ДЕНИС КОЛИСНИЧЕНКО

bhv®

Разработка Linux-приложений

- Разработка приложений на C/C++ в Linux
- Компилятор gcc, отладчик gdb, профайлер gprof
- Создание сетевых приложений клиент/сервер
- Создание модулей ядра
- Межпроцессное взаимодействие (IPC)
- Потоки
- Программирование на языках оболочек bash и tcsh
- Язык программирования TCL
- Библиотека создания графического интерфейса Tk
- Библиотеки glib и GTK+
- Средство создания псевдографического интерфейса dialog

PRO

ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

Денис Колисниченко

Разработка Linux-приложений

Санкт-Петербург
«БХВ-Петербург»
2012

УДК 681.3.06
ББК 32.973.26-018.2
К60

Колисниченко Д. Н.

К60 Разработка Linux-приложений. — СПб.: БХВ-Петербург, 2012. — 432 с. — (Профессиональное программирование)

ISBN 978-5-9775-0747-9

Рассмотрены основные аспекты программирования в Linux: от программирования на языках командных оболочек bash и tcsh до создания приложений с графическим интерфейсом с использованием библиотек Tk, glib, GTK+ и средства dialog, Подробно дано программирование на C/C++ в Linux: использование компилятора gcc, ввод/вывод в Linux, создание многопоточных приложений, сетевых приложений архитектуры клиент/сервер, а также разработка модулей ядра для современной линейки ядер. Описан популярный среди разработчиков утилит язык TCL. Особое внимание уделено отладке и оптимизации программ, рассмотрены отладчик gdb и профайлер gprof.

Для программистов

УДК 681.3.06
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Евгений Рыбаков*
Зав. редакцией	*Григорий Добин*
Редактор	*Владимир Красовский*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Наталия Першакова*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Подписано в печать 30.09.11.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 34,83.
Тираж 1200 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.60.953.Д.005770.05.09 от 26.05.2009 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

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

ISBN 978-5-9775-0747-9

Оглавление

Введение ... **11**

ЧАСТЬ I. ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ КОМАНДНОЙ ОБОЛОЧКИ ... **13**

Глава 1. Командные интерпретаторы ... **14**
1.1. Файл /etc/shells ... 14
1.2. Оболочка *sh* ... 15
1.3. Оболочка *csh* ... 16
1.4. Оболочка *ksh* ... 16
1.5. Оболочка *bash* ... 17
1.6. Оболочка *zsh* ... 17
1.7. Оболочка *tcsh* ... 18
1.8. Оболочка *ash* ... 19
1.9. Выбор оболочки ... 19

Глава 2. Командный интерпретатор *bash* ... **20**
2.1. Настройка *bash* ... 20
2.2. Автоматизация задач с помощью *bash* ... 22
2.3. Привет, мир! ... 23
2.4. Использование переменных в собственных сценариях ... 23
2.5. Передача параметров сценарию ... 25
2.6. Массивы и *bash* ... 25
2.7. Циклы ... 26
2.8. Условные операторы ... 27
2.9. Функции ... 28
2.10. Примеры сценариев ... 28
2.10.1. Сценарий мониторинга журнала ... 28
2.10.2. Переименование файлов ... 29
2.10.3. Преобразование систем счисления ... 30

Глава 3. Создание сценариев на *tcsh* ... **31**
3.1. Использование *tcsh* ... 31
3.2. Конфигурационные файлы *tcsh* ... 32

3.3. Создание сценариев на *tcsh* ... 33
3.3.1. Переменные, массивы и выражения ... 33
3.3.2. Чтение ввода пользователя ... 36
3.3.3. Переменные оболочки *tcsh* ... 36
3.3.4. Управляющие структуры ... 38
Условный оператор *if* ... 38
Условный оператор *if..then..else* ... 39
Оператор *foreach* ... 40
Оператор *while* ... 41
Оператор *switch* ... 41
3.3.5. Встроенные команды *tcsh* ... 42

Глава 4. Пакет dialog: псевдографический интерфейс пользователя ... 45
4.1. Необходимость в графическом интерфейсе ... 45
4.2. Простейшее диалоговое окно ... 46
4.3. Информационное окно ... 47
4.4. Ввод текста ... 49
4.5. Создание меню ... 51
4.6. Проблема выбора: зависимые и независимые переключатели ... 52
4.7. Выбор даты и времени ... 54
4.8. Индикатор ... 55
4.9. Диалог выбора файла ... 56
4.10. Дополнительные возможности ... 57

Глава 5. Компилятор *gcc* и вспомогательные программы ... 60
5.1. Выбор редактора ... 60
5.2. Компилятор *gcc* ... 61
5.2.1. Установка компилятора ... 61
5.2.2. Компиляция первой программы в Linux ... 62
5.2.3. Опции компилятора ... 63
5.3. Автоматическая сборка программ ... 65
5.3.1. Введение в автоматическую сборку ... 65
5.3.2. Синтаксис Makefile ... 66

ЧАСТЬ II. ОСНОВЫ ПРОГРАММИРОВАНИЯ НА C В LINUX ... 59

Глава 6. Библиотеки. Автоматическая сборка библиотек ... 71
6.1. Динамические и статические библиотеки ... 71
6.2. Создание статической библиотеки ... 73
6.3. Создание динамической библиотеки ... 75

Глава 7. Переменные окружения ... 78
7.1. Еще один способ передачи параметров ... 78
7.2. Что такое окружение? ... 78
7.3. Чтение переменных окружения в вашей программе ... 80
7.4. Модификация окружения ... 81

Глава 8. Ввод/вывод в Linux ... 83
8.1. Понятие ввода/вывода. Перенаправление ввода/вывода в командной строке ... 83
8.2. Библиотечные функции C для организации ввода/вывода ... 85

8.3. Низкоуровневый ввод/вывод ... 89
8.3.1. Системные вызовы файлового ввода/вывода ... 89
8.3.2. Системный вызов *creat()* ... 92
8.3.3. Чтение файла: системные вызовы *open()* и *read()* ... 93
8.3.4. Системный вызов *write()* ... 95
8.3.5. Системный вызов *lseek()* ... 97

ЧАСТЬ III. СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ ... 99

Глава 9. Концепция многозадачности ... 100
9.1. Основы многозадачности Linux ... 100
9.1.1. Иерархия процессов ... 100
9.1.2. Аварийное завершение процесса ... 102
9.1.3. Программа *top*: кто больше всех расходует процессорное время ... 105
9.1.4. Команды *nice* и *renice*: изменение приоритета процесса ... 107
9.2. Функция *system()* ... 107

Глава 10. Системные вызовы для работы с процессами ... 109
10.1. Создание и запуск процессов ... 109
10.1.1. Модели описания состояний процессов ... 109
10.1.2. Особенности *fork()* ... 111
10.1.3. Семейство функций *exec* ... 113
10.2. Системный вызов *wait()*: ожидание завершения дочернего процесса ... 117
10.3. Обработка сигналов ... 119
10.4. Получение информации о процессе ... 120

Глава 11. Многопоточные приложения ... 122
11.1. Введение в потоки ... 122
11.2. Функция *pthread_create()* ... 123
11.3. Передача аргументов потоковой функции ... 126
11.4. Правильное завершение потока: функция *pthread_exit()* ... 128
11.5. Избавляемся от бесконечного цикла: функция *pthread_join()* ... 129
11.6. Получение информации о потоке ... 132
11.7. Прерывание потока ... 132

Глава 12. Взаимодействие процессов ... 133
12.1. Способы взаимодействия ... 133
12.2. Каналы ... 133
12.3. Именованные каналы типа FIFO ... 137
12.4. Очереди сообщений ... 140
12.4.1. Межпроцессное взаимодействие System V ... 140
12.4.2. Структуры ядра для работы с очередями ... 141
12.4.3. Создание очереди сообщений ... 143
12.4.4. Постановка и чтение сообщений ... 145
12.5. Семафоры ... 149
12.5.1. Введение в семафоры ... 149
12.5.2. Структуры ядра ... 149
12.5.3. Создание набора семафоров ... 150
12.5.4. Операции над семафорами ... 151
12.5.5. Управление семафором ... 152

12.6. Разделяемые сегменты памяти ... 154
12.6.1. Структуры ядра ... 154
12.6.2. Создание разделяемого сегмента памяти и привязка к нему ... 154
12.6.3. Демонстрационная программа ... 156

Глава 13. Создание модуля ядра ... 158
13.1. Что такое модуль ядра ... 158
13.2. Команды *lsmod*, *insmod*, *modprobe* ... 159
13.3. Установка необходимых пакетов ... 161
13.4. Ваш первый модуль ... 162
13.5. Компиляция модуля ... 165
13.6. Тестируем наш модуль ... 168
13.7. Сборка сложных модулей ... 172
13.8. Настоящее программирование ядра ... 172
13.8.1. Отличие обычных программ от модулей ядра ... 172
13.8.2. Пространства, пространства и еще раз пространства ... 173
13.8.3. Драйверы устройств и ядро ... 174
13.9. Символьные устройства ... 175
13.9.1. Возможные операции ... 175
13.9.2. Регистрация устройства ... 177
13.9.3. Драйвер абстрактного символьного устройства ... 177
13.10. Создание файла в /proc ... 183
13.11. Полезный пример: клавиатурный шпион ... 187

ЧАСТЬ IV. ФАЙЛОВАЯ СИСТЕМА LINUX ... 193

Глава 14. Введение в файловую систему ... 194
14.1. Родные файловые системы Linux ... 194
14.2. Особенности файловой системы Linux ... 195
14.2.1. Имена файлов в Linux ... 195
14.2.2. Файлы и устройства ... 196
14.2.3. Корневая файловая система и монтирование ... 197
14.2.4. Стандартные каталоги Linux ... 197
14.3. Внутреннее строение файловой системы ... 198
14.4. Монтирование файловых систем ... 201
14.4.1. Команды *mount* и *umount* ... 201
14.4.2. Файлы устройств и монтирование ... 202
Жесткие диски ... 202
Приводы оптических дисков ... 204
Дискеты ... 204
Флешки и USB-диски ... 204
14.4.3. Опции монтирования файловых систем ... 205
14.4.4. Монтирование разделов при загрузке ... 206
14.4.5. Подробно о UUID и файле /etc/fstab ... 208
14.4.6. Системный вызов *mount()* ... 210

Глава 15. Операции над каталогами ... 213
15.1. Команды для работы с каталогами ... 213
15.2. Функции для работы с каталогами ... 215
15.2.1. Изменение текущего каталога ... 215

15.2.2. Открываем, читаем и закрываем каталог ... 215
15.2.3. Получение информации о файлах ... 217
15.2.4. Создание и удаление каталога ... 219

Глава 16. Операции с файлами ... 220
16.1. Команды для работы с файлами ... 220
16.2. Системные вызовы для работы с файлами ... 223
16.2.1. Переименование файла: *rename()* ... 223
16.2.2. Удаление файла и каталогов: *unlink()* и *rmdir()* ... 223
16.2.3. Системный вызов *umask()* ... 224
16.2.4. Работа со ссылками ... 224

Глава 17. Получение информации о файловой системе ... 226
17.1. Список смонтированных файловых систем ... 226
17.2. Функции *basename()* и *getcwd()* ... 229

Глава 18. Права доступа к файлам и каталогам ... 230
18.1. Изменение прав доступа. Системный вызов *chmod()* ... 230
18.2. Смена владельца файла. Системный вызов *chown()* ... 233

Глава 19. Псевдофайловые системы ... 234
19.1. Что такое псевдофайловая система ... 234
19.2. Виртуальная файловая система sysfs ... 235
19.3. Виртуальная файловая система /proc ... 235
19.3.1. Информационные файлы ... 236
19.3.2. Файлы, позволяющие изменять параметры ядра ... 236
19.3.3. Файлы, изменяющие параметры сети ... 237
19.3.4. Файлы, изменяющие параметры виртуальной памяти ... 238
19.3.5. Файлы, позволяющие изменить параметры файловых систем ... 238
19.3.6. Как сохранить изменения ... 238

ЧАСТЬ V. СЕТЕВОЕ ПРОГРАММИРОВАНИЕ ... 241

Глава 20. Введение в TCP/IP ... 242
20.1. Модель OSI ... 242
20.2. Что такое протокол ... 244
20.3. Адресация компьютеров ... 245

Глава 21. Программирование сокетов: теория ... 249
21.1. Что такое сокет ... 249
21.2. Создание и связывание сокета ... 250
21.3. Установление связи с удаленным компьютером ... 252
21.4. Передача данных ... 254
21.5. Завершение сеанса связи ... 255

Глава 22. Программирование сокетов: практика ... 256
22.1. Создание приложения клиент/сервер ... 256
22.1.1. Программа-сервер ... 256
22.1.2. Программа-клиент ... 259
22.2. Параметры сокета ... 260

22.3. Сигналы, связанные с сокетами ... 263
22.4. Неблокирующие операции ... 264

ЧАСТЬ VI. СОЗДАНИЕ ГРАФИЧЕСКОГО ИНТЕРФЕЙСА СРЕДСТВАМИ TCL/TK ... 265

Глава 23. Введение в TCL/Tk ... 266
23.1. Знакомство с TCL ... 266
23.2. Установка TCL/Tk ... 266
23.3. Первая программа ... 267

Глава 24. Синтаксис TCL ... 270
24.1. Знакомство с синтаксисом TCL ... 270
24.1.1. Формат TCL-сценария ... 270
24.1.2. Команды *puts* и *format*: вывод и форматирование строки ... 271
24.1.3. Группировка аргументов ... 272
24.1.4. Переменные ... 272
24.1.5. Процедуры ... 274
24.1.6. Получаем ввод пользователя ... 274
24.1.7. Математические операции ... 275
24.1.8. Условная команда *if* ... 276
24.1.9. Команда *while* ... 277
24.1.10. Команда *for* ... 277
24.2. Строки ... 277
24.2.1. Команда *string* ... 277
24.2.2. Сравнение строк ... 279
24.2.3. Получаем информацию о строках ... 280
24.2.4. Модификация строк ... 282
24.2.5. Конкатенация строк ... 283
24.3. Списки ... 283
24.3.1. Команда *list*: создание списка ... 283
24.3.2. Команда *concat*: слияние списков ... 284
24.3.3. Команда *lappend*: добавление элемента в конец списка ... 284
24.3.4. Доступ к элементам списка ... 284
24.3.5. Вставка новых элементов ... 285
24.3.6. Замена и удаление элементов списка ... 285
24.3.7. Поиск элемента ... 285
24.3.8. Сортировка списка ... 287
24.3.9. Преобразование строки в список и обратно ... 287
24.3.10. Цикл *foreach* ... 288
24.4. Массивы ... 289
24.4.1. Отличие массивов от списков ... 289
24.4.2. Команда *array*: обработка массивов ... 289
24.5. Ошибки начинающих TCL-программистов ... 290

Глава 25. Работа с файлами ... 292
25.1. Открываем и закрываем файлы ... 292
25.2. Чтение файла ... 293
25.3. Запись файлов ... 295
25.4. Произвольный доступ к файлу ... 296

Глава 26. Понятие о виджетах ... 297
26.1. Tk-программирование ... 297
26.2. Компоненты Tk-приложения и имена виджетов ... 300

Глава 27. Основные элементы графического интерфейса ... 302
27.1. Команда *pack* ... 302
27.2. Команда *button* ... 304
27.3. Команда *checkbutton* ... 307
27.4. Зависимые переключатели ... 311
27.5. Создание меню ... 313
27.6. Поля ввода ... 315
27.7. Списки и домашнее задание ... 318
27.8. Программирование событий. Команда *bind* ... 319

Глава 28. Многооконный интерфейс ... 321
28.1. Менеджер геометрии *grid* ... 321
28.1.1. Относительное размещение ... 321
28.1.2. Абсолютное размещение ... 323
28.1.3. Объединение ячеек ... 325
28.2. Фреймы ... 326
28.3. Создание окон ... 327
28.4. Сообщения ... 328
28.5. Диалоги открытия и сохранения файла ... 330

Глава 29. Практический пример ... 332
29.1. Постановка задачи ... 332
29.2. Создание оболочки ... 333
29.3. Запуск оболочки ... 336
29.4. Последние штрихи ... 336

ЧАСТЬ VII. БИБЛИОТЕКА GTK+ ... 337

Глава 30. Знакомство с библиотекой ... 338
30.1. Введение в GTK+ ... 338
30.2. Библиотека GLib ... 339
30.2.1. Типы данных ... 339
30.2.2. Строки в GLib ... 340
30.2.3. Функции распределения памяти ... 341
30.2.4. Списки ... 342
30.2.5. Использование таймеров ... 344

Глава 31. Первая программа на GTK+ ... 346
31.1. Виджеты, контейнеры, сигналы и события ... 346
31.2. Создание первой программы ... 347
31.3. Компиляция программы ... 348
31.4. Совершенствование программы. Обработчик сигнала ... 351

Глава 32. Виджеты ... 354
32.1. Подробно о сигналах ... 354
32.1.1. Сигналы и события ... 354
32.1.2. Виджет *EventBox* ... 356

32.2. Русский текст и GTK ... 360
32.3. Состояния виджета ... 361
32.4. Контейнеры, поля ввода и кнопки ... 362
32.5. Зависимые и независимые переключатели ... 370
32.6. Список *CList* ... 375
32.7. Диалог выбора файлов ... 379
32.8. Визуальная разработка интерфейса пользователя ... 381

Глава 33. Редактор интерфейсов Glade ... 382
33.1. Быстрая разработка приложений ... 382
33.2. Установка Glade ... 383
33.3. Использование Glade ... 384
33.4. Создание программы ... 388
33.5. Компиляция программы ... 390
33.6. Рекомендуемая литература ... 390

ЧАСТЬ VIII. ОТЛАДКА И ОПТИМИЗАЦИЯ ПРОГРАММЫ ... 393

Глава 34. Отладка программ. Трассировка системных вызовов ... 394
34.1. Для чего нужна отладка программ ... 394
34.2. Введение в отладчик *gdb* ... 396
34.3. Пример использования *gdb* ... 399
34.4. Трассировка системных вызовов ... 404

Глава 35. Оптимизация программы ... 407
35.1. Назначение и основные опции профайлера *gprof* ... 407
35.2. Практическое использование профайлера ... 408

Заключение ... 413

Приложение. Ядро Linux ... 415
П1. Установка исходных кодов ядра ... 415
П2. Настройка ядра ... 417
П3. Компиляция ядра ... 419

Предметный указатель ... 423

Введение

Процесс разработки приложений для Linux очень многогранный. Вы можете заниматься как разработкой сценариев командной оболочки, так и разработкой модулей ядра. При создании сложных проектов вполне возможно, что оба эти подхода будут использоваться в одном проекте. Все зависит от поставленных целей.

Книга охватывает основные моменты программирования в Linux — от создания простых сценариев на языках оболочек `bash` и `tcsh` до создания программ на C/C++ с графическим интерфейсом.

Предполагается, что читатель уже знаком с языками C и C++: эта книга не учебник по C/C++, в ней вы не найдете описания синтаксиса и стандартных функций этих языков программирования. Зато в ней рассмотрены нюансы создания программ на этих языках в Linux. Поверьте, хотя синтаксис языка и набор стандартных функций остается тем же, таких нюансов достаточно много: у каждой операционной системы есть свои особенности, и если вы раньше программировали в Windows, то будете удивлены, когда узнаете, что в Linux "все не так" (ну, или почти все).

Книга разделена на восемь частей. В *части I* рассматривается синтаксис встроенных языков оболочек `bash` и `tcsh`. Также в этой части будет рассмотрен пакет dialog, позволяющий создавать интерфейс пользователя для сценариев.

Знать синтаксис оболочки нужно каждому уважающему себя программисту. По сути, знание синтаксиса можно приравнять к знанию самой оболочки: не знаете, как программировать в `bash`, значит, не знаете, как использовать эту оболочку. Да и не всегда нужно создавать программу на C: для некоторых простых (и не очень) задач это нецелесообразно. Как вы думаете, почему сценарии системы инициализации `init` написаны на языке командной оболочки, а не на C? Ведь можно было бы написать все эти программы на C, тогда бы сократилось время загрузки системы — ведь скомпилированные программы будут выполняться быстрее. Но тогда, чтобы внести малейшее изменение в процесс загрузки (поверьте, в ранних версиях дистрибутивов это приходилось делать чаще, чем можно представить), нужно перекомпилировать программу. А в случае со сценарием достаточно открыть его в редакторе и изменить код, после чего сценарий "готов к употреблению". В свою очередь это также освобождает пользователя от знания синтаксиса C и от необходимости установки компилятора на каждый компьютер.

Да, были системы инициализации, полностью написанные на C, например `init-ng` (Init Next Generation), но что бы вы думали? Они не прижились...

Часть II посвящена основам программирования на C в Linux. Вы познакомитесь с компилятором `gcc`, утилитой автоматической сборки программ `make`, с переменными окружения, также будет рассмотрен ввод/вывод в Linux. Так сказать, необходимый минимум для начала настоящего программирования в Linux. Прочитав эту часть книги, вы сможете перенести в Linux ваши программы, разработанные с использованием стандартных библиотек C. Все, что нужно для этого — знать, как скомпилировать программу в Linux, а об этом как раз и говорится в *части II*.

В *части III* рассматривается системное программирование в Linux: организация межпроцессного взаимодействия (IPC), потоки, создание модулей ядра и т. д. Все это позволит почувствовать себя настоящим программистом, но ради справедливости нужно отметить, что далеко не всем программистам нужны средства, описанные в этой части книги. Если вы планируете написать текстовый редактор, то нет особого смысла рассматривать программирование ядра.

Файловая система Linux заслуживает отдельного разговора и такой разговор будет — в *части IV*. Сначала будет рассмотрена файловая система глазами пользователя, а затем — программиста. Понимаю, что книга может попасть в руки программиста, ни разу в жизни не видевшего Linux, и в этом случае нет смысла рассматривать системные вызовы для работы с файлами и каталогами, если человек не знает, как скопировать или удалить файл в командной оболочке.

В *части V* рассматривается сетевое программирование. Мы создадим собственный сервер и собственный клиент — две программы, которые могут обмениваться данными по сети. Вам же останется только разработать протокол — набор правил для обмена информацией, и у вас будет полноценное приложение архитектуры клиент/сервер.

Части VI и *VII* посвящены созданию графического интерфейса пользователя. Сначала будет рассмотрен язык программирования TCL (наверное, в Windows вы о таком и не слышали) и графическая библиотека Tk, используемая в паре с TCL. Затем (в *части VII*) будут рассмотрены библиотеки GLib и GTK+ для создания GUI. Также будет рассмотрен редактор интерфейсов Glade, позволяющий за несколько минут создать интерфейс небольшого окна и в результате этого существенно сократить код GTK-программы.

И наконец, *часть VIII* посвящена отладке и оптимизации программы. Будут рассмотрены отладчик `gdb` и профайлер `gprof`.

В *приложении* вы найдете инструкции по перекомпиляции ядра — они вам понадобятся на случай, если вы будете создавать модули ядра.

Вот теперь самое время приступить к чтению книги. О том, как связаться со мной, вы сможете прочитать в *заключении*.

ЧАСТЬ I

Программирование на языке командной оболочки

Глава 1.	Командные интерпретаторы
Глава 2.	Командный интерпретатор *bash*
Глава 3.	Создание сценариев на *tcsh*
Глава 4.	Пакет dialog: псевдографический интерфейс пользователя

Первая часть книги посвящена программированию на языке командного интерпретатора Linux. Кому нужны такие программы? Не всегда целесообразно писать программу на языке C. Некоторые операции можно с легкостью запрограммировать, используя язык командного интерпретатора. А если вы еще освоите дополнительные средства, например язык awk, используемый для обработки текстовых файлов по определенному шаблону, то написание некоторых программ займет у вас всего несколько минут. А вот в C придется изобретать велосипед заново...

ГЛАВА 1

Командные интерпретаторы

1.1. Файл /etc/shells

У командных интерпретаторов есть свои преимущества в плане переносимости. К примеру, пусть у вас будет 64-битная система, вы скомпилировали программу, написанную на C. Чтобы она заработала на 32-битной системе или же вообще на компьютере с другой архитектурой (ведь мир не сошелся клином на x86 и x86-64), то программу придется перекомпилировать. В случае со сценарием командной оболочки этого делать не требуется — просто скопируйте сценарий на другую систему и выполните его. Ведь сценарий — это обычный файл, поэтому проблем с переносимостью у вас не будет.

Конечно, сценарии командного интерпретатора не панацея. Сложных программ на них не напишешь. Я имею в виду действительно сложных, а не больших! Большую программу (свыше 1000 строк) написать на языке командного интерпретатора можно, пример тому — сценарии инициализации Linux, которые в некоторых дистрибутивах длиннее, чем лимузин Билла Гейтса. Что же касается больших программ, то выполняться они будут значительно медленнее, чем их аналоги, написанные на C. Ведь выполнением программы занимается командный интерпретатор, а для выполнения скомпилированной C-программы не нужны какие-либо вспомогательные программы (если, конечно, вы их не вызываете из своей программы).

Также к недостаткам сценариев командной оболочки можно отнести необходимость установки той командной оболочки, на языке которой написан сценарий. Это небольшой недостаток, т. к. обычно сценарии пишутся на `bash` или `tcsh`, а эти командные оболочки почти всегда установлены в Linux. Что же касается FreeBSD, то в ней `bash` по умолчанию не установлен, поэтому его придется установить самостоятельно. Впрочем, о выборе командного интерпретатора для написания сценариев мы и поговорим в данной главе.

По умолчанию во всех современных дистрибутивах используется командный интерпретатор `bash`. Основное предназначение `bash`, как и любой другой оболочки, — выполнение команд, введенных пользователем. Пользователь вводит команду, `bash` ищет программу, соответствующую команде, в каталогах, указанных в переменной

окружения PATH. Если такая программа найдена, то bash запускает ее и передает введенные пользователем параметры. В противном случае выводится сообщение о невозможности выполнения команды.

Кроме bash существуют и другие оболочки — sh, csh, ksh, zsh и пр. Все командные оболочки, установленные в системе, прописаны в файле /etc/shells. Список оболочек может быть довольно длинным. В листинге 1.1 представлен файл /etc/shells дистрибутива Fedora (установка по умолчанию).

Листинг 1.1. Файл /etc/shells дистрибутива Fedora

```
/bin/ash
/bin/bash
/bin/csh
/bin/false
/bin/ksh
/bin/sh
/bin/tcsh
/bin/true
/bin/zsh
/usr/bin/csh
/usr/bin/ksh
/usr/bin/bash
/usr/bin/tcsh
/usr/bin/zsh
```

С точки зрения пользователя указанные оболочки мало чем отличаются. Все они позволяют выполнять введенные пользователем команды. Но оболочки используются не только для выполнения команд, а еще и для автоматизации задач с помощью *сценариев*. Так вот, все эти оболочки отличаются синтаксисом языка описания сценариев.

Примечание

В листинге 1.1 программы /bin/false и /bin/true не являются оболочками. Это "заглушки", которые можно использовать, если вы хотите отключить ту или иную учетную запись пользователя. При входе пользователя в систему запускается установленная для него оболочка. Для каждого пользователя имеется возможность задать свою оболочку (изменить оболочку пользователь может самостоятельно командой chsh). Так вот, если для пользователя задать оболочку /bin/false (или /bin/true), он не сможет войти в систему. Точнее, он войдет в систему, но и сразу выйдет из нее, поскольку обе "заглушки" ничего не делают, а просто возвращают значение 0 (для false) или 1 (для true). Сессия же пользователя длится до завершения работы его оболочки.

1.2. Оболочка *sh*

Самым первым командным интерпретатором (оболочкой) в операционной системе UNIX (да, именно UNIX, поскольку корни Linux уходят в далекие 70-е годы прошлого столетия) была sh (сокращение от *shell*). Данная оболочка до сих пор используется в современных версиях Linux (и FreeBSD).

Оболочка sh была разработана Стивеном Борном (Steve Bourne), поэтому ее второе название — Bourne Shell. Изначально sh была создана для операционной системы AT&T (разработка Bell Labs). Чуть позже sh была усовершенствована и вошла в состав POSIX (Portable Operating System Interface for UNIX — Переносимый интерфейс операционных систем UNIX). Усовершенствованная версия sh до сих пор устанавливается (но не используется по умолчанию) в современных версиях FreeBSD.

С точки зрения пользователя оболочка sh не очень удобна, поэтому пользователи предпочитают другие оболочки, например tcsh или bash.

1.3. Оболочка *csh*

Оболочка csh (C Shell) по умолчанию используется в FreeBSD. Разработка csh началась еще в первых версиях BSD (Linux будет создан лет через 15). Тогда в институте Беркли начали создавать новую оболочку (csh), потому что не захотели мириться с ограничениями sh.

Внутренний синтаксис csh очень напоминает язык программирования C, поэтому он должен был понравиться программистам (а в то время все пользователи компьютеров являлись программистами). Хотя сами программисты отмечали, что синтаксис не очень удобен, даже несмотря на то, что он похож на C.

По сравнению с sh, у csh есть множество преимуществ: она умеет управлять заданиями, хранит историю ранее введенных команд, а также у csh есть сценарии, которые выполняются при входе пользователя (запуске оболочки) и при выходе пользователя (когда пользователь вводит команду exit). В то время у sh не было таких сценариев, которые оказались очень удобными.

С точки зрения обычного использования оболочки (а не программирования) csh тоже была на высоте.

В последних версиях FreeBSD и Linux вместо csh используется ее усовершенствованная версия — tcsh, а файл /bin/csh — это просто ссылка на /bin/tcsh.

1.4. Оболочка *ksh*

Не хочется делать экскурс в историю UNIX, но пару слов сказать все же придется. Изначально система UNIX появилась в лабораториях компании AT&T, позже появились версии UNIX института Беркли (операционная система называлась BSD). Так уж сложилось исторически, что AT&T и институт Беркли постоянно конкурировали между собой. Как только в Беркли разработали оболочку csh, в AT&T принялись разрабатывать собственную оболочку, которая получила название ksh (Korn Shell) — по имени разработчика Дэвида Корна (David Korn).

Оболочка ksh по функциям похожа на csh: есть поддержка управления заданиями, история команд, позволяет назначать командам псевдонимы, а также создавать конфигурационные файлы для подоболочек.

Несмотря на то что оболочка была разработана в 1986 году, она до сих пор используется в некоторых версиях UNIX по умолчанию, а также устанавливается по умолчанию во всех дистрибутивах Linux (но не используется по умолчанию). Правда, изначально ksh — это коммерческий продукт, поэтому в FreeBSD и Linux используется не ksh, а ее бесплатная версия — pdksh, но для краткости исполнимый файл называется ksh.

Начинающим пользователям ksh не понравится (лучше использовать bash) — она слишком неудобна в использовании, зато у нее довольно развитый синтаксис внутреннего языка, что понравится программистам.

1.5. Оболочка *bash*

Командный интерпретатор bash (Bourne Again Shell) был разработан Фондом свободного программного обеспечения (Free Software Foundation, FSF). За основу была взята оболочка sh. Оболочка стала очень популярной и сейчас используется по умолчанию во всех дистрибутивах Linux.

Оболочка bash может использоваться также и для запуска сценариев sh, поэтому sh во многих системах уже не устанавливается, а файл /bin/sh — это ссылка на /bin/bash.

С точки зрения пользователей оболочка bash намного удобнее, чем ksh. Вы можете легко редактировать командную строку, просматривать историю команд, создавать псевдонимы команд, создавать переменные окружения и использовать их в собственных сценариях. Как и в csh, в bash есть сценарии, которые вызываются при запуске оболочки и при выходе из нее.

Синтаксис bash довольно прост, поэтому бóльшая часть сценариев, разрабатываемых в Linux, пишется именно на bash.

1.6. Оболочка *zsh*

Сейчас мы поближе познакомимся с оболочкой zsh, которая становится все более популярной.

До того как я познакомился с zsh, я считал самой удобной оболочку bash. Однако это не так.

Что же удобного в zsh? Во-первых, навигация. В bash для перехода в каталог /dir/subdir1/subdir2 нужно ввести команду:

```
cd /dir/subdir1/subdir2
```

Можно использовать автодополнение bash — вводить начальные символы каталога и нажимать клавишу <Tab>. Это будет выглядеть примерно так:

```
cd /dir/sub [Tab]/subdi [Tab]
```

В zsh можно ввести:

```
/d/s/s
```

Затем нажать клавишу <Tab> — вы перейдете в нужный каталог. Например, для перехода в /etc/sysconfig/network нужно ввести /e/s/n и нажать клавишу <Tab>. Кстати, команда cd уже не нужна.

Покажу еще один трюк. Предположим, у нас есть каталог files, а в нем есть каталоги f1 и f2. Внутри каждого каталога f* есть каталоги sources и last. То есть структура каталогов будет примерно такой:

```
/files/f1/sources/last
/files/f2/sources/last
```

Пусть мы находимся в каталоге /files/f1/sources/last. Для перехода в каталог /files/f2/sources/last введите команду:

```
cd 1 2
```

Но одной лишь навигацией возможности zsh не ограничиваются. Можно, например, использовать вот такое перенаправление:

```
< /var/log/messages
```

Оболочка запустит программу, указанную в переменной $PAGER. В большинстве случаев это аналогично команде:

```
cat /var/log/messages | less
```

Все возможности zsh в этой главе мы рассматривать не будем — их намного больше, чем вам кажется. Если вы заинтересовались, то прочитайте следующие страницы:

- **http://opennet.ru/base/dev/zsh_intro.txt.html**;
- **http://citkit.ru/articles/1083/**;
- **http://alexott.net/ru/writings/zsh/index.html**;
- **http://habrahabr.ru/blogs/linux/82537/**.

1.7. Оболочка *tcsh*

Оболочка tcsh является модифицированной версией csh. Буква t в названии означает TENEX: изначально оболочка была разработана для операционной системы TENEX (использовалась в далеком прошлом на компьютерах DEC PDP-10).

В tcsh усовершенствована функция редактирования командной строки, есть автозавершение команд (как в bash). Кроме того, tcsh может распознавать потенциально опасные команды. Если вы от имени root попытаетесь удалить все файлы, оболочка потребует подтверждения.

Оболочка tcsh очень удобна в использовании, но ее синтаксис сценариев сложнее, чем у bash. Однако далее мы все же рассмотрим разработку сценариев на tcsh, чтобы вы смогли оценить сложность создания разработки сценариев на bash и на tcsh.

1.8. Оболочка *ash*

Оболочка `ash` (Almquist shell) — самая простая командная оболочка. Это самая маленькая оболочка, доступная для UNIX (у нее самые низкие требования к дисковому пространству).

У `ash` всего 24 встроенных команды и 10 опций командной строки. Обычно `ash` используется при загрузке Linux в однопользовательском режиме (или в режиме восстановления).

Оболочка `ash` совместима с `sh`, с ее помощью можно проверить сценарии на совместимость с традиционным синтаксисом `sh`. А в операционной системе NetBSD оболочка `ash` используется вместо /bin/sh.

1.9. Выбор оболочки

Какую оболочку выбрать? Первым делом нужно оценить простоту использования оболочки. Ведь вы будете использовать эту оболочку каждый день, поэтому простота использования должна быть на первом месте.

Затем нужно оценить простоту синтаксиса оболочки. Конечно, это только в том случае, если вы планируете разрабатывать собственные сценарии. Также не нужно забывать, что вы можете использовать одну оболочку, а разрабатывать сценарии — на языке другой оболочки. Например, в повседневной работе вы можете использовать `zsh`, а разрабатывать сценарии на языке `bash`.

Довольно удобны в использовании оболочки `bash`, `tcsh` и `zsh`. Скорее всего, вы выберете одну из них. А вот для программирования вы будете использовать или `bash`, или `tcsh` (синтаксис `zsh` не очень понятен). Именно эти две оболочки будут рассмотрены в последующих двух главах.

ГЛАВА 2

Командный интерпретатор *bash*

2.1. Настройка *bash*

`bash` — это самая популярная командная оболочка (командный интерпретатор) Linux. Основное предназначение `bash` — выполнение команд, введенных пользователем. Пользователь вводит команду, `bash` ищет программу, соответствующую команде, в каталогах, указанных в переменной окружения `PATH`. Если такая программа найдена, то `bash` запускает ее и передает ей введенные пользователем параметры. В противном случае выводится сообщение о невозможности выполнения команды.

Файл /etc/profile содержит глобальные настройки `bash`, он влияет на всю систему — на каждую запущенную оболочку. Обычно /etc/profile не нуждается в изменении, а при необходимости изменить параметры `bash` редактируют один из файлов:

- ~/.bash_profile — обрабатывается при каждом входе в систему;
- ~/.bashrc — обрабатывается при каждом запуске дочерней оболочки;
- ~/.bash_logout — обрабатывается при выходе из системы.

Файл ~/.bash_profile часто не существует, а если и существует, то в нем есть всего одна строка:

```
source ~/.bashrc
```

Данная строка означает, что нужно прочитать файл .bashrc. Поэтому будем считать основным конфигурационным файлом файл .bashrc. Но помните, что он влияет на оболочку текущего пользователя (такой файл находится в домашнем каталоге каждого пользователя, не забываем: "~" означает домашний каталог). Если же вдруг понадобится задать параметры, которые повлияют на всех пользователей, то нужно редактировать файл /etc/profile.

В файле .bash_history (тоже находится в домашнем каталоге) хранится история команд, введенных пользователем. Так что вы можете просмотреть свои же команды, которые накануне вводили.

Какие настройки могут быть в .bashrc? Как правило, в этом файле задаются псевдонимы команд, определяется внешний вид приглашения командной строки, задаются значения переменных окружения.

Псевдонимы команд задаются с помощью команды `alias`, вот несколько примеров:

```
alias h='fc -l'
alias ll='ls -laFo'
alias l='ls -l'
alias g='egrep -i'
```

Псевдонимы работают просто: при вводе команды `l` на самом деле будет выполнена команда `ls -l`.

Теперь рассмотрим пример изменения приглашения командной строки. Глобальная переменная `PS1` отвечает за внешний вид командной строки. По умолчанию командная строка имеет формат:

```
пользователь@компьютер:рабочий_каталог
```

Значение `PS1` при этом будет:

```
PS1='\u@\h:\w$'
```

В табл. 2.1 приведены допустимые модификаторы командной строки.

Таблица 2.1. *Модификаторы командной строки*

Модификатор	Описание
`\a`	ASCII-символ звонка (код 07). Не рекомендуется его использовать — очень скоро начнет раздражать
`\d`	Дата в формате "день недели, месяц, число"
`\h`	Имя компьютера до первой точки
`\H`	Полное имя компьютера
`\j`	Количество задач, запущенных в оболочке в данное время
`\l`	Название терминала
`\n`	Символ новой строки
`\r`	Возврат каретки
`\s`	Название оболочки
`\t`	Время в 24-часовом формате (ЧЧ:ММ:СС)
`\T`	Время в 12-часовом формате (ЧЧ:ММ:СС)
`\@`	Время в 12-часовом формате (AM/PM)
`\u`	Имя пользователя
`\v`	Версия `bash` (сокращенный вариант)
`\V`	Версия `bash` (полная версия: номер релиза, номер патча)
`\w`	Текущий каталог (полный путь)

Таблица 2.1 *(окончание)*

Модификатор	Описание
`\W`	Текущий каталог (только название каталога, без пути)
`\!`	Номер команды в истории
`\#`	Системный номер команды
`\$`	Если UID пользователя равен 0, будет выведен символ `#`, иначе — символ `$`
`\\`	Обратный слэш
`$ ( )`	Подстановка внешней команды

Вот пример альтернативного приглашения командной строки:

```
PS1='[\u@\h] $(date +%m/%d/%y) \$'
```

Вид приглашения будет такой:

```
[denis@host] 12/06/10 $
```

В квадратных скобках выводится имя пользователя и имя компьютера, затем используется конструкция `$()` для подстановки даты в нужном нам формате и символ приглашения, который изменяется в зависимости от идентификатора пользователя.

Установить переменную окружения можно с помощью команды `export`, что будет показано позже.

2.2. Автоматизация задач с помощью *bash*

Представим, что нам предстоит выполнить резервное копирование всех важных файлов, для чего нужно создать архивы каталогов /etc, /home и /usr. Понятно, что понадобятся три команды вида:

```
tar -cvjf имя_архива.tar.bz2 каталог
```

Затем нам нужно записать все эти три файла на DVD с помощью любой программы для прожига DVD.

Если выполнять данную операцию раз в месяц (или хотя бы раз в неделю), то ничего страшного. Но представьте, что вам нужно делать это каждый день или даже несколько раз в день? Думаю, такая рутинная работа вам быстро надоест. А ведь можно написать *сценарий*, который сам будет создавать резервные копии и записывать их на DVD! Все, что вам нужно, — это вставить чистый DVD перед запуском сценария.

Можно пойти и иным путем. Написать сценарий, который будет делать резервные копии системных каталогов и записывать их на другой раздел жесткого диска. Ведь не секрет, что резервные копии делаются не только на случай сбоя системы, но и для защиты от некорректного изменения данных пользователем. Помню, удалил важную тему форума и попросил своего хостинг-провайдера сделать откат. Я был приятно удивлен, когда мне предоставили на выбор три резервные копии — оста-

лось лишь выбрать наиболее подходящую. Не думаете же вы, что администраторы провайдера только и занимались тем, что три раза в день копировали домашние каталоги пользователей? Поэтому автоматизация — штука полезная, и любому администратору нужно знать, как автоматизировать свою рутинную работу.

2.3. Привет, мир!

По традиции напишем первый сценарий, выводящий всем известную фразу: "Привет, мир!" (Hello, world!). Для редактирования сценариев вы можете использовать любимый текстовый редактор, например `nano` или `ee` (листинг 2.1).

Листинг 2.1. Первый сценарий

```
#!/bin/bash
echo "Привет, мир!"
```

Первая строка нашего сценария — это указание, что он должен быть обработан программой /bin/bash. Обратите внимание: если между `#` и `!` окажется пробел, то данная директива не сработает, поскольку будет воспринята как обычный комментарий. Комментарии начинаются, как вы уже догадались, с символа решетки:

```
# Комментарий
```

Вторая строка — это оператор `echo`, выводящий нашу строку. Сохраните сценарий под именем hello и введите команду:

```
$ chmod +x hello
```

Для запуска сценария введите команду:

```
./hello
```

На экране вы увидите строку:

```
Привет, мир!
```

Чтобы вводить для запуска сценария просто `hello` (без `./`), сценарий нужно скопировать в каталог /usr/bin (точнее, в любой каталог из переменной окружения `PATH`):

```
# cp ./hello /usr/bin
```

2.4. Использование переменных в собственных сценариях

В любом серьезном сценарии вы не обойдетесь без использования *переменных*. Переменные можно объявлять в любом месте сценария, но до места их первого применения. Рекомендуется объявлять переменные в самом начале сценария, чтобы потом не искать, где вы объявили ту или иную переменную.

Для объявления переменной используется следующая конструкция:

```
переменная=значение
```

Пример объявления переменной:

```
ADDRESS=www.dkws.org.ua
echo $ADDRESS
```

Обратите внимание на следующие моменты:

- при объявлении переменной знак доллара не ставится, но он обязателен при использовании переменной;
- при объявлении переменной не должно быть пробелов до и после знака =.

Значение для переменной указывать вручную не обязательно — его можно прочитать с клавиатуры:

```
read ADDRESS
```

или со стандартного вывода программы:

```
ADDRESS='hostname'
```

Чтение значения переменной с клавиатуры осуществляется с помощью инструкции `read`. При этом указывать символ доллара не нужно. Вторая команда устанавливает в качестве значения переменной `ADDRESS` вывод команды `hostname`.

В Linux часто используются *переменные окружения*. Это специальные переменные, содержащие служебные данные. Вот примеры некоторых часто используемых переменных окружения:

- `BASH` — полный путь до исполняемого файла командной оболочки `bash`;
- `BASH_VERSION` — версия `bash`;
- `HOME` — домашний каталог пользователя, который запустил сценарий;
- `HOSTNAME` — имя компьютера;
- `RANDOM` — случайное число в диапазоне от 0 до 32 767;
- `OSTYPE` — тип операционной системы;
- `PWD` — текущий каталог;
- `PS1` — строка приглашения;
- `UID` — ID пользователя, который запустил сценарий;
- `USER` — имя пользователя.

Для установки собственной переменной окружения используется команда `export`:

```
# Присваиваем переменной значение
$ADDRESS=ww.dkws.org.ua
# Экспортируем переменную – делаем ее переменной окружения
# После этого переменная ADDRESS будет доступна в других сценариях
export $ADDRESS
```

2.5. Передача параметров сценарию

Очень часто сценариям нужно передавать различные параметры, например режим работы или имя файла/каталога. Для передачи параметров используются следующие специальные переменные:

- `$0` — содержит имя сценария;
- `$n` — содержит значение параметра (`n` — номер параметра);
- `$#` — позволяет узнать количество параметров, которые были переданы.

Рассмотрим небольшой пример обработки параметров сценария. Конструкцию `case-esac` мы еще не рассматривали, но общий принцип должен быть понятен (листинг 2.2).

Листинг 2.2. Пример обработки параметров сценария

```
# Сценарий должен вызываться так:
# имя_сценария параметр

# Анализируем первый параметр
case "$1" in
  start)
        # Действия при получении параметра start
        echo "Запускаем сетевой сервис"
        ;;
  stop)
        # Действия при получении параметра stop
        echo "Останавливаем сетевой сервис"
        ;;
*)
        # Действия в остальных случаях
        # Выводим подсказку о том, как нужно использовать сценарий, и
        # завершаем работу сценария
echo "Usage: $0 {start|stop }"
        exit 1
        ;;
esac
```

Думаю, приведенных комментариев достаточно, поэтому подробно рассматривать работу сценария из листинга 2.2 не будем.

2.6. Массивы и *bash*

Интерпретатор `bash` позволяет использовать *массивы*. Массивы объявляются подобно переменным.

Вот пример объявления массива:

```
ARRAY[0]=1
ARRAY[1]=2

echo $ARRAY[0]
```

2.7. Циклы

Как и в любом языке программирования, в bash можно использовать *циклы*. Мы рассмотрим циклы for и while, хотя вообще в bash доступны также циклы until и select, но они применяются довольно редко.

Синтаксис цикла for выглядит так:

```
for переменная in список
do
команды
done
```

В цикле при каждой итерации переменной будет присвоен очередной элемент списка, над которым будут выполнены указанные команды. Чтобы было понятнее, рассмотрим небольшой пример:

```
for n in 1 2 3;
do
echo $n;
done
```

Обратите внимание: список значений и список команд должны заканчиваться точкой с запятой.

Как и следовало ожидать, наш сценарий выведет на экран следующее:

```
1
2
3
```

Синтаксис цикла while выглядит немного иначе:

```
while условие
do
команды
done
```

Цикл while выполняется до тех пор, пока истинно заданное условие. Подробно об условиях мы поговорим в следующем разделе, а сейчас напишем аналог предыдущего цикла, т. е. нам нужно вывести 1, 2 и 3, но с помощью while, а не for:

```
n=1
while [ $n -lt 4 ]
do
  echo "$n "
  n=$(( $n+1 ));
done
```

2.8. Условные операторы

В bash доступно два *условных оператора*, if и case. Синтаксис оператора if следующий:

```
if условие_1 then
       команды_1
elif условие_2 then
       команды_2
...
elif условие_N then
       команды_N
else
       команды_N+1
fi
```

Оператор if в bash работает аналогично оператору if в других языках программирования. Если истинно первое условие, то выполняется первый список команд, иначе — проверяется второе условие и т. д. Количество блоков elif, понятно, не ограничено.

Самая ответственная задача — это правильно составить условие. Условия записываются в квадратных скобках. Вот пример записи условий:

```
# Переменная N = 10
[ N==10 ]
# Переменная N не равна 10
[ N!=10 ]
```

Операции сравнения указываются не с помощью привычных знаков > или <, а с помощью следующих выражений:

- -lt — меньше;
- -gt — больше;
- -le — меньше или равно;
- -ge — больше или равно;
- -eq — равно (используется вместо ==).

Применять данные выражения нужно следующим образом:

```
[ переменная выражение значение|переменная ]
```

Например:

```
# N меньше 10
[ $N -lt 10 ]
# N меньше A
[ $N -lt $A ]
```

В квадратных скобках вы также можете задать выражения для проверки существования файла или каталога:

- `-e файл` — условие истинно, если файл существует;
- `-d каталог` — условие истинно, если каталог существует;
- `-x файл` — условие истинно, если файл является исполнимым.

С оператором `case` мы уже немного знакомы, но сейчас рассмотрим его синтаксис подробнее:

```
case переменная in
значение_1) команды_1 ;;
...
значение_N) команды_N ;;
*) команды_по_умолчанию;;
esac
```

Значение указанной переменной по очереди сравнивается с приведенными значениями (`значение_1` ... `значение_N`). Если есть совпадение, то будут выполнены команды, соответствующие значению. Если совпадений нет, то будут выполнены команды по умолчанию. Пример использования `case` был приведен в листинге 2.2.

2.9. Функции

В `bash` можно использовать функции. Синтаксис объявления функции следующий:

```
имя() { список; }
```

Вот пример объявления и использования функции:

```
list_txt()
{
echo "Выводим текстовые файлы"
ls *.txt
}
```

2.10. Примеры сценариев

2.10.1. Сценарий мониторинга журнала

Начнем с простого сценария мониторинга журнала (листинг 2.3). Системные журналы постоянно обновляются, а наш сценарий будет каждые три секунды выводить последние 15 строк выбранного нами журнала. Сценарий будет полезен при настройке системы, когда нужно постоянно просматривать журналы, чтобы понять, как система реагирует на новые настройки. Для прекращения работы сценария следует нажать клавиши <Ctrl>+<C>.

Листинг 2.3. Мониторинг системного журнала

```
#!/bin/bash
# Интервал обновления, в секундах
INT=3
```

```
while [ true ]
do
# Выводим последние 15 строк журнала
tail -n 15 $1
# Ждем
sleep $INT
# Две пустые строки
echo; echo
done
```

Формат вызова следующий:

```
./сценарий файл_журнала
```

Например:

```
./script /var/log/messages
```

2.10.2. Переименование файлов

Следующий сценарий сложнее: он ищет файлы в текущем каталоге, в именах которых есть пробелы, и заменяет пробелы на символы подчеркивания (листинг 2.4).

Листинг 2.4. Сценарий rename_blanks

```
#! /bin/bash
#

# Сколько файлов мы переименовали
num=0
# Перебираем все файлы в текущем каталоге
for filename in *
do
# Передаем имя файла фильтру grep
# Если имя файла содержит пробел, то код завершения
# последней операции равен 0
     echo "$filename" | grep -q " "
     if [ $? -eq 0 ]
     then
# Если код завершения равен 0, переименовываем
# файл
       fname=$filename
       n=`echo $fname | sed -e "s/ /_/g"`
       mv "$fname" "$n"
       let "num += 1"
     fi
done
```