Дж. Фуско

Linux

Руководство программиста



PRENTICE HALL

ПИТЕР®

The Linux Programmer's Toolbox

John Fusco



Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City



РУКОВОДСТВО ПРОГРАММИСТА

Дж. Фуско



Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Новосибирск · Ростов-на-Дону · Екатеринбург · Самара
Киев · Харьков · Минск
2011

Дж. Фуско

Linux. Руководство программиста

Перевел с английского В. Иванов

Заведующий редакцией	*А. Буглак*
Руководитель проекта	*К. Галицкая*
Ведущий редактор	*Е. Каляева*
Научный редактор	*Д. Климов*
Литературные редакторы	*А. Вельянинова, Н. Гринчик, Н. Рощина*
Художник	*Л. Адуевская*
Корректоры	*Е. Павлович, Ю. Цеханович*
Верстка	*А. Засулевич*

ББК 32.973.2-018.2
УДК 004.451

Дж. Фуско

Ф95 Linux. Руководство программиста. — СПб.: Питер, 2011. — 448 с.: ил.

ISBN 978-5-49807-794-9

Данное руководство позволит вам освоить обширный перечень инструментов с открытым исходным кодом, доступных для GNU/Linux. Автор книги, Джон Фуско, подробно и доступно описывает наиболее полезные из них, используя наглядные краткие примеры, которые легко видоизменять и использовать на практике.

Начав с самых основ — загрузки, построения и установки, — вы узнаете, как распределены инструменты с открытым кодом и как найти оптимальное решение для той или иной задачи, углубите свои знания о ядре Linux и способах взаимодействия ОС с программным обеспечением. Изложение теоретической информации в книге ориентировано на последующее практическое применение. Освоив ее, вы сможете пользоваться самыми продвинутыми инструментами, применяя их для разработки собственных приложений, а также для увеличения производительности ПО и его отладки.

Права на издание получены по соглашению с Prentice Hall PTR.
Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-5-49807-794-9
ISBN 0132198576 (англ.)

© 2007 Pearson Education, Inc.
© Перевод на русский язык ООО Издательство «Питер», 2011
© Издание на русском языке, оформление ООО Издательство «Питер», 2011

ООО «Лидер», 194044, Санкт-Петербург, Б. Сампсониевский пр., 29а.

Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.

Подписано в печать 24.09.10. Формат 70×100/16. Усл. п. л. 36,12. Тираж 2000. Заказ 0000.

Отпечатано по технологии CtP в ОАО «Печатный двор» им. А. М. Горького.
197110, Санкт-Петербург, Чкаловский пр., 15.

Книга посвящается моей жене Лизе и детям —
Эндрю, Алексу и Саманте.

Краткое содержание

Предисловие 17

Введение 18

Благодарности 22

Об авторе 22

От издательства 22

Глава 1. Загрузка и установка инструментария для работы с открытым исходным кодом 23

Глава 2. Создание приложений на основе исходного кода 53

Глава 3. Поиск справочной документации 96

Глава 4. Редактирование и сопровождение исходных файлов 124

Глава 5. Что должен знать каждый разработчик о ядре Linux 172

Глава 6. Процессы 231

Глава 7. Взаимодействие между процессами 261

Глава 8. Отладка межпроцессного взаимодействия IPC при помощи команд оболочки 308

Глава 9. Настройка производительности 326

Глава 10. Отладка 380

Оглавление

Предисловие 17

Введение 18

Для кого эта книга 19
Цель книги 19
Рекомендации по чтению 20
Структура книги 20

Благодарности 22

Об авторе 22

От издательства 22

Глава 1. Загрузка и установка инструментария для работы с открытым исходным кодом 23

1.1. Введение 23
1.2. Что такое программное обеспечение с открытым исходным кодом? 23
1.3. Что означает программное обеспечение с открытым исходным кодом для обычного пользователя? 24
1.3.1. Поиск инструментария 24
1.3.2. Форматы распространения 25
1.4. Архивные файлы 26
1.4.1. Идентификация архивных файлов 27
1.4.2. Запрос архивного файла 28
1.4.3. Извлечение файлов из архива 31
1.5. Менеджеры пакетов 32
1.5.1. Что лучше: исходный код или двоичные файлы? 34
1.5.2. Работа с пакетами 35
1.6. Кратко о безопасности и пакетах 35

1.6.1. Важность проверки подлинности . . . 36
1.6.2. Базовая проверка подлинности пакетов . . . 36
1.6.3. Проверка подлинности пакетов с использованием цифровых подписей . . . 38
1.6.4. Подпись пакетов RPM с использованием инструмента GPG . . . 38
1.6.5. Что делать, если проверка подлинности пакета невозможна . . . 41
1.7. Проверка содержимого пакетов . . . 42
1.7.1. Как осуществлять проверку пакетов . . . 43
1.7.2. Обзор пакетов RPM . . . 45
1.7.3. Обзор пакетов Debian . . . 46
1.8. Обновление пакетов . . . 47
1.8.1. Инструмент Apt: Advanced Package Tool . . . 48
1.8.2. Инструмент Yum: Yellowdog Updater Modified . . . 49
1.8.3. Synaptic: передовой GUI-интерфейс для APT . . . 49
1.8.4. up2date: инструмент обновления пакетов Red Hat . . . 51
1.9. Заключение . . . 51

Глава 2. Создание приложений на основе исходного кода . . . 53
2.1. Введение . . . 53
2.2. Инструменты для сборки приложений . . . 53
2.2.1. Предыстория . . . 54
2.2.2. Инструмент make . . . 55
2.2.3. Связь программ . . . 71
2.2.4. Понятие библиотек . . . 72
2.3. Процесс сборки . . . 74
2.3.1. GNU-инструменты сборки . . . 74
2.3.2. Этап конфигурирования: сценарий configure . . . 75
2.3.3. Этап сборки: make . . . 76
2.3.4. Этап установки: make install . . . 77
2.4. Понятие ошибок и предупреждений . . . 78
2.4.1. Ошибки, распространенные в файлах Makefile . . . 78
2.4.2. Ошибки на этапе конфигурирования . . . 80
2.4.3. Ошибки на этапе сборки . . . 81
2.4.4. Понятие ошибок компилятора . . . 83
2.4.5. Понятие предупреждений компилятора . . . 85
2.4.6. Понятие ошибок редактора связей . . . 93
2.5. Заключение . . . 95

Глава 3. Поиск справочной документации . . . 96
3.1. Введение . . . 96
3.2. Интерактивные руководства . . . 96

3.2.1. Страницы руководства man . . . 97
3.2.2. Устройство руководства man . . . 97
3.2.3. Поиск страниц руководства man: команда apropos . . . 99
3.2.4. Поиск нужной страницы руководства man: команда whatis . . . 101
3.2.5. Важные разделы страниц руководства man . . . 102
3.2.6. Рекомендуемые страницы руководства man . . . 103
3.2.7. GNU-программа info . . . 105
3.2.8. Просмотр страниц info . . . 105
3.2.9. Поиск страниц info . . . 108
3.2.10. Рекомендуемые страницы info . . . 108
3.2.11. Справочный инструментарий Рабочего стола . . . 109
3.3. Другие источники справочной информации . . . 110
3.3.1. /usr/share/doc . . . 110
3.3.2. Перекрестные ссылки и индексация . . . 110
3.3.3. Запрос пакетов . . . 112
3.4. Форматы документации . . . 113
3.4.1. TeX/LaTeX/DVI . . . 113
3.4.2. Texinfo . . . 113
3.4.3. DocBook . . . 114
3.4.4. HTML . . . 114
3.4.5. PostScript . . . 115
3.4.6. Portable Document Format (PDF) . . . 115
3.4.7. troff . . . 116
3.5. Источники справочной информации в Интернете . . . 116
3.5.1. www.gnu.org . . . 116
3.5.2. SourceForge.net . . . 117
3.5.3. Проект документации The Linux Documentation Project . . . 117
3.5.4. Usenet . . . 118
3.5.5. Списки рассылки . . . 119
3.5.6. Прочие форумы . . . 119
3.6. Поиск информации о ядре Linux . . . 119
3.6.1. Сборка ядра . . . 119
3.6.2. Модули ядра . . . 121
3.6.3. Прочая документация . . . 122
3.7. Заключение . . . 122
3.7.1. Инструментарий, использованный в этой главе . . . 122
3.7.2. Веб-ссылки . . . 123

Глава 4. Редактирование и сопровождение исходных файлов . . . 124

4.1. Введение . . . 124

4.2. Текстовый редактор 124
4.2.1. Важные функциональные возможности текстового редактора 126
4.2.2. Основные текстовые редакторы: vi и Emacs 127
4.2.3. Vim: усовершенствованная версия vi 127
4.2.4. Текстовый редактор Emacs 144
4.2.5. Атака клонов 151
4.2.6. Потребление памяти 153
4.3. Контроль версий 154
4.3.1. Основы контроля версий 155
4.3.2. Терминология в сфере контроля версий 156
4.3.3. Инструменты сопровождения 157
4.3.4. Команды diff и patch 158
4.3.5. Просмотр и объединение изменений 159
4.4. Инструменты для улучшения внешнего вида исходного кода и его просмотра 163
4.4.1. Инструмент для улучшения внешнего вида программного кода indent 164
4.4.2. Художественный стиль astyle 165
4.4.3. Анализ программного кода при помощи cflow 165
4.4.4. Анализ программного кода при помощи ctags 165
4.4.5. Просмотр программного кода при помощи cscope 166
4.4.6. Просмотр и создание документации к программному коду при помощи Doxygen 166
4.4.7. Использование компилятора для анализа программного кода 168
4.5. Заключение 169
4.5.1. Инструментарий, использованный в этой главе 169
4.5.2. Рекомендуемая литература 170
4.5.3. Веб-ссылки 170

Глава 5. Что должен знать каждый разработчик о ядре Linux 172

5.1. Введение 172
5.2. Сравнение режима пользователя и режима ядра 172
5.3. Планировщик процессов 175
5.3.1. Основы планирования 175
5.3.2. Блокировка, вытеснение и уступка 175
5.3.3. Приоритеты и справедливость планирования 176
5.3.4. Приоритеты и значение nice 177
5.3.5. Приоритеты реального времени 177
5.3.6. Создание процессов реального времени 179
5.3.7. Состояния процессов 180

5.3.8. Порядок измерения времени . . . 183
5.4. Понятие устройств и их драйверов . . . 189
5.4.1. Типы драйверов устройств . . . 190
5.4.2. Несколько слов о модулях ядра . . . 191
5.4.3. Узлы устройств . . . 192
5.4.4. Устройства и операции ввода/вывода . . . 197
5.5. Планировщик операций ввода/вывода . . . 202
5.5.1. Элеватор Linus Elevator (также называемый noop) . . . 203
5.5.2. Планировщик операций ввода/вывода «крайнего срока» (Deadline I/O Scheduler) . . . 204
5.5.3. «Ожидающий» планировщик операций ввода/вывода (Anticipatory I/O Scheduler) . . . 204
5.5.4. «Справедливый» планировщик очереди операций ввода/вывода (Complete Fair Queuing I/O Scheduler) . . . 204
5.5.5. Выбор планировщика операций ввода/вывода . . . 205
5.6. Управление памятью в пространстве пользователя . . . 206
5.6.1. Понятие виртуальной памяти . . . 206
5.6.2. Нехватка памяти . . . 219
5.7. Заключение . . . 229
5.7.1. Инструментарий, использованный в этой главе . . . 229
5.7.2. API-интерфейсы, рассмотренные в этой главе . . . 229
5.7.3. Веб-ссылки . . . 230
5.7.4. Рекомендуемая литература . . . 230

Глава 6. Процессы . . . 231
6.1. Введение . . . 231
6.2. Что порождает процессы? . . . 231
6.2.1. Системные вызовы fork и vfork . . . 231
6.2.2. Копирование при записи . . . 232
6.2.3. Системный вызов clone . . . 233
6.3. Функции exec . . . 234
6.3.1. Исполняемые сценарии . . . 234
6.3.2. Исполняемые объектные файлы . . . 236
6.3.3. Разнообразные двоичные файлы . . . 236
6.4. Синхронизация процессов при помощи wait . . . 238
6.5. Объем памяти, занимаемый процессами . . . 239
6.5.1. Дескрипторы файлов . . . 242
6.5.2. Стек . . . 244
6.5.3. Резидентная и блокированная память . . . 246
6.6. Определение лимитов использования ресурсов процессами . . . 246
6.7. Процессы и файловая система procfs . . . 250

6.8. Инструменты для управления процессами 252
6.8.1. Вывод информации о процессах при помощи команды ps. 252
6.8.2. Вывод расширенных сведений о процессах с использованием форматирования 254
6.8.3. Поиск процессов по имени при помощи команд ps и pgrep 256
6.8.4. Просмотр сведений о потреблении памяти процессами при помощи команды pmap 256
6.8.5. Отправка сигналов процессам на основе их имен. 258
6.9. Заключение 259
6.9.1. Системные вызовы и API-интерфейсы, использованные в этой главе 259
6.9.2. Инструментарий, использованный в этой главе 260
6.9.3. Веб-ссылки 260

Глава 7. Взаимодействие между процессами 261

7.1. Введение 261
7.2. Межпроцессное взаимодействие IPC с использованием плоских файлов 261
7.2.1. Блокировка файлов 262
7.2.2. Недостатки подхода, основанного на использовании файлов для обеспечения межпроцессного взаимодействия IPC 262
7.3. Разделяемая память 263
7.3.1. Управление разделяемой памятью при помощи API-интерфейса POSIX 263
7.3.2. Управление разделяемой памятью при помощи API-интерфейса System V 267
7.4. Сигналы 269
7.4.1. Отправка сигналов процессу 270
7.4.2. Обработка сигналов. 271
7.4.3. Маска сигналов и обработка сигналов 272
7.4.4. Сигналы реального времени 275
7.4.5. Расширенные сигналы с использованием функций sigqueue и sigaction 277
7.5. Конвейеры (каналы) 280
7.6. Сокеты 280
7.6.1. Создание сокетов 281
7.6.2. Пример создания локального сокета при помощи функции socketpair 283
7.6.3. Пример пары «клиент/сервер», использующей локальные сокеты . . . 284
7.6.4. Пример пары «клиент/сервер», использующей сетевые сокеты . . . 289
7.7. Очереди сообщений 290
7.7.1. Очередь сообщений System V. 291

7.7.2. Очередь сообщений POSIX 294
7.7.3. Различия между очередями сообщений POSIX и очередями сообщений System V 296
7.8. Семафоры 297
7.8.1. Работа с семафорами с использованием API-интерфейса POSIX . . . 301
7.8.2. Работа с семафорами с использованием API-интерфейса System V . . .304
7.9. Заключение 306
7.9.1. Системные вызовы и API-интерфейсы, использованные в этой главе . . 306
7.9.2. Рекомендуемая литература 307
7.9.3. Веб-ссылки 307

Глава 8. Отладка межпроцессного взаимодействия IPC при помощи команд оболочки 308
8.1. Введение 308
8.2. Инструментарий для работы с открытыми файлами 308
8.2.1. lsof 308
8.2.2. fuser 310
8.2.3. ls 310
8.2.4. file 310
8.2.5. stat 311
8.3. Сброс данных из файла 311
8.3.1. Команда strings 311
8.3.2. Команда xxd 312
8.3.3. Команда hexdump 313
8.3.4. Команда od 313
8.4. Команды оболочки для работы с объектами System V IPC 314
8.4.1. Разделяемая память System V 314
8.4.2. Очереди сообщений System V 316
8.4.3. Семафоры System V 317
8.5. Инструменты для работы с объектами POSIX IPC 318
8.5.1. Разделяемая память POSIX 318
8.5.2. Очереди сообщений POSIX 319
8.5.3. Семафоры POSIX 319
8.6. Инструменты для работы с сигналами 320
8.7. Инструменты для работы с каналами (конвейерами) и сокетами 321
8.7.1. Каналы и FIFO 321
8.7.2. Сокеты 322
8.8. Использование индексных дескрипторов для идентификации файлов и объектов IPC 323
8.9. Заключение 324
8.9.1. Инструментарий, использованный в этой главе 324
8.9.2. Веб-ссылки 325

Глава 9. Настройка производительности 326
9.1. Введение 326
9.2. Производительность системы 326
9.2.1. Аспекты производительности, связанные с оперативной памятью . . . 326
9.2.2. Использование центрального процессора и конкуренция за ресурсы шины 335
9.2.3. Устройства и прерывания 338
9.2.4. Инструменты для выявления причин снижения производительности системы 343
9.3. Производительность приложений 349
9.3.1. Шаг первый: использование команды time 350
9.3.2. Вывод сведений об архитектуре процессора при помощи инструмента x86info 350
9.3.3. Использование Valgrind для проверки эффективности инструкций . 353
9.3.4. Инструмент ltrace 357
9.3.5. Использование strace для мониторинга производительности приложений 358
9.3.6. Традиционные инструменты для настройки производительности: gcov и gprof 359
9.3.7. Инструмент OProfile 365
9.4. Производительность многопроцессорных систем 370
9.4.1. Типы аппаратного обеспечения с поддержкой симметричной многопроцессорной обработки SMP 370
9.4.2. Программирование на компьютере с поддержкой симметричной многопроцессорной обработки SMP 375
9.5. Заключение 378
9.5.1. Аспекты, связанные с производительностью 378
9.5.2. Термины, рассмотренные в этой главе 379
9.5.3. Инструментарий, использованный в этой главе 379
9.5.4. Веб-ссылки 379
9.5.5. Рекомендуемая литература 379

Глава 10. Отладка 380
10.1. Введение 380
10.2. Основной инструмент отладки: printf 380
10.2.1. Проблемы, возникающие при использовании printf 381
10.2.2. Эффективное использование инструмента printf 385
10.2.3. Заключительные положения по отладке с использованием printf . . . 393
10.3. Комфортное использование GNU-отладчика gdb 393
10.3.1. Запуск программ с использованием gdb 394
10.3.2. Остановка и повторный запуск выполнения программы 395

10.3.3. Проверка данных и манипулирование ими . . . 402
10.3.4. Подключение к выполняющемуся процессу при помощи gdb . . . 411
10.3.5. Отладка файлов образов памяти . . . 412
10.3.6. Отладка многопоточных программ при помощи gdb . . . 415
10.3.7. Отладка оптимизированного программного кода . . . 417
10.4. Отладка совместно используемых объектов . . . 418
10.4.1. Когда и зачем необходимы совместно используемые объекты . . . 418
10.4.2. Создание совместно используемых объектов . . . 419
10.4.3. Определение местоположения совместно используемых объектов . . . 420
10.4.4. Переназначение местоположения совместно используемых объектов по умолчанию . . . 421
10.4.5. Безопасность и совместно используемые объекты . . . 421
10.4.6. Инструментарий для работы с совместно используемыми объектами . . . 422
10.5. Поиск проблем с памятью . . . 425
10.5.1. Двойное освобождение . . . 426
10.5.2. Утечки памяти . . . 426
10.5.3. Переполнение буфера . . . 427
10.5.4. Инструментарий, доступный в библиотеке glibc . . . 429
10.5.5. Использование Valgrind для устранения проблем с памятью . . . 432
10.5.6. Выявление переполнения с помощью инструмента Electric Fence . . . 437
10.6. Использование нестандартных методик . . . 439
10.6.1. Создание собственного «черного ящика» . . . 439
10.6.2. Получение трасс стека процессов во время их выполнения . . . 443
10.6.3. Принудительное генерирование файлов образов памяти . . . 443
10.6.4. Использование сигналов . . . 444
10.6.5. Использование procfs при проведении отладки . . . 445
10.7. Заключение . . . 447
10.7.1. Инструментарий, использованный в этой главе . . . 448
10.7.2. Веб-ссылки . . . 448
10.7.3. Рекомендуемая литература . . . 448

Предисловие

Итак, вы знаете основы Linux. Вы уже умеете применять утилиты `ls`, `grep`, `find` и `sort` и, как программист на C или C++, можете использовать системные вызовы Linux. Однако в мире Linux существует еще множество других вещей, которые вам предстоит изучить. Вы просто еще не знаете, как это сделать, и наверняка задаетесь вопросом: «К чему переходить дальше?» Ответ вы найдете в этой книге.

Джон — опытный программист, который помогает пользователям Linux с определенным уровнем знаний подняться на ступеньку выше на пути превращения в профессионалов.

Джон демонстрирует, как можно использовать все, начиная с инструментов отладки и анализа эффективности и заканчивая файловой системой `/proc`, чтобы облегчить повседневную работу с операционной системой Linux и повысить ее производительность.

В этой книге Джон не только отвечает на вопросы, какие инструменты, параметры, файлы используются, но и объясняет, почему что-либо работает именно так, как оно работает. Это, в свою очередь, позволяет читателю понять эффективность системы и постичь самую суть Linux (и UNIX!).

В этой книге собрано множество полезного материала. И я надеюсь, что вы узнаете много нового. Я сам почерпнул из нее немало, а это уже кое о чем говорит.

Приятного чтения.

Арнольд Роббинс (Arnold Robbins),
редактор серии

Введение

Операционная система Linux обладает обширным инструментарием. Многое было унаследовано от UNIX, включая малопонятные двухбуквенные имена, благодаря которым разработчики пытались сэкономить свободное место на перфокартах. К счастью, все это осталось в далеком прошлом, однако наследие все еще живо.

Многие из старых инструментов по-прежнему эффективны. Большинство из них узкоспециализированны. Каждый инструмент используется только для одной задачи, но выполняет ее с большой отдачей. Зачастую узкоспециализированные инструменты обладают множеством параметров, что может отпугнуть пользователя. Допустим, вы впервые воспользовались утилитой `grep` и узнали, что такое регулярное выражение. Вероятно, вы пока еще не освоили синтаксис регулярных выражений (не беспокойтесь, в этом вы не одиноки). Это не имеет значения, поскольку вам не нужно быть специалистом по регулярным выражениям, чтобы эффективно работать с утилитой `grep`.

Есть еще один момент, который, как я надеюсь, вы усвоите из этой книги: в Linux существует много инструментов, применение которых на практике не потребует от вас профессионального владения ими. Чтобы стать хорошим разработчиком, вам не нужно тратить уйму времени на изучение справочных материалов. Я надеюсь, что вы откроете для себя немало новых инструментов. Описанные в этом издании, одни из них являются довольно старыми, другие — новыми. Но все они вам пригодятся. Чем больше вы будете узнавать о каждом из них, тем более полезными они будут становиться для вас.

Термин *инструмент* в этой книге я использую в широком смысле. Для меня создание инструментов так же важно, как и их использование, поэтому я включил в материал разные интерфейсы программирования приложений API, которые в других изданиях обычно не рассматриваются. Кроме того, эта книга также содержит сведения о внутренней работе ядра Linux, которые помогут вам лучше понять суть отдельных изучаемых инструментов. Ядро рассматривается с особой стороны: с точки зрения пользователя. Здесь вы найдете достаточно информации, которая позволит вам усвоить основные правила, определяемые ядром для каждого процесса, и я уверяю вас, что при этом вам абсолютно не придется иметь дело с программной частью ядра.

В данной книге вы не найдете переработанных в обычный текст страниц MAN-руководств или прочей документации. Разработчики GNU и Linux проделали огромную работу по документированию своих проектов, однако неопытному пользователю может быть сложно найти эту информацию. Вместо того чтобы перепечатывать документацию, которая на момент выхода книги может оказаться уста-

ревшей, я расскажу вам о нескольких оригинальных способах отыскать наиболее актуальную информацию такого рода.

Документация по GNU/Linux весьма обширна, но не всегда удобна для чтения. Вы можете прочесть о каком-либо инструменте материал в 10 000 слов, но так и не понять, для чего нужен этот инструмент и как его использовать. Именно здесь я постарался заполнить пробелы. Я буду рассказывать не только о том, как пользоваться конкретным инструментом, но и почему им нужно пользоваться. Везде, где это возможно, я буду приводить простые, краткие примеры, которые вы сможете самостоятельно набрать вручную и модифицировать с целью лучшего понимания сути описываемого инструмента и всей системы Linux в целом.

Все инструменты, рассматриваемые в этой книге, объединяет одно свойство — они бесплатны. Большинство стандартных дистрибутивов Linux уже содержат их внутри, а если они отсутствуют, то я привожу соответствующие URL-ссылки, по которым их можно загрузить.

Насколько это было возможным, я постарался преподнести содержимое этой книги в интересной и развлекательной форме.

Для кого эта книга

Данная книга предназначена для Linux-программистов среднего и высокого уровня, которые стремятся повысить свою квалификацию и лучше понять суть среды программирования Linux. Если вы являетесь опытным программистом на платформе Windows, то в среде Linux вам поначалу будет очень сложно ориентироваться, поэтому эта книга также адресована вам.

Для непрограммистов данная книга также может оказаться полезной, поскольку многие рассматриваемые инструменты и темы применимы и за пределами сферы программирования. Кроме того, много интересного найдут для себя системные администраторы и энтузиасты Linux.

Цель книги

Написанием этой книги я занялся после того, как была опубликована моя статья в журнале *Linux Journal* под названием «Десять команд, которые должен знать программист Linux». К созданию этой статьи меня подтолкнул собственный опыт программирования на платформе Linux. В повседневной работе часть времени я посвящал изучению чего-то нового, даже если это приводило к временной приостановке разрабатываемых проектов. В итоге подобная стратегия принесла свои плоды. Меня всегда удивляло, что если я принимался за изучение какого-либо инструмента или свойства, первоначально казавшихся малопригодными, то через короткое время я отмечал, что они оказывались вполне полезными в работе. Это всегда служило хорошей мотивацией для освоения всего нового. Надеюсь, что, читая эту книгу, вы последуете моему примеру и станете постоянно совершенствовать свои навыки.

Освоение чего-то нового может стать и развлечением. Если вы похожи на меня, то вам должно нравиться работать с Linux. Собственная мотивация на приобретение

новых знаний никогда не была проблемой. Поскольку Linux — система с открытым исходным кодом, вам предоставляется возможность понять ее внутреннее устройство, чего нельзя сделать в случае с системами с закрытым кодом, например Windows.

В этой книге, кроме всего прочего, вы найдете ссылки на бесплатные ресурсы во Всемирной паутине, которые помогут вам пополнить багаж знаний.

Рекомендации по чтению

Каждая глава посвящена отдельной теме. При освоении последующих глав потребуется знание материала, изложенного в предыдущих главах. Там, где было возможно, я привел перекрестные ссылки на материал, чтобы вы могли отыскать необходимые исходные сведения.

Поскольку я считаю, что лучший способ изучить что-либо — это больше практиковаться, то постарался по возможности приводить простые примеры. Я очень одобряю, когда читатель практикуется на примерах и экспериментирует.

Структура книги

Глава 1 под названием «Загрузка и установка инструментария для работы с открытым исходным кодом» посвящена механизму распространения открытого исходного кода. Здесь рассматриваются разнообразные пакетные форматы, используемые в различных дистрибутивах, их преимущества и недостатки. Вы познакомитесь с несколькими инструментами, предназначенными для обслуживания пакетов, и научитесь пользоваться ими.

В главе 2, которая называется «Создание приложений на основе исходного кода», рассматриваются основы построения проекта с открытым исходным кодом. Здесь я расскажу вам об инструментарии для создания программ, а также об имеющихся альтернативах. В этой главе приведены советы и приемы использования команды `make`. Вы также узнаете, как осуществлять конфигурирование в соответствии с требованиями своих проектов, которые распространяются вместе с утилитами GNU `autoconf`. В завершение мы рассмотрим стадии сборки программ, которые многие программисты часто неверно себе представляют. Я приведу примеры ошибок и предупреждений, с которыми вы можете столкнуться, а также научу вас интерпретировать их.

В главе 3 под названием «Поиск справочной документации» мы поговорим о различных форматах справочных сведений, используемых в дистрибутивах Linux, которые могут быть новыми для вас. Вы узнаете об инструментах для чтения этих форматов и научитесь эффективно применять их на практике.

Глава 4, которая называется «Редактирование и сопровождение исходных файлов», посвящена разного рода текстовым редакторам, используемым программистами, а также их преимуществам и недостаткам. Здесь приводится набор свойств, которыми, по мнению программиста, должен обладать текстовый редактор, и на их фоне анализируется каждый текстовый редактор. В этой главе также излагаются

основы ревизионного контроля, который очень важен при работе над программным проектом.

В главе 5 «Что должен знать каждый разработчик о ядре Linux» ядро рассматривается с точки зрения пользователя. Здесь излагаются сведения, позволяющие понять принципы функционирования операционной системы Linux. Вы также познакомитесь с новыми инструментами, дающими возможность взглянуть на процесс взаимодействия программного кода с ядром.

Глава 6 под названием «Процессов» сфокусирована на процессах, их характеристиках и управлении ими. В этой главе приводится много исходного материала для знакомства с новыми инструментами и их полезными свойствами. Кроме того, здесь вы также узнаете о некоторых интерфейсах программирования приложений API, которые можно использовать для создания собственного инструментария.

В главе 7, имеющей название «Взаимодействие между процессами», рассматривается суть межпроцессного взаимодействия IPC. По большей части эта глава содержит материал, необходимый для понимания всего изложенного в главе 8. Вместе с механизмом межпроцессного взаимодействия IPC представлены интерфейсы программирования приложений API, используемые для работы с практическими примерами.

В главе 8 под названием «Отладка межпроцессного взаимодействия IPC при помощи команд оболочки» рассмотрены некоторые инструменты для отладки приложений, использующих межпроцессное взаимодействие IPC. Материал этой главы основан на информации из главы 7, что позволит вам интерпретировать полученный с помощью этих инструментов результат, который иногда бывает трудно понять.

В главе 9, которая называется «Настройка производительности», вы узнаете об инструментах, позволяющих измерять производительность как вашей системы, так и отдельных приложений. Вы увидите несколько примеров, иллюстрирующих, как программная часть может влиять на уровень производительности. Здесь также рассматриваются вопросы производительности в системах с многоядерными процессорами.

Из главы 10 под названием «Отладка» рассказывается о наборе инструментов и методиках для отладки приложений. Здесь рассмотрены средства отладки памяти, включая Valgrind и Electric Fence. Кроме того, подробно рассказывается об отладчике `gdb` и о том, как его можно эффективно использовать.

Благодарности

Благодарю свою жену Лизу, без которой данная книга вообще не увидела бы свет. Пока я был занят работой над этой книгой, она провела много времени, одна воспитывая наших детей. Только благодаря ей я смог осуществить задуманное. Хочу также высказать признательность своим детям — Эндрю, Алексу и Саманте, с которыми мы нечасто общались, пока я писал книгу.

Я благодарю Арнольда Роббинса (Arnold Robbins) за его советы и содействие. Его опыт и впечатляющие знания оказали мне неоценимую помощь при работе над книгой. Я очень признателен ему за то, что создание книги превратилось в увлекательный процесс познания нового.

Хочу также выразить благодарность Дебре Уильямс Коли (Debra Williams Cauley) за ее терпение, которое она проявила, несмотря на сорванный график и смещение сроков выхода книги. Я, как начинающий писатель, признателен ей за то, что все получилось как надо.

И, наконец, благодарю Марка Тоба (Mark Taub) за то, что он нанял меня и предоставил замечательную возможность написать эту книгу.

Об авторе

Джон Фуско (John Fusco) работает программистом в компании GE Healthcare, расположенной в городе Вокеша, штат Висконсин, и специализируется на приложениях и драйверах устройств для операционной системы Linux. Джон более десяти лет работал UNIX-программистом и занимается разработкой программ «под Linux» с момента выхода ядра версии 2.0. Он является автором статей, опубликованных в изданиях *Embedded Systems Programming* и *Linux Journal*.

От издательства

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты gromakovski@minsk.piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На сайте издательства http://www.piter.com вы найдете подробную информацию о наших книгах.

1 Загрузка и установка инструментария для работы с открытым исходным кодом

1.1. Введение

В этой главе мы рассмотрим различные форматы, в которых распространяется бесплатное программное обеспечение. Кроме того, вы узнаете, как с ним обращаться и где его можно отыскать. Я подробно расскажу вам об архивных и пакетных файлах, а также о наиболее используемых командах для управления ими.

Программное обеспечение от незнакомых разработчиков может таить в себе опасность, поэтому я расскажу вам о вопросах безопасности, которые необходимо принять во внимание, и о том, что можно сделать, чтобы защитить себя. Вы узнаете, что такое проверка подлинности и доверие и каким образом они относятся к безопасности. В тех случаях, когда проверка подлинности невозможна, я расскажу, как можно самому проверять пакеты и архивы.

В завершение главы мы поговорим об инструментах для работы с пакетными дистрибутивами и научимся пользоваться ими.

1.2. Что такое программное обеспечение с открытым исходным кодом?

Термин *программное обеспечение с открытым исходным кодом* — маркетинговое название бесплатных программ, создаваемых сообществом Open Source Initiative (OSI) (www.opensource.org). Эта организация была создана с целью пропаганды принципов бесплатного программного обеспечения, которые корнями уходят в проект GNU Project, основанный Ричардом Столманом (Richard Stallman). Одной из важнейших целей организации Open Source Initiative является борьба с негативными стереотипами о бесплатных программах, а также продвижение идеи бесплатного распространения исходного кода.

Поначалу многие компании не решались использовать программное обеспечение с открытым исходным кодом. Несомненно, причиной этому послужили определенные старания отделов маркетинга некоторых крупных разработчиков программ. Есть такое хорошее выражение: «Вы получаете ровно столько, за сколько заплатили». Некоторые фирмы опасались, что лицензионные соглашения (вроде GNU Public License) могут оказаться ловушкой, то есть если вы при создании своих проектов используете бесплатное программное обеспечение, то вам придется сделать их исходный код открытым.

К счастью, большинство этих опасений оказались напрасными. Многие крупные компании бесплатно пользуются идеями программного обеспечения с открытым исходным кодом в своих проектах и продвигают их. Некоторые из них полностью основывают свои продукты на этой идее. Таким образом, джинн был выпущен из бутылки.

1.3. Что означает программное обеспечение с открытым исходным кодом для обычного пользователя?

Для большинства людей программное обеспечение с открытым исходным кодом означает большое количество отличных приложений, за пользование которыми не нужно платить. К сожалению, наряду с действительно хорошими программами существуют и не столь качественные продукты, но это всего лишь издержки процесса. Хорошие программные идеи процветают и развиваются, а плохие — увядают и умирают. Выбор программ с открытым исходным кодом отчасти напоминает выбор фруктов: чтобы определить, зрелые ли они, надо обладать некоторыми знаниями.

Процесс естественного отбора происходит на многих уровнях. Так, на уровне исходного кода отбору подвергается программная часть (на основе «заплаток»), то есть в основе приложений — только качественный программный код. Вы, как пользователь, решаете, какие проекты хотите использовать, а это уже влияет на их дальнейшее существование. Ни один разработчик не станет поддерживать проект, который никому не нужен. Чем меньшим спросом пользуется какая-либо программа, тем меньше подобных приложений будет появляться. Если же приложение становится популярным, то это привлекает разработчиков, которые создадут больше программных продуктов, то есть появится возможность выбрать самые качественные их них. Иногда выбор какого-либо программного проекта напоминает азартную игру, однако в ней вы рискуете лишь своим временем и силами. Ошибки при выборе неизбежны, но не стоит расстраиваться, поскольку это всего лишь издержки процесса.

1.3.1. Поиск инструментария

Перед тем как отправляться во Всемирную паутину, необходимо изучить компакт-диски с дистрибутивами. Если для установки Linux вы использовали несколь-

ко компакт-дисков или один DVD, скорее всего, вам еще предстоит инсталляция множества дополнительных инструментов. На компакт-дисках с дистрибутивами зачастую имеется много программ, которые не устанавливаются по умолчанию. Обычно при установке операционной системы вы можете выбирать компоненты для инсталляции, чтобы в итоге наделить систему необходимыми возможностями. Для этого приходится устанавливать набор различных пакетов в своей системе в зависимости от того, будет это «рабочая станция» или «сервер».

Вы всегда можете установить дополнительные программы вручную, воспользовавшись исходными пакетами на компакт-дисках. Помехой здесь может стать то, что обычно пакеты не располагаются в определенном порядке, поэтому вам самим придется их искать. Некоторые дистрибутивы снабжаются графическим интерфейсом, в котором пакеты распределены по категориям, что облегчает выбор программ для установки.

Если вы не сможете отыскать нужный пакет, то отправляйтесь в Интернет. Некоторые сайты — настоящие порталы программного обеспечения с открытым исходным кодом. Один из таких расположен по адресу www.freshmeat.net. Здесь вы найдете программы, рассортированные по категориям, что позволит легко отыскать то, что вам нужно. Так, например, при работе над этой книгой я использовал сайт www.freshmeat.net для поиска по значению *текстовый процессор* и в результате нашел 71 проект. Только представьте: на выбор вам предоставляется 71 текстовый процессор!

Ресурс www.freshmeat.net позволяет осуществлять фильтрацию результатов, чтобы оптимизировать количество найденных ссылок. Среди тех, что были найдены в моем случае, были ссылки не только на Linux, но и на другие операционные системы, а также на проекты, находящиеся на разных стадиях разработки. Я ограничил диапазон поиска проектами, которые поддерживаются системой Linux, полностью завершены и подразумевают лицензионное соглашение, одобренное организацией Open Source Initiative (поисковик www.freshmeat.net также выдает ссылки на коммерческое программное обеспечение). В результате я получил 12 ссылок на проекты — с таким количеством намного легче управиться. Бегло просмотрев их, я обнаружил, что большинство из них — не совсем то, что я на самом деле искал, поскольку поисковая система слишком широко трактует понятие *текстовый процессор*. Применив дополнительные фильтры, я смог отыскать несколько новых известных и качественных проектов, например AbiWord, а также ряд других, о которых мне не доводилось слышать ранее. Нужно отметить, что среди них не было таких текстовых процессоров, как, например, OpenOffice, который я использую в этой книге. Как оказалось, OpenOffice я не нашел потому, что в поисковик надо было вводить не словосочетание «текстовые процессоры», а фразу «Office/Business :: Office Suites». Мораль такова: если вы не можете найти то, что нужно, продолжайте искать.

1.3.2. Форматы распространения

После того как вы нашли требуемое программное обеспечение, вам приходится сделать новый выбор. Зрелые проекты обычно представлены в виде готовых к установке пакетов, имеющих один или несколько форматов. Менее продуманные

проекты по большей части содержат только архив с исходным кодом или двоичными файлами. Так проекты легко разграничивать. Выбор программного пакета напоминает покупку нового автомобиля: не нужно знать, как он работает; вы просто поворачиваете ключ зажигания, и машина заводится. А если вы решите загрузить из Интернета архив с исходным кодом или двоичными файлами, то это будет сродни покупке подержанной машины: хорошо, если вы разбираетесь в автомобилях, иначе вы рискуете получить далеко не то, что хотите на самом деле.

Альтернатива пакетным файлам — архивные файлы, которые в случае с проектами под Linux обычно имеют формат TAR. Архивный файл представляет собой коллекцию файлов, упакованных в один файл с помощью архиватора, например команды `tar`. Обычно файлы сжимаются с помощью программы gzip, чтобы сэкономить дисковое пространство. Такие файлы часто называют *TAR-файлами* или *тарболами*.

1.4. Архивные файлы

На определенном этапе загрузки и установки программного обеспечения с открытым исходным кодом вы будете сталкиваться с архивными файлами различных форматов. *Архивным файлом* называется файл, содержащий коллекцию других файлов. Если вы работаете в Windows, то вам, несомненно, знаком архиватор этой операционной системы — PKZip. Утилиты архивирования в Linux функционируют аналогичным образом (за тем исключением, что, в отличие от PKZip, они не выполняют сжатие). Вместо этого они концентрируются на архивации, а процедуру сжатия файлов поручают другой программе (обычно gzip или bzip2). Такова философия UNIX.

В Linux вы можете пользоваться разными архиваторами, но, поскольку вы работаете со свободным программным обеспечением, нужно использовать то, что дают. Несмотря на то что вы, скорее всего, будете иметь дело только с TAR-файлами, вам не помешает знать и о других средствах архивации.

Утилиты архивации позволяют сохранять не только имена файлов и данные. Помимо путей к файлам и данных в архиве, сохраняются *метаданные* всех файлов. Они содержат сведения о владельце файла, группе и прочие атрибуты (например, разрешения на чтение/запись/исполнение).

В табл. 1.1 перечислены самые распространенные в операционной системе Linux инструменты архивации. Самый популярный формат архивов — TAR. Его название происходит от сокращения словосочетания *tape archive*, оставшегося в наследство от инструмента резервирования данных tape. В настоящее время программа tar является средством общего назначения для архивации наборов файлов в один файл. В качестве альтернативы для архивации данных можно применять утилиту cpio, но при этом синтаксис будет совершенно другим. Кроме того, существует архиватор pax стандарта POSIX, который умеет работать с файлами формата TAR, CPIO и PAX. Мне никогда не доводилось сталкиваться с архивами формата PAX, но я решил упомянуть его для полноты картины.

Таблица 1.1. Самые распространенные архиваторы

Архиватор	Примечание
tar	Самый популярный
cpio	Служит для работы с форматом RPM; больше нигде не используется
ar	Служит для создания пакетных файлов Debian; также применяется для создания программных библиотек. Файлы формата AR не содержат сведений о файловых путях

Стоит рассказать еще об одном архиваторе — ar, который часто используется для создания библиотек объектного кода, применяемых при разработке программного обеспечения, а также для создания пакетных файлов дистрибутива Debian.

Кроме того, существуют утилиты для работы с файлами формата ZIP, создаваемыми с помощью программы PKZip, а также менее известные архиваторы, например LHA. Однако на самом деле при распространении программ с открытым исходным кодом под Linux эти форматы никогда не используются. Если вы встретите архив ZIP, то он, скорее всего, будет предназначен для операционной системы от компании Microsoft.

1.4.1. Идентификация архивных файлов

Размещаемые в Интернете файлы часто сжаты, чтобы на их загрузку уходило меньше времени. Сжатые файлы имеют соответствующие обозначения; в табл. 1.2 приведены некоторые из них.

Таблица 1.2. Обозначения архивов

Расширение	Тип
.tar	Архив TAR, несжатый
.tar.gz, .tgz	Архив TAR, сжатый с помощью gzip
.tar.bz2	Архив TAR, сжатый с помощью bzip2
.tar.Z, .taz	Архив TAR, сжатый с помощью команды UNIX compress
.ar, .a	Архив AR, используется преимущественно при разработке программного обеспечения
.cpio	Архив CPIO, несжатый

Если вы не можете точно идентифицировать файл, воспользуйтесь командой `file`. Она позволит вам идентифицировать файл, если его имя ни о чем вам не говорит. Это особенно полезно, когда браузер или другая программа искажает имя файла до неузнаваемости. Допустим, у вас имеется сжатый TAR-архив с именем `foo.x`. Из имени этого файла ничего нельзя понять о его содержимом. Воспользуемся следующей командой:

```
$ file foo.x
foo.x: gzip compressed data, from UNIX, max compression
```

Теперь мы знаем, что файл сжат архиватором gzip, однако непонятно, имеет ли он формат TAR. Можно попытаться разархивировать его с помощью gzip и еще раз применить команду `file` либо просто добавить в команду параметр `-z`:

```
$ file -z foo.x
foo.x: tar archive (gzip compressed data, from UNIX, max compression)
```

Теперь нам известны все подробности об этом файле.

Обычно пользователи интуитивно подходят к обозначениям файлов, а имена файлов позволяют точно определять тип архива и то, какой обработке они подверглись при сжатии.

1.4.2. Запрос архивного файла

Содержимое архивных файлов можно отслеживать посредством оглавления, доступ к которому довольно легко получить, добавив флаг `-t` к любому из уже упоминавшихся архиваторов. Далее приведен пример с TAR-файлом установки Debian с использованием команды `cron`:

```
$ tar -tzvf data.tar.gz
drwxr-xr-x root/root         0 2001-10-01 07:53:19 ./
drwxr-xr-x root/root         0 2001-10-01 07:53:15 ./usr/
drwxr-xr-x root/root         0 2001-10-01 07:53:18 ./usr/bin/
-rwsr-xr-x root/root     22460 2001-10-01 07:53:18 ./usr/bin/crontab
drwxr-xr-x root/root         0 2001-10-01 07:53:18 ./usr/sbin/
-rwxr-xr-x root/root     25116 2001-10-01 07:53:18 ./usr/sbin/cron
```

В приведенном примере параметр `-v` использован для включения дополнительной информации, аналогичной *длинному* листингу после ввода команды `ls`. В полученном выводе в первом столбце мы видим разрешения файла, во втором — владение. Затем идет размер файла (в байтах), при этом размер каталогов равен `0`. При проверке архивов всегда следует уделять особое внимание владению и разрешениям каждого файла.

В табл. 1.3 приведены основные команды, используемые для просмотра содержимого архивов различных форматов. Все три формата будут иметь одинаковый итоговый вывод.

Таблица 1.3. Команды для запроса архивных файлов

Формат	Команда	Примечание
TAR	tar -tvf filename	—
Архив TAR, сжатый с помощью gzip	tar -tzvf filename	—
Архив TAR, сжатый с помощью bzip2	tar -tjvf filename	—
Архив CPIO	cpio -tv < filename	CPIO использует stdin и stdout как двоичные потоки

Читать символические представления разрешений файлов станет значительно проще, когда вы привыкнете к ним. А приемы, использованные для отображения

дополнительных сведений до и после обычных разрешений на чтение/запись/исполнение, вам уже должны быть знакомы.

Рассмотрим строку разрешений. Она состоит из десяти символов. Первый символ указывает на тип файла, а остальные три группы из трех символов резюмируют разрешение владельца файла, разрешения членов группы и разрешения других лиц соответственно.

Тип файла определяется одним символом. В табл. 1.4 приведены допустимые значения этого символа и пояснения к ним.

Таблица 1.4. Типы файлов в архивных листингах

Символ	Пояснение	Примечание
-	Обычный файл	Сюда входят текстовые файлы, файлы данных, исполняемые файлы и т. д.
d	Каталог	
c	Символьное устройство	Особый файл, используемый для взаимодействия с драйвером символьного устройства. Традиционно эти файлы сосредоточены в каталоге /dev; в архивах они обычно отсутствуют
b	Блочное устройство	Особый файл, применяемый для взаимодействия с драйвером блочного устройства. По традиции эти файлы сосредоточены в каталоге /dev; в архивах они обычно отсутствуют
l	Символическая ссылка	Имя файла, указывающее на другое имя файла. Файл, на который оно указывает, может располагаться в другой файловой системе либо вообще не существовать

Следующие девять символов можно разбить на три группы, состоящие из трех битов каждая. Каждый бит означает разрешение файла на чтение, запись и исполнение соответственно и представлен в виде символов `r`, `w` и `x`. Символ - на месте бита указывает на то, что разрешение не определено. Например, наличие символа - вместо символа `w` говорит о том, что файл не имеет разрешения на запись. В табл. 1.5 можно увидеть несколько подобных примеров.

Таблица 1.5. Примеры битов разрешений файла

Разрешение	Описание
rwx	Файл имеет разрешение на чтение, запись и исполнение
rw-	Файл имеет разрешение на чтение и запись, но не имеет разрешения на исполнение
r-x	Файл имеет разрешение на чтение и исполнение, но не имеет разрешения на запись
--x	Файл имеет разрешение на исполнение, но не имеет разрешения на запись и чтение

Что касается разрешений, то здесь следует упомянуть о битах `setuid`, `setgid` и `sticky`. Они непосредственно не отображаются, поскольку влияют на поведение файлов только при исполнении.

Если бит setuid установлен, то программный код, содержащийся в файле, будет исполняться, при этом владелец файла выступает как эффективный идентификатор пользователя. Это означает, что программа может выполнять любые действия, разрешенные владельцу файла. Если владельцем файла является корневой пользователь root и установлен бит setuid, то программа может модифицировать или удалять любой файл в системе вне зависимости от того, какой пользователь ее запускает. Звучит угрожающе, не правда ли? В прошлом программы с битом setuid часто подвергались атакам.

Бит setgid выполняет аналогичную роль за тем исключением, что исполнение программного файла осуществляется только с привилегиями группы, к которой он принадлежит. Обычно исполнение программы происходит с привилегиями группы, к которой относится пользователь, ее запустивший. Когда установлен бит setgid, программа работает с такими привилегиями, как если бы пользователь принадлежал к той же самой группе.

Понять, какой бит — setuid или setgid — установлен для файла, можно, взглянув на бит x в строке разрешений. Обычно здесь бит x означает, что файл имеет разрешение на исполнение, а символ - указывает на то, что файл не имеет разрешения на исполнение.

Благодаря битам setuid или setgid данный символ может иметь еще два других значения. Если вместо x в строке разрешений владельца файла поставить символ s, то это будет означать, что установлен бит setuid и владелец данного файла имеет право исполнять этот файл. Если же использовать символ S, то будет установлен бит setuid, однако владелец не будет иметь разрешения на исполнение файла. Это может показаться необычным, однако такое положение вещей вполне допустимо, хотя может таить в себе опасность. Например, владельцем файла может быть корневой пользователь root, однако при этом он может и не иметь разрешения на исполнение этого файла. Linux наделяет корневого пользователя root разрешением на исполнение, если *кто-либо* другой обладает подобным разрешением. Таким образом, даже если бит исполнения не установлен для корневого пользователя root, а текущий пользователь имеет разрешение на исполнение, то программа будет исполняться с привилегиями корневого пользователя root.

Как и в случае с битом setuid, бит setgid указывается посредством подстановки различных символов вместо x в группе разрешений. Использование символа s говорит о том, что для файла установлен бит setgid и члены группы имеют разрешение на исполнение этого файла. Если же использован символ S, то это означает, что установлен бит setgid, но члены группы не обладают разрешением на исполнение файла.

В приведенном ранее в этой главе выводе после использования команды cron вы видели, что crontab — программа setuid, которая находится во владении корневого пользователя root. В табл. 1.6 представлены дополнительные разрешения с соответствующими пояснениями.

Существует также бит sticky, который является пережитком прошлого. Изначально он предназначался для сохранения кода какой-либо программы на диске подкачки, чтобы ускорять ее загрузку. В системе Linux бит sticky используется только для каталогов, где имеет совершенно другое назначение. Когда вы наделяете других пользователей разрешениями на запись и исполнение в каталоге, вла-

дельцем которого являетесь, они получают право создавать и удалять из него файлы. Но вы можете лишить их привилегии, позволяющей удалять из этого каталога файлы других пользователей. Обычно, если пользователь обладает разрешением на запись в каталоге, он может удалять из него не только те файлы, которыми владеет, но и любые другие. Вы можете запретить ему делать это, установив для каталога бит `sticky`. После этого пользователи смогут удалять только те файлы, которые принадлежат им. Как обычно, владелец каталога и корневой пользователь root смогут удалять любые файлы. Именно с этой целью в большинстве систем бит `sticky` устанавливается для каталога /tmp.

Таблица 1.6. Примеры разрешений с пояснениями

Строка разрешения	Разрешение на исполнение	Эффективный идентификатор пользователя	Эффективный идентификатор группы
-rwxr-xr-x	Файл может исполняться всеми пользователями	Текущий пользователь	Текущий пользователь
-rw-r-xr-x	Файл может исполняться всеми членами файловой группы за исключением владельца; файл может исполняться всеми пользователями за исключением владельца	Текущий пользователь	Текущий пользователь
-rwsr-xr-x	Файл может исполняться всеми пользователями	Владелец файла	Текущий пользователь
-rwSr-xr-x	Файл может исполняться всеми пользователями за исключением владельца	Владелец файла	Текущий пользователь
-rwxr-sr-x	Файл может исполняться всеми пользователями	Текущий пользователь	Владелец группы
-rwsr-sr-x	Файл может исполняться всеми пользователями	Владелец файла	Владелец группы
-rwsr-Sr-x	Файл может исполняться всеми пользователями, включая владельца, за исключением членов файловой группы	Владелец файла	Владелец группы

В разрешении на исполнение для других пользователей каталог с битом `sticky` указывается с помощью символа `t` или `T`. Например:

- `-rwxrwxrwt` — все пользователи имеют разрешение на чтение и запись в каталоге, для которого установлен бит `sticky`;
- `-rwxrwx--T` — только владелец и члены группы имеют разрешение на чтение или запись в каталоге, для которого установлен бит `sticky`.

1.4.3. Извлечение файлов из архива

Теперь, когда вы научились проверять содержимое архивов, пора переходить к извлечению из них файлов. В табл. 1.7 вы найдете базовые команды для выполнения этой задачи.

Таблица 1.7. Команды для извлечения файлов из архивов

Формат	Команда	Примечание
TAR	tar -xf filename	Данная команда по умолчанию извлекает файлы в текущий каталог
Архив TAR, сжатый с помощью gzip	tar -xzf filename	
Архив TAR, сжатый с помощью bzip2	tar -xjf filename	
Архив CPIO	cpio -i -d < filename	Следует проявлять внимательность в отношении полных путей
Архив AR	ar x filename	Файлы не содержат сведений о путях

Несмотря на то что извлечение файлов из архивов — безопасная операция, вам необходимо проявлять бдительность в отношении путей, чтобы при распаковке не перезаписать какие-либо данные в своей системе. Так, например, CPIO полностью сохраняет пути в корневой каталог. Это значит, что, если вы попытаетесь извлечь файлы из архива формата CPIO, в котором есть каталог с файлами /etc, вы можете нечаянно удалить важные файлы. Допустим, архив формата CPIO содержит несколько файлов, среди которых есть файл `/etc/hosts`. Если попробовать извлечь файлы из этого архива, то файл `/etc/hosts`, который уже есть в вашей системе, будет перезаписан. Убедиться в этом можно, запросив архив:

```
cpio -t < foo.cpio
/etc/hosts
```

Знак / в самом начале говорит о том, что при распаковке архива будет предпринята попытка перезаписать *именно* файл `/etc/hosts`, а не какой-то другой. Иными словами, если вы извлекаете файлы только с целью проверки, вам вряд ли будет нужно, чтобы при этом они перезаписали аналогичные файлы, которые уже присутствуют в вашей системе. Нужно использовать параметр GNU `--no-absolute-filenames`, и все файлы `hosts` будут извлечены по адресу `./etc/hosts`.

К счастью, столкнуться с архивами CPIO вы можете лишь в пакетных файлах RPM, однако данный формат всегда использует пути в текущий каталог, то есть риск случайной перезаписи системных файлов исключен.

Следует отметить, что некоторые версии TAR-архивов, встречающиеся в UNIX, также позволяют сохранять полные пути. GNU-версия архиватора TAR в Linux автоматически убирает начальный символ / у файлов, извлекаемых из архива TAR. Так что, если вы будете иметь дело с TAR-файлом из UNIX, GNU-версия TAR поможет вам избежать ошибки. Кроме того, она также удаляет начальный символ / из путей в архивах, которые сама создает.

1.5. Менеджеры пакетов

Менеджеры пакетов — это комплексные инструменты, используемые для инсталляции и сопровождения программного обеспечения в операционной системе.

Они помогают отслеживать, какие установлены программы и где располагаются файлы. Менеджеры пакетов позволяют также отслеживать зависимости, чтобы гарантировать совместимость нового программного обеспечения с уже установленным. Если вы, например, попытаетесь установить пакет KDE в среде GNOME, то менеджер пакетов будет «ругаться», упирая на то, что у вас нет необходимых программных библиотек исполнения. Согласитесь, это лучше, чем, установив пакет, сидеть и ломать голову, почему ничего не работает.

Одна из наиболее полезных функций менеджера пакетов заключается в возможности удалять программы. Вы можете устанавливать программы, пробовать их в деле и удалять, если они вам не понравятся. После удаления пакета конфигурация вашей системы вернется в состояние, в котором она пребывала до его установки. Деинсталлируя пакет, можно модернизировать систему. Вы удаляете старую версию и устанавливаете новую. В большинстве менеджеров пакетов имеется возможность использовать команду `upgrade`, которая позволяет за один шаг выполнить модернизацию системы.

Для отслеживания установленных приложений менеджер пакетов создает централизованную базу данных. Она также является ценным источником информации о состоянии системы. К примеру, можно увидеть перечень программ, установленных на компьютере, или проверить целостность приложений. Иногда бывает полезным просто просматривать эту базу данных, в результате чего можно обнаружить такое программное обеспечение, о котором вы не знали, что оно установлено.

Наибольшее распространение получили два пакетных формата: RPM (менеджер пакетов RPM Package Manager[1]) и Debian. Прочие типы форматов перечислены в табл. 1.8. Как вы уже догадались, формат RPM используется в дистрибутивах Red Hat и Fedora, а также в дистрибутиве Suse. Аналогично формат Debian применяется в дистрибутиве Debian и некоторых других популярных дистрибутивах (Knoppix, Ubuntu и др.). Существует еще пакетный формат pkgtool, используемый в дистрибутиве Slackware, а также формат portage, встречающийся в дистрибутиве Gentoo.

Таблица 1.8. Популярные дистрибутивы Linux и используемые в них пакетные форматы

Дистрибутив	Пакетный формат
Red Hat	RPM
Fedora	RPM
Debian	Deb
Knoppix	Deb
Ubuntu	Deb
Gentoo	portage
Xandros	Deb
Mandriva (ранее назывался Mandrake)	RPM
mepis	Deb
Slackware	pkgtool

[1] Ранее назывался менеджером пакетов Red Hat Package Manager.

Когда вы определитесь с форматом, вам придется сделать еще один выбор. Поскольку мы ведем речь о свободном программном обеспечении, нужно решить, какой исходный код мы будем использовать.

1.5.1. Что лучше: исходный код или двоичные файлы?

Если вы работаете в Linux на компьютере на базе 32-битного Intel-совместимого процессора, то для вас оптимальным выбором станут скомпилированные двоичные файлы. В большинстве случаев они объединены в пакетный формат, реже — в архив TAR. Если вы решите установить программное обеспечение на основе двоичных файлов, вам не придется иметь дело с исходным кодом.

Если же вы установили Linux на компьютер на базе процессора другого производителя, то ваш единственный выбор — загрузить файлы с исходным кодом и самостоятельно подобрать необходимое программное обеспечение. Иногда вы сами можете предпочесть эту процедуру, даже если у вас в наличии имеются необходимые двоичные файлы. Разработчики нарочно создают двоичные файлы, совместимые с большинством архитектур, чтобы ими могла воспользоваться максимально широкая аудитория. Если в вашем компьютере установлен новый мощный центральный процессор, вы можете перекомпилировать программный пакет таким образом, чтобы он отвечал его архитектуре и новое программное обеспечение могло функционировать быстрее.

В табл. 1.9 представлены наиболее распространенные названия архитектур, используемые в пакетах RPM. Несмотря на то что эти метки во многом схожи с метками, применяемыми компилятором GNU, на самом деле они отличаются. Из-за того что программы-архиваторы произвольно ставят метки, они не всегда отражают реальное содержимое пакетов. Чаще всего встречаются пакеты с меткой i386, которые предназначены для систем на базе процессоров Pentium или Pentium II. Поскольку сейчас осталось немного людей, которые работают в Linux на компьютерах с процессорами архитектуры 80386, подобное положение вещей всех устраивает.

Таблица 1.9. Обзор архитектур

Метка	Описание
i386	Наиболее распространенная архитектура; в компиляторе gss она служит указателем на совместимость с процессорами 80386. Если пакет имеет такую метку, то для работы вам потребуется минимум процессор Pentium I
i486	Не получила широкого распространения. Данная метка указывает на то, что пакет поддерживается процессорами 80486 (или совместимыми с ними)
i586	Постепенно становится распространенной. GNU-компилятор использует метку i586 для описания процессоров Pentium I. Пакет с такой меткой поддерживается всеми процессорами Pentium или более новыми
i686	GNU-компилятор использует метку i686 для описания процессоров Pentium Pro, которые стали основой для процессоров серии Pentium II и выше. Для работы вам потребуется процессор Pentium II или выше
ix86	Распространена мало, указывает на потребность в процессоре Pentium или выше

Метка	Описание
x86_64	Процессоры AMD Opteron и Intel Pentium 4 с поддержкой инструкций EM64T. Это новейшие процессоры, поддерживающие как 32-, так и 64-битные системы. Программный пакет с этой меткой скомпилирован для работы в 64-битном режиме, то есть он несовместим с 32-битными процессорами, а также не будет работать на компьютерах на базе Opteron или процессоров с поддержкой инструкций EM64T под управлением 32-битной версии ядра Linux
IA64	Данная метка указывает на 64-битные процессоры Itanium. Это архитектура, разработанная компаниями Intel и Hewlett-Packard, которая используется только в дорогостоящих рабочих станциях и суперкомпьютерах
ppc	Процессоры PowerPC G2, G3 и G4, устанавливаемые в компьютеры Apple Macintosh и Apple iMac
ppc64	Процессоры PowerPC G5 в компьютерах Apple iMac
sparc	Процессоры SPARC, используемые в рабочих станциях компании Sun
sparc64	64-битные процессоры SPARC, используемые в рабочих станциях компании Sun
mipseb	Процессоры MIPS, преимущественно встречающиеся в рабочих станциях компании SGI

1.5.2. Работа с пакетами

Поскольку создатели новых дистрибутивов Linux пытаются сделать их простыми в обращении, работать в Linux можно и без менеджера пакетов. Несмотря на это, если вы планируете работать за пределами рабочего каталога вашего дистрибутива, необходимо иметь представление о том, каким образом функционирует данный инструмент. Умение работать с менеджерами пакетов может быть особенно полезным в случае возникновения внештатных ситуаций. К основным функциям, которые выполняет менеджер пакетов, относятся следующие:

- установка в системе новых программ;
- удаление (или деинсталляция) программ из системы;
- проверка — позволяет убедиться в том, что программные файлы не повреждены и не искажены;
- модернизация версий установленных в системе программ;
- запрос установленного программного обеспечения (например, «Из какого пакета установлен это файл?»);
- извлечение — позволяет проверить содержимое пакетов перед установкой.

1.6. Кратко о безопасности и пакетах

Почти каждому пользователю компьютера доводилось когда-нибудь сталкиваться с *вредоносными* программами. Если вы работаете в операционной системе Linux, то наверняка получали по электронной почте неизвестные письма от своих друзей, пользующихся Windows, с вирусами, эпидемия которых иногда охватывает Интернет.

Однако к вредоносному программному обеспечению относятся не только вирусы: сюда также входят любые программы, которые проникают в вашу систему

и самовольно хозяйничают в ней. Это могут быть вирусы, шпионские программы и прочее деструктивное программное обеспечение, которое может попасть в систему.

Мотивы создания вредоносных программ могут быть очень обширны: от совершения спланированных преступлений из мести до простого вандализма. Поэтому не думайте, что вас они могут обойти стороной.

Для обеспечения должного уровня безопасности Linux не позволяет непривилегированным пользователям устанавливать программы. Для этого вы должны обладать привилегиями корневого пользователя root, то есть обязаны войти в систему под его правами либо воспользоваться программой sudo. В данном случае ваша система окажется уязвимой, поскольку большинство программ при установке и удалении предполагают выполнение сценариев. Осознанно или нет, но вы полагаетесь на поставщика пакетов в плане того, что эти сценарии не сделают уязвимой вашу систему. Проверка подлинности автора пакета — ключевой шаг в установлении легитимности содержащегося в нем программного обеспечения.

1.6.1. Важность проверки подлинности

В вопросах безопасности имеет значение не только то, какие приложения вы запускаете, но и когда вы это делаете. Пользователь Linux может создавать любые вредоносные программы, однако без привилегий суперпользователя он не сможет нанести вред всей системе. Важно знать, что все пакетные форматы позволяют сохранять в файлах сценарии, которые исполняются во время установки или удаления пакета. К ним относятся сценарии командной оболочки Борна, которые исполняются с привилегиями корневого пользователя root, то есть им позволено абсолютно все. Подобные сценарии являются потенциальным местом, где могут скрываться вредоносные программы вроде троянских коней, поэтому очень важно проверять подлинность программ не перед запуском, а еще до их установки.

Применять инструменты, которые требуют наличия прав суперпользователя, не рекомендуется, однако исключением является менеджер пакетов. Пакетная база данных является центральным узлом распространения, и доступ к ней может получить лишь корневой пользователь root. Проверку подлинности пакетов можно выполнять несколькими способами. Например, инструмент rpm содержит встроенную функцию проверки подлинности. Что касается дистрибутивов вроде Debian, то в данном случае проверка подлинности будет отдельным этапом.

1.6.2. Базовая проверка подлинности пакетов

Основная форма проверки подлинности пакетов подразумевает использование функции хеширования. Это очень напоминает контрольную сумму, когда набор данных уникально идентифицируется посредством суммы всех байтов. Однако простой контрольной суммы всех байтов данных в файле недостаточно, чтобы обеспечить должный уровень безопасности. Различные наборы данных могут иметь одинаковую контрольную сумму, однако данные можно легко изменить, сохранив

при этом изначальную контрольную сумму. Контрольные суммы никогда не используются для проверки подлинности пакетов, поскольку такую подпись легко подделать.

Величина, получаемая с помощью функции хеширования, называется *хешем*. Подобно контрольной сумме, хеш характеризует набор данных произвольного размера посредством фрагмента данных фиксированной длины. В отличие от контрольной суммы, вывод хеша весьма непредсказуем, то есть очень сложно модифицировать данные и затем сгенерировать идентичный хеш. Большинство алгоритмов хеширования используют длинный ключ (например, 128-битный), то есть вероятность сгенерировать аналогичный хеш настолько мала, что и пытаться не стоит. Если вы загружаете файл из незнакомого источника, но имеете хеш из доверенного источника, то можете быть уверены в двух вещах:

- шансы получить модифицированный файл с идентичным хешем очень невелики;
- вероятность того, что злоумышленник может взять файл, модифицировать его по своему усмотрению и сгенерировать идентичный хеш, бесконечно мала.

Популярным инструментом генерирования хешей является программа md5sum на основе алгоритма MD5[1], которая генерирует 128-битный хеш. Программа md5sum может создавать хеши и проверять их. Для того чтобы сгенерировать хеш, нужно ввести в командную строку имя файла. Затем эта программа сгенерирует для каждого файла отдельную строку, в которой будут итоговый хеш и имя файла:

```
$ md5sum foo.tar bar.tar
af8e7b3117b93df1ef2ad8336976574f *foo.tar
2b1999f965e4abba2811d4e99e879f04 *bar.tar
```

Аналогичные данные необходимо вводить в программу md5sum, если вы хотите выполнить верификацию хешей:

```
$ md5sum foo.tar bar.tar > md5.sums
$ md5sum --check md5.sums
foo.tar: OK
bar.tar: OK
```

Каждый хеш имеет вид шестнадцатеричного числа, состоящего из 32 цифр (каждое шестнадцатеричное число — это 4 бита). Хеш можно проверить, сравнив его со значением из доверенного источника и убедившись, что данные корректны. Если хеши MD5 совпадают, вы можете быть уверены, что данный файл не подвергался изменениям с момента создания хеша MD5. Единственной проблемой является то, что неизвестно, можно ли доверять хешу MD5, который используется для сравнительной проверки. В идеале сумму MD5 следует брать с доверенного сайта, с которого вы не скачивали пакетные файлы. Если загруженные вами с какого-либо сайта файлы заражены троянской программой, можете быть уверены, что все хеши MD5 на этом сайте также подвергались модификации.

---

[1] MD5 означает алгоритм Message Digest номер 5.

1.6.3. Проверка подлинности пакетов с использованием цифровых подписей

Цифровая подпись — это еще одна разновидность хеша вроде MD5, за тем исключением, что здесь не требуется никаких уникальных сведений о данных, подвергаемых проверке подлинности. Все, что нужно для проверки подлинности цифровой подписи, — это единственный общий ключ, предоставляемый лицом или организацией, которую вы желаете аутентифицировать. Имея общий ключ, вы можете подвергать проверке подлинности любые данные, подписанные таким лицом. Даже если каждая подпись этого лица представляет собой уникальный хеш, вы сможете проверять любые подписанные им данные, используя лишь один общий ключ.

Лицо, которое хочет подписать данные, генерирует два ключа: общий и частный. В основе этих ключей лежит фраза-пароль, которая известна только их создателю. Создатель держит частный ключ и фразу-пароль в секрете, а общий ключ доступен всем желающим. При изменении частного ключа или данных проверка подлинности будет неудачной. Вероятность создания действительной подписи вместе с идентичным общим ключом, но с другой фразой-паролем и частным ключом чрезвычайно низка. Шансы подделать подпись для легитимного общего ключа близки к нулю.

При работе с цифровыми подписями в сфере свободного программного обеспечения наибольшей популярностью пользуется инструмент GNU Privacy Guard (GPG). Процесс подписи данных с помощью GPG отражен на рис. 1.1.

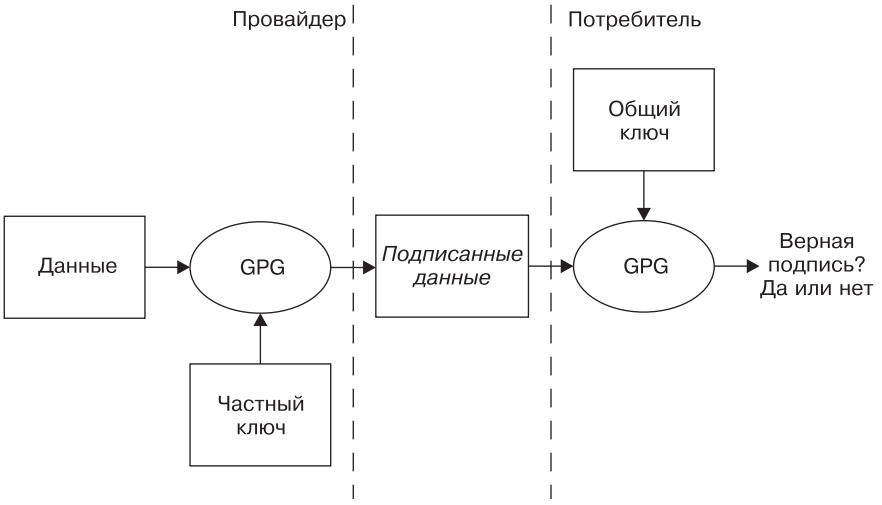


Рис. 1.1. Процесс подписи данных с использованием GPG

1.6.4. Подпись пакетов RPM с использованием инструмента GPG

Пакетный формат RPM позволяет выполнить GPG-подпись данных для последующей проверки подлинности. Формат RPM также использует хеши, включая

MD5, для каждого файла, содержащегося в пакете RPM. Эти хеши служат для проверки целостности передаваемых пакетов RPM, позволяют выявлять в них искажения после установки, однако не обеспечивают проверки подлинности. Только подпись GPG позволяет проверить подлинность пакетов RPM. В качестве альтернативного способа вы можете вручную делать это, используя хеши MD5 пакетных файлов, полученные из доверенных источников.

Утилита rpm имеет флаг `-checksig`, который, к несчастью, смешивает все хеши в одной строке. Иными словами, если у файла нет подписи GPG, rpm будет рапортовать о корректности его суммы MD5. Если же у файла есть подпись GPG, то данная утилита просто включит в строку вывода дополнительный параметр `gpg ok`. Взгляните на этот пример:

```
$ rpm -checksig *.rpm
abiword-2.2.7-1.fc3.i386.rpm: sha1 md5 OK
abiword-plugins-impexp-2.2.7-1.fc3.i386.rpm: sha1 md5 OK

abiword-plugins-tools-2.2.7-1.fc2.i386.rpm: sha1 md5 OK
firefox-1.0-2.fc3.i386.rpm: (sha1) dsa sha1 md5 gpg OK
dpkg-1.10.21-1mdk.i586.rpm: (SHA1) DSA sha1 md5 (GPG) NOT OK (MISSING KEYS:
GPG#26752624)
```

Обратите внимание, что в данном примере представлены пять пакетных файлов RPM, однако подпись GPG есть только у пакетов `firefox` и `dpkg`. У нас есть общий ключ для `firefox`, но нет для `dpkg`. Следовательно, в этой группе только пакет `firefox` может считаться прошедшим проверку подлинности, однако данный факт утилита rpm никак не выделяет. Это плохо. Пакет `dpkg` имеет подпись GPG, но у нас нет общего ключа для нее. Иначе говоря, несмотря на то что данный пакет подписан, мы не сможем проверить действительность этой подписи. В такой ситуации rpm по крайней мере выдаст довольно зловещее предупреждение.

Подпись `firefox` была распознана благодаря тому, что данный пакет был подписан Red Hat и я использовал его в Fedora. Дистрибутив Fedora включает несколько общих ключей, которые используются Red Hat для подписи выпускаемых пакетов. Они копируются на жесткий диск при установке дистрибутива. Если вы загрузите пакет RPM и его подпись можно будет проверить с помощью этих ключей, то можете быть уверены, что пакет действительно исходит от Red Hat и не заражен вредоносными программами. Общий ключ для пакета `dpkg` не был обнаружен потому, что этот пакет происходит из дистрибутива Mandrake, а общий ключ для него отсутствует в моем дистрибутиве Fedora. Этот ключ придется искать самостоятельно.

Если вы загрузили пакет, который имеет недействительную подпись или требует неизвестный общий ключ, как пакет `dpkg` из предыдущего примера, то утилита rpm выдаст предупреждение, когда вы попытаетесь его установить. К сожалению, даже если подпись пакета не прошла проверку подлинности, rpm версии 4.3.2 никак не станет препятствовать его установке. Это большой недостаток, поскольку система оказывается уязвимой в процессе инсталляции, когда вы запускаете сценарии с правами корневого пользователя root. Инструмент GPG не видит разницы между поддельным ключом и отсутствующим общим ключом. Он просто не умеет этого. Как и в случае с обычными подписями на бумаге, два человека могут иметь

одинаковое имя, но расписываются по-разному, однако это не делает одного из них фальсификатором. Кроме того, два человека с одинаковыми именами будут обладать уникальными общими ключами. Единственное сообщение, которое выдает GPG при невозможности проверки подлинности подписи, говорит о том, что общий ключ для нее отсутствует.

Поиск отсутствующих общих ключей

В Сети есть несколько ресурсов, где можно отыскать общие ключи GPG, однако рекомендуется использовать только официальные сайты. У нас отсутствует общий ключ для пакета из дистрибутива Mandrake. Убедиться в этом можно с помощью такого запроса:

```
$ rpm -qip dpkg-1.10.21-1mdk.i586.rpm
Name        : dpkg
Version     : 1.10.21         Vendor: Mandrakesoft
Release     : 1mdk            Build Date: Thu May 20 07:03:20 2004

Host: n1.mandrakesoft.com
Packager    : Michael Scherer <misc@mandrake.org>
URL         : http://packages.debian.org/unstable/base/dpkg.html
Summary     : Package maintenance system for Debian
```

В данном случае есть несколько способов отыскать доверенный общий ключ. Сначала лучше воспользоваться URL-ссылками **mandrakesoft.com** и **mandrake.org**. В приведенных сведениях сказано, что пакет был создан в 2004 году, что, по меркам Интернета, было уже довольно давно. Поскольку название Mandrake было переименовано в Mandriva, их старые сайты исчезли. Оказывается, не все так просто, как можно было предположить. Чтобы найти то, что нам нужно, потребуется дополнительная информация: идентификатор ключа.

```
$ rpm --checksig dpkg-1.10.21-1mdk.i586.rpm
dpkg-1.10.21-1mdk.i586.rpm: ...
... (GPG) NOT OK (MISSING KEYS: GPG 78d019f5)
```

Указанный идентификатор и является искомым.

Следующая остановка — сайт Mandriva. Поисковая система Google выдает адрес **http://mandriva.com**. Вводим в окне поиска на этом сайте фразу `общие ключи`, получаем набор хешей и один общий ключ, который не является тем, что нужен нам. Дальнейшие попытки поиска ни к чему не приведут. Похоже, сайт Mandriva в данной ситуации является тупиком.

Открыв поисковик Google, вводим фразу `общие ключи` и получаем множество ссылок на сайты, где есть общие ключи. Методом проб и ошибок требуемый ключ обнаруживается на сайте **www.keys.pgp.net**, где в окне поиска ключей нужно ввести `0x78d019f5`.

```
Search results for '0x78d019f5'

Type bits/keyID    cr. time exp time key expir
pub 1024D/78D019F5 2003-12-10

uid MandrakeContrib <cooker@linux-mandrake.com>
sig sig3 78D019F5 2003-12-10 __________ __________ [selfsig]
sig sig3 70771FF3 2003-12-10 __________ __________ Mandrake Linux
<mandrake@mandrakesoft.com>
```

```
sig sig3 26752624 2003-12-10 __________ __________ MandrakeCooker <cooker@linuxmandrake.com>
sig sig3 45D5857E 2004-09-22 __________ __________ Fabio Pasquarelli (Lavorro) <fabiopasquarel-
li@tin.it>
sig sig3 17A0F9A0 2004-09-22 __________ __________ Fafo (Personale)
rec.r96@tin.it

sub 1024g/4EE127FA 2003-12-10
sig sbind 78D019F5 2003-12-10 __________ __________ []
```

Если в поисковике ввести слово `Mandrake`, то окажется, что наш ключ присутствует в перечне среди прочих ключей Mandrake. Выбрав гиперссылку **78D019F5**, можно получить ключ PGP в текстовом виде. Чтобы импортировать этот ключ, нужно сохранить его в файле под именем `78D019F5.txt` и ввести следующую команду:

```
rpm --import 78D019F5.txt
```

Если в текстовом файле содержится действительный ключ, то никаких сообщений об ошибках не будет. В итоге мы получаем возможность проверить наш оригинальный пакет:

```
rpm --checksig dpkg-1.10.21-1mdk.i586.rpm
dpkg-1.10.21-1mdk.i586.rpm: (sha1) dsa sha1 md5 gpg OK
```

Краткий параметр `gpg OK` говорит нам о том, что все в порядке.

Прежде чем завершить рассмотрение данной темы, скажу пару слов о доверии. Подпись мы получили из домена **pgp.net**. Мы уверены, что владельцы этой базы данных позаботились о том, что общий ключ исходит из легитимного источника. Иными словами, в итоге все сводится к доверию.

1.6.5. Что делать, если проверка подлинности пакета невозможна

Нужно дважды подумать перед тем, как устанавливать пакет, не прошедший проверку подлинности, хотя надо признаться, что я не следую этой истине, когда устанавливаю пакеты на свой старый компьютер. Однако ситуация будет в корне отличаться, если вы инсталлируете пакеты на крупном предприятии. При возникновении серьезной опасности я немного потеряю, если что-то пропадет с моего старого компьютера, но если речь идет о большой организации, последствия могут быть самыми плачевными.

Порой создатели пакетов не предоставляют никакой информации, которую можно было бы использовать для проверки подлинности. Хотя вероятность заполучить троянского коня вместе с дистрибутивом Linux довольно низка, со временем все может измениться. Вот некоторые практические советы, которым нужно следовать, если проверка подлинности пакета посредством подписи невозможна.

- Используйте исходный код для сборки своих приложений.

  Такой подход, конечно, не защитит от неосторожного или неумелого обращения, как отмечалось ранее в этой главе в случае с OpenSSH. Тем не менее следует осознавать, что использование исходного кода в своих проектах может

быть как легким, так и сложным. Каждый проект имеет свои нюансы, с которыми сталкиваются разработчики. Загрузка исходного кода обычно не представляет сложности, однако после этого вам придется обратиться к дополнительному инструментарию. Работа над незнакомым проектом рискует серьезно затянуться, пока вы будете заняты поиском всех пакетов, необходимых при разработке. О создании приложений на основе исходного кода мы подробно поговорим в главе 2.

- Проверяйте сценарии, исполняемые в процессе установки.

  Данные сценарии являются источником непосредственной опасности, поскольку при инсталляции пакетов они исполняются с привилегиями корневого пользователя root, то есть еще до использования установленного программного обеспечения. Позднее я расскажу вам, как проверять любой пакетный формат.

- Проверяйте содержимое.

  Внимательно изучите двоичные файлы, которые устанавливаете. Типичное пользовательское приложение не должно нуждаться в двоичных файлах в /usr/sbin или /sbin, поскольку данные каталоги зарезервированы для системных демонов и инструментов системного администрирования. Проявляйте бдительность с файлами с установленным битом `setuid` или `setgid`, особенно если их владельцем выступает корневой пользователь root. Эти файлы могут исполняться с правами их владельца, и в них могут скрываться вредоносные программы. Лишь нескольким системным программам требуется такая возможность; ко всему прочему нужно относиться с подозрением.

Независимо от выбранной методики помните, что многое по-прежнему зависит от доверия. Вместо того чтобы доверять не прошедшему проверку подлинности источнику, лучше полагаться на свои способности и выявлять вредоносные программы еще на этапе инспектирования содержимого пакетов.

1.7. Проверка содержимого пакетов

Перед установкой любого загруженного пакета необходимо проверить в нем несколько важных моментов. Большинство пакетных форматов состоят из следующих ключевых частей:

- архива с файлами, которые будут устанавливаться в систему; он может иметь формат TAR, CPIO или любой другой;
- сценариев, исполняемых во время установки или удаления пакета;
- сведений о зависимостях, которые используются инструментом инсталляции, чтобы гарантировать соответствие вашей системы требованиям для установки пакета;
- текстовых данных о самом пакете.

Количество описательных сведений, содержащихся в пакете, в основном зависит от лица, которое его создает. Обычно сюда входят сведения об авторе, дате упаковки файлов и лицензионное соглашение. Более заботливые создатели паке-

тов также вносят информацию о том, для чего предназначено их программное обеспечение, что бывает нечасто.

Зависимости пакетов могут быть обширными, небольшими или вовсе отсутствовать. Пакеты дистрибутивов Slackware, например, вообще не содержат информации о зависимостях. Вы просто устанавливаете пакет и скрещиваете пальцы в ожидании того, будет ли все работать. Пакеты RPM имеют свои крайности. При создании пакета в формате RPM архиватор может автоматически определять зависимости, информация о которых попадет в пакет. Он также позволяет выбирать, сведения о каких зависимостях будут помещены в пакет, либо вовсе не включать их в его содержимое (как в дистрибутиве Slackware).

Каждый пакетный формат обеспечивает определенный порядок исполнения сценариев во время установки или удаления. Сценарии установки необходимо тщательно исследовать. Даже если вы уверены в отсутствии вредоносных программ, плохо проработанные проекты могут содержать дефектные сценарии, которые нанесут ущерб вашей системе. Если вам сложно понять какой-либо сценарий, рекомендуется найти способ проверить подлинность пакета, перед тем как приступать к его установке.

Сценарии установки обычно подразделяются на следующие категории.

- **Прединсталляционные** — исполняются перед распаковкой данных из архива.
- **Постинсталляционные** — выполняются после распаковки данных из архива. Обычно эти сценарии отвечают за второстепенные задачи по настройке инсталляции, например за установку патчей или создание конфигурационных файлов.
- **Преддеинсталляционные** — исполняются, когда вы хотите удалить пакет, однако до того момента, как любые файлы подвергнутся удалению.
- **Постдеинсталляционные** — выполняются после удаления из системы первичных файлов.

Текстовая информация, закладываемая в пакет, варьируется в зависимости от его формата. Обычно пакет содержит дополнительные сведения, касающиеся проверки подлинности, как, например, проект страницы на сайте SourceForge.net. Информация о зависимостях, присутствующая в пакете, также меняется в зависимости от его формата. Это могут быть имена других пакетов или исполняемых программ, которые требуются пакету.

1.7.1. Как осуществлять проверку пакетов

Проверку пакета можно выполнять как перед его установкой, так и после нее. Перед установкой проверяется пакетный файл, который может иметь любое допустимое в Linux имя. Довольно часто, но не всегда имя файла является производной от *официального* имени пакета, то есть это имя отражается в базе данных пакета после его установки. Имя пакета зашифровывается в пакетном файле и доступно для просмотра при его запросе. Несмотря на то что создатели пакетов делают имя пакета частью имени файла, не нужно считать, что имя файла и имя пакета — это одно и то же. После установки пакета к нему можно обращаться только посредством

имени, указанного в пакетном файле, то есть запрос пакетного файла необходимо осуществлять, используя его официальное имя. К примеру, вы устанавливаете компилятор gcc, однако по какой-то причине пакет называется `foo.rpm`. Содержимое этого RPM-файла можно запросить такой командой:

```
$ rpm -qip foo.rpm
```

Однако после его установки команда запроса будет выглядеть уже так:

```
$ rpm -qi gcc
```

Команда `rpm` используется для запроса баз данных RPM, а если нужно запросить пакетный *файл*, к этой команде необходимо добавить параметр `-p`. Один и тот же пакет может называться по-разному, однако после установки в базе данных он будет иметь только одно имя.

Как уже отмечалось, сведения, содержащиеся в пакете, включают имя, версию, информацию о создателе, авторские права и зависимости. К дополнительным сведениям относятся перечни файлов для установки, а также любые сценарии, исполняющиеся во время установки и удаления. Именно такие сведения должны вас интересовать перед инсталляцией пакетов. В табл. 1.10 приведен список базовых команд для запроса пакетных файлов RPM и Debian.

Таблица 1.10. Команды запроса пакетных файлов

Запрос	RPM	Debian
Базовые сведения	rpm -qpi имя_файла	dpkg -s имя_файла
Список файлов для установки	rpm -qpl имя_файла	dpkg -L имя_файла
Дамп сценариев установки/удаления	rpm -qp -scripts имя_файла	dpkg -e
Проверка аутентификационных сведений	rpm --checksig имя_файла	Отсутствует
Показать пакеты, необходимые для данного пакета	rpm -qp --requires имя_файла	dpkg -I
Показать, какой пакет представляет этот файл (например, его имя и версия, отображаемые в базе данных)	rpm -qp --provides имя_файла	dpkg -I

Запрос пакетной базы данных может понадобиться по нескольким причинам. Например, вы можете захотеть просмотреть список всех установленных в вашей системе пакетов или узнать версию какого-либо из них. Следует также проверять содержимое установленных пакетов, чтобы убедиться, что ни один из файлов с момента инсталляции не был поврежден. Команды запроса установленных пакетов немного отличаются от команд запроса пакетных файлов. Некоторые из них вы можете увидеть в табл. 1.11.

Таблица 1.11. Команды запроса установленных пакетов

Запрос	RPM	Debian
Базовые сведения о каком-либо пакете	rpm -qi имя	dpkg -s имя
Показать список всех установленных пакетов	rpm –qa	dpkg –list

Запрос	RPM	Debian
Показать список всех файлов, установленных из какого-либо пакета	rpm -ql имя	dpkg -L имя
Проверить файлы, установленные из какого-либо пакета	rpm -V имя	cd /; md5sum -c < /var/lib/dpkg/info/name.md5sums
К какому пакету относится данный файл?	rpm -qf имя_файла	dpkg -S имя_файла
Какую текущую версию имеет установленный пакет X?	rpm -q X	dpkg-query -W X

1.7.2. Обзор пакетов RPM

RPM — один из самых распространенных пакетных форматов, которые встречаются в Linux. Пакеты RPM могут содержать множество информации, однако ее иногда бывает сложно извлекать. Чтобы извлечь дополнительные данные, нужно воспользоваться утилитой rpm с параметром `--queryformat option` (сокращенно — `--qf`). Большинство тегов, используемых параметром `-qf`, не документированы в справочных руководствах, но их можно просмотреть с помощью такой команды:

```
$ rpm --querytags
HEADERIMAGE
HEADERSIGNATURES
HEADERIMMUTABLE
HEADERREGIONS
HEADERI18NTABLE
SIGSIZE
SIGPGP
SIGMD5
SIGGPG
PUBKEYS
...
```

Необходимо отметить, что теги запроса чувствительны к регистру, несмотря на то что в выводе rpm они имеют верхний регистр. Если, например, вы захотите узнать, кто является поставщиком пакетов для вашего дистрибутива, введите такую команду запроса:

```
$ rpm -qa --qf '%{vendor}' | sort | uniq -c
      1 Adobe Systems, Incorporated
     12 (none)
      1 RealNetworks, Inc
    838 Red Hat, Inc.
      1 Sun Microsystems, Inc.
```

Данный запрос в моей системе Fedora Core 3 показал, что поставщиком 838 пакетов является Red Hat, а источник 12 пакетов неизвестен. Как оказалось, неопознанные пакеты на самом деле являются общими ключами GPG. Каждый из них выглядит как отдельный пакет в базе данных и не имеет идентификатора «поставщика».

Для проверки сценариев установки, содержащихся в пакетах RPM, используется другой запрос. Вот его пример:

```
$ rpm -qp --scripts gawk-3.1.3-9.i386.rpm

postinstall scriptlet (through /bin/sh):
if [ -f /usr/share/info/gawk.info.gz ]; then
    /sbin/install-info /usr/share/info/gawk.info.gz
/usr/share/info/dir
fi
preuninstall scriptlet (through /bin/sh):
if [ $1 = 0 -a -f /usr/share/info/gawk.info.gz ]; then
    /sbin/install-info --delete /usr/share/info/gawk.info.gz
/usr/share/info/dirfi
```

Вывод запроса включает строку, из которой можно понять назначение сценария (постинсталляционный и т. п.) и его тип (например, `/bin/sh`). Таким образом, сценарии можно визуально исследовать еще до их исполнения.

1.7.3. Обзор пакетов Debian

Пакеты Debian имеют более простой формат, чем RPM, а у программы dpkg отсутствуют многие возможности, которыми обладает утилита rpm. Поэтому при работе с пакетами Debian потребуется немного больше усилий. Пакет Debian обычно имеет расширение `.deb`, хотя на самом деле это архив, созданный с помощью команды `ar`. Данная команда применима для проверки содержимого пакета Debian, однако это мало о чем говорит. Например:

```
$ ar -t cron_3.0pl1-72_i386.deb
debian-binary
control.tar.gz
data.tar.gz
```

Файл с именем `debian-binary` содержит ASCII-строку, где указан формат пакета. Файл под названием `control.tar.gz` является сжатым архивом TAR, содержащим сценарии установки, а также другие полезные сведения. Файл с именем `data.tar.gz` представляет собой сжатый архив TAR, где содержатся установочные файлы программы. Чтобы извлечь эти файлы для дальнейшего ознакомления, необходимо ввести такую команду:

```
$ ar -x filename.deb
```

Рассмотрим файлы более подробно. Архив `data.tar.gz` содержит файлы, необходимые для работы программы. Иногда достаточно просто извлечь их и получить рабочую инсталляцию, однако делать этого не рекомендуется. В нашем случае содержимое архива имеет следующий вид:

```
$ tar -tzf data.tar.gz
./
./usr/
./usr/bin/
```

```
./usr/bin/crontab
./usr/sbin/
./usr/sbin/cron
./usr/sbin/checksecurity
./usr/share/
./usr/share/man/
./usr/share/man/man1/
...
```

Архив `control.tar.gz` содержит дополнительные файлы, необходимые для установки, удаления и сопровождения пакета. Извлечь их можно с помощью команды `dpkg` с параметром `-e`:

```
$ dpkg -e cron_3.0pl1-72_i386.deb
$ ls ./DEBIAN/*
./DEBIAN/conffiles
./DEBIAN/control
./DEBIAN/md5sums
./DEBIAN/postinst
./DEBIAN/postrm
./DEBIAN/preinst
./DEBIAN/prerm
```

Как вы уже догадались, файлы `preinst` и `postinst` являются прединсталляционным и постинсталляционным сценариями, о которых мы уже говорили в этой главе. Аналогично файлы `prerm` и `postrm` — это преддеинсталляционный и постдеинсталляционный сценарии соответственно.

1.8. Обновление пакетов

Программа обновления пакетов позволяет отслеживать пакетные файлы и их зависимости. Допустим, вы решили установить пакет `X`, который требует наличия трех дополнительных пакетов, которые у вас не установлены. Чтобы иметь возможность установить пакет `X`, вам придется инсталлировать три недостающих пакета. Однако и эти три пакета могут потребовать установки других пакетов, которых у вас нет, а они, в свою очередь, могут требовать наличия дополнительных пакетов и т. д.

Здесь на помощь приходит программа обновления пакетов. Достаточно запросить пакет `X`, и она сама определит, какие дополнительные пакеты нужны для установки данного пакета, после чего загрузит и установит их.

Программа обновления пакетов использует в своей работе список пакетных репозиториев, в котором осуществляет поиск при запросе какого-либо пакета. Обычно репозитории размещаются в Интернете и сопровождаются распространителем (например, Red Hat). Посредством репозиториев распространители дают возможность пользователям получать исправления и обновления безопасности наряду с обновлениями общего характера.

Репозиторий может также располагаться в локальной файловой системе на компакт-диске или другом компьютере, защищенном брандмауэром, что особенно

удобно, если вам приходится обслуживать большое количество компьютеров в локальной сети. К примеру, можно загрузить необходимые пакеты из Интернета и разместить их в локальной сети, чтобы ваши пользователи смогли быстро установить их на свои компьютеры.

Для работы с дистрибутивами Debian используется инструмент Apt — *Advanced Package Tool*. В действительности он представляет собой набор инструментов командной строки для сопровождения пакетов в дистрибутивах. Apt был портирован для работы с дистрибутивами пакетов RPM. Остается надеяться, что данный инструмент станет популярным средством управления пакетами среди пользователей RPM.

Для работы с дистрибутивами пакетов RPM существует два основных инструмента. Первый — up2date — разработан Red Hat для собственных дистрибутивов Enterprise Server и Fedora Core. Второй называетсяYum — сокращенно от *Yellowdog Updater Modified*[1].

1.8.1. Инструмент Apt: Advanced Package Tool

Apt является одним из самых лучших средств управления пакетами в дистрибутивах Debian, а теперь он также доступен и для дистрибутивов RPM. В отличие от базового инструмента dpkg, он обладает замечательным свойством — автоматически проверяет подлинность пакетов, имеющих подпись GPG. Как вы уже знаете, RPM уже поддерживает подписи GPG. Более того, поскольку Apt выполняет проверку не поддающегося аутентификации пакета, вам не придется беспокоиться о том, что какие-либо репозитории подверглись воздействию со стороны злоумышленников и содержащиеся в них пакеты заражены троянскими программами.

Как и dpkg, инструмент Apt представляет собой набор команд. Чаще всего используются комнады `apt-get` и `apt-cache`. Сначала вас, скорее всего, будет интересовать команда `apt-get`. Это «рабочая лошадка», которая извлекает и устанавливает пакеты. Команда `apt-cache` позволяет запрашивать список пакетов, загруженных в локальный кэш, что будет намного быстрее, чем многократно обращаться к репозиториям в Интернете. Данный список содержит все имеющиеся пакеты, включая те, что вы не инсталлировали, а также установленные обновления.

С помощью команды `apt-key` в базу данных можно добавлять общие подписи, полученные из доверенного источника, и проверять уже имеющиеся. Это позволяет устанавливать подлинность пакетов, имеющих подпись такого источника. Команда `apt-setup` дает возможность выбирать желаемые репозитории, которые Apt должен обследовать при поиске каких-либо пакетов. В моем дистрибутиве Ubuntu `apt-setup` позволяет осуществлять поиск только по репозиториям на сайтах-зеркалах Ubuntu, а репозитории Debian недоступны. В данной ситуации можно вручную отредактировать файл `/etc/apt/sources.list`, чтобы при поиске охватывались и другие репозитории. Данный файл может указывать на сайты в Интернете или на локальные каталоги, расположенные в вашей системе или

[1] Раньше, когда он входил в состав дистрибутива Yellowdog для компьютеров архитектуры PowerPC, он носил название Yup, однако впоследствии был адаптирован под дистрибутивы пакетов RPM и модифицирован.

в локальной сети. Единственное, что требуется Apt, — чтобы файлы были доступны через URL-адреса.

1.8.2. Инструмент Yum: Yellowdog Updater Modified

В настоящее время Yum используется для работы с дистрибутивами RPM. Это утилита командной строки, которая функционирует подобно инструменту Apt. Как и Apt, Yum использует кэш для хранения сведений об имеющихся пакетах. В отличие от Apt, Yum при каждом запуске по умолчанию обследует все репозитории. Однако такой порядок оказывается более затратным по времени, если сравнивать его с кэшем Apt.

Команда `yum` используется для установки, запроса и обновления пакетов. Параметр `-C` говорит использовать кэш при начальном запросе. Если `yum` решит установить программное обеспечение на основе этого запроса, то перед этим она обновит кэш.

При использовании Yum проверка подлинности осуществляется опционально посредством подписей GPG. Это контролируется в отношении всех репозиториев через файлы конфигурации `/etc/yum.conf` и `/etc/yum.repos.d`. Если установлен флаг `gpgcheck=1`, то команда `yum` не будет устанавливать пакеты, не прошедшие проверку подлинности. Как и в случае с Apt, вы можете создавать собственные репозитории в каталоге, который доступен по URL-адресу.

Выбирать параметры для команды `yum` можно интуитивным путем. Например, чтобы просмотреть перечень всех установленных пакетов, которые можно обновить, нужно ввести такую команду:

```
$ yum list updates
```

В результате вы получите список пакетов, доступных для обновления. Наиболее близким эквивалентом этой команды в случае с Apt будет менее интуитивная строка `apt-get --dry-run -u dist-upgrade`, которая выдает более громоздкий результат.

1.8.3. Synaptic: передовой GUI-интерфейс для APT

На момент создания этой книги Synaptic даже не добрался до версии 1.0, но это чрезвычайно полезный GUI-интерфейс для работы с пакетами посредством Apt. Когда я писал книгу, на моей системе Ubuntu был установлен 861 пакет. В любой заданный момент времени многие из них можно было обновить. Для решения этой задачи необходим GUI-интерфейс. Synaptic сортирует пакеты по категориям, в результате чего можно без труда отыскать обновления для часто используемых программ и ознакомиться с ними. Вам, как разработчику, вероятно, хочется узнать, когда компилятор gcc обновится с версии 3.3 до версии 3.4, однако вряд ли будет интересно, что FreeCell обновился с версии 1.0.1 до версии 1.0.2. Категории можно также использовать для поиска новых или еще не установленных программ. Я, как программист, регулярно отслеживаю появление новых версий средств разработки. На рис. 1.2 вы можете наблюдать пример использования Synaptic на практике.

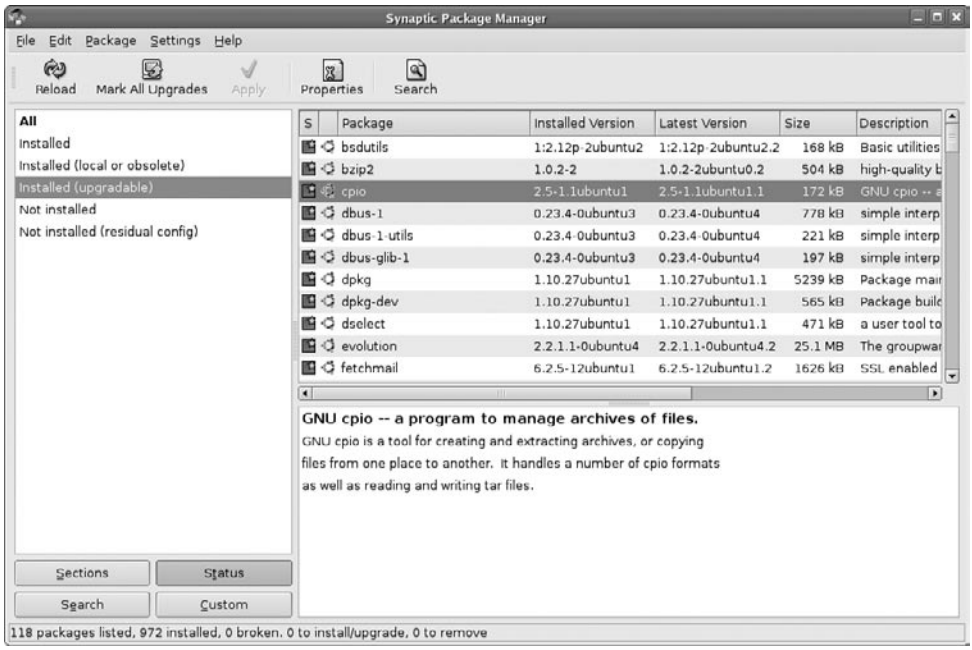


Рис. 1.2. Пример GUI-интерфейса Synaptic

Как и Apt, Synaptic не устанавливает без вашего согласия пакеты, не прошедшие проверку подлинности. Замечательной особенностью Synaptic является то, что он позволяет увидеть возможный результат ваших действий без томительного ожидания, пока программа загрузит множество пакетов, которые вам не нужны. Если зайти в раздел Games and Amusements (Игры и развлечения) и выбрать установку kasteroids, возникнет небольшая проблема: Ubuntu по умолчанию использует GNOME, а kasteroids — это KDE-программа. Иначе говоря, если вы решите установить kasteroids, то вам потребуется установить еще десять дополнительных пакетов. Synaptic охотно выполнит эту задачу, но предварительно выведет предупреждение о том, что необходимо установить десять дополнительных пакетов. Это подразумевает загрузку нескольких лишних мегабайт. Поскольку времени на это нет, то, скорее всего, в данный момент от установки программы kasteroids лучше отказаться.

Другая полезная функция заключается в наличии фильтра, который позволяет запрашивать тома, где содержатся данные обновлений, то есть вы можете легко отыскать нужное обновление. Synaptic находится на раннем этапе развития (на момент написания этой книги он имел версию 0.56), и эта функция еще будет дорабатываться. В настоящее время нельзя отфильтровать важные обновления от второстепенных. Обычно незначительные обновления выпускаются для исправления ошибок, а важные — для добавления новых функций.

Synaptic применим в качестве GUI-интерфейса в системе Debian. Он также может стать популярным и в RPM-дистрибутивах. Однако в настоящее время лишь небольшая часть RPM-репозиториев поддерживает Apt и Synaptic.

1.8.4. up2date: инструмент обновления пакетов Red Hat

GUI-интерфейс для up2date был создан Red Hat для работы с репозиториями Yum. Данным инструментом также можно пользоваться с помощью командной строки. Без добавления каких-либо параметров команда `up2date` выдает список файлов, доступных для обновления, которые могут исчисляться сотнями, и спрашивает у пользователя, какие из них нужно обновить.

По умолчанию в список не включаются неустановленные пакеты, поэтому, даже если доступен новый инструмент, up2date не уведомит вас об этом. Как и Synaptic, up2date проверяет подлинность пакетов посредством подписей GPG и не устанавливает пакеты, которые ее не пройдут.

GUI-интерфейс у up2date оставляет желать лучшего. Функциональность минимальна, обновлять можно только существующие пакеты; увидеть, сколько есть новых, невозможно, а просматривать и удалять уже установленные пакеты нельзя. Весьма досадно, особенно если учитывать, что при использовании в командной строке данный инструмент весьма удобен и интуитивно понятен.

Инструмент up2date старается быть хамелеоном, обеспечивая доступ в репозитории Yum, Apt и up2date. Конфигурация, входящая в дистрибутив Fedora Core 4, по умолчанию направляет пользователя к полному списку репозиториев Yum. Это разумный подход, однако при этом процесс обновления сильно затягивается. Лучше вручную найти несколько репозиториев и добавить их в файл `/etc/sysconfig/rhn/sources`. Положительной чертой инструмента up2date является то, что вы можете применить его к каталогу, содержащему пакеты RPM, и определить все зависимости. Если каталог содержит все необходимые пакеты RPM, то процесс пройдет нормально. Монтируйте установочный DVD на `/mnt/dvd` и добавьте в `/etc/sysconfig/rhn/sources` следующую строку:

```
dir fc-dvd /mnt/dvd/Fedora/RPMS
```

Теперь можно устанавливать пакеты с компакт-диска, а up2date позаботится о зависимостях. С помощью командной строки это делается так:

```
$ up2date --install gcc
```

1.9. Заключение

В этой главе были изложены основы программного обеспечения с открытым исходным кодом. В частности, мы рассмотрели различные форматы распространения, а также инструменты для работы с ними. Была подробно исследована концепция архивного файла, который является ядром любого формата распространения, а в ряде случаев и сам выступает в этой роли. Мы также обсудили основные моменты обеспечения безопасности, которые позволяют избежать проникновения на компьютер вредоносных программ при загрузке пакетов. Вы научились проверять подлинность и предпринимать меры для защиты своей системы. В конце главы были рассмотрены некоторые инструменты для обработки пакетов. Они

используются для управления всеми пакетами, имеющимися в системе. Каждому из них свойственны свои преимущества и недостатки.

В этой главе использовались следующие инструменты:

- dpkg — основной инструмент для установки и запроса пакетов формата Debian, используемый в дистрибутиве Debian и производных от него, например в Ubuntu;
- gpg — GNU-инструмент для шифрования и подписи, который представляет собой средство общего назначения, применяемое для расширения безопасности пакетов посредством их цифровой подписи;
- gzip, bzip2 — GNU-утилиты для сжатия файлов и создания архивов;
- rpm — основной инструмент для установки и запроса программных пакетов, созданных с помощью менеджера пакетов Red Hat Package Manager (RPM); помимо Red Hat, RPM также используется в Suse и других дистрибутивах;
- tar, cpio, ar — UNIX-архиваторы, используемые для создания большинства программных архивов.

Веб-ссылки этой главы:

- www.debian.org — домашняя страница Debian, где можно отыскать ответы на часто задаваемые вопросы, касающиеся этого формата пакетов;
- www.gnupg.org — домашняя страница проекта GNU Privacy Guard, в результате осуществления которого был создан инструмент gpg;
- www.pgp.net — репозиторий общих ключей, используемых gpg и прочими инструментами;
- www.rpm.org — домашняя страница проекта RPM.

2 Создание приложений на основе исходного кода

2.1. Введение

Эта глава посвящена инструментам для создания приложений, а также вопросам, которые следует учитывать при сборке программного обеспечения, распространяемого в виде исходного кода. Несмотря на свои недостатки, программа make по-прежнему является ключевым инструментом для сборки приложений под Linux. Данной программой пользуются практически все разработчики, однако я расскажу вам о важных деталях, которые известны далеко не всем из них.

Вы узнаете, каким образом GNU-код распространяется с помощью сборочного GNU-инструментария, который также используется во многих других проектах разработки свободного программного обеспечения. Кроме того, мы поговорим об альтернативах использованию программы make. Будут также исследованы распространенные ошибки и предупреждения, с которыми вы можете столкнуться при работе со сборочным инструментарием.

2.2. Инструменты для сборки приложений

Разработка программного обеспечения — это циклический процесс. Вы редактируете исходный код, компилируете и исполняете его, находите ошибки и повторяете весь цикл заново. Хотя данный процесс выглядит не очень эффективным, инструменты сборки помогут исправить это. Программа make является «рабочей лошадкой» создателя программ под Linux. Она стала первым инструментом, который поддерживает итеративный процесс сборки программного обеспечения. Несмотря на то что многим программистам он не нравится, до настоящего времени достойной замены инструменту make не придумано. По этой причине любой программист под Linux должен хорошо уметь пользоваться данным инструментом.

2.2.1. Предыстория

Поначалу программа make была довольно примитивным инструментом. UNIX-версия make не поддерживает условные конструкции и языки. Вместо этого она полагается на несколько простых типов выражений, которые управляют ее взаимодействием с оболочкой. В процессе сборки программ всю работу выполняет оболочка, а make лишь играет роль диспетчера.

В простоте make заключаются как ее преимущества, так и недостатки. Отсутствие возможности применять сценарии — проблема для разработчиков, которые хотят развертывать исходный код на разных платформах. Результатом стало то, что в дистрибутивах исходного кода иногда можно встретить несколько файлов Makefile, каждый из которых имеет свою цель. Это не только беспорядочно и некрасиво выглядит, но и доставляет головную боль при сопровождении.

На протяжении нескольких лет разрабатывались различные варианты make, призванные устранить недостатки первой версии. Они сохраняли базовый синтаксис make, но привносили дополнительные ключевые слова, которые наделяли их новыми функциями. К сожалению, нет гарантии, что сценарий сборки, совместимый с одной версией make, подойдет для другой версии, и это служит для разработчиков сдерживающим фактором от применения новых версий инструмента. GNU-вариант make является одной из таких версий, используемых в Linux. Позднее я подробно расскажу вам о некоторых ее особенностях.

Существует также несколько инструментов, которые были созданы на основе утилиты make, но лишены ее недостатков. Они генерируют файлы Makefile на основе высокоуровневого описания проекта.

Imake

Одним из первых инструментов для создания файлов Makefile стала программа imake. Она происходит из проекта X Window и служит для сборки приложений X Window на различных платформах UNIX. Данная программа применяет препроцессор C для разбора сценариев сборки (называемых imakefile) и генерирует файл Makefile, используемый программой make. Это позволяет увеличить переносимость посредством инкапсуляции специфического системного «бюрократического» кода в макросе препроцессора и условных конструкциях. Кроме того, файлы imakefile компактны и просты.

Однако инструмент imake никогда не пользовался популярностью и тоже обладает недостатками. Один из них заключается в том, что каждая цель сборки требует детального набора макросов. Если в целевой системе такой набор описаний отсутствует, то вы не сможете использовать imake. Это препятствует развертыванию проектов, где используется imake, в других системах.

Несмотря на то что imake вполне годится для сборки программ X Window на UNIX-системах, сейчас встретить файлы imakefile можно лишь в старых проектах X Window.

Сборочный инструментарий GNU

В поисках оптимального способа сборки программ на различных архитектурах участники проекта GNU Project создали собственный набор инструментов в до-

полнение к программе make. Принципы остались такими же, что были в imake, за тем исключением, что вместо препроцессора C сборочный инструментарий GNU использует программу m4, которая обладает большей функциональностью. Данные инструменты применяются для создания дистрибутивов исходного кода, сборку которых можно осуществлять в разнообразных системах. Для сборки индивидуального проекта на основе подобного дистрибутива вам потребуется лишь рабочая оболочка и программа make. Сборочный инструментарий GNU де-факто стал стандартом для распространения исходного кода.

Альтернативные инструменты сборки

Альтернативные средства сборки программ не пользуются популярностью, потому что большинство разработчиков не желают связываться с инструментами, которые находятся в процессе развития. У них и так хватает забот, чтобы еще беспокоиться о возможных недоработках в таких инструментах и о совместимости своих сценариев сборки с их будущими версиями. Сборочный инструментарий GNU можно сравнить с тестом производительности: любые альтернативные средства должны быть еще более просты в использовании и работать так же быстро или еще быстрее.

Существует один проект, который может бросить вызов GNU-инструментарию. Он называется Cons[1] (www.gnu.org/software/cons/dev/cons.html). Данный инструмент основан на языке Perl (www.perl.org), из которого его сценарии сборки заимствуют свой синтаксис. Простота его использования по сравнению с GNU-инструментарием объясняется тем, что разработчикам более привычен Perl. Все GNU-инструменты и связанные с ними файлы заменяются одним файлом, который используется одним инструментом за один подход. Недостаток подобного подхода заключается в том, что дополнительное время, которое отнимает этап конфигурирования при сборке программы с помощью GNU-инструментов, будет необходимо каждый раз, когда вы запускаете сборку.

2.2.2. Инструмент make

Унструмент make — это превосходное средство повысить производительность. Грамотно составленный Makefile может значительно ускорить процесс разработки. К сожалению, многие программисты обучаются использованию make методом проб и ошибок, не пользуясь никакой документацией. Они во многом полагаются на интуицию и удачу, о чем свидетельствуют создаваемые ими файлы Makefile. Поскольку вы тоже можете относиться к таким разработчикам, мы начнем с основ и познакомимся с несколькими весьма полезными расширениями GNU.

Основы Makefile: правила и зависимости

В отличие от традиционных сценариев, которые последовательно исполняют программы, файлы Makefile содержат смесь правил и инструкций. Эти правила выглядят следующим образом:

---

[1] Сокращенно от Construction System.

```
target: prerequisite
        commands
```

Правило утверждает зависимость, которая говорит о том, что цель зависит от предпосылки. В самом простом виде цель состоит из одного файлового имени, а предпосылка содержит одно или несколько файловых имен. Если цель старше любой из своих предпосылок, то исполняются команды, ассоциированные с правилом. Команды обычно содержат инструкции для сборки цели. Вот простой пример:

```
foo: foo.c
     gcc -o foo foo.c
```

`foo` — это цель, а `foo.c` — предпосылка. Если `foo.c` новее, чем `foo`, или `foo` не существует, то make исполняет команду `gcc -o foo foo.c`. Если `foo` новее `foo.c`, то ничего не произойдет. Именно так make экономит наше время: не осуществляет сборку целей, которые не нужно собирать.

В более сложном примере предпосылка одного правила может быть целью в другом. В данной ситуации все остальные зависимости должны быть оценены перед тем, как это произойдет в отношении текущей зависимости. Взгляните на следующий пример:

```
# Правило 1
program: object.o
        gcc -o program object.o

# Правило 2
source.c:
        echo 'main() {}' > $@

# Правило 3
object.o: source.c
        gcc -c source.c -o $@

# Правило 4
program2: program2.c
        gcc -o program2 program2.c
```

Начиная с пустого каталога, если вы применили make без аргументов, вы увидите следующий вывод:

```
$ make
echo 'main() {}' > source.c
gcc -c source.c -o object.o
gcc -o program object.o
```

Инструмент make оценивает первое попавшееся в файле Makefile правило и останавливается, если зависимость удовлетворена. Остальные правила подвергаются оценке, если необходимо, чтобы они удовлетворяли правилу 1. Разбор происходит в таком порядке:

- **правило 1**: `program` требуется `object.o`:
  - `object.o` — цель правила 3; оценить правило 3 перед проверкой даты файла;
  - если `object.o` новее, чем `program`, осуществить сборку `program` с помощью `gcc`;

- **правило 3**: `object.o` требуется `source.c`:
  - `source.c` — цель правила 2; оценить правило 2 перед проверкой даты файла;
  - если `source.c` новее, чем `object.o`, осуществить сборку `object.o` с помощью `gcc`;
- **правило 2**: `source.c` не имеет предпосылок:
  - если `source.c` не существует, осуществить его сборку с помощью команды `echo`.

Обратите внимание, что зависимости определяют порядок исполнения команд. Не считая первого правила, порядок правил в Makefile ни на что не влияет. Мы можем, например, поменять правило 2 и правило 3 местами в Makefile, но это не приведет к изменениям в поведении.

Нужно также отметить, что правило 4 не оказывает влияния на сборку. Когда цель первого правила определена как более новая, исполнение make останавливается. Поскольку из правила 4 программе ничего не нужно, сборка `program2` не производится, а `program2.c` не требуется. Но это не значит, что правило 4 здесь лишнее или бесполезное. Можно без труда ввести команду `make program2`, чтобы дать указание произвести оценку правила 4. Правила Makefile могут быть независимыми друг от друга, сохраняя при этом свою полезность.

Инструмент make также может производить сборку определенных целей в Makefile, что позволяет обходить правило по умолчанию. Подобная методика, например, предпочтительна при сборке одиночного объекта в проекте на стадии разработки. Допустим, вы модифицировали `program2.c` и хотите убедиться, что ее компиляция пройдет без осложнений. Введем такую команду:

```
$ make program2.o
```

Если зависимости `program2.o` являются прямыми, данная команда только выполнит компиляцию `program2.c`.

Программа make еще может производить сборку так называемых *псевдоцелей*, которые являются целями, не представляющими файловых имен. Псевдоцель может иметь произвольное имя, но обычно соблюдается определенная система обозначений. Примером здесь может быть использование псевдоцели с именем `all` в качестве первого правила в Makefile. Чтобы понять необходимость псевдоцелей, изучите следующий пример, где у нас имеются две программы, которые необходимо собрать как часть нашего Makefile:

```
program1: a.o b.o
      gcc -o program1 a.o b.o

program2: c.o d.o
      gcc -o program2 c.o d.o
```

Если использовать команду `make` без аргументов, она осуществит сборку `program1` и завершит работу. Чтобы избежать этого, необходимо потребовать от пользователя, чтобы он указал в командной строке обе программы:

```
$ make program1 program2
```

Это сработает, однако возможность применять `make` без аргументов — неплохой способ сделать свой код легким для сборки. Для этого добавим псевдоцель

с именем `all` в качестве первого правила в Makefile, а `program1` и `program2` используем в роли предпосылок:

```
all: program1 program2

program1: a.o b.o
      gcc -o program1 a.o b.o

program2: c.o d.o
      gcc -o program2 c.o d.o
```

Вы можете ввести `make` или `make all`, но в результате все равно будет осуществлена сборка как `program1`, так и `program2`. Несмотря на то что имя `all` является стандартным, псевдоцели можно также присвоить имя `fred` или другое. Поскольку она является первой целью, то подвергнется оценке при запуске make без аргументов. А так как это псевдоцель, ее имя не имеет значения.

Обычно правило, которое содержит псевдоцель, не ассоциировано с командами, а имя псевдоцели выбирается таким образом, чтобы оно не конфликтовало с именами файлов в сборке. Для справки: в GNU-версии make содержится встроенная псевдоцель, которую можно использовать в качестве «ключа»:

```
.PHONY: all
```

Эта строка говорит make не искать файл с именем `all` и всегда считать эту цель устаревшей.

Основы Makefile: определение переменных

Инструмент make позволяет выбирать параметры сборки с помощью переменных. Синтаксис определения переменной довольно прост:

```
VAR = value
VAR := value
```

Оба этих определения полностью совпадают, за исключением того, что форма `:=` позволяет переменным ссылаться на себя без рекурсии. В этом можно убедиться, если к переменной необходимо добавить текст:

```
VAR = value

# Ошибка! Вызывает бесконечную рекурсию
VAR = $(VAR) more

# Все в порядке, := предотвращает рекурсию
VAR := $(VAR) more

# GNU-расширение, которое делает то же самое
VAR = value
VAR += more
```

GNU-версия make допускает альтернативный синтаксис для определения переменных с использованием двух ключевых слов: `define` и `endef`. GNU-эквивалент синтаксиса будет выглядеть так:

```
define VAR
value
endef
```

До и после ключевых слов необходимы новые строки, которые не будут являться частью определения переменной.

При присваивании имен переменным обычно используются буквы в верхнем регистре, хотя это и не обязательно. Значение переменной может содержать любые ASCII-символы, однако они сохраняются дословно и не имеют значения, пока используются в контексте.

Пробелы и новые строки в переменных

Программа make автоматически удаляет начальный и конечный пробелы в значениях переменных, если используется традиционный синтаксис. В частности, любые пробелы после знака = в содержимом переменных удаляются точно так же, как и все пробелы перед новыми строками.

В отношении пробелов в переменных нужно учитывать следующие моменты:

- начальный и конечный пробелы отсутствуют в переменных, определяемых посредством традиционного синтаксиса;
- пробел до и после обратного слеша при использовании традиционного синтаксиса сжимается;
- начальный и конечный пробелы могут сохраняться при использовании традиционного синтаксиса, для чего служит предопределенная переменная $();
- синтаксис define сохраняет все пробелы, включая новые строки, однако переменные с вставленными новыми строками имеют ограниченное применение.

Основы Makefile: ссылка на переменные и их модификация

Вы уже знаете, что в синтаксисе для ссылки на переменную необходимо заключать ее имя в круглые или фигурные скобки, как, например, здесь:

```
$(MACRO)        # наиболее часто встречается
${MACRO}        # менее распространено, но верно
```

Строго говоря, это касается только имен переменных, состоящих более чем из одной буквы, что является оптимальной длиной имени любой переменной. Переменная с именем из одной буквы не нуждается в круглых или фигурных скобках.

В отличие от переменных в сценариях, значения которых меняются по ходу исполнения, значения переменным в Makefile присваиваются лишь раз и никогда не меняются во время осуществления сборки программ[1]. Это сбивает с толку людей, работающих с файлами Makefile и интуитивно предполагающих, что присваивание

[1] Исключением являются так называемые «автоматические» переменные, о которых мы поговорим позднее.

значений переменным осуществляется в той же последовательности, в какой они расположены в Makefile, или что они имеют область и определенное время жизни. Все переменные в файле Makefile имеют глобальную область, а время их жизни определяется продолжительностью процесса сборки. В файле Makefile переменная может определяться множество раз, но лишь последнее определение в Makefile обуславливает ее значение. Взгляните на следующий Makefile:

```
FLAGS = first

all:
      @echo FLAGS=$(FLAGS)

FLAGS = second

other:
      @echo FLAGS=$(FLAGS)
```

Вы можете подумать, что переменная `FLAGS` будет иметь одно значение для цели `all`, а другое — для цели `other`, но на самом деле это не так. Определение переменной `FLAGS` осуществляется еще до того, как будут оценены какие-либо правила. Когда переменная `FLAGS` уже определена, make отбросит старые определения, если наткнется на новое определение, то есть самое последнее определение, содержащееся в файле, и обусловит значение переменной.

Уменьшаем размер файлов Makefiles с помощью неявных правил

По мере создания все большего количества файлов Makefile становится очевидным, что многие правила основаны на нескольких простых шаблонах. В больших файлах Makefile можно разместить большое количество правил, которые будут идентичны во всем, за исключением целей и предпосылок. Это влечет за собой множество операций по их копированию и вставке в текстовый редактор, что, как известно любому разработчику, часто приводит к глупым ошибкам.

К счастью, make предоставляет в распоряжение пользователя неявные правила, которые позволяют описывать подобные шаблоны, не прибегая к копированию и вставке правил. Инструмент make содержит большое количество предопределенных неявных правил, которые покрывают большинство шаблонов, встречающихся в файлах Makefile. Благодаря встроенным неявным правилам написание правил возможно без каких-либо инструкций. При этом make можно использовать даже без Makefile! Ради забавы можете сами попробовать в пустом каталоге следующее:

```
$ echo "main() {}" > foo.c
$ make foo
cc     foo.c   -o foo
```

Используя лишь неявные правила, make определяет, что вы хотите создать из `foo.c`. программу под именем `foo`.

Неявные правила позволяют делать файлы Makefile компактными и облегчают вашу работу. GNU-версия make имеет в своем составе множество неявных правил,

с помощью которых можно сделать почти все, что потребуется. Неявные правила следует использовать при каждом удобном случае.

Один из способов определять неявные правила заключается в использовании правил суффикса. К примеру, правило суффикса, используемое GNU-версией make для создания объектных файлов на основе исходных файлов C, выглядит следующим образом:

```
.c.o:
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

Суть такова: «Если вы видите цель с расширением `.o` и для нее отсутствует явное правило, ищите файл с той же базой и расширением `.c`. Если вы найдете его, введите следующие команды». Как видно из примера, команды для определения неявных правил обычно размещаются в переменных, которые позволяют использовать эти правила в дальнейшем.

Вы можете просмотреть весь перечень встроенных переменных и неявных правил, введя команду `make -p`. Применив ее к файлу `COMPILE.c`, можно увидеть, что он определяется с помощью дополнительных переменных:

```
COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
CC = gcc
```

Что интересно, только одна из этих дополнительных переменных (`CC`) действительно определена; все остальные заменены пустыми строками. Программа make позволяет ссылаться на неопределенные переменные без вывода сообщений об ошибке. Вы можете воспользоваться этим для придания компактности файлу Makefile, просто добавив следующую строку:

```
CFLAGS = -g
```

Данный файл Makefile, содержащий одну строку, можно применять для компилирования объектов при включенной отладке. Но, поскольку make позволяет определять переменные в командной строке, вы можете пропустить Makefile или переопределить `CFLAGS` в командной строке, как показано далее:

```
$ make CFLAGS=-g foo
gcc -g    foo.c   -o foo
```

Переменные в командной строке переопределяют все определения в файле Makefile. Таким образом, если ваш Makefile является громоздким или вообще отсутствует, вы можете полноценно управлять процессом сборки из командной строки. В табл. 2.1 приведены переменные, наиболее часто используемые в неявных правилах, которые могут быть переопределены в командной строке.

Таблица 2.1. Общие переменные, используемые в неявных правилах

Переменная	По умолчанию	Описание
CC	gcc	Компилятор C
CXX	g++	Компилятор C++

Продолжение ⇩

Таблица 2.1 *(продолжение)*

Переменная	По умолчанию	Описание
CFLAGS	none	Флаги передаются в компилятор C с использованием неявных правил
CXXFLAGS	none	Флаги передаются в компилятор C++ с использованием неявных правил
CPPFLAGS	none	Флаги передаются в препроцессор с использованием неявных правил, включая C++, C, несколько правил языков компоновки и некоторых других. Типичные флаги: -I, -D и -U

Благодаря гибкости неявных правил для объектного кода вам может никогда не потребоваться писать правила для объектных файлов. Неявные правила существуют для языков C, C++, Fortran и многих других. В табл. 2.2 приведены наиболее распространенные из них, а с полным списком вы можете ознакомиться, введя команду `make -p` либо обратившись к справочной документации make.

Таблица 2.2. Правила суффикса, по умолчанию используемые для создания объектного кода

Язык	Расширения	Команда
C	.c	$(CC) -c $(CPPFLAGS) $(CFLAGS)
C++	.cpp, .cc, .C	$(CXX) -c $(CPPFLAGS) $(CXXFLAGS)
Assembler	.s	$(AS) $(ASFLAGS)
Assembler6 (это правило не создает объектного кода, а осуществляет препроцессинг файла .S в файл .s, который затем компилируется правилом, указанным выше)	.S	$(CPP) $(CPPFLAGS)
Pascal	.p	$(PC) -c $(PFLAGS)
Fortran	.f	$(FC) -c $(FFLAGS)
Fortran	.F	$(FC) -c $(FFLAGS) $(CPPFLAGS)

Определение пользовательских неявных правил

Несмотря на то что GNU-версия make содержит большое количество неявных правил, которые позволяют решать большинство задач, бывают ситуации, когда необходимое правило среди них отсутствует. В этом случае вы можете написать явное правило либо определить собственное неявное правило.

Допустим, вам необходимо скомпилировать группу модулей C++, каждый из которых имеет расширение `.cxx`. Подобное расширение не относится к трем расширениям C++, которые распознает GNU-версия make[1]. Можно переименовать эти файлы или создать явное правило для каждого из них, но проще будет создать новое неявное правило. Для этого потребуется написать лишь две строки:

[1] Программа make распознает такие расширения C++, как .C, .cpp и .cc.

```
.SUFFIXES: .cxx # let make know that this is a new extension

.cxx.o:
        $(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $<
```

Без псевдоцели `.SUFFIXES:` `make` не сможет использовать неявное правило, поскольку не распознает файлы, заканчивающиеся на `.cxx`. Правило суффикса, приведенное выше, совместимо с другими версиями make, как и переменные, использованные для его определения. GNU-версия make предлагает альтернативный синтаксис, который более гибок и менее разборчив в отношении суффиксов и называется ***правилом шаблона***. Правило шаблона для моих исходных файлов `.cxx` будет иметь следующий вид:

```
%.o : %.cxx
        $(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $<
```

Правила шаблона привносят меньше неразберихи, чем правила суффикса, поскольку им не требуется псевдоцель `.SUFFIXES:`. Они больше напоминают обычные правила по сравнению с правилами суффикса, так как цель в них располагается слева, а предпосылка — справа. Вместо определенных файловых имен для целей и предпосылок правила шаблона используют символ `%` при сопоставлении файловых имен или целей в Makefile, что сильно напоминает использование символов подстановки при сопоставлении имен файлов в оболочке. Когда программа make сталкивается с целью (или файловым именем), которая соответствует шаблону, то она использует часть имени этой цели, которая соответствует символу `%`, чтобы отыскать предпосылку.

Использование автоматических переменных для повышения читабельности

Поскольку явные правила и правила шаблона не используют в своей работе файловые имена, make содержит *автоматические переменные*, которые помогают определять команды для этих правил. В предыдущих примерах я использовал автоматическую переменную `$<`, которая принимает значение первой предпосылки в правиле. В отличие от переменных make, автоматические переменные содержат уникальные значения для каждого правила. В табл. 2.3 приведен перечень наиболее полезных автоматических переменных, используемых в неявных правилах.

Таблица 2.3. Полезные автоматические переменные, входящие в состав GNU-версии make

Переменная	Значение
$@	Файловое имя цели
$^	Имена всех предпосылок без дубликатов
$+	Имена всех предпосылок, включая дубликаты
$<	Имя первой предпосылки в правиле
$?	Имена всех предпосылок, которые новее цели
$*	В случае с правилом суффикса это базовое имя (или ствол) цели. Если правило, например, .c.o:, а соответствующая цель — foo.o, то данное значение будет foo. В случае с некоторыми правилами шаблона это значение может отличаться от предполагаемого

Как вы уже видели, определяемые пользователем переменные могут содержать другие переменные. Неочевидным может быть то, что переменные также могут содержать автоматические переменные. Это означает, что их значения могут изменяться в процессе сборки. Автоматические переменные жизненно необходимы неявным правилам, но могут быть весьма полезны и для явных правил. Вот простой шаблон Makefile для создания программы:

```
program: $(OBJS)
       $(CC) -o $@ $^
```

Командная часть состоит из предопределенной переменной (CC) и двух автоматических переменных. К сожалению, инкапсулировать все это в неявном правиле не представляется возможным из-за неясности шаблона; program может иметь любое имя, а OBJS может содержать любое количество объектов с произвольными именами. В случае использования make отсутствует шаблон, которому можно следовать.

Если вы решите использовать аналогичное правило для создания дополнительных программ в том же файле Makefile, вам придется копировать и вставлять эти команды в отношении каждой программы, как показано ниже:

```
program1: $(PROG1_OBJS)
      $(CC) -o $@ $^

program2: $(PROG2_OBJS)
      $(CC) -o $@ $^
# etc...
```

Посредством использования автоматических переменных вы можете копировать и вставлять строки без изменений в командную строку правила. Это может показаться слегка необычным, но реально позволяет облегчить работу. Вставляя автоматические переменные в переменную с более интуитивным названием, вы можете сделать шаблон понятным для следования. Например:

```
build_C_program=$(CC) -o $@ $^

program1: mod1.o mod2.o
      $(build_C_program)

program2: mod3.o mod4.o
      $(build_C_program)
```

Теперь «громоздкие» автоматические переменные стали частью определения build_C_program, однако они ведут себя так, как и предполагается, когда build_C_program выступает в роли команды. Данному правилу легче следовать, при этом разработчикам нет надобности знать, что собой представляет build_C_program, если они сами того не пожелают.

Манипулирование переменными

Для того чтобы понять, в чем состоит необходимость манипулирования значениями переменных, рассмотрим типичный пример шаблона подстановки текста в UNIX-версии make:

```
SRCS=foo.c bar.c
OBJS=$(SRCS:.c=.o)
```

Данный синтаксис называется *ссылкой подстановки* и может использоваться во всех версиях make. Мы берем список исходных файлов и используем его для создания списка объектных файлов. В данном примере берем значение переменной `SRCS`, вместо расширений `.c` подставляем `.o` и сохраняем результат в переменной `SRCS`. Поскольку у нас имеется два исходных модуля с именами `foo.c` и `bar.c`, нужно, чтобы переменная `SRCS` содержала `foo.o` и `bar.o`. Подстановка текста более удобна, чем повторное впечатывание всех имен во вторую строку, которое утомительно и может привести к ошибкам. Обычно файлы Makefile содержат такие списки, что делает использование подстановок текста необходимостью.

Ссылка подстановки в данном примере сработала, так как все исходные модули имеют одинаковое расширение. А если переменная `SRCS` будет содержать смесь исходных модулей C и C++? Например:

```
SRCS=foo.c bar.cpp
OBJS=$(SRCS:.c=.o)
```

Синтаксис ссылки подстановки не поддерживает более одного шаблона расширения файла, поэтому в данном примере подстановка не осуществляется в отношении `bar.cpp`, поскольку он не соответствует шаблону `.c`. В результате переменная `OBJS` будет содержать `foo.o bar.cpp`, а это не то, что нам нужно.

Типичным решением этой проблемы в UNIX-версии make будет создание дополнительной переменной:

```
CSRCS=foo.c
CXXSRCS=foo.cpp
OBJS=$(CSRCS:.c=.o) $(CXXSRCS:.cpp=.o)
```

Подобная методика привносит неразбериху, особенно если вы имеете дело с множеством списков.

В следующем разделе мы рассмотрим расширения функций GNU-версии make, которые обладают большей гибкостью.

Манипулирование переменными с помощью функций

Как вы уже могли убедиться, синтаксис ссылок подстановки может быть весьма удобен, однако он имеет некоторые ограничения. Расширения функций GNU-версии make являются универсальными инструментами для управления содержимым переменных. Подобную манипуляцию можно осуществлять двумя способами:

```
OBJS=$(SRCS:.c=.o)
OBJS:=$(patsubst %.c, %.o, $(SRCS)))
```

В первой строке использована уже знакомая вам ссылка подстановки. Вторая строка представляет собой эквивалент ссылки подстановки с использованием функции `patsubst`. Данная функция имеет те же ограничения, что и ссылка подстановки: поддерживается только одно расширение единократно. Но функции могут быть вложенными, что невозможно в случае со ссылками подстановки. К примеру, для включения дополнительных расширений можно воспользоваться следующим синтаксисом:

```
OBJS:=$(patsubst %.cpp, %.o, $(patsubst %.c, %.o, $(SRCS)))
```

Теперь мы можем преобразовывать множественные расширения файлов без необходимости в дополнительных переменных.

Рассмотрим еще один способ использования других функций. Он немного более компактен и универсален:

```
OBJS:=$(addsuffix .o, $(basename $(SRCS)))
```

Функция `basename` берет список имен файлов (в данном случае это содержимое переменной `SRCS`) и удаляет расширение у каждого из них. При этом неважно, каково расширение: `basename` удаляет все символы, которые следуют после последней точки в тексте. Мы используем полученный результат и присоединяем `.o` в конце каждого `basename` с помощью функции `addsuffix`. В итоге получаем желаемый список объектов.

Еще одна полезная функция — `shell`. Вспомните, как в примере выше мы использовали обратные кавычки для создания файлового имени на основе даты и времени:

```
NAME = `date +%02d%02m%02y`.dat
```

Вы уже видели, что данный шаблон оказывается неудачным, если обратные кавычки выступают в роли предпосылки или цели. GNU-версия make содержит функцию `shell`, которая служит альтернативным средством достижения необходимого нам результата:

```
NAME = $(shell date +%02d%02m%02y).dat
```

В отличие от обратных кавычек, данная переменная может без проблем использоваться в качестве предпосылки или цели, поскольку GNU-версия make оценивает команду оболочки при инициализации переменной прежде, чем будут оценены какие-либо правила. В данном примере это как раз то, что нам и нужно. Но могут возникнуть ситуации, когда необходимо использовать обратные кавычки. Взгляните на следующее правило:

```
CURRENT_TIME=$(shell date +%T)
something:
      @echo started  $(CURRENT_TIME)
      sleep 5
      @echo finished $(CURRENT_TIME)
```

Вывод будет иметь следующий вид:

```
$ make
started  9:49:45
sleep 5
finished 9:49:45
```

Данный результат, скорее всего, будет отличаться от ожидаемого вами. Проблема заключается в том, что команда оболочки `` `date +%T` `` оценивается только один раз в начале процесса сборки, то есть каждый раз, когда мы будем использовать ее при выполнении сборки, она будет иметь одинаковое значение. Если же вам необходимо, чтобы значение изменялось, придется всякий раз обращаться к оболочке, что как раз и делают обратные кавычки. Например:

```
CURRENT_TIME=`date +%T`
```

Эта команда выдаст необходимый нам результат:

```
$ make
started  9:49:45
sleep 5
finished 9:49:50
```

Обратите внимание, что CURRENT_TIME является переменной, значение которой остается неизменным в обоих примерах. В первом примере CURRENT_TIME содержит вывод команды оболочки, исполнявшейся в начале сборки. Во втором примере CURRENT_TIME содержит текст команды оболочки, исполняемой во время сборки. Изменяется лишь вывод команды оболочки, а содержимое переменной остается прежним.

Полный перечень функций можно просмотреть в справочной системе к make. Некоторые наиболее полезные функции перечислены в табл. 2.4.

Таблица 2.4. Функции для манипулирования текстом

Функция	Применение	Описание
subst	$(subst from, to, text)	Возвращает содержимое аргумента text со всеми случаями аргумента from, замененными содержимым аргумента to
patsubst	$(patsubst from-pattern, to-pattern, filenames)	Возвращает разделенный пробелами список файловых имен с замененными шаблонами файловых имен. Синтаксис шаблона аналогичен синтаксису, используемому для правил шаблона. Хотя данная функция предназначена для файловых имен, она также применима к любому тексту
strip	$(strip string)	Удаляет начальный и конечный пробелы из аргумента string
findstring	$(findstring match, string)	Возвращает аргумент match, если он присутствует в аргументе string; в противном случае возвращает пустую строку
filter	$(filter patterns, filenames)	Возвращает разделенный пробелами список файлов, найденных в аргументе filenames, которые соответствуют аргументу patterns. Аргумент patterns может содержать множественные шаблоны, разделенные пробелами, и использует аналогичный синтаксис, применяемый в случае с правилами шаблона
filter-out	$(filter-out patterns, filenames)	Возвращает разделенный пробелами список файлов из аргумента filenames, которые не соответствуют аргументу patterns. Аргумент patterns может содержать множественные шаблоны, разделенные пробелами, и использует аналогичный синтаксис, применяемый в случае с правилами шаблона
sort	$(sort text)	Возвращает разделенный пробелами список из аргумента text, сортированный в лексическом порядке
word	$(word n, text)	Возвращает n-е слово, найденное в аргументе text. Счет слов начинается с 1
wordlist	$(wordlist first, last, text)	Возвращает набор слов из аргумента text, начиная с first и заканчивая last
words	$(words text)	Возвращает количество слов в аргументе text
error	$(error message)	Используется в условных конструкциях (они будут рассмотрены позднее). Останавливает процесс сборки выводом сообщения об ошибке

Определение и использование собственных функций

В тех маловероятных ситуациях, когда для решения своих задач вам не удастся отыскать встроенную функцию, GNU-версия make дает возможность определять собственные функции. Доступ к определяемым пользователем функциям можно получить с помощью встроенной функции `call`:

```
$(call myfunction, arg1, arg2)
```

Обратите внимание, что вызов определяемой пользователем функции отличается от вызова других функций. Необходимо использовать встроенную функцию `call` в сочетании с именем собственной функции, что передается как аргумент. Дополнительные аргументы отделяются запятыми. Определение `myfunction` выглядит как переменная, за тем исключением, что в данный момент она имеет доступ к позиционным аргументам. Вот более сложный пример:

```
myfunction = @echo $(1) $(2)

all:
      $(call myfunction,hello,world)
```

Вывод данного файла Makefile будет следующим:

```
$ make
hello world
```

Термин *функция* немного вводит в заблуждение. Поскольку программа make — это не язык программирования, она обеспечивает лишь минимальную проверку на ошибки. Подобно переменным, make возвращает пустые строки, если вы допустили ошибку. К примеру, поскольку определение функции не отличается от определения переменной, вы можете использовать функцию как переменную. Звучит странно, однако это действительно так благодаря синтаксису функций. Если функция используется как переменная, она ведет себя как функция, вызванная без аргументов. Например:

```
$(call myfunction) # is the same as...
$(myfunction)
```

В обоих случаях позиционные аргументы — `$(1)`, `$(2)` и т. д. — заменяются пустыми строками.

Условные конструкции

Условные конструкции представляют собой еще одно полезное GNU-расширение для управления поведением make. GNU-версия make поддерживает только четыре типа условных конструкций, которые отражены в табл. 2.5.

Таблица 2.5. Условные конструкции, поддерживаемые GNU-версией make

Условная конструкция	Применение
ifeq	Проверить, равны ли два значения
ifneq	Проверить, не равны ли два значения
ifdef	Проверить определенность переменной
ifndef	Проверить неопределенность переменной

Существует два теста проверки с дополнениями: `if equal` и `if defined`. Поскольку у make отсутствует оператор `not`, необходимо иметь два отдельных ключевых слова для `equal` и `not equal`, а также для `defined` и `not defined`.

Каждый условный блок может иметь опциональное предложение `else`, но завершаться должно на `endif`. Базовый синтаксис таков:

```
conditional test
makefile text evaluated when test is true
else
makefile text evaluated when test is false
endif
```

Текст в условном предложении может представлять собой любые допустимые строки make, однако условные конструкции не могут использоваться в командной части правил или внутри определения переменной/функции. Это устанавливает некоторые ограничения того, каким образом могут использоваться условные конструкции.

Начнем с простого примера, воспользовавшись условной конструкцией `ifeq`:

```
ifeq ($(shell uname -o),GNU/Linux)
      CPPFLAGS += -DLINUX
endif
```

Здесь мы применили условную конструкцию `ifeq` для анализа типа операционной системы. Она использует вывод команды оболочки `uname -o` для индикации типа операционной системы, в данном случае — `GNU/Linux`. Если он указан в файле Makefile в системе Linux, то переменная `CPPFLAGS` будет включать флаг `-DLINUX`. Как вы уже знаете, `CPPFLAGS` употребляется в неявных правилах для целей, сборка которых осуществляется с помощью инструментов, использующих препроцессор C. Это первый способ применения условных конструкций для поддержки сборочных окружений.

Добавление предложения `else` осуществляется весьма просто:

```
ifeq ($(shell uname -o),GNU/Linux)
      CPPFLAGS += -DLINUX
else
      CPPFLAGS += -DOS_UNKNOWN
endif
```

Благодаря тому что условные предложения также можно использовать как обертку для правил, такой синтаксис будет верным:

```
ifeq ($(shell uname -o),GNU/Linux)
      all: linux_programs
else
      all: bsd_programs
endif
```

Командная часть правила может располагаться как внутри, так и за пределами условного предложения.

Нужно также отметить, что условные конструкции оцениваются в соответствии с присваиваемыми переменным значениями, как видно из следующего простого примера:

```
ifeq ($(A),0)  # A isn't defined yet
all:
    @echo $(A) == 0
else
all:
    @echo $(A) != 0
endif
A=0
```

В результате вывод файла Makefile будет иметь нелогичный вид:

```
$ make
0 != 0
```

Проблема заключается в том, что макро A не определено даже после условной конструкции, и в результате проверка `ifeq` потерпит неудачу (`false`). Теперь понимаете, что я имел в виду, когда говорил об отсутствии полноценной проверки на ошибки?

Предложение `ifdef` проверяет определенность значений переменных, которые могут быть полезны при использовании переменных в качестве параметров командной строки. Например:

```
ifdef (debug)
      CFLAGS += -g
else
      CFLAGS += -O2
endif
```

Как вы уже знаете, `CFLAGS` — это встроенная переменная, используемая для неявной сборки объекта из исходного кода C. В данном примере для переменной `CFLAGS` параметром по умолчанию является включение оптимизации посредством переключателя `-O2`. Если вам необходимо осуществлять компиляцию для отладки, в командной строке вызовите make с параметром `debug`:

```
$ make debug=1
```

Предложение `ifdef` можно использовать в больших и сложных файлах Makefile для проверки того, присвоено ли значение необходимой переменной:

```
ifndef (A_VITAL_VARIABLE)
$(error A_VITAL_VARIABLE is not set)
endif
```

Извлечение файлов исходного кода из различных каталогов

Зачастую при работе над проектами с большим количеством файлов последние размещаются в одном каталоге, а их сборка осуществляется в другом. Размещение в каталоге файлов одного типа позволяет упорядочить их. Обычно для этого используется переменная `VPATH`. Но GNU-версия make расширяет данную функциональность несколькими путями.

В принципе, переменная `VPATH` функционирует во многом аналогично переменной оболочки `PATH`. Она содержит список каталогов, разделенных двоеточиями,

в котором осуществляется поиск целей и предпосылок. У вас может иметься проект с двумя каталогами: src для исходных файлов и bin для объектов. В данном случае файл Makefile будет располагаться в каталоге ./bin и выглядеть так:

```
VPATH=../src

foo: foo.c
      $(CC) -o $@ $^
```

Переменная VPATH указывает, что программе make следует осуществлять поиск целей и предпосылок в каталоге ../src, а также в текущем каталоге. Местоположение foo.c указывать не обязательно, если данный файл является предпосылкой. Инструмент make самостоятельно подставит нужный путь. Таким образом, предполагая, что foo.c находится в каталоге ../src, получим вывод, который будет иметь следующий вид:

```
$ make
gcc -o foo ../src/foo.c
```

Если копия foo.c имеется в текущем каталоге, make также попытается его извлечь, но здесь может возникнуть проблема. Программа всегда сначала проверяет текущий рабочий каталог. Иными словами, если получится так, что у вас окажется два разных файла с одним и тем же именем сразу в двух каталогах, VPATH вам уже не поможет.

GNU-версия make добавляет к VPATH расширение посредством директивы vpath (обратите внимание на использование нижнего регистра). С ее помощью вы можете указывать шаблоны файловых имен, то есть make будет выполнять поиск файлов заданного типа в определенных каталогах. В качестве показательного примера можно поместить файл исходного кода C++ в каталог с именем /cxxsrc, а файл исходного кода C — в каталог под именем /csrc. Файл Makefile в данном случае будет таким:

```
vpath %.cpp ./cxxsrc
vpath %.c ./csrc
```

Большинство проектов среднего размера не нуждаются в применении директивы vpath или даже переменной VPATH. Используя их, можно легко запутаться. Они употребляются только в очень крупных проектах либо для обхода некоторых унаследованных ограничений в сборочной среде. Сборку некоторых исходных файлов, распространяемых вместе с GNU-проектами, необходимо осуществлять в пустом каталоге. В подобных проектах для считывания исходных файлов из каталогов, предназначенных только для чтения, применяется директива vpath.

2.2.3. Связь программ

Большинство разработчиков пребывают в блаженном неведении относительно редактора связей, поскольку обычно значительную часть работы выполняет компилятор. Несмотря на то что существует команда вызова редактора связей (ld),

она почти никогда не используется напрямую. Для всех нас важны имена и местоположение библиотек, с которыми необходимо осуществить связку. На самом деле множество мелких операций следует отводить редактору связей. К счастью, компилятор именно так и поступает в фоновом режиме.

Для того чтобы взглянуть, как это все работает, добавьте параметр `-v` к компилятору C. Убрав все лишнее, я попытался продемонстрировать, как это выполняется в моей системе (если вы можете в это поверить):

```
$ gcc -v -o hello hello.c

 /usr/libexec/gcc/i386-redhat-linux/4.0.1/cc1 -quiet -v hello.c -quiet -dumpbase
hello.c -auxbase hello -version -o /tmp/ccoxYr4k.s

 as -V -Qy -o /tmp/ccw3IciH.o /tmp/ccoxYr4k.s

/usr/libexec/gcc/i386-redhat-linux/4.0.1/collect2 --eh-frame-hdr -m elf_i386-
dynamic-linker /lib/ld-linux.so.2 -o hello /usr/lib/gcc/i386-redhatlinux/4.0.1/../
../../crt1.o /usr/lib/gcc/i386-redhat-linux/4.0.1/../../../crti.o /usr/lib/gcc/i386-
redhat-linux/4.0.1/crtbegin.o -L/usr/lib/gcc/i386-redhatlinux/4.0.1 -L/usr/lib/gcc/
i386-redhat-linux/4.0.1 -L/usr/lib/gcc/i386-redhatlinux/4.0.1/../../.. /tmp/ccw3IciH.
o -lgcc --as-needed -lgcc_s --no-as-needed -lc -lgcc --as-needed -lgcc_s --no-as-
needed /usr/lib/gcc/i386-redhatlinux/4.0.1/crtend.o /usr/lib/gcc/i386-redhat-
linux/4.0.1/../../../crtn.o
```

Компилятор проходит три стадии. На первой стадии он конвертирует исходный код на компоновочный язык. На второй стадии запускается компоновщик, создающий объектный файл. На третьей стадии, самой трудной, в действие вступает редактор связей. Несмотря на то что необходимой командой в данном случае является `collect2`, в действительности осуществляется вызов `ld` с аналогичными параметрами.

Компиляторы C и C++ имеют два основных режима работы: со связыванием и без него. Наличие флага `-c` означает, что вы осуществляете компиляцию без связывания. Компилятор лишь создает объектные файлы. Если данный флаг отсутствует, то компилятор создает объектные файлы и вызывает редактор связей для связывания итоговых объектов с программой.

2.2.4. Понятие библиотек

Библиотека — это контейнер, содержащий многократно используемый объектный код. При вызове функции POSIX, например, вам необходимо связать свой программный код с библиотекой POSIX, чтобы она смогла работать. Для каждой функции существует множество библиотек. Некоторые из них, такие как стандартная библиотека C, являются обязательными; большинство прочих опциональны. Путем инкапсуляции функций в библиотеках разработчики могут обеспечивать малый объем занимаемой программой памяти (и исполняемого файла), отбирая лишь те библиотеки, без которых программа вообще не будет работать.

Библиотеки бывают двух типов: динамические и статические. Статическая библиотека представляет собой архив объектных файлов, создаваемый с помощью команды ar. Редактор связей поочередно считывает содержащиеся в архиве объектные файлы и пытается отыскать внешние имена. Редактор связей может отбрасывать любые необязательные объектные файлы. Таким образом, несмотря на то, что статическая библиотека может содержать сотни объектов и мегабайты объектного кода, ваш код может уместиться в одном или двух объектах и занимать намного меньший объем памяти. Если вы когда-нибудь заглянете внутрь статической библиотеки, то увидите, что в ней содержится множество модулей, каждый из которых наделен собственной функцией. Это дает редактору связей максимально широкие возможности по удалению ненужного кода.

Ключевое свойство статических библиотек заключается в том, что их копии нужны лишь тем разработчикам, которые осуществляют сборку исполняемых файлов. Для этого часть объектного кода в библиотеке или он весь копируется в исполняемый файл. Если исполняемый файл связан только со статическими библиотеками, то он называется статически связанным. Это означает, что вы можете скопировать такой файл в другую систему и исполнять его без необходимости в каких-либо дополнительных файлах.

В отличие от статической библиотеки, которая является архивом объектных файлов, динамическая библиотека сама представляет собой объектный файл. Динамические библиотеки создаются редактором связей, а статические — с использованием компилятора и команды ar. Когда вы осуществляете связку с динамической библиотекой, компилятор не копирует каких-либо объектов в ваш исполняемый файл. Вместо этого он записывает сведения о динамической библиотеке в исполняемый файл, который может стать *динамически связанным* во время исполнения. Это означает, что часть процесса связывания откладывается до момента исполнения. В этом заключаются преимущества по сравнению со статической библиотекой. Во-первых, это позволяет сделать исполняемый файл меньше по размеру, поскольку редактор связей не копирует в него объектный код. Во-вторых, в памяти требуется присутствие только одной такой библиотеки. Это позволяет сэкономить память, так как все программы, которые используют эту библиотеку, обращаются к одному разделу физической памяти.

Установка заплаток становится намного более простой, если вы имеете дело с совместно используемыми библиотеками, поскольку при этом вы не только исправляете имеющиеся в них ошибки, но и делаете то же самое в отношении всех программ в системе, которые используют эти библиотеки. Если бы такие библиотеки были статическими, вам пришлось бы производить пересборку всех программ, которые их используют, чтобы с помощью заплаток исправить имеющиеся в них одинаковые ошибки.

При своем исполнении динамически связанная программа проходит финальные стадии связывания, перед тем как перейдет к самой главной. Она осуществляет поиск необходимых динамических библиотек и выдает сообщение об ошибке, если не обнаруживает таковых. Это может стать проблемой, если вы решите перенести двоичные файлы на другие компьютеры, поскольку на них должны иметься все необходимые совместно используемые библиотеки; в противном случае программа не будет работать.

Система присваивания библиотекам имен

Обычно динамические библиотеки можно определить по расширению файлового имени `.so`. В Linux (и UNIX) такие библиотеки также называются совместно используемыми библиотеками или совместно используемыми объектами. Именно отсюда происходит расширение `.so`. Статические библиотеки имеют расширение `.a`, унаследованное от архиватора ar, с помощью которого они создаются. Чтобы можно было отличить архив AR (который едва ли используется где-либо еще) от статической библиотеки, файловые имена библиотек снабжаются префиксом `"lib"`. Таким образом, если вы создаете статическую библиотеку с именем `foo`, файл нужно назвать `libfoo.a`.

Если редактор связей встречает аргумент вроде `-lfoo`, то он автоматически переходит к поиску файла с именем `libfoo.a` по предопределенному пути. По традиции динамические библиотеки также используют префикс `lib`, поэтому редактор связей будет искать файл `libfoo.a` либо `libfoo.so`.

2.3. Процесс сборки

Теперь, когда вы познакомились с инструментами для создания проектов, вернемся на шаг назад и посмотрим, как они работают вместе. Для простоты восприятия сфокусируемся на проектах, сборка которых осуществляется на языке программирования C или C++. Язык C наиболее распространен, поскольку обеспечивает лучшую переносимость программ по сравнению с C++. Несмотря на то что C++ подходит для большинства проектов, разработчики, которые желают охватить как можно более широкий диапазон систем, предпочтут использовать C. Верите вы или нет, но существуют программисты, которые вообще не знакомы с C++.

Невзирая на то что GNU-компиляторы широко распространены в Linux-системах, их не обязательно использовать для сборки приложений на основе исходного кода. В частности, проекты с открытым исходным кодом, а также GNU-проекты акцентируют внимание на переносимости и не требуют применения GNU-компиляторов или операционной системы Linux. Аналогичные приложения, которые собираются и работают под Linux, также могут делать это в системах Solaris, IRIX, Free BSD и даже Windows. Переносимость не является особенностью, модной среди заботливых разработчиков; она жизненно важна для выживания проекта с открытым исходным кодом. Чем больше поддерживается платформ, тем больше пользователей и разработчиков будут обращать на него внимание. В этом заключается рецепт создания хорошего программного обеспечения.

2.3.1. GNU-инструменты сборки

GNU-система сборки стала де-факто стандартом в создании проектов с открытым исходным кодом. Она представляет собой комплексный набор инструментов, используемых разработчиками для генерирования дистрибутивов исходного кода,

сборка и установка которых не представляет труда. Типичный дистрибутив исходного кода формируется с помощью GNU-инструментария, в который входит сценарий `configure`, применяемый конечными пользователями для создания файлов Makefile. Поскольку вы также относитесь к ним, вам нет необходимости знать о существовании этих инструментов. Важно лишь запомнить три простых этапа сборки любой GNU-программы:

```
$ ./configure
$ make
$ make install
```

Однако не всегда все так просто. Иногда вам самим придется настраивать что-либо в соответствии со своими нуждами. Еще хуже то, что на любом из этих трех этапов вас может постигнуть неудача. В этом разделе GNU-инструменты сборки будут рассмотрены с позиции конечного пользователя. Полный охват данной тематики выходит за рамки книги, но вы можете воспользоваться множеством превосходных ресурсов в Интернете или справочной документацией, входящей в состав дистрибутивов GNU-инструментария.

2.3.2. Этап конфигурирования: сценарий configure

На этапе конфигурирования осуществляются все мелкие операции, с которыми нельзя справиться только с помощью файла Makefile. GNU-инструментарий сборки служит для создания программ, которые могут устанавливаться в любых POSIX-совместимых системах, то есть особенности вашей системы мало на что влияют. Исполнение сценария `configure` может отнимать много времени, поскольку он проверяет, какие библиотеки и приложения установлены. Он также может сгенерировать файлы заголовка C, чтобы передать сведения о системе в компилятор.

В некоторых проектах небольшого размера разработчики предпочитают использовать вместо сценария `configure` простой файл Makefile. Вы можете определить это по наличию или отсутствию этого файла в исходном архиве. Если файла Makefile нет, там будет присутствовать файл с именем `Makefile.in`, который используется сценарием `configure` для создания файла Makefile.

Сценарий `configure` анализирует вашу систему и проверяет, установлены ли в ней необходимые для компиляции проекта инструменты, например рабочий компилятор. GNU-инструменты, используемые для создания сценариев `configure`, наделяют разработчиков большой гибкостью в плане проверки всевозможных мелких деталей, касающихся системы. Это позволяет гарантировать, что вы сможете осуществить сборку проекта, не тратя на это слишком много времени. Но это также может быть недостатком, так как этап конфигурирования может оказаться продолжительным, особенно в случаях с небольшими проектами.

Сценарий `configure`, для создания которого применялись сборочные GNU-инструменты, обладает набором параметров, общих для всех проектов. Разработчики могут добавлять к ним новые, но, к сожалению, вовсе не обязательно, что при этом они будут снабжены документацией. В результате сборка проекта может превратиться в запутанный процесс, что нам абсолютно не нужно.

Некоторые базовые параметры, имеющиеся во всех сценариях `configure`, сосредоточены на тонкой настройке установки программного обеспечения (табл. 2.6).

Таблица 2.6. Распространенные параметры команды configure

Параметр	Применение
--prefix	Определяет корневой каталог установки, обычно это /usr/local
--bindir	Определяет местоположение файлов, которые в противном случае будут установлены в ${prefix}/bin. Данный каталог предназначен для исполняемых файлов, которые будут запускаться регулярными пользователями
--sbindir	Определяет местоположение файлов, которые в противном случае будут установлены в ${prefix}/sbin. Данный каталог предназначен для исполняемых файлов, которые будут запускаться администраторами или системными демонами
--datadir	Определяет местоположение служебных файлов с данными; по умолчанию ${prefix}/share
--libdir	Определяет местоположение совместно используемых библиотек; по умолчанию ${prefix}/lib
--mandir	Определяет местоположение установки страниц руководства man; по умолчанию ${prefix}/man
--program-prefix	Добавляет префикс к исполняемым файлам, расположенным в ${bindir} и ${sbindir}. Иногда используется для установки нескольких версий одной программы, так как позволяет присваивать уникальный префикс каждой версии
--program-suffix	Аналог --program-prefix за тем исключением, что добавляет к программе суффикс

Поскольку в обиходе используются различные версии GNU-инструментов сборки, параметры могут иметь небольшие отличия. Чтобы узнать, какие из них доступны в каком-либо проекте, необходимо ввести такую команду:

```
$ ./configure -help
```

Помимо параметров, перечисленных в табл. 2.6, сценарии `configure` допускают применение параметров, определяемых пользователем:

```
--enable-X
--disable-X
--with-X
--without-X (also --with-X=no)
```

Под `X` может скрываться все, что пожелает разработчик, однако данные параметры не всегда документируются. Все может стать еще запутаннее, так как сценарий примет любой параметр на месте `X`, даже если таковой отсутствует. Например, `configure --with-fries` без проблем сработает в любом проекте, разве что не сможет открыть бутылку с кетчупом.

2.3.3. Этап сборки: make

Этап сборки начинается с ввода команды `make` без аргументов. После этого выполняется сборка всех объектных файлов и связывание их с одной или несколькими программами. Целью по умолчанию является псевдоцель с именем `all`, но GNU-инструменты сборки создают другие псевдоцели, которые могут использоваться

в командной строке (табл. 2.7). Обычно они не требуются, но если процесс сборки потерпит неудачу и вам придется исправлять ошибки в исходном коде, то они могут оказаться полезными.

Таблица 2.7. Цели в файле Makefile, созданном с помощью automake

Цель	Применение
all	Цель по умолчанию; осуществляет сборку всех библиотек и исполняемых файлов
clean	Удаляет объектные файлы и библиотеки, сборка которых осуществлялась целью all
distclean	Удаляет все файлы, созданные во время этапа конфигурирования и сборки. В идеале должны остаться только те файлы, которые имелись в исходном архиве tar. После сборки цели distclean файл Makefile будет отсутствовать, и вам придется вновь использовать configure перед выполнением пересборки
mostlyclean	Аналог clean за тем исключением, что не удаляются избранные библиотеки, включая те из них, что определены разработчиками как редко изменяемые и не нуждающиеся в пересборке
install	Устанавливает программы в каталоги, определенные на этапе конфигурирования. Следует проявлять бдительность, поскольку цель uninstall иногда может отсутствовать
uninstall	Если присутствует, то служит для удаления программ, установленных целью install

Если процесс сборки завершился без ошибок, вы можете переходить к следующему этапу — установке.

2.3.4. Этап установки: make install

Если вы решили установить программу для общего пользования, то конфигурации по умолчанию будет достаточно. Для этого вам потребуются привилегии корневого пользователя root. Относительно безопасным для установки по умолчанию будет каталог `/usr/local/bin`. Он не приведет к конфликту или перезаписи системных программ, которые обычно располагаются в папке `/usr/bin`. Но поскольку каталог `/usr/local` используется многими другими программами, всегда следует сначала проверять инсталляцию, поместив ее в тестовом каталоге, владельцем которого вы являетесь. Для этого нужно перезапустить сценарий `configure` с параметром `-prefix`.

Можно использовать `~/usr` в качестве префикса. В таком случае конфигурирование пакета будет осуществляться следующим образом:

```
$ ./configure -prefix ~/usr
```

Это позволит вам испытать в работе новое программное обеспечение, прежде чем делать его общедоступным для других пользователей в системе. Подобная методика также будет полезна, если вы работаете в многопользовательской системе, в которой не являетесь администратором. Если вы захотите установить программу только для личного пользования, возможно, это все, что вам потребуется сделать.

Данную методику необходимо использовать для тестирования цели `uninstall`, чтобы убедиться, что она корректно выполняет свою роль, перед тем как вы поместите файлы в общий каталог. Цель `uninstall` очень важна. Без нее в вашей системе будет нарастать беспорядок, который со временем может привести к конфликтам с другими программами. В мелких проектах цели `install` и `uninstall` просты, но в крупных они могут быть довольно сложными. Следует избегать установки программ в общих каталогах, если вы не уверены в корректности работы цели `uninstall`.

2.4. Понятие ошибок и предупреждений

В этом разделе я расскажу, как интерпретировать сообщения об ошибках и предупреждения, которые выдают инструменты сборки, а также то, как следует на них реагировать. Вы узнаете о распространенных ошибках, при появлении которых не выводится соответствующих сообщений или предупреждений. Особенно часто они встречаются в файлах Makefile. В результате многие разработчики прилагают огромные усилия, чтобы понять, почему поведение их сборки оказывается не таким, каким должно быть.

2.4.1. Ошибки, распространенные в файлах Makefile

Язык зависимостей и предпосылок, используемый make, не всегда интуитивно понятен. Процесс сборки не всегда последователен и ясен. Правила Makefile могут быть просты, но взаимодействие правил и зависимостей может оказаться чрезвычайно сложным. Несмотря на то что на освоение изложенных выше базовых знаний требуется много времени, даже опытные разработчики полностью не застрахованы от ошибок.

Команды оболочки

При работе с командной частью правил make по умолчанию использует оболочку Борна. Точнее говоря, исполнение команд Makefile осуществляется посредством команды, содержащейся в переменной `SHELL`, по умолчанию располагающейся в `/bin/sh`. Теоретически можно применять любую оболочку по своему усмотрению, но никто так не поступает. Вероятно, причиной этому является то, что все неявные правила пишутся для командной оболочки Борна. Если вы предпочтете другую оболочку, то она может оказаться несовместимой с такими правилами.

Другая причина использования командной оболочки по умолчанию менее очевидна. Когда make исполняет командную часть правила, каждая строка команды порождается как новая оболочка. Иначе говоря, когда вы определяете переменную оболочки в одной строке вашей команды, она «забывается» в следующей строке. Это влечет невозможность написания сценариев, если, конечно, вам не удастся вместить весь сценарий в одну строку. К счастью, однострочные сценарии — это конек командной оболочки Борна[1].

[1] Можно использовать символы продолжения строки, однако в оболочку новые строки передаваться не будут. Подобный прием может быть сложным в применении.

Однако программисты часто сталкиваются с проблемами, когда пытаются втиснуть многое в одну строку. Если вам нужен сценарий, создайте файл сценария и вызывайте его из Makefile. Для написания сценария вы можете использовать наиболее подходящий для вас язык. Нужно лишь удостовериться, что ваш сценарий возвращает соответствующее состояние выхода, то есть make будет знать, если сценарий столкнется с ошибкой.

Отсутствие табуляций

К сожалению, разработчики программы make придают большое значение невидимым символам. Все команды в командной части правила должны начинаться с табуляции. С ее помощью make различает команды, цели и присваивания значений переменным.

Начальную табуляцию нельзя заменять пробелами, так как это будет неправильно. Пробел можно ставить после табуляции, но не перед ней. Поскольку табуляции невидимы, трудно понять, когда они заменяются пробелами, хотя на это способны некоторые текстовые редакторы. А что еще хуже, при этом выводится малопонятное сообщение об ошибке, как видно из следующего примера Makefile:

```
all:
@echo Look Ma! No Tabs!
```

Рассмотрим, что выдает GNU-версия make:

```
$ make
Makefile:2: *** missing separator. Stop.
```

Не самое информативное сообщение, не правда ли?

Проблема заключается в том, что после цели не обязательно должны следовать команды. Программа make должна выяснить, содержит ли следующая после цели строка команды, присваивание значения переменной или другую цель. Для выполнения этой задачи make использует разделитель. Им может быть табуляция, двоеточие, знак равенства и т. д. Инструмент make не пытается угадать, что именно *должно* быть там, но если вы получите приведенное выше сообщение об ошибке, то с 99%-ной вероятностью можно утверждать, что в начале строки команды не была поставлена табуляция.

Так оно и оказалось на самом деле. У нашей команды отсутствовал разделитель, поэтому мы и получили сообщение об ошибке. А вот пример, когда перед командой не стоит табуляция, но присутствует другой разделитель:

```
all:
env FOO="Look Ma! No Tabs!" printenv FOO
```

Как вы думаете, каким будет вывод этого Makefile?

```
$ make
make: Nothing to be done for 'all'.
```

Проблема состоит в том, что make видит разделитель = и приходит к выводу, что это присваивания значение переменной. Если вам интересно, то переменная с присвоенным значением имеет имя `$(env FOO)`. Все правильно: идентификаторы переменных могут содержать пробелы, поэтому и нужны круглые скобки.

Проблемы, возникающие при использовании VPATH

Ранее мы уже вели речь о переменной VPATH. Она очень полезна при работе с большими проектами, но может стать причиной затруднений. Основная проблема возникает, когда в вашем проекте имеются конфликты имен, если, например, у нескольких исходных файлов оказываются одинаковые имена (неудачная ситуация, но бывает и такое). Если это произошло, выходом здесь может стать использование другого модуля.

Вне зависимости от того, на какой каталог указывает VPATH, make всегда сначала проверяет текущий каталог, прежде чем исследовать другие. Когда make находит соответствующий объект в текущем каталоге, поиск в других каталогах не осуществляется. Программа make всегда использует VPATH только во вторую очередь.

Проблемы с VPATH могут возникнуть и у препроцессора C. Естественно предполагать, что препроцессор C автоматически обратится в текущий рабочий каталог в поисках включаемых файлов #include, однако путь поиска будет зависеть от синтаксиса. Допустим, имя файла заключено в кавычки:

```
#include "foo.h"
```

Тогда препроцессор автоматически начнет поиск файлов в каталоге, где располагается исходный файл. В обычной ситуации оба файла находятся в текущем каталоге, но в случае с VPATH исходные модули могут размещаться в другой папке. В данной ситуации препроцессор будет исследовать именно такой каталог, а не текущий. Допустим, имя включаемого файла #include заключено в скобки:

```
#include <foo.h>
```

Тогда препроцессор станет искать в стандартных каталогах include, а также во всех других, которые имеют флаг -I, и больше нигде. При использовании VPATH следует отдавать предпочтение именно такому синтаксису, поскольку он дает возможность оптимизировать поиск исходных файлов.

2.4.2. Ошибки на этапе конфигурирования

При работе над проектами с использованием сборочного GNU-инструментария серьезные ошибки наиболее вероятны на этапе конфигурирования. По крайней мере, так предполагается. Идея заключается в том, чтобы предотвратить потерю времени, если вы пытаетесь осуществить сборку при отсутствии необходимых инструментов и библиотек. По завершении этапа конфигурирования вы сможете быть намного более уверены в том, что процесс сборки закончится без ошибок. Если он потерпит неудачу, то вы точно будете знать, что нужно предпринять, вместо того чтобы сидеть и гадать.

Грамотно написанный сценарий autoconf подскажет вам, что для сборки проекта необходимо использовать инструмент xyz и даже где его можно отыскать. Но зачастую выдаваемое сообщение содержит фразу о его отсутствии: can't find xyz. Если вы решили произвести сборку проекта, вам не останется ничего, кроме как самим отыскать xyz и установить его в своей системе.

В больших проектах с множеством конфигурируемых настроек некоторые инструменты разработки могут использоваться для выполнения одной конкретной

задачи, но не для других. На случай таких ситуаций разработчики создают сценарии configure, которые позволяют точно определить весь необходимый инструментарий. Поскольку при этом требуется участие человека, можете не сомневаться, что ошибки неминуемы. Вы не заметите их, пока тесно не соприкоснетесь с проектом. Если вы не разбираетесь в тонкостях проекта (или сценария autoconf), то лучше воспользоваться сценарием configure, а не заниматься самостоятельно его отладкой.

Бывают ситуации, когда сценарий configure «жалуется» на отсутствие xyz. Такое случается, если xyz действительно отсутствует или оказывается в неожиданном месте. Обычно configure самостоятельно ищет подобные вещи. Если требуемый инструмент имеет правильный путь, то он будет найден. Если же необходимая библиотека или включаемые файлы размещаются в нестандартном каталоге, то об этом следует сообщить сценарию configure. По какой-то причине сделать это можно не с помощью параметров в командной строке, а только посредством переменных окружения. К таким переменным относятся следующие:

- CPPFLAGS — служит индикатором нестандартных путей с использованием дополнительных флагов -I;
- LDFLAGS — служит индикатором нестандартных каталогов с библиотеками с использованием дополнительных флагов -L.

2.4.3. Ошибки на этапе сборки

Сценарий configure отвечает за проверку наличия всех необходимых программных инструментов и библиотек и в зависимости от проекта может генерировать файл Makefile. Преимущество проекта, который использует сценарий configure для создания файла Makefile, заключается в том, что данный файл не подвергается вмешательству человека, то есть теоретически в нем не должно быть синтаксических ошибок. Определить такой проект можно по архиву дистрибутива. Если для генерирования файла Makefile используется сценарий configure, то в архиве этот файл будет отсутствовать, а вместо него там будет файл с именем Makefile.in.

Ошибки на этапе сборки подразделяются на две категории: ошибки в файле Makefile и ошибки, обнаруживаемые во время сборки.

Ошибки в файле Makefile

Несмотря на то что GNU-инструментарий сборки может автоматически генерировать файлы Makefile с малой вероятностью синтаксических ошибок, бывают случаи, когда ошибки все-таки возникают. Разработчики, к примеру, могут использовать ключевое слово include для дословной вставки текста в вывод Makefile. В результате этого открывается возможность появления синтаксических ошибок в выводе Makefile.

Если собираемый вами проект не использует автоматически сгенерированный Makefile, изучите файлы README этого проекта, чтобы понять, что требуется для его сборки. Иногда разработчики не используют цели по умолчанию для сборки проекта или же требуют от пользователя ручного определения переменной в командной строке.

Принято считать, что человек, создающий архив с файлами исходного кода, перед размещением в Интернете убеждается в том, что их сборка проходит успешно. Если вы не можете отыскать логического объяснения синтаксическим ошибкам в файле Makefile, то, скорее всего, это указывает на плохое качество всей оставшейся части кода. В таком случае лучше отложить его в сторону и не тратить время впустую.

Ошибки, обнаруживаемые с помощью make

Даже если ваш файл Makefile не содержит синтаксических ошибок, они возможны в командной части правил. Программа make проверяет состояние возврата каждой исполняемой команды и прерывает исполнение, если обнаруживает ошибку. Это является исключением, поскольку разработчики могут выбирать игнорирование ошибок при исполнении каких-либо команд.

Ошибки, которые могут возникать на этапе сборки, подразделяются на различные категории, приведенные в табл. 2.8.

Таблица 2.8. Категории ошибок этапа сборки

Категория ошибок	Источник
Ошибки компилятора	Ошибки препроцессора, синтаксические ошибки
Ошибки редактора связей	Ненайденные имена
Прочие командные ошибки	Ошибки в разрешениях на запись, ненайденные файлы, синтаксические ошибки оболочки, скрытые синтаксические ошибки

Об ошибках компилятора и редактора связей мы поговорим позднее, а сейчас сосредоточимся на «другой» категории.

Синтаксические ошибки оболочки редко встречаются в файлах Makefile, особенно если разработчики пытаются втиснуть в рамки однострочного сценария максимум возможного. Выявить такие ошибки весьма непросто. Рассмотрим пример дефектного фрагмента сценария внутри правила make:

```
t1 t2 t3:
  @if [ "$@"="t1" ]; then echo $@ is target 1; else echo $@ is not target 1;
fi
```

Здесь в одном правиле объединены три цели без предпосылок: `t1`, `t2` и `t3`. Если ввести следующую команду, то можно увидеть забавную картину:

```
$ make t1 t2 t3
t1 is target 1
t2 is target 1
t3 is target 1
```

Вы можете найти ошибку в этом сценарии? Поскольку мы не на уроке по работе со сценариями, я сам отвечу на этот вопрос. Ошибка заключается в скобках. Они являются псевдонимом оболочки для команды `test`, которая требует наличия пробелов между всеми операторами и значениями. Поскольку в приведенном примере вокруг = пробелы отсутствуют, команда `test` видит не операторов, а лишь одну сплошную строку. В результате она возвращает нулевое значение, которое `if` считает `true`.

Благодаря символу @ в начале сценария при исполнении правила данный сценарий не отобразится в stdout. Это поведение можно изменить двумя способами. Один из них — добавить к make параметр -n, который запускает проверочное исполнение, то есть выводит команды, но не исполняет их. Поскольку выводятся даже команды, скрытые посредством @, приведенный выше файл Makefile будет иметь следующий вывод:

```
$ make -n t1 t2 t3
if [ "t1"="t1" ]; then echo t1 is target 1; else echo t1 is not target 1; fi
if [ "t2"="t1" ]; then echo t2 is target 1; else echo t2 is not target 1; fi
if [ "t3"="t1" ]; then echo t3 is target 1; else echo t3 is not target 1; fi
```

Теперь вы можете наблюдать дефектный сценарий, который в других ситуациях может располагаться в нескольких переменных и который трудно будет отыскать внутри Makefile. Используя параметр -n, вы можете просматривать текст, который дословно передается в оболочку. При возникновении сложных проблем вы также можете перенаправить этот вывод в файл и исполнить его как простой сценарий. Это особенно полезно, когда ваши компиляционные строки сильно разрастаются в длину и вам необходимо определить, что вызывает ошибку компилятора. Вывод можно использовать для создания файла сценария, который разрешается редактировать. Когда проблема будет устранена, вы можете внести аналогичные изменения в файл Makefile.

Еще один способ отладки команд оболочки из Makefile заключается в использовании параметра оболочки -x. Он позволяет выводить команды по мере их исполнения. В командной строке это можно сделать посредством следующей модификации переменной SHELL:

```
$ make SHELL="/bin/sh -x" t1 t2 t3
+ [ t1=t1 ]
+ echo t1 is target 1
t1 is target 1
+ [ t2=t1 ]
+ echo t2 is target 1
t2 is target 1
+ [ t3=t1 ]
+ echo t3 is target 1
t3 is target 1
```

В результате оболочка выводит команды по мере их исполнения. Она показывает только исполняемые части сценария за минусом ключевых слов оболочки. В данном примере особо выделяется команда test в скобках, которая позволяет быстрее получить «ключ к разгадке», чем если бы вы просто изучали исходный сценарий. Каждая ситуация имеет свои особенности, поэтому, если неприменим один способ, используйте другой.

2.4.4. Понятие ошибок компилятора

С годами сообщения об ошибках и предупреждения, выводимые компиляторами, становились все более понятными. В былые времена простая ошибка вроде

отсутствия точки с запятой в коде программы на языке C порождала вывод невразумительного сообщения. Современные компиляторы C/C++ выдают намного более внятные сообщения об ошибках, которые могут автоматически подвергаться разбору.

Для начала рассмотрим заурядный пример дефектной программы:

```
1 void foo()
2 {
3       int x,y,z;
4
5       x=1
6       y=2;
7       z=3;
8 }
```

У значения в строке 5 не хватает точки с запятой. В результате компилятор выведет следующее сообщение об ошибке:

```
$ gcc -c foo.c
foo.c: In function 'foo':
foo.c:6: error: parse error before "y"
```

Формат сообщения предназначен для разбора в интегрированной среде разработки IDE (Integrated Development Environment) или в текстовом редакторе, например Emacs или Vim. Сообщение об ошибке состоит из трех частей, разделенных двоеточиями: файлового имени, номера строки и текстового сообщения. Именно так интегрированная среда разработки IDE, и текстовые редакторы указывают вам на местоположение ошибки в исходном коде, когда они ее обнаруживают. Поскольку выводимые сведения легко читаются, для их разбора не требуются какие-либо инструменты. Как уже отмечалось выше, в нашем примере ошибка присутствовала в строке 5, однако компилятор считает, что ошибка в строке 6. Все из-за того, что он не может определить, что точка с запятой отсутствует в строке, предшествующей строке 6.

Что интересно, аналогичный код, скомпилированный с помощью компилятора C++, выдает более информативное сообщение об ошибке:

```
$ g++ -c foo.c
foo.c: In function 'int foo()':
foo.c:6: error: expected ';' before "y"
```

Код, грамотно написанный на языке C, должен без проблем компилироваться с помощью компилятора C++, что позволяет получать более подробные предупреждения и строго регулировать использование прототипов. Вы можете легко переключиться на дефектный объект и посмотреть, появится ли более информативное сообщение об ошибке. Например:

```
$ make CC=g++ foo.o
```

При работе над проектами с открытым исходным кодом компилятор C++ обычно не используется для компиляции кода, написанного на языке C. Одна из причин этого — стандарт C++ не гарантирует совместимости кода на языке C++ с ком-

пилятором C. Это исключает применение C++ для компиляции программных библиотек, используемых клиентами C. Кроме того, косвенно предполагается, что для компиляции кода, написанного на C, будет использоваться именно компилятор C.

Ошибки, встречающиеся в исходном коде, имеют тенденцию к каскадированию, то есть присутствие одной ошибки может привести к тому, что компилятор станет рапортовать о наличии ошибок в последующих строках кода, которые в иной ситуации были бы правильными. При поиске ошибок рекомендуется начинать с первой ошибки в модуле, о которой было выведено сообщение, и двигаться далее. Зачастую оказывается так, что первая ошибка становится единственной, которую приходится исправить.

2.4.5. Понятие предупреждений компилятора

Иногда выводимые предупреждения помогают улучшить программный код. Но бывают ситуации, когда такие предупреждения только мешают. Раньше программистам приходилось использовать программу lint, чтобы иметь возможность просматривать предупреждения. Она проверяла исходный код и выдавала список предупреждений, который можно было читать как телефонную книгу. Факт использования программы lint означал, что разработчик был вынужден следовать информации, содержащейся в выводимых предупреждениях. В настоящее время большинство подобных предупреждений генерируется компилятором, поэтому программисты склонны либо принимать их во внимание, либо рассматривать как помеху. Возможно, именно поэтому в GNU-компиляторе вывод предупреждений деактивирован по умолчанию.

Причина отключения вывода предупреждений стара, как программа lint. У разработчиков вызывало недовольство то, что на одно предупреждение о действительно серьезной проблеме приходилось десять или более малозначительных сообщений. К последним относились неиспользуемые переменные, сравнение беззнаковых целых чисел со знаковыми, а также форматы `printf`. Многие предупреждения поднимали вопросы, имеющие отношение к переносимости, в результате чего некоторые программисты полагали, что они их не касаются.

Даже если вы не планируете поддержку чего-либо, кроме Linux, на архитектуре IA32, в определенный момент вы можете захотеть модернизировать свой компилятор. Аналогичные проблемы с переносимостью, возникающие при смене процессора или операционной системы, также могут появиться при переходе с одной версии компилятора на другую. Реагировать на предупреждения не всегда сложно, при этом вы получаете возможность улучшить свой программный код.

Включить вывод предупреждений в GNU-компиляторе C/C++ можно с помощью параметра `-Wall`, который активирует вывод особой совокупности предупреждений, отобранных участниками проекта GNU.

Обычно в проектах, для сборки которых используется GNU-инструментарий, вывод предупреждений отключен. Возможно, причина заключается в том, что параметры, используемые для активации предупреждений, не всегда одинаковы для всех компиляторов. Это очень неудобно, но вы можете включить вывод предупреждений

в проекте, собранном с помощью GNU-инструментов, используя следующую команду `configure`:

```
$ CFLAGS=-Wall ./configure
```

Предупреждения: неявные объявления

При использовании параметра `-Wall` компилятор выводит предупреждение всякий раз, когда сталкивается с неявно определяемыми переменными или функциями в коде на языке C (в случае с C++ это будет обычное сообщение об ошибке). Большинство опытных программистов предпочитают при компиляции отключать вывод таких предупреждений. Если вы увидели на экране нечто подобное, то это может говорить о том, что вы имеете дело с древним унаследованным кодом, который длительное время не подвергался вмешательству, либо код написан неопытным программистом.

Неявно определяемая переменная — это такая переменная, которая располагается в левой части выражения присваивания до появления в объявлении. Подобная переменная неявно определяется как `int`. Традиционный язык C позволяет так поступать в отношении глобальных переменных, а компилятор `gcc` довольно либерален в этом плане. Например:

```
$ echo "x=1;" > foo.c
$ gcc -c -Wall foo.c
foo.c:1: warning: type defaults to 'int' in declaration of 'x'
foo.c:1: warning: data definition has no type or storage class
```

Компиляция возможна даже при использовании параметра `-ansi`. Единственный способ форсировать появление ошибки в неявном объявлении переменной — использовать параметр `-pedantic`, но это может привести к тому, что компилятор станет «ругаться» на остальную часть кода. Применять данный параметр нужно осторожно.

Неявно определяемая функция — это функция, которая используется без предшествующего объявления функции или прототипа. Когда функция определяется неявно, компилятор считает, что она возвращает `int`, и не делает никаких предположений относительно количества или типа аргументов, используемых этой функцией. Вот эквивалент последующего объявления функции:

```
int foo(); // Объявление функции — никаких упоминаний об аргументах
```

Обратите внимание, что объявление функции отличается от прототипа функции. Объявление является наследованием, предшествующим стандарту ANSI. Оно сообщает компилятору возвращаемый тип, однако ничего не говорит ему насчет аргументов. Если компилятор видит функцию без прототипа, он допускает любое количество аргументов любого типа. Это позволяет унаследованному коду на языке C проходить компиляцию, но отсутствует в современном коде. Прототип определяет точное количество и тип аргументов для функции в дополнение к возвращаемому значению.

Проблема заключается в том, что при использовании `-Wall` компилятору C может удовлетворять как прототип функции, так и объявление функции. Если вы желаете получать предупреждения, когда функция используется без прототипа, применяйте параметр `-Wstrict-prototypes`. Если компилятор наткнется на объявление функции, `gcc` (3.4.4) выдаст следующее сообщение:

```
foo.c:2: warning: function declaration isn't a prototype
```

Теперь, когда вы все знаете об объявлениях и прототипах функции, вы без труда сможете понять, что будет означать такое предупреждение, если оно появится.

Предупреждения, касающиеся формата

Флаг `-Wall`, помимо прочего, позволяет запускать проверку формата. GNU-компилятор проверяет строки формата функций семейства `printf`, для чего ищет правильное количество аргументов и удостоверяется в том, что тип аргументов соответствует формату. Проверка формата поддерживается для всех функций, приведенных в табл. 2.9.

Таблица 2.9. Функции, поддерживающие проверку формата

Варианты printf		**Варианты scanf**	
printf	vprintf	scanf	vscanf
fprintf	vfprintf	fscanf	vfscanf
sprintf	vsprintf	sscanf	vsscanf
snprintf	vsnprintf		

Проверка формата позволяет выявлять ошибки, которые обычным способом весьма непросто отыскать. Формат `%s` в операторе `printf`, например, может привести к нарушению сегментации, если вы передадите ему неверный указатель. Компилятор уведомит вас, если вы, к примеру, нечаянно использовали формат `%s` с аргументом, не являющимся `char *`. Подобные ошибки весьма легко допустить, особенно если вы имеете дело со строками формата с большим количеством аргументов. Ниже приведен простой пример, иллюстрирующий предупреждение:

```
1 #include <stdio.h>
2 void foo(int x)
3 {
4       printf("%s", x);
5 }
```

Если воспользоваться флагом `-Wall`, то компилятор выдаст следующее предупреждение:

```
$ gcc -c -Wall foo.c
foo.c: In function 'foo':
foo.c:4: warning: format argument is not a pointer (arg 2)
```

В приведенном примере формату требуется указатель, однако аргумент (в нашем случае это второй аргумент `printf`) не является указателем. Формату `%s` необходим, в частности, указатель `char *`, и если вы передадите не тот тип указателя, например `int *`, то также получите предупреждение. Такие предупреждения являются ожидаемыми дампами ядра.

Похожие предупреждения выводятся и при других несоответствиях формата, но далеко не все подобные ошибки могут привести к краху вашего кода. К примеру, архитектура IA32 редко терпит крах из-за передачи целочисленному формату указателя неверного типа. IA32 очень либерально подходит к выравниванию данных;

самое худшее, что может случиться, — вы увидите искаженный вывод. Другие процессорные архитектуры не столь снисходительны, и несоответствие типов в форматах `printf` может привести к краху вашего кода.

В предупреждениях, касающихся формата в программном коде, нет ничего необычного, особенно если вы впервые активировали их вывод. Программисты под архитектуру IA32 могут спокойно игнорировать их, если считают не относящимися к делу. В частности, многие имеют привычку использовать для подстановки типы `long int` и `int` как эквивалентные. Компилятор выведет предупреждение, если вы передадите целое число `long` формату `%d` (правильной будет передача целого числа `long` формату `%ld`). В системах на основе архитектуры IA32 это не будет проблемой, поскольку `long` имеет тот же размер, что и `int`. Но в 64-битной системе их размеры отличаются. На платформе x86_64 (Opteron, Xeon/EM64T) `int` остается 32-битным, а `long` будет 64-битным. Это особенно важно знать, поскольку 64-битные системы становятся все популярнее.

Несмотря на то что использование функции `printf` относительно безопасно, со `scanf` следует обходиться намного внимательнее. Независимо от используемой архитектуры, неверный аргумент `scanf` может привести к краху программы. Поскольку аргументы являются указателями в память программы, несоответствие типов может повлечь переполнение буфера и искажение памяти. Ко всем предупреждениям, касающимся форматов `scanf`, нужно подходить очень ответственно.

Проверка формата, запускаемая с помощью `-Wall`, может дополняться еще двумя параметрами.

- `-Wformat-nonliteral` — выдает предупреждение, когда компилятор сталкивается со строкой формата, которая является не литералом, а переменной. Это будет означать, что проверка формата невозможна. Более того, поскольку формат может изменяться во время исполнения, это открывает путь для возникновения ошибок, которые могут привести к краху вашего кода.
- `-Wformat-y2k` — относится к функциям `strftime` и `strptime`, которые используют синтаксис формата, схожий с синтаксисом `printf`. Предупреждение будет выводиться каждый раз, когда будет обнаруживаться формат года из двух цифр.

Прочие предупреждения, выводимые GNU-компилятором C

Помимо предупреждений, касающихся форматов, GNU-компиляторы могут выводить множество других предупреждений. Ниже приведены те из них, которые могут выводиться на индивидуальной основе, если активированы соответствующие параметры.

- Null pointer arguments («Пустые» аргументы указателя со значением NULL) — необходимо проверить аргументы указателя выбранных функций на предмет значений `NULL` во время компиляции. Это не гарантирует, что аргументы `NULL` не будут передаваться во время исполнения. Как и в случае с библиотекой glibc 3.4.4, подобная проверка в стандартной библиотеке C/C++ почти не поддерживается.
- Missing parentheses (Отсутствуют круглые скобки) — данное предупреждение появляется, если присваивание осуществляется в логическом контексте, к при-

меру, с использованием оператора `if`. Зачастую это бывает непреднамеренно. Например:

```
if ( x = y ) //versus...
if ( x == y )
```

Строка 1 одновременно является присваиванием и логической проверкой, однако в ней также может быть типографская ошибка. Возможно, имелась в виду строка 2. Это очень распространенная форма типографской ошибки, поэтому компилятор и обращает ваше внимание на нее. Если применить дополнительные круглые скобки, все становится на свои места:

```
if ( (x = y) )
```

- Missing braces (Отсутствуют фигурные скобки) — данное предупреждение выводится, если операторы `if/else` не используют фигурных скобок. Это может стать причиной ошибок:

```
1 void foo(int x, int y, int *z)
2 {
3       if ( x == y )
4               if ( y == *z ) *z = 0.0;
5       else
6               *z = 1.0;
7 }
```

Использованный в этом примере отступ говорит о том, что оператор `else` в строке 5 относится к оператору `if` в строке 3, однако из-за отсутствия фигурных скобок получается, что этот оператор `else` относится к оператору `if` в строке 4. Предупреждение указывает на необходимость использования фигурных скобок в случае возникновения подобной ошибки:

```
$gcc -c -Wall foo.c
paren.c: In function 'foo':
paren.c:3: warning: suggest explicit braces to avoid ambiguous 'else'
```

Чтобы устранить ошибку, нужно добавить фигурные скобки к каждому предложению `if` и `else`:

```
1 if (x == y) {
2     if (y == *z) {
3         *z = 0.0;
4     }
5 }
6 else
7 {
8     *z = 1.0;
9 }
```

- Uninitialized variables (Неинициализированные переменные) — данное предупреждение появляется, когда переменные используются еще до их инициализации. Тревога также может оказаться ложной, но реальные проблемы можно

без труда обнаружить по указке компилятора. Подобные предупреждения выводятся только в том случае, если компиляция осуществляется вместе с оптимизацией. Например:

```
1 int foo(int y)
2 {
3       int x;
4       if ( y > 0 ) {
5               x = 1;
6       }
7       return x;
8 }
```

В данном примере переменная `x` не инициализирована, так как `y <= 0`. В результате функция возвращает неопределенное значение. При компиляции без оптимизации предупреждение выдаваться не будет, но если включить оптимизацию (с помощью `-O1`, `-O2` и т. д.), вы увидите такое предупреждение:

```
$ gcc -Wall -c -O2 uninit.c
uninit.c: In function 'foo':
uninit.c:3: warning: 'x' might be used uninitialized in this function
```

Без оптимизации код получается дефектным, но компилятор не имеет доступа к такой информации, когда оптимизация не выполняется. Источником необходимых ему сведений служит анализ потока кода, который является частью процесса оптимизации.

Перечень предупреждений, выводимых компилятором gcc, весьма обширен. Вывод большинства из них может быть активирован или деактивирован на индивидуальной основе. Более подробно о них вы можете узнать из справочного руководства к gcc.

Предупреждения GNU-компилятора, характерные для C++

По сравнению с языком C общая надежность C++ была повышена, при этом многие предупреждения, выводимые компилятором C (например, об отсутствии прототипов), в C++ считаются ошибками. Несмотря на это, существует много новых и объектно-ориентированных способов улучшить свой программный код. GNU-компилятор может выводить некоторые предупреждения, характерные для C++. Их вывод можно активировать посредством `-Wall`. Рассмотрим наиболее примечательные предупреждения.

- Nonvirtual destructors (Невиртуальные деструкторы) — данное предупреждение выводится, если у полиморфного класса имеется невиртуальный деструктор. Это может стать источником ошибок исполнения из-за утечек памяти и невысвобождения ресурсов в результате вызова неверного деструктора. Например:

```
1 class Base {
2       char *m_ptr;
3 public:
4       Base() : m_ptr(0) { m_ptr = new char[1024]; };
```

```
 5      virtual char *ptr() { return m_ptr; };
 6      ~Base() { delete [] m_ptr; } ;
 7 };
 8
 9 class Foo : public Base {
10      char *foo_ptr;
11 public:
12      Foo() { foo_ptr = new char[1024]; };
13      virtual char *ptr() { return foo_ptr; };
14      virtual ~Foo() { delete [] foo_ptr; } ;
15 };
```

```
$ gcc -Wall -c polymorph.cpp
polymorph.cpp:1: warning: 'class Base' has virtual functions but nonvirtual
destructor
```

Деструктор в классе `Base` должен быть объявлен как виртуальный, в противном случае он не будет вызываться, если мы удалим указатель типа `Foo *`. Обратите внимание, что имеющийся виртуальный деструктор класса `~Foo` не исправляет ситуации. Класс `Base` по-прежнему остается дефектным.

- Reordered initializers (Переупорядоченные инициализаторы) — данное предупреждение появляется, если вы определяете свои инициализаторы в порядке, отличном от того, в котором они объявлялись. Компилятор должен производить инициализацию членов в том же порядке, в котором они объявляются. Порядок, в котором инициализаторы располагаются в коде, не относится к делу и может ввести в заблуждение. Рассмотрим пример того, как инициализаторы могут вынудить вас сделать ошибку:

```
 1 struct Foo {
 2      int m_two;              // declaration
 3      int m_one;
 4      Foo( int one )
 5      : m_one(one),           // initializer
 6        m_two(m_one+1) {};
 7 };
 8
 9 #include <iostream>
10
11 int main(int argc, char **argv[])
12 {
13      Foo f(1);
14      std::cout << f.m_one << std::endl;
15      std::cout << f.m_two << std::endl;
16 }
```

```
$ gcc -c -Wall order.cpp
order.cpp: In constructor 'Foo::Foo(int)':
order.cpp:3: warning: 'Foo::m_one' will be initialized after
order.cpp:2: warning: 'int Foo::m_two'
order.cpp:6: warning: when initialized here
```

В приведенном выше примере инициализаторы располагаются в строках 5 и 6. Если брать в расчет только эти два инициализатора, то все выглядит как будто правильным. Но если присмотреться внимательнее, то можно заметить, что член `m_two` определяется первым и, следовательно, инициализируется тоже первым. В этом и заключается проблема, поскольку `m_two` инициализируется с использованием значения `m_one`, которое оказывается неинициализированным, когда инициализируется `m_two`. Для решения этой проблемы нужно переупорядочить объявления таким образом, чтобы `m_two` объявлялся после `m_one`.

- Deprecated features (Устаревшие средства) — подобное предупреждение выводится относительно модулей, в которых используются устаревшие средства вроде старомодных заголовков стандартной библиотеки шаблонов STL (Standard Template Library) (например, `iostream.h` вместо `<iostream>`). Предупреждение будет выглядеть следующим образом:

```
warning: #warning This file includes at least one deprecated or antiquated header.
Please consider using one of the 32 headers found in section 17.4.1.2 of the C++
standard. Examples include substituting the <X> header for the <X.h> header for
C++ includes, or <iostream> instead of the deprecated header <iostream.h>. To
disable this warning use -Wnodeprecated[1].
```

Данное предупреждение будет всплывать даже в том случае, если вы осуществляете компиляцию, не активировав вывод предупреждений. Если последовать предложенному совету, то нужно внести изменения, чтобы операторы `#include` использовали заголовки нового стиля. Однако не стоит торопиться. Вместе с новыми заголовками приходят более строгие требования пространства имен. В частности, любые ссылки на символы в пространстве имен `std` не будут компилироваться до тех пор, пока вы не добавите соответствующий классификатор пространства имен или оператор `using`. С подобным можно столкнуться лишь в старом коде, который предшествует стандарту C++ либо портирован из другой системы с использованием более либерального компилятора C++. В таком случае у вас есть три выхода.

- Самый простой — игнорировать предупреждение и/или деактивировать его вывод с помощью параметра `-Wno-deprecated`.
- Простой — исправить неверные заголовки и добавить оператор с использованием пространства имен в тело модуля *(но не в файлы заголовка)*[2].
- Более сложный — исправить неверные заголовки и использовать соответствующие классификаторы пространства имен.

---

[1] Предупреждение: данный файл имеет минимум один устаревший заголовок. Необходимо использовать один из 32 заголовков раздела 17.4.1.2 стандарта C++. Примеры включают подстановку заголовка <X> вместо заголовка <X.h>, а также заголовка <iostream> вместо устаревшего заголовка <iostream.h>. Чтобы отключить вывод данного предупреждения, используйте параметр –Wnodeprecated.

[2] Никогда не внедряйте using namespace std в файл заголовка. Это «загрязняет» пространство имен модуля, который его использует, и может привести к сложным в выявлении ошибкам.

- Incompatible ABI (Несовместимый ABI) — вывод данного предупреждения активируется не с помощью `-Wall`, а посредством флага `-Wabi`. ABI означает *двоичный программный интерфейс (Application Binary Interface)*, который используется для вызова функций и позволяет уживаться разным языкам программирования в одном приложении. Вы можете писать программу на C++, которая осуществляет вызов библиотеки, созданной на языке Fortran, и если даже при этом у вас нет компилятора Fortran, ваша программа все равно будет работоспособна. Это достигается благодаря тому, что компилятор Fortran подчиняется определенному ABI. Интерфейс ABI указывает компилятору, как следует вызывать функции и передавать аргументы.

ABI уникален для каждой комбинации процессора и операционной системы, так как каждая из них имеет свои уникальные требования и возможности. Поскольку большинство процедурных языков не имеют проблем с подчинением общераспространенному ABI, обычно ABI-интерфейсы нейтральны к используемым языкам. Однако языку C++ в силу его достаточной сложности требуются уникальные расширения ABI. Отсутствие единого стандарта приводит к тому, что каждый компилятор создает произвольные расширения для ABI-интерфейса языка C, чтобы обеспечить поддержку C++. В результате может оказаться так, что вы не сможете осуществить связывание программного кода, созданного с помощью GNU-компилятора C++, с библиотекой, скомпилированной коммерческим разработчиком, поскольку ABI-интерфейсы могут быть несовместимы. Связать код на C++, скомпилированный GNU-компилятором версии 3.*x*, с библиотекой, созданной компилятором версии 2.*x*, нельзя, так как у этих версий разные ABI.

2.4.6. Понятие ошибок редактора связей

При работе с командной строкой редактора связей наиболее вероятны такие ошибки, как ненайденные имена. Отсутствующим именем может быть функция или глобальная переменная, на которую объектный файл ссылается без объявления. Это может случиться, например, когда вы пытаетесь осуществить сборку проекта, используя библиотеку несовместимой версии. Отсутствующее имя рапортуется как неопределенная ссылка и выглядит следующим образом:

```
$ g++ -Wall -o main main.o -lfoo
main.o(.text+0x11): In function 'main':
: undefined reference to 'bar()'
```

Если неопределенная ссылка возникает в программе, написанной на языке C, это означает, что в вашей строке связывания отсутствует библиотека или объектный файл либо у вас имеется библиотека несовместимой версии, которая не определяет конкретное имя.

Неопределенные ссылки могут также возникать в программах и библиотеках, созданных на языке C++, из-за разницы в механизме изменения имен между вашим компилятором и тем, с помощью которого создавались библиотеки. *Механизм изменения имен* — это методика, используемая компиляторами C++ для переименования функций на основе их подписей. Поскольку редактор связей ничего не

знает о C++, компилятору приходится добавлять данные к именам функций, чтобы наделить их уникальными подписями, понятными редактору связей. Обычно компилятор добавляет лишние символы в имена функций для указания количества и типа аргументов.

Когда такое случается, обычно сообщается, что отсутствует какое-либо имя, как это было в предыдущем примере, и это может поставить в тупик. Вы можете и не догадываться, что все дело в несовместимости механизмов изменения имен. Здесь на помощь приходит команда `nm`. Как на самом деле выглядит измененное имя `bar()` из приведенного выше примера, можно увидеть, воспользовавшись следующей командой:

```
$ nm -u main.o | grep bar
         U _Z3barv
```

Строка `_Z3barv` показывает, как компилятор g++ 3.3.5 изменяет подпись функции `void bar(void)`. Если скомпилировать `libfoo` с помощью другого, несовместимого компилятора, то подпись может быть совершенной иной. Это следствие того, что в C++ не существует стандартного механизма изменения имен. Если вы установите двоичный пакет, собранный с помощью несовместимого компилятора, то можете столкнуться с подобными ошибками. К сожалению, единственный способ избежать этого заключается в использовании для компиляции кода и библиотек одного и того же компилятора.

Нужно также отметить, что редактор связей очень разборчив в отношении того, в каком порядке библиотеки определяются в его командной строке. Чтобы наглядно продемонстрировать это, создадим две библиотеки и один основной модуль:

```
main.c
1 extern void two();
2 int main(int argc, char *argv[]) { two(); }

one.c
1 void one(void) { }

two.c
1 extern void one(void);
2 void two(void) { one(); }
```

Теперь создадим две библиотеки и попытаемся осуществить связывание:

```
$ ar -clq libone.a one.o
$ ar -clq libtwo.a two.o
$ gcc -o main main.c -L. -lone -ltwo
./libtwo.a(two.o)(.text+0x7): In function 'two':
/home/john/src/linker/two.c:5: undefined reference to 'one'
collect2: ld returned 1 exit status
```

В данном примере функция `two()` вызывает функцию `one()`, при этом они располагаются в `libtwo.a` и `libone.a` соответственно. Редактор связей сначала видит `libone.a`, поскольку это первый параметр `-l` в командной строке. С этого момента он начинает искать ссылку на `two()` из `main()`. Поскольку ссылки на `one()` отсутствуют, редактор связей решает отбросить модуль, содержащий `one()`. Затем он обрабаты-

вает `libtwo.a` и находит ссылку на `two()` из `main()`. Однако теперь он не находит ссылку на `one()`. Поскольку он уже отбросил этот модуль, процесс связывания завершается неудачей из-за неопределенной ссылки на `one()`.

Решить эту проблему весьма просто. Нужно лишь изменить порядок библиотек в командной строке следующим образом:

```
$ gcc -o main main.c -L. -ltwo -lone
```

Теперь редактор связей удовлетворен. Зависимости определяются слева направо, то есть библиотекам, находящимся слева, необходимы библиотеки, расположенные справа.

2.5. Заключение

В этой главе мы рассмотрели инструментарий для сборки приложений с открытым исходным кодом. Программа make — наиболее популярный инструмент, который применяется для работы над проектами с открытым исходным кодом на языке C и C++. Именно по этой причине внушительная часть настоящей главы была посвящена рассказу об этом инструменте и особенностях его использования.

В этой главе использовались следующие инструменты:

- `configure` — сценарий, который входит в состав всех GNU-проектов и служит для генерирования файлов Makefile, используемых инструментом make;
- cons — инструмент сборки на основе языка сценариев Perl, который предполагался как замена make;
- make — наиболее распространенный UNIX-инструмент сборки кода на C и C++;
- scons — попытка повторной реализации cons на языке сценариев Python, но лишенная недостатков cons.

Веб-ссылки этой главы:

- www.gnu.org/software/autoconf — проект GNU autoconf, используемый для распространения исходного кода GNU;
- www.gnu.org/software/make — домашняя страница GNU проекта make;
- www.python.org — домашняя страница языка сценариев Python;
- www.scons.org — домашняя страница инструмента scons.

3 Поиск справочной документации

3.1. Введение

Есть такая шутка: «Программисты не должны документировать свой код. Его должно быть так же сложно использовать, как и писать». Преднамеренно или нет, но открытый исходный код зачастую подтверждает это.

Несмотря на то что многие инструменты снабжаются удобной документацией, существует большой перечень инструментария, у которого она отсутствует. Иногда вам нужно лишь знать, где ее искать.

3.2. Интерактивные руководства

Важным нововведением в UNIX было использование интерактивного руководства[1]. Оно стало отличной альтернативой стопкам печатных справочных руководств, для хранения которых повсеместно приходилось отводить целые полки. В то время, равно как и сейчас, программа man использовалась в качестве основного интерфейса интерактивного руководства. Участники проекта GNU изобрели более гибкий интерактивный формат Texinfo, который генерирует документацию, используемую программой info. Разработчики некоторых инструментов пренебрегают страницами руководства man и инфостраницами info, вместо этого предлагая в дистрибутивах справочные сведения в обычном текстовом или ином формате, которые необходимо искать самостоятельно.

Несмотря на то что man и info — оптимальные форматы для инструментов командной строки, многие графические пользовательские интерфейсы используют собственные справочные меню в качестве единственного источника интерактивной документации.

---

[1] Под термином «интерактивный» в данном случае следует понимать документацию в электронном виде в противопоставлении со справочными материалами на бумаге, а не в современном смысле, связанном с Интернетом.

3.2.1. Страницы руководства man

Идея просмотра документации в текстовых окнах может показаться устаревшей в дни существования форматов HTML и PDF, однако чтение справочных сведений в окне терминала имеет свои преимущества. Это особенно справедливо, если вы, как и многие Linux-разработчики, проводите в нем большинство своего рабочего времени. Получить информацию со страниц руководства man намного проще, чем использовать браузер или программу просмотра PDF-файлов. Если вам нужно быстро получить ответы на интересующие вопросы, то страницы руководства man — это то, что вам нужно.

Страницы руководства man создаются с помощью старого языка разметки под названием troff. Несмотря на свой почтенный возраст, он по-прежнему весьма мощный. Страницы, написанные как на troff, как и на HTML, форматируются под устройство, на котором они отображаются. В дополнение к просмотру в текстовых окнах они также могут быть отформатированы для распечатки или преобразованы в HTML или PDF. Так, например, чтобы отформатировать страницу man для распечатки в формате PostScript, нужно воспользоваться параметром `-t`:

```
$ man -t man | lp -Pps
```

Данный вывод можно передать любому инструменту, который поддерживает PostScript, и манипулировать им по своему усмотрению.

В дистрибутивах используется два типа man. В дистрибутивах Red Hat и прочих, основанных на пакетах RPM, применяется традиционная программа man, тогда как в Debian-дистрибутивах используется man-db. Основная разница между ними заключается в базе данных, используемой для индексации и каталогизирования страниц руководства man. Применение man-db имеет множество преимуществ по сравнению с традиционной базой данных страниц man, однако функционируют эти комплекты инструментов во многом одинаково.

3.2.2. Устройство руководства man

Справочная система Linux состоит из разделов. Она подчиняется стандарту файловой иерархии Filesystem Hierarchy Standard (www.pathname.com/fhs), который, помимо всего прочего, определяет содержимое каждого из разделов man. Общий вид системы разделов приведен в табл. 3.1.

Таблица 3.1. Разделы справочного руководства Linux

Раздел	Описание
1	Пользовательские команды оболочки
2	Системные вызовы программ посредством библиотечных функций
3	Библиотечные функции программ
4	Устройства в каталоге /dev
5	Разнообразные системные файлы (например, /etc)
6	Игры
7	Различная информация
8	Команды администрирования

Разделение справочного руководства на разделы позволяет страницам man избегать конфликтов имен. Команда sync, например, документирована в разделе 1, а функция sync — в разделе 2. Обе эти страницы man называются sync. Если бы они не были документированы в разных разделах, для их разграничения пришлось бы использовать необычную систему присваивания имен. Иногда для того, чтобы отыскать необходимую страницу man, требуются некоторые усилия. Если вы захотите просмотреть страницу man команды sync, то в большинстве случаев вам нужно будет ввести следующее:

```
$ man sync
```

Если вы пожелаете увидеть страницу man *функции* sync, то знайте, что sync — это системный вызов, документированный в разделе 2. Чтобы просмотреть страницу руководства man этой функции, необходимо указать перед именем страницы номер раздела:

```
$ man 2 sync
```

Еще одна примечательная особенность заключается в том, что некоторые дистрибутивы могут содержать свои собственные дополнительные справочные разделы. Первым примером здесь может быть уже упоминавшийся выше Perl. Второй пример — раздел 3p в дистрибутиве Fedora. В нем содержатся справочные сведения о функциях POSIX (отсюда и буква «p» в названии раздела). В данном разделе также имеется страница man функции sync, фактический аналог которой присутствует в разделе 2. Если вы не уверены и просто хотите просмотреть все страницы руководства man, которые доступны в системе, введите такую команду:

```
$ man -a sync
```

Она позволяет выводить страницы man по порядку, и каждый раз при нажатии клавиши Q для выхода со страницы man перед вами будет появляться следующая соответствующая страница.

Для ссылки на страницу, содержащуюся в определенном разделе, необходимо указать номер этого раздела в круглых скобках после ее имени. Например, функция sync в разделе 2 будет обозначаться как sync(2), а функция sync в разделе 3p — sync(3p). Подобная система обозначений используется на всех страницах руководства man, и я буду следовать ей по ходу этой книги.

Вы можете предположить, что, если не указывать раздел, поиск будет осуществляться последовательно в каждом разделе. Однако в большинстве дистрибутивов поиск начинается не с функций, а с команд. Это означает, что man сначала выполняет поиск в разделе 8 (System Administration Commands (Команды системного администрирования)), а потом в разделе 3 (Programming Libraries (Программные библиотеки)). Такой подход имеет смысл, если вы являетесь системным администратором (а если нет?). Но если вы программист, который пишет сокетную программу и хочет просмотреть страницу man функции accept, расположенную в разделе 3, то при вводе man accept вы получите страницу man команды accept из пакета Common UNIX Printer System, содержащуюся в разделе 8. Если вы похожи на меня, то, скорее всего, не сможете запомнить, какой раздел к чему относится. Именно в такой ситуации на помощь нам приходит команда whatis:

```
$ whatis accept
accept (8)          - accept/reject jobs sent to a destination
accept (2)          - accept a connection on a socket
```

Она выводит страницы руководства man в том порядке, в котором они будут обнаружены. В данном случае мы видим, что поиск сначала осуществлялся в разделе 8, а потом в разделе 2. Видно также, что наша искомая страница man располагается в разделе 2.

Порядок поиска страниц руководства man определяется в файлах конфигурации системы, которые варьируются в зависимости от инсталляции. В дистрибутивах Fedora и Ubuntu (на основе Debian) таким файлом является `/etc/man.config`, а в дистрибутиве Knoppix (также на базе Debian) им будет файл `/etc/manpath.config`. С помощью программы mandb вы можете самостоятельно переопределять параметры в `~/.manpath`. Однако в традиционных дистрибутивах менять настройки по умолчанию можно только посредством параметров командной строки или переменных окружения (рис. 3.1).

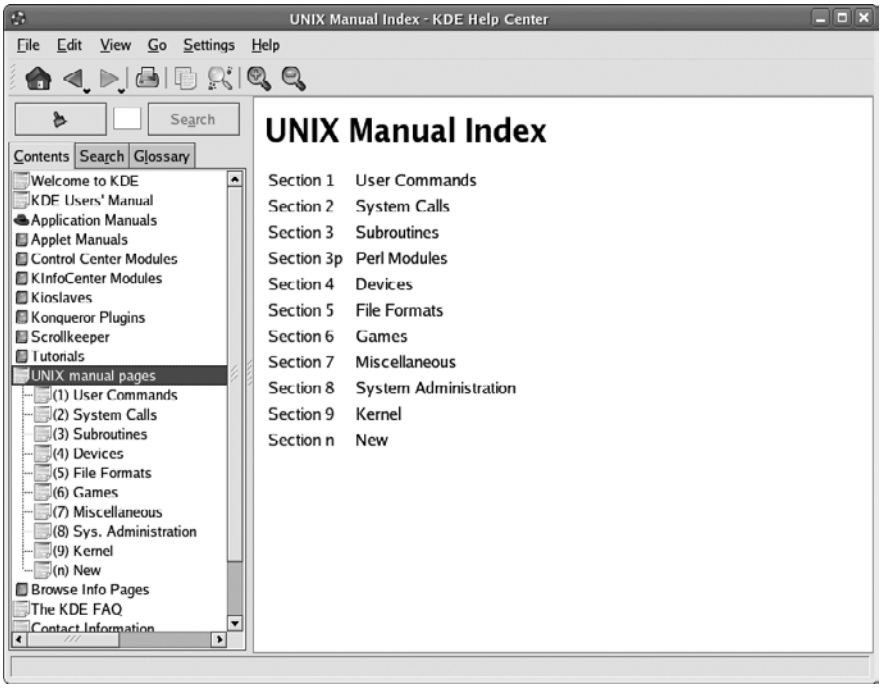


Рис. 3.1. Использование khelpcenter для просмотра разделов man

3.2.3. Поиск страниц руководства man: команда apropos

Если справочные руководства недоступны для просмотра, то нужно организовать в них поиск. Для этого используются два инструмента: `apropos` и `whatis`.

apropos — довольно необычное имя для UNIX-команды не только потому, что в нем более трех букв, но также в силу того, что это слово большинство англоговорящих людей не встречало со времен школьных уроков по лексике. Слово apropos означает «соответствующий», в чем и заключается смысл этой команды. Вы задаете ключевое слово, в результате чего получаете соответствующие результаты (в идеале).

К сожалению, команда apropos позволяет выполнять поиск только в разделе NAME элементов руководства, а в нем содержатся лишь короткие однострочные описания необходимого объекта. Если соответствия вашему ключевому слову отсутствуют в разделе NAME, от команды apropos толку не будет. За один подход осуществляется поиск только по одному ключевому слову. Если вы укажете два ключевых слова, то получите соответствия первому из этих слов, после чего высветятся те, которые соответствуют уже второму слову. По умолчанию при сопоставлении охватывается весь текст, при этом слово man соответствует словам manual («справочное руководство»), command («команда») и т. д. Это не механизм поиска. Любые получаемые на выходе результаты будут соответствовать ключевому слову.

Стоит также предусматривать, чтобы ваше ключевое слово не оказалось слишком ограничивающим. Например, при поиске по слову compression («сжатие») полученный результат должен касаться сведений об утилитах для сжатия файлов, но если при поиске ввести apropos compression, в конечных результатах будут отсутствовать два наиболее популярных архиватора: bzip2 и gzip. Вот результаты, полученные при поиске в системе Fedora Core 4:

```
$ apropos compression
Compress::Zlib       (3pm) - Interface to zlib ...
pbmtopsg3            (1) - convert PBM ...
SSL_COMP_add_compression_method (3ssl) - handle ...
zlib                 (3) - compression/decompression ...
```

Проблема заключается в том, что термин compression («сжатие») не соответствует словам compress («сжимать») и compressor («архиватор»), которые, как оказалось, являются описательными для gzip и bzip2 соответственно. Поэтому поиск по нашему ключевому слову и не выдал таких результатов. Воспользовавшись командой whatis, можно просмотреть информацию об этих архиваторах:

```
$ whatis gzip bzip2
gzip (1)             - compress or expand files
bzip2 (1)            - a block-sorting file compressor, v1.0.2
```

Традиционно команда apropos в дистрибутивах Red Hat используется без параметров. В отличие от этого, mandb-вариант apropos в дистрибутивах Debian может применяться с несколькими параметрами, включая такие, которые ограничивают диапазон результатов точными соответствиями или целыми словами. Оба варианта позволяют использовать регулярные выражения[1]. Например:

```
$ apropos 'mag[tn]'
mt (1)               - control magnetic tape drive operation
mt-gnu (1)           - control magnetic tape drive operation
```

[1] Для получения справки по регулярным выражениям смотрите руководство regex(7).

```
rmt (8)              - remote magtape protocol module
rmt-tar (8)          - remote magtape protocol module
xmag (1x)            - magnify parts of the screen
```

Поскольку выражение `mag[tn]` соответствует magtape («магнитная лента») и magnify («увеличивать»), результаты поиска включают все позиции, которые содержат любое из этих слов. При этом mandb-вариант `apropos` в дистрибутивах Debian позволяет ограничить диапазон результатов точными соответствиями с помощью параметра `-e`. Это, к примеру, дает возможность выполнять поиск по слову compress («сжимать»), не учитывая соответствующее ему слово compressor («архиватор»). В табл. 3.2 приведены возможности поиска в зависимости от используемого дистрибутива.

Таблица 3.2. Возможности поиска в зависимости от дистрибутива

Возможность поиска	Традиционный вариант	mandb-вариант
Регулярные выражения	Да	Да
Точные совпадения		Да
Грубые совпадения	Да	

Традиционная версия man позволяет осуществлять грубый поиск по ключевым словам, проверяя каждое слово в каждой странице руководства. Для этого необходимо добавить к команде `man` параметр `-K` (прописной буквой), но поиск займет много времени даже в очень мощной системе. Если необходим тщательный поиск, то это лучшее, что вы можете сделать.

3.2.4. Поиск нужной страницы руководства man: команда whatis

Вы уже знаете, что в разных разделах могут содержаться страницы man с одинаковыми именами, в результате чего вы можете найти не ту страницу, если не знаете, в каком точно разделе нужно было искать. Если, например, вам необходимо узнать, как пользоваться функцией `readdir`, и вы введете команду `man readdir`, то попадете в раздел 2 руководства, где увидите следующее:

```
This is not the function you are interested in. Look at readdir(3) for the POSIX
conforming C library interface. This page documents the bare kernel system call
interface, which can change, and which is superseded by getdents(2)[1].
```

К счастью, в данном случае страница man достаточно информативна, чтобы вы могли понять, что видите сведения о системном вызове, а не о функции POSIX. Здесь даже можно узнать раздел, в котором следует искать. В других ситуациях, когда страница man окажется не столь полезной, вас выручит команда `whatis`:

---

[1] Это не та функция, информацию о которой вы ищете. Вам необходима функция POSIX readdir(3), соответствующая библиотечному интерфейсу C. Данная страница содержит сведения об интерфейсе системных вызовов ядра, которые могут изменяться и заменяться функцией getdents(2).

```
$ whatis readdir
readdir              (2)  - read directory entry
readdir              (3)  - read a directory
```

Поскольку наша искомая страница man содержится в разделе 3, теперь мы можем указать ее верные данные в команде:

```
$ man 3 readdir
```

Традиционный вариант команды `whatis` не использует аргументов, поэтому совпадения будут грубыми, а mandb-вариант позволяет находить регулярные выражения и соответствия образцов в стиле оболочки. Если вы не помните команду, но запомнили, что она оканчивается на «zip», можете попробовать поискать следующим образом:

```
$ whatis -w "*zip"
funzip (1)           - filter for extracting from a ZIP ..
gunzip (1)           - compress or expand files
gzip (1)             - compress or expand files
unzip (1)            - list, test and extract compressed ...
zip (1)              - package and compress (archive) files
```

Менее элегантный способ отыскать нужную страницу руководства man заключается в изучении всех разделов. Для этого необходимо добавить к команде `man` параметр `-a`, который говорит показать все страницы man, соответствующие критерию поиска. Порядок представляемых разделов определяется локальной конфигурацией, что может быть утомительным. Например, команда `man -a read` в моей системе Fedora Core 4 выводит результат из восьми страниц руководства man. Необходимость тщательного изучения восьми страниц man может стимулировать вас запомнить номер нужного раздела, когда вы будете искать что-либо в следующий раз.

3.2.5. Важные разделы страниц руководства man

Страницы man Linux подчиняются системе стандартов, документированной в разделе 7 руководства[1]. Минимальная страница man содержит раздел `NAME`, который состоит из имени программы или иного документируемого объекта, после чего следует краткое описание. Именно этот текст исследуется при вводе команды `apropos`.

Естественно, большинство страниц man состоят из нескольких разделов `NAME`. Сюда также входит несколько стандартных разделов, к которым программисты могут совершенно свободно добавлять новые.

В разделе `SEE ALSO` страницы man вы встретите множество перекрестных ссылок на документацию. Не удивляйтесь, если какая-либо из них окажется ссылкой на страницу man, не существующую в вашей системе (или где бы то ни было еще). Иногда бывает, что некоторые старые страницы ссылаются на программы, которые были удалены. Может также оказаться, что страница man ссылается на программу, которая еще не установлена в вашей системе. Всегда следует обращать внимание

[1] См. man(7).

на раздел SEE ALSO и искать перекрестные ссылки. Это позволит вам получить больше дополнительной информации по какому-либо вопросу, чем если бы вы изучили лишь одну страницу man.

Из раздела ENVIRONMENT вы сможете почерпнуть ценные сведения о том, как переменные окружения оказывают влияние на поведение программ. Обычно в этом разделе рассматриваются только вопросы локализации, но иногда в переменных окружения могут инкапсулироваться некоторые полезные «сокращения». К примеру, некоторые программы допускают размещение в таких переменных длинных параметров командной строки, чтобы не печатать их заново каждый раз. Изучением раздела ENVIRONMENT никогда не следует пренебрегать.

Другим разделом, в который рекомендуется заглянуть, является раздел CONFORMING TO, поясняющий стандарты, которым соответствуют команды и функции. Это особенно важно, если вы занимаетесь написанием переносимых программ. Существует множество разнообразных функций, которые выполняют в Linux одно и то же действие. Подобное соответствие важно учитывать при решении вопроса, какую из них предпочесть для использования. Например, функция bcopy выполняет те же действия, что и функция memcpy, но если вы заглянете в раздел CONFORMING TO функции bcopy, то увидите, что она соответствует 4.3BSD. Функция memcpy тоже соответствует 4.3BSD, но, поскольку она также является частью стандарта ISO C (ISO 9899), для обеспечения переносимости программ необходимо использовать именно ее.

В нижней части страницы руководства man находится раздел BUGS. Возможно, такое название использовано не совсем правильно, поскольку некоторые пользователи считают, что единственная разница между ошибкой и свойством заключается в наличии документации. Тем не менее раздел BUGS обычно содержит справочные сведения об ограничениях в конструкциях программ, а также свойствах, которые не обладают полноценной функциональностью. Иногда здесь упоминаются свойства, которые планируется внедрить. Данный раздел является необязательным, но, если он присутствует, его следует изучить.

3.2.6. Рекомендуемые страницы руководства man

В любом разделе руководства имеется начальная страница, которая может помочь вам, если вы запутались в разделах. Несмотря на то что на таких страницах содержится немного полезной информации[1], они могут стать удобным справочником по названиям разделов:

```
$ whatis intro
intro          (1) - Introduction to user commands
intro          (2) - Introduction to system calls
intro          (3) - Introduction to library functions
intro          (4) - Introduction to special files
intro          (5) - Introduction to file formats
intro          (6) - Introduction to games
intro          (7) - Introduction to conventions and miscellany section
intro          (8) - Introduction to administration and privileged commands
```

[1] Исключением является intro(2). Здесь можно почерпнуть вводные сведения о системных вызовах.

Если идея время от времени читать руководства кажется вам странной, то, скорее всего, это означает, что вы никогда не уделяли этому занятию должного внимания. Существует несколько страниц, содержащих полезную информацию и справочные материалы, которые вы никогда не нашли бы, если бы они вам не потребовались. Многие из таких страниц располагаются в разделе 7, которому, вероятно, читатели уделяют меньше всего внимания при изучении руководства. В табл. 3.3 приведена выборка страниц из раздела 7, которые могут вас заинтересовать.

Таблица 3.3. Страницы из раздела 7 руководства, рекомендуемые к прочтению

Имя	Описание
ascii(7)	Небольшая и удобная страница кодов ASCII в таблицах. Очень полезна, когда нужно получить подробные сведения, например, о восьмеричной константе для Ctrl+G
boot(7)	Отличный обзор последовательности загрузки ядра. Рекомендуется к прочтению, если вы создаете дистрибутив «с нуля» или просто хотите изучить процесс загрузки Linux
bootparam(7)	Перечень низкоуровневых параметров, которые может принимать ядро через командную строку. Данный список весьма обширен и достаточно информативен, но, скорее всего, не будет исчерпывающим
charsets(7)	Обзор наборов символов, используемых в Linux, с кратким описанием особенностей каждого из них. Рекомендуется к прочтению, если вы работаете в интернационализированных программах (i18n)
hier(7)	Обзор стандарта файловой иерархии Filesystem Hierarchy Standard, который описывает систему расположения каталогов в Linux. Здесь вы найдете ответы на различные актуальные вопросы, например: «Какая разница между /bin и /usr/bin?» Если вы разрабатываете приложение для последующего распространения, то данную страницу необходимо изучить до того, как вы решите, куда будут устанавливаться ваши файлы
man(7)	Вы захотели создать собственную страницу руководства man? Здесь вы найдете инструкции, как это можно сделать с помощью языка разметки troff, а также сведения о системе стандартов, которым подчиняются страницы man в Linux
operator(7)	Содержит перечень операторов C вместе с их приоритетами. Рекомендуется к изучению, если вы не любите использовать лишние круглые скобки в своем программном коде. После прочтения этой страницы вы сможете найти ошибку в этом операторе C: if (1 & 2 == 2) printf(“bit one is set\n”);
regex(7)	Содержит вводные сведения о регулярных выражениях, в которых должен хорошо разбираться любой программист
suffixes(7)	В Linux применяются общепринятые файловые суффиксы. Данная операционная система использует надежные методики определения типов файлов. На этой странице вы сможете найти описание многих известных стандартных суффиксов. Перечень не исчерпывающий, но полезный
units(7)	Содержит список стандартных множителей, определяемых системой SI6 (SI — сокращение от Système International d’unités — Международная система научных измерений). Известно ли вам, что, согласно правилам SI, 1 Мбайт на самом деле равен 1 000 000, а не 1 048 576 байтам? Раз производители жестких дисков знают об этом, то это должны знать и вы. Разработчики программного обеспечения имеют тенденцию использовать префиксы для десятичных множителей, в то время как мы предпочитаем применять двоичные префиксы.

Имя	Описание
units(7)	Система SI определяет уникальные префиксы для двоичных значений, которыми никто не пользуется. Хотя, возможно, стоило бы. Кстати, 1 048 576 байт правильно называть мебибайтом (сокращенно Миб). Несмотря на то что подобная терминология вряд ли станет широко использоваться в ближайшем будущем, вам необходимо владеть ею, чтобы ориентироваться в понятиях при общении с учеными, создателями аппаратного обеспечения или поставщиками жестких дисков. Помимо этого, здесь также описываются разные мелочи вроде префикса для миллиона миллиардов миллиардов. Вы сможете произвести впечатление на друзей своими знаниями
uri(7), url(7), urn(7)	Три ключа для страницы руководства man, где описываются компоненты URL-ссылок вместе с их значениями. Вы можете полагать, что все знаете об URL, но, изучив эту страницу, возможно, узнаете что-то новое

Помните, что все страницы руководства man, приведенные в табл. 3.3, находятся в разделе 7, при этом имена некоторых из них могут вступать в конфликт с аналогичными именами прочих страниц, расположенных в других разделах. Указывайте правильный номер раздела, чтобы отыскать нужную страницу man.

3.2.7. GNU-программа info

При поиске документации по GNU-инструментарию необходимо пользоваться программой info. Зачастую GNU-страницы руководства man содержат сокращенный вариант документации с отсылкой за дополнительной информацией к программе info.

Для создания GNU-страниц info применяется язык текстовой разметки Texinfo. Как и troff, Texinfo является предшественником HTML и XML, но основан на других проектах. Он эволюционировал из языка форматирования TeX (по-прежнему широко используемого и сейчас) и еще одного проекта под названием *Scribe*, который сейчас называется *схемовой разметкой*. Texinfo предназначен не для форматирования, а для разметки содержимого и облегчает написание документов «с нуля» по сравнению с другими языками. Как и в случае с troff, формат Texinfo поддерживает вывод на печать и может быть преобразован в HTML или обычный текстовый формат.

3.2.8. Просмотр страниц info

GNU-система info тесно интегрирована с основным текстовым GNU-редактором Emacs. На рис. 3.2 вы можете увидеть, как выглядит страница info в редакторе Emacs. Если вы не используете Emacs, то для вас существует пара альтернативных решений. Первое из них — текстовая программа info. Людям, которые привыкли к страницам руководства man, пользоваться браузером info неудобно главным образом потому, что он работает во многом не так, как это делает пейджер less, используемый man. Постоянный переход между страницами man и info может сильно раздражать. Программистов, которые используют текстовый редактор vi, вполне устраивает программа less, поскольку принятые в ней сочетания клавиш

во многом аналогичны тем, что применяются в vi. Дело в том, что программа info многое заимствует из текстового редактора Emacs и весьма схожа с ним в поведении. Добавив к info параметр `--vi-keys`, пользователи текстового редактора vi могут просматривать сведения о сочетаниях клавиш, однако функционирует info во многом не так, как это делает vi или less.

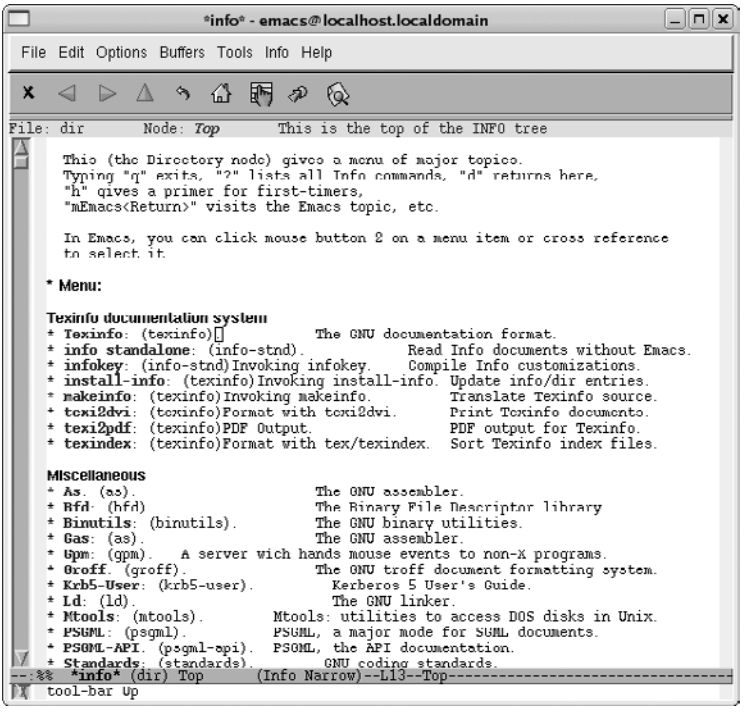


Рис. 3.2. Использование текстового редактора Emacs для просмотра страниц info

Итак, вы решили научиться работать с программой info. Для начала откроем страницу man, которая затрагивает только параметры командной строки. Если вам нужны более подробные сведения, следует обратиться к странице info, но торопиться не стоит. Если вы введете `info info`, то получите сведения о формате info, а не о программе info. Для просмотра документации о программе нужно вводить `info info-stnd`. Страница man должна указывать вам на это, но в силу разных причин она может этого и не делать.

Допустим, программа info вам понравилась. С ее помощью вы можете просматривать как страницы man, так и info. По умолчанию info осуществляет поиск документации в формате info, а при ее отсутствии обращается к страницам man.

Для программистов, которые не используют текстовый редактор Emacs и не питают интереса к программе info, существует еще одно альтернативное решение — программа pinfo (http://pinfo.alioth.debian.org). Пользователям, предпочитающим страницы man, не составит труда научиться пользоваться ею. Как и info, она является текстовой программой, но обладает более продвинутыми терминальными

возможностями, включая цветное выделение гиперссылок. А самое важное заключается в том, что для перемещения курсора в этой программе используются сочетания клавиш, применяемые в текстовом редакторе vi.

Что касается GUI-интерфейсов, то здесь также имеются иные средства просмотра страниц info. Некоторые браузеры поддерживают работу с URL-ссылки вроде `info:topic`. В системе GNOME есть браузер gnome-help, который на момент написания данной книги не поддерживал просмотр страниц man или info. Ужас!

Браузер KDE отсылает пользователя к инструменту khelpcenter, который является отличным средством просмотра страниц info (рис. 3.3).

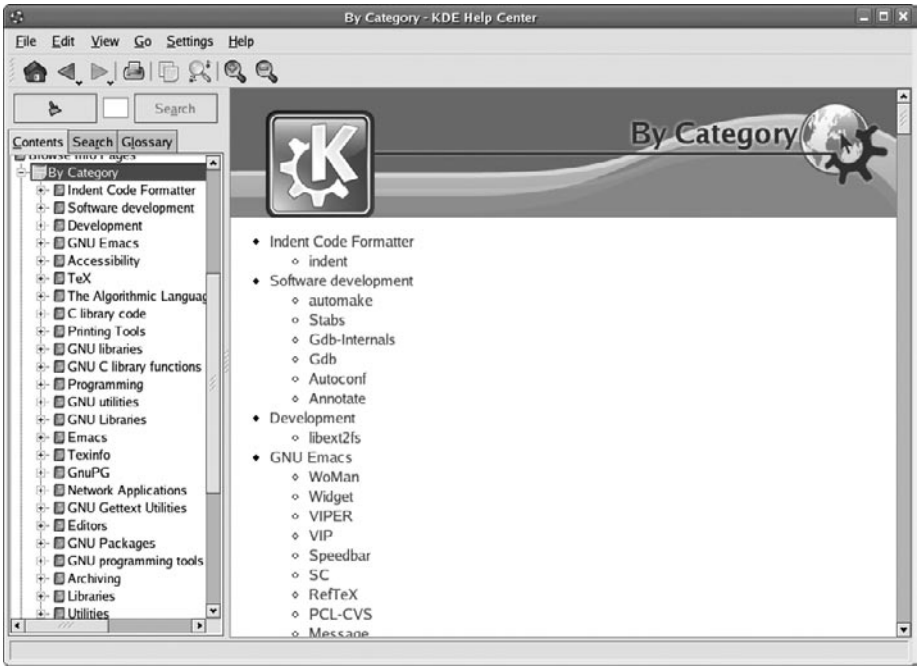


Рис. 3.3. Использование khelpcenter для просмотра страниц info

Обычно документы info состоят из разделов, что облегчает чтение, однако искать нужный материал иногда бывает утомительно. Если вы знаете, какой именно раздел вам нужен, то можете открыть его с помощью такой длинной команды:

```
$ info "(make)Quick Reference"
```

Ввод такой команды требует не так уж мало времени, однако, если вы часто обращаетесь к какому-либо разделу, используйте псевдоним оболочки или сценарий. Программы pinfo и info позволяют с помощью этого синтаксиса «перескакивать» из одного раздела в другой, в отличие от khelpcenter, который на момент написания этой книги не давал такой возможности. Если вам нужен один сплошной вывод, например страница man, то можете заставить программу info сформировать его:

```
$ info --subnodes some-topic | less
```

Параметр `--subnodes` принуждает info отправлять дамп всего содержимого документа на stdout. После передачи этого вывода к less он приобретает вид довольно большой страницы руководства man.

3.2.9. Поиск страниц info

Как и в случае со страницами man, поиск страниц info осуществляется с помощью параметра `--apropos`. Поскольку каждый файл info имеет индекс, который можно отыскать посредством параметра `--apropos`, воспользовавшись этим параметром, вы сможете отыскать страницы info с большей долей вероятности, чем с помощью команды `apropos`, используемой для поиска страниц man. Как и в случае с man, поиск чувствителен к регистру и охватывает весь тест. В отличие от man, использование регулярных выражений невозможно. Таким образом, несмотря на то что поиск по индексу позволяет быстрее получить точные результаты, данное преимущество сводится на нет отсутствием фильтрации посредством регулярных выражений или ключевых слов.

Программа info также может использоваться с параметром `-w`, который выполняет ту же функцию, что и команда `whereis`, однако из-за того, что количество файлов info не очень велико, вы вряд ли часто будете сталкиваться с конфликтами имен, как это бывает в случае с man.

3.2.10. Рекомендуемые страницы info

Несмотря на то что руководство по GNU-инструментарию содержит лишь краткие страницы man со ссылками на документацию info, участники проекта GNU снабдили весьма подробными страницами man многие инструменты, в число которых входит gcc. Однако даже они не могут сравниться со своими Texinfo-аналогами. Страница man компилятора gcc 3.4.4, к примеру, насчитывает около 54 000 слов, в то время как страница info состоит почти из 158 000 слов. Файл info содержит намного больше вспомогательной и исторической информации, а на странице man отображаются лишь самые важные сведения.

Если у вас появится свободное время, ознакомьтесь с некоторыми страницами info. В табл. 3.4 приведена их выборка.

Таблица 3.4. Страницы info, рекомендуемые к прочтению

Страница info	Комментарий
coreutils	Данная страница содержит множество сведений, изложенных понятным языком. Здесь приводятся различные столь любимые программистами UNIX двух- и трехбуквенные команды, которые располагаются последовательно в соответствии со своим назначением. Присутствует и перепечатка статьи Арнольда Роббинса (Arnold Robbins) для журнала Linux Journal, которую можно отыскать в узле Opening the software toolbox (Знакомство с программным инструментарием)
cpp	Данный инструмент наиболее тяжел для понимания, по сравнению с каким-либо другим, используемым программистами. Страница man содержит очень мало сведений о нем. Поскольку этот инструмент весьма сложен, для получения справочной информации о нем необходимо пользоваться страницей info

Страница info	Комментарий
gcc	На этой странице вы найдете намного больше сведений, чем на странице man компилятора gcc, включая подробности реализации. При этом она отлично адаптирована для интерактивного чтения
ld	«Закулисный» инструмент, используемый разработчиками, из которых мало кто понимает, как он работает. Данный инструмент также весьма сложен, и для его изучения лучше пользоваться страницей info
libc	На этой странице содержится множество справочной информации по стандартным библиотекам C наряду с большим количеством вспомогательных сведений

Несмотря на то что страницы man и info покрывают большинство инструментов, существует множество других способов документирования программ.

3.2.11. Справочный инструментарий Рабочего стола

В операционной системе Linux в окружении Рабочего стола существует два основных игрока: GNOME и KDE. Несмотря на то что у каждого из них есть свои почитатели и противники, ни одно из таких окружений Рабочего стола не будет иметь завершенный вид без интерактивного справочного руководства. Справочная система GNOME вызывается программой gnome-help (также известной как yelp), а в версии 2.10.0 основное внимание сосредоточено исключительно на инструментарии GNOME. Однако представляемые ею сведения не всегда актуальны для программистов. Несмотря на то что упомянутый инструмент не может предложить многого в плане информации, у него есть и другое применение. Он служит превосходным компактным браузером для простых HTML-файлов, например `README` на HTML. Применение браузера Mozilla или Firefox для просмотра подобных файлов напоминает использование авианосца для катания на водных лыжах. Кроме того, кому охота возиться с вводом ссылок? Браузеры Mozilla и Firefox требуют ввода полных URL-адресов:

```
$ firefox file:///usr/share/doc/someproject/README.html
```

Как вы уже знаете, после `file:` необходимо ставить три слеша (данный пример не является исключением). Инструмент yelp способен находить файлы в текущем каталоге и запускается намного быстрее, чем браузеры Mozilla и Firefox. Поэтому каждый раз, когда вы будете иметь дело с каталогом, в котором располагается справочная документация, для ее открытия вам достаточно будет ввести следующую команду:

```
$ yelp README.html
```

Поскольку подключаемые модули отсутствуют, yelp сможет открыть нужный HTML-файл почти так же быстро, как программа man — страницу руководства man.

Как и yelp, khelpcenter представляет собой отличный компактный браузер для просмотра простых HTML-файлов. В качестве бонуса khelpcenter обеспечивает

намного менее «близорукое» рассмотрение инструментов по сравнению с yelp. Несмотря на то что khelpcenter служит *источником* справочной информации о KDE-инструментарии, он также позволяет просматривать страницы man и info GNU. Страницы info, как и предполагается, связаны гиперссылками, однако сюрпризом является то, что страницы man тоже оказываются связанными ими. Если найти раздел SEE ALSO страницы man и щелкнуть на любой из перекрестных ссылок на другие страницы man, то на экране появится соответствующий документ. Очень удобно. Инструмент khelpcenter также позволяет осуществлять поиск и содержит словарь. Функция поиска пока несовершенна, а словарь скуден, но эти недостатки обязательно будут устранены.

3.3. Другие источники справочной информации

Если вы имеете дело с новым проектом, то, скорее всего, в нем мало или вообще нет документации в виде страниц man или info. Справочные сведения могут присутствовать, однако их может быть сложно найти. Файлы вроде README постепенно исчезают из двоичных дистрибутивов проектов. Однако существует вероятность того, что справочная документация все-таки где-то имеется, нужно лишь отыскать ее.

3.3.1. /usr/share/doc

Данный подкаталог является частью стандарта файловой иерархии Filesystem Hierarchy Standard, и в нем располагается множество страниц man или info. Он также хорошо подходит для сохранения электронной версии записей «на обратной стороне салфетки», которые мы называем файлами README.

Двоичный дистрибутив обычно содержит подкаталог, имя которого соответствует определенному проекту, и в нем может располагаться всевозможная информация. Зачастую здесь можно найти комментарии к выпуску, журналы регистрации изменений, сведения об авторских правах и файлы README.

Иногда разработчики включают лишь документацию в текстовом или HTML-формате без каких-либо страниц руководства man. Именно в этом подкаталоге они обычно и находятся. Кроме того, здесь вы можете встретить исходные файлы в формате Texinfo, а также файлы других форматов, например DocBook.

В дистрибутивах Debian присутствуют пакеты doc-linux, в которых в подкаталоге /usr/share/doc содержится множество документов HOWTO из проекта The Linux Documentation Project (www.tldp.org) (о нем мы поговорим немного позднее). Эти файлы не связаны с какими-либо устанавливаемыми пакетами и содержат справочную документацию.

3.3.2. Перекрестные ссылки и индексация

Перекрестные ссылки позволяют превратить документацию из словаря в справочник. В словаре можно лишь увидеть определенные искомые слова, а в спра-

вочнике вы рискуете встретить такие слова, о которых и не подозревали. Документация Linux с перекрестными ссылками весьма важна, поскольку обычно существует несколько путей выполнить что-либо, при этом вы можете и не знать этого, пока не воспользуетесь перекрестными ссылками. Документация Linux содержит множество полезных ссылок на дополнительный инструментарий, несмотря на то что она аккумулировалась из большого количества различных источников.

Что касается страниц руководства man, то перекрестные ссылки на прочую информацию можно отыскать в разделе `SEE ALSO`. Допустим, вы настраиваете сервер DHCP и решили изучить страницу `dhcpd man`. По логике именно на этой странице и будут нужные вам сведения, так как `dhcpd` — это имя демона DHCP. Если вы не обратите внимания на перекрестные ссылки, то можете пропустить страницу `dhcpd.conf`, на которой содержится более важная информация. Многие комплексные инструменты и серверы снабжаются документацией из нескольких страниц man, которые также имеют перекрестные ссылки в разделе `SEE ALSO`. Типичная перекрестная ссылка на странице man выглядит следующим образом:

```
SEE ALSO
       dhclient(8), dhcrelay(8), dhcpd.conf(5), dhcpd.leases(5)
```

Систему обозначений, используемую на страницах man, мы рассматривали ранее. Как видно из этого примера, в данном разделе страницы man соблюдается именно такая система.

Еще одним важным инструментом для работы с перекрестными ссылками на страницах man является команда `apropos`, о которой мы уже говорили. Не забывайте, что она применима, только если база данных индексирована. Обычно так и бывает при свежей инсталляции, однако по мере накопления программ, содержащих новые страницы man, и удаления старых синхронизация индексов нарушается. Вследствие этого нужная документация может не обнаружиться при использовании команды `apropos` либо полученные результаты могут касаться программного обеспечения, которое было удалено из системы. Чтобы избежать этого, нужно использовать соответствующий индексатор для инсталляции man. Традиционная версия man использует инструмент makewhatis, который обычно запускается как задание cron. Процесс занимает достаточно много времени, поскольку этому инструменту приходится посещать каждую страницу man в системе. Если вы работаете на лэптопе, который часто включаете и выключаете, то данное задание вряд ли стоит запускать. Если же оно все-таки будет выполняться, то, скорее всего, аккумуляторной батареи вашего лэптопа надолго не хватит.

А вот mandb-вариант man использует инструмент, который называется (вполне подходяще) mandb. Важное преимущество заключается в том, что при использовании программы mandb индексация занимает намного меньше времени, чем в случае с makewhatis. Обычно она также запускается как задание cron. Файлы info связываются между собой с помощью гиперссылок. Программа info также поддерживает использование apropos-функции посредством параметра `--apropos`. Она не обладает гибкостью команды `apropos`, но может оказаться полезной.

3.3.3. Запрос пакетов

В главе 1 мы детально рассмотрели использование пакетов. В этом разделе мы воспользуемся полученными знаниями на практике. Сначала вам может потребоваться узнать основные сведения о каком-либо пакете. Это особенно полезно в ситуациях, когда вас интересует назначение определенной команды. Допустим, вы нашли в каталоге /usr/bin программу diffstat и хотите узнать, для чего она предназначена. Возможно, вы попытаетесь ввести команду man diffstat, но результат может оказаться нулевым. Следующий этап — использовать базовый запрос:

```
$ rpm -qf /usr/bin/diffstat
diffstat-1.38-2
```

Такой запрос позволяет узнать имя пакета, из которого была установлена программа, расположенная в каталоге /usr/bin/diffstat. В нашем случае этот пакет называется diffstat-1.38-2. После этого можно осуществить запрос сведений и получить базовое описание пакета, из которого происходит данная программа:

```
$ rpm -qi diffstat
...
Summary    : A utility which provides statistics based on the output
             of diff.
Description : ...
```

Иногда этого бывает достаточно, чтобы узнать то, что требовалось, однако теперь, когда мы знаем имя пакета, можно запросить его содержимое и увидеть, разрешено ли почерпнуть из него какие-либо другие сведения. В перечне установленных файлов следует искать файлы README или HTML-документы, в которых может содержаться дополнительная информация. Следует также обращать внимание на расположенные не в тех местах страницы руководства man. Они могут оказаться в нестандартных локациях, которые в иной ситуации вы бы не стали исследовать. В случае с пакетом diffstat-1.38-2 выяснилось, что кто-то расположил страницу man в неправильном месте:

```
$ rpm -ql diffstat
/usr/bin/diffstat
/usr/share/man/man1/man1
/usr/share/man/man1/man1/diffstat.1.gz
```

Поскольку при архивировании в подкаталоге man1 был ошибочно создан дополнительный подкаталог man1, команда man не позволяет найти эту страницу. Несмотря на то что эта проблема была устранена в новой версии diffstat, она служит наглядным примером того, насколько могут быть полезны запросы. Ведь архивы с файлами создают люди. Если вам интересно, то для того, чтобы прочесть данную страницу man, нужно ввести следующую команду:

```
$ man /usr/share/man/man1/man1/diffstat.1.gz
```

В большинстве дистрибутивов, если страницу man не удается найти, вы остаетесь ни с чем. В Debian-дистрибутивах вы можете столкнуться со страницей undocumented(7), которая является общей страницей man, где содержится информация о некоторых из рассмотренных в этом разделе предметов.

3.4. Форматы документации

Документация Linux бывает различных форматов. Несмотря на распространенность обычного текстового формата, на протяжении многих лет используются языки разметки, позволяющие создавать документы, которые более красиво выглядят на бумаге или более дружественны к браузерам. Одним из самых первых был язык troff, с помощью которого создаются страницы руководства man. Он генерирует вывод, который можно набирать или просматривать в окне терминала. Другие форматы, например LaTeX, создавались для последующего вывода на бумагу, однако они также поддерживают создание документации для просмотра в браузерах. Наиболее предпочтительным GNU-форматом является Texinfo, который предназначен для создания гипертекстовых документов для просмотра в браузерах или вывода на бумагу. И наконец, упомянем вездесущий формат HTML, предназначенный исключительно для браузеров. С каким из этих форматов вам доведется столкнуться? Ответ прост: со всеми.

3.4.1. TeX/LaTeX/DVI

История TeX и LaTeX уходит своими корнями в UNIX. LaTeX представляет собой расширение системы разметки TeX. В настоящее время большинство документов TeX на самом деле имеют формат LaTeX, однако оба этих названия используются как равнозначные. TeX, первоначально предназначавшийся для создания документов, выводимых на бумагу, широко поддерживает особые наборные требования научных документов, в которых содержатся математические символы и формулы. TeX значительно облегчил работу ученых в те времена, когда текстовые процессоры пребывали в зачаточном состоянии, тем самым создав для себя нишу среди данной категории пользователей. Несмотря на совершенствование стандартных программ для работы с текстом, TeX по-прежнему широко применяется в наши дни.

«Родной» выходной формат TeX называется DVI (сокращенно от *Device Independent — независимый от устройства вывода*). Документы в этом формате можно просматривать с помощью программ xdvi, kdvi и evince, однако DVI создавался как промежуточный формат.

TeX и LaTeX используют расширение `.tex` для исходных файлов, что может приводить к путанице. Компиляция документов TeX может оказаться невозможной с помощью команды `latex`; в такой ситуации необходимо использовать команду `tex`. Во многих случаях документы LaTeX состоят из нескольких исходнях файлов, поэтому, перед тем как использовать для создания документа исходные файлы LaTeX, убедитесь, что у вас есть все необходимые из них. Иногда документ оказывается настолько сложным, что вам потребуется файл Makefile.

3.4.2. Texinfo

Texinfo — наиболее оптимальный формат документации GNU-программ. В качестве основного браузера для просмотра GNU-документации можно применять

программу info либо текстовый редактор Emacs — выбор за вами. Файлы, которые использует info, располагаются в каталоге /usr/share/info, однако они не являются файлами Texinfo. Они создаются из исходных файлов Texinfo с помощью программы makeinfo. В большинстве двоичных дистрибутивов исходные файлы Texinfo отсутствуют, вместо них включаются предварительно форматированные файлы info. Их можно узнать по расширению .info, а файлы Texinfo — по расширению .texi или .texinfo.

Как и следовало ожидать, исходные дистрибутивы могут содержать исходные файлы Texinfo, однако и некоторые двоичные дистрибутивы также включают такие же файлы. Если в двоичном дистрибутиве имеются исходные файлы Texinfo, то они, скорее всего, расположены в подкаталоге /usr/share/doc. В него стоит заглянуть, потому что иногда там можно найти более обстоятельную документацию по сравнению с той, что содержится в руководстве или на страницах info.

Форматирование исходных файлов Texinfo осуществляется с помощью программы makeinfo. Вы можете преобразовывать их в info, HTML, DocBook, XML или обычный текстовый формат.

3.4.3. DocBook

DocBook — это формат языка SGML (сокращение от *Standard Generalized Markup Language*), используемый для создания справочной документации по некоторому инструментарию. Строго говоря, DocBook является не файловым форматом, а определением типов документов SGML Document Type Definition (DTD). SGML — это предшественник и расширенный набор широко распространенного формата XML. DocBook преследует те же цели, что и SGML: обеспечение формата хранения документации, который разделяет содержимое и стиль. Это позволяет осуществлять электронный поиск содержимого без учета стилевой и разметочной информации, которая может исказить результаты.

DocBook DTD совместим как с SGML, так и с XML, хотя обычно файлы DocBook имеют расширение .sgml. DocBook не является дружественным к пользователю форматом, который можно применять для создания документов «с нуля».

3.4.4. HTML

Из-за повсеместной распространенности HTML многие создатели документации предпочитают этот формат другим языкам разметки, например Texinfo, DocBook и troff. Веб-страницы, содержащие документацию проектов, обычно создаются вручную с использованием текстовых редакторов, например vi или Emacs. Обычно они представляют собой один или два файла, которые не содержат рисунков, JavaScript или флэш-анимации. Благодаря наличию компактных браузеров вроде yelp и khelpcenter вам не обязательно запускать для их просмотра какие-либо другие громоздкие веб-браузеры. HTML-документация обычно располагается в подкаталоге /usr/share/doc.

Для создания документации к некоторым инструментам применяется исключительно HTML, страницы руководства man у них отсутствуют. К примеру, пакет

NTP (Network Time Protocol) содержит обширную документацию в формате HTML, пример которой показан на рис. 3.4.



Рис. 3.4. Пример HTML-документации из пакета NTP

3.4.5. PostScript

Язык PostScript, созданный компанией Adobe, является предшественником широко распространенного сейчас формата Portable Document Format (PDF). PostScript когда-то был основным языком принтеров в UNIX. Программные пакеты (как коммерческие, так и с открытым исходным кодом) часто содержали документацию в формате PostScript, точно так же, как в настоящее время используется формат PDF.

Инструментарий GNU Ghostscript обеспечил возможность широкого применения PostScript в UNIX. Ghostscript позволил тем из нас, кому не по карману были принтеры PostScript, печатать документы PostScript на менее дорогих (хотя и довольно медленных) принтерах. Ghostscript дает возможность просматривать файл PostScript на экране компьютера, вместо того чтобы его распечатывать.

3.4.6. Portable Document Format (PDF)

Несмотря на то что формат PDF отлично подходит для формирования наборных документов, выводимых на печать, он также вполне годится для интерактивного просмотра. Одной из причин его популярности является то, что он позволяет раз-

работчикам экономить деньги, вынуждая пользователей самостоятельно распечатывать справочные руководства. Хотя это и доставляет им неудобство, лучше иметь один лист с распечаткой на десяти языках, чем вообще никакого руководства.

Ghostscript может отображать PDF-файлы, однако лишен изящества программы Adobe Acrobat Reader. К примеру, отсутствуют гиперссылки или оглавление. Программа с открытым исходным кодом `xpdf`[1] существует уже давно, однако медленно открывает файлы. GNOME-приложение `evince` появилось относительно недавно, но открывает файлы значительно быстрее, чем `xpdf`. В среде GNOME `evince` планируется использовать как средство просмотра файлов PostScript, PDF и DVI.

3.4.7. troff

troff является «родным» языком разметки страниц руководства `man`. Он имеет долгую историю, предшествующую UNIX[2]. Для этих целей служит инструмент `groff` — GNU-версия troff, однако он вам не потребуется, поскольку для выполнения большинства задач достаточно будет программы `man`. Например, следующие две команды являются эквивалентными:

```
$ man intro
$ gzip -dc /usr/share/man/man1/intro.1.gz | groff -man -Tascii | less
```

Программа `man` позволяет отыскать нужную страницу `man` и распаковать ее при помощи архиватора gzip.

groff может создавать DVI-файлы из исходных файлов troff, способен он генерировать и другие форматы. Большинство из них имеют отношение к принтерным языкам вроде HP PCL и PostScript. Несомненно, все это является наследием UNIX. Поскольку вы также можете преобразовывать свои файлы в формат DVI, используйте данный инструмент, если пожелаете.

3.5. Источники справочной информации в Интернете

Если вы не смогли отыскать ответы на интересующие вопросы на своем жестком диске, за помощью следует обратиться к Интернету. Однако при поиске во Всемирной паутине вы рискуете зайти в тупик, потратив впустую много времени. В этом разделе я расскажу вам о некоторых ресурсах, на которых действительно можно найти полезные справочные материалы.

3.5.1. www.gnu.org

Помимо исходного кода для создания множества программ под Linux, на данном сайте имеются также справочные руководства. Здесь вы найдете руко-

---

[1] См. сайт www.foolabs.com/xpdf.

[2] Те, кому интересно, могут почитать roff(7) интерактивного руководства. При этом следует учитывать, что там есть некоторые неточности.

водства, как входящие в состав пакетов, так и отсутствующие в них. Библиотека GNU C (glibc), например, полностью документирована в опубликованном руководстве из двух томов, насчитывающем более 1300 страниц. Вы можете приобрести их по 60 долларов каждый либо загрузить исходные файлы Texinfo с сайта GNU.

3.5.2. SourceForge.net

Многие проекты с открытым исходным кодом, не поддерживаемые проектом GNU, располагаются на сайте SourceForge.net, включая дистрибутивный инструментарий, который вы используете каждый день. Об этом далеко не всегда можно догадаться, исходя из сведений, содержащихся в том или ином пакете. Например, пакет strace, входящий в состав дистрибутива Ubuntu, не содержит ссылок на вебсайт SourceForge.net.

```
$ dpkg -s strace
Package: strace
Maintainer: Roland McGrath <frob@debian.org>
Description: A system call tracer
 strace is a system call tracer, i.e. a debugging tool ...
```

Фактически единственным упоминанием сайта SourceForge.net, которое здесь можно найти, является адрес электронной почты в списке рассылки в самой последней строке на странице man. Это просто ужасно, поскольку SourceForge.net является ценным ресурсом как для простых пользователей, так и для разработчиков. Что касается RPM-пакетов, то они выглядят более достойно, помещая ссылки на авторов оригиналов в URL-поле заголовка RPM:

```
$ rpm -qi strace
Name            : strace
Packager        : Red Hat, Inc. <http://bugzilla.redhat.com/bugzilla>
URL             : http://sourceforge.net/projects/strace/
Summary         : Tracks and displays system calls ...
```

Если вы имеете дело с дистрибутивом RPM, то стоит обратить особое внимание на его информационный раздел, где могут упоминаться источники справочных сведений. В случае использования дистрибутива Debian вы можете напрямую зайти на сайт SourceForge.net и изучить его.

3.5.3. Проект документации The Linux Documentation Project

Если вам никогда не доводилось посещать сайт www.tldp.org, то вы должны обязательно сделать это. Данный ресурс содержит страницы руководства man, которые используются в большинстве дистрибутивов Linux. Основная часть прочей имеющейся здесь информации носит более общий характер, поэтому вряд ли вы встретите сведения, к примеру, о каком-то конкретном приложении с открытым исходным кодом. Тем не менее здесь есть множество полезных материалов. Они

разбиты на несколько основных групп, включая HOWTO, Guides (Руководства), FAQ (Часто задаваемые вопросы) и, конечно же, страницы man.

Обращайте внимание на дату документов, которые собираетесь читать. Linux развивается настолько стремительно, что документация быстро устаревает. Поскольку написанием и поддержкой документации занимаются добровольцы, она может оставаться неактуальной, пока кто-нибудь не найдет время обновить ее.

Документы HOWTO зачастую содержат весьма полезные сведения, так как они освещают довольно специфические вопросы и пошагово проводят пользователя через процесс, не требуя при этом большого объема вспомогательных знаний. Обычно тематика документа HOWTO позволяет легко понять, стоит ли вам его читать. Поскольку все темы документов HOWTO строго конкретизированы, вы вряд ли натолкнетесь на такой, прочтение которого окажется пустой тратой времени.

Раздел часто задаваемых вопросов FAQ формируется в группах новостей Usenet, где часто задаются одни и те же вопросы[1]. Данный раздел призван снизить количество одинаковых вопросов, задаваемых слишком часто. К сожалению, в настоящее время термин *FAQ* используется не совсем правильно, поскольку в таких разделах зачастую содержатся вопросы, на которые автор желал бы ответить, а не такие, которые действительно были заданы. В отличие от любительских FAQ, которые можно встретить на заурядных веб-страницах, большая часть раздела FAQ на сайте проекта TLDP составлена из реальных вопросов, задаваемых реальными пользователями (некоторые из них как раз часто и задаются). Если вас интересует ответ на определенный вопрос, то стоит заглянуть в FAQ.

Хорошим стартом будет посещение раздела FAQ на сайте проекта The Linux Documentation Project по адресу www.tldp.org/FAQ/LDP-FAQ/index.html.

3.5.4. Usenet

Usenet имеет долгую историю, которая предшествует появлению Интернета. «Группы новостей» Usenet обычно содержат больше слухов и болтовни, чем самих новостей. Здесь любой может высказать свое мнение, и единственной реакцией на него могут быть так называемые «пламенные» постинги, размещаемые другими негодующими пользователями. Вы столкнетесь с множеством убеждений, взглядов на ту или иную проблему и всеобщим отсутствием манер.

Перед тем как задавать свой вопрос в одной из этих новостных групп, рекомендуется ознакомиться с архивами Usenet при помощи Google или другого поисковика. Существует вероятность того, что интересующий вас вопрос уже задавал кто-то ранее. Это также позволит вам узнать, кто регулярно размещает постинги, и посмотреть, оказывается ли помощь тем, кто за ней обращается.

Прежде чем использовать какую-либо информацию, полученную из Usenet, проверьте ее на других ресурсах. Относитесь скептически ко всему, что прочтете. Нет никаких гарантий того, что все прочитанное вами соответствует истине и будет безопасным для применения на практике.

---

[1] На случай, если вы не знаете, FAQ — это сокращение от Frequently Asked Questions.

Имейте в виду, что при размещении постингов в Usenet ваш адрес электронной почты будет виден всем пользователям. Группы Usenet являются настоящими золотыми жилами для спамеров.

3.5.5. Списки рассылки

С моей точки зрения, данный формат форума является наиболее полезным. Со списком рассылки вы получаете сведения о группе людей, которые активно увлечены определенной тематикой и, как правило, действительно стремятся помочь (в противном случае они бы не стали подписываться на него). Как и на любом форуме, прежде чем задавать вопрос, всегда следует подготовиться. Несмотря на то что люди в списках рассылки обычно обладают хорошими манерами, будет невежливо отвлекать их внимание простыми вопросами, ответы на которые вы сами могли бы найти без труда.

Большое количество списков рассылки можно отыскать в архиве (MARC) на сайте http://marc.theaimsgroup.com. Здесь содержатся сотни списков рассылки на тему Linux, а также многие другие, относящиеся к компьютерам.

3.5.6. Прочие форумы

На многих веб-сайтах по Linux имеются похожие на Usenet форумы, которые посвящены рассмотрению вопросов, связанных с этой операционной системой. Обычно для того, чтобы вы могли размещать на них свои постинги, сначала требуется подписаться на них, однако многие из таких форумов позволяют просматривать свои архивы без этой процедуры. Качество постингов может быть самым разнообразным.

3.6. Поиск информации о ядре Linux

Документирование ядра Linux является непростой задачей. На эту тему написано немало хороших книг, однако ядро постоянно развивается. В результате любая книга, посвященная ядру Linux, на момент публикации оказывается уже неактуальной. Наиболее свежую документацию можно найти в дереве исходных файлов ядра Linux. Исходные файлы ядра, распространяемые ресурсом http://kernel.org, содержат определенную долю документации. Кроме того, многие модули ядра предусматривают наличие полезных сведений, которые содержатся в объектах ядра.

3.6.1. Сборка ядра

Процесс сборки ядра хорошо документирован на многих ресурсах в Интернете. Несмотря на то что с выходом версии 2.6 данная процедура значительно упростилась, существует множество тонкостей и мелких деталей, о которых вам следует

знать. Первым делом, конечно же, стоит изучить файл README, расположенный в корне дерева исходных файлов ядра. Он начинается разделом What Is Linux («Что такое Linux»), от которого вам станет не по себе и сразу же расхочется читать дальше. Однако если вы не сдадитесь, то примерно на середине дойдете до рассказа о целях, сборку которых можно осуществить при помощи файла верхнего уровня Makefile. Некоторые из них довольно полезны, а чтобы увидеть общую картину, необходимо ввести:

```
$ make help

Cleaning targets:
  clean           - remove most generated files but keep the config
  mrproper        - remove all generated files + config + various
                    backup files

Configuration targets:
  config          - Update current config utilising a line-oriented
                    program
  menuconfig      - Update current config utilising a menu based
                    program
  xconfig         - Update current config utilising a QT based
                    frontend
  gconfig         - Update current config utilising a GTK based
                    frontend
  oldconfig       - Update current config utilising a provided
                    .config as base
  randconfig      - New config with random answer to all options
  defconfig       - New config with default answer to all options
  allmodconfig    - New config selecting modules when possible
  allyesconfig    - New config where all options are accepted with
                    yes
  allnoconfig     - New minimal config

Other generic targets:
  all             - Build all targets marked with [*]
* vmlinux         - Build the bare kernel
* modules         - Build all modules
  modules_install - Install all modules
  dir/            - Build all files in dir and below
  dir/file.[ois]  - Build specified target only
  rpm             - Build a kernel as an RPM package
  tags/TAGS       - Generate tags file for editors
  cscope          - Generate cscope index

Static analysers
  buildcheck      - List dangling references to vmlinux discarded
                    sections
                    and init sections from non-init sections
  checkstack      - Generate a list of stack hogs
  namespacecheck  - Name space analysis on compiled kernel
```

```
Kernel packaging:
  rpm-pkg         - Build the kernel as an RPM package
  binrpm-pkg      - Build an rpm package containing the compiled
                    kernel & modules
  deb-pkg         - Build the kernel as an deb package

Documentation targets:
  Linux kernel internal documentation in different formats:
  xmldocs (XML DocBook), psdocs (PostScript), pdfdocs (PDF)
  htmldocs (HTML), mandocs (man pages, use installmandocs to
  install)

Architecture specific targets (i386):
* bzImage         - Compressed kernel image (arch/i386/boot/bzImage)
  install         - Install kernel using
                     (your) ~/bin/installkernel or
                     (distribution) /sbin/installkernel or
                     install to $(INSTALL_PATH) and run lilo
  bzdisk         - Create a boot floppy in /dev/fd0
  fdimage        - Create a boot floppy image

  make V=0|1 [targets] 0 => quiet build (default), 1 => verbose build
  make O=dir [targets] Locate all output files in "dir", including
                       .config
  make C=1   [targets] Check all c source with $CHECK (sparse)
  make C=2   [targets] Force check of all c source with $CHECK
                       (sparse)

Execute "make" or "make all" to build all targets marked with [*]
For further info see the ./README file
```

Кроме того, каждое доступное свойство ядра документируется в файлах `Kconfig`, которые располагаются в подкаталогах дерева исходных файлов ядра. Они содержат текст справочных сообщений, которые вы будете видеть при создании новой конфигурации ядра с помощью команд `make config`, `make menuconfig` и т. д. Поскольку файлы `Kconfig` являются обычными текстовыми файлами, их можно просматривать в любом текстовом редакторе.

3.6.2. Модули ядра

Операционная система Linux содержит функциональные средства для модулей ядра, позволяющие осуществлять небольшое самодокументирование. Команда `modinfo` дает возможность просматривать информацию в модуле, которую там мог разместить его создатель. Модули могут принимать параметры при загрузке подобно параметрам командной строки. Данные параметры можно просмотреть при помощи команды `modinfo` наряду с любой документацией, наличие которой обеспечил автор. Для большинства модулей объем предусмотренной документации минимален, но для некоторых он может быть довольно большим. Если модуль принимает какой-либо параметр ядра, то его можно увидеть в перечне, выводимом

при помощи команды `modinfo`, независимо от того, документирован он или нет. Даже если документация отсутствует, то вам по крайней мере будет дано направление для поиска в исходных файлах.

3.6.3. Прочая документация

Дистрибутив исходных файлов ядра содержит ценные справочные материалы, которые можно отыскать в каталоге `Documentation`. В данном каталоге находится более 700 файлов, большинство из которых имеют обычный текстовый формат, а небольшая часть — формат DocBook. Исходные файлы DocBook являются частью сборки ядра. Если вы захотите их просмотреть, помните, что существует ряд целей, которые вы можете использовать для генерирования нужного формата:

```
$ make pdfdocs                  # Создание PDF-файлов
$ make mandocs                  # Создание страниц man
$ make psdocs                   # Создание PostScript-файлов
```

В них содержится множество разнообразной информации, которой могут пользоваться как хакеры ядра, так и системные администраторы.

3.7. Заключение

В данной главе вы узнали об основных источниках документации по инструментарию Linux и о том, как ими пользоваться. Мы рассмотрели различные инструменты для извлечения справочных сведений и исследовали ряд источников документации. Помимо источников, нами также были исследованы форматы справочных документов по Linux. В завершении главы мы заострили внимание на ядре и доступной документации для разных модулей ядра и остальной его части.

3.7.1. Инструментарий, использованный в этой главе

- man — оригинальный справочный UNIX-инструмент.
- apropos — позволяет осуществлять поиск по ключевым словам в заголовках man.
- whatis — позволяет осуществлять поиск в базе данных whatis (созданной при помощи makewhatis) страниц руководства man.
- info — справочный GNU-инструмент, поддерживающий комплексные документы.
- yelp — справочный инструмент среды GNOME, который может использоваться также в качестве функционального веб-браузера.
- khelpcenter — справочный инструмент среды KDE.
- evince — универсальный GNU-инструмент для просмотра файлов в форматах DVI, PDF, PostScript и др.

- makeinfo — используется для преобразования файлов Texinfo в другие форматы (HTML, DocBook и пр.).
- gs — интерфейс командной строки для Ghostscript; используется как GNU-средство просмотра документов в формате PostScript.
- xpdf — программа с открытым исходным кодом для просмотра PDF-файлов в системе X Window.

3.7.2. Веб-ссылки

- www.troff.org — ресурс, посвященный языку форматирования `troff`.
- www.pathname.com/fhs — веб-страница стандарта файловой иерархии Filesystem Hierarchy Standard.
- http://marc.theaimsgroup.com — сайт, где располагается архив множества списков рассылки.
- www.foolabs.com/xpdf — домашняя страница проекта `xpdf`.
- www.gnome.org/projects/evince — домашняя страница проекта `evince`.
- pinfo.alioth.debian.org — домашняя страница `pinfo`.

4 Редактирование и сопровождение исходных файлов

4.1. Введение

Текстовый редактор — это инструмент, которым должен хорошо владеть любой уважающий себя разработчик. У многих подобных программ есть свои приверженцы, которые готовы убеждать всех, что именно тот текстовый редактор, которым они пользуются, и является самым лучшим. В этой главе мы разберемся в том, как на самом деле обстоят дела в данной сфере, и постараемся сделать для себя наиболее оптимальный выбор.

Однако редактированием исходных файлов наша задача не ограничивается. Грамотный процесс разработки подразумевает контроль версий исходных файлов. Для этой цели был разработан ряд как бесплатных, так и коммерческих инструментов, при этом их число растет с каждым днем. Мы рассмотрим наиболее распространенные инструменты с открытым исходным кодом, а также имеющиеся альтернативы.

В завершение главы мы изучим инструментарий, который позволяет осуществлять навигацию между исходными файлами и манипулирование ими. Он особенно пригодится тем, кто работает в командах и имеет дело с чужим исходным кодом.

4.2. Текстовый редактор

Возможно, наиболее важным инструментом разработчика программного обеспечения является именно текстовый редактор, который напрямую влияет на вашу производительность и манеру работать. Использование хорошо знакомого и удобного текстового редактора позволяет почувствовать разницу между подходом к написанию программного кода как к развлечению и как к рутинной работе. Из-за этого разработчики часто неравнодушно относятся к выбору текстового редактора. Большинство из них имеют свои убеждения о том, что следует понимать под хорошим текстовым редактором.

По мнению некоторых разработчиков, современный текстовый редактор должен обладать графическим пользовательским интерфейсом и быть частью интегрированной среды разработки IDE (Integrated Development Environment), для того чтобы его можно было плодотворно использовать для создания программного обеспечения. Другие при написании программ предпочитают пользоваться клавиатурой, не прикасаясь к мыши. Таким образом, лучшим текстовым редактором для вас будет тот, который соответствует методам вашей работы.

Несомненно, недостатка в текстовых редакторах не наблюдается. Их существует довольно много. Вы, как компетентный разработчик, должны владеть наиболее популярными из них, для того чтобы быть продуктивным в любой среде. Текстовые редакторы Linux можно разбить на четыре основные категории, которые приведены в табл. 4.1.

Таблица 4.1. Основные категории текстовых редакторов

Текстовый редактор	Описание
vi/Vim	Vi — это самый старый текстовый редактор UNIX, который является также частью стандарта POSIX. Предназначен для людей, умеющих печатать, и позволяет выполнять все необходимые действия при помощи рук, все время находящихся на клавиатуре. Vim является наиболее распространенным клоном редактора vi с открытым исходным кодом, который входит в большинство дистрибутивов Linux
Emacs и его клоны	Emacs является ведущим GNU-редактором с большим количеством прекрасных возможностей. Благодаря своей расширяемости он с годами аккумулировал множество функций, выходящих за рамки простого редактирования текста. Emacs доступен во всех дистрибутивах GNU/Linux
Прочие терминальные клоны	По существу, в данную категорию входят все остальные терминальные редакторы. Некоторые из них являются клонами более старых программ, которые уже не поддерживаются, например WordStar
Текстовые редакторы с GUI-интерфейсом	Объединение в одну категорию всех текстовых редакторов с GUI-интерфейсом может показаться не совсем правильным, однако на то есть веская причина. Такие текстовые редакторы объединяет одна общая черта: все они обладают интуитивно понятным интерфейсом и у них отсутствуют какие-либо режимы. Клавиатура служит для набора текста, а мышь — для всего остального

Вы непременно должны уметь хорошо работать в одном из терминальных редакторов — предпочтительнее в vi или Emacs. Поскольку vi является частью стандарта POSIX, если вы будете отлично владеть им, то сможете продуктивно работать в любой Linux- или UNIX-системе. Emacs входит во все дистрибутивы программного обеспечения с открытым исходным кодом и обычно устанавливается в каждой Linux-системе. Несмотря на то что Emacs содержится в большинстве основных UNIX-клонов, он не всегда инсталлируется в систему при выборе стандартного типа установки.

4.2.1. Важные функциональные возможности текстового редактора

Если задать трем разработчикам вопрос о том, какова самая важная функциональная возможность текстового редактора, то, скорее всего, вы получите три разных ответа. Этим отчасти можно объяснить существование большого количества текстовых редакторов.

Если вы хотите оценить новый текстовый редактор или какой-либо клон с позиции функциональности, прежде всего следует обращать внимание на возможности, «заточенные» специально под разработчиков. Некоторые из них перечислены в табл. 4.2.

Таблица 4.2. Общие функциональные возможности текстовых редакторов

Функциональная возможность	Описание
Сопоставление фигурных скобок	Если поднести курсор к символу фигурной скобки, например {}, () или [], текстовый редактор выделит соответствующую фигурную скобку. Некоторые текстовые редакторы позволяют перемещать курсор к соответствующей фигурной скобке, что тоже удобно. Это особенно полезно при работе с кодом, в котором мало отступов, а также при написании сложных выражений с большим числом круглых скобок
Выделение синтаксиса	Данная функциональная возможность не просто призвана придать красивый вид программному коду. Она также позволяет визуально привлечь ваше внимание ко многим распространенным ошибкам, например комментариям или кавычкам, которые не были закрыты, а также к синтаксическим ошибкам препроцессора
Автоматическое завершение	Данная функция позволяет серьезно экономить время, особенно тем, кто медленно набирает. Если набрать несколько первых букв какого-либо слова, текстовый редактор автоматически подставит оставшиеся буквы, основываясь на уже встречавшихся ранее подобных словах. Некоторые разработчики вообще не могут обходиться без этой функциональной возможности
Регулярные выражения	Регулярное выражение является точным синтаксисом, который используется для поиска и внесения изменений в текст. Это важная функциональная возможность, применяемая для редактирования программного кода, особенно если приходится производить «хирургические» изменения в больших наборах исходных файлов. Использование менее точного синтаксиса может привести к внесению непредвиденных изменений
Автоматический отступ	Данная функциональная особенность воспринимается большинством программистов как должное. Лишние нажатия клавиш, необходимые для установления отступов в программном коде, сказываются на процессе работы, замедляя его. Обычно редактор сам ставит пробелы во время набора, а некоторые текстовые редакторы позволяют переформатировать ранее набранные фрагменты кода с малым числом отступов

Функциональная возможность	Описание
Просмотр программного кода	Инструменты вроде ctags и etags позволяют генерировать индексы для наборов исходных файлов. Текстовый редактор может использовать такие индексы для навигации по программному коду. Редакторы Emacs и vi применяют уникальные форматы для тегов, при этом большинство редакторов, поддерживающих теги, используют один из таких форматов
Сборка программного кода	Возможность сборки программного кода в текстовом редакторе — это не просто удобство, а весьма важный инструмент разработки. Хороший текстовый редактор не только позволяет осуществлять сборку, но и выводит соответствующие предупреждения и помогает отслеживать ошибки

4.2.2. Основные текстовые редакторы: vi и Emacs

Исторически программы vi и Emacs предшествуют созданию UNIX (а не только Linux). Эти терминальные текстовые редакторы использовались еще до появления системы X, когда терминал был единственным интерфейсом взаимодействия с компьютером. Сегодня некоторые программисты не используют терминальные редакторы (иногда называемые *редакторами текстового режима*), поскольку считают их устаревшими. Однако на самом деле это не так. Редакторы vi и Emacs развивались годами, приобретая все функциональные возможности, наличие которых предполагается в современном текстовом редакторе, включая GUI-интерфейс. Тем не менее текстовый режим по-прежнему остается важным: он позволяет работать даже в тех ситуациях, когда GUI-интерфейс недоступен[1].

4.2.3. Vim: усовершенствованная версия vi

Текстовый редактор vi является частью стандарта POSIX и входит в состав всех дистрибутивов UNIX. Однако до недавнего времени он не был приложением с открытым исходным кодом. С годами это привело к появлению его многочисленных клонов, которые имели открытый исходный код. Vim является клоном, который можно найти во всех основных дистрибутивах Linux. Vim (сокращение от *vi Improved — усовершенствованная версия vi*) не только обладает всеми функциональными возможностями редактора vi, но и содержит множество нововведений, в нем также устранены некоторые недостатки vi.

Функциональные возможности текстового редактора Vim

Неудивительно, что редактор Vim обладает функциональным набором, описанным в табл. 4.2, поскольку именно по данной причине этот наиболее распространенный

[1] Если вы никогда не сталкивались с такими ситуациями, то вам предстоит еще многое узнать.

клон vi включается в дистрибутивы Linux. Одно из преимуществ использования текстового редактора с открытым исходным кодом — то, что его пользователями являются преимущественно программисты. Им также пользуются люди, занимающиеся совершенствованием программных продуктов. Именно таким путем и создается качественное программное обеспечение. Это также означает, что если в одном текстовом редакторе появляется какая-либо полезная особенность, то можно с большой вероятностью ожидать, что вскоре она будет реализована и в прочих программах такого рода.

В табл. 4.3 вы можете увидеть сопоставление некоторых функциональных свойств редакторов vi и Vim. На самом деле Vim обладает намного бóльшим набором функций — их слишком много, для того чтобы уместиться в этой таблице. Когда мы будем говорить о функциональных возможностях, которыми обладают как vi, так и Vim, я буду использовать термин vi, а при упоминании свойств, которые присущи только редактору Vim, будет использоваться термин Vim.

Таблица 4.3. Сопоставление текстовых редакторов vi и Vim

Функциональная особенность	**vi**	**Vim**
Отмена последнего редактирования	Одноуровневая	Многоуровневая
Расширение табуляций	Нет	Да
Количество буферов на сессию	Два	Ограничивается только системными ресурсами
Наличие GUI-интерфейса	Нет	Опционально
Выделение синтаксиса	Нет	Да
Автоматическое завершение	Нет	Да

Ключевой особенностью редактора vi является то, что при работе в нем не нужна мышь. Вы можете выполнить любое действие при помощи рук, все время находящихся на клавиатуре, без использования мыши, в отличие от редакторов с GUI-интерфейсом или Emacs, в котором иногда приходится выворачивать пальцы, для того чтобы одновременно нажать три клавиши. Для этого в текстовом редакторе vi используются три основных режима: *командный режим*, *режим вставки* и *режим редактора Ex*. Возможно, такое положение вещей могут любить (или понимать) только программисты. Среди данной категории людей есть как свои почитатели, так и ненавистники того, что касается сущности формы редактора vi.

Режимы, режимы, режимы

Текстовый редактор vi запускается в командном режиме, а это означает «Набирать пока нельзя!». В данном режиме каждая клавиша клавиатуры выполняет определенную функцию. Если нажать какую-либо клавишу, что-то (возможно) произойдет. Командный режим используется для перемещения курсора, операций вырезания, вставки и прочих действий, не связанных с набором текста. Это единственная особенность, которая делает редактор vi отличным от других и отталкивает новых пользователей от его применения. Вполне естественно ожидать, что

после запуска текстового редактора в нем сразу можно набирать. Однако, перед тем как начинать набирать в редакторе vi, нужно переключиться в режим вставки.

На рис. 4.1 проиллюстрированы режимы редактора vi, а также способы переключения между ними.

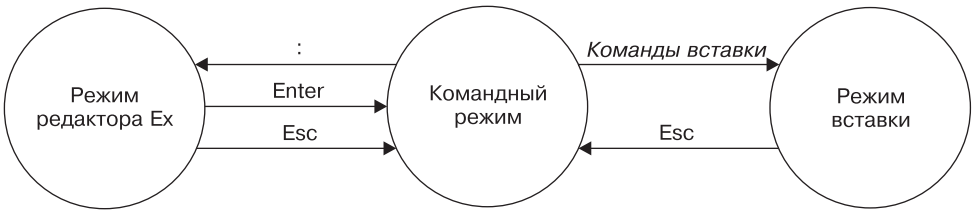


Рис. 4.1. Иллюстрация режимов текстового редактора vi

Обратите внимание на то, что для переключения из командного режима в режим вставки существует много способов, а для выхода из режима вставки только один — нажать клавишу Esc. Также следует отметить, что все пути проходят через командный режим. То есть, например, нельзя из режима вставки напрямую перейти в режим редактора Ex.

Командный режим

Командный режим может заставить понервничать новых пользователей редактора vi. Здесь нет никаких сообщений о состоянии или всплывающих сообщений «Вы действительно хотите?..». Выполнение команды происходит моментально, как только вы нажмете соответствующую клавишу. Например, если в командном режиме нажать клавишу j, курсор переместится на одну строку ниже. Не более. Никаких фанфар или поздравлений — только результат. Для того чтобы привыкнуть к этому, нужно время, но когда вы освоитесь, то заметите, что подобная методика позволяет значительно ускорить работу.

Сочетание клавиш в командном режиме является балансом между «легко запомнить» и «легко использовать». Большинство команд для перехода в режим вставки попадают в категорию «легко запомнить». Сюда относятся команды вроде `i` для операции «вставить» и `a` — для операции «добавить». В то же время команды, связанные с перемещением курсора, могут быть более мудреными, поскольку они призваны быть легкими в использовании. Например, сочетание клавиш jkhl выполняет ту же функцию, что и клавиши со стрелками в командном режиме, которые позволяют перемещать курсор при помощи указательного и среднего пальцев правой руки.

Команды можно повторять, указывая перед ними количество повторений. В данном случае также не выводится никаких сообщений о состоянии или предупреждений. Находясь в командном режиме, просто введите `40j`, и курсор послушно переместится на 40 строк ниже. Для того чтобы привыкнуть к этому, также необходимо время. При ошибочном указании количества повторений может последовать довольно необычная реакция, что часто отпугивает неопытных пользователей редактора vi. К счастью, в редакторе Vim имеется многоуровневая отмена последнего редактирования.

Команды позиционирования курсора

В табл. 4.4 перечислены некоторые основные команды перемещения курсора в командном режиме. Клавиши клавиатуры со стрелками работают, как и предполагается, но на самом деле каждая из них позволяет отдавать уникальную команду. Однако использование клавиш со стрелками требует изменения начальной позиции рук на клавиатуре. Если же воспользоваться сочетанием клавиш jkhl, то позицию можно не менять.

Таблица 4.4. Команды позиционирования курсора, которые могут использоваться отдельно или в сочетании с другими командами

Команда	Описание
j	Перемещение курсора на одну строку вниз
k	Перемещение курсора на одну строку вверх
h	Перемещение курсора на одну колонку влево
l	Перемещение курсора на одну колонку вправо
Enter	Перемещение курсора на одну строку вниз
G	Переход на строку. Если не указано количество повторений, осуществляется переход в конец файла. Если количество повторений указано, то осуществляется переход на соответствующую строку. Например, при вводе 50G осуществляется переход на 50-ю строку файла
+	Перемещение курсора на одну строку вниз с позиционированием его возле первого непустого символа строки
–	Перемещение курсора на одну строку вверх с позиционированием его возле первого непустого символа строки
%	Переход к соответствующей фигурной скобке. Курсор позиционируется у символа фигурной скобки, например (), {} или [], в противном случае терминал подает сигнал и позиционирование не осуществляется
[[	Двухсимвольная команда, которая перемещает курсор обратно к первой фигурной скобке { в первой колонке. Весьма полезна при перемещении между функциями
]]	Аналог [[, за тем исключением, что перемещение осуществляется вперед
`{mark}	Переход на строку, помеченную определенной отметкой при помощи команды m. Отметки определяются пользователем посредством команды m и могут быть любыми символами
``	Переход (возврат) на последнюю строку, с которой осуществлялся переход или поиск. Данная команда особенно полезна, если вы случайно перешли не туда, куда хотели, а также для перехода в какую-либо часть файла и возврата на исходную позицию
/{expr}	Осуществление прямого поиска. Если нажать клавишу /, будет выведено приглашение ввести регулярное выражение для поиска. Поиск начинается после нажатия клавиши Enter. Он стартует с текущей строки и продолжается до первого совпадения. По умолчанию поиск продолжится с начала файла (то есть совершит полный оборот). См. также параметр ws в табл. 4.18

Команда	Описание
?{expr}	Осуществление обратного поиска. Данная команда является аналогом /, за тем исключением того, что поиск осуществляется от текущей строки к началу файла. Здесь также совершается полный оборот и поиск продолжается с конца файла
n	Повторение последнего поиска в том же направлении, начиная с текущей позиции курсора
N	Повторение последнего поиска в противоположном направлении, начиная с текущей позиции курсора
w	Перемещение курсора на одно слово вперед
b	Перемещение курсора на одно слово назад

Все перечисленные команды позиционирования курсора можно с успехом использовать на практике. Как вы еще увидите, все команды перемещения курсора, приведенные в табл. 4.4, могут комбинироваться с другими командами, что позволяет сделать их весьма полезными. Также существует ряд команд позиционирования курсора, которые нельзя комбинировать с прочими командами. Некоторые из них описаны в табл. 4.5.

Таблица 4.5. Отдельные команды позиционирования курсора

Команда	Описание
Ctrl+F	Перемещение на одну экранную страницу вперед
Ctrl+B	Перемещение на одну экранную страницу назад
Ctrl+]	Переход к тегу, на который указывает курсор. Используется с ctags, может выполняться также в режиме редактора Ex с :ta
Ctrl+T	Возврат с предыдущего перехода к тегу (аналог кнопки Back в веб-браузере)

Команды вставки и изменения

Помимо команд позиционирования курсора, наиболее важными командами являются те, которые дают возможность приступить непосредственно к набору текста. Данные команды позволяют переключаться в режим вставки. Существует множество способов перехода в режим вставки, однако большинство программистов выбирают какой-то один и пользуются им в дальнейшем. Неизбежны ситуации, когда предпочтительнее использовать какую-либо другую команду, однако для большинства случаев вполне будет достаточно команды `i`. В табл. 4.6 вы можете увидеть наиболее распространенные команды, используемые для перехода в режим вставки.

Таблица 4.6. Базовые команды для перехода в режим вставки

Команда	Описание
i	Переход в режим вставки, курсор перед символом
a	Переход в режим вставки, курсор после символа

Продолжение ⇩

Таблица 4.6 *(продолжение)*

Команда	Описание
I	Переход в режим вставки, курсор перед первым непустым символом строки
A	Переход в режим вставки, курсор после последнего непустого символа строки
o	Переход в режим вставки, начиная с новой строки под текущей позицией курсора
O	Переход в режим вставки, начиная с новой строки над текущей позицией курсора

Шаблоны, приведенные в табл. 4.8, идентичны шаблонам команд удаления (`d`) из табл. 4.12. В текстовом редакторе Vim единственная разница между изменением (команды приведены в табл. 4.7) и удалением заключается в том, что при операции изменения используется режим вставки. Этим редактор vi отличается от Vim. К примеру, при вводе команды `4s` для изменения следующих четырех символов редактор vi переключится в командный режим после того, как вы закончите набирать четвертый символ. А редактор Vim останется в режиме вставки.

Таблица 4.7. Команды изменения

Команда	Описание
C	Удаление всего, что следует после курсора до конца строки, затем осуществляется переход в режим вставки. Данная команда аналогична команде D (см. табл. 4.10) в сочетании с командой A
c{motion}	Удаление текста, начиная с символа возле курсора в соответствии с аргументом motion, затем происходит переход в режим вставки. Допустимые команды перемещения включают все команды, перечисленные в табл. 4.4
{N}s	Изменение (замена) следующих N символов, начиная с текущего положения курсора. Данная команда аналогична команде c, за тем исключением, что вместо команды перемещения здесь указывается точное количество символов. Команда 5s будет аналогом команды c51
S	Изменение (замена) текущей строки полностью. Данная команда удаляет весь текст текущей строки (за исключением относящегося к новой строке) и выполняет переход в режим вставки

Таблица 4.8. Шаблоны использования команд изменения

Шаблон	Описание
2cw	Удаление двух слов и переход в режим вставки; альтернативный вариант — изменение двух следующих слов
cta	Удаление всего вплоть до следующего появления буквы «a» с последующим переходом в режим вставки. Является комбинацией команды c и команды перемещения ta
5cta	Удаление всего вплоть до пятого появления буквы «a» с последующим переходом в режим вставки
5S	Удаление текущей строки и четырех последующих строк с последующим переходом в режим вставки

Прочие команды, используемые в командном режиме

Вам также необходимо владеть другими командами, которые используются в командном режиме. Пожалуй, самой важной из них будет команда отмены последнего редактирования (u), так как пользователи во время работы в редакторе vi часто допускают ошибки. В редакторе Vim данная функциональная возможность была расширена. Стандартный редактор vi позволяет отменять редактирование, касающееся только последней операции вставки или изменения, в то время как Vim поддерживает многоуровневую отмену последнего редактирования, как и подобает современному текстовому процессору. Если в редакторе vi дважды задействовать отмену последнего редактирования, это приведет лишь к возврату в текст отмененного изменения. Поскольку редактор Vim поддерживает многоуровневую отмену последнего редактирования, в него была добавлена новая команда возврата отмененных изменений в командном режиме — Ctrl+R. В табл. 4.9 приведены эта и прочие команды, которые вы должны знать.

Таблица 4.9. Прочие команды

Команда	Описание
u	Отмена редактирования, касающегося последней операции изменения или вставки. Данная команда может многократно использоваться для последовательной отмены большого числа внесенных изменений
Ctrl+R	Возврат последнего отмененного редактирования (только в редакторе Vim)
m{letter}	Устанавливает закладку на текущей строке посредством задания буквы. Буква может быть любой, но в нижнем регистре в соответствии с текущей локализацией
.	Повтор последней операции изменения или вставки
Ctrl+L	Обновление экрана. Данная команда пригодится в случае, если фоновый процесс искажает выводимые на экран данные
zt	Обновление экрана с размещением текущей строки и курсора в верхней части экрана
zz	Обновление экрана с размещением текущей строки и курсора в средней части экрана
zb	Обновление экрана с размещением текущей строки и курсора в нижней части экрана

Команды вырезания, вставки и удаления

Исторически сложилось так, что в документации к текстовому редактору vi под термином *регистр (register)* понимается то, что мы сейчас называем буфером *(clipboard)*. Несомненно, это является частью его наследия как текстового редактора для программистов. Аналогично современные команды *«вырезать» (cut)* и *«вставить» (paste)* в документации к vi именуются *«вставить в регистр (буфер)» (yank)* и *«поместить» (put)*. К счастью, в этой документации термин для команды *«удалить» (delete)* используется такой же, как и сейчас.

Данные команды отличаются от команд изменения тем, что они не используются в режиме вставки. При вводе любой из команд, приведенных в табл. 4.10, вам

необходимо находиться в командном режиме. Как и большинство команд vi, эти команды также могут повторяться, то есть если, к примеру, ввести 5p, то вставка в редактор содержимого буфера будет выполнена 5 раз.

Таблица 4.10. Команды удаления, вырезания и вставки

Команда	Описание
D	Удаление всего, начиная с текущего положения курсора до конца строки. Данные сохраняются в буфере по умолчанию (регистре)
d{motion}	Удаление определенного количества символов, начиная с текущего положения курсора. Количество символов определяется аргументом motion. Для удаления текущей строки необходимо вводить dd. Данные сохраняются в буфере по умолчанию (регистре). Допустимые команды перемещения включают все команды, перечисленные в табл. 4.4
y{motion}	Копирование определенного количества символов в буфер (вставка в регистр). Количество символов определяется аргументом motion. Для копирования текущей строки в буфер по умолчанию необходимо вводить yy. Допустимые команды перемещения включают все команды, перечисленные в табл. 4.4
p	Вставка (помещение) в буфер символов, начиная с символа, следующего за символом возле курсора
P	Вставка (помещение) в буфер символов, начиная с символа, находящегося перед символом возле курсора

Команды вставки в регистр и удаления могут принимать один аргумент с меткой motion, который сообщает редактору, какой фрагмент текста необходимо удалить или скопировать. Аргументом motion могут быть любая из команд перемещения курсора, приведенных в табл. 4.4, а также дополнительные команды перемещения из табл. 4.11.

Команды, перечисленные в табл. 4.11, могут казаться немного необычными до тех пор, пока вы не станете использовать их на практике. Верите вы или нет, но имена являются мнемоническими. В табл. 4.12 вы найдете ряд базовых шаблонов для команд d и y, которые демонстрируют, как эти команды можно комбинировать с командами перемещения.

Таблица 4.11. Дополнительные команды перемещения

Команда	Описание
f{char}	Позиционирование курсора у первой колонки справа, что соответствует указанному символу
t{char}	Аналог f, за тем исключением, что курсор позиционируется на одну колонку левее соответствующего символа
F{char}	Позиционирование курсора у первой колонки слева, что соответствует указанному символу
T{char}	Аналог F, за тем исключением, что курсор позиционируется на одну колонку правее соответствующего символа

Таблица 4.12. Шаблоны для команд вырезания и вставки

Шаблон	Описание
dfa	Удаление символов, начиная с текущего положения курсора направо, вплоть до первой буквы «a», которая также удаляется. Мнемоническая формулировка: удалять все символы, пока не встретится буква «a»
dta	Удаление символов, начиная с текущего положения курсора направо, вплоть до первой буквы «a», которая удаляться не будет. Мнемоническая формулировка: удалять все символы вплоть до буквы «a»
5yta	Копирование символов, начиная с текущего положения курсора вплоть до пятого появления буквы «a»
yy4p	Копирование текущей строки и вставка в буфер четырех дополнительных копий. В данном случае используются две команды: 1) yy (вставка в регистр текущей строки) и 2) 4p (вставка в регистр по умолчанию четырех копий)
dn	Удаление символов, начиная с текущего положения курсора до первого соответствия самого последнего поиска. В данном случае команда d комбинируется с командой перемещения n
d'a	Удаление символов, начиная с текущего положения курсора вплоть до позиции с отметкой «a». Здесь команда d комбинируется с 'a в качестве команды перемещения
yG	Копирование всех строк, начиняя с текущей строки до конца файла. В данном случае команда y комбинируется с командой перемещения G. Как вы уже знаете, команда G позиционирует курсор в конце файла
y50G	Копирование всех строк, находящихся между текущей и 50-й строкой, включая текущую и 50-ю строки. Здесь команда y комбинируется с командой перемещения 50G
d5l	Удаление следующих пяти символов, начиная с текущего положения курсора (это буква «l» в нижнем регистре, а не единица). Здесь команда d комбинируется с 5l в качестве команды перемещения
5dd	Удаление текущей строки и четырех следующих строк. В данном случае команде dd предшествует количество повторений, равное пяти

В шаблонах, приведенных в табл. 4.12, команды y и d являются взаимозаменяемыми. Шаблоны одинаковы для обеих этих команд, несмотря на то что результаты будут различаться.

Режим редактора Ex

Именно в этом режиме текстовый редактор vi становится немного похожим на монстра доктора Франкенштейна. Ex — это название строчного редактора, на основе которого был создан vi, при этом многие команды, используемые в режиме редактора Ex, были позаимствованы именно из этого оригинального редактора. Трудно представить, что Ex когда-то был текстовым редактором, который люди использовали для работы, но это действительно было. Ex — это строчный редактор, то есть он работает с файлами построчно. Несмотря на то что он напоминает вымершего динозавра, в режиме редактора Ex по-прежнему можно выполнять множество важных действий. Поскольку Ex является полнофункциональным тек-

стовым редактором, многое из того, что возможно в командном режиме, также можно сделать и в режиме редактора Ex. Данный режим используется для ввода комплексных команд, а также команд, принимающих аргументы. В общих чертах, команды Ex относятся к категории «легко запомнить».

На рис. 4.1 видно, что перейти в режим редактора Ex можно, нажав клавишу двоеточия (:). До тех пор пока вы не нажмете клавишу Enter, весь последующий текст будет интерпретироваться как команда Ex. После нажатия Enter команда будет исполнена, а редактор vi вернется в командный режим.

Все команды Ex имеют одинаковую базовую форму: сначала идет опциональный номер строки или диапазон номеров строк, затем следует команда:

```
:[firstline][,lastline]command
```

Большинство наиболее часто используемых команд состоят из одной или двух букв, однако некоторые могут быть и длиннее. Номера строк являются опциональными. Если вы не укажете номер, команда будет применена только к текущей строке. Если указать номер только одной строки, команда будет применена именно к этой строке. Если же вам нужно, чтобы команда затронула весь диапазон строк, нужно указать начальную и конечную строки. Так, например, для того, чтобы удалить строки с 25-й по 30-ю включительно, нужно воспользоваться командой `d` следующим образом:

```
:25,30d
```

Для ввода определенных номеров строк можно также использовать «горячие» клавиши. Некоторые наиболее популярные из них приведены в табл. 4.13.

Таблица 4.13. Горячие клавиши для указания номеров строк в режиме редактора Ex

Символ	Горячая клавиша
.	Номер текущей строки
$	Номер последней строки в файле
%	Горячая клавиша для охвата всего содержимого файла — то же самое, что ввести 1,$
`a	Местоположение тега a. Как вы уже знаете, в командном режиме теги устанавливаются при помощи команды m
/{expr}/	Следующая строка, соответствующая регулярному выражению
?{expr}?	Предыдущая строка, соответствующая регулярному выражению
\/	Следующая за последним регулярным выражением строка
\?	Предшествующая последнему регулярному выражению строка
\&	Следующая за последней подстановкой строка

В табл. 4.14 приведены наиболее важные команды Ex, которыми вам, скорее всего, доведется пользоваться в текстовом редакторе vi. Следует отметить, что почти все команды могут принимать диапазон адресов, независимо от того, имеет он смысл или нет. Например, команда `:w` используется для обновления текущего файла на диске. Эта команда может принимать аргумент для записи текущего содержимого буфера в другой файл, а также диапазон адресов:

```
10,20w foo.dat
```

Данная команда осуществляет запись с 10-й по 20-ю строку в файл под именем `foo.dat`. Не все команды могут принимать адрес или диапазон адресов, однако большинство из них на это способны. Все это может показаться ненужным и необычным для некоторых команд, но все же иногда бывает полезным.

Таблица 4.14. Основные команды редактора Ex

Команда	Краткая форма	Описание
write	:w {filename}	Запись текущего содержимого буфера в файл с указанным именем. Файловое имя является опциональным. Без него текстовый редактор vi обновляет текущий файл на диске. Используйте :w! для того, чтобы форсировать запись в файл, помеченный «только для чтения». :w! позволяет обновлять файлы на диске
quit	:q	Выход из текстового редактора vi (данная команда не сработает, если содержимое буфера не было сохранено). Используйте :q! для того, чтобы форсировать выход из редактора vi и не сохранять внесенные изменения. См. также :e
xit	:x	Выход из текстового редактора vi и сохранение несохраненных данных. Эта команда не сработает, если файл предназначен только для чтения. Используйте :x! для того, чтобы форсировать запись в файл, помеченный «только для чтения». Также допускается использование команды :wq
edit	:e {filename}	Открытие указанного файла в новом буфере. Текущий файл остается открытым. Если имя файла не указано, редактор vi заново открывает текущий файл, не сохраняя каких-либо изменений, но только с вашего предварительного согласия. Если вы решили не сохранять изменения и хотите заново открыть файл, используйте :e!
delete	:d	Удаление текущей строки или диапазона строк
map	:map {a} {b}	Переназначение клавиш, используемых в командном режиме. Без аргументов данная команда использует текущие параметры
set	:set {argument}	Изменение настроек по умолчанию текстового редактора Vim (см. табл. 4.18). Данная команда наиболее полезна в файле .vimrc
help	:help	Вход в справочную систему редактора Vim. Данная команда может принимать аргумент ключевого слова и позволяет находить соответствующую информацию. Справочная система организована в виде гиперссылок с использованием тегов vi, а это означает, что вы должны быть в некоторой степени компетентны в редакторе vi, для того чтобы найти нужные справочные сведения в редакторе Vim

Что интересно, многие команды, которые поддерживает редактор Ex, также совместимы с командой sed (потоковый редактор). Все, что вы почерпнули из данного раздела, может пригодиться вам в процессе работы со сценариями.

Усовершенствованные функциональные возможности текстового редактора Vim в режиме вставки

При работе в режиме вставки клавиатура используется как обычно. Вы нажимаете клавиши, и текст появляется. Все просто, почти. Именно здесь на передний план выходят некоторые значительные преимущества редактора Vim по сравнению с vi. Vim позволяет вводить в режиме вставки некоторые команды, которые нельзя использовать, если вы работаете в vi. Их количество слишком велико для того, чтобы приводить в книге, но в табл. 4.15 вы сможете ознакомиться с наиболее полезными из них.

Таблица 4.15. Команды, поддерживаемые редактором Vim в режиме вставки

Команда	Описание
Ctrl+N/Ctrl+P	Завершение слова на основе слов, ранее набранных в документе. В отличие от большинства редакторов, программе Vim не требуются какие-либо буквы, для того чтобы «догадаться», какое слово желает ввести пользователь, хотя это может помочь сузить рамки поиска. Для того чтобы ввести следующее слово, нужно снова нажать Ctrl+N, а чтобы вернуться к предыдущему — Ctrl+P
Ctrl+T/Ctrl+D	(Данная команда также поддерживается редактором vi). Текущую строку можно сдвинуть вправо или влево посредством параметра shiftwidth. По умолчанию сдвиг осуществляется на 8 колонок, однако этот показатель можно изменять при помощи команды set shiftwidth
Ctrl+R	Вставка в текущий документ содержимого одного из специальных регистров, используемых редактором Vim. Для просмотра имеющихся регистров нужно ввести :help registers. Имя каждого регистра определяется одним произвольным символом. К ним, например, относятся: символ % для текущего файлового имени, а также . (точка) для самой последней вставки
Ctrl+V	(Данная команда также поддерживается редактором vi). Ввод непечатаемых символов, например управляющих символов, которые в иной ситуации могут интерпретироваться как команда vi. Использовать управляющие символы в сценариях не рекомендуется

Поиск и замена

Несмотря на возможность выполнять поиск в командном режиме, команды типа «найти и заменить» можно задействовать только в режиме редактора Ex. Все потому, что данный режим является единственным, который поддерживает длинные аргументы, а регулярные выражения в операциях поиска и замены могут быть довольно длинными.

Базовой командой Ex для выполнения замен в тексте является команда `substitute`, которая может применяться в сокращенной форме — `subst` или, что бывает чаще, `s`. Как и все прочие команды Ex, она может принимать номер строки или диапазон номеров строк с любыми сокращениями, упомянутыми в табл. 4.13. Базовая команда `substitute` выглядит следующим образом:

```
:s/search/replacement/flags
```

Единственным необходимым аргументом будет строка поиска. Параметры `replacement` и `flags` являются опциональными. Если параметр `replacement` не определен, строка поиска заменяется пустой строкой. Все аргументы должны разделяться знаком прямого слеша (`/`), что может стать проблемой, если вы имеете дело с файловыми именами. Например, редактор vi требует, чтобы все прямые слеши *заменялись* знаками обратного слеша, что очень неудобно. По этой причине в редакторе `vi` команда замены с указанным путем будет выглядеть так:

```
:s/\/usr\/bin\/file1/\/usr\/bin\/file2\//
```

Просмотр и сборка программного кода

Текстовый редактор Vim работает в сочетании с программой `ctags`, которая создает индекс исходных файлов, считываемый Vim. Запуск `ctags` осуществляется просто:

```
$ ctags -R
```

После этого она автоматически распознает исходные файлы, располагающиеся в текущем каталоге и подкаталогах, и создает один индексный файл с именем `tags`. Именно этот файл ищет редактор Vim при запуске. Стандарт POSIX предполагает, что `ctags` и редактор vi должны уметь работать с файлами, созданными на языке C и Fortran, однако программа `ctags` из пакета Exuberant, входящая в состав большинства дистрибутивов Linux, может индексировать исходные файлы, созданные на многих других языках, включая C++, Java и Python.

Просматривать программный код в текстовом редакторе vi можно в командном режиме или в режиме редактора Ex. В табл. 4.16 представлены команды, с помощью которых выполняется данная процедура. Некоторые из них уникальны для редактора Vim, функциональные возможности которого, связанные с просмотром программного кода, были значительно усовершенствованы по сравнению с редактором vi. Традиционный vi позволяет делать только базовые переходы и не поддерживает множественные теговые соответствия, которые возможны в коде на языке C++, где используются перегрузка функций или пространства имен. Следует отметить, что Vim поддерживает теговый «стек», то есть вы можете возвращаться обратно после совершенных переходов, подобно тому как если бы вы нажали кнопку Back в веб-браузере.

Таблица 4.16. Команды для просмотра кода

Команда	Краткая форма	Описание
Ctrl+]		Переход к тегу возле курсора

Продолжение ⇩

Таблица 4.16 *(продолжение)*

Команда	Краткая форма	Описание
:tag name	:ta	Переход к указанному тегу. Если тег не указан, переход выполняется к тегу возле курсора
Ctrl+T		Возврат от текущего тега к самой последней точке перехода
:pop	:po	(Поддерживается только в редакторе Vim). Аналог Ctrl+T, за тем исключением, что вы можете задавать количество переходов через несколько уровней посредством одной команды
:tnext	:tn	(Поддерживается только в редакторе Vim). Переход к следующему соответствию, если тег генерирует более одного соответствия, например перегруженную функцию C++. На теговый стек это не влияет
:tprevious	:tp	(Поддерживается только в редакторе Vim). Аналог :tnext, за тем исключением, что переход осуществляется к предыдущему соответствию
:tselect name	:ts	(Поддерживается только в редакторе Vim). Данная команда выводит перечень соответствующих тегов, из которого вы можете выбирать, если тег генерирует более одного соответствия
:tags		(Поддерживается только в редакторе Vim). Эта команда позволяет увидеть текущий теговый стек, где каждому тегу отводится отдельная строка

В версии редактора Vim с графическим пользовательским интерфейсом GUI имеется ряд кнопок, которые позволяют выполнять большинство функций из табл. 4.16, что делает его еще более похожим на веб-браузер.

Возможность чередовать сборку и редактирование программного кода является одной из ключевых особенностей интегрированной среды разработки IDE. Многие разработчики считают ее важнейшим рабочим инструментом. Редактор Vim не претендует на роль среды IDE, однако обладает набором функций, позволяющих одновременно заниматься как редактированием, так и сборкой кода. Вместо того чтобы пытаться охватить весь проект сразу, как это делает интегрированная среда разработки IDE, редактор Vim сначала потребует от вас файл `Makefile`. Затем в Vim вам будет необходимо вызвать `make` при помощи команды `:make`, а это уже не кажется столь удобным. Однако если вы поступите именно так, редактор Vim сохранит вывод компилятора и позволит вам подробно остановиться на каждой строке исходного кода, в отношении которой выводились сообщения об ошибке или предупреждения. Для сборки кода необходимо использовать команду `:make` в режиме редактора Ex:

```
:make arguments
```

Команду `:make` можно использовать как вызов `make` из оболочки. Если потребуется, вы можете указать дополнительные флаги или цели. После ее ввода редактор Vim сохранит все предупреждения и сообщения об ошибке, которые позволят вам отыскать все недочеты в своем исходном коде. Команды, используемые при этом,

приведены в табл. 4.17. Важно отметить, что все они поддерживаются редактором Vim. POSIX-редактор vi не позволяет использовать команду `make`, а также прочие команды из табл. 4.17.

Таблица 4.17. Команды для сборки кода, поддерживаемые редактором Vim

Команда	Краткая форма	Описание
:make arguments	:mak	Запуск make в текущем каталоге и сбор сообщений об ошибках и предупреждений
:cnext	:cn	Переход к строке исходного кода, которой касается следующее сообщение об ошибке или предупреждение в последней сборке
:cprev	:cp	Переход к строке исходного кода, которой касается предыдущее сообщение об ошибке или предупреждение в последней сборке
:cfile filename	:cf	Считывание списка сообщений об ошибках из указанного файла для обработки посредством :cnext и :cprev. Данная команда является альтернативой использованию make

Модификация настроек текстового редактора vi

Многие настройки, отвечающие за работу vi, можно устанавливать в режиме редактора Ex при помощи команды `:set`. Исторически сложилось так, что vi считывает персонализированные настройки пользователя из файла `.exrc`, который располагается в домашнем каталоге, а Vim извлекает их из файла `.vimrc`. Если в системе присутствуют сразу оба этих файла, Vim игнорирует `.exrc` и считывает только файл `.vimrc`. Данное разграничение особенно важно, если вы пользуетесь разными клонами редактора vi. Предполагается, что все подобные клоны будут обращаться за настройками в файл `.exrc` и, скорее всего, ни один из них, за исключением Vim, не станет использовать файл `.vimrc`. В такой ситуации рекомендуется размещать настройки редактора vi в файле `.exrc` и использовать команду `source` из файла `.vimrc` для считывания настроек, содержащихся в `.exrc`:

```
:so ${HOME}/.exrc
```

В дополнение к этой строке в файле `.vimrc` также можно разместить команды, которые понимает только редактор Vim. В табл. 4.18 содержится перечень некоторых важных настроек, которые можно модифицировать в файлах `.vimrc` и `.exrc`.

Таблица 4.18. Настройки, модифицируемые пользователем

Настройка	Краткая форма	Практический пример	Описание
tabstop	ts	set ts=4	Определение количества колонок в табуляции (по умолчанию 8). Это влияет на то, как отображается или преобразуется текст, содержащий табуляции

Продолжение ⇩

Таблица 4.18 *(продолжение)*

Настройка	Краткая форма	Практический пример	Описание
shiftwidth	sw	set sw=4	Определение количества колонок, на которое осуществляет сдвиг при использовании команд отступа (по умолчанию 8). Следует отметить, что данный параметр не зависит от настройки tabstop
autoindent	ai	set ai	Включение или отключение автоматического отступа: ai — включен, noai — отключен (по умолчанию отключен)
expandtabs	et	set et	Не вставлять табуляции, вместо этого использовать число пробелов, определяемое настройкой tabstop. По умолчанию используются жесткие табуляции (ASCII-код \011)
wrapscan	ws	set ws	Изменение порядка поиска: если данная настройка активирована (по умолчанию), при прямом поиске будут выявляться соответствия в предыдущих строках, а при обратном поиске — в последующих строках. В соответствии с данной настройкой поиск выполняется как в командном режиме, так и в режиме редактора Ex. Если она выключена, прямой поиск осуществляется только с текущей строки и до конца файла. Обратный поиск стартует с текущей строки и продолжается до начала файла
syntax	sy	sy on	(Только для редактора Vim). Включение или отключение выделения синтаксиса. Обратите внимание на то, что в данном случае команда set не используется
makeprg	mp	set mp=ant	(Только для редактора Vim). Позволяет выбирать программу, альтернативную make, для выполнения сборок
errorformat	efm	set efm=%f\ %d	(Только для редактора Vim). Позволяет определять для Vim scanf-подобную строку, которая используется для разбора сообщений об ошибках, выдаваемых компилятором. Полное описание данной настройки можно найти в справочной системе редактора Vim, для чего нужно ввести :help efm

Обратите внимание на то, что некоторые настройки устанавливаются при помощи команды `:set`, а часть настроек сами являются командами (например, `:syntax`).

Режим графического пользовательского интерфейса GUI

Текстовый редактор Vim может иметь GUI-интерфейс, обычно выступающий как отдельная программа, которая в среде GNOME называется `gvim`, а в KDE — `kvim`[1].

[1] Что интересно, несмотря на то что многие приложения KDE и GNOME могут мирно сосуществовать, одновременная установка kvim и gvim невозможна.

GUI-интерфейс — это замечательное нововведение и хороший способ освоить редактор vi. Если вы отрицательно относитесь к режимам, но все-таки желаете пользоваться редактором Vim, используйте безрежимный параметр `gvim` (`-y`). Он позволяет превратить Vim в типичный редактор с графическим пользовательским интерфейсом GUI, в котором отсутствуют какие-либо режимы.

Каждое меню, доступное в режиме графического пользовательского интерфейса GUI, содержит подсказки эквивалентных команд редактора vi. Они помогут вам освоить команды, с которыми вы незнакомы. Поскольку в меню приводятся клавиши, с помощью которых команды можно вводить, не используя GUI-интерфейс, они могут стать отличным справочным пособием. Задача по поиску пакета, необходимого для установки GUI-интерфейса, может оказаться далеко не простой и зависит от используемого вами дистрибутива. Из табл. 4.19 вы сможете узнать, какие пакеты доступны для сред KDE и GNOME.

Таблица 4.19. Названия пакетов, содержащих GUI-интерфейс для редактора Vim

Дистрибутив	Пакет
Knoppix (KDE)	vim-gtk
Ubuntu (KDE)	kvim
Ubuntu (GNOME)	vim-gnome
Fedora (GNOME)	vim-X11

Заключительные положения о текстовом редакторе Vim

Давайте вспомним, какие функциональные возможности текстовых редакторов описывались в табл. 4.2, и посмотрим, как их можно использовать на практике. В табл. 4.20 данные функции приводятся наряду с информацией о том, каким образом к ним можно получить доступ.

В текстовом редакторе Vim функциональный набор намного более широк. В справочных меню Vim можно получить исчерпывающую информацию обо всех его возможностях. Данные сведения содержатся в теговых документах, предназначенных только для чтения, которые можно открыть в командном режиме вводом такой команды: `:help` *ключевое слово*. К сожалению, для навигации по справочной системе Vim необходимы определенные знания редактора vi и тегов. Для того чтобы закрыть справочную страницу, нужно ввести команду `:q`.

Таблица 4.20. Доступ к основным возможностям редактора Vim

Функциональная возможность	Способ доступа
Сопоставление фигурных скобок	Находясь в командном режиме, необходимо ввести %
Выделение синтаксиса	Данная возможность обычно активирована по умолчанию. Вы можете вручную активировать или деактивировать ее в режиме редактора Ex вводом команды :syn on или :syn off соответственно
Автоматическое завершение	Необходимо нажать Ctrl+N/Ctrl+P в режиме вставки

Продолжение ⇩

Таблица 4.20 *(продолжение)*

Функциональная возможность	Способ доступа
Регулярные выражения	Доступны при выполнении поиска в командном режиме, а также посредством команды substitute в режиме редактора Ex
Автоматический отступ	Активируется и деактивируется в командном режиме при помощи :set ai и :set noai соответственно
Просмотр программного кода	В командном режиме для перехода к тегу, находящемуся в тексте возле курсора, нужно нажать Ctrl+], а для возврата с предыдущего перехода к тегу — Ctrl+T

4.2.4. Текстовый редактор Emacs

Emacs является ведущим текстовым редактором проекта GNU. Став альтернативой редактору vi, он приобрел своих преданных последователей. Благодаря тому что Emacs содержит обработчик сценариев, основанный на языке программирования Lisp, пользователи (те из них, кто владеет языком Lisp) могут программировать расширения. Поскольку основную категорию пользователей Emacs составляют программисты, данный редактор с годами превратился в своего рода песочницу для разработчиков, аккумулировав многие функциональные возможности, которые мало или вообще не ограничиваются простым редактированием текста. Если вам необходим именно такой функциональный набор, то редактор Emacs — это то, что вам нужно.

Функциональные особенности редактора Emacs

Как вы могли догадаться, Emacs поддерживает все функции, приводившиеся в табл. 4.2, однако отыскать их не так-то просто. Наряду с базовыми возможностями, Emacs обладает набором инструментов для манипулирования исходным кодом, которые вы вряд ли встретите где-то еще. В подавляющем большинстве это команды оболочки, которые были интегрированы в данный редактор.

Режимы? Какие режимы?

Emacs заявлялся как редактор *без режимов*, которые противники редактора vi считают его самым большим недостатком. Поскольку Emacs по своей природе является терминальным редактором, отсутствие режимов было всего лишь иллюзией. Данный редактор имеет ряд режимов. Разница заключается в том, что обычно используемые Emacs режимы являются временными, то есть такими, в которые вы переходите, когда вводите команду, которой могут потребоваться дополнительные аргументы или взаимодействие с вашей стороны. Когда задача выполнена, вы возвращаетесь в режим по умолчанию. В Emacs режим по умолчанию варьируется в зависимости от типа редактируемого файла.

Если вы являетесь пользователем vi, то можете считать, что Emacs постоянно находится в режиме вставки. Командный режим или режим редактора Ex здесь отсутствует. Вместо этого Emacs полагается на комбинации клавиш с использова-

нием Ctrl и метаклавиши на клавиатуре[1]. Подобная методика также не лишена недостатков, так как клавиши управления соответствуют символам ASCII, некоторые из которых отвечают за важные функции. Например, Ctrl+G соответствует ASCII-символу BEL (\007), который заставляет терминал подавать сигнал. Можете убедиться в этом сами — нажмите Ctrl+G, находясь в окне терминала. В редакторе Emacs Ctrl+G используется для отмены последовательности команд.

Метаклавиши также создают проблемы, если вы будете использовать терминальное окно GUI-интерфейса, например gnome-terminal, поскольку в вашем терминале некоторые метаклавиши могут использоваться для других целей. gnome-terminal иногда «перехватывает» определенную метаклавишу, позволяя пользователю с помощью клавиатуры входить в GUI-меню, а это делает Emacs непригодным для использования в текстовом режиме[2]. Если вы открываете Emacs в отдельном GUI-окне, то с такими проблемами не столкнетесь.

О Emacs вам следует знать еще одно: режим по умолчанию может изменяться в зависимости от типа редактируемого файла. Вас как программиста, скорее всего, заинтересует *режим CC*, в котором Emacs позволяет редактировать файлы, созданные на C, C++, Java и прочих языках. Режим CC позволяет задействовать функцию автоматического отступа, при этом он довольно сложен, из-за чего снабжен страницей info. Редактор Emacs по умолчанию запускается в режиме CC, когда определяет, что редактируемый вами файл содержит исходный код. По сравнению с редактором vi, где существует лишь легкий намек на функцию автоматического отступа, режим CC позволяет задействовать ее в полной мере. Данный режим отслеживает, чтобы вы не отклонялись от выбранного стиля отступа. О поддерживаемых стилях можно узнать на странице info режима CC, а также из табл. 4.21.

Таблица 4.21. Стили автоматического отступа, используемые редактором Emacs

Стиль	Описание
gnu	Стиль, по умолчанию используемый Emacs, «благословленный» фондом свободного программного обеспечения Free Software Foundation
K&R	Стиль, используемый в примерах Kerninghan и Ritchie
bsd	Также известен как стиль Allman; схож с K&R
whitesmith	Основан на стиле, использующемся в примерах, созданных посредством компилятора C Whitesmith — коммерческого приложения, применявшегося в PDP-11
Stroustrup	Стиль C++, используемый Stroustrup
ellemtel	Назван в честь компании Ellemtel Telecommunication Systems Laboratories, которая его создала
linux	Стиль, используемый в ядре Linux
python	Стиль, применяемый для написания расширений C для языка Python
java	Стиль, используемый в программном коде на языке Java

[1] На платформе PC к метаклавишам относится Alt.

[2] Этого можно избежать, отключив горячие клавиши клавиатуры, присутствующие меню Edit (Правка) в gnome-terminal.

Emacs по умолчанию использует стиль gnu, который можно встретить в исходном коде, написанном при помощи этого редактора. Однако данный стиль не очень широко распространен, и вы, скорее всего, захотите выбрать какой-то другой. О том, как это сделать, мы поговорим в следующем разделе.

Команды и горячие клавиши редактора Emacs

Для ввода команд в Emacs по большей части используются непечатаемые последовательности клавиш. Сюда относятся клавиши Ctrl, Alt и Esc. Для документирования команд, приведенных в табл. 4.22, я буду пользоваться условной системой, принятой в редакторе Emacs. Обратите внимание на то, что в документации Emacs клавиша Alt считается метаклавишей. В некоторых устаревших системах клавиатура платформы PC не применяется, и клавиша Alt там отсутствует. Использование клавиши Esc отличается от использования Ctrl и метаклавиш, поскольку при нажатии Esc получается символ ASCII (\033). Для сравнения: при нажатии клавиш Ctrl и Alt никакие символы не генерируются. Поэтому при вводе команд, в которых используется клавиша Esc, ее нужно нажимать дважды. Следует отметить, что в системах, где нет метаклавиш (или подобные клавиши там отведены для других целей), вместо них можно использовать клавишу Esc. Например, M-x будет эквивалентом Esc x.

Каждая команда Emacs имеет собственное имя, при этом большинство из них поддерживают «горячие» клавиши. Любую команду можно выполнить, нажав Esc x или M-x и введя имя соответствующей команды. Из-за того что имена команд являются описательными, они получаются довольно длинными. Например, клавиша Backspace отведена под команду `delete-backward-char`. Если же вы не ищете легких путей, то вместо нажатия Backspace можете воспользоваться такой последовательностью клавиш:

```
M-x delete-backward-char
```

Таблица 4.22. Условная система, используемая в Emacs для документирования команд

Нотация	Описание
C-character	Удерживать клавишу Ctrl при нажатии клавиши с указанным символом
M-character	Удерживать клавишу Alt (также называемую метаклавишей) при нажатии клавиши с указанным символом
Esc character	Нажать клавишу Esc, затем нажать клавишу с указанным символом или последовательность управления; является аналогом M-character

В дальнейшем я буду приводить команды в основном в виде их горячих клавиш, особенно это касается команд, которые вы вряд ли когда-нибудь станете набирать вручную (например, `delete-backward-char`).

При нажатии M-x редактор Emacs переходит в так называемый режим *минибуфера*. В таком режиме Emacs позволяет избежать лишнего набора посредством применения функции табуляционного завершения. Нужно набрать несколько первых букв имени какой-либо команды и нажать клавишу Tab. Если будет обнаружено однозначное соответствие, редактор Emacs сам завершит команду, и вам оста-

нется нажать Enter. Если будет выявлено несколько возможных соответствий, редактор выведет их список. Благодаря тому что в режиме мини-буфера также сохраняется история команд, нажав M-x, а затем используя клавишу со стрелкой вверх или вниз, вы можете просматривать историю команд, которые совсем недавно вводили.

Позиционирование курсора

Как и редактор vi, Emacs позволяет пользователю перемещать курсор, не меняя начальной позиции рук на клавиатуре. Основные способы перемещения курсора приведены в табл. 4.23.

Если вы пользуетесь PC-клавиатурой, то для позиционирования курсора также можно задействовать клавиши курсора, равно как и клавиши Page Up и Page Dn. Редактор Emacs также дает возможность повторять команды определенное количество раз. Для выполнения повторения в Emacs нужно воспользоваться предыдущей командой, нажав при этом C-u и указав количество повторений. Приведенная далее последовательность позволяет переместить курсор на пять символов вправо:

```
C-u 5 M-f
```

Подобный шаблон применим ко всем командам Emacs.

Таблица 4.23. Основные способы перемещения курсора в редакторе Emacs

Клавиши	Перемещение
C-b	На одну колонку влево (мнемоника: назад)
C-f	На одну колонку вправо (мнемоника: вперед)
C-n	На одну колонку вниз (мнемоника: далее)
C-p	На одну колонку вверх (мнемоника: назад)
C-v	Вниз на одну экранную страницу
M-v	Вверх на одну экранную страницу
M-f	Вправо на одно слово (мнемоника: вперед)
M-b	Влево на одно слово (мнемоника: назад)

Операции удаления, вырезания и вставки

В силу преклонного возраста Emacs в нем не используются современные термины, описывающие такие функции, как *«поместить в буфер»*, *«вырезать»*, *«копировать»* и *«вставить»*, хотя все они присутствуют в этом редакторе. Как и в редакторе vi, в Emacs применяется другая терминология. Базовая операция *«вырезать»* в Emacs называется *«удалить» (kill)*, а операция *«вставить» (paste)* — *«вставить в регистр» (yank)*, например: *«вставить в регистр текст из буфера» (yank text from the clipboard)*. Те, кто знаком с редактором vi, заметят, что в Emacs команда *«вставить в регистр» (yank)* используется совсем не так, как в vi.

Для того чтобы вырезать или скопировать какой-либо фрагмент текста, сначала нужно установить отметку. Для этого необходимо переместить курсор в один конец фрагмента и отметить его при помощи команды C-@. Другой конец фрагмента определяется положением курсора при вызове соответствующей команды —

«вырезать» или *«копировать»*. Основные команды, используемые для такой операции, приведены в табл. 4.24.

Таблица 4.24. Основные команды вырезания и вставки в редакторе Emacs

Клавиши		Действие
C-@	C-Space	Установка отметки для определения фрагмента текста, который будет удаляться или копироваться при помощи соответствующих команд
C-k		Вырезание (удаление) текста, начиная с текущего положения курсора до конца строки
M-k		Вырезание текста, начиная с текущего положения курсора и до конца предложения
C-w		Вырезание фрагмента текста, начиная с текущего положения курсора до отметки, установленной при помощи команды C-@
ESC w	C-Ins	Копирование фрагмента текста, начиная с текущего положения курсора до отметки, установленной при помощи команды C-@
C-y		Вставка (вставка в регистр) текста из буфера, начиная с текущего положения курсора
C-_	C-x u	Отмена последнего редактирования

Поиск и замена

В текстовом редакторе Emacs порядок поиска имеет две базовые формы, которые описываются в табл. 4.25. Следует отметить, что здесь не используются регулярные выражения. Для того чтобы задействовать регулярное выражение, перед вводом команды поиска необходимо нажать клавишу Esc.

Таблица 4.25. Основные команды поиска и замены в редакторе Emacs

Клавиши	Перемещение
C-s	Поиск вперед с нахождением точных соответствий (без использования регулярных выражений)
C-r	Обратный поиск с нахождением точных соответствий (без использования регулярных выражений)
Esc C-s	Прямой поиск с использованием регулярного выражения
Esc C-r	Обратный поиск с использованием регулярного выражения

Просмотр и сборка программного кода при помощи редактора Emacs

Для просмотра программного кода в Emacs используются те же принципы, что и в редакторе vi. Пользователь создает индекс своих исходных файлов при помощи утилиты наподобие `etags`, которая входит в состав пакета Exuberant Ctags. `etags` берет список файлов, которые необходимо индексировать, и создает в текущем каталоге индексный файл с именем `TAGS`. Редактор Emacs использует его для нави-

гации по программному коду. Некоторые из команд, используемых при просмотре кода в Emacs, приведены в табл. 4.26.

Еще одной полезной функцией Emacs является закладка, которая позволяет отмечать какое-либо место в текстовом файле с использованием особого имени. Например, чтобы установить закладку с именем *review*, нам потребуется следующая команда:

```
C-x r m review
```

Имя закладки *review* будет сохранено, для того чтобы пользователь мог перейти к ней в будущем. Чтобы сделать это, потребуется такая команда:

```
C-x b m review
```

Таблица 4.26. Команды, используемые при просмотре программного кода в редакторе Emacs

Клавиши		**Перемещение**
M-.	Esc-.	Переход к тегу. Данная команда приглашает пользователя ввести тег. Для того чтобы перейти к тегу, находящемуся возле курсора, нужно нажать Return
M-*	Esc-*	Возврат от текущего тега к последней точке перехода
C-u M-.		Поиск следующего тега, альтернативного последнему тегу (например, перегруженным функциям C++)
C-u – M-.		Возврат к предыдущему обнаруженному альтернативному тегу
C-x r name		Установка закладки с указанным именем на месте текущего положения курсора
C-x b name		Переход к закладке с указанным именем

Закладки сохраняются на жестком диске компьютера. Если закрыть редактор Emacs и открыть его снова, он будет «помнить» обо всех закладках, которые вы установили ранее.

Меню текстового режима

Команды Emacs тяжело не только запоминать, но и набирать. Последовательности клавиш, которые необходимо нажимать при вводе команд, могут представлять особую трудность для людей, страдающих от заболеваний, связанных с постоянным напряжением конечностей, например синдрома запястного канала. Если использовать GUI-версию Emacs, то подобные проблемы обойдут вас стороной, поскольку меню графического пользовательского интерфейса GUI позволяют вводить большинство команд при помощи мыши. Новые версии редактора Emacs также поддерживают работу с меню в текстовом режиме. Для того чтобы вызвать меню, нужно нажать на клавиатуре клавишу F10, после чего вам будет предложено сделать новый ввод. Не так удобно, как работать с мышью, но все-таки интуитивно более понятно, если сравнить все это с длинными командами, которые нужно запоминать наизусть, да и вашим запястьям будет легче. На рис. 4.2 вы можете увидеть, как выглядят меню в текстовом режиме редактора Emacs.

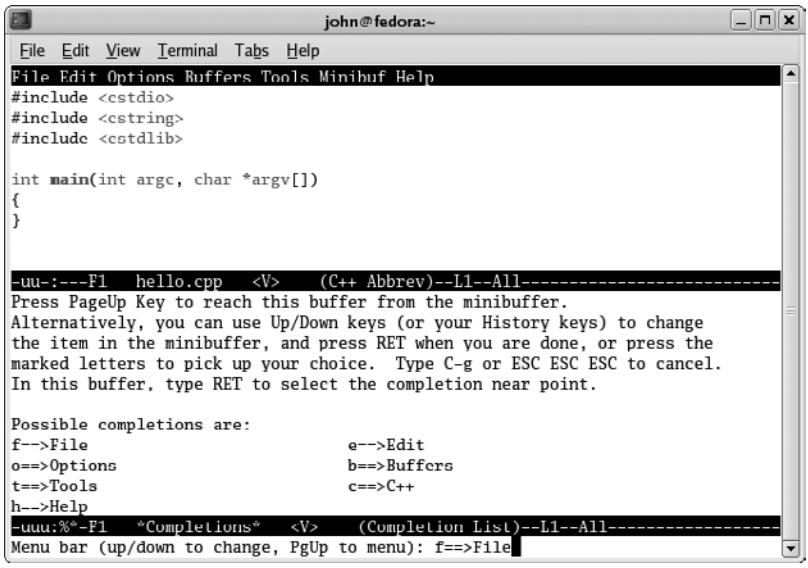


Рис. 4.2. Редактор Emacs в текстовом режиме с использованием меню

Модификация настроек текстового редактора Emacs

Одним из недостатков редактора Emacs является то, что для работы с ним требуются некоторые навыки программирования на языке Lisp. Это означает, что, если вы захотите модифицировать самые простые настройки Emacs, вам придется пользоваться синтаксисом Lisp. При запуске Emacs ищет расположенный в домашнем каталоге файл под именем `.emacs`. Данный файл содержит операторы и функции Lisp, которые можно использовать для изменения настроек по умолчанию редактора Emacs.

Изучение Lisp выходит за рамки настоящей книги, однако на страницах `info` редактора Emacs, а также на интернет-ресурсах можно найти множество материалов на эту тему. Рассмотрим простой пример, наглядно демонстрирующий процесс модификации настроек на языке Lisp и возникающие при этом трудности. Вы увидите, что необходимо сделать для того, чтобы изменить стиль отступа по умолчанию, который, скорее всего, вы первым делом и захотите сменить. Мы выберем вместо него стиль `K&R`, после чего определим количество колонок в табуляции — четыре пробела (вместо восьми по умолчанию) и активируем выделение синтаксиса. В данном случае файл `.emacs` будет выглядеть следующим образом:

```
1. (defun my-c-style ()
2.   (c-set-style "k&r")
3.   (turn-on-font-lock)
4.   (setq c-basic-offset 4))
5.
6. (add-hook 'c-mode-common-hook 'my-c-style)
7. (setq indent-tabs-mode nil)
```

В начале данного файла определяется функция под именем `my-c-style`, которую мы будем вызывать при переходе в режим CC. Первая строка этой функции, располагающаяся на строке 2, изменяет стиль на `k&r`. Строка 3 активирует выделение синтаксиса `font-lock` в редакторе Emacs. И наконец, в строке 4 определяется значение для `c-basic-offset`, которое будет означать количество пробелов в каждом отступе. Данное значение используется функцией `my-c-style`, которая сопровождается заключительными круглыми скобками. Строка 6 инсталлирует `my-c-style` как функцию *перехвата*, вызываемую всякий раз, когда редактор Emacs будет переходить в режим CC. Это глобальная настройка, поэтому ее можно установить отдельным выражением. Строка 7 содержит еще одно отдельное выражение, которое отключает использование табуляции в отступах в программном коде, определяя значение `indent-tabs-mode` как `nil`.

Режим графического пользовательского интерфейса GUI

В состав большинства основных дистрибутивов Linux входит GUI-версия текстового редактора Emacs (технически называемая *Xemacs*), которая используется по умолчанию. Xemacs стал ответвлением редактора Emacs, однако в настоящий момент обе эти программы объединены в одну. Располагая GUI-интерфейсом, Emacs по-прежнему поддерживает текстовый режим, однако он не перейдет в него, если вы не воспользуетесь параметром `-nw` *(no windows — «без окон»)* (также можно ввести `emacs-nox`). Следует отметить, что, даже если у вас не запущен сервер X, данная версия Emacs будет пытаться использовать GUI-режим и, вместо того чтобы перейти в текстовый режим, завершит свою работу выводом сообщения об ошибке.

GUI-режим функционирует аналогично текстовому режиму Emacs. Поддерживаются те же команды, однако, если у вас возникают проблемы с их запоминанием, всегда можно воспользоваться мышью и меню. Emacs можно было бы легко спутать с любым другим GUI-редактором, если бы он походил на окружение Рабочего стола, в котором работает. GUI-версия Emacs не столь совершенна, как прочие современные GUI-редакторы, о которых мы поговорим позднее.

4.2.5. Атака клонов

У редакторов Emacs и vi есть много приверженцев, однако даже самые преданные последователи могут заострять внимание на некоторых недостатках своего любимого редактора. При создании инструментария для программистов всегда существует вероятность того, что кто-то может справиться с этой задачей лучше.

К созданию клонов текстового редактора vi подтолкнуло следующее: несмотря на то что он являлся частью стандарта POSIX, до недавнего времени он не был свободным продуктом. Это означало, что тем, кто хотел использовать vi в операционной системе, отличной от UNIX, требовалась альтернатива. В результате появился ряд клонов vi, среди которых наиболее удачным стал редактор Vim. Его, помимо Linux, поддерживают множество других платформ. У большинства других клонов vi отсутствуют функциональные возможности Vim, однако они хорошо совместимы с оригинальным vi. К ним относятся такие редакторы, как Elvis, Vile

и Nvi. Уникальным в своем роде и любопытным стал двоичный клон редактора vi под названием bvi. Он позволяет просматривать и редактировать файлы с двоичными данными с использованием интерфейса vi. Здесь двоичные данные представляются в виде шестнадцатеричных байтов, при этом вы можете использовать привычные команды vi для навигации и модификации этих данных. Пример окна редактора bvi можно увидеть на рис. 4.3.

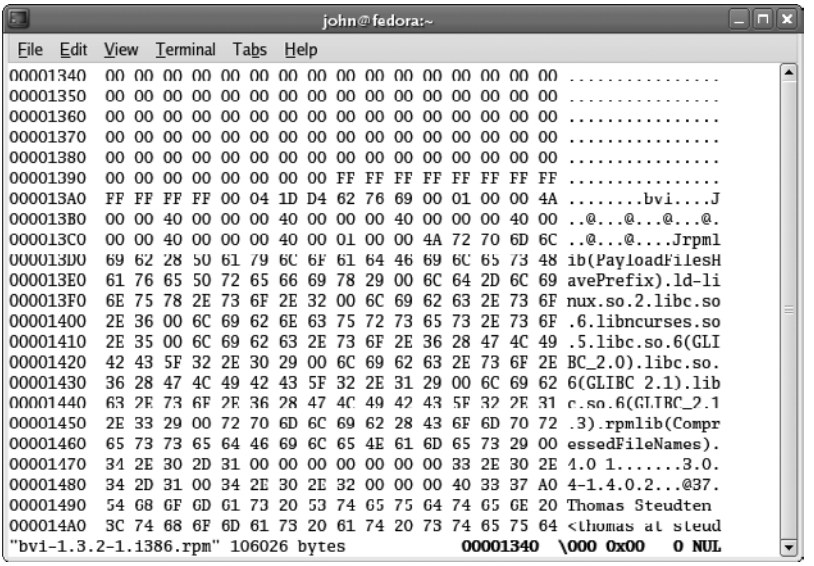


Рис. 4.3. Пример окна редактора bvi

Мотивом «клонирования» Emacs является недостаток, связанный с объемом памяти, который занимает данный редактор. Объем получается слишком большим. Поскольку Emacs является расширяемым приложением, нет смысла создавать его клоны, которые будут обладать большей функциональностью. Проще расширить его посредством программ на языке Lisp. Поэтому нет ничего удивительного в том, что все клоны Emacs являются его «облегченными» версиями. Рядовой пользователь обычно довольствуется интерфейсом Emacs и работает в среде с небольшим объемом памяти, например на встраиваемой системе.

Редактор Pico был частью довольно популярного почтового клиента Pine, который использовался в UNIX-системах до начала эры веб-браузеров. Поскольку Pico входил в состав почтового клиента, он, как вы уже догадались, весьма прост в использовании. Этот редактор устраивал многих пользователей, несмотря на то что у него отсутствовали многие возможности, необходимые для программирования. Тем не менее он пользовался популярностью у программистов. Вы можете загрузить оригинальную версию Pine с сайта Вашингтонского университета[1] либо GNU-клон под названием Nano. Участники проекта GNU решили создать клон Pico, поскольку фонд свободного программного обеспечения Free Software

[1] См. сайт www.washington.edu/pine.

Foundation постановил, что исходная лицензия Pico не соответствует лицензии General Public License (GPL). Клон Nano обладает тем же интерфейсом, что и Pico, но с некоторыми доработками. В табл. 4.27 представлен ряд популярных клонов. Все они, за исключением Vim, работают только в текстовом режиме (у них отсутствует GUI-интерфейс). В следующем разделе мы кратко рассмотрим некоторые текстовые редакторы с GUI-интерфейсом. Из табл. 4.28 вы сможете узнать, какими функциональными возможностями обладает каждый из клонов. Их перечень перекликается с табл. 4.2.

Таблица 4.27. Популярные клоны текстовых редакторов

Редактор	Эмулируемый редактор	Примечание
Vim	vi	Содержит множество нововведений по сравнению с vi
Joe	Emacs, Pico, WordStar	Эмуляция зависит от имени команды. Команды joe и jstar эмулируют WordStar; jmacs эмулирует Emacs, а jpico — Pico. Данные команды указывают на одинаковый исполняемый файл
Zile	Emacs	Zile означает Zile Is Lossy Emacs (Zile — «урезанный» Emacs). Поскольку редактор Zile не имеет текстовых меню, он больше подходит для опытных пользователей Emacs
Jed	Emacs, WordStar и др.	Эмуляция определяется настройками в файле .jedrc. Так как редактор Jed использует текстовые меню, которые одинаковы во всех режимах, он подходит для начинающих пользователей
Nano	Pico	Усовершенствованный GNU-клон текстового редактора Pico

Таблица 4.28. Функциональные возможности, поддерживаемые клонами текстовых редакторов

Редактор	Протестированная версия	Сопоставление фигурных скобок	Выделение синтаксиса	Автоматическое завершение	Регулярные выражения	Автоматический отступ	Просмотр программного кода	Сборка программного кода
Vim	6.3.71	Да	Да	Да	Да	Да	Да	Да
Joe	3.1	Да	Да	—	Да	Да	Да	Да
Zile	2.2	Да	—	—	Да	Да	—	—
Jed	0.99.16	Да	Да	Да	Да	Да	Да	Да
Nano	1.2.4-3	—	Да	—	Да	Да	—	—

4.2.6. Потребление памяти

В некоторых системах объем занимаемой программой памяти является довольно значимым фактором. Если вы работаете на настольном компьютере, который располагает сотнями мегабайт памяти, то, скорее всего, вас мало будет интересовать, сколько памяти потребляет ваш текстовый редактор. Однако на

встраиваемых системах с небольшим объемом памяти, медленным процессором и отсутствием диска подкачки проблема эффективного расходования памяти стоит весьма остро.

На рис. 4.4 вы сможете увидеть данные о потреблении памяти каждым из редакторов, которые мы рассматривали ранее. Как и следовало ожидать, редакторы с GUI-интерфейсом являются лидерами по потреблению памяти, а наименее требовательными к ней оказываются терминальные редакторы.

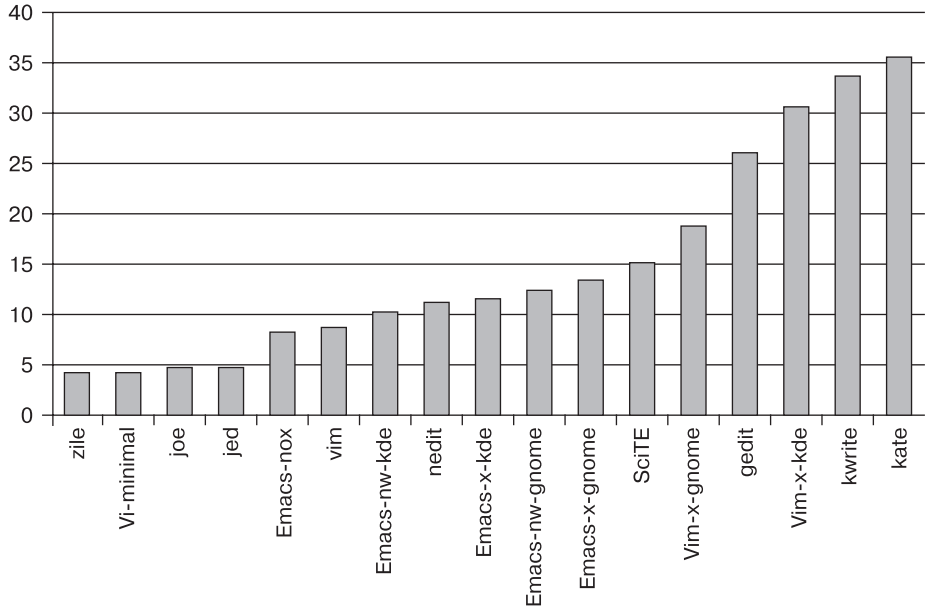


Рис. 4.4. Уровень потребления памяти текстовыми редакторами

4.3. Контроль версий

Любой грамотный процесс разработки подразумевает осуществление контроля версий. Это ключевой показатель для любой компании-разработчика, которая стремится к высокому статусу. Возможность контроля, связанная как с релизом программного продукта, так и с его последующим существованием, жизненно важна для разработки качественного программного обеспечения.

Контроль версий необходим не только крупным компаниям, но и разработчикам-одиночкам. Он может показаться рутинным занятием, однако зачастую это не так, поскольку это важный инструмент, используемый в процессе разработки программ. С точки зрения редактирования текста контроль версий можно рассматривать как функцию *суперотмены внесенных изменений (super undo)*. Она позволяет фиксировать определенные этапы процесса разработки, на которых достигается стабильность функциональности продукта, перед тем как будут реализованы другие функции, которые могут повлиять на весь проект в целом.

4.3.1. Основы контроля версий

Иллюстрация создания версии модуля для проекта с открытым исходным кодом приведена на рис. 4.5. Версии от A до F составляют основную ветвь, где новые функциональные особенности реализуются и подвергаются отладке. На этапе версии C вы решаете выложить свой программный модуль для всеобщего доступа. Пока пользователи привыкают к новому функционалу, вы продолжаете его расширять в версиях от D и до F. Время от времени, пока вы занимаетесь разработкой, пользователи выявляют ошибки, которые необходимо устранить. Для того чтобы исправить эти ошибки без релиза продукта с недоработанным функционалом, вы создаете ответвление от версии C. Это позволяет вам устранить недочеты в уже выпущенном в обращение модуле, одновременно работая над новыми функциональными возможностями. В версии F вы *объединяете* внесенные исправления со своим программным продуктом, что является обычной практикой перед релизом его новой версии. Приведенный на рис. 4.5 пример немного упрощен, однако по своей сути он будет общим для всего инструментария, используемого для контроля версий.

До недавнего времени существовало не так уж много инструментов с открытым исходным кодом, которые использовались для контроля версий. Сейчас все изменилось и для работы над новыми проектами стали доступны современные, хорошо проработанные инструменты. Наиболее популярные из них вы можете увидеть в табл. 4.29.

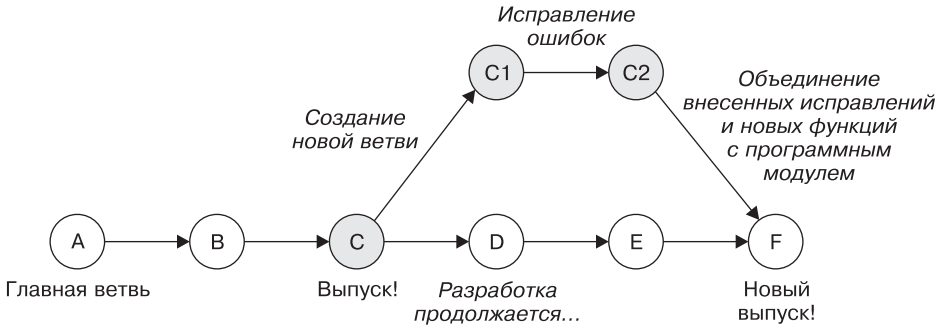


Рис. 4.5. Простой пример «ветвления» версий программного продукта

Таблица 4.29. Популярные инструменты контроля версий

Название	Описание
RCS	Система контроля версий Revision Control System является предком системы CVS (Concurrent Version System) и легла в ее основу. Система RSC не поддерживает проекты и требует, чтобы файлы при внесении в них изменений блокировались
CVS	Система контроля версий Concurrent Version System основана на системе RCS, позволяет группировать файлы в проектах и не требует их блокировки при модификации. Разработчики должны разрешать конфликты, возникающие при объединении, перед тем как смогут внедрять изменения

Продолжение ⇩

Таблица 4.29 *(продолжение)*

Название	Описание
Subversion	Преемник системы CVS, который устраняет многие ее недостатки с сохранением базового пользовательского интерфейса
GNU arch	Альтернативный системе CVS инструмент, который появился примерно в одно время с Subversion. Служит в основном для решения тех же задач, что и Subversion. Возможно, в будущем обретет популярность
monotone	Данный инструмент применяет новый подход к контролю версий, который может показаться спорным: использует хеши SHA1 для регистрации изменений и версий

4.3.2. Терминология в сфере контроля версий

Поскольку терминология варьируется в зависимости от используемого инструмента, я охарактеризую ряд нейтральных определений, позволяющих сравнивать инструменты между собой.

Проект

Обычно под проектом понимается произвольное группирование файлов по усмотрению разработчика, однако на практике часто оказывается, что один проект генерирует одиночный исполняемый файл или библиотеку. Например, каждый инструмент с открытым исходным кодом обладает собственным «проектом» контроля версий.

Любой файл проекта имеет самостоятельную историю, но и сам проект также может иметь историю. В инструментах CVS и Subversion проект называется *модулем*, в GNU arch — *архивом*, а в monotone — *рабочим экземпляром*.

Добавление/Удаление

Данная функциональная возможность является базовой. Разработчикам необходимо иметь возможность добавлять и удалять исходные файлы по мере развития своего проекта. Все инструменты позволяют делать это. Благодаря тому что большинство инструментов сохраняют историю удаленных исходных файлов, они могут быть восстановлены в виде старых версий.

Постановка на учет

Эта функциональная возможность позволяет создавать новые версии определенного файла. Каждый раз, когда вы ставите на учет файл, вы получаете моментальный снимок того, как данный файл выглядит в этот момент. И неважно, сколько изменений вы в него внесли, потому что этот файл всегда можно воспроизвести точно в том виде, в котором он был при постановке на учет.

Снятие с учета

Данная функциональная возможность позволяет «отзывать» файл, который был поставлен на учет ранее. В зависимости от используемого инструмента вы

можете снимать с учета как один файл, так и весь проект целиком. Большинство инструментов по умолчанию используют безблокировочную схему снятия файлов с учета, а блокировки применяются только при определенных обстоятельствах. Если файл блокирован, то никто другой не сможет с ним работать (снять его с учета).

Ветвь

Ветвление является важным инструментом контроля версий. Оно предоставляет в ваше распоряжение множество способов работы с разными версиями программного продукта. Один из них, связанный с устранением найденных ошибок, наглядно продемонстрирован на рис. 4.5. Ветви позволяют создавать новые версии старых исходных файлов, которые не будут конфликтовать с их более новыми собратьями, а также не станут аккумулировать нежелательные изменения, которые не готовы к релизу.

Объединение

Данная функциональная возможность позволяет инструментам наподобие системы контроля версий CVS работать без применения блокировок. Вместо этого используются операции объединения. Идея заключается в том, что, когда вы вносите изменения в определенный файл, соответствующий инструмент проверяет, не внес ли кто-либо другой свои изменения в этот файл с того момента, как вы приступили к его редактированию. Если подобные изменения будут обнаружены, перед постановкой файла на учет необходимо объединить внесенные в него модификации.

Если вы вместе с другим разработчиком снимаете с учета один и тот же файл и при этом он закончит редактирование раньше вас, то вам потребуется выполнить объединение внесенных вами обоими изменений, перед тем как сохранить их в файле. Данная операция осуществляется при помощи инструмента объединения и средства контроля версий. Если вы не произведете объединение, ваши изменения перезапишут модификации, внесенные другим разработчиком.

Метка

Инструменты контроля версий обычно используют произвольные метки для версий файлов, поставленных на учет в системе. Если у вас имеется несколько исходных файлов, то высока вероятность того, что они имеют разные версии. Инструменты контроля версий используют метки 1.1, 1.2 и т. д. Например, инструмент monotone использует 40-символьные хеши. Поскольку они являются произвольными, вы, скорее всего, не захотите применять их в работе с релизными версиями. Метка позволяет присваивать группе файлов обычное имя вроде `release_1.0`, благодаря чему вы можете отслеживать все файлы, которые попадают в релиз, при помощи одной клавиши.

4.3.3. Инструменты сопровождения

Управление изменениями в исходном коде является непростым занятием, которое лишь усложняется по мере вовлечения в проект новых разработчиков. Поэтому неудивительно, что для облегчения этой работы был создан ряд инструментов.

4.3.4. Команды diff и patch

Большинство пользователей знакомы с базовой командой `diff`, которая позволяет сравнивать файлы парами. Зачастую пользователя интересует лишь то, являются ли какие-либо два файла идентичными. В данном случае нужно пользоваться командой `cmp`. При необходимости наглядно увидеть различия, выводимые при помощи команды `diff`, сведения далеко не дружественны к пользователю. По этой причине основное предназначение команды `diff` заключается в создании «заплат» (патчей), которые накладываются при помощи команды `patch`. Как вы увидите в дальнейшем, существуют более оптимальные средства просмотра различий в файлах, чем команда `diff`, однако в данный момент мы рассмотрим то, для чего она хорошо приспособлена. Начнем мы с примера вывода команды `diff`, где описываются небольшие различия в файлах:

```
$ cat -n before.txt
     1 This is a line to be deleted
     2 This is a line that will be changed
     3 This is a line that will be unchanged

$ cat -n after.txt
     1 This is a line that has been changed
     2 This is a line that will be unchanged
     3 This is a line that has been added

$ diff before.txt after.txt
1,2c1
< This is a line to be deleted
< This is a line that will be changed
---
> This is a line that has been changed
3a3
> This is a line that has been added
```

Несмотря на то что данный вывод не особенно дружествен пользователю, он более удобочитаем. По умолчанию строки, которые были подвергнуты изменению или удалению из первого файла, помечены символом <. Строкам, которые были добавлены или остались неизменными во втором файле, предшествует символ >. Дополнительные сведения касаются номеров строк, подвергнутых изменению, которые могут использоваться командой `patch`.

Давайте посмотрим, как данный вывод может быть использован командой `patch`. Допустим, у нас имеется файл, идентичный файлу `before.txt`, на который мы хотим наложить «заплату» (назовем его `new.txt`). Мы хотим увидеть различия между файлами `before.txt` и `after.txt` и наложить их на файл `new.txt`. Это можно сделать с использованием «заплаты» следующим образом:

```
$ cp before.txt new.txt
$ diff before.txt after.txt > mypatch.txt
$ cat mypatch.txt | patch new.txt
```

Другое полезное свойство «заплат» заключается в том, что они обратимы. Для того чтобы отменить изменения, внесенные в файл new.txt при помощи команды patch, можно воспользоваться аналогичной «заплатой» следующим образом:

```
$ patch -R new.txt < mypatch.diff
```

Флаг -R указывает команде patch наложить изменения в обратном направлении, или *отменить* «заплату». При использовании вывода команды diff по умолчанию команде patch необходимо точно указать файлы, на которые необходимо наложить «заплату».

Наиболее простой способ создания большой «заплаты» заключается в использовании команды diff с параметром -r для рекурсивного обхода всех каталогов. Для этого нам потребуются два идентичных дерева каталогов с одинаковыми именами файлов. Единственное различие будет состоять в самих файлах. Используя соответствующие пути, команда diff способна определить, какие пары файлов необходимо сравнить, для того чтобы вывести на экран выявленные различия. Когда команда diff встречает файл, существующий только в одном из деревьев, она по умолчанию пропускает его и отправляет предупреждение на stderr. Такое поведение можно изменить при помощи параметра -N, который заставит команду diff считать, что отсутствующий файл на самом деле присутствует, но пуст. Таким образом, «заплата» сможет охватить созданные нами файлы. Затем ее наложение приведет к созданию новых файлов.

4.3.5. Просмотр и объединение изменений

Команда diff оставляет желать лучшего, когда речь заходит о просмотре изменений. Можно спорить, является ли ее вывод удобным для человеческого восприятия, однако если это касается незначительных изменений, то он вполне адекватен. GNU-команда diff имеет множество параметров, позволяющих сделать вывод более читабельным, но при использовании текстового терминала ваши возможности в этом плане будут ограничены.

Для просмотра большого числа изменений более удобен форматированный вывод, из которого легко понять, что именно подверглось изменению. Когда команда diff видит, что в какой-либо строке изменился только один символ, она выводит всю эту строку (дважды), указывая на внесенное изменение, например:

```
$ diff src1 src2
1c1
< const char *somechars=":,-;+.({)}";
---
> const char *somechars=":,-;+.{()}";
```

В данной строке изменению подверглись только два символа. Вы можете понять, какие именно? Здесь нам придется прибегнуть к помощи других инструментов. Например, текстовый редактор Vim способен показывать различия путем выделения изменений отдельных символов:

```
$ vim -d src1.c src2.c
```

Альтернативой, немного более удобной в использовании, является GUI-версия этого редактора — gvimdiff, пример которой приведен на рис. 4.6. Здесь наглядно показано, как выделяются изменения отдельных символов в дополнение к изменениям целой строки. Теперь различия намного проще заметить. Обе программы, Vim и gvimdiff, выделяют изменения, однако из-за ограниченных возможностей текстового терминала предпочтительнее использовать gvimdiff.

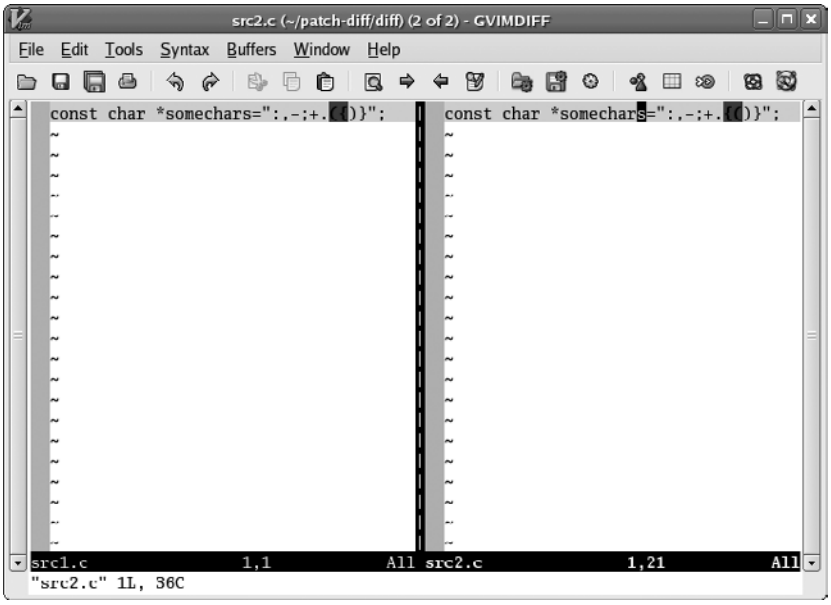


Рис. 4.6. Демонстрация имеющихся различий в программе gvimdiff (выделенный символ s — это курсор)

Еще одним инструментом с открытым исходным кодом, обладающим GUI-интерфейсом, является программа xxdiff[1], доступная для загрузки на сайте sourceforge.net. Данный инструмент имеет ряд дополнительных функциональных возможностей, включая утилиту для осуществления объединений. GUI-интерфейс особенно пригодится при пользовании данной утилитой, в чем вы сами скоро убедитесь.

Со временем процесс разработки все более усложняется и объединение становится необходимостью. Объединение наиболее часто требуется, если в процессе работы используется механизм контроля версий. Обычно потребность в объединении возникает, когда вы работаете с другими членами команды разработчиков или имеете дело с большим количеством ветвей. Для того чтобы вы лучше смогли понять суть объединения, давайте рассмотрим инструмент командной строки, с помощью которого выполняется данная операция.

GNU-средством объединения, используемым инструментами контроля версий наподобие CVS и Subversion, является diff3. Команда diff3 получила свое название из-за того, что ей требуются три аргумента файловых имен:

---

[1] См. сайт http://sourceforge.net/projects/xxdiff.

```
$ diff3 myfile original yourfile
```

Порядок расположения *myfile* и *yourfile* можно изменять, однако второе файловое имя должно быть общим предком. На рис. 4.7 приведена диаграмма, демонстрирующая внешний вид дерева версий.

Для демонстрации нам потребуется файл, с которым мы будем работать. Рассмотрим следующий простой пример:

```
1 void foo(void)
2 {
3         printf("Изменения в данный файл внесу я.\n");
4
5         printf("Данный файл останется неизменным.\n");
6
7         printf("Изменения в данный файл внесете вы.\n");
```

Теперь допустим, что я внес изменения в свой экземпляр данного файла и строка 3 стала выглядеть так:

```
printf("Изменения в данный файл внес я.\n");
```

А вы изменили свой экземпляр файла, и строка 7 теперь выглядит так:

```
printf("Изменения в данный файл внесли вы.\n");
```

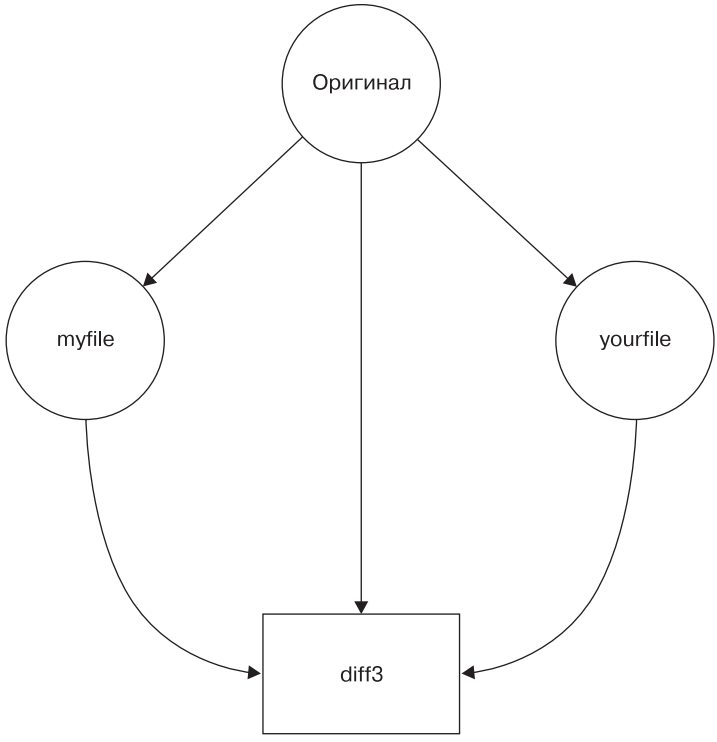


Рис. 4.7. Графическая иллюстрация процесса объединения при помощи команды diff3

Теперь у нас есть два разных изменения, которые необходимо объединить. К счастью, они затрагивают разные строки одного и того же файла, поэтому объединение не потребует особых усилий. Без аргументов команда `diff3` генерирует вывод, демонстрирующий изменения, однако он будет удобочитаемым, только если данные изменения являются незначительными, как показано в следующем примере:

```
$ diff3 me.c orig.c you.c
====1
1:3c
        printf("Изменения в данный файл внес я.\n");
2:3c
3:3c
        printf("Изменения в данный файл внесу я.\n");
====3
1:7c
2:7c
        printf("Изменения в данный файл внесете вы.\n");
3:7c
        printf("Изменения в данный файл внесли вы.\n");
```

Различия разделяются при помощи символов `====1` или `====3` с указанием того, какой из модифицированных файлов стал источником отличий от первоначального файла. Так как нумерация зависит от порядка аргументов, в приведенном примере 1 — это `me.c`, 2 — `orig.c`, а 3 — `you.c`. Номера строк, так же как и типы изменений, указываются слева. Однако данный вывод команды `diff` является далеко не самым лучшим. Более читабельный вывод генерируется в том случае, когда для выполнения объединения задействуется команда `diff3`. Поскольку в нашем примере изменения являются незначительными, `diff3` генерирует простой вывод:

```
$ diff3 -m me.c orig.c you.c | cat -n
1 void foo(void)
2 {
3         printf("Изменения в данный файл внес я.\n");
4
5         printf("Данный файл останется неизменным.\n");
6
7         printf("Изменения в данный файл внесли вы.\n");
8 }
```

Обратите внимание на то, что в данном выводе изменения, которые были внесены мною и вами, отражаются справа. Очень часто при работе с большими исходными файлами возможны мелкие процедуры объединения, которые не требуют от пользователя ввода определенных команд, однако иногда все бывает далеко не так просто.

Давайте еще раз взглянем на инструмент `xxdiff`, который может оказаться весьма полезным при осуществлении объединений. На рис. 4.8 вы можете увидеть, как данный инструмент используется для выполнения этой задачи.

Здесь представлены три файла, и при помощи мыши вы можете сами выбрать, какие изменения нужно принять. Можно даже выбрать все три изменения, что при-

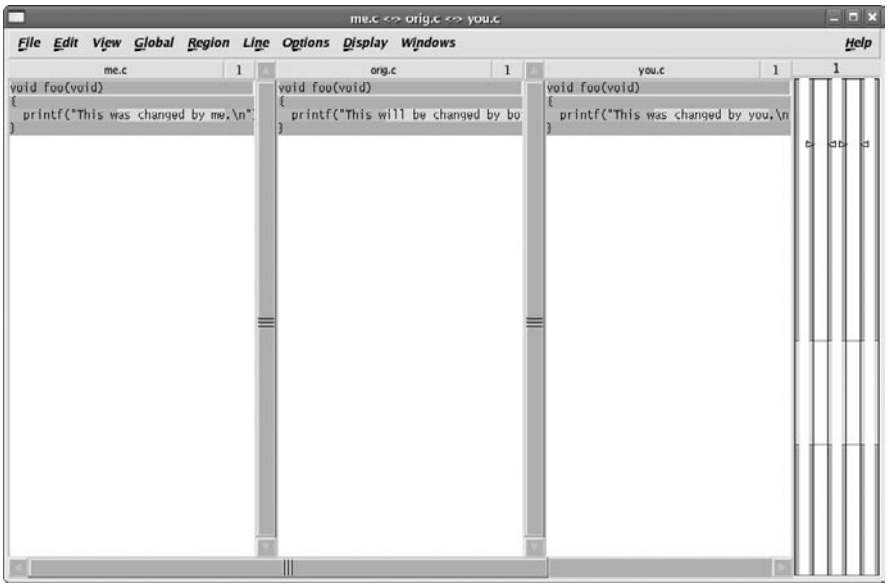


Рис. 4.8. Использование xxdiff для выполнения объединения

ведет к генерации такого же вывода, как после использования команды `diff3`. Либо вы можете дать указание `xxdiff` использовать оператор `#ifdef` в качестве обертки для всех изменений, например:

```
 1      void foo(void)
 2      {
 3      #if defined( ME )
 4         printf("Изменения в данный файл внес я.\n");
 5      #elif defined( ORIG )
 6        printf("Изменения в данный файл мы внесем сообща.\n");
 7       #elif defined( YOU )
 8        printf("Изменения в данный файл внесли вы.\n");
 9       #endif
10       }
```

Что касается системы CVS и большинства инструментов контроля версий, то здесь необходимо отметить, что у пользователя, выполняющего объединение, есть возможность выбора. Это означает, что если я объединяю изменения наряду с использованием механизма контроля версий, у меня будет возможность выбрать, какие именно изменения вступят в силу: ваши или мои.

4.4. Инструменты для улучшения внешнего вида исходного кода и его просмотра

Мы уже обсуждали, какие жесткие правила задания отступа существуют в режиме CC редактора Emacs и то, как их сложно нарушить. Если бы все вокруг пользовались редактором Emacs и одинаковым стилем отступа, вопросов не возникало

бы. Но в реальной жизни пользователи отдают предпочтение разным текстовым редакторам с личными настройками. Чем больше людей занимаются корректированием одного и того же исходного файла, настраивая при этом редактор по-своему, тем больший беспорядок в итоге возникает в программном коде, содержащемся в этом файле. Некоторые редакторы преобразуют табуляции в пробелы; другие смешивают их между собой; при этом все они по-разному определяют количество пробелов в табуляции. То, что выглядит нормально в одном редакторе, может иметь авангардный вид в другом.

Некоторые из существующих инструментов позволяют автоматически расставлять отступы в программном коде, однако здесь следует учитывать один важный момент: переформатирование всего модуля целиком может привести к проблемам с механизмом контроля версий. Все из-за того, что при этом затрагиваются все строки кода без исключения. Даже если вы всего лишь осуществляете перестройку кода, инструмент, используемый для выполнения объединений, не будет этого знать. Если изменения вносятся в *«некрасивую»* версию файла, то попытка объединить эти изменения с «красивой» версией может оказаться довольно неуклюжей.

4.4.1. Инструмент для улучшения внешнего вида программного кода indent

В UNIX применяется команда `cb`, позволяющая переформатировать исходный код на языке C. Она была реализована в виде фильтра, который мог работать только со стандартным вводом и выводом. Это вызывало недовольство у некоторых пользователей, поскольку этот фильтр нельзя было свободно применять к своему исходному коду. Подобный подход имеет некоторые преимущества, особенно для пользователей редактора vi, так как он поддерживает использование фильтров. Например, вы можете отформатировать код, заключенный в фигурные скобки, следующим образом:

```
!%cb
```

Использование инструментов для улучшения внешнего вида программного кода имеет еще одну вескую причину. Допустим, что «улучшенный» код из предыдущего примера нам необходимо объединить с «необлагороженным» кодом. Фильтрация блоков кода позволяет вносить инкрементные изменения в большой модуль, с которым могут работать множество пользователей. Вместо переформатирования всего модуля целиком, приводящего к возникновению конфликтов, вы можете подправить какую-либо одну функцию или блок кода и избежать многих проблем, когда дело дойдет до объединения.

В Linux эквивалентом фильтра `cb` является команда `indent`, которая более универсальна в применении. Прежде всего она способна переформатировать код, написанный как на языке C++, так и на C. Она может функционировать как фильтр подобно `cb`, а также позволяет переформатировать файлы на месте. Несмотря на то что делать это не рекомендуется, вы можете переформатировать группу файлов при помощи одной команды, приведенной далее:

```
$ indent *.c
```

Хотя команда `indent` и не поддерживает все стили, используемые редактором Emacs (они приводились в табл. 4.21), она совместима со стилями K&R, GNU и BSD. Любым аспектом переформатирования можно управлять посредством более чем 80 параметров командной строки, благодаря чему вы можете настроить свой любимый стиль таким образом, чтобы команда `indent` стала поддерживать его.

4.4.2. Художественный стиль astyle

Еще одним перспективным инструментом для улучшения внешнего вида программного кода является `astyle`[1]. Как и `indent`, `astyle` «понимает» языки C и C++, а также Java и (что удивительно) C#. Данный инструмент не поддерживает все форматы, с которыми совместим редактор Emacs, однако имеет предопределенные форматы для стилей K&R, GNU и Linux, включая так называемый стиль ANSI.

4.4.3. Анализ программного кода при помощи cflow

Если вам придется работать с программным кодом, который вы не создавали либо с которым мало знакомы, простого его изучения не всегда может оказаться достаточно для того, чтобы понять его. К счастью, здесь нам на помощь приходит ряд инструментов.

POSIX-команда `cflow` преобразует исходный код в диаграмму, позволяющую увидеть обзор программного потока. Это очень удобно, если вы имеете дело с незнакомым кодом. Участники проекта GNU создали собственную версию POSIX-команды `cflow`[2], которая хотя и не полностью соответствует POSIX, однако может вполне успешно использоваться.

4.4.4. Анализ программного кода при помощи ctags

Если мы заговорили о перекрестных ссылках, то давайте вновь обратимся к пакету Exuberant Ctags. Несмотря на то что `ctags` обычно формирует вывод для текстового редактора, данная утилита также может генерировать удобочитаемые перекрестные ссылки, если задействован параметр `-x`. Воспользовавшись предыдущим примером еще раз, мы увидим такой вывод:

```
$ ctags -x ex4-5.c
afunc           function 5   ex4-5.c    void afunc(void) { afunc(); }
mainfunc        function 9   ex4-5.c    void mainfunc()
recurs          function 7   ex4-5.c    void recurs(void) { recurs(); }
xfunc           function 3   ex4-5.c    void xfunc(void) { zfunc(); }
zfunc           function 1   ex4-5.c    void zfunc(void) { afunc(); }
```

---

1 См. сайт http://sourceforge.net/projects/astyle.

2 См. сайт www.gnu.org/software/cflow.

Обратите внимание на разницу в подходе между cflow и ctags, которая заключается в том, что ctags фокусируется исключительно на определениях, а не на ссылках. Несмотря на то что cflow предоставляет дополнительные сведения о ссылках, ctags обладает большей функциональностью. Во-первых, ctags поддерживает большое количество языков, помимо C и C++, в то время как cflow — только один C. Во-вторых, ctags позволяет осуществлять фильтрацию кода на языке C посредством параметра --c-kinds.

4.4.5. Просмотр программного кода при помощи cscope

cscope представляет собой текстовый браузер для просмотра кода. Он создает собственную базу данных из предоставленного списка исходных файлов, после чего переходит в систему текстовых меню ncurses. Для того чтобы пользоваться cscope, вам потребуется функциональный терминальный эмулятор. Как выглядит система меню браузера cscope, можно увидеть на рис. 4.9.

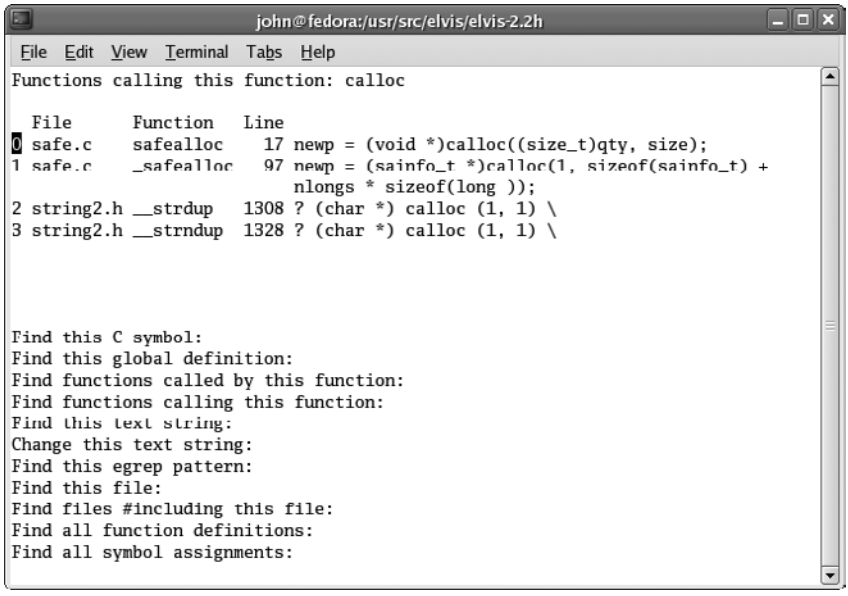


Рис. 4.9. Система меню браузера cscope

4.4.6. Просмотр и создание документации к программному коду при помощи Doxygen

Doxygen является замечательным инструментом, предназначенным прежде всего для генерирования документации к программным проектам. Он позволяет выполнять разбор кода на языках C и C++ и создавать документацию на основе гиперссылок (обычно в формате HTML), которую можно просматривать. Кроме того,

вы можете добавить «легкую» разметку в свой программный код, и Doxygen включит ее в документацию. Вот пример документации, достаточной для функции:

```
/**
 ** Это функция.
*/
void func(void)
{
}
```

Java-программисты сразу узнают здесь синтаксис javadoc. На самом деле Doxygen многое заимствует из javadoc. Это синтаксис «легкой» разметки, позволяющий писать комментарии, которые можно прочесть в текстовом редакторе, который можно использовать и для создания качественной текстовой документации.

Инструмент Doxygen генерирует вывод в форматах HTML, LaTeX, PDF, RTF, а также страницы руководства `man` (например, troff). Он также может использовать инструмент Graphviz (dot)[1] для создания комплексных UML-диаграмм для классов C++. Это превосходный инструмент для проверки конструкций, в которых используется синтаксис UML. Если вы будете иметь дело с подобной конструкцией, то сможете создавать аналогичные диаграммы из исходного кода.

Сначала Doxygen создает файл Doxyfile, в котором содержатся ваши предпочтения относительно определенного проекта.

В табл. 4.30 представлен ряд полезных настроек, которые вы можете использовать в своем файле Doxyfile.

Doxygen — это очень полезный инструмент для создания документации на основе исходного кода. Из-за того что документация является частью исходных файлов, ее легче обновлять, чем если бы она содержалась в отдельных файлах. Подобное расположение документации позволяет обновлять ее по мере создания новых версий исходных файлов. Задача по обновлению содержимого документации попрежнему возлагается на разработчиков, но поскольку она находится под пристальным вниманием пользователей, у разработчиков все меньше причин пренебрегать этой своей обязанностью.

Таблица 4.30. Полезные теги Doxygen

Тег	Назначение
USE_PDFLATEX	Генерирование PDF-файлов из LaTeX-файлов; требуется GENERATE_LATEX (значение по умолчанию NO)
PDF_HYPERLINKS	Генерирование PDF-файлов с гиперссылками; требуется USE_PDFLATEX (значение по умолчанию NO)
GENERATE_HTML	Генерирование HTML-файлов (значение по умолчанию YES)
GENERATE_TREEVIEW	Генерирование иерархического представления классов в формате HTML (значение по умолчанию NO)
GENERATE_LATEX	Генерирование исходных файлов в формате LaTeX (значение по умолчанию YES)
GENERATE_RTF	Генерирование RTF-файлов (значение по умолчанию NO)
GENERATE_MAN	Генерирование страниц руководства man (значение по умолчанию NO)

Продолжение ⇩

[1] См. сайт www.graphviz.org.

Таблица 4.30 *(продолжение)*

Тег	Назначение
HAVE_DOT	Использование программы dot Graphvis для генерирования диаграмм кооперации (значение по умолчанию NO)
UML_LOOK	Придание диаграммам кооперации UML-вида (значение по умолчанию NO)

4.4.7. Использование компилятора для анализа программного кода

Компилятор GNU Compiler Collection (GCC) предоставляет новые возможности анализа исходного кода. Прежде всего это препроцессор C. Большинство параметров препроцессора, используемых в командной строке `gcc`, аналогичны используемым в командной строке `cpp`, однако по большому счету для взаимодействия с препроцессором рекомендуется применять `gcc`.

Зависимости

Компилятор позволяет генерировать зависимости при помощи параметра `-M`. Данный параметр сам по себе включает в зависимости системные заголовки, что создает беспорядок. Скорее всего, вы захотите показать зависимости, касающиеся только исходных файлов. Для этого можно воспользоваться параметром `-MM`. Вот пример из дерева исходных файлов `strace`[1]:

```
$ gcc -MM -I ./linux syscall.c
syscall.o: syscall.c defs.h ./linux/syscall.h ./linux/dummy.h \
  ./linux/syscallent.h ./linux/errnoent.h
```

Обратите внимание на то, что данный вывод предназначен для использования в файле `Makefile`, поэтому каждая строка заканчивается обратным слешем. Существует совсем немного параметров, позволяющих сделать его более дружественным пользователю. Данный вывод не только позволяет создавать файлы `Makefile` или дополнения к ним, он также помогает понять, что творится в незнакомом проекте. В приведенном ранее примере видно, что `syscall.c` требует наличия файла с именем `dummy.h`, несмотря на то что он не был задействован посредством `syscall.c`; на самом деле он задействуется посредством `syscall.h`. Вывод также зависит от указываемого пользователем пути поиска, что выполняется с помощью параметра `I`. По умолчанию, если необходимый файл не обнаруживается, выполнение команды завершается неудачей. Вы можете исправить такое поведение, применив параметр `-MG`, который предполагает, что ненайденные файлы будут созданы во время компиляции и помещены в текущий каталог.

Макрорасширения

Отладка макросов препроцессора осуществляется при помощи параметра `-d` (табл. 4.31), который может использоваться только вместе с препроцессором (па-

[1] См. сайт http://sourceforge.net/projects/strace.

раметр -E). Для того чтобы просмотреть список предопределенных макросов в произвольном порядке, нужно ввести следующую команду:

```
$ echo | gcc -E -dM -
```

Таблица 4.31. Флаги, используемые вместе с параметром -d

Параметр	Описание
-dM	Позволяет выводить список операторов #define из исходного кода, а также встроенные макросы. Сведения выводятся в произвольном порядке
-dD	По сути, аналогичен параметру –dM. В GNU-документации говорится, что он не позволяет просматривать встроенные макросы, однако на самом деле он выводит большинство из них. Макросы, обнаруживаемые в исходном коде, выводятся в том порядке, в каком они объявлены
-dN	Генерирует тот же вывод, что и –dD, за исключением того, что выводятся только имена макросов. Значения макросов опускаются
-dI	В сочетании с –E данный флаг позволяет включать в итоговый вывод операторы #include; в выводе препроцессора они обычно отсутствуют

4.5. Заключение

Содержание данной главы сосредоточено на инструментах, используемых для манипулирования исходным кодом. Здесь рассмотрен ряд функциональных возможностей, которыми должен обладать текстовый редактор, чьей целевой аудиторией являются программисты. Мы изучили и сравнили два наиболее популярных текстовых редактора, предназначенных для операционной системы Linux: Vim и Emacs. Также были рассмотрены некоторые альтернативные им программы со всеми их преимуществами и недостатками.

Кроме того, мы поговорили о механизме контроля версий, познакомились с его базовыми концепциями и инструментарием. Вы научились создавать и накладывать «заплаты» (патчи), что является важнейшим аспектом деятельности, связанной с использованием инструментов контроля версий.

В завершение главы мы рассмотрели инструментарий, позволяющий извлекать из исходного кода сведения в виде перекрестных ссылок, браузерного вывода и даже набранной документации.

4.5.1. Инструментарий, использованный в этой главе

В данной главе рассмотрены два основных текстовых редактора:

- Vim — наиболее популярный клон редактора vi, который является стандартным текстовым редактором;
- Emacs — ведущий текстовый GNU-редактор.

Также был упомянут ряд клонов редакторов vi и Emacs. Большинство из них обладают ограниченной функциональностью, однако потребляют меньше памяти:

- клоны редактора vi: Elvis, nvi, Vile;
- клоны редактора Emacs: Zile, joe, jed.

Программы vi и Emacs первоначально были терминальными текстовыми редакторами, которые впоследствии *обзавелись* графическим пользовательским интерфейсом GUI. Из-за этого они по-прежнему сохраняют терминальную «внешность». Все более современные редакторы обладают GUI-интерфейсом и могут оказаться интуитивно более понятными для пользователей, незнакомых с vi и Emacs:

- GNOME: Gedit;
- KDE: Kate, Kwrite;
- X (общий): NEdit, SciTE.

В этой главе мы изучили механизм контроля версий и инструментарий, который при этом используется.

- Инструменты для выполнения объединений и выявления различий: `diff`, `diff3`, `patch`, `xxdiff`, `vimdiff`, `gvimdiff`.
- Инструменты для управления проектами: Subversion, CVS, monotone, GNU arch.

Мы также рассмотрели ряд инструментов, предназначенных для улучшения внешнего вида и просмотра программного кода:

- `indent`;
- `astyle`;
- `cflow`;
- `ctags`.

4.5.2. Рекомендуемая литература

- *Cameron D. et al.* Learning GNU Emacs. 3d ed. — Sebastopol, Calif.: O'Reilly Media, Inc., 2004.
- *Dougherty D. and Robbins A.* Sed and awk. 2d ed. — Sebastopol, Calif.: O'Reilly Media, Inc., 1997.
- *Friedl J. E. F.* Mastering Regular Expressions. 3d ed. — Sebastopol, Calif.: O'Reilly Media, Inc., 2006.
- *Lamb L. and Robbins A.* Learning the vi Editor. 6th ed. — Sebastopol, Calif.: O'Reilly Media, Inc., 1998.

4.5.3. Веб-ссылки

Текстовые редакторы:

- Emacs — www.gnu.org/software/emacs;
- Vim — www.vim.org.

Клоны текстовых редакторов:

- bvi — http://bvi.sourceforge.net;
- gedit — www.gnome.org/projects/gedit;
- JED — www.jedsoft.org/jed;
- joe — http://joe-editor.sourceforge.net;
- Kate — www.kate-editor.org;
- nano — www.gnu.org/software/nano;
- NEdit — www.nedit.org;
- SciTE — www.scintilla.org/SciTE.html;
- vile — http://invisible-island.net/vile;
- WordStar — www.wordstar.org;
- Zile — http://zile.sourceforge.net.

Инструменты для просмотра и улучшения внешнего вида программного кода:

- astyle — http://astyle.sourceforge.net;
- cflow — www.gnu.org/software/cflow;
- cscope — http://cscope.sourceforge.net;
- Doxygen — www.stack.nl/~dimitri/doxygen;
- Exuberant Ctags — http://ctags.sourceforge.net, http://xxdiff.sourceforge.net.

Инструменты контроля версий:

- arch — www.gnuarch.org/arch;
- cvs — www.nongnu.org/cvs;
- monotone — http://venge.net/monotone;
- Subversion — http://subversion.tigris.org;
- xxdiff — http://xxdiff.sourceforge.net.

5 Что должен знать каждый разработчик о ядре Linux

5.1. Введение

При изучении данной главы предполагается, что вы обладаете определенным опытом написания приложений под Linux, а также базовыми знаниями о ядре этой операционной системы. Мы затронем здесь некоторые вопросы, касающиеся ядра, которые часто рассматриваются в книгах, целиком посвященных ядру Linux. Однако, в отличие от таких книг, в материале этой главы внимание фокусируется на приложениях.

Тематика данной главы включает рассмотрение планировщика Linux, который претерпел немало изменений за последнее время. Я расскажу вам о приоритетах и вытеснении процессов, их роли, а также о приложениях реального времени.

В прошлом 32-битное адресное пространство было достаточно большим для того, чтобы большинство приложений никогда не сталкивалось с какими-либо ограничениями. В настоящее время, когда в 32-битных системах может устанавливаться более 4 Гбайт оперативной памяти, многие программисты, впервые встречающие подобные ограничения, не всегда хорошо представляют себе, с чем они столкнулись на самом деле. После прочтения этой главы вы сможете намного лучше разбираться в таких проблемах, а также узнаете, как их решить.

В данной главе мы рассмотрим системные вводы и выводы и особенности их связи с процессами. Вполне возможно, что вас впечатляют запредельные тактовые частоты современных процессоров, однако вы можете разочароваться в производительности маломощных устройств. Я расскажу вам о некоторых неэффективных сторонах модели программирования Linux и о том, как их можно избежать. Вы также узнаете об усовершенствованиях в планировщике операций ввода/вывода Linux 2.6 и научитесь пользоваться всеми его преимуществами.

5.2. Сравнение режима пользователя и режима ядра

Процессы выполняются в двух режимах: режиме пользователя и режиме ядра. Программный код, который вы создаете, и библиотеки, с которыми вы осуществля-

ете связывание, исполняются в режиме пользователя. Если ваш процесс нуждается в службах из ядра, то здесь потребуется выполнение программного кода ядра, что возможно только в режиме ядра. Все это может показаться несложным, однако суть всегда кроется в мелочах. Прежде всего, зачем нужны два режима работы?

Первая причина — безопасность. При выполнении процесса в режиме пользователя доступная ему область памяти принадлежит исключительно ему. Поскольку Linux — это многопользовательская операционная система, ни один из процессов не должен иметь возможности «подглядывать» в память другого процесса, в которой могут содержаться пароли и прочая важная информация. Режим пользователя позволяет гарантировать, что процессу доступна только та область памяти, которая ему принадлежит. Кроме того, в случае, если будут нарушены внутренние структуры процесса, крах потерпит только он один, а не утянет за собой другие процессы и, естественно, всю систему в целом. Память, которая доступна процессу в режиме пользователя, называется *пространством пользователя*.

Для того чтобы система в целом могла функционировать, ядро должно иметь возможность поддерживать структуры данных для управления всеми процессами в системе. Для этого необходима область памяти, которая будет общей для всех процессов. Поскольку ядро выполняется всеми процессами в системе, каждый из них нуждается в доступе к этой общей области памяти. Однако в целях безопасности программный код ядра и структуры данных должны быть строго изолированы от пользовательского программного кода и данных. Вот почему понадобился режим ядра. В данном режиме выполняется только код ядра, там он может получить доступ к общим данным ядра и исполнять привилегированные инструкции. Память, доступная для процесса в режиме ядра, называется *пространством ядра*. Существует только одно пространство ядра, доступное для любых процессов в режиме ядра, в отличие от пространства пользователя, которое является уникальным для каждого процесса.

На рис. 5.1 приведен пример распределения виртуальных адресов между процессами и ядром. В данном примере ядру отводится 1 Гбайт виртуальных адресов, а все остальное — процессам. Данное разделение (так называемое разделение «1 Гбайт/3 Гбайт») может осуществляться при сборке ядра, оно является обычным для большинства ядер. В приведенной конфигурации все адреса выше `0xC0000000` располагаются в ядре. Для того чтобы иметь возможность использовать их, процесс должен выполняться в режиме ядра.

Системные вызовы. Процессы переходят и выходят из режима ядра посредством системных вызовов. Многие POSIX-функции, например `open`, `close`, `read`, `ioctl` и `write`, являются обычными обертками для системных вызовов. Драйверы устройств, к примеру, функционируют только в режиме ядра. Программный код приложений не может осуществлять вызов функций драйверов напрямую. Вместо этого приложения используют один из предопределенных системных вызовов, для того чтобы косвенным путем получить доступ к программному коду драйверов. Например, вызов `read` будет выглядеть следующим образом:

```
#include <syscall.h>
...
n = syscall(SYS_read, fd, buffer, length);
```

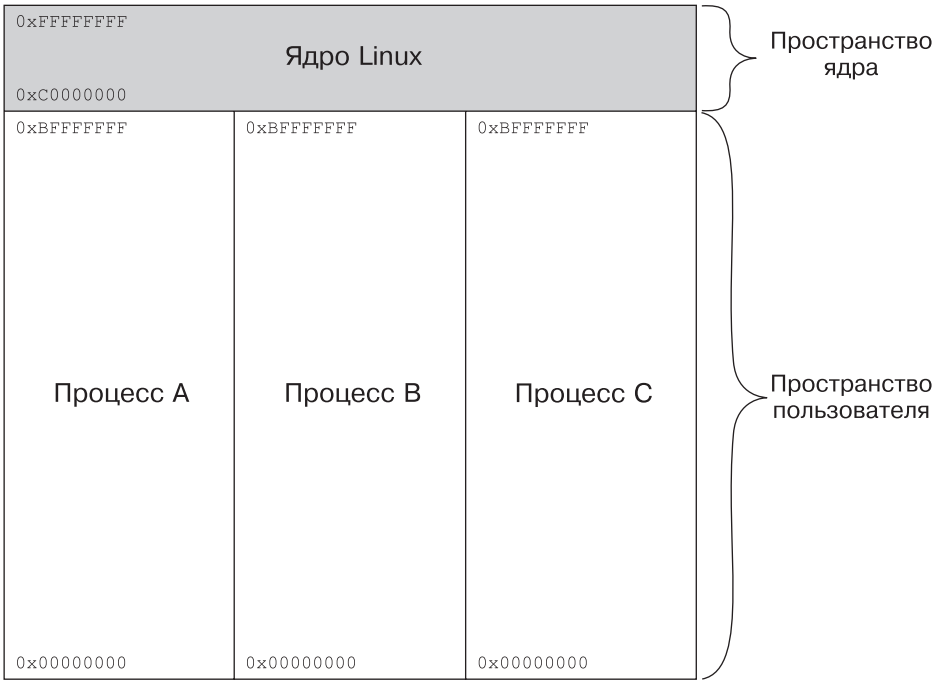


Рис. 5.1. Виртуальные адреса в типичном 32-битном окружении

Каждому системному вызову ядро присваивает номер — в приведенном примере он определяется посредством макроса `SYS_read`. Макросы для системных вызовов определяются в `syscall.h`. Перечень системных вызовов, доступных в Linux, зависит от версии ядра, со временем он претерпел небольшие изменения. Однако механизм совершения системных вызовов уникален для каждой процессорной архитектуры. Функция `syscall` является оберткой для ассемблерного кода, используемого для совершения системных вызовов.

Технология, используемая в Linux для осуществления системных вызовов, называется *двоичным программным интерфейсом* (*Application Binary Interface*, сокращенно *ABI*), однако она свойственна не только для Linux. Аналогичная технология применяется в других операционных системах и даже в базовой системе ввода/вывода BIOS. В отличие от интерфейса программирования приложений (*Application Programming Interface, API*), который требует привязки к совместимым функциям, ABI-интерфейс не потребует от вас связывать свой программный код с другим программным кодом, который предполагается исполнять. Благодаря этому ваша исполняемая программа сможет функционировать на множестве различных ядер без необходимости в пересборке. Большинство проблем совместимости с разными дистрибутивами Linux связаны с различиями в API-интерфейсах библиотек, а не в ядре. Если, к примеру, у вас есть статически связанный исполняемый файл, который работоспособен на ядре Linux 2.2, то высока вероятность того, что он будет совместим и с ядром версии 2.6, поскольку большинство наиболее общих интерфейсов системных вызовов никогда не меняются.

5.3. Планировщик процессов

Во времена DOS и CP/M типичная настольная операционная система могла выполнять только один процесс одномоментно. Планирование не было актуальным, так как система выполняла только по одной задаче за один подход в порядке, в котором они запрашивались. Однако все это осталось в прошлом. В настоящее время даже самая скромная встраиваемая операционная система поддерживает многозадачность.

Проблема многозадачных операционных систем заключается в том, что им приходится распределять процессорное время между различными задачами. Используемый при этом алгоритм называется *планировщиком*. Каждая операционная система обладает собственным алгоритмом планирования, иногда даже более чем одним, поскольку единого алгоритма, который идеально подходил бы для всех программ, не существует. Планировщик может без проблем работать с одним набором процессов и при этом не годиться для другого набора. Ядро Linux содержит несколько алгоритмов планирования и позволяет пользователю выбирать во время загрузки тип планировщика, который будет использоваться системой.

5.3.1. Основы планирования

В сфере Linux планировщик иногда рассматривается как отдельный процесс. На самом деле программный код планировщика выполняется всеми процессами. Всякий раз, когда какой-либо процесс приостанавливается или блокируется в ожидании устройства, он осуществляет вызов планировщика для определения того, какой процесс будет выполняться следующим. Вызовы планировщика часто внедряются в системные вызовы и осуществляются, когда процессу необходимо дождаться определенного события.

Процесс, который активно взаимодействует с устройствами, будет часто вызывать планировщик. Ввод/вывод устройств неизменно связан с некоторым периодом ожидания. Если устройство медленное, бóльшая часть времени выполнения процесса будет потрачена на ожидание. Подобный процесс не потребляет много процессорного времени по сравнению с общим временем выполнения. Если бы все процессы поступали аналогичным образом, операционная система могла бы позволить им самим вызывать планировщик, и все замечательно работало бы. Такая схема существует и называется *кооперативной многозадачностью*.

5.3.2. Блокировка, вытеснение и уступка

Каждому процессу Linux отводится временной интервал (или *квант*), в течение которого он будет выполняться, прежде чем будет остановлен ядром и уже другой процесс получит возможность выполняться. Когда ядро останавливает процесс по причине того, что его временной интервал истек, мы говорим, что процесс *вытесняется*. Ядро может вытеснять процесс еще до того, как истечет его временной интервал, если готов к выполнению процесс с более высоким приоритетом. Когда

такое происходит, мы говорим, что процесс с более высоким приоритетом *вытесняет* процесс с более низким приоритетом.

Процесс может также добровольно освобождать ресурсы процессора. Когда такое случается, мы говорим, что процесс *уступает* ресурсы центрального процессора. Процесс может воспользоваться системным вызовом `sched_yield`, для того чтобы явным образом освободить ресурсы процессора от нагрузки. Довольно часто ресурсы процессора освобождаются посредством других системных вызовов, совершаемых процессом. Например, если процесс вызывает `read` или `write`, высока вероятность того, что ему придется ожидать устройство. «Порядочный» драйвер устройства приостановит процесс и уступит ресурсы процессора до тех пор, пока соответствующее устройство не будет готово.

Когда процесс ожидает определенного события в режиме ядра, мы говорим, что данный процесс *блокируется*. Это означает, что он не будет готов к выполнению до тех пор, пока не произойдет определенное событие. Отсюда следует, что блокированный процесс не потребляет ресурсы процессора и не будет запланирован, пока не произойдет конкретное событие и не «разбудит» его.

5.3.3. Приоритеты и справедливость планирования

Во всех операционных системах с поддержкой вытесняющей многозадачности, включая Linux, реализована схема приоритетов планирования. Проще говоря, приоритеты разрешают конфликты планирования в ситуациях, когда к выполнению оказываются готовы два и более процесса. Всякий раз, когда такое происходит, процесс с более высоким приоритетом получает возможность выполняться раньше процесса с более низким приоритетом. Приоритеты процессов могут изменяться пользователем, однако в конечном счете они определяются ядром.

Основная задача планировщика Linux заключается в обеспечении каждому процессу возможности выполняться — ни один из них не должен быть лишен процессорного времени. Планировщик обращает внимание на поведение каждого процесса, и если среди них выявляются интерактивные процессы, их приоритет повышается. Хорошим примером интерактивного процесса будет ввод с клавиатуры. Этот процесс проводит основную часть своего времени в ожидании ввода и обрабатывается очень быстро. Он легко определяется планировщиком благодаря тому, что никогда не вытесняется и не потребляет много процессорного времени. Он всегда добровольно освобождает ресурсы процессора. Для присваивания интерактивным процессам более высокого приоритета планировщик использует бонусное значение в дополнение к *статическому приоритету* процессов — при создании процесса ему присваивается приоритет, который планировщик оставляет неизменным на протяжении всей жизни этого процесса. Эффективный приоритет процесса является суммой его статического приоритета и бонусного значения[1]. Бонусное значение бывает положительным или отрицательным, то есть эффективный приоритет может быть как выше, так и ниже статического приоритета.

---

[1] Здесь не учитывается значение nice, о котором мы вскоре поговорим.

Краткое описание параметров команды ps

В приведенном примере вместе с командой использован ряд необычных параметров. Параметр `-C` указывает команде `ps` выводить на экран только процессы с исполняемыми именами, которые соответствуют аргументу. Для того чтобы команда `ps` осуществляла поиск более одного командного имени, необходимо указывать параметр `-C` несколько раз.

Параметр `-o` позволяет определять формат вывода. После него указываются поля, которые пользователь желает видеть в итоговом выводе.

5.3.4. Приоритеты и значение nice

Если у вас запущено большое количество процессов, это вовсе не означает, что ваша система будет медленно функционировать. Процессы на самом деле могут *не выполнять* никаких особых действий, поэтому вполне естественно, что они не будут вызывать снижение скорости работы системы.

Однако подобные заурядные процессы все же *могут* отрицательно сказаться на производительности системы при определенных условиях. Одно из них заключается в том, чтобы присвоить такому процессу высокий приоритет. Ядро позволяет пользователям оказывать влияние на решения планировщика о присвоении приоритетов посредством использования так называемого *значения* `nice`. Если наделить процесс положительным значением `nice`, планировщик присвоит ему более низкий приоритет; если же процесс получит отрицательное значение, планировщик присвоит ему более высокий приоритет.

Значение `nice` вычитается из суммы бонусного значения и статического приоритета, в результате чего получается эффективный приоритет. Любой непривилегированный пользователь может задавать положительное значение с использованием команды `nice`, однако только суперпользователь имеет право присваивать отрицательное значение `nice`.

5.3.5. Приоритеты реального времени

Планировщик позволяет осуществлять разные виды планирования процессов, которые имеют строгие требования к латентности. Под *латентностью* понимается время, которое требуется программе для того, чтобы отреагировать на внешние события, например прерывания. Приложения со строгими требованиями к латентности часто называются *приложениями реального времени*. Они гарантируют, что программы будут реагировать на события в течение определенного интервала времени, в противном случае возникнут проблемы.

К приложениям реального времени, которые вы можете встретить на своем компьютере, относится медиапроигрыватель. При воспроизведении видео проигрывателю приходится обновлять экран через определенные интервалы времени, иначе ваш любимый фильм будет идти рывками. Это называется *мягким приложением*

реального времени, поскольку если программа изредка запаздывает в работе, она всегда может наверстать упущенное. Если ваш медиапроигрыватель пропускает кадры, это еще не конец света. А вот *жесткое приложение реального времени* не имеет права запаздывать никогда. Примером такого приложения может быть бортовой компьютер управления полетом, который должен мгновенно реагировать на действия пилота. Запаздывание в работе такого приложения может стоить человеческих жизней.

Планировщик Linux обеспечивает реализацию планирования реального времени, которая очень близка к стандарту POSIX 1003.1. Здесь мы получаем в свое распоряжение 100 дополнительных уровней приоритетов, каждый из которых является более высоким по сравнению с приоритетами нормальных процессов (0–39). Процессы реального времени в Linux могут иметь приоритеты в диапазоне от 41 до 139 (по некоторым причинам приоритет, равный 40, не используется). Как и в случае с нормальными процессами, более высокие значения означают более высокий приоритет, однако приоритеты реального времени отличаются тем, что они никогда не меняются на протяжении всей жизни процесса. Из-за этого процессы реального времени не имеют значений `nice`, а также бонусных значений. Приоритет будет таким, каков он есть в данный момент.

Когда вы определяете процесс как процесс реального времени, необходимо также указывать политику планирования. Стандарт POSIX устанавливает две политики планирования для процессов реального времени: *FIFO* и *циклическое планирование*.

Планирование FIFO

Термин FIFO — это сокращение от *first in, first out*, означающее то, как процессы размещаются в очереди выполнения. Когда два процесса FIFO с одинаковым приоритетом готовы к выполнению, тот из них, который первым оказался готовым, и будет выполняться первым, причем всегда. Процесс FIFO никто не может вытеснить, за исключением другого процесса с более высоким приоритетом, который по определению будет еще одним процессом реального времени. Если вы запустите сценарий `cruncher` как процесс FIFO, то рискуете сделать вашу систему непригодной к использованию.

Циклическое планирование

Циклическое планирование — это вторая политика для процессов реального времени, которая почти идентична планированию FIFO, за исключением того, что циклические процессы не могут запускаться в произвольном порядке. Вместо этого им выделяется определенный временной интервал, в течение которого они будут выполняться. Процесс, запущенный посредством циклического планирования, может быть вытеснен, только когда его временной интервал истечет или будет готов к выполнению процесс с более высоким приоритетом. Если циклический процесс вытесняется процессом с более высоким приоритетом, планировщик позволяет этому циклическому процессу использовать остаток его временно́го интервала до того, как будут запланированы другие процессы с аналогичным приоритетом. Только когда циклический процесс уступит ресурсы центрального процессора, процессы с более низким приоритетом получат возможность выполняться.

5.3.6. Создание процессов реального времени

Ранее мы рассмотрели первый способ создания процесса реального времени при помощи команды chrt. Для присваивания приоритетов данная команда использует базовые вызовы fork и exec в сочетании с дополнительным POSIX-вызовом. Для того чтобы присвоить приоритет реального времени, приложению потребуются следующие POSIX-функции:

```
int sched_setscheduler(pid_t pid, int pol, const struct sched_param*p);
int pthread_setschedparam(pthread_t thread, int policy,
       const struct sched_param *param);
```

Функция sched_setscheduler используется процессами и принимает в качестве аргумента идентификатор процесса. Функция pthread_setschedparam предназначена для потоков и принимает в качестве аргумента идентификатор потока вместо идентификатора процесса. Обе эти функции требуют наличия политики и указателя на структуру sched_param. Индикатором политики может быть один из макросов, приведенных в табл. 5.1.

Единственное значение в структуре sched_param, вводимое пользователем, располагается в поле priority, при этом оно должно относиться к определенному диапазону допустимых значений. Определить диапазон допустимых значений можно при помощи следующих POSIX-функций:

```
int sched_get_priority_min(int policy);
int sched_get_priority_max(int policy);
```

Стандарт POSIX разрешает любой политике планирования реального времени иметь уникальный диапазон приоритетов, несмотря на то что Linux использует аналогичный диапазон для обоих политик реального времени (SCHED_FIFO и SCHED_RR). При указании политики, не относящейся к политикам реального времени (SCHED_OTHER), функция sched_setscheduler не позволяет задавать приоритет. Любое значение приоритета, отличное от нуля, вызовет ошибку, при этом переменной errno будет присвоено значение EINVAL. Вместо этого процесс может определить значение nice при помощи системных вызовов nice или setpriority. Имейте в виду: несмотря на то что имя setpriority подразумевает, что вы задаете приоритет, на самом деле данный системный вызов позволяет определять только значение nice.

Таблица 5.1. Макросы, используемые в качестве индикаторов политики планирования POSIX

Макрос	Значение
SCHED_FIFO	Использовать планирование FIFO
SCHED_RR	Использовать циклическое планирование
SCHED_OTHER	Использовать обычное планирование Linux

Операционная система Linux использует диапазон от 1 до 99 для приоритетов реального времени POSIX, которые передаются функции sched_setscheduler. Это немного смущает, поскольку планировщик Linux использует только один непрерывный диапазон приоритетов как для нормальных процессов, так и для процессов реального времени. Полный диапазон абсолютных приоритетов, используемый планировщиком,

содержит значения от 0 до 139. К примеру, если вы присвоите приоритет, равный 1, планировщик будет использовать абсолютный приоритет, равный 41.

5.3.7. Состояния процессов

На протяжении своей жизни процессы проходят через несколько состояний. Пользователь видит только те состояния, которые выводят на экран инструменты вроде команды `ps` либо которые доступны в файловой системе `/proc` (ее использует `ps`). В табл. 5.2 приведены состояния и их сокращенные названия.

Таблица 5.2. Состояния процессов, доступные пользователю для просмотра

Состояние	Сокращенное название	Значение
Запущен (Running)	R	Запущен или готов к запуску
Прерываемый (Interruptible)	S	Блокирован и ожидает события, но может быть «разбужен» сигналом
Непрерывный (Uninterruptible)	D	Блокирован и ожидает события, но не может быть «разбужен» сигналом
Остановлен (Stopped)	T	Остановлен из-за управления работами или внешней трассировки (например, ptrace)
Зомби (Zombie)	Z	Завершен, но его родительский процесс еще не совершал системный вызов wait (не уничтожал этот процесс)

Сравнение состояний, когда процесс приостановлен («спит») и запущен

Если процесс пребывает в «работоспособном» состоянии, это не означает, что он действительно выполняется в данный момент. Это означает лишь, что данный процесс не приостановлен и не ожидает события. В подобном состоянии может пребывать множество процессов. Например, процесс дробления чисел всегда будет в «работоспособном» состоянии. Вы всегда должны иметь это в виду, когда будете использовать команду `ps`, как показано в следующем примере:

```
$ ./cruncher & ./cruncher & ./cruncher &
$ ps -C cruncher -p $$ -o pid,state,cmd
  PID S CMD
 2588 S bash
 2657 R /bin/sh ./cruncher
 2658 R /bin/sh ./cruncher
 2659 R /bin/sh ./cruncher
```

В приведенном примере три процесса дробления чисел запускаются как фоновые задачи, после чего выполняется команда `ps`, выводящая на экран состояния процессов, а также состояние процесса родительской оболочки. Как видно в выводе, все эти три процесса находятся в состоянии `R`, то есть все они «работоспособны». Это вполне ожидаемо, так как они никогда не приостанавливаются (не «спят»). Однако, поскольку мы используем однопроцессорную систему, только один из этих процессов может выполняться в данный момент. В выводе также указывается, что родительская оболочка (`bash`) приостановлена («спит»). Это также вполне ожида-

емо. Поскольку оболочка является создателем процесса, она, вероятно, «спит» и не совершает системный вызов `wait`, ожидая завершения работы команды `ps`.

Состояние «сна» оболочки `bash`, которое наблюдается в предыдущем примере, может быть прервано. Это означает, что, если эта оболочка получит сигнал, она отреагирует на него (то есть запустит обработчик сигналов). Например, сигнал `SIGTERM` или `SIGQUIT` приведет к завершению ее работы. Большая часть времени, в течение которого ваш программный код блокирован из-за ввода/вывода или просто приостановлен, будет представлять собой состояние «сна», которое может быть прервано.

Состояние непрерывного «сна» встречается реже и используется, когда код ядра (чаще всего драйвер устройства) решает, что определенный процесс не должен прерываться, пока идет выполнение операции. Обычно оно представляет собой переходное состояние, которое драйвер использует в течение коротких промежутков времени, для того чтобы гарантировать, что процесс завершит начатое. Так, например, ваш драйвер может отправлять биты данных на конкретное устройство и ожидать ответа посредством опроса. Драйверу необходимо обеспечить пребывание устройства в известном состоянии, чего нельзя гарантировать, если процесс может быть завершен во время «сна». Для предотвращения этого драйвер отправляет процесс в непрерывную «спячку» до тех пор, пока аппаратное устройство не вернется в известное состояние.

Процесс, пребывающий в непрерывной «спячке», может стать источником проблем. В обычной ситуации подобное состояние длится совсем недолго, однако если аппаратное устройство или носитель информации окажется дефектным, подобное состояние вызовет проблемы. Оно может никогда не возникнуть, пока вы не столкнетесь с дефектным устройством или носителем информации. Рассмотрим пример не совсем удачно написанного драйвера, который использует состояния непрерывного «сна» без каких-либо интервалов времени. Когда такой драйвер попытается считать данные с дефектного устройства, он может не получить необходимо ответа и оставить процесс в непрерывном «сне» на неопределенное время. А что еще хуже, пользователь не будет понимать, в чем дело. Все, что ему известно, — процесс завис и его не получается ни принудительно завершить, ни «разбудить».

Если вы столкнетесь с процессом, который принудительно не завершается даже при помощи `kill -9`, то высока вероятность того, что он «застрял» в состоянии непрерывного «сна». Единственным выходом из такой ситуации является перезагрузка и ремонт устройства (или, возможно, корректировка его драйвера).

Зомби и системный вызов wait

Когда процесс завершается, он полностью не исчезает до тех пор, пока его родительский процесс не совершит один из системных вызовов `wait`. Пока это не произойдет, процесс будет оставаться в так называемом состоянии зомби, ожидая, пока родитель не подтвердит его завершение. Причудливым термином *зомби* называется процесс, который завершился, но пребывает ни в «живом», ни в «мертвом» состоянии, как его тезка-нежить из фильмов ужасов. Зомби-процессы не поглощают ресурсов памяти или центрального процессора[1], однако они не включаются в вывод команды `ps`. Если родительский процесс завершается, не дожидаясь своих дочерних

[1] Равно как и человеческой плоти и мозгов.

процессов, такие процессы «усыновляются» процессом `init`, который периодически совершает вызов `wait` для их *уничтожения* (еще одна мрачная метафора).

Какой смысл в зомби-процессах?

Вы можете задать вопрос: «Зачем все эти хлопоты по сохранению зомби-процессов?» В конце концов, единственная полезная информация, которую они несут, — информация о состоянии их завершения.

Однако именно в этих сведениях заключается вся суть. Вас как программиста приложений может не волновать состояние завершения процесса, который вы «разветвляли» (хотя должно быть как раз наоборот), но ядро не будет знать об этом. Поскольку ядро считает, что родительскому процессу необходимо знать, в каком состоянии находится его дочерний процесс, оно отправляет родительскому процессу сигнал (`SIGCHLD`) и сохраняет состояние, которое собирает родительский процесс. И до тех пор, пока родительский процесс не извлечет возвращаемое состояние путем вызова одной из функций `wait`, дочерний процесс будет оставаться в состоянии зомби.

Если родительский процесс завершается раньше дочернего процесса, последний ***усыновляется*** процессом `init`, который немедленно собирает его состояние и эффективно удаляет зомби-процесс.

Остановленные процессы

Процессы могут останавливаться по разным причинам. Вам, вероятно, приходилось использовать сочетание клавиш Ctrl+Z для остановки процесса, выполняющегося в приоритетном режиме. По традиции терминалы определяют данное сочетание в виде так называемого символа `SUSP`, используемого для остановки процессов, запущенных в приоритетном режиме. В операционной системе Linux (и UNIX) нажатие данных клавиш приводит к тому, что псевдотерминал посылает процессу сигнал `SIGTSTP`[1]. Вы можете приспособить данную комбинацию под свои нужды либо использовать ее «настройки» по умолчанию[2].

Ряд сигналов позволяет переводить процесс в остановленное состояние; их перечень можно увидеть на странице руководства man `signal(7)`. Процесс выходит из данного состояния и продолжает выполняться, когда получает сигнал `SIGCONT`. По-другому вывести процесс из остановленного состояния можно только при помощи сигнала завершения. Обычно сигнал, принимаемый в остановленном состоянии, фиксируется ядром, и процесс не запустит свой обработчик событий до тех пор, пока не выйдет из такого состояния. Однако в операционной системе Linux имеется ряд исключений. Например, сигналы `SIGTERM` и `SIGKILL` обрабатываются незамедлительно, даже если процесс остановлен[3].

---

1 Не путать с SIGSTOP. Сигнал SIGTSTP может быть «пойман», а SIGSTOP — нет.

2 См. stty(1).

3 В отличие от SIGTERM, сигнал SIGKILL не может перехватываться обработчиком сигналов, определяемым пользователем.

Процессы могут останавливаться также терминалом. Терминал управляет процессами, используя систему процессов в *фоновом* и *приоритетном* режимах. Каждый терминал имеет один, и только один, процесс, выполняющийся в приоритетном режиме, который является единственным процессом, принимающим ввод с клавиатуры. Любой другой процесс, запущенный в данном терминале, будет считаться фоновым. Если подобный процесс попытается считать стандартный ввод, терминал остановит его при помощи сигнала SIGTTIN, поскольку имеется только одно устройство ввода (клавиатура), которое связано с процессом в приоритетном режиме. Процесс, остановленный посредством сигнала SIGTTIN, остается в таком состоянии до тех пор, пока не будет переведен в приоритетный режим при помощи команды fg. Следует отметить, что концепция процессов в *фоновом* и в *приоритетном* режимах используется только для терминалов. Ядро не отслеживает процессы подобным образом, однако обеспечивает сигналы, позволяющие облегчить управление процессами при помощи терминала. Вот пример, демонстрирующий сигнал SIGTTIN в действии:

```
$ read x &
[1] 5851
```

Здесь предпринимается попытка выполнения встроенной bash-команды read в фоновом режиме. Поскольку фоновому процессу не разрешается считывать стандартный ввод с терминала, он получает сигнал SIGTTIN, который переводит этот процесс в остановленное состояние. Следующая команда показывает, что данный процесс действительно пребывает в состоянии T (то есть *«Остановлен» (Stopped)*):

```
$ jobs -x ps -p %1 -o pid,state,cmd
  PID S CMD
 5851 T bash
```

Если процесс остановлен посредством сигнала SIGTTIN, его можно «разбудить» при помощи сигнала SIGCONT, однако он будет вновь заблокирован в случае попытки считать стандартный ввод. Он сможет завершить свой ввод только в том случае, если будет переведен в приоритетный режим.

Аналогичный способ может применяться для остановки фоновых процессов, для того чтобы они не загромождали экран терминала, что бывает довольно часто. В терминале имеется настройка tostop (сокращение от terminal output stop), которая по умолчанию деактивирована в большинстве систем. Если ее не активировать, фоновые процессы смогут осуществлять запись в терминал в любое время. Если установить флаг tostop при помощи команды stty, фоновые процессы будут останавливаться при каждой попытке осуществить запись в стандартный вывод.

5.3.8. Порядок измерения времени

Ядро отслеживает продолжительность времени выполнения всех процессов. Оно отдельно фиксирует, сколько времени каждый из процессов проводит в режиме пользователя и в режиме ядра. Команда time позволяет выяснить, где тот или иной процесс проводит свое время. Данная функциональная возможность встроена в оболочку bash и некоторые другие инструменты, а также доступна

в виде команды в /bin/time. В следующем примере используется ее встроенная bash-версия:

```
$ time sleep 1

real    0m1.042s
user    0m0.000s
sys     0m0.020s

$ time dd if=/dev/urandom of=/dev/null count=1000
1000+0 records in
1000+0 records out

real    0m0.527s
user    0m0.000s
sys     0m0.500s
```

Команда sleep позволяет осуществлять соответствующий системный вызов, который приводит к блокировке процесса на 1 с. В приведенном примере выполнение этого процесса занимает в общей сложности 1,042 с. Пока процесс блокирован, во время своей «спячки» он не потребляет ресурсов центрального процессора. В то же время команда dd осуществляет передачу 1000 блоков во время копирования данных из /dev/urandom в /dev/null. В этом случае выполнение процесса занимает 527 мс, при этом 500 мс он проводит в режиме ядра. По всей видимости, драйвер /dev/urandom выполнялся от имени нашего процесса. Помимо этого, команда dd имеет с ним мало общего, за исключением копирования данных, которое занимает остальные 27 мс.

Если ваша программа медленно выполняется из-за блокирующего устройства или системного вызова, повысить скорость ее работы можно несколькими способами. Наиболее очевидный из них — избегать применения таких вызовов. Если это невозможно, то попробуйте запараллелить свое приложение с потоками или асинхронным вводом/выводом.

Единицы измерения системного времени

Стандарт POSIX определяет базовый интервал времени хода часов (clock_t) в качестве единицы измерения системного времени в приложениях пространства пользователя. Однако у данной единицы есть два определения. ANSI-определение используется в сочетании с ANSI-функцией clock, которая не должна применяться в программах под Linux. Данная функция возвращает значение количества процессорного времени, потраченного на выполнение определенного процесса, которое приблизительно равно сумме времени выполнения программного кода пользователя и времени выполнения программного кода ядра.

Функция clock возвращает значение, измеряемое в CLOCKS_PER_SEC, что представляет собой системный макрос, определяемый стандартной библиотекой GNU как равный 1000000. В операционной системе Linux возвращаемое функцией clock значение измеряется в микросекундах, хотя реальная частота интервалов времени обычно бывает намного меньше. Частота интервалов хода часов, используемая

функциями, возвращающими значение clock_t, задается при помощи функции sysconf[1]:

```
sysconf(_SC_CLK_TCK);
```

sysconf возвращает значение в интервалах времени в секунду (или герцах), при этом любые переменные типа clock_t будут привязаны к этому значению. Это особенно важно, если вы измеряете уровень производительности, поскольку это значение определяет степень точности для функций, возвращающих значение типа clock_t.

Недостаток ANSI-функции clock заключается в том, что она переполняется в пределах чуть более одного часа. Для процессов, выполнение которых длится намного дольше этого времени, функция clock неприменима. Более того, данная функция не учитывает объем ресурсов центрального процессора, потребляемых дочерними процессами, и не отличает пространство пользователя от пространства ядра. Из-за всех этих недостатков применять функцию clock в Linux-системах не имеет смысла, несмотря на то что она является частью стандарта ANSI языка C. К счастью, в Linux имеется ряд альтернативных инструментов.

POSIX-функция times также использует тип clock_t, однако определяет единицы измерения по-другому. Вместо CLOCKS_PER_SEC или микросекунд возвращаемые функцией times значения clock_t измеряются в интервалах времени хода системных часов. Это значительно снижает вероятность переполнения.

Прототип функции times выглядит следующим образом:

```
clock_t times(struct tms *buf);
```

Возвращаемое значение является количеством интервалов времени хода настенных часов, прошедших с момента какого-либо события в прошлом. Linux под этим событием понимает загрузку операционной системы. Для обеспечения переносимости возвращаемое значение следует использовать в качестве эталона для относительного, а не абсолютного измерения времени выполнения процессов. Важные детали, возвращаемые функцией times, располагаются в структуре struct tms, адрес которой передается посредством процесса вызова. Структура tms определяется следующим образом:

```
struct tms {
       clock_t tms_utime;  /* время выполнения кода пользователя */
       clock_t tms_stime;  /* время выполнения кода ядра */
       clock_t tms_cutime; /* время выполнения дочерних процессов кода
                              пользователя */
       clock_t tms_cstime; /* время выполнения дочерних процессов кода
                              ядра */
};
```

Значение tms_utime представляет собой количество времени, которое процесс потратил на выполнение программного кода пользователя с момента своего запуска. В поле tms_stime содержится значение времени, которое затрачено процессом

[1] Необходимо отметить, что стандарт POSIX считает ANSI-макрос CLK_TCK устаревшим, несмотря на то что он по-прежнему определяется как sysconf(_SC_CLK_TCK).

на выполнение программного кода ядра с момента своего запуска. Вывод ANSI-функции clock является эквивалентом суммы этих двух значений, умноженной на интервал времени хода часов. Значения tms_cutime и tms_cstime являются аналогичными значениями, за тем исключением, что они высчитываются в отношении процессов-ответвлений, которые завершились и были уничтожены посредством одного из системных вызовов wait.

Альтернативой times является функция getrusage, впервые внедренная в системе BSD:

```
int getrusage(int who, struct rusage *usage);
```

В отличие от функции times, функция getrusage позволяет извлекать конкретные сведения, касающиеся как родительских, так и дочерних процессов.

Продолжая разговор о «часовых» функциях, необходимо упомянуть еще одну. Расширения реального времени POSIX, определяемые стандартом POSIX 1003.1, привносят функцию clock_gettime, которая обладает некоторыми преимуществами. Прототип clock_gettime выглядит следующим образом:

```
int clock_gettime(clockid_t clk_id, struct timespec *tp);
```

Первое, на что стоит обратить внимание, — это другая структура времени. timespec исчисляет время в наносекундах:

```
struct timespec {
      time_t      tv_sec;
      long        tv_nsec;
};
```

Здесь опять-таки исчисление времени в наносекундах не означает, что часы «тикают» каждую наносекунду. Данный API-интерфейс поддерживает различные типы часов, все они могут использовать разные интервалы хода и эталоны. Тип часов указывается посредством параметра clockid_t. В отличие от getrusage, функцию clock_getres можно использовать для определения интервала хода часов. Прототип функции clock_getres имеет следующий вид:

```
int clock_getres( clockid_t clk_id, struct timespec *res);
```

Здесь интервал хода часов указывается в структуре timespec. Несмотря на то что данный API-интерфейс поддерживает различные типы часов, стандарт POSIX требует, чтобы имелась поддержка только часов CLOCK_REALTIME.

В табл. 5.3 вы можете увидеть перечень часов, которые могут с успехом применяться на практике.

Таблица 5.3. Часы, используемые наряду с функцией clock_gettime

Идентификатор	Описание	Примечание
CLOCK_REALTIME	Стандарт POSIX предусматривает обязательную поддержку данных часов; количество секунд возвращается в формате Coordinated Universal Time (UTC) с более высокой частотой интервалов хода часов, чем в случае использования ANSI-функции time	Частота интервалов хода часов обычно аналогична SC_CLK_TCK

Идентификатор	Описание	Примечание
CLOCK_MONOTONIC	Простые часы, показывающие, какое время прошло с момента произвольного (и неопределенного) события в прошлом	Частота интервалов хода часов обычно аналогична SC_CLK_TCK
CLOCK_PROCESS_CPUTIME_ID	Показывают, сколько процессорного времени потребляет процесс. Как и в случае применения ANSI-функции clock, в значение потребленного времени включается время выполнения программного кода пользователя и программного кода ядра. Для многопоточных процессов в данное значения включается время, потребляемое потоками	В данном случае отсутствуют недостатки ANSI-функции clock. Частота интервалов хода часов, которую показывает clock_getres, составляет 1 нс, однако мои измерения на ядре 2.6.14 показали результат 1/100 с
CLOCK_THREAD_CPUTIME_ID	Показывают, сколько процессорного времени потребляет текущий процесс; данные часы аналогичны приведенным ранее, за исключением того, что время измеряется только относительно текущего выполняющегося потока	Частота интервалов хода часов, которую показывает clock_getres, составляет 1 нс, но на самом деле интервал будет намного больше. Частота интервалов хода часов, показываемая clock_getres, равна 1 нс, однако мои измерения на ядре 2.6.14 показали результат 1/100 с

Переносимые приложения должны использовать `clock_getres` для проверки интервала хода часов и доступности определенных часов, перед тем как они будут задействованы. Функция `clock_gettime` не принимает во внимание дочерние процессы, поэтому системные вызовы `wait` никак не повлияют на возвращаемые ею значения.

Интервал времени хода часов в ядре

Стандартная единица измерения времени в ядре называется *мгновением*. Одно мгновение представляет собой интервал времени хода внутренних часов, через который аппаратный таймер запрограммирован генерировать прерывания на определенной частоте. Частота определяется при сборке ядра и остается неизменной. В большинстве дистрибутивов используется значение по умолчанию, которое хранится в макросе `HZ`. Каждая архитектура определяет уникальное значение `HZ` по умолчанию. До недавнего времени данное значение было довольно сложно изменить в ядре. Многих пользователей вполне устраивало значение по умолчанию, которое для архитектуры IA32 равнялось 100 Гц. По случайному совпадению аналогичная частота используется стандартной библиотекой GNU для `clock_t`.

В табл. 5.4 приведены примеры реального распределения интервалов времени в сравнении с частотой 100 Гц, по умолчанию используемой в операционной системе Linux.

Таблица 5.4. Примеры распределения интервалов времени

Приложение	Частота, Гц	Интервал, мс
Часы по умолчанию в Linux	100	10

Продолжение ⇩

Таблица 5.4 *(продолжение)*

Приложение	Частота, Гц	Интервал, мс
Видеокадр в стандарте вещания NTSC	60	16,67
Видеокадр в стандарте вещания PAL	50	20
Частота обновления монитора с электронно-лучевой трубкой (средняя)	80	12,5

Частоту таймера можно устанавливать только при сборке ядра. На рис. 5.2 показано, как это происходит, если сборка ядра осуществляется с использованием цели `menuconfig`.

Как уже отмечалось, частота «работы» часов, используемая функциями, возвращающими тип `clock_t`, не зависит от интервалов хода часов в ядре. Данная частота определяется макросом `USER_HZ` во время сборки ядра. Ее значение возвращается при вызове `sysconf(_SC_CLK_TCK)`. Вне зависимости от того, какое значение `HZ` установлено в ядре, данное значение останется неизменным. Как правило, разумно предполагать, что частота интервалов хода часов в ядре равна аналогичной пользовательской частоте или превышает ее.

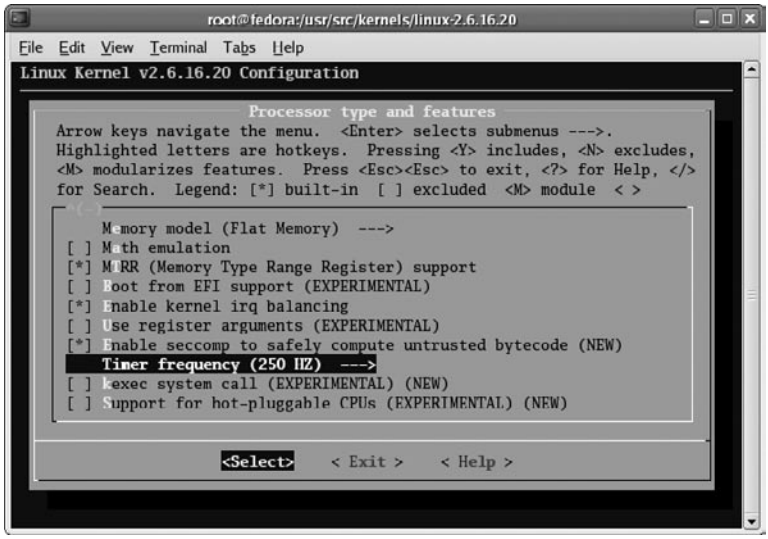


Рис. 5.2. Изменение частоты таймера во время сборки ядра

Измерение времени выполнения приложений

Оболочка `bash` содержит встроенные команды, позволяющие отслеживать производительность приложений, включая команду `time`, которая дает возможность осуществлять мониторинг загрузки процессора при выполнении какой-либо команды или сценария без необходимости вмешательства в программный код. Выводимое значение является результатом измерения времени, начиная с запуска процесса и заканчивая его завершением.

Если вы измеряете время выполнения приложения изнутри, то применяйте функцию getrusage, которая позволяет использовать флаги для указания того, какие данные вы желаете увидеть. Для этого функции getrusage передается первый аргумент, которым может быть RUSAGE_SELF или RUSAGE_CHILDREN. При этом, однако, значение времени, возвращаемое при использовании RUSAGE_CHILDREN, будет включать время выполнения только тех дочерних процессов, которые были уничтожены родительским процессом. До тех пор пока родительский процесс не осуществит вызов wait, возвращаемое значение времени выполнения дочерних процессов будет нулевым. Это не касается процессов, задействующих потоки. Поскольку поток не является дочерним процессом, время, затраченное на выполнение потоков, будет считаться временем, которое потребовалось на выполнение основного процесса. Выводимые при помощи getrusage значения будут повышенными, хотя потоки выполняются без необходимости совершения каких-либо дополнительных системных вызовов.

Измерять время выполнения приложения можно извне, при помощи оболочки и команды time. Она реализована в виде встроенной bash-функции и команды общего назначения в /usr/bin/time. Обе выполняют одну и ту же задачу, за тем исключением, что bash-версия фокусируется исключительно на измерении времени выполнения приложений, а команда time позволяет также просматривать сведения, извлекаемые при помощи системного вызова getrusage. Для того чтобы воспользоваться встроенной командой time, необходимо передать ей в качестве аргументов свою командную строку, как показано далее:

```
$ time sleep 1

real    0m1.007s
user    0m0.000s
sys     0m0.004s
```

Встроенную команду можно «доработать», *изменив* ее следующим образом:

```
$ \time sleep 1
0.00user 0.00system 0:01.00elapsed 0%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+199minor)pagefaults 0swaps
```

Данный вывод содержит намного больше информации из структуры rusage, включая поля, которые «не заполняет» операционная система Linux.

Обе использованные команды time выводят три значения: real (реальное время выполнения), user (время выполнения программного кода пользователя) и sys (время выполнения программного кода ядра). Последние две позиции мы уже рассматривали, а значение real, как вы уже догадались, представляет собой количество времени, которое прошло с момента запуска процесса и до его завершения. Это время выполнения вашего приложения, которое вы *ощутите*.

5.4. Понятие устройств и их драйверов

Каждому приложению рано или поздно приходится «общаться» с устройствами. Намеренно или нет, но трудно избежать контакта с одним или более устройствами в системе. Вы можете написать сложный алгоритм моделирования, который будет

выполняться исключительно в памяти компьютера, однако, если вы решите сохранить результаты, вам придется зафиксировать свои данные в файловой системе. Не исключено, что ваше приложение будет обращаться к файлу подкачки на диске из-за высокой нагрузки на систему. Любой вывод, который вы захотите отправить на консоль, скорее всего, потребует наличия драйвера псевдотерминала. Таким образом, как бы вы ни старались избежать их, запускаемые вами процессы все равно будут обращаться к драйверам устройств.

API-интерфейс драйверов устройств Linux берет свое начало в ранний период существования UNIX и с тех пор не претерпел значительных изменений. Стандарт POSIX формализует данный интерфейс и служит в качестве основы операционной системы Linux.

Многие устройства «открываются» подобно файлам на диске. Взаимодействие с устройствами начинается с момента открытия одного из таких файлов. Приложение использует дескриптор файла, возвращаемый посредством системного вызова `open`, а также все системные вызовы, которые принимают в качестве аргумента дескриптор файла.

5.4.1. Типы драйверов устройств

Драйверы устройств подразделяются на несколько основных категорий. Наиболее распространены *блочные* и *символьные* устройства, доступ к которым осуществляется посредством специальных файлов, размещенных на диске. Эти файлы называются *узлами*, что отличает их от обычных файлов и каталогов. Прочие устройства включают системные и сетевые драйверы, к которым приложение обычно не обращается напрямую и которые, однако, тесно взаимодействуют с другими драйверами, имеющимися в системе.

Символьные устройства

Типичным примером символьного устройства является последовательный порт компьютера или терминальное устройство, которые используется для ввода команд оболочки. Данные принимаются и отправляются по одному байту. При записи на устройство байты данных передаются в том же порядке, в котором идет их запись. Представьте, какая неразбериха может возникнуть, если буквы, которые вы набираете в терминале, будут выводиться в произвольном порядке. Аналогично при чтении с устройства байты данных принимаются в том же порядке, в котором они были отправлены.

Однако не все символьные устройства осуществляют запись и чтение таким образом. Символьные устройства охватывают широкий спектр аппаратного обеспечения и функций. Поскольку некоторые из таких устройств обеспечивают произвольный доступ к данным подобно устройствам хранения, их драйверы поддерживают дополнительные системные вызовы, например `mmap` или `lseek`[1].

[1] Некоторые драйверы «не признают» lseek, однако все же реализуют этот системный вызов. Вместо возврата значения –1 такие драйверы обычно возвращают значение 0, которое означает, что положение не изменилось, то есть технически данный системный вызов завершается без ошибок.

Драйвер mem является хорошим примером. Он обеспечивает функционирование ряда устройств, которые выполняют простые, свободно связанные функции, приведенные в табл. 5.5. Некоторые из этих устройств, например /dev/null, вообще не привязаны к какому-либо аппаратному обеспечению.

Блочные устройства

Блочное устройство — это устройство хранения с фиксированным объемом пространства. Как следует из его названия, оно оперирует блоками фиксированного размера. Главное назначение подобных устройств заключается во взаимодействии с дисковыми накопителями, хотя они могут использоваться также наряду с другими типами носителей информации, например с флеш-дисками. Если дисковый накопитель использует *логическую адресацию блоков* (Logical Block Addressing, — LBA), то между блоками на устройстве и логическими блоками на диске будет существовать взаимная корреляция. Уникальная особенность блочных устройств заключается в том, что на них могут размещаться файловые системы, что требует от них тесного взаимодействия с драйверами таких систем.

Сетевые устройства

В отличие от символьных и блочных устройств сетевые устройства не используют узлов устройств. Подобные устройства относятся к особому классу. Приложения редко обращаются к сетевым драйверам, если же это происходит, то они используют специальные имена вроде eth0, которые передаются функции ioctl посредством анонимного сокета. В этой книге мы не будем рассматривать сетевые устройства. Если вас интересует данный вопрос, можете обратиться к странице руководства man netdevice(7), которая станет отличным стартом для изучения сетевых устройств.

Драйверы файловой системы

Несмотря на то что файловая система технически не является устройством, ей требуется соответствующий драйвер. Драйверы файловых систем требуют отдельного блочного устройства. Хотя приложения постоянно взаимодействуют с файлами, они редко напрямую обращаются к драйверу файловой системы. Существует ряд исключений, касающихся определенных файловых систем. Так, например, файловая система XFS позволяет предопределять размеры файлов в целях увеличения производительности. Для этого используется функция ioctl в сочетании с дескриптором открытого файла в файловой системе.

5.4.2. Несколько слов о модулях ядра

Модуль ядра — это весьма популярное средство «доставки» драйверов устройств, однако *модуль ядра* и *драйвер устройства* не являются синонимами. Модули могут содержать любой программный код ядра, а не только драйверы устройств, несмотря на то что они чаще всего именно для этого используются. Привлекательность модулей заключается в том, что их можно компилировать после сборки ядра и устанавливать уже на выполняющемся ядре. Благодаря этому пользователи могут испытывать новые драйверы, не прибегая к пересборке ядра и не боясь загубить свою систему.

Модули ядра имеют расширение .ko (то есть *kernel object — объект ядра*) и могут помещаться непосредственно в ядро при помощи команды insert. Многие (но не все) модули можно удалить из ядра посредством команды rmmod. Если вы решите сохранить модуль, произведите перманентную инсталляцию при помощи следующей командной строки:

```
$ make -C /lib/modules/$(uname -r)/build M=$(pwd) modules_install
```

В результате ваш модуль будет размещен в соответствующем каталоге /lib/modules. В зависимости от модуля вам может потребоваться команда depmod, для того чтобы обновить его зависимости.

5.4.3. Узлы устройств

Доступ к блочным и символьным устройствам осуществляется, как к файлам на диске, посредством узлов устройств. Узел содержит целое число, которое служит индикатором старшего и младшего номеров. По традиции старший номер идентифицирует драйвер устройства в ядре, а младший номер используется драйвером для идентификации конкретных устройств. В версии Linux 2.4 и ниже значение, используемое для хранения старшего и младшего номеров, было 16-битным, при этом и для старшего, и для младшего номеров отводилось по 8 бит. В Linux 2.6 данное значение стало уже 32-битным, где 12 бит приходится на старший номер, а 20 бит — на младший.

Создание узлов на диске осуществляется при помощи команды mknod, которая принимает тип устройства (символьного или блочного), а также старший и младший номера в качестве аргументов. Из-за того что узлы устройств обеспечивают наличие интерфейса для драйверов устройств, могут возникнуть проблемы с безопасностью. В конце концов, вы же не хотите, чтобы любой пользователь имел доступ к блочному устройству, где располагается ваша корневая файловая система. В силу этих причин узлы устройств имеет право создавать только суперпользователь. Например, синтаксис создания /dev/mem, где используются старший номер 1 и младший номер 1, будет выглядеть следующим образом:

```
$ mknod /dev/mem c 1 1
```

Каталог /dev по умолчанию содержит все узлы устройств, имеющиеся в системе, при этом эти узлы могут создаваться в любой файловой системе[1].

При открытии процессом узла устройства ядро определяет местоположение соответствующего драйвера при помощи старшего номера. Младший номер, передаваемый драйверу, используется различными драйверами по-разному. Как правило, данный номер применяется для разграничения функций и устройств. Одним из наглядных примеров является драйвер mem, который реализует несколько различных функций на основе младшего номера (см. табл. 5.5). Доступ к каждой из функций осуществляется посредством узлов устройств. В принципе, из-за того,

[1] Узлы устройств в файловой системе можно сделать неиспользуемыми, применив при монтировании параметр nodev; см. mount(8).

что все узлы устройств имеют один общий старший номер, они будут связаны с одним и тем же драйвером.

Узлы определяются в каталоге /dev посредством имен, приведенных в табл. 5.5. Так, например, при открытии /dev/mem ядро вызывает функцию open драйвера mem с младшим номером 1. Это позволяет драйверу понять, что вы хотите увидеть физическую память, доступную в системе.

По общему правилу драйвер сам по себе не навязывает никаких политик доступа. Вместо этого он полагается на системный вызов open при проверке разрешений узла устройства, относящихся к текущему пользователю, так же как это делается в отношении всех прочих файлов в системе. Так, например, вы вряд ли захотите разрешить доступ к системной памяти всем пользователям, поскольку они могут воспользоваться этим устройством для того, чтобы «подглядывать» пароли или портить систему. В данной ситуации иметь право открывать /dev/mem и /dev/kmem должен только корневой пользователь *root*, а все остальные пользователи будут иметь доступ к большинству прочих функций устройства mem. Проверить это можно, взглянув на файловые разрешения узлов устройств.

Таблица 5.5. Символьные устройства, реализуемые драйвером mem

Имя узла устройства	Младший номер	Функция
mem	1	Предоставляет доступ к физической памяти
kmem	2	Предоставляет доступ к виртуальной памяти ядра
null	3	«Сточная труба» для данных. Все данные, записываемые на это устройство, удаляются
port	4	Предоставляет доступ к портам ввода/вывода (имеющимся на некоторых архитектурах)
zero	5	При чтении с данного устройства буфер заполняется нулями
	6	Устаревшее устройство /dev/core, замененное на /proc/kcore
full	7	Запись на это устройство всегда будет завершаться неудачей и выводом ENOSPC
random	8	При чтении с данного устройства буфер заполняется произвольными байтами; возвращаются только те байты, которые драйвер считает произвольными. Для получения подробных сведений см. random(4)
urandom	9	Аналог random, за тем исключением, что возвращает все запрошенные данные, независимо от того, являются ли они высококачественными произвольными данными. Более подробно см. random(4)
	10	Не поддерживается драйвером mem
kmsg	11	Позволяет приложениям вести запись в журнале сообщений ядра вместо использования системного вызова syslog

Несомненно, с точки зрения обеспечения должного уровня безопасности только корневой пользователь *root* должен иметь право создавать узлы устройств и изменять их разрешения. В противном случае любой пользователь получит возможность создать узел устройства, указывающий на /dev/mem или другое «уязвимое» устройство, и нанести вред системе. Кроме того, корневой пользователь *root* может воспрепятствовать распознаванию узлов устройств в файловых системах,

монтируемых пользователями, например /mnt/floppy, добавив параметр nodev к mount в /etc/fstab.

Младшие номера устройств

Младшие номера используются драйверами по-разному, при этом пользователю не всегда понятно как. Блочные устройства особенно сложны в этом смысле, поскольку младший номер служит уникальным идентификатором для конкретного накопителя и раздела.

Рассмотрим пример IDE-драйвера (/dev/hd). В версии Linux 2.6 принято использовать наименее значимые 6 бит для кодирования раздела, а 14 наиболее значимых бит — для кодирования накопителя. Это означает, что IDE-устройство поддерживает до 16 384 (214) накопителей, каждый из которых может иметь до 63 разделов (26 – 1); нулевой раздел будет использоваться для обращения ко всему накопителю. При обозначении узлов устройств дисков используются уникальные буквы, которые идентифицируют накопитель, а также десятичные числа, идентифицирующие раздел. При обращении ко всему диску в целом (раздел 0) номер раздела отбрасывается. В этом можно убедиться при помощи команды ls:

```
$ ls -l /dev/hd[ab]*
brw-------  1 john disk 3,  0 Dec 21 10:00 /dev/hda
brw-rw----  1 root disk 3,  1 Dec 21 03:59 /dev/hda1
brw-------  1 john disk 3, 64 Dec 21 10:00 /dev/hdb
brw-rw----  1 root disk 3, 65 Dec 21 03:59 /dev/hdb1
brw-rw----  1 root disk 3, 66 Dec 21 03:59 /dev/hdb2
```

Когда команда ls находит узел устройства, она выводит старший и младший номера в той колонке, где обычно высвечивается размер файла. В приведенном примере видно, что старший номер IDE-устройства равен 3, а метка первого диска имеет вид hda. Доступ к диску в целом осуществляется посредством узла устройства /dev/hda в сочетании со старшим номером 3 и младшим номером 0. Раздел 1 первого накопителя обладает узлом с младшим номером 1, который называется /dev/hda1. В нашем случае hda имеет только один раздел. Узел устройства второго накопителя называется hdb, при этом его младшие номера начинаются с 64. Подробности вы сможете узнать из табл. 5.6.

SCSI-драйвер подчиняется той же системе, что и IDE-драйвер, за тем исключением, что в данном случае для кодирования раздела используются только 4 бита младшего номера, позволяя иметь лишь 15 разделов. Для кодирования номера накопителя используются 16 бит, то есть SCSI-драйвер поддерживает до 65 536 (216) устройств. Для SCSI-устройств формула младшего номера будет такой: 16 × на накопитель + раздел.

Таблица 5.6. «Анатомия» младших номеров IDE-устройств (для версии Linux 2.6 и выше)

Накопитель	Раздел	Младший номер = 64 × на накопитель + раздел	Имя узла
0	0	0	/dev/hda
0	1	1	/dev/hda1

Накопитель	Раздел	Младший номер = 64 × на накопитель + раздел	Имя узла
1	0	64	/dev/hdb
1	1	65	/dev/hdb1
1	2	66	/dev/hdb2

Старшие номера устройств

Обычно старший номер в сочетании с типом драйвера (символьного или блочного устройства) идентифицирует один, и только один, драйвер устройства. Если, к примеру, старший номер присвоен определенному драйверу символьного устройства, этим номером не сможет воспользоваться никакой другой драйвер символьного устройства. Драйвер устройства, которому присвоен старший номер, «владеет» всеми младшими номерами независимо от того, нуждается он в них или нет. Так, например, драйвер `mem` обслуживает ряд псевдоустройств с одинаковым старшим номером. В противном случае каждому устройству пришлось бы отводить уникальный старший номер.

Это было серьезной проблемой в ядрах до появления версии Linux 2.6, в которых допускалось использование только 256 старших номеров в любой заданный момент времени. Несмотря на то что на самом деле никто не нуждается в большом количестве драйверов устройств в одной системе, проблема заключается в том, что многие старшие номера являются статически определяемыми, то есть они одинаковы для всех систем. Это ограничивает общее количество возможных устройств.

В Linux 2.6 данная проблема решается двумя путями. Первый из них, о котором мы говорили ранее, заключается в увеличении количества допустимых номеров до 1024. Второй состоит в том, что драйвер теперь имеет возможность регистрировать только диапазон младших номеров. Большое количество младших номеров будет кстати для драйвера диска, однако большинство драйверов не способны оперировать более чем несколькими младшими номерами.

Перечень перманентно присваиваемых старших номеров доступен в Интернете по адресу www.lanana.org/docs/devicelist. Он включается в каждый релиз исходных файлов ядра и располагается в файле `Documentation/devices.txt`.

Причина того, что драйверам вроде IDE присваиваются фиксированные старшие номера, заключается в согласованности. Фактически все чипсеты материнских плат архитектуры x86 поддерживают IDE-интерфейс. Представьте, что было бы, если бы каждый драйвер чипсета использовал произвольные старшие номера. Значения в `/dev` должны были бы быть уникальными для каждой конфигурации системы. К счастью, реальная ситуация является обратной и дистрибутивы могут создавать стандартный набор узлов в `/dev`, которые будут поддерживаться любой конфигурацией. Перманентное присваивание старших номеров, вероятно, сохранится навсегда.

Источники узлов устройств

Многие дистрибутивы на базе Linux 2.4 и ниже включают пакет, который содержит сотни или даже тысячи узлов устройств, извлекаемых в каталог `/dev`. Если среди этих узлов отсутствует тот, который описывает необходимое вам устройство,

вы можете самостоятельно добавить новый узел. Заглянув в данный каталог, вы обнаружите, что в нем присутствуют сотни узлов, указывающих на драйверы, которые вы не устанавливали и, скорее всего, не собираетесь устанавливать в будущем. Столь неэлегантный и грубый подход привел многих пользователей Linux на неверный путь и вызвал появление ряда альтернативных решений. Первым из них стал devfs, реализованный в виде драйвера файловой системы и используемый для создания псевдофайловой системы в /dev. Он динамически заполняет данный каталог лишь теми узлами, которые действительно присутствуют в системе. Таким образом, вместо тысяч узлов в каталоге /dev будут располагаться только те из них, которые имеют установленные в системе драйверы. Узлы создаются и удаляются по мере того, как новые устройства добавляются в систему и убираются из нее.

udev и hotplug. Функциональное свойство ядра hotplug прежде всего отвечает за определение местоположения и загрузки модулей драйверов для аппаратного обеспечения по мере того, как они подключаются и отключаются. Реализация hotplug полагается на минимальное вмешательство со стороны ядра; основная масса работы проделывается в пространстве пользователя. Ядро лишь определяет, когда аппаратные устройства становятся доступными и недоступными, и порождает процесс пространства пользователя, который обрабатывает соответствующее событие. Данный процесс распознает устройство, находит для него модуль драйвера и загружает этот модуль.

Основная функция udev заключается в заполнении каталога /dev узлами устройств, которые точно отражают аппаратные средства, доступные в системе на текущий момент. Вполне ожидаемо, что все это было реализовано в виде расширения hotplug[1].

Полную информацию о правилах udev вы сможете отыскать на странице руководства man udev(8).

sysfs. sysfs является новым функциональным ключом к реализации hotplug, который необходимо вкратце рассмотреть, поскольку позднее мы вновь с ним столкнемся. sysfs представляет собой основанную на памяти файловую систему вроде procfs, в которой содержатся текстовые файлы с системной информацией. Она базируется на объектах ядра *(kobjects — kernel objects)*, что является нововведением в ядре 2.6, поэтому sysfs недоступна в ядрах версии 2.4 и ниже.

Файловая система sysfs монтируется в каталоге /sys, благодаря чему приложения пространства пользователя смогут легко отыскать ее. Данная точка монтирования строго регламентирована, поскольку от нее зависят многие инструменты, то же как /proc имеет большое значение для файловой системы procfs. Файловая система sysfs во многом схожа с procfs, несмотря на то что она обладает интуитивно более понятным форматом и не привносит слишком много (в случае необходимости) дополнительного программного кода в модули и драйверы для обеспечения их поддержки.

procfs обычно содержит обыкновенные файлы с большим количеством информации, а sysfs — иерархию небольших файлов, в каждом из которых содержится минимальный объем сведений. В большинстве случаев сама структура каталогов позволяет получить представление о системе.

---

[1] См. сайт www.kernel.org/pub/linux/utils/kernel/hotplug/udev.html.

5.4.4. Устройства и операции ввода/вывода

Обычно перед тем, как данные попадут в буфер приложения, они проходят через пространство ядра. Например, при чтении с устройства данные копируются минимум два раза: один раз в буфер ядра, затем еще раз — в буфер пользователя. Такова цена, которую приходится платить за надежность и безопасность. Для того чтобы процесс не привел к краху ядра или другого процесса, все операции ввода/вывода должны обрабатываться ядром, которое служит контрольной точкой безопасности.

Операции ввода/вывода и символьные устройства

Дополнительное время, которое затрачивается на копирование данных через низкоскоростной последовательный порт, является несущественным. В случае с высокоскоростными устройствами подобные излишние операции копирования могут серьезно сказаться на производительности.

При использовании специального аппаратного обеспечения зачастую оптимальным выбором будет применение символьного устройства в качестве драйвера, поскольку оно является наиболее простым. Чтение и запись выполняются синхронно, что означает, что, если процесс вызывает `read` или `write`, он блокируется и не возвращается из системного вызова до тех пор, пока операция не будет завершена[1]. При записи на символьное устройство драйвер может осуществлять копирование данных напрямую из буфера пространства пользователя на устройство. Это означает, что драйвер не позволит вам продолжить работу, пока он не завершит эту процедуру. Ваш процесс блокируется и ожидает завершения процесса записи. Несмотря на то что время ожидания устройства можно потратить на выполнение других программ, обычно делать это не рекомендуется.

Устройства, к которым возможен произвольный доступ, зачастую поддерживают системный вызов `mmap`. Он позволяет приложению получать доступ ко всем данным на устройстве как к одной большой области памяти. Драйвер символьного устройства может поддерживать исключительно `mmap` и «не признавать» системных вызовов `read` и `write`. Если драйвер несовместим с `mmap`, данный системный вызов возвращает значение `MAP_FAILED`, а переменной `errno` присваивается значение `ENODEV`.

Блочные устройства, файловые системы и операции ввода/вывода

Блочные устройства являются основой для дисков и прочих устройств хранения, которые используются файловыми системами. По этой причине доступ к блочным устройствам может осуществляться двумя способами: напрямую или посредством файловой системы. Зачастую прямое обращение к блочному устройству необходимо лишь для его разбивки на разделы или для создания файловой системы.

Файловая система может создаваться на любом блочном устройстве или его разделе. Драйвер дисковода гибких дисков уникален тем, что он не позволяет работать с разделами, несмотря на то что дискеты их поддерживают. Вместо разделов этот драйвер использует старшие номера. С их помощью помечаются всевозможные типы дисководов для гибких дисков, которые появлялись и исчезали

[1] Подобное поведение является стандартным для большинства символьных устройств.

на протяжении эры существования персональных компьютеров. Большинства из них уже не существует, однако их поддержка сохраняется[1].

Для создания файловой системы используется команда `mkfs`, при этом необходимо указать тип системы при помощи параметра `-t`. Для того чтобы отформатировать дискету, которую можно будет использовать также, например, в операционной системе Windows, нужна такая команда:

```
$ mkfs -t vfat /dev/fd0
```

После того как на блочном устройстве создана файловая система, его можно монтировать в каталог при помощи команды `mount`:

```
$ mount -t vfat /dev/fd0 /mnt/floppy
```

Обычно команда `mount` самостоятельно определяет тип файловой системы, имеющейся на устройстве, поэтому параметр `-t` является опциональным. После того как блочное устройство смонтировано, к нему можно обращаться двумя способами: посредством узла устройства (например, `/dev/fd0`) или его точки монтирования (например, `/mnt/floppy`). При чтении с узла устройства вы будете получать необработанные («сырые») данные, которые включают все, что имеется в файловой системе. Это может показаться бессмысленным, однако может сослужить вам хорошую службу. Например, в случае проведения архивирования. Обычно подобная методика не очень эффективна для архивирования файловой системы, однако сброс дампа «сырого» устройства позволяет сохранить данные, чего не могут сделать утилиты архивирования вроде `tar`. Несмотря на то что программа `tar` может создавать архивы, включающие все файлы и каталоги, присутствующие в системе, сохраняя при этом все метаданные, она не способна сохранить загрузочный сектор, который не является частью файловой системы. Для того чтобы получить точную копию каждого байта с дискеты, включая загрузочный сектор, данные необходимо копировать с узла устройства.

Роль буферного кэша и кэша файловой системы

Одним из отличий блочных устройств от их символьных собратьев является то, что они используют в качестве кэша системную память. Linux поддерживает множество общих кэшей посредством структур данных в памяти, однако наибольший интерес для программистов представляют буферный кэш и кэш файловой системы[2].

Буферный кэш — это область, используемая для хранения блоков, которые считываются с блочных устройств или записываются на них. Когда процесс осуществляет запись на блочное устройство, данные сначала копируются в блок в буферном кэше. Блочный драйвер не вызывается до тех пор, пока ядро не решит, что пришло время записать этот блок на устройство, что может произойти через некоторое время. Ядро сохраняет копию каждого записанного или считанного блока в буферном кэше как можно дольше. Для физических устройств наподобие дисков это означает, что данные не сохраняются на диске в течение определенного времени после того, как завершается операция записи. Преимущество такого подхода заключается в том, что данные окажутся доступны любому процессу, который бу-

---

[1] См. страницу руководства man fd(4).

[2] Большинство накопителей имеют собственный аппаратный кэш, который «невидим» для ядра Linux.

дет осуществлять считывание с этого сектора диска позднее. Недостаток состоит в том, что, если система потерпит крах или пропадет электропитание до того, как данные будут записаны на устройство, они будут потеряны.

Кэширование увеличивает производительность несколькими путями. Один из них заключается в сохранении данных в памяти, в результате чего системе не приходится повторно считывать их с устройств наподобие дисковых накопителей, которые работают намного медленнее памяти. Это также позволяет ядру объединять смежные блоки данных, записанных в кэш, в одну большую операцию записи на диск вместо нескольких небольших, что обычно позволяет более эффективно использовать дисковый накопитель. Кэш также позволяет снизить количество избыточных операций записи на диск, поскольку, если блок обновляется до записи на диск, ядру приходится выполнять только одну процедуру записи на диск вместо двух. Все это обеспечивается ценой дополнительных операций копирования, затраты времени на которые во многих приложениях окажутся незначительными по сравнению с потерями времени, связанными с неэффективным использованием дискового накопителя.

Увидеть кэш «в действии» можно при помощи команды `vmstat`:

```
$ vmstat
procs -----------memory---------- ...
 r  b   swpd   free      buff       cache
 1  0      0  93412      5736       38096

                                        Запись 4 Мбайт на устройство ramdisk.
$ dd if=/dev/zero of=/dev/ram0 bs=1k count=4096
4096+0 records in
4096+0 records out

$ vmstat
procs -----------memory---------- ...
 r  b   swpd   free      buff       cache
 0  0      0  89272      9832       38096
                                     4 Мбайт добавлены в буферы.
```

Команда `vmstat` позволяет выводить и другие сведения, однако в данный момент мы сосредоточимся на информации, касающейся использования памяти[1]. Здесь мы копируем 4 Мбайт с `/dev/zero` (символьного устройства) на `ramdisk`-устройство `/dev/ram0` (блочное устройство). Поскольку мы непосредственно взаимодействуем с блочным устройством, нам необходимо выделить в буферном кэше область хранения, в которой будут размещаться записываемые данные. В выводе команды `vmstat` видно, что размер буферного кэша увеличился с 5736 Кбайт до 9832 Кбайт, то есть точно на 4096 Кбайт. Особенность устройства `ramdisk` заключается в том, что после того, как ему отводится определенная область памяти, оно ее занимает навсегда, из-за чего буферы продолжают «отображаться» в буферном кэше. Однако не все блочные устройства поступают подобным образом.

В приведенном далее примере показано, что произойдет, если создать на этом устройстве файловую систему с последующим монтированием:

---

[1] См. также free(1), что входит в состав пакета procps. См. также /proc/meminfo.

```
$ mkfs -t ext2 /dev/ram0            Создание файловой системы.
mke2fs 1.37 (21-Mar-2005)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
4096 inodes, 16384 blocks
...

$ vmstat                            Базовый уровень использования кэша.
procs -----------memory----------
 r  b   swpd   free     buff     cache
 1  0      0  88792    10164     38272

$ mount /dev/ram0 /mnt/tmp          Монтирование файловой системы.

$ vmstat                            Значительного увеличения степени
procs -----------memory----------   использования кэша не произошло.
 r  b   swpd   free     buff     cache
 0  0      0  88792    10168     38272
                                    Создание в файловой системе файла
                                    размером 2 Мбайт.
$ dd if=/dev/zero of=/mnt/tmp/zero.dat bs=1k count=2048
2048+0 records in
2048+0 records out

$ vmstat                            Уровень использования кэша файловой
procs -----------memory----------   системы возрастает примерно на 2 Мбайт.
 r  b   swpd   free     buff     cache
 1  0      0  86572    10200     40324
```

Созданный нами файл будет находиться в кэше до тех пор, пока не произойдет одно из следующих событий.

- Он будет «вытолкнут» более новыми данными, помещаемыми в кэш.
- Данный файл будет удален.
- Файловая система будет размонтирована.
- Ядро сбрасывает этот файл на диск, для того чтобы освободить память для процессов.
- Приложение явным образом сбрасывает этот файл на диск при помощи `sync` или `fdatasync`.

До тех пор пока не произойдет одно из этих событий, наш файл не будет записан непосредственно на диск.

Еще один важный момент, на который следует обратить внимание, заключается в том, что кэши соперничают с процессами за системную память. С точки зрения процесса буферный кэш и кэш файловой системы представляют собой свободные области памяти, которые могут быть очищены, для того чтобы освободить память для других процессов. Система, в которой протекают интенсивные операции вво-

да/вывода, может использовать бóльшую часть системной памяти в качестве буферного кэша или кэша файловой системы. Если процессам требуется больше памяти, чем свободно в данный момент, системе приходится искать пути, чтобы как-то высвободить необходимую память. Для того чтобы сделать это, ядро может регенерировать кэш, сбросив блоки на диск.

В нормальных условиях ядро берет свободную память из кэша, сбрасывая самые «старые» блоки непосредственно на диск. Блоки, которые относятся к файлу, записываются на диск, после чего ядро регенерирует память. Так называемые «чистые» блоки могут быть регенерированы непосредственно, без необходимости в операциях ввода/вывода. *«Чистый» блок* — это блок, который был считан с диска и не подвергся модификации, либо блок, который был записан на диск, но не регенерирован. *«Грязный» блок* — это модифицированный (или созданный) блок, изменения в котором еще не были записаны на диск.

Сравнение устройства ramdisk и файловой системы tmpfs

Особенность устройства `ramdisk` состоит в том, что оно не позволяет регенерировать отведенную ему память, то есть его блоки всегда будут занимать свободную память до перезагрузки системы. Поскольку блоки в буферном кэше, которые оно использует, никогда не возвращаются в систему, размеры устройства `ramdisk` нельзя изменить, так же как и удалить его. Это позволяет выполнять размонтирование и перемонтирование устройства `ramdisk` без потерь данных.

Недостаток данного устройства заключается в том, что при создании на нем файловой системы оно начинает потреблять как буферный кэш, так и кэш файловой системы, то есть теоретически оно может «поглотить» вдвое больше оперативной памяти, чем файловая система на диске. Поэтому вместо устройства `ramdisk` большинство приложений, которым требуется временная область хранения в оперативной памяти, используют файловую систему `tmpfs`.

Файловая система `tmpfs` уникальна тем, что ей не требуются блочные устройства для хранения данных. В `tmpfs` данные располагаются целиком в кэше файловой системы. Поскольку данная область памяти также может использоваться для подкачки, вы одновременно получаете в свое распоряжение преимущества высокоскоростного устройства `ramdisk` и гибкость виртуальной памяти.

`tmpfs` является файловой системой по умолчанию для устройства с разделяемой памятью (`/dev/shm`) фактически в любом дистрибутиве.

Как ядро управляет кэшем файловой системы

Обычно ядро полагается на пользовательские процессы при выполнении программных операций, необходимых для обслуживания системы. Так как подобный подход ненадежен в отношении процедур, которые должны осуществляться периодически, в большинстве систем имеются процессы-демоны, которые отвечают за критически важные функции и выполняются в фоновом режиме. Одним из таких

демонов является pdflush, который следит за тем, чтобы данные слишком долго не задерживались в кэше и записывались на диск.

Потоки ядра

Уникальной особенностью демона pdflush является то, что он технически является не процессом, а потоком ядра. Как вы уже знаете, процесс существует, так сказать, в двух «мирах»: пространстве пользователя и пространстве ядра. Поток ядра — это процесс, который не имеет пространства пользователя и выполняется целиком в пространстве ядра. В силу этих причин программный код этого потока ядра должен полностью располагаться в ядре. В отличие от демона пространства пользователя, который обычно запускается процессом init посредством исполняемого файла, имеющегося на диске, поток ядра активируется непосредственно ядром при помощи функций, определенных в ядре, и не имеет исполняемого файла.

Потоки ядра выглядят и ведут себя так же, как обычные процессы, однако отсутствие у них пространства пользователя выдает их. Выявить поток ядра можно, заглянув в файл /proc/PID/maps, который для нормального процесса содержит карту виртуальной памяти. Из-за отсутствия пространства пользователя файл maps потока ядра всегда будет пуст.

Ядро может заблаговременно записывать блоки на устройство, если необходимо высвободить определенный объем памяти. В данном случае сначала регенерируются самые «старые» блоки, или, точнее говоря, «наиболее давно использовавшиеся». Поскольку «грязные» блоки, скорее всего, использовались недавно, ядро вряд ли станет сбрасывать их на диск, однако это возможно в системах с большим числом операций ввода/вывода. В данной ситуации демон pdflush может вообще не запускаться. Когда такое происходит, работа по записи «грязных» блоков ложится на текущий процесс.

Приложения также могут форсировать заблаговременную запись блоков определенного файла посредством системных вызовов fsync, fdatasync и sync. Команда sync позволяет пользователям осуществлять системный вызов sync из оболочки. Эти системные вызовы дают возможность программам (или пользователям) более тесно управлять кэшем файловой системы путем форсирования операций ввода/вывода в определенные моменты времени, вместо того чтобы ожидать запуска демона pdflush.

5.5. Планировщик операций ввода/вывода

При управлении операциями ввод/вывода весьма важную роль играет их планировщик. При записи блоков на устройство или чтении с него соответствующие запросы ставятся в очередь и завершаются позднее. Каждое блочное устройство имеет собственную очередь. Планировщик операций ввода/вывода отвечает за сортировку этих очередей в целях обеспечения максимально эффективного использования дискового накопителя. Для жесткого диска это особенно важно, поскольку это позволяет сократить чрезмерное количество перемещений головок,

которые являются наиболее затратными с точки зрения времени операциями в любой системе. Даже в случае с накопителями другого типа, например флеш-дисками, упорядочение операций ввода/вывода позволяет организовать наиболее эффективное использование этих устройств.

В Linux алгоритм планирования операций ввода/вывода по умолчанию называется *элеваторным* алгоритмом, так как задача по планированию операций записи и чтения с диска весьма схожа с планированием остановок элеватора (лифта). Головки жестких дисков движутся взад и вперед по дорожкам почти так же, как лифт в здании движется вверх и вниз. Подобно тому как лифт останавливается на разных этажах, принимая и высаживая пассажиров, головки жесткого диска останавливаются у цилиндров, для того чтобы считывать и записывать данные. Задача по планированию будет одинаковой в обоих случаях. Если запросы просто будут обрабатываться в том же порядке, в котором они поступают, результатом будет снижение скорости работы соответствующего устройства.

5.5.1. Элеватор Linus Elevator (также называемый noop)

До появления версии Linux 2.4 существовал только один планировщик операций ввода/вывода, который назывался Linus Elevator. Данный планировщик осуществляет сортировку запросов ввода/вывода так же, как обычный элеватор, то есть по мере поступления запросов он объединяет смежные запросы, что позволяет собирать в одну группу все запросы, касающиеся одного и того же сектора дискового накопителя. Запросы, которые не поддаются объединению с каким-либо из запросов ввода/вывода, ставятся в конец очереди.

Это довольной простой алгоритм, однако он не лишен недостатков: объединение запроса с уже имеющимся запросом приводит к тому, что он перемещается в начало очереди. То есть если непрерывно поступающие новые запросы будут объединяться с запросом, находящимся в начале очереди, то запрос, располагающийся в ее конце, будет удерживаться там неопределенно долгое время. Когда такое происходит, мы говорим, что запрос *откладывается*.

Linus Elevator предпочитает откладывать запросы чтения, а не записи, поскольку запросы записи проходят легче, чем запросы чтения. Благодаря кэшу файловой системы процессу не приходится ожидать завершения одной операции записи, перед тем как он сможет выполнять следующую такую операцию. Системный вызов `write` копирует данные в кэш, а операция записи планируется к завершению. Так как каждый запрос может быть объединен только со своим предшественником, запросы записи могут быстро загромоздить очередь ввода/вывода. Для сравнения: процессу, осуществляющему чтение из файла, приходится ожидать завершения всех операций чтения, прежде чем он сможет выполнить следующую подобную операцию. Между запросами чтения может быть интервал в несколько миллисекунд, а за это время может поступить множество новых запросов, которые отложат выполнение следующего запроса чтения. Таким образом, процесс, осуществляющий запись больших блоков данных на дисковый накопитель (что встречается довольно часто), может монополизировать это устройство. Запросы записи больших

блоков будут объединяться, что позволит постоянно отодвигать любые ожидающие запросы чтения в конец очереди.

Это может показаться странным, но приоритет процесса не влияет на расположение его запросов в очереди ввода/вывода. Процесс с высоким приоритетом не получает никаких привилегий со стороны планировщика операций ввода/вывода. Причина этого заключается в том, что, поскольку данные располагаются в кэше файловой системы, процесс, завершающий операции ввода/вывода, может оказаться уже не тем процессом, который осуществлял запись этих данных в кэш. В силу этого планировщик операций ввода/вывода не может определить уровень приоритета текущего процесса.

Для решения этих проблем в версию Linux 2.6 была добавлена функция перезаписи блочного слоя ввода/вывода. Linus Elevator по-прежнему доступен и называется «*планировщик* noop», однако теперь для выбора доступны сразу три разных планировщика операций ввода/вывода и вы можете сами решить, какой из них вам больше подходит.

5.5.2. Планировщик операций ввода/вывода «крайнего срока» (Deadline I/O Scheduler)

Данный планировщик сортирует и объединяет запросы точно так же, как планировщик noop, за тем исключением, что запросы также сортируются по дате и по соответствующей области дискового накопителя. Этот планировщик гарантирует, что запросы будут обслуживаться в течение фиксированного отрезка времени (крайнего срока). Пользователь может сам определять крайний срок, однако по умолчанию крайний срок обслуживания запросов чтения будет короче крайнего срока обслуживания запросов записи. В результате запросы записи не приводят к откладыванию запросов чтения, как это делает планировщик noop.

5.5.3. «Ожидающий» планировщик операций ввода/вывода (Anticipatory I/O Scheduler)

Это новый встроенный планировщик, который по своей сути идентичен предыдущему планировщику, за тем исключением, что после завершения последней операции записи он ждет 6 мс, прежде чем продолжить обработку следующих запросов ввода/вывода. Таким образом, он *ожидает* поступления нового запроса чтения от приложения. Это позволяет увеличить скорость чтения за счет некоторого снижения скорости записи.

5.5.4. «Справедливый» планировщик очереди операций ввода/вывода (Complete Fair Queuing I/O Scheduler)

Это новейший планировщик операций ввода/вывода, который был включен в ядро Linux. Он присваивает запросам ввода/вывода приоритеты, во многом ана-

логичные приоритетам процессов. Приоритет ввода/вывода, которым наделяется запрос, не зависит от приоритета процесса, поэтому запросы чтения и записи, поступающие от процесса с высоким приоритетом, автоматически не наследуют высокий приоритет ввода/вывода.

5.5.5. Выбор планировщика операций ввода/вывода

В Linux 2.4 и предыдущих версиях 2.6 для формирования очередей ввода/вывода был доступен только один планировщик, который необходимо было выбирать во время загрузки при помощи параметра `elevator`. Он передавался ядру в качестве загрузочного параметра (и располагался обычно в файле `lilo.conf` или `grub.conf`). На момент написания данной книги ядро поддерживало параметры, приведенные в табл. 5.7, которые также можно найти в файле `Documentation/kernel-parameters.txt`.

В более поздних версиях 2.6 необходимость использования параметра `elevator` во время загрузки отпала. Теперь можно было выбирать планировщика отдельно для каждого блочного устройства и менять его «на лету». Узнать, какой планировщик используется в настоящий момент для конкретного блочного устройства, можно, заглянув в `/sys/block/{устройство}/queue/scheduler`, например:

```
$ cat /sys/block/hdb/queue/scheduler
noop [anticipatory] deadline cfq
```

Таблица 5.7. Планировщики операций ввода/вывода, доступные в Linux 2.6

Параметр	Описание
noop	Элеватор Linus Elevator из версии Linux 2.4 и ниже. Осуществляет сортировку запросов и объединение новых смежных запросов с целью минимизации количества перемещений головок жесткого диска
deadline	Аналог noop, за исключением того, что форсирует обслуживание запросов ввода/вывода в течение фиксированного отрезка времени (крайнего срока)
as	«Ожидающий» планировщик операций ввода/вывода (Anticipatory I/O Scheduler); аналог deadline, за исключением того, что между обработкой запросов чтения выдерживается пауза 6 мс
cfq	«Справедливый» планировщик очереди операций ввода/вывода (Complete Fair Queuing I/O Scheduler); аналог deadline, за исключением того, что запросам ввода/вывода присваиваются приоритеты точно так же, как процессам

В приводившемся ранее примере было видно, что в случае с устройством `hdb` используется «ожидающий» планировщик операций ввода/вывода (Anticipatory I/O Scheduler). Для этого устройства можно выбрать другой планировщик, записав в соответствующий файл другой параметр. Например, для того, чтобы сменить его на планировщик `cfg`, потребуется такая команда:

```
$ echo cfq > /sys/block/hdb/queue/scheduler
```

Выбор планировщика зависит от приложения. Приложениям реального времени, работающим с дисковым накопителем, требуется либо планировщик `deadline`, либо `cfq`. Что касается встраиваемых систем, работающих с RAM-устройствами и флеш-дисками, то им вполне подойдет планировщик `noop`.

5.6. Управление памятью в пространстве пользователя

Одной из примечательных особенностей такой операционной системы с защищенной памятью, как Linux, является то, что разработчикам не нужно беспокоиться о том, в какой области располагается их программный код или что именно является источником памяти. Все распределяется по своим местам без вмешательства со стороны программиста. Большинство программистов не задумываются о том, как много скрытых от их глаз процессов протекают на уровне ядра, библиотек и загрузочного кода.

В данном разделе мы сосредоточимся на 32-битных процессорах, которые имеют уникальные особенности взаимодействия с приложениями, работающими с большими наборами данных. На момент написания этой книги 32-битные процессоры были основой для платформ, которые использовались для установки операционной системы Linux. Что касается 64-битных процессоров, то здесь некоторые проблемные моменты будут аналогичными, но уже в других рамках. Границы применения 64-битных процессоров достаточно обширны для того, чтобы охватывать большинство сложных аспектов, которые наблюдаются у 32-битных процессоров и будут актуальны в обозримом будущем.

5.6.1. Понятие виртуальной памяти

Ключевая концепция *виртуальной памяти* заключается в том, что здесь адреса памяти, используемые приложениями, не имеют ничего общего с физическим расположением данных. Данные, относящиеся к какому-либо приложению, могут вообще не занимать физической памяти, а сохраняться на диске *(подкачки)*, для того чтобы другие процессы могли использовать эту память для своей работы. То, что может выглядеть как цельный блок данных в виртуальной памяти, скорее всего, окажется записанным в виде мелких фрагментов в множестве областей физической памяти или на диске подкачки. Наглядный пример этого можно увидеть на рис. 5.3.

Благодаря использованию виртуальной памяти операционная система Linux обеспечивает каждый процесс уникальными для него данными, защищенными от других процессов. Каждый процесс выполняется как единственный запущенный в системе. В пространстве пользователя адрес А одного процесса будет указывать на совершенно иные области физической памяти, чем адрес А другого процесса. Каждый раз, когда центральный процессор загружает данные в память или сохраняет их, виртуальный адрес, используемый программой, преобразуется в физический адрес. Преобразование виртуальных адресов в физические осуще-

ствляет аппаратный блок управления памятью MMU (Memory Management Unit).

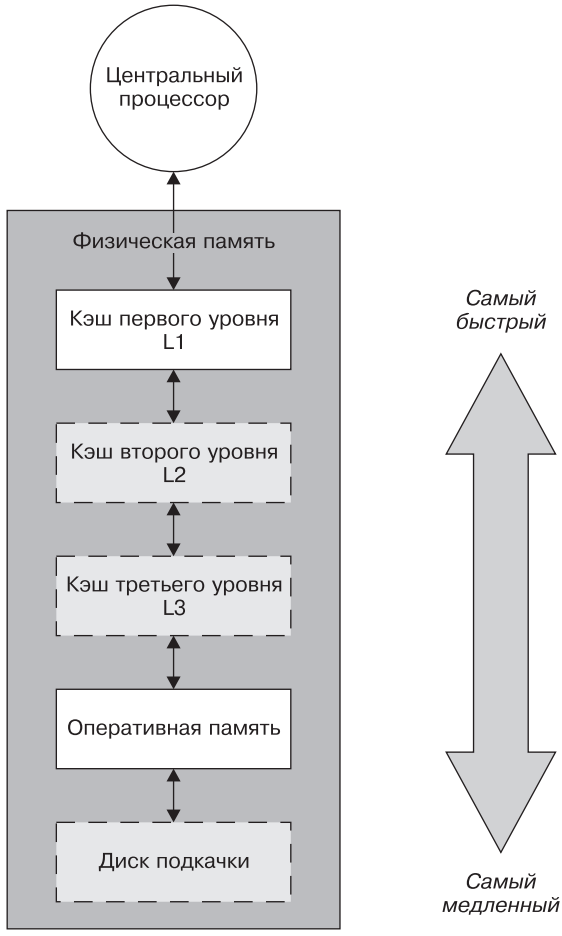


Рис. 5.3. Физическая память, какой ее «видит» центральный процессор

Роль аппаратного блока управления памятью MMU

Аппаратный блок управления памятью MMU работает в тесном сотрудничестве с кэшами, перемещая данные между оперативной памятью и кэшем по мере необходимости. Если ваш процессор имеет кэш, то он также будет иметь и MMU-блок, и наоборот. Современные настольные процессоры обладают определенным объемом встроенного кэша и аппаратным блоком управления памятью MMU.

При преобразовании адресов MMU-блок разделяет память на страницы, которые представляют собой минимальные блоки физической памяти, которыми он манипулирует. Для того чтобы преобразовать виртуальный адрес в физический, MMU-блок разбивает его на две части: номер фрейма страницы и смещение, как показано на рис. 5.4.

Размер страницы зависит от используемой архитектуры, однако 4 Кбайт является довольно распространенным размером для многих архитектур, включая PowerPC и IA32. В нашем случае номер фрейма страницы составляет 20 бит, а смещение — 12 бит.

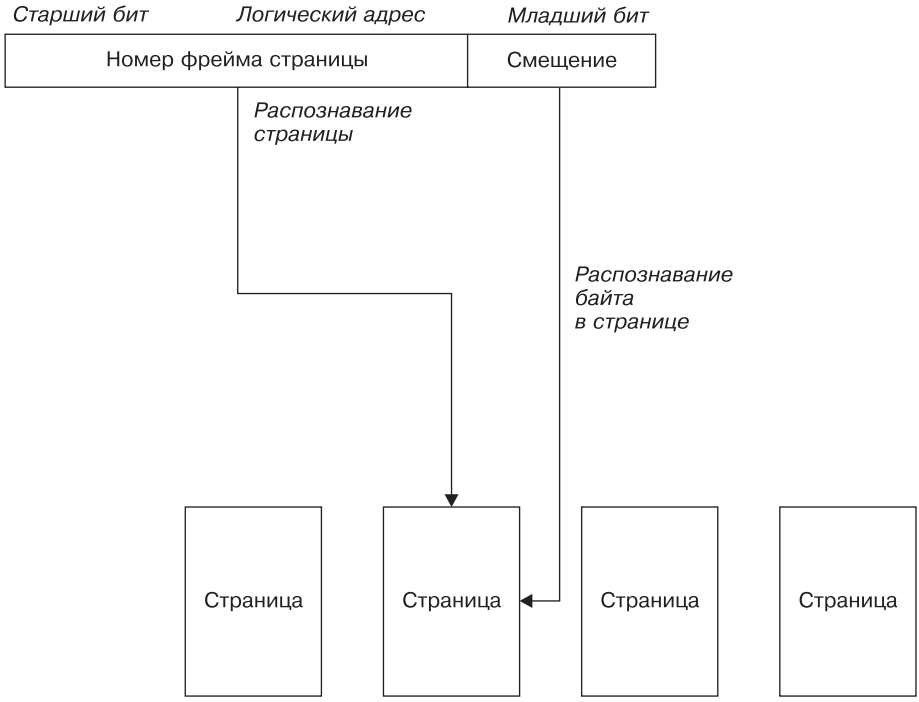


Рис. 5.4. Логический адрес, разбиваемый на номер фрейма страницы и смещение

Располагая номером фрейма страницы, MMU-блок может определить ее физический адрес, используя *таблицу страниц*, создаваемую ядром. Смещение берется из виртуального адреса и добавляется в физический адрес страницы, для того чтобы сгенерировать полный физический адрес.

Всякий раз, когда центральный процессор загружает инструкцию или сохраняет ее в памяти, MMU-блок преобразует виртуальный адрес. Если MMU-блок не находит нужный адрес в таблице страниц, то возникает *ошибка страницы*. Такое случается, когда страница отсутствует в памяти или когда процесс использует неверный логический адрес. Если ошибка страницы вызвана неверным адресом, ядро посылает процессу сигнал *нарушения сегментации* (`SIGSEGV`). Ошибки страницы могут происходить также, когда запрашиваемая страница расположена на диске подкачки. В такой ситуации ядро должно обратиться к дисковому устройству для извлечения этой страницы в физическую память и обновления таблицы страниц таким образом, чтобы она указывала на новую область в физической памяти.

Ошибки страницы, связанные с дисковыми операциями ввода/вывода, в Linux называются *значительными* ошибками страницы. Операционная система Linux

отслеживает также так называемые *незначительные* ошибки страниц, которые случаются, когда запрашиваемая страница располагается в физической оперативной памяти, а не во встроенном в центральный процессор кэше. В результате латентность системы немного повышается, поскольку ей приходится перемещать эту страницу из оперативной памяти в кэш процессора. Однако из-за того, что все это происходит полностью на аппаратном уровне, данная ошибка страницы будет считаться незначительной.

Ассоциативный буфер преобразования TLB (Translation Lookaside Buffer)

Каждый процесс в системе обладает собственным виртуальным адресом, поэтому каждый из них должен также обладать уникальной таблицей страниц. Важный аспект переключения контекста с одного процесса на другой связан с таким изменением таблиц страниц, чтобы они указывали на соответствующую область в виртуальной памяти. На самом деле данная процедура довольно сложна, однако я расскажу вам о ней подробнее.

Каждый раз, когда центральный процессор обращается к памяти, MMU-блок преобразует соответствующий адрес при помощи таблиц страниц до того, как он завершит данную операцию, однако таблицы страниц могут располагаться и в самой памяти. Это означает, что в наихудшем случае каждая операция загрузки или сохранения потребует проведения двух транзакций памяти: чтения из таблицы страниц и последующей непосредственной загрузки или сохранения.

Если бы таблицы страниц располагались исключительно в памяти, это привело бы к существенному уменьшению скорости работы системы. Таблицы страниц могут разрастаться до довольно больших размеров, и, поскольку операционная система поддерживает неограниченное количество запускаемых процессов, размещение таблиц страниц полностью во встроенном в процессор кэше также не является выходом из этой ситуации.

В качестве компромисса центральный процессор сохраняет кэш элементов таблиц страниц в так называемом *ассоциативном буфере преобразования* TLB (Translation Lookaside Buffer). TLB-буфер позволяет процессам использовать обширную область памяти, а критически важная информация о преобразовании адресов при этом сохраняется в кэше, встроенном в процессор. TLB-буфер должен иметь размер, достаточный для того, чтобы полностью покрывать кэш процессора. Таким образом, процессору с 512 Кбайт кэша и страницей размером 4 Кбайт для эффективной работы потребуется 128 TLB-элементов. На рис. 5.5 вы можете увидеть, как процессор преобразует виртуальный адрес при помощи TLB-буфера.

Так как TLB-буфер содержит кэш элементов таблиц страниц, относящихся к текущему процессу, то следует ожидать, что его содержимое будет сбрасываться на диск при переключении контекста. Сброс содержимого TLB-буфера на диск и его повторное заполнение является довольного ресурсоемкой операцией, поэтому ядро любой ценой избегает сброса содержимого этого буфера на диск, для того чтобы обеспечивать быстрое переключение контекста. Данная процедура иногда называется *ленивым сбросом содержимого TLB-буфера на диск*.

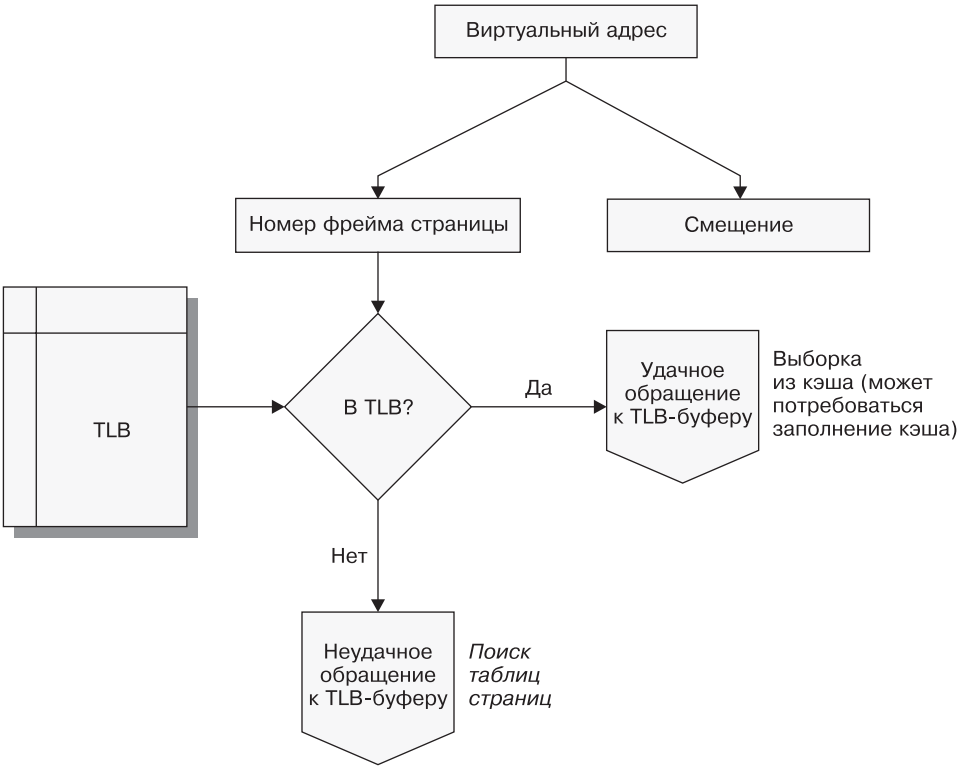


Рис. 5.5. Диаграмма использования ассоциативного буфера преобразования TLB

Кэш центрального процессора

Скорость работы процессоров намного превосходит скорость работы DRAM-устройств, поэтому все современные процессоры обладают определенным объемом встроенной кэш-памяти, позволяющей им функционировать на высоких частотах, избегая замедления из-за влияния DRAM-устройств.

Самый «близкий» к процессору кэш называется *кэшем первого уровня L1*, он располагается внутри и обычно имеет сравнительно небольшой объем (в среднем от 8 до 32 Кбайт), однако работает с нулевой латентностью. Это означает, что операции загрузки или сохранения в этот кэш выполняются за один тактовый цикл. Иначе говоря, кэш первого уровня L1 работает на одной частоте с процессором. На некоторых архитектурах данный кэш может быть единственным, который присутствует в процессоре. Так, например, самые дешевые версии процессоров x86 имеют только кэш первого уровня L1.

В целях увеличения размера кэша процессоры многих архитектур стали снабжаться дополнительным уровнем кэш-памяти, который больше по объему, но работает медленнее. Этот кэш называется *кэшем второго уровня L2*, он превышает по размеру кэш первого уровня L1, хотя и имеет более высокую латентность, что приводит к тому, что операции загрузки и сохранения выполняются более чем за один тактовый цикл.

После появления в процессорах кэша второго уровня L2 производители придумали новый термин — *кэш третьего уровня L3*, который позволяет обращаться к кэш-памяти, располагающейся вне пределов центрального процессора. Не так давно компания Intel стала встраивать кэш третьего уровня L3 в свои процессоры Xeon. Возможно, данная тенденция сохранится и в будущем мы увидим процессоры с кэшем четвертого L4 и пятого L5 уровней.

Строки кэша. Центральный процессор никогда не считывает из оперативной памяти и не записывает в нее байты данных или даже целые слова. Каждая операция чтения или записи из процессора в оперативную память сначала проходит в кэше первого уровня L1, который считывает и записывает данные в оперативную память строками. Строка кэша является единицей измерения всех транзакций, выполняемых с оперативной памятью. Несмотря на то что стандартная страница виртуальной памяти может иметь размер 4 Кбайт, размер стандартной строки кэша может быть равен 32 или 62 Кбайт. Размер страницы и строки кэша является уникальным для каждой марки и модели используемого процессора.

Далее мы рассмотрим упрощенную диаграмму того, как все это работает. При выполнении обычной строки кода, которая считывает единичный байт из памяти, процессору может потребоваться считать всю строку кэша (возможно, равную 64 Кбайт). Если последующие инструкции будут считываться из той же строки кэша, то заполнение данной строки имело смысл; в противном случае дополнительные ресурсы будут напрасно потрачены на данную операцию. Однако «неудачи» кэша первого уровня L1 не всегда приводят к таким последствиям. Данные также могут располагаться в кэше второго L2 или третьего L3 уровня, а в этом случае заполнение строки будет осуществляться намного быстрее считывания из оперативной памяти. Обычно память, установленная на материнской плате, устроена таким образом, что при пакетном чтении из нее поступает объем данных, равный строке кэша. В этом случае заполнение строк кэша из оперативной памяти будет максимально эффективным.

Даже если бы при выполнении программного кода, приведенного на рис. 5.6, осуществлялась операция записи в память, порядок ее выполнения был бы аналогичным, то есть при записи единичного байта данных вам пришлось бы заполнять всю строку кэша целиком. При обратной записи этой строки кэша в память центральный процессор запишет полностью всю строку, даже если изменению подвергся только один байт данных.

Все это может показаться незначительным, если вы работаете на компьютере, в котором установлен мощный процессор с тактовой частотой в 3 ГГц, однако и ему придется тяжело потрудиться, особенно при обработке больших объемов данных.

Обратная запись, сквозная запись и упреждающая выборка. Кэши функционируют в различных режимах, при этом каждая процессорная архитектура имеет свои особенности. Общими для них являются следующие базовые режимы.

- **Обратная запись** — это наиболее распространенный режим максимальной производительности. В данном режиме кэш не записывает содержащиеся в нем данные в память до тех пор, пока более новый элемент кэша или программа не сбросит их на диск. Это повышает производительность, поскольку процессору

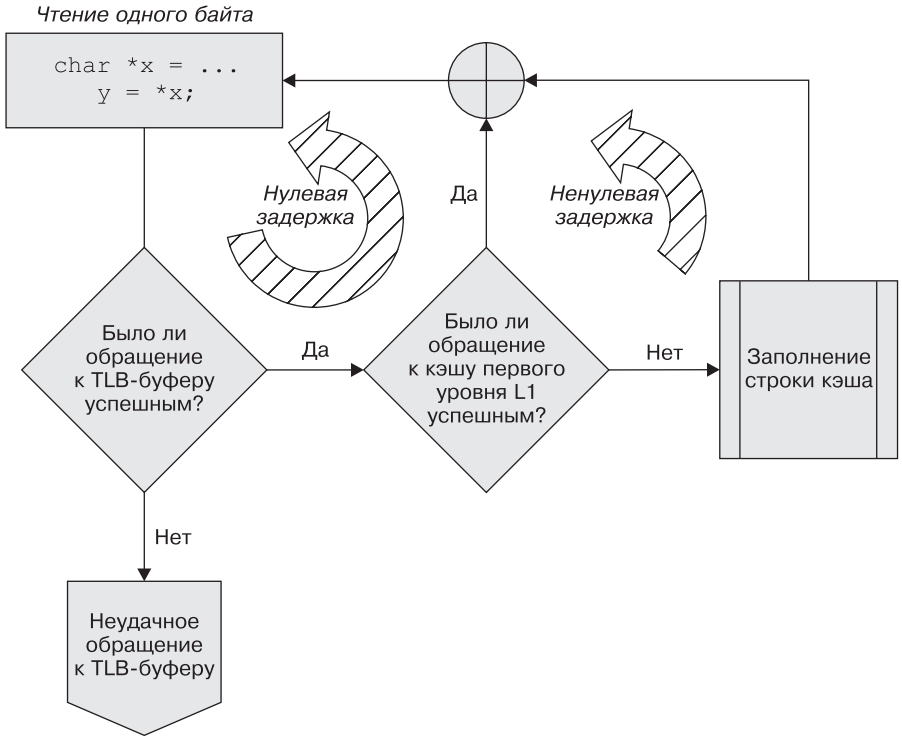


Рис. 5.6. «Неудача» кэша процессора: считывание единичного байта данных может привести к заполнению строки кэша

не нужно выполнять дополнительные операции записи, когда строка кэша подвергается модификации более чем один раз. Кроме того, несмотря на то что запись строк кэша может осуществляться в произвольном порядке, их сброс на диск должен выполняться в определенной последовательности, что также может положительно сказаться на производительности. Эта процедура иногда называется *комбинированной записью* и поддерживается не всеми процессорными архитектурами[1].

- **Сквозная запись** — менее производительный режим, чем обратная запись, поскольку в нем форсируется запись данных не только в кэш, но и в оперативную память. Из-за этого операции записи занимают больше времени, однако чтение из кэша по-прежнему остается быстрым. Данный режим используется, когда необходимо, чтобы в основной памяти и в кэше постоянно содержались одинаковые данные.
- **Упреждающая выборка** — некоторые кэши позволяют процессору осуществлять упреждающую выборку строк кэша в ответ на запросы чтения, то есть считывание смежных блоков памяти происходит одновременно. Пакетное чте-

[1] Комбинированная запись схожа с объединением запросов ввода/вывода в планировщике операций ввода/вывода, который мы рассматривали ранее в этой главе.

ние сразу нескольких строк кэша обычно более эффективно, чем чтение лишь одной такой строки. Производительность повышается, если программа последовательно считывает данные из этих областей памяти. Однако если считывание будет осуществляться в произвольном порядке, упреждающая выборка может замедлить работу процессора. Центральные процессоры, которые поддерживают упреждающую выборку, обычно содержат специальные встроенные инструкции, позволяющие программам инициировать упреждающую выборку в фоновом режиме в целях достижения максимального параллелизма[1].

Поскольку большинство кэшей позволяют выбирать режим для отдельных областей, одна из них может использовать режим обратной записи, другая — режим сквозной записи, а третья — оставаться некэшируемой. Обычно данные операции являются привилегированными, поэтому пользовательские программы никогда сами непосредственно не переключают режим обратной или сквозной записи кэша. Подобное регулирование зачастую требуется только драйверам устройств.

Программирование приложений-подсказок кэша

Управление упреждающей выборкой осуществляется на программном уровне посредством так называемых приложений-подсказок кэша с использованием функции `madvise`. Данный API-интерфейс позволяет сообщать операционной системе, как вы планируете использовать блоки памяти. Нет никаких гарантий того, что операционная система последует вашим рекомендациям, однако если она все же так поступит, то при определенных обстоятельствах это может привести к увеличению производительности. Для того чтобы сообщить операционной системе, что вы хотите задействовать упреждающую выборку, потребуется команда следующего вида:

```
madvise( pointer, size, MADV_WILLNEED | MADV_SEQUENTIAL);
```

Согласованность памяти

Согласованность памяти связана с отдельной проблемой сохранения актуальности содержимого кэшей в многопроцессорных системах. Когда один процессор изменяет данные в кэше, второй процессор не получит к ним доступ до тех пор, пока содержимое этого кэша не будет записано обратно в память. Теоретически, если второй процессор попытается осуществить считывание из этой области, он получит некорректные данные. На самом деле современные процессоры на аппаратном уровне содержат сложные механизмы, позволяющие избежать этого. В нормальных условиях данный факт будет очевиден для программного обеспечения, особенно в пространстве пользователя. В системах с поддержкой *симметричной многопроцессорной обработки* (Symmetric Multiprocessing, SMP) за сохранение согласованности кэшей между разными процессорами отвечает аппаратное обеспечение.

Проблемы с согласованностью памяти возникают даже в однопроцессорных системах, поскольку некоторые периферийные устройства могут сыграть роль

[1] Некоторые новые версии BIOS позволяют активировать и деактивировать упреждающую выборку строк кэша на уровне системы.

своеобразных «дополнительных процессоров». Любые аппаратные устройства, обращающиеся к памяти посредством *прямого доступа* (Direct Memory Access, DMA), могут осуществлять чтение и запись без ведома центрального процессора.

Если данные в кэше оказываются старше, чем данные, содержащиеся в памяти, они называются *устаревшими*. Если программа инициирует передачу данных посредством DMA-доступа с устройства в оперативную память, она должна «сообщить» центральному процессору о том, что элементы, содержащиеся в кэше, должны быть удалены (отброшены). В некоторых системах это называется *аннулированием* элементов кэша.

Данные в кэше, которые окажутся новее, чем те, что располагаются в оперативной памяти, называются «*грязными*». Перед тем как драйвер позволит устройству считать данные посредством DMA-доступа, он должен убедиться в том, что все «грязные» элементы были записаны в память. Это называется сбросом *на диск содержимого* (*очисткой*) или *синхронизацией* кэша.

К счастью, большинство программистов защищены от проблем с согласованностью кэшей на уровне аппаратного обеспечения и операционной системы. Лишь специфические драйверы могут привести к возникновению подобных трудностей у приложения, осуществляющего системный вызов `mmap`. Одним из примеров может стать файл, отображаемый (загружаемый) в память. Если процесс выполняет совместное отображение файла в память, изменения, произошедшие в этом файле, не станут сразу же «видны» другим процессам. Данный процесс должен явным образом синхронизировать память с этим файлом, прежде чем изменения, которые в нем произошли, станут доступными для прочих процессов.

По этой причине стандарт POSIX предусматривает наличие функции `msync`, которая позволяет приложениям осуществлять операции, эквивалентные сбросу содержимого на диск (очистке) и аннулированию.

Роль подкачки

Добавление раздела подкачки равносильно установке дополнительного объема памяти в своей системе. Идея заключается в том, что бóльшая часть памяти, выделяемая процессам, не используется основную часть времени. Поэтому вполне разумно перенести эти блоки оперативной памяти и временно разместить их на жестком диске, высвободив таким образом эту память для других пользователей. Если эта оперативная память вновь потребуется, данные могут считываться с диска и помещаться обратно в память, в то время как другой неиспользуемый блок памяти может «извлекаться» из нее и размещаться на жестком диске. Эти два блока памяти меняются местами, то есть подкачиваются *(swap)*, отсюда и происходит название «*подкачка*». Программисты часто используют слово «подкачка» для обозначения и «предмета», и действия. Область жесткого диска, используемая для хранения страниц, называется *разделом подкачки*, однако термин «подкачка» также используется для описания операции перемещения данных как в раздел подкачки, так и из него. В сфере операционных систем слово «подкачка» никогда не используется для обозначения действия. Процедура перемещения данных из памяти в раздел подкачки просто называется *подкачкой страниц (paging)*.

С подкачкой связана проблема, называемая *переполнением памяти*, которая наблюдается, когда несколько запущенных процессов одновременно пытаются

получить доступ к бóльшему объему памяти, чем физически доступно. Системе приходится перемещать страницы в раздел подкачки и из него при каждом переключении контекста, то есть она тратит больше времени на эти процедуры, чем на непосредственное выполнение программы. Это приводит к замедлению работы системы, поскольку центральный процессор занят перемещением данных на диск подкачки и обратно. В качестве альтернативного решения можно принудительно завершить эти процессы при помощи так называемого *«убийцы» процессов из-за нехватки памяти* (*Out-Of-Memory Killer (OOM)*; подробнее о нем поговорим позднее в этой главе).

Еще одна проблема может возникнуть, когда система испытывает тяжелую нагрузку в виде большого числа операций ввода/вывода. В данной ситуации кэш файловой системы может потреблять бóльшую часть доступного объема памяти, в то время как выполняющиеся процессы будут запрашивать дополнительное пространство в оперативной памяти. Если процесс запрашивает большой блок памяти и этот запрос не может быть удовлетворен немедленно, ядру приходится решать, что делать: выполнять подкачку или регенерацию буферов кэша файловой системы.

Активировать и деактивировать подкачку можно в любой момент, воспользовавшись командами `swapon` и `swapoff`. Поскольку операционная система Linux позволяет иметь более одного раздела подкачки, эти команды дают возможность активировать и деактивировать отдельные разделы подкачки. Отключение также возможно при помощи параметра `-a`, который позволяет применить соответствующую команду ко всем разделам в файле `/etc/fstab`, помеченным как разделы подкачки.

Устройствами подкачки не обязательно должны быть разделы жесткого диска. Для форматирования разделов подкачки используется команда `mkswap`, которая также позволяет форматировать плоские файлы.

Своевременное создание файлов подкачки может оказаться полезным, если возникает необходимость во временном увеличении размера раздела подкачки. Это позволяет обойтись без повторной разбивки своего дискового накопителя на разделы.

Процессы и виртуальная память

С точки зрения программиста каждый процесс в Linux должен обладать собственной виртуальной памятью. Поскольку пространство ядра является общим для всех процессов, то при их выполнении в режиме ядра им будет доступен один и тот же объем памяти. Это имеет большое значение, так как позволяет ядру поручать задачи выполняющемуся в данный момент процессу. Здесь необходимо сделать оптимальный выбор, поскольку адресное пространство ограничено и его нужно разделить на пространство ядра и пространство пользователя.

Адреса пространства пользователя начинаются с нуля и ограничены фиксированным верхним пределом. Верхний предел обозначает максимальный теоретический объем памяти, который может быть доступен процессу пространства пользователя. Все виртуальные адреса ядра берут свое начало именно с этого адреса и недоступны в режиме пользователя. Для большинства 32-битных архитектур характерно отведение по умолчанию 3 Гбайт под пространство пользователя

и 1 Гбайт — под пространство ядра. Такое распределение осуществляется при сборке ядра и не может быть изменено без его пересборки.

Теоретически 32-битный процесс может «поглотить» до 3 Гбайт памяти. На практике большой объем памяти, используемый простой программой C, отнимают стандартная библиотека и прочие включенные библиотеки, а также динамическая память. Пример можно увидеть в листинге 5.1.

Листинг 5.1. pause.c: пример простой программы, иллюстрирующий использование памяти

```
int main()
{
      return pause();
}
```

Программа из листинга 5.1 не выполняет каких-либо особых действий, а лишь делает паузу, для того чтобы вы могли изучить ее. Вы можете запустить ее из оболочки в фоновом режиме и ознакомиться с ее картами памяти. Карту памяти любого процесса (пространства пользователя) можно отыскать в файле /proc/PID/maps, однако более дружественный пользователю вывод можно получить при помощи команды pmap, которая является частью пакета procps:

```
$ ./pause &
[1] 6321
$ pmap 6321
6321:   ./pause
004d0000    104K r-x--  /lib/ld-2.3.5.so
004ea000      4K r----  /lib/ld-2.3.5.so
004eb000      4K rw---  /lib/ld-2.3.5.so
004ee000   1168K r-x--  /lib/libc-2.3.5.so
00612000      8K r----  /lib/libc-2.3.5.so
00614000      8K rw---  /lib/libc-2.3.5.so
00616000      8K rw---    [ anon ]
08048000      4K r-x-- /home/john/examples/mm/pause
08049000      4K rw--- /home/john/examples/mm/pause
b7f08000      4K rw---    [ anon ]
b7f1a000      4K rw---    [ anon ]
bfb05000     88K rw---    [ stack ]
ffffe000      4K -----    [ anon ]
 total     1412K
```

Команда pmap позволяет выводить на экран виртуальные адреса и размеры различных сегментов виртуальной памяти. Как вы можете видеть, каждая область памяти подобно файлу имеет набор разрешений. Рядом с каждой областью команда pmap приводит файл, ассоциированный с этим отображением, если он имеется в наличии. Из приведенного вывода видно, что процесс, который не выполняет никаких действий, потребляет примерно 1,4 Мбайт виртуальной памяти, бóльшая часть которой отводится для стандартной библиотеки C (/lib/libc-2.3.5.so). Немалый объем памяти также отнимает редактор связей (ld-2.3.5.so), который потребляет 112 Кбайт. Использованный мной программный код занимает лишь 4 Кбайт, что составляет одну страницу памяти — минимально возможный объем. Необходимо отметить следующее: несмотря на то что libc занимает 1,1 Мбайт вир-

туальной памяти, секции, доступные только для чтения, распределяются между всеми процессами в системе, которые их используют, то есть данная библиотека потребляет только 1,1 Мбайт физической памяти во всей системе. В этом заключается одно из главных преимуществ использования разделяемых библиотек.

Еще один момент, на который следует обратить внимание в приведенной карте памяти, заключается в больших промежутках между адресами виртуальной памяти. Это означает, что общий размер цельного блока виртуальной памяти, отводимый процессу, будет меньше, чем в случае, если бы эти области были расположены «ближе» друг к другу. Один из таких промежутков имеется между областью, расположенной по адресу 616000, и исполняемым сегментом, находящимся по адресу 8048000 (примерно 122 Мбайт). Для большинства приложений это не будет представлять проблемы, однако если вашей программе необходимо размещать в памяти большие объемы данных, то подобные промежутки могут стать помехой.

Рассмотрим аналогичный пример на ассемблере. Для упрощения мы воспользуемся ассемблером 80x86, однако результаты должны быть схожими на любой платформе. В листинге 5.2 приведена та же программа, что и в листинге 5.1, написанная на ассемблере 80x86. Разница заключается в том, что она задействует созданные вручную системные вызовы и не использует стандартную библиотеку C.

Листинг 5.2. pause.s: пример простой программы, написанной на ассемблере 80x86

```
.text

# Редактор связей использует _start в качестве точки входа.
.global _start
       .type _start, @function

# Обработчик сигналов. Не выполняет никаких действий

sighdlr:
      ret

_start:

# Использовать системный вызов BSD signal() syscall; аналог : signal(SIGCONT,sighdlr)

      movl $sighdlr, %ecx     # 3-й аргумент, sighdlr
      movl $18, %ebx          # 2-й аргумент, 18 = SIGCONT
      movl $48, %eax          # 1-й аргумент, 48 = системный вызов BSD
                                signal()
      Int  $0x80              # выполнить системный вызов

# Выполнить системный вызов pause()

      movl $29, %eax          # 1-й аргумент, системный вызов 29 = pause()
      int  $0x80              # выполнить системный вызов

# Завершить системный вызов
```

```
# Мы достигаем этого этапа, если посылается сигнал SIGCONT.

      movl $0,%ebx            # 2-й аргумент, код завершения
      movl $1,%eax            # 1-й аргумент, 1 = exit()
      int  $0x80              # выполнить системный вызов
```

Сборка этой программы выполняется при помощи следующей команды:

```
$ gcc -nostdlib -o pause pause.s
```

После ее ввода вы получите гораздо более краткую карту памяти:

```
$ ./pause &
[1] 6992
$ pmap 6992
6992:   ./pause
08048000      4K r-x--  /home/john/examples/mm/pause
08049000      4K rwx--  /home/john/examples/mm/pause
bf8f5000     88K rwx--    [ stack ]
ffffe000      4K -----    [ anon ]
 total      100K
```

Единственное, что мы здесь видим, — это то, что было создано редактором связей и системным вызовом Linux exec. Редактор связей добавил раздел данных с доступом для записи, поскольку мы сами не сделали этого. Системный вызов exec загрузил наш программный код на страницу памяти с доступом только для чтения по адресу 8049000. Указанные биты разрешений очень схожи с битами разрешений файла, при этом они демонстрируют, что программный код, загруженный на данную страницу, доступен для чтения и исполнения. Рядом с битами разрешений команда pmap приводит исполняемое имя, благодаря чему мы знаем, откуда происходит данная страница. Системный вызов exec также выделил одну страницу с доступом для записи для нашего сегмента данных. В завершение exec создал стек, который составляет самый большой фрагмент карты, равный 88 Кбайт. *Анонимное* отображение, расположенное по адресу ffffe000, используется в Linux 2.6 как часть нового, более эффективного механизма осуществления системных вызовов на платформе IA32. Что касается приведенного примера, то в нем использован старый метод.

Обзор расширений физического адреса PAE (Physical Address Extension), реализованных компанией Intel

Объем оперативной памяти, которую можно установить в компьютер, ограничен не только количеством DIMM-разъемов, доступных на материнской плате. Он ограничен также объемом физической памяти, который способен адресовать центральный процессор. Этот лимит определялся размером слов, поддерживаемым процессором. Так, например, 32-битные процессоры могут оперировать лишь 32-битными указателями, то есть предел физического адреса составляет 232 бит, или 4 Гбайт. Когда 32-битные процессоры только появились, то, что кому-либо может потребоваться целых 4 Гбайт оперативной памяти, которая в то время стоила немалых денег, казалось неправдоподобным.

Шло время, и емкость модулей оперативной памяти росла, в результате чего вскоре стало возможным создание систем с 4 Гбайт оперативной памяти по разумной цене. Программное обеспечение с легкостью освоило подобные объемы памяти, из-за чего пользователям требовалось нарастить ее еще больше. Очевидным решением стал бы переход на 64-битную архитектуру. Однако в то время подобная процедура означала необходимость переноса всех приложений на новую платформу. Это было весьма затратным решением, особенно если принять во внимание тот факт, что большинству пользователей требовалась возможность запуска большего количества процессов, а не увеличения их размеров. Это подтолкнуло компанию Intel к реализации механизма, позволяющего расширить физическую память без необходимости в затратном переходе на 64-битную архитектуру.

Расширения физического адреса PAE (Physical Address Extension), реализованные компанией Intel, позволяют процессору адресовать до 64 Гбайт (236 байт) оперативной памяти путем увеличения адреса страницы с 20 до 24 бит. Так как размер страницы остается прежним, смещение по-прежнему должно составлять 12 бит. Это означает, что эффективный физический адрес будет равен 36 битам. Поскольку логический адрес должен укладываться в рамки 32-битного регистра, отдельные процессы могут адресовать только 4 Гбайт виртуальной памяти.

Аппаратный блок управления памятью MMU (Memory Management Unit) и операционная система используют адреса страниц исключительно для манипулирования страницами, то есть операционная система свободно может оперировать 24-битными страничными адресами при распределении страниц для кэша или процессов. Таким образом, совокупная виртуальная память, доступная в системе, будет составлять 64 Гбайт (236 байт).

5.6.2. Нехватка памяти

Любая система находится в постоянном движении, при этом блоки памяти все время занимаются и высвобождаются. Многие процессы занимают небольшие блоки памяти на короткий промежуток времени, а другие процессы могут занять какую-либо область памяти и находиться в ней постоянно. Процесс может испытывать нехватку памяти, даже когда в системе доступен ее приличный объем, при этом сама система может также испытывать недостаток памяти, в то время как некоторые процессы будут продолжать выполняться без каких-либо проблем. Все зависит от конкретных обстоятельств.

Ситуации, в которых процессы могут испытывать нехватку памяти

Недостаток памяти у процессов может наблюдаться в двух случаях: когда они испытывают нехватку виртуальных адресов или нехватку физической памяти. На первый взгляд недостаток виртуальных адресов кажется маловероятным. В конце концов, если вы располагаете 1 Гбайт оперативной памяти и у вас нет диска подкачки, разве выполнение `malloc` не должно завершиться неудачей задолго до исчерпания запаса виртуальных адресов? Программа, приведенная в листинге 5.3,

наглядно демонстрирует обратную ситуацию. Она распределяет оперативную память блоками по 1 Мбайт до тех пор, пока выполнение malloc не потерпит неудачу.

Листинг 5.3. crazy-malloc.c: распределение максимально возможного объема памяти

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

int main(int argc, char *argv[])
{
  void *ptr;
  int n = 0;

  while (1) {
    // Распределять память блоками по 1 Мбайт
    ptr = malloc(0x100000);

    // Приостановиться, когда дальнейшее распределение невозможно
    if (ptr == NULL)
      break;

    n++;
  }
  // Каков итоговый объем распределенной памяти?
  printf("распределено %d Мб\n", n);

  // Приостановиться, чтобы можно было оценить убыток.
  pause();
}
```

Программа из листинга 5.3 выполнялась на 32-битной системе с 160 Мбайт оперативной памяти, подкачка при этом не использовалась. Вам интересно, каков результат? Выполнение malloc *не* завершилось неудачей до тех пор, пока процессу не было выделено почти 3 Гбайт памяти! Я добавил вызов pause, для того чтобы вы могли ознакомиться с картой памяти:

```
$ ./crazy-malloc &
[1] 2817
malloced 3056 MB
(распределено 3056 Мбайт)
$ jobs -x pmap %1
2823:   ./crazy-malloc
000cc000   4112K rw---    [ anon ]
004d0000    104K r-x--  /lib/ld-2.3.5.so
004ea000      4K r----  /lib/ld-2.3.5.so
004eb000      4K rw---  /lib/ld-2.3.5.so
004ee000   1168K r-x--  /lib/libc-2.3.5.so
00612000      8K r----  /lib/libc-2.3.5.so
00614000      8K rw---  /lib/libc-2.3.5.so
00616000      8K rw---    [ anon ]
006cf000 124388K rw---    [ anon ]
08048000      4K r-x--  /home/john/examples/mm/crazy-malloc
```

```
08049000      4K rw---  /home/john/examples/mm/crazy-malloc
08051000 2882516K rw---    [ anon ]
b7f56000 125424K rw---    [ anon ]
bfa43000     84K rw---    [ stack ]
bfa58000   5140K rw---    [ anon ]
ffffe000      4K -----    [ anon ]
 total  3142980K
```

Стандартная библиотека C и динамическая память. Термин *динамическая память (heap)* служит для описания пула памяти, используемого программами на языке C и C++ для динамических распределений памяти. Реализовать динамическую память можно несколькими способами, основным из которых является стандартная GNU-библиотека C. Классический способ, описанный в книге Б. В. Кернигана и Д. Ритчи «Язык программирования C» («The C Programming Language», Prentice Hall), заключается в распределении большого пула памяти и отслеживании в нем свободных блоков с использованием соответствующего связанного списка. У данного подхода имеется недостаток, состоящий в том, что процесс может потреблять память, которая ему на самом деле не требуется. Для повышения эффективности в большинстве реализаций динамической памяти ее распределение осуществляется только по мере необходимости посредством системного вызова `brk`. Это позволяет приложению начинать с небольшого объема динамической памяти, который может увеличиваться при получении дополнительных запросов памяти. После распределения эта память редко возвращается обратно в систему.

Другой недостаток заключается в том, что монолитный пул имеет тенденцию со временем становиться *фрагментированным*. Это случается, когда небольшие блоки не высвобождаются после распределения, как показано на рис. 5.7. Если подобные блоки не высвобождаются после того, как они были распределены, как, например, блоки 2 и 4, они вызывают дробление более крупных блоков. Когда мы распределяем блок 1, его размер ограничен только размером пула памяти. После того как произойдут три процедуры распределения и две — высвобождения, максимальный размер следующего распределяемого блока будет намного меньше всей доступной памяти. Небольшие блоки 2 и 4 вызывают фрагментацию пула памяти. В системе, не имеющей виртуальной памяти, это может продолжаться неопределенно долго, до тех пор, пока процедуры распределения не начнут заканчиваться неудачей. К счастью, стандартная библиотека позволяет предотвращать фрагментацию.

Подробное рассмотрение динамической памяти выходит за рамки настоящей книги, но я все же расскажу вам об уловке, которую стандартная GNU-библиотека C использует для предотвращения фрагментации, а ее практическое применение вы увидите на примере программы, приведенной в листинге 5.4.

Стандартная GNU-библиотека C использует обычный пул памяти при распределении небольших блоков, а для распределения больших блоков памяти она задействует системный вызов `mmap`. Она обеспечивает предотвращение фрагментации, проиллюстрированной на рис. 5.7, благодаря тому что отделяет небольшие блоки от крупных, размещая их в разных пулах[1]. Для большинства приложений

[1] Такой подход также может поставить в тупик некоторые инструменты, отвечающие за проверку динамической памяти, поскольку они не будут «знать» о данной уловке.

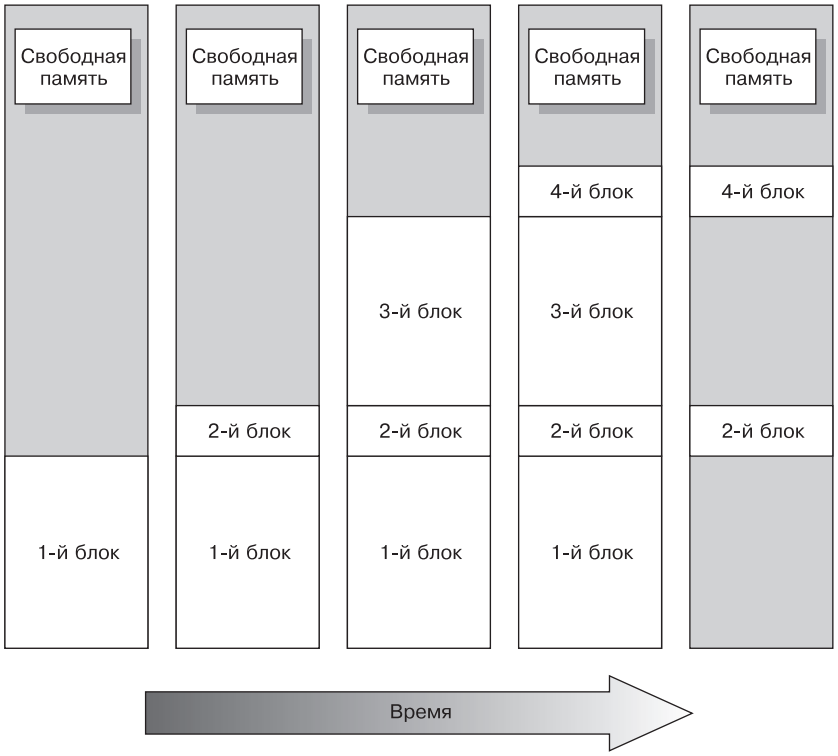


Рис. 5.7. Иллюстрация фрагментации памяти

размер пула виртуальной памяти будет намного превышать тот, который вы желали бы иметь для динамической памяти, из-за чего вы никогда не будете испытывать недостатка в виртуальных адресах. Для большинства программ этого будет вполне достаточно. Однако при определенных условиях пространство виртуальных адресов также может становиться фрагментированным подобно динамической памяти. Поэтому стандартная библиотека C дает возможность некоторым приложениям управлять использованием `mmap` посредством функции `mallopt`. Эта функция не имеет страниц руководства man, но вы можете отыскать сведения о ней на странице info, посвященной `libc`, вводом следующей команды:

```
$ info libc mallopt
```

Функция `mallopt` является частью стандарта SVR4, хотя поддерживаемые ею параметры варьируются в зависимости от используемой системы. Пара таких полезных GNU-параметров приведена в табл. 5.8.

Таблица 5.8. Настраиваемые GNU-параметры функции mallopt()

Параметр	Цель использования
M_MMAP_THRESHOLD	Определение величины порога в байтах. Любой распределенный блок, превышающий этот порог, вместо динамической памяти будет использовать mmap

Параметр	Цель использования
M_MMAP_MAX	Определение максимального числа распределенных при помощи mmap блоков, которые могут задействоваться в любой заданный момент времени. При превышении этого порога все распределенные блоки будут использовать динамическую память. Для отключения использования mmap значение данного порога должно быть равным нулю

Виртуальная память и динамическая память. В примере `crazy-malloc` программа отвела почти все пространство пользователя под динамическую память в системе, где имелось только 160 Мбайт оперативной памяти и не было раздела подкачки. Приведенная карта наглядно демонстрировала, как стандартная библиотека C создает *анонимные* отображения в пространстве пользователя посредством системного вызова `mmap`. Воспользовавшись инструментом `strace`, можно увидеть, что каждый вызов `malloc` приводит к вызову `mmap`, как показано далее:

```
mmap2(NULL,1052672,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_ANONYMOUS,-1,0)
```

Системный вызов `mmap2` позволяет выделять процессу память путем установки флага `MAP_ANONYMOUS`. В данном случае драйвера устройства в качестве хранилища не требуется, из-за чего аргумент дескриптора файла равен `-1`. Для обеспечения эффективности ядро откладывает поиск свободной памяти для этих страниц до тех пор, пока они не будут использованы, поэтому `mmap2` возвращает указатель на виртуальную память, которой пока еще не существует. Только после того, как вы воспользуетесь этим виртуальным адресом, будет выделена необходимая физическая память. Когда это случится, произойдет ошибка страницы, которая, в свою очередь, побудит ядро искать свободную область в физической памяти для этой страницы. Подобный подход весьма эффективен, поскольку он увеличивает скорость выполнения многих операций, предотвращая излишние процедуры обращения к памяти.

Если модифицировать программу из листинга 5.3 таким образом, чтобы она изменяла распределяемые данные, это приведет к возникновению ошибок страницы, в результате чего вы увидите совершенно другое поведение, которое показано в листинге 5.4.

Листинг 5.4. `crazy-malloc2.c`: Распределение и использование памяти

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

int main(int argc, char *argv[])
{
  void *ptr;
  int n = 0;

  while (1) {
    // Распределять память блоками по 1 Мбайт
    ptr = malloc(0x100000);

    //Приостановиться, когда дальнейшее распределение невозможно
```

```
    if (ptr == NULL)
      break;

    // Модифицировать данные.
    memset(ptr, 1, 0x100000);
    printf("распределено %d Мб\n", ++n);
  }
  // Приостановиться, чтобы можно было оценить убыток.
  pause();
}
```

При выполнении программы из листинга 5.4 результат может оказаться не таким, как можно было ожидать. Вместо выполнения приостановки, дающей возможность оценить убыток, данная программа принудительно завершается, прежде чем сделать это:

```
$ ./crazy-malloc2
malloced 1 MB
malloced 2 MB
malloced 3 MB
...
malloced 74 MB
Killed
$
```

Выполнение данной программы было принудительно завершено грозным *«убийцей» процессов Out-Of-Memory Killer* (который часто сокращенно называется *OOM*). Модификация данных привела к тому, что *система* стала испытывать нехватку памяти. С точки зрения процесса в системе имеется очень много памяти в виде виртуальных адресов. Когда система начинает испытывать нехватку ресурсов при обращении к оперативной памяти или разделу подкачки, ядро реагирует на это, принудительно завершая соответствующие процессы. Ядро сообщает пользователю, по какой причине оно так поступило (возможно, не по его вине). Среди множества малопонятных сообщений, которые можно отыскать в `/var/log/messages`, вы встретите следующее:

```
Out of Memory: Killed process 2995 (crazy-malloc2)
```

Единственный раз, когда выполнение `malloc` потерпело неудачу, связан с нехваткой виртуальной памяти. Инструмент `malloc` продолжал возвращать указатели на виртуальную память, которая лежала далеко за пределами возможностей системы. Можно сказать, что `malloc` выписывает чеки, по которым не сможет расплатиться. В операционной системе Linux такая ситуация называется *избыточным использованием памяти*, что говорит о том, что ядро позволяет отводить процессу больше памяти, чем доступно в данный момент. Ядро полагает, что нужный объем памяти будет всегда доступен в случае необходимости. Пользователи могут изменить такое поведение, если пожелают.

Избыточное использование памяти можно отключить в ядре, выполнив следующую команду с правами корневого пользователя *root*:

```
$ echo 2 > /proc/sys/vm/overcommit_memory
```

Ситуации, в которых система может испытывать нехватку памяти

Вы уже видели, как ядро использует «убийцу» процессов Out-Of-Memory Killer для принудительного завершения процессов, когда система испытывает нехватку памяти. До того как это произойдет, ядро сбрасывает на диск содержимое кэша файловой системы для высвобождения места наряду с содержимым других кэшей, которое также может быть отправлено на диск. После этого она переносит страницы на диск, если это будет возможно.

Данные операции отнимают много времени, которое обычно записывается «на счет» процесса, вызвавшего проблему (запросившего всю имеющуюся память). Когда объем свободной памяти снижается, фактически любой процесс может вызвать обращение к разделу подкачки при простом переключении контекста. Это называется *переполнением памяти*, о нем упоминалось ранее. Когда система испытывает нехватку ресурсов при обращении к разделу подкачки, запускается «убийца» процессов Out-Of-Memory Killer, в чем можно было убедиться в одном из предыдущих примеров. Прежде чем это произойдет, система может потерять впустую много времени из-за переполнения памяти.

Блокировка памяти

Страницы пространства ядра и пространства пользователя могут переноситься на диск, однако некоторые из них не поддаются этой процедуре. Такие страницы называются *блокированными*. Например, страницы, содержащиеся в памяти, отведенной устройству `ramdisk`, не могут быть перемещены (подкачаны) на диск. Ядро позволяет процессам пространства пользователя блокировать память посредством системных вызовов `mlock` и `munlock`. Блокировка памяти отнимает определенный объем оперативной памяти и уменьшает количество страниц, которые могут быть перенесены на диск. Это может привести к переполнению памяти, поскольку объем физической памяти, доступной для неблокированных страниц, уменьшается. По этой причине лишь процессам с привилегиями суперпользователя разрешается использовать системные вызовы `mlock` и `munlock`.

Для того чтобы предотвратить перемещение (подкачку) содержимого какой-либо области памяти на диск, необходим следующий вызов:

```
r = mlock( ptr , size );
```

Подобно POSIX-функциям, `mlock` возвращает нулевое значение в случае успеха, а в случае возникновения ошибки — значение `-1`. Если она возвращает ненулевое значение, то вы можете быть уверены в том, что страницы располагаются в оперативной памяти, то есть доступ к ним будет осуществляться с высокой скоростью. Точнее говоря, при доступе к этой области памяти никогда не будут возникать ошибки страниц.

Именно таким образом критически важные процессы продолжают выполняться, даже когда система начинает испытывать нехватку памяти. Страницы блокируются, когда в системе происходит переполнение памяти. Контекстное переключение на процесс, который блокирует большинство или все свои страницы, будет отнимать столько же ресурсов, сколько переключение на другой процесс. По этой

причине для блокировки страниц была внедрена еще одна полезная функция — mlockall. Она принимает только аргументы флагов, которые могут быть комбинацией MCL_CURRENT для блокировки всех текущих распределенных страниц процесса, и MCL_FUTURE — для блокировки всех будущих распределенных страниц процесса. Какой-нибудь «несносный» демон может потребовать постоянной блокировки всех его страниц, содержащихся в памяти. В этом случае необходимо применять оба флага:

```
r = mlockall( MCL_CURRENT | MCL_FUTURE );
```

После совершения этого вызова все страницы, используемые процессом, будут оставаться в оперативной памяти до тех пор, пока не будут разблокированы. Новые страницы, создаваемые в результате применения системного вызова brk (который обычно является результатом использования malloc) или любого другого, который распределяет новые страницы, также будут оставаться в оперативной памяти неопределенно долгое время. По этой причине утечки памяти недопустимы.

В отличие от вызова mlock, вызов функции mlockall может осуществляться процессом, выполняющимся без привилегий суперпользователя. Ограничение заключается в том, что процесс без подобных привилегий не сможет блокировать страницы. Непривилегированный процесс может осуществлять вызов mlockall (MCL_FUTURE), который не приведет к блокировке текущих распределенных страниц, однако заставляет ядро блокировать все новые распределенные страницы этого процесса. Если процесс не имеет привилегий суперпользователя, то в данной ситуации распределение потерпит неудачу. Например, если malloc приводит к вызову brk, то данный вызов потерпит неудачу, что, в свою очередь, приведет к тому, что malloc вернет указатель NULL. Именно таким образом можно проверить, как обрабатываются ошибки, возникающие при нехватке памяти.

Как вы уже могли догадаться, у mlock и mlockall есть аналоги, которые позволяют выполнять разблокирование страниц и называются munlock и munlockall. Блокировка страниц осуществляется только по мере необходимости, а снятие блокировки выполняется в целях высвобождения физической памяти для других процессов.

Если ваша система располагает объемом памяти, намного меньшим, чем пространство виртуальных адресов центрального процессора, то ваши процессы вряд ли столкнутся с нехваткой памяти раньше, чем это случится с самой системой. Это прискорбно, поскольку для разрешения подобных ситуаций в приложениях можно было бы использовать функцию обработки ошибок. Вы не сможете ничего разрешить, если ваш процесс был принудительно завершен «убийцей» процессов Out-Of-Memory Killer. Альтернативное решение содержится в GNU-библиотеке C, что является частью библиотечной функции sysconf. Запросить число доступных физических страниц можно при помощи следующего вызова:

```
num_pages = sysconf( _SC_AVPHYS_PAGES );
```

В результате вы получите количество страниц, которые система может распределять без необходимости сброса содержимого кэша или страниц на диск. Оно

будет приблизительно равно значению MemFree в /proc/meminfo. Поскольку в данное значение не включается память, высвобождаемая при сбросе на диск страниц из кэша файловой системы, оно будет весьма умеренным.

При распределении памяти между процессами существует еще одна «линия обороны» — системный вызов setrlimit, позволяющий администраторам и даже простым пользователям определять лимиты ресурсов, которые могут выделяться одиночному процессу. В листинге 5.5 вы сможете увидеть пример переработанной программы crazy-malloc, которая в данном случае использует системный вызов setrlimit, взяв за основу объем памяти, доступный в системе.

Листинг 5.5. crazy-malloc3.c: распределение памяти с установлением соответствующих лимитов

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <stdlib.h>
 4 #include <limits.h>
 5 #include <signal.h>
 6 #include <unistd.h>
 7 #include <sys/types.h>
 8 #include <sys/stat.h>
 9 #include <sys/resource.h>
10
11 int main(int argc, char *argv[])
12 {
13     void *ptr;
14     int n = 0;
15     int r = 0;
16     struct rlimit rl;
17     u_long pages, max_pages, max_bytes;
18
19     pages = sysconf(_SC_AVPHYS_PAGES);
20
21     /* Вычислить max_bytes, но следить за переполнением */
22     max_pages = ULONG_MAX / sysconf(_SC_PAGE_SIZE);
23     if (pages > max_pages)
24         pages = max_pages;
25     max_bytes = pages * sysconf(_SC_PAGE_SIZE);
26
27     r = getrlimit(RLIMIT_AS, &rl);
28
29     printf("текущий жесткий лимит равен %ld Мб\n",
30            (u_long) rl.rlim_max / 0x100000);
31
32     /* Модифицировать мягкий лимит и не изменять жесткий лимит. */
33     rl.rlim_cur = max_bytes;
34
35     r = setrlimit(RLIMIT_AS, &rl);
36     if (r) {
37         perror("setrlimit");
38         exit(1);
```

```
39     }
40
41      printf("значение лимита составляет %ld Мб\n", max_bytes / 0x100000);
42
43      while (1) {
44          // Распределять память блоками по 1 Мбайт
45          ptr = malloc(0x100000);
46
47          // Приостановиться, когда дальнейшее распределение невозможно
48          if (ptr == NULL) {
49              perror("malloc");
50              break;
51          }
52
53          memset(ptr, 1, 0x100000);
54          printf("распределено %d Мб\n", ++n);
55     }
56     // Приостановиться, чтобы можно было оценить убыток.
57     printf("приостановлено\n");
58     raise(SIGSTOP);
59     return 0;
60 }
```

При выполнении программа `crazy-malloc3` вместо того, чтобы принудительно завершиться «убийцей» процессов Out-Of-Memory Killer, в итоге терпит неудачу. На моей системе я получил следующий вывод:

```
$ ./crazy-malloc3
current hard limit is 4095 MB
limit set to 53 MB
malloced 1 MB
malloced 2 MB
malloced 3 MB
...
malloced 50 MB
malloced 51 MB
malloc: Cannot allocate memory
paused
```

Структура `rlimit` состоит из мягкого и жесткого лимитов. Жесткий лимит обычно определяется при запуске системы; лимиты по умолчанию отсутствуют. Непривилегированный пользователь может устанавливать любое значение в качестве мягкого лимита, но оно не должно превышать значения жесткого лимита. Такой пользователь может также устанавливать более низкое значение жесткого лимита для текущего процесса и его дочерних процессов, однако после этого лимит данного процесса не может быть вновь увеличен. По этой причине, перед тем как осуществлять системный вызов `setrlimit`, необходимо воспользоваться `getrlimit` (строка 27), для того чтобы нечаянно не установить более низкое значение жесткого лимита. Непривилегированные процессы должны иметь возможность изменять значение только мягкого лимита.

5.7. Заключение

В данной главе мы подробно рассмотрели особенности функционирования процессов в операционной системе Linux. Нами были изучены такие концепции, как режим пользователя и режим ядра. Вы также ознакомились с основными принципами системных вызовов и узнали о том, что многие библиотечные функции, которые воспринимаются пользователями как сами собой разумеющиеся, на самом деле являются обертками системных вызовов.

Мы также рассмотрели планировщика Linux и то, каким образом он оперирует программами. Был приведен ряд пользовательских команд, которые позволяют влиять на поведение данного планировщика. Были также рассмотрены особенности того, каким образом ядро отслеживает ход времени. Мы изучили различные типы системных часов, которые «тикают» с разной частотой. Благодаря этому вы теперь можете решить, какие из них вам наиболее подходят.

В этой главе были изложены основы драйверов устройств и узлов устройств, а также системных операций ввода/вывода с использованием таких драйверов. Мы рассмотрели ряд планировщиков операций ввода/вывода и особенности их настройки.

В завершение главы мы обсудили виртуальную память и ее значение для процессов. По ходу были приведены различные ситуации, в которых процессы могут испытывать нехватку памяти; вы также познакомились с грозным «убийцей» процессов Out-Of-Memory Killer.

5.7.1. Инструментарий, использованный в этой главе

- `mkswap`, `swapon`, `swapoff` — инструменты для манипулирования разделами подкачки.
- `nice`, `renice`, `chrt` — инструменты для управления поведением планировщика.
- `pmap` — средство для просмотра карты виртуальной памяти процесса.
- `ps`, `time`, `times` — используются для вывода сведений о количестве времени, которое процессы проводят в пространстве пользователя и пространстве ядра.
- `strace` — превосходный инструмент для анализа поведения системных вызовов, которые используются программами.

5.7.2. API-интерфейсы, рассмотренные в этой главе

- `clock_getres`, `clock_gettime` — высокоточные часы стандарта POSIX.
- `getrusage`, `times` — библиотечные функции для просмотра показателей использования ресурсов системы.
- `mallopt` — API-интерфейс GNU, позволяющий управлять поведением `malloc`.
- `mlock`, `mlockall` — позволяют блокировать страницы в памяти.

- `mmap`, `msync`, `madvise` — API-интерфейсы, позволяющие управлять размещением областей хранения в оперативной памяти и на дисковом накопителе.
- `pthread_setschedparam` — позволяет выбирать политику планирования для потока.
- `sched_get_priority_min/max` — определяет текущий минимальный и максимальный приоритет определенной политики планирования.
- `sched_setscheduler` — позволяет выбирать политику планирования для процесса.
- `sysconf` — сообщает подробности о настройках системной конфигурации.

5.7.3. Веб-ссылки

- www.kernel.org/pub/linux/utils/kernel/hotplug/udev.html — отличный источник сведений о `udev`.
- http://linux-hotplug.sourceforge.net — ресурс, где можно отыскать документацию по Linux-инструменту `hotplug`.

5.7.4. Рекомендуемая литература

- *Cesati M. and Bovet D. P.* Understanding the Linux Kernel. 3d ed. — Sebastopol, Calif.: O'Reilly Media, Inc., 2005.
- *Kernighan B.W. and RitchieD.* The C Programming Language. — Englewood Cliffs, N.J.: Prentice Hall, 1988.
- *Kroah-Hartman G., Corbet J. and Rubini A.* Linux Device Drivers. — Sebastopol, Calif.: O'Reilly Media, Inc., 2005.
- *Love R.* Linux Kernel Development. 3d ed. — Indianapolis: Novell Press, 2005.
- *Rodriguez C. S., Fischer G. and Smolski S.* The Linux Kernel Primer: A Top-Down Approach for x86 and PowerPC Architectures. — Englewood Cliffs, N. J.: Prentice Hall, 2006.

6 Процессы

6.1. Введение

Модель процессов мы рассмотрели в главе 5. Большинство изложенного там материала касалось взаимодействия процессов с ядром. В этой главе сфокусируемся на процессах пространства пользователя. Вы узнаете о жизненном цикле процесса от запуска до завершения, а также о том, что происходит в этом промежутке. Мы подробно рассмотрим, сколько памяти занимают процессы, вы также познакомитесь с инструментами и API-интерфейсами, позволяющими определять уровень использования ресурсов процессами.

6.2. Что порождает процессы?

В операционной системе Linux взаимоотношения процессов строятся по схеме «родительский процесс — дочерний процесс». Процесс имеет только одного родителя, однако он может иметь (почти) любое количество дочерних процессов. У всех процессов есть один общий предок — процесс `init`. Данный процесс запускается первым при загрузке системы и продолжает выполняться до ее выключения. Он отвечает за обеспечение корректного функционирования системы путем осуществления ее корректных загрузки и выключения. Завершить процесс `init` нельзя, даже если вы обладаете правами суперпользователя. Вы должны вежливо «попросить» его завершиться, причем сделать это можно несколькими способами. После этого он корректно отключит вашу систему.

Linux создает процессы при помощи одного из трех системных вызовов. Два из них являются традиционными системными вызовами, используемыми в других UNIX-системах: `fork` и `vfork`. Третий вызов является специфическим для Linux и может создавать как потоки, так и процессы. Он называется *системный вызов* `clone`.

6.2.1. Системные вызовы fork и vfork

Системный вызов `fork` является наиболее оптимальным средством создания новых процессов. В результате его применения создаются два процесса — родительский и дочерний, которые будут идентичными клонами. `fork` возвращает

идентификатор процесса (pid_t), который может иметь как нулевое, так и ненулевое значение. С точки зрения программиста, единственная разница между родительским и дочерним процессом заключается в значении, возвращаемом функцией fork. Родительский процесс «видит» ненулевое возвращаемое значение, которое является идентификатором его дочернего процесса (или значение -1 в случае возникновения ошибки). Дочерний процесс «видит» нулевое возвращаемое значение, которое указывает на то, что перед нами дочерний процесс. Дальнейший ход событий зависит от приложения. Наиболее простой способ действия заключается в использовании одного из системных вызовов exec (о нем мы вскоре поговорим), хотя это совсем не обязательно делать.

Системный вызов vfork является своего рода артефактом. По своей сути он идентичен системному вызову fork, за тем исключением, что гарантирует, что страницы пространства пользователя не будут подвергаться копированию. Раньше fork мог инициировать копирование страниц процессов пространства пользователя в новые страницы. В этом не было особенного смысла, если дочерний процесс осуществлял лишь вызов exec. В таком случае копирование происходило впустую. Это касается, к примеру, процесса init. Дочерние процессы процесса init не нуждаются в копиях страниц пространства пользователя этого процесса, поэтому их копирование является пустой тратой времени. Системный вызов vfork был призван устранить этот этап копирования, для того чтобы повысить эффективность процессов вроде init.

Недостаток vfork заключается в том, что он требует, чтобы дочерние процессы осуществляли вызов exec немедленно, не модифицируя каких-либо страниц памяти. Это легче сказать, чем сделать, особенно если учесть тот факт, что выполнение вызова exec может завершиться неудачей. Те, кого заинтересовал данный аспект, могут обратиться к странице руководства man vfork(2).

Все современные UNIX-системы используют так называемую методику *копирования при записи*, в результате чего обычный системный вызов fork функционирует во многом аналогично vfork и применение vfork становится не только нежелательным, но и излишним.

6.2.2. Копирование при записи

Назначение методики копирования при записи заключается в повышении эффективности путем устранения ненужных процедур копирования. Идея довольно проста. Когда процесс разветвляется, он и его дочерний процесс совместно используют одну и ту же область физической памяти максимально возможное время, то есть ядро копирует только элементы таблицы страниц и помечает все страницы, копируемые при записи. Это будет приводить к возникновению ошибок страниц, когда какой-либо процесс попытается их модифицировать. Когда ошибка страницы возникает из-за копирования при записи, ядро выделяет новую страницу физической памяти и копирует соответствующую страницу до того, как она будет модифицирована. То, как это работает, можно увидеть на рис. 6.1.

Если после разветвления процесса его дочерний процесс модифицирует лишь небольшую часть страниц памяти, это очень хорошо, поскольку дает возможность сэкономить время на копировании всех этих данных. Это также позволяет экономить физическую память, так как немодифицированные страницы располагаются

в той ее области, которую совместно используют оба этих процесса. Без использования методики копирования при записи системе пришлось бы выделять вдвое больше памяти для родительского и дочернего процессов.



Рис. 6.1. Использование методики копирования при записи: если процесс попытается модифицировать страницу, это приведет к возникновению ее ошибки

Только представьте себе, сколько времени занимал бы запуск, если бы процессу `init` приходилось копировать все свои страницы при каждом инициировании дочернего процесса. Основная работа процесса `init` заключается в осуществлении системных вызовов `fork` и `exec`. Дочерние процессы не нуждаются в копиях страниц памяти своего родительского процесса `init`. Кроме того, существует множество системных демонов, которые также осуществляют `fork` и `exec` подобно процессу `init` (они также имеют свои положительные стороны). К ним относятся демоны `xinetd`, `sshd` и `ftpd`.

6.2.3. Системный вызов clone

Системный вызов `clone` является специфическим для операционной системы Linux и используется для создания процессов и потоков. Я решил упомянуть его здесь для полноты картины. Переносимые программы никогда не должны задействовать системный вызов `clone`. API-интерфейсов стандарта POSIX будет вполне достаточно, если пользователю потребуется создать поток или процесс.

`clone` представляет собой сложный системный вызов, реализованный как разновидность системного вызова общего назначения `fork`. Он позволяет приложениям полностью контролировать то, какие блоки дочернего процесса будут совместно использоваться вместе с его родительским процессом. Это делает его пригодным для создания как процессов, так и потоков. Поток можно рассматривать как особый процесс, который делит свое пространство пользователя с родительским процессом.

6.3. Функции exec

Функции exec позволяют передавать управление процессами от одной программы к другой. В операционной системе Linux функции с именем exec не существует, однако здесь я использую данный термин для обозначения семейства библиотечных вызовов. Документацию к этим вызовам можно отыскать на странице руководства man exec(3)[1]. Несмотря на то что exec реализуется посредством множества библиотечных функций, существует только один системный вызов — execve. Все функции, содержащиеся в стандартной библиотеке, являются всего лишь обертками этого системного вызова. Доступ к системному вызову execve осуществляется при помощи функции:

```
int execve(const char *filename, char *const argv [], char *const envp[]);
```

Системный вызов execve осуществляет поиск указываемого пользователем файла, определяет, является ли он исполняемым, и, если это так, пытается загрузить его и выполнить. Для системного вызова такой объем работы непривычен, однако execve уникален по своей природе. В данной главе я буду использовать термин execve для обозначения соответствующего системного вызова. А термин exec будет применяться для обозначения любой из функций exec, описанных на странице руководства exec(3).

На первом этапе ядро проверяет разрешения файла. Владелец процесса должен обладать разрешением на выполнение этого файла до того, как ядро будет его считывать. Если данная проверка завершится неудачей, execve вернет значение ошибки (-1), а errno будет присвоено значение EPERM.

Если проверка разрешений завершится удачно, ядро исследует содержимое файла и удостоверяется в том, что он действительно является исполняемым. Исполняемые файлы подразделяются на три категории: *исполняемые сценарии*, *исполняемые объектные файлы* и *разнообразные двоичные файлы*.

6.3.1. Исполняемые сценарии

Исполняемые сценарии представляют собой текстовые файлы, направляющие ядро к интерпретатору, который должен быть исполняемым объектным файлом. В противном случае выполнение execve терпит неудачу, а переменной errno присваивается значение ENOEXEC (формат ошибки exec). В качестве интерпретатора не может выступать другой сценарий.

Ядро распознает исполняемый сценарий по двум первым символам файла. Если ядро «видит» символы #!, оно разбирает эту первую строку на один или два дополнительных маркера, разделенных пробелом. Типичным примером является сценарий оболочки, который начинается со строки

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

Ядро интерпретирует маркер, следующий после #!, как путь к исполняемому объектному файлу. Если данный файл отсутствует или не является исполняемым объектным файлом, execve возвращает значение ошибки -1. В противном случае ядро разбивает строку 1 сценария на три маркера:

---

[1] Обратите внимание на то, что эта страница man располагается в разделе 3 (библиотеки), а не в разделе 2 (системные вызовы).

- argv[0] — путь к исполняемому объектному файлу интерпретатора;
- argv[1] — все, что следует после имени интерпретатора (аргумент);
- argv[2] — имя файла сценария и генерирует вектор argv для интерпретатора.

Операционная система Linux ограничивает первую строку сценария 128 символами[1], включая пробелы, а в случае превышения строка обрезается и используется «как есть». Любые аргументы, которые следуют после 128-го символа, отбрасываются. В других операционных системах максимальная длина первой строки сценария может быть больше.

Опечатки в сценариях

При написании сценариев мне самому доводилось совершать опечатки и впоследствии наблюдать странное поведение. Оболочка скрывает некоторые такие моменты от внимания ничего не подозревающих пользователей. Вот ряд примеров «антишаблонов» сценариев, которые работают из оболочки, но потерпят неудачу в случае применения execve:

```
# !/bin/sh Обратите внимание на пробел между # и !.
#!/bin/sh Обратите внимание на пробел перед #.
```

Если попытаться выполнить один из этих сценариев при помощи системного вызова execve, он завершится неудачей с ошибкой ENOEXEC. Однако если попытаться запустить любой из них из оболочки, все будет в порядке.

При запуске таких сценариев из оболочки она осуществляет вызов execve точно так же, как вы делали бы при использовании собственного приложения. Как и в случае с вашим приложением, выполнение execve потерпит неудачу. Однако, в отличие от приложения, дочерний процесс оболочки будет представлять собой полноценный командный интерпретатор, благодаря чему он способен определить, является ли соответствующий файл текстовым, после чего интерпретирует его содержимое в виде команд. Первая строка, которую вы могли бы посчитать параметром для execve, будет игнорироваться как комментарий оболочки, а интерпретация остальных операторов пройдет без ошибок. Вуаля! Ваш сценарий оболочки работает — случайно.

Оболочка bash использует простой алгоритм для определения того, будет ли файл, отвергнутый из-за ENOEXEC, передан интерпретатору. Версия 3.00.17 считывает первые 80 символов или первую строку (что будет короче), выискивая не-ASCII символы. Если все символы будут соответствовать стандарту ASCII, данный файл передается интерпретатору оболочки, в противном случае выдается ошибка.

Другой «антишаблон» срабатывает в случае, когда вы используете текстовые редакторы Windows, которые задействуют возвраты каретки в ваших файлах. Системный вызов execve считает возврат каретки в первой строке частью имени файла интерпретатора. Вполне ожидаемо, что его выполнение потерпит неудачу с ошибкой ENOENT (означающей, что данный файл или каталог не найден). Оболочка принимает это за чистую монету и завершает работу.

[1] Данный лимит определяется посредством BINFMT_BUF_SIZE в ядре.

6.3.2. Исполняемые объектные файлы

Исполняемые объектные файлы представляют собой файлы, связывание которых осуществляется без применения неразрешенных ссылок, а только при помощи ссылок динамических библиотек. Ядро распознает лишь ограниченное число форматов, использование которых поддерживается наряду с системным вызовом `execve`. В зависимости от процессорной архитектуры форматы могут различаться, однако общим является формат ELF[1]. До появления ELF общим форматом для всех Linux-систем был формат `a.out` (сокращение от *assembly output — ассемблерный вывод*), который может опционально использоваться и в наши дни.

Поддержка прочих форматов зависит от используемого ядра и процессорной архитектуры. Например, процессоры, у которых отсутствует аппаратный блок управления памятью Memory Management Unit (MMU), применяют так называемый плоский формат, поддерживаемый ядром. При использовании архитектуры MIPS ядро также поддерживает формат ECOFF, который является разновидностью формата Common Object File Format (COFF) — предшественника формата ELF.

Для идентификации объектного файла ядро ищет в нем подпись, обычно называемую *магическим числом*. Например, в файлах формата ELF подпись содержится в их первых четырех байтах, точнее, в байте `0x7f`, за которым следует строка `'ELF'`. Несмотря на то что все ELF-файлы обладают такой подписью, не все они являются исполняемыми. Например, скомпилированные модули (файлы с расширением `.o`) не относятся к исполняемым, хотя они представляют собой двоичные объектные файлы формата ELF. Когда ядро сталкивается с ELF-файлом, то в дополнение к магическому числу оно проверяет также ELF-заголовок, для того чтобы удостовериться, что данный файл действительно является исполняемым, перед тем как он будет загружен и исполнен. Поскольку компиляторы генерируют объектные файлы без каких-либо разрешений на исполнение, системный вызов `execve` никогда не должен случайным образом задействовать подобные файлы.

6.3.3. Разнообразные двоичные файлы

При помощи параметра `BINFMT_MISC` ядро позволяет расширять методику, посредством которой системный вызов `execve` оперирует исполняемыми файлами. Данный параметр указывается при сборке ядра и дает возможность суперпользователю определять приложения-помощники, которые `execve` может задействовать для запуска других программ. Это особенно полезно, если вы запускаете Windows-приложения при помощи `wine`[2] или будете иметь дело с исполняемыми Java-файлами, или `jar`-файлами.

Вполне очевидно, что ядро не сможет загрузить и выполнить исполняемый Windows-файл, однако при помощи `wine` можно запускать Windows-программы, подобно тому как интерпретатор запускает выполнение сценариев. Исполнение двоичных Java-файлов похожим образом осуществляет интерпретатор Java.

---

[1] Сокращенно от Executable and Linkable Format — исполняемый и связываемый формат.

[2] См. сайт www.winehq.org.

Если при сборке ядра был использован параметр BINFMT_MISC, вы сможете указать ядру, как оно должно распознавать «неродные» Linux-файлы и какую программу-помощник оно должно использовать для исполнения таких файлов. Поскольку все это осуществляется в рамках системного вызова execve, то приложению, использующему данный системный вызов, нет необходимости знать, что программный файл, который оно собирается исполнить, не относится к «родным» файлам Linux.

Для начала нам потребуется смонтировать особый элемент procfs, как показано далее:

```
$ mount binfmt_misc -t binfmt_misc /proc/sys/fs/binfmt_misc
```

При помощи этой команды осуществляется монтирование каталога с двумя элементами:

```
$ ls -l /proc/sys/fs/binfmt_misc/
total 0
--w-------  1 root root 0 Feb 12 15:19 register
-rw-r--r--  1 root root 0 Feb 11 20:06 status
```

Псевдофайл register будет использоваться для записи новых правил в ядро, а элемент status позволит активировать и деактивировать обработку ядром разнообразных двоичных файлов. Путем считывания этого файла можно также запрашивать соответствующее состояние.

Добавление новых правил осуществляется посредством записи строки особого формата в псевдофайл register. Данный формат включает ряд маркеров, разделяемых двоеточиями:

```
:name:type:offset:magic:mask:interpreter:flags
```

В поле name можно указать любое имя, которое будет отображаться в каталоге binfmt_misc для последующей ссылки. Поле type указывает ядру, как данное правило должно использоваться для распознавания типа файла. В этом поле можно поставить M, то есть *magic number (магическое число)*, или E — *extention (расширение)*. Если вы укажете магическое число (M), оставшаяся часть правила будет включать строку байтов для поиска, а также ее местоположение в файле. Если же в этом поле вы укажете расширение (E), остальная часть правила будет содержать указание ядру на то, какое расширение файла следует искать. Данный подход наиболее часто используется в случае с исполняемыми файлами DOS и Windows.

Поля offset, magic и mask служат для обработки так называемого магического числа. Поле offset является опциональным и указывает на первый байт файла, содержащий значение, которое ядро будет искать в качестве магического числа. Поле mask также является опциональным. В нем указывается битовая маска, которую ядро применяет по отношению к магическому числу (посредством поразрядного AND) перед тем, как тестировать его. Это дает возможность одним правилом определять целое семейство магических чисел. Магическое число и битовая маска указываются с использованием символов ASCII. При необходимости можно воспользоваться двоичными байтами, если они используют шестнадцатеричные последовательности переключения кода (*escape*-последовательности).

Для автоматической обработки исполняемых Windows-файлов можно задействовать wine, для чего потребуется следующее правило:

```
$ echo ':Windows:M::MZ::/usr/bin/wine:' >  /proc/sys/fs/binfmt_misc/register
```

В данном случае используется двухбайтовое магическое число (M) в сопровождении Z. Поскольку смещение не указывается, ядро считывает магическое число из начала файла. Также не указана маска, а это значит, что магическое число будет таким, каково оно в файле[1].

6.4. Синхронизация процессов при помощи wait

При создании процессов значительное внимание следует уделить тому, как происходит их завершение. Когда процесс завершается, он посылает своему родительскому процессу сигнал SIGCHLD. Сигнал SIGCHLD по умолчанию игнорируется, хотя соответствующая информация сохраняется. Состояние процесса остается в памяти до тех пор, пока родительский процесс не соберет его посредством одной из функций wait, приведенных далее:

```
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
pid_t wait3(int *status, int options, struct rusage *rusage);
pid_t wait4(pid_t pid, int *status, int options, struct rusage *rusage);
```

Как уже отмечалось в главе 5, ожидание завершения дочернего процесса называется его *уничтожением*. Если родительский процесс не дожидается завершения своего дочернего процесса, то последний переходит в так называемое состояние зомби, при котором ядро сохраняет достаточное количество сведений для информирования родительского процесса о состоянии выхода его дочернего процесса.

В операционной системе Linux (и UNIX) неважно, завершился дочерний процесс до или после того, как родительский процесс осуществлял вызов wait. Поведение функции wait в обоих случаях будет одинаковым, за исключением того, что она может блокироваться, если дочерний процесс не завершился при ее вызове. Если родительский процесс завершается раньше дочернего процесса, последний продолжает свое выполнение, однако при этом он *усыновляется* процессом init (pid 1). Когда дочерний процесс завершается, процесс init уничтожает соответствующее состояние. Любой дочерний зомби-процесс, выполняющийся после завершения его родительского процесса, усыновляется и уничтожается процессом init.

В табл. 6.1 приведен свод функций wait с пояснениями. Их общая черта заключается в том, что все они связаны с одним или несколькими одинаковыми системными вызовами Linux. Каждый из них принимает указатель на переменную int, в которой будет содержаться состояние процесса, и возвращает идентификатор завершенного процесса. В этом заключается базовая функция вызова wait. Функ-

[1] Следует отметить, что в RPM-пакете wine из дистрибутива Fedora содержится служба запуска, которая обеспечивает автоматическую обработку данных параметров.

ции `waitpid` и `wait4` добавляют в итоговый вывод идентификатор процесса, по этой причине вызывающий оператор может явно ожидать завершения одного из множества дочерних процессов. Функции `wait` и `wait3` не принимают `pid` в качестве аргумента. Данные функции возвращают значение, как только завершается какой-либо из дочерних процессов.

Таблица 6.1. Свод функций wait

Функция	Идентификатор процесса	Параметры	rusage	Пояснение
wait	Нет	Нет	Нет	Возвращает значение, как только завершается дочерний процесс, или сразу же, если выполняющиеся дочерние процессы отсутствуют
waitpid	Да	Да	Нет	Аналог wait, за тем исключением, что может возвращать значение немедленно, без блокирования, если потребуется. Также может возвращать значение, когда выполнение дочернего процесса приостанавливается
wait3	Нет	Да	Да	Поддерживает те же параметры, что и waitpid, однако не принимает pid в качестве аргумента. Возвращает значение, когда дочерний процесс завершается или приостанавливается в зависимости от того, что определено параметрами. Также возвращает структуру rusage struct, которая демонстрирует уровень потребления ресурсов дочерним процессом
wait4	Да	Да	Да	Аналог wait3, за тем исключением, что может принимать pid в качестве аргумента

Функции `waitpid`, `wait3` и `wait4` принимают аргумент `options`, который может иметь один из двух следующих флагов:

- `WNOHANG` — при использовании данного флага функция не блокируется и возвращает значение незамедлительно. Возвращаемое значение состояния будет равно `-1`, если уничтожению не подвергся ни один процесс;
- `WUNTRACED` — если задействован этот флаг, функция возвращает значения, касающиеся процессов, которые пребывают в приостановленном состоянии и не трассируются (например, отладчиком).

6.5. Объем памяти, занимаемый процессами

Как уже отмечалось в главе 5, каждый процесс обладает уникальным пространством виртуальной памяти (пространством пользователя). Помимо этого, процессы также обладают рядом других свойств и потребляют ресурсы не только виртуальной памяти.

Когда ядро создает процесс, он наделяется начальными значениями данных свойств и получает для себя рабочее пространство в виртуальной памяти. Все это частично определяется ядром, частично — компилятором и библиотеками. Пример типичной карты памяти на архитектуре IA32 можно увидеть на рис. 6.2.

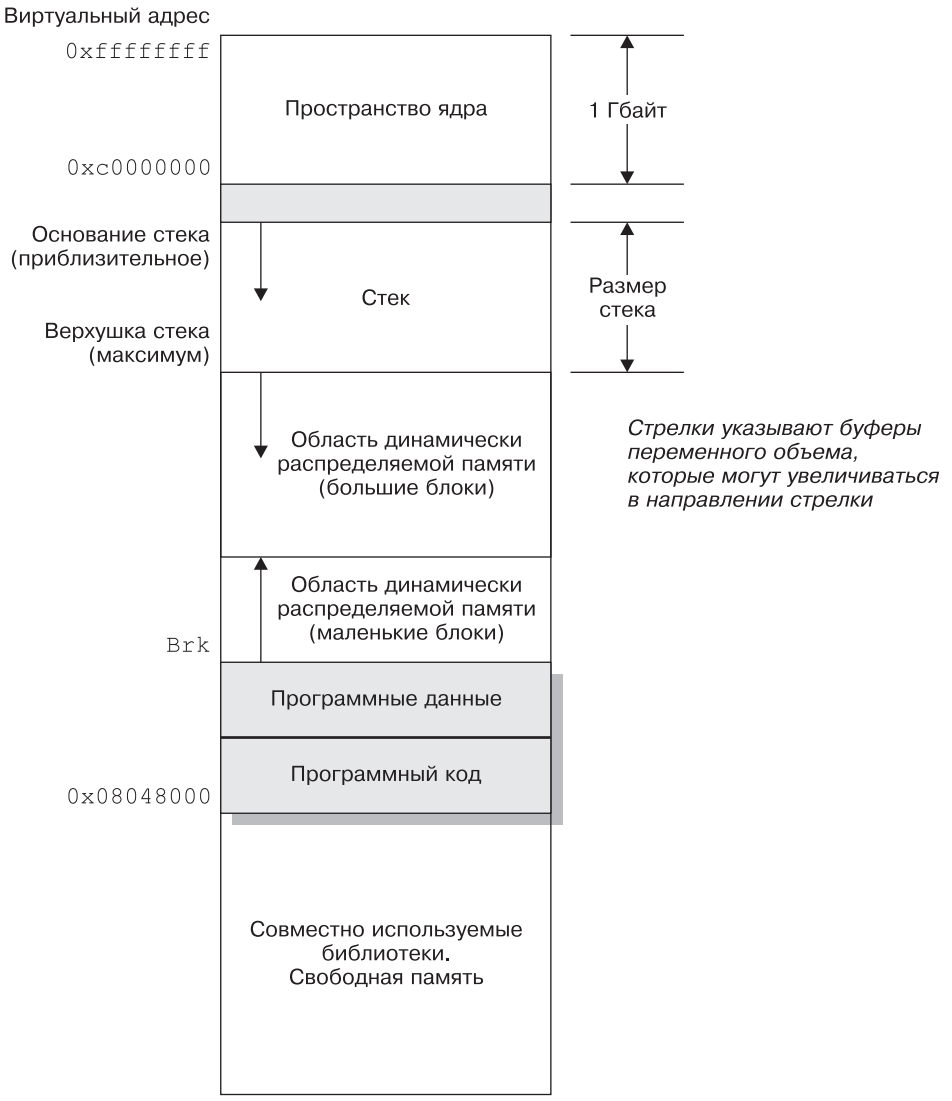


Рис. 6.2. Типичная карта памяти на архитектуре IA32

На рис. 6.2 приведено ядро, при сборке которого было произведено так называемое разделение «3 Гбайт/1 Гбайт», то есть нижняя часть виртуальных адресов размером 3 Гбайт принадлежит пространству пользователя, а верхняя размером 1 Гбайт — пространству ядра, которое совместно используется всеми процессами.

Подобное разделение будет справедливо для любого процесса, выполняющегося в данной системе.

Вы уже знаете, что идентификатор процесса уникально идентифицирует каждый процесс, протекающий в системе. Ядро использует его при поиске во внутренних таблицах информации, касающейся какого-либо процесса.

На рис. 6.2 приведена карта виртуальной памяти. Данная память не вся используется процессом; диаграмма демонстрирует лишь *предполагаемое* потребление. Память не «расходуется», пока она не распределена. Объем стека, например, может увеличиваться в предопределенных рамках (обычно это 1 Мбайт и больше), однако на начальном этапе ядро отводит ему лишь несколько страниц памяти. По мере роста стека ему будут выделяться дополнительные страницы. Более подробно о стеке мы поговорим позднее в этой главе.

Жизнь любого процесса, за исключением `init`, начинается в виде ответвления от другого процесса, то есть они стартуют не с чистой страницы. Карта памяти, изображенная на рис. 6.2, изначально заполнена родительскими отображениями. Данные будут идентичны родительским до тех пор, пока дочерний процесс не модифицирует память или не вызовет `exec`. Если дочерний процесс вызывает `exec`, страница очищается (так сказать) и карта заполняется только программным кодом, данными и стеком. В случае с программой на языке C процесс обычно заполняет карту некоторым количеством разделяемых библиотек и динамической памяти.

Что касается пространства ядра, то здесь все обстоит немного иначе. Когда происходит разветвление, дочерний процесс получает копию таблиц страниц в пространстве ядра. Память, необходимая для этого, уникальна для процесса, в силу чего он будет уникальным, несмотря на то что является все тем же клоном как в пространстве пользователя, так и в пространстве ядра. Дочерний процесс обзаводится собственным набором дескрипторов файлов, которые первоначально являются клонами принадлежащих родительскому процессу (подробнее об этом мы поговорим позднее в этой главе).

Объем памяти, занимаемой процессом, включает следующие элементы:

- страницы таблиц;
- стек (содержит переменные окружения);
- резидентную память;
- блокированную память.

Помимо этого, каждый процесс обладает свойствами, которые случайным образом могут совпадать с аналогичными свойствами других процессов, однако будут уникальными для каждого процесса в отдельности. К ним относятся:

- корневой каталог;
- текущий рабочий каталог;
- дескрипторы файлов;
- терминал;
- маска прав доступа `umask`;
- маска сигналов.

6.5.1. Дескрипторы файлов

Дескрипторы файлов представляют собой обычные целочисленные значения, возвращаемые посредством системного вызова open. Некоторые системные вызовы принимают дескрипторы файлов в качестве аргументов, которые они используют в виде индексов в важных структурах ядра. Вообще говоря, дескриптор файла — это простой индекс в таблице, которым ядро управляет для каждого процесса в отдельности.

Любой процесс обладает собственным набором дескрипторов файлов. При создании процесса он обычно располагает тремя открытыми дескрипторами файлов: 0, 1 и 2. Они означают стандартный ввод, стандартный вывод и стандартную ошибку, а соответственно все вместе именуются *стандартным вводом/выводом*. Они изначально наследуются от родительского процесса. Задача такого процесса, как sshd, заключается в том, чтобы гарантировать, что эти три дескриптора файлов будут ассоциированы с соответствующим псевдотерминалом или сокетом. Перед тем как дочерний процесс sshd станет вызывать exec, он должен закрыть эти дескрипторы файлов и открыть новые для стандартного ввода/вывода.

Каждый дескриптор файла имеет уникальные свойства, например разрешения на чтение или запись. Они указываются при совершении вызова open с использованием аргумента flags, например:

```
fd = open("foo",O_RDONLY);          Открыть файл с правом только на чтение
fd = open("foo",O_WRONLY);          Открыть файл с правом только на запись
fd = open("foo",O_RDWR);            Открыть файл с правом на чтение и запись
```

Флаги должны соответствовать разрешениям файла, в противном случае выполнение вызова open потерпит неудачу. Например, файл нельзя открыть с правом на запись, если текущий пользователь не обладает разрешением на запись в этот файл. Когда такое случается, вызов open возвращает значение ошибки -1, при этом переменной errno присваивается значение EACCESS («нет разрешения»).

Если выполнение вызова open оказывается удачным, атрибуты чтения/записи будут «приводиться в исполнение» исключительно дескриптором файла. Даже если в процессе выполнения программы разрешения файлов изменятся, это неважно. Применение разрешений файлов форсируется только при совершении вызова open.

Любой дескриптор файла обладает уникальными свойствами, даже если несколько таких дескрипторов указывают на одинаковый файл. Допустим, процесс располагает двумя открытыми файловыми дескрипторами, указывающими на один и тот же файл. Один из них был открыт с использованием O_RDONLY, другой — O_WRONLY. Любая попытка осуществить запись в дескриптор файла с правом только на чтение завершится неудачей с ошибкой EBADF («неверный дескриптор файла»). Аналогично попытка чтения из дескриптора файла с правом только на запись потерпит неудачу с такой же ошибкой.

Поскольку дескрипторы файлов уникальны для каждого процесса, они не могут совместно использоваться сразу несколькими процессами. Единственным исключением из данного правила являются родительские и дочерние процессы.

Дескрипторы файлов могут указывать на открытые файлы, устройства или сокеты. Каждый из них копируется при вызове fork и остается открытым после вызова exec[1]. Если оставлять дескрипторы файлов открытыми, особенно неиспользуемые, это может привести к необычным побочным эффектам. Здесь нам на помощь приходит ряд инструментов. Файловый элемент /proc для каждого процесса содержит подкаталог fd, в котором текущие открытые файлы представлены в виде символических ссылок.

Команда lsof

Команда lsof[2] позволяет просматривать открытые файлы любых процессов в системе. Для этого вам потребуются разрешения суперпользователя, в противном случае вам будут доступны файлы лишь тех процессов, владельцем которых вы являетесь. Команда lsof дает возможность просматривать не только дескрипторы файлов, но и множество других сведений. Сравните вывод данной команды с тем, который можно увидеть в /proc/pid/fd:

```
$ ls -l /proc/26231/fd
total 4
lrwx------ 1 root root 64 Feb 17 19:40 0 -> /dev/pts/0
lrwx------ 1 root root 64 Feb 17 19:40 1 -> /dev/pts/0
lrwx------ 1 root root 64 Feb 17 19:40 2 -> /dev/pts/0
lrwx------ 1 root root 64 Feb 17 19:40 255 -> /dev/pts/0

$ lsof -p 26231
COMMAND   PID USER   FD   TYPE DEVICE      SIZE NODE   NAME
bash    26231 root  cwd    DIR  253,0      4096 542913 /root
bash    26231 root  rtd    DIR  253,0      4096      2 /
bash    26231 root  txt    REG  253,0    686520 415365 /bin/bash
bash    26231 root  mem    REG  253,0    126648 608855 /lib/ld-2.3.5.so
bash    26231 root  mem    REG  253,0   1489572 608856 /lib/libc-2.3.5.so
bash    26231 root  mem    REG  253,0     16244 608859 /lib/libdl...
bash    26231 root  mem    REG  253,0     12924 606897 /lib/libtermcap...
bash    26231 root  mem    REG    0,0         0 [heap] (stat: ...
bash    26231 root  mem    REG  253,0  48501472 801788 /usr/lib/locale...
bash    26231 root  mem    REG  253,0     46552 606837 /lib/libnss_fil...
bash    26231 root  mem    REG  253,0     22294 862494 /usr/lib/gconv/...
bash    26231 root    0u   CHR  136,0         2 /dev/pts/0
bash    26231 root    1u   CHR  136,0         2 /dev/pts/0
bash    26231 root    2u   CHR  136,0         2 /dev/pts/0
bash    26231 root  255u   CHR  136,0         2 /dev/pts/0
```

Из приведенного примера видно, что команда lsof выводит намного больше информации, чем просто дескрипторы файлов. Обратите внимание на то, что в колонке FD данного вывода приводится ряд файлов в дополнение к тем, у которых есть файловые дескрипторы. Эти файлы имеют сокращения mem и txt, что говорит

---

[1] Примечательным исключением является дескриптор файла, возвращаемый функцией shm_open, что использует флаг FD_CLOEXEC.

[2] См. сайт ftp://lsof.itap.purdue.edu/pub/tools/unix/lsof.

о том, что они были отображены (загружены) в пространство процесса при помощи системного вызова mmap. Данные файлы не имеют дескрипторов, даже если они отображаются в память. Их можно удалять подобно файлам, открываемым при помощи дескрипторов, однако они будут продолжать занимать пространство в файловой системе до тех пор, пока какой-нибудь процесс не откроет их. В приведенном выводе также можно видеть сокращения cwd — текущий рабочий каталог и rtd — корневой каталог. Как отмечалось ранее, каждый процесс обладает собственными уникальными корневым и текущим рабочими каталогами (подробнее о них мы поговорим далее в этой главе).

Команда lsof представляет собой довольной сложный инструмент, поддерживающий множество параметров. Помимо страницы руководства man, в состав пакета lsof входит также файл QUICKSTART. В нем содержатся весьма полезные справочные сведения.

Лимиты использования дескрипторов файлов

Действительное число открытых файлов, которые может иметь процесс, определяется ядром, однако это можно сделать и при помощи функции sysconf:

```
sysconf(_SC_OPEN_MAX);
```

Функцию sysconf нам уже доводилось использовать ранее для определения размера страниц и частоты интервалов хода часов. Если данная функция вызывается с использованием аргумента _SC_OPEN_MAX, она возвращает максимальное число открытых файлов, которым может располагать процесс в любой момент времени. Когда процесс достигает своего лимита, все последующие вызовы open будут оканчиваться неудачей с ошибкой EMFILE («слишком много открытых файлов»). Неважно, будет это файл, устройство или сокет, лимит определяется разрешенным количеством файловых дескрипторов.

6.5.2. Стек

Стек представляет собой область памяти в пространстве пользователя, которая используется процессом для временного размещения данных. Стек получил свое название из-за того, что его поведение аналогично стеку элементов. Как и в случае с реальным стеком объектов, последний элемент, помещаемый в стек, станет первым, который будет удален из него. Иногда его также называют *буфером LIFO (last in, first out)*. Помещение данных в стек называется *проталкиванием*, а их удаление из стека — *выталкиванием*.

С точки зрения программиста, стек является местом, в котором переменные располагаются внутри функций. Переносимая программа C/C++ никогда не использует память стека напрямую, вместо этого полагаясь на компилятор, который распределяет переменные. Такой подход отлично работает в случае с языками функционального программирования, поскольку проталкивание переменных в стек может осуществляться в течение жизни функции, при этом компилятору нужно лишь удостовериться в том, что указатель стека восстановлен в своей исходной локации до завершения работы функции. Иными словами, распределение и высвобождение памяти осуществляется автоматически. Программа C/C++ обращается

к локальным переменным, размещаемым в стеке, как к *автоматическому* хранилищу, идентифицируемому при помощи ключевого слова `auto`. Оно по умолчанию является хранилищем локальных переменных, поэтому ключевое слово `auto` используется очень редко. Альтернативой автоматическому хранилищу является *статическое* хранилище, которое идентифицируется ключевым словом `static`. Локальные переменные, помеченные как `static`, не используют стек для хранения данных, а полагаются в этом на перманентное хранилище, которое выделяется редактору связей и/или загрузчику[1].

Основание (нижняя часть) стека размещается возле самого верхнего виртуального адреса пространства пользователя, и с этой точки стек начинает уменьшаться. Максимальный размер стека определяется в момент старта процесса. Предельный размер стека можно заново определять для новых процессов, однако после того, как процесс будет запущен, данную величину нельзя будет изменить. Если процесс начинает потреблять слишком много пространства стека, возникает так называемое *переполнение стека*. Операционная система Linux реагирует на переполнение стека посылкой соответствующему процессу сигнала `SIGSEGV`.

Местоположение основания стека нельзя предсказать со стопроцентной уверенностью, поскольку в силу некоторых особенностей ядра оно может произвольно меняться (см. врезку «Колоризация стека»). Поэтому, даже если вы точно не знаете, где располагается основание стека, можно предположить, что оно будет находиться где-то возле самого верхнего виртуального адреса пространства пользователя.

Колоризация стека

Адрес основания стека (его нижней части) будет разным для различных процессов благодаря использованию методики колоризации стека. Если основание стека будет всегда размещаться по одному и тому же виртуальному адресу, процессы, осуществляющие выполнение одинакового исполняемого файла, будут стремиться получить для переменных стека одни и те же виртуальные адреса при каждом своем запуске.

Это отрицательно сказывается на производительности систем на базе процессоров Intel с поддержкой многопоточной обработки Hyperthreading. Данная технология позволяет одиночному процессору функционировать подобно двум независимым процессорам в рамках одного чипа. В отличие от настоящих двухъядерных процессоров, которые содержат по два независимых ядра, процессор с поддержкой многопоточной обработки Hyperthreading распределяет большинство своих ресурсов между двумя логическими ядрами, включая кэш.

Когда два потока или процесса используют один и тот же адрес стека, они начинают бороться за одинаковые строки кэша, что приводит к конфликтам и снижению производительности.

[1] Ключевое слово static известно тем, что несет большую смысловую нагрузку (то есть имеет разные значения в разных контекстах). Вам достаточно знать, что в любом контексте оно будет означать перманентное хранилище и ограниченную область.

Благодаря произвольному выбору (рандомизации) адреса основания стека разные процессы будут обращаться к разным строкам кэша, избегая избыточного использования памяти.

Несмотря на то что колоризация стека не относится к средствам обеспечения безопасности, она все же позволяет немного увеличить ее. Некоторые атаки, связанные с переполнением буфера, основаны на предположении, что виртуальные адреса будут всегда одинаковыми. Рандомизация адреса основания стека снижает (но не исключает вовсе) вероятность успеха подобных атак.

Лимит размеров стека определяется при помощи системного вызова `setrlimit`, доступ к которому также можно получить посредством встроенной `bash`-команды `ulimit`. Новый установленный лимит будет касаться всех дочерних процессов текущего процесса.

6.5.3. Резидентная и блокированная память

Использование виртуальной памяти подразумевает, что некоторые части процесса не будут располагаться в оперативной памяти. Они будут размещаться на диске подкачки или их вовсе не будет. Например, страницы памяти, которые подверглись инициализации или доступу, не нуждаются в физическом распределении. Такие страницы не будут распределяться до тех пор, пока при доступе к памяти не произойдет ошибка страницы, после чего ядро приступит к их распределению.

Объем памяти, занимаемой частью процесса, включает потребляемое ею количество оперативной памяти, что характеризуется количеством *резидентной памяти*, которое имеет непосредственное отношение к частям памяти процесса, размещаемым в оперативной памяти. Сюда не входят части процесса, которые располагаются на диске подкачки или вообще нигде.

Подмножество резидентной памяти называется *блокированной памятью*, она имеет отношение к любой виртуальной памяти, явным образом блокированной процессом в оперативной памяти. Блокированная страница не может перемещаться на диск подкачки и всегда обитает в оперативной памяти. Процесс блокирует страницу для предотвращения роста латентности (задержек), который может быть вызван подкачкой. Блокировка страниц означает, что другим процессам будет доступен меньший объем оперативной памяти. По этой причине данную процедуру могут осуществлять лишь процессы с привилегиями корневого пользователя *root*.

6.6. Определение лимитов использования ресурсов процессами

Функция `setrlimit` применяется для установления лимитов ресурсов, которые могут отводиться процессам. Узнать текущие значения лимитов можно при помо-

щи вызова функции getrlimit. Об этих функциях мы вели речь еще в главе 5. Они определяются следующим образом:

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

Как отмечалось ранее, структура rlimit имеет *мягкий* и *жесткий* лимиты. Жесткий лимит обычно устанавливается при загрузке системы и остается неизменным. Если снизить значение жесткого лимита, то его нельзя будет увеличить для соответствующего процесса. Мягкий лимит можно увеличивать и понижать по желанию, однако он не должен превышать жесткий лимит. Структура rlimit определяется так:

```
struct rlimit {
        rlim_t rlim_cur; /* Мягкий лимит */
        rlim_t rlim_max; /* Жесткий лимит (предельное значение
                            для rlim_cur) */
};
```

Обратите внимание на то, что процесс может устанавливать лимиты только для себя самого; API-интерфейса для изменения значений лимитов других процессов не существует. Обычно лимиты использования ресурсов устанавливаются после вызова fork в дочернем процессе, но до вызова execve, например:

```
pid_t pid = fork();
if ( pid == 0 ) {
        struct rlimit limits = {…};
        getrlimit( RLIMIT_..., &limits);

        Модификация мягкого лимита.

        setrlimit( RLIMIT_..., &limits );
        exec( ... );
}
```

Вызывающий оператор указывает ресурсы, использование которых ограничивается в первом аргументе. Управляемые ресурсы включают многое, что касается текущего пользователя, а не только текущего процесса. Полный перечень можно увидеть в табл. 6.2.

Таблица 6.2. Флаги ресурсов, используемые в сочетании с setrlimit и getrlimit

Флаг ресурса	Описание
RLIMIT_AS	Ограничивает объем виртуальной памяти (адресного пространства), отводимой процессу. Устанавливаемое ограничение касается стека и динамической памяти. При превышении мягкого лимита попытки распределения динамической памяти (включая анонимные отображения посредством вызова mmap) будут заканчиваться неудачей с ошибкой ENOMEM. В случае если размер стека превысит установленный лимит, соответствующий процесс будет принудительно завершен при помощи сигнала SIGSEGV

Продолжение ⇩

Таблица 6.2 *(продолжение)*

Флаг ресурса	Описание
RLIMIT_CORE	Ограничивает размер файла образа памяти. Если указать нулевое значение, это приведет к отключению генерирования файлов образов памяти, что может быть уместно по соображениям безопасности, однако нежелательно при разработке программного обеспечения
RLIMIT_CPU	Ограничивает количество процессорного времени, отводимого процессу. Исчисление осуществляется в секундах. При превышении мягкого лимита процесс станет каждую секунду получать сигнал SIGXCPU до тех пор, пока не будет превышен жесткий лимит, после чего поступающий сигнал сменится на SIGKILL
RLIMIT_DATA	Максимальный размер сегмента данных. Влияет на совершение вызовов brk и sbrk, что (теоретически) означает, что попытки распределения динамической памяти будут терпеть неудачу при достижении мягкого лимита. Переменной errno в таком случае будет присваиваться значение ENOMEM. Библиотека glibc использует вызов mmap, если совершение вызова brk заканчивается неудачей, эффективно нейтрализуя данное свойство
RLIMIT_FSIZE	Максимальный размер отдельного файла, создаваемого процессом. Если процесс превышает мягкий лимит, он получает сигнал SIGXFSZ, а выполнение системных вызовов write и truncate будет заканчиваться неудачей с ошибкой EFBIG
RLIMIT_LOCKS	Ограничивает количество блокировок, которые может устанавливать процесс за раз; не используется в версии Linux 2.6
RLIMIT_MEMLOCK	Определяет максимальное число байт, которые процесс может блокировать за раз; данное значение может быть изменено привилегированными пользователями
RLIMIT_NOFILE	Ограничивает количество дескрипторов файлов, которые процесс может открыть за один раз
RLIMIT_NPROC	Максимальное число процессов, которое может быть создано с использованием реального идентификатора пользователя вызывающего процесса. При достижении мягкого лимита выполнение вызова fork будет терпеть неудачу с присваиванием переменной errno значения EAGAIN
RLIMIT_RSS	Ограничивает размер резидентной памяти процесса, который представляет собой объем всей резидентной памяти, доступной процессу; предположительно используется в операционной системе Linux, однако в версии 2.6.14 отсутствует
RLIMIT_SIGPENDING	Ограничивает число сигналов в очереди (ожидающих) для конкретного процесса. См. главу 7
RLIMIT_STACK	Определяет максимальный размер стека процесса

Если процесс попытается превысить один из установленных лимитов использования ресурсов, его дальнейшее поведение будет зависеть от ограничиваемого ресурса. Если, к примеру, стек имеет ограниченный размер, выполнение процесса будет отменено посредством сигнала `SIGSEGV`, когда пользователь попытается отвести слишком большой объем под автоматическое хранилище. В то же время, если ограничить число дескрипторов файлов, то процесс, скорее всего, потерпит неудачу при совершении вызова `open` с возвратом значения `-1`.

Поверка значений лимитов посредством `getrlimit` является способом обеспечить устойчивое выполнение операций. Теоретически весьма разумно было бы использовать для этого системный вызов `getrusage`. К сожалению, в Linux данный

вызов позволяет выводить лишь малую часть сведений о потреблении ресурсов каким-либо приложением. Столь краткую информацию можно получить посредством вызова `exit`, что делает `getrusage` малопригодным инструментом в данной ситуации.

Вам может быть интересно, зачем нужно устанавливать лимиты на использование ресурсов процессами? На это есть несколько причин. Первая заключается в необходимости предотвратить снижение производительности системы из-за распределения и использования слишком больших объемов памяти. Вредоносные (или написанные с ошибками) программы могут привести к замедлению работы системы, если им выделяется много памяти. Чрезмерное количество ошибок страниц, вызванное одним процессом, может привести к серьезному увеличению латентности других процессов и общему снижению производительности. Вторая причина использования лимитов состоит в необходимости отключать генерирование файлов образов памяти, в которых могут содержаться пароли и другие важные сведения. Думаете, я преувеличиваю? Взгляните на листинг 6.1, где приведен простой пример того, как даже зашифрованный пароль может быть выставлен напоказ в файле образа памяти.

Листинг 6.1. `insecure.c: пароль, выставленный напоказ в файле образа памяти`

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

// Зашифрованное сообщение, взлом которого может занять годы.
unsigned char secret_message[] = {
  0x8f, 0x9e, 0x8c, 0x8c, 0x88, 0x90, 0x8d, 0x9b, 0xc2, 0x8b, 0x90,
  0x8f, 0xdf, 0x8c, 0x9a, 0x9c, 0x8d, 0x9a, 0x8b
};

int main(int argc, char *argv[])
{
  int i;
  for (i = 0; i < sizeof(secret_message); i++) {
    // На взлом могут и не уйти годы, но идею вы поняли.
    secret_message[i] ^= 0xff;
  }
  abort();
}
```

Пароль может быть защищен шифрованием, однако если программа осуществляет его расшифровку в памяти, он станет видимым, когда она сбросит дамп файла образа памяти. Пароль будет защищен ровно настолько, насколько будет защищен файл образа памяти. Если дамп этого файла будет сброшен на диск, к нему сможет получить доступ любой пользователь, обладающий разрешением на чтение этого файла. По этой причине ядро создает файлы образов памяти с ограниченными разрешениями, благодаря чему их сможет прочесть только их владелец. Это единственная скромная защита от неавторизованных пользователей, которые могут попытаться получить доступ к важной информации. Иногда бывает так, что даже владелец исполняемого файла может оказаться не авторизованным на просмотр файла образа памяти.

6.7. Процессы и файловая система procfs

procfs представляет собой псевдофайловую систему, из которой пользователь может почерпнуть сведения о системе в целом и отдельных процессах. Некоторые операционные системы известны тем, что содержат множество заумных системных вызовов, позволяющих извлекать информацию из procfs. Однако в Linux для этого используются лишь несколько системных вызовов: open, close, read и write. Следует отметить, что содержимое файловой системы procfs не подчиняется каким-либо стандартам. Теоретически программа, которая позволяет считывать данные из procfs, может превосходно функционировать на одном ядре и оказаться неработоспособной на другом.

Файловая система procfs по общему правилу монтируется в каталог /proc. В ней имеется дерево данных о системе и процессах. Большинство представленной здесь информации нельзя получить посредством системных вызовов, поэтому пользователям зачастую приходится обращаться сюда.

Информация, содержащаяся в файловой системе procfs, главным образом представлена в формате ASCII.

В каталоге /proc каждому процессу отводится отдельный подкаталог, называемый в соответствии с идентификатором процесса, то есть каталог процесса 123 будет называться /proc/123 и существовать на протяжении жизненного цикла данного процесса. Кроме того, здесь также имеется каталог /proc/self, который представляет собой ссылку на каталог текущего выполняющегося процесса.

Внутри каждого подкаталога вы сможете найти сведения, о которых мы говорили в разделе, посвященном объему памяти, занимаемому процессами, а также некоторую дополнительную информацию. Часть этих сведений довольно полезна, при этом некоторые из них достаточно специфичны. Тем не менее ознакомиться с ними никогда не помешает.

Каталог /proc является превосходным средством отладки, позволяющим извлекать сведения о системе, не прибегая к использованию дополнительного инструментария. Реальное содержимое этого каталога может варьироваться в зависимости от параметров и версии ядра. В табл. 6.3 вы сможете увидеть перечень наиболее характерных файлов, которые здесь содержатся, а также ознакомиться с их описанием.

Таблица 6.3. Файлы, содержащиеся в каталоге /proc/PID

Файл	Описание	Формат
auxv	Вектор значений, используемый инструментами вроде gdb, содержащий информацию о системе	Двоичный
cmdline	ASCII-строка с разделителями NUL (0) ASCII, представляющая собой вектор argv, используемый программами на языке C	ASCII
cwd	Символическая ссылка на текущий рабочий каталог процесса	Нет данных
environ	Окружение процесса, «упакованное» в одну ASCII-строку с разделителями NUL ASCII. Каждый маркер имеет такой же вид, как и в векторе envp, передаваемом программам на языке C (например, PATH=xyz:abc)	ASCII

Файл	Описание	Формат
exe	Символическая ссылка на файл, содержащий программный код для данного процесса	Нет данных
fd	Каталог, содержащий символическую ссылку на каждый дескриптор файла, открываемый процессом. Эти ссылки включают все, что открывается посредством дескриптора файла, в том числе плоские файлы, сокеты и конвейеры (каналы)	Нет данных
maps	Текстовое представление памяти пространства пользователя, занимаемой процессом. В случае с потоками ядра данный файл будет пуст, поскольку такие потоки не имеют пространства пользователя	Текстовый ASCII
mem	Файл, позволяющий другим процессам получать доступ к пространству пользователя данного процесса; используется программами наподобие gdb	Двоичный
mounts	Список смонтированных файловых систем, например /etc/mtab. Одинаков для всех процессов	Текстовый
oom_adj	Позволяет пользователям настраивать oom_score (см. далее)	Текстовый
oom_score	«Негодность» процесса, определяемая «убийцей» процессов OOM (Out-Of-Memory). Когда система начинает испытывать нехватку памяти, первыми принудительно завершаются («убиваются») процессы с высокими значениями оценки	Текстовый
root	Символическая ссылка на корневую файловую систему процесса; обычно указывает на / или другую локацию, если процесс осуществлял вызов chroot	Нет данных
smaps	Подробный список отображений (загрузок) в память разделяемых библиотек, используемых данным процессом. В отличие от maps, здесь содержится больше детальной информации об отображениях, включая количество «чистых» и «грязных» страниц	Текстовый ASCII
stat	Однострочное, scanf-дружественное представление состояний процессов, используемое командой ps	Текстовый ASCII
statm	Содержит сведения о суммарном использовании памяти процессами, большая часть которых приведена в stat	Текстовый ASCII
status	Содержит ту же информацию, что и stat, но в удобочитаемом виде	Текстовый ASCII
wchan	Служит индикатором функции ядра, которая вызывает блокировку процесса (если применяется)	Текстовый

Формально Linux может вполне нормально функционировать и без файловой системы `procfs`, хотя немалая часть инструментов зависит от содержимого `/proc`. Любой из них может внезапно оказаться неработоспособным, если вы попытаетесь использовать систему без `procfs`.

6.8. Инструменты для управления процессами

Проект procps[1] содержит множество инструментов для работы с файловой системой procfs и входит в состав большинства дистрибутивов. Некоторые инструменты, содержащиеся в этом пакете, например команда ps, подчиняются стандарту POSIX. Прочие инструменты пока являются нестандартными, однако не менее полезными. Зачастую намного удобнее пользоваться ими, чем самостоятельно копаться в каталогах /proc. Прежде чем приступать к написанию сценария для обследования каталогов /proc, сначала следует заглянуть в данный пакет.

6.8.1. Вывод информации о процессах при помощи команды ps

Команда ps из пакета procps реализует опции, определяемые как стандартом POSIX, так и другими стандартами. Несмотря на то что данные стандарты имеют некоторые общие черты, каждый из них предъявляет свои требования к аргументам, которые нельзя легко изменить, не затрагивая основную часть клиентского кода (сценариев). В результате вы обнаружите, что команда ps может как минимум двумя разными способами выполнить одинаковую задачу, присваивать одному и тому же полю два разных названия и т. д. Неудивительно, что страница руководства man для этой команды довольно плотно заполнена справочной информацией.

При вводе команды ps без аргументов она выводит сведения лишь о тех процессах, владельцем которых является текущий пользователь и которые связаны с текущим терминалом. Обычно сюда входят все процессы, которые были запущены в текущей оболочке, хотя данная ситуация не имеет ничего общего с концепцией оболочки о процессах, выполняющихся в фоновом и приоритетном режимах. Как отмечалось ранее, оболочка отслеживает процессы как *задания*, выполняющиеся в приоритетном или фоновом режиме. Две разные оболочки могут использовать один и тот же терминал, однако выводимые задания будут касаться только текущей оболочки, в то время как вывод команды ps по умолчанию будет включать процессы, запущенные из обеих оболочек, например:

```
$ tty                          Как называется наш терминал?
/dev/pts/1
$ ps
  PID TTY          TIME CMD
21563 pts/1    00:00:00 bash
21589 pts/1    00:00:00 ps     В данном терминале запущено только
                               два процесса.
$ sleep 1000 &
[1] 21590                      Выполняющихся в фоновом режиме заданий - 1,
$ jobs -l                      идентификатор процесса - 21590.
[1]+ 21590 Running  sleep 1000 &
```

[1] См. сайт http://procps.sourceforge.net.

```
$ ps
  PID TTY          TIME CMD
21563 pts/1    00:00:00 bash
21590 pts/1    00:00:00 sleep
21591 pts/1    00:00:00 ps      Сведения о новом процессе выводятся командой
                                ps и jobs.
$ bash                          Запуск новой оболочки в том же терминале.
$ jobs                          Список заданий пуст, поскольку это новая
                                оболочка.
$ ps                            Команда ps выводит сведения о процессах,
  PID TTY          TIME CMD     запущенных из обеих оболочек.
21563 pts/1    00:00:00 bash
21590 pts/1    00:00:00 sleep
21592 pts/1    00:00:00 bash
21609 pts/1    00:00:00 ps
```

В большинстве случаев вам потребуются более подробные сведения, чем те, что по умолчанию выводит команда `ps`. Вы можете захотеть узнать, какие процессы протекают вне вашего терминала, или увидеть более детальную информацию о состоянии своего процесса. Для этого необходимо воспользоваться параметром `-l`. Он позволяет получить более обширный перечень свойств процесса, например:

```
$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY        TIME CMD
0 S   500 21563 21562  0  75   0 -  1128 wait   pts/1  00:00:00 bash
0 S   500 21590 21563  0  76   0 -  974  -      pts/1  00:00:00 sleep
0 R   500 21623 21563 96  85   0 -  1082 -      pts/1  00:00:26 cruncher
0 R   500 21626 21563  0  75   0 -  1109 -      pts/1  00:00:00 ps
```

В данном выводе заголовки полей содержат краткое, порой малопонятное описание колонок. В табл. 6.4 вы сможете отыскать пояснения к описаниям колонок из этого расширенного вывода по умолчанию, которые приведены в нем, по порядку слева направо.

В данном выводе могут как присутствовать дополнительные сведения помимо тех, которые вас интересуют, так и отсутствовать важные для вас. К счастью, команда `ps` позволяет «настраивать» внешний вид итогового вывода в точном соответствии с пожеланиями пользователя.

Таблица 6.4. Колонки, содержащиеся в расширенном выводе команды ps

Заголовок колонки	Описание
F	Флаги (см. sched.h)
S	Состояние процесса: R — запущен; S — находится в «спячке» (прерываемой); T — приостановлен; D — находится в «спячке» (непрерывной); Z — завершен, но не уничтожен
UID	Эффективный идентификатор пользователя процесса

Продолжение ⇩

Таблица 6.4 *(продолжение)*

Заголовок колонки	Описание
PID	Идентификатор процесса
PPID	Идентификатор родительского процесса
C	Использование ресурсов центрального процессора в процентном отношении
PRI	Приоритет процесса
NI	Значение nice процесса
ADDR	Не используется в Linux
SZ	Приблизительный объем виртуальной памяти процесса в страницах
WCHAN	Системный вызов или функция ядра, отправляющие процесс в «спячку» (если таковые имеются)
TTY	Управляющий терминал
TIME	Количество процессорного времени, потребляемого процессом
CMD	Имя команды, соответствующее указанному в /proc/stat (сокращенное до 15 символов)

6.8.2. Вывод расширенных сведений о процессах с использованием форматирования

Пакет `procps` также предназначен для использования в операционных системах, отличных от Linux, о чем можно судить по команде `ps`, которая входит в его состав. В частности, поля форматирования, используемые параметром `-o`, представляют собой разношерстную связку мнемоник, производных от тех, что использовались в различных UNIX-системах на протяжении долгих лет. Многие из них являются синонимами, поскольку каждый производитель выбирал немного отличавшуюся мнемонику. Компания SGI использовала одну мнемонику, Sun — вторую, а Hewlett-Packard — третью. Команда `ps` из пакета `procps` версии 3.2.6 поддерживает 236 разных параметров форматирования (86 из которых не позволяют генерировать каких-либо полезных сведений). На странице руководства man можно отыскать документацию лишь к некоторым из них.

В табл. 6.5 можно увидеть перечень наиболее полезных параметров форматирования. Часто бывает так, что некоторые из них позволяют выводить идентичные сведения. Иногда выходной формат может немного отличаться; в других ситуациях один и тот же параметр форматирования может иметь несколько псевдонимов.

Таблица 6.5. Параметры форматирования, поддерживаемые командой ps

Параметр форматирования		Описание
Касающийся времени	start, start_time, lstart, bsdstart	Время и дата запуска процесса. Каждый параметр форматирования генерирует немного отличающийся вывод. Некоторые из них включают в итоговый вывод дату, другие — секунды, а все вместе — часы и минуты
	etime	Время, прошедшее с момента запуска процесса

Параметр форматирования		Описание
Касающийся времени	time, cputime, atime, bsdtime	Совокупное количество процессорного времени, отнятого процессом, в часах, минутах и секундах. При использовании bsdtime время выводится только в минутах и секундах
Касающийся памяти	size	Приблизительный суммарный объем подкачиваемой памяти процесса; сюда входят стек и динамическая память
	m_size	Объем виртуальной памяти процесса, сообщаемый ядром; немного отличается от size
	pmem, %mem	Объем резидентной памяти процесса, выражаемый в процентах от общего количества физической памяти, доступной в системе
	majflt, maj_flt, pagein	Количество значительных ошибок страниц, определяемых ядром
	minflt, min_flt	Количество незначительных ошибок страниц, определяемых ядром
	sz, vsz, vsize	Суммарный объем виртуальной памяти, используемой процессом. sz выражается в страницах, а vsz и vsize — в килобайтах
	rss, rssize, rsz	Суммарный объем резидентной памяти, который отводится процессу в оперативной памяти; выражается в килобайтах
	lim	Лимит резидентной памяти процесса rss, устанавливаемый посредством setrlimit
	stackp	Адрес основания выделенного стека; остается фиксированным до момента увеличения размера стека
Касающийся планировщиков	cpu	В системах с поддержкой симметричной многопроцессорной обработки SMP идентифицирует центральный процессор, который осуществляет обработку процессов; также применим для однопроцессорных систем
	policy, class, cls, sched	Определяет класс планировщика процесса в виде числа или мнемоники: TS (0) — нормальный процесс, с использованием интервалов времени; FF (1) — процесс реального времени, с использованием политики планирования FIFO; RR (2) — процесс реального времени, с использованием политики циклического планирования
	cp, %cpu, c, util	Использование ресурсов центрального процессора в процентах
	pri	Приоритет, выражаемый в виде положительного целого числа. Чем выше значение, тем выше будет приоритет (нормальный процесс имеет приоритет в диапазоне от 0 до 39; процесс реального времени — от 41 до 99; приоритет, равный 40, в операционной системе Linux не используется). Данные значения аналогичны тем, что содержатся в ядре

Продолжение ⇩

Таблица 6.5 *(продолжение)*

Параметр форматирования		Описание
Касающийся планировщиков	priority	Приоритет, в случае с которым более низкие значения означают более высокий приоритет (нормальный процесс будет иметь приоритет в диапазоне от 39 до 0, а процесс реального времени — от –1 до –99). Данные значения аналогичны тем, что содержатся в /proc/PID/stat/
	opri, intpri	Инвертированная версия параметра форматирования priority (диапазон от –39 до 0 — для нормальных процессов, от 1 до 99 — для процессов реального времени). Положительные значения используются для процессов реального времени, а отрицательные — для нормальных процессов
	s, state, stat	Состояние процесса. state может иметь значение D, R, S, T или Z. При использовании stat добавляется еще один символ для более подробного отчета
	tid, spid, lwp	Определяет идентификаторы потоков в многопоточных процессах. Потоки доступны для просмотра только при помощи параметра –T
	wchan, wname	Имя системного вызова или функции ядра, вызывающей блокировку процесса если он не находится в «спячке»

6.8.3. Поиск процессов по имени при помощи команд ps и pgrep

Иногда вам может потребоваться узнать, что происходит с программой (процессом), выполнение которой вы запускали из другого терминала или, быть может, во время загрузки системы. Вы можете помнить имя этой программы, но не знать соответствующего идентификатора процесса. В такой ситуации необходимо воспользоваться командой следующего вида:

```
$ ps -ef | grep myprogram
```

Аналогичной, но менее гибкой командой является `pidof`, которая также принимает имя программы в качестве аргумента. Она не входит в состав `procps`, а является частью пакета `SysVinit`, который используется загрузочными сценариями. Данная команда располагается в `/sbin` и предназначена для использования сценариями загрузки. Поскольку пакет `SysVinit` присутствует далеко не во всех дистрибутивах, разумно будет избегать его, если вы не хотите, чтобы у вас возникли проблемы с переносимостью программ.

6.8.4. Просмотр сведений о потреблении памяти процессами при помощи команды pmap

Ранее мы уже прибегали к использованию команды `pmap` для просмотра информации о том, как процессы используют память. Интересующие нас сведения содер-

жатся в /proc/PID/maps, где можно отыскать карту виртуальной памяти любого процесса, например:

```
$ cat &
[1] 3989
$ cat /proc/3989/maps
009db000-009f0000 r-xp 00000000 fd:00 773010     /lib/ld-2.3.3.so
009f0000-009f1000 r-xp 00014000 fd:00 773010     /lib/ld-2.3.3.so
009f1000-009f2000 rwxp 00015000 fd:00 773010     /lib/ld-2.3.3.so
009f4000-00b15000 r-xp 00000000 fd:00 773011     /lib/tls/libc-2.3.3.so
00b15000-00b17000 r-xp 00120000 fd:00 773011     /lib/tls/libc-2.3.3.so
00b17000-00b19000 rwxp 00122000 fd:00 773011     /lib/tls/libc-2.3.3.so
00b19000-00b1b000 rwxp 00b19000 00:00 0
08048000-0804c000 r-xp 00000000 fd:00 4702239    /bin/cat
0804c000-0804d000 rwxp 00003000 fd:00 4702239    /bin/cat
0804d000-0806e000 rwxp 0804d000 00:00 0          [heap]
b7d1e000-b7f1e000 r-xp 00000000 fd:00 1232413    /usr/../locale-archive
b7f1e000-b7f20000 rwxp b7f1e000 00:00 0
bfd1b000-bfd31000 rw-p bfd1b000 00:00 0          [stack]
ffffe000-fffff000 ---p 00000000 00:00 0          [vdso]
```

Формат данного вывода может оказаться сложным для восприятия, поэтому в случае необходимости вы можете отыскать его полное описание на странице руководства man proc(5). Каждая представленная здесь строка является диапазоном виртуальной памяти. Диапазон адресов приводится в первой колонке слева. Во второй колонке демонстрируются разрешения, а s и p означают совместные *(shared)* и частные *(private)* отображения. В следующей колонке указывается смещение устройства, которое будет использоваться для эквивалентного вызова mmap. В случае с анонимными картами памяти это будет аналогом виртуального адреса. Если в виртуальную память загружены (отображены) файл или устройство, то в следующих полях будет указан идентификатор устройства в виде старшего/младшего номера, после чего следует индексный дескриптор[1] и, наконец, имя файла/устройства.

Аналогичная карта, генерируемая при помощи команды pmap, имеет немного более дружественный пользователю вид:

```
$ pmap 3989
3989:   cat
009db000     84K r-x--  /lib/ld-2.3.3.so
009f0000      4K r-x--  /lib/ld-2.3.3.so
009f1000      4K rwx--  /lib/ld-2.3.3.so
009f4000   1156K r-x--  /lib/tls/libc-2.3.3.so
00b15000      8K r-x--  /lib/tls/libc-2.3.3.so
00b17000      8K rwx--  /lib/tls/libc-2.3.3.so
00b19000      8K rwx--    [ anon ]
08048000     16K r-x--  /bin/cat
0804c000      4K rwx--  /bin/cat
0804d000    132K rwx--    [ anon ]
```

[1] Подробнее об индексных дескрипторах мы поговорим в главе 7.

```
b7d1e000   2048K r-x--  /usr/lib/locale/locale-archive
b7f1e000      8K rwx--    [ anon ]
bfd1b000     88K rw---    [ stack ]
ffffe000      4K -----    [ anon ]
 total     3572K
```

6.8.5. Отправка сигналов процессам на основе их имен

Команды `skill` и `pkill` функционируют аналогично команде `kill`, за тем исключением, что при поиске соответствий они полагаются на имена процессов, а не на их идентификаторы. С данными командами следует обращаться как с заряженным оружием. То есть *использовать их с осторожностью.*

Команда `skill` принимает имя процесса в качестве аргумента и осуществляет поиск только точных соответствий. При этом она руководствуется содержимым `/proc/PID/stat`. Операционная система Linux сохраняет в `/proc/PID/stat` только первые 15 символов имени программы[1], поэтому, если вы ищете процесс с необыкновенно длинным командным именем, вам не удастся сделать это при помощи `skill`. Ситуация может стать еще запутаннее, если у вас запущены программы, длина имен которых точно равна 15 символам, и программы, длина имен которых превышает это значение, но включает аналогичные 15 первых символов, например:

```
$ ./image_generator &                  Длина имени составляет ровно
 [1] ...                               15 символов
$ ./image_generator1 &                 Длина имени составляет ровно
 [2] ...                               16 символов
$ skill image_generator                Принудительно завершаются
[1] Terminated ./image_generator       оба процесса!
[2] Terminated ./image_generator1
```

Функционирование команды `pkill` основано на аналогичном принципе, за исключением того, что она использует регулярное выражение при поиске имени процесса. Это еще более рискованно, поскольку в данном случае она будет искать любые соответствия в командной строке. `pkill`, в отличие от команды `skill`, использует `/proc/cmdline`, где вектор `argv` сохраняется целиком таким, каков он есть. Если не проявлять внимательность, можно по неосторожности принудительно завершить не те процессы, например:

```
$ ./proc_abc &
[1] ...
$ ./abc_proc &
[2] ...

$ pkill abc                    Принудительно завершаются оба процесса!
$ pkill ^abc                   Принудительно завершается только процесс
                               abc_proc.
```

[1] Данный лимит определяется посредством TASK_COMM_LEN в ядре.

```
$ pkill abc\$                    Принудительно завершается только процесс
                                 proc_ abc. ($ экранируется при помощи
                                 обратного слеша.)
```

Поскольку аргумент является регулярным выражением, первая из приведенных ранее команд принудительно завершает оба процесса, так как последовательность букв `abc` присутствует в именах обоих процессов. Для того чтобы конкретизировать параметры поиска соответствий, можно указать имя процесса целиком либо использовать уточняющий синтаксис в регулярном выражении. Такой подход применен в последующих командах. Регулярное выражение `^abc` указывает на то, что имя процесса должно начинаться с букв `abc`, а это позволяет избежать принудительного завершения процесса `proc_abc`. Аналогично регулярное выражение `abc$` говорит о том, что имя процесса должно заканчиваться буквами `abc`. Это предотвращает принудительное завершение процесса `abc_proc`.

6.9. Заключение

В данной главе вы изучили аспекты процессов, связанные с пространством пользователя. Были детально рассмотрены принципы использования вызова `exec`, а также ряд ухищрений, к которым операционная система Linux прибегает при выполнении различных программ. Кроме того, вы увидели примеры «подводных камней», с которыми можно столкнуться при использовании `exec`.

Мы подробно рассмотрели потребление ресурсов процессами и особенности извлечения сведений о них. В заключение вы изучили ряд инструментов, с помощью которых осуществляется управление процессами из оболочки.

6.9.1. Системные вызовы и API-интерфейсы, использованные в этой главе

- `execve` — системный вызов Linux, применяемый для инициализации пространства пользователя процессов и выполнения программного кода. Стандарт POSIX определяет ряд функций со схожими подписями, однако все они используют этот системный вызов. Обычно он задействуется после совершения вызова `fork`. См. `exec(3)`.
- `fcntl` — используется для установки флагов в дескрипторах файлов. С ее помощью мы устанавливали флаг `FD_CLOEXEC`.
- `fork` — системный вызов для создания клона текущего процесса. Это первый этап в создании нового процесса.
- `kill` — системный вызов для отправки сигнала выполняющемуся процессу.
- `setrlimit`, `getrlimit` — функции для проверки и установления лимитов использования ресурсов процессами.
- `sysconf` — возвращает системные константы, используемые в работе.
- `wait`, `waitpid`, `wait3`, `wait4` — позволяют родительскому процессу синхронизироваться с дочерним процессом.

6.9.2. Инструментарий, использованный в этой главе

- `pgrep` — осуществляет поиск процессов, имена которых соответствуют регулярному выражению.
- `pmap` — выводит на экран карту памяти процесса.
- `ps` — команда, широко используемая для вывода сведений о состоянии процесса.
- `ulimit` — встроенная `bash`-функция для проверки и определения лимитов использования ресурсов процессами.

6.9.3. Веб-ссылки

- http://procps.sourceforge.net — домашняя страница проекта `procps`, в состав которого входит множество полезных инструментов для отслеживания процессов и ресурсов системы.
- www.unix.org — ресурс, где публикуются спецификации Single UNIX Specification.
- www.unix.org и www.opengroup.org — ресурсы, на которых можно отыскать публикации, касающиеся стандарта POSIX (IEEE Standard 1003.2) и многих других (требуется регистрация).

7 Взаимодействие между процессами

7.1. Введение

Каждый из процессов обладает отдельным адресным пространством, поэтому добиться взаимодействия между ними не всегда оказывается легко. Существует ряд методик, обеспечивающих функционирование механизма межпроцессного взаимодействия IPC *(Interprocess communication)*, каждая из которых имеет свои преимущества и недостатки.

Ключевая проблема, возникающая в приложениях, оперирующих большим числом процессов или потоков, заключается в состоянии гонки. *Состояние гонки* — это ситуация, когда несколько процессов (или потоков) пытаются одновременно внести изменения в одни и те же данные. Без обеспечения синхронизации нет никаких гарантий того, что один процесс не станет «затирать» вывод другого процесса. А что, возможно, еще важнее, так это то, что состояния гонки могут вызвать непредсказуемость итогового вывода любого из процессов.

Обычно причиной возникновения состояний гонки является отсутствие синхронизации, что может привести к варьированию содержимого итогового вывода в зависимости от нагрузки на систему и прочих факторов. Ряд примеров можно было наблюдать в главе 6, где итоговый вывод для родительского и дочерних процессов каждый раз выглядел по-другому при повторном выполнении одной и той же команды. Состояния гонки никогда не проявляются подобным образом. Во многих ситуациях они могут не проявиться даже после того, как соответствующая программа будет выпущена в обращение.

Механизм межпроцессного взаимодействия IPC жизненно необходим для предотвращения возникновения состояний гонки. Однако в случае неправильного использования он может даже способствовать, а не препятствовать их возникновению. В этой главе вы узнаете, как следует пользоваться механизмом межпроцессного взаимодействия IPC, а также освоите ряд инструментов, позволяющих осуществлять отладку процессов во время его использования.

7.2. Межпроцессное взаимодействие IPC с использованием плоских файлов

Плоские файлы являются примитивным, но эффективным способом взаимодействия между процессами. Если необходимо организовать взаимодействие

между процессами, выполняющимися в разное время, то единственным выбором для нас станет использование такого файла. Примером здесь может стать компилятор C. Когда вы, к примеру, компилируете программу при помощи `gcc`, он генерирует ассемблерный файл, который передается в ассемблер. Поскольку промежуточный файл удаляется после завершения компоновки, пользователь обычно его не видит, однако его можно просмотреть, добавив с `gcc` параметр `-v`:

```
$ gcc -v -c hello.c
...
.../cc1 ... hello.c ... -o /tmp/ccPrPSPE.s   Компилятор генерирует
                                             временный файл.
...
as -V -Qy -o hello.o /tmp/ccPrPSPE.s         Ассемблер использует временный
                                             файл при выполнении компоновки.
```

Данный пример справедлив для компилятора C, так как он работает последовательно, то есть он должен завершить свою задачу прежде, чем запустится ассемблер. Таким образом, несмотря на то что это разные процессы, они выполняются не одновременно.

7.2.1. Блокировка файлов

Существует два вида блокировок: рекомендательные и обязательные. *Рекомендательные блокировки* задействуются, когда процесс вызывает `lockf` для блокировки файла, перед тем как он будет подвергнут чтению или записи. Если процесс не выполняет вызова `lockf`, блокировка игнорируется. *Обязательные блокировки* решают эту проблему следующим образом: когда какой-либо процесс пытается получить доступ к блокированному файлу, он сам подвергается блокировке при совершении вызова `read` или `write`. Поскольку блокировка осуществляется ядром, вам нет необходимости беспокоиться о некооперирующихся процессах, игнорирующих ваши рекомендательные блокировки. Для использования обязательных блокировок GNU/Linux требует, чтобы при монтировании файловой системы применялся флаг `mand`, а у создаваемого файла должен отсутствовать бит на исполнение группой, но должен быть установлен бит `setgid`. Если какое-то из этих условий не выполняется, обязательные блокировки не применяются.

7.2.2. Недостатки подхода, основанного на использовании файлов для обеспечения межпроцессного взаимодействия IPC

Использование файлов для обеспечения межпроцессного взаимодействия IPC имеет ряд недостатков. Применение файлов означает, что вы, скорее всего, столкнетесь с увеличением латентности (задержек) выполнения поставленной задачи вследствие необходимости доступа к дисковому накопителю. Кэш файловой системы позволяет в некоторой степени избежать данной проблемы, однако вы можете столкнуться с ней в самый неподходящий момент.

Другой недостаток применения файлов для обеспечения межпроцессного взаимодействия IPC заключается в угрозе безопасности. Размещение незашифрованных данных в файлах делает их уязвимыми перед пытливым взглядом случайных пользователей. Если среди таких данных будет важная информация, то размещать ее в файлах в незашифрованном виде не рекомендуется. Однако в случае с задачами, не имеющими критического значения, использование файлов для реализации IPC-взаимодействия может стать довольно неплохим решением.

7.3. Разделяемая память

Как вы уже знаете, процессы не могут просто разрешать другим процессам вторгаться в свою память и осуществлять в ней операции чтения/записи из-за механизма защиты памяти, реализованного в операционной системе Linux. Поскольку указателем на область памяти процесса служит виртуальный адрес, нет необходимости обращаться к физической локации в памяти. Передача данного адреса другому процессу может привести лишь к его краху. Виртуальный адрес имеет смысл только для того процесса, который его создал.

Linux и все UNIX-системы дают возможность позволять процессам совместно использовать память при помощи средств управления *разделяемой памятью*. Для разделения памяти между процессами существует два основных API-интерфейса: System V и POSIX. Они оба используют одинаковые принципы, но разные функции. Основная идея заключается в том, что любая память, которая будет совместно использоваться процессами, должна быть явным образом распределена таковой. Это означает, что нельзя просто взять переменную из стека или динамической памяти и «поделить» ее между разными процессами. Для того чтобы процессы могли совместно использовать память, необходимо распределить ее как разделяемую при помощи специальных функций.

Упомянутые выше API-интерфейсы используют ключи или имена для создания или подключения к областям разделяемой памяти. Процессы, которые будут совместно использовать память, должны «договориться» между собой о системе присваивания имен, для того чтобы иметь возможность отобразить в память соответствующие разделяемые области. API-интерфейс System V использует ключи, которые представляют собой определяемые приложением целочисленные значения. API-интерфейс POSIX использует символические имена, которые подчиняются тем же правилам, что и имена файлов.

7.3.1. Управление разделяемой памятью при помощи API-интерфейса POSIX

API-интерфейс POSIX является наиболее интуитивно понятным средством для управления разделяемой памятью из двух имеющихся. Его обзор можно увидеть в табл. 7.1. Функции `shm_open` и `shm_unlink` работают во многом аналогично системным вызовам `open` и `unlink`, используемым для обычных файлов. Они даже могут возвращать дескрипторы файлов, поддерживаемые обычными системными вызовами вроде `read` и `write`. Фактически строгой необходимости в использовании

функций shm_open и shm_unlink в Linux нет, однако, если вы создаете переносимые приложения, их следует применять вместо некоторых других инструментов.

Поскольку API-интерфейс POSIX основан на дескрипторах файлов, нам не потребуются какие-либо другие API-интерфейсы для выполнения дополнительных операций. Для этого мы можем воспользоваться любым системным вызовом, поддерживающим работу с файловыми дескрипторами. В листинге 7.1 приведен исчерпывающий пример того, как можно создать область разделяемой памяти, которая будет доступна для разных процессов.

Таблица 7.1. API-интерфейс POSIX, используемый для управления разделяемой памятью

Функция	Назначение
shm_open	Применяется для создания областей разделяемой памяти или подключения к уже существующим. Области определяются по имени, при этом данная функция возвращает дескриптор файла подобно системному вызову open
shm_unlink	Служит для удаления области разделяемой памяти с использованием дескриптора файла, возвращаемого функцией shm_open. Как и в случае с системным вызовом unlink, применяемым в отношении файлов, подобная область памяти не удаляется, пока все процессы не отсоединятся от нее. После вызова shm_unlink новые процессы уже не смогут подключаться к этой области
mmap	Отображает (загружает) в память процесса соответствующий файл. При вводе используется дескриптор файла, возвращаемый функцией shm_open. Данная функция возвращает указатель на новую область памяти, в которую были загружены данные. mmap поддерживает также использование файловых дескрипторов, которые принадлежат плоским файлам и некоторым другим устройствам
munmap	Отменяет отображение (загрузку) данных в память, которое было выполнено при помощи вызова mmap. Объем памяти, отображение в которую отменяется, может быть меньше или равен объему памяти, в которую были загружены данные посредством вызова mmap, при условии, что «отменяемая» область памяти удовлетворяет всем требованиям операционной системы, касающимся выравнивания и размера
msync	Синхронизирует доступ к области разделяемой памяти, в которую были отображены (загружены) данные при помощи вызова mmap, и записывает любые кэшированные данные в физическую память (или другое устройство), что позволяет другим процессам «видеть» произошедшие изменения

Листинг 7.1. posix-shm.c: пример создания области разделяемой памяти с использованием API-интерфейса POSIX

```
 1 /* posix-shm.c : gcc -o posix posix.c -lrt */
 2 #include <stdio.h>
 3 #include <string.h>
 4 #include <stdlib.h>
 5 #include <unistd.h>             // POSIX
 6 #include <sys/file.h>           // Включает open(2) с "сотоварищами".
 7 #include <sys/mman.h>          // Включает mmap(2) с "сотоварищами".
 8 #include <sys/wait.h>
 9
```

```
10 void error_out(const char *msg)
11 {
12     perror(msg);
13     exit(EXIT_FAILURE);
14 }
15
16 int main(int argc, char *argv[])
17 {
18     int r;
19
20     // shm_open рекомендует использовать начальный символ '/'
21     // в имени области для обеспечения переносимости, однако Linux
22     // этого не требует.
23     const char *memname = "/mymem";
24
25     // Использовать одну страницу для данного примера
26     const size_t region_size = sysconf(_SC_PAGE_SIZE);
27
28     // Создать область разделяемой памяти.
29     // Необходимо обратить внимание на то, что аргументы
       // идентичны тем, которые используются системным вызовом open(2).
30     int fd = shm_open(memname, O_CREAT | O_TRUNC | O_RDWR, 0666);
31     if (fd == -1)
32         error_out("shm_open");
33
34     // Распределить пространство в этой области. Мы используем
35     // ftruncate, однако вполне подойдет и write(2).
36     r = ftruncate(fd, region_size);
37     if (r != 0)
38         error_out("ftruncate");
39
40     // Отобразить область в память.
41     void *ptr =
42         mmap(0, region_size, PROT_READ | PROT_WRITE, MAP_SHARED, fd,
43              0);
44     if (ptr == MAP_FAILED)
45         error_out("mmap");
46
47     // fd после вызова mmmap нам не потребуется.
48     close(fd);
49
50     pid_t pid = fork();
51
52     if (pid == 0) {
53         // Дочерний процесс наследует отображение разделяемой памяти.
54         u_long *d = (u_long *) ptr;
55         *d = 0xdeadbeef;
56         exit(0);
57     }
58     else {
59         // Синхронизировать с дочерним процессом.
```

```
60          int status;
61          waitpid(pid, &status, 0);
62
63          // Родительскому процессу доступна аналогичная область памяти.
64          printf("дочерний процесс осуществил запись %#lx\n", *(u_long *) ptr);
65      }
66
67      // Разобравшись с областью памяти, отменить отображение данных в нее.
68      r = munmap(ptr, region_size);
69      if (r != 0)
70          error_out("munmap");
71
72      // Удалить область разделяемой памяти.
73      r = shm_unlink(memname);
74      if (r != 0)
75          error_out("shm_unlink");
76
77      return 0;
78  }
```

shm_open создает область разделяемой памяти точно так же, как вы создаете файл. Создаваемая область, подобно файлу, пуста. Распределение пространства во вновь созданной области разделяемой памяти аналогично заполнению файла данными. Для осуществления записи можно воспользоваться системным вызовом write, однако в случае с разделяемой памятью оптимальным инструментом зачастую оказывается системный вызов ftruncate. В заключение нужно отметить, что области разделяемой памяти продолжают существовать подобно файлам, то есть они не исчезают после завершения процесса, а должны быть явным образом удалены.

В листинге 7.1 создание равноценного процесса осуществляется посредством вызова fork. Несмотря на то что это может показаться своего рода хитростью, здесь иллюстрируется важный момент. Дочерние процессы наследуют отображения разделяемой памяти. Поэтому, в отличие от отображений стека или динамической памяти, которые клонируются при помощи методики копирования при записи, отображение разделяемой памяти будет указывать на одну и ту же физическую область в случае как с родительским, так и с дочерним процессами:

```
$ gcc -o posix-shm posix-shm.c -lrt
$ ./posix-shm
$ child wrote 0xdeadbeef
```

Память можно разделять между процессами, существование которых не строится по схеме «родительский процесс — дочерний процесс», то есть между равноценными процессами. Равноценный процесс, которому необходимо подключиться к области разделяемой памяти, по сути, будет использовать код, аналогичный приведенному в листинге 7.1. Единственное отличие заключается в том, что такой процесс не требует создания или очистки области памяти, поэтому флаг O_CREAT нужно будет удалить из системного вызова shm_open, так же как и вызов ftruncate.

Процессы должны применять соответствующие меры, для того чтобы обеспечивать должную синхронизацию во избежание возникновения состояний гонки. Как уже отмечалось, вызов wait в листинге 7.1 синхронизирует родительский и до-

черний процессы. В противном случае возникнет состояние гонки. Значение, выводимое дочерним процессом, зависит от того, какой процесс выполняется первым: родительский или дочерний. Вставкой вызова `waitpid` мы задаем завершение дочернего процесса первым, что гарантирует, что наше значение будет соответствующим. Подобная методика годится для данного примера, однако позднее вы узнаете, как обеспечивать более сложную синхронизацию.

Внутренне устройство разделяемой памяти, управляемой посредством API-интерфейса POSIX, в Linux

Реализация совместно используемой памяти в операционной системе Linux зависит от файловой системы разделяемой памяти, которая обычно монтируется в `/dev/shm`. В случае отсутствия данной точки монтирования выполнение `shm_open` потерпит неудачу. Поддерживаются файловые системы любых типов, однако в большинстве дистрибутивов по умолчанию монтируется файловая система `tmpfs`. Даже если `tmpfs` не смонтирована в `/dev/shm`, `shm_open` будет использовать файловую систему на основном дисковом накопителе. Вы можете и не заметить, что это происходит, поскольку, в зависимости от размера областей памяти, они могут бóльшую часть времени располагаться в кэше файловой системы. Как отмечалось ранее, `tmpfs` представляет собой файловую систему, у которой имеется кэш, но отсутствует дисковый накопитель.

Поскольку `shm_open` создает файлы в `/dev/shm`, каждая область разделяемой памяти будет иметь вид файла в данном каталоге. Имя файла будет аналогично тому, которое использовалось процессом, создавшим соответствующую область памяти. Это весьма полезное средство отладки, так как все инструменты, применяемые для отладки файлов, применимы также для отладки разделяемой памяти.

7.3.2. Управление разделяемой памятью при помощи API-интерфейса System V

API-интерфейс System V широко используется системой X Window, а также многими ее приложениями. Для большинства других приложений предпочтительным будет API-интерфейс POSIX. Обзор API-интерфейса System V приведен в табл. 7.2. Данный API-интерфейс применяется также при использовании семафоров и очередей сообщений, о которых мы поговорим в дальнейшем в этой главе.

Таблица 7.2. API-интерфейс System V, используемый для управления разделяемой памятью

Функция	Назначение
shmget	Применяется для создания областей разделяемой памяти или подключения к уже существующим (аналог shm_open)
shmat	Возвращает указатель на область разделяемой памяти (аналог mmap)
shmdt	Отменяет отображение (загрузку) данных в область разделяемой памяти, которое было выполнено с использованием shmat (аналог munmap)
shmctl	Имеет многоцелевое применение, включая отсоединение (удаление) областей разделяемой памяти, создаваемых при помощи shmget (аналог shm_unlink)

Листинг 7.2 представляет собой эквивалент листинга 7.1, однако на этот раз в нем используется API-интерфейс System V. Этапы остаются прежними, за тем исключением, что функция shmget одновременно отвечает за создание и распределение области разделяемой памяти, поэтому вызов ftruncate не применяется.

Листинг 7.2. sysv-shm.c: пример создания области разделяемой памяти с использованием API-интерфейса System V

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <stdlib.h>
 4 #include <unistd.h>
 5 #include <sys/ipc.h>
 6 #include <sys/shm.h>
 7 #include <sys/wait.h>
 8
 9 void error_out(const char *msg)
10 {
11     perror(msg);
12     exit(EXIT_FAILURE);
13 }
14
15 int main(int argc, char *argv[])
16 {
17
18     // Определяемый приложением ключ, аналог файлового имени,
       // используемого в случае с shm_open()
19     key_t mykey = 12345678;
20
21     // Использовать одну страницу для данного примера
22     const size_t region_size = sysconf(_SC_PAGE_SIZE);
23
24     // Создать область разделяемой памяти.
25     int smid = shmget(mykey, region_size, IPC_CREAT | 0666);
26     if (smid == -1)
27         error_out("shmget");
28
29     // Отобразить область в память.
30     void *ptr;
31     ptr = shmat(smid, NULL, 0);
32     if (ptr == (void *) -1)
33         error_out("shmat");
34
35     pid_t pid = fork();
36
37     if (pid == 0) {
38         // Дочерний процесс наследует отображение разделяемой памяти.
39         u_long *d = (u_long *) ptr;
40         *d = 0xdeadbeef;
41         exit(0);
42     }
```

```
43     else {
44         // Синхронизировать с дочерним процессом.
45         int status;
46         waitpid(pid, &status, 0);
47
48         // Родительскому процессу доступна аналогичная область памяти.
49         printf("дочерний процесс осуществил запись %#lx\n", *(u_long *) ptr);
50     }
51
52     // Разобравшись с областью памяти, отменить отображение данных в нее.
53     int r = shmdt(ptr);
54     if (r == -1)
55         error_out("shmdt");
56
57     // Удалить область разделяемой памяти.
58     r = shmctl(smid, IPC_RMID, NULL);
59     if (r == -1)
60         error_out("shmdt");
61
62     return 0;
63 }
```

Ключ, используемый `shmget`, функционально аналогичен файловому имени, используемому `shm_open`. Идентификатор `shmid`, возвращаемый `shmget`, функционально соответствует дескриптору файла, возвращаемому `shm_open`. В каждом случае первый определяется приложением, а второй — операционной системой. Поскольку листинг 7.2 почти идентичен листингу 7.1, итоговый вывод также будет похожим:

```
$ gcc -o sysv-shm sysv-shm.c
$ ./sysv-shm
child wrote 0xdeadbeef
```

В отличие от области памяти, созданной при помощи API-интерфейса POSIX, область памяти, сформированная посредством API-интерфейса System V, не будет «видна» в какой-либо файловой системе. Команда `ipcs` специально предназначена для манипулирования объектами разделяемой памяти System V. Об этом инструменте мы поговорим немного позднее в этой главе.

7.4. Сигналы

В своей книге[1] Арнольд Роббинс отрицательно высказался об использовании сигналов для обеспечения межпроцессного взаимодействия IPC, и я не стану оспаривать эту точку зрения. Проблема с сигналами заключается в том, что их обработчики не могут свободно осуществлять вызов всех доступных стандартных

[1] «Основы программирования Linux на практике» (Linux Programming by Example: The Fundamentals).

библиотечных функций. Время поступления сигнала нельзя заранее предсказать, а когда он все-таки поступает, операционная система прерывает текущий процесс с целью обработки сигнала независимо от того, какую задачу выполняет этот процесс. Это означает, что при запуске обработчика сигналов состояние библиотек окажется неизвестным. Так, например, ваш процесс может осуществлять вызов `malloc` или `printf`, и вдруг он прерывается обработчиком сигналов. Когда такое происходит, глобальные или статические переменные могут оказаться в рассогласованном состоянии. Если обработчик сигналов осуществляет вызов библиотечной функции, он может неправильно воспользоваться этими величинами и вызвать крах процесса. Вызов некоторых функций из обработчика сигналов может быть небезопасным. Стандарт POSIX определяет, какие функции безопасны для вызова подобным образом. Их можно отыскать на странице руководства man `signal(2)`.

Дополнительные трудности создает то, что функции, вызываемые из обработчика, на самом деле «не знают», что их вызов осуществляется подобным образом. Из этого следует, что функция, которая не *безопасна для вызова из обработчика сигналов*, не может безопасно завершиться, то есть возвратить состояние ошибки при вызове из обработчика сигналов. Вместо этого такие функции ведут себя непредсказуемо. Ваша программа может потерпеть крах, выдать бессодержательный вывод («мусор») или же функционировать абсолютно нормально. Это налагает ряд ограничений на операции, которые обработчик сигналов может безопасно выполнять.

7.4.1. Отправка сигналов процессу

Сигналы использовались с момента создания системы UNIX, и, несмотря на то что данный API-интерфейс эволюционировал, механизм реализации сигналов остался простым. Когда процесс получает сигнал, происходит установка внутреннего флага, который служит индикатором того, что данный сигнал принят. Когда ядро занимается планированием процесса, оно, вместо того чтобы возобновлять процесс с того места, где он был прерван, вызывает обработчик сигналов. После вызова обработчика сигналов флаг удаляется.

Процесс посылает сигнал другому процессу посредством системного вызова `kill`, который принимает в качестве аргументов идентификатор процесса и номер сигнала:

```
int kill( pid_t pid, int signal );
```

Каждый сигнал представлен в виде уникального целочисленного значения, из-за чего единовременно системный вызов `kill` может отправлять только один сигнал. Значение сигнала, равное `0`, означает его отсутствие, что является весьма полезным приемом для проведения проверки на предмет существования процесса, как показано в следующем примере:

```
int r = kill(pid,0);          Использование сигнала со значением 0
                              не оказывает никакого эффекта на процесс.
if ( r == 0 )
       /* process exists! */
              (/* процесс существует! */)
```

```
else if ( errno == ESRCH )     Перед тем как делать какие-либо заключения,
                               необходимо проверить значение переменной
                               errno.
    /* процесс не существует */
```

Системный вызов `kill`, подобно POSIX-функциям, возвращает `-1` в случае неудачи, при этом переменной `errno` присваивается соответствующее значение. В приведенном примере возвращаемое им значение `0` просто означает, что процесс существовал на момент отправки сигнала. Поскольку значение сигнала равно `0`, никаких сигналов отправлено не было, то есть это лишь свидетельствует о том, что процесс существовал на момент проверки. Если системный вызов `kill` возвращает значение `-1`, а переменной `errno` присваивается значение `ESRCH`, это говорит о том, что процесса с указанным идентификатором не существует.

7.4.2. Обработка сигналов

Для обработки сигналов существует два основных API-интерфейса: POSIX и System V. API-интерфейс POSIX был унаследован из системы BSD и является предпочтительным инструментом, поскольку он более гибок и устойчив к возникновению состояний гонки. API-интерфейс System V был выбран ANSI для стандартной библиотеки C. Он более прост, но менее гибок по сравнению с API-интерфейсом POSIX.

API-интерфейсы POSIX и System V позволяют пользователю определять одно из следующих поведений для любого сигнала:

- вызывать определяемую пользователем функцию;
- игнорировать сигнал;
- вернуться к поведению сигнала по умолчанию.

API-интерфейс POSIX также дает возможность блокировать и разблокировать отдельные сигналы без смены основного обработчика сигналов. Это ключ к предотвращению состояний гонки. Необходимо отметить, что сигналы `SIGKILL` и `SIGSTOP` не могут подвергаться обработке, блокировке или игнорированию, поскольку они жизненно важны для управления процессами.

В табл. 7.3 представлены шаблоны, посредством которых можно определять любое из поведений для сигналов, о которых мы говорили ранее.

Таблица 7.3. Определение поведения для сигналов

Действие	Шаблон System V	Шаблон POSIX
Вызвать определяемую пользователем функцию	old = signal(signo,handler)	sigaction(signo,new,old)
Игнорировать сигнал	old = signal(signo,SIG_IGN)	sigaction(signo,new,old)
Восстановить обработчик по умолчанию (вернуться к нему)	old = signal(signo,SIG_DFL)	sigaction(signo,new,old)

7.4.3. Маска сигналов и обработка сигналов

Маска сигналов используется ядром для определения того, каким образом сигналы доставляются процессу. По сути, это очень длинное слово, где на каждый сигнал приходится один бит. Если процесс устанавливает маску для конкретного сигнала, данный сигнал не будет доставляться процессу. Если сигнал обладает маской, мы также говорим, что он *блокирован*.

В соответствии с POSIX для управления маской сигналов используется `sigset_t`. Для обеспечения портативности никогда не стоит модифицировать `sigset_t` напрямую, а нужно пользоваться следующими функциями:

```
int sigemptyset(sigset_t *set);                 Удаляет все сигналы из маски
int sigfillset(sigset_t *set);                  Определяет все сигналы в маске
int sigaddset(sigset_t *set, int signum);       Определяет один сигнал в маске
int sigdelset(sigset_t *set, int signum);       Удаляет один сигнал из маски
int sigismember(sigset_t *set, int signum);     Возвращает значение 1,
                                                если в маске определено signum;
                                                в противном случае возвращается
                                                значение 0
```

`sigaddset`, `sigdelset` и `sigismember` принимают в качестве аргумента номер одного сигнала. Таким образом, каждая функция влияет только на один сигнал в маске. Процесс должен вызывать одну из этих функций для каждого сигнала, который необходимо модифицировать. Следует отметить, что данные функции воздействуют только на аргумент `sigset_t` и не влияют на обработку сигналов.

Когда процесс завершает модификацию маски, он передает ее функции `sigprocmask` для внесения изменений в маску сигналов процесса. Это позволяет приложению одновременно воздействовать на все сигналы, используя при этом единственный системный вызов и избегая возникновения состояний гонки. Прототип `sigprocmask` выглядит следующим образом:

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

Функция принимает два указателя на `sigset_t`. Первый представляет собой новую маску, которая будет применена, а второй — старую маску, которая будет использоваться для последующего восстановления маски сигналов в исходном состоянии. Аргумент `how` указывает на то, каким образом входная маска сигналов должна применяться к маске сигналов процесса. Данное значение может быть одним из следующих:

- `SIG_BLOCK` — указывает, что сигналы из входной маски сигналов должны быть добавлены в маску сигналов процесса. Сигналы, содержащиеся во входной маске сигналов, будут заблокированы в дополнение к текущим блокированным сигналам;
- `SIG_UNBLOCK` — указывает, что сигналы, содержащиеся во входной маске сигналов, должны быть удалены из маски сигналов процесса. Сигналы из входной маски сигналов не будут блокироваться, однако оставшаяся часть маски сигналов процесса останется неизменной;

- SIG_SET — указывает, что текущая маска сигналов должна быть перезаписана значениями входной маски сигналов. Блокироваться будут только сигналы, содержащиеся во входной маске сигналов. Все прочие сигналы не будут подвергаться блокировке, а текущая маска сигналов будет отброшена.

Следует отметить, что функция sigprocmask не принимает в качестве аргумента обработчик сигналов. Блокировка сигнала лишь откладывает доставку сигнала, она не отбрасывает сигналы, отправленные процессу. Пример можно увидеть в листинге 7.3.

Листинг 7.3. sigprocmask.c: использование функции sigprocmask для отсрочки доставки сигнала

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <stdlib.h>
 4 #include <signal.h>
 5 #include <unistd.h>
 6
 7 volatile int done = 0;
 8
 9 // Обработчик сигналов
10 void handler(int sig)
11 {
12     // Ref signal(2) — функция write() безопасна, а printf() — нет.
13     const char *str = "handled...\n";
14     write(1, str, strlen(str));
15     done = 1;
16 }
17
18 void child(void)
19 {
20     // Дочерний процесс завершается незамедлительно и посылает
       // родительскому процессу сигнал SIGCHLD
21     printf("дочерний процесс завершается\n");
22     exit(0);
23 }
24
25 int main(int argc, char *argv[])
26 {
27     // Осуществить обработку сигнала SIGCHLD, когда дочерний процесс
       // завершится.
28     signal(SIGCHLD, handler);
29     sigset_t newset, oldset;
30
31     // Определить все сигналы в наборе
32     sigfillset(&newset);
33
34     // Блокировать все сигналы и сохранить старую маску сигналов,
35     // чтобы ее можно было восстановить позднее
36     sigprocmask(SIG_BLOCK, &newset, &oldset);
37
```

```
38     // Породить дочерний процесс
39     pid_t pid = fork();
40     if (pid == 0)
41     child();
42
43     printf("родительский процесс "засыпает"\n");
44
45     // Погрузиться в "спячку" с блокировкой всех сигналов.
46     int r = sleep(3);
47
48     // r == 0 указывает на то, что "спячка" длилась полный период.
49     printf("пробуждение! r=%d\n", r);
50
51     // Восстановить старую маску сигналов,
52     // что приведет к вызову нашего обработчика сигналов.
53     sigprocmask(SIG_SETMASK, &oldset, NULL);
54
55     // Дождаться запуска обработчика сигналов.
56     while (!done) {
57     };
58
59     printf("завершение\n");
60     exit(0);
61 }
```

Программа из листинга 7.3 порождает дочерний процесс, который сразу же завершается. В результате этого родительский процесс получает сигнал SIGCHLD. Однако, перед тем как будет порожден дочерний процесс, необходимо определить обработчик сигналов для SIGCHLD и блокировать SIGCHLD на 3 с. Незамедлительно по возвращении из «спячки» нужно восстановить исходный обработчик сигналов, который снимет блокировку с сигнала SIGCHLD. После этого запускается обработчик сигналов, демонстрирующий, что данный сигнал доставлен:

```
$ ./sigprocmask
child exiting                    При этом родительскому процессу
                                 посылается сигнал SIGCHLD.
parent sleeping                  Сигнал SIGCHLD блокируется,
                                 и родительский процесс отправляется
                                 в "спячку".
woke up! r=0                     Возвращаемое значение 0 говорит о том,
                                 что "спячка" длилась 3 с; сигнал
                                 по-прежнему отсутствует.
handled...                       Сигнал доставляется после восстановления
                                 маски сигналов.
exiting
```

API-интерфейс POSIX включает две дополнительные функции для оперирования масками сигналов во время блокировки сигналов. Их прототипы имеют следующий вид:

```
int sigpending(sigset_t *set);
int sigsuspend(const sigset_t *mask);
```

sigpending дает возможность просматривать сигналы, которые были отправлены, но не доставлены. В листинге 7.3 маску сигналов можно было бы протестировать перед погружением в «спячку» и, возможно, увидеть сигнал. Однако гарантий, что это произойдет, нет никаких, поскольку функция sigpending не дожидается сигналов, а просто делает моментальный снимок сигналов и представляет их вызывающему оператору. Данная методика называется также *опросом*. Если вам необходимо дожидаться определенного набора сигналов и больше никаких других, функция sigpending — это то, что вам нужно. Она временно заменяет маску сигналов входной маской сигналов, пока не будет доставлен необходимый сигнал. Перед возвратом соответствующего значения она восстанавливает исходное состояние сигнальной маски.

7.4.4. Сигналы реального времени

Проницательные читатели могли заметить, что в результате блокировки возникает ряд специфических моментов. Что, например, произойдет, если во время блокировки сигналов будет отправлено более одного сигнала? Если во время блокировки сигналов поступает одиночный сигнал, он сохраняется в ядре, то есть не утрачивается и доставляется сразу же, как только процесс снимет маску с сигнала. Если аналогичный сигнал дважды отправляется во время своей блокировки, второй сигнал обычно отбрасывается. Если у сигнала отсутствует маска, вызов обработчика сигналов осуществляется только один раз.

POSIX ввел в употребление сигналы реального времени, для того чтобы обеспечить приложениям возможность многократно принимать сигнал, даже если он блокируется. Когда сигнал реального времени подвергается блокировке, ядро отслеживает количество раз, которое он принимается, и вызывает обработчик сигналов то же количество раз, когда сигнал оказывается разблокированным.

Сигналы реального времени идентифицируются по диапазону номеров сигналов. Этот диапазон определяется макросами SIGRTMIN и SIGRTMAX. Если указать номер сигнала в промежутке между двумя этими значениями, то данный сигнал будет поставлен в очередь. Пример можно увидеть в листинге 7.4.

Листинг 7.4. rt-sig.c: Блокировка и сигналы реального времени

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <stdlib.h>
 4 #include <signal.h>
 5 #include <unistd.h>
 6
 7 volatile int done = 0;
 8
 9 // Обработчик сигналов
10 void handler(int sig)
11 {
12     // Ref signal(2) — функция write() безопасна, а printf() — нет.
13     const char *str = "handled...\n";
14     write(1, str, strlen(str));
15     done = 1;
```

```
16 }
17
18 void child(void)
19 {
20     int i;
21     for (i = 0; i < 3; i++) {
22         // Послать родительскому процессу "высокоскоростные" сигналы.
23         kill(getppid(), SIGRTMIN);
24         printf("дочерний процесс — BANG!\n");
25     }
26     exit(0);
27 }
28
29 int main(int argc, char *argv[])
30 {
31     // Осуществить обработку сигнала SIGRTMIN, отправленного дочерним процессом.
32     signal(SIGRTMIN, handler);
33     sigset_t newset, oldset;
34
35     // Блокировать все сигналы и сохранить старую маску сигналов,
36     // для того чтобы ее можно было восстановить позднее
37     sigfillset(&newset);
38     sigprocmask(SIG_BLOCK, &newset, &oldset);
39
40     // Породить дочерний процесс
41     pid_t pid = fork();
42     if (pid == 0)
43         child();
44
45     printf("родительский процесс "засыпает"\n");
46
47     // Погрузиться в "спячку" с блокировкой всех сигналов.
48     int r = sleep(3);
49
50     // r == 0 указывает на то, что "спячка" длилась полный период.
51     printf("пробуждение! r=%d\n", r);
52
53     // Восстановить старую маску сигналов,
54     // что приведет к вызову нашего обработчика сигналов.
55     sigprocmask(SIG_SETMASK, &oldset, NULL);
56
57     // Дождаться запуска обработчика сигналов.
58     while (!done) {
59     };
60
61     printf("завершение\n");
62     exit(0);
63 }
```

Листинг 7.4 весьма схож с листингом 7.3, за исключением того, что в качестве сигнала в нем используется `SIGRTMIN`, а не `SIGCHLD`, а дочерний процесс трижды от-

правляет этот сигнал родительскому процессу, перед тем как завершается. Для использования размещаемых в очереди сигналов не требуется каких-либо специальных флагов. Номер сигнала указывает ядру использовать поставленные в очередь сигналы. После выполнения данного примера можно увидеть, как все происходит:

```
$ ./rt-sig
child – BANG!
child – BANG!
child – BANG!
parent sleeping
woke up! r=0
handled...
handled...
handled...
exiting
```

Здесь видно, что родительский процесс не прерывается сигналами даже после восстановления маски сигналов. Когда маска сигналов восстановлена, обработчик вызывается трижды — по одному разу при каждом совершении дочерним процессом вызова `kill`.

7.4.5. Расширенные сигналы с использованием функций sigqueue и sigaction

Мы наконец подошли к инструменту, альтернативному системному вызову `kill`, который называется `sigqueue`. Прототип данной функции приведен далее, при этом он довольно сильно напоминает системный вызов `kill`, за тем исключением, что принимает дополнительный аргумент:

```
int sigqueue(pid_t pid, int sig, const union sigval value);
```

Аргументы `pid` и `sig` идентичны системному вызову `kill`, однако аргумент `value` позволяет включать данные наряду с сигналом. Как вы уже знаете, сигналы, не относящиеся к сигналам реального времени, не ставятся в очередь, поэтому, только когда вы будете использовать значение номера сигнала в промежутке между `RTSIGMIN` и `RTSIGMAX`, соответствующие сигналы будут помещаться в очередь вместе с их ассоциированными данными.

Дополнительная информация, наличие которой обеспечивается посредством функции `sigqueue`, может извлекаться только обработчиком, определенным при помощи функции `sigaction`, которая уже упоминалась ранее как инструмент, альтернативный системному вызову `signal`. Определение обработчика с использованием `sigaction` является немного более сложной процедурой. Если взять за основу листинг 7.4, то функция `signal` может быть заменена функцией `sigaction` следующим образом:

```
struct sigaction sa = {              Использование синтаксиса
                                     инициализатора структуры C99…
     .sa_handler = handler,          Обработчик остался прежним
     .sa_flags = SA_RESTART         "Перевооружение" обработчика сигналов
                                     после его вызова…
};
```

```
sigemptyset(&sa.sa_mask);          Создание пустой маски сигналов
sigaction(SIGRTMIN,&sa,NULL);      Отмена старого действия посредством
                                   NULL
```

Заполнение структуры sigaction является дополнительным этапом, необходимым для использования функции sigaction. Поле sa_mask — это маска сигналов, которая используется во время работы обработчика сигналов. Обычно при вызове обработчика сигналов обрабатываемый сигнал блокируется. Приведенная маска указывает дополнительные сигналы, которые будут блокироваться во время выполнения обработчика сигналов. Обработчик в структуре sigaction служит указателем на функцию sigaction. На самом деле это союз, содержащий указатели на обработчики двух разных типов. sa_handler указывает на обработчик сигналов стиля System V, который использовался в приведенном ранее примере. sa_sigaction указывает на обработчик нового стиля, прототип которого имеет следующий вид:

```
void handler(int sig, struct siginfo *si, void *ptr)
```

Для того чтобы извлечь максимальную выгоду от использования sigqueue, необходимо определить обработчик нового стиля. Для обозначения того, что такой обработчик используется, нужно установить флаг SA_SIGINFO в поле sa_flags, как показано далее:

```
struct sigaction sa = {                Использование инициализатора
                                       структуры C99…
    .sa_sigaction = handler,           Использование обработчика нового
                                       стиля
    .sa_flags = SA_RESTART|SA_SIGINFO  НЕ ЗАБУДЬТЕ УСТАНОВИТЬ ЭТОТ ФЛАГ!
};
sigemptyset(&sa.sa_mask);
```

Когда ядро встречает флаг SA_SIGINFO, оно помещает в стек разные аргументы для обработчика сигналов. Без этого обработчик сигналов будет вызываться с использованием неверных аргументов, что, скорее всего, приведет к краху вашего приложения с сигналом SIGSEGV.

Теперь у нас есть обработчик нового стиля и мы можем заменить kill функцией sigqueue следующим образом:

```
union sigval sv = {
       .sival_int = 42                 Подойдет любой определяемый
                                       пользователем номер.
};
sigqueue(getppid(), SIGRTMIN, sv);     Можно использовать любой сигнал,
                                       однако в очередь будут помещаться
                                       только сигналы реального времени.
```

Если мы снова посмотрим на обработчик, то увидим, что структура siginfo содержит разнообразные элементы, определяемые разными стандартами. Далее приведены набор значений, используемых в Linux, и их POSIX-определения:

```
int           si_signo          Номер сигнала — SIGINT и т. д.
int           si_code           Код сигнала (см. содержание)
```

```
int             si_errno        Если значение errno не является нулевым,
                                оно ассоциировано с данным сигналом
pid_t           si_pid          Идентификатор посылающего сигнал процесса
uid_t           si_uid          Реальный идентификатор
                                пользователя процесса, посылающего сигнал
void           *si_addr         (См. содержание)
int             si_status       Значение завершения или сигнал
long            si_band         Событие band для SIGPOLL
union sigval    si_value        Значение сигнала
```

Приведенные примечания говорят о том, что некоторые относящиеся к сигналам поля в структуре `siginfo` не определяются в любых ситуациях. Из всех определяемых полей только `si_signo`, `si_errno` и `si_code` всегда содержат соответствующие данные согласно POSIX. Если сигнал посылается другим процессом, в полях `si_pid` и `si_uid` будут отображаться идентификатор процесса и идентификатор пользователя процесса, который отправил сигнал.

`si_code` принимает одно из значений, приведенных в табл. 7.4, и служит индикатором подробностей об источнике сигнала и причине его отправки. Сигнал, который был послан другим процессом или из-за вызова функции `raise`, приводит к тому, что `si_code` принимает значение `SI_USER`. Результатом сигнала, посланного при участии функции `sigqueue`, является то, что `si_code` принимает значение `SI_QUEUE`.

Примером поля, чувствительного к контексту, является поле `si_value`, которое определяется только тогда, когда вызывающий оператор использовал для отправки сигнала функцию `sigqueue`. Когда такое происходит, `si_value` будет содержать копию `sigval`, отправленного этой функцией. Большинство других полей имеют отношение к различным исключительным ситуациям, которые никак не связаны с межпроцессным взаимодействием IPC и исходят из текущего процесса.

Еще одним чувствительным к контексту полем является `si_errno`, которое может содержать ненулевое значение, если произошедшая ошибка ассоциирована с данным сигналом. Поле `si_band` определяется только для `SIGPOLL`, что может оказаться полезным для некоторых IPC-приложений.

Таблица 7.4. Значения, принимаемые si_code

si_code	**Характеристика**
SI_USER	Сигнал отправлен посредством kill() или raise()
SI_KERNEL	Сигнал отправлен из ядра
SI_QUEUE	Сигнал отправлен посредством sigqueue()
SI_TIMER	Время в таймере POSIX истекло
SI_MESGQ	Состояние очереди сообщений POSIX изменилось
SI_ASYNCIO	Асинхронный ввод/вывод (AIO — Asynchronous I/O) завершен
SI_SIGIO	Сигнал SIGIO поставлен в очередь (не используется в Linux)
SI_TKILL	Сигнал отправлен посредством tkill() или tgkill(); используется только в Linux

7.5. Конвейеры (каналы)

Конвейеры (каналы) весьма просто создавать и использовать. Они бывают двух видов: неименованные каналы, используемые для связи между родительскими и дочерними процессами, и именованные каналы, служащие для связи между равноценными процессами, протекающими на одном и том же компьютере. Создать неименованный канал в процессе можно при помощи системного вызова `pipe`, который возвращает пару файловых дескрипторов. Прототип функции `pipe` имеет следующий вид:

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

Вызывающий оператор передает массив из двух целочисленных значений, который при успешном возврате будет содержать два файловых дескриптора. Первый дескриптор файла в массиве предназначен только для чтения, второй — только для записи. Неименованные каналы пригодны только для взаимодействия между родительским и дочерним процессами. Поскольку дочерний процесс наследует файловые дескрипторы родительского процесса, последний создает конвейер перед разветвлением для того, чтобы организовать канал взаимодействия между собой и дочерним процессом.

Системы Linux и UNIX поддерживают работу с именованными каналами, которые идентифицируются посредством особых файлов на диске, созданных при помощи функции `mkfifo` или `mknod`:

```
int mkfifo(const char *pathname, mode_t mode);
```

`mkfifo` и `mknod` доступны также в виде команд оболочки. Именованный канал подобно обычному файлу может подвергаться открытию, чтению и записи с использованием обычных системных вызовов `open`, `read` и `write`. Соответствующий файл, размещаемый на диске, используется только для присваивания имени; в файловую систему никакие данные не записываются.

Обычно, когда процесс открывает именованный канал для чтения, он блокируется до тех пор, пока другой процесс не откроет именованный канал для записи, и наоборот. Это делает именованные каналы пригодными для синхронизации с другими процессами.

7.6. Сокеты

Сокеты — это универсальные средства для обеспечения взаимодействия между процессами; они могут использоваться локально или через сеть. Сокеты функционируют во многом аналогично каналам, за тем исключением, что они являются двунаправленными. Если вы осуществляете распространение процессов посредством сети, сокеты позволяют использовать аналогичный API-интерфейс для организации взаимодействия между всеми процессами независимо от того, являются они местными или нет.

Всестороннее изучение сокетов выходит за рамки данной книги[1], однако мы все же рассмотрим ряд базовых примеров.

---

[1] Отличный материал можно найти на info-странице glibc: info libc sockets.

7.6.1. Создание сокетов

Системным вызовом для создания универсальных сокетов является функция `socket`, которая генерирует сокеты, используемые для локальных и сетевых соединений. Также существует функция `socketpair`, которая применяется исключительно для создания локальных сокетов. Далее приведены прототипы обеих этих функций:

```
int socket(int domain, int type, int protocol);
int socketpair(int domain, int type, int protocol, int fd[2]);
```

Данным функциям необходимо, чтобы пользователь указал *домен* (`domain`), *тип* (`type`) и *протокол* (`protocol`) для сокета, о которых мы будем говорить в дальнейшем. Функция `socketpair`, подобно системному вызову `pipe`, возвращает массив из двух файловых дескрипторов.

Домены сокетов

Параметр `domain`, указываемый для сокета, позволяет определить интерфейс, который сможет использовать данный сокет, то есть решить, будет это сетевой, локальный или какой-то другой интерфейс. *Домен* — это термин, используемый POSIX, однако GNU применяет вместо него термин *пространство имен*, для того чтобы не создавать дополнительную смысловую нагрузку на термин *домен*, который имеет много других значений. POSIX-константы, определяемые в качестве данного параметра, используют префикс `PF` (сокращенно от *protocol family — семейство протоколов*). В табл. 7.5 приведен неполный перечень таких констант. Вам, скорее всего, придется столкнуться с `PF_UNIX` и `PF_INET`. `PF_UNIX` используется для взаимодействия между процессами, выполняющимися на одном и том же компьютере, а `PF_INET` — для взаимодействия через IP-сеть[1].

Таблица 7.5. Домены, указываемые при создании сокетов с помощью функций

Домен	Протокол/использование
PF_UNSPEC	Неопределенный. Выбор в данном случае делает операционная система. Определяется в виде нулевого значения
PF_UNIX, PF_LOCAL	Обеспечивает взаимодействие между процессами, протекающими на одном и том же компьютере
PF_INET	Указывает на использование интернет-протоколов IPv4
PF_INET6	Указывает на использование интернет-протоколов IPv6
PF_NETLINK	Устройство пользовательского интерфейса ядра
PF_PACKET	Низкоуровневый пакетный интерфейс, обеспечивающий прямой доступ к устройству

Типы сокетов

Параметр `type` сокета — это второй аргумент, указываемый при создании сокетов с использованием соответствующих функций, который определяет тип службы,

[1] Точнее, через IPv4-сеть. IPv4 — это оригинальный IP-протокол, где на каждый адрес приходится 32 бита, в то время как в протоколе IPv6 для адресации используется уже 128 бит.

необходимой приложению. В табл. 7.6 приведено подробное описание существующих типов сокетов. Возможно, наиболее распространенным и простым в использовании является тип SOCK_STREAM. Данный тип требует наличия *соединения*, что, по сути, означает, что у выполняющегося процесса все концы сокета должны быть открыты для функционирования; в противном случае возникнет ошибка. Надежность соединения определяется третьим аргументом — protocol.

Таблица 7.6. Типы сокетов

Тип сокета	Описание
SOCK_STREAM	Обеспечивает «надежную» передачу данных при установлении соединений. Лежащий в основе протокол гарантирует, что считывание данных будет осуществляться в том же порядке, в котором они передаются. Протокол также может поддерживать передачу так называемых «срочных» (out of band) данных
SOCK_DGRAM	Обеспечивает «ненадежную» передачу данных без установления соединений, не гарантируя при этом обязательной доставки данных или соблюдения порядка, в котором они доставляются
SOCK_SEQPACKET	Аналогичен SOCK_STREAM, однако при этом требуется, чтобы читатель данных считывал их целыми пакетами
SOCK_RAW	Обеспечивает доступ к «сырым» сетевым пакетам
SOCK_RDM	Аналогичен SOCK_STREAM, за тем исключением, что не гарантирует упорядоченной доставки данных

Протоколы сокетов

Протоколы обладают разными возможностями, при этом не все из них поддерживают любые типы сокетов. Стандарт POSIX предусматривает набор протоколов, которые поддерживаются всеми реализациями сокетов (табл. 7.7). Макросы для данных протоколов определяются в <netinet/in.h>. Поддержка других протоколов также может обеспечиваться, однако они будут относиться к нестандартным. В любом случае список известных протоколов можно найти в /etc/protocols. Для выбора протокола в соответствии с именем, указанным в /etc/protocols, необходимо воспользоваться одной из библиотечных функций семейства getprotoent(3) и значением, возвращаемым для определения протокола.

Таблица 7.7. Протоколы сокетов

Макрос	Имя протокола	Описание
IPPROTO_IP	ip	Межсетевой протокол (Internet Protocol), формально не являющийся протоколом. Указанный макрос применяется для локальных сокетов любого типа, может сочетаться также с не использующими соединений сетевыми и локальными сокетами
IPPROTO_ICMP	icmp	Протокол управляющих сообщений (Internet Control Message Protocol), используемый приложениями вроде команды ping

Макрос	Имя протокола	Описание
IPPROTO_TCP	tcp	Протокол управления передачей (Transmission Control Protocol), который основан на соединениях и является надежным в использовании в сочетании с сокетами SOCK_STREAM
IPPROTO_UDP	udp	Протокол пользовательских дейтаграмм (User Datagram Protocol), который не использует соединений, является ненадежным и применяется для межпроцессного взаимодействия IPC, что обеспечивает низкую латентность за счет надежности, при этом данный протокол наиболее часто используется в сочетании с сокетами типа SOCK_DGRAM
IPPROTO_IPV6	ipv6	Как и протокол IP, формально не является протоколом, однако указывает на то, что пакеты должны использовать IPv6-адресацию
IPPROTO_RAW	raw	Дает возможность приложениям получать «сырые» пакеты; обычно недоступен для непривилегированных пользователей

Псевдопротокол `ip` имеет значение `0`, которое используется для всех локальных сокетов (`PF_LOCAL`). Вследствие этого многие программисты указывают для протокола значение `0`, когда им требуется локальный сокет.

7.6.2. Пример создания локального сокета при помощи функции socketpair

Наиболее простой способ создать локальный сокет заключается в использовании функции `socketpair`, что наглядно продемонстрировано в листинге 7.5.

Листинг 7.5. `socketpair.c`: создание локального сокета посредством функции `socketpair`

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <stdlib.h>
 4 #include <errno.h>
 5 #include <unistd.h>
 6 #include <sys/types.h>
 7 #include <sys/socket.h>
 8 #include <sys/wait.h>
 9 #include <netdb.h>
10
11 int main(int argc, char *argv[])
12 {
13     int fd[2];
14
15     // ref. socketpair(2)
16     // домен (также называемый семейством протоколов) = PF_LOCAL
       // (аналог PF_UNIX))
17     // тип = SOCK_STREAM (см. таблицу)
```

```
18      // протокол = нулевое значение (использовать протокол по умолчанию)
19      int r = socketpair(PF_LOCAL, SOCK_STREAM, 0, fd);
20      if (r == -1) {
21          perror("socketpair");
22      }
23
24      pid_t pid = fork();
25
26      if (pid == 0) {
27          // Дочерний процесс осуществляет чтение.
28          char buf[32];
29          int n = read(fd[1], buf, sizeof(buf));
30          if (n == -1) {
31              perror("read");
32          }
33          printf("прочитано %d байт '%s'\n", n, buf);
34      }
35      else {
36          // Родительский процесс осуществляет запись.
37          char msg[] = "Hello World";
38          int n = write(fd[0], msg, sizeof(msg));
39          if (n == -1) {
40              perror("write");
41          }
42
43          // Ожидать завершения работы дочернего процесса.
44          int status;
45          wait(&status);
46      }
47
48      exit(0);
49      return 0;
50  }
```

В листинге 7.5 наглядно продемонстрированы основы создания сокета при помощи функции `socketpair`. Как отмечалось ранее, сокеты весьма схожи с каналами (конвейерами), но они, в отличие от каналов, являются двунаправленными. Вы уже знаете, что вызов `pipe` возвращает два файловых дескриптора: один для записи, другой для чтения. Если вы хотите организовать двунаправленное взаимодействие между двумя процессами посредством каналов, для каждого направления необходимо создать отдельный канал. Поскольку сокет изначально является двунаправленным, для обеспечения полнодуплексного пути взаимодействия между двумя процессами вам потребуется только один сокет.

7.6.3. Пример пары «клиент/сервер», использующей локальные сокеты

Сокеты наиболее часто используются в приложениях типа «клиент/сервер», которые требуют применения более универсального системного вызова `socket`. В от-

личие от socketpair, который возвращает пару файловых дескрипторов, socket возвращает один такой дескриптор. Функция socketpair избавляет программистов от необходимости вдаваться в подробности, касающиеся сокетов, однако она может использоваться только тогда, когда речь идет о взаимодействии между родительским и дочерним процессами.

Перед тем как приступать к использованию функции socket, нам необходимо обратиться к ряду дополнительных функций. На рис. 7.1 представлена потоковая диаграмма базового клиента и сервера. Следует отметить, что функция socketpair обеспечивает удобный кратчайший путь для всех дополнительных API-вызовов, необходимых основному клиенту и серверу.

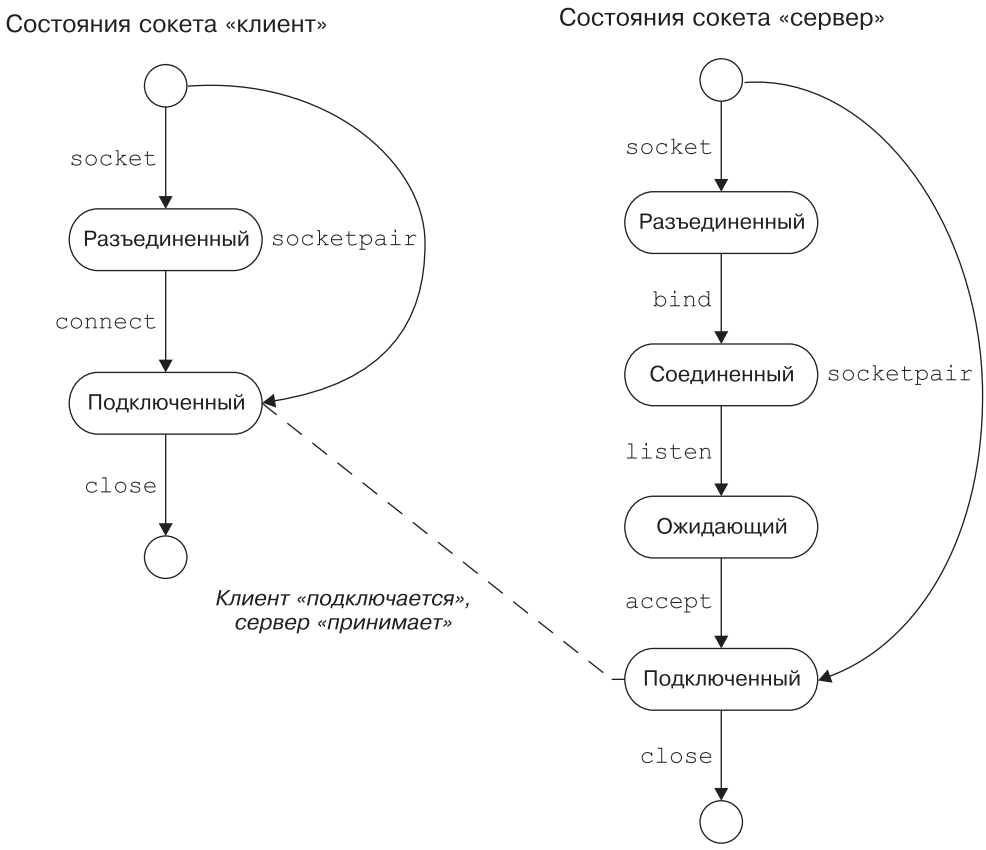


Рис. 7.1. API-интерфейс сокетов «клиент/сервер» вместе с ассоциированными состояниями сокетов

Поскольку данные функции снабжены исчерпывающими страницами руководства man в Linux, я не стану подробно рассматривать их здесь, а лучше приведу ряд практических примеров. Однако сначала я расскажу вам еще об одной функции, не упомянутой на рис. 7.1: select. Эта функция применяется в отношении разнообразных дескрипторов файлов с использованием дополнительного интервала

ожидания[1]. Она также используется в сочетании с многими другими дескрипторами файлов, а не только с сокетами, однако написание сокетных программ без нее является непростым занятием. Функция select определяется следующим образом:

```
int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
           struct timeval *timeout);
```

Посредством типа fd_set приложение указывает функции select, мониторинг каких файловых дескрипторов необходимо осуществлять. Каждым fd_set можно манипулировать при помощи группы макросов, приведенных далее:

```
FD_SET(int fd, fd_set *set)          Добавить fd в fd_set.
FD_CLR(int fd, fd_set *set)          Удалить fd из fd_set.
FD_ISSET(int fd, fd_set *set)        Произвести проверку на предмет
                                     наличия fd в fd_set.
FD_ZERO(fd_set *set)                 Удалить все fd из fd_set.
```

В примере вы увидите, как все эти функции работают сообща. В листинге 7.6 приведен простой сервер, использующий сокеты. Следует отметить, что дескриптор файла сокета, возвращаемый функцией socket, передается функции listen, которая преобразует его в так называемый сокет прослушивания. *Сокеты прослушивания* используются серверами для принятия соединений. Функция accept принимает сокет прослушивания в качестве ввода и ожидает установки соединения с клиентом. После этого она возвращает новый дескриптор файла, который будет сокетом, соединенным с клиентом.

Листинг 7.6. server_un.c: сервер, использующий локальный сокет

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <signal.h>
 4 #include <stdlib.h>
 5 #include <unistd.h>
 6 #include <sys/types.h>
 7 #include <sys/socket.h>
 8 #include <sys/un.h>
 9
10 // Вызвать perror и exit, если возвращаемое системным вызовом
   // значение равно -1
11 #define ASSERTNOERR(x,msg) do {\
12     if ((x) == -1) { perror(msg); exit(1); }} while(0)
13
14 // Локальным сокетам требуется именованный файл, который должен
   // быть отсоединен (удален) позднее
15 #define SOCKNAME "localsock"
16
17 int main(int argc, char *argv[])
18 {
19     // ref. socket(2)
```

[1] «Родственницей» select является функция poll.

```
20      // Создать локальный потоковый сокет
21      // домен (также называемый семейством протоколов) = PF_LOCAL
        // (аналог PF_UNIX)
22      // тип = SOCK_STREAM (см. таблицу)
23      // протокол = нулевое значение (использовать протокол по умолчанию)
24      int s = socket(PF_LOCAL, SOCK_STREAM, 0);
25
26      ASSERTNOERR(s, "socket");
27
28      // Локальным сокетам дается имя, которое располагается
        // в файловой системе.
29      struct sockaddr_un sa = {
30          .sun_family = AF_LOCAL,
31          .sun_path = SOCKNAME
32      };
33
34      // Создание файла.
35      // Если файл существует, возникает ошибка EADDRINUSE,
36      // поэтому не забудьте по завершении осуществить вызов unlink!
37      int r = bind(s, (struct sockaddr *) &sa, sizeof(sa));
38
39      ASSERTNOERR(r, "bind");
40
41      // Разрешить клиентам устанавливать соединения. Разрешить
        // размещение одного в backlog-очереди.
42      // Данный вызов не блокируется. Это происходит во время принятия.
43      r = listen(s, 0);
44
45      ASSERTNOERR(r, "listen");
46
47      // Мы используем структуру sockaddr_un для адреса сокета UNIX.
48      // Необходимо указать путь к файлу в смонтированной файловой системе.
49      struct sockaddr_un asa;
50      size_t addrlen = sizeof(asa);
51
52      // Осуществлять блокировку до установления соединения с клиентом.
        // Возвращается дескриптор файла
53      // нового соединения, а также его адрес.
54      int fd = accept(s, (struct sockaddr *) &asa, &addrlen);
55
56      ASSERTNOERR(fd, "accept");
57
58      while (1) {
59          char buf[32];
60          fd_set fds;
61
62          // Использовать select для ожидания данных от клиента.
63          FD_ZERO(&fds);
64          FD_SET(fd, &fds);
65          int r = select(fd + 1, &fds, NULL, NULL, NULL);
66
```

```
67          ASSERTNOERR(r, "select");
68
69          // Прочесть данные
70          int n = read(fd, buf, sizeof(buf));
71          printf("сервером прочитано %d байт\n", n);
72
73          // Нулевая продолжительность времени чтения означает, что
            // клиент закрыл сокет.
74          if (n == 0)
75              break;
76      }
77
78      // Достаточно использовать обычный системный вызов unlink.
79      unlink(SOCKNAME);
80
81      return 0;
82  }
```

В листинге 7.7 приведен пример простого клиента, используемого в сочетании с сервером из листинга 7.6. Здесь дескриптор файла, используемый функцией socket, передается функции connect, которая возвращает соответствующее значение после того, как сокет соединяется с сервером.

Листинг 7.7. client_un.c: простой клиент, использующий локальные сокеты

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <stdlib.h>
 4 #include <signal.h>
 5 #include <unistd.h>
 6 #include <sys/types.h>
 7 #include <sys/socket.h>
 8 #include <sys/un.h>
 9
10 #define ASSERTNOERR(x,msg) do {\
11     if ((x) == -1) { perror(msg); exit(1); }} while(0)
12
13 // Имя должно соответствовать серверу, с которым необходимо установить
   // соединение.
14 #define SOCKNAME "localsock"
15
16 int main(int argc, char *argv[])
17 {
18     // Параметры ust соответствуют server_un.c
19     int s = socket(PF_LOCAL, SOCK_STREAM, 0);
20
21     ASSERTNOERR(s, "socket");
22
23     // Адрес сокета использует AF_ macros
24     struct sockaddr_un sa = {
25         .sun_family = AF_LOCAL,
26         .sun_path = SOCKNAME
```

```
27     };
28
29     // В случае неудачи возвращается значение -1.
30     int r = connect(s, (struct sockaddr *) &sa, sizeof(sa));
31
32     ASSERTNOERR(r, "connect");
33
34     const char data[] = "Hello World";
35     r = write(s, data, sizeof(data));
36
37     printf("клиентом записано %d байт\n", r);
38
39     return 0;
40 }
```

При запуске сервер создает в локальном каталоге файл с именем `localsock`. Он необходим сокету домена UNIX и определяется в структуре `sockaddr_un`. Клиент и сервер оба определяют имя сокета, однако только сервер создает сокет. Локальный сокет можно просмотреть при помощи команды `ls`, а опознать его можно по символу `s` в первой колонке разрешений, как показано в примере:

```
$ ./server_un &
$ ls -l localsock
srwxrwxr-x  1 john john 0 Apr 29 22:28 localsock
```

Клиент, использующий локальные сокеты, довольно прост, в чем можно было убедиться, просматривая листинг 7.7. Запуск клиента (после сервера) приводит к генерированию следующего вывода:

```
$ ./client_un
client wrote 12 bytes
server read 12 bytes
server read 0 bytes
 [1]+  Done                    ./server_un
```

7.6.4. Пример пары «клиент/сервер», использующей сетевые сокеты

По счастливому стечению обстоятельств сетевые клиент и сервер не слишком отличаются от локальных клиента и сервера, которые мы создавали в листингах 7.7 и 7.6. В этом разделе вы узнаете, какие различия существуют между ними. По сути, разница заключается в используемом семействе протоколов и адресе. За исключением этого другие различия отсутствуют.

- Включаемый файл `sys/un.h` заменяется файлом `netinet/in.h`.
- Семейство протоколов, передаваемое функции `socket`, изменяется с `PF_LOCAL` на семейство сетевых протоколов (обычно это `PF_INET`).
- Структура адреса сокета изменяется с `sockaddr_un` на `sockaddr_in`. Она содержит семейство сетевых адресов (обычно это `AF_INET`) и инициализируется с использованием сетевого адреса и порта.

- По завершении вызов unlink в отношении сокета не осуществляется. Сетевой сокет не прекращает существования после завершения работы сервера.

Адрес сокета для клиента и сервера немного различается, поэтому его стоит рассмотреть более пристально. Поскольку это сетевой адрес, он должен определяться с применением адреса и порта с соответствующим номером. Сервер может разрешать установку соединений посредством любого интерфейса при помощи специального адресного макроса INADDR_ANY. При инициализации структуры необходимо также учитывать сетевой порядок байтов. Программный код для инициализации адреса порта адреса сервера со значением 5000 выглядит следующим образом:

```
struct sockaddr_in sa = {
    .sin_family = AF_INET,
    .sin_port=htons(5000),
    .sin_addr = {INADDR_ANY} };
```

Имя макроса htons сокращенно означает *host to network short (преобразование узлового порядка в коротком числе в сетевой порядок байтов)*; он выполняет преобразование порядка байтов в идентификаторе порта в сетевой порядок байтов[1]. Вместо INADDR_ANY можно указать специальный адрес интерфейса в поле sin_addr и использовать структуру in_addr, которая инициализируется при помощи команды, выполненной по следующему шаблону:

```
struct in_addr ifaddr;
int r = inet_aton("192.168.163.128",&ifaddr);
```

Затем можно воспользоваться значением ifaddr для инициализации sin_addr в struct sockaddr. Клиент подвергается аналогичным изменениям, за исключением того, что вместо INADDR_ANY он обычно использует специальный адрес.

7.7. Очереди сообщений

Очереди сообщений представляют собой еще один способ передачи данных между двумя процессами. Как вы уже могли догадаться, создавать очереди сообщений можно посредством двух API-интерфейсов: System V и POSIX. Они имеют небольшие различия, однако используют один и тот же принцип.

Оба типа очередей сообщений делают акцент на сообщениях, обладающих фиксированным размером и соответствующим приоритетом. Получатель должен иметь возможность осуществлять чтение именно того количества данных, которое передано, в противном случае чтение очереди сообщений будет неудачным. Это обеспечивает для приложения некоторую гарантию того, что процесс, с которым оно взаимодействует, использует ту же версию структуры сообщений.

API-интерфейсы реализуют приоритеты немного по-разному. API-интерфейс System V предоставляет получателю чуть бóльшую гибкость в присваивании приоритета входящим сообщениям, в то время как API-интерфейс POSIX навязывает строгий порядок расстановки приоритетов.

---

[1] См. inet(3).

7.7.1. Очередь сообщений System V

Создание очередей сообщений и подключение к ним осуществляется при помощи функции msgget, которая принимает целочисленное значение в качестве определяемого пользователем ключа для очереди сообщений, подобному тому как это происходит в случае с API-интерфейсом System V и разделяемой памятью. Данная функция возвращает определяемый системой ключ для очереди, который используется при последующих операциях чтения и записи в очередь. Прототип функции msgget выглядит следующим образом:

```
int msgget(key_t key, int msgflg);
```

Аргументом key может выступать определяемый приложением ключ, который всегда остается неизменным. Значение, возвращаемое функцией msgget, представляет собой определяемый системой идентификатор очереди сообщений, используемый процессом во всех процедурах чтения и записи. Этот ключ может заменяться макросом IPC_PRIVATE. Использование данного значения в качестве ключа всегда будет приводить к созданию новой очереди сообщений. Идентификаторы очередей сообщений, возвращаемые функцией msgget, в отличие от дескрипторов файлов, подходят для нескольких процессов. Иначе говоря, идентификатор очереди сообщений может совместно использоваться родительским и дочерним процессами, а также равноценными процессами.

Аргумент msgflag, принимаемый функцией msgget, очень схож с флагами, передаваемыми системному вызову open. Фактически нижние 9 бит аналогичны битам разрешений, используемым вызовом open. Среди прочих поддерживаются флаги IPC_CREAT и IPC_EXCL. Флаг IPC_CREAT применяется для создания новой очереди сообщений; если очередь сообщений уже существует, будет возвращен идентификатор существующей очереди сообщений. Когда используются сразу оба флага, IPC_CREAT и IPC_EXCL, выполнение функции msgget потерпит неудачу, если очередь сообщений уже существует. Данное поведение аналогично тому, которое можно наблюдать при использовании флагов IPC_CREAT и IPC_EXCL в сочетании с системным вызовом open(2).

Эквивалента функции close, которую можно было бы применять в отношении очередей сообщений, не существует, поскольку идентификаторы очередей не используют файловых дескрипторов. Идентификаторы «видны» для всех процессов, а операционная система не отслеживает очередей сообщений отдельно по каждому процессу.

Процесс может удалить очередь сообщений при помощи функции msgctl, которая, если исходить из ее имени, способна выполнять, кроме этой задачи, еще множество других:

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

Аргументом cmd, применяемым для удаления очереди сообщений, является IPC_RMID, аргументом buf в этом случае может быть NULL. Для получения дополнительных подробностей обращайтесь к msgctl(2).

Чтение сообщений из очереди и запись в нее осуществляются при помощи следующих функций:

```
int msgsnd(int qid, void *msg, size_t msgsz, int msgflg);
ssize_t msgrcv(int qid, void *msg, size_t msgsz, long typ, int msgflg);
```

Очереди сообщений System V могут содержать сообщения переменной длины при условии, что отправитель и получатель «сходятся во взглядах» на данный вопрос. *Тип* сообщений, как и их приоритет, бывает двух видов. Таким образом, в зависимости от приложения вы можете выбрать присваивание приоритетов сообщениям в очереди, использовать приоритет для идентификации типа сообщения либо комбинировать эти методики.

Пример применения данных методик приведен в листинге 7.8.

Листинг 7.8. sysv-msgq-example.c: пример работы с очередью сообщений с использованием API-интерфейса System V

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <stdlib.h>
 4 #include <unistd.h>
 5 #include <sys/wait.h>
 6 #include <sys/stat.h>
 7 #include <sys/file.h>
 8 #include <sys/msg.h>
 9 #include <sys/ipc.h>
10
11 // Для простоты установить фиксированный размер сообщений.
12 struct message {
13     long int mtype;
14     char mtext[128];
15 };
16
17 // Вставить текст в сообщение и отправить его.
18 int send_msg(int qid, int mtype, const char text[])
19 {
20     struct message msg = {
21         .mtype = mtype
22     };
23     strncpy(msg.mtext, text, sizeof(msg.mtext));
24
25     int r = msgsnd(qid, &msg, sizeof(msg), 0);
26     if (r == -1) {
27         perror("msgsnd");
28     }
29     return r;
30 }
31
32 // Осуществить чтение сообщения из очереди в структуру сообщений.
33 int recv_msg(int qid, int mtype, struct message *msg)
34 {
35     int r = msgrcv(qid, msg, sizeof(struct message), mtype, 0);
36     switch (r) {
37     case sizeof(struct message):
```

```
38          /* okay */
39          break;
40      case -1:
41          perror("msgrcv");
42          break;
43      default:
44          printf("получено только %d байт\n", r);
45      }
46      return r;
47  }
48
49  void producer(int mqid)
50  {
51      // Необходимо обратить внимание на порядок, в котором мы отправляем
        // эти сообщения.
52      send_msg(mqid, 1, "type 1 — first");
53      send_msg(mqid, 2, "type 2 — second");
54      send_msg(mqid, 1, "type 1 — third");
55  }
56
57  void consumer(int qid)
58  {
59      struct message msg;
60      int r;
61      int i;
62      for (i = 0; i < 3; i++) {
63          // -2 означает, что будут приниматься сообщения типа 2 или ниже.
64          r = msgrcv(qid, &msg, sizeof(struct message), -2, 0);
65          printf("'%s'\n", msg.mtext);
66      }
67  }
68
69  int main(int argc, char *argv[])
70  {
71      // Создать частную (неименованную) очередь сообщений.
72      int mqid;
73      mqid = msgget(IPC_PRIVATE, S_IREAD | S_IWRITE);
74      if (mqid == -1) {
75          perror("msgget");
76          exit(1);
77      }
78
79      pid_t pid = fork();
80      if (pid == 0) {
81          consumer(mqid);
82          exit(0);
83      }
84      else {
85          int status;
86          producer(mqid);
87          wait(&status);
```

```
88      }
89      // Удалить очередь сообщений.
90      int r = msgctl(mqid, IPC_RMID, 0);
91      if (r)
92          perror("msgctl");
93      return 0;
94  }
```

В приведенном примере следует отметить ряд моментов: здесь применяются неименованная очередь и сообщения фиксированного размера. Получатель использует функцию `msgrcv`, о которой еще не упоминалось. Указание `-2` в качестве типа допустимых сообщений говорит о том, что будут приниматься сообщения, относящиеся к типу 2 и ниже. После выполнения программы можно наблюдать любопытный итоговый вывод:

```
$ cc -o sysv-msgq-example sysv-msgq-example.c
$ ./sysv-msgq-example
'type 1 – first'
'type 1 – third'
'type 2 – second'
```

Обратите внимание на то, что сообщения типа 1 получаются первыми. Более низкое значение типа сообщения означает, что оно имеет более высокий приоритет и будет получено первым. В результате мы получаем сообщения в ином порядке, чем тот, в котором они были отправлены. Для того что иметь возможность получать сообщения в том же порядке, в котором они были отправлены (FIFO), необходимо установить нулевое значение для аргумента `type` в `msgrcv`.

Если аргумент `type` функции `msgrcv` имеет ненулевое значение, сообщения будут приниматься в соответствии с их приоритетом, как показано далее:

- положительный — принимаются только сообщения указанного типа;
- отрицательный — принимаются только сообщения указанного типа с абсолютным или более низким значением.

Операционная система Linux поддерживает также использование флага `MSG_EXCEPT`, который не является частью какого-либо стандарта. Когда тип сообщений указывается как положительный, а флаг `MSG_EXCEPT` передается в качестве аргумента `flags` функции `msgrcv`, это производит эффект инвертированного отбора, то есть вместо сообщений типа N `msgrcv` будет принимать сообщения любых других типов, кроме относящихся к типу N.

7.7.2. Очередь сообщений POSIX

API-интерфейс POSIX функционально схож с API-интерфейсом System V, за исключением того, что в нем моделирование очередей сообщений очень похоже на формирование файловой модели POSIX. Как и в файловой модели, очереди сообщений обладают именами, которые должны подчиняться тем же правилам, что и имена файлов. Очереди сообщений можно открывать, закрывать, создавать и отсоединять (удалять) точно так же, как файлы. В Linux очереди сообщений используют дескрипторы файлов подобно открытым файлам. API-интерфейс очередей

сообщений POSIX должен быть интуитивно понятен программисту, для того чтобы его можно было успешно использовать для работы с файлами.

Создание, открытие, закрытие и удаление очередей сообщений POSIX

Стандарт POSIX не предусматривает наличия функции creat для создания очередей сообщений. Вместо этого они создаются при помощи функции mq_open с флагом O_CREAT. Функция mq_open определяется следующим образом:

```
mqd_t mq_open(const char *name, int oflag, ...);
```

Подпись функции во многом схожа с системным вызовом open. Стандарт POSIX допускает (но не требует), чтобы возвращаемое значение было дескриптором файла. В операционной системе Linux оно как раз и будет дескриптором файла. Нет ничего удивительного в том, что аргумент oflag принимает те же аргументы, что и системный вызов open, включая O_CREAT, O_READ, O_WRITE и O_RDWR. Но когда вызывающий оператор использует флаг O_CREAT, функции mq_open для создания очереди сообщений потребуются два дополнительных параметра. Третьим аргументом является режим, который, как и в случае с вызовом open, определяет разрешения на чтение/запись для очереди сообщений. Он принимает значения, аналогичные флагам разрешений (например, S_IREAD), и «навязывается» соответствующей очереди сообщений посредством последующих вызовов функции mq_open. Четвертым аргументом для mq_open, наличие которого требует O_CREAT, является указатель на структуру mq_attr, которая определяется так:

```
struct mq_attr
{
  long int mq_flags;       Определяемые реализацией флаги, включая O_NONBLOCK
  long int mq_maxmsg;      Максимальное число сообщений, ожидающих в очереди
  long int mq_msgsize;     Максимальный размер любого сообщения в очереди
  long int mq_curmsgs;     Текущее число сообщений в очереди
\
  long int __pad[4];
};
```

Аргумент mq_attr является опциональным и может быть опущен, при этом система будет применять в качестве атрибутов определяемые реализацией значения по умолчанию. Поскольку использовать очередь сообщения с неизвестными атрибутами нельзя, стандарт POSIX предусматривает наличие функции mq_getattr для извлечения данной структуры для конкретной очереди сообщений. Также существует функция mq_setattr, которая позволяет «настраивать» флаги очереди. Обе эти функции определяются следующим образом:

```
int mq_setattr(mqd_t mqdes, const struct mq_attr *iattr, struct mq_attr *oattr);
int mq_getattr(mqd_t mqdes, struct mq_attr *oattr);
```

Поля mq_maxmsg и mq_msgsize используются только вызовом mq_open при создании очереди сообщений. Эти поля определяют максимальное число сообщений, которое будет содержать очередь, а также размер каждого сообщения соответственно. Поскольку данные значения остаются фиксированными на протяжении всей жизни

очереди, они игнорируются функцией mq_setattr. Поле mq_curmsgs при передаче в качестве ввода функции mq_open или mq_setattr также игнорируется; оно заполняется лишь выводом, генерируемым системным вызовом mq_getattr, когда тот используется.

Так как идентификатор очереди сообщения, возвращаемый Linux-версией функции mq_open, является дескриптором файла, вам потребуется функция close для высвобождения этого дескриптора и всех связанных ресурсов. Для выполнения этой задачи стандарт POSIX предусматривает функцию mq_close:

```
int mq_close(mqd_t mqdes);
```

Как вы уже догадались, Linux-версия функции mq_close представляет собой псевдоним системного вызова close. Для перманентного удаления очереди сообщения используется функция mq_unlink, которая формируется по образцу системного вызова unlink:

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

Очередь сообщений, подобно обычному файлу, не удаляется до тех пор, пока все процессы не отсоединятся от нее. Любой процесс, который обладает открытой очередью сообщений во время отсоединения, сможет продолжать использовать ее.

Чтение и запись в очереди сообщений POSIX

Функции, используемые для чтения и записи сообщений, во многом схожи со своими System V-аналогами. Далее приведены прототипы функций mq_send и mq_receive, слегка укороченные с целью экономии места:

```
int mq_send(mqd_t mqdes, char *ptr, size_t len, unsigned prio);
ssize_t mq_receive(mqd_t mqdes, char *ptr, size_t len, unsigned *prio);
```

Как и вызовы System V, данные функции включают приоритет, однако, в отличие от System V, POSIX устанавливает верхний предел размера сообщений. Функция mq_send принимает аргументы аналогично системному вызову write с дополнением в виде аргумента, касающегося приоритета. mq_send осуществляет запись сообщений, размер которых меньше того максимального лимита, который установлен для конкретной очереди сообщений. Функция mq_receive требует, чтобы получатель обеспечил наличие пространства, достаточного для принятия сообщения максимального размера, и возвращает фактический размер полученного сообщения.

Чтение из пустой очереди сообщений по умолчанию приводит к блокировке соответствующего процесса до тех пор, пока в ней не появятся сообщения. Аналогично запись в наполненную очередь сообщений также вызовет блокировку вашего процесса. Для того чтобы изменить данное поведение, при открытии очереди сообщений необходимо воспользоваться флагом O_NONBLOCK. Данный флаг можно динамически менять при помощи функции mq_setattr.

7.7.3. Различия между очередями сообщений POSIX и очередями сообщений System V

В поведении очередей System V и POSIX имеется ряд существенных различий. Очереди System V позволяют содержащимся в них сообщениям варьироваться по

размеру при условии, что размер прочитанного сообщения будет соответствовать размеру записанного сообщения. Очереди POSIX позволяют отправителю осуществлять запись сообщений с переменной длиной, несмотря на то что от считывателя данных требуется обеспечивать наличие пространства, достаточного для приема сообщений фиксированного размера, то есть выполнение вызова `msg_receive` потерпит неудачу, если объем данного пространства недостаточен для того, чтобы сообщение поместилось целиком.

Второе различие состоит в том, что System V позволяет считывателю данных осуществлять элементарную фильтрацию сообщений на основе определенного приоритета, а POSIX-сообщения доставляются в порядке строгой приоритетности, то есть считыватель данных не может выбирать, сообщения с каким приоритетом читать, или же осуществлять блокировку до тех пор, пока не поступит сообщение с определенным приоритетом. При чтении очереди сообщений POSIX всегда извлекаются сообщения с самым высоким приоритетом. Если в очереди оказывается более одного сообщения с данным приоритетом, первым в очереди станет то из них, которое будет прочитано первым (подобно FIFO).

7.8. Семафоры

Когда два процесса совместно используют ресурсы, важно, чтобы они делали это на основе организованного доступа; в противном случае может возникнуть неразбериха в форме искаженного вывода и краха программ. В вычислительной сфере слово *семафор* используется в качестве термина для обозначения флага особого типа, который применяется для синхронизации параллельных процессов. Семафоры выступают в роли своего рода «светофоров» для таких процессов.

В данном случае мы также имеем два API-интерфейса, посредством которых можно использовать семафоры: System V и POSIX. Основным семафором является счетчик. Он используется для отслеживания каких-либо ограниченных ресурсов. Поскольку обычно на один ресурс приходится по одному семафору, счетчик никогда не увеличивается более чем на единицу. Иногда он называется *двоичным семафором*, так как значение счетчика семафора всегда равно 1 или 0.

Детальное рассмотрение семафоров и параллелизма выходит за рамки данной книги, однако мы все же ознакомимся с примером их использования посредством API-интерфейса POSIX. В листинге 7.9 приведен пример двух процессов, которые пытаются отправить две половины одного сообщения на стандартный вывод. В данном случае стандартным выводом является совместно используемый ресурс, который необходимо *защитить* при помощи семафора.

Листинг 7.9. `hello-unsync.c: два несинхронизированных процесса, пытающихся осуществить запись на стандартный вывод`

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5 #include <sys/file.h>
```

```
 6 #include <sys/times.h>
 7 #include <sys/stat.h>
 8 #include <semaphore.h>
 9 #include <assert.h>
10
11 // Простой ждущий цикл busywait для исключения синхронизации.
12 void busywait(void)
13 {
14     clock_t t1 = times(NULL);
15     while (times(NULL) – t1 < 2);
16 }
17
18 /*
19 ** Простое сообщение. 1-я часть выводится одним процессом
20 ** 2-я часть выводится другим процессом. Синхронизация отсутствует,
21 ** в результате итоговый вывод будет бессодержательным (то есть "мусором").
22 */
23 int main(int argc, char *argv[])
24 {
25     const char *message = "Hello World\n";
26     int n = strlen(message) / 2;
27
28     pid_t pid = fork();
29     int i0 = (pid == 0) ? 0 : n;
30     int i;
31
32     for (i = 0; i < n; i++) {
33         write(1, message + i0 + i, 1);
34         busywait();
35 }
36 }
```

При выполнении программы из листинга 7.9 вы неизменно будете наблюдать на экране «мусор». Обратите внимание на то, что я задействовал ждущий цикл `busywait` для того, чтобы слегка рандомизировать этапы выполнения. В противном случае планировщик может ненамеренно позволить этой программе выполняться в правильной последовательности:

```
$ cc -o hello-unsync hello-unsync.c
$ $ ./hello-unsync
HWelolo rld            "Мусор", вызванный несинхронизированным доступом
                       к стандартному выводу
```

Временна́я диаграмма к листингу 7.9 приведена на рис. 7.2. Основная проблема заключается в том, что выполнение обоих вызовов `write` может происходить в произвольном порядке. Я специально усугубил ситуацию, осуществляя запись по одному байту за один раз. Для наведения порядка нам потребуется «полицейский-регулировщик», который не позволит более чем одному процессу получить доступ к стандартному выводу в один и тот же момент. В листинге 7.10 вы сможете увидеть, как применить для этого семафор при помощи API-интерфейса POSIX.

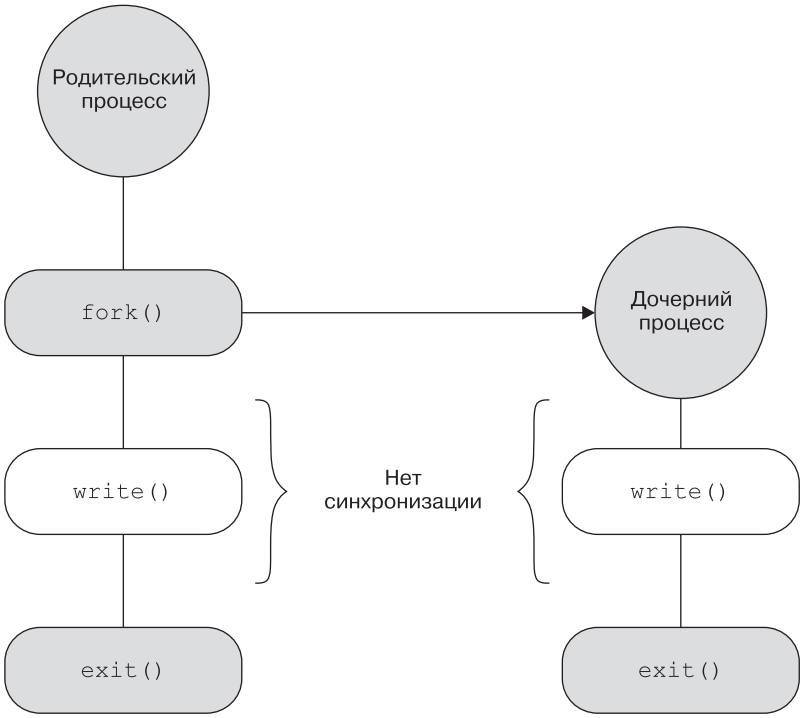


Рис. 7.2. Временная диаграмма к листингу 7.9: несинхронизированный программный код приводит к генерированию «мусора»

Листинг 7.10. hello-sync.c: упорядоченный вывод с участием двух процессов

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <stdlib.h>
 4 #include <unistd.h>
 5 #include <sys/file.h>
 6 #include <sys/times.h>
 7 #include <sys/stat.h>
 8 #include <semaphore.h>
 9 #include <assert.h>
10
11 // Простой ждущий цикл busywait для исключения синхронизации.
12 void busywait(void)
13 {
14     clock_t t1 = times(NULL);
15     while (times(NULL) – t1 < 2);
16 }
17
18 /*
19 ** Простое сообщение. 1-я часть выводится одним процессом
20 ** 2-я часть выводится другим процессом. Синхронизация
21 ** обеспечивается при помощи семафора.
```

```
22 */
23 int main(int argc, char *argv[])
24 {
25     const char *message = "Hello World\n";
26     int n = strlen(message) / 2;
27
28     // Создать семафор.
29     sem_t *sem = sem_open("/thesem", O_CREAT, S_IRUSR | S_IWUSR);
30     assert(sem != NULL);
31
32     // Инициализировать счетчик семафора со значением, равным нулю.
33     int r = sem_init(sem, 1, 0);
34     assert(r == 0);
35
36     pid_t pid = fork();
37     int i0 = (pid == 0) ? 0 : n;
38     int i;
39
40     // Родительский процесс ожидает увеличения значения семафора.
41     if (pid)
42         sem_wait(sem);
43
44     for (i = 0; i < n; i++) {
45         write(1, message + i0 + i, 1);
46         busywait();
47     }
48
49     // Дочерний процесс увеличивает значения семафора по завершении.
50     if (pid == 0)
51         sem_post(sem);
52 }
```

На рис. 7.3 можно увидеть откорректированную временну́ю диаграмму. Более подробно об API-интерфейсе POSIX мы поговорим немного позднее, а здесь необходимо отметить, что приведенный пример с использованием семафоров довольно прост. Счетчик семафора инициализируется со значением, равным нулю, поэтому любой процесс, ожидающий семафор (когда тот примет ненулевое значение), будет блокироваться. В приведенном примере я решил позволить родительскому процессу блокироваться и дать возможность дочернему процессу осуществить вывод первой половины сообщения. Таким образом, первое, что делает родительский процесс, — ожидает семафор с использованием функции `sem_wait`, что вызывает его блокировку.

Дочерний процесс по завершении увеличивает значение семафора при помощи POSIX-функции `sem_post`. Это приводит к снятию блокировки с родительского процесса и дает ему возможность вывести вторую половину сообщения. После этого данное сообщение всякий раз будет выводиться в корректном виде:

```
$ cc -o hello-sync hello-sync.c
$ ./hello-sync
Hello World
```

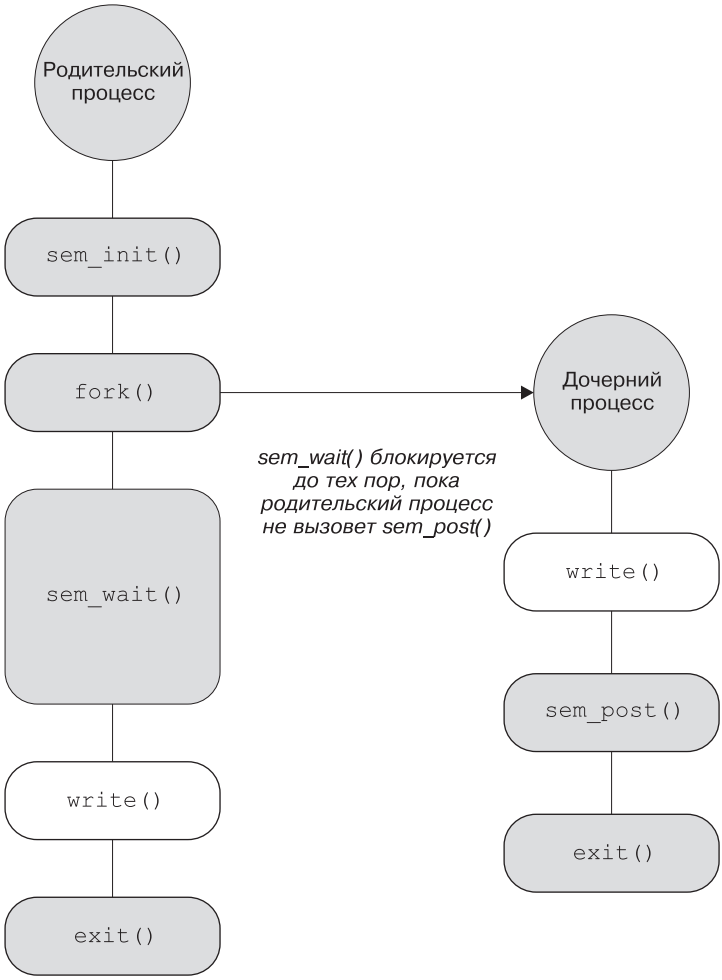


Рис. 7.3. Временная диаграмма к листингу 7.10: семафор выступает в роли «полицейского-регулировщика»

7.8.1. Работа с семафорами с использованием API-интерфейса POSIX

В листинге 7.10 вы могли наблюдать пример использования семафора для синхронизации родительского и дочернего процессов. Семафоры POSIX имеют имена, которые доступны для просмотра в системе. Семафор существует с момента создания и вплоть до своего отсоединения (удаления) или перезагрузки системы, в чем можно убедиться на примере программы из листинга 7.11.

Листинг 7.11. posix_sem.c: простая программа, работающая с семафором POSIX

```
1 #include <stdio.h>
2 #include <string.h>
```

```
 3 #include <stdlib.h>
 4 #include <assert.h>
 5 #include <errno.h>
 6 #include <unistd.h>
 7 #include <sys/file.h>
 8 #include <sys/stat.h>
 9 #include <semaphore.h>
10
11 int main(int argc, char *argv[])
12 {
13     const char *semname = "/mysem";
14
15     // Создать семафор и инициализировать счетчик семафора со значением,
       // равным нулю.
16     // Поскольку мы используем флаг O_EXCL, выполнение
       // соответствующего вызова потерпит неудачу, если
17     // семафор существует, при этом переменной errno будет
       // присвоено значение EEXIST.
18     sem_t *sem = sem_open(semname,
19                           O_CREAT | O_EXCL,
20                           S_IRUSR | S_IWUSR,
21                           0);
22     if (sem != SEM_FAILED) {
23         printf("создание нового семафора\n");
24     }
25     else if (errno == EEXIST) {
26         // Семафор существует, поэтому при его открытии не использовать
           // флаг O_EXCL
27         printf("семафор существует\n");
28         sem = sem_open(semname, 0);
29     }
30
31     assert(sem != SEM_FAILED);
32
33     int op = 0;
34
35     // Аргумент пользователя: нулевой, положительный или отрицательный
36     if (argc > 1)
37         op = atoi(argv[1]);
38
39     if (op > 0) {
40         printf("увеличение значения семафора\n");
41         sem_post(sem);
42     }
43     else if (op < 0) {
44         printf("уменьшение значения семафора\n");
45         sem_wait(sem);
46     }
47     else {
48         printf("значение семафора не подверглось модификации\n");
```

```
49     }
50     int val;
51     sem_getvalue(sem, &val);
52     printf("значение семафора равно %d\n", val);
53
54     return 0;
55 }
```

Семафор создается при первом запуске программы при помощи функции sem_open. Первые три аргумента, принимаемые функцией sem_open, идентичны тем, что используются системным вызовом open(2). Четвертый аргумент применяется только при создании семафора и содержит счетчик семафора. Прототип выглядит следующим образом:

```
sem_t *sem_open( const char *name, int oflag, ...);
```

Я использовал флаг O_EXCL для того, чтобы форсировать неудачное выполнение sem_open в случае, если семафор существует. Это вовсе не обязательно делать, и я мог пропустить этот этап. Благодаря использованию этого флага при создании семафоров можно выводить разные сообщения.

Данная программа выполняет одну операцию в отношении семафора на основе аргумента пользователя. Пользователь может увеличивать значение семафора посредством использования положительного целого числа в качестве аргумента или уменьшать его, используя отрицательный аргумент, например:

```
$ cc -o posix_sem posix_sem.c -lrt
$ ./posix_sem 1                    Создание семафора и увеличение
created new semaphore              его значения.
incrementing semaphore
semaphore value is 1

$ ./posix_sem 1                    Повторное увеличение значения семафора.
semaphore exists
incrementing semaphore
semaphore value is 2               Значение семафора отражает число вызовов
                                   sem_post.

$ ./posix_sem -1                   Уменьшение значения семафора.
semaphore exists
decrementing semaphore
semaphore value is 1

$ ./posix_sem 0                    Никаких операций не производится
semaphore exists
not modifying semaphore
semaphore value is 1               Значение = 2 x sem_post - 1 x sem_wait

$ ./posix_sem -1
semaphore exists
decrementing semaphore
semaphore value is 0
```

```
$ ./posix_sem -1
semaphore exists
decrementing semaphore                Семафор блокируется!
```

В листинге 7.11 используется именованный семафор. В данном примере я не стал прибегать к вызову функции sem_close, которая, как вы уже могли догадаться, высвобождает ресурсы пространства пользователя, занятые семафором. Также существует функция sem_unlink, которая используется для удаления семафора из системы и высвобождения системных ресурсов, занятых им.

API-интерфейс POSIX также поддерживает работу с неименованными семафорами, однако здесь необходимо учитывать один момент: в операционной системе Linux подобные семафоры могут работать только с потоками. Неименованный семафор определяется без использования функции sem_open, которая требует указания имени. Вместо этого приложение вызывает функцию sem_init, прототип которой выглядит так:

```
int sem_init( sem_t *sem, int pshared, int value );
```

7.8.2. Работа с семафорами с использованием API-интерфейса System V

API-интерфейс System V для работы с семафорами соответствует API-интерфейсам, используемым для взаимодействия с разделяемой памятью и очередями сообщений, то есть приложение определяет ключ, а система присваивает идентификатор, когда создается семафор. В листинге 7.12 вы можете увидеть эквивалент приложения, которое приводилось в листинге 7.11.

Листинг 7.12. sysv_sem.c: приложение, работающее с семафорами System V

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <stdlib.h>
 4 #include <assert.h>
 5 #include <errno.h>
 6 #include <unistd.h>
 7 #include <sys/stat.h>
 8 #include <sys/sem.h>
 9
10 int main(int argc, char *argv[])
11 {
12     // Создать ключ при помощи ftok
13     key_t semkey = ftok("/tmp", 'a');
14
15     // Создать семафор – "массив" с длиной, равной 1.
16     // Поскольку мы используем флаг IPC_EXCL, выполнение
       // соответствующего вызова потерпит неудачу, если
17     // семафор существует, при этом переменной errno будет присвоено
       // значение EEXIST.
18     int semid =
19         semget(semkey, 1, IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);
```

```
20      if (semid != -1) {
21          printf("создание нового семафора\n");
22      }
23      else if (errno == EEXIST) {
24          // Семафор существует, поэтому при его открытии не использовать
            // флаг IPC_EXCL
25          printf("семафор существует\n");
26          semid = semget(semkey, 1, 0);
27      }
28
29      assert(semid != -1);
30
31      // Примечание: "ожидание нулевого значения" является легитимной
        // операцией System V в отношении семафора
32      // Мы выполняем операцию только при наличии аргумента
33      if (argc == 2) {
34          int op = atoi(argv[1]);
35
36          // Инициализировать структуру операций,
37          // которая применяется по отношению к массиву семафоров,
38          // но в данном случае мы воспользуемся только одним.
39          struct sembuf sb = {
40              .sem_num = 0, // индекс в массиве.
41              .sem_op = op, // значение, суммированное со значением счетчика
42              .sem_flg = 0 // флаги (например, IPC_NOWAIT)
43          };
44
45          // Все задачи решаются посредством одного вызова!
46          int r = semop(semid, &sb, 1);
47
48          assert(r != -1);
49          printf("операция %d выполнена\n", op);
50      }
51      else {
52          printf("операций не производилось\n");
53      }
54
55      printf("semid %d value %d\n", semid, semctl(semid, 0, GETVAL));
56
57      return 0;
58  }
```

Указанные функции почти в точности соответствуют тем, которые используются в случае с API-интерфейсом POSIX, при этом имеются некоторые важные отличия. Во-первых, API-интерфейс System V использует только функцию `semop` в качестве эквивалента операций `wait` и `post`. Во-вторых, API-интерфейс System V включает так называемую операцию *ожидания нулевого значения (wait for zero)*, которая не модифицирует значение семафора, однако блокирует вызывающий оператор до тех пор, пока значение счетчика семафора не станет равным нулю.

7.9. Заключение

В данной главе мы рассмотрели основы межпроцессного взаимодействия. Вы познакомились с рядом API-интерфейсов и увидели пример использования каждого из них. В большинстве ситуаций для решения одинаковых задач можно применять минимум два API-интерфейса. Я рассказал вам их предысторию, а также привел всестороннее толкование, которого должно быть вполне достаточно для того, чтобы вы смогли выбрать для себя наиболее подходящий API-интерфейс.

7.9.1. Системные вызовы и API-интерфейсы, использованные в этой главе

В данной главе мы рассмотрели множество API-интерфейсов, которые подразделяются на несколько категорий.

Разное

- `flock` — помещает рекомендательную блокировку на файл.
- `ftok` — создает уникальный ключ, используемый в случае с System V IPC.
- `lockf` — помещает обязательную блокировку на файл.
- `select` — функция, используемая для блокировки или опроса разнообразных дескрипторов файлов.

Разделяемая память

- `shm_open`, `shm_unlink`, `mmap` — инструменты для работы с разделяемой памятью POSIX.
- `shmget`, `shmat`, `shmdt`, `shmctl` — инструменты для работы с разделяемой памятью System V.

Сигналы

- `kill`, `sigqueue` — функции для отправки сигналов.
- `sigaction`, `signal` — функции для определения обработчиков сигналов.
- `sigpending`, `sigsuspend` — функции для ожидания сигналов.
- `sigprocmask`, `sigemtpyset`, `sigfillset`, `sigaddset`, `sigdelset`, `sigismember` — функции для манипулирования масками сигналов.

Каналы (конвейеры)

- `mkfifo` — используется для создания именованных каналов.
- `pipe` — используется для создания неименованных каналов.

Сокеты

- `bind`, `listen`, `accept`, `close` — важнейшие функции для создания серверов, ориентированных на соединения.

- `connect` — клиентская функция для установления соединений с сокетом сервера.
- `socket` — основная функция для создания сокетов.

Очереди сообщений

- `mq_open`, `mq_close`, `mq_unlink`, `mq_send`, `mq_receive`, `mq_setattr`, `mq_getattr` — функции для работы с очередями сообщений POSIX.
- `msgget`, `msgsend`, `msgrcv`, `msgctl` — функции для работы с очередями сообщений System V.

Семафоры

- `sem_open`, `sem_close`, `sem_post`, `sem_wait` — функции для работы с семафорами POSIX.
- `semget`, `semop` — функции для работы с семафорами System V.

7.9.2. Рекомендуемая литература

- *Gallmeister B.* POSIX 4 Programmers Guide. — Sebastopol, Calif.: O'Reilly Media, Inc., 1995.
- *Robbins A.* Linux Programming by Example, The Fundamentals. — Englewood Cliffs, N. J.: Prentice Hall, 2004.
- *Stevens W. R. et al.* UNIX Network Programming. — Boston, Mass.: Addison-Wesley, 2004.

7.9.3. Веб-ссылки

- www.opengroup.org — ресурс, где можно отыскать публикации, касающиеся стандарта POSIX (IEEE Standard 1003.2) и многих других (требуется регистрация).
- www.unix.org — ресурс, где публикуются спецификации Single UNIX Specification.

8 Отладка межпроцессного взаимодействия IPC при помощи команд оболочки

8.1. Введение

В данной главе рассмотрим методики и команды, которые используются из оболочки для отладки межпроцессного взаимодействия IPC. При отладке данного механизма всегда будет нелишним иметь нейтральное стороннее решение на случай возникновения каких-либо проблем.

8.2. Инструментарий для работы с открытыми файлами

Процессы, которые оставляют файлы открытыми, могут стать источником проблем. Дескрипторы файлов также могут страдать от «утечек», как и память, например потреблять излишние ресурсы. Каждый процесс обладает ограниченным набором дескрипторов файлов, которые он может держать открытыми, поэтому если сбойное приложение будет продолжать открывать дескрипторы файлов, не закрывая их впоследствии, то это приведет к его краху с присваиванием переменной `errno` значения `EMFILE`. Если вы позаботились о включении в свой программный код обработчика ошибок, то без труда сможете понять, в чем причина возникших проблем. Однако как же следует поступать далее?

Файловая система `procfs` является весьма полезным средством для устранения подобных проблем. В каталоге `/proc/PID/fd` можно увидеть все открытые файлы любого конкретного процесса. Здесь каждый открытый файл имеет вид символической ссылки. Имя ссылки является номером дескриптора файла, при этом она будет указывать на открытый файл.

8.2.1. lsof

Более информативный вывод можно получить при помощи команды `lsof`. Если не указывать никаких аргументов, данная команда выведет перечень всех открытых

файлов в системе, который может оказаться весьма длинным. Но даже в этом случае будут выведены лишь те файлы, на просмотр которых у пользователя имеется разрешение. При помощи параметра `-p` можно выводить сведения, касающиеся только какого-то одного процесса:

```
$ lsof -p 16894
COMMAND   PID   USER    FD   TYPE   DEVICE       SIZE     NODE NAME
cat     16894   john   cwd    DIR    253,0       4096   575355 /home/john
cat     16894   john   rtd    DIR    253,0       4096        2 /
cat     16894   john   txt    REG    253,0      21104   159711 /bin/cat
cat     16894   john   mem    REG    253,0     126648   608855 /lib/ld-2.3.5.so
cat     16894   john   mem    REG    253,0    1489572   608856 /lib/libc-2.3.5.so
cat     16894   john   mem    REG      0,0                  0 [heap]
cat     16894   john   mem    REG    253,0   48501472   801788 .../locale-archive
cat     16894   john    0r   FIFO      0,5                176626 pipe
cat     16894   john    1u    CHR    136,2                     4 /dev/pts/2
cat     16894   john    2w    CHR      1,3                  1510 /dev/null
cat     16894   john    3r    REG    253,0        167   575649 /home/john/.bashrc
```

В данном выводе представлены не только дескрипторы файлов, но и файлы, отображаемые (загруженные) в память. В колонке с заголовком `FD` *(дескриптор файла)* можно узнать, является выведенный объект дескриптором файла или отображением. Отображение не требует наличия дескриптора файла после совершения вызова `mmap`, поэтому колонка `FD` относительно каждого отображения содержит текстовые данные, которые служат индикатором его типа. Дескрипторы файлов приводятся с номерами и типом доступа, а в табл. 8.1 можно увидеть пояснение к текстовым данным, которые содержатся в колонке `FD`.

Таблица 8.1. Текстовые данные, приводимые в колонке FD вывода команды lsof

Идентификатор	**Значение**
cwd	Текущий рабочий каталог
ltx	Текст совместно используемой библиотеки (код и данные)
mem	Отображаемый в память файл
mmap	Отображаемое в память устройство
pd	Родительский каталог
rtd	Корневой каталог
txt	Текст программы (код и данные)
{digit}r	Дескриптор файла, открытого только для чтения
{digit}w	Дескриптор файла, открытого только для записи
{digit}u	Дескриптор файла, открытого для чтения и записи

Команду `lsof` можно использовать также для выяснения того, каким процессом был открыт тот или иной файл, если ввести в качестве аргумента имя этого файла. Команда `lsof` поддерживает множество других параметров. Для получения дополнительной информации см. страницу руководства `lsof(8)`.

8.2.2. fuser

Еще одним инструментом для «отслеживания» открытых файлов является команда `fuser`. Допустим, вам необходимо отследить процесс, который осуществляет запись файла большого размера в файловую систему. В этом случае команду `fuser` можно использовать следующим образом:

```
$ fuser some-huge-file.txt   Какой процесс открыл данный файл?
some-huge-file.txt: 17005
```

Если полученный результат вас удовлетворяет, можно пойти дальше и принудительно завершить данный процесс. Команда `fuser` позволяет сделать это при помощи параметра `-k`:

```
]$ fuser -k -KILL some-huge-file.txt
some-huge-file.txt: 17005
[1]+ Killed                  cat some-huge-file.txt
```

8.2.3. ls

Использование параметра `-l` в сочетании с командой `ls` дает возможность пользователю получать *информативный* вывод. Вы, несомненно, уже знаете, что в итоговом выводе будут присутствовать сведения, касающиеся файлового имени, разрешений и размера определенного файла. Вывод также содержит информацию о том, какой тип файла видит перед собой пользователь, например:

```
$ ls -l /dev/log /dev/initctl /dev/sda /dev/zero
prw------- 1 root root    0 Oct 8 09:13 /dev/initctl Канал (p)
srw-rw-rw- 1 root root    0 Oct 8 09:10 /dev/log     Сокет (s)
brw-r----- 1 root disk 8, 0 Oct 8 04:09 /dev/sda     Блочное устройство (b)
crw-rw-rw- 1 root root 1, 5 Oct 8 04:09 /dev/zero    Символьное
                                                     устройство(c)
```

8.2.4. file

Данная команда позволяет получать весьма дружественный к пользователю вывод, содержащий сведения о типе файла, который он перед собой видит, например:

```
file /dev/log /dev/initctl /dev/sda /dev/zero\
/dev/log:     socket
/dev/initctl: fifo (named pipe)
/dev/sda:     block special (8/0)        Приводятся старший/младший номера
/dev/zero:    character special (1/5)    Приводятся старший/младший номера
```

Каждый файл приводится наряду с простым, удобочитаемым описанием своего типа. Команда `file` также может выводить сведения о плоских файлах различных типов, например о файлах в формате ELF или файлах образов. Для распознавания типов файлов данная команда использует обширную базу данных магических чисел. Эта база данных может быть расширена пользователем. Для получения дополнительной информации см. страницу руководства `file(1)`.

8.2.5. stat

Команда stat является оберткой системного вызова stat, которая может использоваться из оболочки. Итоговый вывод будет содержать в удобочитаемом формате те же данные, которые можно было бы извлечь при помощи системного вызова stat, например:

```
stat /dev/sda
  File: `/dev/sda'
  Size:  0               Blocks: 0 IO         Block: 4096   block special file
Device:  eh/14d  Inode: 1137        Links: 1    Device type: 8,0
Access:  (0640/brw-r-----)  Uid: (    0/    root)  Gid: (    6/    disk)
Access:  2006-10-08 04:09:34.750000000 -0500
Modify:  2006-10-08 04:09:34.750000000 -0500
Change:  2006-10-08 04:09:50.000000000 -0500
```

Команда stat, подобно функции printf, также позволяет осуществлять форматирование итогового вывода с использованием специально определяемых символов форматирования, о которых можно прочесть на странице руководства man stat(1).

8.3. Сброс данных из файла

Вам, скорее всего, уже доводилось пользоваться некоторыми инструментами для сброса данных из файла, среди которых присутствовал текстовый редактор для работы с текстовыми файлами. Любые инструменты, которые используются для обработки обычных текстовых файлов, могут применяться для работы с текстовыми файлами ASCII. Некоторые из них поддерживают и другие кодировки — если не посредством параметров командной строки, то, возможно, используя настройки локализации, например:

```
$ wc -w konnichiwa.txt                  Содержит японскую фразу приветствия
                                        konnichiwa (одно слово)
0 konnichiwa.txt                        wc выдает 0 слов, основываясь
                                        на текущей локализации
$ LANG=ja_JP.UTF-8 wc -w konnichiwa.txt
1 konnichiwa.txt                        Форсирование японской локализации
                                        приводит к выводу корректного
                                        результата
```

Некоторые инструменты могут оказаться пригодными для просмотра двоичных данных, однако не все они способны интерпретировать данные.

8.3.1. Команда strings

Зачастую оказывается так, что тестовые строки в файле с данными могут стать ключом к пониманию его содержимого. Иногда их бывает вполне достаточно для того, чтобы пользователь узнал все, что ему необходимо. Однако когда текстовые

строки находятся в компании двоичных данных, потребуется кое-что получше простой команды `cat`.

Команда `strings` по умолчанию включает в итоговый вывод только строки длиной четыре символа или более; все, что будет меньше, игнорируется. Такое поведение можно изменить при помощи параметра `-n`, который позволяет определять минимальную длину искомой строки. Для просмотра двоичных данных, содержащихся в файле, нам потребуются другие инструменты.

8.3.2. Команда xxd

Команда `xxd` является частью текстового редактора Vim и позволяет получать вывод, весьма напоминающий тот, который генерируется редактором `bvi`. Разница заключается лишь в том, что `xxd` не является текстовым редактором. Как и `bvi`, эта команда выводит данные в шестнадцатеричном представлении и позволяет просматривать только ACSII-символы:

```
$ xxd floats-ints.dat
0000000: 4865 6c6c 6f20 576f 726c 640d 2020 2020 Hello World.
0000010: 2020 2020 2020 2020 0abf e381 93e3 8293         ........
0000020: e381 abe3 81a1 e381 af0d 2020 2020 2020     ..........
0000030: 2020 2020 0a0a 0000 0000 0000 0000 0000     ............
0000040: 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000050: 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000060: 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000070: 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000080: 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000090: 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000a0: 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000b0: 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000c0: 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000d0: 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000e0: 0000 0000 0000 0000 0000 0000 0000 0000 ................
00000f0: 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000100: 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000110: 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000120: 0000 0000 0000 0000 0000 0000 0000 0000 ................
0000130: 0090 5f01 0091 5f01 0092 5f01 0093 5f01 .._..._..._..._.
0000140: 0000 c8af 4780 c8af 4700 c9af 4780 c9af ....G...G...G...
0000150: 47                                      G
```

`xxd` по умолчанию выводит 16-битные слова (значения), однако данное поведение можно регулировать при помощи параметра `-g`. Например, чтобы просмотреть данные в группах по 4 байта, необходимо ввести `-g4`. При этом, однако, необходимо убедиться в том, что группы сохраняют порядок байтов в файле. Это означает, что 32-битные слова, выводимые на архитектуре IA32, будут некорректны. Данная архитектура сначала сохраняет слова с наименее значимым байтом, то есть использует порядок байтов, обратный тому, который применяется в памяти. Иногда такой порядок называется *Little Endian (прямой порядок байтов)*. Для того чтобы на эк-

ране отображались корректные слова, порядок байтов необходимо реверсировать, чего команда xxd не делает.

В некоторых ситуациях это может оказаться полезным. Если, к примеру, необходимо просмотреть данные с обратным порядком байтов *(Big Endian)* на компьютере, где используется прямой порядок байтов, вам не потребуется переупорядочивать байты. Сетевые протоколы для передачи данных используют так называемый *сетевой порядок байтов*, который является аналогом обратного порядка байтов. Таким образом, если вы будете просматривать файл, содержащий протокольные заголовки из сокета, вам потребуется инструмент вроде xxd, который не изменяет порядка байтов.

8.3.3. Команда hexdump

Команда hexdump, как свидетельствует ее имя, позволяет сбрасывать содержимое файла в шестнадцатеричном виде. Подобно команде xxd, по умолчанию hexdump выводит данные в 16-битном шестнадцатеричном представлении, однако порядок байтов корректируется на архитектурах, где используется прямой порядок байтов, поэтому итоговый вывод, выполненный с помощью команд xxd и hexdump, может различаться.

Команда hexdump больше, чем xxd, подходит для терминального вывода сведений, поскольку она устраняет пропущенные дубликаты строк данных, для того чтобы не загромождать экран. hexdump поддерживает генерирование вывода и в других форматах, а не только в 16-битном шестнадцатеричном представлении, однако они будут не столь удобочитаемы.

8.3.4. Команда od

od — это традиционная команда Unix для *сброса данных в восьмеричном представлении*. Несмотря на свое имя, команда od способна представлять данные в множестве других форматов с использованием различных размеров слов. Параметр -t является универсальным переключателем для изменения формата вывода данных и размера элементов (хотя существуют псевдонимы на основе наследуемых параметров). Содержимое текстового файла из предыдущего примера в данном случае будет выглядеть следующим образом:

```
$ od -tc floats-ints.dat                Вывод данных в виде ASCII-символов
0000000   H   e   l   l   o       W   o   r   l   d  \r
0000020                                  \n 277 343 201 223 343 202 223
0000040 343 201 253 343 201 241 343 201 257  \r
0000060                 \n  \n  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000100  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
*       Пропуск дубликатов строк (в качестве индикатора используется
        символ *)
0000460  \0 220    _ 001  \0 221    _ 001  \0 222   _ 001   \0 223    _ 001
0000500  \0  \0 310 257   G 200 310 257   G \0 311 257    G 200 311 257
0000520   G
0000521
```

8.4. Команды оболочки для работы с объектами System V IPC

Наиболее оптимальными инструментами для работы с объектами System V IPC являются команды `ipcs` и `ipcrm`. Команда `ipcs` представляет собой основной инструмент для оперирования любыми из рассматриваемых нами объектов System V IPC. `ipcrm` применяется для удаления объектов System V IPC, которые могли остаться после завершения или краха процесса.

8.4.1. Разделяемая память System V

В случае с объектами разделяемой памяти команда `ipcs` позволяет выводить определяемые приложением ключи (если таковые имеются), а также определяемые системой идентификаторы ключей. Она также дает возможность выявлять наличие процессов, подключенных к объекту разделяемой памяти. Поскольку система X Window активно эксплуатирует разделяемую память System V IPC, если вы произведете проверку своей системы, то с большой долей вероятности обнаружите немало используемых объектов разделяемой памяти. Например:

```
$ ipcs -m -m    -m указывает на то, что на экран должны быть выведены только
                объекты разделяемой памяти
------ Shared Memory Segments --------
key        shmid      owner      perms   bytes      nattch     status
0x00000000 163840     john       600     196608     2          dest
0x66c8f395 32769      john       600     1          0
0x237378db 65538      john       600     1          0
0x5190ec46 98307      john       600     1          0
0x31c16fd1 131076     john       600     1          0
0x00000000 196613     john       600     393216     2          dest
0x00000000 229382     john       600     393216     2          dest
0x00000000 262151     john       600     196608     2          dest
0x00000000 294920     john       600     393216     2          dest
0x00000000 327689     john       600     393216     2          dest
0x00000000 360458     john       600     196608     2          dest
0x00000000 393227     john       600     393216     2          dest
0x00000000 425996     john       600     196608     2          dest
0x00000000 884749     john       600     12288      2          dest
0x00000000 2031630    john       600     393216     2          dest
0x00000000 2064399    john       600     196608     2          dest
0x00000000 2097168    john       600     16384      2          dest
```

В данном примере можно наблюдать и частные, и общие объекты разделяемой памяти. Частные объекты имеют в колонке `key` нулевые значения, хотя при этом каждый объект обладает уникальным `shmid`-идентификатором. Колонка `nattch` содержит сведения о количестве процессов, которые в текущий момент подключены к объекту разделяемой памяти. Если команду `ipcs` применить в сочетании с параметром `-p`, то итоговый вывод будет содержать идентификатор процесса создателя объекта и такой же идентификатор самого последнего процесса, который подключался к каждому из объектов разделяемой памяти или отключался от него. Например:

```
$ ipcs -m -p
------ Shared Memory Creator/Last-op --------
shmid      owner      cpid       lpid
163840     john       2790       2906
32769      john       2788       0
65538      john       2788       0
98307      john       2788       0
131076     john       2788       0
196613     john       2897       2754
229382     john       2899       2921
262151     john       2899       2921
294920     john       2907       2754
327689     john       2921       2923
360458     john       2893       2754
393227     john       2893       2754
425996     john       2921       2754
884749     john       2893       2754
2031630    john       8961       9392
2064399    john       8961       9392
2097168    john       8961       9392
```

Заголовком колонки идентификатора процесса создателя объекта разделяемой памяти является `cpid`, а заголовком колонки идентификатора процесса, который последним подключался к объекту разделяемой памяти или отключался от него, — `lpid`. Можно подумать, что если в колонке `nattch` стоит значение 2, то именно столько процессов в настоящий момент подключено к объекту разделяемой памяти. Не стоит забывать, что нет никаких гарантий того, что создатель объекта будет по-прежнему оставаться подключенным (или выполняющимся). Аналогичная ситуация может сложиться для процесса, который последним подключался к данному объекту или отключался от него.

Если в колонке `nattch` присутствует значение `0` и ни один из процессов, упомянутых в выводе команды `ipcs`, не выполняется, *можно* безопасно удалить соответствующий объект при помощи команды `icprm`. Однако инструмент `ipcs` не позволяет получить ответ на вопрос: «Какой именно процесс подключен к объекту разделяемой памяти *в настоящий момент*?» Ответить на него поможет грубый поиск при помощи команды `lsof`. Рассмотрим пример:

```
$ ipcs -m
------ Shared Memory Segments --------
key        shmid      owner      perms     bytes     nattch     status
...
0xdeadbeef 2752529    john      666       1048576   3
     К данному объекту подключено всего три процесса, но каких именно?
$ ipcs -m -p
------ Shared Memory Creator/Last-op --------
shmid      owner      cpid       lpid
2752529    john       10155      10160
     Нашими «подозреваемыми» являются процессы 10155 и 10160.
     Для того чтобы окончательно разобраться, воспользуемся командой lsof.
```

```
$ lsof | head -1 ; lsof | grep 2752529
COMMAND     PID    USER   FD   TYPE   DEVICE   SIZE   NODE NAME
sysv-shm  10155    john  DEL    REG      0,7        2752529 /SYSVdeadbeef
sysv-clie 10158    john  DEL    REG      0,7        2752529 /SYSVdeadbeef
sysv-clie 10160    john  DEL    REG      0,7        2752529 /SYSVdeadbeef
```

Дополнительные сведения, касающиеся объекта разделяемой памяти, можно извлечь, применив команду `ipcs` с параметром `-i`. Если вы решите, что объект разделяемой памяти System V можно безопасно удалить, то вам потребуется `shmid`-идентификатор этого объекта (не ключ). Рассмотрим пример:

```
$ ipcs -m -i 32769                          Поиск сведений об объекте
                                            разделяемой памяти с указанным
                                            shmid-идентификатором
Shared memory Segment shmid=32769
uid=500 gid=500 cuid=500      cgid=500
mode=0600       access_perms=0600
bytes=1 lpid=0  cpid=2406     nattch=0      Создателем является процесс 2406…
att_time=Not set
det_time=Not set
change_time=Sat Apr   8  15:48:24 2006

$ kill -0 2406                              Выполняется ли данный процесс
                                            в настоящий момент?
bash: kill: (2406) – No such process        Не выполняется

$ ipcrm -m 32769                            Удаление объекта разделяемой
                                            памяти
```

8.4.2. Очереди сообщений System V

Для просмотра любых очередей сообщений System V используется команда `ipcs` с параметром `-q`:

```
$ ipcs -q
------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0x00000000 131072     john       600        0            0
0x00000000 163841     john       600        0            0
0x00000000 196610     john       600        0            0
0x00000000 229379     john       600        132          1
```

Значения, приведенные в колонке `key`, являются определяемыми приложением ключами, в то время как значения в колонке `msqid` представляют собой определяемые системой ключи. Как вы уже могли догадаться, определяемые системой ключи уникальны. В приведенном примере все определяемые приложением ключи равны `0`, а это значит, что данные очереди сообщений были созданы с использованием ключа `IPC_PRIVATE`.

Одна из таких очередей (с `msqid`-идентификатором `229379`) содержит данные в колонках с заголовками `used-bytes` и `messages`. Это может быть признаком проблем, поскольку большинство приложений не позволяют сообщениям надолго

задерживаться в очередях. Здесь опять на помощь приходит команда ipcs с параметром -i:

```
$ ipcs -q -i 229379

Message Queue msqid=229379
uid=500 gid=500 cuid=500          cgid=500          mode=0600
cbytes=132      qbytes=16384      qnum=1     lspid=12641     lrpid=0
send_time=Sun Oct   22   15:25:53   2006
rcv_time=Not set
change_time=Sun Oct 22   15:25:53   2006
```

Обратите внимание на то, что поля lspid и lrpid содержат *идентификатор процесса последнего отправителя* и *идентификатор процесса последнего получателя* соответственно. Если вы решите, что необходимости в данной очереди больше нет, можете удалить ее, введя идентификатор этой очереди:

```
$ ipcrm -q 229379
```

Поскольку команда ipcrm применима не только к очередям сообщений, необходимо указать системный идентификатор объекта наряду с тем фактом, что он является очередью сообщений, воспользовавшись для этого параметром -q.

8.4.3. Семафоры System V

Как и в случае с очередями сообщений и разделяемой памятью, команда ipcs также может использоваться для просмотра семафоров в системе, если ввести ее с параметром -s:

```
$ ipcs -s
------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x6100f981 360448     john       600        1
```

Ранее уже отмечалось, что семафоры System V объявляются как массивы. В колонке nsems вы можете увидеть длину массива. Данный вывод очень напоминает тот, что генерируется для очередей сообщений. Семафор также можно удалить при помощи команды ipcrm:

```
$ ipcrm -s 360448
```

При удалении семафора необходимо указывать его системный идентификатор (не ключ). Дополнительные сведения о семафоре можно извлечь, воспользовавшись параметром -i:

```
$ ipcs -s -i 393216

Semaphore Array semid=393216
uid=500  gid=500         cuid=500         cgid=500
mode=0600, access_perms=0600
nsems = 1
otime = Tue May  9 22:23:30 2006
```

```
ctime = Tue May  9 22:22:23 2006
semnum     value      ncount     zcount     pid
0          3          0          1          32578
```

Данный вывод практически аналогичен тому, что генерируется командой `stat` для файлов, за тем исключением, что здесь присутствуют дополнительные сведения, специфичные для семафора. Колонка `ncount` содержит показатель количества процессов, которые блокируются и ожидают увеличения значения семафора. В колонке `zcount` указано количество процессов, которые блокируются и ожидают, когда значение семафора станет равным нулю. Колонка `pid` отражает идентификатор процесса, который последним *завершил* операцию, касающуюся семафора; идентификаторы процессов, *ожидающих* семафор, не приводятся.

Команда `ps` может использоваться для идентификации процессов, ожидающих семафор. Параметр форматирования `wchan` демонстрирует, какая системная функция вызывает блокировку процесса. Для процесса, который блокируется в ожидании семафора, можно ввести команду

```
$ ps -o wchan -p 32746
WCHAN
semtimedop
```

Системный вызов `semtimedop` используется в операциях с семафорами. К сожалению, способа, позволяющего определить, какой именно процесс находится в ожидании определенного семафора, не существует. Карты процессов, а также дескрипторы файлов не дают возможности выявлять идентификаторы семафоров.

8.5. Инструменты для работы с объектами POSIX IPC

POSIX IPC использует дескрипторы файлов для каждого объекта. Стандартом POSIX предполагается, что каждый дескриптор файла обладает ассоциированным с ним файлом или устройством, при этом в операционной системе Linux данный подход дополнен использованием специальных файловых систем при реализации механизма межпроцессного взаимодействия IPC. Поскольку каждый IPC-объект может быть отслежен с использованием плоского файла, инструментарий, используемый для работы с плоскими файлами, зачастую оказывается вполне пригодным для оперирования объектами POSIX IPC.

8.5.1. Разделяемая память POSIX

Для работы с разделяемой памятью POSIX существует ряд специальных инструментов. В Linux объекты разделяемой памяти POSIX располагаются в псевдофайловой системе `tmpfs`, которая обычно монтируется в `/dev/shm`. Это означает, что для отладки данных объектов вы можете использовать любые инструменты, применяемые для обработки файлов. Практически все сказанное в разделе, посвященном работе с открытыми файлами, справедливо и в данном случае. Единственная

разница заключается в том, что все файлы, которыми вы будете оперировать, сосредоточены в одной файловой системе.

В силу особенностей реализации операционной системы Linux объекты разделяемой памяти можно создавать и использовать только посредством стандартных системных вызовов: `open`, `close`, `mmap`, `unlink` и т. д. Просто имейте в виду, что данные вызовы являются специфичными для Linux. Стандартом POSIX данный подход поддерживается, но не навязывается, поэтому в случае с переносимым программным кодом необходимо придерживаться использования системных вызовов для работы с разделяемой памятью POSIX.

8.5.2. Очереди сообщений POSIX

Для демонстрации очередей сообщений POSIX в Linux используется псевдофайловая система `mqueue`. К сожалению, стандартной точки монтирования данной системы не существует. Если вы захотите осуществлять отладку очередей сообщений POSIX из оболочки, то вам придется вручную заниматься ее монтированием. Так, например, для того, чтобы смонтировать данную файловую систему в каталог с именем `/mnt/mqs`, потребуется следующая команда:

```
$ mkdir /mnt/mqs
$ mount -t mqueue none /mnt/mqs      Для того чтобы иметь возможность
                                     использовать команду mount, необходимо
                                     обладать правами корневого пользователя
                                     root
```

Когда файловая система будет смонтирована, перед вами предстанут элементы всех очередей сообщений POSIX в системе. Однако они не будут являться обычными файлами. Если вы выполните команду `cat` в отношении такого файла, то увидите не сообщения, а свойства очереди, например:

```
$ ls -l /mnt/mqs
total 0
-rw-------  1 john john 80 Apr  9  00:20 myq
$ cat /mnt/mqs/myq
QSIZE:6          NOTIFY:0      SIGNO:0      NOTIFY_PID:0
```

В поле `QSIZE` указано количество байтов в очереди. Если в нем ненулевое значение, это может быть признаком тупика или иных проблем. Поля `NOTIFY`, `SIGNO` и `NOTIFY_PID` используются функцией `mq_notify`, которая в книге не рассматривается.

Для того чтобы удалить очередь сообщений POSIX из системы при помощи оболочки, необходимо воспользоваться командой `rm` и убрать ее из файловой системы `mqueue` вводом соответствующего имени.

8.5.3. Семафоры POSIX

Именованные семафоры POSIX реализуются в Linux в виде файлов, размещаемых в файловой системе `tmpfs`, подобно объектам разделяемой памяти. В Linux, в отличие от API-интерфейса System V, отсутствуют какие-либо системные вызовы для создания семафоров. Семафоры реализуются преимущественно в пространстве

пользователя при помощи существующих системных вызовов. Это означает, что реализация в значительной степени определяется библиотекой реального времени GNU (`librt`), которая входит в состав пакета `glibc`.

К счастью, библиотека реального времени предоставляет ряд довольно предсказуемых альтернативных методов, которых весьма легко придерживаться. С выходом библиотеки `glibc` версии 2.3.5 именованные семафоры создаются в виде файлов, располагаемых в `/dev/shm`. Путь к семафору с именем `mysem` будет иметь вид `/dev/shm/sem.mysem`. Поскольку API-интерфейс POSIX оперирует дескрипторами файлов, используемые семафоры можно увидеть как открытые файлы в файловой системе `procfs`; из этого следует, что такие инструменты, как `lsof` и `fuser`, также смогут их «увидеть».

Счетчик семафора POSIX нельзя просмотреть напрямую. Тип `sem_t`, применяемый в GNU, не содержит полезных элементов данных, а только массив целых чисел. Однако разумно предположить, что среди этих данных присутствует счетчик семафора.

8.6. Инструменты для работы с сигналами

Полезным инструментом, используемым для отладки сигналов из оболочки, является команда `ps`, которая позволяет просматривать маску сигналов процесса наряду с ожидающими (необработанными) сигналами. Вы также можете узнать, у каких сигналов есть определяемые пользователем обработчики, а у каких — нет.

Как вы уже могли догадаться, для просмотра масок сигналов используется параметр `-o`:

```
$ ps -o pending,blocked,ignored,caught
        PENDING          BLOCKED          IGNORED           CAUGHT
0000000000000000 0000000000010000 0000000000384004 000000004b813efb
0000000000000000 0000000000000000 0000000000000000 0000000073d3fef9
```

В более «компактном» варианте данной команды используется синтаксис BSD, который является немного нестандартным, поскольку в нем не используется тире для обозначения аргументов. Тем не менее он легок в применении и генерирует более содержательный вывод:

```
$ ps s                          Обратите внимание на отсутствие тире перед s.
UID   PID   PENDING   BLOCKED   IGNORED    CAUGHT ...

500  6487  00000000  00000000  00384004  4b813efb ...
500  6549  00000000  00000000  00384004  4b813efb ...
500 12121  00000000  00010000  00384004  4b813efb ...
500 17027  00000000  00000000  00000000  08080002 ...
500 17814  00000000  00010000  00384004  4b813efb ...
500 17851  00000000  00000000  00000000  73d3fef9 ...
```

Четыре значения, приводимых для каждого из процессов, являются *масками*, хотя при этом ядро сохраняет только одну маску, которая показана в данном примере в колонке сигналов `BLOCKED`. Остальные маски извлекаются из данных, распо-

ложенных в системе. Каждая маска содержит 1 или 0 для каждого сигнала N в битовой позиции N-1:

- CAUGHT — сигналы, у которых нет обработчика по умолчанию;
- IGNORED — сигналы, которые явным образом игнорируются посредством команды signal(N,SIG_IGN);
- BLOCKED — сигналы, которые явным образом блокируются посредством команды sigprocmask;
- PENDING — сигналы, посланные процессу, но еще не обработанные.

Еще одним полезным инструментом для работы с сигналами является команда strace. Она показывает переходы из режима пользователя в режим ядра в текущем процессе наряду с системным вызовом или сигналом, который стал причиной этого перехода. Команда strace является весьма гибким инструментом, который, однако, имеет некоторые ограничения при выводе сведений о сигналах.

Команда strace позволяет узнать лишь время, когда происходит переход из режима пользователя в режим ядра. А это значит, что она может сообщить лишь, когда сигнал был доставлен, а не когда он был отправлен. Кроме того, сигналы, помещаемые в очередь, выглядят точно так же, как и обычные сигналы; сведения об отправителе в выводе команды strace отсутствуют.

8.7. Инструменты для работы с каналами (конвейерами) и сокетами

Предпочтительным инструментом пространства пользователя для отладки сокетов является команда netstat, которая в значительной степени полагается на сведения, содержащиеся в каталоге /proc/net. С каналами и FIFO дела обстоят немного сложнее, поскольку не существует какой-либо единой локации, в которой можно отследить их существование. Единственным индикатором существования канала или FIFO является каталог /proc/pid/fd процесса, который использует данный канал или FIFO.

8.7.1. Каналы и FIFO

В каталоге /proc/pid/fd каналы и FIFO приводятся в соответствии с номером индексного дескриптора. Вот пример выполняющейся программы, осуществляющей вызов функции pipe для создания пары файловых дескрипторов (один из них предназначен только для записи, второй — только для чтения):

```
$ ls -l !$
ls -l /proc/19991/fd
total 5
lrwx------ 1 john john 64 Apr 12 23:33 0 -> /dev/pts/4
lrwx------ 1 john john 64 Apr 12 23:33 1 -> /dev/pts/4
lrwx------ 1 john john 64 Apr 12 23:33 2 -> /dev/pts/4
lr-x------ 1 john john 64 Apr 12 23:33 3 -> pipe:[318960]
l-wx------ 1 john john 64 Apr 12 23:33 4 -> pipe:[318960]
```

В данном случае «файл» имеет имя `pipe:[318960]`, где `318960` — это номер индексного дескриптора канала. Обратите внимание: несмотря на то что функция `pipe` возвращает два файловых дескриптора, мы видим только один номер индексного дескриптора, который идентифицирует канал. Подробнее об индексных дескрипторах мы поговорим далее в этой главе.

Функция `lsof` может использоваться для отслеживания процессов наряду с каналами.

8.7.2. Сокеты

Наиболее полезными инструментами для отладки сокетов являются команды `netstat` и `lsof`. Инструмент `netstat` больше всего подходит для просмотра общей картины системного использования сокетов. Так, например, для того, чтобы увидеть все TCP-соединения в системе, потребуется следующая команда:

```
$ netstat --tcp -n
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address             Foreign Address          State
tcp        0     48 ::ffff:192.168.163.128:22   ::ffff:192.168.163.1:1344  ESTABLISHED
```

А вот вывод, генерируемый командой `lsof`:

```
$ lsof -n -i tcp
COMMAND    PID    USER   FD   TYPE DEVICE  SIZE NODE NAME
portmap   1853     rpc    4u  IPv4   4847        TCP *:sunrpc (LISTEN)
rpc.statd 1871 rpcuser    6u  IPv4   4881        TCP *:32769 (LISTEN)
smbd      2120    root   20u  IPv4   5410        TCP *:microsoft-ds (LISTEN)
smbd      2120    root   21u  IPv4   5411        TCP *:netbios-ssn (LISTEN)
X         2371    root    1u  IPv6   6310        TCP *:x11 (LISTEN)
X         2371    root    3u  IPv4   6311        TCP *:x11 (LISTEN)
xinetd   20338    root    5u  IPv4 341172        TCP *:telnet (LISTEN)
sshd     23444    root    3u  IPv6 487790        TCP *:ssh (LISTEN)
sshd     23555    root    3u  IPv6 502673        ...
       TCP 192.168.163.128:ssh->192.168.163.1:1344 (ESTABLISHED)
sshd     23557    john    3u  IPv6 502673        ...
       TCP 192.168.163.128:ssh->192.168.163.1:1344 (ESTABLISHED)
```

Вывод команды `lsof` содержит `pid`-идентификаторы всех перечисленных процессов. Один и тот же сокет приводится дважды, поскольку два процесса `sshd` совместно используют один и тот же дескриптор файла. Обратите внимание на то, что вывод, выполненный командой `lsof`, по умолчанию включает сокеты прослушивания, в отличие от вывода, генерируемого по умолчанию командой `netstat`.

Команда `lsof` не выводит на экран сокеты, которые не принадлежат ни одному из процессов. Это TCP-сокеты, пребывающие в так называемом состоянии *ожидания (wait)*, которое наблюдается при закрытии сокетов. Например, когда процесс «умирает», его сокет может перейти в состояние `TIME_WAIT`. В этом случае вывод, выполненный командой `lsof`, не будет содержать сведений о данном сокете, так как он больше не принадлежит процессу. Вывод, генерируемый командой `netstat`, наоборот, будет включать эту информацию. Для просмотра всех TCP-сокетов необходимо добавить к команде `netstat` параметр `-tcp`:

```
$ netstat -n --tcp
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address     Foreign Address     State
tcp        0      0 127.0.0.1:60526   127.0.0.1:5000      TIME_WAIT
```

Если вы будете использовать упомянутые инструменты для просмотра сокетов, то имейте в виду, что каждый сокет, подобно файлу, имеет номер индексного дескриптора. Это касается как сетевых, так и локальных сокетов, однако наибольшую важность индексный дескриптор имеет именно для локальных сокетов, так как зачастую он является единственным уникальным идентификатором сокета.

При извлечении сведений о сервере с помощью команды `netstat` итоговый вывод можно ограничить сокетами прослушивания, задействовав параметр `-l`, а благодаря использованию параметра `-p` на экране будет отображена идентификационная информация о процессе:

```
$ netstat --unix -lp | grep /tmp/.ICE-unix/
unix 2 [ACC] STREAM LISTENING 7600 2408/gnome-session /tmp/.ICE-unix/2408
```

8.8. Использование индексных дескрипторов для идентификации файлов и объектов IPC

Linux поддерживает виртуальную файловую систему `vfs`, которая является общей для всех файловых систем. Она дает возможность работать с файловыми системами, не ассоциированными с физическими устройствами (например, `tmpfs` и `procfs`), обеспечивая при этом API-интерфейс для физических дисков. В результате этого виртуальные файлы неотличимы от файлов, располагающихся на дисковом накопителе.

Понятие *индексный дескриптор* происходит из терминологии файловых систем UNIX. Он обозначает структуру, сохраненную на диске, которая содержит учетные данные файлов — разрешения, касающиеся размеров файлов, и т. д. Каждый объект в файловой системе обладает уникальным индексным дескриптором, который представлен в пространстве пользователя в виде уникального целого числа. Говоря в общем, все обладающие файловым дескриптором в операционной системе Linux будут обладать также индексным дескриптором.

Номера индексных дескрипторов могут оказаться полезными для объектов, которые не имеют файловых имен, включая сетевые сокеты и каналы. Номера индексных дескрипторов уникальны в рамках файловой системы, однако нет никаких гарантий того, что они окажутся таковыми в разных файловых системах. Сетевые сокеты, в отличие от каналов, могут быть однозначно идентифицированы по своим номерам портов и IP-адресам. Для идентификации двух процессов, использующих один и тот же канал, необходимо сопоставить номера индексных дескрипторов.

Команда `lsof` позволяет просматривать номера индексных дескрипторов всех дескрипторов файлов, которые содержатся в итоговом выводе. Эти сведения для большинства файлов и прочих объектов отражаются в колонке `NODE`. Команда

netstat также выводит номера индексных дескрипторов, но только для сокетов доменов UNIX. Это вполне естественно, поскольку сокеты прослушивания доменов UNIX представлены в виде файлов, размещаемых на дисковом накопителе.

Подход к сетевым сокетам немного отличается от рассмотренного. В Linux сетевые сокеты обладают индексными дескрипторами, однако команды lsof и netstat (которые могут также выполняться не только в Linux, но и в других операционных системах) «притворяются», что на самом деле это не так. Вывод команды netstat не содержит номеров индексных дескрипторов сетевых сокетов, однако в выводе команды lsof эти сведения присутствуют в колонке DEVICE. Рассмотрим пример TCP-сокетов, открываемых демоном xinetd (для этого необходимо обладать правами корневого пользователя *root*):

```
$ lsof -i tcp -a -p $(pgrep xinetd)
COMMAND  PID USER   FD   TYPE DEVICE SIZE NODE NAME
xinetd  2838 root    5u  IPv4  28178       TCP *:telnet (LISTEN)
```

Здесь видно, что демон xinetd использует для прослушивания сокет telnet (порт 23). Несмотря на то что в колонке NODE содержится только одно слово TCP, в колонке DEVICE можно наблюдать номер индексного дескриптора. Также можно просматривать индексные дескрипторы сетевых сокетов, которые располагаются в различных местах файловой системы procfs, например:

```
$ ls -l /proc/$(pgrep xinetd)/fd
total 7
lr-x------  1 root root 64 Oct 22 22:24 0 -> /dev/null
lr-x------  1 root root 64 Oct 22 22:24 1 -> /dev/null
lr-x------  1 root root 64 Oct 22 22:24 2 -> /dev/null
lr-x------  1 root root 64 Oct 22 22:24 3 -> pipe:[28172]
l-wx------  1 root root 64 Oct 22 22:24 4 -> pipe:[28172]
lrwx------  1 root root 64 Oct 22 22:24 5 -> socket:[28178]
lrwx------  1 root root 64 Oct 22 22:24 7 -> socket:[28175]
```

8.9. Заключение

В данной главе мы изучили ряд инструментов и методик, применяемых при отладке различных механизмов межпроцессного взаимодействия IPC, включая плоские файлы. Несмотря на то что System V IPC предполагает использование специального инструментария, в случае с POSIX IPC для отладки применимы те же инструменты, которые используются для работы с плоскими файлами.

8.9.1. Инструментарий, использованный в этой главе

- ipcs, ipcrm — утилиты командной строки, используемые в случае с System V IPC.
- lsof, fuser — инструменты для просмотра открытых файлов и использования файловых дескрипторов.

- `ltrace` — средство для отслеживания вызовов функций, совершаемых процессом, при работе с разделяемыми объектами.
- `pmap` — позволяет выводить на экран удобочитаемые карты памяти процессов.
- `strace` — отслеживает использование процессом системных вызовов.

8.9.2. Веб-ссылки

- http://procps.sourceforge.net — домашняя страница `procps`, источник сведений о команде `pmap`.
- http://sourceforge.net/projects/strace — домашняя страница `strace`.

9 Настройка производительности

9.1. Введение

В этой главе мы поговорим о производительности как системы, так и приложений. Иногда возникают ситуации, когда выполнение «медленного» приложения невозможно значительно ускорить ни более производительным центральным процессором, ни более скоростной памятью или дисковым накопителем. После прочтения данной главы вы сможете отчетливо увидеть разницу между приложением, которое медленно работает вследствие своей неэффективности, и приложением, выполнение которого тормозится «медленной» аппаратной начинкой компьютера.

9.2. Производительность системы

Неоптимальная производительность системы сказывается на всех процессах. Пользователи такой системы хорошо чувствуют это: обновление окон «подтормаживает», соединения с серверами очень медленны и т. д. Многие инструменты позволяют определять текущий уровень производительности системы. К сожалению, некоторые из них отягощены таким количеством деталей, что у пользователей зачастую пропадает охота пользоваться ими. После того как вы усвоите основные аспекты производительности, эти детали не будут казаться вам столь непривычными.

9.2.1. Аспекты производительности, связанные с оперативной памятью

При консультации в службе технической поддержки часто можно услышать совет, от которого иногда бывает мало толку: «Вам необходимо увеличить объем оперативной памяти». Почему увеличение объема памяти должно помочь? Повышение уровня производительности вовсе не означает увеличения тактовой частоты центрального процессора или скорости работы системной шины компьютера. Так почему же наращивание объема памяти должно повысить производительность компьютера?

Увеличение объема памяти иногда помогает устранить проблемы с производительностью, однако никто не захочет выбрасывать деньги на ветер, покупая оперативную память лишь для того, чтобы впоследствии выяснить, что это не решит возникших проблем. Кроме того, на материнских платах может оказаться не слишком много разъемов для установки модулей оперативной памяти, поэтому ее покупка окажется напрасной и в этом случае. Для того чтобы заранее быть уверенным, что дополнительный объем оперативной памяти поможет устранить проблемы, нельзя строить свои предположения на одних догадках. Вам требуется понять, как система использует память, для того чтобы точно определить, связан ли низкий уровень производительности с недостаточным количеством оперативной памяти.

Ошибки страниц

Подсчет количества возникающих ошибок страниц может стать неплохим способом определить, насколько эффективно ваша система использует память. Как уже отмечалось ранее, ошибка страниц случается, когда центральный процессор запрашивает страницу памяти, которая отсутствует в оперативной памяти. Обычно это бывает следствием того, что данная страница не инициализирована или была перенесена из оперативной памяти на устройство подкачки.

Когда в системе возникает большое число ошибок страниц, это сказывается на всех процессах, которые в ней протекают. Данная ситуация может быть проиллюстрирована при помощи пары простых программ. Первой из них является программа, распределяющая память, которую она будет потреблять. Она обратится к памяти лишь один раз и в дальнейшем не будет ее использовать. Пример этой программы (она носит имя `hog`) можно увидеть в листинге 9.1. Она распределяет память при помощи `malloc`, модифицирует по одному байту на каждой странице, после чего погружается в «спячку».

Признаки возникновения ошибок страниц можно выявить при помощи команды GNU `time`. Как вы уже знаете, она не является аналогом встроенной команды `bash`-оболочки `time`. Для того чтобы воспользоваться GNU-версией этой программы, необходимо использовать знак обратного слеша, как показано далее:

```
$ \time ./hog 127
allocated 127 mb
Command terminated by signal 2
0.01user 0.32system 0:01.17elapsed 28%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+32626minor)pagefaults 0swaps
```

В данном примере я выделил интересующую нас информацию — количество незначительных ошибок страниц. К *значительным* ошибкам страниц относятся ошибки страниц, связанные с дисковыми операциями ввода/вывода, а к *незначительным* — все остальные.

Если использовать более понятную терминологию, то, когда программа `hog` запрашивает страницу памяти при помощи `malloc`, ядро генерирует отображения таблицы страниц для пространства пользователя процесса. На данном этапе распределение хранилища не производится — создается только отображение. Лишь после того, как процесс предпримет попытку впервые осуществить чтение со страницы

или запись на нее, произойдет *ошибка страницы*, в результате чего ядру придется искать хранилище для данной страницы. Этот механизм используется ядром для распределения новых хранилищ для процессов.

Листинг 9.1. hog.c: программа, которая распределяет память, но не использует ее в дальнейшем

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <stdlib.h>
 4 #include <unistd.h>
 5
 6 int main(int argc, char *argv[])
 7 {
 8     if (argc != 2)
 9         exit(0);
10
11     size_t mb = strtoul(argv[1], NULL, 0);
12
13     // Выполнить распределение памяти
14     size_t nbytes = mb * 0x100000;
15
16     char *ptr = (char *) malloc(nbytes);
17     if (ptr == NULL) {
18         perror("malloc");
19         exit(EXIT_FAILURE);
20     }
21
22     // Обратиться к памяти (также возможно применение calloc())
23     size_t i;
24     const size_t stride = sysconf(_SC_PAGE_SIZE);
25     for (i = 0; i < nbytes; i += stride) {
26         ptr[i] = 0;
27     }
28
29     printf("allocated %d mb\n", mb);
30     pause();
31     return 0;
32 }
```

Подкачка

До настоящего момента был доступен большой объем свободной памяти, из которой брались новые страницы, при этом латентность оставалась низкой. Однако если запустить указанный процесс снова и на этот раз свободной памяти окажется недостаточно для нормальной работы, то при возникновении каждой ошибки страницы ядру придется сохранять страницы в разделе подкачки. Оно, в частности, сохраняет в данный раздел самые последние из использовавшихся страниц (иногда сокращенно называемые LRU — от *least recently used*).

Перемещение страниц на диск и подкачка — это самое худшее, что может случиться, когда производительность падает до критического уровня. Когда системе приходится прибегать к перемещению страниц, операции загрузки или сохранения

в память, которые обычно длятся несколько наносекунд, могут занимать уже десятки миллисекунд и даже больше. Если системе необходимо считать несколько страниц с устройства подкачки или записать их на него, последствия не будут слишком серьезными; если же требуется переместить большое количество страниц, это может привести к значительному падению производительности системы.

Для того чтобы проиллюстрировать сказанное, вновь обратимся к программе hog, но на этот раз сделаем так, чтобы распределение памяти вызвало подкачку страниц:

```
$ free -m
             total       used       free     shared    buffers     cached
Mem:           250        109        140          0          0         11
-/+ buffers/cache:         97        152
Swap:          511          0        511
```

Обратите внимание на то, что вначале доступно 250 Мбайт памяти, 140 Мбайт свободны, при этом подкачка не используется.

```
$ \time ./hog 200
allocated 200 mb
Command terminated by signal 2
0.03user 0.61system 0:03.94elapsed 16%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+51313minor)pagefaults 0swaps
```

Заметьте, произошло 51 313 незначительных ошибок страниц и ни одной значительной!

```
$ free -m
             total       used       free     shared    buffers     cached
Mem:           250         48        201          0          0          6
-/+ buffers/cache:         42        208
Swap:          511         68        443
```

На диск подкачки записано 68 Мбайт страниц!

В данном примере видно, что программа hog стала причиной отправки на диск подкачки 68 Мбайт страниц памяти, при этом команда time рапортует о том, что не произошло ни одной значительной ошибки страниц. На самом деле эти сведения не совсем точны, однако их нельзя считать и ошибочными. Значительные ошибки страниц возникают, когда процесс запрашивает страницы, которые располагаются на диске. В данном случае страниц не существовало, следовательно, они отсутствовали на диске, в силу чего не были отнесены к значительным ошибкам страниц. Несмотря на то что процесс hog привел к тому, что системе пришлось записывать страницы на диск, данная операция на самом деле проводилась не системой. Непосредственная запись осуществлялась потоком kswapd.

Поток ядра kswapd берет на себя тяжелую работу по перемещению данных из памяти на диск. Только когда процесс, который обладает такими страницами памяти, попытается использовать их снова, произойдет ошибка страницы. Виновником такой ошибки будет процесс, запросивший страницу, при этом ошибка будет считаться *значительной*.

Команда top. Еще одним полезным инструментом, входящим в состав пакета procps, является команда top. Она использует библиотеку ncurses, которая максимально задействует возможности текстового терминала[1]. Формат итогового вывода команды top во многом напоминает формат вывода команды ps, за тем исключением, что вывод top содержит множество полей, не поддерживаемых инструментом ps. Поскольку вывод команды top периодически обновляется, пользователь обычно отводит одно окно для просмотра вывода top, а второе — для других целей. Вот пример типичного окна top:

```
top -  20:27:24 up  3:06,  4 users,  load average: 0.17, 0.27, 0.41
Tasks:  64 total,   3 running,  61 sleeping,   0 stopped,  0 zombie
Cpu(s):  6.8% us,  5.1% sy,  0.8% ni, 80.8% id,  6.2% wa, 0.3% hi,  0.0% si
Mem:    158600k total,    30736k used,   127864k free,    1796k buffers
Swap:   327672k total,    10616k used,   317056k free,   16252k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM   TIME+  COMMAND
    1 root      16   0  1744   96   72 R  0.0  0.1  0:00.89 init
    2 root      34  19     0    0    0 S  0.0  0.0  0:00.00 ksoftirqd/0
    3 root      RT   0     0    0    0 S  0.0  0.0  0:00.00 watchdog/0
    4 root      10  -5     0    0    0 S  0.0  0.0  0:00.10 events/0
    5 root      13  -5     0    0    0 S  0.0  0.0  0:00.03 khelper
    6 root      10  -5     0    0    0 S  0.0  0.0  0:00.00 kthread
    8 root      20  -5     0    0    0 S  0.0  0.0  0:00.00 kacpid
   61 root      10  -5     0    0    0 S  0.0  0.0  0:01.37 kblockd/0
   64 root      10  -5     0    0    0 S  0.0  0.0  0:00.00 khubd
...
```

Поскольку вывод содержит большое количество сведений, команда top разбивает его на четыре экрана с информацией, называемых *полевыми группами*. Нажав сочетание клавиш Shift+G, на основном экране вы увидите следующее приглашение:

```
Choose field group (1 – 4):
```

Как видно из приглашения, мы можем выбирать для просмотра любой из четырех экранов с информацией. Такой подход необходим из-за того, что все колонки не помещаются в один текстовый экран терминала. Одновременно просматривать все четыре экрана возможно, лишь отбросив некоторые из рядов, которые в них содержатся. Для этого необходимо нажать сочетание клавиш Shift+A, что позволит отобразить больше экранов, пожертвовав при этом некоторыми рядами. Например:

```
1:Def - 22:26:14 up  1:41,  3 users,  load average: 0.02, 0.01, 0.10
Tasks:  69 total,   1 running,  68 sleeping,   0 stopped,  0 zombie
Cpu(s):  0.7% us,  0.0% sy,  0.0% ni, 99.3% id,  0.0% wa, 0.0% hi,  0.0% si
Mem:    256292k total,   174760k used,    81532k free,    8860k buffers
Swap:   524280k total,    28120k used,   496160k free,  113608k cached
```

[1] Существует также GNOME-инструмент gtop, который в системе Fedora называется gnome-system-monitor. При использовании GUI-интерфейса выбор доступных параметров сужается.

```
1  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM   TIME+  COMMAND
  4026 john      15   0 42232  11m 6504 S  0.7  4.7  0:05.13 gnome-terminal
 30696 john      16   0  1936  956  764 R  0.3  0.4  0:00.01 top
  3583 john      15   0  7056  660  396 S  0.0  0.3  0:04.98 sshd
2  PID  PPID    TIME+  %CPU %MEM  PR  NI S  VIRT SWAP RES  UID COMMAND
 30696  4034   0:00.01  0.3  0.4  16   0 R  1936  980 956  500 top
  4054  4026   0:00.03  0.0  0.3  16   0 S  4276 3600 676  500 bash
  4034  4026   0:00.09  0.0  0.3  15   0 S  4276 3484 792  500 bash
  4033  4026   0:00.00  0.0  0.1  16   0 S  2068 1920 148  500 gnome-pty-helpe
3  PID %MEM  VIRT SWAP  RES CODE DATA  SHR nFLT nDRT S PR  NI %CPU COMMAND
  4026  4.7 42232  29m  11m  256  17m 6504  542    0 S 15   0  0.7 gnome-termin
  4030  0.5  4576 3248 1328   48 1604  896   47    0 S 16   0  0.0 gconfd-2
 30696  0.4  1936  980  956   48  268  764    0    0 R 16   0  0.3 top
  3588  0.3  4276 3460  816  580  260  624   10    0 S 16   0  0.0 bash
4  PID  PPID  UID USER     RUSER    TTY          TIME+ %CPU %MEM S COMMAND
  4026  3588  500 john     john     pts/1      0:05.13 0.7  4.7 S gnome-termina
  4030     1  500 john     john     pts/1      0:00.32 0.0  0.5 S gconfd-2
 30696  4034  500 john     john     pts/2      0:00.01 0.3  0.4 R top
  3588  3583  500 john     john     pts/1      0:00.29 0.0  0.3 S bash
```

Использование команды top для отслеживания hog-процессов. Для вывода сведений, касающихся статистики времени, а также получения большего уровня контроля над поведением нам потребуется еще одна программа, схожая с приложением `hog.c` из листинга 9.1. Пример такой программы (она называется `son-of-hog.c`) приведен в листинге 9.2. В данном случае я прибегну к парочке хитростей, для того чтобы вы могли как можно лучше рассмотреть инструмент `top` в действии. Сначала нам потребуется сделать следующее:

```
$ cc -O2 -o son-of-hog son-of-hog.c -lrt  librt требуется для clock_gettime
$ ln -s son-of-hog hog-a                 Вводим два разных имени для
                                         процессов, сведения о которых
                                         будем просматривать с помощью top
$ ln -s son-of-hog hog-b
```

Теперь переходим к более сложному этапу. Нам необходимо «опустошить» раздел подкачки, для того чтобы лучше увидеть картину происходящего. Воспользуемся командами `swapon` и `swapoff`:

```
$ free -m
             total       used       free     shared    buffers     cached
Mem:           250        246          3          0         14        165
-/+ buffers/cache:         66        183
Swap:          511          4        507
                      Используется 4 Мбайт раздела подкачки

$ sudo swapoff -a  Отключение использования разделов подкачки (необходимо
                   наличие прав корневого пользователя root); перенос всех
                   подкачанных страниц в RAM
$ sudo swapon -a   Повторное включение использования всех разделов подкачки
$ free -m
             total       used       free     shared    buffers     cached
Mem:           250        246          3          0         14        160
-/+ buffers/cache:         71        178
Swap:          511          0        511
```

В случае с данной системой около 178 Мбайт оперативной памяти могут использоваться без необходимости в подкачке страниц. Об этом свидетельствует значение, которое содержится в ряду +/- buffers/cache вывода команды free. Буферы и кэш представляют собой хранилище, которое может быть регенерировано без отправки его содержимого в раздел подкачки[1]. Далее нам необходимо дать указание процессу hog-a использовать 150 Мбайт памяти, чего должно хватить для предотвращения подкачки страниц:

```
$ ./hog-a 150 &
[1] 30825
$ touched 150 mb; in 0.361459 sec
$ free -m
             total       used       free     shared    buffers     cached
Mem:           250        246          3          0         11         17
-/+ buffers/cache:        217         32
Swap:          511          0        511
```

Обратите внимание на то, что мы смогли воспользоваться 150 Мбайт страниц памяти за приемлемый промежуток времени (около 361 мс). Прибегать к подкачке страниц не потребовалось, однако в кэше осталось много данных, и нам необходимо произвести регенерацию. Отправим процессу сигнал SIGUSR1, в результате чего он выйдет из «спячки» и еще раз обратится к своим страницам памяти:

```
$ kill -USR1 %1
$ touched 150 mb; in 0.009929 sec
```

Заметьте, что аналогичная процедура заняла на этот раз лишь 9 мс! При повторном обращении к буферу все страницы уже располагаются в оперативной памяти, то есть повода для возникновения ошибок страниц нет. Теперь запустим еще один процесс hog и посмотрим, что произойдет:

```
$ ./hog-b 150 &
[2] 30830
$ touched 150 mb; in 5.013068 sec
$ free -m
             total       used       free     shared    buffers     cached
Mem:           250        246          3          0          0         10
-/+ buffers/cache:        235         14
Swap:          511        136        375
```

Только посмотрите, как меняется картина, когда задействуется подкачка страниц! То, что прежде продолжалось 361 мс, теперь длится более 5 с. Это вполне ожидаемо, поскольку нам было известно, что процессу hob-b потребуется переместить на диск большое количество страниц процесса hog-a, для того чтобы высвободить память. Иначе говоря, вывод команды free свидетельствует о том, что процесс hob-b форсировал перемещение на диск 136 Мбайт страниц. Неудивительно, что на это потребовалось так много времени! Второй заход должен пройти намного быстрее:

---

[1] Это в идеале. На самом деле существует несколько сдерживающих факторов, однако такое вполне возможно.

```
$ pkill -USR1 hog-b
touched 150 mb; in 0.019061 sec
```

Нет ничего удивительного в том, что итоговый результат весьма близок к тому, который наблюдался при втором запуске процесса hog-a. Теперь посмотрим, что на все это «скажет» команда top. Введем ее в сочетании с параметром -p, для того чтобы в выводе отобразились только процессы hog:

```
$ top -p $(pgrep hog-a) -p $(pgrep hog-b)
```

Воспользовавшись сочетанием клавиш Shift+A, мы можем вывести на экран компьютера сразу все четыре окна. Итоговый вывод будет иметь следующий вид:

```
1:Def - 23:21:16 up  2:36,  2 users,  load average: 0.00, 0.02, 0.01
Tasks:   2 total,   0 running,   2 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0% us,  0.0% sy,  0.0% ni, 100.0% id,  0.0% wa,  0.0% hi, 0.0% si
Mem:    256292k total,   252048k used,     4244k free,      692k buffers
Swap:   524280k total,   139760k used,   384520k free,    11908k cached

1  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 30825 john      16   0  151m  63m  172 S  0.0 25.3   0:00.36 hog-a
 30830 john      16   0  151m 150m  296 S  0.0 60.1   0:00.57 hog-b
2  PID  PPID    TIME+  %CPU %MEM  PR  NI S  VIRT SWAP  RES  UID COMMAND
 30830  4054   0:00.57  0.0 60.1  16   0 S  151m 1160 150m  500 hog-b
 30825  4054   0:00.36  0.0 25.3  16   0 S  151m  88m  63m  500 hog-a
3  PID %MEM  VIRT SWAP  RES CODE DATA  SHR nFLT nDRT S  PR  NI %CPU COMMAND
 30830 60.1  151m 1160 150m    4 150m  296    9    0 S  16   0  0.0 hog-b
 30825 25.3  151m  88m  63m    4 150m  172    2    0 S  16   0  0.0 hog-a
4  PID  PPID  UID USER     RUSER    TTY         TIME+  %CPU %MEM S COMMAND
 30830  4054  500 john     john     pts/3      0:00.57  0.0 60.1 S hog-b
 30825  4054  500 john     john     pts/3      0:00.36  0.0 25.3 S hog-a
```

Обратите внимание: что оба процесса потребляют по 151 Мбайт виртуальной памяти, в чем можно убедиться, обратившись к колонке VIRT на экране 1. После переноса на диск 130 Мбайт страниц в обоих процессах hog количество значительных ошибок страниц менее 10, о чем свидетельствуют значения в колонке nFLT на экране 3. Колонка RES (экраны 1 и 2) содержит показатель количества пространства, отводимого процессам в оперативной памяти (то есть объем доступной им резидентной памяти). Здесь также есть колонка SWAP, из которой можно узнать, какой объем раздела подкачки использует соответствующий процесс.

Листинг 9.2. son-of-hog.c: модифицированная программа hog.c

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <stdlib.h>
 4 #include <signal.h>
 5 #include <time.h>
 6 #include <unistd.h>
 7 #include <sys/time.h>
 8
 9 void handler(int sig)
10 {
```

```
11      // Не выполняет никаких действий
12  }
13
14      // преобразовать структуру timespec в двойную для более удобного
        // использования
15  #define TIMESPEC2FLOAT(tv) ((double) (tv).tv_sec + (double) (tv).tv_nsec * 1e-9)
16
17  int main(int argc, char *argv[])
18  {
19      if (argc != 2)
20          exit(0);
21
22      // Фиктивный обработчик сигналов
23      signal(SIGUSR1, handler);
24
25      size_t mb = strtoul(argv[1], NULL, 0);
26
27      // Выполнить распределение памяти
28      size_t nbytes = mb * 0x100000;
29      char *ptr = (char *) malloc(nbytes);
30      if (ptr == NULL) {
31          perror("malloc");
32          exit(EXIT_FAILURE);
33      }
34
35      int val = 0;
36      const size_t stride = sysconf(_SC_PAGE_SIZE);
37
38      // Обращение к памяти происходит при каждом цикле, после чего
        // делается пауза и ожидается сигнал SIGUSR1
39      while (1) {
40          int i;
41          struct timespec t1, t2;
42
43       // t1 — когда мы начинаем использовать память
44          clock_gettime(CLOCK_REALTIME, &t1);
45
46       // Используется только по одному байту каждой страницы!
47          for (i = 0; i < nbytes; i += stride) {
48              ptr[i] = val;
49          }
50          val++;
51
52       // t2 — когда мы заканчиваем использовать память
53          clock_gettime(CLOCK_REALTIME, &t2);
54
55          printf("touched %d mb; in %.6f sec\n", mb,
56                 TIMESPEC2FLOAT(t2) - TIMESPEC2FLOAT(t1));
57
58       // Ожидать сигнал
59          pause();
```

```
60      }
61      return 0;
62 }
```

В качестве завершающей иллюстрации того, как процессы могут вызывать взаимное снижение скорости выполнения, давайте рассмотрим, как чередующиеся сигналы между процессами hog-a и hog-b могут нарочно вынудить систему прибегнуть к операциям записи на дисковый накопитель. Ситуацию, при которой система начинает уделять больше времени перемещению страниц на диск, чем выполнению пользовательского кода, мы называем *избыточным использованием памяти*. Применив команды, приведенные далее, вы сможете увидеть действие данного механизма в окне top, которое здесь не приводится:

```
$ pkill -USR1 hog-a                    Процесс hog-b был последним, которому
$ touched 150 mb; in 14.286939 sec     мы посылали сигнал
$ pkill -USR1 hog-                     Теперь процесс hog-b по большей части
$ touched 150 mb; in 16.731990 sec     "располагается" на диске
pkill -USR1 hog-a
$ touched 150 mb; in 16.944799 sec
```

Раздел, посвященный подкачке, я завершу рассмотрением вещей в перспективе. Каждый процесс hog использует по 150 Мбайт памяти, однако при этом задействует только 1 байт каждой страницы. Приведенный пример выполнялся на компьютере на базе процессора Pentium 4, размер страниц равнялся 4 Кбайт, то есть всего было 38 400 страниц. Иначе говоря, на модификацию 37 Кбайт памяти ушло около 17 с. Скорость функционирования памяти в данном примере не имеет решающего значения — время выполнения каждой операции определяется исключительно скоростью работы устройства подкачки.

Если вы уверены, что проблемы с производительностью вызваны тем, что система часто прибегает к подкачке страниц, то наиболее вероятным выходом из данной ситуации станет увеличение объема оперативной памяти. Если вы занимаетесь написанием приложения, которое может слишком часто инициировать подкачку страниц, то вам, скорее всего, стоит переработать программный код таким образом, чтобы обеспечить более эффективное использование оперативной памяти вместо наращивания ее объема. Воспользовавшись инструментами, рассмотренными в данном разделе, вы сможете выбрать правильное направление действий.

9.2.2. Использование центрального процессора и конкуренция за ресурсы шины

В предыдущем разделе мы рассмотрели аспекты подкачки страниц, которая инициируется, когда процессы начинают соперничать за обладание ограниченными ресурсами оперативной памяти. В системе имеются и другие «дефицитные» ресурсы, за которые могут бороться процессы. Одним из них является пропускная способность шины.

На рис. 9.1 приведена упрощенная схема шины материнской платы типичного компьютера, которая поддерживает шину PCI Express (PCIe). *Фронтальная шина*

(*Frontside Bus*, FSB) — это точка входа для всех данных, которые поступают в центральный процессор и выходят из него. Оперативная память может функционировать как в одноканальном, так и в многоканальном режиме, поскольку доступ к ней осуществляется через центральный процессор или периферийные компоненты, а также (в некоторых системах) через видеоконтроллер. Видеоконтроллеры часто снабжаются достаточным объемом памяти и собственной высокоскоростной шиной (PCI Express, или AGP), благодаря чему им нет необходимости конкурировать за обладание ресурсами шины оперативной памяти.

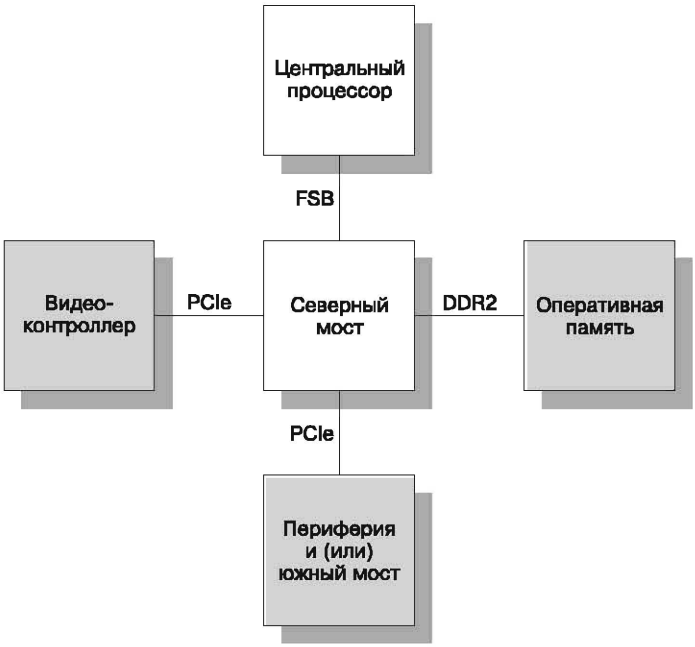


Рис. 9.1. Пример конструкции типичной материнской платы компьютера с поддержкой процессоров Intel и шины PCI Express

Многопроцессорная обработка и FSB-шина

Скорость FSB-шины является решающим фактором, от которого зависит производительность компьютера, поскольку в современных системах частота ее функционирования оказывается значительно меньше тактовой частоты центрального процессора. Скорость работы FSB-шины определяет максимальную быстроту функционирования подсистемы ввода/вывода компьютера.

С появлением многопроцессорных систем FSB-шина превратилась в «узкое место», в котором в значительной степени тормозилась их производительность. Устройство материнской платы типичной многопроцессорной системы аналогично устройству материнской платы, приведенной на рис. 9.1, за тем исключением, что блок, помеченный как центральный процессор *(CPU)*, содержит не один, а два или более процессора, которые совместно используют одну FSB-шину. Это означает, что вместо одного быстрого центрального процессора, за которым не поспевает более медлен-

ная FSB-шина, мы имеем целых два. Таким образом, с увеличением количества процессоров в системе конкуренция за ресурсы FSB-шины еще более обостряется.

Подобный тип вычислительной системы называется *компьютером с поддержкой симметричной многопроцессорной обработки SMP* (Symmetric Multiprocessing). Раньше такая функциональность была доступна лишь в высокопроизводительных серверах и рабочих станциях. Операционная система Linux обрела поддержку симметричной многопроцессорной обработки SMP с выходом версии 2.0. Совсем недавно многоядерные процессоры стали доступны для сегмента настольных компьютеров, что позволило расширить круг пользователей, которые могут воспользоваться преимуществами SMP-обработки. Компьютер, в котором установлен многоядерный центральный процессор, функционально идентичен компьютеру с поддержкой симметричной многопроцессорной обработки SMP, за исключением того, что в нем процессорные ядра располагаются не в отдельных корпусах, а в одном.

Определить, как процесс использует FSB-шину, можно путем подсчета количества ошибок страниц. Однако это не даст полной картины. Процесс использует шину памяти, когда пытается осуществить чтение или запись в область памяти, которая отсутствует в кэше, что зачастую приводит не к возникновению ошибки страницы, а скорее к *неудачному обращению к кэшу*.

Использование центрального процессора и эффективность

Использование ресурсов центрального процессора выражается в процентах от времени, которое он тратит на выполнение программного кода. Процессор, который загружен на 100 %, осуществляет выполнение программ постоянно. Пока система включена, процессор, естественно, всегда будет занят выполнением программного кода. Каждая процессорная архитектура поддерживает собственную функцию `cpu_idle`, которая вызывается операционной системой Linux, когда планировщик не может отыскать следующий процесс, который необходимо запустить. Использование ресурсов процессора может выражаться в виде количества времени, которое он затрачивает на выполнение программного кода, не связанного с функцией `cpu_idle`.

Ранее вы уже видели, что операции, связанные с модификацией нескольких килобайт оперативной памяти, могут протекать с совершенно разной скоростью. В каждом случае одни и те же инструкции выполняются с различной быстротой. В каждом случае загрузка процессора составляет 100 %. Однако одного уровня нагрузки на процессор недостаточно для того, чтобы охарактеризовать *эффективность* процесса или системы.

Концепцию эффективности использования ресурсов центрального процессора понять несложно. С подобными ситуациями вы сталкиваетесь постоянно в реальной жизни. Вспомните, когда вы в последний раз ходили в магазин за продуктами. Даже если вы самый практичный человек в мире, все равно можете задержаться на какое-то время в длинной очереди у кассы. В любом случае вы тратите 100 % своего времени на достижение поставленной цели, однако ваша эффективность в значительной степени лежит вне рамок вашего контроля. Аналогом длинной очереди в магазине в компьютерной сфере является увеличение латентности выполнения инструкций, что может быть вызвано соперничеством за какие-либо ресурсы (например, с другим процессором или устройством) или же обусловлено «медленным» ресурсом (например, оперативной памятью).

Возможности планировщика ядра по выявлению неэффективных процессов довольно скромны. Как и в реальной жизни, вовсе не обязательно, что виной тому будет сам процесс, он сам является жертвой обстоятельств. При присваивании приоритетов планировщик в основном полагается на данные, касающиеся создаваемой на процессор нагрузки. Эффективный приоритет процессов, которые захватывают ресурсы процессора, понижается[1], в то время как так называемым *интерактивным процессам* (к ним относятся процессы, которые большинство времени своего существования проводят в ожидании) присваивается более высокий приоритет.

Именно здесь от нас, пользователей, потребуются некоторые усилия. В следующих разделах мы рассмотрим инструменты, которые помогают определять эффективность процессов. Процессоры компании Intel обладают богатым набором регистров для мониторинга производительности, которыми можно воспользоваться в аналогичных целях, при этом доступно лишь несколько инструментов, посредством которых их можно задействовать. В силу этих причин многие из имеющихся инструментов применимы только на архитектуре Intel.

9.2.3. Устройства и прерывания

Когда речь заходит об устройствах и производительности, вы, скорее всего, полагаете, что устройства являются независимыми от всей остальной системы, то есть они не оказывают влияния на процессы, которые их используют. Зачастую это действительно так, однако устройства могут стать источником «побочных эффектов», о которых вы не подозреваете.

Конкуренция за ресурсы шины

В большинстве стандартных современных компьютеров, независимо от лежащей в основе процессорной архитектуры, для подключения периферийных устройств применяется PCI-шина. Я буду использовать данную шину в качестве примера, однако все сказанное справедливо и для таких типов шин, как SBUS, ISA, VME и др. Все они имеют один общий признак — они являются параллельными шинами, а это означает, что подключенные к ним устройства используют одни и те же линии. Устройствам, которые хотят «пообщаться» с центральным процессором, необходимо согласовывать между собой время, когда они будут использовать соответствующую шину. Пропускная способность шины является фиксированной и распределяется между разными устройствами, которые к ней подключены. Подобно тому как вычислительные ядра в многопроцессорной системе конкурируют за одну общую FSB-шину, устройства, подключенные к периферийной шине, соперничают между собой за обладание ее ресурсами.

PCI-шина разбивается компьютером на сегменты, которые располагаются в виде древовидной структуры. Как показано на рис. 9.2, данные сегменты образуют иерархию, во главе которой находится северный мост. Два устройства, связанных с разными сегментами, не станут конкурировать между собой за пропускную способность в своих собственных сегментах шины. Если таким устройствам потребуется обратиться к центральному процессору или памяти, ареной их борьбы

[1] Если они не относятся к процессам реального времени.

за пропускную способность шины станет северный мост. Может показаться неправдоподобным, однако наиболее эффективное использование такой шинной схемы возможно в ситуациях, когда два устройства взаимодействуют между собой без вовлечения центрального процессора. В данном случае в зависимости от схематической выкладки шины конкуренции за ее пропускную способность может не возникнуть, поскольку каждый сегмент является отдельным.

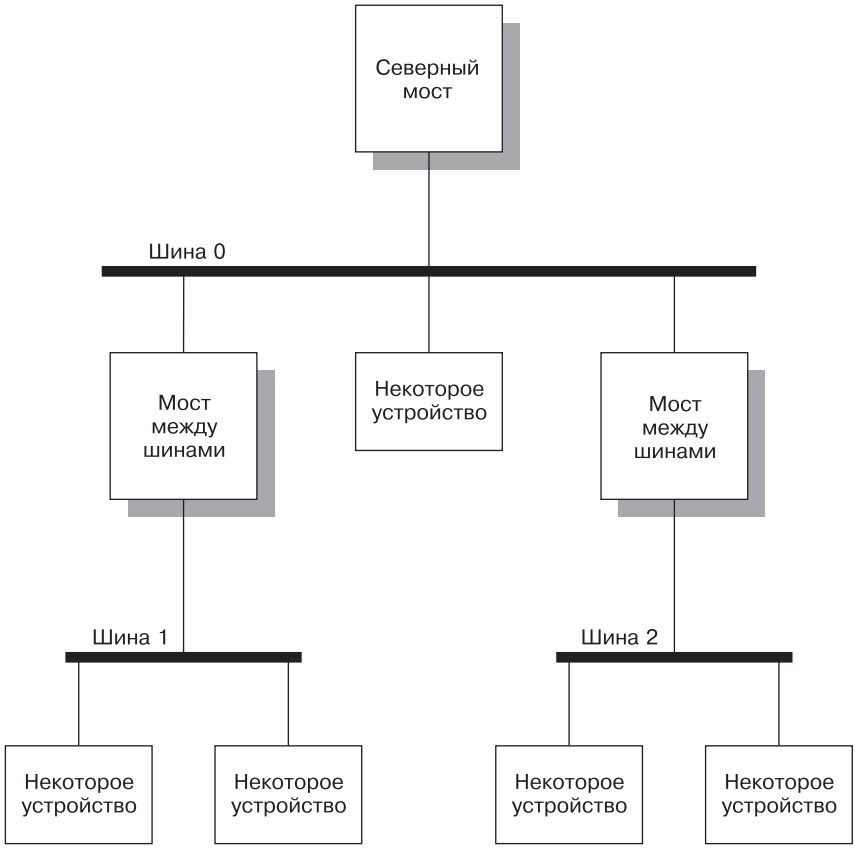


Рис. 9.2. Гипотетическая иерархия сегментов шины

С недавнего времени среди производителей комплектующих наметилась тенденция перехода от параллельных шин к высокоскоростным последовательным соединениям типа «точка — точка». Примерами могут послужить шина PCI Express от компании Intel и шина Hypertransport от компании AMD. Поскольку они реализуют соединения типа «точка — точка», конкуренция за ресурсы шины отсутствует. Здесь используется система разбивки шины на сегменты, похожая на ту, о которой мы говорили ранее. Каждое устройство можно рассматривать как имеющее выделенный сегмент, которым ему не нужно «делиться» с другими устройствами. Это так называемые архитектуры *коммутируемых комплексов*, которые основаны на тех же концепциях, что и обычная сеть Ethernet. В подобных конфигурациях каждый мост (включая

северный) функционирует скорее как коммутатор, чем как мост. Устройства с поддержкой шины PCI Express соперничают с другими устройствами на каждом мосту, который они «пересекают». При операциях, проводимых через северный мост (например, при обращении к памяти), устройству PCI Express приходится конкурировать за ресурсы со всеми другими такими же устройствами, установленными в системе.

Прерывания

Раньше, до появления интерфейсов USB и FireWire, для подключения периферийного оборудования вроде сканеров и устройств ввода двухмерных изображений требовался специальный адаптер, устанавливаемый в разъем ISA. У всех карт-адаптеров имелось одно общее свойство — им требовалась линия прерывания к центральному процессору. В устаревших компьютерах было доступно только 15 прерываний, многие из которых резервировались для системных функций, что делало их недоступными для других устройств. Выходом из данной ситуации стало использование одного прерывания двумя картами одновременно при условии, что это позволяет драйвер. Однако это отрицательно сказывается на производительности, поскольку операционной системе приходится поочередно вызывать каждый драйвер, пока один из них занят обработкой соответствующего прерывания. А что еще хуже, некоторые некачественно написанные драйверы не поддерживают работу с разделяемыми прерываниями, то есть если в вашей системе доступно мало прерываний, то вам не повезло.

Недостаток прерываний в компьютерах на базе типичной архитектуры подтолкнул разработчиков к созданию интерфейсов USB и FireWire, которые позволяют подключать периферийное оборудование без карт-адаптеров. Тем не менее некоторые из современных систем могут потребовать установки дополнительных карт-адаптеров, например сетевых карт. Определение прерываний для этих карт может оказаться непростым занятием.

Состояние прерываний и драйверов можно просмотреть в псевдофайле `/proc/interrupts`. Здесь можно точно узнать, для какого устройства назначено то или иное прерывание. В данном псевдофайле возле каждого прерывания отображается счетчик, по которому можно определить, сколько раз соответствующее прерывание подвергалось обработке с момента запуска системы, например:

```
$ cat /proc/interrupts
           CPU0
  0:      27037        XT-PIC  timer
  1:         10        XT-PIC  i8042
  2:          0        XT-PIC  cascade
  7:          1        XT-PIC  parport0
  8:          1        XT-PIC  rtc
  9:          0        XT-PIC  acpi
 11:       1184        XT-PIC  uhci_hcd:usb1, uhci_hcd:usb2, uhci_hcd:usb3, eth0
 12:          0        XT-PIC  VIA8233
 14:       6534        XT-PIC  ide0
 15:        269        XT-PIC  ide1
NMI:          0
LOC:      27008
ERR:          0
MIS:
```

Почти всегда вы будете видеть строку счетчика прерываний от таймера с наибольшими количественными показателями, которые увеличиваются с каждым интервалом хода системных часов. Как уже отмечалось, частота интервалов хода часов определяется при конфигурировании ядра. Если во время этой процедуры установить частоту интервалов хода 250, то значение данного счетчика будет увеличиваться 250 раз в секунду.

Сравнение контроллеров PIC и APIC

Архитектура прерываний представляет собой один из немногих осколков ISA-наследия, сохранившихся в современных персональных компьютерах. Большинство компьютерных чипсетов содержит реализацию устаревшего *программируемого контроллера прерываний 8259 PIC (Programmable Interrupt Controller)*, который встраивается в южный мост (см. рис. 9.1). Прерывания, обеспечиваемые данным контроллером, проходят через оба моста, прежде чем достигнут центрального процессора. Оба моста повторно пересекаются при подтверждении прерывания. Эти дополнительные «прыжки» через мосты увеличивают латентность прерываний, что может отрицательно сказаться на производительности системы.

Процессоры Pentium выпускаются со встроенным контроллером прерываний — *усовершенствованным программируемым контроллером прерываний APIC (Advanced Programmable Interrupt Controller)*, который дает возможность системным разработчикам обеспечивать для прерываний низколатентные «маршруты» к центральному процессору. Компания Intel также реализовала интерфейс для подключения внешнего APIC-контроллера, который необходим для многопроцессорных систем. Он будет присутствовать на материнской плате только в том случае, если она поддерживает установку двух и более процессоров. Встраиваемый APIC-контроллер обычно называется *локальным усовершенствованным программируемым контроллером прерываний LAPIC (Local Advanced Programmable Interrupt Controller)*, а внешний APIC-контроллер, как правило, именуется *усовершенствованным программируемым контроллером прерываний ввода/вывода I/O APIC (I/O Advanced Programmable Interrupt Controller)*.

Ни один из упомянутых APIC-контроллеров не требуется в однопроцессорной системе, при этом все они могут отключаться на программном уровне. В целях обеспечения совместимости поддержка APIC-контроллеров отключается на уровне BIOS во многих материнских платах, а вместо него задействуется устаревший контроллер 8259 PIC, встраиваемый в южный мост. В данном случае операционная система Linux будет использовать именно этот PIC-контроллер. В `/var/log/messages` вы можете увидеть следующее сообщение:

```
localhost kernel: Local APIC disabled by BIOS -- you can enable it with "lapic"
```

Как можно понять из сообщения, для того чтобы активировать LAPIC-контроллер, который был отключен на уровне BIOS, необходимо указать в загрузочной строке параметр `lapic`. В результате `/etc/grub.conf` должен иметь следующий вид:

```
Fedora Core (2.6.16np)
        root (hd1,0)
          kernel /vmlinuz-2.6.16np ro root=/dev/VolGroup00/LogVol00 rhgb quiet
lapic
        initrd /initrd-2.6.16np.img
```

Включение APIC-контроллера положительно скажется на латентности прерываний, что может иметь большое значение для приложений реального времени. Это также позволяет задействовать большее количество прерываний, чем при использовании устаревшего PIC-контроллера. Таким образом, активация APIC-контроллера означает, что установленным в системе картам не придется совместно использовать одно прерывание, о чем мы и поговорим в дальнейшем.

По мнению части пользователей, APIC-контроллер мешает работе некоторых драйверов. Если в вашем компьютере этот контроллер активирован на уровне BIOS, можете отключить его при помощи параметра `nolapic`.

Устройства и разъемы

Если в разъемах на материнской плате вашего компьютера имеются установленные адаптеры, следует иметь в виду несколько важных моментов. Как уже отмечалось, разъем часто определяет прерывание, которое будет использовать карта. Если ваша карта использует прерывание совместно с другим устройством, ее переустановка в другой разъем может помочь избежать этого. Совместное использование прерывания не является проблемой, однако это неоптимально с точки зрения производительности.

Параллельным шинам вроде PCI и PCI-X приходится разделять доступные циклы шины между установленными картами. «Медленная» карта в сочетании с «быстрой» будет замедлять работу последней.

Шина PCI Express уникальна тем, что использует соединение типа «точка — точка», что позволяет избежать многих проблем с качеством сигналов параллельных шин. Пропускная способность шины PCI Express выражается в *линиях*, которые имеют фиксированную пропускную способность (2,5 Гбайт/с). Физический размер разъема определяет максимальное число линий, доступных устанавливаемой в него карте.

Вам необходимо помнить, что все разъемы имеют разные размеры. Особенности конфигурации уникальны для каждой материнской платы, однако производители качественных продуктов всегда заботятся о наличии документации, к которой может обратиться заинтересованный пользователь. Никогда не стоит пренебрегать чтением руководства.

Инструменты для работы с разъемами и устройствами

Весьма полезным инструментом для определения конфигурации разъема является утилита `lspci`, позволяющая просматривать сведения о сегментах общей шины, которые совместно используются более чем одним устройством. Иногда выясняется, что определенный разъем использует сегмент общей шины совместно с устройством, распаянным на материнской плате. В данном случае пропускная способность шины может снижаться по вине пользователя, который не будет даже подозревать об этом.

Без использования каких-либо параметров команда `lspci` выводит перечень устройств, задействующих PCI-шину. Каждое PCI-устройство имеет идентификатор производителя и идентификатор устройства. Идентификатор производителя представляет собой 16-битное число, по которому можно определить компанию, изготовившую устройство. В состав инструмента `lspci` входит таблица производителей и устройств, благодаря которой он может выводить точную информацию об устройствах в удобочитаемом виде.

PCI является параллельной шиной, а это означает, что устанавливаемым картам приходится совместно использовать общий набор сигналов. Номер шины можно увидеть в поле bus. PCI позволяет использовать несколько сегментов шин посредством мостов. Шина 0 является наиболее «близкой» к процессору (северный мост). Каждый мост генерирует новый сегмент шины с более высоким номером.

Несмотря на то что инструмент lspci не позволяет узнать скорость работы устройства, подключенного к параллельной PCI-шине, с его помощью можно определить скорость устройств PCI Express. Для того чтобы просмотреть эти сведения, необходимо воспользоваться параметром -vv (двойным). В результате будет выведена информация, из которой можно узнать о скоростных возможностях карты и ее текущих настройках.

9.2.4. Инструменты для выявления причин снижения производительности системы

Для отслеживания процессов, вызывающих снижение производительности системы, существует ряд инструментов. В одном из предыдущих разделов при рассмотрении некоторых детализированных примеров использовалась команда top. В этом разделе вы узнаете об инструментах, которые позволяют наглядно увидеть, чем занята система. Большинство функциональных возможностей данных инструментов частично совпадают. В результате этого полезность одного инструмента по сравнению с другим часто будет зависеть от приложения.

Использование vmstat для извлечения информации о состоянии виртуальной памяти и прочих сведений

Ранее по ходу книги я использовал команду vmstat для вывода сведений о том, как система использует дисковый кэш. Это довольно известный инструмент, существующий уже длительное время. Одно из его преимуществ заключается в простоте: достаточно лишь ввести команду vmstat, и вы получите вывод, состоящий из одной компактной, но весьма полезной информационной строки. Для вывода сведений каждую секунду необходимо ввести vmstat 1, например:

```
$ vmstat 1
procs -----------memory---------- ---swap-- -----io---- --system-- ----cpu----
 r  b swpd   free   buff  cache   si   so    bi    bo   in    cs us sy id wa
 1  1    0   3940  14740 147440    0    0    51    15  263    91  1  8 90  1
 0  1    0   3764  15236 145516    0    0  2952     0  534   606  3 10  0 87
 0  1    0  12368  16320 132804    0    0  1576     0  565   644  0  9  0 91
 0  1    0   8216  16728 135116    0    0  2768     0  455   430  2  5  0 93
 0  1    0   3984  16732 139260    0    0  4164     0  402   354  1  5  0 94
 0  1    0  14452  16764 128484    0    0  3596  1020  438   397  1  5  0 94
 0  1    0  11236  17796 129696    0    0  2196     0  588   695  2  9  0 89
 0  1    0   8028  18724 130944    0    0  2220     0  556   610  2  7  0 91
 0  1    0   4040  19604 132240    0    0  2180     0  532   601  1 12  0 87
 1  1    0   4140  19744 131624    0    0  2964     0  491   506  1  5  0 94
 0  1    0   4368  20024 131004    0    0  3484   932  481   488  1  6  0 93
 0  1    0   9748  20212 124968    0    0  3840     0  502   535  1  8  0 91
```

Информация о процессах в выводе команды vmstat. Информация, касающаяся процессов, отображается в выводе по умолчанию команды `vmstat` в колонках `r` и `b` под заголовком `procs`. В колонке `r` можно увидеть количество процессов, которые в данный момент пребывают в «работоспособном» состоянии, а в колонке `b` — количество процессов, находящихся в состоянии непрерывного «сна».

Процессы, пребывающие в состоянии непрерывной «спячки», блокируются системным вызовом (наиболее вероятно — драйвером устройства). Самой распространенной причиной подобного состояния является ожидание ввода/вывода. Таким образом, если в колонке `b` при каждой выборке сведений часто отображается ненулевое значение, это может быть признаком того, что процесс блокируется «медленным» устройством.

Информация об использовании памяти в выводе команды vmstat. Под заголовком `memory` располагаются колонки `swpd`, `free`, `buff` и `cache`. Они содержат текущие количества страниц памяти, подкачанных на диск, а также свободных и находящихся в системных буферах или кэше соответственно. Все эти значения по умолчанию приводятся в килобайтах. Такое поведение можно изменить при помощи параметра `-S` (для получения дополнительных подробностей см. `vmstat(8)`).

Объем оперативной памяти, выделяемый приложениям и не требующий обращения к подкачке, можно узнать более точно, сложив показатель количества свободной памяти со значениями из колонок `buffers` и `cache`. Именно таким образом аналогичная информация определяется в выводе команды `free`.

Информация об операциях ввода/вывода в итоговом выводе команды vmstat. Сведения об операциях ввода/вывода отображаются в итоговом выводе по умолчанию под двумя заголовками. Под заголовком `swap` можно узнать значение частоты операций перемещения страниц памяти как на устройство подкачки, так и обратно. Сюда относятся *значительные* ошибки страниц, о которых мы говорили ранее в данной главе. В колонке `si` приводится частота операций чтения с устройства подкачки (чтения страниц с данного устройства — *page in*), а в колонке `so` — частота операций записи на устройство подкачки (записи страниц на данное устройство — *page out*). Приводимые в них значения по умолчанию выражаются в килобайтах в секунду. Наиболее оптимальным, конечно же, будет наличие нулевых значений в обеих колонках. Максимум для данных значений определяется скоростью и возможностями устройства подкачки.

Рядом с заголовком `swap` имеется заголовок, обозначенный `io`, — здесь отражаются показатели, касающиеся остальных системных операций ввода/вывода. Сюда входят операции чтения с диска, не являющегося устройством подкачки, и записи на него. К ним относятся все операции ввода/вывода, а не только те, причиной которых стали системные вызовы `read` и `write`, а также ошибки страниц, связанные с отображениями, созданными при помощи вызовов `mmap`. В колонке `bi` выводится частота операций ввода, а в колонке `bo` — частота операций вывода. Значения по умолчанию выражаются в килобайтах в секунду.

Прочая информация, содержащаяся в выводе команды vmstat. Последние два заголовка в выводе по умолчанию обозначены `system` и `cpu`. Под заголовком `system` располагается колонка `in`, в которой приводится значение количества прерываний в секунду. Данное значение всегда выражается в прерываниях в секунду независимо от того, какой итоговый интервал указывается в командной строке.

В бездействующей системе данное значение должно быть очень близким к частоте интервалов хода системных часов, когда на каждый такой интервал должно приходиться одно прерывание.

В колонке cs можно узнать количество переключений контекста в секунду. Переключение контекста ведет к увеличению нагрузки на систему, однако данная нагрузка может быть уменьшена применением методики ленивого сброса содержимого TLB-буфера на диск, о которой речь шла в главе 5. Частота переключения контекста сама по себе ничего не значит, однако в сочетании с другими данными может послужить неплохим источником сведений о поведении системы.

И наконец, под заголовком cpu приводятся показатели времени (в процентах), которое система провела в режиме пользователя (us), режиме ядра (sy) и в бездействии (id). В последней колонке (wa) отражается количество времени, потраченного системой на ожидание ввода/вывода. Данное значение может выводиться любыми версиями инструмента vmstat, однако действительным оно будет только при использовании ядра версии 2.6. В ядрах версии 2.4 и ниже оно всегда будет равно нулю; показатель количества времени, затраченного на ожидание ввода/вывода, будет недоступен и включается в значение времени *бездействия*.

Значение в колонке wa является весьма полезным показателем, поскольку, если ядро проводит много времени в ожидании ввода/вывода, следует предложить, что процессы будут соперничать между собой в процессе ввода/ввода. Устройство, за которое они соперничают, является предметом отдельного обсуждения.

Инструментарий из пакета sysstat

sysstat представляет собой важный пакет, который входит в состав любого дистрибутива. Он содержит инструменты для мониторинга производительности системы как в интерактивном, так и в ретроспективном режиме. Основным инструментом в этом пакете является sar (сокращение от *system activity reporting*) — *средство отчетности об активности системы*. Данные, которые выводит этот инструмент, весьма схожи с теми, что выводит команда vmstat, за исключением того, что он поддерживает множество дополнительных параметров. Уникальное свойство инструмента sar заключается в его способности собирать данные о системе за определенный промежуток времени и представлять их позднее.

Инструмент sar. Как и vmstat, инструмент sar в качестве аргумента принимает интервал, а также счетный аргумент. Однако в выводе по умолчанию будут содержаться сведения, касающиеся только использования ресурсов центрального процессора. Рассмотрим пример:

```
$ sar 1 4
Linux 2.6.16np (redhat)       06/01/2006

09:41:05 PM       CPU     %user     %nice   %system   %iowait  %steal   %idle
09:41:06 PM       all      6.00      0.00     11.00     83.00    0.00    0.00
09:41:07 PM       all      0.99      0.00      8.91     90.10    0.00    0.00
09:41:08 PM       all     95.92      0.00      1.02      3.06    0.00    0.00
09:41:09 PM       all      5.94      0.00      8.91     85.15    0.00    0.00
Average:          all     26.75      0.00      7.50     65.75    0.00    0.00
```

Приведенная команда дает указание sar выводить статистику четыре раза, по разу в секунду. Каждая строка демонстрирует значения использования ресурсов центрального процессора наряду со временем суток по каждой выборке сведений (образчику) отдельно. Эти значения отражают среднюю величину, превышающую указанный интервал (в данном случае это 1 с). Этот вывод, по сути, аналогичен тому, что генерирует команда vmstat, за тем исключением, что в нем имеются дополнительные колонки %nice и %steal. В колонке %nice указывается процент времени, потраченного на выполнение процессов с положительным значением nice (то есть с более низким приоритетом). В колонке %steal отражается процент времени ожидания, форсированного гипервизором[1] на виртуальной машине.

Вывод сведений о виртуальной памяти при помощи инструмента sar. Если задействовать параметр -r, это приведет к генерированию информации о виртуальной памяти, подобно тому как это делает команда vmstat, однако в данном случае вывод будет более детальным:

```
$ sar -r 1 4
Linux 2.6.16np (redhat)    05/31/2006
11:04:54 PM kbmemfree kbmemused  %memused kbbuffers  kbcached kbswpfree kbswpused %swpused  kbswpcad
11:04:55 PM      5504    250120     97.85     20708    128008    524280         0     0.00         0
11:04:56 PM      5504    250120     97.85     20708    128008    524280         0     0.00         0
11:04:57 PM      5504    250120     97.85     20708    128008    524280         0     0.00         0
11:04:58 PM      5504    250120     97.85     20708    128008    524280         0     0.00         0
Average:         5504    250120     97.85     20708    128008    524280         0     0.00         0
```

Воспользовавшись параметром -B, можно просмотреть отчет о подкачке страниц и активности виртуальной памяти:

```
$ sar -B 1 5
Linux 2.6.16np (redhat)         05/31/2006
11:08:53 PM  pgpgin/s pgpgout/s   fault/s  majflt/s
11:08:54 PM      0.00      0.00     53.00      0.00
11:08:55 PM      0.00      0.00     31.00      0.00
11:08:56 PM      0.00      0.00     11.00      0.00
11:08:57 PM      0.00      0.00     11.00      0.00
11:08:58 PM      0.00     48.00     11.00      0.00
Average:         0.00      9.60     23.40      0.00
```

Приведенный вывод, в отличие от генерируемого командой vmstat, содержит значения, выражаемые в страницах (или ошибках) в секунду, при этом такое поведение нельзя изменить. Аналогичный вывод может быть сгенерирован для всех блочных устройств, доступных в системе, или каких-либо других специфических устройств.

Вывод сведений о процессах при помощи инструмента sar. sar, как и команда top, позволяет просматривать сведения о процессах, для чего необходимо воспользоваться параметром -x. В данном случае требуется один аргумент, в роли которого может выступать идентификатор процесса, либо можно указать несколько таких идентификаторов, применив параметр -x соответствующее количество раз:

---

[1] Гипервизор — это программа, используемая для запуска операционной системы на виртуальной машине. Сюда относятся такие приложения с открытым исходным кодом, как Xen, QEMU и Bochs.

```
$ jobs -x sar -x 3942 -x 3970 1 1
Linux 2.6.16np (redhat)         06/01/2006
10:04:09 PM       PID  minflt/s  majflt/s    %user   %system   nswap/s   CPU
10:04:10 PM      3942  43231.00      0.00     5.00     48.00      0.00     0
10:04:10 PM      3970      0.00      0.00    46.00      0.00      0.00     0
Average:          PID  minflt/s  majflt/s    %user   %system   nswap/s
Average:         3942  43231.00      0.00     5.00     48.00      0.00
Average:         3970      0.00      0.00    46.00      0.00      0.00
```

Замечательная особенность команды `sar` состоит в том, что единицы измерения в генерируемом ею выводе всегда однозначно понятны. Эти единицы уже должны быть вам знакомы. Вывод включает колонки `minflt/s`, `majflt/s`, а также колонки, в которых отражаются значения времени (в процентах), проведенного в пространстве пользователя (`%user`) и пространстве ядра (`%system`). Значение в колонке `nswap/s` будет действительным только при использовании ядра версии 2.4 и ниже. В колонке `CPU` указывается идентификатор процессора, с которым ассоциировано выполнение соответствующего процесса. В приведенном примере использовался однопроцессорный компьютер, поэтому все значения в этой колонке равны нулю. Но и при использовании компьютера с поддержкой симметричной многопроцессорной обработки SMP итоговый вывод будет содержать сведения только об одном процессоре, даже если процесс «мигрирует» между разными процессорами. Поддержка симметричной многопроцессорной обработки SMP лишь недавно была внедрена в инструмент `sar`, который в этом плане нуждается в доработке.

При указании параметра `x` в нижнем регистре итоговый вывод будет содержать сведения только об указанных процессах, в то время как параметр `X` в верхнем регистре обуславливает вывод информации *только* о дочерних процессах, порожденных этими процессами. Если у процесса отсутствуют дочерние процессы, в итоговом выводе будут содержаться нулевые значения. Этим можно воспользоваться, когда необходимо просмотреть сведения о порождаемых процессах, поскольку при этом иногда может потребоваться слишком быстрый ввод команд с клавиатуры. Отслеживая общий родительский процесс при помощи параметра `X`, можно узнавать информацию о выполняющихся в данный момент дочерних процессах, а также обо всех новых порождаемых процессах.

Наряду с инструментом `sar` доступен пакет `sysstat`[1], который включает инструменты `mpstat` и `iostat`. В состав дистрибутива Fedora входит служба запуска `sysstat`, которая периодически заносит сведения об активности системы в файлы журналов, располагающиеся в каталоге `/var/log/sa`. Данные файлы используются при «посмертной» отладке активности системы.

Просмотр сведений, касающихся ввода/вывода, при помощи инструмента sar. `sar` может выводить подробную информацию о блочных и сетевых устройствах. Кроме того, вы также сможете просматривать сведения об активности прерываний за определенный промежуток времени. В идеале на одну линию прерывания должно приходиться одно устройство, хотя, как уже отмечалось, так бывает далеко не всегда.

При генерировании вывода в отношении блочных устройств используется подход «все или ничего». Пользователь не может выбирать, какие блочные устройства

[1] См. сайт http://perso.wanadoo.fr/sebastien.godard.

должны подвергаться мониторингу, однако итоговый вывод будет полезен, если вы занимаетесь отладкой использования ресурсов дискового накопителя. Рассмотрим ситуацию, когда для вывода имен устройств вместо их старшего/младшего номера используется параметр -p:

```
$ sar -d -p 1 1
Linux 2.6.16np (redhat)      06/01/2006
10:40:12 PM     DEV     tps  rd_sec/s  wr_sec/s  avgrq-sz  avgqu-sz  await svctm %util
10:40:13 PM     hda    0.00      0.00      0.00      0.00      0.00   0.00  0.00  0.00
10:40:13 PM     hdb  457.00      8.00   8248.00     18.07      1.00   2.19  2.18 99.60
10:40:13 PM     hdc    0.00      0.00      0.00      0.00      0.00   0.00  0.00  0.00
10:40:13 PM   nodev 1037.00      8.00   8288.00      8.00      3.54   3.41  0.96 99.60
10:40:13 PM   nodev    0.00      0.00      0.00      0.00      0.00   0.00  0.00  0.00
```

При помощи команды sar можно выполнять мониторинг использования сетевого устройства, применив параметр -n, для чего требуется один аргумент. Данный аргумент позволяет извлекать сведения о специфических типах сетевых объектов, определяемых программой sar. Упомянутым аргументом может быть один из следующих:

- DEV — статистика об устройствах Ethernet;
- EDEV — статистика об ошибках устройств Ethernet;
- NFS — статистика о клиентах NFS;
- NFSD — статистка о серверах NFS;
- SOCK — статистика о сокетах;
- ALL — статистика обо всех перечисленных объектах.

Активность прерываний может быть важным индикатором активности системы. Инструмент sar позволяет выполнять мониторинг активности прерываний при помощи параметра -I, например:

```
$ sar -I 0 1 1
Linux 2.6.16np (redhat)          06/01/2006
11:00:52 PM      INTR     intr/s
11:00:53 PM         0     250.00
Average:            0     250.00
```

В данном случае я выбрал прерывание 0, которое представляет собой прерывание от таймера. Значение, содержащееся в приведенном выводе, соответствует частоте интервалов хода системных часов, которая в этой системе равна 250 Гц. Для мониторинга определенных прерываний необходимо указать соответствующий аргумент либо воспользоваться аргументом ALL, для того чтобы просмотреть сведения обо всех прерываниях. Если вам необходимо узнать, какое именно прерывание связано с тем или иным устройством, загляните в /proc/interrupts.

Заключительные положения об инструменте sar. Инструмент sar чаще всего используется для мониторинга производительности ретроактивным способом. Система может создать задание cron, которое будет периодически запускать службу sysstat с сохранением результатов в файл. Задание cron, которое входит в состав пакета sysstat, располагается в /etc/cron.d/sysstat и содержит снабженные комментариями элементы, доступные для использования.

Сценарий sysstat использует несколько программ-помощников для сохранения результатов в файлы. Этим файлам присваиваются имена, которые состоят из сочетания sa с порядковым номером дня из двух цифр. Во избежание чрезмерного увеличения размеров этих файлов в них не включается информация о процессах. Здесь вы сможете найти любые сведения, о которых шла речь в этой главе, за исключением тех, что касаются процессов.

Инструменты iostat и mpstat

В состав пакета sysstat входят два полезных инструмента для наблюдения за производительностью системы: iostat и mpstat. Данные, которые позволяют просматривать эти инструменты, аналогичны тем, что содержатся в выводе команды sar, однако они используются исключительно как средства командной строки. Они не позволяют сохранять или извлекать архивные сведения о системе, однако могут оказаться намного удобнее в применении, не требуя при этом запоминания большого количества параметров.

При использовании параметра -d генерируется тот же вывод, что и в случае с командой sar, за тем исключением, что пользователь может сфокусироваться на каком-либо одном устройстве при помощи параметра -p. Например, если ввести команду iostat без аргументов, то итоговый вывод будет содержать системную информацию, касающуюся устройств ввода/вывода, за все время с момента загрузки этой системы:

```
# iostat
Linux 2.6.16np (redhat) 06/03/2006
avg-cpu:  %user   %nice %system %iowait  %steal   %idle
          4.56    0.42    1.55    1.65    0.00   91.83
Device:            tps  Blk_read/s   Blk_wrtn/s   Blk_read   Blk_wrtn
hda               0.01        0.04         0.00        201          0
hdb               5.66      169.67        39.79     833342     195444
hdc               0.00        0.01         0.00         28          0
dm-0              9.42      169.51        39.60     832570     194512
dm-1              0.03        0.01         0.19         72        928
```

Как и в случае с выводом команды sar, используемые единицы измерения не нуждаются в дополнительном разъяснении. Обеспечить периодическое обновление сведений каждую секунду можно, указав соответствующий интервал в секундах в качестве первого аргумента. При запросе периодических обновлений в первой выборке сведений (образчике) всегда будет содержаться совокупная статистика, охватывающая промежуток времени с момента загрузки системы. В последующих позициях выборки будет отражаться текущая активность устройств ввода/вывода.

9.3. Производительность приложений

Вы уже знаете, как выполнять мониторинг производительности системы в целом, а как следует поступить, если вы захотите сфокусироваться на каком-то одном приложении? Например, вы пришли к выводу, что производительность системы снижается из-за какого-то одного «непослушного» процесса. Что вы будете делать дальше? Можно воспользоваться некоторыми из доступных инструментов, однако

в них порой бывает сложно разобраться. Нам необходимо понимать архитектуру системы и центрального процессора, которые мы используем для работы.

Все существующие платформы позволяют использовать те или иные инструменты, однако если вы работаете на компьютере на базе процессорной архитектуры от компании Intel, то вам особенно повезло: здесь будет доступен ряд инструментов, которые используют встроенные регистры для мониторинга производительности и позволяют устранять проблемы, приводящие к снижению скорости работы. В этом разделе мы подробно рассмотрим инструменты, которые поддерживаются всеми архитектурами, а также те из них, что применимы исключительно на архитектуре Intel.

9.3.1. Шаг первый: использование команды time

При настройке производительности использование команды `time`, возможно, окажется наиболее быстрым и легким способом получить ответ на основные из интересующих нас вопросов. Данная команда активно применялась в предыдущих примерах, поэтому я не стану останавливаться на ней, а лишь отмечу, что она всегда должна использоваться в первую очередь при отладке проблем, связанных с производительностью.

9.3.2. Вывод сведений об архитектуре процессора при помощи инструмента x86info

`x86info`[1] выдает сведения о конфигурации процессора на основе информации, обеспечиваемой инструкцией `cupid`, которая реализована в процессорной архитектуре Intel. Данный инструмент входит в состав большинства дистрибутивов, однако архитектура Intel находится в постоянном развитии. Если ваш дистрибутив старше процессора, то, скорее всего, представляемая им информация будет неполной или вообще неверной.

В частности, одна из проблем связана с подходом архитектуры Intel к характеристике кэша. Рассмотрим пример:

```
$ x86info
x86info v1.17.  Dave Jones 2001-2005
Feedback to <davej@redhat.com>.

Found 1 CPU
--------------------------------------------------------------------------
Found unknown cache descriptors: 64 80 91 102 112 122
Family: 15 Model: 1 Stepping: 2 Type: 0 Brand: 8
CPU Model: Pentium 4 (Willamette) [D0] Original OEM
Processor name string: Intel(R) Pentium(R) 4 CPU 1.70GHz

Feature flags:
 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflsh ds
acpi mmx fxsr sse sse2 ss ht tm
```

[1] См. сайт http://sourceforge.net/projects/x86info.

```
Extended feature flags:

Instruction trace cache:
        Size: 12K uOps  8-way associative.
L1 Data cache:
        Size: 8KB       Sectored, 4-way associative.
        line size=64 bytes.
L2 unified cache:
        Size: 256KB     Sectored, 8-way associative.
        line size=64 bytes.
Instruction TLB: 4K, 2MB or 4MB pages, fully associative, 64 entries.
Found unknown cache descriptors: 64 80 91 102 112 122
Data TLB: 4KB or 4MB pages, fully associative, 64 entries.
The physical package supports 1 logical processors
```

Так выглядит информация о моем скромном процессоре Pentium 4 с тактовой частотой 1,7 ГГц, который уже считается устаревшим. Однако инструмент `x86info` по-прежнему не распознает некоторые дескрипторы кэша. Сведения о дескрипторах кэша процессоров Intel можно почерпнуть из веб-документа Intel Application Note 485[1], который довольно часто обновляется.

Аналогичные сведения можно отыскать в `/proc/cpuinfo`, однако данная информация является «встроенной» в ядро. Высока вероятность того, что она будет даже еще более устаревшей. Если ваш процессор новее, чем используемое ядро, то данная информация, скорее всего, будет неполной или неверной. Кроме того, она содержит менее конкретизированные данные, чем те, что выводит команда `x86info`. Так, в частности, в `/proc/cpuinfo` не проводится различий между уровнями кэша L1, L2 или L3. Воспользуемся моим скромным процессором Pentium 4 в качестве «подопытного» еще раз:

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 15
model           : 1
model name      : Intel(R) Pentium(R) 4 CPU 1.70GHz
stepping        : 2
cpu MHz         : 1700.381
cache size      : 256 KB
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 2
wp              : yes
flags         : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36
clflush dts acpi mmx fxsr sse sse2 ss ht tm
bogomips        : 3406.70
```

[1] См. сайт http://developer.intel.com.

Давайте теперь воспользуемся представленной информацией. Сначала нам потребуется программа для работы — cache-miss.c, которая приведена в листинге 9.3.

Листинг 9.3. cache-miss.c: программа для демонстрации инструментов, используемых для отслеживания удачных и неудачных обращений к кэшу

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <stdlib.h>
 4 #include <unistd.h>
 5 #include <malloc.h>
 6 #include <assert.h>
 7
 8 int main(int argc, char *argv[])
 9 {
10    // Использовать по умолчанию 64 TLB-элемента (максимальное количество,
      // поддерживаемое моим процессором)
11     int num_pages = 64;
12
13    // Пользователь может указать любое количество страниц
14     if (argc > 1) {
15         num_pages = atoi(argv[1]);
16     }
17
18     const size_t page_size = sysconf(_SC_PAGESIZE);
19
20     int num_bytes = num_pages * page_size;
21     int alloc_bytes = num_bytes;
22
23     if (alloc_bytes == 0) {
24         // Выполнить распределение одной страницы в качестве основы,
25         // но использовать ее мы не будем
26         alloc_bytes = page_size;
27     }
28
29     // Выполнить распределение памяти с выравниванием в соответствии
       // с границами страницы
30     char *buf = memalign(page_size, alloc_bytes);
31     assert(buf != NULL);
32
33     printf("%d pages %d KB\n", num_pages, num_bytes / 1024);
              (printf("%d страниц %d Кб\n", num_pages, num_bytes / 1024);
34
35     // Пользователь запросил нулевое количество страниц. Мы выполнили
       // распределение одной страницы, но
36     // не использовали ее, благодаря чему не возникло дополнительных
       // ошибок страниц
37     if (num_pages == 0) {
38         exit(0);
39     }
40
41     // Необходимо хранилище для размещения считаных байтов
42     static volatile char store;
```

```
43
44      /*
45       ** С каждой страницы осуществляется чтение по одному байту
46       ** до тех пор, пока число операций чтения не достигнет 1 000 000
47       */
48
49      int i;
50      char *c = buf;
51
52      for (i = 0; i < 1000000; i++) {
53          store = *c;
54          c += page_size;
55          if (c - buf >= num_bytes)
56              c = buf;
57      }
58
59      return 0;
60  }
```

Обратите внимание на то, что данная программа выполняет чтение по одному байту каждой распределенной страницы до тех пор, пока число операций чтения не достигнет 1 млн. Запись байт осуществляется в переменную, благодаря чему компилятор не отбрасывает инструкции чтения из-за оптимизации. Данный пример я буду использовать также для наглядной демонстрации некоторых инструментов.

9.3.3. Использование Valgrind для проверки эффективности инструкций

Инструмент `Valgrind` доступен для использования на архитектурах Intel (IA32 и X86_64), а также на 32-битных архитектурах Power PC. На самом деле он представляет собой набор инструментов для поиска утечек и повреждений памяти. В этом разделе мы сфокусируемся на инструменте под названием `cachegrind`, который позволяет определить кэш-эффективность кода. Выполним компиляцию примера из листинга 9.3:

```
$ cc -O2 -o cache-miss cache-miss.c
```

Для получения наилучшего результата компиляцию следует производить с активированной оптимизацией. Данная программа принимает один аргумент, который представляет собой значение количества страниц, подлежащих распределению. При запуске программа выполняет 1 млн операций чтения — по байту с каждой страницы, повторяя цикл до тех пор, пока не будет достигнут указанный предел. Сначала запустим ее с аргументом в виде нулевого значения количества страниц, что даст нам основу для сравнения. Команда для вызова `Valgrind` при помощи инструмента `cachegrind` выглядит следующим образом:

```
$ valgrind --tool=cachegrind ./cache-miss 0

==18902== Cachegrind, an I1/D1/L2 cache profiler.
==18902== Copyright (C) 2002-2005, and GNU GPL'd, by Nicholas Nethercote et al.
```

```
==18902== Using LibVEX rev 1471, a library for dynamic binary translation.
==18902== Copyright (C) 2004-2005, and GNU GPL'd, by OpenWorks LLP.
==18902== Using valgrind-3.1.0, a dynamic binary instrumentation framework.
==18902== Copyright (C) 2000-2005, and GNU GPL'd, by Julian Seward et al.
==18902== For more details, rerun with: -v
==18902==
--18902-- warning: Pentium 4 with 12 KB micro-op instruction trace cache
--18902--          Имитация 16 Кбайт I-кэша с 32 B строками
0 pages 0 KB
==18902==
==18902== I   refs:      137,569                          Кэш инструкций
==18902== I1  misses:      1,216
==18902== L2i misses:        694
==18902== I1  miss rate:    0.88%
==18902== L2i miss rate:    0.50%
==18902==
==18902== D   refs:       62,808   (47,271 rd + 15,537 wr) Кэш данных
                                                           уровней L1 и L2
==18902== D1 misses: 2,253 ( 1,969 rd + 284 wr)
==18902== L2d misses:      1,251   ( 1,041 rd +    210 wr)
==18902== D1  miss rate:     3.5%  (   4.1%   +    1.8%  )
==18902== L2d miss rate:     1.9%  (   2.2%   +    1.3%  )
==18902==
==18902== L2  refs:        3,469   ( 3,185 rd +    284 wr)  Кэш данных
                                                            только уровня L2
==18902== L2  misses:      1,945   ( 1,735 rd +    210 wr)
==18902== L2  miss rate:     0.9%  ( 0.9%     +    1.3%  )
```

Данный вывод содержит много информации, однако я выделил наиболее важную ее часть. Это поможет понять, что именно здесь нас интересует. В данном случае программа незамедлительно завершает работу, поэтому большинство видимой активности генерируется процессом загрузки приложения. Если использовать аналогичные входные параметры, то в итоговом выводе будут отображаться одинаковые сведения при повторном выполнении программы (за исключением, конечно же, `pid`-идентификатора).

Из приведенного вывода видно, что общее количество обращений, касающихся данных, составляет 62 808. Среди них 47 271 — это запросы на чтение, а оставшиеся 15 537 — запросы на запись. Самое интересное содержится в следующей строке, где говорится, что при обращении к кэшу данных первого уровня L1 (сокращенно обозначен `D1`) произошло 2253 неудачных попытки. *Неудачное обращение к кэшу* — это операция чтения или записи, при которых запрашивались данные, отсутствующие в этом кэше. Неудачные обращения к кэшу случаются при выполнении первой операции чтения или записи строки кэша. Поскольку в нашем случае осуществлялось чтение лишь одного байта каждой страницы, учитываются только эти операции чтения.

Когда происходит неудачное обращение к кэшу, инструкция *простаивает*, пока идет заполнение строки кэша. Это приводит к тому, что выполнение инструкции занимает больше времени и может отсрочить завершение других инструкций. Итоговые значения можно увидеть в строках `miss rate`, где количество неудачных попыток

отражается в процентном отношении от общего числа обращений. В приведенном примере частота неудачных обращений при операциях чтения составила 3,5 %.

Теперь давайте посмотрим, что произойдет, когда программа на самом деле будет выполнять какие-либо действия. На этот раз используем аргумент количества страниц, равный 2, это означает, что необходимые данные, к которым осуществляется доступ, умещаются в кэше первого уровня L1:

```
$ valgrind --tool=cachegrind ./cache-miss 2       Распределение двух страниц
==18908== Cachegrind, an I1/D1/L2 cache profiler.
...
1 pages 4 KB
==18908==
==18908== I   refs:      10,136,737
==18908== I1  misses:         1,213
==18908== L2i misses:           692
==18908== I1  miss rate:       0.01%
==18908== L2i miss rate:       0.00%
==18908==
==18908== D refs: 2,062,373 (1,046,952 rd + 1,015,421 wr)
==18908== D1 misses: 2,236 ( 1,951 rd + 285 wr)
==18908== L2d misses:     1,252      (    1,041 rd +       211 wr)
==18908== D1  miss rate:    0.1%     (      0.1%   +       0.0%  )
==18908== L2d miss rate:    0.0%     (      0.0%   +       0.0%  )
==18908==
==18908== L2  refs:       3,449      (    3,164 rd +       285 wr)
==18908== L2  misses:     1,944      (    1,733 rd +       211 wr)
==18908== L2  miss rate:    0.0%     (        0.0% +       0.0%  )
```

Обратите внимание на то, что количество обращений, связанных с данными, превышает 2 млн. В данном случае приложение считывает и записывает байты 1 млн раз, как и планировалось. Поскольку необходимые данные умещаются в кэше, количество неудачных обращений к кэшу данных первого уровня L1 не увеличивается. На самом деле количество неудачных обращений при операциях чтения даже уменьшилось — с 1969 до 1951. По всей вероятности, это связано с тем, что упреждающая выборка при чтении осуществляется в контроллере кэша, хотя уверенности в этом нет. Количество неудачных обращений при операциях записи увеличилось только на единицу — с 284 до 285. Поскольку количество удачных обращений к кэшу довольно велико по сравнению с количеством неудачных попыток, число последних считается незначительным, в результате чего частота неудачных обращений в приведенном примере равна нулю.

Теперь давайте внесем дополнительную интригу. Используемый в данном случае процессор располагает кэшем размером 256 Кбайт (64 страницы), поэтому распределим полностью весь кэш второго уровня L2 и посмотрим, каков будет результат:

```
$ valgrind --tool=cachegrind ./cache-miss 64
...
64 pages 256 KB
==18914==
==18914== I   refs:       9,152,148
==18914== I1  misses:         1,176
```

```
==18914== L2i misses:           670
==18914== I1  miss rate:       0.01%
==18914== L2i miss rate:       0.00%
==18914==
==18914== D   refs:       2,062,236   (1,046,866 rd + 1,015,370 wr)
==18914== D1  misses:     1,002,231   (1,001,947 rd +       284 wr)
==18914== L2d misses:         1,315   (    1,105 rd +       210 wr)
==18914== D1  miss rate:       48.5%  (     95.7%   +       0.0%  )
==18914== L2d miss rate:        0.0%  (      0.1%   +       0.0%  )
==18914==
==18914== L2  refs:       1,003,407  (1,003,123 rd +       284 wr)
==18914== L2  misses:         1,985  (    1,775 rd +       210 wr)
==18914== L2  miss rate:        0.0% (      0.0%   +       0.0%  )
```

Теперь практически все попытки обращения к кэшу данных первого уровня L1 при операциях чтения являются неудачными, а большинство обращений к аналогичному кэшу при операциях записи — успешными. А все из-за того, что буфер чтения больше не умещается в кэше данных первого уровня L1, но, поскольку запись все время осуществляется в одну и ту же область памяти, все попытки обращения к кэшу при операциях записи всегда оказываются удачными. Однако в примере с кэшем в 256 Кбайт необходимые данные все еще умещаются в кэше данных второго уровня L2, о чем свидетельствуют выведенные сведения, которые следуют за статистикой по кэшу данных первого уровня L1.

Кэш второго уровня L2 используемого процессора является *унифицированным*, что означает, что данный кэш используется для работы как с инструкциями, так и с данными. В приведенном примере видно, что произошло 1 003 407 обращений к контроллеру кэша второго уровня L2, в то время как в приведенном ранее примере их было только 3449. В этот показатель включаются операции как выборки инструкций, так и чтения данных. Процессор не проводит различия между этими двумя видами обращений, однако делает это в отношении двух типов неудачных попыток. Для нас интерес представляют неудачные обращения к кэшу данных, обозначенные в выводе `L2d misses`.

Большинство обращений к кэшу второго уровня L2, как и ожидалось, составили операции чтения. Количество неудачных обращений является незначительным, а это говорит о том, что бóльшая часть обращений к кэшу при операциях записи оказалась успешной. Это отражается в виде значения `0.0%` в строке `L2 miss rate`. В завершение давайте посмотрим, как будет выглядеть итоговый вывод, когда необходимые данные перестанут умещаться в кэше второго уровня L2:

```
$ valgrind --tool=cachegrind ./cache-miss 256
...
256 pages 1024 KB
==18918==
==18918== I    refs:     9,140,561
==18918== I1   misses:       1,176
==18918== L2i  misses:         670
==18918== I1   miss rate:     0.01%
==18918== L2i  miss rate:     0.00%
==18918==
```

```
==18918== D    refs:      2,062,295  (1,046,911 rd + 1,015,384 wr)
==18918== D1   misses:    1,002,230  (1,001,946 rd +       284 wr)
==18918== L2d  misses:    1,001,251  (1,001,041 rd +       210 wr)
==18918== D1   miss rate:      48.5% (     95.7%   +       0.0%  )
==18918== L2d  miss rate:      48.5% (     95.6%   +       0.0%  )
==18918==
==18918== L2   refs:      1,003,406  (1,003,122 rd +       284 wr)
==18918== L2   misses:    1,001,921  (1,001,711 rd +       210 wr)
==18918== L2   miss rate:       8.9% (      9.8%   +       0.0%  )
```

В данном примере использовалось 256 страниц, что в 4 раза превосходит размер кэша второго уровня L2. Можно предположить, что каждая операция чтения блока памяти будет заканчиваться неудачной попыткой обращения к кэшу. Приведенный вывод свидетельствует о том, что общее количество неудачных обращений к кэшу данных второго уровня L2 превышает 1 млн. Инструмент `Valgrind` выражает частоту неудачных попыток в процентах от общего числа обращений к кэшу, указанного в строке `D refs`. Сюда включаются операции чтения, связанные с загрузкой кода, в силу чего частота неудачных обращений к `L2d` в данном примере равна 95,6 %, а не 100 %.

Итоговая частота неудачных обращений к кэшу данных второго уровня L2 (`L2d miss rate`) — 48,5 %, поскольку сюда входят операции как чтения, так и записи. Как отмечалось ранее, программа осуществляет равное количество операций чтения и записи, за тем исключением, что все процедуры записи выполняются на одну страницу, которая всегда располагается в кэше.

Частота неудачных обращений к кэшу второго уровня L2 в приведенном примере может отчасти ввести в заблуждение. Здесь указывается, что она составляет 8,9 %, что является отношением количества неудачных обращений к кэшу второго уровня L2 (касающихся как инструкций, так и данных) к общему числу всех операций чтения и записи (9 140 561). Если взглянуть на это значение, то может показаться, что не все так плохо. Однако этот показатель обманчив, поскольку при всех операциях выборки инструкций, происходящих во время цикла, наблюдаются удачные попытки обращения к кэшу, что приводит к занижению итогового значения средней частоты неудачных обращений.

`Valgrind` содержит и другие инструменты, заслуживающие внимания. Он является превосходным средством, которое постоянно совершенствуется и должно присутствовать в инструментарии любого программиста. Вывод, генерируемый `Valgrind`, будет содержать полезные сведения даже в том случае, если вы исследуете архитектуру, которую данный инструмент не поддерживает. Если вы сможете перенести свой программный код на поддерживаемую платформу и запустить его выполнение, то сведения, почерпнутые из вывода `Valgrind`, пригодятся вам при отладке собственного исходного кода.

9.3.4. Инструмент ltrace

`ltrace` позволяет отслеживать использование библиотечных вызовов приложениями. Как и инструмент `strace`, который фигурировал в примерах, приводившихся ранее, `ltrace` выводит сведения о вызываемых функциях, а также их аргументах.

Разница заключается в том, что strace показывает сведения только о системных вызовах, в то время как ltrace может выводить информацию, касающуюся библиотечных вызовов. Поскольку системные вызовы C и C++ совершаются с использованием стандартных библиотечных оберток, инструмент ltrace также способен выводить на экран аналогичну́ю информацию.

Особенность применения ltrace заключается в том, что при этом выполнение программы существенно замедляется. Функционирование ltrace во многом напоминает подключение к процессу с использованием отладчика, что требует дополнительного времени. Каждый вызов функции — это как точка останова, которая вынуждает инструмент ltrace сохранять временную статистику.

По умолчанию ltrace генерирует подробный вывод, содержащий сведения обо всех библиотечных вызовах. Для применения фильтра с целью вывода информации только об интересующих вас функциях необходимо воспользоваться параметром -e. Также можно отследить продолжительность каждого вызова функции в отдельности при помощи параметра -T, например:

```
$ ltrace -T -e read,write dd if=/dev/urandom of=/dev/null count=1
read(0,
"\007\354\037\024b\316\t\255\001\322\222O\b\251\224\335\357w\302\351\207\323\n$\
032\342\211\315\2459\006{"..., 512) = 512 <0.000327>
write(1,
"\007\354\037\024b\316\t\255\001\322\222O\b\251\224\335\357w\302\351\207\323\n$\
032\342\211\315\2459\006{"..., 512) = 512 <0.000111>
1+0 records in
1+0 records out
+++ exited (status 0) +++
```

В данном выводе вы можете наблюдать дополнительную информацию по каждому вызову, при этом благодаря использованию параметра -T также отображается продолжительность вызовов (в угловых скобках). Кроме того, здесь видно, что длительность вызова read превышает таковую у вызова write, но лишь в 3 раза. На точность выводимых значений могут влиять «накладные расходы», связанные с выполнением программы. Поэтому не стоит слишком зацикливаться на них.

Ограничение инструмента ltrace состоит в том, что он дает возможность отслеживать только вызовы функций, относящихся к динамическим библиотекам, и не позволяет делать этого в отношении вызовов функций, содержащихся в статически связанных библиотеках.

9.3.5. Использование strace для мониторинга производительности приложений

strace и ltrace используются в сочетании с многими одинаковыми параметрами и во многом схожими способами. Разница заключается в том, что strace отслеживает исключительно системные вызовы. Но, в отличие от ltrace, данному инструменту не требуется динамическая библиотека для отслеживания системных вызовов. Вызовы отслеживаются независимо от того, используют ли они функции-обертки, поскольку системные вызовы задействуют прерывания для перехода меж-

ду режимом пользователя и режимом ядра. В результате «накладные расходы» оказываются меньше при использовании `ltrace`, даже если `strace` замедляет выполнение программы. Однако это замедление не будет настолько существенным, как при использовании `ltrace`.

Показатели времени, приводимые в выводе команды `strace`, не стоит воспринимать слишком серьезно. Отрицательный «побочный эффект» подхода, на основе которого `strace` осуществляет мониторинг выполнения программ, заключается в том, что данный инструмент склонен сильно недооценивать значения системного времени.

Возможность подключаться к выполняющемуся процессу является весьма полезным свойством. Оно придется особенно кстати, когда вы будете иметь дело с процессом, который завис. Подключение к процессу осуществляется при помощи команды `strace` с параметром `-p`:

```
$ strace -ttt -p 23210
Process 23210 attached - interrupt to quit
1149372951.205663 write(1, "H", 1)      = 1
1149372951.206036 select(0, NULL, NULL, NULL, {1, 0}) = 0 (Timeout)
1149372952.209712 write(1, "e", 1)      = 1
1149372952.210045 select(0, NULL, NULL, NULL, {1, 0}) = 0 (Timeout)
1149372953.213802 write(1, "l", 1)      = 1
1149372953.214133 select(0, NULL, NULL, NULL, {1, 0}) = 0 (Timeout)
1149372954.217911 write(1, "l", 1)      = 1
1149372954.218244 select(0, NULL, NULL, NULL, {1, 0}) = 0 (Timeout)
1149372955.221994 write(1, "o", 1)      = 1
Process 23210 detached
```

В этом своеобразном примере я создал процесс, который осуществляет запись приветствия `Hello World` по 1 байту за раз, погружаясь между этими этапами «в спячку». Это может показаться глупым, однако данный пример наглядно демонстрирует, как будет выглядеть итоговый вывод. Воспользовавшись параметром `-c`, вы также можете вывести на экран гистограмму в данном режиме.

9.3.6. Традиционные инструменты для настройки производительности: gcov и gprof

Инструменты `gcov` и `gprof`, применяемые еще в системе UNIX, весьма полезны, но довольно сложны в использовании. Проблема, связанная с данными утилитами, состоит в том, что они требуют от пользователя инструментировать его исполняемые файлы, что может оказаться сложным и понизить производительность. Оптимизация способна привести к тому, что вывод не будет удобочитаемым, при этом исходный код может не совсем точно отражать то, что делает центральный процессор. Отключение оптимизации и активация отладки позволят гарантировать, что каждая строка исходного кода будет снабжена машинными инструкциями, ассоциированными с ней. Однако запуск исполняемого файла без оптимизации во многих приложениях является недопустимым. К счастью, участники проекта GNU проделали огромную работу, для того чтобы гарантировать пользователям возможность оптимизировать свой код и по-прежнему получать приемлемые результаты.

Использование инструмента gprof

gprof представляет собой инструмент для профилирования исполняемых файлов, то есть он позволяет определить, на что та или иная программа тратит основную часть своего времени при выполнении. Особенность его заключается в том, что «измерению» подвергается лишь программный код, который был оснащен компилятором. Эта процедура генерируется при компиляции посредством gcc с флагом -pg. Любые неинструментированные программные файлы не «замеряются», то есть все модули, при компиляции которых не использовался параметр -pg, не будут подвергаться оценке. Так обычно бывает со связываемыми библиотеками. Если программный код пользователя вызывает библиотечную функцию, на что уходит определенное время, данный временной промежуток записывается «на счет» вызываемой функции, а пользователю приходится самостоятельно выяснять, почему на вызов этой функции уходит так много времени.

В листинге 9.4 приведен простой пример, наглядно демонстрирующий, насколько полезным может оказаться профайлер, а также некоторые ограничения последнего.

Листинг 9.4. profme.c: программа для демонстрации профилирования

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <stdlib.h>
 4 #include <math.h>
 5
 6 /* Установить x для чего-либо. Не очень быстро */
 7 double slow(double x)
 8 {
 9 return pow(x, 1.12345);
10 }
11
12 /* Деление с плавающей точкой — очень медленно */
13 double slower(double x)
14 {
15     return 1.0 / x;
16 }
17
18 /* Квадратный корень — возможно, наиболее медленно */
19 double slowest(double x)
20 {
21     return sqrt(x);
22 }
23
24 int main(int argc, char *argv[])
25 {
26     int i;
27     double x;
28
29     /* Здесь нужно большое число для обеспечения наглядности */
30     for (i = 0; i < 3000000; i++) {
31         x = 100.0;
```

```
32          x = slow(x);
33          x = slower(x);
34          x = slowest(x);
35      }
36 }
```

Перед тем как использовать профайлер, вы должны осуществить сборку и связывание своего программного кода при помощи параметра `-pg`. Кроме того, вы должны использовать те же флаги, которые обычно применяете при компилировании. Если вам привычнее заниматься компилированием с активированной оптимизацией, поступайте так и в дальнейшем, если же вы при этом отключаете оптимизацию, делайте так и в этот раз. Если вы воспользуетесь другими флагами для профилирования, то итоговый результат может оказаться не тем, которого вы ожидаете. Для сборки приведенного примера необходимо использовать флаги `-O2` и `pg`:

```
$ cc -pg -O2 -o profme profme.c -lm
$ ./profme
```

Каждый раз при выполнении данной программы в текущем каталоге будет генерироваться файл с именем `gmon.out`. Он, наряду с инструментированным исполняемым файлом, используется в качестве ввода в программу `gprof`. Наиболее простым и полезным выводом `gprof` является плоский профиль. По умолчанию он выглядит следующим образом:

```
$ gprof ./profme
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
time   seconds     seconds    calls  ns/call  ns/call name
 46.88     0.15       0.15  3000000    50.00    50.00 slowest
 37.50     0.27       0.12  3000000    40.00    40.00 slower
 12.50     0.31       0.04                             main
  3.12     0.32       0.01  3000000     3.33     3.33 slow
...
```

Использование gcov и gprof для профилирования

Обычно инструмент `gcov` применяется для определения *охвата программного кода*, то есть того количества кода, который оказался выполненным во время конкретного прогона. Охват кода служит неплохим показателем того, насколько хорошо ваш код был протестирован, а также может оказаться полезным при осуществлении оптимизации, особенно в ситуациях, когда вы будете иметь дело с незнакомыми программами.

Допустим, у вас есть приложение, которое было написано студентом-практикантом. Оно является полезным, но работает медленно. Попытки оптимизировать эту программу при помощи компилятора ненамного увеличивают ее производительность, так как практикант, создавший ее, внимания оптимизации не уделил. В данном случае вам придется углубиться в программный код и вручную выполнить оптимизацию. При этом вы отметите следующее: помимо того что данная

программа неэффективна, ее исходный код также непонятен. И здесь вам на помощь придет инструмент gcov.

Вместо того чтобы корпеть над тысячами строк программного кода, пытаясь улучшить его, что может оказаться маловероятным, имеет смысл направить свои усилия на те строки, которые выполняются наиболее часто. Совсем не обязательно, что они будут являться тем «местом», где программа тратит основную часть своего времени при выполнении, однако станут хорошей отправной точкой. Оптимизация вручную предполагает комбинирование методики тестирования охвата программного кода с профилированием.

В листинге 9.5 представлен гипотетический проект, которым мог заниматься ваш практикант.

Листинг 9.5. summer-proj.c: пример для иллюстрации профилирования и охвата программного кода

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <stdlib.h>
 4 #include <time.h>
 5 #include <math.h>
 6
 7 volatile double x;
 8
 9 int main(int argc, char *argv[])
10 {
11     int i;
12     for (i = 0; i < 16000000; i++) {
13         x = 1000.0;
14
15         /* 0 <= r < 16 */
16         int r = i & 0xf;
17
18         if (r <= 8) {
19             x = pow(x, 1.234);  /* Подвергается вызову 9/16 времени */
20         }
21         else if (r <= 11) {
22             x = sqrt(x);        /* Подвергается вызову 3/16 времени */
23         }
24         else {
25             x = 1.0 / x;        /* Подвергается вызову 4/16 времени */
26         }
27     }
28 }
```

Для обеспечения возможности тестирования охвата программного кода компилирование должно осуществляться с использованием двух специальных флагов:

```
$ cc -g -O2 -ftest-coverage -fprofile-arcs summer-proj.c -o summer-proj -pg -lm
```

Для тестирования охвата программного кода рекомендуется не активировать оптимизацию — это позволит лучше сохранить структуру строк кода. В нашем случае это не имеет значения, поэтому мы включаем оптимизацию. Благодаря исполь-

зованию флага -ftest-coverage компилятор создает файл с именем summer-proj.gcno в дополнение к исполняемому файлу. Флаг -fprofile-arcs приводит к тому, что при запуске исполняемого файла генерируется файл с именем summer-proj.gcda. Оба этих файла используются в качестве ввода в программу gcov[1]. В конце мы указали параметр -pg, благодаря чему можно использовать также утилиту gprof. Перед запуском gcov необходимо выполнить программу минимум один раз:

```
$ ./summer-proj                    Создание файла summer-proj.gcda
$ gcov ./summer-proj               Создание файла summer-proj.c.gcov
File 'summer-proj.c'
Lines executed:100.00% of 9
summer-proj.c:creating 'summer-proj.c.gcov'
```

Как и ожидалось, в данном случае охват программного кода составил 100 %, то есть все исполняемые строки кода были выполнены во время прогона. Однако нас интересует другое. Нам необходимо взглянуть на значения счетчиков, которые содержатся в summer-proj.c.gcov, для того чтобы узнать, что произошло. Обратимся к листингу 9.6.

Листинг 9.6. summer-proj.c.gcov: пример вывода gcov

```
        -:    0:Source:summer-proj.c
        -:    0:Graph:summer-proj.gcno
        -:    0:Data:summer-proj.gcda
        -:    0:Runs:1
        -:    0:Programs:1
        -:    1:#include <stdio.h>
        -:    2:#include <string.h>
        -:    3:#include <stdlib.h>
        -:    4:#include <time.h>
        -:    5:#include <math.h>
        -:    6:
        -:    7:volatile double x;
        -:    8:
        -:    9:int main(int argc, char *argv[])
        1:   10:{
        -:   11:    int i;
 16000000:   12:    for (i = 0; i < 16000000; i++) {
 16000000:   13:        x = 1000.0;
        -:   14:
        -:   15:        /* 0 <= r < 16 */
 16000000:   16:        int r = i & 0xf;
        -:   17:
 16000000:   18:        if (r <= 8) {
  9000000:   19:           x = pow(x, 1.234); /* Подвергается вызову 9/16
                                                времени */
        -:   20:        }
  7000000:   21:        else if (r <= 11) {
```

[1] Файл .gcda также может использоваться в качестве обратной связи с компилятором для проведения оптимизации на основе вероятностей переходов при помощи параметра fbranch-probabilities.

```
3000000:   22:            x = sqrt(x);      /* Подвергается вызову 3/16
                                               времени */
      -:   23:        }
      -:   24:        else {
4000000:   25:            x = 1.0 / x;      /* Подвергается вызову 4/16
                                               времени */
      -:   26:        }
      -:   27:    }
      -:   28:}
```

Обратите внимание на то, что значения счетчиков хорошо согласуются с предварительной информацией, приведенной в комментариях. Поскольку большинство строк кода не имеют таких полезных комментариев, необходимо воспользоваться инструментом gcov. Из этого вывода следует, что строка 19 вызывается наиболее часто. Как вы уже знаете, строки 19, 22 и 25 соответствуют функциям slow, slower и slowest из листинга 9.4, но без оберток. Из приведенного вывода понятно, что вызов sqrt значительно «медленнее» двух других. Но где же данная программа тратит основную часть своего времени при выполнении? Здесь опять на помощь приходит инструмент gprof:

```
$ gprof --no-graph -l summer-proj| head -10
Flat profile:
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
time  seconds     seconds    calls  Ts/call  Ts/call name
34.00    0.17       0.17                   main (summer-proj.c:22 @ 8048926)
20.00    0.27       0.10                   main (summer-proj.c:25 @ 8048879)
16.00    0.35       0.08                   main (summer-proj.c:19 @ 80488c6)
11.00    0.41       0.06                   main (summer-proj.c:18 @ 80488b3)
 6.00    0.43       0.03                   main (summer-proj.c:21 @ 8048870)
...
```

Обратите внимание на то, что, поскольку данная программа не вызывает каких-либо инструментированных функций, наряду с командой gprof необходимо указывать параметр nograph. В противном случае вы увидите на экране лишь следующее сообщение:

```
gprof: gmon.out file is missing call-graph data
```

Из рассмотренного вывода видно, что строки ранжируются в соответствии с временем, затраченным на выполнение каждой из них. Теперь нам ясно: несмотря на то что самые большие значения счетчика — в строке 19, программа тратит основную часть своего «рабочего» времени при выполнении строки 22 (sqrt). Таким образом, значения счетчика выполнения строк, содержащиеся в выводе gcov, не совсем точны. В случае с большими и более сложными программами вывод, генерируемый инструментом gcov, будет служить неплохим источником сведений о том, где они тратят основную часть своего времени при выполнении. Аннотированные листинги зачастую легче интерпретировать, чем вывод, генерируемый gprof. Однако существует инструмент, который может выполнять обе эти задачи с меньшими потерями, — OProfile.

9.3.7. Инструмент OProfile

В ядро Linux совсем недавно был внедрен встроенный профайлер под названием `OProfile`. Для того чтобы им можно было пользоваться, при компиляции ядра необходимо активировать поддержку данного инструмента[1]. `OProfile` представляет собой весьма сложный инструмент, который обладает множеством преимуществ по сравнению с `gcov` и `gprof`.

Несмотря на то что `OProfile` является наиболее полезным инструментом для разработчиков ядра, позволяющим определить, где ядро тратит основную часть своего времени при выполнении, он вполне подходит и для программистов, поскольку может выводить сведения о том, что происходит в пространстве пользователя. Более того, при желании вы можете работать исключительно в пространстве пользователя. Вам не потребуются исходные файлы ядра или инструментированные объектные файлы ни в пространстве ядра, ни в пространстве пользователя. Другими словами, для профилирования своего программного кода вам не нужно будет проводить его пересборку.

Однако не все так просто. Для запуска профайлера (`oprofiled`) необходимо обладать привилегиями корневого пользователя *root*. При его запуске работа приложения замедлится, однако не столь серьезно, как в случае с инструментированными объектными файлами с использованием флагов компилятора. В последующих разделах мы рассмотрим, как именно функционирует инструмент `OProfile`.

`OProfile` состоит из пакета инструментов пространства пользователя, куда входят демон и ряд инструментов командной строки, о которых мы поговорим немного позже. Однако данные инструменты работают только на ядре, при сборке которого была активирована поддержка `OProfile`.

К инструментам командной строки относятся:

- `opcontrol` — запускает и приостанавливает работу демона `oprofiled`;
- `opreport` — выводит отчет на основе последних статистических данных, собранных `oprofiled`;
- `opgprof` — генерирует файл `gmon.out`, используемый утилитой `gprof`;
- `opannotate` — создает аннотированные исходные файлы из данных профиля.

Программа `opcontrol` является основным инструментом для управления демоном `oprofiled`, который собирает статистические сведения.

Управление профайлером при помощи opcontrol. Инструмент `opcontrol` по умолчанию поддерживает множество параметров, что позволяет максимально облегчить проведение профилирования. Для выбора также доступны и другие параметры, однако применять их рекомендуется только опытным пользователям. Кроме того, `opcontrol` позволяет сохранять параметры командной строки после каждого прогона, в результате чего они будут использоваться по умолчанию при следующем запуске программы `opcontrol`.

---

[1] Соответствующий флаг можно найти при конфигурировании ядра: Instrumentation Support □ Profiling Support. После того как вы активируете профилирование, будет предложено выбрать опцию OProfile.

Для запуска демона потребуется совершить вызов `opcontrol` с параметром `start`. При первом вызове `opcontrol` следует указать, желаете ли вы выполнить профилирование ядра или же собираетесь поработать в пространстве пользователя.

`opcontrol` сохраняет последние использовавшиеся параметры в домашнем каталоге с именем `.oprofile`. При следующем запуске `OProfile` вам не потребуется вводить какие-либо параметры, кроме `--start`, так как данный инструмент сохраняет их в вашем домашнем каталоге. Запускать и приостанавливать работу демона может только корневой пользователь *root*, поэтому данная информация располагается в домашнем каталоге именно такого пользователя. Простые пользователи могут только осуществлять сброс статистических данных при помощи параметра `-dump`, но не будем слишком забегать вперед.

Когда демон запускается, он начинает собирать статистические данные обо всем, что протекает в системе, — *без исключения*. На первый взгляд, итоговый вывод может показаться чересчур громоздким. Его может просматривать любой пользователь, однако предварительно необходимо сбросить статистические сведения при помощи параметра `-dump`. После этого их можно просматривать посредством инструмента `opreport`, `opgprof` или `opannotate`. Сброс статистических сведений или вызов `opreport` может совершать любой пользователь, а не только корневой пользователь *root*.

`oprofiled` начинает записывать сведения начиная с момента, когда пользователь вводит команду `opcontrol --start`. Посредством вызова `opcontrol -dump` можно сгенерировать моментальный снимок статистических сведений, однако при этом `oprofiled` продолжит собирать и аккумулировать данные. Последующие вызовы `opcontrol -dump` сделают свежие аккумулированные данные доступными для изучения. Только когда пользователь введет команду `opcontrol -stop`, демон `oprofiled` перестает собирать данные.

Профилирование приложения при помощи OProfile. Этапы профилирования приложения довольно просты, однако должна строго соблюдаться очередность этапов, если вы хотите получить приемлемый результат. Поскольку `oprofiled` аккумулирует данные непрерывно, важно своевременно производить очистку и выполнять сброс статистических данных. Для получения корректных результатов необходимые процедуры нужно проводить на бездействующей системе, в противном случае другие выполняющиеся процессы могут повлиять на итоговый результат. Существуют следующие этапы:

- запуск демона/начало сбора данных;
- очистка буфера статистики (при необходимости);
- запуск приложения;
- сброс статистических данных и/или приостановка демона.

После того как вы собрали статистические данные при помощи соответствующей программы, можете приступать к работе с инструментами отчетности. Профилированию может подвергаться приложение, которое было скомпилировано без применения каких-либо специальных флагов. Так, например, вы можете выполнить компиляцию программы из листинга 9.4 с включенной оптимизацией и без дополнительных флагов и получить при этом приемлемый результат:

```
$ cc -o profme -O2 -lm
$ opcontrol --start -no-vmlinux          Начало сбора статистических данных
Using default event: GLOBAL_POWER_EVENTS:100000:1:1:1
Using 2.6+ OProfile kernel interface.
Using log file /var/lib/oprofile/oprofiled.log
Daemon started.
Profiler running.
$ ./profme                               Запуск приложения, которое будет
                                         подвергаться профилированию
$ opcontrol dump                         Запись моментального снимка
                                         статистических данных для того,
                                         чтобы затем можно было запустить
                                         opreport
$ opreport
Counted GLOBAL_POWER_EVENTS events (time during which processor is not stopped)
with a unit mask of 0x01 (mandatory) count 100000
GLOBAL_POWER_E...|
  samples|      %|
------------------
    16477 52.9807 libm-2.3.3.so
    11597 37.2894 profme
     1795  5.7717 no-vmlinux
      325  1.0450 libc-2.3.3.so
      162  0.5209 bash
      140  0.4502 libglib-2.0.so.0.400.8
      103  0.3312 oprofiled
```

Вывод `opreport` свидетельствует о том, что «местом», где приложение тратит основную часть своего времени при выполнении, является стандартная математическая библиотека `libm`. Это вполне ожидаемо, поскольку нам известно, что самые «медленные» функции располагаются именно здесь. Разница заключается в том, что мы осуществляем запуск полностью оптимизированного исполняемого файла, который не был инструментирован тем или иным образом.

Однако это далеко не все, что мы можем сделать. В состав пакета `OProfile` входит еще один инструмент, `opgprof` — который создает файл `gmon.out`, используемый утилитой `gprof`. Это требует дополнительного этапа:

```
$ opgprof profme                         Создание файла gmon.out на основе
                                         сохраненных статистических данных
$ gprof --no-graph profme                В итоговом файле gmon.out отсутствуют
                                         данные графа вызовов
Flat profile:

Each sample counts as 1 samples.
  %   cumulative   self              self     total
 time   samples   samples     calls  T1/call  T1/call name
 70.37   8161.00  8161.00                              slow
 22.78  10803.00  2642.00                              slower
  5.40  11429.00   626.00                              slowest
  1.41  11593.00   164.00                              main
  0.03  11597.00     4.00                              _init
...
```

Необходимо отметить ряд моментов данного вывода. Первый из них заключается в том, что здесь отсутствуют показатели времени. Все базируется исключительно на значениях счетчиков образчиков. Поскольку эти образчики (и проценты времени) отражают все, что протекает в системе, они могут не совсем точно информировать пользователя о том, сколько времени отнимает выполнение его программного кода. Если вы собираете подобные данные в нагруженной системе, где ваш процесс выполняется с низким приоритетом, данный профиль может быть некорректным.

Также следует отметить, что в данном выводе отсутствуют какие-либо счетные показатели, касающиеся вызовов. Нам известно, что демон `oprofiled` собрал 8161 образчик, когда центральный процессор выполнял функцию `slow`, однако непонятно, сколько раз данная функция на самом деле подвергалась вызову. Таков «побочный эффект» того, что инструмент `OProfile` охватывает всю систему в целом, а не один лишь процесс.

Нами также был использован параметр `--no-graph`, поскольку в файле `gmon.out`, генерируемом инструментом `opgprof`, отсутствуют данные графы вызовов. Как и в случае с обычной инструменированной программой, если вам требуется построчное профилирование, то компилирование исполняемого файла необходимо осуществлять с активированной отладкой (`-g`). В качестве примера снова обратимся к листингу 9.5:

```
$ cc -g -O2 -o summer-proj summer-proj.c -lm
$ opcontrol --reset                          Очистка буфера статистики
Signalling daemon... done
$ opcontrol --start
Profiler running.
$ ./summer-proj                              Запуск программного кода
$ opcontrol --dump                           Демон получает указание на сброс
                                             статистических данных.
$ opgprof ./summer-proj                      Создание файла gmon.out для gprof
$ gprof --no-graph -l ./summer-proj | head
Flat profile:

Each sample counts as 1 samples.
  %   cumulative   self              self     total
 time   samples   samples    calls  T1/call  T1/call name
 46.81   3496.00   3496.00              main (summer-proj.c:19 @ 8 048444)
 20.10   4997.00   1501.00              main (summer-proj.c:12 @ 8 0483fb)
 14.82   6104.00   1107.00              main (summer-proj.c:19 @ 8 04842c)
  8.19   6716.00    612.00              main (summer-proj.c:22 @ 8 048410)
  4.69   7066.00    350.00              main (summer-proj.c:21 @ 8 0483ea)
```

Здесь мы наблюдаем уже знакомый нам построчный профиль. И вновь показатели времени отсутствуют — в наличии только значения счетчиков образчиков. Однако `OProfile` может «улучшить» вывод `gprof` путем предоставления аннотированных исходных листингов с помощью `opannotate`. При этом даже не потребуется задействовать инструмент `opgprof`:

```
$ opannotate --source ./summer-proj          Вывод отправляется на stdout
                                             (см. далее)
/*
```

```
 * Command line: opannotate --source ./summer-proj
*
 * Interpretation of command line:
* Output annotated source file with samples
* Output all files
*
 * CPU: P4 / Xeon, speed 1700.38 MHz (estimated)
* Counted GLOBAL_POWER_EVENTS events (time during which processor is not
stopped) with a unit mask of 0x01 (mandatory) count 100000
*/
/*
 * Total samples for file : "/home/john/examples/ch-07/prof/summer-proj.c"
*
 *   7458 99.8527
 */
               :#include <stdio.h>
               :#include <string.h>
               :#include <stdlib.h>
               :#include <time.h>
               :#include <math.h>
               :
               :volatile double x;
               :
               :int main(int argc, char *argv[])
               :{ /* main total:   7458 99.8527 */
               :    int i;
  1554 20.8060 :    for (i = 0; i < 16000000; i++) {
    27  0.3615 :        x = 1000.0;
               :
               :        /* 0 <= r < 16 */
     6  0.0803 :        int r = i & 0xf;
               :
   115  1.5397 :        if (r <= 8) {
  4678 62.6322 :            x = pow(x, 1.234); /* Подвергается вызову 9/16
                                                  времени */
               :        }
   350  4.6860 :        else if (r <= 11) {
   612  8.1939 :            x = sqrt(x);       /* Подвергается вызову 3/16
                                                  времени */
               :        }
               :        else {
   116  1.5531 :            x = 1.0 / x;       /* Подвергается вызову 4/16
                                                  времени */
               :        }
               :    }
               :}
```

Здесь вы можете наблюдать лучшее из того, на что способны `gcov` и `gprof`. Каждая строка аннотирована значением счетчика образчиков, то есть указывается, сколько раз демон `oprofiled` смог «поймать» процессор за выполнением этой строки кода. Рядом с данным значением указывается процент времени, потраченный

на выполнение соответствующей строки кода. Эти процентные значения нормализуются относительно файла, обрабатываемого в текущий момент, поэтому они могут увеличиваться вплоть до 100 % независимо от того, что еще могло выполняться в то же время.

Заключительные положения об инструменте OProfile. Вы познакомились лишь с малой частью возможностей `OProfile`. Так, например, мы не рассматривали события других типов, мониторинг которых можно осуществлять при помощи данного инструмента. Конкретные события специфичны для различных процессоров, при этом вы можете узнать, какие из них поддерживаются вашим процессором, с помощью ввода следующей команды:

```
$ opcontrol --list-events
```

Итоговый перечень должен быть внушительным. Событие по умолчанию хорошо подходит при профилировании приложений, однако в случае с профилированием системы вам потребуются знания о других доступных событиях. Более подробную информацию можно отыскать на домашней странице инструмента `OProfile`[1].

9.4. Производительность многопроцессорных систем

Если операционная система может использовать ресурсы более чем одного процессора, установленного в компьютере, мы говорим, что она обладает поддержкой *симметричной многопроцессорной обработки SMP*, которая существует уже довольно продолжительное время[2]. При использовании симметричной многопроцессорной обработки SMP общая производительность системы, выполняющейся на нагруженном сервере, возрастает, поскольку планировщик может распределять процессы между несколькими процессорами. В то же время некоторые однопоточные приложения могут использовать ресурсы только одного процессора в каждый конкретный момент. Наибольший выигрыш от симметричной многопроцессорной обработки SMP получают многопоточные приложения, которые позволяют операционной системе распределять потоки между разными процессорами и тем самым обеспечивать более высокий уровень производительности.

9.4.1. Типы аппаратного обеспечения с поддержкой симметричной многопроцессорной обработки SMP

Симметричная многопроцессорная обработка SMP реализуется на основе нескольких подходов, к тому же появляются все новые. Каждый из них имеет как преимущества, так и недостатки. Понимание того, в чем они заключаются, поможет

---

[1] См. сайт http://oprofile.sourceforge.net.

[2] Версия Linux 2.0, вышедшая в 1996 году, стала первой, которая поддерживает SMP-обработку.

вам разрабатывать программы таким образом, чтобы они могли получить максимальную выгоду от использования многопроцессорных систем.

Перед тем как приступить к подробному изучению реализаций SMP-обработки, нам необходимо рассмотреть ряд основополагающих понятий, которые будут использоваться в дальнейшем. Если вы уже владеете соответствующей терминологией, то можете пропустить следующие разделы.

Параллелизм в центральных процессорах

Для увеличения скорости работы и без того весьма мощных процессоров производители прибегают к различным уловкам. Их понимание поможет вам осознать, в чем заключается важность существования многоядерных процессоров с поддержкой симметричной многопроцессорной обработки SMP.

Сравнение архитектур RISC и CISC. Трудно представить, однако было время, когда разработчики избегали использования компиляторов. Программисты писали код на ассемблере, даже если был доступен компилятор. Ассемблер считался самым эффективным языком, при этом большинство программистов полагало, что компиляторы не могут способствовать созданию эффективного и безошибочного программного кода.

В те времена основная часть процессоров относилась к архитектуре CISC (Complex Instruction Set Computers — *компьютер с комплексным набором инструкций*), которая поддерживала широкий набор инструкций с множеством режимов адресации, облегчавший программирование на ассемблере. Двумя выдающимися представителями архитектуры CISC являются процессоры Motorola 68000 и Intel 8086. Процессор Motorola 68000 использовался в первых компьютерах Macintosh, а чип Intel 8086 — в компьютерах IBM PC[1]. Набор инструкций Intel 8086 сохранился и до наших дней в виде подмножества в архитектуре IA32.

Исследования, проводившиеся компанией IBM, а также университетами Стэнфорд и Беркли, показали, что транзисторы центрального процессора можно более эффективно использовать путем внедрения набора с меньшим количеством инструкций и ограниченными режимами адресации, обеспечивая при этом более низкую латентность инструкций. Это стало побудительным мотивом к созданию процессоров с архитектурой RISC (Reduced Instruction Set Computer — *компьютер с сокращенным набором инструкций*). Если говорить в общем, то инструкции RISC более дружественны к компилятору, нежели к пользователю. К представителям архитектуры RISC относятся такие процессоры, как PowerPC, SPARC и MIPS.

Конвейерная обработка и параллелизм выполнения. Инновацией, позволившей процессорам на основе архитектур CISC и RISC работать более быстро, стало использование конвейерной обработки — компьютеризированного эквивалента широко известной производственной сборочной линии, использовавшейся Генри Фордом в начале XX века для ускорения процесса сборки автомобилей. При разбивке процесса сборки автомобиля на этапы каждый рабочий специализировался на каком-то одном из них. Поскольку все рабочие были специалистами в своей области, они могли более эффективно выполнять свои обязанности, чем если бы каждому из них пришлось заниматься сборкой автомобиля от начала до конца.

---

[1] Формально это был процессор 8088 — 8-битная версия процессора 8086.

Вместо сборки автомобилей конвейер центрального процессора поэтапно выполняет инструкции. Конструкция процессора подразумевает разбивку каждой инструкции на отдельные этапы, которые выполняются независимо друг от друга. Каждый этап «сопровождается» специализированным аппаратным средством, выделенным для него в конвейере. Специализированное аппаратное средство называется *блоком выполнения* (иногда — *суперскалярным* блоком). В данном случае конвейер можно рассматривать как сборочную линию, а блок выполнения — как рабочего.

Однако сходство со сборочной линией на этом заканчивается. Разные инструкции подразумевают разные этапы, при этом выполнение некоторых из них требует больше времени по сравнению с остальными. Конвейеры инструкций можно сравнить с одной сборочной линией, на которой одновременно выпускаются автомобили, грузовики, велосипеды и самолеты. Время, которое инструкция затрачивает на прохождение через конвейер, называется *латентностью* инструкции.

Каждая ступень конвейера завершается за один такт центрального процессора, то есть процессору с 20-ступенчатым конвейером может потребоваться совершить до 20 тактов, для того чтобы завершить одну инструкцию. На первый взгляд может показаться, что это много, однако вы не будете так считать, когда узнаете, что процессор может обрабатывать до 20 инструкций за раз, то есть чистая *производительность* составит одну инструкцию за такт. Подобная максимальная теоретическая производительность редко достигается из-за *простаивания* конвейера.

Чаще всего конвейер простаивает из-за инструкций условного перехода. Пока происходит декодирование инструкции перехода и ее условия, за ней в конвейере могут расположиться другие инструкции. В результате перехода эти инструкции могут быть отвергнуты. В данном случае уровень производительности упадет, поскольку конвейер должен быть заполнен до того, как процессор снова начнет выполнять инструкции. От глубины конвейера зависит, насколько сильным будет это падение (рис. 9.3).

Ситуации простаивания конвейера неизбежны: написание программного кода невозможно без использования операторов `if`. Многие процессоры имеют свои особенности, призванные минимизировать эффекты, связанные с переходами, однако итоговый результат будет варьироваться в зависимости от приложения. В общем, конвейер позволяет процессору достигать пика производительности при выполнении инструкций при пакетном чтении. В случае с реальными приложениями обеспечить подобный уровень производительности на постоянной основе не представляется возможным.

Общая черта процессоров на основе архитектуры RISC заключается в том, что они поддерживают набор инструкций с низкой латентностью, предполагая использование более коротких конвейеров инструкций по сравнению с CISC-процессорами. Это означает, что RISC-процессоры не будут так сильно страдать от простаивания конвейера, как процессоры на основе архитектуры CISC. Чем больше глубина конвейера, тем выше должна быть тактовая частота, для того чтобы обеспечивалась низкая латентность, измеряемая в наносекундах. Иногда для повышения тактовой частоты может потребоваться увеличить глубину конвейера, но такой подход является неоптимальным.

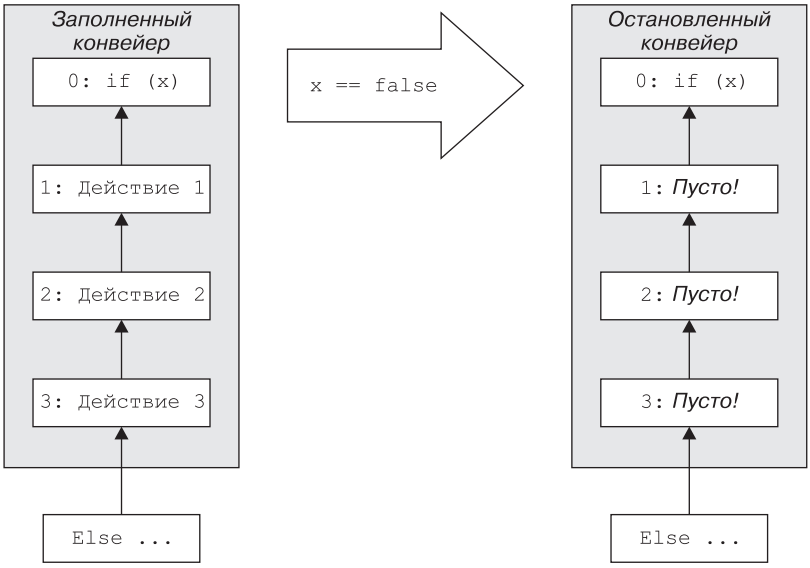


Рис. 9.3. Упрощенный пример ситуации простаивания четырехступенчатого конвейера: когда процессору «известно», что x = false, все другие ступени конвейера должны быть сброшены и заполнены снова

Кэш. О кэше мы говорили в главе 5, поэтому здесь на нем подробно останавливаться не будем. Кэш представляет собой важную часть процессора, позволяющую ему функционировать быстрее, чем в случае использования оперативной памяти. Благодаря кэшу ядро процессора может работать на очень высоких тактовых частотах, превышающих возможности современных модулей оперативной памяти.

Материнские платы, поддерживающие установку нескольких процессоров

Самая первая реализация симметричной многопроцессорной обработки SMP подразумевает размещение на одной материнской плате двух и более процессоров. Эти процессоры совместно используют общую FSB-шину, оперативную память и северный мост (см. рис. 9.1), с чем связано много конструкторских проблем. Прежде всего это выражается в ограничении скорости FSB-шины, поскольку при использовании несколькими устройствами одной общей шины возникают проблемы с качеством сигналов. В результате скорость FSB-шины в самых быстрых многопроцессорных системах обычно оказывается ниже этого показателя в самых быстрых однопроцессорных системах.

Процессоры Opteron от компании AMD обладают встроенным контроллером оперативной памяти, который обеспечивает каждому такому процессору, установленному на многопроцессорной материнской плате, высокоскоростной доступ к оперативной памяти. Кроме того, при использовании чипов Opteron шина FSB заменяется шиной Hypertransport, которая представляется собой пакетный интерфейс типа «точка — точка». Это позволяет устранить множество проблем с качеством сигналов, имеющихся у традиционной FSB-шины, а также дает возможность

компании AMD размещать большее количество процессоров на одной материнской плате.

Недостаток данного подхода заключается в том, что, в отличие от скорости FSB-шины, которая может варьироваться, скорость шины Hypertransport является фиксированной и может возрасти лишь с выходом новой версии данного стандарта. Hypertransport — это общепринятый стандарт, на который ориентируются другие производители электронных компонентов, поэтому изменения в него вносятся нечасто. Встроенный в процессоры Opteron контроллер оперативной памяти имеет тенденцию отставать от темпов развития технологии производства оперативной памяти, поскольку адаптация модулей оперативной памяти, построенных на базе новой технологии, потребует заново перерабатывать конструкцию центрального процессора.

Как видите, каждый подход имеет как преимущества, так и недостатки.

Симметричная многопоточность SMT

Используемый в Linux термин *симметричная многопоточность SMT (Symmetric Multithreading)* характеризует технологию Hyperthreading, поддерживаемую некоторыми моделями процессоров Pentium и Xeon, которые выпускаются компанией Intel. Процессор, в котором реализована технология Hyperthreading, «виден» в операционной системе как многоядерный процессор. Однако процессоры с поддержкой симметричной многопоточности SMT не являются полноценными многоядерными чипами, так как их логические ядра совместно используют ресурсы одного физического процессора и, следовательно, конкурируют за них между собой.

Каждый логический процессор обладает лишь собственным регистром и конвейером инструкций. Такие процессоры совместно используют встроенный в физический чип кэш, а также аппаратный блок управления памятью MMU, TLB-буфер и блоки выполнения. Результатом этого является, например, то, что два логических ядра не могут обрабатывать инструкции вдвое быстрее, как это могут делать два одноядерных процессора.

Симметричная многопоточность SMT является уникальной возможностью реализации параллелизма в процессорах архитектуры CISC. Поскольку глубина конвейера у CISC-процессоров больше, они будут сильнее страдать от простаивания конвейера, чем процессоры RISC, у которых более короткий конвейер. Благодаря наличию нескольких конвейеров в форме нескольких логических процессоров можно поддерживать занятость блоков выполнения при простаивании конвейера.

Технология Hyperthreading (а также симметричная многопоточность SMT) получила такое название из-за того, что она является наиболее эффективным средством ускорения работы программ, поддерживающих многопоточность. Потоки, в отличие от процессов, совместно используют память и элементы таблицы страниц, что делает их идеальными кандидатами на распределение между несколькими логическими процессорами. Поскольку в процессоре с поддержкой симметричной многопоточности SMT имеется только один аппаратный блок управления памятью MMU, потоки получают большее ускорение при выполнении, чем процессы.

Многоядерные процессоры

Производители процессоров достигли потолка в повышении тактовых частот своих продуктов. Проблема, сдерживающая дальнейший рост тактовой частоты,

в основном заключается в количестве тепла, которое выделяют мощные процессоры. При использовании нескольких ядер процессор может работать на пониженной тактовой частоте, но выполнять при этом большее количество программ за аналогичное число тактов. Двухъядерные процессоры функционируют на более низких частотах, но считаются более быстрыми, чем их одноядерные собратья, работающие на более высокой тактовой частоте.

Первое поколение многоядерных процессоров, выпущенных компаниями Intel и AMD, были двухъядерными. Функционально двухъядерный процессор аналогичен двум одноядерным процессорам (например, установленным на одной многопроцессорной материнской плате). Каждое ядро имеет собственные регистры, кэш, конвейер инструкций, блоки выполнения, аппаратный блок управления памятью MMU и т. д. Двухъядерный процессор, в принципе, должен обладать той же производительностью, что и система с поддержкой симметричной многопроцессорной обработки SMP, в которой установлены два одноядерных процессора с одинаковой тактовой частотой.

Ядра будущих двух- и четырехъядерных процессоров будут совместно использовать встроенный кэш определенного уровня, что имеет как недостатки, так и преимущества. С одной стороны, это ограничивает размер кэша, к которому ядро может получить доступ, не вступая в «противоборство» за него с другим ядром. С другой — совместное использование кэша снижает количество тактов, необходимых при синхронизации раздельных кэшей. Некоторые приложения смогут выиграть от совместного использования кэша несколькими ядрами, а некоторые, наоборот, пострадают от этого. Ответить на вопрос о том, какой подход является оптимальным, непросто.

9.4.2. Программирование на компьютере с поддержкой симметричной многопроцессорной обработки SMP

Основной части приложений никогда не потребуется «знать», что они выполняются на компьютере, в котором установлено несколько процессоров. Большинство из них связаны с аппаратным обеспечением и операционной системой. Операционная система отвечает за распределение задач и уравнивание нагрузки на разные процессоры. Однако существует ряд приложений, которым в процессе работы в пространстве пользователя потребуется информация о количестве и типе установленных в системе процессоров, для того чтобы аппаратные ресурсы были задействованы максимально эффективно.

Планировщик Linux и симметричная многопроцессорная обработка SMP

Операционная система Linux получила возможность поддержки симметричной многопроцессорной обработки SMP с выходом ядра версии 2.0. SMP-планировщик рационально распределяет задачи и потоки между разными процессорами, обеспечивая максимально эффективное использование аппаратных ресурсов. Применяемая эвристика основывается на ряде предположений, основным из которых является то, что все установленные в системе процессоры равны. Именно это и подразумевается под словом *симметричная* в термине *симметричная многопроцессорная обработка*.

SMP-планировщик старается обеспечить выполнение процесса на одном и том же процессоре, поскольку (благодаря «ленивому» сбросу содержимого TLB-буфера на диск) существует высокая вероятность того, что он будет способен повторно использовать TLB-элементы. Фактически, если в системе установлен процессор с поддержкой симметричной многопоточности SMT, все это будет напрасным, так как оба логических процессора используют один и тот же аппаратный блок управления памятью MMU и кэш. Это может означать, что процесс будет вынужден ожидать ресурсы определенного процессора, поскольку планировщик полагает, что будет слишком «накладным» ставить процесс в очередь на выполнение на другом процессоре.

Один из способов, посредством которого приложения пользователя могут дать «ниточку» планировщику, является использование *маски привязки*. Планировщик поддерживает маску привязки для любого процесса (и потока), протекающего в системе. Она представляет собой битовую маску, где на каждый процессор, доступный в системе, приходится по одному биту. По умолчанию маска привязки является именно такой, это означает, что процесс может выполняться на любом процессоре.

Если планировщик обнаруживает процесс в «работоспособном» состоянии, он проверяет, на какой процессор можно возложить его выполнение. Затем он сравнивает последний с маской привязки процесса, для того чтобы определить, какой процессор будет выполнять данный процесс. Пользователь может ограничить выполнение процесса на одном или большем числе процессоров настройкой маски привязки соответствующим образом.

По мере совершенствования ядра планировщик постепенно начинает поддерживать определенные технологии. В Linux версии 2.6.7 в дополнение к поддержке симметричной многопроцессорной обработки SMP была внедрена поддержка симметричной многопоточности SMT[1]. Таким образом, планировщик сможет принимать оптимальные решения, когда речь зайдет о том, какие процессоры будут выполнять те или иные задачи.

Использование маски привязки для принуждения выполнения процесса на определенном процессоре

В состав пакета `schedutils` входит команда `taskset`, которая может использоваться для настройки маски привязки для конкретного процесса. Ее можно применить как к выполняющемуся процессу, так и к одиночной команде. Так, например, команда `taskset` применима для того, чтобы заставить процесс выполняться только на первом процессоре в системе с поддержкой симметричной многопроцессорной обработки SMP, для чего необходимо установить бит `0` маски привязки и больше никаких:

```
$ taskset 1 ./myprogram                    Определение значения маски привязки,
                                           равного 1
```

Команду `taskset` в сочетании с параметром `-p` можно использовать для проверки маски привязки выполняющихся процессов:

```
$ taskset -p 1234
pid 1234's current affinity mask: 1
```

[1] В меню конфигурирования ядра данная установка доступна в Processor Type and Features (Тип и характеристики процессора).

Операционная система Linux позволяет любому пользователю проверять маску привязки любого процесса, однако только корневой пользователь *root* имеет право изменять маску привязки процесса, независимо от того, кто является его владельцем.

Когда и зачем следует изменять маску привязки процесса

Всегда, когда это возможно, предпочтительнее оставлять маску привязки без изменений и позволить планировщику Linux заниматься своими прямыми обязанностями.

Изменять маску привязки следует только при определенных обстоятельствах. Примером подобной ситуации является наличие приложения, интенсивно использующего память, выполнять который пользователь желает на каком-то одном процессоре. Несмотря на то что планировщик Linux будет пытаться обеспечить выполнение этого процесса на одном процессоре, нет никаких гарантий, что он на самом деле будет выполняться только одним процессором. Если у вас имеются два таких процесса, протекающих в двухъядерной системе, разумным будет принудить их выполняться на разных процессорах.

API-интерфейс маски привязки процесса

Процессы и потоки могут проверять и модифицировать свои маски привязки посредством системных вызовов, однако, для того чтобы внести в нее изменения, они должны обладать привилегиями корневого пользователя *root*. Процесс может проверять как свою маску привязки, так и аналогичную маску другого процесса при помощи следующих расширений GLIBC:

```
int sched_setaffinity(pid_t  pid,  size_t setsize,  cpu_set_t *cpuset);
int sched_getaffinity(pid_t  pid,  size_t setsize,  cpu_set_t *cpuset);
```

Данные функции в случае успеха возвращают значение 0, а в случае неудачи — значение -1. cpu_set_t представляет собой битовую маску, о которой говорилось ранее, а параметр setsize — размер маски. cpu_set_t определяется для обеспечения битовой маски, которая поддерживает намного больше процессоров, чем может уместиться в unsigned long. В итоге нам потребуются специальные макросы для установки битов в данную маску и удаления их из нее. К ним относятся следующие:

```
CPU_ZERO(p)     – Удаляет маску, указанную посредством p
CPU_SET(n,p)    – Устанавливает бит для процессора n в маске, указанной
                  посредством p
CPU_CLR(n,p)    – Удаляет бит для процессора n в маске, указанной
                  посредством p
CPU_ISSET(n,p) – Возвращает ненулевое значение, если установлен бит n маски,
                  указанной посредством p
```

При вызове одной из функций setaffinity необходимо использовать эти макросы, для того чтобы инициализировать cpu_set_t. Процесс должен обладать привилегиями корневого пользователя *root*, в противном случае функция возвратит значение ошибки. Данные функции не следует применять к потокам. Для этих целей в библиотеке GLIBC существуют расширения для API-интерфейса POSIX pthreads.

API-интерфейс маски привязки потока

GNU-библиотека потоков Native POSIX Threads Library (NPTL) содержит функции для работы с масками привязки потоков. Используя их, вы можете строго задать выполнение потока на одном или более процессорах, доступных в системе. Это может оказаться полезным для максимизации производительности путем принуждения потоков, использующих общую память, к выполнению на каком-то одном процессоре, что может снизить количество неудачных обращений к кэшу.

Стандарт POSIX `pthreads` в настоящий момент не поддерживает работу с масками привязки, поэтому функции, содержащиеся в библиотеке NPTL, являются расширениями, на что указывает суффикс `_np` (не-POSIX). Данные функции определяются следующим образом:

```
int pthread_setaffinity_np(pthread_t tid, size_t setsize, cpu_set_t *cpuset);
int pthread_setaffinity_np(pthread_t tid, size_t setsize, cpu_set_t *cpuset);
```

Для корректной работы эти функции требуют наличия выполняющегося потока, определяемого посредством `tid`-идентификатора. Для того чтобы воздействовать на текущий поток, вызывающий оператор должен передать возвращаемое значение `pthread_self` в качестве значения для `tid`. Если пользователь желает инициализировать маску привязки *до* запуска потока, это можно сделать посредством атрибутов потока. При наличии должным образом инициализированного объекта `pthread_attr_t` маску привязки можно настроить с помощью следующих функций:

```
int pthread_attr_setaffinity_np(pthread_attr_t *attr, size_t setsize, cpu_set_t
*cpuset);
int pthread_attr_getaffinity_np(pthread_attr_t *attr, size_t setsize, cpu_set_t
*cpuset);
```

Обратите внимание на то, что данные функции не принимают в качестве аргумента идентификатор потока. Вызывающий оператор обеспечивает хранилище для `attr` и использует его в качестве аргумента для `pthread_create`. Созданный поток будет обладать маской привязки со значением, которое было определено в атрибутах.

9.5. Заключение

В данной главе мы рассмотрели основы мониторинга и настройки производительности. Кроме того, мы затронули ряд базовых концепций симметричной многопроцессорной обработки SMP, а также изучили необходимые мероприятия, позволяющие приложениям пользователя максимально задействовать ресурсы многопроцессорных и многоядерных систем.

9.5.1. Аспекты, связанные с производительностью

- Конкуренция за ресурсы шины — вы изучили ряд примеров архитектур, которые базируются на шине PCI, и рассмотрели моменты, на которые стоит обращать внимание, для того чтобы выявить конкуренцию за ресурсы шины, которая может возникнуть в системе.

- Прерывания — нами были исследованы архитектура прерываний типичного компьютера, а также сложные моменты, связанные с прерываниями, с которыми может столкнуться программное обеспечение.
- Память, ошибки страниц, подкачка — вы узнали, как эти аспекты влияют на приложения и как их следует оценивать.

9.5.2. Термины, рассмотренные в этой главе

- *Привязка* — методика, используемая операционной системой для принуждения процесса к выполнению на определенном подмножестве процессоров в многопроцессорной системе.
- *Многоядерный* — процессор с более чем одним вычислительным ядром.
- *Многопроцессорный* — компьютер, в котором установлено более одного процессора.
- *Симметричная многопроцессорная обработка SMP (Symmetric Multiprocessing), симметричная многопоточность SMT (Symmetric Multithreading)* — термины, используемые для характеристики операционной системы, выполняющейся на многопроцессорной машине.

9.5.3. Инструментарий, использованный в этой главе

- `gprof`, `gcov` — инструменты для оптимизации программного кода на уровне исходных файлов.
- `OProfile` — мощный системный инструмент (в этой главе вы узнали, как он может использоваться для оптимизации приложений).
- `strace`, `ltrace` — служат для мониторинга поведения программ с минимальным вмешательством.
- `time`, `top`, `vmstat`, `iostat`, `mpstat` — используются для выявления проблем, связанных с памятью и производительностью системы.

9.5.4. Веб-ссылки

- http://sourceforge.net/projects/procps — домашняя страница проекта `procps`, в котором есть множество полезных инструментов для отслеживания ресурсов процессов и системы.
- http://sourceforge.net/projects/strace — домашняя страница проекта `strace`.

9.5.5. Рекомендуемая литература

Dowd K. and Severance C. High Performance Computing, RISC Architectures, Optimization & Benchmarks. — Sebastopol, Calif.: O'Reilly Media, Inc., 1998.

10 Отладка

10.1. Введение

В этой главе мы рассмотрим наиболее распространенные инструменты и методики отладки, используемые в операционной системе Linux.

В былые времена не существовало каких-либо средств отладки и программистам приходилось полагаться почти исключительно на сообщения, выводимые в окне терминала. В некоторых современных встраиваемых окружениях наблюдается острая нехватка ресурсов памяти и центрального процессора, которые можно было бы задействовать для выполнения дополнительных задач. Использование методик отладки всегда оправданно, независимо от того, с какой средой имеет дело пользователь. В этой главе вы узнаете, какие методики необходимо применять для того, чтобы максимально эффективно распорядиться сведениями, содержащимися в выводимых на экран сообщениях.

Помимо этого, я подробно расскажу вам об особенностях отладчика `gdb` и приведу соответствующие примеры. Несмотря на то что существуют превосходные графические пользовательские интерфейсы GUI, расширяющие функциональность отладчика `gdb`, текстовый интерфейс по-прежнему остается исключительно мощным и полнофункциональным средством. К сожалению, GUI-интерфейсы не позволяют раскрыть все возможности отладчика `gdb`. А многие программисты не уделяют времени изучению текстового интерфейса, который все еще доступен в GUI-версиях соответствующих инструментов.

В данной главе мы проведем сравнение ряда инструментов, используемых для проверки памяти, и обсудим их особенности и ограничения. А в завершение поговорим о некоторых нестандартных методиках, которые можно применить в случае, если прочие методики отладки не принесли желаемого результата.

10.2. Основной инструмент отладки: printf

Многие программисты прибегают к использованию инструмента `printf` из-за лени. Когда объем программного кода невелик, легче ввести пару операторов `printf`, чем задействовать отладчик, устанавливать точки останова и возиться с переменными. Во встраиваемой системе может оказаться сложным, а то и вовсе невозможным использование отладчика в отношении программного кода. В таком окруже-

нии выводимые на экран сообщения будут единственной методикой, которую можно применить для отладки. Кстати, именно таким образом разработчики ядра и осуществляют большинство мероприятий, направленных на проведение отладки.

10.2.1. Проблемы, возникающие при использовании printf

Первая проблема, связанная с использованием инструмента `printf`, заключается в том, что он заполняет экран большим количеством информации. Если вы будете полагаться исключительно на данный инструмент, то, скорее всего, вам придется добавить операторы `printf` в различные сегменты своего программного кода. Большинство программистов неохотно удаляют полезные сообщения после того, как они сыграли свою роль: всегда существует вероятность, что они пригодятся в будущем. Внимательный программист должен по крайней мере позаботиться о снабжении сообщений комментариями таким образом, чтобы при этом не загромождался экран, однако нередко возникают ситуации, когда подобные сообщения появляются на экране с пугающей частотой.

Влияние инструмента printf на производительность

Оператор `printf` в зависимости от используемого устройства вывода может оказать определенное влияние на производительность вашей программы. Псевдотерминалы по-своему помогают пользователю уменьшить это влияние. Если вывод вашего программного кода осуществляется в окно терминала X, то высока вероятность того, что запись выполняется в «глубокий» канал:

```
$ time od -v /dev/zero -N200000
                                    Много, много строк спустя…

0606440 000000 000000 000000 000000 000000 000000 000000 000000
0606460 000000 000000 000000 000000 000000 000000 000000 000000
0606500
real    0m1.257s
user    0m0.052s
sys     0m0.092s
```

Как указано в выводе, выполнение данной команды заняло 1,257 с, что не совсем соответствует действительности. Поскольку выполнение осуществлялось в терминале X, реальное время, которое ушло на прокрутку всех данных, будет немного бóльшим, чем время выполнения программы. Из-за буферизации в терминале текст продолжает прокручиваться и после того, как выполнение программы было завершено. Аналогичная команда, перенаправленная на `/dev/null`, выполняется намного быстрее:

```
$ time od -v /dev/zero -N200000 > /dev/null
real    0m0.059s
user    0m0.048s
sys     0m0.012s
```

Несмотря на то что данный пример является весьма нестандартным, он свидетельствует о том, что вывод текста в любой форме может сказаться на производительности.

Аспекты синхронизации и инструмент printf

При рассмотрении межпроцессного взаимодействия IPC в главе 7 мы затронули проблемы, имеющие отношение к буферизации. Если вывод программы направляется на экран, пользователь считает само собой разумеющимся, что соответствующие текстовые сведения отобразятся там при запуске данной программы. Такое поведение типично для коротких сообщений, которые заканчиваются новой строкой, однако оно меняется, когда вывод перенаправляется на другое устройство. Рассмотрим простой пример:

```
for (i = 0; i < 3; i++) {
    printf("Hello World\n");
    sleep(1);
}
```

При выполнении данной команды вывод `Hello World` будет осуществляться раз в секунду на протяжении 3 с. Теперь представим, что мы занимаемся отладкой данной программы и при этом нам необходимо сохранить итоговый вывод в файл, пока мы будем его просматривать. Для этого воспользуемся командой `tee`. Однако здесь нас ожидает сюрприз:

```
$ ./hello | tee hello.out
```

В течение первых 3 с мы не увидим ничего, а затем появятся три строки, содержащие фразу `Hello World`. Если вы ожидали увидеть сообщения при выполнении оператора `printf`, то вы ошибаетесь. Проблема заключается в том, что стандартная библиотека C использует потоки, основанные на дескрипторах плоских файлов. Данные потоки применяются для стандартного ввода/вывода любого файла, использующего указатель `FILE*`. Библиотека C поддерживает буфер для каждого потока и использует разные стратегии буферизации в зависимости от того, является ли терминал устройством вывода. Когда устройством вывода выступает терминал, файловые потоки используют так называемую *строчную буферизацию*. В результате этого поток откладывает запись символов на устройство до того момента, пока не встретится новая строка. Затем все соответствующие символы, включая новую строку, выводятся на экран. Поскольку большинство выводимых на экран сообщений содержат новую строку, это может навести пользователя на ошибочную мысль о том, что операторы `printf` будут оставаться синхронизированными все время. Если устройством вывода является не терминал, символы не будут отправляться на данное устройство до тех пор, пока буфер будет оставаться заполненным или программа явным образом не сбросит содержимое буфера на диск посредством `fflush`.

Буферизация и файловые потоки C

Отправка символов на устройства блоками всегда более эффективна, чем поодиночке. Буферы пространства пользователя позволяют драйверу отправлять

символы на устройство вывода блоками. Поскольку терминальные устройства зачастую оказываются символьными устройствами, они не используют буферизацию так, как это делают блочные устройства (например, дисковый накопитель). В качестве альтернативного средства в данном случае используется буфер пространства пользователя.

Буферизация не всегда должным образом работает с терминалами, поскольку люди привыкли видеть результат на экране сразу после того, как они нажали клавишу, а не после того, как введут некоторое количество байт. В силу этих причин библиотека C позволяет приводить стратегию буферизации в соответствие с типом устройства, подключенного к файловому потоку. Файловые потоки C поддерживают использование трех базовых стратегий, которые можно выбирать при помощи функции `setvbuf`, определяемой следующим образом:

```
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

Аргумент `steam` указывает функции на то, какой поток желает модифицировать пользователь, — например, `stdout`. Аргументы `buf` и `size` позволяют предусмотреть наличие собственного буфера работы с потоками. Существует ряд приложений, для которых это будет желательным, однако проще разрешить библиотеке C самой заняться распределением буфера. В качестве параметров по умолчанию вы можете передать `NULL` как значение для `buf` и `0` — для `size`. Однако наиболее примечательным аргументом является `mode`, который может принимать одно из следующих значений:

- `_IONBF` — буферизация не используется. При работе с потоком буферизация не применяется. Запись символов осуществляется поочередно;
- `_IOLBF` — используется строчная буферизация. Символы буферизуются до первого символа новой строки;
- `_IOFBF` — используется полная буферизация. Символы буферизуются до полного заполнения буфера.

`stdout` по умолчанию использует строчную буферизацию (`_IOLBF`), если подключение осуществляется к терминалу. Если перенаправить `stdout` в файл или канал (как это было сделано ранее), такое поведение изменится и будет использоваться уже полная буферизация (`_IOFBF`). В большинстве случаев это как раз то, что вам и требуется. Влияние на производительность из-за записи в терминал минимизируется посредством использования буферизации.

Если вы решите принудительно обеспечить синхронность вывода информации, то у вас есть два пути. Первый заключается в задании возврата к режиму использования строчной буферизации при помощи `setvbuf`:

```
setvbuf(stdout,NULL,_IOLBF,0);
```

Вы также можете вообще отключить буферизацию при помощи `_IONBF`, однако в любом случае выполнение программного кода будет осуществляться медленнее, чем при использовании буферизации.

Второй способ задания синхронности вывода информации состоит в сбросе содержимого буфера на диск вручную при помощи команды `fflush`. Преимущество данного способа заключается в том, что вы можете нацелиться на сброс на диск

определенных итоговых данных и извлечь выгоду из использования буферизации в других ситуациях. Вы, к примеру, можете захотеть, чтобы сообщение выводилось, когда счетчик достигает определенного значения, и тут же ознакомиться с ним. В данном случае использование вызова `fflush` один раз будет разумным решением. Недостаток команды `fflush` заключается в том, что она привносит дополнительные строки кода и способствует еще большему заполнению экрана.

Буферизация и файловые системы

В дополнение к буферам пространства пользователя, предусматриваемым библиотекой C, файловая система поддерживает буферы в ядре. На рис. 10.1 вы можете увидеть диаграмму, на которой приведены местоположение буферов, а также библиотечные вызовы, посредством которых между ними перемещаются данные. Обратите внимание на то, что сброс на диск содержимого буфера пространства пользователя при помощи `fflush` не вызывает выполнения аналогичной операции в отношении содержимого буфера файловой системы. Данные не записываются на диск до тех пор, пока система не решит, что это нужно сделать, или приложение не вызовет `fsync`.

Если программа, которую вы используете для просмотра файла, выполняется в той же системе, где располагается этот файл, это не будет иметь особого значения. Кэш файловой системы «видим» для всех процессов, протекающих в системе, благодаря чему каждый из них может получить доступ к содержащимся в кэше данным, как если бы они были записаны на диск. Они могут просто оказаться пока что не записанными на физический дисковый накопитель.

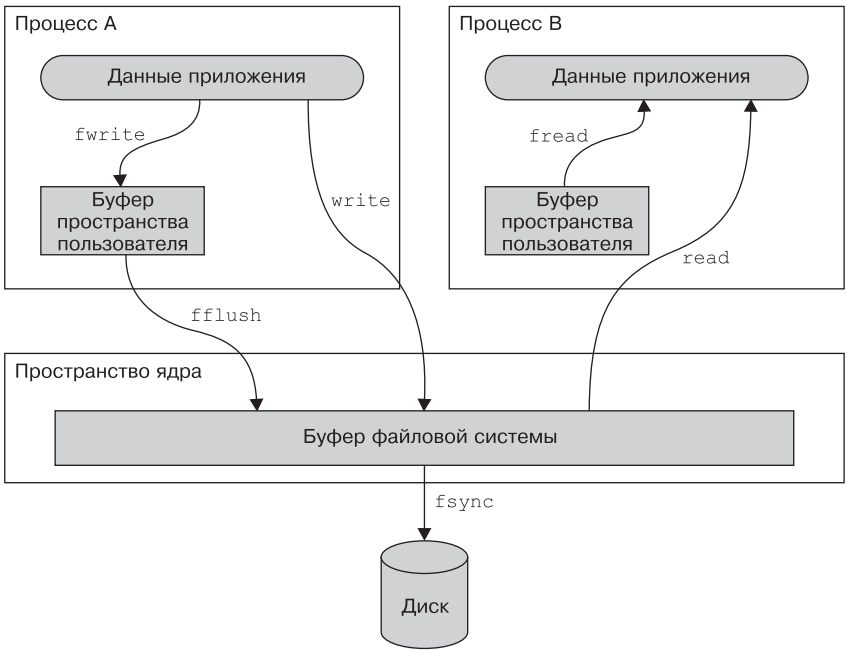


Рис. 10.1. Диаграмма операций ввода/вывода при использовании буферов

Буфер файловой системы может стать проблемой, когда вы решите просмотреть файл с другого компьютера. Представьте, что дисковый накопитель с интересующими нас данными находится на удаленном сервере NFS. В этом случае нам будут доступны лишь результаты самых последних операций записи на такой накопитель. Данные, располагающиеся в кэше файловой системы на другом клиенте, будут нам «не видны». С подобной проблемой можно столкнуться лишь при очень специфических обстоятельствах, однако если это все же произошло, то вы можете форсировать запись обновленных данных в файловую систему при помощи функций `fsync` и `fdatasync`, которые определяются следующим образом:

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

Обе эти функции обеспечивают запись на соответствующее устройство пользовательских данных, содержащихся в кэше файловой системы. Они блокируют вызывающий оператор до тех пор, пока драйвер устройства не сообщит, что данные записаны на устройство[1]. Разница между ними заключается в том, что функция `fdatasync` задает только запись на устройство пользовательских данных, в то время как `fsync` — еще и обновление метаданных файловой системы.

Необходимо отметить, что данные функции принимают в качестве аргумента файловый дескриптор, а не файловый поток, то есть они не являются заменой вызовов `fflush` и `setvbuf`, используемых в случае с файловыми потоками, а служат дополнением к ним. Извлечь файловый дескриптор любого файлового потока C можно при помощи функции `fileno`.

10.2.2. Эффективное использование инструмента printf

Если исходить из изложенного, то, возможно, наиболее важная проблема заключается в определении того, когда инструмент `printf` лучше не использовать. Если избежать его применения не получается, то по крайней мере можно деактивировать его, когда он не требуется.

Помощь препроцессора

Препроцессор C может оказаться весьма полезным при форматировании и управлении отладочными сообщениями. Один из шаблонов, которым рекомендуется следовать, заключается в использовании макроса в качестве обертки для вызовов `printf`, что может способствовать меньшему заполнению экрана текстовой информацией и позволит без особых усилий удалять сообщения, когда потребность в них исчезнет. Рассмотрим пример:

---

[1] Жесткие диски также обладают внутренним кэшем, содержимое которого может оказаться не сброшенным непосредственно на накопитель, даже когда задействуется соответствующая функция. Стандартного способа управления внутренним кэшем жестких дисков не существует.

```
#ifdef DEBUG
#define DEBUGMSG(...) printf(__VA_ARGS__)    /* Используется C99 / GNU-
                                                расширение */
#else
#define DEBUGMSG /* nop */
#endif
```

Использование списков переменных аргументов в макросах начиналось как GNU-расширение, однако было перенято также стандартом C99. Если вы используете не-GNU-компилятор или такой, который не поддерживает расширения C99, можно воспользоваться более грубым альтернативным синтаксисом, который позволяет получить эффект списка переменных аргументов:

```
#define DEBUGMSG(msg) printf msg        /* Вызывающий оператор задействует
                                           круглые скобки */
/* Это возможно лишь таким образом... */
DEBUGMSG(("Hello World %d\n",123));     /* Превращается в printf("Hello World
                                           %d\n",123); */
```

Обратите внимание на то, что аргументы заключены в круглые скобки. Внутренние круглые скобки стали частью аргумента макроса, и именно по этой причине они приведены в определении. Все это выглядит немного неуклюже и некрасиво и трудно поддается толкованию, однако это отличный заменитель списков переменных аргументов в макросах.

Использование макроса в качестве обертки для вызовов `printf` позволяет в значительной степени скрывать «некрасивый» программный код, который в иной ситуации может оказаться полезным. В любое сообщение можно включить файловое имя и номер строки:

```
#define DEBUGMSG(fmt,...) printf("%s %d " fmt, __FILE__,__LINE__, ## __VA_ARGS__)
```

Двойной знак фунта ##, использованный в данном контексте, представляет собой еще одно GNU-расширение. Обычно он используется для конкатенации строк в макросе, однако в этом контексте GNU предусматривает для него другое значение. При использовании в сочетании с `__VA_ARGS__` он удаляет стоящую в конце запятую, когда вы используете макрос вместе со строкой формата, которая не принимает никаких аргументов. То есть если `__VA_ARGS__` пуст, то компилятор `gcc` сам удалит стоящую в конце запятую. Без этого расширения нам пришлось бы вводить минимум один аргумент даже в том случае, если строки формата не принимают никаких аргументов.

Следует также отметить, что строка формата `"%s %d"` конкатенирована с аргументом `fmt` компилятором C, а не с препроцессором C. Поддержка конкатенации фиксированных строк наподобие упомянутой была внедрена в C еще со времен оригинального стандарта ANSI (1989 г.).

Хитрые приемы с использованием препроцессора. Далее приведены мои излюбленные `printf`-приемы с использованием препроцессора C.

Вывод переменных на экран при минимуме нажатий на клавиатуре

```
#define PHEX(x)printf("%#10x %s\n", x, #x)
...
```

```
PHEX(foo);
PHEX(bar);
PHEX(averylongname);
```

В данном случае хитрость заключается в том, что символ # используется для заключения аргумента макроса в кавычки. Препроцессор преобразует аргумент, перед которым имеется знак #, путем заключения его в кавычки. Это позволяет избежать лишних нажатий на клавиши. Таким образом, строка

```
PHEX(averylongname);
```

преобразуется в

```
printf("%#10x %s\n", averylongname, "averylongname");
```

Второй хитрый прием подразумевает размещение значений в полях с фиксированным отступом слева. Я считаю, что вывод в таком формате более удобочитаем, поскольку имена переменных постоянно варьируются, однако значения всегда выстраиваются по порядку, например:

```
       0x1 foo
       0x2 bar
0xdeadbeef averylongname
```

В этом примере я использовал шестнадцатеричный формат, однако вы можете воспользоваться аналогичным приемом для генерирования вывода в любом желаемом формате. Весь фокус заключается в правой стороне строки формата.

Встраиваемая синхронизация. Основываясь на материале о синхронизации, изложенном ранее, также можно использовать макрос в качестве обертки для «некрасивого» программного кода, для того чтобы не слишком загромождать экран информацией. Решение в данном случае может выглядеть следующим образом:

```
#define DEBUGMSG(...) \
        do { \
        printf(__VA_ARGS__);\
        fflush(stdout);\
        } while(0)
```

В данный блок кода вы можете вставлять все, что пожелаете. Если он будет разрастаться со временем, это может сказаться на объеме вашего программного кода. Возможно, вам придется взамен этого создать функцию.

Использование do/while(0) в макросах

Данный шаблон активно используется в исходных файлах ядра Linux. Когда макрос содержит более одного оператора, его определение без заключения в скобки может быть рискованным. Допустим, нам необходимо, чтобы перед завершением программы на экран выводилось сообщение. Приведенный далее макрос синтаксически корректен, однако содержит дефект:

```
#define EXITMSG(msg)  printf(msg); exit(EXIT_FAILURE)
```

Компиляция этого макроса пройдет успешно, однако при использовании на практике он, скорее всего, будет неправильно функционировать. Обратимся к следующему примеру:

```
if ( x != 0 ) EXITMSG("x is not zero \n");
```

После обработки препроцессором итоговый программный код будет выглядеть так:

```
if ( x != 0 ) printf("x is not zero \n");
exit(EXIT_FAILURE);
```

При использовании подобным образом вызов `exit` будет безусловным. Программа всегда будет завершаться в этой точке, а это далеко не то, что нам нужно. Доверившись интуиции, откорректируем наш макрос следующим образом:

```
#define EXITMSG(msg)  { printf(msg); exit(EXIT_FAILURE); }
```

Использование дополнительных скобок позволит данному макросу функционировать подобно единому оператору. В большинстве случаев так оно и будет, однако в некоторых ситуациях могут возникнуть проблемы, например:

```
if ( x != 0 )
   EXITMSG("x is not zero \n");
else
   printf("no problem \n");
```

Компиляция окажется неудачной. После преобразования программный код будет выглядеть следующим образом:

```
if (x != 0)
   { printf("x is not zero \n"); exit(1); };
else                                /* Здесь содержится синтаксическая
                                       ошибка! */
   printf("no problem \n");
```

Точка с запятой, идущая после заключительных скобок, завершает выполнение оператора `if`. Предложение `else` воспринимается компилятором как отдельный оператор, что, конечно же, неверно.

Для того чтобы устранить дефект, необходимо разместить заключенный в скобки программный код внутри оператора `do/while`. Поведение блока `do/while` направлено на выполнение кода, содержащегося в этом блоке, минимум один раз.

Условие (находится после ключевого слова `while`) оценивается после выполнения блока. Таким образом, если используется условие со значением `0` ***(false)***, блок будет выполняться только один раз. После переопределения макрос будет выглядеть так:

```
#define EXITMSG(msg) \
      do { printf(msg); exit(EXIT_FAILURE); } while(0)
```

Программный код, содержащийся внутри блока, идеально инкапсулирован, благодаря чему он может использоваться в любом контексте. Поскольку условие является литералом, оптимизатор может избежать включения любого совершающего цикл программного кода, который в противном случае ему пришлось бы сгенерировать.

Использование функции-обертки

Подход к управлению генерируемым выводом, основанный исключительно на использовании препроцессора, имеет ряд недостатков. В самом последнем примере мы воспользовались макросом в качестве обертки для блока программного кода. Это примитивная форма встраивания функций. Недостаток в данном случае заключается в том, что макрос не имеет подписи функции, то есть в нем отсутствует проверка аргументов. Любые синтаксические ошибки, которые будут возникать при вызове подобных макросов, могут привести к генерированию вводящих в заблуждение сообщений.

В качестве альтернативного решения можно использовать функцию как обертку для упомянутого ранее программного кода. Такой подход может оказаться не очень привлекательным на некоторых архитектурах, поскольку это требует дополнительных «накладных расходов», связанных с вызовами функции. Данный недостаток минимизируется благодаря ключевому слову `inline`, поддерживаемому языком программирования C++, стандартом C99 и компилятором `gcc`. *Встраиваемая функция* — это функция, которая на самом деле не подвергается вызову. Вместо этого каждый раз, когда компилятор сталкивается с вызовом встраиваемой функции, он размещает скомпилированные инструкции в данном сегменте программного кода. В действительности функция генерирует тот же программный код, что и макрос, за тем исключением, что здесь появляется подпись функции, обеспечивающая расширенную проверку ошибок и вывод предупреждающих сообщений. Встраивание позволяет избежать «накладных расходов», связанных с вызовом функции, однако увеличивает объем программного кода, поскольку скомпилированные инструкции размещаются не в одном, а во многих сегментах объектного файла. По этой причине встраивание рассматривается компилятором как рекомендация, а не директива. Оптимизатор волен использовать собственную эвристику при решении вопроса о том, когда встраивание будет целесообразным[1].

Вторым препятствием, которое предстоит преодолеть при использовании функции-обертки, являются списки переменных аргументов. Функция `printf` использует такой список для поддержки строк формата, которые могут принимать любое количество аргументов. API-интерфейс для использования переменных аргументов определяется в `stdarg.h`. Простая обертка выглядит следующим образом:

```
inline int myprintf( const char *fmt, ... )
{
    int n;
    va_list ap;              va_list содержит сведения, необходимые
                             для API-интерфейса
    va_start(ap, fmt);       Здесь указывается, где переменные аргументы
                             берут свое начало (то есть после fmt)
    n = vprintf(fmt,ap);     vprintf принимает строку формата и va_list
    va_end(ap);              Данный вызов необходимо совершить до того,
                             как будет завершено выполнение функции
    return n;
}
```

[1] Компилятор gcc не будет прибегать к встраиванию, если он запущен с параметром -03 или для него не задан параметр -finline-functions.

Данная функция не выполняет никаких особых действий, однако наглядно демонстрирует то, каким образом переменные аргументы можно использовать для создания собственной обертки printf. Ключевой момент заключается в том, что основные функции, которые подвергаются вызову, должны принимать va_list в качестве аргумента. В стандартной библиотеке C содержатся эквиваленты практически всех printf-подобных функций. Их можно узнать по тому, что все они начинаются с v: vprintf, vsprintf, vfprintf и т. д.

Процедура создания функции-обертки для printf имеет недостаток, заключающийся в отключении проверки типов в отношении переменных аргументов. Данной проблемы не существует при использовании макросов, поскольку строка формата представляет собой строчный литерал и никогда не сохраняется в переменной. Таким образом, преобразованный макрос будет содержать вызов printf с форматом в строчном литерале и всеми аргументами. Как вы увидите в дальнейшем, GNU предусматривает наличие расширения, которое позволяет обойти данный недостаток.

printf является исключением, когда речь заходит о проверке переменных аргументов. Обычно компилятор не способен делать каких-либо предположений относительно типа и количества аргументов в списке переменных аргументов. В случае с вызовом printf данные предположения могут быть сделаны на основе строки формата. Компилятор gcc способен осуществлять разбор литерных строк форматов и проверять их правильность на фоне переменных аргументов. vprintf и другие stdarg-дружественные функции не позволяют выполнять проверку строк формата из-за того, что они могут принимать только va_list, где содержатся аргументы для строки формата. Более того, при использовании myprintf функции vprintf необходимо передавать строчную переменную, а не строчный литерал, в силу чего компилятор не сможет осуществить разбор строки формата.

GNU допускает применение проверки формата printf к любой функции посредством использования директивы __attribute__. Данный синтаксис аналогичен используемому в stdio.h для printf и ей подобных. В случае с myprintf атрибут можно добавить в прототип функции следующим образом:

```
inline int myprintf( const char *fmt, ... )
        __attribute__ ((format (printf, 1, 2)));
```

Директива __attribute__ является директивой общего назначения. В приведенном примере компилятору сообщается, что функция подчиняется правилам форматирования printf и что строка формата находится в аргументе 1, при этом первый аргумент формата берет свое начало в аргументе 2. Необходимо отметить, что директива __attribute__ доступна для использования только в GNU-компиляторах и не является стандартной, по этой причине ее нельзя перенести в другие компиляторы.

Важность предупреждающих сообщений, касающихся формата printf

Добавление предупреждающих сообщений, касающихся формата printf, является целесообразным, однако, к сожалению, большинство из них содержат сведения о проблемах с переносимостью, а не только об имеющихся ошибках. Одно из наи-

более распространенных предупреждающих сообщений, с которым вам доведется столкнуться, будет содержать информацию о несоответствии типов целых чисел:

```
warning: int format, long int arg (arg 2)
warning: long unsigned int format, int arg (arg 2)
```

Стандарт C точно не определяет величины для `int` и `long`, поэтому такие сообщения относятся к разряду предупреждающих. В действительности, если говорить о компиляторе `gcc`, практически на всех 32-битных архитектурах применяются одинаковые величины для `long` и `int`. Поскольку `printf` использует переменные аргументы, компилятор относит все небольшие целые числа к `int`, а все числа с плавающей точкой — к `double`. Таким образом, несмотря на то что `printf` обеспечивает поддержку формата для этих различающихся типов, ничего страшного не произойдет, если в итоговом выводе вы увидите неверное число, обусловленное размером аргумента.

Однако существует ряд предупреждающих сообщений, заслуживающих внимания, они могут затеряться на фоне сообщений, о которых говорилось ранее. Между типами данных могут возникнуть несоответствия, что приведет к искажению итогового вывода и даже возможному краху программы.

64-битные типы. Тип `off_t`, к примеру, используется в таких API-интерфейсах POSIX, как `lseek` и `mmap`. Он может быть как 32-, так и 64-битным в зависимости от флагов компилятора. По умолчанию задействуется 32-битный тип `off_t`, однако GNU позволяет заменять его 64-битным посредством использования при компилировании следующего флага:

```
-D_FILE_OFFSET_BITS=64
```

Когда 64-битный тип используется в качестве аргумента для формата целого и длинного целого чисел, итоговый вывод будет некорректным. А что еще хуже, из-за различий в размерах последующие аргументы также будут некорректными или даже приведут к краху вашего приложения. Содержание предупреждающего сообщения в такой ситуации изменяется в зависимости от используемой версии компилятора `gcc`:

```
gcc 3.x: warning: int format, different type arg
gcc 4.x: warning: format '%x' expects type 'unsigned int', but argument 2 has type 'off_t'
```

Подобные предупреждающие сообщения могут выводиться на экран независимо от того, на какой архитектуре осуществляется компиляция — 64- или 32-битной. Тип `long long` используется `gcc` и другими компиляторами для представления 64-битных типов на 32-битных архитектурах. Например, формат для типа `long long` будет `11`:

```
printf("%lld\n", x);
```

Однако в существующей ситуации не так уж легко разобраться. На архитектурах x86_64 компилятор `gcc` способен распознавать 64-битные типы `long` и `long long`. При этом пользователь может и не знать, что `printf` требует, чтобы формат совпадал с типом даже в том случае, если типы имеют одинаковый размер. На архитектуре IA32, к примеру, компилятор `gcc` реализует тип `int64_t` стандарта C99 в виде типа `long long`, однако на архитектуре x86_64 `gcc` использует тип `long`. Таким образом, для обеспечения компиляции на архитектуре IA32 без вывода предупреждающих

сообщений формат printf подразумевает использование "%lld", в то время как на архитектуре x86_64 — "%ld".

Типы с плавающей точкой. Смешение типов с плавающей точкой и целочисленных типов так же небезопасно, как и смешение 64- и 32-битных типов. Все потому, что тип double является 64-битным. Самое интересное заключается в том, что смешение аргументов типа float и double не является проблемой, поскольку стандарт C относит аргумент float к double, если тот имеется в списке переменных аргументов. По той же причине использование float в качестве аргумента для формата целого числа может привести к катастрофе, так как аргумент float относится к double, а все целочисленные аргументы являются 32-битными. Выводимые предупреждающие сообщения также будут разниться в зависимости от используемой версии компилятора gcc:

```
gcc 3.x: warning: int format, double arg (arg 2)
gcc 4.x: warning: format '%d' expects type 'int', but argument 2 has type 'double'
```

Обратите внимание на то, что в обоих случаях аргумент относится к типу float, однако в предупреждающем сообщении, выводимом компилятором, принимается во внимание тот факт, что этот компилятор относит его к типу double.

Строчные типы. Строчные типы являются, пожалуй, наиболее вероятным источником ошибок, поскольку они принимают в качестве аргументов указатели, что открывает широкий простор для возникновения несоответствий. Здесь опять-таки содержание предупреждающего сообщения будет зависеть от используемой версии компилятора gcc:

```
gcc 3.x: warning: format argument is not a pointer (arg 2)
gcc 3.x: warning: char format, different type arg (arg 2)
gcc 4.x: warning: format '%s' expects type 'char *', but argument 2 has type 'int'
gcc 4.x: warning: format '%s' expects type 'char *', but argument 2 has type 'int *'
```

Выполнение программного кода, который провоцирует вывод подобных предупреждающих сообщений, с большой долей вероятности будет заканчиваться крахом. Если на экране отобразится одно из таких сообщений, то вам следует проверить свой программный код и устранить имеющиеся ошибки. Если же вы не выявите ошибок, то попытайтесь применить явное преобразование типов, чтобы это предупреждающее сообщение больше не появлялось.

Советы по созданию оптимальных отладочных сообщений

Следуя определенным правилам форматирования при подготовке выводимых сообщений, вы обеспечите пользователям возможность с первого взгляда понять, в чем заключается проблема, не читая каждую строчку.

- *Следует использовать легко узнаваемый формат.*

  Взрослые пользователи при чтении распознают слова по форме, а не прочитывают каждую букву в отдельности. Можно использовать это, придавая «хорошим» сообщениям одну форму, а «плохим» — другую.

  Создание собственной обертки printf является идеальным решением для применения определенного форматирования к выводимым на экран сведениям. Это позволит вам отделить информацию, имеющую отношение к делу, от той, что в данной ситуации будет неуместна.

- *Каждое сообщение должно состоять из одной строки.*

  Выводимые сообщения могут оказаться малопонятными, если вы попытаетесь втиснуть в одну строку всю необходимую информацию, однако такой подход имеет свои преимущества. Так, например, если программа сбрасывает в файл большое количество информации, для поиска конкретных сообщений можно воспользоваться командой `grep`. Если нужные сведения заключены более чем в одной строке, то может потребоваться прибегнуть к их ручному инспектированию.

- *Все лишнее необходимо свести к минимуму.*

  Всегда хочется добавить работе неформальности, однако когда лишняя информация начинает загромождать экран или, что еще хуже, замедлять работу приложения, необходимо ее сократить.

10.2.3. Заключительные положения по отладке с использованием printf

Как вы могли убедиться, использование `printf` не лишено «побочных эффектов». Один из них, не упомянутый мною, заключается в непреднамеренной синхронизации. Данная проблема чаще всего встречается в многопоточных приложениях, однако может возникнуть и в однопоточных программах. Возможно, вы сталкивались с ситуациями, когда определенная проблема исчезала, если задействовался инструмент `printf`. Такое может произойти, когда благодаря стратегическому размещению `printf` скрывается состояние гонки в многопоточном приложении. В случае с однопоточными приложениями `printf` может привести к тому, что компилятор станет сохранять числа с плавающей точкой в памяти, а не в регистре.

10.3. Комфортное использование GNU-отладчика gdb

`gdb` — это текстовый отладчик, основанный на использовании интуитивно понятных команд. Большинство программистов Linux обладают тем или иным опытом работы с ним. Все команды могут сокращаться — некоторые даже до одной буквы. Зачастую пользователю приходится вводить первые несколько букв соответствующей команды. Отладчику `gdb` требуется достаточное количество букв имени команды, для того чтобы точно идентифицировать ее, поэтому если пользователь не введет необходимое количество букв, `gdb` выдаст подсказку. Так, например, если ввести `sh` в приглашении отладчика `gdb`, то появится следующее сообщение:

```
(gdb) sh

Ambiguous command "sh": sharedlibrary, shell, show.
```

После того как вы введете несколько букв, нажмите клавишу Tab, возможно, это позволит автоматически завершить ввод команды на основе введенных вами букв. Если завершения не произошло, нажмите клавишу Tab еще раз, для того чтобы просмотреть список всех команд, начинающихся с введенных букв, например:

```
(gdb) b<Tab><Tab>
backtrace  break      bt
(gdb) b
```

Я ввел букву b, после чего отладчик предложил мне такие варианты завершения ввода команды, как backtrace, break и bt[1]. Затем я вернулся к приглашению gdb и буквам, которые я ввел. Мне по-прежнему требовалось ввести достаточное количество букв, для того чтобы отладчик gdb смог однозначно идентифицировать команду.

10.3.1. Запуск программ с использованием gdb

При запуске отладчика gdb вы можете указать имя своей программы в командной строке или просто ввести команду gdb и загрузить свое приложение. Любой из приведенных далее шаблонов будет правильным:

```
$ gdb ./hello ... или

$ gdb
(gdb) file ./hello
Reading symbols from /home/john/hello...done.
```

При выполнении данных команд соответствующий файл и содержащиеся в нем символы считываются в память. Последний из приведенных шаблонов позволяет также осуществлять переключение программ без необходимости завершения работы gdb.

Находясь в приглашении отладчика gdb, вы можете при необходимости запустить выполнение своей программы при помощи команды run. Это будет целесообразным, если вам известно, что данная программа потерпит крах. После того как это случится, вы сможете просмотреть трассу стека и узнать, где именно произошел сбой. Нередки ситуации, когда вам может потребоваться пройтись по всему программному коду или установить точку останова.

Для запуска программ с использованием gdb вам понадобятся следующие базовые команды:

- set args — особый вариант команды set для передачи аргументов командной строки. gdb сохраняет передаваемые пользователем аргументы, которые будут использоваться в сочетании с последующими вызовами команды run, но в случае необходимости также можно указать новые аргументы при любом вызове run. Однако для того, чтобы удалить использовавшиеся аргументы, необходимо ввести команду set args;
- run — запускает выполнение программы с самого начала. Программа будет выполняться с указанными пользователем аргументами или теми, что использовались при последнем вызове set args. Выполнение осуществляется до тех пор, пока программа не завершится, не будет отменена или не достигнет точки останова;

[1] Следует отметить, что буква b служит сокращением для команды break, а bt — это сокращение для команды backtrace.

- `start` — аналог `run`, за исключением того, что выполнение программы может останавливаться, как если бы в `main` была установлена точка останова. Это позволяет пошагово осуществлять выполнение программы с самого начала;
- `step` — запускает выполнение какой-то одной строки программного кода. Если в такой строке содержится функция, скомпилированная с включенной отладкой, `gdb` перейдет внутрь тела этой функции. Для того чтобы использовать данную команду, запускать программу необходимо при помощи `run` или `start`;
- `next` — аналог `step`, за исключением того, что `gdb` не переходит внутрь тела функций, независимо от того, были ли они скомпилированы с включенной отладкой или нет;
- `kill` — завешает работу программы. Данная команда не является эквивалентом *системного вызова* `kill`, который посылает сигнал выполняющейся программе. Дополнительные детали можно узнать из пояснения к команде `signal` в следующем разделе.

10.3.2. Остановка и повторный запуск выполнения программы

Выполнение программы можно остановить, нажав сочетание клавиш Ctrl+C. В результате этого `gdb` получает указание остановить текущую программу, но не посылать ей сигнал `SIGINT`, благодаря чему выполнение программы может быть возобновлено с того места, где она была остановлена. Некоторые приложения поддерживают подобный тип отладки без использования точек останова, однако большинство программ на это не способны. Именно для последних в приглашении отладчика `gdb` предусмотрен ряд команд для работы с точками останова.

К базовым командам относятся следующие:

- `break`, `tbreak` — позволяют устанавливать точки останова (`break`) или временные точки останова (`tbreak`). Если не используется никаких аргументов, то в результате применения этих команд точка останова устанавливается на следующей инструкции, подлежащей выполнению; нередко сам пользователь указывает функцию или номер строки, на которой должна произойти остановка. Данные команды могут сопровождаться логическим выражением при создании *условных точек останова* (см. раздел «Использование условных точек останова» далее в этой главе);
- `watch` — эта команда схожа с теми, что служат для установления точек останова, однако в данном случае для мониторинга конкретных областей на предмет изменений используются аппаратные регистры (при наличии таковых). Это позволяет программам выполняться намного быстрее, чем при использовании стандартных точек останова (см. раздел «Использование точек наблюдения» далее в этой главе);
- `continue` — продолжает выполнение программы с того места, где она была остановлена. Команда `continue` поддерживает использование опционального числового аргумента (*N*), посредством которого отладчику `gdb` дается указание игнорировать последующие точки останова $N - 1$. Иначе это звучит так: «Продолжать

выполнение до тех пор, пока процесс не достигнет точки останова в течение *N*-го времени»;

- `signal` — посылает сигнал программе и продолжает выполнение. Данная команда принимает один аргумент, которым может быть номер сигнала или его имя. Список имен сигналов можно просмотреть с помощью ввода команды `info signal`;
- `info breakpoints` — выводит список текущих активных точек останова и точек наблюдения;
- `delete` — удаляет все точки останова. Для удаления какой-то одной точки останова необходимо узнать ее номер при помощи команды `info breakpoints` и передать его команде `delete`.

Синтаксис, используемый при обращении с точками останова

Команда `breakpoint` сокращена до одной буквы `b`, поскольку она довольно часто используется. Аргументом, принимаемым данной командой, всегда выступает адрес инструкции, который можно указать одним из следующих способов:

```
(gdb) b                  Установить точку останова на следующей инструкции
                         в текущем кадре стека
gdb) b foo               Установить точку останова на функции foo
(gdb) b foobar.c:foo     Установить точку останова на функции foo,
                         определенной в foobar.c
(gdb) b 10               Установить точку останова на строке 10 в текущем
                         модуле
(gdb) b foobar.c:10      Установить точку останова на строке 10 в foobar.c
(gdb) b *0xdeadbeef      Установить точку останова по адресу 0xdeadbeef
```

Точки останова могут устанавливаться почти в любой области памяти. Компиляция основного программного кода не требует использования отладки, хотя это может помочь.

Использование условных точек останова

Отладка некоторых программ также возможна с использованием условной точки останова. К примеру, выполнение программы можно остановить на функции `nasty`, когда `setlen` будет превышать `buflen`. Для этого необходимо добавить оператор `if` после точки останова, как показано далее:

```
(gdb) b nasty if setlen > buflen
Breakpoint 1 at 0x804843e: file nasty.c, line 8.
(gdb) run 100
Starting program: /home/john/examples/ch-10/debug/nasty 100
Reading symbols from shared object read from target memory...done.
Loaded system supplied DSO at 0xffffe000
Breakpoint 1, nasty (buf=0x804a008 "", setlen=100) at nasty.c:8
8           return memset(buf, 'a', setlen);
```

В условную точку останова можно заложить любой адрес или условие. Единственное ограничение заключается в том, что любые используемые переменные

должны располагаться в рамках той же области, что и адрес точки останова. Вот пример неправильной условной точки останова:

```
(gdb) b nasty if len > buflen
No symbol "len" in current context.
```

В данном случае `len` представляет собой локальную переменную внутри `main` и лежит вне требуемых рамок при вызове функции `nasty`, из-за чего отладчик `gdb` не «признает» ее. Необходимые рамки области можно точно указать посредством использования соответствующих операций на языке C++. Аналогичная условная точка останова может быть установлена следующим образом:

```
(gdb) b nasty if main::len > buflen
Breakpoint 1 at 0x804845e: file nasty.c, line 8.
```

Следует отметить, что при использовании подобного синтаксиса программный код не обязательно должен быть написан на языке C++.

Установка точек останова с использованием C++

Программы, написанные на языке C++, могут оказаться сложными в отладке. Все эти пространства имен, перегрузки и шаблоны усложняют анализ символов, касающихся точек останова. К счастью, инструмент `gdb` предусматривает наличие «коротких путей», позволяющих упростить отладку.

Попытайтесь произвести отладку программы, приведенной в листинге 10.1. Это окажется особенно непростым занятием из-за длинных имен функций, которые весьма похожи. Программа помещает все их в пространство имен и производит перегрузку одной из них, максимизируя количество нажатий на клавиатуре, которое нам потребуется сделать. `gdb` позволяет использовать клавишу Tab для автоматического завершения ввода любых команд и символов, что придется весьма кстати и даст возможность сократить объем работы на клавиатуре и исключит вероятность опечаток. К сожалению, по той причине, что все функции располагаются в пространстве имен, нам необходимо знать, в каком именно, прежде чем мы сможем воспользоваться клавишей Tab для завершения ввода. Если просто ввести `annoy<Tab>`, это не принесет желаемого результата.

Листинг 10.1. `cppsym.c: такое придется по вкусу только программисту C++`

```
 1 // Три "неудобно" названные функции,
 2 // помещенные в пространство имен для того, чтобы они стали еще более
   // раздражающими
 3 // А в довершение одна из этих функций подвергается перегрузке
 4
 5 namespace inconvenient {
 6     void *annoyingFunctionName1(void *ptr) {
 7         return ptr;
 8     };
 9     void *annoyingFunctionName2(void *ptr) {
10         return ptr;
11     };
12     void *annoyingFunctionName3(void *ptr) {
13         return ptr;
```

```
14     };
15     void *annoyingFunctionName3(int x) {
16         return (void *) x;
17     };
18 };
19
20 // Жаль, что оператор 'using' недоступен в отладчике gdb...
21 using namespace inconvenient;
22
23 int main(int argc, char *argv[])
24 {
25     annoyingFunctionName1(0);
26     annoyingFunctionName2(0);
27     annoyingFunctionName3(0);
28     annoyingFunctionName3((int) 0);
29 }
```

Поскольку данный модуль довольно мал, можно без труда увидеть, что данные функции расположены в пространстве имен. На самом деле такого обычно не бывает. При необходимости можно воспользоваться весьма полезной командой `info`, например:

```
(gdb) info function annoy      Поиск функций, в именах которых встречается
                               слово"annoy"
All functions matching regular expression "annoy":
File cppsym.cpp:
(Файл cppsym.cpp:)
void *inconvenient::annoyingFunctionName1(void*);
void *inconvenient::annoyingFunctionName2(void*);
void *inconvenient::annoyingFunctionName3(int);
void *inconvenient::annoyingFunctionName3(void*);
```

В данном выводе приводится пространство имен наряду со всеми соответствующими именами функций. Теперь, когда известно пространство имен, мы можем установить точку останова, используя клавишу Tab для завершения ввода. Правда, здесь необходимо учитывать еще один момент: использование клавиши Tab подобным образом возможно в ситуации с указанным пространством имен (`inconvenient`), однако здесь имеется ограничение, которое состоит в том, что данная клавиша в отладчике `gdb` не задействует ввод двоеточий, являющихся частью пространства имен. Для решения данной проблемы необходимо начинать ввод имени функции с одинарных кавычек, после чего использовать для его завершения клавишу Tab, как показано далее:

```
(gdb) b 'inc<Tab><Tab>
inconvenient
inconvenient::annoyingFunctionName1(void*)
inconvenient::annoyingFunctionName2(void*)
inconvenient::annoyingFunctionName3(int)
inconvenient::annoyingFunctionName3(void*)
```

Клавиша Tab «работает» до первого двоеточия. Ее повторное нажатие выведет список возможных соответствий. Для того чтобы получить возможность продол-

жать использовать клавишу Tab для завершения ввода, необходимо вручную ввести два двоеточия, после чего такая возможность появится, например:

```
(gdb) b 'incon<tab>              В результате получается b 'inconvenient
(gdb) b 'inconvenient::<Tab>     В результате получается b
                                 'inconvenient::annoyingFunctionName
(gdb) b 'inconvenient::annoyingFunctionName3<Tab><Tab>
inconvenient::annoyingFunctionName3(int)
inconvenient::annoyingFunctionName3(void*)
(gdb) b 'inconvenient::annoyingFunctionName3(
```

И наконец, когда нужная функция выбрана, можно закрыть кавычки и нажать клавишу Enter. В итоге команда будет иметь следующий вид:

```
(gdb) b 'inconvenient::annoyingFunctionName3(void*)'
Breakpoint 2 at 0x804836f: file cppsym.cpp, line 13.
```

Благодаря использованию клавиши Tab для завершения ввода нам удалось значительно сократить объем работы на клавиатуре.

Использование точек наблюдения

Многие процессоры содержат специальные регистры, которые задействуются при отладке с использованием точек останова. Отладчик gdb позволяет получать доступ к регистрам при помощи команды watchpoint. *Точка наблюдения* дает возможность останавливать выполнение программы всякий раз, когда определенная область памяти будет подвергаться чтению или записи. Сравните это с точкой останова, которая принимает в качестве аргумента адрес инструкции и останавливает программу, когда она начинает выполняться в данной области. Точки наблюдения особенно полезны при поиске повреждений памяти, вызванных дефектным программным кодом.

Для установки точки наблюдения в целях остановки выполнения программы, когда она будет изменять значение переменной foo, необходима следующая команда:

```
(gdb) watch foo    Ведение наблюдения за изменениями значения переменной foo
```

Важно учитывать, что точки наблюдения срабатывают только при изменении значения в памяти. Если первоначальное значение переменной foo равно 123 и при этом программа, к примеру, запишет значение 123, данная точка наблюдения не сработает. Необходимо отметить, что команда watch автоматически задействует адрес foo в качестве точки наблюдения. Если foo окажется указателем на область, которая будет подвергаться мониторингу, то потребуется следующая команда:

```
(gdb) watch *foo     Ведение наблюдения за изменениями в области,
                     на которую указывает foo
```

Если вы не поставите в команде знак звездочки, то будет осуществляться наблюдение за изменениями значения указателя foo! Подобные точки наблюдения позволяют останавливать выполнение программы всякий раз, когда происходит модификация значения переменной, помеченной в выражении, независимо от того, где выполняется данная программа. Это означает, что точку останова можно установить

в модуле, скомпилированном без активированной отладки. В данном случае пользователь может заглянуть в стек и (в идеале) отыскать кадр, содержащий полезную отладочную информацию.

Точки наблюдения могут комбинироваться с логическими условиями с целью создания условных точек наблюдения. Обеспечить остановку выполнения программы можно каждый раз, когда она будет записывать значение foo как равное 123:

```
(gdb) watch foo if foo == 123
```

Пример использования точек наблюдения. Детализированный пример поможет наглядно продемонстрировать полезность точек наблюдения. В листинге 10.2 приведена дефектная программа, которая время от времени вызывает переполнение буфера динамической памяти. Подобный тип ошибки может оказаться весьма сложным для обнаружения даже при использовании отладчика.

Я создал функцию ovrrun, которая служит «тонкой» оберткой для memcpy. Поскольку в данной функции отсутствует какая-либо проверка пределов, существует вероятность возникновения переполнения целевого буфера. Я добавил memcpy с целью снижения производительности и имитации программы, интенсивно потребляющей ресурсы процессора. Распределение целевого буфера осуществляется из динамической памяти с использованием buflen в качестве размера. Я искусственно создал ситуацию, когда вероятность переполнения целевого буфера на один байт составляет 1 к 800 000. Подобный тип превышения пределов буфера зачастую лишен «побочных эффектов» благодаря заполнению, производимому функцией malloc. Переполнение буфера можно выявить и впоследствии, путем использования strlen, однако обычно это бывает уже слишком поздно. Если в данном случае произошло серьезное переполнение буфера, то итогом может стать крах выполнения программы.

Если не используются точки наблюдения, то первым решением может стать установка условной точки останова. Остановку выполнения программы можно обеспечить при каждом вызове функции ovrrun, когда значение msglen будет превышать значение buflen. В данном случае нам потребуется строка со следующим синтаксисом:

```
(gdb) b ovrrun if msglen > buflen
```

Все будет работать, как и ожидалось, однако весьма медленно. Причина заключается в том, что *каждый* вызов msglen приводит к остановке выполнения программы и передаче управления gdb. Отладчик gdb проверяет значение msglen и сравнивает его со значением buflen, каждый раз решая, будет выполнение программы продолжено или остановлено. Поскольку данная программа совершает вызов ovrrun 800 000 раз, «накладные расходы», связанные с этой условной точкой останова, значительно сказываются на уровне производительности. На моей системе, в которой установлен процессор Pentium 4 с тактовой частой 1,7 ГГц, выполнение программы overrun занимает лишь около 700 мс, если при этом используется gdb и нет никаких точек останова. Если же установить условную точку останова, то выполнение программы будет занимать уже 2 мин 17 с.

Аналогичная процедура, совершаемая с использованием точки наблюдения, вообще не влияет на производительность приложения; выполнение программы

происходит так же быстро, как и при использовании gdb без точек наблюдения. Устанавливается точка наблюдения следующим образом:

```
(gdb) watch buf[buflen]
```

В данном случае мы желаем наблюдать за тем, когда произойдет операция записи в область buf+buflen, что будет свидетельствовать о переполнении буфера. Высокая скорость выполнения программы объясняется тем, что управление соответствующим триггером осуществляется на уровне процессора, который не генерирует его до тех пор, пока не произойдет операция записи. Таким образом, вместо остановки выполнения программы 800 000 раз отладчик gdb делает это всего лишь один раз.

Точки наблюдения бывают трех видов:

- watch — останавливает выполнение программы, когда она осуществляет запись в соответствующую область, в результате чего изменяется значение;
- rwatch — останавливает выполнение программы, когда она осуществляет чтение соответствующей области;
- awatch — останавливает выполнение программы, когда она осуществляет чтение соответствующей области или запись в нее.

Отладчик gdb управляет точками наблюдения так же, как и точками останова. Список точек наблюдения можно просмотреть при помощи команды info watchpoints, которая является синонимом команды info breakpoints. Удаление точек наблюдения, как и точек останова, осуществляется посредством команды delete.

Листинг 10.2. overrun.c: пример дефектной программы для иллюстрации использования точек наблюдения

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <stdlib.h>
 4 #include <time.h>
 5
 6     // Исходный текст для копирования
 7 const char text[] = "0123456789abcdef";
 8
 9     // Данная функция будет вызывать переполнение буфера, если дать ей
       // соответствующее указание
10 void ovrrun(char *buf, const char *msg, int msglen)
11 {
12     // memcpy – используется лишь в целях снижения производительности
13     // и наглядной иллюстрации полезности точек наблюдения
14     char dummy[4096];
15     memset(dummy, msglen, sizeof(dummy));
16
17     // А вот и наш "нарушитель"...
18     memcpy(buf, msg, msglen);
19 }
20
21     // Тщательно выбранный при помощи malloc размер распределения.
22     // Выполнить распределение небольшого буфера при помощи malloc
       // из динамической памяти (не использовать mmap).
```

```
23     // При помощи malloc осуществляется также заполнение буфера, что
       // означает, что его переполнение на 1 байт
24     // не должно привести к краху программы
25 const int buflen = 13;
26
27 int main(int argc, char *argv[])
28 {
29     char *buf = malloc(buflen);
30     int i;
31
32     // Использовать генератор случайных чисел, чтобы при каждом прогоне
       // результат отличался от предыдущего.
33     srand(time(NULL));
34
35     // Количество циклов — большое число.
36     int n = 800000;
37
38     // Вероятность переполнения буфера должна составлять 1 шанс из N для
39     // того, чтобы его было сложно отловить.
40     int thresh = RAND_MAX / n;
41
42     for (i = 0; i < n; i++) {
43     // Переполнение буфера должно произойти, если случайное число
       // окажется меньше порогового значения
44        int len = (rand() < thresh) ? buflen + 1 : buflen;
45        ovrrun(buf, text, len);
46     }
47
48     // Переполнение буфера легко обнаружить, но сложно отловить.
49     int overran = (strlen(buf) > buflen);
50     if (overran)
51         printf("OVERRUN!\n");
52     else
53         printf("No overrun\n");
54
55     free(buf);
56     return overran;
57 }
```

10.3.3. Проверка данных и манипулирование ими

Отладчик gdb обладает весьма мощными функциональными возможностями для проверки данных с использованием всего лишь нескольких команд с богатым синтаксисом. Перед тем как приступить к их изучению, давайте рассмотрим базовые команды, которые применяются в данном случае:

- print — обеспечивает уникальный синтаксис с разнообразным форматированием, посредством которого можно отображать на экране любые типы данных, например включаемые строки и массивы. Отображаемые объекты могут располагаться в памяти или в любом пригодном выражении C или C++;

- x — сокращение от слова *examine (проверка)*; является аналогом команды print, за тем исключением, что работает с адресами памяти и «сырыми» данными, в то время как print позволяет обрабатывать абстрактные выражения. Обе эти команды поддерживают модификаторы, о которых мы поговорим в следующем разделе;
- printf — похожа на функцию C с аналогичным именем. Соблюдает идентичные правила при форматировании. В строку формата необходимо включать новую строку, за исключением ситуаций, когда она действительно не требуется;
- whatis — сообщает пользователю все то, что отладчику gdb «известно» о типе определенного символа;
- backtrace — позволяет просматривать стек вызовов текущей программы, включая при необходимости и локальные переменные;
- up, down — изменяет кадр стека, благодаря чему пользователь получает возможность ознакомиться с локальными переменными в различных сегментах стека вызовов;
- frame — альтернатива командам up и down, которая позволяет указывать именно тот кадр, к которому желает перейти пользователь. Кадры указываются посредством номеров, список которых выводится при помощи команды backtrace;
- info locals — подкоманда команды info, которая выводит список всех локальных переменных в текущем кадре стека.

Синтаксис выражений print

Отображение одиночной переменной на экране или сброс дампов памяти осуществляется при помощи команд print и x (print сокращенно именуется p). Команда print может принимать почти любое допустимое выражение C или C++ в качестве аргумента[1], в то время как команда x принимает адрес в качестве аргумента, после чего выводит на экран сведения о памяти, расположенной по данному адресу. Если в качестве аргумента для команды x используется переменная, то она будет восприниматься как адрес даже в том случае, если не является указателем.

Формат сведений, выводимых по умолчанию, можно варьировать, при этом отладчик gdb позволяет с легкостью изменять поведение данных команд по умолчанию. Команды print и x поддерживают использование модификаторов, которые дают возможность влиять на их вывод. Команда x также позволяет указывать количественный счетчик. При использовании обеих этих команд отладчик gdb требует, чтобы пользователь отделял модификаторы от них при помощи прямого слеша, например:

```
(gdb) p/x foo           Вывод значения foo в шестнадцатеричном представлении
$2 = 0x1000
(gdb) x/d &foo          Сброс дампа памяти по адресу &foo в десятичном
                        представлении
0x22eec4:       4096
```

[1] gdb также «понимает» выражения, написанные на языках, отличных от C и C++. Для получения дополнительных подробностей см. info gdb languages support.

Полный перечень модификаторов приведен в табл. 10.1.

Таблица 10.1. Модификаторы вывода, используемые в сочетании с командами print и x

Модификатор	Формат	print	x
x	Шестнадцатеричный	Да	Да
d	Знаковый десятичный	Да	Да
u	Беззнаковый десятичный	Да	Да
o	Восьмеричный	Да	Да
t	Двоичный	Да	Да
a	Адресный	Выводит значения в шестнадцатеричном представлении и показывает их связь с соседними символами	Выводит значения в шестнадцатеричном представлении и показывает их связь с соседними символами
c	Символьный	Наименее значимый байт	Осуществляет сброс дампов памяти парами — ASCII-символы сочетаются с десятичными байтами
f	С плавающей точкой	Осуществляет сброс дампов памяти в виде значений типа double	Осуществляет сброс дампов памяти в виде значений с плавающей точкой типа float, используя текущий размер слов. При работе на 32-битных машинах для IEEE double необходимо использовать g, а для IEEE float — w
i	Инструкции	Нет	Дизассемблирует память по указанному адресу
s	Завершаемая NUL-символом строка	Нет	Осуществляет сброс дампов памяти в виде ASCII-строк. Вывод останавливается на первом символе NUL

Кроме того, команда `x` позволяет также указывать размер слов, который будет использоваться при сбросе дампов памяти, а также количество слов, подлежащих сбросу. Счетчик количества указывается непосредственно после слеша, например:

```
(gdb) x/8bx &foo      Сброс дампа памяти в 8 байт в шестнадцатеричном
                      представлении по адресу &foo
0x22eec4:       0x00   0x10   0x00   0x00   0xd8   0xef   0x22   0x00
```

Поскольку команда `x` позволяет осуществлять сброс дампов памяти, для отображения на экране данных при этом используется фиксированный размер слов. Размер слов может указываться посредством одного из суффиксов, приведенных в табл. 10.2. В примере команда `x` использована в сочетании с флагом `b`, который определяет размер слов. Команда `print` определяет размер слов, исходя из типа переменной.

Таблица 10.2. Размеры слов, используемые с командой x

Суффикс	Размер слов
b	Байт (8 бит)
h	Половина слова (2 байта)
w	Слово (4 байта)
g	Большое слово (8 байт)

Отладчик gdb запоминает использованные модификаторы для применения в следующий раз, когда пользователь введет соответствующую команду, в силу чего указать нужные модификаторы достаточно лишь однажды. Если вам необходимо, чтобы данный подход сохранялся на протяжении оставшейся части сеанса, какие-либо дополнительные модификаторы указывать не следует. Модификаторы, указываемые после счетчика количества, могут иметь любой порядок, то есть 8bx — это то же самое, что 8xb.

Примеры с использованием команды print

Воспользовавшись табл. 10.1 и 10.2, приведу ряд практических примеров. Как отмечалось ранее в этой главе, команда print может принимать в качестве аргумента любые допустимые выражения C. Отладчик gdb также может вызывать функции, а это означает, что мы можем сделать кое-что интересное из командной строки gdb:

```
(gdb) p getpid()              Вывод pid-идентификатора текущего процесса
$1 = 12903
(gdb) p kill(getpid(),0)      Тест на существование процесса
$2 = 0
(gdb) p kill(getpid(),9)      Принудительное завершение процесса посредством
                              API-интерфейса C (отладчику gdb это
                              не понравится)
Program terminated with signal SIGKILL, Killed.
The program no longer exists.
The program being debugged stopped while in a function called from GDB.
When the function (kill) is done executing, GDB will silently
stop (instead of continuing to evaluate the expression containing
the function call).
```

print руководствуется типом отображаемой на экране переменной для придания выводу определенного формата, в то время как x осуществляет сброс дампа памяти с использованием явного размера слов, определяемого форматом. Это наглядно продемонстрировано на примере следующих переменных C:

```
double dblarr[] = {1,2,3,4};
float  fltarr[] = {1,2,3,4};
int    intarr[] = {1,2,3,4};
```

Теперь давайте рассмотрим разницу между x и print в gdb:

```
(gdb) p intarr
$5 = {10, 20, 30, 40}         Выводу придается формат в виде массива целых
                              чисел
```

```
 (gdb) x/4wx intarr          Вывод имеет вид 32-битного шестнадцатеричного
                             представления (в соответствии с запросом)
0x8049610 <intarr>:     0x0000000a      0x00000014      0x0000001e
0x00000028
(gdb) x/2gx intarr          Вывод имеет вид 64-битного шестнадцатеричного
                             представления (в соответствии с запросом)
0x8049610 <intarr>:     0x000000140000000a      0x000000280000001e
```

Использование команды `x` в сочетании с числами с плавающей точкой может привести к странным результатам, если не соблюдать некоторую осмотрительность. Так, например, числа с плавающей точкой IEEE типа `float` имеют размер слова 4 байта, однако если вы по невнимательности укажете 8-байтовый размер слова (`g`) в случае с числами такого типа, то увидите на экране абсолютно непонятный результат. В случае с числами IEEE типа `double` используется размер слова 8 байт, благодаря чему вывод с применением аналогичного формата выглядит абсолютно нормально, когда речь идет о массиве значений типа `double`:

```
(gdb) p fltarr              У команды p не возникает каких-либо проблем
                            с числами с плавающей точкой
$7 = {10, 20, 30, 40}
(gdb) x/4wf fltarr          Размер слов w является таким же, что
                            и у sizeof(float)
0x8049600 <fltarr>:     10      20      30      40
(gdb) x/2gf fltarr          Размер слов g слишком велик для чисел типа
                            float
0x8049600 <fltarr>:     134217760.5625  34359746808
(gdb) x/4gf dblarr          Размер слов g как раз подходит для чисел типа
                            double
0x80495e0 <dblarr>:     10      20
0x80495f0 <dblarr+16>:  30      40
```

В приведенных примерах формат указывался явным образом, что является необходимым действием, о котором нельзя забывать. Проблема заключается в том, что если забыть указать формат, то итоговый результат будет непонятным. В подобной ситуации стоит лишний раз перепроверить, чтобы убедиться в том, что вы использовали корректный формат, прежде чем делать какие-либо умозаключения.

Это лишь малая часть из многих вариантов, которые можно задействовать при выводе данных. Команда `print` обеспечивает даже еще бóльшую гибкость при работе с переменными благодаря тому, что позволяет использовать синтаксис на языке C. Так, например, в случае с массивами данный синтаксис можно применить для вывода на экран отдельных значений или же делать то же самое в отношении множественных элементов путем использования амперсандного суффикса:

```
(gdb) p *intarr             Как и в случае с синтаксисом на языке C, массив
                            может использоваться в качестве указателя
$4 = 10
(gdb) p intarr[1]           Использование индексной нотации C для просмотра
                            второго элемента в массиве
$5 = 20
(gdb) p intarr[1]@2         Использование комбинации индексов и знака @ для
                            просмотра двух элементов, начиная с элемента 1
$6 = {20, 30}
```

Однако при выводе строк необходимо учитывать ряд едва заметных различий. Некоторые форматы распознают NUL-символы ASCII, а некоторые игнорируют их. Рассмотрим следующие объявления:

```
const char ccarr[] = "This is NUL terminated.\000Oops! you shouldn't see this.";
const char *ccptr = ccarr;
```

ccarr является массивом, при этом NUL-символ располагается посреди ASCII-текста. ccarr — это указатель на одну и ту же область памяти. Необходимо отметить, что команда print проводит различие между переменными, исходя из их типов, в то время как команда x в сочетании с явным модификатором /s будет считать их принадлежащими к одному типу:

```
(gdb) p ccarr                 Тип массива не распознает NUL-символ ASCII
$1 = "This is NUL terminated.\000Oops! you shouldn't see this".
(gdb) p ccptr                Указатель на char распознает NUL-символ
$2 = 0x8048440 "This is NUL terminated".
(gdb) x/s ccarr              Модификатор /s явным образом дает указание
                             команде x вывести на экран строку, завершаемую
                             NUL-символом
0x8048440 <ccarr>:           "This is NUL terminated".
```

При использовании команды print можно также навязывать те или иные типы с использованием регулярного синтаксиса C в целях придания выводу требуемого вида, например:

```
(gdb) p (char*) ccarr
$3 = 0x403040 "This is NUL terminated."
```

И наконец, следует отметить, что, хотя вам вряд ли когда-нибудь потребуется это делать, вы можете произвести дизассемблирование машинного кода в любой области памяти путем использования формата i в сочетании с командой x:

```
(gdb) x/10i main
0x401050 <main>:       push   %ebp
0x401051 <main+1>:     mov    %esp,%ebp
0x401053 <main+3>:     sub    $0x28,%esp
0x401056 <main+6>:     and    $0xfffffff0,%esp
0x401059 <main+9>:     mov    $0x0,%eax
0x40105e <main+14>:    add    $0xf,%eax
0x401061 <main+17>:    add    $0xf,%eax
0x401064 <main+20>:    shr    $0x4,%eax
0x401067 <main+23>:    shl    $0x4,%eax
0x40106a <main+26>:    mov    %eax,0xffffffe4(%ebp)
```

Это может помочь, если вы решите отслеживать возможные атаки, связанные с переполнением буфера.

Вызов функций из gdb

Отладчик gdb позволяет вызывать любую функцию, которая доступна в вашей программе. Выполнение функции происходит в контексте запущенного процесса, и на это расходуется определенная часть стека и других ресурсов, отводимых

процессу, который подвергается отладке. На первый взгляд все это выглядит здорово, однако невнимательность может спровоцировать возникновение непредвиденных «побочных эффектов».

Вызов функции может быть включен в качестве аргумента почти в любую команду. Подобный подход я использовал ранее в этой главе при иллюстрации применения команды `print`, когда задействовал функцию `kill` в роли аргумента для `print`. Если вам нужно всего лишь совершить вызов функции, используйте команду `call`:

```
(gdb) call getpid()
$1 = 27274
```

`$1` — это временное значение, которое распределяется отладчиком `gdb` для размещения возвращаемого функцией значения. Оно располагается в пространстве `gdb` (а не выполняющейся программы). `gdb` автоматически осуществляет распределение подобных переменных всякий раз, когда ему необходимо сохранить какое-либо возвращаемое значение. Данные значения можно использовать в качестве аргументов для функций. Вы можете передать результат `getpid` из предыдущего примера команде `kill`, как показано далее:

```
(gdb) call kill($1,0)
$2 = 0
```

Если вам потребуется модифицировать значения в пространстве выполняющейся программы, используйте команду `set`. Данная команда может принимать множество различных аргументов, а также, как и большинство команд `gdb`, способна принимать в качестве аргумента любое допустимое выражение C. Благодаря свободному синтаксису `gdb` вы можете присваивать значения переменным с помощью любой команды, поддерживающей использование выражений C в роли аргумента, а не только посредством команды `set`. Вам не составит труда запомнить, что команду `set` следует использовать в сочетании с выражениями присваивания.

Примечания, касающиеся C++ и шаблонов

Шаблоны C++ представляют собой уникальное средство, используемое при отладке. Они позволяют программисту определять код в общей форме, в силу чего компилятор может генерировать исходный код из более абстрактных спецификаций. Рассмотрим простой пример, в котором два значения меняются местами:

```
template <class Typ>
void swapvals( Typ &a, Typ &b)
{
        Typ tmp = a;
        a = b;
        b = tmp;
}
```

Маркер `Tmp` представляет собой заполнитель для имени типа. Определение данного шаблона в исходном коде не приведет к генерированию какого-либо программного кода, пока вы не воспользуетесь им. Когда вы все же сделаете это, вам

потребуется указать тип, который займет место Typ. Это называется *реализацией*. Для создания функции, меняющей местами два значения типа double, необходимо ввести следующее:

```
swapvals<double>(a,b);
```

В результате компилятор создаст функцию swapvals, которая работает исключительно со значениями типа double. Если вам необходимо поменять местами две переменные типа int, введите swapvals<int>, и компилятор сгенерирует абсолютно другую функцию с уникальной подписью. Поскольку шаблон определяет все семейство функций, установление точки останова на одной из таких функции потребует определенного мастерства. Сначала необходимо отыскать нужную функцию, воспользовавшись gdb-командой info functions:

```
(gdb) info func swapvals
All functions matching regular expression "swapvals":
File templ.cpp:
void void swapvals<Foo>(Foo&, Foo&);  gdb 6.3 prints 'void' twice, for some
reason.
void void swapvals<double>(double&, double&);
void void swapvals<int>(int&, int&);
```

Обратите внимание на то, что для каждого типа приводится своя уникальная функция. Команды, посредством которой можно было бы установить точку останова сразу на всех функциях, генерируемых данным шаблоном, не существует. Установка точки останова возможна только на одной из этих функций за один раз. Для того чтобы установить точку останова на функции, касающейся типа int, необходимо открыть кавычки и воспользоваться методикой преобразования табуляций:

```
(gdb) b 'void swap<Tab>
(gdb) b 'void swapvals<<Tab>
(gdb) b 'void swapvals<int>(int&, int&)'
Breakpoint 3 at 0x8048434: file templ.cpp, line 8.
```

Приблизиться к возможности установки точки останова на всех функциях, соответствующих какому-либо шаблону, позволяет команда rbreak. С ее помощью можно установить точку останова на каждой из функций, которые соответствуют определенному регулярному выражению, например:

```
(gdb) rbreak swapvals
Breakpoint 2 at 0x8048456: file templ.cpp, line 8.
void void swapvals<Foo>(Foo&, Foo&);
Breakpoint 3 at 0x8048400: file templ.cpp, line 8.
void void swapvals<double>(double&, double&);
Breakpoint 4 at 0x8048434: file templ.cpp, line 8.
void void swapvals<int>(int&, int&);
```

Если ваш шаблон обладает коротким именем, которое соответствует многим другим функциям, в результате может оказаться, что будут установлены не те точки останова, которые были нужны. Поэтому будьте внимательны.

Примечания, касающиеся стандартной библиотеки шаблонов C++

Несмотря на то что C++ содержит стандартную библиотеку C ANSI, он имеет также собственную стандартную библиотеку, реализованную почти исключительно с использованием шаблонов. Формально в настоящее время *стандартная библиотека шаблонов STL (Standard Template Library)* является стандартной библиотекой C++, хотя многие программисты по-прежнему называют ее *библиотекой STL*.

Одной из особенностей стандартной библиотеки шаблонов STL являются контейнеры. *Контейнер* представляет собой шаблон, посредством которого реализуется динамическое хранилище. Контейнеры позволяют избежать лишней работы по написанию программного кода путем реализации общих алгоритмов хранения вроде списков, очередей и карт. Однако при отладке программного кода, использующего контейнеры, могут возникнуть проблемы.

Проблема заключается в том, что контейнеры C++ скрывают от пользователя основную реализацию. Доступ к данным в контейнере возможен только посредством вызова методов. Вызов методов тем же путем, которым вызываются функции, возможен при помощи `gdb`. Аналогичным образом можно проверять контейнеры во время выполнения.

Рассмотрим наиболее простой контейнер C++: `vector`. Поведение контейнера `vector` соответствует поведению обычного массива C за тем исключением, что данное хранилище является динамическим. Давайте сравним их на примере с использованием типа `int`:

```
int myarray[3];                    Массив C, куда входят три целых числа
std::vector<int> myvector(3);      Контейнер C++ vector, где располагаются
                                   три целых числа
```

При отладке программного кода, в котором присутствуют два подобных объявления, разобраться в том, с чем вы имеете дело, вам поможет команда `whatis`:

```
(gdb) whatis myarray
type = int [3]                                  Размер является фиксированным
 (gdb) whatis myvector
type = std::vector<int,std::allocator<int> > Размер является динамическим,
                                             об этом не сообщается
```

Все контейнеры C++ включают метод `size`, который сообщает пользователю о том, сколько элементов располагается в контейнере. Данный метод можно вызвать из `gdb` точно так же, как и любую другую функцию:

```
(gdb) p myvector.size()
$1 = 3
```

Однако при этом следует учитывать один важный момент: шаблоны C++ не генерируют код то тех пор, пока он не будет использоваться (реализовываться). Таким образом, если в вашем программном коде не задействован метод `size`, он не будет реализован и в исполняемом файле будет отсутствовать метод `size` для вызова. В такой ситуации на экране вы можете увидеть следующее сообщение:

```
(gdb) p myvector.size()
Cannot evaluate function -- may be inlined
```

Дополнительную путаницу вносит то, что реализация метода возможна косвенным образом, для чего необходимо воспользоваться другими методами. Подобную методику можно использовать для отладки одной программы, но она может оказаться непригодной в случае с другим приложением. Для того чтобы избежать трудностей, вы можете сознательно увеличивать количество излишних методов в своем программном коде с целью отладки. Также учитывайте то, что реализация должна осуществляться в отношении каждого уникального типа, то есть реализация `vector<int>::size` не повлечет реализации `vector<float>::size`.

Программа display

`display` — это инструмент, используемый для вывода определенного выражения при каждой остановке выполнения программы. Количество выводимых сообщений не ограничивается, при этом им можно придавать нужный формат с помощью того же синтаксиса, что и в случае с командой `x`.

10.3.4. Подключение к выполняющемуся процессу при помощи gdb

Находясь в командной строке, вы можете подключиться к выполняющемуся процессу при помощи `gdb` вводом следующей команды:

```
$ gdb programname pid
```

`pid` — это идентификатор процесса, к которому будет осуществляться подключение, а `programname` — файловое имя исполняемого файла. Оба этих значения должны оставаться соответствующими на протяжении всего сеанса отладки, для того чтобы был достигнут желаемый результат. Если вы попытаетесь поработать с исполняемым файлом, который был перекомпилирован после запуска процесса, нет никаких гарантий, что итоговый результат будет иметь какой-то смысл.

Если вам потребуется произвести отладку другой программы, то выходить из отладчика `gdb` или завершать выполнение текущего процесса не нужно. Отладку можно остановить и позволить процессу продолжать выполняться, применив команду `detach`. Она завершает сеанс отладки, не завершая при этом выполнения процесса. После отключения `gdb` вы можете сменить программу и заняться отладкой другого процесса при помощи команды `attach`. При необходимости также можно воспользоваться командой `file` и выбрать исполняемый файл, который будет соответствовать новому процессу.

В качестве решения, альтернативного запуску отладчика `gdb`, как это было сделано ранее в этой главе, можно воспользоваться такими командами:

```
(gdb)file programname
(gdb)attach pid
```

Следует отметить, что, если компиляция программы осуществлялась с включенной отладкой, можно пропустить этап ввода команды `file` и позволить отладчику `gdb` самому выяснить, где располагается нужный исполняемый файл.

10.3.5. Отладка файлов образов памяти

Когда процесс осуществляет сброс файла образа памяти на диск, он не включает в него информацию, касающуюся его виртуальной памяти. Генерирование файлов образов памяти всегда осуществляется в результате отправки определенного сигнала. Среди наиболее распространенных сигналов генерирования файлов образов памяти при проведении отладки встречаются следующие:

- `SIGSEGV` — *нарушение сегментации*. Данный сигнал посылается, когда процесс пытается осуществить чтение или запись по неверному адресу памяти, а также в ситуациях, когда процесс предпринимает попытку записи на страницу, предназначенную только для чтения, или старается выполнить чтение страницы, предназначенной только для записи;
- `SIGFPE` — *ошибка при операции с плавающей точкой*. Довольно странно, однако данный сигнал обычно *не* генерируется функциями с плавающей точкой. Отправка сигнала `SIGFPE` производится на архитектуре x86, когда процесс пытается произвести деление *целого числа* на ноль;
- `SIGABRT` — *отмена*. Данный сигнал используется функциями `abort` и `assert`;
- `SIGILL` — *недопустимая инструкция*. Отправка этого сигнала наиболее вероятна в ситуациях, когда написанные вручную ассемблерные программы пытаются использовать привилегированные инструкции из режима пользователя;
- `SIGBUS` — *ошибка шины*. Здесь имеется в виду не аппаратный уровень. Данный сигнал может стать результатом возникающей ошибки страницы, например, когда ощущается нехватка пространства, используемого для подкачки.

В файлы образов памяти включается информация о том, какой сигнал привел к их генерированию. Если вы располагаете экземпляром *того же* исполняемого файла, который сгенерировал файл образа памяти (а также всеми теми же совместно используемыми объектами), то можете произвести отладку программы, используя соответствующий файл образа памяти. Для выполнения отладки необходима следующая командная строка:

```
$ gdb exec-filename core-filename
```

Файлы образов памяти по традиции именуются просто `core`, хотя во многих дистрибутивах ядро сконфигурировано таким образом, что при сбросе подобных файлов на диск в их имя включается идентификатор процесса, например `core.pid`[1]. Когда такой файл используется при выполнении *«посмертной» отладки*, отладчик `gdb` в первую очередь сообщает пользователю о том, что послужило причиной сброса этого файла образа памяти на диск. По возможности `gdb` даже укажет строку в исходном файле, на которой произошла отправка сигнала, например:

```
$ gdb seldomcrash core.27078
...
Reading symbols from shared object read from target memory...done.
Loaded system supplied DSO at 0xffffe000
Core was generated by './seldomcrash'.
Program terminated with signal 8, Arithmetic exception.
```

[1] Данную опцию можно отключить при помощи команды sysctl -w kernel.core_uses_pid = 0.

```
...
#0  0x0804836c in main (argc=1, argv=0xbfc9acd4) at seldomcrash.c:10
10              return someint / 0;                  Ой, деление на ноль!
(gdb)
```

Как вы могли догадаться, запуск программ для проведения «посмертной» отладки из gdb возможен при помощи команды file, после которой необходимо ввести команду core-file.

Когда программа вызывает серьезное повреждение стека, использование генерируемого файла образа памяти в сочетании с отладчиком gdb часто оказывается ограниченным, поскольку последнему необходим стек для навигации по локальным переменным. Однако даже в случае повреждения стека глобальные и статические переменные не перестают быть полезными.

Информация о том, что ваша программа вызвала повреждение стека, весьма полезна, однако, к сожалению, отладчик gdb напрямую вам об этом не сообщит. Судить о том, что данное событие произошло, можно либо по тому факту, что перечень функций, выводимый командой backtrace, не имеет смысла по отношению к выполняющейся программе, либо когда отладчик gdb выдает сообщение no stack.

Наиболее распространенной причиной повреждения стека является переполнение буфера локальной переменной. Вы можете объявить массив в стеке и воспользоваться memset или memcpy для инициализации этого массива, для того чтобы значительно увеличить размер. Если говорить в общем, то всякий раз, когда вы передаете указатель на локальную переменную другой функции, вы создаете угрозу переполнения буфера. Возникновение подобной угрозы вероятно в случаях с функциями scanf, read и многими другими.

Программа, приведенная в листинге 10.3, является реализацией факториальной функции, о которой говорилось ранее в этой главе. Обычно она используется в статистических приложениях, а также в сфере компьютерного образования в качестве примера рекурсивного программирования[1].

Листинг 10.3. factorial.c: пример ситуации, когда происходит повреждение стека

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <stdlib.h>
 4
 5 // Факториальная функция. Пример традиционного программирования
 6 // рекурсивной функции
 7 int factorial(int x)
 8 {
 9     int overflow;
10     static int depth = 0;
11
12     if (x <= 1)
13         return 1;
14
```

[1] Бесконечная рекурсия представляет собой широко распространенную ошибку программирования, которая приводит к генерированию сигнала SIGSEGV, но не обязательно к повреждению стека.

```
15     if (++depth > 6) {
16         // Переполнение размером n байт. Приводит к генерированию сигнала
           // SIGSEGV, однако нам неизвестно, где именно
17         int n = 0x100;
18         memset((char *) &overflow, 0xa5, sizeof(overflow) + n);
19     }
20
21     return x * factorial(x - 1);
22 }
23
24 int main(int argc, char *argv[])
25 {
26     int n = 3;
27     printf("Аргумент командной строки > 7 приведет к повреждению стека\n");
28     if (argc > 1) {
29         n = atoi(argv[1]);
30     }
31     printf("%d! == %d\n", n, factorial(n));
32     return 0;
33 }
```

В данном примере повреждение стека было нарочно спровоцировано после того, как функция вызвала сама себя 7 раз. Для этого мы записали 256 байт по адресу, по которому располагается целочисленная локальная переменная. Вот как это все выглядит в действии:

```
$ ./factorial 7
Command line argument > 7 will cause stack corruption
7! == 5040
$ ./factorial 8
Command line argument > 7 will cause stack corruption
Segmentation fault (core dumped)                    Как и следовало ожидать!
$ gdb ./factorial core
GNU gdb Red Hat Linux (6.3.0.0-1.21rh)
...
Core was generated by `./factorial 8'.
Program terminated with signal 11, Segmentation fault.
...
#0 0xa5a5a5a5 in ?? ()                              Это выглядит не слишком
                                                    хорошо
 (gdb) bt                                           Давайте посмотрим, что
#0 0xa5a5a5a5 in ?? ()                              нам "скажет" backtrace
#1 0xa5a5a5a5 in ?? ()
#2 0xa5a5a5a5 in ?? ()
#3 0xa5a5a5a5 in ?? ()
...                                                 Вывод оказался не очень
#21 0xa5a5a5a5 in ?? ()                             полезным
#22 0xa5a5a5a5 in ?? ()
---Type <return> to continue, or q <return> to quit---q
Quit
 (gdb)p factorial::depth                            Статические переменные
$1 = 7                                              по-прежнему в порядке
```

После того как программа сбрасывает файл образа памяти на диск, мы задействуем gdb, пытаясь произвести «посмертную» отладку, и выясняем, что стек оказывается непригодным к использованию. На это указывает тот факт, что вывод команды backtrace (bt) не содержит ничего полезного. Однако мы по-прежнему можем использовать упомянутый файл образа памяти. Для просмотра локальных переменных нам потребуются лишь кадры стека. Все статические и глобальные переменные по-прежнему доступны для просмотра. Для того чтобы увидеть статическую переменную depth, определенную в функции factorial, необходимо воспользоваться следующим синтаксисом C++: factorial::depth. Изучив эту переменную, мы узнаем, что вызов factorial осуществлялся 7 раз. В данном случае это будет не намного хуже, чем просмотр трассы стека.

К сожалению, большинство программ не столь просты в отладке. Если вам доведется заниматься отладкой приложения, которое вызывает повреждение стека, вам следует рассмотреть возможность добавления глобальных или статических переменных в целях проведения «посмертной» отладки. Подробнее об этом мы поговорим позднее, в разделе «*Создание собственного "черного ящика"*» данной главы.

10.3.6. Отладка многопоточных программ при помощи gdb

Отладка многопоточных программ может оказаться непростым занятием. Отладчик gdb поддерживает работу с потоками и предусматривает ряда опций, позволяющих упростить данную процедуру.

Все многопоточные программы вначале являются однопоточными, то есть отладка не будет представлять сложности до тех пор, пока не будут созданы потоки. Поведение отладчика gdb по умолчанию в ситуации с многопоточным приложением, скорее всего, будет именно таким, как вы ожидаете. Когда вы находитесь в приглашении командной строки, выполнение всех потоков останавливается. Если вы начнете пошаговое выполнение своего программного кода, отладчик gdb будет пытаться остановиться на текущем потоке. Однако если точка останова окажется на другом потоке, приглашение командной строки переключится на кадр стека этого потока.

Команда info threads позволяет выводить список выполняющихся потоков. В сокращенном виде он выглядит следующим образом:

```
(gdb) info threads
  4 Thread -1225585744 (LWP 6703) 0x0804854c in the_thread (ptr=0x2)
    at thread-demo.c:17
  3 Thread -1217193040 (LWP 6702) 0x0804854c in the_thread (ptr=0x1)
    at thread-demo.c:17
  2 Thread -1208800336 (LWP 6701) 0x0804854c in the_thread (ptr=0x0)
    at thread-demo.c:17
* 1 Thread -1208797504 (LWP 6698) main (argc=1, argv=0xbf940e84)
    at thread-demo.c:45
```

Первая колонка содержит определяемые отладчиком gdb идентификаторы потоков. Это простые инкрементные значения, которые gdb использует для более легкого переключения между потоками для пользователя. Значение идентификатора увеличивается с каждым новым потоком, что помогает разграничивать

последние. Большое отрицательное значение в приведенном примере — это pthread_t, что используется API-интерфейсом pthreads.

В приглашении командной строки отладчик gdb использует кадр стека текущего потока для локальных переменных. *Текущий поток* представляет собой «место», на котором останавливается отладчик gdb. Для переключения с одного потока на другой используйте команду thread в сочетании с gdb-идентификатором соответствующего потока.

Любая точка, которую пользователь устанавливает без квалификатора, применяется ко всем потокам. Первый поток, который достигнет точки останова, вызовет остановку выполнения программы, а gdb сделает этот поток текущим. Для того чтобы применить точку останова только к одному потоку, необходимо воспользоваться квалификатором thread в сочетании с командой break. Если вы хотите, чтобы остановка выполнения программы произошла на функции foo в потоке 3, введите команду:

```
(gdb) break foo thread 3
```

Каждый раз, когда gdb сталкивается с точкой останова, поток, который послужил причиной остановки, становится текущим потоком. С этого момента пошаговое выполнение программы будет происходить в текущем потоке.

Использование gdb-опции scheduler-lock при работе с потоками

При пошаговом выполнении многопоточной программы с использованием команды step или next важно знать, что другие потоки также будут выполняться наряду с потоком, который выполняется поэтапно. Обычно такая ситуация приветствуется, поскольку она обеспечивает некоторую степень синхронности потоков, когда пользователь осуществляет пошаговое выполнение приложения.

Однако бывают случаи, когда подобное может оказаться нежелательным. При пошаговом выполнении программы с использованием отладчика gdb в одном потоке другие потоки также могут выполняться и достигать точек останова. Если это случается, отладчик gdb осуществляет автоматическое переключение потоков. Для предотвращения этого необходимо изменить поведение планировщика, когда будет происходить пошаговое выполнение приложения, воспользовавшись командой set scheduler-lock. Она может принимать один из трех параметров:

```
set scheduler-lock off
set scheduler-lock on
set scheduler-lock step
```

Параметром по умолчанию является off, о котором мы только что говорили. Параметр on приводит к тому, что gdb останавливает все остальные потоки при пошаговом выполнении приложения. Эти потоки не продолжат свое выполнение до тех пор, пока не будет введена команда continue.

Использование параметра step влечет остановку потоков при вводе не команды step, а команды next, то есть выполнение прочих потоков не будет осуществляться, если пользователь «не перешагнет» ***через*** функцию в текущем потоке при помощи функции next.

10.3.7. Отладка оптимизированного программного кода

gdb допускает, чтобы компиляция программного кода осуществлялась с активированной оптимизацией и отладкой, несмотря на то что эти опции могут показаться взаимоисключающими. Бывают ситуации, когда снижение уровня производительности из-за того, что не используется оптимизация, является недопустимым. Вместе с тем поведение оптимизированного программного кода, выполняющегося наряду с отладчиком, может оказаться откровенно странным. Иногда можно пойти на компромисс, скомпилировав без применения оптимизации только избранные модули, однако это не всегда возможно.

Оптимизация зависит от используемой архитектуры, поэтому в ее проведении легких путей не существует. Обычно следует обращать внимание на следующие аспекты:

- *неиспользуемые переменные* — оптимизатор самостоятельно удаляет неиспользуемые переменные из программного кода, что может вызвать путаницу, когда задействуется отладка. Иногда программисты оставляют без внимания переменные, в которых размещаются данные, необходимые исключительно при проведении отладки. Подобные переменные, предназначенные только для записи, являются первыми кандидатами на удаление оптимизатором. Это, естественно, усложняет отладку. Для того чтобы «защитить» такие переменные от оптимизатора, их необходимо объявить как volatile;
- *встраиваемые функции* — они представляют собой типичный способ оптимизации, к которому компиляторы прибегают в целях снижения «накладных расходов», связанных с вызовами функций. Если функция является встраиваемой, она не будет отражаться в стеке вызовов. Для того чтобы деактивировать встраивание функций, компилирование программного кода необходимо осуществлять с использованием gcc с флагом -fno-inline;
- *неполадки* — пошаговое выполнение оптимизированной функции может бросить вызов вашему самообладанию. Там, где выполнение неоптимизированного программного кода осуществляется нормально от начала до конца, оптимизированная функция зачастую внезапно начинает «прыгать» с места на место по той причине, что оптимизатор перестраивает программный код для того, чтобы его выполнение происходило более эффективно.

Иногда может оказаться полезным обратиться к ассемблеру. gdb не обеспечивает объединенных исходных и ассемблерных листингов, однако вы можете увидеть, что компилятор сообщает отладчику gdb, применив команду objdump. Рассмотрим пример дефектной функции:

```
1 void vscale2(double *vec, int len, double arg)
2 {
3     int i;
4
5     // Временные переменные, сбивающие с толку читателя...
6     double a, b, c;
7     for (i = 0; i < len; i++) {
```

```
 8
 9         // Неважно, какую именно роль это играет
10         a = arg * arg;
11         b = arg * a / 2;
12         c = a * b * arg;
13         vec[i] *= c;
14     }
15 }
```

Временные переменные, определенные в строке 6, используются лишь для того, чтобы сбить с толку читателя. Использованием временных переменных в разных строках мы даем оптимизатору возможность объединить эти строки в одну общую операцию. Компиляция данной функции с включенной оптимизацией и последующий ввод команды `objdump` покажет нам, что все эти ссылки будут сжаты в одну (с точки зрения отладчика) строку кода, то есть оптимизатор будет способен понять, что они являются временными переменными, и использует эту информацию при объединении строк программного кода.

В данном случае команда `objdump` является довольно полезной. Она позволяет генерировать дамп машинного кода, производить дизассемблирование и выводить отладочную информацию, аннотированную оригинальным исходным кодом C. Если применить данную команду к объектному файлу, скомпилированному с включенными оптимизацией и отладкой, можно увидеть более четкую картину.

10.4. Отладка совместно используемых объектов

Совместно используемые объекты по-разному называются на различных платформах, однако сама концепция остается неизменной. В операционной системе Windows от компании Microsoft они называются *динамическими связываемыми библиотеками*, а во многих UNIX-инструментах их именуют *динамическими библиотеками*. Общим является термин *совместно используемые объекты*, так как в данном случае речь не обязательно должна идти о библиотеках, несмотря на то что именно они используются наиболее часто. Отладка программного кода, содержащегося в совместно используемом объекте, не слишком отличается от отладки обычной программы. Однако иногда при работе с двоичным кодом могут возникнуть определенные трудности.

10.4.1. Когда и зачем необходимы совместно используемые объекты

Все Linux-приложения по умолчанию задействуют совместно используемые объекты в качестве стандартных библиотек, поскольку это наиболее эффективный способ расходования памяти. В этом можно убедиться при помощи команды `ldd`[1]. В качестве примера возьмем приложение `Hello World`:

[1] ldd — это сокращение от list dynamic dependencies — просмотр динамических зависимостей.

```
$ ldd hello
        linux-gate.so.1 => (0xffffe000)
        libc.so.6 => /lib/libc.so.6 (0xb7e32000)
        /lib/ld-linux.so.2 (0xb7f69000)
```

Здесь видно, что файл связан со стандартной библиотекой `libc.so`, при этом «рабочая» версия располагается в `/lib/libc.so.6`. Программный файл также должен быть связан с динамическим редактором связей, путь к которому выглядит как `/lib/ld-linux.so.2` в приведенном ранее примере. `linux-gate.so.1` — это совместно используемый псевдообъект, который имеет отношение к архитектуре Intel. Он позволяет совместно используемым библиотекам задействовать более быстрые операции `sysenter` и `sysexit`, если процессор их поддерживает. Они представляют собой более скоростное решение по сравнению с обычным механизмом совершения системных вызовов, который использует программные прерывания.

Обычно совместно используемые объекты применяются как средство разделения программной библиотеки между несколькими процессами. Это позволяет экономить физическую память, поскольку сегменты совместно используемой библиотеки, предназначенные только для чтения, занимают одну и ту же область физической памяти, когда она задействуется сразу несколькими процессами.

Менее распространено применение совместно используемых объектов для реализации оверлеев. В былые времена оверлеи были общепринятой методикой, позволявшей экономить память на 16-битных платформах, не имеющих виртуальной памяти. В настоящее время данная методика вряд ли актуальна, однако для справки необходимо отметить, что POSIX предусматривает наличие соответствующего API-интерфейса для этих целей. Те, кому интересен данный вопрос, могут обратиться к странице руководства man `dlopen(3)`.

10.4.2. Создание совместно используемых объектов

В принципе, единственная разница между совместно используемым объектом и программой заключается в том, что разделяемый объект обычно лишен функции `main`. Однако это не является обязательным требованием. Вы можете создавать совместно используемые объекты, которые можно вызывать подобно исполняемым файлам, сохраняя при этом возможность их динамического связывания с более крупной программой. Динамический редактор связей сам по себе является совместно используемым объектом; он задействуется командой `ldd`, о которой говорилось ранее в этой главе.

Создать заурядный совместно используемый объект довольно просто — процедура аналогична сборке программы, однако в данном случае следует использовать флаги `-shared` и `-fpic`, например:

```
$ cc -shared -fpic -o libmylib.so mylib1.c mylib2.c
```

Флаг `-shared` адресован редактору связей и указывает ему на необходимость создания совместно используемого объекта, а не исполняемого файла. Флаг `-fpic` информирует компилятор о том, что он должен сгенерировать *позиционно-независимый*

программный код. Это весьма важный момент, поскольку, в отличие от стандартных исполняемых файлов, виртуальные адреса совместно используемых объектов не будут известны до начала выполнения.

Связывание программы с совместно используемым объектом осуществляется обманчиво просто:

```
$ cc -o myprog myprog.o -L . -lmylib
```

С помощью параметра `-L` мы сообщаем редактору связей о том, что совместно используемый объект располагается в текущем каталоге. Проблема заключается в том, что редактору связей `ld-linux.so` также необходимо знать, где искать данный совместно используемый объект. В этом вы можете убедиться сами, если попытаетесь выполнить такую программу:

```
$ ./myprog
./myprog: error while loading shared libraries: libmylib.so: cannot open shared
object file: No such file or directory
```

Проблема заключается в том, что система не имеет понятия о том, где расположен совместно используемый объект. Программы, связываемые с такими объектами, не содержат информации об их местонахождении. Это делается намеренно, так как совместно используемые объекты в разных системах расположены совершенно по-особому. Если приложение будет запускаться в другой системе, то данная система не должна сама делать какие-либо предположения о местонахождении соответствующего совместно используемого объекта. Вместо этого задействуется так называемое *специальное имя* `soname` — им обладает каждый такой объект и оно представляет собой имя, по которому динамический редактор связей идентифицирует объект. Библиотека, которую мы создали, не имеет специального имени `soname`, поскольку мы не указали его. Это делать не обязательно, так как редактор связей обратится к файловому имени, если не сможет распознать специальное имя `soname`.

10.4.3. Определение местоположения совместно используемых объектов

Определением местоположения совместно используемых объектов занимается динамический редактор связей `ldlinux.so`, расположенный в каталоге `/lib`. Динамический редактор связей всегда использует для поиска стандартные пути `/lib` и `/usr/lib`. Если вы желаете разместить совместно используемые объекты в других местах, то воспользуйтесь переменной окружения `LD_LIBRARY_PATH`. Зачастую совместно используемые библиотеки располагаются в системе в нескольких местах. Для того чтобы динамическому редактору связей не приходилось обследовать большое число путей, система сохраняет кэш специальных имен `soname`, а также сведения о местонахождении совместно используемых объектов в `/etc/ld.so.cache`. Создание и обновление данного кэша осуществляется посредством программы `/sbin/ldconfig`, которая исследует каталоги, указанные в `/etc/ld.so.conf`.

При каждой установке новых библиотек необходимо запускать программу `ldconfig`, для того чтобы кэш обновлялся. Кроме того, данная программа создает символические ссылки, благодаря чему файловые имена совместно используемых

объектов уникально отличаются от специальных имен soname. При использовании программы hello на моем компьютере, где установлен дистрибутив Fedora Core 3, libc.so.6 указывает на файл с именем libc-2.3.6.so:

```
$ ls -l /lib/libc.so.6
lrwxrwxrwx  1 root root 13 Jul  2 16:03 /lib/libc.so.6 -> libc-2.3.6.so
```

libc.so.6 — это общее специальное имя soname, с которым сталкивается компилятор, в то время как libc-2.3.6.so представляет собой файловое имя, используемое стандартной библиотекой GNU C (glibc). В принципе, вам не обязательно использовать библиотеку glibc для того, чтобы иметь доступ к libc.so.6. В качестве замены вы можете воспользоваться собственной библиотекой. Если она обладает корректным специальным именем soname и располагается в соответствующем каталоге, динамический редактор связей сможет ее задействовать. Однако в действительности замена библиотеки glibc может нарушить работу всех GNU-инструментов, которым требуются расширения glibc.

10.4.4. Переназначение местоположения совместно используемых объектов по умолчанию

Непривилегированные пользователи могут использовать переменную окружения LD_LIBRARY_PATH для того, чтобы указать динамическому редактору связей путь для поиска. Рассмотрим ситуацию на примере с myprog:

```
$ LD_LIBRARY_PATH=./ ./myprog
```

Здесь динамический редактор связей получает указание осуществлять поиск совместно используемых объектов в текущем каталоге, в котором располагается libmylib.so. Что касается объектов, обладающих специальным именем soname, то в этом случае можно избежать лишней работы на клавиатуре, воспользовавшись переменной окружения LD_PRELOAD следующим образом:

```
$ LD_PRELOAD=libc.so.6 ./hello-world      libc.so.6 – это специальное имя
                                          soname библиотеки glibc в моей
                                          системе
```

Здесь мы даем указание динамическому редактору связей произвести связывание с libc перед тем, как осуществлять связывание остальной части hello-world. Подобный подход может оказаться целесообразным, если ваша программа связывается с библиотекой, которая повторно реализует важную функцию из libc. Переменная окружения LD_PRELOAD работает только с библиотеками, которые имеют специальные имена soname, указанные в /etc/ld.so.cache.

10.4.5. Безопасность и совместно используемые объекты

Мы рассмотрели преимущества совместно используемых объектов, однако, неправильно применяя их, можно поставить под удар безопасность своей системы. Некоторые разделяемые объекты совместно используются многими программами,

например, libc используется фактически всеми системными командами. Множество программ задействуют подобные объекты, включая такие, которые выполняются с привилегиями корневого пользователя *root*. Если программисту-злоумышленнику удастся скомпрометировать широко применяющийся разделяемый объект, он может поставить под удар безопасность всей вашей системы.

Допустим, вы создали программу с привилегиями setuid root, позволив обычным пользователям выполнять в своей системе задания, связанные с ее сопровождением. Всякий раз, когда обычный пользователь будет запускать данную программу, процесс будет выполняться с привилегиями корневого пользователя *root*. Возможно, эта программа будет использовать разделяемый объект, расположенный в небезопасном месте. Программист-злоумышленник теоретически может подменить данный объект таким, который содержит вредоносный программный код. Исходная программа останется нетронутой, однако она будет представлять угрозу для всей системы, вызывая функции, содержащиеся в совместно используемом объекте, который был подменен злоумышленником.

По этой причине динамический редактор связей внимательно следит за тем, чтобы разделяемые объекты, используемые подобными программами, были безопасны. Так, например, разрешается использовать только те объекты, которые располагаются в стандартных каталогах (LD_LIBRARY_PATH игнорируется), а владельцем всех совместно используемых объектов должен выступать корневой пользователь *root*, при этом объекты должны обладать разрешениями только на чтение[1].

10.4.6. Инструментарий для работы с совместно используемыми объектами

Динамический редактор связей Linux сам по себе является инструментом командной строки. В силу исторических причин его страница руководства man располагается в ld.so(8), несмотря на то что реальное имя данной программы в текущих дистрибутивах звучит как ld-linux.so.2. При вызове из командной строки редактор связей способен принимать ряд параметров и поддерживает работу с многими переменными окружения, которые вы можете использовать, чтобы лучше понять свою программу. Эти параметры рассматриваются на странице руководства man ld.so(8), однако в большинстве случаев предпочтительным является инструмент ldd, который представляет собой сценарий-обертку для вызова ld-linux.so.2 и может принимать более дружественные к пользователю параметры.

Просмотр списка совместно используемых объектов, необходимых определенному исполняемому файлу

Если команда ldd вводится без каких-либо параметров, то итоговый вывод будет включать перечень всех совместно используемых объектов, необходимых конкретному исполняемому файлу:

```
$ ldd hello
        linux-gate.so.1 => (0xffffe000)
```

[1] Получить дополнительные подробности можно в ld.so(8).

```
        libm.so.6 => /lib/libm.so.6 (0xb7f1b000)
        libpthread.so.0 => /lib/libpthread.so.0 (0xb7f09000)
        libc.so.6 => /lib/libc.so.6 (0xb7de0000)
        /lib/ld-linux.so.2 (0xb7f4e000)
```

Строго говоря, это будет список объектов, с которыми данный исполняемый файл был связан. Вовсе не обязательно, что это будут объекты, необходимые этому файлу. В данном случае я намеренно произвел связывание программы `Hello World` с математической библиотекой, а также с библиотекой `pthread`, однако ни одна из них не является необходимой:

```
$ gcc -o hello hello.c -lm -lpthread
```

В отличие от статических библиотек, редактор связей не удаляет код совместно используемых объектов из исполняемого файла. Как уже отмечалось, статическая библиотека представляет собой архив. Редактор связей использует такой архив для того, чтобы задействовать лишь те объектные файлы, которые он считает необходимыми. Таким образом, редактор связей способен удалять ненужные объектные файлы из исполняемого файла. Если указать в командной строке совместно используемый объект, редактор связей включит его в исполняемый файл независимо от того, является он необходимым или нет.

В этом можно убедиться при помощи команды `ldd`, использованной ранее в этой главе. В данном случае нам известно, что `libpthread.so` и `libm.so` не являются необходимыми, а если бы мы не знали об этом? Применив параметр `-u`, можно просмотреть неиспользуемые зависимости, как показано далее[1]:

```
$ ldd -u ./hello
Unused direct dependencies:
        /lib/libm.so.6
        /lib/libpthread.so.0
```

Почему следует обращать особое внимание на неиспользуемые разделяемые объекты?

В случае с каждым совместно используемым объектом, с которым связывается программа, динамический редактор связей должен осуществлять поиск неразрешенных ссылок и совершать вызов инициализационных приложений. Это приводит к увеличению количества времени, которое уходит на запуск программы. На мощном настольном компьютере неиспользуемые разделяемые объекты вряд ли серьезно увеличат данный интервал, однако, если их будет слишком много, это может значительно увеличить время, необходимое для запуска программы.

Вторая проблема с неиспользуемыми разделяемыми объектами заключается в ресурсах, которые они потребляют. Подобный объект, независимо от того, используется он или нет, может вызвать распределение и инициализацию большого объема физической памяти. Если такая память будет инициализирована, но не использована, это вызовет обращение к разделу подкачки. Объекты, которые не

[1] Любопытно, что параметр –u не упоминается на странице руководства man, посвященной инструменту ldd, в отличие от справочных сведений, выводимых посредством –help.

потребляют физической памяти, могут отнимать виртуальную память. Это память, которая подвергается распределению, но не инициализируется. Если она не используется, то не будет вызывать обращений к разделу подкачки и отнимать физическую оперативную память, однако ограничит число доступных виртуальных адресов, которые может использовать программа. Обычно такая проблема возникает лишь на 32-битных архитектурах, которые требуют очень больших наборов данных — около нескольких гигабайт оперативной памяти.

Поиск символов в совместно используемых объектах

Возможны ситуации, когда вы загрузили определенный исходный код, который удачно компилируется, но не проходит связывание из-за отсутствующего символа. Команды `nm` и `objdump` являются отличными инструментами для просмотра таблицы символов программы. Кроме того, существует команда `readelf`. Все эти инструменты в основном позволяют решать одинаковые задачи, однако может случиться, что в зависимости от того, что желает узнать пользователь, лишь какой-то один из них окажется пригодным.

Допустим, у вас есть совместно используемый объект и вы желаете узнать его специальное имя `soname` перед установкой. Как уже отмечалось, команда `ldconfig` позволяет считывать имена и помещать их в кэш. Перед установкой своей библиотеки вам необходимо узнать, не будет ли ее специальное имя `soname` конфликтовать с аналогичными именами уже существующих объектов. Для этого нужно заглянуть в кэш с помощью ввода команды `ldconfig p`. Для того чтобы отыскать специальное имя `soname` какого-либо одиночного (неустановленного) совместно используемого объекта, необходимо обратиться к так называемой секции `DYNAMIC`. Инструмент `nm` для этого не годится, в отличие от `objdump` и `readelf`:

```
$ objdump -x some-obj-1.0.so | grep SONAME
  SONAME      libmylib.so
$ readelf  -a libmylib.so |grep SONAME
 0x0000000e (SONAME)                     Library soname: [libmylib.so]
```

Еще одна проблема заключается в том, что редактор связей «ругается» на неразрешенные символы. Наиболее частой причиной этого является отсутствие библиотеки или же попытка осуществить связывание с библиотекой неверной версии. Существует множество причин возникновения подобной ситуации. В случае с C++ проблема может быть вызвана подписью функции, которая подверглась изменению, либо простой опечаткой.

Все три упомянутых инструмента позволяют просматривать таблицы символов, однако `nm` является наиболее простым в использовании. Для просмотра объектного кода с целью поиска ссылок на определенный символ необходимо ввести команду:

```
$ nm -uA *.o | grep foo
```

Параметр `-u` ограничивает круг выводимых сведений неразрешенными символами в каждом объектном файле. Параметр `-A` позволяет выводить информацию об имени файла по каждому символу, то есть если передать итоговый вывод команде `grep`, то можно увидеть, в каком из объектных файлов содержится тот или

иной символ. Что касается C++, то здесь также доступен параметр -C, который позволяет придавать закодированным («искаженным») символам их изначальную форму. Это может оказаться полезным при отладке библиотек с неразумно выбранными подписями функций, как показано далее:

```
int foo(char p);
int foo(unsigned char p);
```

C++ позволяет обеим функциям иметь уникальные подписи, однако при этом осуществляет преобразование типов входных параметров для того, чтобы была возможность воспользоваться одним при отсутствии другого[1]. Для поиска библиотек, которые содержат эти функции, необходимо применить параметр -u. Добавление параметра -C никогда не повредит, за исключением ситуаций, когда вам необходимо увидеть закодированные («искаженные») имена функций:

```
$ nm -gCA lib*.a | grep foo
libFoolib.a:somefile.o:00000000 T foo(char)
libFoolib.a:somefile.o:00000016 T foo(unsigned char)
```

Как вы уже догадались, команды objdump и readelf позволяют выполнять аналогичные действия. Эквивалентом команды nm будет использование инструмента objdump следующим образом:

```
$ objdump -t
```

objdump также поддерживает применение параметра -C для придания закодированным («искаженным») символьным именам их исходной формы. Здесь эквивалентом будет команда readelf с таким параметром:

```
$ readelf -s
```

В отличие от nm и objdump, с выходом версии 2.15.94 инструмент readelf не стал поддерживать параметры для придания изначальной формы закодированным («искаженным») символьным именам. Все три упомянутые утилиты входят в состав пакета binutils.

10.5. Поиск проблем с памятью

Проблемы с памятью могут принимать множество форм, начиная с переполнения буфера и заканчивая утечками памяти. Для их устранения доступно много инструментов, однако существуют определенные ограничения в том, что вы можете с их помощью сделать. Несмотря на это, некоторые инструменты довольно легки в применении, в силу чего ими стоит воспользоваться. Иногда бывает так, что если один инструмент не годится, то поможет другой. Даже в библиотеке glibc содержатся инструменты, которые могут оказаться полезными при устранении проблем с динамической памятью.

---

[1] Кстати, если изменить типы входных аргументов на ссылки const, то входные типы будут строго навязаны.

10.5.1. Двойное освобождение

При освобождении указателя дважды довольно легко ошибиться, однако последствия этого будут весьма серьезными. Проблема заключается в том, что до недавнего времени библиотека `glibc` не проверяла указатели и слепо принимала любые указатели, которые ей передавал пользователь. Освобождение указателя на неверный виртуальный адрес приведет к генерированию сигнала `SIGSEGV` в той точке, где это произошло. Однако выявить это не составит особого труда. Куда сложнее разобраться в ситуации, когда освобождается указатель на корректный виртуальный адрес.

Зачастую бывает так, что неверный указатель, который в данный момент подвергается освобождению, был инициализирован посредством вызова `malloc`, а освобожден при помощи вызова `free`. Вполне возможно, что проведение освобождения указателя дважды приведет к искажениям в списке освобождения, который `glibc` использует для отслеживания распределений динамической памяти. Когда такое случается, генерируется сигнал `SIGSEGV`, однако подобная ситуация может не возникнуть до *следующего* вызова `malloc` или `free`! Именно такие ошибки намного труднее выявить.

Последние версии библиотеки `glibc` поддерживают проведение проверки для выявления неверных освобожденных указателей, которые провоцируют завершение программы со сбросом файла образа памяти на диск вне зависимости от обстоятельств. Подробнее на эту тему мы поговорим в дальнейшем.

10.5.2. Утечки памяти

Утечки памяти случаются, когда процесс после распределения блока памяти отбрасывает его, но не освобождает. Зачастую небольшие утечки безвредны и выполнение программы продолжается без проявления каких-либо отрицательных эффектов. Тем не менее по прошествии определенного времени даже незначительная утечка может превратиться в серьезную и стать проблемой. Простая программа, выполнение которой занимает небольшой промежуток времени, сможет «пережить» небольшие утечки, поскольку она отбрасывает выделенный ей блок динамической памяти по завершении работы. Однако процесс-демон, который может выполняться в системе месяцами, нетерпимо относится к любым утечкам, поскольку они имеют тенденцию накапливаться со временем.

Эффект утечки памяти проявляется в том, что объем памяти, занимаемой процессом, постоянно растет. Если обращения к распределенной памяти отсутствуют, то в случае с памятью, которая страдает от утечки, могут использоваться только виртуальные адреса. Если ваше приложение не испытывает нехватки виртуальных адресов в пространстве пользователя (обычно это 3 Гбайт на 32-битной машине), вы никогда не увидите проявления каких-либо отрицательных эффектов. В большинстве случаев страницы памяти, которая страдает от утечки, подвергаются модификациям со стороны программы, поэтому они будут занимать определенный объем физической памяти (оперативной или на устройстве подкачки). По мере того как неиспользуемые страницы памяти «стареют» и растут потребности в системной памяти, данные страницы записываются на диск подкачки.

Подкачка, вероятно, является наиболее коварным «побочным эффектом» утечек памяти, поскольку даже одна вызывающая утечку программа может замедлить работу всей системы. К счастью, утечки памяти легко выявить. Позднее мы рассмотрим инструменты, которые для этого используются.

10.5.3. Переполнение буфера

Переполнение буфера происходит, когда при выполнении операций записи программа выходит за пределы определенного блока памяти, перезаписывая страницы, которые могли бы быть использованы для других целей. Переполнение может стать результатом осуществления записи в память, отображение в которую отменено, или в память, предназначенную только для чтения, что приводит к генерированию сигнала `SIGSEGV`. Переполнение представляет собой широко распространенную разновидность ошибки и может случиться с любым типом памяти: стеком, динамической или статической памятью. Для выявления переполнения в динамической памяти существует ряд инструментов, однако обнаружить его в случае со статической памятью и локальными переменными будет намного сложнее.

Наилучший совет, который можно дать относительно переполнения: избегайте ситуаций, в которых оно может возникнуть. Определенные функции, содержащиеся в стандартной библиотеке, предоставляют широкие возможности для возникновения переполнения, по этой причине ими не следует пользоваться. Зачастую бывают доступны более безопасные альтернативные решения. Неплохим инструментом, позволяющим выявлять уязвимые места, является `flawfinder`[1]. Это сценарий на языке Python, который выполняет разбор исходного программного кода, ищет в нем небезопасные функции и сообщает о них пользователю.

Переполнение буфера стека

Переполнение буфера стека угрожает безопасности системы. Некоторые злоумышленники используют уязвимости в коммерческих программных продуктах, связанные с переполнением, для того чтобы внедрять вредоносный код в системы, которые в других ситуациях могли оказаться абсолютно надежными.

Типичным уязвимым местом является текстового поле для ввода данных, определяемое в качестве переменной с помощью функции, которая не проверяется на предмет переполнения. Если вводимые данные представляют собой обычный текст или всякий «мусор», программа просто потерпит крах. Это скверно, однако умный злоумышленник может ввести в текстовое поле двоичный машинный код, для того чтобы вызвать переполнение буфера ввода. Методом проб и ошибок такой злоумышленник способен выяснить, что именно нужно ввести, чтобы убедить программу выполнить нужный ему код и завладеть процессом.

Подробное рассмотрение всех деталей того, как это происходит, лежит вне рамок настоящей книги, однако следует учитывать, что переполнение буфера стека может нарушить безопасность системы. В обычных условиях переполнение буфера стека довольно трудно выявить. Характерным признаком того, что это случилось, является генерирование сигнала `SIGSEGV`, после чего следует сброс файла образа памяти

[1] См. сайт www.dwheeler.com/flawfinder.

на диск, в котором отсутствует трасса стека, из которой можно почерпнуть полезные сведения. В такой ситуации предпочтительнее выполнить просмотр программного кода с целью поиска известных проблемных функций, чем пытаться произвести его отладку.

Переполнение буфера динамической памяти

Если программа вызывает переполнение буфера динамической памяти, последствия не всегда проявляются сразу. Когда `malloc` использует вызов `mmap` для распределения блока памяти (как это бывает в случае с большими блоками), происходит заполнение запрошенного размера блока и он становится кратным объему страниц. Следовательно, если запрошенный размер блока не будет интегральным числом страниц, пространство, отведенное блоку, будет включать дополнительные байты. Ваш программный код может стать причиной выхода за пределы блока, а вы об этом так и не узнаете. И лишь когда данный код спровоцирует переполнение с достижением адреса, расположенного вне границ последней страницы, его выполнение будет завершено посредством отправки сигнала `SIGSEGV`. Хорошей новостью является то, что завершение произойдет немедленно и при этом не будет повреждена динамическая память. Что касается небольших блоков, когда вызовы `mmap` не используются, то проблема может оказаться еще более сложной.

При работе с блоками малого размера незначительное переполнение также может пройти незамеченным. Большинство реализаций динамической памяти заполняют размер блока таким образом, чтобы он вписывался в эффективные рамки в памяти. Благодаря этому случайное переполнение буфера на несколько байт не приведет к появлению каких-либо отрицательных эффектов. Подобные ошибки провоцируют крах программ лишь время от времени. Все зависит от реализации стандартной библиотеки, размера блока и величины переполнения.

Когда программа выходит за пределы небольшого блока, превышая заполненный размер, это приводит к искажению внутренних списков, которые `malloc` и `free` используют для поддержки динамической памяти. Обычно подобное переполнение не выявляется до тех пор, пока не будет совершен следующий вызов `malloc` или `free`. Дополнительную путаницу вносит то, что вызов `free`, который терпит неудачу, не обязательно должен освобождать блок, за пределы которого произошел выход. Если переполнение окажется серьезным, оно может охватить неверные виртуальные адреса, что приведет к генерированию сигнала `SIGSEGV`.

Проблемы с переполнением динамической памяти будут аналогичными и для C++. По сути, операторы по умолчанию для `new` и `delete` — это традиционная динамическая память, которая даже может использовать вызовы `malloc` и `free` из библиотеки C. GNU-реализация `new` и `delete` нетерпимо относится к переполнению даже на один байт, несмотря на то что, как и в ситуации с C, оно будет оставаться незамеченным до следующей операции `delete`. Однако C++ позволяет осуществлять перегрузку этих операторов. Если вы будете это делать, то создадите условия для возникновения переполнения. По этой причине решение о перегрузке операторов `new` и `delete` следует принимать, лишь хорошо все обдумав.

Для выявления переполнения буфера динамической памяти существует ряд инструментов; о них мы поговорим в дальнейшем.

10.5.4. Инструментарий, доступный в библиотеке glibc

Стандартная библиотека GNU (glibc) включает встроенные средства для отладки динамической памяти уже довольно давно. До недавнего времени эти функциональные возможности были по умолчанию деактивированы, а их активация осуществлялась только посредством переменной окружения MALLOC_CHECK_. Основная причина отсутствия проверки динамической памяти при каждой процедуре распределения или освобождения заключалась в том, что это приводило к снижению производительности. Некоторые средства проверки не создают большой нагрузки на систему, поэтому в последних версиях glibc основные из них активированы по умолчанию.

Использование MALLOC_CHECK_

glibc проверяет переменную окружения MALLOC_CHECK_ и изменяет ее поведение следующим образом:

- MALLOC_CHECK_=0 — полное отключение проверки;
  MALLOC_CHECK_=1 — вывод сообщения на stderr при обнаружении ошибки;
- MALLOC_CHECK_=2 — отмена при обнаружении ошибки; сообщение не выводится.

Если значение переменной окружения MALLOC_CHECK_ не присвоено, более старые версии glibc считают, что ее значение равно 0. Более свежие версии данной библиотеки полагают, что это значение равно 2. Сброс файла образа памяти на диск выполняется при обнаружении какой-либо рассогласованности наряду с выводом подробной трассы стека. Генерируемый по умолчанию вывод более информативен, чем тот, который можно получить, когда переменной окружения MALLOC_CHECK_ присваивается значение 2.

Поиск утечек памяти при помощи mtrace

mtrace — это инструмент, входящий в состав библиотеки glibc, которая в дистрибутиве Fedora располагается в пакете glibc-utils. По умолчанию в системе он не устанавливается. В других дистрибутивах он также может располагаться в аналогичном пакете. Основное назначение mtrace — поиск утечек. Для этих целей существуют и более подходящие инструменты, но, поскольку mtrace входит в состав библиотеки glibc, на нем стоит заострить внимание.

Для того чтобы использовать утилиту mtrace, необходимо инструментировать свой программный код посредством функций mtrace и muntrace из библиотеки glibc. Кроме того, переменной окружения MALLOC_TRACE должно быть присвоено значение в соответствии с именем файла, в котором glibc будет размещать данные, необходимые утилите mtrace. После запуска программного кода данные будут сохраняться в указанном вами файле и перезаписываться при каждом прогоне программы. Я не стану приводить здесь листинг, а лишь продемонстрирую, как функционирует mtrace:

```
$ MALLOC_TRACE=foo.dat ./ex-mtrace        Данные mtrace будут сохраняться
                                          в foo.dat
```

```
leaking 0x603 bytes
leaking 0x6e2 bytes
leaking 0x1d8 bytes
(утечка 0x1d8 байт)
(утечка 0xd9f байт)
leaking 0xc3 bytes
leaking 0x22f bytes
$ mtrace ./ex-mtrace foo.dat            Утилите mtrace необходимы имя
                                        исполняемого файла и данные
Memory not freed:
-----------------
   Address     Size     Caller
0x0804a378    0x603  at /home/john/examples/ch-10/memory/ex-mtrace.c:23
0x0804a980    0x6e2  at /home/john/examples/ch-10/memory/ex-mtrace.c:23
0x0804b068    0x1d8  at /home/john/examples/ch-10/memory/ex-mtrace.c:23
0x0804b248    0xd9f  at /home/john/examples/ch-10/memory/ex-mtrace.c:23
0x0804bff0     0xc3  at /home/john/examples/ch-10/memory/ex-mtrace.c:23
0x0804c0b8    0x22f  at /home/john/examples/ch-10/memory/ex-mtrace.c:23
```

Когда утилита `mtrace` имеет дело с кодом на C++, пользы от нее не очень много. Она корректно докладывает о числе и размерах утечек памяти в коде C++, однако не может идентифицировать номер строки утечки. Возможно, причиной является то, что `mtrace` следует за вызовом `malloc`, а не за вызовом `new`. Поскольку вызов `malloc` совершается стандартной библиотекой C++, возвращаемый указатель не указывает на модуль с отладочными символами.

Сбор статистических сведений о памяти при помощи memusage

Для того чтобы пользоваться утилитой `memusage`, вам вообще не нужно инструментировать свой программный код. В дистрибутиве Fedora данную утилиту можно отыскать в пакете `glibc-utils`. Она сообщает, какой объем памяти использует определенная программа, оформляя итоговый вывод в виде гистограммы. Генерируемые по умолчанию сведения отправляются на стандартный вывод, при этом для демонстрации графической гистограммы используется ASCII-текст. Рассмотрим пример:

```
$ memusage awk 'BEGIN{print "Hello World"}'  Вывод сведений об использовании
                                             памяти программой awk
Hello World

Memory usage summary: heap total: 3564, heap peak: 3548, stack peak: 8604
         total calls   total memory   failed calls
 malloc|         28           3564              0
realloc|          0              0              0 (in place: 0, dec: 0)
 calloc|          0              0              0
   free|         10             48
Histogram for block sizes:
    0-15            21  75% ================================================
   16-31             3  10% =======
   32-47             1   3% ==
```

```
  48-63             1   3% ==
 112-127            1   3% ==
3200-3215           1   3% ==
```

Как и все функции, имеющиеся в библиотеке `glibc-utils`, утилита `memusage` не имеет страницы руководства man и страницы info. С помощью параметра `--help` можно вывести на экран информацию о некоторых полезных свойствах `memusage`, например о том, что данная утилита способна отслеживать вызовы `mmap` и `munmap` в дополнение к вызовам `malloc` и `free`:

```
$ memusage --help
Usage: memusage [OPTION]... PROGRAM [PROGRAMOPTION]...
Profile memory usage of PROGRAM.
   -n,--progname=NAME     Имя файла программы для профилирования
   -p,--png=FILE          Генерирование графической информации в формате PNG
                          с сохранением в указанный файл
   -d,--data=FILE         Генерирование двоичных данных с сохранением
                          в указанный файл
   -u,--unbuffered        Не буферизировать вывод
   -b,--buffer=SIZE       Сбор элементов указанного размера перед их записью
      --no-timer          Не собирать дополнительные сведения по таймеру
   -m,--mmap              Отслеживание mmap и ему подобных
   -?,--help              Вывод справочной информации и завершение работы
      --usage             Вывод короткого сообщения об использовании
   -V,--version           Вывод информации о версии и завершение работы

The following options only apply when generating graphical output:
   -t,--time-based        Генерирование линейного графика во времени
   -T,--total             Генерирование графика суммарного использования
                          памяти
      --title=STRING      Использовать указанную строку в качестве заголовка
                          графика
   -x,--x-size=SIZE       Изменить ширину графических пикселов
   -y,--y-size=SIZE       Изменить высоту графических пикселов

Mandatory arguments to long options are also mandatory for any corresponding
short options.

For bug reporting instructions, please see:
<http://www.gnu.org/software/libc/bugs.html>.
```

Одна из примечательных возможностей утилиты `memusage` заключается в способности генерировать графический вывод с сохранением его в формате PNG[1], пример которого можно увидеть на рис. 10.2.

Утилита `memusagestat` позволяет создавать PNG-файлы из данных, которые генерируются при помощи `memusage` с параметром `d`. Оба этих инструмента постоянно совершенствуются.

---

[1] PNG — это сокращение от Portable Network Graphics, так называется открытый формат для распространения изображений.

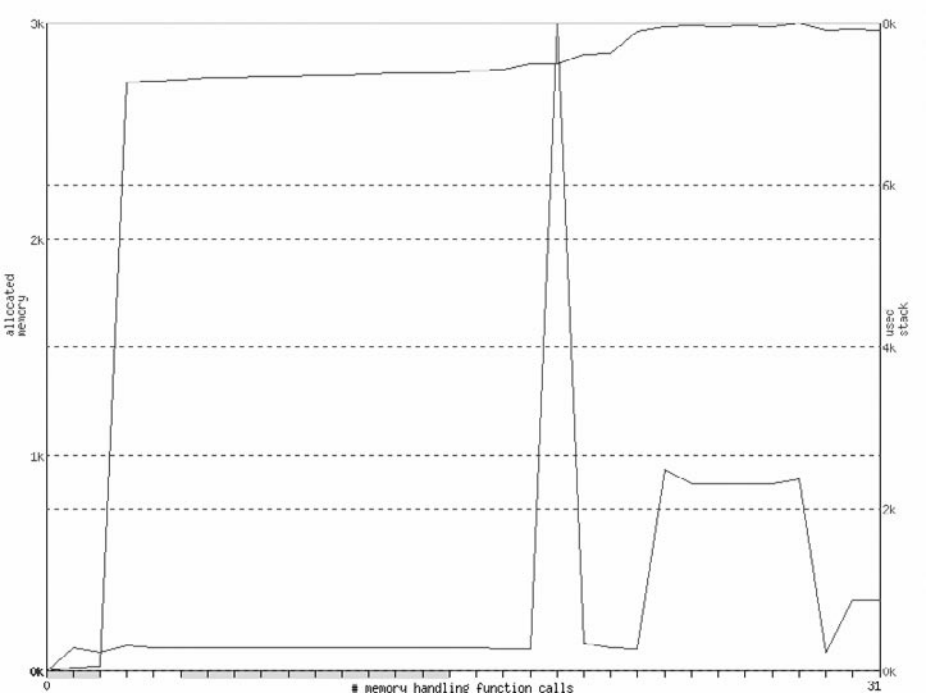


Рис. 10.2. Графический вывод, генерируемый утилитой memusage

10.5.5. Использование Valgrind для устранения проблем с памятью

В главе 9 я использовал Valgrind[1] для демонстрации возможностей по отладке проблем с кэшем, однако данный инструмент чаще применяется для устранения проблем с памятью. Фактически именно это происходит по умолчанию, когда пользователь вводит команду `valgrind` без параметра `--tool`. Вызов `valgrind` без применения аргументов будет эквивалентом следующей команды:

```
$ valgrind --tool=memcheck ./myprog
```

Преимущество Valgrind заключается в том, что вам не нужно инструментировать программный код, для того чтобы произвести отладку своего приложения. Платой за это является снижение производительности. Valgrind значительно замедляет скорость выполнения программного кода. Вторым недостатком использования данного инструмента является то, что некоторые итоговые результаты нельзя увидеть до тех пор, пока выполнение программы не завершится. В частности, утечки в силу их природы не всегда могут быть выявлены раньше, чем завершится программа.

[1] См. сайт www.valgrind.org.

Выявление утечек при помощи Valgrind

Если команда `valgrind` вводится без аргументов, итоговый вывод будет включать сведения о выявленных утечках после завершения выполнения программы. Для получения более детализированного вывода необходимо использовать параметр `--leakcheck=full`. В листинге 10.4 содержатся примеры двух типов утечек памяти, которые может выявлять Valgrind.

Листинг 10.4. `leaky.c: примеры утечек`

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <stdlib.h>
 4
 5 char *possible_leak(int x)
 6 {
 7   // Статический указатель — инструменту valgrind неизвестно, нужен ли он
 8     static char *lp;
 9     char *p = (char *) malloc(x);
10     lp = p + x / 2;
11     return p;
12 }
13
14 int main(int argc, char *argv[])
15 {
16   // Определенная утечка
17     char *p1 = malloc(0x1000);
18
19   // Возможная утечка
20     char *p2 = possible_leak(0x1000);
21     return 0;
22 }
```

К первому типу утечки, о которой сообщает Valgrind, относится простая утечка. В листинге 10.4 указатель `p1` был распределен и отброшен, но не освобожден. Поскольку данный указатель выходит за пределы области, блок *определенно* имеет утечку. Второе распределение происходит внутри функции `possible_leak`. На этот раз я включил статический указатель, который указывает на место где-то в середине наиболее недавно распределенного блока. Это может быть значение, которое будет использоваться при следующем вызове, или просто оплошность, допущенная при программировании. Valgrind не видит разницы, поэтому сообщает о наличии *возможной* утечки:

```
$ valgrind --quiet --leak-check=full ./leaky
==22309==
==22309== 4,096 bytes in 1 blocks are possibly lost in loss record 1 of 2
==22309==    at 0x40044C9: malloc (vg_replace_malloc.c:149)
==22309==    by 0x804838D: possible_leak (leaky.c:9)
==22309==    by 0x80483E8: main (leaky.c:20)
==22309==
==22309==
==22309== 4,096 bytes in 1 blocks are definitely lost in loss record 2 of 2
```

```
==22309==    at 0x40044C9: malloc (vg_replace_malloc.c:149)
==22309==    by 0x80483D5: main (leaky.c:17)
```

Обратите внимание на то, что *возможная* утечка указывает на функцию possible_leak как на источник утечки. Если вам доведется столкнуться с подобным в своем программном коде, вы будете знать, что именно нужно искать.

Поиск повреждений памяти при помощи Valgrind

Valgrind позволяет наряду с повреждениями динамической памяти выявлять повреждения основной памяти. Так, в частности, данный инструмент может обнаруживать переполнение величиной 1 байт, которое может пройти незамеченным для glibc. В листинге 10.5 приведен пример программы, при выполнении которой произошло однобайтовое переполнение.

Листинг 10.5. new-corrupt.cpp: пример повреждения динамической памяти в программе C++

```
1 #include <string.h>
2
3 int main(int argc, char *argv[])
4 {
5     int *ptr = new int;
6     memset(ptr, 0, sizeof(int) + 1);     // Переполнение величиной 1 байт
7     delete ptr;
8 }
```

Запустив данную программу с использованием команды valgrind, мы выявляем следующую проблему:

```
$ valgrind --quiet ./new-corrupt
==14780== Invalid write of size 1
==14780==    at 0x80484B9: main (new-corrupt.cpp:6)
==14780==  Address 0x402E02C is 0 bytes after a block of size 4 alloc'd
==14780==    at 0x4004888: operator new(unsigned) (vg_replace_malloc.c:163)
==14780==    by 0x80484A9: main (new-corrupt.cpp:5)
```

Необходимо отметить, что инструмент Valgrind может выявлять переполнение только динамической памяти. Он не способен обнаруживать переполнение стека или статической памяти.

Анализ динамической памяти при помощи massif

Инструмент massif входит в состав Valgrind и применяется для вывода сведений о суммарном использовании динамической памяти определенной функцией. В качестве иллюстрации рассмотрим пример. Программа, приведенная в листинге 10.6, называется funalloc и содержит две функции, которые вызывают утечку памяти.

Листинг 10.6. funalloc.c: функции распределения памяти

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #include <unistd.h>
5
6 void func1(void)
```

```
 7 {
 8     malloc(1024);             // Преднамеренная утечка
 9 }
10
11 void func2(void)
12 {
13     malloc(1024);             // Преднамеренная утечка
14 }
15
16 int main(int argc, char *argv[])
17 {
18     srand(0);
19     int i;
20     for (i = 0; i < 256; i++) {
21         int r = rand() % 100;
22         if (r > 75) {
23             func2(); // Подвергается вызову около 25 % от общего времени
24         }
25         else {
26             func1(); // Подвергается вызову около 75 % от общего времени
27         }
28         // Визуализировать образчики с использованием графика
29         usleep(1);
30     }
31 }
```

Программа `funalloc` использует псевдопроизвольную последовательность для обеспечения непредсказуемости, однако статически вызывает `func1` на протяжении примерно 75 % общего времени. Поскольку обе функции распределяют одинаковое количество памяти, в итоге мы увидим, что функция `func1` будет «отвечать» приблизительно за 75 % используемой динамической памяти.

Для выполнения приведенной программы необходимо вести следующую командную строку:

```
$ valgrind --tool=massif ./funalloc
```

В результате будут созданы два файла: `massif.PID.ps` и `massif.PID.txt`. В текстовом файле можно будет отыскать сведения, которые нам уже известны:

```
Command: ./funalloc

== 0 ===========================
Heap allocation functions accounted for 97.9% of measured spacetime

Called from:
(Вызов из:)
  73.5% : 0x8048432: func1 (funalloc.c:8)

  24.4% : 0x804844A: func2 (funalloc.c:13)
(дополнительные данные были удалены)
```

Весьма интересным также может оказаться и графическое представление, пример которого можно увидеть на рис. 10.3. Каждая функция в программе здесь

выделена отдельным цветом. Высота графика отражает общее количество используемой динамической памяти. Распределения динамической памяти уменьшают использование стека, что может быть особенно важно в случае с многопоточными приложениями.

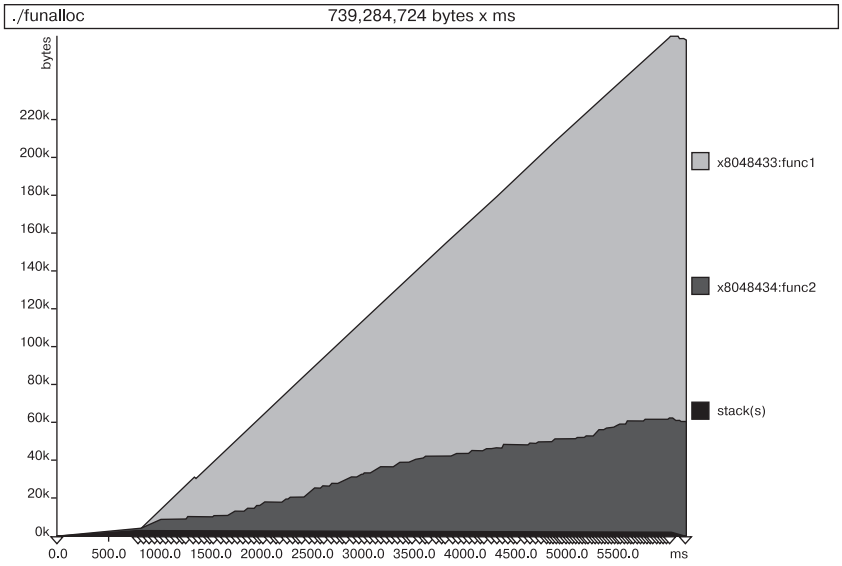


Рис. 10.3. График использования динамической памяти функциями, сгенерированный с помощью инструмента massif

Заключительные положения по Valgrind

При использовании любых Valgrind-инструментов информация о номере строки будет доступна только в отношении модулей, компиляция которых производилась с включенной отладкой. Если же отладка не была активирована, итоговый вывод будет содержать ошибки. В большинстве случаев рекомендуется перенаправлять вывод в файл журнала для последующего просмотра с использованием редактора. Для этого необходимо задействовать параметр `--log-file`.

Valgrind представляет собой весьма мощное средство. Мы лишь слегка затронули его функциональные возможности. В некоторых ситуациях снижение уровня производительности может убедить вас отказаться от его использования. Существуют и другие инструменты, которые работают быстрее и генерируют грубый вывод, однако лишь малая их часть может сравниться с Valgrind по уровню детализированности итоговых сведений, выводимых на экран.

Кроме `memcheck`, `massif` и `cachegrind`, вы также можете использовать `callgrind` и `helgrind`. Инструмент `callgrind` является профайлером функций, напоминающим `gprof`, но при этом он также содержит ряд особенностей инструмента `cachegrind`. `helgrind` — это средство, используемое для поиска состояний гонки в многопоточных приложениях. Данные инструменты слишком сложны для того, чтобы их рассматривать в этой книге, однако вы можете самостоятельно исследовать их в собственной системе.

10.5.6. Выявление переполнения с помощью инструмента Electric Fence

Инструмент Electric Fence использует ряд хитрых методик для выявления ситуаций переполнения динамической памяти сразу же, как только они происходят, в отличие от библиотеки glibc, которая может делать это только постфактум. Несмотря на то что Valgrind также позволяет выполнять аналогичные действия, важная особенность Electric Fence заключается в том, что для отслеживания вызывающего проблему программного кода он использует аппаратный блок управления памятью Memory Management Unit (MMU). Поскольку MMU-блок выполняет основную работу, влияние инструмента Electric Fence на уровень производительности будет минимальным.

Для того чтобы использовать Electric Fence, вам не потребуется инструментировать свой программный код. Вместо этого данный инструмент предусматривает наличие динамической библиотеки, которая реализует альтернативные версии функций распределения динамической памяти. Сценарий-обертка с именем ef позаботится о присваивании переменной окружения LD_PRELOAD необходимого значения. Как и в случае с Valgrind, все, что нужно сделать, — запустить необходимый процесс при помощи команды ef. Вы можете запустить программу из листинга 10.5, воспользовавшись при этом инструментом Electric Fence, как показано далее:

```
$ ef ./new-corrupt
  Electric Fence 2.2.0 Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>
/usr/bin/ef: line 20: 23227 Segmentation fault   (core dumped) ( export
LD_PRELOAD=libefence.so.0.0; exec $* )
```

Вы можете перевести gdb в режим «посмертной» отладки и получить точную трассу стека, для того чтобы узнать, где именно произошло переполнение. Если вам не нужен файл образа памяти, то вы можете запустить свою программу в gdb с использованием Electric Fence двумя способами. Первый из них заключается в связывании своей программы со статической библиотекой, которая входит в состав Electric Fence:

```
$ g++ -g -o new-corrupt new-corrupt.cpp -lefence    Связывание с libefence.a
```

Второй способ состоит в задействовании динамических библиотек из gdb путем присваивания переменной окружения LD_PRELOAD значения в командной оболочке gdb, как показано далее:

```
$ gdb ./new-corrupt

...
(gdb) set environment LD_PRELOAD libefence.so
(gdb) run
Starting program: /home/john/examples/ch-10/memory/new-corrupt

  Electric Fence 2.2.0 Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>
Reading symbols from shared object read from target memory...done.
Loaded system supplied DSO at 0xffffe000

  Electric Fence 2.2.0 Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>
```

```
Program received signal SIGSEGV, Segmentation fault.
0x080484b9 in main (argc=1, argv=0xbfb8a404) at new-corrupt.cpp:7
7               memset(ptr,0,sizeof(int)+1);
(gdb)
```

Это позволит избежать генерирования файла образа памяти, что произошло бы в иной ситуации при выполнении данной программы. Интуиция подсказывает, что запуск gdb наряду с использованием Electric Fence возможен при помощи команды ef. Конечно, сделать это не получится, если вы имеете дело с версией Electric Fence 2.2.2, в силу конструктивных особенностей сценария ef, а также по той причине, что gdb версии 6.3 содержит 0 байт malloc:

```
$ ef gdb ./new-corrupt

  Electric Fence 2.2.0 Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>

ElectricFence Aborting: Allocating 0 bytes, probably a bug.
/usr/bin/ef: line 20: 23307 Illegal instruction   (core dumped) ( export
LD_PRELOAD=libefence.so.0.0; exec $* )
```

Этого можно избежать, если присвоить переменной окружения EF_ALLOW_MALLOC значение, равное 0, благодаря чему libefence не будет «беспокоиться» о распределениях размером 0 байт. Для изменения поведения Electric Fence можно также присваивать соответствующие значения и другим переменным окружения, о чем можно подробнее узнать на странице руководства man efence(3).

Еще одна особенность Electric Fence заключается в способности выявлять ситуации не только переполнения, но и опустошения буфера. Опустошение происходит, когда процесс осуществляет запись по адресу, предшествующему блоку памяти. Подобный тип ошибки случается с арифметикой указателей, как показано далее:

```
 char *buf = malloc(1024);
...
char *ptr = buf + 10;     Стиль не совсем удачный, но вы можете использовать
                          отрицательные индексы с ptr
...
*(ptr - 11) = '\0';       Произошло то, чего мы и добивались: опустошение!
```

Для того чтобы выявить подобное опустошение с использованием Electric Fence, необходимо присвоить переменной окружения EF_PROTECT_BELOW соответствующее значение:

```
$ EF_PROTECT_BELOW=1 ef ./underrun

  Electric Fence 2.2.0 Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>
/usr/bin/ef: line 20: 4644 Segmentation fault  (core dumped) ( export LD_
PRELOAD=libefence.so.0.0; exec $* )
```

Я сократил данный пример, однако механизм отладки остался таким же, как и прежде. Ошибка приводит к генерированию сигнала SIGSEGV, в результате чего мы можем узнать, какая именно строка породила эту ошибку.

Electric Fence осуществляет распределение дополнительной страницы, предназначенной только для чтения, вслед за каждым распределенным блоком, для того

чтобы в случае переполнения был сгенерирован сигнал SIGSEGV. Поскольку данный инструмент распределяет дополнительную страницу для каждого блока независимо от его размера, библиотека может заставить ваше приложение потреблять намного больше памяти, чем обычно. Что касается приложений, которые распределяют множество небольших блоков, то использование ими динамической памяти значительно возрастет.

Другая проблема возникает в связи с выравниванием блоков в библиотеках распределения. Переполнение величиной 1 байт бывает весьма сложно выявить при определенных обстоятельствах, например в ситуации с программой из листинга 10.2. Инструмент Electric Fence также не способен детектировать переполнение подобного рода.

В завершение необходимо отметить, что единственный механизм, который Electric Fence использует для информирования пользователя о произошедших ошибках, заключается в генерировании сигнала SIGSEGV, что не очень информативно. Таким образом, несмотря на то что выявить подобные проблемы можно и при помощи Electric Fence, для того чтобы разобраться в них, вам может потребоваться другой инструмент. Тем не менее Electric Fence обеспечивает пусть и не совершенный, но весьма неплохой способ проверки программного кода на предмет серьезных ошибок.

10.6. Использование нестандартных методик

Несмотря на доступность множества инструментов для проведения отладки, бывают ситуации, когда приходится прибегать к нестандартным методикам. В случае с некоторыми приложениями использование отладчика оказывается слишком сложным, а замещение одних библиотек другими вызывает проблемы.

10.6.1. Создание собственного «черного ящика»

Вам, вероятно, доводилось слышать о так называемых «черных ящиках», которые устанавливаются в коммерческих авиалайнерах. Подобные устройства используются при расследовании катастроф и позволяют определить причину трагедии. В «черном ящике» содержится история измерения различных показателей начиная с определенного времени в прошлом и до момента катастрофы. Исследовав эти данные, запись которых ведется вплоть до падения самолета, можно выявить возможную причину катастрофы.

Вы можете создать программный эквивалент «черного ящика», который будет использоваться для приложений. Подобная методика имеет ряд преимуществ по сравнению с применением отладчика.

- Вы сможете полностью контролировать то, какие сведения будут регистрироваться, а какие — нет. В результате можно сохранить высокий уровень производительности, обеспечивая при этом наличие некоторого объема отладочной информации.

- Данную методику можно использовать в сочетании с оптимизированными исполняемыми файлами, при этом вовсе не обязательно, чтобы компиляция осуществлялась с активированной отладкой.
- Подобная методика может оказаться особенно эффективной при устранении переполнения стека. Как уже отмечалось, переполнение стека обычно приводит к тому, что информация, содержащаяся в трассе стека, оказывается неверной, что делает отладку почти бесполезной.

В листинге 10.7 приведен пример полноценного приложения, которое позволяет создавать программный «черный ящик» в форме *буфера трассировки* (данный термин более распространен в сфере программного обеспечения).

Листинг 10.7. trace-buffer.c: пример полноценного приложения для создания программного «черного ящика»

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <stdlib.h>
 4 #include <stdarg.h>
 5
 6 // Глобальный буфер сообщений. Необходимо присвоить легко запоминающееся имя
 7 // Выбрать подходящий размер
 8 char tracebuf[4096] = "";
 9 char *mstart = tracebuf;
10
11 // Прототип printf-подобной функции. Мы будем использовать директиву
   // GNU __attribute__
12 // для включения свободной проверки формата
13 int dbgprintf(const char *fmt, ...)
14     __attribute__ ((__format__(__printf__, 1, 2)));
15
16 // printf-подобная функция отправляет данные в буфер трассировки
17 int dbgprintf(const char *fmt, ...)
18 {
19     int n = 0;
20
21     // ref. stdarg(3)
22     va_list ap;
23     va_start(ap, fmt);
24
25     // Количество символов, доступных для snprintf
26     int nchars = sizeof(tracebuf) - (mstart - tracebuf);
27
28     if (nchars <= 2) {
29         // Циклический буфер
30         mstart = tracebuf;
31         nchars = sizeof(tracebuf);
32     }
33
34     // Запись сообщения в буфер
35     n = vsnprintf(mstart, nchars, fmt, ap);
```

```
36      mstart += n + 1;
37
38      va_end(ap);
39      return n;
40  }
41
42  int defective(int x)
43  {
44      int y = 1;
45      dbgprintf("defective(%u)", x);
46      if (x == 10) {
47          dbgprintf("time to corrupt the stack!");
48
49          // Осуществить переполнение стека на 128 байт (этого должно быть
            // достаточно)
50          memset(&y, 0xa5, sizeof(y) + 128);
51
52          // Скорее всего, все не закончится до тех пор, пока мы не
            // попытаемся выполнить возврат
53          dbgprintf("I'm still here; returning now.");
54          return 0;
55      }
56
57      return defective(x + 1);
58  }
59
60  int main(int argc, char *argv[])
61  {
62      defective(1);
63      dbgprintf("exiting...");
64      return 0;
65  }
```

Функция `dbgprintf` представляет собой `printf`-подобную функцию, которая осуществляет запись в блок глобальной памяти, вместо того чтобы записывать на стандартный вывод. Память используется в качестве циклического буфера, поэтому, когда буфер окажется заполненным, самые старые сообщения будут перезаписаны самыми новыми. Количество «исторических» сведений, которое мы получаем, будет определяться размером распределяемой памяти.

Оставшаяся часть программы содержит одиночную функцию, которая при определенных обстоятельствах вызывает переполнение стека. Это рекурсивная функция, которая увеличивает свой входной параметр и осуществляет самовызов уже с новым значением. Когда данное значение достигает `10`, программа вызывает переполнение стека. Как вы уже догадались, в результате генерируется файл образа памяти. При этом также вполне ожидаемо, что в данном файле будет отсутствовать пригодная к использованию трасса стека. Рассмотрим пример:

```
 $ cc -g -o trace-buffer trace-buffer.c
./trace-buffer
Segmentation fault (core dumped)
```

```
$ gdb ./trace-buffer core.25347
...
#0  0xa5a5a5a5 in ?? ()              Бессмысленный адрес является первым
                                     признаком проблемы!
(gdb) bt                             Вводим команду bt (backtrace)
#0  0xa5a5a5a5 in ?? ()              и смотрим на результат…
#1  0xa5a5a5a5 in ?? ()
...                                  Снова то же самое
#22 0xa5a5a5a5 in ?? ()
---Type <return> to continue, or q <return> to quit---q
Quit
 (gdb) x/15s &tracebuf               Просмотр первых 15 сообщений,
                                     содержащихся в буфере трассировки
                                     tracebuf
0x8049720 <tracebuf>:    "defective(1)"
0x804972d <tracebuf+13>:         "defective(2)"
0x804973a <tracebuf+26>:         "defective(3)"
0x8049747 <tracebuf+39>:         "defective(4)"
0x8049754 <tracebuf+52>:         "defective(5)"
0x8049761 <tracebuf+65>:         "defective(6)"
0x804976e <tracebuf+78>:         "defective(7)"
0x804977b <tracebuf+91>:         "defective(8)"
0x8049788 <tracebuf+104>:        "defective(9)"
0x8049795 <tracebuf+117>:        "defective(10)"
0x80497a3 <tracebuf+131>:        "time to corrupt the stack!"
0x80497be <tracebuf+158>:        "I'm still here; returning now."
0x80497dd <tracebuf+189>:        ""
0x80497de <tracebuf+190>:        ""
0x80497df <tracebuf+191>:        ""
(gdb)
```

Видно, что в данном примере стек является бесполезным «благодаря» функции `defective`, однако с глобальными данными все в порядке. Вы можете воспользоваться этим, заглянув в буфер трассировки при помощи `gdb`. Содержащиеся в нем сообщения представят вам историю того, что происходило до момента краха. Нет гарантий, что вы получите именно интересующие вас сведения, однако вы всегда сможете модифицировать содержимое таким образом, чтобы добиться желаемого результата. Это также сработает в случае с программным кодом, компиляция которого осуществлялась без включенной оптимизации. Кроме того, здесь не обязателен и отладчик — вы можете воспользоваться командой `strings`, для того чтобы сбросить дамп всех ASCII-строк из файла образа памяти и просмотреть сообщения в том порядке, в котором они были записаны.

Реализация, приведенная в листинге 10.7, неидеальна, но довольно проста. Например, когда буфер оказывается заполненным, последнее сообщение будет отбрасываться. Если полное заполнение буфера никогда не будет достигнуто, вы с этой проблемой не столкнетесь. Если буфер все же окажется заполненным, то, скорее всего, отброшено будет одно сообщение. В качестве упражнения вы можете самостоятельно доработать приведенную в данном листинге реализацию.

10.6.2. Получение трасс стека процессов во время их выполнения

Самый простой способ получения трассы стека свой программы заключается в использовании отладчика gdb. Недостатком данного подхода является то, что подключение к выполняющемуся процессу при помощи отладчика требует остановки процесса, ввода определенных команд и повторного запуска процесса. Время, затрачиваемое на остановку выполнения, может оказаться довольно значительным, если прибегать к ней приходится часто. В состав gdb входит недокументированный сценарий gstack, который сам выполняет данную процедуру и тем самым уменьшает время, затрачиваемое на получение трассы стека. Для того чтобы получить трассу стека выполняющегося процесса, необходимо ввести следующую команду:

```
$ gstack pid        Нужно лишь ввести идентификатор соответствующего процесса
```

Преимущество сценария gstack заключается в том, что благодаря его использованию вы затратите намного меньше времени на получение трассы, чем если бы вручную вводили команды в gdb в интерактивном режиме. В трассе стека инструмент gstack отражает только имена функций, то есть для того, чтобы получить полезный вывод, вовсе не обязательно, чтобы исполняемый файл был скомпилирован с включенной отладкой.

10.6.3. Принудительное генерирование файлов образов памяти

Может показаться парадоксальным, что программу нужно заставлять генерировать файл образа памяти, однако это может быть оправданным в ситуациях с ошибками, сложными для «отлова». Примером может стать регрессивное тестирование, проходящее в ночное время. В данном случае вы не сможете предсказать, что может пойти не так, однако, если это все же случится, необходимо получить максимум информации. Именно такой объем информации сможет обеспечить файл образа памяти.

Как вы уже знаете, в большинстве новых дистрибутивов опция генерирования файлов образов памяти активирована по умолчанию. Для того чтобы включить генерирование таких файлов в оболочке bash, необходимо ввести следующую команду:

```
$ ulimit -c unlimited
```

Значения системных лимитов, скорее всего, можно отыскать в /etc/rc.local, однако, как отмечалось в главе 6, существуют веские причины, по которым генерирование файлов образов памяти по умолчанию лучше отключить.

Функции abort и assert

Обе эти функции позволяют завершать выполнение программы и создавать файл образа памяти. Так, в частности, при их использовании генерируется сигнал SIGABRT. Вызов функции abort будет эквивалентом следующего вызова:

```
raise(SIGABRT);
```

Данный вызов следует использовать, когда вы уверены в том, что программа пребывает в неизвестном состоянии (возможно, по причине переполнения памяти) и продолжать ее выполнение неразумно. Вы также можете воспользоваться вызовом exit, однако в этом случае не будет сгенерирован файл образа памяти. Типичный шаблон использования abort выглядит так:

```
if ( ! program is sane ) abort();
```

ANSI C определяет макрос assert, который позволяет «завернуть» все это в одну строку. assert принимает одиночный аргумент, который оценивается как логическое выражение. Если данное выражение — false, то он совершает вызов abort. Итоговый вывод будет содержать номер строки и выражения, которое потерпело неудачу:

```
assert(!insane);
```

Если insane — true, то вывод будет выглядеть следующим образом:

```
$./foo
assert: foo.c:7: main: Assertion '!insane' failed.
Aborted (core dumped)
```

Вы можете деактивировать assert путем компиляции программного кода с использованием препроцессорного макроса NDEBUG. В такой ситуации выражение, заключенное в круглые скобки, будет отбрасываться до того, как компилятор его «увидит». Это может привести к «побочным эффектам», которые возникают, когда пользователь деактивирует assert. Окажите себе услугу: упрощайте свои предположения.

Использование gcore

Обычно файл образа памяти генерируется ядром, когда завершение процесса происходит ненормально. Отладчик gdb позволяет осуществлять сброс файла образа памяти на диск без необходимости завершения процесса. Для этого следует использовать в gdb команду gcore. Когда данная команда задействуется в gdb, при записи файла образа памяти ему будет присваиваться имя core.pid или любое другое по выбору пользователя. В состав gdb входит сценарий оболочки с именем gcore, который позволяет подключаться к процессу с помощью gdb и сбрасывать файл образа памяти на диск посредством команды gcore; затем данный сценарий отсоединяется и позволяет процессу продолжить выполнение.

gcore может принимать в качестве аргумента один или более идентификаторов процессов, при этом имя файла образа памяти можно изменять при помощи параметра -o.

10.6.4. Использование сигналов

Чаще всего сигналы используются для сброса на диск отладочных сведений, когда поступает определенный сигнал. Так, например, вы можете вынудить программу сбросить файл образа памяти на диск, послав ей соответствующий сигнал

вроде SIGBART[1]. В данном случае вы сможете осуществлять управление из оболочки, не прибегая к инструментированию своего программного кода. Для выполнения более активных действий вы можете создать обработчик сигналов, который будет реагировать на один из определяемых пользователем сигналов (SIGUSR1, SIGUSR2) и выводить при этом на экран отладочную информацию, позволяя программе продолжать выполнение.

Упростить своевременное проведение отладки можно также при помощи сигналов SIGSTOP и SIGCONT. В некоторых примерах, приведенных в данной книге, я использовал сигнал SIGSTOP для приостановки процесса с целью обеспечения читателю возможности исследовать его состояние перед тем, как он будет завершен. Это так же легко сделать, как добавить в свой программный код следующую строку:

```
raise(SIGSTOP);
```

Это будет сродни нажатию сочетания клавиш Ctrl+Z в отношении процесса в приоритетном режиме в оболочке. Остановить процесс, выполняющийся в фоновом режиме, можно, послав ему из оболочки сигнал SIGSTOP:

```
$ kill -STOP pid
```

После того как процесс остановлен, ему необходимо послать сигнал SIGCONT, для того чтобы продолжить его выполнение. Из оболочки это сделать очень просто:

```
$ kill -CONT pid
```

Если процесс остановлен, вы можете ознакомиться с его состоянием без каких-либо состояний гонки, то есть взглянуть на вещи, не изменяя их. В частности, для вас, скорее всего, будет представлять интерес файловая система /proc.

10.6.5. Использование procfs при проведении отладки

Как уже отмечалось, procfs обращается к драйверу файловой системы, который используется для формирования отчетов о процессах и монтируется в /proc. Данная файловая система содержит внушительное количество системной информации; для каждого процесса здесь отводится по одному каталогу. Каждому подкаталогу присваивается имя в соответствии с идентификатором процесса, в нем размещаются сведения, свойственные только данному процессу. Единственным исключением является /proc/self, который представляет собой символическую ссылку на каталог выполняющегося процесса. Точнее, когда выполняющийся процесс открывает файлы в /proc/self, на самом деле он открывает файлы в каталоге, который ему отведен.

Большинство информации о состоянии процесса можно почерпнуть из вывода команды ps, а также команд, входящих в состав пакета procps. Настоятельно рекомендуется использовать эти команды каждый раз, когда это возможно, так

[1] На самом деле вы можете послать любой сигнал генерирования файла образа памяти, например SIGSEGV или SIGBUS, и процесс будет завершен со сбросом на диск такого файла.

как в некоторых ситуациях, характерных исключительно для файловой системы /proc, они могут оказаться бесценными отладочными инструментами.

Выполняющийся процесс — это движущаяся цель, и вы должны учитывать это, когда будете использовать данные, извлекаемые из файловой системы procfs. При исследовании таких сведений всегда рекомендуется убедиться в том, что соответствующий процесс в данный момент не выполняется. Процесс не обязательно должен быть приостановлен, он в равной степени может быть в состоянии «спячки» или блокировки из-за какого-либо системного вызова. Если при ознакомлении с интересующими вас сведениями процесс не выполняется, то им можно доверять.

Карты памяти

Ранее в данной книге я рассказывал о команде pmap, с помощью которой можно просматривать карту памяти какого-либо процесса. Это может оказаться полезным при отслеживании утечек памяти, которые обычно вызывают фрагментацию. Так, в частности, когда malloc использует вызовы mmap для распределения блоков, эти блоки будут «видны» на карте памяти. Блоки *могут* непосредственно соответствовать блокам, распределенным при выполнении вашей программы.

«Сырые», нефильтрованные данные, извлекаемые посредством pmap, располагаются в /proc/PID/maps. В большинстве случаев использование команды pmap является более предпочтительным. Для получения дополнительных подробностей относительно формата карт памяти procfs см. consult proc(5).

Окружение процесса

Внутри каждого каталога можно отыскать полезные сведения, касающиеся окружения процесса, включая переменные окружения, а также текущий рабочий каталог и имя исполняемого файла. Переменные окружения располагаются в /proc/PID/environ в виде обычного массива символов, в котором переменные окружения отделены друг от друга NUL-разделителями. Проще всего увидеть их из оболочки при помощи команды strings:

```
$ strings -n1 /proc/PID/environ
MANPATH=:/home/john/usr/share/man
HOSTNAME=redhat
TERM=xterm
SHELL=/bin/bash
HISTSIZE=1000 ...                              Вывод имеет продолжение
```

Данный файл предназначен только для чтения, а это означает, что вы не сможете модифицировать окружение выполняющегося процесса из оболочки. Я использовал параметр -n1 в сочетании с командой strings потому, что по умолчанию эта команда фильтрует строки длиной менее 4 символов. В данном примере она выводит на экран строки длиной даже 1 символ, то есть все имеющиеся строки.

Еще одним полезным файлом является /proc/cmdline, в котором хранится командная строка процесса в аналогичном формате. Благодаря этому вы можете точно узнать, как выглядит вектор argv процесса при его выполнении, например:

```
$ strings -n1 /proc/self/cmdline | cat -n
     1  strings                                  Содержимое argv[0]
```

```
2  -n1                                  Содержимое argv[1]
3  /proc/self/cmdline                   Содержимое argv[2]
```

Здесь потребуется кое-что прикинуть в уме, поскольку в строке 1 содержится `argv[0]`, но идея должна быть вам понятна.

Открытые файлы

Из каталога `/proc` процесса можно узнать также сведения об открытых файлах и каталогах. В `/proc/PID/exe` содержится символическая ссылка на исполняемый файл, который использовался для запуска процесса, например:

```
$ ls -l /proc/self/exe
lrwxrwxrwx  1 john john 0 Jul 30 11:53 /proc/self/exe -> /bin/ls
```

Аналогично `/proc/PID/cwd` представляет собой символическую ссылку на текущий рабочий каталог процесса. Она пригодится вам при подключении к серверному процессу, который осуществляет запись в свой текущий рабочий каталог. Если данная ссылка не позволит вам узнать, где находится этот каталог, вы можете обратиться к `procfs` и самостоятельно отыскать его.

И наконец, вы можете узнать, какие файлы были открыты процессом, взглянув на его файловые дескрипторы в `/proc/PID/fd`. Это подкаталог, в котором содержатся символические ссылки на дескриптор каждого открытого файла. Имя ссылки — это путь к открытому файлу, если он располагается в файловой системе. Если файл является сокетом или каналом, `readlink` этого файла обычно будет содержать текстовые индикаторы, позволяющие определить природу файлового дескриптора, например:

```
$ ls -l /proc/self/fd | tee /dev/null
total 4
lrwx------  1 john john 64 Jul 30 12:00 0 -> /dev/pts/3
l-wx------  1 john john 64 Jul 30 12:00 1 -> pipe:[33209]
lrwx------  1 john john 64 Jul 30 12:00 2 -> /dev/pts/3
lr-x------  1 john john 64 Jul 30 12:00 3 -> /proc/5487/fd
```

Обратите внимание на то, что файловые дескрипторы `0` и `2` (стандартный вывод и стандартная ошибка соответственно) указывают на псевдотерминальное устройство. Стандартный вывод передается команде `tee`, поэтому файловый дескриптор `1` выглядит как `pipe:[33209]`. Это не файл, который можно открыть, а канал, при этом `33209` представляет собой индексный дескриптор этого канала. Файловый дескриптор `3` — это открытый каталог. Точнее, это `/proc/self/fd`, что в данном случае указывает на `/proc/5487/fd`.

В главе 6 мы рассмотрели инструменты `lsof` и `fuser`, которые активно используют в своей работе приведенные ранее сведения. Выбор того или иного инструмента зависит от обстоятельств, однако иногда бывает так, что все необходимое можно узнать при помощи простой команды `ls` в `/proc`.

10.7. Заключение

В данной главе мы изучили ряд инструментов и вопросов, связанных с отладкой пользовательского программного кода. Нами были рассмотрены методики

использования printf для отладки программ, а также некоторые нежелательные «побочные эффекты», которые возникают при этом.

Кроме того, в этой главе мы детально исследовали GNU-отладчик gdb на примере как основного, так и второстепенного его использования. Вы увидели примеры того, как можно наиболее эффективно применять gdb при определенных условиях, а также то, что в иных ситуациях он может довольно значительно уменьшить производительность программ пользователя. В дополнение к этому мы также рассмотрели специфические аспекты, касающиеся gdb-отладки программного кода, написанного на языке C++.

Рассмотрены были также уникальные проблемы, связанные с совместно используемыми объектами, которые бывает сложно устранить. Вы узнали, как следует запускать временное выполнение процесса в сочетании с определенным совместно используемым объектом, который может находиться на стадии разработки, без необходимости замены общесистемного экземпляра этого объекта. Я также продемонстрировал, как это делается при помощи отладчика gdb.

В завершение вы изучили ряд инструментов для разрешения проблем с памятью, возникающих в процессе работы программного кода, таких как переполнение памяти и повреждение стека. Были рассмотрены как популярные инструменты, так и нестандартные методики.

10.7.1. Инструментарий, использованный в этой главе

- Electric Fence — инструмент для поиска случаев переполнения или опустошения буфера памяти, основанный на использовании аппаратного блока управления памятью Memory Management Unit.
- gdb — GNU-отладчик, служит для интерактивной отладки процессов.
- mtrace — входит в состав библиотеки glibc и позволяет выявлять утечки памяти после выполнения программ.
- Valgrind — мощный инструмент для поиска утечек памяти, повреждений памяти и решения многих других задач.

10.7.2. Веб-ссылки

- http://duma.sourceforge.net — дочерний ресурс, где можно отыскать сведения об инструменте Electric Fence.
- http://perens.com/FreeSoftware/ElectricFence — оригинальная домашняя страница инструмента Electric Fence.
- www.valgrind.org — домашняя страница проекта Valgrind.

10.7.3. Рекомендуемая литература

Robbins A. GDB Pocket Reference. — Sebastopol, Calif.: O'Reilly Media, Inc., 2005.