



# Программирование для Linux<sup>®</sup>

Профессиональный подход

New  
Riders



CODESOURCERY  
LLC

Данная книга в основном посвящена программированию в среде GNU/Linux. Авторы применяют обучающий подход, последовательно излагая самые важные концепции и методики использования расширенных возможностей системы GNU/Linux в прикладных программах. Читатели научатся писать программы, к интерфейсу которых привыкли пользователи Linux; освоят такие технологии, как многозадачность, многопоточное программирование, межзадачное взаимодействие и взаимодействие с аппаратными устройствами; смогут улучшить свои программы, сделав их быстрее, надежнее и безопаснее; поймут особенности системы GNU/Linux, ее ограничения, дополнительные возможности и специфические соглашения.

Книга предназначена для программистов, уже знакомых с языком C и имеющих базовый опыт работы в GNU/Linux.

- 
- [Программирование для Linux](#)
    - 
    - [Об авторах](#)
    - [Введение](#)
    - [От издательства](#)
    - [Часть I](#)
      - [Глава 1](#)
        - 
        - [1.1. Редактор Emacs](#)
          - 
          - [1.1.1. Открытие исходного файла C/C++](#)
          - [1.1.2. Автоматическое форматирование](#)
          - [1.1.3. Синтаксические выделения](#)
        - [1.2. Компиляторы GCC](#)
          - 
          - [1.2.1. Компиляция одного исходного файла](#)
          - [1.2.2. Компоновка объектных файлов](#)
        - [1.3. Автоматизация процесса с помощью GNU-утилиты make](#)
        - [1.4. GNU-отладчик gdb](#)
          - 
          - [1.4.1. Компиляция с включением отладочной информации](#)
          - [1.4.2. Запуск отладчика](#)
        - [1.5. Поиск дополнительной информации](#)
          - 
          - [1.5.1. Интерактивная документация](#)
          - [1.5.2. Система Info](#)
          - [1.5.3. Файлы заголовков](#)
          - [1.5.4. Исходные тексты](#)
      - [Глава 2](#)
        - 
        - [2.1. Взаимодействие со средой выполнения](#)
          - 
          - [2.1.1. Список аргументов](#)

- [2.1.2. Соглашения по работе с командной строкой в GNU/Linux](#)
  - [2.1.3. Функция getopt\\_long\(\)](#)
  - [2.1.4. Стандартный ввод-вывод](#)
  - [2.1.5. Коды завершения программы](#)
  - [2.1.6. Среда выполнения](#)
  - [2.1.7. Временные файлы](#)
- [2.2. Защита от ошибок](#)
  - 
  - [2.2.1. Макрос assert\(\)](#)
  - [2.2.2. Ошибки системных вызовов](#)
  - [2.2.3. Коды ошибок системных вызовов](#)
  - [2.2.4. Ошибки выделения ресурсов](#)
- [2.3. Создание и использование библиотек](#)
  - 
  - [2.3.1. Архивы](#)
  - [2.3.2. Совместно используемые библиотеки](#)
  - [2.3.3. Стандартные библиотеки](#)
  - [2.3.4. Зависимости между библиотеками](#)
  - [2.3.5. Преимущества и недостатки библиотек](#)
  - [2.3.6. Динамическая загрузка и выгрузка](#)
- [Глава 3](#)
  - 
  - [3.1. Знакомство с процессами](#)
    - 
    - [3.1.1. Идентификаторы процессов](#)
    - [3.1.2. Получение списка активных процессов](#)
    - [3.1.3. Уничтожение процесса](#)
  - [3.2. Создание процессов](#)
    - 
    - [3.2.1. Функция system\(\)](#)
    - [3.2.2. Функции fork\(\) и exec\(\)](#)
    - [3.2.3. Планирование процессов](#)
  - [3.3. Сигналы](#)
  - [3.4. Завершение процесса](#)
    - 
    - [3.4.1. Ожидание завершения процесса](#)
    - [3.4.2. Системные вызовы wait\(\)](#)
    - [3.4.3. Процессы-зомби](#)
    - [3.4.4. Асинхронное удаление дочерних процессов](#)
- [Глава 4](#)
  - 
  - [4.1. Создание потока](#)
    - 
    - [4.1.1. Передача данных потоку](#)
    - [4.1.2. Ожидание завершения потоков](#)
    - [4.1.3. Значения, возвращаемые потоками](#)
    - [4.1.4. Подробнее об идентификаторах потоков](#)

- [4.1.5. Атрибуты потоков](#)
- [4.2. Отмена потока](#)
  - 
  - [4.2.1. Синхронные и асинхронные потоки](#)
  - [4.2.2. Неотменяемые потоки](#)
  - [4.2.3. Когда необходимо отменять поток](#)
- [4.3. Потокосовые данные](#)
  - 
  - [4.3.1. Обработчики очистки](#)
  - [4.3.2. Очистка потокосовых данных в C++](#)
- [4.4. Синхронизация потоков и критические секции](#)
  - 
  - [4.4.1. Состояние гонки](#)
  - [4.4.2. Исключающие семафоры](#)
  - [4.4.3. Взаимоблокировки исключающих семафоров](#)
  - [4.4.4. Неблокирующие проверки исключающих семафоров](#)
  - [4.4.5. Обычные потокосовые семафоры](#)
  - [4.4.6. Сигнальные \(условные\) переменные](#)
  - [4.4.7. Взаимоблокировки двух и более потоков](#)
- [4.5. Реализация потоков в Linux](#)
  - 
  - [4.5.1. Обработка сигналов](#)
  - [4.5.2. Системный вызов clone\(\)](#)
- [4.6. Сравнение процессов и потоков](#)
- [Глава 5](#)
  - 
  - [5.1. Совместно используемая память](#)
    - 
    - [5.1.1. Быстрое локальное взаимодействие](#)
    - [5.1.2. Модель памяти](#)
    - [5.1.3. Выделение сегментов памяти](#)
    - [5.1.4. Подключение и отключение сегментов](#)
    - [5.1.5. Контроль и освобождение совместно используемой памяти](#)
    - [5.1.6. Пример программы](#)
    - [5.1.7. Отладка](#)
    - [5.1.8. Проблема выбора](#)
  - [5.2. Семафоры для процессов](#)
    - 
    - [5.2.1. Выделение и освобождение семафоров](#)
    - [5.2.2. Инициализация семафоров](#)
    - [5.2.3. Операции ожидания и установки](#)
    - [5.2.4. Отладка семафоров](#)
  - [5.3. Отображение файлов в памяти](#)
    - 
    - [5.3.1. Отображение в памяти обычного файла](#)
    - [5.3.2. Примеры программ](#)
    - [5.3.3. Совместный доступ к файлу](#)

- [5.3.4. Частные отображаемые области](#)
- [5.3.5. Применения функции mmap\(\)](#)
- [5.4. Каналы](#)
  - 
  - [5.4.1. Создание каналов](#)
  - [5.4.2. Взаимодействие родительского и дочернего процессов](#)
  - [5.4.3. Перенаправление стандартных потоков ввода, вывода и ошибок](#)
  - [5.4.4. Функции popen\(\) и pclose\(\)](#)
  - [5.4.5. Каналы FIFO](#)
- [5.5. Сокеты](#)
  - 
  - [5.5.1. Концепции сокетов](#)
  - [5.5.2. Системные вызовы](#)
  - [5.5.3. Серверы](#)
  - [5.5.4. Локальные сокеты](#)
  - [5.5.5. Примеры программ, работающих с локальными сокетами](#)
  - [5.5.6. Internet-сокеты](#)
  - [5.5.7. Пары сокетов](#)

○ [Часть II](#)

■ [Глава 6](#)

- 
- [6.1. Типы устройств](#)
- [6.2. Номера устройств](#)
- [6.3. Файловые ссылки на устройства](#)
  - 
  - [6.3.1. Каталог /dev](#)
  - [6.3.2. Доступ к устройству путем открытия файла](#)
- [6.4. Аппаратные устройства](#)
- [6.5. Специальные устройства](#)
  - 
  - [6.5.1. /dev/null](#)
  - [6.5.2. /dev/zero](#)
  - [6.5.3. /dev/full](#)
  - [6.5.4. Устройства генерирования случайных чисел](#)
  - [6.5.5. Устройства обратной связи](#)
- [6.6. Псевдотерминалы](#)
  - 
  - [6.6.1. Пример работы с псевдотерминалом](#)
- [6.7. Функция ioctl\(\)](#)

■ [Глава 7](#)

- 
- [7.1. Извлечение информации из файловой системы /proc](#)
- [7.2. Каталоги процессов](#)
  - 
  - [7.2.1. Файл /proc/self](#)
  - [7.2.2. Список аргументов процесса](#)
  - [7.2.3. Переменные среды процесса](#)

- [7.2.4. Исполняемый файл процесса](#)
  - [7.2.5. Дескрипторы файлов процесса](#)
  - [7.2.6. Статистика использования процессом памяти](#)
  - [7.2.7. Статистика процесса](#)
- [7.3. Аппаратная информация](#)
  - 
  - [7.3.1. Центральный процессор](#)
  - [7.3.2. Аппаратные устройства](#)
  - [7.3.3. Шина PCI](#)
  - [7.3.4. Последовательные порты](#)
- [7.4. Информация о ядре](#)
  - 
  - [7.4.1. Версия ядра](#)
  - [7.4.2. Имя компьютера и домена](#)
  - [7.4.3. Использование памяти](#)
- [7.5. Дисководы, точки монтирования и файловые системы](#)
  - 
  - [7.5.1. Файловые системы](#)
  - [7.5.2. Диски и разделы](#)
  - [7.5.3. Точки монтирования](#)
  - [7.5.4. Блокировки](#)
- [7.6. Системная статистика](#)
- [Глава 8](#)
  - 
  - [8.1. Команда strace](#)
  - [8.2. Функция access\(\): проверка прав доступа к файлу](#)
  - [8.3. Функция fcntl\(\): блокировки и другие операции над файлами](#)
  - [8.4. Функции fsync\(\) и fdatasync\(\): очистка дисковых буферов](#)
  - [8.5. Функции getrlimit\(\) и setrlimit\(\): лимиты ресурсов](#)
  - [8.6. Функция getrusage\(\): статистика процессов](#)
  - [8.7. Функция gettimeofday\(\): системные часы](#)
  - [8.8. Семейство функций mlock\(\): блокирование физической памяти](#)
  - [8.9. Функция mprotect\(\): задание прав доступа к памяти](#)
  - [8.10. Функция nanosleep\(\): высокоточная пауза](#)
  - [8.11. Функция readlink\(\): чтение символических ссылок](#)
  - [8.12. Функция sendfile\(\): быстрая передача данных](#)
  - [8.13. Функция setitimer\(\): задание интервальных таймеров](#)
  - [8.14. Функция sysinfo\(\): получение системной статистики](#)
  - [8.15. Функция uname\(\)](#)
- [Глава 9](#)
  - 
  - [9.1. Когда необходим ассемблерный код](#)
  - [9.2. Простая ассемблерная вставка](#)
    - 
    - [9.2.1. Преобразование функции asm\(\) в ассемблерные инструкции](#)
  - [9.3. Расширенный синтаксис ассемблерных вставок](#)
  -

- [9.3.1. Ассемблерные инструкции](#)
- [9.3.2. Выходные операнды](#)
- [9.3.3. Входные операнды](#)
- [9.3.4. Модифицируемые регистры](#)
- [9.4. Пример](#)
- [9.5. Вопросы оптимизации](#)
- [9.6. Вопросы сопровождения и переносимости](#)
- [Глава 10](#)
  - 
  - [10.1. Пользователи и группы](#)
    - 
    - [10.1.1. Суперпользователь](#)
  - [10.2. Идентификаторы пользователей и групп, закрепленные за процессами](#)
  - [10.3. Права доступа к файлам](#)
    - 
    - [10.3.1. Проблема безопасности: программы без права выполнения](#)
    - [10.3.2. Sticky-бит](#)
  - [10.4. Реальные и эффективные идентификаторы](#)
    - 
    - [10.4.1. Программы с установленным битом SUID](#)
  - [10.5. Аутентификация пользователей](#)
  - [10.6. Дополнительные проблемы безопасности](#)
    - 
    - [10.6.1. Переполнение буфера](#)
    - [10.6.2. Конкуренция доступа к каталогу /tmp](#)
    - [10.6.3. Функции system\(\) и popen\(\)](#)
- [Глава 11](#)
  - 
  - [11.1. Обзор](#)
    - 
    - [11.1.1. Существующие ограничения](#)
  - [11.2. Реализация](#)
    - 
    - [11.2.1. Общие функции](#)
    - [11.2.2. Загрузка серверных модулей](#)
    - [11.2.3. Сервер](#)
    - [11.2.4. Основная программа](#)
  - [11.3. Модули](#)
    - 
    - [11.3.1. Отображение текущего времени](#)
    - [11.3.2. Отображение версии Linux](#)
    - [11.3.3. Отображение объема свободного дискового пространства](#)
    - [11.3.4. Статистика выполняющихся процессов](#)
  - [11.4. Работа с сервером](#)
    - 
    - [11.4.1. Файл Makefile](#)
    - [11.4.2. Создание сервера](#)

- [11.4.3. Запуск сервера](#)
- [11.5. Вместо эпилога](#)
- [Часть III](#)
  - [Приложение А](#)
    - 
    - [А.1. Статический анализ программы](#)
    - [А.2. Поиск ошибок в динамической памяти](#)
      - 
      - [А.2.1. Программа для тестирования динамической памяти](#)
      - [А.2.2. Проверка функции malloc\(\)](#)
      - [А.2.3. Поиск потерянных блоков памяти с помощью утилиты mtrace](#)
      - [А.2.4. Библиотека csmalloc](#)
      - [А.2.5. Библиотека Electric Fence](#)
      - [А.2.6. Выбор средств отладки](#)
      - [А.2.7. Исходный текст программы, работающей с динамической памятью](#)
    - [А.3. Профилирование](#)
      - 
      - [А.3.1. Простейший калькулятор](#)
      - [А.3.2. Сбор профильной информации](#)
      - [А.3.3. Отображение профильных данных](#)
      - [А.3.4. Как работает утилита gprof](#)
      - [А.3.5. Исходные тексты программы-калькулятора](#)
  - [Приложение Б](#)
    - 
    - [Б.1. Чтение и запись данных](#)
      - 
      - [Б.1.1. Открытие файла](#)
      - [Б.1.2. Заккрытие файла](#)
      - [Б.1.3. Запись данных](#)
      - [Б.1.4. Чтение данных](#)
      - [Б.1.5. Перемещение по файлу](#)
    - [Б.2. Функция stat\(\)](#)
    - [Б.3. Векторные чтение и запись](#)
    - [Б.4. Взаимосвязь с библиотечными функциями ввода-вывода](#)
    - [Б.5. Другие низкоуровневые операции](#)
    - [Б.6. Чтение содержимого каталога](#)
  - [Приложение В](#)
  - [Приложение Г](#)
    - 
    - [Г.1. Общая информация](#)
    - [Г.2. Информация о программном обеспечении GNU/Linux](#)
    - [Г.3. Другие ресурсы](#)
  - [Приложение Д](#)
    - [I. Требования к модифицированной и немодифицированной версиям](#)
    - [II. Авторские права](#)
    - [III. Область действия Лицензии](#)
    - [IV. Требования к модифицированным материалам](#)



- [V. Рекомендации](#)
- [VI. Предусмотренные ограничения](#)
- [Дополнение, касающееся политики публикации](#)
- [Приложение E](#)
  - 
  - [Преамбула](#)
  - [Условия копирования, распространения и модификации программных продуктов](#)
  - [Конец условий](#)
  - [Как применить эти требования к новым программным продуктам](#)

○

- [notes](#)

- [1](#)
- [2](#)
- [3](#)
- [4](#)
- [5](#)
- [6](#)
- [7](#)
- [8](#)
- [9](#)
- [10](#)
- [11](#)
- [12](#)
- [13](#)
- [14](#)
- [15](#)
- [16](#)
- [17](#)
- [18](#)
- [19](#)
- [20](#)
- [21](#)
- [22](#)
- [23](#)
- [24](#)
- [25](#)
- [26](#)
- [27](#)
- [28](#)
- [29](#)
- [30](#)
- [31](#)
- [32](#)
- [33](#)
- [34](#)
- [35](#)

- [36](#)
  - [37](#)
  - [38](#)
  - [39](#)
  - [40](#)
  - [41](#)
  - [42](#)
-

Благодарим Вас за использование нашей библиотеки [Librs.net](http://Librs.net).

**Программирование для Linux**

**Профессиональный подход**

**Марк Митчелл**

**Джеффри Оулдем**

**Алекс Самьюэл**

**Марк Митчелл (Mark Mitchell)** получил степень бакалавра вычислительной техники в Гарвардском университете в 1994 году и степень магистра в Станфордском университете в 1999 году. Его научные исследования касались теории сложности вычислений и компьютерной безопасности. Марк принимал участие в разработке коллекции GNU-компиляторов (GCC).

**Джефффри Оулдем (Jeffrey Oldham)** получил степень бакалавра вычислительной техники в университете Райс в 1991 году. После работы в Центре исследования параллельных вычислений он получил степень доктора философии в Станфордском университете в 2000 году. Его научные исследования касались теории алгоритмов. В настоящее время он продолжает разработку коллекции GNU-компиляторов и пишет программы для научных расчетов.

**Алекс Самьюэл (Alex Samuel)** окончил физический факультет Гарвардского университета в 1995 году. Он работал инженером-программистом в компании BBN, после чего вернулся к изучению физики в Станфордском университете. Алекс является администратором проекта Software Carpentry и занимается рядом других проектов, в частности оптимизацией коллекции GCC.

Марк и Алекс основали компанию **CodeSourcery LLC** в 1999 году. Джефффри пришел в компанию в 2000 году. Целями компании являются создание средств разработки для GNU/Linux и других операционных систем; превращение семейства GNU-утилит в стандартный набор средств разработки промышленного качества; выполнение работ под заказ и предоставление консультаций. Адрес Web-узла компании: [www.codesourcery.com](http://www.codesourcery.com).

## О научных консультантах

Эти люди внесли значительный вклад в написание книги. Они просмотрели материал книги на предмет технической грамотности и организации. Их советы и рецензии позволили авторам убедиться в том, что они не обманут ожидания читателей.

**Гленн Бекер (Glenn Becker)** имеет много научных степеней, все в области театрального искусства. В настоящее время он работает онлайн-продюсером в SCIFI.COM интерактивном компоненте канала SCI FI в Нью-Йорке. Дома у него установлена система Debian Linux, и он интересуется такими темами, как системное администрирование, безопасность, локализация программного обеспечения и XML.

**Джон Дин (John Dean)** получил степень бакалавра естественных наук в Шеффилдском университете в 1974 году. В 1986 г. он получил степень магистра систем автоматического управления в Институте наук и технологий в Кранфилде. Работая в компании Roll Royce and Associates, Джон разрабатывал программное обеспечение для систем автоматизированного управления ядерной техникой. После ухода из компании в 1978 г. он работал в нефтехимической промышленности, занимаясь созданием систем управления технологическими процессами. С 1996 по 2000 гг. Джон был добровольным разработчиком компании MySQL, после чего перешел на работу в эту компанию. Джон занимается переносом MySQL в Windows и написанием нового графического клиента MySQL для платформ Windows и X11.

Мы выражаем глубокую признательность Ричарду Сталлману (Richard Stallman), без которого никогда не было бы проекта GNU, и Линусу Торвальдсу (Linus Torvalds) без которого никогда не было бы ядра Linux. Огромное число людей внесло свой вклад в операционную систему Linux и мы благодарим их всех.

Мы благодарим преподавателей университетов Гарварда, Станфорда и Райс, которые учили нас. Без них мы никогда не рискнули бы учить других!

Ричард Стивенс (W. Richard Stevens) написал три великолепные книги по программированию в UNIX, которыми мы постоянно пользуемся. Роланд Маграт (Roland McGrath), Ульрих Дреппер (Ulrich Drepper) и многие другие написали GNU-библиотеку языка C и превосходную документацию к ней.

Роберт Бразил (Robert Brazile) и Сэм Кендалл (Sam Kendall) просмотрели ранние наброски нашей книги и дали советы по ее направленности и содержанию. Наши научные консультанты и рецензенты (особенно Гленн Бекер и Джон Дин) находили ошибки и оказывали нам техническую поддержку. Естественно, оставшиеся ошибки целиком на нашей совести!

Благодарим сотрудников издательства New Riders: Энн Куинн (Ann Quinn) за решение всех вопросов, связанных с публикацией книги: Лору Ловолл (Laura Loveall) за то, что помогла нам уложиться в сроки; Стефани Уолл (Stephanie Wall) за то что вдохновила нас на написание этой книги.

# Введение

Операционная система Linux вихрем ворвалась в мир компьютеров. Было время, когда выбор пользователей был ограничен коммерческими операционными системами и приложениями. У пользователей не было возможности исправлять или улучшать эти программы и часто они были вынуждены принимать довольно жесткие лицензионные условия. С появлением GNU/Linux и других систем с открытым кодом все изменилось. Теперь в распоряжении пользователей, администраторов и разработчиков есть бесплатная операционная система с множеством утилит приложений и со всеми исходными текстами.

Большая доля успеха GNU/Linux приходится на открытую природу этой системы. Поскольку исходные тексты программ общедоступны, любой может принять участие в их разработке, либо путем исправления незначительной ошибки либо путем распространения целого приложения. Это подвигло тысячи разработчиков во всем мире на создание новых программных компонентов и улучшение операционной системы до такой степени, что она сравнялась по своим возможностям с любой коммерческой ОС. В дистрибутивы Linux входят тысячи программ и приложений.

Своим успехом ОС Linux обязана также философии UNIX. Многие программные компоненты, появившиеся в AT&T UNIX и BSD UNIX, продолжили свое существование в Linux и заложили основу для написания новых программ. Философия UNIX, заключающаяся в организации взаимодействия множества небольших утилит командной строки, является главным принципом организации ОС Linux, делающим эту систему столь мощной. Даже когда программы оснащены пользовательским интерфейсом, лежащие в их основе команды доступны для написания сценариев автоматизации. Множество сложных задач можно решить, объединяя существующие команды и программы в простых сценариях.

## *GNU и Linux*

Операционная система Linux названа в честь Линуса Торвальдса, ее автора и создателя ядра системы. Ядро это программа, которая выполняет основные функции операционной системы. Оно взаимодействует с аппаратными устройствами, выделяет память и другие ресурсы, позволяет нескольким программам работать одновременно, управляет файловыми системами и т.д.

Ядро само по себе не располагает средствами взаимодействия с пользователями. Оно не может выдать даже простую строку приглашения на ввод команд. Ядро не позволяет пользователям редактировать файлы, взаимодействовать с другими компьютерами или писать программы. Для решения всех этих задач требуется большое число других программ, включая интерпретаторы команд, редакторы и компиляторы. Многие из этих программ пользуются библиотеками функций общего назначения, не включенными в ядро.

В системах GNU/Linux большинство таких программ разработано в рамках проекта GNU.<sup>[1]</sup> Многие из них были написаны раньше чем появилось ядро Linux. Цель проекта GNU "создание полноценной операционной системы наподобие UNIX оснащенной бесплатным программным обеспечением" (цитата с Web-узла [www.gnu.org](http://www.gnu.org)).

Ядро Linux и GNU программы составляют очень мощную комбинацию которую чаще всего называют просто "Linux". Но без GNU-программ система не будет работать, как и без ядра. Поэтому во многих случаях мы говорим *GNU/Linux*.

Исходные тексты программ, приведенные в этой книге, распространяются на условиях общей лицензии GNU (GPL, GNU General Public License), которая приведена в приложении Е, "Общая лицензия GNU". Таким же способом лицензируется большинство бесплатных программ, особенно в рамках GNU/Linux, например ядро системы. Прежде чем использовать представленные исходные тексты, ознакомьтесь с условиями данной лицензии.

Общая лицензия GNU обсуждается на Web-узле [www.gnu.org/copyleft](http://www.gnu.org/copyleft) наряду с другими лицензиями на бесплатное распространение программного обеспечения. Найти информацию о лицензиях на распространение программ с открытым кодом можно по адресу <http://www.opensource.org/licenses/index.html>.

### Для кого предназначена эта книга

Эта книга предназначена для трех категорий читателей.

■ Возможно, наш читатель является разработчиком, имеющим опыт создания программ для GNU/Linux и стремящимся узнать о более сложных возможностях и особенностях системы. Таких читателей заинтересуют главы, посвященные программированию процессов и потоков, а также межзадачному программированию и взаимодействию с аппаратными устройствами. Мы поможем им сделать свои программы быстрее, надежнее и безопаснее.

■ Возможно, наш читатель является разработчиком, имеющим опыт программирования для другой UNIX-системы и желающим создавать программы для GNU/Linux. Такой читатель уже знаком со стандартными API-функциями, и ему нужно узнать об особенностях системы, ее ограничениях, дополнительных возможностях и специфических соглашениях.

■ Возможно, наш читатель является разработчиком, пришедшим в среду UNIX из другой платформы, например Win32. Такой читатель знаком с общими принципами разработки программного обеспечения, но ему нужно узнать о специфических методиках, применяемых в Linux-программах для взаимодействия с операционной системой и другими программами. Ему нужно научиться писать программы, которые ведут себя так, как того ожидают пользователи Linux.

Эта книга не является исчерпывающим руководством или справочником по программированию в GNU/Linux. Мы применяем обучающий подход, последовательно излагая самые важные концепции и методики и приводя примеры их использования. В разделе 1.5, "Поиск дополнительной информации", указано, где можно найти дополнительную информацию по данной теме.

Поскольку в книге рассматриваются довольно сложные вопросы, мы предполагаем, что читатели знакомы с языком программирования C и знают, как использовать функции стандартной библиотеки языка C. Этот язык является основным средством разработки программного обеспечения для GNU/Linux. Большинство команд и функций, описанных в книге, а также большая часть ядра Linux написаны на C.

Изложенная в книге информация и равной степени применима к программам написанным на C++, так как этот язык является надмножеством языка C. Библиотечные функции языка C являются основным "средством общения" в среде GNU/Linux.

Те, кто уже программировали в UNIX, возможно, сталкивались с низкоуровневыми функциями ввода-вывода (`open()`, `read()`, `stat()` и т.д.). Они отличаются от стандартных



библиотечных функций языка C (`fopen()`, `fprintf()`, `fscanf()` и др.). Оба семейства функций находят применение в GNU/Linux, поэтому мы не будем делать акцент на каком-то одном семействе. Низкоуровневые функции описаны в приложении Б, "Низкоуровневый ввод-вывод".

В книге отсутствует вводная информация об операционной системе Linux. Мы предполагаем, что читатели имеют общее представление о том, как взаимодействовать с системой и выполнять базовые операции в графической среде и в режиме командной строки.

### *Соглашения, принятые в книге*

В книге используются следующие типографские соглашения.

- Новые термины выделяются *курсивом*.

- Тексты программ, названия функций, переменных и других элементов "компьютерного языка" выделены моноширинным шрифтом, например `printf("Hello, world!\n")`.

- Имена команд, файлов и каталогов также даны моноширинным шрифтом, например `cd /`.

- Когда мы показываем взаимодействие пользователя с интерпретатором команд, то ставим в начале строки приглашения символ `%` (в реальной системе вместо него может стоять другое выражение). Все, что находится далее в этой строке, вводится пользователем. Остальные строки это реакция системы. Например, в диалоге

```
% uname
Linux
```

система выдала приглашение `%`, пользователь ввел команду `uname`, а система ответила выдачей строки `Linux`.

- В заголовках к примерам программ указывается имя исходного файла (в скобках). Все листинги можно загрузить по адресу <http://www.advancedlinuxprogramming.com>.

Мы писали программы в Red Hat Linux версии 6.2. В этот дистрибутив входит ядро Linux версии 2.2.14, GNU-библиотека языка C версии 2.1.3 и семейство компиляторов EGCS версии 1.1.2. Приведенные программы в общем случае должны работать и в других версиях Linux, в частности в ядре версии 2.4 и с GNU-библиотекой языка C версии 2.2.

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше в что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать электронное письмо или просто посетить наш Web-сервер, оставив свои замечания, одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более подходящими для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш e-mail. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

E-mail: [info@williamspublishing.com](mailto:info@williamspublishing.com)

WWW <http://www.williamspublishing.com>

**Часть I**

**Сложные вопросы программирования в среде  
UNIX/Linux**

# Глава 1

## Начало

В этой главе рассказывается о том, как выполнять базовые операции, связанные с написанием программы на языке C или C++ в среде Linux. В частности, описываются процессы создания и модифицирования исходного текста на C/C++, компиляции этого текста и отладки полученного результата. Те, кто уже знакомы с программированием в Linux, могут смело переходить к главе 2, "Написание качественных программ для среды GNU/Linux".

При изложении материала всей книги мы предполагаем, что читатели знакомы с языком программирования C и C++ и наиболее распространенными функциями стандартной библиотеки языка C. Тексты программ в книге написаны на C, за исключением случаев, когда рассматривается конкретная особенность программирования на C++. Мы также предполагаем, что читатели умеют выполнять базовые операции в интерпретаторе команд Linux, в частности создавать каталоги и копировать файлы. В связи с тем что многие Linux-программисты начинали свой путь в среде Windows, мы иногда будем указывать на сходства и различия между двумя операционными системами.

### 1.1. Редактор Emacs

*Редактор* это программа, используемая для модификации исходных текстов. В Linux множество редакторов, но, очевидно, наиболее популярный и многофункциональный среди них GNU Emacs.

#### *Несколько слов о Emacs*

Emacs нечто гораздо большее, чем просто редактор. Это необычайно мощная программа, работая в которой можно, к примеру, читать и отправлять электронные сообщения. Способы настройки и расширения ее функциональных возможностей столь обширны, что заслуживают написания отдельной книги. Представьте, что, находясь в Emacs, вы можете путешествовать в Internet!

Впрочем, мы не ограничиваем свободу читателей, привыкших работать в другом редакторе. Ни один из примеров книги не зависит от использования Emacs. Представленный ниже небольшой вводный курс предназначен для тех из вас, кто еще не успел выбрать свой любимый редактор в Linux.

#### 1.1.1. Открытие исходного файла C/C++

Чтобы запустить редактор Emacs, наберите `emacs` в окне терминала и нажмите <Enter>. Появится окно редактора, в верхней части которого имеется строка меню. Перейдите в меню **Files**, выберите команду **Open Files** и наберите имя требуемого файла в строке "мини-буфера" в нижней части экрана.<sup>[2]</sup> При создании исходного файла на языке C используйте расширения `.c` или `.h`. В случае C++ придерживайтесь расширений `.cpp`, `.hpp`, `.c` или `.h`. Когда файл будет открыт, введите нужный текст и сохраните результат, выбрав команду **Save Buffer** в меню **Files**.

Чтобы выйти из редактора, воспользуйтесь командой **Exit Emacs** в меню **Files**.

Те, кто испытывают раздражение от необходимости постоянно щелкать мышью, могут воспользоваться клавиатурными сокращениями, ускоряющими открытие и сохранение файлов, а также выход из редактора. Операции открытия файла соответствует сокращение `C-x C-f`. (Запись `C-x` означает нажатие клавиши `<Control>` с последующим нажатием клавиши `<x>`.) Чтобы сохранить файл, введите `C-x C-s`, а чтобы выйти из Emacs `C-x C-c`. Лучше узнать о возможностях редактора можно с помощью встроенного учебника, доступного через команду **Emacs Tutorial** в меню **Help**. В нем приведено множество советов, которые помогут пользователям научиться эффективнее работать с Emacs.

### 1.1.2. Автоматическое форматирование

Программисты, привыкшие работать в интегрированной среде разработки, оценят имеющиеся в Emacs средства автоматического форматирования кода. При открытии исходного файла, написанного на C/C++, редактор самостоятельно определяет наличие в нем программного кода, а не просто текста. Если нажать клавишу `<Tab>` в пустой строке, редактор переместит курсор в нужную позицию, определяемую положением предыдущей строки. Когда клавиша `<Tab>` нажимается в строке, содержащей какой-то текст, сдвигается вся строка. Предположим, к примеру, что набран такой текст:

```
int main() {  
    printf("Hello, world\n");  
}
```

Нажатие клавиши `<Tab>` в строке вызова функции `printf()` приведет к следующему результату:

```
int main() {  
    printf("Hello, world\n");  
}
```

По мере работы с редактором Emacs читатели изучат и другие средства форматирования. Особенность редактора заключается в том, что он позволяет программировать практически любые операции, связанные с автоматическим форматированием. Благодаря этому были реализованы режимы редактирования множества видов документов, разработаны игры<sup>[3]</sup> и даже СУБД.

### 1.1.3. Синтаксические выделения

Помимо форматирования программного кода Emacs упрощает чтение файлов, написанных на C/C++, выделяя цветом различные синтаксические элементы. Например, ключевые слова могут быть выделены одним цветом, названия встроенных типов данных другим, а комментарии третьим. Подобный подход облегчает нахождение некоторых широко распространенных синтаксических ошибок.

Чтобы включить режим цветовых выделений, откройте файл `~/.emacs` и вставьте в него такую строку:

```
(global-font-lock-mode t)
```

Сохраните файл, выйдите из Emacs и перезапустите редактор. Теперь можете открыть нужный исходный файл и наслаждаться!

Внимательные читатели, возможно, обратили внимание на то, что строка, вставленная в файл `.emacs`, выглядит написанной на языке LISP. Это и есть LISP! Большая часть редактора Emacs реализована именно на этом языке. На нем же можно писать расширения к редактору.

## 1.2. Компиляторы GCC

Компилятор превращает исходный текст программы, понятный человеку, в объектный код, исполняемый компьютером. Компиляторы, доступные в Linux-системах, являются честью коллекции GNU-компиляторов, известной как GCC (GNU Compiler Collection).<sup>[4]</sup> В нее входят компиляторы языков C, C++, Java, Objective-C, Fortran и Chill. В этой книге нас будут интересовать лишь первые два.

Предположим, имеется проект, в который входят два исходных файла: один написан на C (main.c; листинг 1.1), а другой на C++ (reciprocal.cpp; листинг 1.2). После компиляции оба файла компонуются вместе, образуя программу reciprocal,<sup>[5]</sup> которая вычисляет обратное заданного целого числа.

### *Листинг 1.1. (main.c) Исходный файл на языке C*

```
#include <stdio.h>
#include "reciprocal.hpp"

int main(int argc, char **argv) {
    int i;
    i = atoi(argv[1]);
    printf("The reciprocal of %d is %g\n", i, reciprocal(i));
    return 0;
}
```

### *Листинг 1.2. (reciprocal.cpp) Исходный файл на языке C++*

```
#include <cassert>
#include "reciprocal.hpp"

double reciprocal (int i) {
    // Аргумент не должен быть равен нулю
    assert(i != 0);
    return 1.0/i;
}
```

Есть также файл заголовков, который называется reciprocal.hpp (листинг 1.3).

### *Листинг 1.3. (reciprocal.hpp) Файл заголовков*

```
#ifdef __cplusplus
extern "C" {
#endif

extern double reciprocal(int i);

#ifdef __cplusplus
}
#endif
```

Первый шаг заключается в превращении исходных файлов в объектный код.

### 1.2.1. Компиляция одного исходного файла

Компилятор языка C называется `gcc`. При компиляции исходного файла нужно указывать опцию `-c`. Вот как, например, в режиме командной строки компилируется файл `main.c`:

```
% gcc -c main.c
```

Полученный объектный файл будет называться `main.o`.

Компилятор языка C++ называется `g++`. Он работает почти так же, как и `gcc`. Следующая команда предназначена для компиляции файла `reciprocal.cpp`:

```
% g++ -c reciprocal.cpp
```

Опция `-c` говорит компилятору о необходимости получить на выходе объектный файл (он будет называться `reciprocal.o`). Без неё компилятор `g++` попытается скомпоновать программу и создать исполняемый файл.

В процессе написания любой более-менее крупной программы обычно задействуется ряд дополнительных опций. К примеру, опция `-I` сообщает компилятору о том, где искать файлы заголовков. По умолчанию компиляторы GCC просматривают текущий каталог, а также каталоги, где установлены файлы стандартных библиотек. Предположим, наш проект состоит из двух каталогов: `src` и `include`. Следующая команда даст компилятору `g++` указание дополнительно искать файл `reciprocal.hpp` в каталоге `../include`:

```
% g++ -c -I ../include reciprocal.cpp
```

Иногда требуется задать макроконстанты в командной строке. Например, в коммерческой версии программы нет необходимости осуществлять избыточную проверку утверждения в файле `reciprocal.cpp`; она нужна лишь в целях отладки. Эта проверка отключается путем определения макроконстанты `NDEBUG`. Можно, конечно, явно добавить в файл директиву `#define`, но это означает изменение исходного текста программы. Проще сделать то же самое в командной строке:

```
% g++ -c -D NDEBUG reciprocal.cpp
```

Аналогичным образом можно задать конкретный уровень отладки:

```
% g++ -c -D NDEBUG=3 reciprocal.cpp
```

При написании коммерческих программ оказываются полезными средства оптимизации кода, имеющиеся в компиляторах GCC. Есть несколько уровней оптимизации; для большинства программ подходит второй. Следующая команда компилирует файл `reciprocal.cpp` с включенным режимом оптимизации второго уровня:

```
% g++ -c -O2 reciprocal.cpp
```

Учтите, что средства оптимизации усложняют отладку программы. Кроме того, бывают случаи, когда наличие оптимизации приводит к проявлению скрытых ошибок, незаметных ранее.

Компиляторы `gcc` и `g++` принимают множество различных опций. Получить их полный список можно в интерактивной документации. Для этого введите следующую команду:

```
% info gcc
```

### 1.2.2. Компоновка объектных файлов

После того как файлы `main.c` и `reciprocal.cpp` скомпилированы, необходимо скомпоновать их. Если в проект входит хотя бы один файл C++, компоновка всегда осуществляется с помощью компилятора `g++`. Если же все файлы написаны на языке C, нужно использовать компилятор `gcc`. В нашем случае имеются файлы обоих типов, поэтому требуемая команда выглядит так:

```
% g++ -o reciprocal main.o reciprocal.o
```

Опция `-o` задает имя файла, создаваемого в процессе компоновки. Теперь можно осуществить запуск программы `reciprocal`:

```
% ./reciprocal 7
The reciprocal of 7 is 0.142857
```

Как видите, компилятор `g++` автоматически подключил к проекту стандартную библиотеку языка `C`, содержащую реализацию функции `printf()`. Для компоновки дополнительных библиотек (например, модуля функций графического интерфейса пользователя) необходимо воспользоваться опцией `-l`. В `Linux` имена библиотек почти всегда начинаются с префикса `lib`. Например, файл подключаемого модуля аутентификации (`Pluggable Authentication Module`, `PAM`) называется `libpam.a`. Чтобы скомпоновать его с имеющимися файлами, введите такую команду:

```
% g++ -o reciprocal main.o reciprocal.o -lpam
```

Компилятор автоматически добавит к имени библиотеки префикс `lib` и суффикс `.a`.

Как и в случае с файлами заголовков, компилятор ищет библиотечные файлы в стандартных каталогах, в частности `/lib` и `/usr/lib`. Для задания дополнительных каталогов предназначена опция `-L`, которая аналогична рассматривавшейся выше опции `-I`. Следующая команда сообщает компоновщику о том, что поиск библиотечных файлов нужно осуществлять прежде всего в каталоге `/usr/local/lib/pam`:

```
% g++ -o reciprocal main.o reciprocal.o -L/usr/local/lib/pam -lpam
```

В отличие от препроцессора, автоматически ведущего поиск файлов заголовков в текущем каталоге, компоновщик просматривает лишь стандартные каталоги. Поэтому, если библиотечный файл находится в текущем каталоге, об этом нужно сообщить явно с помощью опции `-L`. Например, после выполнения следующей команды компоновщик будет искать в текущем каталоге библиотеку `test`:

```
% gcc -o app app.o -L. -ltest
```

### 1.3. Автоматизация процесса с помощью GNU-утилиты `make`

Те, кто программируют в `Windows`, привыкли работать в той или иной интегрированной среде разработки. Программист добавляет в нее исходные файлы, а среда автоматически создает проект. Аналогичные среды доступны и в `Linux`, но мы не будем рассматривать их. Вместо этого мы научим читателей работать с GNU-утилитой `make`, знакомой большинству `Linux`-программистов. Она позволяет автоматически перекомпилировать программу.

Основная идея утилиты `make` проста. Ей указываются *целевые модули*, участвующие в процессе построения исполняемого файла, и *правила*, по которым протекает этот процесс. Также задаются *зависимости*, определяющие, когда конкретный целевой модуль должен быть перестроен.

В нашем тестовом проекте `reciprocal` три очевидных целевых модуля: `reciprocal.o`, `main.o` и сама программа `reciprocal`. Правила нам уже известны: это рассмотренные выше командные строки. А вот над зависимостями нужно немного подумать. Ясно, что файл `reciprocal` зависит от файлов `reciprocal.o` и `main.o`, поскольку нельзя скомпоновать программу, не создав оба объектных файла. Последние должны перестраиваться при изменении соответствующих исходных файлов. Нельзя также забывать о файле `reciprocal.hpp`: он включается в оба исходных файла, поэтому его изменение тоже затрагивает объектные файлы.

Помимо очевидных целевых модулей должен также существовать модуль `clean`. Он предназначен для удаления всех сгенерированных объектных файлов и программ, чтобы можно было начать все сначала. Правило для данного модуля включает команду `rm`, удаляющую перечисленные файлы.

Чтобы передать всю эту информацию утилите `make`, необходимо создать файл `Makefile`.



Его содержимое будет таким:

```
reciprocal: main.o reciprocal.o  
g++ $(CFLAGS) -o reciprocal main.o reciprocal.o
```

```
main.o: main.c reciprocal.hpp  
gcc $(CFLAGS) -c main.c
```

```
reciprocal.o: reciprocal.cpp reciprocal.hpp  
g++ $(CFLAGS) -c reciprocal.cpp
```

```
clean:  
rm -f *.o reciprocal
```

Целевые модули перечислены слева. За именем модуля следует двоеточие и существующие зависимости. В следующей строке указано правило, по которому создается модуль (назначение записи `$(CFLAGS)` мы пока проигнорируем). Строка правила должна начинаться с символа табуляции, иначе утилита `make` проинтерпретирует ее неправильно.

Если удалить созданные нами выше объектные файлы и ввести

```
% make
```

будет получен следующий результат:

```
% make  
gcc -c main.c  
g++ -c reciprocal.cpp  
g++ -o reciprocal main.o reciprocal.o
```

Утилита `make` автоматически создала объектные файлы и скомпоновала их. Попробуйте теперь внести какое-нибудь простейшее изменение в файл `main.c` и снова запустить утилиту. Вот что произойдет:

```
% make  
gcc -c main.c  
g++ -o reciprocal main.o reciprocal.o
```

Как видите, утилита `make` повторно создала файл `main.o` и перекомпоновала программу, но не стала перекомпилировать файл `reciprocal.cpp`, так как в этом не было необходимости.

Запись `$(CFLAGS)` обозначает переменную утилиты `make`. Ее можно определить либо в файле `Makefile`, либо в командной строке. Утилита подставит на место переменной реальное значение во время выполнения правила. Вот как, например, можно осуществить перекомпиляцию с включённой оптимизацией:

```
% make clean  
rm -f *.o reciprocal  
% make CFLAGS=-O2  
gcc -O2 -c main.c  
g++ -O2 -c reciprocal.cpp  
g++ -O2 -o reciprocal main.o reciprocal.o
```

Обратите внимание на то, что вместо записи `$(CFLAGS)` в правилах появился флаг `-O2`.

В этом разделе мы рассмотрели лишь самые основные возможности утилиты `make`. Чтобы получить о ней более подробную информацию, обратитесь к интерактивной документации, введя такую команду:

```
% info make
```

В документации можно найти полезные сведения о том, как упростить управление файлом `Makefile`, уменьшить число необходимых правил и автоматически вычислять зависимости.

## 1.4. GNU-отладчик gdb

Отладчик это программа, с помощью которой можно узнать, почему написанная вами

программа ведет себя не так, как было задумано. Работать с отладчиком приходится очень часто. Большинство Linux-программистов имеет дело с GNU-отладчиком (GNU Debugger, GDB), который позволяет пошагово выполнять программу, создавать точки останова и проверять значения локальных переменных.

### 1.4.1. Компиляция с включением отладочной информации

Чтобы можно было воспользоваться GNU-отладчиком, необходимо скомпилировать программу с включением в нее отладочной информации. Этой цели служит опция `-g` компилятора. Если имеется описанный выше файл Makefile, достаточно задать переменную `CFLAGS` равной `-g` при запуске утилиты `make`:

```
% make CFLAGS=-g
gcc -g -c main.c
g++ -g -c reciprocal.cpp
g++ -g -o reciprocal main.o reciprocal.o
```

Встречая в командной строке флаг `-g`, компилятор включает дополнительную информацию в объектные и исполняемые файлы. Благодаря этой информации отладчик узнает, какие адреса соответствуют тем или иным строкам в том или ином исходном файле, как отобразить значение локальной переменной, и т.д.

### 1.4.2. Запуск отладчика

Отладчик `gdb` запускается следующим образом:

```
% gdb reciprocal
```

После запуска появится строка приглашения такого вида:

```
(gdb)
```

В первую очередь необходимо запустить программу под отладчиком. Для этого введите команду `run` и требуемые аргументы. Попробуем вызвать программу без аргументов:

```
(gdb) run
```

```
Starting program: reciprocal
```

```
Program received signal SIGSEGV, Segmentation fault.
__strtol_internal (nptr=0x0, endptr=0x0, base=10, group=0)
at strtol.c:287
287 strtol.c: No such file or directory.
(gdb)
```

Проблема заключается в том, что в функции `main()` не предусмотрены средства контроля ошибок. Программа ожидает наличия аргумента, а в данном случае его нет. Получение сигнала `SIGSEGV` означает крах программы. Отладчик определяет, что причина краха находится в функции `__strtol_internal()`. Эта функция является частью стандартной библиотеки, но ее исходный файл отсутствует. Вот почему появляется сообщение "No such file or directory". С помощью команды `where` можно просмотреть содержимое стека:

```
(gdb) where
#0 __strtol_internal (nptr=0x0, endptr=0x0, base=10, group=0)
at strtol.c:287
#1 0x40096fb6 in atoi (nptr=0x0) at ../stdlib/stdlib.h:251
#2 0x804863e in main (argc=1, argv=0xbffff5e4) at main.c:8
```

Как нетрудно заметить, функция `main()` вызвала функцию `atoi()`, передав ей нулевой указатель, что и стало причиной ошибки.

С помощью команды `up` можно подняться по стеку на два уровня, дойдя до функции `main()`:

```
(gdb) up 2
#2 0x804863e in main (argc=1, argv=0xbffff5e4) at main.c:8
8 i = atoi(argv[1]);
```

Заметьте, что отладчик нашел исходный файл `main.c` и отобразил строку, где располагается ошибочный вызов функции. Узнать значение нужной локальной переменной позволяет команда `print`:

```
(gdb) print argv[1]
$2 = 0x0
```

Это подтверждает нашу догадку о том, что причина ошибки передача функции `atoi()` указателя `NULL`.

Установка контрольной точки осуществляется посредством команды `break`:

```
(gdb) break main
Breakpoint 1 at 0x804862e: file main.c, line 8.
```

В данном случае контрольная точка размещена в первой строке функции `main()`. Давайте теперь заново запустим программу, передав ей один аргумент:

```
(gdb) run 7
Starting program: reciprocal 7
```

```
Breakpoint 1, main (argc=2, argv=0xbffff5e4) at main.c:8
8i = atoi(argv[1]);
```

Как видите, отладчик остановился на контрольной точке- Перейти на следующую строку можно с помощью команды `next`:

```
(gdb) next
9 printf("The reciprocal of %d is %g\n", i,
reciprocal(i));
```

Если требуется узнать, что происходит внутри функции `reciprocal()`, воспользуйтесь командой `step`:

```
(gdb) step
reciprocal (i=7) at reciprocal.cpp:6
6 assert(i != 0);
```

Иногда удобнее запускать отладчик `gdb` непосредственно из редактора `Emacs`, а не из командной строки. Для этого следует ввести в редакторе команду `M-x gdb`. Когда отладчик останавливается в контрольной точке, редактор `Emacs` автоматически открывает соответствующий исходный файл. Не правда ли, проще разобраться в происходящем, глядя на весь файл, а не на одну его строку?

## 1.5. Поиск дополнительной информации

В каждый дистрибутив `Linux` входит масса полезной документации. В ней можно прочесть почти все из того, о чем говорится в этой книге (хотя это, очевидно, займет больше времени). Документация не всегда хорошо организована, поэтому поиск нужной информации требует определенной изобретательности. Иногда представленные факты оказываются устаревшими, так что не стоит всему слепо верить.

Ниже описаны наиболее полезные источники информации о программировании в `Linux`.

### 1.5.1. Интерактивная документация

В дистрибутивы `Linux` входят `man`-страницы с описанием большинства стандартных команд, системных вызовов и стандартных библиотечных функций. Интерактивная документация разбита на разделы, которым присвоены номера. Для программистов наиболее важными

являются следующие разделы:

- (1)пользовательские команды;
- (2)системные вызовы;
- (3)стандартные библиотечные функции;
- (8)системные/административные команды.

Числа обозначают номера разделов. Для доступа к страницам интерактивной документации применяется команда `man`. Она имеет вид `man имя`, где *имя* название команды или функции. Иногда одно и то же имя встречается в разных разделах. В этом случае номер раздела нужно указать явно, поставив его перед именем. К примеру, так вызывается страница с описанием команды `sleep` (находящаяся в первом разделе):

```
% man sleep
```

А следующая команда вызывает страницу с описанием библиотечной функции `sleep()`:

```
% man 3 sleep
```

Каждая `man`-страница содержит однострочное резюме команды или функции. Команда `whatis имя` отображает список всех `man`-страниц (во всех разделах), связанных с указанным именем. Если не известно точно, описание какой команды или функции требуется, можно выполнить поиск по ключевому слову в строках резюме с помощью команды `man -k ключевое_слово`.

Страницы интерактивной документации содержат множество полезной информации и являются первым источником, к которому следует обращаться за помощью. В случае команды `man`-страница описывает ее флаги и аргументы, входные и выходные значения, коды ошибок установки по умолчанию и т.п. В случае системного вызова или библиотечной функции описываются параметры и возвращаемые значения, коды ошибок и побочные эффекты, а также указывается, какие файлы заголовков нужно включать в программу при использовании функции.

### 1.5.2. Система Info

Система Info содержит гораздо более подробную документацию ко многим базовым компонентам GNU/Linux, а также к ряду других программ. Информационные страницы представляют собой гипертекстовые документы, напоминающие Web-страницы. Для запуска текстовой версии справочной системы Info достаточно ввести `info` в командной строке. Появится меню с описанием иерархии документов, установленных в системе. Нажав <Ctrl+N>, можно получить список клавиш, посредством которых осуществляется навигация по документам системы Info.

Среди наиболее полезных документов перечислим следующие:

- `gcc` описание компилятора `gcc`;
- `libc` описание GNU-библиотеки языка C, содержащей множество системных вызовов,
- `gdb` описание GNU-отладчика;
- `emacs` описание редактора Emacs;
- `info` описание самой системы Info.

Можно сразу вызвать нужную страницу, задав ее имя в командной строке:

```
% info libc
```

Те, кто в основном работают в Emacs, могут вызвать встроенный модуль просмотра документов Info, набрав `M-x info` или `C-h i`.

### 1.5.3. Файлы заголовков

Много информации о системных функциях можно почерпнуть из системных файлов заголовков. Они находятся в каталогах `/usr/include` и `/usr/include/sys`. Например, если компилятор сообщает об ошибке вызова системной функции, загляните в соответствующий файл заголовков и убедитесь, что реальный прототип функции соответствует описанному на ман-странице.

В Linux множество деталей функционирования системных вызовов отражено в файлах заголовков расположенных в каталогах `/usr/include/bits`, `/usr/include/asm` и `/usr/include/linux`. В частности, номера сигналов (механизм сигналов рассматривается в разделе 3.3, "Сигналы") определены в файле `/usr/include/bits/signal.h`.

#### **1.5.4. Исходные тексты**

Linux система с открытым кодом, не так ли? Верховным судьей, определяющим, как работает система, является исходный код самой системы. К нашему счастью, он доступен бесплатно. В имеющийся дистрибутив Linux могут входить исходные тексты всей системы и всех установленных в ней программ. (Правда, они не всегда записываются на жесткий диск. Инструкции по установке исходных текстов содержатся в документации дистрибутива.) Если это не так, у вас есть право запросить их на основании общей открытой GNU-лицензии.

Исходный код ядра Linux обычно хранится в каталоге `/usr/src/linux`. Это хороший источник информации о том, как работают процессы, виртуальная память и системные устройства. Большинство системных функций, упоминаемых в книге, реализовано в GNU-библиотеке языка C. Местоположение ее исходных текстов можно узнать в документации к дистрибутиву.

## Глава 2

# Написание качественных программ для среды GNU/Linux

В этой главе описываются базовые методики, применяемые большинством Linux-программистов. Придерживаясь данных методик, читатели смогут писать программы, которые не только хорошо работают в среде GNU/Linux, но и соответствуют представлениям пользователей о том, как должны работать такие программы.

## 2.1. Взаимодействие со средой выполнения

Те, кто изучали языки C и C++, знают, что специальная функция `main()` является главной точкой входа в программу. Когда операционная система запускает программу на выполнение, она автоматически предоставляет определенные средства, позволяющие программе взаимодействовать как с самой системой, так и с пользователем. Читатели наверняка знают о том, что у функции `main()` есть два параметра, `argc` и `argv`, через которые в программу передаются входные данные. Имеются также стандартные потоки `stdout` и `stdin` (или `cout` и `cin` в C++), реализующие консольный ввод-вывод. Все эти элементы существуют в языках C и C++, и работа с ними в среде GNU/Linux происходит строго определенным образом.

### 2.1.1. Список аргументов

Для запуска программы достаточно ввести ее имя в командной строке. Дополнительные информационные элементы, передаваемые программе, также задаются в командной строке и отделяются от имени программы и друг от друга пробелами. Такие элементы называются *аргументами командной строки*. (Аргумент, содержащий пробел, должен заключаться в кавычки.) Вообще-то, если быть более точным, правильнее говорить о *списке аргументов*, поскольку они не обязательно поступают из командной строки. В главе 3, "Процессы", рассказывается об ином способе вызова программы, при котором другая программа может передавать ей список аргументов напрямую.

Когда программа запускается из командной строки, список аргументов охватывает все содержимое строки, включая имя программы и любые присутствующие аргументы. Допустим, вызывается программа `ls`, отображающая содержимое корневого каталога и размеры соответствующих файлов:

```
% ls -s /
```

В данном случае список аргументов программы `ls` состоит из трех элементов. Первый это имя самой программы, указанное в командной строке, а именно `ls`. Вторым и третьим элементами аргументы командной строки `-s` и `/`.

Функция `main()` получает доступ к списку аргументов благодаря своим параметрам `argc` и `argv` (если они не используются, их можно не указывать). Параметр `argc` это целое число, равное количеству элементов в списке. Параметр `argv` это массив символьных указателей. Размер массива равен `argc`, а каждый элемент массива указывает на соответствующий элемент списка. Все аргументы представляются в виде строк, оканчивающихся нулевым символом.

Работа с аргументами командной строки сводится к просмотру параметров `argc` и `argv`. Если имя программы не должно учитываться, не забудьте пропустить первый элемент списка.

Использование параметров `argc` и `argv` демонстрируется в листинге 2.1.

```
#include <stdio.h>

int main (int argc, char* argv[]) {
    printf("The name of this program is \"%s*.\n", argv[0]);
    printf("This program was invoked with %d arguments.\n", argc - 1);

    /* Имеется ли хоть один аргумент? */
    if (argc > 1) {
        /* Да; отображаем содержимое. */
        int i;
        printf("The arguments are:\n");
        for (i = 1; i < argc; ++i)
            printf(" %s\n", argv[i]);
        }
    return 0;
}
```

### 2.1.2. Соглашения по работе с командной строкой в GNU/Linux

Практически все Linux-программы подчиняются соглашениям об интерпретации аргументов командной строки. Аргументы подразделяются на две категории: *опции* (или *флаги*) и все остальные. Опции меняют поведение программы, а остальные аргументы содержат разного рода входные данные (например, названия входных файлов).

Опции бывают двух видов.

■ *Короткие опции* состоят из дефиса и одиночного символа (обычно это буква в нижнем или верхнем регистре). Такие опции быстрее и проще набирать.

■ *Длинные опции* состоят из двух дефисов, после которых следует имя, содержащее буквы нижнего и верхнего регистров и дефисы. Такие опции легче запоминать и читать (например, в командных сценариях).

Обычно программа поддерживает обе разновидности каждой опции: первую для краткости, вторую для ясности. Например, большинство программ понимает опции `-h` и `--help` и трактует их одинаково. Как правило, опции указываются сразу после имени программы. После опций могут идти другие аргументы, в частности имена входных файлов и входные данные.

Некоторые опции предполагают наличие собственных аргументов. Так, рассмотренная выше команда `ls -s /` выводит содержимое корневого каталога. Опция `-s` сообщает программе `ls` о необходимости отображения размера (в килобайтах) каждого элемента каталога. Аргумент `/` задает имя каталога. Опция `--size` является синонимом опции `-s`, поэтому та же самая команда может быть задана так: `ls -- size /`.

В документе *GNU Coding Standards* перечислены имена некоторых наиболее часто используемых опций командной строки. При написании GNU-программ рекомендуется сверяться с этим документом. Пользователям удобно работать с программами, у которых много общих черт. Получить доступ к упомянутому документу в большинстве Linux-систем позволяет команда

```
% info "(standards)User Interfaces"
```

### 2.1.3. Функция getopt\_long()

Синтаксический анализ аргументов командной строки утомительная задача. К счастью. в GNU-библиотеке языка C есть функция `getopt_long()`, упрощающая ее решение. Эта функция понимает как короткие, так и длинные опции. Ее объявление находится в файле `<getopt.h>`.

Предположим, требуется написать программу, которая поддерживает три опции (табл. 2.1).

**Таблица 2.1. Опции тестовой программы**

Короткая форма	Длинная форма	Назначение
-h	--help	Отображение справки по использованию программы и выход
-o имя_файла	--output имя_файла	Задание имени выходного файла
-v	--verbose	Отображение развернутых сообщений

Кроме того, программе могут быть переданы дополнительные аргументы, задающие имена входных файлов

Функции `getopt_long()` нужно передать две структуры. Первая это строка с описанием возможных коротких опций (каждая из них представлена одной буквой). Если опция предполагает наличие аргумента, после нее ставится двоеточие. В нашем случае строка будет иметь вид `ho:v`. Это говорит о том, что программа поддерживает опции `-h`, `-o` и `-v`, причем для второй из них требуется аргумент.

Список возможных длинных опций задается в виде массива структур `option`. Каждый элемент массива соответствует одной опции и состоит из четырех полей. Чаще всего первое поле содержит имя опции (строка символов без ведущих дефисов), второе 1, если опция принимает аргумент, и 0 в противном случае: третье `NULL`; четвертое символьная константа, задающая короткий эквивалент данной длинной опции. Последний элемент массива должен содержать одни нули. Наш массив будет выглядеть так:

```
const struct option long_options[] = {
    { "help", 0, NULL, 'h' },
    { "output", 1, NULL, 'o' },
    { "verbose", 0, NULL, 'v' },
    { NULL, 0, NULL, 0 }
};
```

Функции `getopt_long()` передаются также параметры `argc` и `argv` функции `main()`. Ниже перечислены особенности ее работы.

■ При каждом вызове функция `getopt_long()` анализирует очередную опцию, возвращая букву, которая соответствует короткому эквиваленту опции. При отсутствии опций возвращается `-1`.

■ Обычно функция `getopt_long()` вызывается в цикле для обработки всех опций командной строки. Выбор конкретной опции осуществляется посредством конструкции `switch`.

■ Если опция `getopt_long()` обнаруживает неправильную опцию (т.е. она не указана в списке коротких и длинных опций), она выводит сообщение об ошибке и возвращает символ `?` (знак вопроса). В ответ на это большинство программ завершает свою работу, обычно отображая справку по работе с программой.

■ При обработке опции, имеющей аргумент, в глобальную переменную `optarg` помещается указатель на строку с содержимым аргумента.

■ Когда функция `getopt_long()` завершает анализ опций, в глобальную переменную `optind` записывается индекс того элемента массива `argv`, в котором содержится первый аргумент, не являющийся опцией.



В листинге 2.2 приведен пример обработки аргументов программы с помощью функции `getopt_long()`.

### *Листинг 2.2. (getopt\_long.c) Использование функции `getopt_long()`*

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

/* Имя программы. */
const char* program_name;

/* Вывод информации об использовании программы в поток STREAM
(обычно stdout или stderr) и завершение работы с выдачей кода
EXIT_CODE. Возврат в функцию main() не происходит */
void print_usage(FILE* stream, int exit_code) {
    fprintf(stream, "Usage: %s options [ inputfile ... ]\n",
        program_name);
    fprintf(stream,
        " -h --help Display this usage
information.\n"
        " -o --output filename Write output to file.\n"
        " -v --verbose Print verbose messages.\n");
    exit(exit_code);
}

/* Точка входа в основную программу, параметр ARGV содержит размер
списка аргументов; параметр ARGV -- это массив указателей
на аргументы. */
int main(int argc, char* argv[]) {
    int next_option;

    /* Строка с описанием возможных коротких опций. */
    const char* const short_options = "ho:v";
    /* Массив с описанием возможных длинных опций. */
    const struct option long_options[] = {
        { "help", 0, NULL, 'h' },
        { "output", 1, NULL, 'o' },
        { "verbose", 0, NULL, 'v' },
        { NULL, 0, NULL, 0 } /* Требуется в конце массива. */
    };

    /* Имя файла, в который записываются результаты работы
программы, или NULL, если вывод направляется в поток
stdout. */
    const char* output_filename = NULL;
    /* Следует ли выводить развернутые сообщения. */
    int verbose = 0;

    /* Запоминаем имя программы, которое будет включаться
в сообщения. Оно хранится в элементе argv[0] */
    program_name = argv[0];

    do {
        next_option =
```

```

getopt_long(argc, argv, short_options,
long_options, NULL);
switch(next_option) {
case 'h': /* -h или --help */
/* Пользователь запросил информацию об использовании
программы, нужно вывести ее в поток stdout и завершить
работу с выдачей кода 0 (нормальное завершение). */
print_usage(stdout, 0);
case 'o': /* -o или --output */
/* Эта опция принимает аргумент -- имя выходного файла. */
output_filename = optarg;
break;
case 'v': /* -v или --verbose */
verbose = 1;
break;
case '?': /* Пользователь ввел неверную опцию. */
/* Записываем информацию об использовании программы в поток
stderr и завершаем работу с выдачей кода 1
(аварийное завершение). */
print_usage(stderr, 1);
case -1: /* Опций больше нет. */
break;
default: /* Какой-то непредвиденный результат. */
abort();
}
}
while (next_option != -1);

/* Обработка опций завершена, переменная OPTIND указывает на
первый аргумент, не являющийся опцией. В демонстрационных
целях отображаем эти аргументы, если задан режим VERBOSE. */
if (verbose) {
int i;
for (i = optind; i < argc; ++i)
printf("Argument: %s\n", argv[i]);
}

/* Далее идет основное тело программы... */

return 0;
}

```

Может показаться, что использование функции `getopt_long()` приводит к написанию громоздкого кода, но, поверьте, самостоятельный синтаксический анализ опций командной строки гораздо более трудоемкая задача. Функция `getopt_long()` достаточно универсальна и гибка в работе с опциями, но лучше не заниматься сложными вещами. Старайтесь придерживаться традиционной структуры задания опций.

#### 2.1.4. Стандартный ввод-вывод

В стандартной библиотеке языка C определены готовые потоки ввода и вывода (`stdin` и `stdout` соответственно). Они используются функциями `scanf()`, `printf()` и целым рядом других библиотечных функций. Согласно идеологии UNIX, стандартные потоки можно перенаправлять. Это позволяет образовывать цепочки программ, связанных посредством каналов (конкретный синтаксис перенаправления потоков и работы с каналами можно узнать в документации к интерпретатору команд).

Есть также стандартный поток ошибок: `stderr`. Программы должны направлять предупреждающие сообщения и сообщения об ошибках в него, а не в поток `stdout`. Это позволяет отделять обычные выводные данные от разного рода служебных сообщений. К примеру, стандартный поток вывода можно направить в файл, а сообщения об ошибках, по-прежнему отображать на консоли. Запись в поток `stderr` осуществляется с помощью функции `fprintf()`:

```
fprintf(stderr, ("Error: ..."));
```

Все три стандартных потока доступны низкоуровневым функциям ввода-вывода (`read()`, `write()` и т.д.) через дескрипторы файлов. В частности, поток `stdin` имеет дескриптор 0, `stdout` 1, а `stderr` 2.

При вызове программы иногда требуется одновременно перенаправить потоки вывода и ошибок в файл или канал. Синтаксис подобной операции зависит от используемого интерпретатора команд. В интерпретаторах семейства Bourne shell (включая `bash`, который по умолчанию установлен в большинстве дистрибутивов Linux) это делается так:

```
% program > output_file.txt 2>&1
% program 2>&1 | filter
```

Запись `2>&1` означает, что файл с дескриптором 2 (`stderr`) объединяется с файлом, имеющим дескриптор 1 (`stdout`). Обратите внимание на то, что эта запись должна идти после операции перенаправления в файл (первый пример), но перед операцией перенаправления в канал (второй пример).

Поток `stdout` является буферизуемым. Записываемые в него данные не посылаются на консоль (или на другое устройство в случае перенаправления), пока буфер не заполнится, программа не завершит работу нормальным способом или файл `stdout` не будет закрыт. Осуществить принудительное "выталкивание" буфера позволяет функция `fflush()`:

```
fflush(stdout);
```

В то же время поток `stderr` не буферизуется. Записываемые в него данные сразу попадают на консоль. [\[6\]](#)

Указанная особенность потока `stdout` может приводить к неожиданным результатам. Например, в следующем цикле точка не выводится каждую секунду. Вместо этого все символы сначала помещаются в буфер, а затем целая их группа одновременно выводится на экран, когда буфер оказывается заполненным.

```
while (1) {
    printf(".");
    sleep(1);
}
```

А в этом цикле происходит то, что нам нужно:

```
while (1) {
    fprintf(stderr, ".");
    sleep(1);
}
```

### 2.1.5. Коды завершения программы

Когда программа завершает работу, она уведомляет операционную систему о своем состоянии, посылая ей код завершения, который представляет собой 16-разрядное целое число. По существующему соглашению нулевой код свидетельствует об успешном завершении, а ненулевой указывает на наличие ошибки. Некоторые программы возвращают различные ненулевые коды, обозначая разные ситуации.

В большинстве интерпретаторов команд код завершения последней выполненной

программы содержится в специальной переменной `$?`. В показанном ниже примере программа `ls` вызывается дважды, и оба раза запрашивается код ее завершения. В первом случае программа завершается корректно и возвращает нулевой код, во втором случае она сталкивается с ошибкой (указанный в командной строке файл не найден), поэтому код завершения оказывается ненулевым:

```
% ls /
bin coda etc lib misc nfs proc sbin usr
boot dev home lost+found mnt opt root tmp var
% echo $?
0
% ls bogusfile
ls: bogusfile: No such file or directory
% echo $?
1
```

Программа, написанная на языке C или C++, указывает код завершения в операторе `return` в функции `main()`. Есть и другие методы задания кодов завершения. Они обсуждаются в главе 3, "Процессы". Например, программе назначается определенный код, когда она завершается аварийно (вследствие получения сигнала).

### 2.1.6. Среда выполнения

Операционная система Linux предоставляет каждой запущенной программе *среду выполнения*. Под средой подразумевается совокупность пар переменная-значение. Имена переменных среды и их значения являются строками. По существующему соглашению переменные среды записываются прописными буквами.

Некоторые переменные должны быть знакомы большинству читателей, например:

- `USER` содержит имя текущего пользователя;

- `HOME` содержит путь к начальному каталогу текущего пользователя;

- `PATH` содержит разделенный двоеточиями список каталогов, которые операционная система просматривает в поиске вызванной программы;

- `DISPLAY` содержит имя и номер экрана сервера X Window, на котором отображаются окна графических программ.

Интерпретатор команд, как и любая другая программа, располагает своей средой. Имеются средства просмотра и редактирования переменных среды из командной строки. Например, программа `printenv` отображает текущую среду интерпретатора. В разных интерпретаторах есть свой встроенный синтаксис работы с переменными среды. Ниже демонстрируется синтаксис интерпретаторов семейства Bourne shell.

- Интерпретатор автоматически создает локальную переменную (называемую переменной интерпретатора) для каждой обнаруживаемой им переменной среды. Благодаря этому возможен доступ к переменным среды через выражения вида `$переменная`. Например:

```
% echo $USER
samuel
% echo $HOME
/home/samuel
```

- С помощью команды `export` можно экспортировать переменную интерпретатора в переменную среды. Вот как, например, задается значение переменной `EDITOR`:

```
% EDITOR=emacs
% export EDITOR
```

Или короче:

```
% export EDITOR=emacs
```

В программе доступ к переменным среды осуществляет функция `getenv()`, объявленная в файле `<stdlib.h>`. В качестве аргумента она принимает имя переменной и возвращает ее значение в строковом виде или `NULL`, если переменная не определена в данной среде. Для установки и сброса значений переменных среды предназначены функции `setenv()` и `unsetenv()` соответственно.

Получить список всех переменных среды немного сложнее. Для этого нужно обратиться к специальной глобальной переменной `environ`, определенной в GNU-библиотеке языка C. Данная переменная имеет тип `char**` и представляет собой массив указателей на символьные строки, последним элементом которого является `NULL`. Каждая строка имеет вид `ПЕРЕМЕННАЯ=значение`.

Программа, представленная в листинге 2.3, отображает всю свою среду, просматривая в цикле массив `environ`.

### ***Листинг 2.3. (print-env.c) Вывод переменных среды***

```
#include <stdio.h>

/* Массив ENVIRON содержит среду выполнения. */
extern char** environ;

int main() {
    char** var;
    for (var = environ; *var != NULL; ++var)
        printf("%s\n", *var);
    return 0;
}
```

Не пытайтесь модифицировать массив `environ` самостоятельно. Пользуйтесь для этих целей функциями `setenv()` и `unsetenv()`.

Обычно при запуске программа получает копию среды своей родительской программы (интерпретатора команд, если она была запущена пользователем). Таким образом, программы, запущенные из командной строки, могут исследовать среду интерпретатора команд.

Переменные среды чаще всего используют для передачи программам конфигурационной информации. Предположим, к примеру, что требуется написать программу, подключающуюся к серверу Internet. Имя сервера может задаваться в командной строке, но, если оно меняется нечасто, имеет смысл определить специальную переменную среды скажем, `SERVER_NAME`, которая будет хранить имя сервера. При отсутствии переменной программа берет имя, заданное по умолчанию. Интересующая нас часть программы показана в листинге 2.4.

### ***Листинг 2.4. (client.c) Часть сетевой клиентской программы***

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

int main() {
    char* server_name = getenv("SERVER_NAME");
    if (server_name == NULL)
        /* переменная среды SERVER_NAME не задана. Используем
        установки по умолчанию. */
        server_name = "server.my-company.com";
}
```

```
printf("accessing server %s\n", server_name);  
/* Здесь осуществляется доступ к серверу... */  
return 0;  
}
```

Допустим, программа называется `client`. Если переменная `SERVER_NAME` не задана, используется имя сервера, заданное по умолчанию:

```
% client  
accessing server server.my-company.com
```

Вот как задается другой сервер:

```
% export SERVER_NAME=backup-server.elsewhere.net  
% client  
accessing server backup-server.elsewhere.net
```

### 2.1.7. Временные файлы

Иногда программе требуется создать временный файл, например для промежуточного хранения большого объема данных или для передачи данных другой программе. В системах GNU/Linux временные файлы хранятся в каталоге `/tmp`. Работая с временными файлами, необходимо помнить о следующих ловушках.

- Одновременно может быть запущено несколько экземпляров программы (одним и тем же пользователем или разными пользователями). Все они должны использовать разные имена временных файлов, чтобы не было конфликтов.

- Права доступа к временным файлам должны задаваться таким образом, чтобы неавторизованные пользователи не могли влиять на работу программы путем модификации или замены временного файла.

- Имена временных файлов должны генерироваться так, чтобы посторонние пользователи не могли их предугадать. В противном случае хакер может воспользоваться задержкой между проверкой факта использования данного имени файла и открытием нового временного файла.

В Linux имеются функции `mkstemp()` и `tmpfile()`, решающие все вышеперечисленные проблемы. Выбор между ними делается на основании того, должен ли временный файл передаваться другой программе и какие функции ввода-вывода будут применяться при работе с файлом: низкоуровневые (`read()`, `write()` и т.д.) или потоковые (`fopen()`, `fprintf()` и т.д.).

#### Функция *mkstemp()*

Функция `mkstemp()` генерирует уникальное имя файла на основании переданного ей шаблона, создает временный файл с правами, разрешающими доступ к нему только для текущего пользователя, и открывает файл в режиме чтения/записи. Шаблон имени это строка, оканчивающаяся последовательностью "XXXXXX" (шесть прописных букв "X"). Функция `mkstemp()` заменяет каждую букву произвольным символом таким образом, чтобы получилось уникальное имя, и возвращает дескриптор файла. Запись в файл осуществляется с помощью функций семейства `write()`.

Временные файлы, создаваемые функцией `mkstemp()`, не удаляются автоматически. Ответственность за это возлагается на того, кто запускает программу. (Программисты должны внимательно следить за удалением временных файлов, иначе файловая система `/tmp` рано или поздно переполнится, приведя всю систему в нерабочее состояние.) Если файл создан для внутреннего использования и не предназначен для передачи другой программе, по окончании

работы с ним нужно сразу же вызвать функцию `unlink()`. Она удаляет из каталога ссылку на файл, но сам файл остается до тех пор, пока не будут закрыты все ссылающиеся на него дескрипторы. Таким образом, программа может продолжать использовать временный файл; он будет удален автоматически сразу после закрытия дескриптора. Операционная система закрывает дескрипторы файлов по окончании работы программы, так что временный файл будет удален даже в случае аварийного завершения программы.

В листинге 2.5 показаны две функции, работающие с временным файлом. Будучи примененными в связке, они позволяют легко переносить содержимое буферов из операторной памяти во временный файл (это дает возможность освободить и повторно использовать память), а затем загружать данные из файла обратно в память.

### ***Листинг 2.5. (temp\_file.c) Использование функции `mkstemp()`***

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

/* дескриптор временного файла, созданного в функции
write_temp_file(). */
typedef int temp_file_handle;

/* Запись указанного числа байтов из буфера во временный файл.
Ссылка на временный файл немедленно удаляется. Возвращается
дескриптор временного файла. */
temp_file_handle write_temp_file(char* buffer, size_t length) {
/* Создание имени файла и самого файла. Цепочка XXXXXX будет
заменена символами, которые сделают имя уникальным. */
char temp_filename() = "/tmp/temp_file.XXXXXX";
int fd = mkstemp(temp_filename);
/* немедленное удаление ссылки на файл, благодаря чему он будет
удален сразу же после закрытия дескриптора файла. */
unlink(temp_filename);
/* Сначала в файл записывается число, определяющее размер
буфера. */
write(fd, &length, sizeof(length));
/* теперь записываем сами данные. */
write(fd, buffer, length);
/* Возвращаем дескриптор файла. */
return fd;
}

/* Чтение содержимого временного файла, созданного в функции
write_temp_file(). Создается и возвращается буфер с содержимым
файла. Этот буфер должен быть удален в вызывающей подпрограмме
с помощью функции free(). В параметр LENGTH записывается размер
буфера в байтах. В конце временный файл удаляется. */
char* read_temp_file(temp_file_handle temp_file, size_t* length) {
char* buffer;
/* TEMP_FILE -- это дескриптор временного файла. */
int fd = temp_file;
/* переход в начало файла. */
lseek(fd, 0, SEEK_SET);
/* Определение объема данных, содержащихся во временном файле. */
read(fd, length, sizeof(*length));
/* Выделение буфера и чтение данных. */
```

```
buffer = (char*)malloc(*length);
read(fd, buffer, *length);
/* Закрытие дескриптора файла, что приведет к уничтожению
временного файла. */
close(fd);
return buffer;
}
```

## Функция *tmpfile()*

Если в программе используются функции потокового ввода-вывода библиотеки языка С и передавать временный файл другой программе не нужно, то для работы с временным файлом больше подойдет функция *tmpfile()*. Она создает и открывает временный файл, возвращая файловый указатель на него. Ссылка на файл уже оказывается удаленной, благодаря чему он уничтожается автоматически при закрытии указателя (с помощью функции *fclose()*) или при завершении программы.

В Linux есть ряд других функций, предназначенных для генерирования временных файлов или их имен, в частности *mktemp()*, *tmpnam()* и *tempnam()*. Работать с ними нежелательно, поскольку возникают упоминавшиеся выше проблемы, связанные с надежностью и безопасностью.

## 2.2. Защита от ошибок

Написать программу, которая корректно работает при "разумном" использовании, трудная задача. Написать программу, которая ведет себя "разумно" при возникновении ошибок, еще труднее. В этом разделе описываются методики программирования, позволяющие выявлять ошибки на ранних стадиях и решать проблемы, возникающие в ходе выполнения программы.

В представленные ниже фрагменты программ сознательно не включены громоздкие коды проверки ошибок и восстановления после них. так как это привело бы к потере наглядности при рассмотрении основных методик. Тем не менее мы вернемся к данной теме в главе 11, "Демонстрационное Linux-приложение", где будут приведены полностью работающие программы.

### 2.2.1. Макрос *assert()*

При написании программы следует помнить о том, что ошибки и непредвиденные ситуации могут радикально менять работу программы на самых ранних стадиях ее выполнения. Нужно стараться выявлять такие ошибки как можно раньше, на этапах разработки и отладки. Остальные ошибки, влияние которых на работу программы незначительно, обычно остаются незамеченными до тех пор, пока пользователи не начнут запускать программу.

Простейший способ выявления ненормальных ситуаций стандартный макрос *assert()* в языке С. Его аргументом является булево выражение. Программа завершается, если выражение оказывается ложным, при этом выводится сообщение об ошибке с указанием исходного файла и номера строки, а также текста выражения, приведшего к ошибке. Макрос *assert()* оказывается очень полезным для самых разных проверок целостности, выполняемых внутри программы. Например, с помощью этого макроса проверяют правильность аргументов функций, выполнение входных и выходных условий при вызове функций (а также методов С++) и наличие



непредвиденных возвращаемых значений.

Каждый вызов макроса `assert()` является не только проверкой, осуществляемой на этапе выполнения, но и документацией, описывающей работу программы непосредственно в ее исходном тексте. Когда программа содержит строку `assert (условие)`, то любой, кто читает исходный текст, будет знать, что в данной точке программы указанное условие всегда должно быть истинным. Если же условие не выполняется, то, очевидно, в программе присутствует ошибка.

В программах, критических к требованиям производительности, проверки `assert()` на этапе выполнения могут представлять собой слишком большую нагрузку. В таких случаях программа компилируется с установленной макроконстантой `NDEBUG`; для этого в командной строке компилятора указывается флаг `-DNDEBUG`. При наличии данной макроконстанты препроцессор убирает из тела программы все вызовы макроса `assert()`. И все же помните: делать это имеет смысл только тогда, когда производительность является узким местом программы, причем макрос нужно отключать лишь в наиболее критических файлах.

В связи с тем что макрос `assert()` может удаляться препроцессором из программы, необходимо тщательно проверить, не имеют ли выражения с макросом побочных эффектов. В частности, в этих выражениях не следует вызывать функции, присваивать значения переменным и пользоваться модифицирующими операторами наподобие `++`.

Предположим, к примеру, что в цикле вызывается функция `do_something()`. В случае успешного выполнения она возвращает 0, иначе ненулевое значение. Легкомысленный программист считает, что функция всегда завершается успешно, поэтому возникает соблазн написать так:

```
for (i = 0; i < 100; ++i)
    assert(do_something() == 0);
```

Позднее, забыв о данной особенности, программист решает, что проверки на этапе выполнения заметно снижают производительность программы и нужно перекомпилировать программу с включенной макроконстантой `NDEBUG`. В результате из программы будут удалены все макросы `assert()`, и функция `do_something()` вообще не будет вызвана. На самом деле необходимо использовать следующий подход:

```
for (i = 0; i < 100; ++i) {
    int status = do_something();
    assert(status == 0);
}
```

Еще один важный момент: макрос `assert()` не следует применять для проверки данных, вводимых пользователем. Пользователи не любят, когда программа аварийно завершает свою работу, выдавая малопонятное сообщение об ошибке, даже если причиной этого стали неправильно введенные данные. Конечно, входные данные всегда нужно проверять, но другими способами. Макрос `assert()` предназначен лишь для внутренних проверок.

Дадим несколько полезных советов.

- Проверяйте наличие пустых указателей, например в списке аргументов функции. Сообщение об ошибке, генерируемое строкой `{assert (pointer != NULL)}`,  
`Assertion 'pointer != ((void *)0)' failed.`

более информативно, чем сообщение, выдаваемое в ответ на попытку раскрытия пустого указателя:

```
Segmentation fault (core dumped)
```

- Проверяйте значения параметров функции. Например, если в функции предполагается, что параметр `foo` имеет только положительные значения, поставьте следующую проверку в самом начале тела функции:

```
assert(foo > 0);
```

Это поможет обнаружить случаи неправильного использования функции, а также даст понять любому, кто просматривает исходный текст программы, что функция накладывает ограничение на значение параметра.

### **2.2.2. Ошибки системных вызовов**

Большинство из нас училось писать программы, которые выполняются по четко намеченному алгоритму. Мы разделяли программу на задачи и подзадачи, и каждая функция решала свою задачу, вызывая другие функции для решения соответствующих подзадач. Мы ожидали, что, получив нужные входные данные, функция выдаст правильный результат с определенными побочными эффектами.

Реалии развития компьютерных систем разрушили этот идеал. Ресурсы компьютеров ограничены; иногда происходят аппаратные сбои; многие программы выполняются одновременно; пользователи и программисты делают ошибки. Часто все это проявляется на границе между приложением и операционной системой. Следовательно, используя системные вызовы для доступа к ресурсам, осуществления операций ввода-вывода или других целей, нужно понимать не только то, что именно происходит при успешном завершении вызова, но также при каких обстоятельствах он может завершиться неуспешно.

Сбои системных вызовов происходят в самых разных ситуациях.

■ В системе могут закончиться ресурсы (или же программа может исчерпать лимит ресурсов, наложенный на нее системой). Например, программа может запросить слишком много памяти, записать чересчур большой объем данных на диск или открыть чрезмерное количество файлов одновременно.

■ Операционная система Linux блокирует некоторые системные вызовы, когда программа пытается выполнить операцию при отсутствии должных привилегий. Например, программа может попытаться осуществить запись в доступный только для чтения файл, обратиться к памяти другого процесса или уничтожить программу другого пользователя.

■ Аргументы системного вызова могут оказаться неправильными либо по причине ошибочно введенных пользователем данных, либо из-за ошибки самой программы. Например, программа может передать системному вызову неправильный адрес памяти или неверный дескриптор файла. Другой вариант ошибки попытка открыть каталог вместо обычного файла или передать имя файла системному вызову, ожидающему имя каталога.

■ Системный вызов может аварийно завершиться по причинам, не зависящим от самой программы. Чаще всего это происходит при доступе к аппаратным устройствам. Устройство может работать некорректно или не поддерживать требуемую операцию, либо в дисковод просто не вставлен диск.

■ Выполнение системного вызова иногда прерывается внешними событиями, к каковым относится, например, получение сигнала. Это не обязательно означает ошибку, но ответственность за перезапуск системного вызова возлагается на программу.

В хорошо написанной программе, часто обращающейся к системным вызовам, большая часть кода посвящена обнаружению и обработке ошибок, а не решению основной задачи.

### **2.2.3. Коды ошибок системных вызовов**

Большинство системных вызовов возвращает 0, если операция выполнена успешно, и ненулевое значение в случае сбоя. (В некоторых случаях используются другие соглашения.

Например, функция `malloc()` при возникновении ошибки возвращает нулевой указатель. Никогда не помешает прочесть man-страницу, посвященную требуемому системному вызову.) Обычно этой информации достаточно для того, чтобы решить, следует ли продолжать привычное выполнение программы. Но для более специализированной обработки ошибок необходимы дополнительные сведения.

Практически все системные вызовы сохраняют в специальной переменной `errno` расширенную информацию о произошедшей ошибке.<sup>[7]</sup> В эту переменную записывается число, идентифицирующее возникшую ситуацию. Поскольку все системные вызовы работают с одной и той же переменной, необходимо сразу же после завершения функции скопировать значение переменной в другое место. Переменная `errno` модифицируется после каждого системного вызова.

Коды ошибок являются целыми числами. Возможные значения задаются макроконстантами препроцессора, которые, по существующему соглашению, записываются прописными буквами и начинаются с литеры "E", например `EACCESS` и `EINVAL`. При работе со значениями переменной `errno` следует всегда использовать макроконстанты, а не реальные числовые значения. Все эти константы определены в файле `<errno.h>`.

В Linux имеется удобная функция `strerror()`, возвращающая строковый эквивалент кода ошибки. Эти строки можно включать в сообщения об ошибках. Объявление функции находится в файле `<string.h>`.

Есть также функция `perror()` (объявлена в файле `<stdio.h>`), записывающая сообщение об ошибке непосредственно в поток `stderr`. Перед собственно сообщением следует размещать строковый префикс, содержащий имя функции или модуля, ставших причиной сбоя.

В следующем фрагменте программы делается попытка открыть файл. Если это не получается, выводится сообщение об ошибке и программа завершает свою работу. Обратите внимание на то, что в случае успеха операции функция `open()` возвращает дескриптор открытого файла, иначе -1.

```
fd = open("inputfile.txt", O_RDONLY);
if (fd == -1) {
    /* Открыть файл не удалось.
    Вывод сообщения об ошибке и выход. */
    fprintf(stderr, "error opening file: %s\n", strerror(errno));
    exit(1);
}
```

В зависимости от особенностей программы и используемого системного вызова конкретные действия, предпринимаемые в случае ошибки, могут быть разными: вывод сообщения об ошибке, отмена операции, аварийное завершение программы, повторная попытка и даже игнорирование ошибки. Тем не менее важно включить в программу код, обрабатывающий все возможные варианты ошибок.

Одни из кодов, с которым приходится сталкиваться наиболее часто, особенно в функциях ввода-вывода, это `EINTR`. Ряд функций, в частности `read()`, `select()` и `sleep()`, требует определенного времени на выполнение. Они называются блокирующими, так как выполнение программы приостанавливается до тех пор, пока функция не завершится. Но если программа, будучи заблокированной, принимает сигнал, функция завершается, не закончив выполнение операции. В данном случае в переменную `errno` записывается значение `EINTR`. Обычно в подобной ситуации следует повторно выполнить системный вызов.

Ниже приведен фрагмент программы, в котором функция `chown()` меняет владельца файла, определяемого переменной `path`, назначая вместо него пользователя с идентификатором `user_id`. Если функция завершается неуспешно, дальнейшие действия программы зависят от

значения переменной `errno`. Обратите внимание на интересный момент: при обнаружении возможной ошибки в самой программе ее выполнение завершается с помощью функции `abort()` или `assert()`, вследствие чего генерируется файл дампа оперативной памяти. Анализ этого файла может помочь выяснить природу таких ошибок. В случае невозстанавливаемых ошибок, например нехватки памяти, программа завершается с помощью функции `exit()`, указывая ненулевой код ошибки: в подобных ситуациях файл дампа оказывается бесполезным.

```
rval = chown(path, user_id, -1);
if (rval != 0) {
    /* Сохраняем переменную errno, поскольку она будет изменена
    при следующем системном вызове. */
    int error_code = errno;
    /* Операция прошла неуспешно; в случае ошибки функция chown()
    должна вернуть значение -1. */
    assert(rval == -1);
    /* Проверяем значение переменной errno и выполняем
    соответствующее действие. */
    switch (error_code) {
        case EPERM: /* Доступ запрещен. */
        case EROFS: /* Переменная PATH ссылается на файловую
        систему, доступную только для чтения. */
        case ENAMETOOLONG: /* Переменная PATH оказалась слишком
        длинной. */
        case ENOENT: /* Переменная PATH ссылается на
        несуществующий файл. */
        case ENOTDIR: /* Один из компонентов переменной PATH
        не является каталогом. */
        case EACCES: /* Один из компонентов переменной PATH
        недоступен. */
            /* Что-то неправильно с файлом, выводим сообщение
            об ошибке. */
            fprintf(stderr, "error changing ownership of %s: %s\n",
                path, strerror(error_code));
            /* Не завершаем программу; можно предоставить пользователю
            шанс открыть другой файл. */
            break;
        case EFAULT:
            /* Переменная PATH содержит неправильный адрес. Это, скорее
            всего, ошибка программы. */
            abort();
        case ENOMEM:
            /* Ядро столкнулось с нехваткой памяти. */
            fprintf(stderr, "%s\n", strerror(error_code));
            exit(1);
        default:
            /* Произошла какая-то другая, непредвиденная ошибка. Мы
            пытались обработать все возможные коды ошибок. Если
            что-то пропущено, то это ошибка программы! */
            abort();
    };
}
```

В самом начале программного фрагмента можно было поставить следующий код:

```
rval = chown(path, user_id, -1);
assert(rval == 0);
```

Но в таком случае, если функция завершится неуспешно, у нас не будет возможности обработать или исправить ошибку и даже просто сообщить о ней. Какую форму проверки использовать зависит от требований к обнаружению и последующему исправлению ошибок в программе.

## 2.2.4. Ошибки выделения ресурсов

Обычно при неудачном выполнении системного вызова наиболее приемлемое решение отменить текущую операцию, но не завершить программу, так как можно восстановить ее нормальную работу. Один из способов сделать это выйти из текущей функции, передав через оператор `return` код ошибки вызывающему модулю.

В случае, когда выход осуществляется посреди функции, важно убедиться в том, что ресурсы, выделенные в функции ранее, освобождены. К таким ресурсам относятся буферы памяти, дескрипторы и указатели файлов, временные файлы, объекты синхронизации и т.д. В противном случае, если программа продолжит выполняться, ресурсы окажутся потерянными.

В качестве примера рассмотрим функцию, загружающую содержимое файла в буфер. Функция выполняет такую последовательность действий:

- 1. выделяет буфер;
- 2. открывает файл;
- 3. читает содержимое файла и записывает его в буфер;
- 4. закрывает файл;
- 5. возвращает буфер вызывающему модулю.

Если файл не существует, этап 2 закончится неудачей. Подходящая реакция в этом случае вернуть из функции значение `NULL`. Но если буфер уже был выделен на этапе 1, существует опасность потери этого ресурса. Нужно не забыть освободить буфер где-то в программе. Если же неудачей завершится этап 3, требуется не только освободить буфер перед выходом из функции, но и закрыть файл.

В листинге 2.6 показан пример реализации такой функции.

**Листинг 2.6. (*readfile.c*) Освобождение ресурсов при возникновении аварийных ситуаций**

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

char* read_from_file(const char* filename, size_t length) {
    char* buffer;
    int fd;
    ssize_t bytes_read;
    /* Выделяем буфер. */
    buffer = (char*)malloc(length);
    if (buffer == NULL)
        return NULL;
    /* Открываем файл. */
    fd = open(filename, O_RDONLY);
    if (fd == 1) {
        /* Открыть файл не удалось. Освобождаем буфер
        перед выходом. */
        free(buffer);
        return NULL;
    }
    /* Чтение данных. */
    bytes_read = read(fd, buffer, length);
    if (bytes_read != length) {
```

```

/* Чтение не удалось. Освобождаем буфер и закрываем файл
перед выходом. */
free(buffer);
close(fd);
return NULL;
}
/* Все прошло успешно. Закрываем файл и возвращаем буфер
в программу. */
close(fd);
return buffer;
}

```

При завершении программы операционная система Linux освобождает выделенную память, ссылки на открытые файлы и большинство других ресурсов, поэтому перед вызовом функции `exit()` нет необходимости удалять буферы и закрывать файлы. Но некоторые другие совместно используемые ресурсы приходится все же освобождать вручную. В частности, это относится к временным файлам и совместным буферам памяти: они способны "пережить" программу.

## 2.3. Создание и использование библиотек

Практически со всеми программами компонуется одна или несколько библиотек. К любой программе, использующей функции языка C (например, `printf()` или `malloc()`), подключается библиотека времени выполнения. Если у программы есть графический интерфейс, вместе с ней компонуются библиотеки функций работы с окнами. Когда программа обращается к СУБД, она делает это посредством функции библиотеки, предоставленной разработчиком данной СУБД.

В каждом из перечисленных случаев необходимо решить, как компоновать библиотеку: *статически* или *динамически*. В первом случае программы станут громоздкими и их будет труднее обновлять, зато проще распространять. Во втором случае программы окажутся меньше и доступнее для изменений, но распространять придется большее число файлов. В данном разделе рассказывается о том, как осуществлять статическую и динамическую компоновку, на какие компромиссы при этом приходится идти и как решить, какой тип компоновки лучше всего подходит для конкретного случая.

### 2.3.1. Архивы

*Архив* (или статическая библиотека) это коллекция объектных файлов, хранящаяся в виде одного файла (он является примерным эквивалентом LIB-файла в Windows). Когда архив поступает на вход компоновщика, тот ищет в нем нужные объектные файлы, извлекает их и подключает к программе так, как если бы они были указаны непосредственно.

Архив создается посредством команды `ar`. Архивные файлы традиционно имеют расширение `.a`, а не `.o`, которое закреплено за отдельными объектными файлами. Вот как объединить файлы `test1.o` и `test2.o` в единый архив `libtest.a`:

```
% ar cr libtest.a test1.o test2.o
```

Флаги `cr` сообщают команде `ar` о необходимости создать архив. <sup>[8]</sup> Теперь можно подключать этот архив к программам с помощью флага `-ltest` компилятора `gcc` или `g++`, как описывалось в разделе 1.2.2, "Компоновка объектных файлов".

Обнаруживая в командной строке архив, компоновщик ищет в нем определения всех символических констант (функций или переменных), на которые дается ссылка в уже обработанных объектных файлах. Объектные файлы, содержащие определения этих констант, извлекаются из архива и включаются в исполняемый файл. В связи с тем что компоновщик

просматривает архив один раз, архивные файлы нужно указывать в конце командной строки. Предположим, например, что имеются два файла: `test.c` (листинг 2.7) и `app.c` (листинг 2.8).

### *Листинг 2.7. (test.c) Первый исходный файл*

```
int f() {  
    return 3;  
}
```

### *Листинг 2.8. (app.c) Второй исходный файл*

```
int main() {  
    return f();  
}
```

Теперь допустим, что файл `test.o` включен вместе с другими объектными файлами в архив `libtest.a`. Тогда следующая команда не будет работать:

```
% gcc -o app -L. -ltest app.o  
app.o: In function 'main':  
app.o(.text+0x4): undefined reference to 'f'  
collect2: ld returned 1 exit status
```

Как следует из сообщения об ошибке, несмотря на то что файл `libtest.a` содержит определение функции `f()`, компоновщик не нашел ее. Это объясняется тем, что компоновщик анализирует свои аргументы последовательно, слева направо, просматривая архив сразу же, как только он встречается в командной строке. На тот момент компоновщик еще не знал, что в дальнейшем ему встретится ссылка на функцию `f()`. Если сделать небольшую перестановку, все заработает:

```
% gcc -o app app.o -L. -ltest
```

Теперь наличие в файле `app.o` ссылки на функцию `f()` заставляет компоновщик включить в программу объектный файл `test.o` из архива `libtest.a`.

## **2.3.2. Совместно используемые библиотеки**

*Совместно используемая библиотека* (известная также как динамически подключаемая библиотека) напоминает архив тем, что она представляет собой группу объектных файлов. Но между ними есть ряд важных различий. Самое основное из них заключается в том, что, когда совместно используемая библиотека подключается к программе, в исполняемый файл не включается код самой библиотеки: в нем присутствует лишь ссылка на библиотеку. Если с несколькими программами компонуется одна и та же библиотека, все они будут ссылаться на нее, но ни в одну из них она не будет включена. Так расшифровывается термин "совместное использование".

Второе важное отличие состоит в том, что совместно используемая библиотека это не просто коллекция объектных файлов, из которых компоновщик выбирает требуемый для разрешения ссылки. В данном случае все объектные файлы, входящие в библиотеку, объединяются в единый объектный файл. Благодаря этому программы, компонуемые вместе с библиотекой, всегда имеют доступ ко всему ее содержимому, а не только к одной конкретной части.

Чтобы создать совместно используемую библиотеку, нужно сначала скомпилировать

составляющие ее объектные файлы с указанием опции -fPIC, например:

```
% gcc -c -fPIC test1.c
```

Опция -fPIC сообщает компилятору о том, что файл test1.o станет частью совместно используемой библиотеки.

### *Позиционно-независимый код*

Аббревиатура PIC (Position-Independent Code) в названии опции расшифровывается как "позиционно-независимый код". Функции в совместно используемой библиотеке могут загружаться по разным адресам разными программами, поэтому код библиотеки не должен зависеть от адреса (или позиции), по которому она загружена. Все это никак не касается программистов, просто нужно не забывать указывать флаг -fPIC при компиляции файлов, которые могут включаться в совместно используемую библиотеку.

Затем следует объединить объектные файлы в библиотеку:

```
% gcc -shared -fPIC -o libtest.so test1.o test2.o
```

Опция -shared заставляет компоновщик создать совместно используемую библиотеку, а не обычный исполняемый файл. Такие библиотеки имеют расширение .so. Подобно статическому архиву, имя библиотеки всегда начинается с префикса lib, указывающего на то, что файл является библиотекой.

Компоновка совместно используемой библиотеки аналогична компоновке архива. Например, следующая команда подключает к программе файл libtest.so, если он находится в текущем каталоге или одном из стандартных системных библиотечных каталогов:

```
% gcc -o app app.o -L. ltest
```

Предположим, имеются оба файла: libtest.a и libtest.so. Каким образом компоновщик принимает решение? Он просматривает каждый заданный каталог (сначала те, что указаны в опции -L, затем стандартные) и, как только обнаруживает хотя бы один из файлов, тут же прекращает поиск. Если в найденном каталоге присутствует только один из файлов, он и выбирается. В противном случае выбор делается в пользу совместно используемой библиотеки, если явно не указано обратное. Отдать приоритет статическому архиву позволяет опция -static. Например, следующая команда подключит к программе архив libtest.a, даже если присутствует библиотека libtest.so:

```
% gcc -static -o app app.o -L. -ltest
```

Команда ldd выводит список совместно используемых библиотек, подключенных к заданному исполняемому файлу. Все они должны быть доступны при запуске программы. Обратите внимание на то, что команда ldd сообщает о наличии дополнительной библиотеки: ld-linux.so. Она является частью механизма динамической компоновки в Linux.

### *Переменная LD\_LIBRARY\_PATH*

Когда к программе подключается совместно используемая библиотека, компоновщик помещает в исполняемый файл ссылку на нее, но в этой ссылке указан не полный путь к библиотеке, а только имя файла. При запуске программы система сама находит библиотеку и загружает ее. По умолчанию система просматривает лишь каталоги /lib и /usr/lib. Если библиотека находится в другом каталоге, она не будет найдена и система откажется загружать



программу.

Одно из решений заключается в компоновке программы с указанием флага `-wl, -rpath:`  
`% gcc -o app app.o -L. -ltest -wl, -rpath, /usr/local/lib`

Теперь в случае запуска программы `app` система будет искать требуемые библиотеки также в каталоге `/usr/local/lib`.

Но есть и другое решение: устанавливать переменную `LD_LIBRARY_PATH` при запуске программы. Подобно переменной среды `PATH`, переменная `LD_LIBRARY_PATH` представляет собой разделенный двоеточиями список каталогов. Если, к примеру, она равна `/usr/local/lib:/opt/lib`, то каталоги `/usr/local/lib` и `/opt/lib` будут просматриваться перед стандартными каталогами `/lib` и `/usr/lib`. Необходимо также учитывать, что при наличии данной переменной компоновщик будет просматривать заданные в ней каталоги, обнаруживая опцию `-L` в командной строке.<sup>[9]</sup>

### 2.3.3. Стандартные библиотеки

Даже если при компоновке программы не были заданы библиотеки, все равно одна из них почти наверняка присутствует. Дело в том, что компилятор `gcc` автоматически подключает к программе стандартную библиотеку языка C: `libc`. В нее, однако, не входят математические функции. Они находятся в отдельной библиотеке, `libm`, которую нужно компоновать явно. Например, чтобы скомпилировать и скомпоновать программу `compute`, использующую тригонометрические функции (такие как `sin()` и `cos()`), необходимо задать следующую команду:

```
% gcc -o compute compute.c -lm
```

При компоновке программ, написанных на C++, компилятор `c++` или `g++` автоматически подключает к ним стандартную библиотек языка C++: `libstdc++`.

### 2.3.4. Зависимости между библиотеками

Библиотеки часто связаны одна с другой. Например, во многих Linux-системах есть библиотека `libtiff`, содержащая функции чтения и записи графических файлов формата TIFF. Она, в свою очередь, использует библиотеки `libjpeg` (подпрограммы обработки JPEG-изображений) и `libz` (подпрограммы сжатия).

В листинге 2.9 показана небольшая программа, использующая функции библиотеки `libtiff` для работы с TIFF-файлом.

#### *Листинг 2.9. (tifftest.c) Применение библиотеки libtiff*

```
#include <stdio.h>
#include <tiffio.h>

int main(int argc, char** argv) {
    TIFF* tiff;
    tiff = TIFFOpen(argv[1], "r");
    TIFFClose(tiff);
    return 0;
}
```

При компиляции этого файла необходимо указать флаг `-ltiff`:

```
% gcc -o tiffptest tiffptest.c -ltiff
```

По умолчанию будет скомпонована совместно используемая версия библиотеки: `/usr/lib/libtiff.so`. В связи с тем что она обращается к библиотекам `libjpeg` и `libz` (одна совместно используемая библиотека может ссылаться на другие аналогичные библиотеки, от которых она зависит), будут также подключены их совместно используемые версии. Чтобы проверить это, воспользуемся командой `ldd`:

```
% ldd tiffptest
libtiff.so.3 => /usr/lib/libtiff.so.3 (0x4001d000)
libc.so.6 => /lib/libc.so.6 (0x40060000)
libjpeg.so.62 => /usr/lib/libjpeg.so.62 (0x40155000)
libz.so.1 => /usr/lib/libz.so.1 (0x40174000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

В противоположность этому статические библиотеки не могут указывать на другие библиотеки. Если попытаться подключить к программе статическую версию библиотеки `libtiff`, указав в командной строке опцию `-static`, компоновщик столкнется с нераспознаваемыми символическими константами:

```
% gcc -static -o tiffptest tiffptest.c -ltiff
/usr/bin/../lib/libtiff.a(tif_jpeg.o): In function
'tIFFjpeg_error_exit':
tif_jpeg.o(.text+0x2a): undefined reference to 'jpeg_abort'
/usr/bin/../lib/libtiff.a (tif_jpeg.o): In function
'tIFFjpeg_create_compress':
tif_jpeg.o(.text+0x8d): undefined reference to 'jpeg_std_error'
tif_jpeg.o(.text+0xcf): undefined reference to
'jpeg_CreateCompress'
...
```

В случае статической компоновки программы нужно самостоятельно указать две другие библиотеки:

```
% gcc -static -o tiffptest tiffptest.c -ltiff -ljpeg -lz
```

Иногда между двумя библиотеками образуется взаимная зависимость. Другими словами, первый архив ссылается на символические константы, определенные во втором архиве, и наоборот. Такая ситуация, как правило, является следствием неправильного проектирования. В этом случае нужно указать в командной строке одну и ту же библиотеку несколько раз. Компоновщик просмотрит библиотеку столько раз, сколько она присутствует в командной строке. Рассмотрим пример:

```
% gcc -o app app.o -lfoo -lbar -lfoo
```

Теперь, даже если библиотека `libfoo.a` ссылается на символические константы в библиотеке `libbar.a` и наоборот, программа будет успешно скомпонована.

### 2.3.5. Преимущества и недостатки библиотек

Познакомившись со статическими архивами и совместно используемыми библиотеками, читатели, очевидно, задумались: какие же из них лучше использовать? Есть несколько важных моментов, о которых следует помнить.

Большим преимуществом совместно используемой библиотеки является то, что она экономит место на диске при инсталляции программы. Когда устанавливаются десять программ и все они работают с одной и той же библиотекой, экономия может оказаться весьма существенной, тогда как статический архив будет включен во все десять программ. Уменьшается также время загрузки, если программа загружается из Internet.

С этим связано еще одно преимущество совместно используемых библиотек: пользователи могут обновлять библиотеки, не затрагивая связанные с ними программы. Предположим, к

примеру, что была создана библиотека, содержащая функции управления HTTP- соединениями. Потенциально с ней может работать множество программ. Если впоследствии в библиотеке обнаружится ошибка, достаточно будет просто заменить ее файл, и все программы, использующие данную библиотеку, немедленно обновятся. Не придется выполнять перекомпиловку всех программ, как в случае статического архива.

Описанные преимущества могут заставить читателей подумать, будто статические архивы бесполезны. Но их существование обусловлено вескими причинами. Тот факт, что обновление совместно используемой библиотеки отражается на всех связанных с нею программах, может на самом деле оказаться недостатком. Некоторые программы тесно связаны с используемыми библиотеками и не должны зависеть от произвольных изменений в системе.

Если библиотеки не должны устанавливаться в каталог `/lib` или `/usr/lib`, нужно дважды подумать, стоит ли их делать совместно используемыми. (Библиотеки нельзя помещать в указанные каталоги, если предполагается, что программу будут устанавливать пользователи, не имеющие привилегий системного администратора.) В частности, прием с флагом `-w1, -rpath` не будет работать, поскольку не известно, где именно окажутся библиотеки. А просить пользователей устанавливать переменную `LD_LIBRARY_PATH` не выход из положения, так как это означает для них выполнение дополнительного (для некоторых не самого тривиального) действия.

Оценивать преимущества и недостатки двух типов библиотек нужно для каждой создаваемой программы отдельно.

### 2.3.6. Динамическая загрузка и выгрузка

Иногда на этапе выполнения программы требуется загрузить некоторый код без явной компоновки. Рассмотрим приложение, поддерживающее подключаемые модули: Web-браузер. Архитектура браузера позволяет сторонним разработчикам создавать дополнительные модули, расширяющие функциональные возможности браузера. Модуль реализуется в виде совместно используемой библиотеки и размещается в заранее известном каталоге. Браузер автоматически загружает код из этого каталога.

Для этих целей в Linux существует специальная функция `dlopen()`. Например, открыть библиотеку `libtest.so` можно следующим образом:

```
dlopen("libtest.so", RTLD_LAZY)
```

Второй параметр это флаг, определяющий способ привязки символических констант в библиотеке. Данная установка подходит в большинстве случаев. Подробнее узнать о ней можно в документации.

Объявление функций работы с динамическими библиотеками находится в файле `<dlfcn.h>`. Используя их программы должны компоноваться с флагом `-ldl`, обеспечивающим подключение библиотеки `libdl`.

Функция `dlopen()` возвращает значение типа `void*`, используемое в качестве дескриптора динамической библиотеки. Это значение можно передавать функции `dlsym()`, которая возвращает адрес функции, загружаемой из библиотеки. Например, если в библиотеке `libtest.so` определена функция `my_function()`, то она вызывается следующим образом:

```
void* handle = dlopen("libtest.so", RTLD_LAZY);
void (*test)() = dlsym(handle, "my_function");
(*test)();
dlclose(handle);
```

С помощью функции `dlsym()` можно также получить указатель на статическую переменную, содержащуюся в совместно используемой библиотеке.

Обе функции, `dlopen()` и `dlsym()`, в случае неудачного завершения возвращают `NULL`. В данной ситуации можно вызвать функцию `dLError()` (без параметров), чтобы получить текстовое описание возникшей ошибки.

Функция `dldclose()` выгружает совместно используемую библиотеку. Строго говоря, функция `dlopen()` загружает библиотеку лишь в том случае, если она еще не находится в памяти. В противном случае просто увеличивается число ссылок на файл. Аналогичным образом функция `dldclose()` сначала уменьшает счетчик ссылок, и только если он становится равным нулю, выгружает библиотеку.

Когда совместно используемая библиотека пишется на C++, имеет смысл объявлять общедоступные функции со спецификатором `extern "C"`. Например, если функция `my_function()` написана на C++ и находится в совместно используемой библиотеке, а нужно обеспечить доступ к ней с помощью функции `dlsym()`, объявите ее следующим образом:

```
extern "C" void my_function();
```

Тем самым компилятору C++ будет запрещено подменять имя функции. При отсутствии спецификатора `extern "C"` компилятор подставит вместо имени `my_function` совершенно другое имя, в котором закодирована информация о данной функции. Компилятор языка C не заменяет имена; он работает с теми именами, которые назначены пользователем.

# Глава 3

## Процессы

Выполняющийся экземпляр программы называется *процессом*. Если на экране отображаются два терминальных окна, то, скорее всего, одна и та же терминальная программа запущена дважды ей просто соответствуют два процесса. В каждом окне, очевидно, работает интерпретатор команд это еще один процесс. Когда пользователь вводит команду в интерпретаторе, соответствующая ей программа запускается в виде процесса. По завершении работы программы управление вновь передается процессу интерпретатора.

Опытные программисты часто создают несколько взаимодействующих процессов в рамках одного приложения, чтобы оно могло выполнять группу действий одновременно. Это повышает надежность приложения и позволяет ему использовать уже написанные программы.

Большинство описанных в данной главе функций управления процессами доступно и в других UNIX-системах. В основном они объявлены в файле `<unistd.h>`, но не помешает проверить это в документации.

### 3.1. Знакомство с процессами

Пользователю достаточно войти в систему, чтобы в ней начали выполняться процессы. Даже если пользователь ничего не запускает, а просто сидит перед экраном и пьет кофе. в системе все равно "теплится жизнь". Любой выполняющейся программе соответствует один или несколько процессов. Давайте для начала познакомимся с теми из них, которые присутствуют по умолчанию.

#### 3.1.1. Идентификаторы процессов

Каждый процесс в Linux помечается уникальным идентификатором (PID, process identifier). Идентификаторы это 16-разрядные числа, назначаемые последовательно по мере создания процессов.

У всякого процесса имеется также родительский процесс (за исключением специального демона `init`, о котором рассказывается в разделе 3.4.3, "Процессы-зомби"). Таким образом, все процессы Linux организованы в виде древовидной иерархии, на вершине которой находится процесс `init`. К атрибутам процесса относится идентификатор его предка (PPID, parent process identifier).

Работая с идентификаторами процессов в программах, написанных на языках C и C++, следует объявлять соответствующие переменные как имеющие тип `pid_t` (определен в файле `<sys/types.h>`). Программа может узнать идентификатор своего собственного процесса с помощью системного вызова `getpid()`, а идентификатор своего родительского процесса с помощью вызова `getppid()`. В листинге 3.1 показано, как это сделать.

#### Листинг 3.1. (*print-pid.c*) Вывод идентификатора процесса

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

```
int main() {
printf("The process ID is %d\n", (int)getpid());
printf("The parent process ID is %d\n", (int)getppid());
return 0;
}
```

Обратите внимание на важную особенность: при каждом вызове программа сообщает о разных идентификаторах, поскольку всякий раз запускается новый процесс. Тем не менее, если программа вызывается из одного и того же интерпретатора команд, то родительский идентификатор оказывается одинаковым.

### 3.1.2. Получение списка активных процессов

Команда `ps` отображает список процессов, работающих в данный момент в системе. Версия этой команды в GNU/Linux имеет множество опций, так как пытается быть совместимой со своими "родственниками" в других UNIX-системах. С помощью опций можно указывать, о каких процессах и какую именно требуется получить информацию.

Будучи вызванной без аргументов, команда `ps` выводит список тех процессов, управляющим терминалом которых является ее собственный терминал:

```
% ps
PID TTY TIME CMD
21693 pts/8 00:00:00 bash
21694 pts/8 00:00:00 ps
```

В данном случае мы видим два процесса. Первый из них. `bash`, это интерпретатор команд, запущенный на данном терминале. Второй выполняющийся экземпляр самой команды `ps`. В самом левом столбце, `PID`, отображаются идентификаторы процессов.

Более полный список можно получить с помощью следующей команды:

```
% ps -e -o pid,ppid,command
```

Опция `-e` заставляет команду `ps` отображать все процессы, выполняющиеся в системе. Опция `-o pid,ppid,command` сообщает о том, какая информация о процессе нас интересует. В данном случае это идентификатор самого процесса, идентификатор его родительского процесса и команда, посредством которой был запущен процесс.

#### **Форматы вывода команды `ps`**

В опции `-o` через запятую указываются столбцы которые должны быть включены в вывод команды `ps`. Например, команда `ps -o pid,user,start_time,command` отображает идентификатор процесса, имя его владельца, время запуска а также команду, соответствующую процессу. Полный список опций и столбцов можно узнать на [man-странице](#) команды `ps`. Имеются три predefined формата вывода: `-f` (полный листинг), `-l` (длинный листинг) и `-j` (вывод заданий)

Ниже приведено несколько первых и последних строк, выдаваемых этой командой в нашей системе:

```
% ps -e -o pid,ppid,command
PID PPID COMMAND
1 0 init [5]
2 1 [kflushd]
3 1 [kupdate]
...
```

```
21725 21693 xterm
21727 21725 bash
21728 21727 ps -e -o pid,ppid,command
```

Заметьте: родительский идентификатор команды `ps`, 21727, соответствует интерпретатору `bash`, из которого была вызвана команда. В свою очередь, родительский идентификатор интерпретатора, 21725, принадлежит программе `xterm` эмулятору терминала, в котором выполняется интерпретатор.

### 3.1.3. Уничтожение процесса

Для уничтожения процесса предназначена команда `kill`. Ей достаточно указать идентификатор требуемого процесса.

Команда `kill` посылает процессу сигнал `SIGTERM`, являющийся запросом на завершение.<sup>[10]</sup> По умолчанию, если в программе отсутствует обработчик данного сигнала, процесс просто завершает свою работу. О сигналах речь пойдет в разделе 3.3, "Сигналы".

## 3.2. Создание процессов

Существуют два способа создания процессов. Первый из них относительно прост, но применяется редко, поскольку неэффективен и связан со значительным риском для безопасности системы. Второй способ сложнее, но избавлен от недостатков первого.

### 3.2.1. Функция `system()`

Функция `system()` определена в стандартной библиотеке языка C и позволяет вызывать из программы системную команду, как если бы она была набрана в командной строке. По сути, эта функция запускает стандартный интерпретатор Bourne shell (`/bin/sh`) и передает ему команду на выполнение. Например, программа, представленная в листинге 3.2, вызывает команду `ls -l /`, отображающую содержимое корневого каталога.

#### *Листинг 3.2. (system.c) Использование функции `system()`*

```
#include <stdlib.h>

int main() {
    int return_value;
    return_value = system("ls -l /");
    return return_value;
}
```

Функция `system()` возвращает код завершения указанной команды. Если интерпретатор не может быть запущен, возвращается значение 127, а в случае возникновения других ошибок -1.

Поскольку функция `system()` запускает интерпретатор команд, она подвержена всем тем ограничениям безопасности, что и системный интерпретатор. Рассчитывать на наличие какой-то конкретной версии Bourne shell не приходится. В большинстве UNIX-систем программа `/bin/sh` представляет собой символическую ссылку на другой интерпретатор. В Linux это `bash` (Bourne-Again SHell), причем в разных дистрибутивах присутствуют разные его версии. Вызов из функции `system()` программы с привилегиями пользователя `root` также может иметь

неодинаковые последствия в разных системах. Таким образом, лучше создавать процессы с помощью функций `fork()` и `exec()`.

### 3.2.2. Функции `fork()` и `exec()`

В DOS и Windows API имеется семейство функций `spawn()`. Они принимают в качестве аргумента имя программы, создают новый экземпляр ее процесса и запускают его. В Linux нет функции, которая делала бы все это за один заход. Вместо этого имеется функция `fork()`, создающая дочерний процесс, который является точной копией родительского процесса, и семейство функций `exec()`, заставляющих требуемый процесс перестать быть экземпляром одной программы и превратиться в экземпляр другой программы. Чтобы создать новый процесс, нужно сначала с помощью функции `fork()` создать копию текущего процесса, а затем с помощью функции `exec()` преобразовать одну из копий в экземпляр запускаемой программы.

#### *Вызов функции `fork()`*

Вызывая функцию `fork()`, программа создает свой дубликат, называемый *дочерним процессом*. Родительский процесс продолжает выполнять программу с той точки, где была вызвана функция `fork()`. То же самое делает и дочерний процесс.

Как же различить между собой оба процесса? Во-первых, дочерний процесс это новый, только что появившийся в системе процесс, поэтому его идентификатор отличается от идентификатора родительского процесса. Таким образом, программа может вызвать функцию `getpid()` и узнать, где именно она находится. Но сама функция `fork()` реализует другой способ: она возвращает разные значения в родительском и дочернем процессах. Родительский процесс получает идентификатор своего потомка, а дочернему процессу возвращается 0. В системе нет процессов с нулевым идентификатором, так что программа легко разбирается в ситуации.

В листинге 3.3 приведен пример ветвления программы с помощью функции `fork()`. Учтите, что первая часть инструкции `if` выполняется только в родительском процессе, тогда как ветвь `else` только в дочернем.

#### *Листинг 3.3. (fork.c) Ветвление программы с помощью функции `fork()`*

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

int main() {
    pid_t child_pid;
    printf("The main program process ID is %d\n",
        (int)getpid());

    child_pid = fork();
    if (child_pid != 0) {
        printf("This is the parent process, with ID %d\n",
            (int)getpid());
        printf("The child's process ID is %d\n", (int)child_pid);
    } else
        printf("This is the child process, with ID %d\n",
```



```
(int)getpid());  
return 0;  
}
```

## Семейство функций *exec()*

Функции семейства *exec()* заменяют программу, выполняющуюся в текущем процессе, другой программой. Когда программа вызывает функцию *exec()*, ее выполнение немедленно прекращается и начинает работу новая программа.

Функции, входящие в семейство *exec()*, немного отличаются друг от друга по своим возможностям и способу вызова.

■ Функции, в названии которых присутствует суффикс 'p' (*execvp()* и *execlp()*), принимают в качестве аргумента имя программы и ищут эту программу в каталогах, определяемых переменном среды *PATH*. Всем остальным функциям нужно передавать полное путевое имя программы.

■ Функции, в названии которых присутствует суффикс 'v' (*execv()*, *execvp()* и *execve()*), принимают список аргументов программы в виде массива строковых указателей, оканчивающегося *NULL*-указателем. Функции с суффиксом 'l' (*exec1()*, *execlp()* и *execle()*) принимают список аргументов переменного размера.

■ Функции, в названии которых присутствует суффикс 'e' (*execve()* и *execle()*), в качестве дополнительного аргумента принимают массив переменных среды. Этот массив содержит строковые указатели и оканчивается пустым указателем. Каждая строка должна иметь вид "ПЕРЕМЕННАЯ=значение".

Поскольку функция *exec()* заменяет одну программу другой, она никогда не возвращает значение только если вызов программы оказался невозможен в случае ошибки.

Список аргументов, передаваемых программе, аналогичен аргументам командной строки, указываемым при запуске программы в интерактивном режиме. Их тоже можно получить с помощью параметров *argc* и *argv* функции *main()*. Не забывайте, когда программу запускает интерпретатор команд, первый элемент массива *argv* будет содержать имя программы, а далее будут находиться переданные программе аргументы. Аналогичным образом следует поступить, формируя список аргументов для функции *exec()*.

## Совместное использование функций *fork()* и *exec()*

Стандартная методика запуска одной программы из другой такова: сначала с помощью функции *fork()* создается дочерний процесс, затем в нем вызывается функция *exec()*. Это позволяет главной программе продолжать выполнение в родительском процессе.

Программа, показанная в листинге 3.4, отображает содержимое корневого каталога с помощью команды *ls*, как и программа в листинге 3.2. Но на этот раз команда *ls* вызывается не из интерпретатора, а напрямую; ей передаются аргументы *-l* и */.*

### Листинг 3.4. (*fork-exec.c*) Совместное использование функций *fork()* и *exec()*

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

```

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

/* Запуск дочернего процесса в виде новой программы. Параметр
PROGRAM это имя вызываемой программы; ее поиск будет
осуществляться в каталогах, определяемых переменной среды PATH.
Параметр ARG_LIST -- это список строковых аргументов,
передаваемых программе (должен оканчиваться указателем NULL).
Функция возвращает идентификатор порожденного процесса. */
int spawn(char* program, char** arg_list) {
    pid_t child_pid;
    /* Создание копии текущего процесса. */
    child_pid = fork();
    if (child_pid != 0)
        /* Это родительский процесс. */
        return child_pid;
    else {
        /* Выполнение указанной программы. */
        execvp(program, arg_list);
        /* Функция execvp() возвращает значение только в случае
        ошибки. */
        fprintf(stderr, "an error occurred in execvp\n");
        abort();
    }
}

int main() {
    /* Список аргументов, передаваемых команде ls. */
    char* arg_list[] = {
        "ls", /* argv[0] -- имя программы. */
        "-l",
        NULL /* Список аргументов должен оканчиваться указателем
        NULL. */
    };
    /* Порождаем дочерний процесс, который выполняет команду ls.
    Игнорируем возвращаемый идентификатор дочернего процесса. */
    spawn("ls", arg_list);
    printf("done with main program\n");
    return 0;
}

```

### 3.2.3. Планирование процессов

Операционная система Linux планирует работу родительских и дочерних процессов независимо друг от друга. Нет гарантии, что один процесс будет запущен раньше другого, и неизвестно, как долго один процесс будет выполняться, прежде чем Linux прервет его работу и передаст управление другому процессу. В частности, к моменту завершения родительского процесса может оказаться, что команда `ls` еще не выполнена, выполнена частично или уже закончила свою работу. [\[11\]](#) Linux лишь гарантирует, что любой процесс когда-нибудь получит свой "кусочек пирога": ни один процесс не окажется полностью лишенным доступа к процессору.

Можно сообщить системе о том, что процесс не очень важен и должен выполняться с пониженным приоритетом. Это делается путем повышения *фактора уступчивости* процесса. По умолчанию у каждого процесса нулевой фактор уступчивости. Повышение этого значения свидетельствует о снижении приоритета процесса, и наоборот: процессы с низким (т.е.

отрицательным) фактором уступчивости получают больше времени на выполнение.

Для запуска программы с ненулевым фактором уступчивости необходимо воспользоваться командой `nice -n`. Рассмотрим следующий пример:

```
% nice -n 10 sort input.txt > output.txt
```

Здесь активизируется длительная операция сортировки, которая, благодаря пониженному приоритету, не приведет к сильному снижению производительности системы. Изменить фактор уступчивости выполняющегося процесса позволяет команда `renice`.

Если требуется менять фактор уступчивости программным путем, воспользуйтесь функцией `nice()`. Ее аргумент это величина приращения, добавляемая к фактору уступчивости вызывающего процесса. В результате приоритет процесса снижается.

Только программа с привилегиями пользователя `root` может запускать процессы с отрицательным фактором уступчивости или понижать это значение у выполняющегося процесса. Это означает, что вызывать команды `nice` и `renice` с отрицательными аргументами можно, лишь войдя в систему как пользователь `root`, и только процесс, выполняемый от имени суперпользователя, может передавать функции `nice()` отрицательное значение. Таким образом, обычные пользователи не могут помешать работать процессам других пользователей и монополизировать системные ресурсы.

### 3.3. Сигналы

Сигналы это механизм связи между процессами в Linux. Данная тема очень обширна, поэтому здесь мы рассмотрим лишь наиболее важные сигналы и методики управления процессами.

Сигнал представляет собой специальное сообщение, посылаемое процессу. Сигналы являются асинхронными: когда процесс принимает сигнал, он немедленно обрабатывает его, прерывая выполнение текущей функции и даже текущей строки программы. Есть несколько десятков различных сигналов, каждый из которых имеет свое функциональное назначение. Все они распознаются по номерам, но в программах для ссылки на сигналы пользуются символическими константами. В Linux эти константы определены в файле `/usr/include/bits/signum.h` (его не нужно включать в программы, для этого есть файл `<signal.h>`).

В ответ на полученный сигнал процесс выполняет ряд действий в зависимости от типа сигнала. У каждого сигнала есть стандартный *обработчик*, определяющий, что произойдет с процессом, если он попытается проигнорировать сигнал. Для большинства сигналов можно также задавать явную функцию обработки. В этом случае при поступлении сигнала выполнение программы приостанавливается, выполняется обработчик, а потом программа возобновляет свою работу.

Операционная система Linux посылает процессам сигналы в случае возникновения определенных ситуаций. Например, сигналы `SIGBUS` (ошибка на шине), `SIGSEGV` (нарушение сегментации) и `SIGFPE` (ошибка операции с плавающей запятой) могут быть посланы процессу, пытающемуся выполнить неправильную операцию. По умолчанию эти сигналы приводят к завершению процесса и созданию дампа оперативной памяти.

Процесс может сам послать сигнал другому процессу. Чаще всего возникает необходимость завершить требуемый процесс с помощью сигнала `SIGTERM` или `SIGKILL`.<sup>[12]</sup> С помощью сигналов можно также передавать команды выполняющимся программам. Для этого существуют "пользовательские" сигналы `SIGUSR1` и `SIGUSR2`. Иногда в аналогичных целях применяется сигнал `SIGURG`, с помощью которого можно заставить программу повторно прочитать свои файлы конфигурации.

Функция `sigaction()` определяет правила обработки указанного сигнала. Первый ее аргумент — это номер сигнала. Следующие два аргумента представляют собой указатели на структуру `sigaction`; первый из них регистрирует новый обработчик сигнала, а второй содержит описание предыдущего обработчика. Наиболее важным полем структуры `sigaction` является `sa_handler`. Оно может содержать одно из трех значений:

- `SIG_DFL` — выбор стандартного обработчика сигнала;

- `SIG_IGN` — игнорирование сигнала,

- указатель на функцию обработки сигнала; эта функция должна принимать один параметр (номер сигнала) и возвращать значение типа `void`.

Поскольку сигнал может прийти в любой момент, он способен заставить программу "врасплох" за выполнением критической операции, не подразумевающей прерывание. Такой операцией, к примеру, является обработка предыдущего сигнала. Отсюда правило: следует избегать операций ввода-вывода и вызовов большинства библиотечных и системных функций в обработчиках сигналов.

Обработчик должен выполнять минимум действий в ответ на получение сигнала и как можно быстрее возвращать управление в программу (или просто завершать ее работу). В большинстве случаев обработчик просто фиксирует факт поступления сигнала, а основная программа периодически проверяет, был ли сигнал, и реагирует должным образом.

Тем не менее возможность прерывания обработчика никогда нельзя исключать. Это очень сложная ситуация для диагностирования и отладки (и наглядный пример состояния гонки, о котором пойдет речь в разделе 4.4. "Синхронизация потоков и критические секции"). Необходимо внимательно следить за тем, что именно делается в обработчике.

Даже присвоение значения глобальной переменной несет потенциальную опасность, так как данная операция может занять два или три такта процессора, а за это время успеет прийти следующий сигнал, вследствие чего переменная окажется поврежденной. Если обработчик использует какую-то переменную в качестве флага поступления сигнала, она должна иметь специальный тип `sig_atomic_t`. Linux гарантирует, что операция присваивания значения такой переменной займет ровно один такт и не будет прервана. На самом деле тип `sig_atomic_t` в Linux эквивалентен типу `int`; более того, операции присваивания целочисленных переменных (32- и 16-разрядных) и указателей всегда атомарны. Использовать тип `sig_atomic_t` необходимо для того, чтобы программу можно было перенести в любую стандартную UNIX-систему.

В листинге 3.5 представлен шаблон программы, в которой функция-обработчик подсчитывает, сколько раз программа получает сигнал `SIGUSR1`.

### ***Листинг 3.5. (sigusr1.c) Корректное применение обработчика сигнала***

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

sig_atomic_t sigusr1_count = 0;

void handler(int signal_number) {
    ++sigusr1_count;
}
```

```

int main() {
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = &handler;
    sigaction(SIGUSR1, &sa, NULL);

    /* далее идет основной текст. */
    /* ... */

    printf("SIGUSR1 was raised %d times\n", sigusr1_count);
    return 0;
}

```

### 3.4. Завершение процесса

Обычно процесс завершается одним из двух способов: либо выполняющаяся программа вызывает функцию `exit()`, либо функция `main()` заканчивается. У каждого процесса есть код завершения число, возвращаемое родительскому процессу. Этот код передается в качестве аргумента функции `exit()` или возвращается функцией `main()`.

Возможно также аварийное завершение процесса, в ответ на получение сигнала. Таковыми могут быть, например, упоминавшиеся выше сигналы `SIGBUS`, `SIGSEGV` и `SIGFPE`. Есть сигналы, явно запрашивающие прекращение работы процесса. В частности, сигнал `SIGINT` посылается, когда пользователь нажимает `<Ctrl+C>`. Сигнал `SIGTERM` посылается процессу командой `kill` по умолчанию. Если программа вызывает функцию `abort()`, она посылает сама себе сигнал `SIGABRT`. Самый "могучий" из всех сигналов `SIGKILL`: он приводит к безусловному уничтожению процесса и не может быть ни блокирован, ни обработан.

Любой сигнал можно послать с помощью команды `kill`, указав дополнительный флаг. Например, чтобы уничтожить процесс, послав ему сигнал `SIGKILL`, воспользуйтесь следующей командой:

```
% kill -KILL идентификатор_процесса
```

Для отправки сигнала из программы предназначена функция `kill()`. Ее первым аргументом является идентификатор процесса. Второй аргумент номер сигнала (стандартному поведению команды `kill` соответствует сигнал `SIGTERM`). Например, если переменная `child_pid` содержит идентификатор дочернего процесса, то следующая функция, вызываемая из родительского процесса, вызывает завершение работы потомка:

```
kill(child_pid, SIGTERM);
```

Для использования функции `kill()` необходимо включить в программу файлы `<sys/types.h>` и `<signal.h>`.

По существующему соглашению код завершения указывает на то, успешно ли выполнялась программа. Нулевой код говорит о том, что все в порядке, ненулевой код свидетельствует об ошибке. В последнем случае конкретное значение кода может подсказать природу ошибки. Подобным образом функционируют все компоненты GNU/Linux. Например, на это рассчитывает интерпретатор команд, когда в командных сценариях вызовы программ объединяются с помощью операторов `&&` (логическое умножение) и `||` (логическое сложение). Таким образом, функция `main()` должна явно возвращать 0 при отсутствии ошибок.

Помните о следующем ограничении: несмотря на то что тип параметра функции `exit()`, как и тип возвращаемого значения функции `main()`, равен `int`, операционная система Linux записывает код завершения лишь в младший из четырех байтов. Это означает, что значение кода должно находиться в диапазоне от 0 до 127. Коды, значение которых больше 128,

интерпретируются особым образом: когда процесс уничтожается вследствие получения сигнала, его код завершения равен 128 плюс номер сигнала.

### 3.4.1. Ожидание завершения процесса

Читатели, запускаявшие программу `fork-exec` (см. листинг 3.4), должно быть, обратили внимание на то, что вывод команды `ls` часто появляется после того, как основная программа уже завершила свою работу. Это связано с тем, что дочерний процесс, в котором выполняется команда `ls`, планируется независимо от родительского процесса. Linux многозадачная операционная система, процессы в ней выполняются одновременно, поэтому нельзя заранее предсказать, кто предок или потомок завершится раньше.

Но бывают ситуации, когда родительский процесс должен дождаться завершения одного или нескольких своих потомков. Это можно сделать с помощью функций семейства `wait()`. Они позволяют родительскому процессу получать информацию о завершении дочернего процесса. В семейство входят четыре функции, различающиеся объемом возвращаемой информации, а также способом задания дочернего процесса.

### 3.4.2. Системные вызовы `wait()`

Самая простая функция в семействе называется `wait()`. Она блокирует вызывающий процесс до тех пор, пока один из его дочерних процессов не завершится (или не произойдет ошибка). Код состояния потомка возвращается через аргумент, являющийся указателем на целое число. В этом коде зашифрована различная информация о потомке. Например, макрос `WEXITSTATUS()` возвращает код завершения дочернего процесса. Макрос `WIFEXITED()` позволяет узнать, как именно завершился процесс: обычным образом (с помощью функции `exit()` или оператора `return` функции `main()`) либо аварийно вследствие получения сигнала. В последнем случае макрос `WTERMSIG()` извлекает из кода завершения номер сигнала.

Ниже приведена доработанная версия функции `main()` из файла `fork-exec.c`. На этот раз программа вызывает функцию `wait()`, чтобы дождаться завершения дочернего процесса, в котором выполняется команда `ls`.

```
int main() {
    int child_status;
    /* Список аргументов, передаваемых команде ls. */
    char* arg_list[] = {
        "ls", /* argv[0] имя программы. */
        "-l",
        "/",
        NULL /* Список аргументов должен оканчиваться указателем
        NULL. */
    };

    /* Порождаем дочерний процесс, который выполняет команду ls.
    Игнорируем возвращаемый идентификатор дочернего процесса. */
    spawn("ls*", arg_list);
    /* Дожидаемся завершения дочернего процесса. */
    wait(&child_status);
    if (WIFEXITED(child_status));
    printf("the child process exited normally, with exit code %d\n",
        WEXITSTATUS(child_status));
    else
```

```
printf("the child process exited abnormally\n");
return 0;
}
```

Расскажем о других функциях семейства. Функция `waitpid()` позволяет дожидаться завершения конкретного дочернего процесса, а не просто любого. Функция `wait3()` возвращает информацию о статистике использования центрального процессора завершившимся дочерним процессом. Функция `wait4()` позволяет задать дополнительную информацию о том, завершения каких процессов следует дожидаться.

### 3.4.3. Процессы-зомби

Если дочерний процесс завершается в то время, когда родительский процесс заблокирован функцией `wait()`, он успешно удаляется и его код завершения передается предку через функцию `wait()`. Но что произойдет, если потомок завершился, а родительский процесс так и не вызвал функцию `wait()`? Дочерний процесс просто исчезнет? Нет, ведь в этом случае информация о его завершении (было ли оно аварийным или нет и каков код завершения) пропадет. Вместо этого дочерний процесс становится процессом-зомби.

Зомби это процесс, который завершился, но не был удален. Удаление зомби возлагается на родительский процесс. Функция `wait()` тоже это делает, поэтому перед ее вызовом не нужно проверять, продолжает ли выполняться требуемый дочерний процесс. Предположим, к примеру, что программа создает дочерний процесс, выполняет нужные вычисления и затем вызывает функцию `wait()`. Если к тому времени дочерний процесс еще не завершился, функция `wait()` заблокирует программу. В противном случае процесс на некоторое время превратится в зомби. Тогда функция `wait()` извлечет код его завершения, система удалит процесс и функция немедленно завершится.

Что же всё-таки случится, если родительский процесс не удалит своих потомков? Они останутся в системе в виде зомби. Программа, показанная в листинге 3.6, порождает дочерний процесс, который немедленно завершается, тогда как родительский процесс берет минутную паузу, после чего тоже заканчивает работу, так и не позаботившись об удалении потомка.

#### Листинг 3.6. (*zombie.c*) Создание процесса-зомби

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

int main() {
    pid_t child_pid;
    /* Создание дочернего процесса. */
    child_pid = fork();
    if (child_pid > 0) {
        /* Это родительский процесс делаем минутную паузу. */
        sleep(60);
    } else {
        /* Это дочерний процесс немедленно завершаем работу. */
        exit(0);
    }
    return 0;
}
```

Скомпилируйте этот файл и запустите программу. Пока программа работает, перейдите в

другое окно и просмотрите список процессов с помощью следующей команды:

```
% ps -e -o pid,ppid,stat,cmd
```

Эта команда отображает идентификатор самого процесса и его предка, а также статус процесса и его командную строку. Обратите внимание на присутствие двух процессов с именем `zombie`. Один из них предок, другой потомок. У последнего идентификатор родительского процесса равен идентификатору основного процесса `zombie`, при этом потомок обозначен как `<defunct>` (несуществующий), а его код состояния равен `Z` (т.е. `zombie` зомби).

Итак, мы хотим узнать, что будет, когда программа `zombie` завершится, не вызвав функцию `wait()`. Останется ли процесс-зомби? Нет выполните команду `ps` и убедитесь в этом: оба процесса `zombie` исчезли. Дело в том, что после завершения программы управление ее дочерними процессами принимает на себя специальный процесс демон `init`, который всегда работает, имея идентификатор 1 (это первый процесс, запускаемый при загрузке Linux). Демон `init` автоматически удаляет все унаследованные им дочерние процессы-зомби.

#### 3.4.4. Асинхронное удаление дочерних процессов

Если дочерний процесс просто вызывает другую программу с помощью функции `exec()`, то в родительском процессе можно сразу же вызвать функцию `wait()` и пассивно дожидаться завершения потомка. Но очень часто нужно, чтобы родительский процесс продолжал выполняться одновременно с одним или несколькими своими потомками. Как в этом случае получать сигналы об их завершении?

Один подход заключается в периодическом вызове функции `wait3()` или `wait4()`. Функция `wait()` в данной ситуации не подходит, так как в случае отсутствия завершившегося дочернего процесса она заблокирует основную программу. А вот упомянутые две функции принимают дополнительный флаг `WNOHANG`, переводящий их в неблокируемый режим, в котором функция либо удаляет дочерний процесс, если он есть, либо просто завершается. В первом случае возвращается идентификатор процесса, во втором 0.

Более элегантный подход состоит в асинхронном уведомлении родительского процесса о завершении потомка. Существуют разные способы сделать это, но проще всего воспользоваться сигналом `SIGCHLD`, посылаемым как раз тогда, когда завершается дочерний процесс. По умолчанию программа никак не реагирует на этот сигнал, поэтому раньше вы могли и не знать о его существовании.

Таким образом, нужно организовать удаление дочерних процессов в обработчике сигнала `SIGCHLD`. Естественно, код состояния удаляемого процесса следует сохранять в глобальной переменной, если эта информация необходима основной программе. В листинге 3.7 показана программа, в которой реализована данная методика.

#### Листинг 3.7. (`sigchld.c`) Удаление дочерних процессов в обработчике сигнала `SIGCHLD`

```
#include <signal.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>

sig_atomic_t child_exit_status;

void clean_up_child_process(int signal_number) {
```



```
/* Удаление дочернего процесса. */
int status;
wait(&status);
/* Сохраняем статус потомка в глобальной переменной. */
child_exit_status = status;
}

int main() {
/* Обрабатываем сигнал SIGCHLD, вызывая функцию
clean_up_child_process(). */
struct sigaction sigchld_action;
memset(&sigchld_action, 0, sizeof(sigchld_action));
sigchld_action.sa_handler = &clean_up_child_process;
sigaction(SIGCHLD, &sigchld_action, NULL);

/* Далее выполняются основные действия, включая порождение
дочернего процесса. */
/* ... */

return 0;
}
```

# Глава 4

## Потоки

Потоки, как и процессы, это механизм, позволяющий программам выполнять несколько действий одновременно. Потоки работают параллельно. Ядро Linux планирует их работу асинхронно, прерывая время от времени каждый из них, чтобы дать шанс остальным.

С концептуальной точки зрения поток существует внутри процесса, являясь более мелкой единицей управления программой. При вызове программы Linux создает для нее новый процесс, а в нем единственный поток, последовательно выполняющий программный код. Этот поток может создавать дополнительные потоки. Все они находятся в одном процессе, выполняя ту же самую программу, но, возможно, в разных ее местах.

Мы уже знаем, как программа порождает дочерний процесс. Первоначально он находится в родительской программе, получая копии ее виртуальной памяти, дескрипторов файлов и т.п. Модификация содержимого памяти, закрытие файлов и другие подобные действия в дочернем процессе не влияют на работу родительского процесса и наоборот. С другой стороны, когда программа создает поток, ничего не копируется. Оба потока старый и новый имеют доступ к общему виртуальному пространству, общим дескрипторам файлов и другим системным ресурсам. Если, к примеру, один поток меняет значение переменной, это изменение отражается на другом потоке. Точно так же, когда один поток закрывает файл, второй поток теряет возможность работать с этим файлом. В связи с тем что процесс и все его потоки могут выполнять лишь одну программу одновременно, как только одни из потоков вызывает функцию семейства `exec()`, все остальные потоки завершаются (естественно, новая программа может создавать собственные потоки).

В Linux реализована библиотека API-функций работы с потоками, соответствующая стандарту POSIX (она называется Pthreads). Все функции и типы данных библиотеки объявлены в файле `<pthread.h>`. Эти функции не входят в стандартную библиотеку языка C, поэтому при компоновке программы нужно указывать опцию `-lpthread` в командной строке.

### 4.1. Создание потока

Каждому потоку в процессе назначается собственный идентификатор. При ссылке на идентификаторы потоков в программах, написанных на языке C или C++, нужно использовать тип данных `pthread_t`.

После создания поток начинает выполнять *потокową функцию*. Это самая обычная функция, которая содержит код потока. По завершении функции поток уничтожается. В Linux потоковые функции принимают единственный параметр типа `void*` и возвращают значение аналогичного типа. Этот параметр называется *аргументом потока*. Через него программы могут передавать данные потокам. Аналогичным образом через возвращаемое значение программы принимают данные от потоков.

Функция `pthread_create()` создает новый поток. Ей передаются следующие параметры.

- Указатель на переменную типа `pthread_t`, в которой сохраняется идентификатор нового потока.

- Указатель на объект *атрибутов потока*. Этот объект определяет взаимодействие потока с остальной частью программы. Если задать его равным `NULL`, поток будет создан со стандартными атрибутами. Подробнее данная тема обсуждается в разделе 4.1.5, "Атрибуты потоков".

■ Указатель на потоковую функцию. Функция имеет следующий тип:

```
void* (*)(void*)
```

■ Значение аргумента потока (тип void\*). Данное значение без каких-либо изменений передается потоковой функции.

Функция pthread\_create() немедленно завершается, и родительский поток переходит к выполнению инструкции, следующей после вызова функции. Тем временем новый поток начинает выполнять потоковую функцию. ОС Linux планирует работу обоих потоков асинхронно, поэтому программа не должна рассчитывать на какую-то согласованность между ними.

Программа, представленная в листинге 4.1, создает поток, который непрерывно записывает символы 'x' в стандартный поток ошибок. После вызова функции pthread\_create() основной поток начинает делать то же самое, но вместо символов 'x' печатаются символы 'o'.

#### *Листинг 4.1. (thread-create.c) Создание потока*

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

/* Запись символов 'x' в поток stderr.
Параметр не используется.
Функция никогда не завершается. */

void* print_xs(void* unseed) {
while (1)
fputc('x', stderr);
return NULL;
}

/* Основная программа. */
int main() {
pthread_t thread_id;
/* Создание потока. Новый поток выполняет
функцию print_xs(). */
pthread_create(&thread_id, NULL, &print_xs, NULL);
/* Непрерывная запись символов 'o' в поток stderr. */
while (1)
fputc('o', stderr);
return 0;
}
```

Компиляция и компоновка программы осуществляются следующим образом:

```
% cc -o thread-create thread-create.c -lpthread
```

Запустите программу, и вы увидите, что символы 'x' и 'o' чередуются самым непредсказуемым образом.

При нормальных обстоятельствах поток завершается одним из двух способов. Один из них выход из потоковой функции. Возвращаемое ею значение считается значением, передаваемым из потока в программу. Второй способ вызов специальной функции pthread\_exit(). Это может быть сделано как в потоковой функции, так и в любой другой функции, явно или неявно вызываемой из нее. Аргумент функции pthread\_exit() является значением, которое возвращается потоком.

#### **4.1.1. Передача данных потоку**

Потоковый аргумент это удобное средство передачи данных потокам. Но поскольку его тип `void*`, данные содержатся не в самом аргументе. Он лишь должен указывать на какую-то структуру или массив. Лучше всего создать для каждой потоковой функции собственную структуру, в которой определялись бы "параметры", ожидаемые потоковой функцией.

Благодаря наличию потокового аргумента появляется возможность использовать одну и ту же потоковую функцию с разными потоками. Все они будут выполнять один и тот же код, но с разными данными.

Программа, приведенная в листинге 4.2, напоминает предыдущий пример. На этот раз создаются два потока: один отображает символы 'x', а другой символы 'o'. Чтобы вывод на экран не длился бесконечно, потокам передается дополнительный аргумент, определяющий, сколько раз следует отобразить символ. Одна и та же функция `char_print()` эксплуатируется обоими потоками, но каждый из них конфигурируется независимо с помощью структуры `char_print_parms`.

#### ***Листинг 4.2. (thread-create2.c) Создание двух потоков***

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

/* Параметры для функции char_print(). */
struct char_print_parms {
/* Отображаемый символ. */
char character;
/* Сколько раз его нужно отобразить. */
int count;
};

/* Запись указанного числа символов в поток stderr. Аргумент
PARAMETERS является указателем на структуру char_print_parms. */
void* char_print(void* parameters) {
/* Приведение указателя к нужному типу. */
struct char_print_parms* p =
(struct char_print_parms*)parameters;
int i;
for (i = 0; i < p->count; ++i)
fputc(p->character, stderr);
return NULL;
}

/* Основная программа. */
int main() {
pthread_t thread1_id;
pthread_t thread2_id;
struct char_print_parms thread1_args;
struct char_print_parms thread2_args;

/* Создание нового потока, отображающего 30000
символов 'x'. */
thread1_args.character = 'x';
thread1_args.count = 30000;
pthread_create(&thread1_id, NULL, &char_print, &thread1_args);

/* Создание нового потока, отображающего 20000
```

```

символов 'o'. */
thread2_args.character = 'o';
thread2_args.count = 20000;
pthread_create(&thread2_id, NULL, &char_print, &thread2_args);
return 0;
}

```

Но постойте! Приведенная программа имеет серьезную ошибку. Основной поток (выполняющий функцию `main()`) создает структуры `thread1_args` и `thread2_args` в виде локальных переменных, а затем передает указатели на них дочерним потокам. Что мешает Linux распланировать работу потоков так, чтобы функция `main()` завершилась до того, как будут завершены другие два потока? Ничего! Но если это произойдет, структуры окажутся удаленными из памяти, хотя оба потока все еще ссылаются на них.

#### 4.1.2. Ожидание завершения потоков

Одно из решений описанной выше проблемы заключается в том, чтобы заставить функцию `main()` дожидаться завершения обоих потоков. Нужна лишь функция наподобие `wait()`, которая работает не с процессами, а с потоками. Такая функция называется `pthread_join()`. Она принимает два аргумента: идентификатор ожидаемого потока и указатель на переменную `void*`, в которую будет записано значение, возвращаемое потоком. Если последнее не важно, задайте в качестве второго аргумента `NULL`.

В листинге 4.3 приведена исправленная версия функции `main()` из предыдущего, неправильного примера. В данном случае функция `main()` не завершается, пока оба дочерних потока не выполнят свои задания и не перестанут ссылаться на переданные им структуры.

#### *Листинг 4.3. Исправленная функция `main()` из файла `thread-create.c`*

```

int main() {
pthread_t thread1_id;
pthread_t thread2_id;
struct char_print_parms thread1_args;
struct char_print_parms thread2_args;

/* Создание нового потока, отображающего 30000
символов 'x'. */
thread1_args.character = 'x';
thread1_args.count = 30000;
pthread_create(&thread1_id, NULL, &char_print, &thread1_args);

/* Создание нового потока, отображающего
20000 символов 'o'. */
thread2_args.character = 'o';
thread2_args.count = 20000;
pthread_create(&thread2_id, NULL, &char_print, &thread2_args);

/* Убеждаемся, что завершился первый поток. */
pthread_join(thread1_id, NULL);

/* Убеждаемся, что завершился второй поток. */
pthread_join(thread2_id, NULL);
}

```

```
/* Теперь можно спокойно завершать работу. */
return 0;
}
```

Мораль сей басни такова: убедитесь, что любые данные, переданные потоку по ссылке, не удаляются (*даже другим потоком*) до тех пор, пока поток не завершит свою работу с ними. Это относится как к локальным переменным, удаляемым автоматически при выходе за пределы своей области видимости, так и к динамическим переменным, удаляемым с помощью функции `free()` (или оператора `delete` в C++).

### 4.1.3. Значения, возвращаемые потоками

Если второй аргумент функции `pthread_join()` не равен `NULL`, то в него помещается значение, возвращаемое потоком. Как и потоковый аргумент, это значение имеет тип `void*`. Если поток возвращает обычное число типа `int`, его можно свободно привести к типу `void*`, а затем выполнить обратное преобразование по завершении функции `pthread_join()`. [\[13\]](#)

Программа, представленная в листинге 4.4, в отдельном потоке вычисляет  $n$ -е простое число и возвращает его в программу. Тем временем функция `main()` может продолжать свои собственные вычисления. Сразу признаемся: алгоритм последовательного деления, используемый в функции `compute_prime()`, весьма неэффективен. В книгах по численным методам описаны более мощные алгоритмы (например, "решето Эратосфена").

#### *Листинг 4.4. (primes.c) Вычисление простых чисел в потоке*

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

/* Находим простое число с порядковым номером N, где N -- это
значение, на которое указывает параметр ARG. */
void* compute_prime(void* arg) {
    int candidate = 2;
    int n = *((int*)arg);

    while (1) {
        int factor;
        int is_prime = 1;

        /* Проверка простого числа путем последовательного деления. */
        for (factor = 2; factor < candidate; ++factor)
            if (candidate % factor == 0) {
                is_prime = 0;
                break;
            }
        /* Это то простое число, которое нам нужно? */
        if (is_prime) {
            if (--n == 0)
                /* Возвращаем найденное число в программу. */
                return (void*)candidate;
            ++candidate;
        }
    }
    return NULL;
}
```

```

int main() {
pthread_t thread;
int which_prime = 5000;
int prime;

/* Запускаем поток, вычисляющий 5000-е простое число. */
pthread_create(&thread, NULL, &compute_prime, &which_prime);
/* Выполняем другие действия. */
/* Дожидаемся завершения потока и принимаем возвращаемое им
значение. */
pthread_join(thread, (void*)&prime);
/* Отображаем вычисленный результат. */
printf("The %dth prime number is %d.\n", which_prime, prime);
return 0;
}

```

#### 4.1.4. Подробнее об идентификаторах потоков

Иногда в программе возникает необходимость определить, какой поток выполняет ее в данный момент. Функция `pthread_self()` возвращает идентификатор потока, в котором она вызвана. Для сравнения двух разных идентификаторов предназначена функция `pthread_equal()`.

Эти функции удобны для проверки соответствия заданного идентификатора текущему потоку. Например, поток не должен вызывать функцию `pthread_join()`, чтобы ждать самого себя (в подобной ситуации возвращается код ошибки `EDEADLK`). Избежать этой ошибки позволяет следующая проверка:

```

if (!pthread_equal(pthread_self(), other_thread)) pthread_join(other_thread,
NULL);

```

#### 4.1.5. Атрибуты потоков

Потоковые атрибуты это механизм настройки поведения отдельных потоков. Вспомните, что функция `pthread_create()` принимает аргумент, являющийся указателем на объект атрибутов потока. Если этот указатель равен `NULL`, поток конфигурируется на основании стандартных атрибутов.

Для задания собственных атрибутов потока выполните следующие действия.

- 1.Создайте объект типа `pthread_attr_t`.
- 2.Вызовите функцию `pthread_attr_init()`, передав ей указатель на объект. Эта функция присваивает неинициализированным атрибутам стандартные значения.
- 3.Запишите в объект требуемые значения атрибутов.
- 4.Передайте указатель на объект в функцию `pthread_create()`.
- 5.Вызовите функцию `pthread_attr_destroy()`, чтобы удалить объект из памяти. Сама переменная `pthread_attr_t` не удаляется; ее можно проинициализировать повторно с помощью функции `pthread_attr_init()`.

Один и тот же объект может быть использован для запуска нескольких потоков. Нет необходимости хранить объект после того, как поток был создан.

Для большинства Linux-приложений интерес представляет один-единственный атрибут (остальные используются в приложениях реального времени): *статус отсоединения потока*. Поток может быть создан как *ожидаемый* (по умолчанию) или *отсоединенный*. Ожидаемый

поток, подобно процессу, после своего завершения не удаляется автоматически операционной системой Linux. Код его завершения хранится где-то в системе (как у процесса-зомби), пока какой-нибудь другой поток не вызовет функцию `pthread_join()`, чтобы запросить это значение. Только тогда ресурсы потока считаются освобожденными. С другой стороны, отсоединенный поток, завершившись, сразу уничтожается. Другие потоки не могут вызвать по отношению к нему функцию `pthread_join()` или получить возвращаемое им значение.

Чтобы задать статус отсоединения потока, воспользуйтесь функцией `pthread_attr_setdetachstate()`. Первый ее аргумент это указатель на объект атрибутов потока, второй требуемый статус. Ожидаемые потоки создаются по умолчанию, поэтому в качестве второго аргумента имеет смысл указывать только значение `PTHREAD_CREATE_DETACHED`.

Программа, представленная в листинге 4.5, создает отсоединенный поток, устанавливая соответствующим образом атрибуты потока.

#### ***Листинг 4.5. (detached.c) Шаблон программы, создающей отсоединенный поток***

```
#include <pthread.h>

void* thread_function(void* thread_arg) {
/* Тело потоковой функции... */
}

int main() {
pthread_attr_t attr;
pthread_t thread;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_create(&thread, &attr, &thread_function, NULL);
pthread_attr_destroy(&attr);

/* Тело основной программы... */

/* Дождаться завершения второго потока нет необходимости. */
return 0;
}
```

Даже если поток был создан ожидаемым, его позднее можно сделать отсоединенным. Для этого нужно вызвать функцию `pthread_detach()`. Обратное преобразование невозможно.

## **4.2. Отмена потока**

Обычно поток завершается при выходе из потоковой функции или вследствие вызова функции `pthread_exit()`. Но существует возможность запросить из одного потока уничтожение другого. Это называется отменой, или принудительным завершением, потока.

Чтобы отменить поток, вызовите функцию `pthread_cancel()`, передав ей идентификатор требуемого потока. Далее можно дождаться завершения потока. Вообще-то, это обязательно нужно делать с целью освобождения ресурсов, если только поток не является отсоединенным. Отмененный поток возвращает специальное значение `PTHREAD_CANCELED`.

Во многих случаях поток выполняет код, который нельзя просто взять и прервать. Например, поток может выделить какие-то ресурсы, поработать с ними, а затем удалить. Если отмена потока произойдет где-то посередине, освободить занятые ресурсы станет невозможно,



вследствие чего они окажутся потерянными для системы. Чтобы учесть эту ситуацию, поток должен решить, где и когда он может быть отменен.

С точки зрения возможности отмены поток находится в одном из трех состояний.

■ *Асинхронно отменяемый*. Такой поток можно отменить в любой точке его выполнения.

■ *Синхронно отменяемый*. Поток можно отменить, но не везде. Запрос на отмену помещается в очередь, и поток отменяется только по достижении определенной точки.

■ *Неотменяемый*. Попытки отменить поток игнорируются. Первоначально поток является синхронно отменяемым.

#### 4.2.1. Синхронные и асинхронные потоки

Асинхронно отменяемый поток "свободен" в любое время. Синхронно отменяемый поток, наоборот, бывает "свободным", только когда ему "удобно". Соответствующие места в программе называются точками отмены. Запрос на отмену помещается в очередь и находится в ней до тех пор, пока поток не достигнет следующей точки отмены.

Чтобы сделать поток асинхронно отменяемым, воспользуйтесь функцией `pthread_setcanceltype()`. Эта функция влияет на тот поток, в котором она была вызвана. Первый ее аргумент должен быть `PTHREAD_CANCEL_ASYNCHRONOUS` в случае асинхронных потоков и `PTHREAD_CANCEL_DEFERRED` в случае синхронных потоков. Второй аргумент это указатель на переменную, в которую записывается предыдущее состояние потока.

Вот как можно сделать поток асинхронным:

```
pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
```

Что такое точка отмены и где она должна находиться? На этот вопрос нельзя дать прямой ответ. Точка отмены создается с помощью функции `pthread_testcancel()`. Все, что она делает, это обрабатывает отложенный запрос на отмену в синхронном потоке. Ее следует периодически вызывать в потоковой функции в ходе длительных вычислений, там, где поток можно завершить без риска потери ресурсов или других побочных эффектов.

Некоторые функции неявно создают точки отмены. О них можно узнать на `man`-странице, посвященной функции `pthread_cancel()`. Учтите, что они могут вызываться в других функциях, которые, тем самым, косвенно станут точками отмены.

#### 4.2.2. Неотменяемые потоки

Поток может вообще отказаться удаляться, вызвав функцию `pthread_setcancelstate()`. Как и в случае функции `pthread_setcanceltype()`, это оказывает влияние только на вызывающий поток. Первый аргумент функции должен быть `PTHREAD_CANCEL_DISABLE`, если нужно запретить отмену потока, и `PTHREAD_CANCEL_ENABLE` в противном случае. Второй аргумент это указатель на переменную, в которую записывается предыдущее состояние потока.

Вот как можно запретить отмену потока:

```
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
```

Функция `pthread_setcancelstate()` позволяет организовывать *критические секции*. Критической секцией называется участок программы, который должен быть либо выполнен целиком, либо вообще не выполнен. Другими словами, если поток входит в критическую секцию, он во что бы то ни стало должен дойти до ее конца.

Предположим, к примеру, что для банковской программы требуется написать функцию, осуществляющую перевод денег с одного счета на другой. Для этого нужно добавить заданную сумму на баланс одного счета и вычесть аналогичную сумму с баланса другого счета. Если

между этими двумя операциями произойдет отмена потока, выполняющего функцию, программа ложно увеличит суммарный депозит банка вследствие незавершенной транзакции. Чтобы этого не случилось, обе операции должны выполняться в критической секции.

В листинге 4.6 показан пример функции `process_transaction()`, осуществляющей данную задумку. Функция запрещает отмену потока до тех пор, пока баланс обоих счетов не будет изменен.

#### ***Листинг 4.6. (critical\_section.c) Защита банковской транзакции с помощью критической секции***

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

/* Массив балансов счетов, упорядоченный по номеру счета. */
float* account_balances;

/* перевод денежной суммы, равной параметру DOLLARS, со счета
FROM_ACCT на счет TO_ACCT. Возвращается 0, если транзакция
завершена успешно, или 1, если баланс счета FROM_ACCT
слишком мал. */
int process_transaction(int from_acct, int to_acct,
float dollars) {
int old_cancel_state;

/* Проверяем баланс на счету FROM_ACCT. */
if (account_balances[from_acct] < dollars)
return 1;

/* Начало критической секции. */
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &old_cancel_state);

/* переводим деньги. */
account_balances[to_acct] += dollars;
account_balances[from_acct] -= dollars;
/* Конец критической секции. */
pthread_setcancelstate(old_cancel_state, NULL);

return 0;
}
```

Обратите внимание на то, что по окончании критической секции восстанавливается предыдущее состояние потока, а не режим `PTHREAD_CANCEL_ENABLE`. Это позволит безопасно вызывать функцию `process_transaction()` из другой критической секции.

### **4.2.3. Когда необходимо отменять поток**

В общем случае не рекомендуется отменять поток, если его можно просто завершить. Лучше всего каким-то образом просигнализировать потоку о том, что он должен прекратить работу, а затем дождаться его завершения. Подробнее о способах взаимодействия с потоками речь пойдет ниже в этой главе.

## 4.3. Потокковые данные

В отличие от процессов, все потоки программы делят общее адресное пространство. Это означает, что если один поток модифицирует ячейку памяти (например, глобальную переменную), то это изменение отразится на всех остальных потоках. Таким образом, потоки могут работать с одними и теми же данными, не используя механизмы межзадачного взаимодействия (рассматриваются в главе 5, "Взаимодействие процессов").

Тем не менее у каждого потока свой собственный стек вызова. Это позволяет всем потокам выполнять разный код, а также вызывать функции традиционным способом. При каждом вызове функции в любом потоке создается отдельный набор локальных переменных, которые сохраняются в стеке этого потока.

Иногда все же требуется продублировать определенную переменную, чтобы у каждого потока была ее собственная копия. С этой целью операционная система Linux предоставляет потокам *область потоковых данных*. Переменные, сохраняемые в этой области, дублируются для каждого потока, что позволяет потокам свободно работать с ними, не мешая друг другу. Доступ к потоковым данным нельзя получить с помощью ссылок на обычные переменные, ведь у потоков общее адресное пространство. В Linux имеются специальные функции для чтения и записи значений, хранящихся в области потоковых данных.

Можно создать сколько угодно потоковых переменных, при этом все они должны иметь тип `void*`. Ссылка на каждую переменную осуществляется по ключу. Для создания нового ключа, т.е. новой переменной, предназначена функция `pthread_key_create()`. Первым ее аргументом является указатель на переменную типа `pthread_key_t`. В нее будет записано значение ключа, посредством которого любой поток сможет обращаться к своей копии данных. Вторым аргументом это указатель на функцию очистки ключа. Она будет автоматически вызываться при уничтожении потока; ей передается значение ключа, соответствующее данному потоку. Это очень удобно, так как функция очистки вызывается даже в случае отмены потока в произвольной точке. Если потоковая переменная равна `NULL`, функция очистки не вызывается. Если же такая функция не нужна, задайте в качестве второго параметра функции `pthread_key_create()` значение `NULL`.

После того как ключ создан, каждый поток может назначать ему собственное значение, вызывая функцию `pthread_setspecific()`. Ее первый аргумент это ключ, а второй требуемое значение типа `void*`. Для чтения потоковых переменных предназначена функция `pthread_getspecific()`, единственным аргументом которой является ключ.

Предположим, имеется приложение, распределяющее задачу между несколькими потоками. В целях аудита за каждым потоком закреплен отдельный журнальный файл, куда записываются сообщения о ходе выполнения поставленной задачи. Область потоковых данных удобное место для хранения указателя на журнальный файл каждого потока.

В листинге 4.7 показано, как осуществить задуманное. Для хранения файлового указателя в функции `main()` создается ключ, запоминаемый в переменной `thread_log_key`. Эта переменная является глобальной, поэтому она доступна всем потокам. Когда поток начинает выполнять свою потоковую функцию, он открывает журнальный файл и сохраняет указатель на него в своем ключе. Позднее любой поток может вызвать функцию `write_to_thread_log()`, чтобы записать сообщение в свой журнальный файл. Эта функция извлекает из области потоковых данных указатель на журнальный файл и помещает в файл требуемое сообщение.

```

#include <malloc.h>
#include <pthread.h>
#include <stdio.h>

/* Ключ, связывающий указатель журнального файла с каждым
потоком. */
static pthread_key_t thread_log_key;

/* Запись параметра MESSAGE в журнальный файл текущего потока. */
void write_to_thread_log(const char* message) {
FILE* thread_log =
(FILE*)pthread_getspecific(thread_log_key);
fprintf(thread_log, "%s\n", message);
}

/* Закрытие журнального файла, на который указывает параметр
THREAD_LOG. */
void close_thread_log(void* thread_log) {
fclose((FILE*)thread_log);
}

void* thread_function(void* args) {
char thread_log_filename[20];
FILE* thread_log;
/* Создание имени журнального файла для текущего потока. */
sprintf(thread_log_filename, "thread%d.log",
(int)pthread_self());
/* Открытие журнального файла. */
thread_log = fopen(thread_log_filename, "w");
/* Сохранение указателя файла в области потоковых данных,
под ключом thread_log_key. */
pthread_setspecific(thread_log_key, thread_log);
write_to_thread_log("Thread starting.");
/* Далее идет основное тело потока... */
return NULL;
}

int main() {
int i;
pthread_t threads[5];

/* Создание ключа, который будет связывать указатели
журнальных файлов с областью потоковых данных. Функция
close_thread_log() закрывает все файлы. */
pthread_key_create(&thread_log_key, close_thread_log);
/* Создание потоков. */
for (i = 0; i < 5; ++i)
pthread_create(&(threads[i]), NULL, thread_function, NULL);
/* Ожидание завершения всех потоков. */
for (i = 0; i < 5; ++i)
pthread_join(threads[i], NULL);
return 0;
}

```

Обратите внимание на то, что в функции thread\_function() не нужно закрывать журнальный файл. Просто когда создавался ключ, функция close\_thread\_log() была назначена

функцией очистки данного ключа. Когда бы поток ни завершился, операционная система Linux вызовет эту функцию, передав ей значение ключа, соответствующее данному потоку. В функции `close_thread_log()` и происходит закрытие файла.

### 4.3.1. Обработчики очистки

Функции очистки ключей гарантируют, что в случае завершения или отмены потока не произойдет потерн ресурсов. Но иногда возникает необходимость в создании функции, которая будет связана не с ключом, дублируемым между потоками, а с обычным ресурсом. Такая функция называется *обработчиком очистки*.

Обработчик очистки вызывается при завершении потока. Он принимает один аргумент типа `void*`, который передается обработчику при его регистрации. Это позволяет использовать один и тот же обработчик для удаления нескольких экземпляров ресурса.

Обработчик очистки это временная мера, требуемая только тогда, когда поток завершается или отменяется, не закончив выполнять определенный участок кода. При нормальных обстоятельствах ресурс должен удаляться явно.

Для регистрации обработчика следует вызвать функцию `pthread_cleanup_push()`, передав ей указатель на обработчик и значение его аргумента. Каждому такому вызову должен соответствовать вызов функции `pthread_cleanup_pop()`, которая отменяет регистрацию обработчика. Для удобства эта функция принимает дополнительный целочисленный флаг. Если он не равен нулю, при отмене регистрации выполняется операция очистки.

В листинге 4.8 показан фрагмент программы, в котором обработчик очистки применяется для удаления динамического буфера при завершении потока.

#### Листинг 4.8. (*cleanup.c*) Фрагмент программы, содержащий обработчик очистки потока

```
#include <malloc.h>
#include <pthread.h>

/* Выделение временного буфера. */
void* allocate_buffer(size_t size) {
    return malloc(size);
}

/* Удаление временного буфера. */
void deallocate_buffer(void* buffer) {
    free(buffer);
}

void do_some_work() {
    /* Выделение временного буфера. */
    void* temp_buffer = allocate_buffer(1024);
    /* Регистрация обработчика очистки для данного буфера. Этот
    обработчик будет удалять буфер при завершении или отмене
    потока. */
    pthread_cleanup_push(deallocate_buffer, temp_buffer);

    /* Выполнение других действий... */

    /* Отмена регистрации обработчика. Поскольку функции передается
```

```
ненулевой аргумент, она выполняет очистку, вызывая функцию  
deallocate_buffer(). */  
pthread_cleanup_pop(1);  
}
```

В данном случае функции `pthread_cleanup_pop()` передается ненулевой аргумент, поэтому функция очистки `deallocate_buffer()` вызывается автоматически. В данном простейшем случае можно было в качестве обработчика непосредственно использовать стандартную библиотечную функцию `free()`.

#### 4.3.2. Очистка потоковых данных в C++

Программисты, работающие на C++, привыкли к тому, что очистку за них делают деструкторы объектов. Когда объект выходит за пределы своей области видимости, либо по достижении конца блока, либо вследствие возникновения исключительной ситуации, среда выполнения C++ гарантирует вызов деструкторов для тех автоматических переменных, у которых они есть. Это удобный механизм очистки, работающий независимо от того, как осуществляется выход из конкретного программного блока.

Тем не менее, если поток вызывает функцию `pthread_exit()`, среда выполнения C++ не может гарантировать вызов деструкторов для всех автоматических переменных, находящихся в стеке потока. Чтобы этого добиться, нужно вызвать функцию `pthread_exit()` в рамках конструкции `try/catch`, охватывающей все тело потоковой функции. При этом перехватывается специальное исключение `ThreadExitException`.

Программа, приведенная в листинге 4.9, иллюстрирует данную методику. Потоковая функция сообщает о своем намерении завершить поток, генерируя исключение `ThreadExitException`, а не вызывая функцию `pthread_exit()` явно. Поскольку исключение перехватывается на самом верхнем уровне потоковой функции, все локальные переменные, находящиеся в стеке потока, будут удалены правильно.

#### Листинг 4.9. (cxx-exit.cpp) Безопасное завершение потока в C++

```
#include <pthread.h>  
  
class ThreadExitException {  
public:  
/* Конструктор, принимающий аргумент RETURN_VALUE, в котором  
содержится возвращаемое потоком значение. */  
ThreadExitException(void* return_value) :  
thread_return_value_(return_value) {  
}  
  
/* Реальное завершение потока. В программу возвращается  
значение, переданное конструктору. */  
void* DoThreadExit() {  
pthread_exit(thread_return_value_);  
}  
  
private:  
/* Значение, возвращаемое в программу при завершении потока. */  
void* thread_return_value_;  
};
```

```

void do_some_work() {
while (1) {
/* Здесь выполняются основные действия... */

if (should_exit_thread_immediately())
throw ThreadExitException(/* поток возвращает */NULL);
}
}

void* thread_function(void*) {
try {
do_some_work();
} catch (ThreadExitException ex) {
/* Возникла необходимость завершить поток. */
ex.DoThreadExit();
}
return NULL;
}

```

## 4.4. Синхронизация потоков и критические секции

Программирование потоков нетривиальная задача, ведь большинство потоков выполняется одновременно. К примеру, невозможно определить, когда система предоставит доступ к процессору одному потоку, а когда другому. Длительность этого доступа может быть как достаточно большой, так и очень короткой, в зависимости от того, как часто система переключает задания. Если в системе есть несколько процессоров, потоки могут выполняться одновременно в буквальном смысле.

Отладка потоковой программы также затруднена, ведь не всегда можно воссоздать ситуацию, приведшую к проблеме. В одном случае программа работает абсолютно правильно, а в другом вызывает системный сбой. Нельзя заставить систему распланировать выполнение потоков так, как она сделала при предыдущем запуске программы.

Большинство ошибок, возникающих при работе с потоками, связано с тем, что потоки обращаются к одним и тем же данным. Как уже говорилось, это одно из главных достоинств потоков, оно же является их бедствием. Если один поток заполняет структуру данными в то время, когда второй поток обращается к этой же структуре, возникает хаос. Очень часто неправильно написанные потоковые программы корректно работают только в том случае, когда один поток планируется системой с более высоким приоритетом, т.е. чаще или быстрее обращается к процессору, чем другой поток. Подобного рода ошибки называются *состоянием гонки*: потоки преследуют друг друга в попытке изменить одни и те же данные.

### 4.4.1. Состояние гонки

Предположим, что в программу поступает группа запросов, которые обрабатываются несколькими одновременными потоками. Очередь запросов представлена связанным списком объектов типа `struct job`.

Когда каждый поток завершает свою операцию, он обращается к очереди и проверяет, есть ли в ней еще необработанные запросы. Если указатель `job_queue` не равен `NULL`, поток удаляет из списка самый верхний элемент и перемещает указатель на следующий элемент. Потоковая функция, работающая с очередью заданий, представлена в листинге 4.10.

```
#include <malloc.h>

struct job {
/* Ссылка на следующий элемент связанного списка. */
struct job* next;

/* Другие поля, описывающие требуемую операцию... */
};

/* Список отложенных заданий. */
struct job* job_queue;

/* Обработка заданий до тех пор, пока очередь не опустеет. */
void* thread_function(void* arg) {
while (job_queue != NULL) {
/* Запрашиваем следующее задание. */
struct job* next_job = job_queue;
/* Удаляем задание из списка. */
job_queue = job_queue->next;
/* выполняем задание. */
process_job(next_job);
/* Очистка. */
free(next_job);
}
return NULL;
}
```

Теперь предположим, что два потока завершают свои операции примерно в одно и то же время, а в очереди остается только одно задание. Первый поток проверяет, равен ли указатель `job_queue` значению `NULL`, и, обнаружив, что очередь не пуста, входит в цикл, где сохраняет указатель на объект задания в переменной `next_job`. В этот момент Linux прерывает первый поток и активизирует второй. Он тоже проверяет указатель `job_queue`, устанавливает, что он не равен `NULL`, и записывает тот же самый указатель в свою переменную `next_job`. Увы, теперь мы имеем два потока, выполняющих одно и то же задание.

Далее ситуация только ухудшается. Первый поток удаляет последнее задание из очереди, делая переменную `job_queue` равной `NULL`. Когда второй поток попытается выполнить операцию `job_queue->next`, возникнет фатальная ошибка сегментации.

Это наглядный пример гонки за ресурсами. Если программе "повезет", система не распланирует потоки именно таким образом и ошибка не проявится. Возможно, только в сильно загруженной системе (или в новой многопроцессорной системе важного клиента!) произойдет "необъяснимый" сбой.

Чтобы исключить возможность гонки, необходимо сделать операции *атомарными*. Атомарная операция неделима и непрерывна; если она началась, то уже не может быть приостановлена или прервана, пока, наконец, не завершится. Выполнение других операций в это время становится невозможным. В нашем конкретном примере проверка переменной `job_queue` и удаление задания должны выполняться как одна атомарная операция.

#### 4.4.2. Исключающие семафоры

Решение проблемы гонки заключается в том, чтобы позволить только одному потоку



обращаться к очереди в конкретный момент времени. Когда поток начинает просматривать очередь, все остальные потоки вынуждены дожидаться, пока он удалит очередное задание из списка.

Реализация такого решения требует поддержки от операционной системы. В Linux имеется специальное средство, называемое *исключающим семафором*, или мьютексом (MUTual EXclusion взаимное исключение). Это специальная блокировка, которую в конкретный момент времени может устанавливать только один поток. Если исключаящий семафор захвачен каким-то потоком, другой поток, обращающийся к семафору, оказывается заблокированным или переведенным в режим ожидания. Как только семафор освобождается, поток продолжает свое выполнение. ОС Linux гарантирует, что между потоками, пытающимися захватить исключаящий семафор, не возникнет гонка. Такой семафор может принадлежать только одному потоку, а все остальные потоки блокируются.

Чтобы создать исключаящий семафор, нужно объявить переменную типа `pthread_mutex_t` и передать указатель на нее функции `pthread_mutex_init()`. Вторым аргументом этой функции является указатель на объект атрибутов семафора. Как и в случае функции `pthread_create()`, если объект атрибутов пуст, используются атрибуты по умолчанию. Переменная исключаящего семафора инициализируется только один раз. Вот как это делается:

```
pthread_mutex_t mutex;  
pthread_mutex_init(&mutex, NULL);
```

Более простой способ создания исключаящего семафора со стандартными атрибутами — присвоение переменной специального значения `PTHREAD_MUTEX_INITIALIZER`. Вызывать функцию `pthread_mutex_init()` в таком случае не требуется. Это особенно удобно для глобальных переменных (а в C++ статических переменных класса). Предыдущий фрагмент программы эквивалентен следующей записи:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Поток может попытаться захватить исключаящий семафор, вызвав функцию `pthread_mutex_lock()`. Если семафор свободен, он переходит во владение данного потока и функция немедленно завершается. Если же семафор уже был захвачен другим потоком, выполнение функции `pthread_mutex_lock()` блокируется и возобновляется только тогда, когда семафор вновь становится свободным. Сразу несколько потоков могут ожидать освобождения исключаящего семафора. Когда это событие наступает, только один поток (выбираемый произвольным образом) разблокируется и получает возможность захватить семафор; остальные потоки остаются заблокированными.

Функция `pthread_mutex_unlock()` освобождает исключаящий семафор. Она должна вызываться только из того потока, который захватил семафор.

В листинге 4.11 представлена другая версия программы, работающей с очередью заданий. Теперь очередь "защищена" исключаящим семафором. Прежде чем получить доступ к очереди (для чтения или записи), каждый поток сначала захватывает семафор. Только когда вся последовательность операций проверки очереди и удаления задания из нее будет закончена, произойдет освобождение семафора. Благодаря этому не возникает описанное выше состояние гонки.

**Листинг 4.11. (*job-queue2.c*) Работа с очередью заданий, защищенной исключаящим семафором**

```
#include <malloc.h>  
#include <pthread.h>
```

```

struct job {
/* Ссылка на следующий элемент связанного списка. */
struct job* next;

/* Другие поля, описывающие требуемую операцию... */
};

/* Список отложенных заданий. */
struct job* job_queue;

/* Исключающий семафор, защищающий очередь. */
pthread_mutex_t job_queue_mutex = PTHREAD_MUTEX_INITIALIZER;

/* Обработка заданий до тех пор, пока очередь не опустеет. */
void* thread_function(void* arg) {
while (1) {
struct job* next_job;
/* Захват семафора, защищающего очередь. */
pthread_mutex_lock(&job_queue_mutex);
/* Теперь можно проверить, является ли очередь пустой. */
if (job_queue == NULL)
next_job = NULL;
else {
/* Запрашиваем следующее задание. */
next_job = job_queue;
/* Удаляем задание из списка. */
job_queue = job_queue->next;
}

/* Освобождаем семафор, так как работа с очередью окончена. */
pthread_mutex_unlock(&job_queue_mutex);
/* Если очередь пуста, завершаем поток. */
if (next_job == NULL)
break;
/* Выполняем задание. */
process_job(next_job);
/* Очистка. */
free(next_job);
}
return NULL;
}

```

Все операции доступа к совместно используемому указателю `job_queue` происходят между вызовами функций `pthread_mutex_lock()` и `pthread_mutex_unlock()`. Объект задания, ссылка на который хранится в переменной `next_job`, обрабатывается только после того, как ссылка на него удаляется из очереди, что позволяет обезопасить этот объект от других потоков.

Обратите внимание на то, что, если очередь пуста (т.е. указатель `job_queue` равен `NULL`), цикл не завершается немедленно. Это привело бы к тому, что исключающий семафор так и остался бы в захваченном состоянии и не позволил бы ни одному другому потоку получить доступ к очереди заданий. Мы действуем иначе: записываем в переменную `next_job` значение `NULL` и выходим из цикла только после освобождения семафора.

Исключающий семафор блокирует доступ к участку программы, а не к переменной. В обязанности программиста входит написать код для захвата семафора перед доступом к переменной и последующего его освобождения. Вот как, например, может выглядеть функция, добавляющая новое задание к очереди:

```

void enqueue_job(struct job* new_job) {
    pthread_mutex_lock(&job_queue_mutex);
    new_job->next = job_queue;
    job_queue = new_job;
    pthread_mutex_unlock(&job_queue_mutex);
}

```

#### 4.4.3. Взаимоблокировки исключающих семафоров

Исключающие семафоры являются механизмом, позволяющим одному потоку блокировать выполнение другого потока. Это приводит к возникновению нового класса ошибок, называемых *взаимоблокировками* или *тупиковыми ситуациями*. Смысл ошибки в том, что один или несколько потоков ожидают наступления события, которое на самом деле никогда не произойдет.

Простейшая тупиковая ситуация когда один поток пытается захватить тот же самый исключающий семафор дважды подряд. Дальнейшие действия зависят от типа исключающего семафора. Их всего три.

- *Захват быстрого семафора* (используется по умолчанию) приведет к взаимоблокировке. Функция, обращающаяся к захваченному семафору данного типа, заблокирует поток до тех пор, пока семафор не будет освобожден. Но семафор принадлежит самому потоку, поэтому блокировка никогда не будет снята.

- *Захват рекурсивного семафора* не приведет к взаимоблокировке. Семафор данного типа запоминает, сколько раз функция `pthread_mutex_lock()` была вызвана в потоке, которому принадлежит семафор. Чтобы освободить семафор и позволить другим потокам обратиться к нему, необходимо аналогичное число раз вызвать функцию `pthread_mutex_unlock()`.

- Операционная система Linux обнаруживает попытку повторно захватить *контролирующий семафор* и сигнализирует об этом: при очередном вызове функции `pthread_mutex_lock()` возвращается код ошибки `EDEADLK`.

По умолчанию в Linux создается быстрый семафор. В двух других случаях требуется предварительно создать объект атрибутов семафора, объявив переменную типа `pthread_mutexattr_t` и передав указатель на нее функции `pthread_mutexattr_init()`. Затем нужно задать тип исключающего семафора с помощью функции `pthread_mutexattr_setkind_np()`. Первым ее аргументом является указатель на объект атрибутов семафора; второй аргумент равен `PTHREAD_MUTEX_RECURSIVE_NP` в случае рекурсивного семафора и `PTHREAD_MUTEX_ERRORCHECK_NP` в случае контролирующего семафора. Указатель на полученный объект атрибутов необходимо передать функции `pthread_mutex_init()`, которая создаст семафор. После этого нужно удалить объект атрибутов с помощью функции `pthread_mutexattr_destroy()`.

Следующий фрагмент программы иллюстрирует процесс создания контролирующего семафора:

```

pthread_mutexattr_t attr;
pthread_mutex_t mutex;

pthread_mutexattr_init(&attr);
pthread_mutexattr_setkind_np(&attr, PTHREAD_MUTEX_ERRORCHECK_NP);
pthread_mutex_init(&mutex, &attr);
pthread_mutexattr_destroy(&attr);

```

Как подсказывает префикс "np" (not portable), исключающие семафоры рекурсивного и контролирующего типов специфичны для Linux и непереносимы в другие операционные

системы. Поэтому не рекомендуется использовать их в программах широкого назначения.

#### 4.4.4. Неблокирующие проверки исключających семафоров

Иногда нужно, не заблокировав программу, проверить, захвачен ли исключающий семафор. Для потока не всегда приемлемо находиться в режиме пассивного ожидания, ведь за это время можно сделать много полезного! Функция `pthread_mutex_lock()` не возвращает значения до тех пор, пока семафор не будет освобожден, поэтому она нам не подходит.

То, что нам нужно, это функция `pthread_mutex_trylock()`. Если она обнаруживает, что семафор свободен, то захватывает его так же, как и функция `pthread_mutex_lock()`, возвращая при этом 0. Если же оказывается, что семафор уже захвачен другим потоком, функция `pthread_mutex_trylock()` не блокирует программу, а немедленно завершается, возвращая код ошибки `EBUSY`. "Права собственности" другого потока при этом не нарушаются. Можно попытаться захватить семафор позднее.

#### 4.4.5. Обычные потоковые семафоры

В предыдущем примере, в котором группа потоков обрабатывает задания из очереди, потоковая функция запрашивает задания до тех пор, пока очередь не опустеет, после чего поток завершается. Эта схема работает в том случае, когда все задания помещаются в очередь заранее или новые задания поступают по крайней мере так же часто, как их запрашивают потоки. Но если потоки начнут работать слишком быстро, очередь опустеет и потоки завершатся. Может оказаться, что задание поступило, а потоков, готовых его обработать, уже нет. Следовательно, необходим механизм, который позволит блокировать потоки в случае, когда очередь пуста, а новые задания еще не поступили.

Такой механизм называется *семафором*. Семафор это счетчик, используемый для синхронизации потоков. Операционная система гарантирует, что проверка и модификация значения семафора могут быть выполнены безопасно и не приведут к возникновению гонки.

Счетчик семафора является неотрицательным целым числом. Семафор поддерживает две базовые операции.

■ Операция *ожидания* (`wait`) уменьшает значение счетчика на единицу. Если счетчик уже равен нулю, операция блокируется до тех пор, пока значение счетчика не станет положительным (вследствие действий, выполняемых другими потоками). После снятия блокировки значение семафора уменьшается на единицу и операция завершается.

■ Операция *установки* (`post`) увеличивает значение счетчика на единицу. Если до этого счетчик был равен нулю и существовали потоки, заблокированные в операции ожидания данного семафора, один из них разблокируется и завершает свою операцию (т.е. счетчик семафора снова становится равным нулю).

В Linux имеются две немного отличающиеся реализации семафоров. Та, которую мы опишем ниже, соответствует стандарту POSIX. Такие семафоры применяются для организации взаимодействия потоков. Другая реализация, служащая целям межпроцессного взаимодействия, рассмотрена в разделе 5.2, "Семафоры для процессов". При работе с семафорами необходимо включить в программу файл `<semaphore.h>`.

Семафор представляется переменной типа `sem_t`. Семафор следует предварительно инициализировать с помощью функции `sem_init()`, передав ей указатель на переменную семафора. Второй параметр этой функции должен быть равен нулю,<sup>[14]</sup> а третий это начальное

значение счетчика семафора.

Чтобы выполнить операцию ожидания семафора, необходимо вызвать функцию `sem_wait()`. Функция `sem_post()` устанавливает семафор. Есть также функция `sem_trywait()`, реализующая операцию неблокирующего ожидания. Она напоминает функцию `pthread_mutex_trylock()`: если операция ожидания приведет к блокированию потока из-за того, что счетчик семафора равен нулю, функция немедленно завершается, возвращая код ошибки `EAGAIN`.

В Linux имеется функция `sem_getvalue()`, позволяющая узнать текущее значение счетчика семафора. Это значение помещается в переменную типа `int`, на которую ссылается второй аргумент функции. Не пытайтесь на основании данного значения определять, стоит ли выполнять операцию ожидания или установки, так как это может привести к возникновению гонки: другой поток способен изменить счетчик семафора между вызовами функции `sem_getvalue()` и какой-нибудь другой функции работы с семафором. Доверяйте только атомарным функциям `sem_wait()` и `sem_post()`.

Но вернемся к нашему примеру. Можно сделать так, чтобы с помощью семафора потоковая функция проверяла, сколько заданий находится в очереди. Измененная версия программы приведена в листинге 4.12.

#### ***Листинг 4.12. (job-queue3.c) Работа с очередью заданий с применением семафора***

```
#include <malloc.h>
#include <pthread.h>
#include <semaphore.h>

struct job {
    /* Ссылка на следующий элемент связанного списка. */
    struct job* next;

    /* Другие поля, описывающие требуемую операцию... */
};

/* Список отложенных заданий. */
struct job* job_queue;

/* Исключающий семафор, защищающий очередь. */
pthread_mutex_t job_queue_mutex =
PTHREAD_MUTEX_INITIALIZER;

/* Семафор, подсчитывающий число гаданий в очереди. */
sem_t job_queue_count;

/* Начальная инициализация очереди. */
void initialize_job_queue() {
    /* Вначале очередь пуста. */
    job_queue = NULL;
    /* Устанавливаем начальное значение счетчика семафора
    равным 0. */
    sem_init(&job_queue_count, 0, 0);
}

/* Обработка заданий до тех пор, пока очередь не опустеет. */
```

```

void* thread_function(void* arg) {
while (1) {
struct job* next_job;
/* Дожидаемся готовности семафора. Если его значение больше
нуля, значит, очередь не пуста; уменьшаем счетчик на 1.
В противном случае операция блокируется до тех пор, пока
в очереди не появится новое задание. */
sem_wait(&job_queue_count);
/* Захват исключающего семафора, защищающего очередь. */
pthread_mutex_lock(&job_queue_mutex);
/* Мы уже знаем, что очередь не пуста, поэтому без лишней
проверки запрашиваем новое задание. */
next_job = job_queue;
/* Удаляем задание из списка. */
job_queue = job_queue->next;
/* освобождаем исключающий семафор, так как работа с
очередью окончена. */
pthread_mutex_unlock(&job_queue_mutex);
/* Выполняем задание. */
process_job(next_job);
/* Очистка. */
free(next_job);
}
return NULL;
}

```

```

/* Добавление нового задания в начало очереди. */
void enqueue_job(/* Передача необходимых данных... */) {
struct job* new_job;

```

```

/* Выделение памяти для нового объекта задания. */
new_job = (struct job*)malloc(sizeof(struct job));
/* Заполнение остальных полей структуры JOB... */

```

```

/* Захватываем исключающий семафор, прежде чем обратиться
к очереди. */
pthread_mutex_lock(&job_queue_mutex);
/* Помещаем новое задание в начало очереди. */
new_job->next = job_queue;
job_queue = new_job;

```

```

/* Устанавливаем семафор, сообщая о том, что в очереди появилось
новое задание. Если есть потоки, заблокированные в ожидании
семафора, один из них будет разблокирован и
обработает задание. */
sem_post(&job_queue_count);

```

```

/* Освобождаем исключающий семафор. */
pthread_mutex_unlock(&job_queue_mutex);
}

```

Прежде чем извлекать задание из очереди, каждый поток дожидается семафора. Если счетчик семафора равен нулю, т.е. очередь пуста, поток блокируется до тех пор, пока в очереди не появится новое задание и счетчик не станет положительным.

Функция `enqueue_job()` добавляет новое задание в очередь. Подобно потоковой функции, она захватывает исключающий семафор, перед тем как обратиться к очереди. После добавления задания функция `enqueue_job()` устанавливает семафор, сообщая потокам о том, что задание доступно. В программе, показанной в листинге 4.12, потоки никогда не завершаются: если

задания не поступают в течение длительного времени, все потоки переводятся в режим блокирования функцией `sem_wait()`.

#### 4.4.6. Сигнальные (условные) переменные

Мы узнали, как с помощью исключающего семафора защитить переменную от одновременного доступа со стороны двух и более потоков и как посредством обычного семафора реализовать счетчик обращений, доступный нескольким потокам. *Сигнальная переменная* (называемая также *условной переменной*) это третий элемент синхронизации в Linux. Благодаря ему можно задавать более сложные условия выполнения потоков.

Предположим, требуется написать потоковую функцию, которая входит в бесконечный цикл, выполняя на каждой итерации какие-то действия. Но работа цикла должна контролироваться флагом: действие выполняется только в том случае, когда он установлен.

В листинге 4.13 показан вариант такой программы. На каждой итерации цикла потоковая функция проверяет, установлен ли флаг. Поскольку к флагу обращается сразу несколько потоков, он защищается исключающим семафором. Подобная реализация является корректной, но она неэффективна. Если флаг не установлен, потоковая функция будет впустую тратить ресурсы процессора, занимаясь бесцельными проверками флага, а также захватывая и освобождая семафор. На самом деле необходимо как-то перевести функцию в неактивный режим, пока какой-нибудь другой поток не установит этот флаг.

#### *Листинг 4.15. (spin-condvar.c) Простейшая реализация сигнальной переменной*

```
#include <pthread.h>

int thread_flag;
pthread_mutex_t thread_flag_mutex;

void initialize_flag() {
    pthread_mutex_init(&thread_flag_mutex, NULL);
    thread_flag = 0;
}

/* Если флаг установлен, многократно вызывается функция do_work().
В противном случае цикл работает вхолостую. */
void* thread_function(void* thread_arg) {
    while (1) {
        int flag_is_set;
        /* Защищаем флаг с помощью исключающего семафора. */
        pthread_mutex_lock(&thread_flag_mutex);
        flag_is_set = thread_flag;
        pthread_mutex_unlock(&thread_flag_mutex);
        if (flag_is_set)
            do_work();
        /* Если флаг не установлен, ничего не делаем. Просто переходим
на следующую итерацию цикла. */
    }
    return NULL;
}

/* Задаем значение флага равным FLAG_VALUE. */
```

```

void set_thread_flag(int flag_value) {
/* Защищаем флаг с помощью исключающего семафора. */
pthread_mutex_lock(&thread_flag_mutex);
thread_flag = flag_value;
pthread_mutex_unlock(&thread_flag_mutex);
}

```

Сигнальная переменная позволяет организовать такую проверку, при которой поток либо выполняется, либо блокируется. Как и в случае семафора, поток может ожидать сигнальную переменную. Поток А, находящийся в режиме ожидания, блокируется до тех пор, пока другой поток, Б, не просигнализирует об изменении состояния этой переменной. Сигнальная переменная не имеет внутреннего счетчика, что отличает ее от семафора. Поток А должен перейти в состояние ожидания до того, как поток Б пошлет сигнал. Если сигнал будет послан раньше, он окажется потерянным и поток А заблокируется, пока какой-нибудь поток не пошлет сигнал еще раз.

Вот как можно сделать предыдущую программу более эффективной.

- Функция `thread_function()` в цикле проверяет флаг. Если он не установлен, поток переходит в режим ожидания сигнальной переменной.

- Функция `set_thread_flag()` устанавливает флаг и сигнализирует об изменении условной переменной. Если функция `thread_function()` была заблокирована в ожидании сигнала, она разблокируется и снова проверяет флаг.

Но существует одна проблема: возникает гонка между операцией проверки флага и операцией сигнализирования или ожидания сигнала. Предположим, что функция `thread_function()` проверяет флаг и обнаруживает, что он не установлен. В этот момент планировщик Linux прерывает выполнение данного потока и активизирует главную программу. По стечению обстоятельств программа как раз находится в функции `set_thread_flag()`. Она устанавливает флаг и сигнализирует об изменении условной переменной. Но поскольку в данный момент нет потока, ожидающего получения этого сигнала (вспомните, что функция `thread_function()` была прервана перед тем, как перейти в режим ожидания), сигнал окажется потерянным. Когда Linux вновь активизирует дочерний поток, он начнет ждать сигнал, который, возможно, никогда больше не придет.

Чтобы избежать этой проблемы, необходимо одновременно захватить и флаг, и сигнальную переменную с помощью исключающего семафора. К счастью, в Linux это предусмотрено. Любая сигнальная переменная должна использоваться совместно с исключающим семафором для предотвращения состояния гонки. Наша потоковая функция должна следовать такому алгоритму:

- В цикле необходимо захватить исключающий семафор и прочитать значение флага.
- Если флаг установлен, нужно разблокировать семафор и выполнить требуемые действия.
- Если флаг не установлен, одновременно выполняются операции освобождения семафора и перехода в режим ожидания сигнала.

Вся суть заключена в третьем этапе, на котором Linux позволяет выполнить атомарную операцию освобождения исключающего семафора и перехода в режим ожидания сигнала. Вмешательство других потоков при этом не допускается.

Сигнальная переменная имеет тип `pthread_cond_t`. Не забывайте о том, что каждой такой переменной должен быть сопоставлен исключающий семафор. Ниже перечислены функции, предназначенные для работы с сигнальными переменными.

- Функция `pthread_cond_init()` инициализирует сигнальную переменную. Первый ее аргумент это указатель на объект типа `pthread_cond_t`. Вторым аргументом (указатель на объект атрибутов сигнальной переменной) игнорируется в Linux. Исключающий семафор должен



инициализироваться отдельно, как описывалось в разделе 4.4.2, "Исключающие семафоры".

■ Функция `pthread_cond_signal()` сигнализирует об изменении переменной. При этом разблокируется один из потоков, находящийся в ожидании сигнала. Если таких потоков нет, сигнал игнорируется. Аргументом функции является указатель на объект типа `pthread_cond_t`.

Похожая функция `pthread_cond_broadcast()` разблокирует все потоки, ожидающие данного сигнала.

■ Функция `pthread_cond_wait()` блокирует вызывающий ее поток до тех пор, пока не будет получен сигнал об изменении заданной переменной. Первым ее аргументом является указатель на объект типа `pthread_cond_t`. Второй аргумент это указатель на объект исключаящего семафора (тип `pthread_mutex_t`).

В момент вызова функции `pthread_cond_wait()` исключаящий семафор уже должен быть захвачен вызывающим потоком. Функция в рамках единой "транзакции" освобождает семафор и блокирует поток в ожидании сигнала. Когда поступает сигнал, функция разблокирует поток и автоматически захватывает семафор.

Перечисленные ниже этапы должны выполняться всякий раз, когда программа тем или иным способом меняет результат проверки условия, контролируемого сигнальной переменной (в нашей программе условие это значение флага):

1. Захватить исключаящий семафор, дополняющий сигнальную переменную.
2. Выполнить действие, включающее изменение результата проверки условия (в нашем случае установить флаг).
3. Послать сигнал (возможно, широковещательный) об изменении условия.
4. Освободить исключаящий семафор.

В листинге 4.14 показана измененная версия предыдущего примера, в которой на этот раз флаг защищается сигнальной переменной. Обратите внимание на то, что в функции `thread_function()` исключаящий семафор захватывается до того, как будет проверено значение переменной `thread_flag`. Захват автоматически снимается функцией `pthread_cond_wait()` перед тем, как поток оказывается заблокированным, и также автоматически восстанавливается по завершении функции:

#### ***Листинг 4.14. (condvar.c) Управление работой потока с помощью сигнальной переменной***

```
#include <pthread.h>

int thread_flag;
pthread_cond_t thread_flag_cv;
pthread_mutex_t thread_flag_mutex;

void initialize_flag() {
    /* Инициализация исключаящего семафора и сигнальной
    переменной. */
    pthread_mutex_init(&thread_flag_mutex, NULL);
    pthread_cond_init(&thread_flag_cv, NULL);
    /* Инициализация флага. */
    thread_flag = 0;
}

/* Если флаг установлен, многократно вызывается функция
do_work(). В противном случае поток блокируется. */
void* thread_function(void* thread_arg) {
```

```

/* Бесконечный цикл. */
while (1) {
/* Захватываем исключаящий семафор, прежде чем обращаться
к флагу. */
pthread_mutex_lock(&thread_flag_mutex);
while (!thread_flag)
/* Флаг сброшен. Ожидаем сигнала об изменении условной
переменной, указывающего на то, что флаг установлен.
При поступлении сигнала поток разблокируется и снова
проверяет флаг. */
pthread_cond_wait(&thread_flag_cv, &thread_flag_mutex);
/* При выходе из цикла освобождаем исключаящий семафор. */
pthread_mutex_unlock(&thread_flag_mutex);
/* Выполняем требуемые действия. */
do_work();
}
return NULL;
}

/* Задаем значение флага равным FLAG_VALUE. */
void set_thread_flag(int flag_value) {
/* Захватываем исключаящий семафор, прежде чем изменять
значение флага. */
pthread_mutex_lock(&thread_flag_mutex);
/* Устанавливаем флаг и посылаем сигнал функции
thread_function(), заблокированной в ожидании флага.
Правда, функция не сможет проверить флаг, пока
исключаящий семафор не будет освобожден. */
thread_flag = flag_value;
pthread_cond_signal(&thread_flag_cv);
/* освобождаем исключаящий семафор. */
pthread_mutex_unlock(&thread_flag_mutex);
}

```

Условие, контролируемое сигнальной переменной, может быть произвольно сложным. Но перед выполнением любой операции, способной повлиять на результат проверки условия, необходимо захватить исключаящий семафор, и только после этого можно посылать сигнал.

Сигнальная переменная может вообще не быть связана ни с каким условием, а служить лишь средством блокирования потока до тех пор, пока какой-нибудь другой поток не "разбудит" его. Для этой же цели может использоваться и семафор. Принципиальная разница между ними заключается в том, что семафор "запоминает" сигнал, даже если ни один поток в это время не был заблокирован, а сигнальная переменная регистрирует сигнал только в том случае, если его ожидает какой-то поток. Кроме того, семафор всегда разблокирует лишь один поток, тогда как с помощью функции `pthread_cond_broadcast()` можно разблокировать произвольное число потоков.

#### 4.4.7. Взаимоблокировки двух и более потоков

Взаимоблокировка происходит, когда два (или более) потока блокируются в ожидании события, наступление которого на самом деле зависит от действий одного из заблокированных потоков. Например, если поток А ожидает изменения сигнальной переменной, устанавливаемой в потоке Б, а поток Б, в свою очередь, ждет сигнала от потока А, возникает тупиковая ситуация. Ни один из потоков никогда не пошлет сигнал другому. Необходимо тщательно избегать таких ситуаций, потому что их очень трудно обнаруживать.

Чаще всего взаимоблокировка возникает, когда группа потоков пытается захватить один и тот же набор объектов. Рассмотрим, к примеру, программу, в которой два потока, выполняющих разные потоковые функции, должны захватить одни и те же два исключających семафора. Предположим, поток А захватывает сначала семафор 1, а затем семафор 2, в то время как поток Б захватывает семафоры в обратном порядке. Возможна достаточно неприятная ситуация, когда после захвата семафора 1 потоком А операционная система активизирует поток Б, который захватит поток 2. Далее оба потока окажутся заблокированными, так как им будет закрыт доступ к семафорам друг друга.

Это пример более общей проблемы взаимоблокировки, которая касается не только объектов синхронизации, таких как исключające семафоры, но и ряда других ресурсов, в частности блокировок файлов и устройств. Проблема возникает, когда потоки пытаются захватить один и тот же набор ресурсов, но в разной последовательности. Выход заключается в том, чтобы обеспечить согласованный протокол доступа к ресурсам во всех потоках.

## 4.5. Реализация потоков в Linux

Потоковые функции, соответствующие стандарту POSIX, реализованы в Linux не так, как в большинстве других версий UNIX. Суть в том, что в Linux потоки реализованы в виде процессов. Когда вызывается функция `pthread_create()`, операционная система на самом деле создает новый процесс, выполняющий поток. Но это не тот процесс, который создается функцией `fork()`. Он, в частности, делит общее адресное пространство и ресурсы с исходным процессом, а не получает их копии.

Сказанное иллюстрирует программа `thread-pid`, показанная в листинге 4.15. Она отображает идентификатор главного потока с помощью функции `getpid()` и создает новый поток, в котором тоже выводится значение идентификатора, после чего оба потока входят в бесконечный цикл.

### Листинг 4.15. (*thread-pid.c*) Вывод идентификаторов потоков

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

void* thread_function(void* arg) {
    fprintf(stderr, "child thread pid is %d\n", (int) getpid());
    /* Бесконечный цикл. */
    while (1);
    return NULL;
}

int main() {
    pthread_t thread;
    fprintf(stderr, "main thread pid is %d\n", (int) getpid());
    pthread_create(&thread, NULL, &thread_function, NULL);
    /* Бесконечный цикл. */
    while (1);
    return 0;
}
```

Запустите программу в фоновом режиме, а затем вызовите команду `ps` x, чтобы увидеть

список выполняющихся процессов. Не забудьте затем уничтожить программу thread-pid, так как она потребляет ресурсы процессора. Вот что мы получим:

```
% cc thread-pid.c -o thread-pid -lpthread
% ./thread-pid &
[1] 14608
main thread pid is 14608
child thread pid is 14610
% ps x
PID TTY STAT TIME COMMAND
14042 pts/9 S 0:00 bash
14068 pts/9 R 0:01 ./thread-pid
14069 pts/9 S 0:00 ./thread-pid
14610 pts/9 R 0:01 ./thread-pid
14611 pts/9 R 0:00 ps x
% kill 14608
[1]+ Terminated ./thread-pid
```

### *Сообщения интерпретатора команд касающиеся управления заданиями*

Строки, начинающиеся с записи [1], поступают от интерпретатора команд. Если программа запускается в фоновом режиме, интерпретатор назначает ей номер задания в данном случае 1 и сообщает ее идентификатор. Когда фоновое задание завершается, интерпретатор сообщает об этом при вызове первой же команды

Обратите внимание на то, что программе thread-pid соответствуют три процесса. Первый из них, с идентификатором 14608, это основной поток программы. Третий, с идентификатором 14610, это дочерний поток, выполняющий функцию thread\_function(). Что же такое тогда второй поток, с идентификатором 14609? Это "управляющий поток", являющийся частью внутреннего механизма реализации потоков в Linux. Он создается, когда программа вызывает функцию pthread\_create().

#### **4.5.1. Обработка сигналов**

Предположим, что многопоточковая программа принимает сигнал. В каком потоке будет вызван обработчик сигнала? Это зависит от версии UNIX. В Linux поведение программы объясняется тем, что потоки на самом деле реализуются в виде процессов.

Каждый поток в Linux является отдельным процессом, а сигнал доставляется конкретному процессу, поэтому никакой неоднозначности на самом деле нет. Обычно сигнал, поступающий от внешней программы, посылается процессу, управляющему главным потоком программы. Например, если программа с помощью функции fork() делится на два процесса и дочерний процесс запускает многопоточковую программу, в родительском процессе будет храниться идентификатор главного потока дочернего процесса, и этот идентификатор будет включаться во все сигналы, посылаемые от предка потомку. Этим правилом следует руководствоваться при написании многопоточковых программ для Linux.

Тем не менее подобная особенность реализации библиотеки Pthreads в Linux не согласуется со стандартом POSIX. Нельзя полагаться на нее в программах, рассчитанных на то, чтобы быть переносимыми.

В многопоточковой программе один поток может послать сигнал другому. Для этого предназначена функция pthread\_kill(). Ее первым параметром является идентификатор

потока, а второй параметр это номер сигнала.

### 4.5.2. Системный вызов `clone()`

Все потоки, создаваемые в одной программе, являются отдельными процессами, которые делят общее адресное пространство и другие ресурсы. Но дочерний процесс, создаваемый с помощью функции `fork()`, получает в свое распоряжение копии ресурсов. Как же реализуются процессы первого типа?

В Linux имеется функция `clone()`, являющаяся обобщением функций `fork()` и `pthread_create()`. Она позволяет вызывающему процессу указывать, какие ресурсы он согласен делить с дочерним процессом. Необходимо также задать область памяти, в которой будет расположен стек выполнения нового процесса. Вообще говоря, мы упоминаем функцию `clone()` лишь для того, чтобы удовлетворить любопытство читателей. Использовать ее в программах не следует. Создавайте процессы с помощью функции `fork()`, а потоки с помощью функции `pthread_create()`.

## 4.6. Сравнение процессов и потоков

В некоторых программах, связанных с параллельным выполнением операций, сделать выбор в пользу процессов или потоков может оказаться достаточно сложно. Приведем ряд правил, которые помогут читателям выбрать наилучшую модель для своих программ.

- Все потоки программы должны выполнять один и тот же код. В то же время дочерний процесс может запустить другой исполняемый файл с помощью функции `exec()`.

- Неправильно работающий поток способен помешать другим потокам того же процесса, поскольку все они используют одни и те же ресурсы. Например, неверное обращение к указателю может привести к искажению области памяти, используемой другим потоком. Процесс лишен возможности это делать, так как у него своя копия памяти,

- Копирование памяти, требуемой для дочернего процесса, приводит к снижению производительности процессов в сравнении с потоками. Но на самом деле операция копирования выполняется только тогда, когда содержимое памяти изменяется, поэтому снижение производительности оказывается минимальным, если дочерний процесс обращается к памяти только для чтения данных.

- Потоки требуются программам, в которых необходима тонкая настройка параллельной работы. Потоки, например, хорошо подходят в том случае, когда задание можно разбить на ряд почти идентичных задач. Процессы в основном работают не зависимо друг от друга.

- Совместное использование данных несколькими потоками тривиальная задача, ведь потоки имеют общий доступ к ресурсам (необходимо, правда, внимательно следить за тем, чтобы не возникало состояние гонки). В случае процессов требуется задействовать особый механизм взаимодействия, описанный в главе 5, "Взаимодействие процессов". Это делает программы более громоздкими, зато уменьшает вероятность ошибок, связанных с параллельной работой.

# Глава 5

## Взаимодействие процессов

В главе 3, "Процессы", описывалась процедура создания процесса и рассказывалось о том, как родительский процесс может получить код завершения дочернего процесса. Это простейшая форма взаимодействия двух процессов, но не самая эффективная. Рассмотренные в главе 3 механизмы позволяли процессу-предку общаться с процессом-потомком только посредством аргументов командной строки и переменных среды, а все, что мог сделать для предка потомок, вернуть свой код завершения. Такие механизмы не позволяют контролировать выполняющийся процесс или обращаться к внешнему, независимому процессу.

В этой главе будет показано, как обойти упомянутые ограничения путем организации взаимодействия процессов. Между собой могут общаться не только родительский и дочерний процессы, но также "неродственные" процессы и даже процессы, выполняющиеся на разных компьютерах.

*Взаимодействие процессов* это механизм обмена данными между процессами. Взяв, к примеру, ситуацию, когда браузер запрашивает Web-страницу у сервера, который в ответ высылает HTML-данные. Обычно при этом используются сокет, работающие через телефонное соединение. Или другой пример: пользователь вводит команду `ls | lpr`, чтобы вывести на печать список файлов в каталоге. Интерпретатор команд создает два отдельных процесса `ls` и `lpr` и соединяет их *каналом*, который представлен символом `'|'`. Канал это однонаправленный способ передачи данных от одного процесса к другому. Процесс `ls` записывает данные в канал, а процесс `lpr` читает данные из него.

В этой главе рассматриваются пять способов взаимодействия процессов.

- Совместно используемая память процессы могут просто читать и записывать данные в рамках заданной области памяти.

- Отображаемая память напоминает совместно используемую память, но организуется связь с файлами.

- Каналы позволяют последовательно передавать данные от одного процесса к другому.

- FIFO-файлы в отличие от каналов, с ними работают несвязанные процессы, поскольку у такого файла есть имя в файловой системе и к нему может обратиться любой процесс.

- Сокеты соединяют несвязанные процессы, работающие на разных компьютерах.

Различия между способами взаимодействия определяются следующими критериями:

- ограничено ли взаимодействие рамками связанных процессов (имеющих общего предка) или же соединяются процессы, выполняющиеся в одной файловой системе либо на разных компьютерах:

- ограничен ли процесс только чтением либо только записью данных;

- число взаимодействующих процессов;

- синхронизируются ли взаимодействующие процессы (например, должен ли читающий процесс перейти в режим ожидания при отсутствии данных на входе).

### 5.1. Совместно используемая память

Простейшим способом взаимодействия процессов является совместный доступ к общей области памяти. Это выглядит так, как если бы два или более процесса вызвали функцию `malloc()` и получили указатели на один и тот же блок памяти. Когда один из процессов меняет содержимое памяти, другие процессы замечают это изменение.

### 5.1.1. Быстрое локальное взаимодействие

Совместное использование памяти самый быстрый способ взаимодействия. Процесс обращается к общей памяти с той же скоростью, что и к своей собственной памяти, и никаких системных вызовов или обращений к ядру не требуется. Устраняется также ненужное копирование данных.

Ядро не синхронизирует доступ процессов к общей памяти об этом следует позаботиться программисту. Например, процесс не должен читать данные из совместно используемой памяти, пока в нее осуществляется запись, и два процесса не должны одновременно записывать данные в одну и ту же область памяти. Стандартная стратегия предотвращения подобной конкуренции заключается в использовании семафоров, о которых пойдет речь в разделе 5.2, "Семафоры для процессов". Тем не менее в приводимом далее примере программы доступ к памяти осуществляет только один процесс: просто мы хотим сконцентрировать внимание читателей на механизме совместного использования памяти и не перегружать программу кодом синхронизации.

### 5.1.2. Модель памяти

При совместном использовании сегмента памяти один процесс должен сначала выделить память. Затем все остальные процессы, которые хотят получить доступ к ней, должны подключить сегмент. По окончании работы с сегментом каждый процесс отключает его. Последний процесс освобождает память.

Для того чтобы понять принципы выделения и подключения сегментов памяти, необходимо разобраться в модели памяти Linux. В Linux виртуальная память (ВП) каждого процесса разбита на страницы. Все процессы хранят таблицу соответствий между своими адресами памяти и страницами ВП, содержащими реальные данные. Несмотря на то что за каждым процессом закреплены свои адреса, разным процессам разрешается ссылаться на одни и те же страницы. Это и есть совместное использование памяти.

При выделении совместно используемого сегмента памяти создаются страницы ВП. Это действие должно выполняться только один раз, так как все остальные процессы будут обращаться к этому же сегменту. Если запрашивается выделение существующего сегмента, новые страницы не создаются; вместо этого возвращается идентификатор существующих страниц. Чтобы сделать сегмент общедоступным, процесс подключает его, при этом создаются адресные ссылки на страницы сегмента. По окончании работы с сегментом адресные ссылки удаляются. Когда все процессы завершили работу с сегментом, один (и только один) из них должен освободить страницы виртуальной памяти.

Размер совместно используемого сегмента кратен *размеру страницы* ВП. В Linux последняя величина обычно равна 4 Кбайт, но никогда не помешает это проверить с помощью функции `getpagesize()`.

### 5.1.3. Выделение сегментов памяти

Процесс выделяет сегмент памяти с помощью функции `shmget()`. Первым аргументом функции является целочисленный ключ, идентифицирующий создаваемый сегмент. Если несвязанные процессы хотят получить доступ к одному и тому же сегменту, они должны указать одинаковый ключ. К сожалению, ничто не мешает посторонним процессам выбрать тот же

самый ключ сегмента, а это приведет к системному конфликту. Указание специальной константы `IPC_PRIVATE` в качестве ключа позволяет гарантировать, что будет создан совершенно новый сегмент.

Во втором аргументе функции задается размер сегмента в байтах. Это значение округляется, чтобы быть кратным размеру страницы ВП.

Третий параметр содержит набор битовых флагов. Перечислим наиболее важные из них.

- `IPC_CREAT`. Указывает на то, что создается новый сегмент, которому присваивается заданный ключ.

- `IPC_EXCL`. Всегда используется совместно с флагом `IPC_CREAT` и заставляет функцию `shmget()` выдать ошибку в случае, когда сегмент с указанным ключом уже существует. Если флаг не указан и возникает описанная ситуация, функция `shmget()` возвращает идентификатор существующего сегмента, не создавая новый сегмент.

- *Флаги режима*. В эту группу входят 9 флагов, задающих права доступа к сегменту для владельца, группы и остальных пользователей. Биты выполнения игнорируются. Проще всего задавать права доступа с помощью констант, определенных в файле `<sys/stat.h>` (они описаны на man-странице функции `stat()`).<sup>[15]</sup> Например, флаги `S_IRUSR` и `S_IWUSR` предоставляют право чтения и записи владельцу сегмента, а флаги `S_IROTH` и `S_IWOTH` предоставляют аналогичные права остальным пользователям.

В следующем фрагменте программы функция `shmget()` создает новый совместно используемый сегмент памяти (или возвращает идентификатор существующего, если значение `shm_key` уже зарегистрировано в системе), доступный для чтения/записи только его владельцу:

```
int segment_id = shmget(shm_key, getpagesize(),  
IPC_CREAT | S_IRUSR | S_IWUSR);
```

В случае успешного завершения функция возвращает идентификатор сегмента. Если сегмент уже существует, проверяются права доступа к нему.

#### 5.1.4. Подключение и отключение сегментов

Чтобы сделать сегмент памяти общедоступным, процесс должен подключить его с помощью функции `shmat()`. В первом ее аргументе передается идентификатор сегмента, возвращенный функцией `shmget()`. Второй аргумент это указатель, определяющий, где в адресном пространстве процесса необходимо создать привязку на совместно используемую область памяти. Если задать значение `NULL`, ОС Linux выберет первый доступный адрес. Третий аргумент может содержать следующие флаги.

- `SHM_RND`. Указывает на то, что адрес, заданный во втором параметре, должен быть округлен, чтобы стать кратным размеру страницы. Если этот флаг не указан, необходимо самостоятельно позаботиться о выравнивании сегмента по границе страницы.

- `SHM_RDONLY`. Указывает на то, что сегмент доступен только для чтения, но не для записи.

В случае успешного завершения функция возвращает адрес подключенного сегмента. Дочерний процесс, созданный функцией `fork()`, унаследует этот адрес и в случае необходимости сможет отключить сегмент.

По завершении работы с сегментом его необходимо отключить с помощью функции `shmdt()`. Ей следует передать адрес, возвращаемый функцией `shmat()`. Если текущий процесс был последним, кто ссылался на сегмент, сегмент удаляется из памяти. Функции `exit()` и `_exit()` автоматически отключают сегменты.

#### 5.1.5. Контроль и освобождение совместно используемой памяти



Функция `shmctl()` возвращает информацию о совместно используемом сегменте и способна модифицировать его. Первым параметром является идентификатор сегмента.

Чтобы получить информацию о сегменте, укажите в качестве второго параметра константу `IPC_STAT`, а в третьем параметре передайте указатель на структуру `shmid_ds`.

Чтобы удалить сегмент, передайте во втором параметре константу `IPC_RMID`, а в третьем параметре `NULL`. Сегмент удаляется, когда последний подключивший его процесс отключает сегмент.

Каждый совместно используемый сегмент должен явно освобождаться с помощью функции `shmctl()`, чтобы случайно не был превышен системный лимит на общее число таких сегментов. Функции `exit()` и `exec()` отключают сегменты, но не освобождают их.

Описание других операций, выполняемых над совместно используемыми сегментами памяти, можно найти на [man-странице](#) функции `shmctl()`.

5.1.6. Пример программы

Программа, приведенная в листинге 5.1, иллюстрирует методику совместного использования памяти.

Листинг 5.1. (*shm.c*) Пример совместного использования памяти

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main() {
    int segment_id;
    char* shared_memory;
    struct shmid_ds shmbuffer;
    int segment_size;
    const int shared_segment_size = 0x6400;

    /* Выделение совместно используемого сегмента. */
    segment_id =
    shmget(IPC_PRIVATE, shared_segment_size,
    IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);
    /* Подключение сегмента. */
    shared_memory = (char*)shmat(segment_id, 0, 0);
    printf("shared memory attached at address %p\n",
    shared_memory);
    /* Определение размера сегмента. */
    shmctl(segment_id, IPC_STAT, &shmbuffer);
    segment_size = shmbuffer.shm_segsz;
    printf("segment size: %d\n", segment_size);
    /* Запись строки в сегмент. */
    sprintf(shared_memory, "Hello, world.");
    /* Отключение сегмента. */
    shmdt(shared_memory);

    /* Повторное подключение сегмента, но по другому адресу! */
    shared_memory =
    (char*)shmat(segment_id, (void*) 0x50000000, 0);
    printf("shared memory reattached at address %p\n",
```

```

shared_memory);
/* Отображение строки, хранящейся в совместно используемой
памяти. */
printf("%s\n", shared_memory);
/* Отключение сегмента. */
shmdt(shared_memory);
/* Освобождение сегмента. */
shmctl(segment_id, IPC_RMID, 0);
return 0;
}

```

### 5.1.7. Отладка

Команда `ipcs` выдает информацию о взаимодействии процессов, включая сведения о совместно используемых сегментах (для этого следует задать флаг `-m`). Например, в показанном ниже случае сообщается о том, что используется один такой сегмент, с номером 1627649:

```

% ipcs -m
----- Shared Memory Segments -----
key shmid owner perms bytes nattch status
0x00000000 1627649 user 640 25600 0

```

Если этот сегмент был по ошибке "забыт" какой-то программой, его можно удалить с помощью команды `ipcrm`:

```

% ipcrm shm 1627649

```

### 5.1.8. Проблема выбора

Благодаря совместному использованию памяти можно организовать быстрое двустороннее взаимодействие произвольного числа процессов. Любой пользователь сможет получать доступ к сегментам памяти для чтения/записи, но для этого программа должна следовать определенным правилам, позволяющим избегать конкуренции (чтобы, например, информация не оказалась перезаписанной до того, как будет прочитана). К сожалению, Linux не гарантирует монопольный доступ к сегменту, даже если он был создан с указанием флага `IPC_PRIVATE`.

Кроме того, чтобы несколько процессов могли совместно работать с общим сегментом, они должны "договориться" о выборе одинакового ключа.

## 5.2. Семафоры для процессов

Как говорилось в предыдущем разделе, процессы должны координировать свои усилия при совместном доступе к памяти. Вспомните: в разделе 4.4.5, "Обычные потоковые семафоры", рассказывалось о семафорах, которые являются счетчиками, позволяющими синхронизировать работу потоков. В Linux имеется альтернативная реализация семафоров (иногда называемых семафорами System V), предназначенных для синхронизации процессов. Такие семафоры выделяются, используются и освобождаются подобно совместно используемым сегментам памяти. Для большинства случаев достаточно одного семафора, тем не менее они работают группами. В этом разделе мы опишем системные вызовы, позволяющие реализовать двоичный семафор.

### 5.2.1. Выделение и освобождение семафоров

Функции `semget()` и `semctl()` выделяют и освобождают семафоры, функционируя подобно функциям `shmget()` и `shmctl()`. Первым аргументом функции `semget()` является ключ, идентифицирующий группу семафоров; второй аргумент это число семафоров в группе; третий аргумент флаги прав доступа, как в функции `shmget()`. Функция `semget()` возвращает идентификатор группы семафоров. Если задан ключ, принадлежащий существующей группе, будет возвращен ее идентификатор. В этом случае второй аргумент (число семафоров) может равняться нулю.

Семафоры продолжают существовать даже после того, как все работавшие с ними процессы завершились. Чтобы система не исчерпала лимит семафоров, последний процесс должен явно удалить группу семафоров. Для этого нужно вызвать функцию `semctl()`, передав ей идентификатор группы, число семафоров в группе, флаг `IPC_RMID` и произвольное значение типа `union semun` (оно игнорируется). Значение `EUID` (эффективный идентификатор пользователя) процесса, вызвавшего функцию, должно совпадать с аналогичным значением процесса, создавшего группу семафоров (либо вызывающий процесс должен быть запущен пользователем `root`). В отличие от совместно используемых сегментов памяти, удаляемая группа семафоров немедленно освобождается.

В листинге 5.2 представлены функции, выделяющие и освобождающие двоичный семафор.

### ***Листинг 5.2. (sem\_all\_deall.c) Выделение и освобождение двоичного семафора***

```
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/types.h>

/* Тип union semun необходимо определить самостоятельно. */
union semun {
    int val;
    struct semid_ds *buf;
    unsigned short int* array;
    struct seminfo *__buf;
};

/* Получаем идентификатор семафора и создаем семафор,
если идентификатор оказывается уникальным. */
int binary_semaphore_allocation(key_t key, int sem_flags) {
    return semget(key, 1, sem_flags);
}

/* Освобождаем семафор, подразумевая, что пользователи
больше не работают с ним. В случае ошибки
возвращается -1. */
int binary_semaphore_deallocate(int semid) {
    union semun ignored_argument;
    return semctl(semid, 1, IPC_RMID, ignored_argument)
}
```

## **5.2.2. Инициализация семафоров**

Выделение и инициализация семафора две разные операции. Чтобы проинициализировать семафор, вызовите функцию `semctl()`, задав второй аргумент равным нулю, а третий аргумент

равным константе SETALL. Четвертый аргумент должен иметь тип union semun, поле array которого указывает на массив значений типа unsigned short. Каждое значение инициализирует один семафор из набора.

В листинге 5.3 представлена функция, инициализирующая двоичный семафор.

### **Листинг 5.3. (sem\_init.c) Инициализация двоичного семафора**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

/* Тип union semun необходимо определить самостоятельно. */
union semun {
    int val;
    struct semid_ds* buf;
    unsigned short int *array;
    struct seminfo *__buf;
};

/* Инициализация двоичного семафора значением 1. */
int binary_semaphore_initialize(int semid) {
    union semun argument;
    unsigned short values(1);
    values[0] = 1;
    argument.array = values;
    return semctl(semid, 0, SETALL, argument);
}
```

### **5.2.3. Операции ожидания и установки**

Каждый семафор имеет неотрицательное значение и поддерживает операции ожидания и установки. Системный вызов semop() реализует обе операции. Первым аргументом функции является идентификатор группы семафоров. Вторым аргументом это массив значений типа struct sembuf, задающих выполняемые операции. Третий аргумент это длина массива.

Ниже перечислены поля структуры sembuf.

- sem\_num номер семафора в группе.
- sem\_op число, задающее операцию.

Если данное поле содержит положительное число, оно немедленно добавляется к значению семафора.

Если данное поле содержит отрицательное число, то модуль числа вычитается из значения семафора. Операции, приводящие к установке отрицательного значения, блокируются до тех пор, пока значение семафора не станет достаточно большим (вследствие действий других процессов).

Если данное поле равно нулю, операция блокируется до тех пор, пока значение семафора не станет равным нулю.

■ sem\_flg это значение флага. Флаг IPC\_NOWAIT предотвращает блокирование операции. Если запрашиваемая операция приведет к блокированию, функция semop() завершится выдачей кода ошибки. При наличии флага SEM\_UNDO ОС Linux автоматически отменит выполненную операцию по завершении процесса.

В листинге 5.4 иллюстрируются операции ожидания и установки двоичного семафора.

#### **Листинг 5.4. (sem\_pv.c) Ожидание и установка двоичного семафора**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

/* Ожидание семафора. Операция блокируется до тех пор, пока
значение семафора не станет положительным, после чего
значение уменьшается на единицу. */
int binary_semaphore_wait(int semid) {
    struct sembuf operations[1];
    /* Оперировать одним-единственным семафором. */
    operations[0].sem_num = 0;
    /* Уменьшаем его значение на единицу. */
    operations[0].sem_op = -1;
    /* Разрешаем отмену операции. */
    operations[0].sem_flg = SEM_UNDO;
    return semop(semid, operations, 1);
}

/* Установка семафора: его значение увеличивается на единицу.
Эта операция завершается немедленно. */
int binary_semaphore_post(int semid) {
    struct sembuf operations[1];
    /* оперировать одним-единственным семафором. */
    operations[0].sem_num = 0;
    /* Увеличиваем его значение на единицу. */
    operations[0].sem_op = 1;
    /* Разрешаем отмену операции. */
    operations[0].sem_flg = SEM_UNDO;
    return semop(semid, operations, 1);
}
```

Флаг SEM\_UNDO позволяет решить проблему, возникающую при завершении процесса, которого есть ресурсы, связанные с семафором. Как бы ни завершился процесс принудительно или естественным образом, значение семафора автоматически корректируется. "отменяя" эффект операции, выполненной над семафором. Например, если процесс уменьшил значение семафора, а затем был уничтожен командой kill, значение семафора будет снова увеличено.

#### **5.2.4. Отладка семафоров**

С помощью команды `ipcs -s` можно получить информацию о существующих группах семафоров. Команда `ipcrm sem` позволяет удалить заданную группу, например:

```
% ipcrm sem 5790517
```

### **5.3. Отображение файлов в памяти**

Благодаря механизму отображаемой памяти процессы получают возможность общаться друг с другом посредством совместно используемого файла. Схематически это можно представить как совместный доступ к именованному сегменту памяти, хотя технически оба механизма реализованы по-разному.

При отображении файла в памяти формируется связь между файлом и памятью процесса. ОС Linux разбивает файл на страничные блоки и копирует их в страницы виртуальной памяти, чтобы они стали доступны в адресном пространстве процесса. Таким образом, процесс сможет обращаться к содержимому файла как к обычной памяти. При записи данных в соответствующую область памяти содержимое файла будет меняться. Это ускоряет доступ к файлам.

Отображаемую память можно представить как буфер, в который загружается все содержимое файла. Если данные, находящиеся в буфере, модифицируются, они записываются обратно в файл. Операции чтения и записи ОС Linux обрабатывает самостоятельно.

Файлы, отображаемые в памяти, можно использовать не только для организации взаимодействия процессов. О других применениях таких файлов пойдет речь в разделе 5.3.5. "Другие применения функции `mmap()`".

### 5.3.1. Отображение в памяти обычного файла

Для отображения обычного файла в памяти процесса предназначена функция `mmap()`. Ее первым аргументом является адрес, который будет соответствовать началу отображаемого файла в адресном пространстве процесса. Если задать значение `NULL`, ОС Linux выберет первый доступный адрес. Вторым аргументом это длина отображаемой области в байтах. Третий аргумент задает степень защиты диапазона отображаемых адресов. Он может содержать объединение битовых констант `PROT_READ`, `PROT_WRITE` и `PROT_EXEC`, соответствующих разрешению на чтение, запись и выполнение соответственно. Четвертый аргумент содержит дополнительные флаги. Пятый аргумент это дескриптор открытого файла. В последнем аргументе задается смещение от начала файла, с которого начинается отображаемая область. Можно перенести в память весь файл или только часть его, должным образом корректируя начальное смещение и длину отображаемой области.

Ниже перечислены дополнительные флаги, задаваемые в четвертом аргументе.

■ **`MAP_FIXED`**. При наличии этого флага ОС Linux использует значение первого аргумента как точный адрес размещения отображаемого файла. Этот адрес должен соответствовать началу страницы.

■ **`MAP_PRIVATE`**. Изменения, вносимые в отображаемую память, записываются не в присоединенный файл, а в частную копию файла, принадлежащую процессу. Другие процессы не узнают об этих изменениях. Данный режим не совместим с режимом `MAP_SHARED`.

■ **`MAP_SHARED`**. Изменения, вносимые в отображаемую память, немедленно фиксируются в файле, минуя буфер. Этот режим используется при организации взаимодействия процессов и не совместим с режимом `MAP_PRIVATE`.

При успешном завершении функция возвращает указатель на начало области памяти. В противном случае возвращается флаг `MAP_FAILED`.

По окончании работы с отображаемым файлом его необходимо освободить с помощью функции `munmap()`. Ей передается начальный адрес и длина отображаемой области. ОС Linux автоматически освобождает отображаемые области при завершении процесса.

### 5.3.2. Примеры программ

В этом разделе рассматриваются две программы, в которых иллюстрируются чтение и запись файлов, отображаемых в памяти. Первая программа (листинг 5.5) генерирует случайное

число и записывает его в отображаемый файл. Вторая программа (листинг 5.6) читает число из файла, выводит его на экран, а затем умножает на 2 и записывает обратно в файл. Обе программы принимают имя файла из командной строки.

### ***Листинг 5.5. (mmap-write.c) Запись случайного числа в файл, отображаемый в памяти***

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <time.h>
#include <unistd.h>
#define FILE_LENGTH 0x100

/* получение случайного числа в диапазоне [low,high]. */
int random_range(unsigned const low, unsigned const high) {
    unsigned const range = high - low + 1;
    return
    low + (int)((((double)range) * rand() / (RAND_MAX + 1.0)));
}

int main (int argc, char* const argv[]) {
    int fd;
    void* file_memory;

    /* Инициализация генератора случайных чисел. */
    srand(time(NULL));

    /* подготовка файла, размер которого будет достаточен для
    записи беззнакового целого числа. */
    fd = open(argv[1], O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
    lseek(fd, FILE_LENGTH+1, SEEK_SET);
    write(fd, "", 1);
    lseek(fd, 0, SEEK_SET);

    /* Создание отображаемой области. */
    file_memory =
    mmap(0, FILE_LENGTH, PROT_WRITE, MAP_SHARED, fd, 0);
    close(fd);
    /* Запись случайного числа в отображаемую память. */
    sprintf((char*)file_memory,
    "%d\n", random_range(-100, 100));
    /* Освобождение памяти (не обязательно, так как программа
    завершается). */
    munmap(file_memory, FILE_LENGTH);
    return 0;
}
```

Программа `mmap-write` пытается открыть файл и, если он не существует, создает его. Третий аргумент функции `open()` указывает на то, что файл доступен для чтения/записи. Поскольку длина файла неизвестна, с помощью функции `lseek()` мы убеждаемся в том, что файл имеет достаточную длину для записи беззнакового целого числа, а затем возвращаемся в начало файла.

Программа закрепляет файл за областью памяти и закрывает его дескриптор, так как в нем

больше нет необходимости. После этого программа записывает случайное число в отображаемую память, т.е. в файл, и освобождает память. В принципе, вызывать функцию `mmap()` нет необходимости, так как ОС Linux автоматически освободит память при завершении программы.

#### ***Листинг 5.6. (mmap-read.c) Чтение случайного числа из файла, отображаемого в памяти***

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>
#define FILE_LENGTH 0x100

int main(int argc, char* const argv[]) {
    int fd;
    void* file_memory;
    int integer;

    /* Открытие файла. */
    fd = open(argv[1], O_RDWR, S_IRUSR | S_IWUSR);
    /* Создание отображаемой области. */
    file_memory =
        mmap(0, FILE_LENGTH, PROT_READ | PROT_WRITE,
            MAP_SHARED, fd, 0);
    close(fd);

    /* Чтение целого числа и вывод его на экран. */
    sscanf(file_memory, "%d", &integer);
    printf("value: %d\n", integer);
    /* Удваиваем число и записываем его обратно в файл. */
    sprintf((char*)file_memory, "%d\n", 2 * integer);
    /* Освобождение памяти (не обязательно, так как программа
    завершается). */
    munmap(file_memory, FILE_LENGTH);
    return 0;
}
```

Программа `mmap-read` читает число из файла, а затем удваивает его и записывает обратно в файл. Сначала файл открывается для чтения/записи. Поскольку предполагается, что файл содержит число, проверка с помощью функции `lseek()`, как в предыдущей программе, не требуется. Чтение содержимого памяти и его анализ выполняет функция `lseek()`. Функция `sprintf()` форматирует число и записывает его в память.

Ниже показан пример запуска обеих программ. Им на вход передается файл `/tmp/integer-file`.

```
% ./mmap-write /tmp/integer-file
% cat /tmp/integer-file
42
% ./mmap-read /tmp/integer-file
value: 42
% cat /tmp/integer-file
84
```

Обратите внимание: значение 42 оказалось записано в файл на диске, хотя функция



`write()` не вызывалась. Последующее чтение файла осуществлялось без функции `read()`. Целое число записывалось в файл и извлекалось из него в текстовом виде (с помощью функций `sprintf()` и `scanf()`). Это сделано исключительно в демонстрационных целях. В действительности отображаемый файл может содержать не только текст, но и двоичные данные.

### 5.3.3. Совместный доступ к файлу

Процессы могут взаимодействовать друг с другом через области отображаемой памяти, связанные с одним и тем же файлом. Если в функции `mmap()` указать флаг `MAP_SHARED`, все данные, заносимые в отображаемую память, будут немедленно записываться в файл, т.е. становиться видимыми другим процессам. При отсутствии этого флага ОС Linux может осуществлять предварительную буферизацию записываемых данных.

С другой стороны, с помощью функции `msync()` можно заставить операционную систему перенести содержимое буфера в дисковый файл. Первые два параметра этой функции такие же, как и в функции `mmap()`. Третий параметр может содержать следующие флаги.

- **MS\_ASYNC.** Операция обновления ставится в очередь планировщика и будет выполнена, но не обязательно до того, как функция завершится.

- **MS\_SYNC.** Операция обновления выполняется немедленно. До ее завершения функция блокируется. Флаги `MS_ASYNC` и `MS_SYNC` нельзя указывать одновременно.

- **MS\_INVALIDATE.** Все остальные отображаемые области помечаются как недействительные и подлежащие обновлению.

Следующая функция обновляет файл, область отображения которого начинается с адреса `mem_addr` и имеет длину `mem_length`:

```
msync(mem_addr, mem_length, MS_SYNC | MS_INVALIDATE);
```

Как и в случае совместного использования сегментов памяти, при работе с отображаемыми областями необходимо придерживаться определенного порядка во избежание конкуренции. Например, можно создать семафор, который позволит только одному процессу обращаться к отображаемой памяти в конкретный момент времени. Можно также воспользоваться функцией `fcntl()` и поставить на файл блокировку чтения или записи (об этом рассказывается в разделе 8.3, "Функция `fcntl()`: блокировки и другие операции над файлами").

### 5.3.4. Частные отображаемые области

Если в функции `mmap()` указан флаг `MAP_PRIVATE`, отображаемая область создается в режиме "копирование при записи". Любые операции записи в эту область имеют эффект только в адресном пространстве текущего процесса. Другие процессы, связанные с тем же самым отображаемым файлом, не узнают об изменениях. Изменения заносятся не на страницу, доступную всем процессам, а в частную копию этой страницы. Все последующие операции чтения и записи в данном процессе будут выполняться по отношению к этой копии.

### 5.3.5. Применения функции `mmap()`

Функция `mmap()` может использоваться не только для организации взаимодействия процессов. Часто она выступает в качестве замены функциям `read()` и `write()`. Например, вместо того чтобы непосредственно загружать содержимое файла в память, программа может связать файл с отображаемой памятью и сканировать его путем обращения к памяти. Иногда это

удобнее и быстрее, чем выполнять операции файлового ввода-вывода.

Некоторые программы формируют в отображаемом файле структуры данных. При каждом следующем запуске программа повторно инициализирует файл в памяти, вследствие чего восстанавливается начальное состояние структур. В подобной ситуации следует помнить о том, что указатели на структуры будут некорректными, если они не локализованы в пределах одной отображаемой области и если файл не загружается по одному и тому же адресу.

Другой удобный прием отображение в памяти файла `/dev/zero` (описывается в разделе 6.5.2, `"/dev/zero"`). Этот файл ведет себя так, как будто содержит бесконечное число нулевых байтов. Операции записи в него игнорируются. Описываемый прием часто применяется в пользовательских функциях выделения памяти, которым необходимо инициализировать блоки памяти.

## 5.4. Каналы

*Канал* это коммуникационное устройство, допускающее однонаправленное взаимодействие. Данные, записываемые на "входном" конце канала, читаются на "выходном" его конце. Каналы являются последовательными устройствами: данные всегда читаются в том порядке, в котором они были записаны. Канал обычно используется как средство связи между двумя потоками одного процесса или между родительским и дочерним процессами.

В интерпретаторе команд канал создается оператором `|`. Например, показанная ниже команда заставляет интерпретатор запустить два дочерних процесса, один для программы `ls`, а второй для программы `less`:

```
% ls | less
```

Интерпретатор также формирует канал, соединяющий стандартный выходной поток подпроцесса `ls` со стандартным входным потоком подпроцесса `less`. Таким образом, имена файлов, перечисляемые программой `ls`, посылаются программе постраничной разбивки `less` в том порядке, в котором они отображались бы нетерминале.

Информационная емкость канала ограничена. Если пишущий процесс помещает данные в канал быстрее, чем читающий процесс их извлекает, и буфер канала переполняется, то пишущий процесс блокируется до тех пор, пока буфер не освободится. И наоборот: если читающий процесс обращается к каналу, в который еще не успели поступить данные, он блокируется в ожидании данных. Таким образом, канал автоматически синхронизирует оба процесса.

### 5.4.1. Создание каналов

Канал создается с помощью функции `pipe()`. Ей необходимо передать массив из двух целых чисел. В элементе с индексом 0 функция сохраняет дескриптор файла, соответствующего выходному концу канала, а в элементе с индексом 1 сохраняется дескриптор файла, соответствующего входному концу канала. Рассмотрим следующий фрагмент программы

```
int pipe_fds[2];
int read_fd;
int write_fd;
```

```
pipe(pipe_fds);
read_fd = pipe_fds[0];
write_fd = pipe_fds[1];
```

Данные, записываемые в файл `write_fd`, могут быть прочитаны из файла `read_fd`.

## 5.4.2. Взаимодействие родительского и дочернего процессов

Функция `pipe()` создает два файловых дескриптора, которые действительны только в текущем процессе и его потомках. Эти дескрипторы нельзя передать постороннему процессу. Дочерний процесс получает копии дескрипторов после завершения функции `fork()`.

В программе, показанной в листинге 5.7. родительский процесс записывает в канал строку, а дочерний процесс читает ее. С помощью функции `fdopen()` файловые дескрипторы приводятся к типу `FILE*`. Благодаря этому появляется возможность использовать высокоуровневые функции ввода-вывода, такие как `printf()` и `fgets()`.

### *Листинг 5.7. (pipe.c) Общение с дочерним процессом посредством канала*

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

/* Запись указанного числа копий (COUNT) сообщения (MESSAGE)
в поток (STREAM) с паузой между каждой операцией. */
void writer(const char* message, int count, FILE* stream) {
    for (; count > 0; --count) {
        /* Запись сообщения в поток с немедленным "выталкиванием"
из буфера. */
        fprintf(stream, "%s\n", message);
        fflush(stream);
        /* Небольшая пауза. */
        sleep(1);
    }
}

/* Чтение строк из потока, пока он не опустеет. */
void reader(FILE* stream) {
    char buffer[1024];
    /* Чтение данных, пока не будет обнаружен конец потока.
Функция fgets() завершается, когда встречает символ
новой строки или признак конца файла. */
    while (!feof(stream)
        && !ferror(stream)
        && fgets(buffer, sizeof (buffer), stream) != NULL)
        fputs(buffer, stdout);
}

int main() {
    int fds[2];
    pid_t pid;

    /* Создание канала. Дескрипторы обоих концов канала
помещаются в массив FDS. */
    pipe(fds);
    /* порождение дочернего процесса. */
    pid = fork();
    if (pid == (pid_t)0) {
        FILE* stream;
        /* Это дочерний процесс. Закрываем копию входного конца
канала. */
        close(fds[1]);
```

```

/* Приводим дескриптор выходного конца канала к типу FILE*
и читаем данные из канала. */
stream = fdopen(fds[0], "r");
reader(stream);
close(fds[0]);
} else {
/* Это родительский процесс. */
FILE* stream;
/* Закрываем копию выходного конца канала. */
close(fds[0]);
/* Приводим дескриптор входного конца канала к типу FILE*
и записываем данные в канал. */
stream = fdopen(fds[1], "w");
writer("Hello, world.", 5, stream);
close(fds[1]);
}
return 0;
}

```

Сначала в программе объявляется массив `fds`, состоящий из двух целых чисел. Функция `pipe()` создает канал и помещает в массив дескрипторы входного и выходного концов канала. Затем функция `fork()` порождает дочерний процесс. После закрытия выходного конца канала родительский процесс начинает записывать строки в канал. Дочерний процесс читает строки из канала, предварительно закрыв его входной конец.

Обратите внимание на то, что в функции `writer()` родительский процесс принудительно "выталкивает" буфер канала, вызывая функцию `fflush()`. Без этого строка могла бы ""застыть" в буфере и отправиться в канал только после завершения родительского процесса.

При вызове команды `ls | less` функция `fork()` выполняется дважды: один раз для дочернего процесса `ls`, второй раз для дочернего процесса `less`. Оба процесса наследуют копии дескрипторов канала, поэтому могут общаться друг с другом. О соединении несвязанных процессов речь пойдет ниже, в разделе 5.4.5, "Каналы FIFO".

### 5.4.3. Перенаправление стандартных потоков ввода, вывода и ошибок

Часто требуется создать дочерний процесс и сделать один из концов канала его стандартным входным или выходным потоком. В этом случае на помощь приходит функция `dup2()`, которая делает один файловый дескриптор равным другому. Вот как, например, можно связать стандартный входной поток с файлом `fd`:

```
dup2(fd, STDIN_FILENO);
```

Символическая константа `STDIN_FILENO` представляет дескриптор файла, соответствующий стандартному потоку ввода (значение этого дескриптора равно 0). Показанная функция закрывает входной поток, а затем открывает его под видом файла `fd`. Оба дескриптора (0 и `fd`) будут указывать на одну и ту же позицию в файле и иметь одинаковый набор флагов состояния, т.е. дескрипторы станут взаимозаменяемыми.

Программа, представленная в листинге 5.8, с помощью функции `dup2()` соединяет выходной. Конец канала со входом команды `sort`.<sup>[16]</sup> После создания канала программа "делится" функцией `fork()` на два процесса. Родительский процесс записывает в канал различные строки, а дочерний процесс соединяет выходной конец канала со своим входным потоком, после чего запускает команду `sort`.

### Листинг 5.8. (`dup2.c`) Перенаправление выходного потока канала с помощью функции `dup2()`

```

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

int main() {
int fds[2];
pid_t pid;

/* Создание канала. Дескрипторы обоих концов канала
помещаются в массив FDS. */
pipe (fds);
/* Создание дочернего процесса. */
pid = fork();
if (pid == (pid_t)0) {
/* Это дочерний процесс. Закрываем копию входного конца
канала */
close(fds[1]);
/* Соединяем выходной конец канала со стандартным входным
потокom. */
dup2(fds[0], STDIN_FILENO);
/* Замещаем дочерний процесс программой sort. */
execlp("sort", "sort", 0);
} else {
/* Это родительский процесс. */
FILE* stream;
/* Закрываем копию выходного конца канала. */
close(fds[0]);
/* Приводим дескриптор входного конца канала к типу FILE*
и записываем данные в канал. */
stream = fdopen(fds[1], "w");
fprintf(stream, "This is a test.\n");
fprintf(stream, "Hello, world.\n");
fprintf(stream, "My dog has fleas.\n");
fprintf(stream, "This program is great.\n");
fprintf(stream, "One fish, two fish.\n");
fflush(stream);
close(fds[1]);
/* Дожидаемся завершения дочернего процесса. */
waitpid(pid, NULL, 0);
}
return 0;
}

```

#### 5.4.4. Функции `open()` и `pclose()`

Каналы часто используются для передачи данных программе, выполняющейся как подпроцесс (или приема данных от нее). Специально для этих целей предназначены функции `open()` и `pclose()`, устраняющие необходимость в вызове функций `pipe()`, `dup2()`, `exec()` и `fdopen()`.

Сравните листинг 5.9 с предыдущим примером (листинг 5.8).

#### *Листинг 5.9. (open.c) Использование функций `open()` и `pclose()`*

```

#include <stdio.h>

```

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

```
int main() {  
FILE* stream = popen("sort", "w");  
fprintf(stream, "This is a test.\n");  
fprintf(stream, "Hello, world.\n");  
fprintf(stream, "My dog has fleas\n");  
fprintf(stream, "This program is great.\n");  
fprintf(stream, "One fish, two fish.\n");  
return pclose(stream);  
}
```

Функция `popen()` создает дочерний процесс, в котором выполняется команда `sort`. Один этот вызов заменяет вызовы функций `pipe()`, `fork()`, `dup2()` и `exec1p()`. Вторым аргументом, "w", указывает на то, что текущий процесс хочет осуществлять запись в дочерний процесс. Функция `popen()` возвращает указатель на один из концов канала; второй конец соединяется со стандартным входным потоком дочернего процесса. Функция `pclose()` закрывает входной поток дочернего процесса, дожидается его завершения и возвращает код статуса.

Первый аргумент функции `popen()` является командой интерпретатора, выполняемой в подпроцессе `/bin/sh`. Интерпретатор просматривает переменную среды `PATH`, чтобы определить, где следует искать команду. Если вторым аргументом равен "r", функция возвращает указатель на стандартный выходной поток дочернего процесса, чтобы программа могла читать данные из него. Если вторым аргументом равен "w", функция возвращает указатель на стандартный входной поток дочернего процесса, чтобы программа могла записывать данные в него. В случае ошибки возвращается пустой указатель.

Функция `pclose()` закрывает поток, указатель на который был возвращен функцией `popen()`, и дожидается завершения дочернего процесса.

#### 5.4.5. Каналы FIFO

Файл FIFO (First-In, First-Out первым пришел, первым обслужен) это канал, у которого есть имя в файловой системе. Любой процесс может открыть и закрыть такой файл. Процессы, находящиеся на противоположных концах канала, не обязаны быть связанными друг с другом. FIFO-файлы называют *именованными каналами*.

FIFO-файл создается с помощью команды `mkfifo`. Путь к файлу указывается в командной строке, например:

```
% mkfifo /tmp/fifo  
% ls -l /tmp/fifo  
prw-rw-rw- 1 samuel users 0 Jan 16 14:04 /tmp/fifo
```

Первый символ в строке режима (p) указывает на то, что файл имеет тип FIFO (именованный канал). Теперь в одном терминальном окне можно осуществлять чтение из файла с помощью команды

```
% cat < /tmp/fifo
```

а в другом окне можно выполнять запись в файл:

```
% cat > /tmp/fifo
```

Попробуйте во втором окне ввести какой-то текст и нажать <Enter>. Введенный текст немедленно отобразится в первом окне. Канал закрывается нажатием клавиш <Ctrl+D> во втором окне. FIFO-файл удаляется с помощью следующей команды:

```
% rm /tmp/fifo
```

FIFO-файл можно создать программным путем с помощью функции `mkfifo()`. Первым аргументом является путь к файлу. Второй аргумент задает права доступа к каналу со стороны его владельца, группы и остальных пользователей (об этом пойдет речь в разделе 10.3, "Права доступа к файлам"). Поскольку у канала есть читающая и записывающая стороны, права доступа должны учитывать оба случая. Если канал не может быть создан (например, файл с таким именем уже существует), функция `mkfifo()` возвращает -1. Для работы функции требуется подключить к программе файлы `<sys/types.h>` и `<sys/stat.h>`.

### *Доступ к FIFO-файлу*

К FIFO-файлу можно обращаться как к обычному файлу. При организации межзадачного взаимодействия одна программа должна открыть файл для записи, а другая - для чтения. Над файлом можно выполнять как низкоуровневые (`open()`, `write()`, `read()`, `close()` и др.), так и высокоуровневые (`fopen()`, `fprintf()`, `fscanf()`, `fclose()` и др.) функции.

Например, на низком уровне запись блока данных в FIFO-файл осуществляется следующим образом:

```
int fd = open(fifo_path, O_WRONLY);
write(fd, data, data_length);
close(fd);
```

А так выполняется чтение строки из FIFO-файла на высоком уровне:

```
FILE* fifo = fopen(fifo_path, "r");
fscanf(fifo, "%s", buffer);
fclose(fifo);
```

У FIFO-файла одновременно может быть несколько читающих и записывающих программ. Входные потоки разбиваются на атомарные блоки, размер которых определяется константой `PIPE_BUF` (4 Кбайт в Linux). Если несколько программ параллельно друг другу осуществляют запись в файл, их блоки будут чередоваться. То же самое относится к программам, одновременно читающим данные из файла.

### *Отличия от именованных каналов в Windows*

Каналы операционных систем семейства Win32 очень напоминают каналы Linux. Основное различие касается именованных каналов, которые в Win32 функционируют скорее как сокеты. Именованные каналы Win32 способны соединять по сети процессы, выполняющиеся на разных компьютерах. В Linux для этой цели используются именно сокеты. Кроме того, в Win32 допускается, чтобы несколько программ чтения или записи работали с именованным каналом, не перекрывая потоки друг друга, а сами каналы поддерживают двунаправленный обмен данными.<sup>[\[17\]](#)</sup>

## **5.5. Сокеты**

*Сокет* это устройство двунаправленного взаимодействия, которое предназначено для связи с другим процессом, выполняющимся на этом же или на другом компьютере. Сокеты используются Internet-программами, такими как `telnet`, `rlogin`, `ftp`, `talk` и Web-броузеры.

Например, с помощью программы telnet можно получить от Web-сервера HTML-страницу, поскольку обе программы общаются по сети при помощи сокетов. Чтобы установить соединение с Web-сервером `www.codesourcery.com`, следует ввести команду `telnet www.codesourcery.com 80`. Загадочная константа 80 обозначает порт, который прослушивается Web-сервером. Когда соединение будет установлено, введите команду `GET /`. В результате через сокет будет послан запрос Web-серверу, который в ответ вернет начальную HTML-страницу, после чего закроет соединение.

```
% telnet www.codesourcery.com 80
Trying 206.168.99.1...
Connected to merlin.codesourcery.com (206.168.99.1).
Escape character is '^]'.
GET /
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset="iso-8659-1">
...
```

### 5.5.1. Концепции сокетов

При создании сокета необходимо задать три параметра, тип взаимодействия, пространство имен и протокол.

Тип взаимодействия определяет способ интерпретации передаваемых данных и число абонентов. Данные, посылаемые через сокет, формируются в блоки, называемые пакетами. Тип взаимодействия указывает на то, как обрабатываются пакеты и как они передаются от отправителя к получателю.

■ При взаимодействии с установлением *соединения* гарантируется доставка пакетов в том порядке, в каком они были отправлены. Если пакеты теряются или приходят в неправильном порядке из-за проблемы в сети, принимающая сторона автоматически запрашивает у отправителя повторную отправку данных.

Сокеты, ориентированные на соединения, функционируют наподобие телефонного звонка: адреса запрашивающей и принимающей сторон фиксируются в самом начале, на этапе установки соединения.

■ При передаче *дейтаграмм* не гарантируется доставка и правильный порядок пакетов. Пакеты могут теряться и приходить в произвольном порядке. Операционная система лишь обещает сделать "все возможное".

Дейтаграммные сокеты функционируют подобно почтовой службе: отправитель указывает адрес получателя каждого сообщения и не контролирует доставку пакетов.

Пространство имен сокета определяет способ записи адресов. Например, в локальном пространстве имен адреса это обычные имена файлов. В пространстве имен Internet адрес сокета состоит из IP-адреса компьютера, подключенного к сети, и номера порта. Благодаря номерам портов можно различать сокеты, созданные на одном компьютере.

Протокол определяет способ передачи данных. Основными семействами протоколов являются TCP/IP (ключевые сетевые протоколы, используемые в Internet) и AppleTalk (протоколы, используемые системами Macintosh). Сокеты могут также работать в соответствии с локальным коммуникационным протоколом UNIX. Не все комбинации типов взаимодействия, пространств имен и протоколов поддерживаются.

### 5.5.2. Системные вызовы



Сокеты являются более гибкими в управлении, чем рассмотренные выше механизмы межзадачного взаимодействия. При работе с сокетами используются следующие функции:

- `socket()` создает сокет;
- `close()` уничтожает сокет;
- `connect()` устанавливает соединение между двумя сокетами;
- `bind()` назначает серверному сокету адрес;
- `listen()` переводит сокет в режим приема запросов на подключение;
- `accept()` принимает запрос на подключение и создает новый сокет, который будет обслуживать данное соединение.

Сокеты представляются в программе файловыми дескрипторами.

### ***Создание и уничтожение сокетов***

Функции `socket()` и `close()` создают и уничтожают сокет соответственно. В первом случае необходимо задать три параметра: пространство имен, тип взаимодействия и протокол. Константы, определяющие пространство имен, начинаются с префикса `PF_` (сокращение от "protocol family" семейство протоколов). Например, константы `PF_LOCAL` и `PF_UNIX` соответствуют локальному пространству имен, а константа `PF_INET` пространству имен Internet. Константы, определяющие тип взаимодействия, начинаются с префикса `SOCK_`. Сокетам, ориентированным на соединения, соответствует константа `SOCK_STREAM`, а дейтаграммным сокетам константа `SOCK_DGRAM`.

Выбор протокола определяется связкой "пространство имен тип взаимодействия". Поскольку для каждой такой пары, как правило, лучше всего подходит какой-то один протокол, в третьем параметре функции `socket()` обычно задается значение 0 (выбор по умолчанию). В случае успешного завершения функция `socket()` возвращает дескриптор сокета. Чтение и запись данных через сокеты осуществляется с помощью обычных файловых функций, таких как `read()`, `write()` и т.д. По окончании работы с сокетом его необходимо удалить с помощью функции `close()`.

### ***Вызов функции connect()***

Чтобы установить соединение между двумя сокетами, следует на стороне клиента вызвать функцию `connect()`, указав адрес серверного сокета. Клиент это процесс, инициирующий соединение, а сервер это процесс, ожидающий поступления запросов на подключение. В первом параметре функции `connect()` задается дескриптор клиентского сокета, во втором адрес серверного сокета, в третьем длина (в байтах) адресной структуры, на которую ссылается второй параметр. Формат адреса будет разным в зависимости от пространства имен.

### ***Передача данных***

При работе с сокетами можно применять те же самые функции, что и при работе с файлами. О низкоуровневых функциях ввода-вывода, поддерживаемых в Linux, рассказывается в приложении Б, "Низкоуровневый ввод-вывод". Имеется также специальная функция `send()`,

являющаяся альтернативой традиционной функции `write()`.

### 5.5.3. Серверы

Жизненный цикл сервера можно представить так:

- 1)создание сокета, ориентированного на соединения (функция `socket()`);
- 2)назначение сокету адреса привязки (функция `bind()`);
- 3)перевод сокета в режим ожидания запросов (функция `listen()`);
- 4)прием поступающих запросов (функция `accept()`);
- 5)закрывание сокета (функция `close()`).

Данные не записываются и не читаются напрямую через серверный сокет. Вместо этого всякий раз, когда сервер принимает запрос на соединение, ОС Linux создает отдельный сокет, используемый для передачи данных через это соединение.

Серверному сокету необходимо с помощью функции `bind()` назначить адрес, чтобы клиент смог его найти. Первым аргументом функции является дескриптор сокета. Вторым аргументом это указатель на адресную структуру, формат которой будет зависеть от выбранного семейства адресов. Третий аргумент это длина адресной структуры в байтах. После получения адреса сокет, ориентированный на соединения, должен вызвать функцию `listen()`, тем самым обозначив себя как сервер. Первым аргументом этой функции также является дескриптор сокета. Вторым аргументом определяет, сколько запросов может находиться в очереди ожидания. Если очередь заполнена, все последующие запросы отвергаются. Этот аргумент задает не предельное число запросов, которое способен обработать сервер, а максимальное количество клиентов, которые могут находиться в режиме ожидания.

Сервер принимает от клиента запрос на подключение, вызывая функцию `accept()`. Первый ее аргумент это дескриптор сокета. Вторым аргументом указывает на адресную структуру, заполняемую адресом клиентского сокета. Третий аргумент содержит длину (в байтах) адресной структуры. Функция `accept()` создает новый сокет для обслуживания клиентского соединения и возвращает его дескриптор. Исходный серверный сокет продолжает принимать запросы от клиентов. Чтобы прочитать данные из сокета, не удалив их из входящей очереди, воспользуйтесь функцией `recv()`. Она принимает те же аргументы, что и функция `read()`, плюс дополнительный аргумент `FLAGS`. Флаг `MSG_PEEK` задает режим "неразрушающего" чтения, при котором прочитанные данные остаются в очереди.

### 5.5.4. Локальные сокеты

Сокеты, соединяющие процессы в пределах одного компьютера, работают в локальном пространстве имен (`PF_LOCAL` или `PF_UNIX`, это синонимы). Такие сокеты называются *локальными* или *UNIX-сокетами*. Их адресами являются имена файлов, указываемые только при создании соединения.

Имя сокета задается в структуре типа `sockaddr_un`. В поле `sun_family` необходимо записать константу `AF_LOCAL`, указывающую на то, что адрес находится в локальном пространстве имен. Поле `sun_path` содержит путевое имя файла и не может превышать 108 байтов. Длина структуры `sockaddr_un` вычисляется с помощью макроса `SUN_LEN()`. Допускается любое имя файла, но процесс должен иметь право записи в каталог, где находится файл. При подключении к сокету процесс должен иметь право чтения файла. Несмотря на то что файловая система может экспортироваться через NFS на разные компьютеры, только процессам,

работающим в пределах одного компьютера, разрешается взаимодействовать друг с другом посредством локальных сокетов.

При работе в локальном пространстве имен допускается только протокол с номером 0.

Локальный сокет является частью файловой системы, поэтому он отображается командой `ls` (обратите внимание на букву `s` в строке режима):

```
% ls -l /tmp/socket
srwxrwx--x 1 user group 0 Nov 13 19:16 /tmp/socket
```

Если локальный сокет больше не нужен, его файл можно удалить с помощью функции `unlink()`.

#### 5.5.5. Примеры программ, работающих с локальными сокетами

Работу с локальными сокетами мы проиллюстрируем двумя программами. Первая (листинг 5.10) это сервер. Он создает локальный сокет и переходит в режим ожидания запросов на подключение. Приняв запрос, сервер читает сообщения из сокета и отображает на экране, пока соединение не будет закрыто. Если поступает сообщение "quit", сервер удаляет сокет и завершает свою работу. Программа `socket-server` ожидает путевое имя сокета в командной строке.

##### *Листинг 5.10. (socket-server.c) Сервер локального сокета*

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

/* Чтение сообщений из сокета и вывод их на экран. Функция
   продолжает работу до тех пор, пока сокет не будет закрыт.
   Функция возвращает 0, если клиент послал сообщение "quit",
   в противном случае возвращается ненулевое значение. */
int server(int client_socket) {
    while (1) {
        int length;
        char* text;

        /* Сначала читаем строку, в которой записана длина сообщения.
           Если возвращается 0, клиент закрыл соединение. */
        if (read(client_socket, &length, sizeof(length)) == 0)
            return 0;
        /* Выделение буфера для хранения текста. */
        text = (char*)malloc(length);
        /* Чтение самого сообщения и вывод его на экран. */
        read(client_socket, text, length);
        printf("%s\n", text);
        /* Очистка буфера. */
        free(text);
        /* Если клиент послал сообщение "quit.", работа сервера
           завершается. */
        if (!strcmp(text, "quit"))
            return 1;
    }
}
```

```

}
}

int main(int argc, char* const argv[]) {
const char* const socket_name = argv[1];
int socket_fd;
struct sockaddr_un name;
int client_sent_quit_message;

/* Создание локального сокета. */
socket_fd = socket(PF_LOCAL, SOCK_STREAM, 0);
/* Переход в режим сервера. */
name.sun_family = AF_LOCAL;
strcpy(name.sun_path, socket_name);
bind(socket_fd, SUN_LEN(&name));
/* Ожидание запросов. */
listen(socket_fd, 5);

/* Непрерывный прием запросов на подключение. Для каждого
клиента вызывается функция server(). Цикл продолжается,
пока не будет получено сообщение "quit". */
do {
struct sockaddr_un client_name;
socklen_t client_name_len;
int client_socket_fd;

/* Прием запроса. */
client_socket_fd =
accept(socket_fd, &client_name, &client_name_len);

/* Обработка запроса. */
client_sent_quit_message = server(client_socket_fd);
/* Закрытие серверной стороны соединения. */
close(client_socket_fd);
} while(!client_sent_quit_message);

/* Удаление файла локального сокета. */
close(socket_fd);
unlink(socket_name);
return 0;
}

```

Клиентская программа, показанная в листинге 5.11, подключается к локальному сокету и посылает сообщение. Путь к сокету и текст сообщения задаются в командной строке.

#### ***Листинг 5.11. (socket-client.c) Клиент локального сокета***

```

#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

/* Запись строки TEXT в сокет, заданный
дескриптором SOCKET_FD. */
void write_text(int socket_fd, const char* text) {

```

```

/* Сначала указывается число байтов в строке, включая
завершающий символ NULL. */
int length = strlen(text) + 1;
write(socket_fd, &length, sizeof(length));
/* Запись строки. */
write(socket_fd, text, length);
}

int main(int argc, char* const argv[]) {
const char* const socket_name = argv[1];
const char* const message = argv[2];
int socket_fd;
struct sockaddr_un name;

/* Создание сокета. */
socket_fd = socket(PF_LOCAL, SOCK_STREAM, 0);
/* Сохранение имени сервера в адресной структуре. */
name.sun_family = AF_LOCAL;
strcpy(name.sun_path, socket_name);
/* Подключение к серверному сокету. */
connect(socket_fd, &name, SUN_LEN(&name));
/* передача сообщения, заданного в командной строке. */
write_text(socket_fd, message);
close(socket_fd);
return 0;
}

```

Прежде чем отправить текст сообщения, клиент записывает в сокет число (хранится в переменной `length`), определяющее длину сообщения в байтах. На противоположной стороне сервер выясняет длину сообщения и выделяет для него буфер соответствующего размера, после чего читает само сообщение.

Чтобы проверить этот пример, запустите в одном терминальном окне серверную программу, указав путь к сокету, например:

```
% ./socket-server /tmp/socket
```

В другом окне запустите несколько раз клиентскую программу, задав тот же путь к сокету плюс требуемое сообщение:

```
% ./socket-client /tmp/socket "Hello, world."
% ./socket-client /tmp/socket "This is a test."
```

Сервер получит и отобразит эти сообщения. Чтобы закрыть сервер, пошлите ему сообщение "quit":

```
% ./socket-client /tmp/socket "quit"
```

### 5.5.6. Internet-сокеты

UNIX-сокеты используются для организации взаимодействия двух процессов, выполняющихся на одном компьютере. С другой стороны. Internet-сокеты позволяют соединять между собой процессы, работающие на разных компьютерах.

Пространству имен Internet соответствует константа `PF_INET`. Internet-сокеты чаще всего работают по протоколам TCP/IP. Протокол IP (Internet Protocol) отвечает за низкоуровневую доставку сообщений, осуществляя при необходимости их разбивку на пакеты и последующую компоновку. Доставка пакетов не гарантируется, поэтому они могут исчезать или приходить в неправильном порядке. Каждый компьютер в сети имеет свой IP-адрес. Протокол TCP (Transmission Control Protocol) функционирует поверх протокола IP и обеспечивает надежную доставку сообщений, ориентированную на установление соединений.

Легче запоминать имена а не числа, поэтому служба DNS (Domain Name Service) закрепляет за IP-адресами доменные имена вида `www.codesourcery.com`. Служба DNS организована в виде всемирной иерархии серверов имен. Чтобы использовать доменные имена в программах, нет необходимости разбираться в протоколах DNS

Адрес Internet-сокета состоит из двух частей: адреса компьютера и номера порта. Эта информация хранится в структуре типа `sockaddr_in`. В поле `sin_family` необходимо записать константу `AF_INET`, указывающую на то, что адрес принадлежит пространству имен Internet. В поле `sin_addr` хранится IP-адрес компьютера в виде 32-разрядного целого числа. Благодаря номерам портов можно различать сокеты, создаваемые на одном компьютере. В разных системах многобайтовые значения могут храниться с разным порядком следования байтов, поэтому с помощью функции `htons()` необходимо преобразовать номер порта в число с сетевым порядком следования байтов.

Функция `gethostbyname()` преобразует адрес компьютера из текстового представления стандартного точечного (например, `10.10.10.1`) или доменного (например, `www.codesourcery.com`) во внутреннее 32-разрядное. Функция возвращает указатель на структуру типа `hostent`. IP-адрес находится в ее поле `h_addr`.

Программа, представленная в листинге 5.12, иллюстрирует работу с Internet-сокетами. Программа запрашивает начальную страницу у Web-сервера, адрес которого указан в командной строке.

### ***Листинг 5.12. (socket-inet.c) Чтение страницы с Web-сервера***

```
#include <stdlib.h>
#include <stdio.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/socket.h>
#include <unistd.h>
#include <string.h>

/* Отображение содержимого Web-страницы, полученной из
серверного сокета. */
void get_home_page(int socket_fd) {
    char buffer[10000];
    ssize_t number_characters_read;

    /* Отправка HTTP-команды GET с запросом начальной страницы. */
    sprintf(buffer, "GET /\n");
    write(socket_fd, buffer, strlen(buffer));
    /* Чтение данных из сокета. Функция read() может вернуть
не все данные сразу, поэтому продолжаем чтение, пока
не будут получены все данные. */
    while (1) {
        number_characters_read = read(socket_fd, buffer, 10000);
        if (number_characters_read == 0)
            return;
        /* Запись данных в стандартный выходной поток. */
        fwrite(buffer, sizeof(char), number_characters_read, stdout);
    }
}
```

```

}
}

int main(int argc, char* const argv[]) {
int socket_fd;
struct sockaddr_in name;
struct hostent* hostinfo;

/* Создание сокета. */
socket_fd = socket(PF_INET, SOCK_STREAM, 0);
/* Запись имени сервера в адресную структуру. */
name.sin_family = AF_INET;
/* Преобразование адреса из текстового представления во
внутреннюю форму. */
hostinfo = gethostbyname(argv[1]);
if (hostinfo == NULL)
return 1;
else
name.sin_addr = *((struct in_addr*)hostinfo->h_addr);
/* Web-серверы используют порт 80. */
name.sin_port = htons(80);

/* Подключаемся к Web-серверу. */
if (connect(socket_fd, &name,
sizeof(struct sockaddr_in)) == -1) {
perror("connect");
return 1;
}
/* получаем содержимое начальной страницы сервера. */
get_home_page(socket_fd);
return 0;
}

```

Программа извлекает имя Web-сервера из командной строки (имя не является URL-адресом, т.е. в нем отсутствует префикс `http://`). Далее вызывается функция `gethostbyname()`, которая преобразует имя сервера в числовое представление. После этого программа подключает потоковый (TCP) сокет к порту 80 сервера. Web-серверы общаются по протоколу HTTP (Hypertext Transfer Protocol), поэтому программа посылает HTTP-команду GET, в ответ на которую сервер возвращает текст начальной страницы.

### **Стандартные номера портов**

По существующему соглашению Web-серверы ожидают поступления запросов на порт 80. За большинством Internet-сервисов закреплены стандартные номера портов. Например, защищенные Web-серверы работающие по протоколу SSL. прослушивают порт 443 а почтовые серверы (протокол SMTP) прослушивают порт 25

В Linux связи между именами протоколов/сервисов и номерами портов устанавливаются в файле `/etc/services`. В первой колонке файла указано имя протокола или сервисе. Во второй колонке приведен номер порта и тип взаимодействия: `tcp` для сервисов ориентированных на соединения, и `udp` для дейтаграмм.

При реализации собственных сетевых сервисов используйте номере портов, большие чем 1024

Например, чтобы получить начальную страницу с сервера `www.codesourcery.com`, введите следующую команду:

```
% ./socket-inet www.codesourcery.com
<html>
<meta http-equiv="Content-Type"
content="text/html; charset=iso-8859-1">
...
```

### 5.5.7. Пары сокетов

Как было показано выше, функция `pipe()` создает два дескриптора для входного и выходного концов канала. Возможности каналов ограничены, так как с файловыми дескрипторами должны работать связанные процессы и данные через канал передаются только в одном направлении. Функция `socketpair()` создает два дескриптора для двух связанных сокетов, находящихся на одном компьютере. С помощью этих дескрипторов можно организовать двунаправленное взаимодействие процессов.

Первые три параметра функции `socketpair()` такие же, как и в функции `socket()`: пространство имен (должно быть `PF_LOCAL`), тип взаимодействия и протокол. Последний параметр это массив из двух целых чисел, куда будут записаны дескрипторы сокетов, подобно функции `pipe()`.





# Глава 6

## Устройства

Linux, как и большинство операционных систем, взаимодействует с аппаратными устройствами посредством модульных программных компонентов, называемых *драйверами*. Драйвер скрывает от операционной системы детали взаимодействия с устройством и предоставляет в распоряжение системы стандартный интерфейс обращения к устройству.

В Linux драйверы устройств являются частью ядра и могут подключаться к ядру статически либо по запросу в виде модулей. Драйверы недоступны напрямую пользовательским процессам. Но в Linux имеется особый механизм специальные файловые объекты, позволяющие процессам взаимодействовать с драйверами, а через них с аппаратными устройствами. Такие объекты являются частью операционной системы, поэтому программы могут открывать их, читать из них данные и осуществлять запись в них точно так же, как если бы это были обычные файлы. С помощью низкоуровневых вызовов (описаны в приложении Б, "Низкоуровневый ввод-вывод") или стандартных библиотечных функций ввода-вывода программы могут обмениваться данными с устройствами через файловые объекты

В Linux есть также ряд файловых объектов, предназначенных для доступа к ядру, а не к драйверам устройств. Такие объекты не связаны с аппаратными устройствами. Они реализуют специальные функции, используемые приложениями и системными программами.

***Будьте осторожны при доступе к устройствам!***

Описанные в этой главе методики позволяют непосредственно взаимодействовать с драйверами устройств, работающими в ядре Linux, а через них с аппаратными устройствами, подключенными к системе. Применить эти методики следует осторожно, чтобы не нарушить работоспособность системы

### 6.1. Типы устройств

Файлы устройств не являются обычными файлами: с ними не связаны блоки данных на диске. Данные, помещаемые в такой файл или извлекаемые из него, передаются соответствующему драйверу устройства или принимаются от него, а драйвер, в свою очередь, осуществляет обмен данными с обслуживаемым устройством. Устройства классифицируются по двум типам.

■ *Символьные (байт-ориентированные)* устройства читают и записывают данные в виде потока байтов. Сюда входят последовательные и параллельные порты, накопители на магнитной ленте, терминалы и звуковые платы.

■ *Блочные (блок-ориентированные)* устройства читают и записывают данные блоками фиксированного размера. В отличие от символьных устройств блочные устройства предоставляют произвольный доступ к своим данным. В качестве примера можно назвать жесткий диск.

Как правило, приложения не работают с блочными устройствами. В каждом разделе жесткого диска содержится файловая система, которая монтируется к дереву корневой файловой системы Linux. Лишь ядро, реализующее функции файловой системы, получает прямой доступ к блочному устройству. Программы обращаются к содержимому диска через

обычные файлы и каталоги.

## Опасность доступа к блочному устройству

Драйверы блочных устройств имеют прямой доступ к данным, хранящимся на диске. В большинстве Linux-систем прямой доступ к таким устройствам разрешен лишь процессам, выполняющимся от имени пользователя `root`, но и они способны нанести непоправимый ущерб, изменив содержимое диска. Осуществляя запись в блочное устройство, программа может модифицировать или уничтожить не только управляющую информацию, хранящуюся в файловой системе, но и таблицу разделов диска и даже главную загрузочную запись. Вследствие этого жесткий диск или вся система может оказаться разрушенной.

Приложениям иногда приходится иметь дело с символьными устройствами: об этом пойдет речь в разделе 6.5, "Специальные устройства".

## 6.2. Номера устройств

ОС Linux идентифицирует устройства двумя числами: *старшим номером устройства* и *младшим номером устройства*. Старший номер указывает на то, какой драйвер соответствует устройству. Соответствие между старшими номерами устройств и драйверами жестко зафиксировано в исходных файлах ядра Linux. Двум разным драйверам может соответствовать одинаковый старший номер. Это значит, что один драйвер управляет символьным устройством, а второй блочным. Младшие номера позволяют различать отдельные устройства или аппаратные компоненты, управляемые одним драйвером. Значение младшего номера зависит от драйвера.

Например, устройству со старшим номером 3 соответствует основной контроллер IDE. К этому контроллеру могут быть подключены два устройства (жесткие диски, накопитель на магнитной ленте или дисковод CD-ROM). "Главному" устройству будет соответствовать младший номер 0, а "подчиненному" устройству номер 64. Отдельные разделы главного устройства (если он поддерживает разбивку на разделы) будут иметь младшие номера 1, 2, 3 и т.д. Разделы подчиненного устройства представляются младшими номерами 65, 66, 67 и т.д.

Список старших номеров устройств можно узнать в документации к исходным текстам ядра Linux. Во многих дистрибутивах эта информация хранится в файле `/usr/src/Linux/Documentation/devices.txt`. В специальном файле `/proc/devices` перечислены старшие номера устройств, соответствующие загруженным в данный момент драйверам (о файловой системе `/proc` рассказывается в главе 7, "Файловая система `/proc`").

## 6.3. Файловые ссылки на устройства

Ссылки на устройства напоминают обычные файлы. Их можно перемещать с помощью команды `mv` и удалять командой `rm`. Правда, если попытаться скопировать такую ссылку с помощью команды `cp`, из устройства будут прочитаны данные (при условии что устройство поддерживает операцию чтения) и эти данные перенесутся в указанный файл. При попытке перезаписи ссылки в соответствующее устройство будут записаны данные.

Ссылка на устройство создается с помощью команды `mknod` (документация вызывается так: `man 1 mknod`) или функции `mknod()` (документация вызывается так: `man 2 mknod`). Создание

ссылки не означает, что драйвер устройства или само устройство автоматически станут доступными. Ссылка является лишь своего рода порталом, через который происходит взаимодействие с драйвером. Создавать такие ссылки разрешается только процессам суперпользователя.

Первый аргумент команды `mknod` задает путь, под которым ссылка появится в файловой системе. Второй аргумент равен `b` для блочного устройства и `c` для символического устройства. Старший и младший номера устройства задаются в третьем и четвертом аргументах соответственно. Например, следующая команда создает в текущем каталоге ссылку на символическое устройство `lp0`. Старший номер устройства 6, младший 0. Эти номера соответствуют первому параллельному порту Linux.

```
% mknod ./lp0 c 6 0
```

Помните, что лишь суперпользователю разрешено создавать ссылки на устройства, поэтому для успешного выполнения показанной команды необходимо зарегистрироваться в системе под именем `root`.

Команда `ls` особым образом помечает ссылки на устройства. Если вызвать ее с флагом `-l` или `-o`, то первый символ в каждой строке будет обозначать тип записи. Знак `-` (дефис) соответствует обычному файлу, буква `d` каталогу, `b` блочному устройству, `c` символическому устройству. В последних двух случаях команда `ls` вместо размера файла отображает старший и младший номера устройства. Давайте, к примеру, получим информацию о ссылке на символическое устройство, которую мы только что создали:

```
% ls -l lp0
crw-r----- 1 root root 6, 0 Mar 7 17:03 lp0
```

В распоряжении программ имеется функция `stat()`, которая позволяет не только узнать, какому устройству символическому или блочному соответствует ссылка, но и определить номера устройства. Эта функция описана в приложении Б, "Низкоуровневый ввод-вывод".

Удалить ссылку на устройство (не сам драйвер) можно с помощью команды `rm`:

```
% rm ./lp0
```

### 6.3.1. Каталог `/dev`

В Linux имеется каталог `/dev`, в котором содержатся ссылки на все символичные и блочные устройства, известные системе. Имена этих ссылок стандартизированы

Например, главное устройство, подключенное к основному контроллеру IDE, имеет старший и младший номера 3 и 0 соответственно, а его стандартное имя `/dev/hda`. Если данное устройство поддерживает разделы, то первый раздел (младший номер 1) будет называться `/dev/hda1`. Проверим это:

```
% ls -l /dev/hda /dev/hda1
brw-rw---- 1 root disk 3, 0 May 5 1998 /dev/hda
brw-rw---- 1 root disk 3, 1 May 5 1998 /dev/hda1
```

Здесь же будет находиться и ссылка на параллельный порт, которую мы создали выше:

```
% ls -l /dev/lp0
crw-rw---- 1 root daemon 6, 0 May 5 1998 /dev/lp0
```

В большинстве случаев нет необходимости с помощью команды `mknod` создавать собственные ссылки. Достаточно скопировать нужные ссылки из каталога `/dev`. У программ, не располагающих привилегиями суперпользователя, нет другого выбора, кроме как пользоваться имеющимися ссылками. Обычно новые ссылки создаются только системными администраторами и разработчиками драйверов. В Linux имеются специальные средства, упрощающие администраторам процесс создания ссылок с правильными именами.

6.3.2. Доступ к устройству путем открытия файла

Как работать с аппаратными устройствами? В случае символьного устройства ответ прост: откройте ссылку на устройство как обычный файл и осуществляйте чтение-запись традиционным образом. Например, если к первому параллельному порту подключен принтер, то распечатать файл document.txt можно, направив его непосредственно на устройство /dev/lp0:

```
% cat document.txt > /dev/lp0
```

Чтобы эта команда завершилась успешно, необходимо иметь право записи в файл принтера. Во многих Linux-системах таким правом обладают лишь пользователь root и системный демон печати (lpd). Кроме того, результат работы принтера зависит от того, как он интерпретирует посылаемые ему данные. Одни принтеры распечатывают текстовые файлы, [\[18\]](#) другие нет. PostScript-принтеры распечатывают файлы формата PostScript.

Послать устройству данные из программы несложно. В приведенном ниже фрагменте программы с помощью низкоуровневых функций ввода-вывода содержимое буфера направляется в устройство /dev/lp0:

```
int fd = open("/dev/lp0", O_WRONLY);
write(fd, buffer, bufffer_length);
close(fd);
```

6.4. Аппаратные устройства

В табл. 6.1 перечислены распространенные блочные устройства. "Родственные" устройства именуются схожим образом (например, второй раздел первого SCSI-диска называется /dev/sda2). Эта информация будет полезна при анализе файла /proc/mounts на предмет того, какие файловые системы смонтированы в настоящий момент (об этом рассказывается в разделе 7.5, "Дисководы, точки монтирования и файловые системы").

Таблица 6.1. Распространенные блочные устройства

Устройство	Имя	Старший номер	Младший номер
Первый дисковод гибких дисков	/dev/fd0	2	0
Второй дисковод гибких дисков	/dev/fd1	2	1
Основной IDE-контроллер, главное устройство	/dev/hda	3	0
Основной IDE-контроллер, главное устройство, первый раздел	/dev/hda1	3	1
Основной IDE-контроллер, подчиненное устройство	/dev/hdb	3	64
Основной IDE-контроллер, подчиненное устройство, первый раздел	/dev/hdb1	3	65
Дополнительный IDE-контроллер, главное устройство	/dev/hdc	22	0
Дополнительный IDE-контроллер, подчиненное устройство	/dev/hdd	22	64
Первый SCSI-диск	/dev/sda	8	0
Первый SCSI-диск, первый раздел	/dev/sda1	8	1
Второй SCSI диск	/dev/sdb	8	16
Второй SCSI-диск, первый раздел	/dev/sdb1	8	17

Первый SCSI-дисковод CD-ROM	/dev/scd0 11	0
Второй SCSI-дисковод CD-ROM	/dev/scd1 11	1

В табл. 6.2 перечислены распространенные символьные устройства.

Таблица 6.2. Распространенные символьные устройства

Устройство	Имя	Старший номер	Младший номер
Параллельный порт 0	/dev/lp0 или /dev/par0	6	0
Параллельный порт 1	/dev/lp1 или /dev/par1	6	1
Первый последовательный порт	/dev/ttyS0	4	64
Второй последовательный порт	/dev/ttyS1	4	65
IDE-накопитель на магнитной ленте	/dev/ht0	37	0
Первый SCSI-накопитель на магнитной ленте	/dev/st0	9	0
Второй SCSI-накопитель на магнитной ленте	/dev/st1	9	1
Системная консоль	/dev/console	5	1
Первый виртуальный терминал	/dev/tty1	4	1
Второй виртуальный терминал	/dev/tty2	4	2
Текущее терминальное устройство процесса	/dev/tty	5	0
Звуковая плата	/dev/audio	14	4

К некоторым аппаратным компонентам можно получить доступ сразу через несколько символьных устройств. Чаще всего этим устройствам соответствует разная семантика доступа. Например, если в системе есть ленточное IDE-устройство /dev/ht0, то Linux автоматически перематывает ленту в дисковомде, когда программа закрывает дескриптор файла устройства. С помощью ссылки /dev/nht0 можно обратиться к тому же ленточному накопителю, но режим автоматической перемотки в нем будет отключен. Иногда в системе есть ссылки наподобие /dev/cua0. Это старые интерфейсы последовательных портов, таких как /dev/ttyS0.

Иногда требуется записывать данные непосредственно в символьные устройства. Рассмотрим примеры.

■Терминальная программа напрямую обращается к модему через устройство последовательного порта. Данные, записываемые в устройство, передаются по модему на удаленный компьютер.

■Программа резервного копирования записывает данные непосредственно на ленту. Такая программа может реализовывать свои собственные алгоритмы сжатия и проверки ошибок.

■Программа обращается к первому виртуальному терминалу,<sup>[19]</sup> записывая данные в устройство /dev/tty1.

Терминальным окнам, работающим в графической среде, и окнам сеансов удаленной регистрации назначаются не виртуальные терминалы, а псевдотерминалы (о них говорится в разделе 6.6, "Псевдотерминалы")

■Иногда программе требуется получить доступ к терминальному устройству, с которым она связана.

Например, программа может попросить пользователя ввести пароль. Из соображений безопасности требуется проигнорировать перенаправление стандартных потоков ввода и вывода и прочитать пароль с терминала независимо от того, как пользователь вызвал программу. Для этого можно открыть файл `/dev/tty`, всегда соответствующий текущему терминальному устройству процесса. Запишите в данный файл строку приглашения, а затем прочитайте пароль. Это не позволит пользователю передать программе пароль из файла с помощью следующего синтаксиса:

```
% secure_program < my-password.txt
```

■ Программа воспроизводит аудиофайл через звуковую плату, посылая аудиоданные в устройство `/dev/audio`. Эти данные должны быть представлены в формате Sun (такие файлы обычно имеют расширение `.au`).

Например, во многие дистрибутивы Linux входит файл `/usr/share/sndconfig/sample.au`. Попробуйте воспроизвести его с помощью такой команды:

```
% cat /usr/share/sndconfig/sample.au > /dev/audio
```

Те, кто хотят включить звук в свои программы, должны использовать специальные сервисы и библиотеки функций работы со звуком, имеющиеся в Linux. В графической среде Gnome есть демон Esound (доступен по адресу <http://www.tux.org/~riclude/Esound.html>), в KDE программа aRts (<http://space.twc.de/~stefan/kde/arts-mcop-doc/>). Благодаря этим средствам приложения, обращающиеся к звуковой плате, лучше взаимодействуют друг с другом.

## 6.5. Специальные устройства

В Linux есть также ряд специальных символьных устройств, которым не соответствуют никакие аппаратные компоненты. Старший номер всех таких устройств равен 1. Это означает, что обращение к устройству переадресуется ядру Linux.

### 6.5.1. `/dev/null`

Устройство `/dev/null` служит двум целям.

■ Linux удаляет любые данные, направляемые в устройство `/dev/null`. В тех случаях, когда выводные данные программы не нужны, в качестве выходного файла назначают устройство `/dev/null`, например:

```
% verbose_command > /dev/null
```

■ При чтении из устройства `/dev/null` всегда возвращается признак конца строки. Если открыть файл `/dev/null` с помощью функции `open()` и попытаться прочесть данные из него с помощью функции `read()`, функция вернет 0 байтов. При копировании файла `/dev/null` в другое место будет создан пустой файл нулевой длины:

```
% cp /dev/null empty-file
```

```
% ls -l empty-file
```

```
-rw-rw---- 1 samuel samuel 0 Mar 8 00:27 empty-file
```

### 6.5.2. `/dev/zero`

Устройство `/dev/zero` ведет себя так, как если бы оно было файлом бесконечной длины, заполненным одними нулями. Сколько бы данных ни запрашивалось из этого файла, ОС Linux "сгенерирует" достаточное количество кулевых байтов.

Чтобы проверить это, запустите программу `hexdump`, представленную в листинге Б.4

приложения Б, "Низкоуровневый ввод-вывод". Программа отображает содержимое файла /dev/zero в шестнадцатеричном виде:

```
% ./hexdump /dev/zero
0x000000 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000010 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000020 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000030 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
...
```

Чтобы прервать работу программы, нажмите <Ctrl+C>.

Файл /dev/zero используется в функциях выделения памяти, которые отображают этот файл в памяти, чтобы инициализировать выделяемые сегменты нулями. Об этом рассказывается в разделах 5.3.5, "Другие применения функции mmap()", и 8.9. "Функция mprotect(): задание прав доступа к памяти".

### 6.5.3. /dev/full

Устройство /dev/full ведет себя так, как если бы оно было файлом в файловой системе, где не осталось свободного места. Операция записи в этот файл завершается ошибкой, и в переменную errno помещается код ENOSPC, обычно свидетельствующий о том, что устройство записи переполнено.

Вот что получится, если попытаться осуществить запись в устройство /dev/full с помощью команды cp:

```
% cp /etc/fstab /dev/full
cp: /dev/full: No space left on device
```

Этот файл удобен для проверки того, как программа будет вести себя в случае, если при записи в файл возникнет нехватка места.

### 6.5.4. Устройства генерирования случайных чисел

Специальные устройства /dev/random и /dev/urandom предоставляют доступ к средствам генерирования случайных чисел, встроенным в ядро Linux.

Большинство аналогичных программных функций, например функция rand() стандартной библиотеки языка C, в действительности генерируют *псевдослучайные* числа. Такие числа имеют некоторые свойства случайных последовательностей, но их можно воспроизвести: достаточно задать то же самое инициализирующее значение, чтобы получить одинаковую последовательность чисел. Такое поведение неизбежно, ведь внутренняя работа компьютера жестко определена и предсказуема. Но в ряде приложений это крайне нежелательно. Например, можно взломать криптографический шифр, если воспроизвести последовательность случайных чисел, лежащих в его основе.

Чтобы получить настоящие случайные числа, необходим внешний "источник хаоса". Ядро Linux знает о таком источнике: это *вы сами*! Замеряя задержки между действиями пользователя, в частности нажатиями клавиш и перемещениями мыши, ядро способно генерировать непредсказуемый поток действительно случайных чисел. Получить доступ к этому потоку можно путем чтения из устройств /dev/random и /dev/urandom.

Разница между устройствами проявляется, когда запас случайных чисел в ядре Linux заканчивается. Если попытаться прочесть большое количество байтов из устройства /dev/random и при этом не выполнять никаких пользовательских действий (не нажимать клавиши, не перемещать мышь и т.п.), система заблокирует операцию чтения. Только когда



пользователь проявит какую-то активность, система сгенерирует дополнительные случайные числа и передаст их программе.

Попытайтесь, к примеру, отобразить содержимое файла `/dev/random` с помощью команды `od`.<sup>[20]</sup> В каждой строке выходных данных содержится 16 случайных байтов.

```
% od -t x1 /dev/random
00000000 2c 9c 7a db 2e 79 3d 65 36 c2 e3 1b 52 75 1e 1a
00000020 d3 6d 1e a7 91 05 2d 4d c3 a6 de 54 29 f4 46 04
00000040 b3 b0 8d 94 21 57 f3 90 61 dd 26 ac 94 c3 b9 3a
00000060 05 a3 02 cb 22 0a be c9 45 dd a6 59 40 22 53 d4
```

Число строк в выводе команды будет разным (их может оказаться очень мало). Главное то, что, в конце концов, вывод прекратится, поскольку операционная система исчерпает запас случайных чисел. Попробуйте теперь переместить мышь или нажать что-нибудь на клавиатуре, и вы увидите, что появляются новые случайные числа.

В противоположность этому операция чтения из устройства `/dev/urandom` никогда не блокируется. Если в системе кончаются случайные числа, Linux использует криптографический алгоритм, чтобы сгенерировать псевдослучайные числа из последней цепочки случайных байтов.

Следующая команда будет выполняться до тех пор, пока пользователь не нажмет `<Ctrl+C>`:

```
% od -t x1 /dev/urandom
00000000 62 71 d6 3e af dd de 62 c0 42 78 bd 29 9c 69 49
00000020 26 3b 95 be b9 6c 15 16 38 fd 7e 34 f0 ba ee c3
00000040 95 31 e5 2c 8d 8a dd f4 c4 3b 9b 44 2f 20 d1 54
...
```

Получить доступ в программе к генератору случайных чисел несложно. В листинге 6.1 показана функция, которая генерирует случайное число, читая байты из файла `/dev/random`. Помните, что операция чтения из этого файла окажется заблокированной в случае нехватки случайных чисел. Если важна скорость работы функции и можно смириться с тем, что некоторые числа окажутся псевдослучайными, воспользуйтесь файлом `/dev/urandom`.

**Листинг 6.1. (*random\_number.c*) Генерирование случайного числа с помощью файла `/dev/random`**

```
#include <assert.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>

/* Функция возвращает случайное число в диапазоне от MIN до MAX
включительно. Случайная последовательность байтов читается из
файла /dev/random. */
int random_number(int min, int max) {
    /* Дескриптор файла /dev/random сохраняется в статической
    переменной, чтобы не приходилось повторно открывать файл
    при каждом следующем вызове функции. */
    static int dev_random_fd = -1;
    char* next_random_byte;
    int bytes_to_read;
    unsigned random_value;

    /* Убеждаемся, что аргумент MAX больше, чем MIN. */
    assert(max > min);
```

```

/* Если функция вызывается впервые, открываем файл /dev/random
и сохраняем его дескриптор. */
if (dev_random_fd == -1) {
dev_random_fd = open("/dev/random", O_RDONLY);
assert(dev_random_fd != -1);
}

/* Читаем столько байтов, сколько необходимо для заполнения
целочисленной переменной. */
next_random_byte = (char*)&random_value;
bytes_to_read = sizeof(random_value);
/* Цикл выполняется до тех пор, пока не будет прочитано
требуемое количество байтов. Поскольку файл /dev/random
заполняется в результате пользовательских действий,
при длительном отсутствии активности операция чтения
может быть заблокирована или возвращать
лишь один байт за раз. */
do {
int bytes_read;
bytes_read =
read(dev_random_fd, next_random_byte, bytes_to_read);
bytes_to_read -= bytes_read;
next_random_byte += bytes_read;
} while (bytes_to_read > 0);
/* Вычисляем случайное число в правильном диапазоне. */
return min + (random_value % (max - min + 1));
}

```

### 6.5.5. Устройства обратной связи

*Устройство обратной связи* позволяет симитировать блочное устройство с помощью обычного дискового файла. Представьте жесткий диск, в котором данные находятся не в дорожках и секторах, а в файле с именем `disk-image` (естественно, сам этот файл должен размещаться на реальном диске, размер которого больше имитируемого).

Устройства обратной связи называются `/dev/loop0`, `/dev/loop1` и т.д. Каждому из них соответствует одно виртуальное блочное устройство. Создавать такие устройства может только суперпользователь.

Устройство обратной связи используется так же, как и любое другое блочное устройство. В частности, на нем можно создать файловую систему и смонтировать ее подобно файловой системе обычного диска или раздела. Такая файловая система, целиком размещаемая в дисковом файле, называется *виртуальной файловой системой* (ВФС).

Ниже описана последовательность действий, которые необходимо выполнить, чтобы создать виртуальную файловую систему и смонтировать ее на устройстве обратной связи.

1. Создайте пустой файл, который будет содержать образ ВФС. Размер файла должен соответствовать видимому размеру виртуальной файловой системы после ее монтирования.

Проще всего создать файл фиксированного размера с помощью команды `dd`. Эта команда копирует блоки (по умолчанию каждый из них имеет размер 512 байтов) из одного файла в другой. Лучший источник байтов для копирования устройство `/dev/zero`.

Файл `disk-image` размером 10 Мбайт создается следующим образом:

```

% dd if=/dev/zero of=/trap/disk-image count=20480
20480+0 records in
20480+0 records out

```

```
% ls -l /tmp/disk-image  
-rw-rw---- 1 root root 10485760 Mar 8 01:56 /trap/disk-image
```

2. Только что созданный файл заполнен нулевыми байтами. Теперь следует сформировать в нем файловую систему. При этом будут созданы управляющие структуры, предназначенные для организации и хранения файлов, и корневой каталог.

Файловая система может иметь любой тип. Команда `mke2fs` создает файловую систему типа `ext2` (чаще всего используется в жестких дисках Linux-систем). Поскольку команда обычно работает с блочными устройствами, она потребует подтверждение:

```
% mke2fs -q /tmp/disk-image  
mke2fs 1.18, 11-Nov-1999 for EXT2 FS 0.5b, 95/08/09  
disk-image is not a block special device.  
Proceed anyway? (y,n) y
```

Опция `-q` подавляет вывод статистики файловой системы.

Теперь файл `disk-image` содержит новую файловую систему, как если бы это был жесткий диск емкостью 10 Мбайт.

3. Смонтируйте файловую систему с использованием устройства обратной связи. Для этого введите команду `mount`, указав файл образа диска в качестве устройства монтирования. Необходимо также задать опцию `-o loop=устройство_обратной_связи`. Ниже показаны команды, которые это делают. Помните, что только суперпользователь может работать с устройством обратной связи. Первая команда создает каталог `/tmp/virtual-fs`, который станет точкой монтирования ВФС.

```
% mkdir /tmp/virtual-fs  
% mount -o loop=/dev/loop0 /tmp/disk-image /tmp/virtual-fs
```

Теперь образ диска смонтирован подобно обычному жесткому диску емкостью 10 Мбайт.

```
% df -h /tmp/virtual-fs  
Filesystem Size Used Avail Use% Mounted on  
/tmp/disk-image 9.7M 13k 9.2M 0% /tmp/virtual-fs
```

Для работы с новой файловой системой применяются обычные команды:

```
% cd /tmp/virtual-fs  
% echo 'Hello, world!' > test.txt  
% ls -l total 13  
drwxr-xr-x 2 root root 12288 Mar 8 02:00 lost+found  
-rw-rw---- 1 root root 14 Mar 8 02:12 test.txt  
% cat test.txt  
Hello, world!
```

Каталог `lost+found` автоматически добавляется командой `mke2fs`.[\[21\]](#)

По завершении работы с виртуальной файловой системой ее следует демонтировать:

```
% cd /tmp  
% umount /tmp/virtual-fs
```

При желании файл `disk-image` можно удалить или смонтировать позднее, чтобы получить доступ к файлам ВФС. Можно даже скопировать файл на другой компьютер и смонтировать его там вся файловая система будет воссоздана в неизменном виде.

Файловую систему можно не создавать с нуля, а скопировать непосредственно с устройства, например с компакт-диска. Если в системе есть IDE-дисковод CD-ROM, ему будет соответствовать имя устройства наподобие `/dev/hda`. Имя устройства для SCSI-дисковода будет примерно таким: `/dev/scd0`. В системе может также существовать символическая ссылка `/dev/cdrom`. Чтобы узнать, какое конкретно устройство закреплено за дисководом CDROM, просмотрите файл `/etc/fstab`.

Достаточно скопировать содержимое устройства в файл. В результате будет создан полный образ файловой системы компакт-диска, вставленного в дисковод. Например:

```
% cp /dev/cdrom /tmp/cdrom-image
```

Такая команда может выполняться несколько минут, в зависимости от емкости компакт-диска и скорости дискового.

Теперь можно монтировать образ компакт-диска даже при отсутствии самого накопителя в дисковом. Например, следующая команда назначает точкой монтирования каталог `/mnt/cdrom`:

```
% mount -o loop=/dev/loop0 /tmp/cdrom-image /mnt/cdrom
```

Поскольку образ файловой системы находится на жестком диске, доступ к ней будет осуществляться гораздо быстрее, чем к исходному компакт-диску. В большинстве компакт-дисков файловая система имеет тип `iso9660`.

## 6.6. Псевдотерминалы

Если запустить команду `mount` без аргументов, будет выдан список всех смонтированных файловых систем. Одна из строк выглядит примерно так:

```
none on /dev/pts type devpts (rw,gid=5,mode=620)
```

Она указывает на то, что файловая система специального типа `devpts` смонтирована в каталоге `/dev/pts`. Эта файловая система не связана ни с каким аппаратным устройством, создается ядром Linux и напоминает файловую систему `/proc` (о ней пойдет речь в главе 7, "Файловая система `/proc`").

Подобно каталогу `/dev` каталог `/dev/pts` содержит ссылки на устройства, но создается ядром динамически. Его "наполнение" меняется, отражая состояние работающей системы. Все записи этого каталога соответствуют псевдотерминалам. ОС Linux создает псевдотерминал для каждого открываемого терминального окна и помещает ссылку на него в каталог `/dev/pts`. Псевдотерминалы ведут себя аналогично терминальным устройствам: они принимают данные с клавиатуры и отображают текст, передаваемый им программами. Номер псевдотерминала является именем его записи в каталоге `/dev/pts`.

### 6.6.1. Пример работы с псевдотерминалом

Узнать, какое терминальное устройство закреплено за процессом, можно с помощью команды `ps`. Укажите в опции `-o` столбец `tty`, чтобы он был включен в отчет команды. Например, следующая команда отображает идентификаторы процессов, терминалы, на которых они работают, и командные строки их вызова:

```
% ps -o pid,tty,cmd
PID TTY CMD
28832 pts/4 bash
29287 pts/4 ps -o pid,tty,cmd
```

В данном случае терминальному окну соответствует псевдотерминал 4.

У каждого псевдотерминала есть запись в каталоге `/dev/pts`:

```
% ls -l /dev/pts/4
crw--w---- 1 samuel tty 136, 4 Mar 8 02:56 /dev/pts/4
```

Обратите внимание на то, что псевдотерминал это символьное устройство, а его владельцем является владелец процесса, для которого был создан псевдотерминал.

С псевдотерминалом можно обмениваться данными. При чтении перехватываются символы, вводимые с клавиатуры, а при записи данные отображаются в окне терминала.

Попробуйте открыть новое терминальное окно и определить номер псевдотерминала, выполнив команду `ps -o pid,tty,cmd`. Теперь откройте другое окно и направьте какие-то данные на псевдотерминал. Например, если его номер 7, введите такую команду:

```
% echo "Hello, other window!" > /dev/pts/7
```

Заданная строка отобразится в первом окне. Когда терминальное окно будет закрыто, запись 7 исчезнет из каталога /dev/pts.

Если ввести команду ps в терминальном окне, работающем в текстовом режиме, окажется, что ему соответствует обычное терминальное устройство, а не псевдотерминал:

```
% ps -o pid, tty, cmd
PID TTY CMD
29325 tty1 -bash
29353 tty1 ps -o pid, tty, cmd
```

**6.7. Функция ioctl()**

Системный вызов ioctl() это универсальное средство управления аппаратными устройствами. Первым аргументом функции является дескриптор файла того устройства, которым требуется управлять. Второй аргумент это код запроса, обозначающего выполняемую операцию. Разным устройствам соответствуют разные запросы. В зависимости от запроса функции ioctl() могут потребоваться дополнительные аргументы.

Многие коды запросов перечислены на man-странице ioctl\_list. При работе с функцией ioctl() нужно хорошо понимать, как работает драйвер соответствующего устройства. В принципе, эти вопросы выходят за рамки нашей книги, но все же приведем небольшой пример.

*Листинг 6.2. (cdrom-eject.c) Извлечение компакт-диска из дисковод*

```
#include <fcntl.h>
#include <linux/cdrom.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char* argv[]) {
/* Открытие файла устройства, указанного в командной строке. */
int fd = open(argv[1], O_RDONLY);
/* Извлечение компакт-диска из дисковод. */
ioctl(fd, CDROMEJECT);
/* Закрытие файла. */
close(fd);
return 0;
}
```

В листинге 6.2 представлена короткая программа, которая запрашивает извлечение компакт-диска из дисковод CD-ROM. Программа принимает единственный аргумент командной строки: имя дисковод CD-ROM. Программа открывает файл устройства и вызывает функцию ioctl() с кодом запроса CDROMEJECT. Этот код определен в файле <linux/cdrom.h> и служит устройству указанием извлечь компакт-диск из дисковод.

Например, если в системе имеется IDE-дисковод CD-ROM, подключенный в качестве главного устройства к дополнительному IDE-контроллеру, соответствующий файл устройства будет называться /dev/hdc. Тогда компакт-диск извлекается из дисковод с помощью такой команды:

```
% ./cdrom-eject /dev/hdc
```

# Глава 7

## Файловая система /proc

Попробуйте запустить команду `mount` без аргументов она выдаст список файловых систем, смонтированных в настоящий момент. Среди прочих строк будет и такая:

```
none on /proc type proc (rw)
```

Она указывает на специальную файловую систему `/proc`. Поле `none` говорит о том, что эта система не связана с аппаратным устройством, например жестким диском. Она является своего рода "окном" в ядро Linux. Файлам в системе `/proc` не соответствуют реальные файлы на физическом устройстве. Это особые объекты, которые ведут себя подобно файлам, открывал доступ к параметрам, служебным структурам и статистической информации ядра. "Содержимое" таких файлов генерируется ядром динамически в процессе чтения из файла. Осуществляя запись в некоторые файлы, можно менять конфигурацию работающего ядра системы. Рассмотрим пример:

```
% ls -l /proc/version
-r--r--r-- 1 root root 0 Jan 17 18:09 /proc/version
```

Обратите внимание на то, что размер файла равен нулю. Поскольку содержимое файла создается ядром "на лету", понятие размера файла здесь неприменимо. Соответственно время модификации файла равно времени запуска команды.

Что находится в файле `/proc/version`? Он содержит строку, описывающую номер версии ядра Linux. Сюда входит информация, возвращаемая системным вызовом `uname()` (описан в разделе 8.15, "Функция `uname()`"), а также номер версии компилятора, с помощью которого было создано ядро. Чтение из файла `/proc/version` осуществляется самым обычным образом, например с помощью команды `cat`:

```
% cat /proc/version
Linux version 2.2.14-5.0 (root@porky.devel.redhat.com)
(gcc version egcs-2.91.66 19990314/Linux
(egcs-1.1.2 release)) #1 Tue Mar 7 21:07:39 EST 2000
```

Многие элементы файловой системы `/proc` описаны на man-странице `proc` (раздел 5). В этой главе будут рассмотрены те из них, которые чаще всего используются программистами и полезны при отладке.

Читатели, которых интересуют детали функционирования файловой системы `/proc`, могут просмотреть ее исходные коды в каталоге `/usr/src/linux/fs/proc/`.

### 7.1. Извлечение информации из файловой системы /proc

Большинство элементов файловой системы `/proc` выдает информацию в отформатированном виде. Например, файл `/proc/cpuinfo` содержит сведения о процессоре (или процессорах, если это многопроцессорный компьютер). Выходная информация представляется в виде таблицы значений, по одному на строку. Каждое значение сопровождается символическим идентификатором.

При обращении к файлу `/proc/cpuinfo` будет выдана примерно следующая информация:

```
% cat /proc/cpuinfo
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 5
model name : Pentium II (Deschutes)
stepping : 2
```

```
cpu MHz : 400.913520
cache size: 512 KB
fdiv_bug : no
hlt_bug : no
sep_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 mce cmov pat pse36 mmx fxsr
bogomips : 399.77
```

Интерпретация некоторых значений даны в разделе 7.3.1. "Центральный процессор". Если нужно получить одно из этих значений в программе, проще всего загрузить файл в память и просканировать его функцией `sscanf()`. В листинге 7.1 показано, как это сделать. В программе имеется функция `get_cpu_clock_speed()`, которая загружает файл `/proc/cpuinfo` и определяют частоту процессора.

***Листинг 7.1. (clock-speed.c) Определение частоты процессора путем анализа файла /proc/cpuinfo***

```
#include <stdio.h>
#include <string.h>

/* Определение частоты процессора в мегагерцах на
основании данных файла /proc/cpuinfo. В
многопроцессорной системе будет найдена частота
первого процессора. В случае ошибки возвращается нуль. */
float get_cpu_clock_speed() {
    FILE* fp;
    char buffer[1024];
    size_t bytes_read;
    char* match;
    float clock_speed;

    /* Загрузка всего файла /proc/cpuinfo в буфер. */
    fp = fopen("/proc/cpuinfo", "r");
    bytes_read = fread(buffer, 1, sizeof(buffer), fp);
    fclose(fp);
    /* Выход, если прочитать файл не удалось или буфер оказался
слишком маленьким. */
    if (bytes_read == 0 || bytes_read == sizeof(buffer))
        return 0;
    /* Буфер завершается нулевым символом. */
    buffer[bytes_read] = '\0';
    /* Поиск строки, содержащей метку "cpu MHz". */
    match = strstr(buffer, "cpu MHz");
    if (match == NULL)
        return 0;
    /* Анализ строки и выделение из нее значения частоты
процессора. */
    sscanf(match, "cpu MHz ; %f" &clock_speed);
    return clock_speed;
}
```

```
int main() {
printf("CPU clock speed: %4.0f Mhz\n",
get_cpu_clock_speed());
return 0;
}
```

Не забывайте о том, что имена, семантика и формат представления элементов файловой системы `/proc` меняются при обновлении ядра Linux. Программа должна вести себя корректно в случае, если нужный файл отсутствует или имеет иной формат.

## 7.2. Каталоги процессов

Файловая система `/proc` содержит по одному каталогу для каждого выполняющегося в данный момент процесса. Именем каталога является идентификатор процесса. [\[22\]](#) Каталоги появляются и исчезают динамически по мере запуска и завершения процессов. В каждом каталоге имеются файлы, предоставляющие доступ к различной информации о процессе. Собственно говоря, на основании этих каталогов файловая система `/proc` и получила свое имя.

В каталогах процессов находятся следующие файлы.

- `cmdline`. Содержит список аргументов процесса; описан в разделе 7.2.2, "Список аргументов процесса".

- `cwd`. Является символической ссылкой на текущий рабочий каталог процесса (задаётся, к примеру, функцией `chdir()`).

- `environ`. Содержит переменные среды процесса; описан в разделе 7.2.3, "Переменные среды процесса".

- `exe`. Является символической ссылкой на исполняемый файл процесса; описан в разделе 7.2.4, "Исполняемый файл процесса".

- `fd`. Является подкаталогом, в котором содержатся ссылки на файлы, открытые процессом: описан в разделе 7.2.5, "Дескрипторы файлов процесса".

- `maps`. Содержит информацию о файлах, отображаемых в адресном пространстве процесса. О механизме отображения файлов в памяти рассказывалось в главе 5. "Взаимодействие процессов". Для каждого такого файла выводится соответствующий диапазон адресов в адресном пространстве процесса, права доступа, имя файла и пр. К числу отображаемых файлов относятся исполняемый файл процесса, а также загруженные библиотеки.

- `root`. Является символической ссылкой на корневой каталог процесса (обычно это `/`). Корневой каталог можно сменить с помощью команды `chroot` или функции `chroot()`.

- `stat`. Содержит статистическую информацию о процессе. Эти же данные представлены в файле `status`, но здесь они находятся в неотформатированном виде и записаны в одну строку. Такой формат труден для восприятия, зато проще в плане синтаксического анализа.

- `statm`. Содержит информацию об использовании памяти процессом, описан в разделе 7.2.6, "Статистика использования процессом памяти".

- `status`. Содержит статистическую информацию о процессе, причем в отформатированном виде; описан в разделе 7.2.7, "Статистика процесса".

- `cpu`. Этот файл появляется только в симметричных многопроцессорных системах и содержит информацию об использовании процессорного времени (пользователями и системой).

Из соображений безопасности права доступа к некоторым файлам предоставляются только владельцу процесса и суперпользователю.

### 7.2.1. Файл `/proc/self`



В файловой системе /proc есть дополнительный элемент, позволяющий программам находить информацию о своем собственном процессе. Файл /proc/self является символической ссылкой на каталог, соответствующий текущему процессу. Естественно, содержимое ссылки меняется в зависимости от того, кто к ней обращается.

Например, программа, представленная в листинге 7.2, с помощью файла /proc/self определяет свой идентификатор процесса (это делается лишь в демонстрационных целях, гораздо проще пользоваться функцией getpid(), описанной в разделе 3.1.1, "Идентификаторы процессов"). Для чтения содержимого символической ссылки вызывается функция readlink() (описана в разделе 8.11, "Функция readlink(): чтение символических ссылок").

**Листинг 7.2. (get-pid.c) Получение идентификатора процесса из файла /proc/self**

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

/* Определение идентификатора вызывающего процесса
на основании символической ссылки /proc/self. */
pid_t get_pid_from_proc_self() {
    char target[32];
    int pid;
    /* Чтение содержимого символической ссылки. */
    readlink("/proc/self", target, sizeof(target));
    /* Адресатом ссылки является каталог, имя которого соответствует
идентификатору процесса. */
    sscanf(target, "%d", &pid);
    return (pid_t)pid;
}

int main() {
    printf("/proc/self reports process id %d\n",
(int)get_pid_from_proc_self());
    printf("getpid() reports process id %d\n", (int)getpid());
    return 0;
}
```

**7.2.2. Список аргументов процесса**

Файл cmdline в файловой системе /proc содержит список аргументов процесса (см. раздел 2.1.1. "Список аргументов"). Этот список представлен одной строкой, в которой аргументы отделяются друг от друга нулевыми символами. Большинство функций работы со строками предполагает, что нулевым символом оканчивается вся строка, поэтому они не смогут правильно обработать файл cmdline.

В листинге 2.1 приводилась программа, которая отображала переданный ей список аргументов. Теперь, когда мы узнали назначение файлов cmdline файловой системы /proc, можно написать программу, отображающую список аргументов другого процесса. Ее текст показан в листинге 7.3. Поскольку в строке файла cmdline может содержаться несколько нулевых символов, ее длину нельзя определить с помощью функции strlen() (она лишь подсчитывает число символов, пока не встретится нулевой символ). Приходится полагаться на

функцию `read()`, которая возвращает число прочитанных байтов.

### ***Листинг 7.3. (print-arg-list.c) Отображение списка аргументов указанного процесса***

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

/* Вывод списка аргументов (по одному в строке) процесса
с заданным идентификатором. */
void print_process_arg_list(pid_t pid) {
    int fd;
    char filename[24];
    char arg_list[1024];
    size_t length;
    char* next_arg;

    /* Определение полного имени файла cmdline
    для заданного процесса. */
    snprintf(filename, sizeof(filename), "/proc/%d/cmdline",
    (int)pid);
    /* Чтение содержимого файла. */
    fd = open(filename, O_RDONLY);
    length = read(fd, arg_list, sizeof(arg_list));
    close(fd);
    /* Функция read() не помещает в конец текста нулевой символ,
    поэтому его приходится добавлять отдельно. */
    arg_list[length] = '\0';
    /* Перебор аргументов. Аргументы отделяются друг от друга
    нулевыми символами. */
    next_arg = arg_list;
    while (next_arg < arg_list + length) {
        /* Вывод аргументов. Каждый из них оканчивается нулевым
        символом и потому интерпретируется как обычная строка. */
        printf("%s\n", next_arg);
        /* Переход к следующему аргументу. Поскольку каждый аргумент
        заканчивается нулевым символом, функция strlen() вычисляет
        длину отдельного аргумента, а не всего списка. */
        next_arg += strlen(next_arg) + 1;
    }
}
```

```
int main(int argc, char* argv[]) {
    pid_t pid = (pid_t)atoi(argv[1]);
    print_process_arg_list(pid);
    return 0;
}
```

Предположим, к примеру, что номер процесса системного демона `syslogd` равен 372.

```
% ps 372
PID TTY STAT TIME COMMAND
372 ? S 0:00 syslogd -m 0
% ./print-arg-list 372
syslogd
-m
```

0

В данном случае программа `print-arg-list`, сообщает о том, что демон `syslogd` вызван с аргументами `-m 0`.

### 7.2.3. Переменные среды процесса

Файл `environ` содержит список переменных среды, в которой работает процесс (см. раздел 2.1.6, "Среда выполнения"). Как и в случае файла `cmdline`, элементы списка разделяются нулевыми символами. Формат элемента таков: *ПЕРЕМЕННАЯ=значение*.

Представленная в листинге 7.4 программа является обобщением программы, которая была показана в листинге 2.3. В данном случае программа принимает в командной строке идентификатор процесса и отображает список его переменных среды, извлекаемый из файловой системы `/proc`.

#### *Листинг 7.4. (`print-environment.c`) Отображение переменных среды процесса*

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

/* Вывод переменных среды (по одной в строке) процесса
с заданным идентификатором. */
void print_process_environment(pid_t pid) {
    int fd;
    char filename[24];
    char environment[8192];
    size_t length;
    char* next_var;

    /* Определение полного имени файла environ
    для заданного процесса. */
    snprintf(filename, sizeof(filename), "/proc/%d/environ",
    (int)pid);
    /* Чтение содержимого файла. */
    fd = open(filename, O_RDONLY);
    length = read(fd, environment, sizeof (environment));
    close(fd);
    /* Функция read() не помещает в конец текста нулевой символ,
    поэтому его приходится добавлять отдельно. */
    environment[length] = '\0';

    /* Перебор переменных. Элементы списка отделяются друг от друга
    нулевыми символами. */
    next_var = environment;
    while (next_var < environment + length) {
        /* Вывод элементов списка. Каждый из них оканчивается нулевым
        символом и потому интерпретируется как обычная строка. */
        printf("%s\n", next_var);
        /* Переход к следующей переменной. Поскольку каждый элемент
        списка заканчивается нулевым символом, функция strlen()
```

```

вычисляет длину отдельного элемента, а не всего списка. */
next_var += strlen(next_var) + 1;
}
}

int main(int argc, char* argv[]) {
pid_t pid = (pid_t)atoi(argv[1]);
print_process_environment(pid);
return 0;
}

```

## 7.2.4. Исполняемый файл процесса

Файл `exe` указывает на исполняемый файл процесса. В разделе 2.1.1, "Список аргументов", говорилось о том, что имя исполняемого файла обычно передается в качестве первого элемента списка аргументов. Но это лишь распространенное соглашение. Программу можно запустить с произвольным списком аргументов. Файл `exe` файловой системы `/proc` это более надежный способ узнать, какой исполняемый файл запущен процессом.

Во многих программах путь ко вспомогательным файлам задан относительно исполняемого файла, поэтому важно знать, где именно он находится. Функция `get_executable_path()` в листинге 7.5 определяет путевое имя текущего исполняемого файла, проверяя символическую ссылку `/proc/self/exe`.

### *Листинг 7.5. (get-exe-path.c) Определение путевого имени текущего исполняемого файла*

```

#include <limits.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

/* Нахождение путевого имени текущего исполняемого файла.
   путевое имя помещается в строку BUFFER, длина которой
   равна LEN. Возвращается число символов в имени либо
   -1 в случае ошибки. */
size_t get_executable_path(char* buffer, size_t len) {
char* path_end;
/* чтение содержимого символической ссылки /proc/self/exe. */
if (readlink("/proc/self/exe", buffer, len) <= 0)
return -1;
/* Нахождение последней косой черты, отделяющей путевое имя. */
path_end = strrchr(buffer, '/');
if (path_end == NULL)
return -1;
/* Переход к символу, стоящему за последней косой чертой. */
++path_end;
/* Усечение полной строки до путевого имени. */
*path_end = '\0';
/* Длина путевого имени это число символов до последней
   косой черты. */
return (size_t)(path_end - buffer);
}

int main() {

```

```

char path[PATH_MAX];
get_executable_path(path, sizeof (path));
printf("this program is in the directory %e\n", path);
return 0;
}

```

## 7.2.5. Дескрипторы файлов процесса

Элемент `fd` файловой системы `/proc` это подкаталог, в котором содержатся записи обо всех файлах, открытых процессом. Каждая запись представляет собой символическую ссылку на файл или устройство. Через эти ссылки можно осуществлять чтение и запись данных. Имена ссылок соответствуют номерам дескрипторов.

Рассмотрим небольшой трюк. Откройте новое терминальное окно и найдите с помощью команды `ps` идентификатор процесса, соответствующий интерпретатору команд:

```

% ps
PID TTY TIME CMD
1261 pts/4 00:00:00 bash
2455 pts/4 00:00:00 ps

```

В данном случае процесс идентификатора команд (`bash`) имеет идентификатор 1261. Теперь откройте второе окно и просмотрите содержимое подкаталога `fd` этого процесса:

```

% ls -l /proc/1261/fd total 0
lrwx----- 1 samuel samuel 64 Jan 30 01:02 0 -> /dev/pts/4
lrwx----- 1 samuel samuel 64 Jan 30 01:02 1 -> /dev/pts/4
lrwx----- 1 samuel samuel 64 Jan 30 01:02 2 -> /dev/pts/4

```

(В выводе могут присутствовать дополнительные строки, соответствующие другим открытым файлам.) Вспомните в разделе 2.1.4, "Стандартный ввод-вывод", рассказывалось о том, что дескрипторы 0, 1 и 2 закрепляются за стандартными потоками ввода, вывода и ошибок соответственно. Таким образом, при записи в файл `/proc/1261/fd/1` данные будут направляться в устройство, связанное с потоком `stdout` интерпретатора команд, т.е. на псевдотерминал первого окна. Попробуйте ввести следующую команду

```
% echo "Hello, world." >> /proc/1261/fd/1
```

Сообщение "Hello, world." появится в первом окне.

В подкаталоге `fd` могут присутствовать ссылки и на другие файлы. В листинге 7.6 показана программа, которая открывает файл, указанный в командной строке, и переходит в бесконечный цикл.

### Листинг 7.6. (*open-and-spin.c*) Открытие файла для чтения

```

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

int main(int argc, char* argv[]) {
    const char* const filename = argv[1];
    int fd = open(filename, O_RDONLY);
    printf("in process %d, file descriptor %d is open to %s\n",
        (int)getpid(), (int)fd, filename);
    while (1);
    return 0;
}

```

```

}
Запустите программу в терминальном окне:
% ./open-and-spin /etc/fstab
in process 2570, file descriptor 3 is open to /etc/fstab
Теперь откройте другое окно и проверьте подкаталог fd процесса с указанным номером:
% ls -l /proc/2570/fd
total 0
lrwx----- 1 samuel samuel 64 Jan 30 01:30 0 -> /dev/pts/2
lrwx----- 1 samuel samuel 64 Jan 30 01:30 1 -> /dev/pts/2
lrwx----- 1 samuel samuel 64 Jan 30 01:30 2 -> /dev/pts/2
lr-x----- 1 samuel samuel 64 Jan 30 01:30 3 -> /etc/fstab

```

Как видите, появилась, ссылка 3, которая соответствует дескриптору файла /etc/fstab, открытого программой.

Программа может открывать дескрипторы не только файлов, но также сокетов и каналов. В таких случаях адресатом символической ссылки будет строка "socket" или "pipe", а не имя файла либо устройства.

### 7.2.6. Статистика использования процессом памяти

Файл `statm` содержит список из семи чисел, разделенных пробелами. Каждое число это счетчик числа страниц памяти, используемых процессом и попадающих в определенную категорию. Соответствующие категории перечислены ниже (в порядке следования счетчиков):

- общий размер процесса;
- размер резидентной части процесса;
- память, совместно используемая с другими процессами (например, загруженные библиотеки или нетронутые страницы, созданные в режиме "копирование при записи");
- текстовый размер процесса, т.е. размер сегмента кода исполняемого файла;
- размер совместно используемых библиотек, загруженных процессом;
- память, выделенная под стек процесса;
- число недействительных страниц, т.е. страниц памяти, которые были модифицированы программой.

### 7.2.7. Статистика процесса

Файл `status` содержит всевозможную информацию о процессе, отформатированную в понятном для пользователя виде. Сюда входит идентификатор процесса, идентификатор родительского процесса, реальный и эффективный идентификаторы пользователя и группы, статистика использования памяти, а также битовые маски, определяющие, какие сигналы перехватываются, игнорируются или блокируются.

## 7.3. Аппаратная информация

В файловой системе `/proc` есть ряд других элементов, позволяющих получить доступ к информации о системных аппаратных средствах. Обычно это интересно лишь системным администраторам, но иногда такая информация используется и в приложениях. Ниже описано несколько наиболее полезных файлов.

### 7.3.1. Центральный процессор

Как уже говорилось, файл `/proc/cpuinfo` содержит информацию о центральном процессоре (или процессорах, если их больше одного). В поле "processor" перечислены номера процессоров. В случае однопроцессорной системы там будет стоять 0. Благодаря полям "vendor\_id", "cpu family", "model" и "stepping" можно точно узнать модель и модификацию процессора. В поле "flags" показано, какие флат процессора установлены. Это самая важная информация. Она определяет, какие функции процессора доступны. Например, флаг "mmx" говорит о том, что поддерживаются расширенные инструкции MMX.<sup>[23]</sup>

Большая часть информации, содержащейся в файле `/proc/cpuinfo`, извлекается с помощью ассемблерной инструкции `cuid` процессоров семейства x86. С помощью этой низкоуровневой инструкции программы могут получать сведения о центральном процессоре. Подробнее узнать об этой инструкции можно в руководстве *IA-32 Intel Architecture Software Developer's Manual, Volume2: Instruction Set Reference*, доступном по адресу <http://developer.intel.com/design>.

Последний элемент файла, `bogomips`, характерен для Linux. Это показатель скорости работы процессора в поглощающем цикле (когда программы обращаются к процессору, но не выполняют вычислений). Он не отражает общую производительность процессора.

### 7.3.2. Аппаратные устройства

В файле `/proc/devices` содержится список старших номеров символьных и блочных устройств, имеющих в системе. Подробнее об этом рассказывалось в главе 6. "Устройства".

### 7.3.3. Шина PCI

В файле `/proc/pci` перечислены устройства, подключенные к шине (или шинам) PCI. Сюда входят реальные PCI-платы, а также устройства, встроенные в материнскую плату, плюс графические платы AGP. В каждой строке указан тип устройства, идентификатор устройства и его поставщика, имя устройства (если есть), информация о функциональных возможностях устройства и сведения о ресурсах PCI-шины, используемых устройством

### 7.3.4. Последовательные порты

Ф а й л `/proc/tty/driver/serial` содержит конфигурационную и статистическую информацию о последовательных портах. Эти порты нумеруются начиная с нуля.<sup>[24]</sup> Работать с настройками порта позволяет также команда `setserial`, но файл `/proc/tty/driver/serial`, помимо всего прочего, включает дополнительные статистические данные о счетчиках прерываний каждого порта.

Например, следующая строка описывает последовательный порт 1 (COM2 в Windows):

```
1: uart:16550A port:2F8 irq:3 baud:9600 tx:11 rx:0
```

Здесь говорится о том, что последовательный порт оснащен микросхемой UART 16550A, использует порт ввода-вывода 0x218 и прерывание 3 и работает со скоростью 9600 бод. Через этот порт было передано 11 запросов на прерывание и получено 0 таких запросов.

## 7.4. Информация о ядре

В файловой системе `/proc` есть много элементов, содержащих информацию о настройках и состоянии ядра. Некоторые из них находятся на верхнем уровне файловой системы, а некоторые

скрыты в каталоге /proc/sys/kernel.

### 7.4.1. Версия ядра

В файле /proc/version находится строка, описывающая номер версии и модификации ядра. В нее также включены сведения о создании ядра: имя пользователя, скомпилировавшего ядро, адрес компьютера, на котором это было сделано, дата компиляции и версия компилятора. Например:

```
% cat /proc/version
Linux version 2.2.14-5.0 (root@porky.devel.redhat.com)
(gcc version egcs-2.91.66 19990314/Linux (egcs-1.1.2 release))
#1 Tue Mar 7 21:07:39 EST 2000
```

Здесь сказано, что в системе используется ядро Linux версии 2.2.14, которое было скомпилировано программой EGCS версии 1.1.2 (эта программа является предшественницей широко распространенного в настоящее время пакета GCC).

Для наиболее важных параметров, а именно названия операционной системы и номера версии/модификации ядра, созданы отдельные записи в файловой системе /proc. Это файлы /proc/sys/kernel/ostype, /proc/sys/kernel/osrelease и /proc/sys/kernel/version.

```
% cat /proc/sys/kernel/ostype Linux
% cat /proc/sys/kernel/osrelease 2.2.14-5.0
% cat /proc/sys/kernel/version #1 Tue Mar 7 21:07:39 EST 2000
```

### 7.4.2. Имя компьютера и домена

В файлах /proc/sys/kernel/hostname и /proc/sys/kernel/domainname содержатся имя компьютера и имя домена соответственно. Эту же информацию возвращает функция uname(), описанная в разделе 8.15, "Функция uname()".

### 7.4.3. Использование памяти

Файл /proc/meminfo хранит сведения об использовании системной памяти. Указываются данные как о физической памяти, так и об области подкачки. Во второй и третьей строках значения даны в байтах, в остальных строках в килобайтах. Приведем пример:

```
% cat /proc/meminfo
total: used: free:shared: buffers: cached:
Mem: 529694720 519610368 10084352 82612224 10977280 82108416
Swap: 271392766 44003328 227389440
MemTotal: 517280 kB
MemFree: 9848 kB
MemShared: 80676 kB
Buffers: 10720 kB
Cached: 80184 kB
BigTotal: 0 kB
BigFree: 0 kB
SwapTotal: 265032 kB
SwapFree: 222060 kB
```

Как видите, в системе имеется 512 Мбайт ОЗУ, из которых 9 Мбайт свободно. Для области подкачки выделено 258 Мбайт, из которых свободно 216 Мбайт. В строке, соответствующей физической памяти, показаны три других значения.

■ В колонке "shared" отображается общий объем совместно используемой памяти,



- выделенной в системе.
- В колонке "buffers" отображается объем памяти, выделенной для буферов блочных устройств. Эти буферы используются драйверами устройств для временного хранения считываемых и записываемых блоков данных.
  - В колонке "cached" отображается объем памяти, выделенной для страничного кэш-буфера. В этом буфере сохраняются страницы файлов, отображаемых в памяти.
- Ту же самую информацию можно получить с помощью команды free.

## 7.5. Дисководы, точки монтирования и файловые системы

В файловой системе /proc находится также информация о присутствующих в системе дисковых устройствах и смонтированных на них файловых системах.

### 7.5.1. Файловые системы

Файл /proc/filesystems хранит информацию об известных ядру типах файловых систем. Этот список не очень полезен, так как он не полный: файловые системы могут подключаться и отключаться динамически в виде модулей ядра. В файле /proc/filesystems перечислены типы файловых систем, которые либо статически подключены к ядру, либо присутствуют в настоящий момент.

### 7.5.2. Диски и разделы

В файловой системе /proc находятся данные об устройствах, подключенных как к IDE-так и к SCSI-контроллерам (если таковые имеются). Обычно в каталоге /proc/ide есть один или два подкаталога (ide0 и ide1) для основного и дополнительного IDE-контроллеров системы.<sup>[25]</sup> В этих подкаталогах будут другие подкаталоги, которые соответствуют физическим устройствам, подключенным к контроллерам. В случае, если устройство не распознано системой, подкаталог не создается. В табл. 7.1 указаны путевые имена каталогов для четырех возможных IDE-устройств.

**Таблица 7.1. Каталоги, соответствующие четырем возможным IDE-устройствам**

Контроллер	Устройство	Подкаталог
Основной	Главное	/proc/ide/ide0/hda/
Основной	Подчиненное	/proc/ide/ide0/hdb/
Дополнительный	Главное	/proc/ide/ide1/hdc/
Дополнительный	Подчиненное	/proc/ide/ide1/hdd/

В каталоге каждого IDE-устройства есть несколько файлов, хранящих конфигурационные данные устройства. Перечислим наиболее важные из них.

- model. Содержит строку идентификации устройства.
- media. Описывает тип носителя. Возможные значения: disk, cdrom, tape, floppy и UNKNOWN.
- capacity. Определяет емкость устройства (в 512-байтовых блоках). Для дисководов CD-ROM значением будет  $2^{31}-1$ , а не емкость компакт-диска, вставленного в дисковод. Находящееся в данном файле значение представляет емкость всего физического диска. Емкость файловых

систем, содержащихся в разделах диска, будет меньше.

Ниже показано, как определить тип носителя и идентификатор главного устройства, подключенного к дополнительному IDE-контроллеру:

```
% cat /proc/ide/ide1/hdc/media  
cdrom  
% cat /proc/ide/ide1/hdc/model  
TOSHIBA CD-ROM XM-6702B
```

В данном случае это дисковод CDROM компании Toshiba.

Если в системе есть SCSI-устройства, в файле /proc/scsi/scsi будет находиться сводка их идентификаторов. Содержимое этого файла выглядит примерно так

```
% cat /proc/scsi/scsi  
Attached devices:  
Host: scsi0 Channel: 00 Id: 00 Lun: 00  
Vendor: QUANTUM Model: ATLAS_V__9_WLS Rev: 0230  
Type: Direct-Access ANSI SCSI revision: 03  
Host: scsi0 Channel: 00 Id: 04 Lun: 00  
Vendor: QUANTUM Model: QM39100TD-SW Rev: N491  
Type: Direct-Access ANSI SCSI revision: 02
```

В системе присутствует один одноканальный SCSI-контроллер (обозначен как scsi0), к которому подключены два дисковых накопителя Quantum со SCSI-номерами 0 и 4.

В файле /proc/partitions содержатся сведения о разделах распознанных дисковых устройств. Для каждого раздела указываются старший и младший номера, число однокилобайтовых блоков, а также имя устройства, соответствующего этому разделу.

Файл /proc/sys/dev/cdrom/info хранит различные данные о возможностях дисководов CD ROM. Записи этого файла не требуют особых пояснений:

```
% cat /proc/sys/dev/cdrom/info  
CD-ROM information, Id: cdrom.c 2.56 1999/09/09
```

```
drive name: hdc  
drive speed: 48  
drive # of slots: 0  
Can close tray: 1  
Can open tray: 1  
Can lock tray: 1  
Can change speed: 1  
Can select disk: 0  
Can read multisession: 1  
Can read MCN: 1  
Reports media changed: 1  
Can play audio: 1
```

### 7.5.3. Точки монтирования

В файле /proc/mounts находится перечень смонтированных файловых систем. Каждая строка соответствует одному дескриптору монтирования и содержит имя устройства, имя точки монтирования и прочие сведения. Та же самая информация хранится в обычном файле /etc/mtab, который автоматически обновляется командой mount.

Ниже перечислены элементы дескриптора монтирования.

■Первый элемент строки это имя смонтированного устройства. Для специальных файловых систем, например /proc, здесь стоит значение none.

■Второй элемент это имя точки монтирования, т.е. места в корневой файловой системе, где появится содержимое монтируемой файловой системы. Для самой корневой системы точка

монтирования обозначается символом /. Разделам подкачки соответствует точка монтирования swap.

■Третий элемент это тип файловой системы. В настоящее время на жестких дисках Linux в основном устанавливаются файловые системы типа ext2, но диски DOS и Windows могут монтироваться с файловыми системами других типов, например fat или vfat. Тип файловых систем большинства компакт-дисков iso9660. Список типов файловых систем приведен на man-странице команды mount.

■Четвертый элемент это флаги монтирования. Они указываются при добавлении точки монтирования. Пояснение этих флагов также дано на man-странице команды mount.

В файле /proc/mounts последние два элемента всегда равны нулю и никак не интерпретируются.

Подробнее о формате дескрипторов монтирования можно узнать на man-странице fstab. В Linux есть функции, позволяющие анализировать содержимое дескрипторов монтирования. За дополнительной информацией обратитесь к man-странице функции getmntent().

#### 7.5.4. Блокировки

В файле /proc/locks перечислены все блокировки файлов, установленные в настоящий момент в системе. Каждая строка соответствует одной блокировке.

Для блокировок, созданных функцией fcntl() (описана в разделе 8.3. "Функция fcntl(): блокировки и другие операции над файлами"), первыми двумя элементами строки будут слова POSIX и ADVISORY. Третьим элементом будет WRITE или READ, в зависимости от типа блокировки. Следующее число это идентификатор процесса, установившего блокировку. За ним идут три числа, разделенные двоеточиями. Это старший и младший номера устройства, на котором расположен файл, а также номер индексного дескриптора, оказывающий на местоположение файла в файловой системе. Оставшиеся числа используются внутри ядра и не представляют интереса.

Чтобы понять, как работает файл /proc/locks, запустите программу, приведенную в листинге 8.2. и поставьте блокировку записи на файл /tmp/test-file.

```
% touch /tmp/test-file
% ./lock-file /tmp/test-file
file /tmp/test-file
opening /tmp/test-file
locking
locked; hit enter to unlock...
```

В другом окне просмотрите содержимое файла /proc/locks:

```
% cat /proc/locks
ls POSIX ADVISORY WRITE 5467 08:05:181288 0 2147483647 d1b5f740
00000000 dfea7d40 00000000 00000000
```

В файле могут присутствовать дополнительные строки, если какие-то программы устанавливали свои блокировки. В данном случае идентификатор процесса программы lock-file 5467. Убедимся в этом с помощью команды ps:

```
% ps 5467
PID TTY STAT TIME COMMAND
5467 pts/28 S 0:00 ./lock-file /tmp/test-file
```

Заблокированный файл /tmp/test-file находится на устройстве со старшим и младшим номерами 8 и 5 соответственно. Это номера устройства /dev/sda5:

```
% df /tmp
Filesystem 1k-blocks Used Available Use% Mounted on
/dev/sda5 8459764 5094292 2935736 63% /
```

```
% ls -l /dev/sda5
brw-rw---- 1 root disk 8, 5 May 5 1998 /dev/sda5
На этом устройстве с файлом /tmp/test-file связав индексный дескриптор 181288:
% ls --inode /tmp/test-file
181288 /tmp/test-file
```

## 7.6. Системная статистика

Два элемента файловой системы `/proc` содержат полезную статистическую информацию. В файле `/proc/loadavg` находятся данные о загрузенности системы. Первые три показателя это число *активных задач* (выполняющихся процессов) за последние 1, 5 и 15 минут. Следующая строка отображает число *выполняемых задач* (процессов, запланированных к выполнению, а не заблокированных в каком-нибудь системном вызове) в данный момент времени и общее число процессов в системе. Последняя строка содержит идентификатор самого недавнего процесса.

В файле `/proc/uptime` отражено, сколько времени прошло с момента загрузки системы и сколько времени с тех пор система пребывала в неактивном состоянии. Оба показателя выражены в секундах и представлены числами с плавающей запятой:

```
% cat /proc/uptime
3248936.18 3072330.49
```

Программа, показанная в листинге 7.7, определяет общее время работы и время простоя системы и отображает эти значения в понятном формате.

### Листинг 7.7. (`print-uptime.c`) Отображение времени работы и времени простоя системы

```
#include <stdio.h>

/* Запись результата в стандартный выходной поток.
Параметр TIME это количество времени, а параметр LABEL --
короткая описательная строка. */
void print_time(char* label, long time) {
/* Константы преобразования. */
const long minute = 60;
const long hour = minute * 60;
const long day = hour * 24; /* Вывод результата. */
printf("%s: %ld days, %ld:%02ld:%02ld\n", label, time / day,
(time % day) / hour, (time % hour) / minute, time % minute);
}

int main() {
FILE* fp;
double uptime, idle_time;
/* Чтение показателей времени из файла /proc/uptime. */
fp = fopen("/proc/uptime", "r");
fscanf(fp, "%lf %lf\n", &uptime, &idle_time);
fclose(fp);
/* Форматирование и вывод. */
print_time("uptime ", (long)uptime);
print_time("idle time", (long)idle_time);
return 0;
}
```

Общее время работы системы отображают также команда `uptime` и функция `sysinfo()` (описана в разделе 8.14, "Функция `sysinfo()`: получение системной статистики"). Команда `uptime`

дополнительно выдает показатели средней загрузки, извлекаемые из файла /proc/loadavg.

Мы уже познакомились с большим количеством функций, реализующих различные системные задачи, например анализ командной строки, манипулирование процессами и отображение файлов в памяти. Если присмотреться повнимательнее, то окажется, что все они подпадают под две категории в зависимости от способа реализации.

■ *Библиотечная функция* это обычная функция, которая находится во внешней библиотеке, подключаемой к программе. Большинство рассмотренных нами функций содержится в стандартной библиотеке языка C, `libc`. Вызов библиотечной функции реализуется традиционно: ее аргументы помещаются в регистры процессора или в стек и управление передается в начало кода функции (этот код находится в библиотеке, загруженной в память).

■ *Системный вызов* реализован в ядре Linux. Аргументы вызова упаковываются и передаются ядру, которое берет на себя управление программой, пока вызов не завершится. Системный вызов это не обычная функция, и для передачи управления ядру требуется специальная подпрограмма. В GNU-библиотеке языка C (реализация стандартной библиотеки, имеющаяся в Linux) для системных вызовов созданы функции-оболочки, упрощающие обращение к ним. В качестве примеров системных вызовов можно привести низкоуровневые функции ввода-вывода, такие как `open()` и `read()`.

Совокупность системных вызовов Linux формирует основной интерфейс между программами и ядром. Каждому вызову соответствует некая элементарная операция или функция.

Некоторые системные вызовы оказывают очень большое влияние на систему. Например, есть вызовы, позволяющие завершить работу Linux, выделить системные ресурсы или запретить другим пользователям доступ к ресурсам. С такими вызовами связано ограничение: только процессы, выполняющиеся с привилегиями суперпользователя (учетная запись `root`), имеют право обращаться к ним. В противном случае вызовы завершатся ошибкой.

Внутри себя библиотечная функция может обращаться к другим функциям или системным вызовам.

В настоящее время в Linux есть около 200 системных вызовов. Их список находится в файле `/usr/include/asm/unistd.h`. Некоторые из них используются только внутри системы, а некоторые предназначены лишь для реализации специализированных библиотечных функций. В этой главе будут рассмотрены те системные вызовы, которые чаще всего используются системными программистами.

### 8.1. Команда `strace`

Прежде чем изучать системные вызовы, полезно познакомиться с командой `strace`, которая отслеживает выполнение заданной программы, выводя список всех запрашиваемых системных вызовов и получаемых сигналов. Эта команда ставится в начале строки вызова программы, например:<sup>[26]</sup>

```
% strace hostname
```

В результате будет получено несколько экранов выходной информации. Каждая строка соответствует одному системному вызову. В строке указываются имя вызова, его аргументы (или их сокращенные обозначения, если аргументы слишком длинные) и возвращаемое значение. По возможности команда `strace` старается отображать не числовые значения, а символические

константы. Показываются также поля структур, переданных по указателю. Вызовы обычных функций не регистрируются.

В случае команды `strace hostname` первая строка сообщает о системном вызове `execve()`, загружающем программу `hostname`:<sup>[27]</sup>

```
execve("/bin/hostname", ["hostname"], [/* 49 vars */]) = 0
```

Первый аргумент это имя запускаемой программы. За ним идет список аргументов, состоящий из одного элемента. Дальше указан список переменных среды, который команда `strace` опустила для краткости.

Следующие примерно 30 строк отражают работу механизма загрузки стандартной библиотеки языка C из библиотечного файла. Ближе к концу наконец встречаются системные вызовы, связанные непосредственно с работой программы. Системный вызов `uname()` запрашивает имя компьютера у ядра:

```
uname({sys="Linux", node="myhostname", ...}) = 0
```

Заметьте, что команда `strace` показала метки полей структуры, в которой хранятся аргументы. Эта структура заполняется в системном вызове: `Linux` помещает в поле `sys` имя операционной системы, а в поле `node` имя компьютера. Функция `uname()` будет описана ниже, в разделе 8.15. "Функция `uname()`".

Системный вызов `write()` выводит полученные результаты на экран. Вспомните, что дескриптор 1 соответствует стандартному выходному потоку. Третий аргумент это количество отображаемых символов. Функция возвращает число действительно записанных символов.

```
write(1, "myhostname\n", 11) = 11
```

Эта строка может отобразиться искаженной, поскольку вывод программы `hostname` смешивается с результатами работы команды `strace`. Если запускаемая программа создает слишком много выходных данных, лучше перенаправить вывод команды `strace` в файл с помощью опции `-o имя_файла`.

## 8.2. Функция `access()`: проверка прав доступа к файлу

Функция `access()` определяет, имеет ли вызывающий ее процесс право доступа к заданному файлу. Функция способна проверить любую комбинацию привилегий чтения, записи и выполнения, а также факт существования файла.

Функция `access()` принимает два аргумента: путь к проверяемому файлу и битовое объединение флагов `R_OK`, `W_OK` и `X_OK`, соответствующих правам чтения, записи и выполнения. При наличии у процесса всех необходимых привилегий функция возвращает 0. Если файл существует, а нужные привилегии на доступ к нему у процесса отсутствуют, возвращается -1 и в переменную `errno` записывается код ошибки `EACCES` (или `EROFS`, если проверяется право записи в файл, который расположен в файловой системе, смонтированной только для чтения).

Если второй аргумент равен `F_OK`, функция `access()` проверяет лишь факт существования файла. В случае обнаружения файла возвращается 0, иначе -1 (в переменную `errno` помещается также код ошибки `ENOENT`). Когда один из каталогов на пути к файлу недоступен, в переменную `errno` будет помещён код `EACCES`.

Программа, показанная в листинге 8.1, с помощью функции `access()` проверяет существование файла и определяет, разрешен ли к нему доступ на чтение/запись. Имя файла задается в командной строке.

```

#include <errno.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char* argv[]) {
char* path = argv[1];
int rval;

/* Проверка существования файла. */
rval = access(path, F_OK);
if (rval == 0)
printf("%s exists\n", path);
else {
if (errno == ENOENT)
printf("%s does not exist\n", path);
else if (errno == EACCES)
printf("%s is not accessible\n", path);
return 0;
}

/* Проверка права доступа. */
rval = access(path, R_OK);
if (rval == 0)
printf("%s is readable\n", path);
else
printf("%s is not readable (access denied)\n", path);

/* проверка права записи. */
rval = access(path, W_OK);
if (rval == 0)
printf("%s is writable\n", path);
else if (errno == EACCES)
printf("%s is not writable (access denied)\n", path);
else if (errno == EROFS)
printf("%s is not writable (read-only filesystem)\n",
path);
return 0;
}

```

Вот как, к примеру, проверить права доступа к файлу README, расположенному на компакт-диске:

```

% ./check-access /mnt/cdrom/README
/mnt/cdrom/README exists
/mnt/cdrom/README is readable
/mnt/cdrom/README is not writable (read-only filesystem)

```

### 8.3. Функция `fcntl()`: блокировки и другие операции над файлами

Функция `fcntl()` это точка доступа к нескольким особым операциям над файлами. Первым аргументом функции является дескриптор файла, вторым указывается код операции. Для некоторых операций требуется также дополнительный, третий аргумент. В этом разделе описана наиболее распространенная операция, выполняемая с помощью функции `fcntl()`: блокирование файлов.

Функция `fcntl()` позволяет программе поставить на файл блокировку чтения иди записи. Это напоминает применение исключающих семафоров, которые описывались в главе 5, "Взаимодействие процессов". Блокировка чтения ставится на файл, доступный для чтения.



Соответственно блокировка записи ставится на файл, доступный для записи. Несколько процессов могут удерживать блокировку чтения одного и того же файла, но только одному процессу разрешено ставить блокировку записи. Файл не может быть одновременно заблокирован и для чтения, и для записи. Учтите, что наличие блокировки не мешает другим процессам открывать файл и осуществлять чтение/запись его данных, если только они сами не попытаются вызвать функцию `fcntl()`.

Прежде чем ставить блокировку на файл, необходимо создать и обнулить структуру типа `flock`. В поле `l_type` должна быть записана константа `F_RDLCK` в случае блокировки чтения и константа `F_WRLCK` в случае блокировки записи. Далее следует вызвать функцию `fcntl()`, передав ей дескриптор файла, код операции `F_SETLKW` и указатель на структуру типа `flock`. Если аналогичная блокировка уже была поставлена другим процессом, функция `fcntl()` перейдет в режим ожидания, пока "мешающая" ей блокировка не будет снята.

В листинге 8.2 показана программа, которая открывает для записи указанный файл, а затем ставит на него блокировку записи. Программа ждет нажатия клавиши `<Enter>`, после чего снимает блокировку и закрывает файл.

### *Листинг 8.2. (lock-file.c) Установка блокировки записи с помощью функции `fcntl()`*

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char* argv[]) {
    char* file = argv[1];
    int fd;
    struct flock lock;

    printf("opening %s\n", file);
    /* Открытие файла. */
    fd = open(file, O_WRONLY);
    printf("locking\n");
    /* инициализация структуры flock. */
    memset(&lock, 0, sizeof(lock));
    lock.l_type = F_WRLCK;
    /* Установка блокировки записи. */
    fcntl(fd, F_SETLKW, &lock);

    printf("locked; hit Enter to unlock... ");
    /* Ожидание нажатия клавиши <Enter>. */
    getchar();

    printf("unlocking\n");
    /* Снятие блокировки. */
    lock.l_type = F_UNLCK;
    fcntl(fd, F_SETLKW, &lock);

    close(fd);
    return 0;
}
```

Скомпилируйте программу и запустите ее с каким-нибудь тестовым файлом, скажем, `/tmp/test-file`:

```
% cc -o lock-file lock-file.c
% touch /tmp/test-file
% ./lock-file /tmp/test-file
opening /tmp/test-file
locking
locked; hit Enter to unlock...
```

Теперь откройте другое окно и вызовите программу еще раз с тем же файлом:

```
% ./lock-file /tmp/test-file
opening /tmp/test-file
locking
```

Пытаясь поставить блокировку на файл, программа сама окажется заблокированной.

Вернитесь в первое окно и нажмите <Enter>:

```
unlocking
```

В результате программа, запущенная во втором окне, немедленно продолжит свою работу. Если необходимо, чтобы функция `fcntl()` не переходила в режим ожидания в случае, когда блокировку поставить невозможно, задайте в качестве кода операции константу `F_SETLCK`, а не `F_SETLKW`. Если функция обнаружит, что запрашиваемый файл уже заблокирован, она немедленно вернет -1.

В Linux имеется системный вызов `flock()`, также реализующий операцию блокирования файла. Но у функции `fcntl()` есть большое преимущество: она работает с файловыми системами NFS<sup>[28]</sup> (при условии, что сервер NFS имеет относительно недавнюю версию и сконфигурирован правильно). Так что, имея доступ к двум компьютерам, которые монтируют одну и ту же файловую систему через NFS, можно повторить показанный выше пример на двух разных машинах.

## 8.4. Функции `fsync()` и `fdatasync()`: очистка дисковых буферов

В большинстве операционных систем при записи в файл данные не передаются на диск немедленно. Вместо этого операционная система помещает их в резидентный кэш-буфер с целью сокращения числа обращений к диску и повышения оперативности программы. Когда буфер заполнится или произойдет какое-нибудь другое событие (например, истечет определенный промежуток времени), система запишет содержимое буфера на диск в ходе одной непрерывной операции.

В Linux тоже поддерживается такой тип кэширования. Обычно он способствует существенному повышению производительности. Но он же делает ненадежными программы, зависящие от целостности дисковых данных. Если система внезапно выйдет из строя, например вследствие сбоя ядра или отключения питания, любые данные, находящиеся в памяти и еще не записанные на диск, будут потеряны.

Предположим, создается программа обработки транзакций, которая ведет журнальный файл. В этот файл помещаются записи обо всех транзакциях, завершившихся на данный момент, чтобы в случае системного сбоя можно было восстановить целостность данных. Очевидно, не менее важна и целостность самого журнального файла: как только транзакция завершена, запись о ней должна быть немедленно занесена в дисковый файл.

Для реализации такого поведения ОС Linux предоставляет системный вызов `fsync()`. Эта функция принимает один аргумент дескриптор записываемого файла и принудительно переносит на диск все данные этого файла, находящиеся в кэш-буфере. Функция не завершается до тех пор, пока данные не окажутся на диске.

В листинге 8.3 показана функция, использующая данный системный вызов. Она записывает переданную ей строку в журнальный файл.

### Листинг 8.3. (write\_journal\_entry.c) Запись строки в журнальный файл с последующей синхронизацией

```
#include <fcntl.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

const char* journal_filename = "journal.log";

void write_journal_entry(char* entry) {
    int fd =
    open(journal_filename,
    O_WRONLY | O_CREAT | O_APPEND, 0660);
    write(fd, entry, strlen(entry));
    write(fd, "\n", 1);
    fsync(fd);
    close(fd);
}
```

Аналогичное действие выполняет другой системный вызов: `fdatasync()`. Но если функция `fsync()` гарантирует, что дата модификации файла будет обновлена, то функция `fdatasync()` этого не делает, а лишь гарантирует запись данных. В принципе это означает, что функция `fdatasync()` способна выполняться быстрее, чем `fsync()`, так как ей требуется выполнить одну операцию записи на диск, а не две. Но в настоящее время в Linux обе функции работают одинаково, обновляя дату модификации.

Файл можно также открыть в режиме *синхронного ввода-вывода*, при котором все операции записи будут немедленно фиксироваться на диске. Для этого в функции `open()` следует указать флаг `O_SYNC`.

## 8.5. Функции `getrlimit()` и `setrlimit()`: лимиты ресурсов

Функции `getrlimit()` и `setrlimit()` позволяют процессу определять и задавать лимиты использования системных ресурсов. Аналогичные действия выполняет команда `ulimit`, которая ограничивает доступ запускаемых пользователем программ к ресурсам.

У каждого ресурса есть два лимита: *жесткий* и *нежесткий*. Второе значение никогда не может быть больше первого, и лишь процессы с привилегиями супер пользователя имеют право менять жесткий лимит. Обычно приложение уменьшает нежесткий лимит, ограничивая потребление системных ресурсов.

Обе функции принимают два аргумента: код, задающий тип ограничения, и указатель на структуру типа `rlimit`. Функция `getrlimit()` заполняет поля этой структуры, тогда как функция `setrlimit()` проверяет их и соответствующим образом меняет лимит. У структуры `rlimit` два поля: в поле `rlim_cur` содержится значение нежесткого лимита, а в поле `rlim_max` значение жесткого лимита.

Ниже перечислены коды наиболее полезных лимитов, допускающих возможность изменения.

■ **RLIMIT\_CPU**. Это максимальный интервал времени центрального процессора (в секундах), занимаемый программой. Именно столько времени отводится программе на доступ к процессору. В случае превышения данного ограничения программа будет завершена по сигналу

SIGXCPU.

■RLIMIT\_DATA. Это максимальный объем памяти, который программа может запросить для своих данных. Запросы на дополнительную память будут отвергнуты системой.

■RLIMIT\_NPROC. Это максимальное число дочерних процессов, которые могут быть запущены пользователем. Если процесс вызывает функцию `fork()`, а лимит уже исчерпан, функция завершается ошибкой.

■RLIMIT\_NOFILE. Это максимальное число файлов, которые могут быть одновременно открыты процессом.

Программа, приведенная в листинге 8.4, задает одnoseкундный лимит использования центрального процессора, после чего переходит в бесконечный цикл. Как только программа превышает установленный ею же лимит, ОС Linux уничтожает ее.

#### ***Листинг 8.4. (limit-cpu.c) Задание ограничения на использование нейтрального процессора***

```
#include <sys/resource.h>
#include <sys/time.h>
#include <unistd.h>

int main() {
    struct rlimit rl;

    /* Определяем текущие лимиты. */
    getrlimit(RLIMIT_CPU, &rl);
    /* Ограничиваем время доступа к процессору
    одной секундой. */
    rl.rlim_cur = 1;
    setrlimit(RLIMIT_CPU, &rl);
    /* Переходим в бесконечный цикл. */
    while(1);

    return 0;
}
```

Когда программа завершается по сигналу SIGXCPU, интерпретатор команд выдает поясняющее сообщение:

```
% ./limit_cpu
CPU time limit exceeded
```

## **8.6. Функция `getrusage()`: статистика процессов**

Функция `getrusage()` запрашивает у ядра статистику работы процессов. Если первый аргумент функции равен `RUSAGE_SELF`, процесс получит информацию о самом себе. Если же первым аргументом является константа `RUSAGE_CHILDREN`, будет выдана информация обо всех его завершившихся дочерних процессах. Второй аргумент это указатель на структуру типа `rusage`, в которую заносятся статистические данные.

Перечислим наиболее интересные поля этой структуры.

■`ru_utime`. Здесь находится структура типа `timeval`, в которой указано, сколько пользовательского времени (в секундах) ушло на выполнение процесса. Это время, затраченное центральным процессором на выполнение программного кода, а не системных вызовов.

■`ru_stime`. Здесь находится структура типа `timeval`, в которой указано, сколько

системного времени (в секундах) ушло на выполнение процесса. Это время, затраченное центральным процессором на выполнение системных вызовов от имени данного процесса.

■ `ru_maxrss`. Это максимальный объем физической памяти, которую процесс занимал в какой-то момент своего выполнения.

В листинге 8.5 приведена функция, которая показывает, сколько пользовательского и системного времени потребил текущий процесс.

**Листинг 8.5. (*printf-cpu-times.c*) Определение пользовательского и системного времени, затраченного на выполнение текущего процесса**

```
#include <stdio.h>
#include <sys/resource.h>
#include <sys/time.h>
#include <unistd.h>

void print_cpu_time() {
    struct rusage usage;
    getrusage(RUSAGE_SELF, &usage);
    printf("CPU time: %ld.%06ld sec user, %ld.%06ld sec system\n",
        usage.ru_utime.tv_sec, usage.ru_utime.tv_usec,
        usage.ru_stime.tv_sec, usage.ru_stime.tv_usec);
}
```

## 8.7, Функция `gettimeofday()`: системные часы

Функция `gettimeofday()` определяет текущее системное время. В качестве аргумента она принимает структуру типа `timeval`, в которую записывается значение времени (в секундах), прошедшее с начала эпохи UNIX (1-е января 1970 г., полночь по Гринвичу). Это значение разделяется на два поля. В поле `tv_sec` хранится целое число секунд, а в поле `tv_usec` дополнительное число микросекунд. У функции есть также второй аргумент, который должен быть равен `NULL`. Функция объявлена в файле `<sys/time.h>`.

Результат, возвращаемый функцией `gettimeofday()`, мало подходит для отображения на экране, поэтому существуют библиотечные функции `localtime()` и `strftime()`, преобразующие это значение в нужный формат. Функция `localtime()` принимает указатель на число секунд (поле `tv_sec` структуры `timeval`) и возвращает указатель на структуру типа `tm`. Эта структура содержит поля, заполняемые параметрами времени в соответствии с локальным часовым поясом:

- `tm_hour`, `tm_min`, `tm_sec` текущее время (часы, минуты, секунды);
- `tm_year`, `tm_mon`, `tm_day` год, месяц, день;
- `tm_wday` день недели (значение 0 соответствует воскресенью);
- `tm_yday` день года;
- `tm_isdst` флаг, указывающий, учтено ли летнее время.

Функция `strftime()` на основании структуры `tm` создает строку, отформатированную по заданному правилу. Формат напоминает тот, что используется в функции `printf()`: указывается строка с кодами, определяющими включаемые поля структуры. Например, форматная строка вида

```
"%Y-%m-%d %H:%M:%S"
```

соответствует такому результату:

```
2001-01-14 13:09:42
```

Функции `strftime()` необходимо задать указатель на текстовый буфер, куда будет помещена полученная строка, длину буфера, строку формата и указатель на структуру типа `tm`. Следует учесть, что ни функция `localtime()`, ни функция `strftime()` не учитывают дробную часть текущего времени (поле `tv_usec` структуры `timeval`). Об этом должен позаботиться программист.

Объявления функций `localtime()` и `strftime()` находятся в файле `<time.h>`.

Программа, показанная в листинге 8.6, отображает текущие дату и время с точностью до миллисекунды.

#### **Листинг 8.6. (`print-time.c`) Отображение даты и времени**

```
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <unistd.h>

void print_time() {
    struct timeval tv;
    struct tm* ptm;
    char time_string[40];
    long milliseconds;

    /* Определение текущего времени и преобразование полученного
    значения в структуру типа tm. */
    gettimeofday(&tv, NULL);
    ptm = localtime(&tv.tv_sec);
    /* Форматирование значения даты и времени с точностью
    до секунды. */
    strftime(time_string, sizeof(time_string),
    "%Y-%m-%d %H:%M:%S", ptm);
    /* Вычисление количества миллисекунд. */
    milliseconds = tv.tv_usec / 1000;
    /* Отображение даты и времени с указанием
    числа миллисекунд. */
    printf("%s.%03ld\n", time_string, milliseconds);
}
```

### **8.8. Семейство функций `mlock()`: блокирование физической памяти**

Функции семейства `mlock()` позволяют программе блокировать часть своего адресного пространства в физической памяти. Заблокированные страницы не будут выгружены операционной системой в раздел подкачки, даже если программа долго к ним не обращалась.

Блокирование физической памяти важно в программах реального времени, поскольку задержки, связанные с выгрузкой и подкачкой страниц, могут оказаться слишком длинными или возникать в самый неподходящий момент. Приложения, заботящиеся о безопасности своих данных, могут устанавливать запрет на выгрузку важных данных в файл подкачки, в котором они станут доступны злоумышленнику после завершения программы.

Чтобы заблокировать область памяти, достаточно вызвать функцию `mlock()`, передав ей указатель на начало области и значение длины области. ОС Linux разбивает память на страницы и соответственно блокирует ее постранично: любая страница, которую захватывает (хотя бы частично) заданная в функции `mlock()` область памяти, окажется заблокированной. Определить

размер системной страницы позволяет функция `getpagesize()`. В Linux-системах, работающих на платформе x86, эта величина составляет 4 Кбайт.

Вот как можно выделить и заблокировать 32 Мбайт оперативной памяти:

```
const int alloc_size = 32 * 1024 * 1024;
char* memory = malloc(alloc_size);
mlock(memory, alloc_size);
```

Выделение страницы и блокирование ее с помощью функции `mlock()` еще не означает, что эта страница будет предоставлена данному процессу, поскольку выделение памяти может происходить в режиме копирования при записи.<sup>[29]</sup> Следовательно, каждую страницу необходимо проинициализировать:

```
size_t i;
size_t page_size = getpagesize();
for (i = 0; i < alloc_size; i += page_size)
    memory[i] = 0;
```

Процессу, осуществляющему запись на страницу, операционная система предоставит в монопольное использование ее уникальную копию.

Для разблокирования области памяти следует вызвать функцию `munlock()`, передав ей те же аргументы, что и функции `mlock()`.

Функция `mlockall()` блокирует все адресное пространство программы и принимает единственный флаговый аргумент. Флаг `MCL_CURRENT` означает блокирование всей выделенной на данный момент памяти, но не той, что будет выделяться позднее. Флаг `MCL_FUTURE` задает блокирование всех страниц, выделенных после вызова функции `mlockall()`. Сочетание флагов `MCL_CURRENT | MCL_FUTURE` позволяет блокировать всю память программы, как текущую, так и будущую.

Блокирование больших объемов памяти, особенно с помощью функции `mlockall()`, несет потенциальную угрозу всей системе. Несправедливое распределение оперативной памяти приведет к катастрофическому снижению производительности системы, так как остальным процессам придется сражаться друг с другом за небольшой "ключок" памяти, вследствие чего они будут постоянно выгружаться на диск и загружаться обратно. Может даже возникнуть ситуация, когда оперативная память закончится и система начнет уничтожать процессы. По этой причине функции `mlock()` и `mlockall()` доступны лишь суперпользователю. Если какой-нибудь другой пользователь попытается вызвать одну из этих функций, она вернёт значение -1, а в переменную `errno` будет записан код `ENOMEM`.

Функция `munlockall()` разблокирует всю память текущего процесса.

Контролировать использование памяти удобнее всего с помощью команды `top`. В колонке `SIZE` ее выходных данных показывается размер виртуального адресного пространства каждой программы (общий размер сегментов кода, данных и стека с учетом выгруженных страниц). В колонке `RSS` приводится объем резидентной части программы. Сумма значений в столбце `RSS` не может превышать имеющийся объем ОЗУ, а суммарный показатель по столбцу `SIZE` не может быть больше 2 Гбайт (в 32-разрядных версиях Linux).

Функции семейства `mlock()` объявлены в файле `<sys/mman.h>`.

## 8.9. Функция `mprotect()`: задание прав доступа к памяти

В разделе 5.3, "Отображение файлов в памяти", рассказывалось о том, как осуществляется отображение файла в памяти. Вспомните, что третьим аргументом функции `mmap()` является битовое объединение флагов доступа: флаги `PROT_READ`, `PROT_WRITE` и `PROT_EXEC` задают права чтения, записи и выполнения файла, а флаг `PROT_NONE` означает запрет доступа. Если программа

пытается выполнить над отображаемым файлом недопустимую операцию, ей посылается сигнал SIGSEGV (нарушение сегментации), который приводит к завершению программы.

После того как файл был отображен в памяти, изменить права доступа к нему позволяет функция `mprotect()`. Ее аргументами является адрес области памяти, размер области и новый набор флагов доступа. Область должна состоять из целых страниц, т.е. начинаться и заканчиваться на границе между страницами.

### ***Корректное выделение памяти***

Учтите, что память, выделяемая функцией `malloc()`, обычно не выравнивается по границе страниц, даже если размер области кратен размеру страницы. Если требуется защищать память, выделяемую функцией `malloc()`, нужно запросить более крупный блок, а затем найти в нем участок, выровненный по границе страниц.

Кроме того, с помощью функции `mmap()` можно обойти функцию `malloc()` и запрашивать память непосредственно у ядра Linux.

Предположим, к примеру, что программа выделяет страницу, отображая в памяти файл `/dev/zero`. Память инициализируется как для чтения, так и для записи:

```
int fd = open("/dev/zero", O_RDONLY);
char* memory =
mmap(NULL, page_size, PROT_READ | PROT_WRITE,
MAP_PRIVATE, fd, 0);
close(fd);
```

Далее программа запрещает запись в эту область памяти, вызывая функцию `mprotect()`:

```
mprotect(memory, page_size, PROT_READ);
```

Существует оригинальная методика контроля памяти: можно защитить область памяти с помощью функций `mmap()` и `mprotect()`, а затем обрабатывать сигнал SIGSEGV, посылаемый при попытке доступа к этой памяти. Эта методика иллюстрируется в листинге 8.7.

### ***Листинг 8.7. (mprotect.c) Обнаружение попыток доступа к памяти благодаря функции mprotect()***

```
#include <fcntl.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

static int alloc_size;
static char* memory;

void segv_handler(int signal_number) {
    printf("memory accessed!\n");
    mprotect(memory, alloc_size, PROT_READ | PROT_WRITE);
}

int main() {
```



```

int fd;
struct sigaction sa;

/* Назначение функции segv_handler() обработчиком сигнала
SIGSEGV. */

memset(&sa, 0, sizeof(sa));
sa.sa_handler = &segv_handler;
sigaction(SIGSEGV, &sa, NULL);

/* Выделение одной страницы путем отображения в памяти файла
/dev/zero. Сначала память доступна только для записи. */
alloc_size = getpagesize();
fd = open("/dev/zero", O_RDONLY);
memory =
mmap(NULL, alloc_size, PROT_WRITE, MAP_PRIVATE, fd, 0);
close(fd);
/* Запись на страницу для получения ее копии в частное
использование. */
memory[0] = 0;
/* Запрет на запись в память. */
mprotect(memory, alloc_size, PROT_NONE);

/* Попытка записи в память. */
memory[0] = 1;

/* Удаление памяти. */
printf("all done\n");
munmap(memory, alloc_size);
return 0;
}

```

Программа работает по следующей схеме.

1.Задается обработчик сигнала SIGSEGV.

2.Файл /dev/zero отображается в памяти, из которой выделяется одна страница. В эту страницу записывается инициализирующее значение, благодаря чему программе предоставляется частная копия страницы.

3.Программа защищает память, вызывая функцию mprotect() с флагом PROT\_NONE.

4.Когда программа впоследствии обращается к памяти, Linux посылает ей сигнал SIGSEGV, который обрабатывается в функции segv\_handler(). Обработчик сигнала отменяет защиту памяти, разрешая выполнить операцию записи.

5.Программа удаляет область память с помощью функции munmap().

## 8.10. Функция nanosleep(): высокоточная пауза

Функция nanosleep() является более точной версией стандартной функции sleep(), принимая указатель на структуру типа timespec, где время задается с точностью до наносекунды, а не секунды. Правда, особенности работы ОС Linux таковы, что реальная точность оказывается равной 10 мс, но это все равно выше, чем в функции sleep(). Функцию nanosleep() можно использовать в приложениях, где требуется запускать различные операции с короткими интервалами между ними.

В структуре timespec имеются два поля:

■ tv\_sec целое число секунд;

■ `tv_nsec` дополнительное число миллисекунд (должно быть меньше, чем  $10^9$ ).

Работа функции `nanosleep()`, как и функции `sleep()`, прерывается при получении сигнала. При этом функция возвращает значение -1, а в переменную `errno` записывается код `EINTR`. Но у функции `nanosleep()` есть важное преимущество. Она принимает дополнительный аргумент еще один указатель на структуру `timespec`, в которую (если указатель не равен `NULL`) заносится величина оставшегося интервала времени (т.е. разница между запрашиваемым и прошедшим промежутками времени). Благодаря этому можно легко возобновлять прерванные операции ожидания.

В листинге 8.8 показана альтернативная реализация функции `sleep()`. В отличие от стандартного системного вызова эта функция может принимать дробное число секунд и возобновлять операцию ожидания в случае прерывания по сигналу.

### ***Листинг 8.8. (better\_sleep.c) Высокоточная реализация функции sleep()***

```
#include <errno.h>
#include <time.h>

int better_sleep(double sleep_time) {
    struct timespec tv;
    /* Заполнение структуры timespec на основании указанного числа
    секунд. */
    tv.tv_sec = (time_t)sleep_time;
    /* добавление неучтенных выше наносекунд. */
    tv.tv_nsec = (long)((sleep_time - tv.tv_sec) * 1e+9);

    while (1) {
        /* Пауза, длительность которой указана в переменной tv.
        В случае прерывания по сигналу величина оставшегося
        промежутка времени заносится обратно в переменную tv. */
        int rval = nanosleep(&tv, &tv);
        if (rval == 0)
            /* пауза успешно окончена. */
            return 0;
        else if (errno == EINTR)
            /* Прерывание по сигналу. Повторная попытка. */
            continue;
        else
            /* Какая-то другая ошибка. */
            return rval;
    }
    return 0;
}
```

## **8.11. Функция readlink(): чтение символических ссылок**

Функция `readlink()` определяет адресата символической ссылки. Она принимает три аргумента: путь к символической ссылке, буфер для записи адресата и длина буфера. Как ни странно, путевое имя, помещаемое в буфер, не завершается нулевым символом. Но поскольку в третьем аргументе возвращается длина буфера, добавить этот символ несложно.

Если первый аргумент не является символической ссылкой, функция `readlink()` возвращает -1, а в переменную `errno` записывается константа `EINVAL`.

Программа, представленная в листинге 8.9, показывает адресата символической ссылки, заданной в командной строке.

### *Листинг 8.9. (print-symlink.c) Отображение адресата символической ссылки*

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

int main(int argc, char* argv[]) {
    char target_path[256];
    char* link_path = argv[1];

    /* Попытка чтения адресата символической ссылки. */
    int len =
        readlink(link_path, target_path, sizeof(target_path));

    if (len == -1) {
        /* Функция завершилась ошибкой. */
        if (errno == EINVAL)
            /* Это не символическая ссылка. */
            fprintf(stderr, "%s is not a symbolic link\n", link_path);
        else
            /* Произошла какая-то другая ошибка. */
            perror("readlink");
        return 1;
    } else {
        /* Завершаем путевое имя нулевым символом. */
        target_path[len] = '\0';
        /* Выводим результат. */
        printf("%s\n", target_path);
        return 0;
    }
}
```

Ниже показано, как создать символическую ссылку и проверить ее с помощью программы

print-symlink:

```
% ln -s /usr/bin/wc my_link
% ./print-symlink my_link
/usr/bin/wc
```

## **8.12. Функция sendfile(): быстрая передача данных**

Функция `sendfile()` это эффективный механизм копирования данных из одного файлового дескриптора в другой. Дескрипторам могут соответствовать дисковые файлы, сокеты или устройства.

Обычно цикл копирования реализуется следующим образом. Программа выделяет буфер фиксированного размера, перемещает в него данные из исходного дескриптора, затем записывает содержимое буфера во второй дескриптор и повторяет описанную процедуру до тех пор, пока не будут скопированы все данные. Такая схема неэффективна как с точки зрения времени, так и с точки зрения затрат памяти, поскольку выделяется дополнительный буфер и над его содержимым выполняются операции копирования.

Функция `sendfile()` устраняет потребность в создании промежуточного буфера. Ей

передаются дескриптор для записи, дескриптор для чтения, указатель на переменную смещения и число копируемых данных. Переменная смещения определяет позицию входного файла, с которой начинается копирование (0 это начало файла). После окончания копирования переменная будет содержать смещение конца блока. Функция `sendfile()` объявлена в файле `<sys/sendfile.h>`.

Программа, показанная в листинге 8.10, представляет собой простую, но очень эффективную реализацию механизма файлового копирования. Она принимает в командной строке два имени файла и копирует содержимое первого файла во второй. Размер исходного файла определяется с помощью функции `fstat()`.

#### ***Листинг 8.10. (copy.c) Копирование файла с помощью функции `sendfile()`***

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

int main(int argc, char* argv[]) {
    int read_fd;
    int write_fd;
    struct stat stat_buf;
    off_t offset = 0;

    /* Открытие входного файла. */
    read_fd = open(argv[1], O_RDONLY);
    /* Определение размера входного файла. */
    fstat(read_fd, &stat_buf);
    /* Открытие выходного файла для записи. */
    write_fd =
        open(argv[2], O_WRONLY | O_CREAT, stat_buf.st_mode);
    /* Передача данных из одного файла в другой. */
    sendfile(write_fd, read_fd, &offset, stat_buf.st_size);
    /* Закрытие файлов. */
    close(read_fd);
    close(write_fd);

    return 0;
}
```

Функция `sendfile()` часто используется для повышения эффективности копирования. Она широко применяется Web-серверами и сетевыми демонами, предоставляющими файлы по сети клиентским программам. Запрос обычно поступает через сокет. Серверная программа открывает локальный дисковый файл, извлекает из него данные и записывает их в сокет. Благодаря функции `sendfile()` эта операция существенно ускоряется.

### **8.13. Функция `setitimer()`: задание интервальных таймеров**

Функция `setitimer()` является обобщением системного вызова `alarm()`. Она планирует доставку сигнала по истечении заданного промежутка времени.

С помощью функции `setitimer()` можно создавать таймеры трех типов.

■ `ITIMER_REAL`. По истечении указанного времени процессу посылается сигнал `SIGALRM`.

■ `ITIMER_VIRTUAL`. После того как процесс отработал требуемое время, ему посылается сигнал `SIGVTALRM`. Время, когда процесс не выполнялся (работало ядро или другой процесс), не учитывается.

■ `ITIMER_PROF`. По истечении указанного времени процессу посылается сигнал `SIGPROF`. Учитывается время выполнения самого процесса, а также запускаемых в нем системных вызовов.

Код таймера задается в первом аргументе функции `setitimer()`. Второй аргумент это указатель на структуру типа `itimerval`, содержащую параметры таймера. Третий аргумент либо равен `NULL`, либо является указателем на другую структуру `itimerval`, куда будут записаны прежние параметры таймера.

В структуре `itimerval` два поля.

■ `it_value`. Здесь находится структура типа `timeval`, где записано время отправки сигнала. Если это поле равно нулю, таймер отменяется.

■ `it_interval`. Это еще одна структура `timeval`, определяющая, что произойдет после отправки первого сигнала. Если она равна нулю, таймер будет отменен. В противном случае здесь записан интервал генерирования сигналов.

Структура `timeval` была описана в разделе 8.7. "Функция `gettimeofday()`: системные часы"

В листинге 8.11 показано, как с помощью функции `setitimer()` отслеживать выполнение программы. Таймер настроен на интервал 250 мс, по истечении которого генерируется сигнал `SIGVTALRM`.

### *Листинг 8.11. (itimer.c) Пример создания таймера*

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/time.h>

void timer_handler(int signum) {
    static int count = 0;
    printf("timer expired %d times\n", ++count);
}

int main() {
    struct sigaction sa;
    struct itimerval timer;

    /* Назначение функции timer_handler обработчиком сигнала
    SIGVTALRM. */
    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = &timer_handler;
    sigaction(SIGVTALRM, &sa, NULL);

    /* Таймер сработает через 250 миллисекунд... */
    timer.it_value.tv_sec = 0;
    timer.it_value.tv_usec = 250000;
    /* ... и будет продолжать активизироваться каждые 250
```

```

миллисекунд. */
timer.it_interval.tv_sec = 0;
timer.it_interval.tv_usec = 250000;
/* Запуск виртуального таймера. Он подсчитывает фактическое
время работы процесса. */
setitimer(ITIMER_VIRTUAL, &timer, NULL);

/* Переход в бесконечный цикл. */
while (1);
}

```

## 8.14. Функция sysinfo(): получение системной статистики

Функция `sysinfo()` возвращает системную статистику. Ее единственным аргументом является указатель на структуру типа `sysinfo`. Перечислим наиболее интересные поля этой структуры.

- `uptime` время в секундах, прошедшее с момента загрузки системы;
- `totalram` общий объем оперативной памяти;
- `freeram` свободный объем ОЗУ;
- `procs` число процессов, работающих в системе.

Для использования функции `sysinfo()` требуется включить в программу файлы `<linux/kernel.h>`, `<linux/sys.h>` и `<sys/sysinfo.h>`.

Программа, приведенная в листинге 8.12, отображает статистическую информацию о текущем состоянии системы.

### *Листинг 8.12. (sysinfo.c) Вывод системной статистики*

```

#include <linux/kernel.h>
#include <linux/sys.h>
#include <stdio.h>
#include <sys/sysinfo.h>

int main() {
/* Константы преобразования. */
const long minute = 60;
const long hour = minute * 60;
const long day = hour * 24;
const double megabyte = 1024 * 1024;
/* Получение системной статистики. */
struct sysinfo si;
sysinfo(&si);
/* Представление информации в понятном виде. */
printf("system uptime : %ld days, %ld:%02ld:%02ld\n",
si.uptime / day, (si.uptime % day) / hour,
(si.uptime % hour) / minute, si.uptime % minute);
printf("total RAM : %5.1f MB\n", si.totalram / megabyte);
printf("free RAM : %5.1f MB\n",
si.freeram / megabyte);
printf("process count : %d\n", si.procs);
return 0;
}

```

## 8.15. Функция uname()

Функция `uname()` возвращает информацию о системе, в частности сетевое и доменное имена компьютера, а также версию операционной системы. Единственным аргументом функции является указатель на структуру типа `utsname`. Функция заполняет следующие поля этой структуры (все эти поля содержат текстовые строки).

- `sysname`. Здесь содержится имя операционной системы (например, `Linux`).

- `release, version`. В этих полях указываются номера версии и модификации ядра.

- `machine`. Здесь приводится информация о платформе, на которой работает система. В случае Intel-совместимых компьютеров это будет либо `i386`, либо `i686`, в зависимости от процессора.

- `node`. Это имя компьютера.

- `__domain`. Это имя домена.

Функция `uname()` объявлена в файле `<sys/utsname.h>`.

В листинге 8.13 показана небольшая программа, которая отображает номера версии и модификации ядра `Linux`, а также сообщает тип платформы.

#### *Листинг 8.15. (`print-uname.c`) Вывод информации о ядре и платформе*

```
#include <stdio.h>
#include <sys/utsname.h>

int main() {
    struct utsname u;
    uname(&u);
    printf("%s release %s (version %s) on %s\n", u.sysname,
        u.release, u.version, u.machine);
    return 0;
}
```

# Глава 9

## Встроенный ассемблерный код

Сегодня лишь немногие программисты используют в своей практике язык ассемблера. Языки высокого уровня, такие как C и C++, поддерживаются практически на всех архитектурах и обеспечивают достаточно высокую производительность программ. Для тех редких случаев, когда требуется встроить в программу ассемблерные инструкции, в коллекции GNU-компиляторов (GCC) предусмотрены специальные средства, учитывающие особенности конкретной архитектуры.

Встроенными ассемблерными инструкциями следует пользоваться осторожно, так как они являются системно-зависимыми. Например, программу с инструкциями архитектуры x86 не удастся скомпилировать на компьютерах PowerPC. В то же время такие инструкции позволяют напрямую обращаться к аппаратным устройствам, вследствие чего программный код выполняется чуть быстрее.

В программы, написанные на языках C и C++, ассемблерные инструкции встраиваются с помощью функции `asm()`. Например, на платформе x86 команда

```
asm("fsin" : "=t" (answer) : "0" (angle));
```

является эквивалентом следующей инструкции языка C:<sup>[30]</sup>

```
answer = sin(angle);
```

Обратите внимание на то, что, в отличие от обычных ассемблерных инструкций, функция `asm()` позволяет указывать входные и выходные операнды, используя синтаксис языка C.

Подробнее узнать об инструкциях архитектуры x86, используемых в настоящей главе, можно по следующим адресам: <http://developer.intel.com/design/pentiumii/manuals> и <http://www.x86-64.org/documentation>.

### 9.1. Когда необходим ассемблерный код

Инструкции, указываемые в функции `asm()`, позволяют программам напрямую обращаться к аппаратным устройствам, поэтому полученные программы выполняются быстрее. Ассемблерные инструкции используются при написании кода операционных систем. Например, файл `/usr/include/asm/io.h` содержит объявления команд, осуществляющих прямой доступ к портам ввода-вывода. Можно также назвать один из исходных файлов ОС Linux `/usr/src/linux/arch/i386/kernel/process.s`; в нем с помощью инструкции `hlt` реализуется пустой цикл ожидания.

Прибегать к ассемблерным инструкциям как к средству ускорения работы программы следует лишь в крайнем случае. Современные компиляторы достаточно сложны и прекрасно осведомлены об особенностях работы процессоров, для которых они генерируют код. Часто они создают цепочки кодов, которые кажутся неэффективными или неоптимальными, но на самом деле такие последовательности инструкций выполняются быстрее других. В подавляющем большинстве случаев можно положиться на оптимизирующие способности компиляторов.

Иногда одна или две ассемблерные команды способны заменить целую группу высокоуровневых инструкций. Например, чтобы определить позицию самого старшего значащего бита целого числа в языке C, требуется написать цикл, тогда как во многих ассемблерных языках для этой цели существует операция `bsrl`. Ее использование будет продемонстрировано в разделе 9.4, "Пример".



## 9.2. Простая ассемблерная вставка

Вот как с помощью функции `asm()` осуществляется сдвиг числа на 8 битов вправо:

```
asm("shrl $8, %0" : "=r" (answer) : "r" (operand) : "cc");
```

Выражение в скобках состоит из секций, разделенных двоеточиями. В первой секции указана ассемблерная инструкция и ее операнды. Команда `shrl` осуществляет сдвиг первого операнда на указанное число битов вправо. Первый операнд представлен выражением `%0`. Второй операнд это константа `$8`.

Во второй секции задаются выходные операнды. Единственный такой операнд будет помещен в `C-переменную` `answer`, которая должна быть адресуемым (левосторонним) значением. В выражении `"=r"` знак равенства обозначает выходной операнд, а буква `r` указывает на то, что значение переменной `answer` заносится в регистр.

В третьей секции перечислены входные операнды. Переменная `operand` содержит значение, подвергаемое битовому сдвигу. Выражение `"r"` означает, что значение переменной записывается в регистр.

Выражение `"cc"` в четвертой секции говорит о том, что инструкция меняет значение регистра `cc` (содержит код завершения).

### 9.2.1. Преобразование функции `asm()` в ассемблерные инструкции

Компилятор `gcc` интерпретирует функцию `asm()` очень просто: он генерирует ассемблерные инструкции, обрабатывающие указанные входные и выходные операнды, после чего заменяет вызов функции заданной инструкцией. Никакой дополнительный анализ не выполняется.

Например, следующий фрагмент программы:

```
double foo, bar;
asm("mycool_asm %1, %0" : "=r" (bar) : "r" (foo));
будет преобразован в такую последовательность команд x86:
movl -8(%ebp),%edx
movl -4(%ebp),%ecx
#APP
mycool_asm %edx, %edx
#NO_APP
movl %edx, -16(%ebp)
movl %ecx, -12(%ebp)
```

Переменные `foo` и `bar` занимают по два слова в стеке в 32-разрядной архитектуре `x86`. Регистр `ebp` ссылается на данные, находящиеся в стеке.

Первые две команды копируют переменную `foo` в регистры `edx` и `ecx`, с которыми работает инструкция `mycool_asm`. Компилятор решил поместить результат в те же самые регистры. Последние две команды копируют результат в переменную `bar`. Выбор нужных регистров и копирование операндов осуществляются автоматически.

## 9.3. Расширенный синтаксис ассемблерных вставок

В следующих подразделах будет описан синтаксис правил, по которым строятся выражения в функции `asm()`. Секции выражения отделяются друг от друга двоеточиями. Мы будем ссылаться на следующую инструкцию, которая вычисляет результат булевого выражения `x > y`:

```
asm("fucomip %%st(1), %%st; seta %%al" :
    "=a" (result) : "u" (y), "t" (x) : "cc", "st");
```

Сначала инструкция `fcomip` сравнивает два операнда, `x` и `y`, и помещает значение, обозначающее результат, в регистр `ss`, после чего инструкция `seta` преобразует это значение в 0 или 1.

### 9.3.1. Ассемблерные инструкции

Первая секция содержит ассемблерные инструкции, заключенные в кавычки. В рассматриваемом примере таких инструкций две: `fcomip` и `seta`. Они разделены точкой с запятой. Если текущий вариант языка ассемблера не допускает такого способа разделения инструкций, воспользуйтесь символом новой строки (`\n`).

Компилятор игнорирует содержимое первого раздела, разве что один уровень символов процента удаляется, т.е. вместо `%%` будет `%`. Смысл выражения `%%st(1)` и ему подобных зависит от архитектуры компьютера.

Если при компиляции программы, содержащей функцию `asm()`, указать опцию `-traditional` или `-ansi`, компилятор `gcc` выдаст предупреждение. Чтобы этого избежать, используйте альтернативное имя `__asm__`.

### 9.3.2. Выходные операнды

Во второй секции указаны выходные операнды инструкции. Каждому операнду соответствует строка адресации и выражение языка C, записанное в скобках. В случае выходных операндов (все они должны быть левосторонними значениями) строка адресации должна начинаться со знака равенства. Компилятор проверяет, действительно ли каждый выходной операнд является левосторонним значением (т.е. может стоять в левой части оператора присваивания).

Список обозначений регистров для конкретной архитектуры можно найти в исходных текстах компилятора `gcc` (конкретнее в определении макроса `REG_CLASS_FROM_LETTER`). Например, в файле `gcc/config/i386/i386.h` содержатся обозначения, соответствующие архитектуре `x86` (табл. 9.1).

Таблица 9.1. Обозначения регистров в архитектуре Intel x86	
Символ регистра	Регистры, которые могут использоваться компилятором gcc
R	Регистры общего назначения (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP)
q	Общие регистры хранения данных (EAX, EBX, ECX, EDX)
f	Регистр для чисел с плавающей запятой
t	Верхний стековый регистр для чисел с плавающей запятой
u	Второй после верхнего стековый регистр для чисел с плавающей запятой
a	Регистр EAX
b	Регистр EBX
c	Регистр ECX
d	Регистр EDX
x	Регистр SSE (регистр потокового расширения SIMD)
y	Мультимедийные регистры MMX
A	Восьмибайтовое значение, формируемое из регистров EAX и EDX

D	Указатель приемной строки в строковых операциях (EDI)
S	Указатель исходной строки в строковых операциях (ESI)

Если есть несколько однотипных операндов, то они разделяются запятыми, как показано в секции входных операндов. Всего можно задавать до десяти операндов, адресуемых как %0, %1, %9. Если выходные операнды отсутствуют, но есть входные операнды или модифицируемые регистры, то вторую секцию следует оставить пустой или пометить ее комментарием наподобие `/* нет выходных данных */`.

### 9.3.3. Входные операнды

В третьей секции задаются входные операнды. Строка адресации такого операнда не должна содержать знака равенства, в остальном синтаксис совпадает с синтаксисом выходных операндов.

Если требуется указать, что в одной инструкции осуществляется как чтение регистра, так и запись в него, необходимо в строке адресации входного операнда поставить номер выходного операнда. Например, если входной регистр должен быть тем же, что и регистр первого выходного операнда, назначьте ему номер 0. Выходные операнды нумеруются слева направо, начиная с нуля. Если просто указать одинаковое C-выражение для входного и выходного операндов, то это еще не означает, что оба значения будут помещены в один и тот же регистр.

Данную секцию можно пропустить, если входные операнды отсутствуют и следующая секция модифицируемых регистров пуста.

### 9.3.4. Модифицируемые регистры

Если в качестве побочного эффекта инструкция модифицирует значение одного или нескольких регистров, в функции `asm()` должна присутствовать четвертая секция. Например, инструкция `fusomip` меняет регистр кода завершения, обозначаемый как `ss`. Строки, представляющие затираемые регистры, разделяются запятыми. Если инструкция способна изменить произвольную ячейку памяти, в этой секции должно стоять ключевое слово `memory`. На основании этой информации компилятор определяет, какие значения должны быть загружены повторно после завершения функции `asm()`. При отсутствии данной секции компилятор может сделать неверное предположение о том, что регистры содержат прежние значения, и это скажется на работе программы.

## 9.4. Пример

В архитектуре x86 есть инструкции, определяющие позицию старшего и младшего значащих битов в слове. Процессор выполняет эти инструкции очень быстро. С другой стороны, чтобы сделать то же самое на языке C, потребуется написать цикл с операциями побитового сдвига.

Инструкция `bsrl` вычисляет местоположение старшего значащего бита в первом операнде и записывает результат (номер позиции начиная с нуля) во второй операнд. Например, следующая команда анализирует переменную `number` и помещает результат в переменную `position`:

```
asm("bsrl %1, %0" : "=r" (position) : "r" (number));
```

Ей соответствует такой фрагмент на языке C:

```
long i;
for (i = (number >> 1), position = 0; i != 0; ++position)
    i >>= 1;
```

Чтобы сравнить скорость выполнения двух фрагментов, мы поместили их в цикл, где перебирается большое количество чисел. В листинге 9.1 приведена реализация на языке C. Программа перебирает значения от единицы до числа, указанного в командной строке. Для каждого значения переменной `number` вычисляется позиция старшего значащего бита. В листинге 9.2 показано, как сделать то же самое с помощью ассемблерной вставки. Обратите внимание на то, что в обоих случаях результат вычислений заносится в переменную `result`, объявленную со спецификатором `volatile`. Это необходимо для подавления оптимизации со стороны компилятора, который удалит весь блок вычислений, если их результаты не используются или не заносятся в память.

***Листинг 9.1. (bit-pos-loop.c) Нахождение позиции старшего значащего бита в цикле***

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

int main(int argc, char* argv[]) {
    long max = atoi(argv[1]);
    long number;
    long i;
    unsigned position;
    volatile unsigned result;

    /* Повторяем вычисления для большого количества чисел. */
    for (number = 1; number <= max; ++number) {
        /* Сдвигаем число вправо, пока результат не станет
        равным нулю.
        Запоминаем количество операций сдвига. */
        for (i = (number >> 1), position = 0; i != 0; ++position)
            i >>= 1;
        /* Позиция старшего значащего бита это общее число
        операций сдвига, кроме первой. */
        result = position;
    }
    return 0;
}
```

***Листинг 9.2. (bit-pos-asm.c) Нахождение позиции старшего значащего бита с помощью инструкции `bsrl`***

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

int main(int argc, char* argv[]) {
    long max = atoi(argv[1]);
    long number;
    unsigned position;
    volatile unsigned result;

    /* Повторяем вычисления для большого количества чисел. */
    for (number = 1; number <= max; ++number) {
        /* Вычисляем позицию старшего значащего бита с помощью
        ассемблерной инструкции bsrl. */
```

```
asm("bsrl %1, %0" : "=r" (position) : "r" (number));
result = position;
}
return 0;
}
```

Скомпилируем обе версии программы в режиме полной оптимизации:

```
% cc -O2 -o bit-pos-loop bit-pos-loop.c
```

```
% cc -O2 -o bit-pos-asm bit-pos-asm.c
```

Теперь запустим их с помощью команды `time`, которая замеряет время выполнения. В командной строке каждой программы указано большое значение, чтобы программа выполнялась хотя бы несколько секунд.

```
% time ./bit-pos-loop 250000000
```

```
19.51user 0.00system 0:20.40elapsed 95%CPU (0avgtext+0avgdata
0maxresident)k0inputs+0outputs (73major+11minor)pagefaults 0swaps
```

```
% time ./bit-pos-asm 250000000
```

```
3.19user 0.00system 0:03.32elapsed 95%CPU (0avgtext+0avgdata
0maxresident)k0inputs+0outputs (73major+11minor)pagefaults 0swaps
```

Приведенные результаты могут немного меняться в зависимости от загруженности системы, но хорошо видно, что ассемблерная версия выполняется гораздо быстрее.

## 9.5. Вопросы оптимизации

Даже при наличии в программе ассемблерных вставок модуль оптимизации компилятора пытается переупорядочить и переписать код программы, чтобы минимизировать время ее выполнения. Когда оптимизатор обнаруживает, что выходные данные функции `asm()` не используются, он удаляет ее, если только ему не встречается ключевое слово `volatile`. Любой вызов функции `asm()` может быть перемещен самым непредсказуемым образом. Единственный способ гарантировать конкретный порядок ассемблерных инструкций включить все нужные инструкции в одну функцию `asm()`.

Применение функции `asm()` ограничивает эффективность оптимизации, поскольку компилятор не понимает семантику используемых в ней ассемблерных выражений. Помните об этом!

## 9.6. Вопросы сопровождения и переносимости

Если вы решили включить в программу архитектурно-зависимые ассемблерные вставки, поместите их в отдельные макросы или функции, что облегчит сопровождение программы. Когда все макросы находятся в одном файле и задокументированы, программу легче будет перенести в другую систему, так как придется переписать один-единственный файл. Например, большинство вызовов `asm()` в исходных текстах Linux сгруппировано в файлах `/usr/src/linux/include/asm` и `/usr/src/linux/include/asm-i386`.

# Глава 10

## Безопасность

Одним из основных достоинств Linux является поддержка одновременной работы нескольких пользователей, в том числе по сети. Но у всякой медали есть обратная сторона. В данном случае это угрозы безопасности, возникающие, когда система подключена к Internet. При благоприятном стечении обстоятельств хакер сможет войти в систему и прочитать, модифицировать или удалить хранящиеся в ней файлы. Кроме того, сами пользователи системы могут получать несанкционированный доступ к чужим файлам.

В ядре Linux есть множество средств, позволяющих предотвратить подобные события. Но защищать следует и обычные приложения. Предположим, к примеру, что разрабатывается бухгалтерская программа. Любой пользователь может зарегистрировать в программе отчет о расходах, но далеко не каждый имеет право утвердить этот отчет. Пользователям разрешается просматривать информацию о своих зарплатах, но, естественно, не о зарплатах своих коллег. Менеджерам может быть разрешено получать данные о зарплатах служащих их отделов, но не служащих других отделов.

Все эти меры предосторожности требуют особого внимания. Очень легко допустить ошибку, из-за которой пользователи смогут делать то, что им не разрешено. Конечно, лучше всего воспользоваться помощью экспертов по системам безопасности. Но базовыми знаниями должен владеть любой разработчик программного обеспечения.

### 10.1. Пользователи и группы

В Linux каждому пользователю назначается уникальный номер, называемый его *идентификатором* (UID, user identifier). При регистрации в системе, естественно, вводится имя пользователя, а не идентификатор. Система преобразовывает введенное имя в соответствующий идентификатор и дальше работает только с ним.

С одним идентификатором может быть связано несколько пользовательских имен. С точки зрения системы это не имеет никакого значения и не представляет для нее проблемы: она учитывает только идентификатор. Пользовательские имена совершенно равноправны, если им соответствует одинаковый идентификатор.

Доступ к файлу или другому ресурсу контролируется путем закрепления за ним конкретного идентификатора пользователя. Только пользователь с этим идентификатором имеет привилегированный доступ к ресурсу. Например, можно создать файл, который будет открыт для чтения лишь его владельцу, либо создать каталог, в котором только владелец сможет создавать новые файлы. Это самые простые способы защиты данных.

Иногда требуется делить ресурсы с несколькими пользователями. К примеру, менеджер может создать файл, предназначенный для чтения другими менеджерами, но не рядовыми служащими. ОС Linux не позволяет закреплять за файлом несколько пользовательских идентификаторов, поэтому нельзя задать список лиц, имеющих доступ к данному конкретному файлу.

Но выход все же есть это создание *группы*. Ей также назначается уникальный номер, называемый *идентификатором группы* (GID, group identifier). В каждую группу входит один или несколько идентификаторов пользователей. Один и тот же пользователь может быть членом множества групп, но членами групп не могут быть другие группы. У групп, как и у пользователей, есть имена, но они не играют практически никакой роли, так как система

работает с идентификаторами групп.

Например, можно создать группу `managers` и включить в нее идентификаторы всех менеджеров компании. Тогда любой файл, принадлежащий этой группе, будет доступен только менеджерам и никому другому. Всякому системному ресурсу соответствует только одна группа.

Команда `id` позволяет узнать идентификатор текущего пользователя и группы, в которые он входит:

```
% id
uid=501(mitchell) gid=501(mitchell) groups=501(mitchell), 503(csl)
```

В первой части выходных данных указано, что идентификатор пользователя равен 501. В скобках приведено соответствующее этому идентификатору имя пользователя. Как следует из результатов работы команды, пользователь `mitchell` входит в две группы: с номером 501 (`mitchell`) и с номером 503 (`csl`). Читатели, возможно, удивлены тем, что группа 501 появляется дважды: в поле `gid` и в поле `groups`. Объяснение этому факту будет дано позже.

### 10.1.1. Суперпользователь

Одна учетная запись имеет для системы особое значение.<sup>[31]</sup> Пользователь, чей идентификатор равен 0, обычно носит имя `root` (его еще иногда называют суперпользователем). Этот пользователь обладает исключительными правами: он может читать и удалять любой файл, добавлять новых пользователей, отключать сетевые интерфейсы и т.п. Множество специальных операций разрешено выполнять лишь процессам, работающим с привилегиями суперпользователя.

К сожалению, этих специальных операций так много, что очень большое число программ должно принадлежать пользователю `root`. Если какая-то из этих программ ведет себя неправильно, система может погрузиться в хаос. Не существует способа воспрепятствовать работе такой программы: она может делать *все что угодно*. Поэтому программы, принадлежащие пользователю `root`, следует писать очень внимательно.

## 10.2. Идентификаторы пользователей и групп, закрепленные за процессами

До сих пор речь шла о командах, выполняемых конкретными пользователями. Это не совсем точно, поскольку компьютер в действительности никогда не знает, кто из пользователей за ним работает. Если пользователь Ева узнает имя и пароль пользователя Элис, она сможет войти в систему под ее именем, и компьютер позволит Еве выполнять те действия, которые разрешены для Элис. Системе известен лишь идентификатор пользователя, а не то, какой именно пользователь вводит команды. Таким образом, ответственность за безопасность системы распределяется между разработчиками приложений, пользователями и системными администраторами.

С каждым процессом связаны идентификаторы пользователя и группы. Когда пользователь вызывает программу, запускается процесс, идентификаторы которого совпадают с идентификаторами этого пользователя. Когда мы говорим, что пользователь выполняет операцию, то на самом деле имеется в виду, что операцию выполняет процесс с идентификатором соответствующего пользователя. Когда процесс делает системный вызов, ядро проверяет идентификаторы процесса и определяет, имеет ли процесс право доступа к запрашиваемым ресурсам.

Теперь становится понятным смысл поля `gid` в выводе команды `id`. В нем показан идентификатор группы текущего процесса. Пользователь 501 может входить в несколько групп,

но текущему процессу соответствует только один идентификатор группы. В рассматривавшемся примере это 501.

В программах значения идентификаторов пользователей и групп имеют типы `uid_t` и `gid_t`. Оба типа определены в файле `<sys/types.h>`. Несмотря на то что эти идентификаторы являются, по сути, всего лишь целыми числами, избегайте делать какие-либо предположения о том, сколько битов они занимают, и выполнять над ними арифметические операции

Узнать идентификаторы пользователя и группы текущего процесса позволяют функции `geteuid()` и `getegid()`, объявленные в файле `<unistd.h>`. Они не принимают никаких аргументов и всегда работают, так что проверять ошибки не обязательно. В листинге 10.1 показана программа, которая частично дублирует работу команды `id`.

### **Листинг 10.1. (*simpleid.c*) Отображение идентификаторов пользователя и группы**

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

int main() {
    uid_t uid = geteuid();
    gid_t gid = getegid();
    printf("uid=%d gid=%d\n", (int) uid, (int)gid);
    return 0;
}
```

Если программу запустит тот же пользователь, который ранее запустил команду `id`, результат будет таким:

```
% ./simpleid
uid=501 gid=501
```

## **10.3. Права доступа к файлам**

Хороший способ разобраться в назначении идентификаторов пользователей и групп изучить права доступа к файловой системе. В частности, нужно узнать, как система устанавливает права доступа к файлам и как ядро определяет, кому разрешено обращаться к запрашиваемым файлам.

У каждого файла есть лишь один *пользователь-владелец* и одна *группа-владелец*. При создании файла за ним закрепляются идентификаторы пользователя и группы того процесса, в котором происходит эта операция.

Основные операции, производимые над файлами в Linux, это *чтение*, *запись* и *выполнение* (создание и удаление файлов считаются операциями над каталогами, где находятся эти файлы). Если файл недоступен для чтения, Linux не позволит узнать его содержимое, а если файл защищен от записи, то нельзя будет модифицировать его содержимое. Программу, у которой отсутствует право выполнения, нельзя будет запустить.

Linux позволяет задавать, какие действия чтение, запись, выполнение разрешено осуществлять над файлом его владельцу, группе и остальным пользователям. Например, можно указать, что владелец имеет все права доступа к файлу, пользователям группы разрешено читать и выполнять файл (но не записывать в него), а остальные пользователи не должны получать к нему доступ.

Совокупность прав доступа к файлу называется *кодом режима*. Он состоит из трех триад



битов, соответствующих владельцу, группе и остальным пользователям. В каждой триаде первый бит означает право чтения, второй право записи, третий право выполнения. Символьное представление этих битов называется *строкой режима*. Просмотреть ее можно с помощью команды `ls -l` или системного вызова `stat()`. Задание прав доступа к файлу осуществляется с помощью команды `chmod` или одноименного системного вызова. Допустим, имеется файл `hello` и требуется узнать права доступа к нему. Вот как это делается:

```
% ls -l hello
-rwxr-x--- 1 samuel cs1 11734 Jan 22 16:29 hello
```

Третье и четвертое поля выводных данных сообщают о том, что файл принадлежит пользователю `samuel` и группе `cs1`. В первом поле отображается строка режима. Начальный дефис указывает на то, что это обычный файл. В случае каталога здесь будет стоять буква `d`. Специальные файлы, например файлы устройств (см. главу 6, "Устройства") или каналы (см. раздел 5.4, "Каналы"), обозначаются другими буквами. Следующие три символа соответствуют правам владельца файла. В данном случае пользователь `samuel` имеет право чтения, записи и выполнения файла. Далее указаны права группы, которой принадлежит файл. Пользователям группы разрешено читать и выполнять файл. Последние три символа в строке режима обозначают права остальных пользователей, которым запрещен доступ к файлу.

Давайте проверим, действительно ли все вышесказанное правда. Для начала попробуем обратиться к файлу от имени пользователя `nobody`, не входящего в группу `cs1`:

```
% id
uid=99(nobody) gid=99(nobody) groups=99(nobody)
% cat hello
cat: hello: Permission denied
% echo hi > hello
sh: ./hello: Permission denied
% ./hello
sh: ./hello: Permission denied
```

Команда `cat` не смогла выполниться, потому что у нас нет права чтения файла. Запись в файл тоже не разрешена, поэтому потерпела неудачу команда `echo`. А поскольку право выполнения также отсутствует, запустить программу `hello` не удалось.

Посмотрим, что будет, если к файлу обратится пользователь `mitchell`, являющийся членом группы `cs1`:

```
% id
uid=501 (mitchell) gid=501 {mitchell} groups=501 (mitchell), 503 (cs1)
% cat hello
#!/bin/bash
echo "Hello, world."
% ./hello
Hello, world.
% echo hi > hello
bash: ./hello: Permission denied
```

В данном случае можно отобразить содержимое файла и запустить его на выполнение (файл является простейшим командным сценарием), но осуществить перезапись файла невозможно. Доступ для записи разрешен только владельцу файла (пользователь `samuel`):

```
% id
uid=502(samuel) gid=502(samuel) groups=502(samuel), 503(cs1)
% echo hi > hello
% cat hello
hi
```

Менять режим доступа к файлу может только его владелец, а также суперпользователь. Если требуется разрешить всем пользователям запускать файл на выполнение, то это делается так:

```
% chmod o+k hello
% ls -l hello
-rwxr-x--x 1 samuel csl 3 Jan 22 16:38 hello
```

Обратите внимание на появление буквы x в конце строки режима. Флаг o+x команды chmod означает добавление (+) права выполнения (x) для остальных пользователей (o). Если требуется, к примеру, отнять право записи у группы, следует задать такой флаг: g-w.

Функция stat() позволяет определить режим доступа к файлу программным путем. Она принимает два аргумента: имя файла и адрес структуры, заполняемой информацией о файле. Подробнее функция stat() описана в приложении Б, "Низкоуровневый ввод-вывод". Пример ее использования показан в листинге 10.2.

### ***Листинг 10.2. (stat-perm.c) Проверка того, имеет ли владелец право записи в файл***

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

int main(int argc, char* argv[]) {
    const char* const filename = argv[1];
    struct stat buf;
    /* Получение информации о файле. */
    stat(filename, &buf);
    /* Если владельцу разрешена запись в файл,
    отображаем сообщение. */
    if (buf.st_mode & S_IWUSR)
        printf("Owning user can write '%s'.\n", filename);
    return 0;
}
```

Если запустить программу с файлом hello, будет выдано следующее:

```
% ./stat-perm hello
Owning user can write 'hello'.
```

Константа S\_IWUSR соответствует праву записи для владельца. Для каждого бита в строке режима существует своя константа. Например, константа S\_IRGRP обозначает право чтения для группы, а константа S\_IXOTH право выполнения для остальных пользователей. Если невозможно получить информацию о файле, функция stat() возвращает -1 и помещает код ошибки в переменную errno.

С помощью функции chmod() можно менять режим доступа к существующему файлу. Функции передаётся имя файла и набор флагов, соответствующих устанавливаемым битам доступа. Например, в следующей строке файл hello делается доступным для чтения и выполнения владельцу, а права группы и остальных пользователей отменяются:

```
chmod("hello", S_IRUSR | S_IXUSR);
```

Те же самые права доступа действуют и в отношении каталогов, но имеют несколько иной смысл. Если у пользователя есть право чтения каталога, то это означает разрешение на получение списка содержимого каталога. Право записи означает возможность добавлять и удалять файлы в каталоге. Пользователь, которому разрешена запись в каталог, может удалять из него файлы даже в том случае, когда у него нет права доступа к этим файлам. Право выполнения применительно к каталогам называется правом поиска. Пользователю, имеющему это право, разрешается входить в каталог и обращаться к его файлам. Если пользователь не может перейти в каталог, то ему не удастся получить доступ к находящимся в нем файлам независимо от их собственных прав доступа.

Подводя итог, рассмотрим, как ядро определяет, имеет ли процесс право обратиться к

заданному файлу. Сначала выясняется, кем является пользователь, запустивший процесс: владельцем файла, членом его группы или кем-то другим. В зависимости от категории пользователя проверяется соответствующий набор битов чтения/записи/выполнения и на его основании принимается окончательное решение.<sup>[32]</sup>

Есть, правда, одно важное исключение: процессы, запускаемые пользователем root (его идентификатор равен нулю), всегда получают доступ к требуемым файлам независимо от их атрибутов.

### 10.3.1. Проблема безопасности: программы без права выполнения

Есть один хороший пример того, как обмануть неопытного пользователя, пытающегося защитить свои программы от несанкционированного запуска. Сброс бита выполнения файла еще не означает, что файл нельзя будет запустить. Дело в том, что при копировании файла копия переходит в распоряжение нового владельца. Как вы понимаете, ему не составляет никакого труда изменить права доступа к скопированному файлу и снова сделать его исполняемым. Вывод: защищайте программы не от несанкционированного запуска, а от несанкционированного копирования!

### 10.3.2. Sticky-бит

Помимо обычных битов режима есть один особый бит, называемый sticky-битом ("липучкой").<sup>[33]</sup> Он применим только в отношении каталогов.

Обычно удалять файлы могут пользователи, имеющие право записи в каталог. Каталог, для которого установлен sticky-бит, допускает удаление файла только в том случае, когда пользователь является владельцем этого файла или самого каталога и имеет право записи в каталог.

В типичной Linux-системе есть несколько таких каталогов. Один из них каталог /tmp, в котором любой пользователь может размещать временные файлы. Этот каталог специально сделан доступным для всех пользователей, поэтому он полностью открыт для записи. Однако нельзя допустить, чтобы пользователи удаляли чужие файлы, поэтому для каталога /tmp установлен sticky-бит.

О наличии sticky-бита говорит буква t в конце строки режима:

```
% ls -ld /tmp
```

```
drwxrwxrwt 12 root root 2048 Jan 24 17:51 /tmp
```

Соответствующий флаг функций stat() и chmod() называется S\_ISVTX.

Если требуется установить для каталога sticky-бит, следует воспользоваться такой командой:

```
% chmod o+t каталог
```

А вот как можно назначить каталогу те же права доступа, что и к каталогу /tmp:

```
chmod(dir_path, S_IRWXU | S_IRWXG | S_IRWXO | S_ISVTX);
```

## 10.4. Реальные и эффективные идентификаторы

До сих пор подразумевалось, что у процесса один идентификатор пользователя и один идентификатор группы. На самом деле не все так просто. У каждого процесса есть два пользовательских идентификатора: *реальный* и *эффективный*. То же самое справедливо и в отношении идентификаторов групп. В большинстве случаев ядро работает с эффективным

идентификатором. Например, если процесс пытается открыть файл, ядро проверяет допустимость этой операции именно на основании эффективного идентификатора.

Упомянутые выше функции `geteuid()` и `getegid()` возвращают эффективные идентификаторы пользователя и группы. Для определения реальных идентификаторов предназначены функции `getuid()` и `getgid()`.

Раз ядро работает только с эффективным идентификатором, какой смысл в существовании еще и реального идентификатора? Есть один специальный случай, когда он необходим: ядро проверяет его при попытке изменения эффективного идентификатора выполняющегося процесса.

Прежде чем выяснять, как менять эффективный идентификатор процесса, рассмотрим, зачем это необходимо. Предположим, имеется серверный процесс, который может просматривать любой файл независимо от того, кто является его владельцем. Такой процесс должен быть запущен пользователем `root`, так как только у него есть подобные привилегии. А теперь допустим, что запрос к файлу поступает от конкретного пользователя (`mitchell`, к примеру). Серверный процесс может проверить, есть ли у данного пользователя соответствующие разрешения, но это означает дублирование того кода, который уже реализован в ядре.

Гораздо лучший подход временно поменять эффективный идентификатор пользователя `root` на `mitchell` и попытаться выполнить требуемую операцию. Если пользователь `mitchell` не имеет нужных прав доступа, ядро само даст об этом знать. После завершения (или отмены) операции серверный процесс восстановит первоначальный эффективный идентификатор.

Механизм временной замены идентификаторов используется программами, которые выполняют аутентификацию пользователей, пытающихся зарегистрироваться в системе. Такие программы работают с правами пользователя `root`. Когда пользователь вводит свое имя и пароль, программа аутентификации сверяет их с записями в системной базе паролей. Если проверка прошла успешно, программа меняет свои эффективные и реальные идентификаторы на пользовательские, после чего выполняет функцию `exec()`, которая запускает интерпретатор команд. В результате пользователь оказывается в среде интерпретатора, идентификаторы которого соответствуют пользовательским.

Функция, меняющая пользовательский идентификатор процесса, называется `setreuid()` (имеется, конечно же, и функция `setregid()`). Она принимает два аргумента: устанавливаемый реальный идентификатор и требуемый эффективный идентификатор. Вот как, к примеру, можно поменять эффективный и реальный идентификаторы:

```
setreuid(geteuid(), getuid());
```

Естественно, ядро не позволит первому попавшемуся процессу изменить свои идентификаторы. Если бы это было возможно, любой пользователь легко мог бы получить доступ к чужим ресурсам, сменив эффективный идентификатор одного из своих процессов. Поэтому ядро делает исключение лишь для процессов, чей эффективный идентификатор пользователя равен нулю (опять-таки, обратите внимание на то, какой властью обладают процессы суперпользователя!) Всем остальным процессам разрешается следующее:

- заменять эффективный идентификатор реальным;
- заменять реальный идентификатор эффективным;
- переставлять местами значения реального и эффективного идентификаторов.

Первый вариант будет использован серверным процессом, когда он закончит работать от имени пользователя `mitchell` и захочет снова "стать" пользователем `root`. Второй вариант используется программой аутентификации после того, как она сделала эффективный идентификатор равным пользовательскому. Изменение реального идентификатора необходимо

для того, чтобы пользователь не смог стать обратно пользователем root. Последний, третий, вариант в современных программах не встречается.

В качестве любого из аргументов функции `setreuid()` можно указать значение `-1`. Это означает, что соответствующий идентификатор нужно оставить без изменений. Есть также вспомогательная функция `seteuid()`, которая меняет эффективный идентификатор, но не трогает реальный. Например, следующие две строки эквивалентны:

```
seteuid(id);
setreuid(-1, id);
```

#### 10.4.1. Программы с установленным битом SUID

Выше было показано, как процесс пользователя `root` может временно принять на себя права другого пользователя или отказаться от специальных привилегий, изменив свои реальный и эффективный идентификаторы. Но вот загадка: может ли непривилегированный пользователь стать суперпользователем? Это кажется невозможным, но следующий пример свидетельствует об обратном:

```
% whoami
mitchell
% su
Password: ...
% whoami
root
```

Команда `whoami` аналогична команде `id`, но отображает только эффективный идентификатор пользователя. Команда `su` позволяет вызвавшему ее пользователю стать суперпользователем, если введен правильный пароль.

Как же работает команда `su`? Ведь мы знаем, что интерпретатор команд был запущен с реальным и эффективным идентификаторами, равными `mitchell`. Функция `setreuid()` не позволит ему поменять ни один из них.

Дело в том, что у программы `su` установлен бит *смены идентификатора пользователя* (SUID, set user identifier). Это значит, что при запуске ее эффективным идентификатором станет идентификатор владельца (реальный идентификатор останется тем же, что у пользователя, запустившего программу). Для установки бита SUID предназначены команда `chmod +s` и флаг `S_SUID` функции `chmod()`.<sup>[34]</sup>

В качестве примера рассмотрим программу, показанную в листинге 10.3.

#### Листинг 10.3. (`setuid-test.c`) Проверка идентификаторов

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

int main() {
    printf("uid=%d euid=%d\n", (int)getuid(), (int)geteuid());
    return 0;
}
```

Теперь предположим, что у программы установлен бит SUID и она принадлежит пользователю `root`. В этом случае вывод команды `ls` будет примерно таким:

```
-rwsrws--x 1 root root 11931 Jan 24 18:25 setuid-test
```

Буквы `s` в строке режима означают, что этот файл не только является исполняемым, но для него установлены также биты SUID и SGID. Результат работы программы будет таким:

```
% whoami
mitchell
% ./setuid-test
uid=501 euid=0
```

Обратите внимание на то, что эффективный идентификатор стал равным нулю. Устанавливать биты SUID и SGID позволяют команда `chmod u+s` и `chmod g+s` соответственно. Приведем пример:

```
% ls -l program
-rwxr-xr-x 1 samuel cs1 0 Jan 30 23:38 program
% chmod g+s program
% ls -l program
-rwxr-sr-x 1 samuel cs1 0 Jan 30 23:38 program
% chmod u+s program
% ls -l program
-rwsr-sr-x 1 samuel cs1 0 Jan 30 23:38 program
```

Аналогичным целям служат флаги `S_ISUID` и `S_ISGID` функции `chmod()`.

Именно так работает команда `su`. Ее эффективный идентификатор пользователя равен нулю. Если введенный пароль совпадает с паролем пользователя `root`, команда меняет свой реальный идентификатор на `root`, после чего запускает новый интерпретатор команд. В противном случае ничего не происходит.

Рассмотрим атрибуты программы `su`:

```
% ls -l /bin/su
-rwsr-xr-x 1 root root 14188 Mar 7 2000 /bin/su
```

Как видите, она принадлежит пользователю `root` и для нее установлен бит SUID. Обратите внимание на то, что команда `su` не меняет идентификатор интерпретатора команд, в котором она была вызвана, а запускает новый интерпретатор с измененным идентификатором. Первоначальный интерпретатор будет заблокирован до тех пор, пока пользователь не введет `exit`.

## 10.5. Аутентификация пользователей

Программы, у которых установлен бит SUID, не должны запускаться кем попало. Например, программа `su`, прежде чем менять идентификатор пользователя, заставляет его ввести пароль. Это называется *аутентификацией* программа проверяет, получил ли пользователь права суперпользователя.

Администраторам высоконадежных систем недостаточно, чтобы пользователи просто вводили пароли. У пользователей есть вредная привычка записывать свои пароли на бумажке, приклеенной к монитору, или выбирать пароли, в которых закодирован день рождения, имя любимой собаки, жены и т.п. Все это облегчает задачу злоумышленникам, пытающимся незаконно проникнуть в систему.

Во многих организациях требуется использовать "одноразовые" пароли, генерируемые специальными электронными карточками, которые пользователи хранят при себе. Одни и тот же пароль не может встретиться дважды, а прежде чем получить пароль, требуется ввести личный код. Следовательно, для взлома системы хакеру требуется раздобыть электронную карточку и узнать соответствующий личный код. В сверхсекретных учреждениях используются устройства сканирования сетчатки глаза или другие биометрические приборы.

При написании аутентификационной программы важно позволить системному администратору использовать тот механизм аутентификации, который он считает приемлемым. В Linux этой цели служат *подключаемые модули аутентификации* (PAM, pluggable authentication modules). Рассмотрим простейшее приложение (листинг 10.4).

```
#include <security/pam_appl.h>
#include <security/pam_misc.h>
#include <stdio.h>

int main() {
    pam_handle_t* pamh;
    struct pam_conv pamc;

    /* Указание диалоговой функции. */
    pamc.conv = &misc_conv;
    pamc.eppdata_ptr = NULL;
    /* Начало сеанса аутентификации. */
    pam_start("su", getenv("USER"), &pamc, &pamh);
    /* Аутентификация пользователя. */
    if (pam_authenticate(pamh, 0) != PAM_SUCCESS)
        fprintf(stderr, "Authentication failed!\n");
    else
        fprintf(stderr, "Authentication OK.\n");
    /* Конец сеанса. */
    pam_end(pamh, 0);
    return 0;
}
```

Чтобы скомпилировать эту программу, необходимо подключить к ней две библиотеки:

libpam и libpam\_misc:

```
% gcc -o para pam.c -lpam -lpam_misc
```

Сначала программа создает *объект диалога*, который используется библиотекой PAM, когда ей требуется запросить у пользователя данные. Функция `misc_conv()`, адрес которой записывается в объект, это стандартная диалоговая функция, осуществляющая терминальный ввод-вывод. Можно написать собственную функцию, отображающую всплывающее окно, использующую голосовой ввод-вывод или реализующую другие способы общения с пользователем.

Затем вызывается функция `pam_start()`, которая инициализирует библиотеку PAM. Первый аргумент функции это имя сервиса. Оно должно уникальным образом идентифицировать приложение. Программа не будет работать, пока системный администратор не настроит систему на использование указанного сервиса. В данном случае задействуется сервис `su`, при котором программа аутентифицирует пользователей так же, как это делает команда `su`. В реальных программах так поступать не следует. Выберите реальное имя сервиса и создайте сценарий инсталляции, который позволит системному администратору правильно настраивать механизм аутентификации.

Второй аргумент функции это имя пользователя, которого требуется аутентифицировать. В данном примере имя пользователя берется из переменной среды `USER` (обычно это имя соответствует эффективному идентификатору текущего процесса, но так бывает не всегда). В большинстве реальных программ в данном месте выдается запрос на ввод имени. Третьим аргументом является ссылка на объект диалога. В четвертом аргументе указывается переменная, в которую функция `pam_start()` запишет дескриптор сеанса. Этот дескриптор должен передаваться всем последующим функциям библиотеки PAM.

Далее в программе вызывается функция `pam_authenticate()`. Во втором ее аргументе указываются различные флаги. Значение 0 означает стандартные установки. Возвращаемое значение функции говорит о том, как прошла аутентификация. В конце программы вызывается

функция `ram_end()`, которая удаляет выделенные ранее структуры данных.

Предположим, что пользователь должен ввести пароль "password". Если это будет сделано, получим следующее:

```
% ./pam
Password: password
```

Authentication OK.

Будучи запущенной в терминальном окне, программа не покажет введенный пароль, чтобы кто-нибудь посторонний не смог его подглядеть.

Вот что произойдет, если в систему попытается вломиться хакер:

```
% ./pam
Password: badguess
```

Authentication failed!

Полное описание работы модулей аутентификации приведено в каталоге `/usr/doc/pam`.

## 10.6. Дополнительные проблемы безопасности

В этой главе мы рассматриваем лишь несколько наиболее общих проблем, связанных с безопасностью. Но существует великое множество других "дыр", и далеко не все из них еще раскрыты. Поэтому в ответственных случаях без помощи экспертов не обойтись.

### 10.6.1. Переполнение буфера

Почти все основные Internet-демоны, включая демоны таких программ, как `sendmail`, `finger`, `talk` и др., подвержены атакам типа переполнение буфера. О них следует обязательно помнить при написании программ, которые должны выполняться с правами пользователя `root`, а также программ, осуществляющих межзадачное взаимодействие или читающих файлы, которые не принадлежат пользователю, запустившему программу.

Суть атаки заключается в том, чтобы заставить программу выполнить код, который она не собиралась выполнять. Типичный механизм достижения этой цели перезапись части стека программы. В стеке, помимо всего прочего, сохраняется адрес памяти, по которому программа передает управление после завершения текущей функции. Следовательно, если поместить код взлома в памяти, а затем изменить адрес возврата так, чтобы он указывал на этот код, то по завершении текущей функции программа начнет выполнять код хакера с правами текущего процесса. Если процесс принадлежит пользователю `root`, последствия будут катастрофическими. Если атаке подвергся процесс другого пользователя, катастрофа наступит "только" для него (а также для любого пользователя, который работает с файлами пострадавшего).

Хуже всего обстоит дело с программами, которые работают в режиме демона и ожидают поступление запросов на подключение. Демоны обычно принадлежат пользователю `root`. Если в программе есть описываемая "дыра", любой, кто сможет подключиться к этой программе, способен захватить контроль над компьютером, послав по сети "смертельную" последовательность данных. Программы, не работающие с сетью, гораздо безопаснее, так как их могут атаковать только пользователи, уже зарегистрировавшиеся в системе.

Старым версиям программ `finger`, `talk` и `sendmail` присущ один общий недостаток: все они работают со строковым буфером фиксированной длины. Предельный размер строки предполагается по умолчанию, но ничто не мешает сетевым клиентам вводить строки,



вызывающие переполнение буфера. В программах содержится примерно такой код, как показан ниже.

```
#include <stdio.h>

int main() {
/* Никто, будучи в здравом уме, не выбирает имя пользователя
длиной более 32 символов. Кроме того, я думаю, что в UNIX
допускаются только 8-символьные имена. Поэтому выделенного
буфера должно быть достаточно. */
char username[32];
/* Предлагаем пользователю ввести свое имя. */
printf("Enter your username: ");
/* Читаем введенную строку. */
gets(username);
/* Выполняем другие действия... */

return 0;
}
```

Комбинация 32-символьного буфера и функции `gets()` делает возможным переполнение буфера. Функция `gets()` читает вводимые данные до тех пор, пока не встретится символ новой строки, после чего помещает весь результат в массив `username`. В комментариях к программе предполагается, что пользователи выбирают себе короткие имена, не превышающие в длину 32 символа. Но при написании защищенных программ необходимо помнить о существовании хакеров. В данном случае хакер может выбрать сколь угодно длинное имя. Локальные переменные, в частности `username`, сохраняются в стеке, поэтому выход за пределы массива оборачивается тем, что в стек помещаются произвольные данные.

К счастью, предотвратить переполнение буфера относительно несложно. При чтении строк следует всегда пользоваться функцией наподобие `getline()`, которая либо динамически выделяет буфер достаточной длины, либо прекращает принимать входные данные, когда буфер оказывается заполнен. Вот вариант выхода из положения:

```
char* username = getline(NULL, 0, stdin);
```

Функция `getline()` автоматически вызывает функцию `malloc()`, которая выделяет буфер для введенной строки и возвращает указатель на него. Естественно, следует не забыть вызвать функцию `free()`, чтобы по окончании работы с буфером вернуть память системе.

Ситуация еще проще, если используется язык C++, где есть готовые строковые примитивы. В C++ ввод строки осуществляется так:

```
string username;
getline(cin, username);
```

Буфер строки `username` удаляется автоматически, поэтому даже не придется вызывать функцию `free()`.

Проблема переполнения буфера возникает при работе с любыми статическими массивами, а не только со строками. При написании безопасных программ следует тщательно проверять, не осуществляется ли запись в массив за его пределами.

### 10.6.2. Конкуренция доступа к каталогу /tmp

Другая распространенная проблема безопасности связана с созданием файлов с предсказуемыми именами, в основном в каталоге `/tmp`. Предположим, что программа `prog`, выполняющаяся от имени пользователя `root`, всегда создает временный файл `/tmp/prog` и помещает в него важную информацию. Тогда злоумышленник может заранее создать

символическую ссылку /tmp/prog на любой другой файл в системе. Когда программа попытается создать временный файл, функция open() завершится успешно, но в действительности вернет дескриптор символической ссылки. Любые данные, записываемые во временный файл, окажутся перенаправленными в файл злоумышленника.

В такой ситуации говорят о конкуренции. Она неявно существует между автором программы и хакером. Кто первым успеет создать временный файл, тот и победит.

Посредством этой атаки часто уничтожаются системные файлы. Создав нужную символическую ссылку, хакер может заставить программу, выполняющуюся с правами суперпользователя, затереть важный системный файл, например /etc/passwd.

Один из способов избежать такой атаки создавать временные файлы со случайными именами. Например, можно прочитать из устройства /dev/random случайные данные и включить их в имя файла. Это усложнит задачу хакеру, но не остановит его полностью. Он может попытаться создать большое число символических ссылок с потенциально верными именами. Даже если их будет 10000, одна верная догадка приведет к непоправимому.

Другой подход заключается в вызове функции open() с флагом O\_EXCL. Он заставляет функцию завершиться неудачей, если обнаруживается факт существования файла. К сожалению, это не срабатывает, если программа работает через NFS. Нельзя заранее предсказать, в какой файловой системе будет находиться программа, поэтому рассчитывать только на флаг O\_EXCL нельзя.

В разделе 2.1.7, "Временные файлы", рассказывалось о применении функции mkstemp() для создания временных файлов. К сожалению, в Linux эта функция открывает файл с флагом O\_EXCL после того, как было выбрано трудно угадываемое имя. Другими словами, применять функцию небезопасно, если каталог /tmp смонтирован через NFS.<sup>[35]</sup>

Прием, который всегда работает, заключается в вызове функции lstat() (рассматривается в приложении Б, "Низкоуровневый ввод-вывод") для созданного файла. Она отличается от функции stat() тем, что возвращает информацию о самой символической ссылке, а не о файле, на который она ссылается. Если функция сообщает, что новый файл является обычным файлом, а не символической ссылкой, и принадлежит владельцу программы, то все в порядке.

В листинге 10.5 представлена функция, которая пытается безопасно открыть файл в каталоге /tmp. Возможно, у этой функции есть свои слабости. Мы не рекомендуем читателям включать показанный код в свои программы без дополнительной экспертизы, просто мы хотим убедить читателей в том, что создание безопасных приложений непростая задача,

#### ***Листинг 10.5. (temp-file.c) Безопасное создание временного файла***

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

/* Функция возвращает дескриптор созданного временного файла.
Файл будет доступен для чтения и записи только тому
пользователю, чей идентификатор равен эффективному
идентификатору текущего процесса. Если файл не удалось создать,
возвращается -1. */
int secure_temp_file() {
/* Этот дескриптор ссылается на устройство /dev/random, из
которого будут получены случайные данные. */
static int random_fd = -1;
```

```

/* Случайное целое число. */
unsigned int random;
/* Буфер для преобразования числа в строку. */
char filename[128];
/* дескриптор создаваемого временного файла. */
int fd;
/* информация о созданном файле. */
struct stat stat_buf;

/* Если устройство /dev/random еще не было открыто,
открываем его. */
if (random_fd == -1) {
/* Открытие устройства /dev/random. Предполагается, что
это устройство является источником случайных данных,
а не файлом, созданным хакером. */
random_fd = open("/dev/random", O_RDONLY);
/* Если устройство /dev/random не удалось открыть,
завершаем работу. */
if (random_fd == -1)
return -1;
}

/* чтение целого числа из устройства /dev/random. */
if (read(random_fd, &random, sizeof(random)) != sizeof(random))
return -1;
/* Формирование имени файла из случайного числа. */
sprintf(filename, "/tmp/%u", random);
/* Попытка открытия файла. */
fd = open(filename,
/* Используем флаг O_EXCL. */
O_RDWR | O_CREAT | O_EXCL,
/* Разрешаем доступ только владельцу файла. */
S_IRUSR | S_IWUSR);
if (fd == -1)
return -1;

/* Вызываем функцию lstat(), чтобы проверить, не является ли
файл символической ссылкой */
if (lstat(filename, &stat_buf) == -1)
return -1;
/* Если файл не является обычным файлом, кто-то пытается
обмануть нас. */
if (!S_ISREG(stat_buf.st_mode))
return -1;
/* Если файл нам не принадлежит, то, возможно, кто-то успел
подменить его. */
if (stat_buf.st_uid != geteuid() ||
stat_buf.st_gid != getegid())
return -1;
/* Если у файла установлены дополнительные биты доступа,
что-то не так. */
if ((stat_buf.st_mode & ~(S_IRUSR | S_IWUSR)) != 0)
return -1;

return fd;
}

```

Рассмотренная функция вызывает функцию `open()` для создания файла, а затем функцию `lstat()` для проверки того, что файл не является символической ссылкой. Внимательный

читатель обнаружит здесь то, что называется состоянием гонки. Между вызовами функций `open()` и `lstat()` злоумышленник может успеть удалить файл и заменить его символической ссылкой. Это не вызовет разрушающих последствий, но приведет к тому, что функция завершится ошибкой и не сможет выполнить свою задачу. Такой тип атаки называется *отказом от обслуживания*.

В данной ситуации на помощь приходит sticky-бит. Поскольку для каталога `/tmp` он установлен, никто не сможет удалить файлы из этого каталога, не будучи их владельцем. Естественно, пользователю `root` разрешено делать все что угодно, но если хакер сумел получить привилегии суперпользователя, вас уже ничто не спасет.

Грамотный системный администратор не допустит, чтобы каталог `/tmp` был смонтирован через NFS, поэтому на практике можно пользоваться функцией `mkstemp()`. Если же речь идет о другом каталоге, то нельзя ни доверять флагу `O_EXCL`, ни рассчитывать на установку sticky-бита.

### 10.6.3. Функции `system()` и `popen()`

Третья распространенная проблема безопасности, о которой должен помнить каждый программист, заключается в несанкционированном запуске программ через интерпретатор команд. В качестве наглядной демонстрации рассмотрим сервер словарей. Серверная программа ожидает поступления запросов через Internet. Клиент посылает слово, а сервер сообщает, является ли оно корректным словом английского языка. В любой Linux-системе имеется файл `/usr/dict/words`, в котором содержится список 45000 слов, поэтому серверу достаточно выполнить такую команду:

```
% grep -x слово /usr/dict/words
```

Код завершения команды `grep` сообщит о том, обнаружено ли указанное слово в файле `/usr/dict/words`.

В листинге 10.6 показан пример реализации поискового модуля сервера.

#### *Листинг 10.6. (grep-dictionary.c) Поиск слова в словаре*

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

/* Функция возвращает ненулевое значение, если аргумент WORD
встречается в файле /usr/dict/words. */
int grep_for_word(const char* word) {
    size_t length;
    char* buffer;
    int exit_code;
    /* Формирование строки 'grep -x WORD /usr/dict/words'.
    Строка выделяется динамически во избежание
    переполнения буфера. */
    length =
        strlen("grep -x ") + strlen(word) +
        strlen(" /usr/dict/words") + 1;
    buffer = (char*)malloc(length);
    sprintf(buffer, "grep -x %s /usr/dict/words", word);

    /* Запуск команды. */
    exit_code = system(buffer);
    /* Очистка буфера. */
```

```
free(buffer);  
/* Если команда grep вернула значение 0, значит, слово найдено  
в словаре. */  
return exit_code == 0;  
}
```

Обратите внимание на подсчет числа символов в строке и динамическое выделение буфера, что позволяет обезопасить программу от переполнения буфера. К сожалению, небезопасна сама функция `system()` (описана в разделе 3.2.1, "Функция `system()`"). Функция вызывает стандартный интерпретатор команд и принимает от него код завершения. Но что произойдет, если злоумышленник вместо слова введет показанную ниже строку?

```
foo /dev/null; rm -rf /
```

В этом случае сервер выполнит такую команду:

```
grep -x foo /dev/null; rm -rf / /usr/dict/words
```

Теперь проблема стала очевидной. Пользователь запустил одну команду, якобы `grep`, а на самом деле их оказалось две, так как интерпретатор считает точку с запятой разделителем команд. Первая команда это по-прежнему безобидный вызов утилиты `grep`, зато вторая команда пытается удалить все файлы в системе. Даже если серверная программа не имеет привилегий суперпользователя, она удалит все файлы, доступные запустившему ее пользователю. Похожая проблема возникает и при использовании функции `popen()` (описана в разделе 3.4.4, "Функции `popen()` и `pclose()`"), которая создает канал между родительским и дочерним процессами, но тоже вызывает интерпретатор для запуска команды.

Существуют два способа устранения подобных проблем. Первый заключается в использовании функции семейства `exec()` вместо функции `system()` или `popen()`. Специальные символы интерпретатора команд (например, точка с запятой) не подвергаются обработке, если они присутствуют в списке аргументов функции `exec()`. Естественно, при этом пропадают преимущества таких функций, как `system()` и `popen()`.

Второй способ проверка строки на предмет "благонадежности". В случае сервера словарей следует убедиться в том, что слово содержит только буквы (для этого предназначена функция `isalpha()`). Такое слово не представляет угрозы.

# Глава 11

## Демонстрационное Linux-приложение

В этой главе кусочки мозаики сложатся в единую композицию. Мы опишем и реализуем законченную Linux-программу, в которой объединятся многие рассмотренные в данной книге методики. Программа через протокол HTTP выдает информацию о системе, в которой она работает.

### 11.1. Обзор

Демонстрационная программа является частью пакета мониторинга Linux-системы и предоставляет следующие возможности.

- Программа реализует минимально необходимые функции Web-сервера. Локальные и удаленные клиенты получают доступ к системной информации, запрашивая Web-страницы у сервера по протоколу HTTP.

- Программа не работает со статическими HTML-страницами. Все страницы динамически генерируются модулями, каждый из которых вычисляет итоговую информацию о какой-либо характеристике системы.

- Все модули подключаются к серверу динамически, загружаясь из совместно используемых библиотек. Их можно добавлять, удалять и заменять по ходу работы сервера.

- Для каждого запроса на подключение сервер создает дочерний процесс. Это позволяет серверу продолжать реагировать на запросы, а также защищает его от ошибок в модулях.

- Серверу не требуются привилегии суперпользователя (он не работает с привилегированным портом). Это ограничивает его в доступе к системной информации.

Программу сопровождают четыре модуля, в которых иллюстрируются методики сбора системной информации. В модуле `time` используется системный вызов `gettimeofday()`. В модуле `issue` применяются функции низкоуровневого ввода-вывода и системный вызов `sendfile()`. В модуле `diskfree` показано, как с помощью функций `fork()`, `exec()` и `dup2()` выполнять команды в дочерних процессах. В модуле `processes` продемонстрирована работа с файловой системой `/proc`.

#### 11.1.1. Существующие ограничения

Программа обладает многими функциональными возможностями, которые ожидаются от полноценного приложения. В частности, она имеет средства анализа командной строки и проверки ошибок. Одновременно с этим она немного упрощений, так как нам хотелось сделать ее понятнее и сосредоточить внимание читателей на представленных в книге методиках. При анализе программного кода помните о следующих ограничениях.

- Мы не пытались создать полноценную реализацию протокола HTTP. Воплощены лишь те его функции, которые достаточны для организации взаимодействия Web-сервера и клиентов. В реальных приложениях используются готовые реализации Web-сервера. [\[36\]](#)

- Программа не претендует на полную совместимость со спецификациями HTML (<http://www.w3.org/MarkUp/>). Она генерирует простые HTML-страницы, которые могут обрабатываться популярными Web-браузерами.

- Сервер не настроен на максимальную производительность или минимальное потребление

ресурсов. В частности, мы сознательно опустили код сетевой настройки, обычно имеющийся у Web-сервера. Рассмотрение этой темы выходит за рамки нашей книги.

■Мы не пытаемся регулировать объем ресурсов (число процессов, объем используемой памяти), потребляемых сервером или его модулями. Многие многозадачные Web-серверы обслуживают запросы посредством фиксированного пула процессов, а не создают новый дочерний процесс для каждого соединения.

■Всякий раз, когда поступает запрос, сервер загружает библиотеку с модулем, которая немедленно выгружается по окончании обработки запроса. Эффективнее было бы кэшировать загруженные модули.

## ***Протокол HTTP***

Протокол HTTP (Hypertext Transport Protocol) используется для организации взаимодействия Web-клиентов и серверов. Клиент подключается к серверу, устанавливая соединение с заранее известным портом (обычно его номер 80). Запросы и заголовки HTTP представляются в виде обычного текста.

Подключившись к серверу, клиент посылает запрос. Типичный запрос выглядит так: `GET /page HTTP/1.0`. Метод `GET` означает запрос на получение Web-страницы. Второй элемент это путь к странице. В третьем элементе указан протокол и его версия. В последующих строках содержатся поля заголовка отформатированные наподобие заголовков почтовых сообщений. В них приведена дополнительная информация о клиенте. Заголовок оканчивается пустой строкой.

В ответ сервер сообщает результат обработки запроса. Типичный ответ таков: `HTTP/1.0 200 OK`. Первый элемент это версия протокола. В следующих двух элементах описан результат. В данном случае код 200 означает успешное выполнение запроса. Далее идут поля заголовка, который, оканчивается пустой строкой. После заголовка сервер может передать произвольные данные.

Обычно сервер возвращает HTML-код Web-страницы. В рассматриваемом примере в заголовке ответа будет указано следующее: `Content-type: text/html`.

Спецификацию протокола HTTP можно получить по адресу <http://www.w3.org/Protocols>.

## **11.2. Реализация**

Во всех более-менее сложных С-программах требуется тщательно продумать организацию, чтобы сохранить модульность и обеспечить удобство сопровождения. Наша демонстрационная программа разделена на четыре главных исходных файла.

В каждом исходном файле экспортируются функции и переменные, используемые в других частях программы. Для простоты все они объявлены в одном файле заголовков: `server.h` (листинг 11.1). Функции, применяемые в рамках только одного модуля, объявлены со спецификатором `static` и не включены в файл `server.h`.

### ***Листинг 11.1. (server.h) Объявления функций и переменных***

```
#ifndef SERVER_H
```

```

#define SERVER_H

#include <netinet/in.h>
#include <sys/types.h>

/**** Символические константы файла common.c. *****/

/* Имя программы. */
extern const char* program_name;

/* Если не равна нулю, отображаются развернутые сообщения. */
extern int verbose;

/* Напоминает функцию malloc(), не прерывает работу программы,
если выделить память не удалось. */
extern void* xmalloc(size_t size);

/* Напоминает функцию realloc(), но прерывает работу программы,
если выделить память не удалось */
extern void* xrealloc(void* ptr, size_t size);

/* Напоминает функцию strdup(), но прерывает работу программы,
если выделить память не удалось. */
extern char* xstrdup(const char* s);

/* Выводит сообщение об ошибке заданного системного вызова
и завершает работу программы. */
extern void system_error(const char* operation);

/* Выводит сообщение об ошибке и завершает работу программы. */
extern void error(const char* cause, const char* message);

/* Возвращает имя каталога, содержащего исполняемый файл
программы. Поскольку возвращается указатель на область памяти,
вызывающая подпрограмма должна удалить ее с помощью
функции free(). В случае неудачи выполнение программы
завершается. */
extern char* get_self_executable_directory();

/**** Символические константы файла module.c *****/

/* Экземпляр загруженного серверного модуля. */
struct server_module {
/* Дескриптор библиотеки, в которой находится модуль. */
void* handle;
/* Описательное имя модуля. */
const char* name;
/* Функция, генерирующая HTML-код для модуля. */
void (*generate_function)(int);
};

/* Каталог, из которого загружаются модули. */
extern char* module_dir;

/* Функция, пытающаяся загрузить указанный серверный модуль.
Если модуль существует, возвращается структура

```



```

с его описанием, в противном случае возвращается NULL. */
extern struct server_module* module_open(const char* module_path);

/* Закрытие модуля и удаление объекта MODULE. */
extern void module_close(struct server_module* module);

/**/ Символические константы файла server.c. /**/

/* Запуск сервера по адресу LOCAL_ADDRESS и порту PORT. */
extern void server_run(struct in_addr local_address, uint16_t port);

#endif /* SERVER_H */

```

### 11.2.1. Общие функции

Файл common.c (листинг 11.2) содержит функции общего назначения, используемые в разных частях программы.

#### *Листинг 11.2. (common.c) Функции общего назначения*

```

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "server.h"

const char* program_name;
int verbose;

void* xmalloc(size_t size) {
void* ptr = malloc(size);
/* Аварийное завершение, если выделить память не удалось. */
if (ptr == NULL)
abort();
else
return ptr;
}

void* xrealloc(void* ptr, size_t size) {
ptr = realloc(ptr, size);
/* Аварийное завершение, если выделить память не удалось. */
if (ptr == NULL)
abort();
else
return ptr;
}

char* xstrdup(const char* s) {
char* copy = strdup(s);
/* Аварийное завершение, если выделить память не удалось. */
if (copy == NULL)
abort();
else

```

```

return copy;
}

void system_error(const char* operation) {
/* Вывод сообщения об ошибке на основании значения
переменной errno. */
error(operation, strerror(errno));
}


void error(const char* cause, const char* message) {
/* Запись сообщения об ошибке в поток stderr. */
fprintf(stderr, "%s: error: (%s) %s\n", program_name,
cause, message);
/* Завершение программы */
exit(1);
}

char* get_self_executable_directory() {
int rval;
char link_target[1024];
char* last_slash;
size_t result_length;
char* result;

/* Чтение содержимого символической ссылки /proc/self/exe. */
rval =
readlink("/proc/self/exe", link_target,
sizeof(link_target));
if (rval == -1)
/* Функция readlink() завершилась неудачей, поэтому выходим
из программы. */
abort();
else
/* Запись нулевого символа в конец строки. */
link_target[rval] = '\0';
/* Удаление имени файла,
чтобы осталось только имя каталога. */
last_slash = strrchr(link_target, '/');
if (last_slash == NULL || last_slash == link_target)
/* Формат имени некорректен. */
abort();
/* Выделение буфера для результирующей строки. */
result_length = last_slash - link_target;
result = (char*)xmalloc(result_length + 1);
/* Копирование результата. */
strncpy(result, link_target, result_length);
result[result_length] = '\0';
return result;
}

```

Приведенные здесь функции можно использовать в самых разных программах.

■ Функции `xmalloc()`, `xrealloc()`  `strdup()` являются расширенными версиями стандартных функций `malloc()`, `realloc()` и `strdup()`, в которые дополнительно включен код проверки ошибок. В отличие от стандартных функций, которые возвращают пустой указатель в случае ошибки, наши функции немедленно завершают работу программы, если в системе недостаточно памяти.

Раннее обнаружение нехватки памяти хорошая идея. Если этого не делать, пустые указатели будут появляться в самых неожиданных местах программы. Ситуации, связанные с нехваткой

памяти, непросто воспроизвести, поэтому их отладка будет затруднена. Ошибки выделения памяти обычно имеют катастрофические последствия для программы, так что аварийное ее завершение вполне приемлемый вариант реакции.

■Функция `error()` сообщает о фатальной ошибке, произошедшей в программе. При этом в поток `stderr` записывается сообщение об ошибке, и работа программы завершается. Для ошибок, произошедших в системных вызовах или библиотечных функциях, предназначена функция `system_error()`, которая генерирует сообщение об ошибке на основании значения переменной `errno` (см. раздел 2.2.3, "Коды ошибок системных вызовов").

■Функция `get_self_executable_directory()` определяет каталог, в котором содержится исполняемый файл текущего процесса. Это позволяет программе находить свои внешние компоненты. Функция проверяет содержимое символической ссылки `/proc/self/exe` (см. раздел 7.2.1, "Файл `/proc/self`").

В файле `common.c` определены также две полезные глобальные переменные.

■Переменная `program_name` содержит имя выполняемой программы, указанное в списке аргументов командной строки (см. раздел 2.1.1, "Список аргументов").

■Переменная `verbose` не равна нулю, если программа работает в режиме выдачи развернутых сообщений. В таком случае многие компоненты будут записывать в поток `stdout` сообщения о ходе выполнения задачи.

## 11.2.2. Загрузка серверных модулей

В файле `module.c` (листинг 11.3) содержится реализация динамически загружаемых серверных модулей. Загруженному модулю соответствует структура типа `server_module`, который определен в файле `server.h`.

### *Листинг 11.3. (module.c) Загрузка и выгрузка серверных модулей*

```
#include <dlfcn.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "server.h"

char* module_dir;

struct server_module* module_open(const char* module_name) {
    char* module_path;
    void* handle;
    void (*module_generate)(int);
    struct server_module* module;
    /* Формирование путевого имени библиотеки, в которой содержится
    загружаемый модуль. */
    module_path =
        (char*)xmalloc(strlen(module_dir) +
            strlen(module_name) + 2);
    sprintf(module_path, "%s/%s", module_dir, module_name);

    /* Попытка открыть файл MODULE_PATH как совместно используемую
    библиотеку. */
    handle = dlopen(module_path, RTLD_NOW);
```

```

free (module_path);
if (handle == NULL) {
/* Ошибка: либо путь не существует, либо файл не является
совместно используемой библиотекой. */
return NULL;
}

/* Чтение константы module_generate из библиотеки. */
module_generate =
(void(*)int)dlsym(handle,
"module_generate");
/* Проверяем, найдена ли константа. */
if (module_generate == NULL) {
/* Константа отсутствует в библиотеке. Очевидно, файл не
является серверным модулем. */
dlclose(handle);
return NULL;
}

/* Выделение и инициализация объекта server_module. */
module =
(struct server_module*)xmalloc
(sizeof (struct server_module));
module->handle = handle;
module->name = xstrdup(module_name);
module->generate_function = module_generate;
/* Успешное завершение функции. */
return module;
}

void module_close(struct server_module* module) {
/* Заккрытие библиотеки. */
dlclose(module->handle);
/* Удаление строки с именем модуля. */
free((char*)module->name);
/* Удаление объекта module. */
free(module);
}

```

Каждый модуль содержится в файле совместно используемой библиотеки (см. раздел 2.3.2, "Совместно используемые библиотеки") и должен экспортировать функцию `module_generate()`. Эта функция генерирует HTML-код Web-страницы и записывает его в сокет, дескриптор которого передан ей в качестве аргумента.

В файле `module.c` определены две функции.

■ Функция `module_open()` пытается загрузить серверный модуль с указанным именем. Файл модуля имеет расширение `.so`, так как это совместно используемая библиотека. Функция открывает библиотеку с помощью функции `dlopen()` и ищет в библиотеке константу `module_generate` посредством функции `dlsym()` (описаны в разделе 2.3.6, "Динамическая загрузка и выгрузка"). Если библиотеку не удалось открыть или в ней не обнаружена экспортируемая константа `module_generate`, возвращается значение `NULL`. В противном случае выделяется и возвращается объект `module`.

■ Функция `module_close()` закрывает совместно используемую библиотеку, соответствующую указанному модулю, и удаляет объект `module`.

В файле `module.c` определена также глобальная переменная `module_dir`. В ней записано имя каталога, в котором функция `module_open()` будет искать совместно используемые библиотеки.

## 11.2.3. Сервер

Файл `server.c` (листинг 11.4) представляет собой реализацию простейшего HTTP-сервера.

### *Листинг 11.4. (server.c) Реализация HTTP-сервера*

```
#include <arpa/inet.h>
#include <assert.h>
#include <errno.h>
#include <netinet/in.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <unistd.h>
#include "server.h"

/* HTTP-ответ и заголовок, возвращаемые в случае
успешной обработки запроса. */
static char* ok_response =
"HTTP/1.0 100 OK\n"
"Content-type: text/html\n"
"\n";

/* HTTP-ответ, заголовок и тело страницы на случай
непонятого запроса. */
static char* bad_request_response =
"HTTP/1.0 400 Bad Request\n"
"Content-type: text/html\n"
"\n"
"<html>\n"
" <body>\n"
" <h1>Bad Request</h1>\n"
" <p>This server did not understand your request.</p>\n"
" </body>\n"
"</html>\n";

/* HTTP-ответ, заголовок и шаблон страницы на случай,
когда запрашиваемый документ не найден. */
static char* not_found_response_template =
"HTTP/1.0 404 Not Found\n"
"Content-type: text/html\n"
"\n"
"<html>\n"
" <body>\n"
" <h1>Not Found</h1>\n"
" <p>The requested URL %s was not found on this server.</p>\n"
" </body>\n"
"</html>\n";

/* HTTP-ответ, заголовок к шаблон страницы на случай,
когда запрашивается непонятный метод */
static char* bad_method_response_template =
"HTTP/1.0 501 Method Not Implemented\n"
```

```

"Content-type: text/html\n"
"\n"
"<html>\n"
" <body>\n"
" <h1>Method Not Implemented</h1>\n"
" <p>The method %s is not implemented by this server.</p>\n"
" </body>\n"
"</html>\n";

```

```

/* Обработчик сигнала SIGCHLD, удаляющий завершившиеся
дочерние процессы. */
static void clean_up_child_process(int signal_number) {
int status;
wait(&status);
}

```

```

/* Обработка HTTP-запроса "GET" к странице PAGE и
запись результата в файл с дескриптором CONNECTION_FD. */
static void handle_get(int connection_fd, const char* page) {
struct server_module* module = NULL;

```

```

/* Убеждаемся, что имя страницы начинается с косой черты и
не содержит других символов косой черты, так как
подкаталоги не поддерживаются. */
if (*page == '/' && strchr(page + 1, '/') == NULL) {
char module_file_name[64];

```

```

/* Имя страницы правильно. Формируем имя модуля, добавляя
расширение ".so" к имени страницы. */
snprintf(module_file_name, sizeof(module_file_name),
"%s.so", page + 1);
/* Попытка открытия модуля. */
module = module_open(module_file_name);
}
if (module == NULL) {
/* Имя страницы неправильно сформировано или не удалось
открыть модуль с указанным именем. В любом случае
возвращается HTTP-ответ "404. Not Found". */
char response[1024];

```

```

/* Формирование ответного сообщения. */
snprintf(response, sizeof(response),
not_found_response_template, page);
/* Отправка его клиенту. */
write(connection_fd, response, strlen(response));
} else {
/* Запрашиваемый модуль успешно загружен. */

```

```

/* Выдача HTTP-ответа, обозначающего успешную обработку
запроса, и HTTP-заголовка для HTML-страницы. */
write(connection_fd, ok_response, strlen(ok_response));
/* Вызов модуля, генерирующего HTML-код страницы и
записывающего этот код в указанный файл. */
(*module->generate_function)(connection_fd);
/* Работа с модулем окончена. */
module_close(module);
}
}

```

```

/* Обработка клиентского запроса на подключение. */
static void handle_connection(int connection_fd) {
char buffer[256];
ssize_t bytes_read;

/* Получение данных от клиента. */
bytes_read =
read(connection_fd, buffer, sizeof(buffer) 1);
if (bytes_read > 0) {
char method[sizeof(buffer)];
char url[sizeof(buffer)];
char protocol[sizeof(buffer)];

/* Часть данных успешно прочитана. Завершаем буфер
нулевым символом, чтобы его можно было использовать
в строковых операциях. */
buffer[bytes_read] = '\0';
/* Первая строка, посылаемая клиентом, -- это HTTP-запрос.
В запросе указаны метод, запрашиваемая страница и
версия протокола. */
sscanf(buffer, "%s %s %s", method, url, protocol);
/* В заголовке, стоящем после запроса, может находиться
любая информация. В данной реализации HTTP-сервера
эта информация не учитывается. Тем не менее необходимо
прочитать все данные, посылаемые клиентом. Данные читаются
до тех пор, пока не встретится конец заголовка,
обозначаемый пустой строкой. В HTTP пустой строке
соответствуют символы CR/LF. */
while (strstr(buffer, " \r\n\r\n") == NULL)
bytes_read = read(connection_fd, buffer, sizeof(buffer));
/* Проверка правильности последней операции чтения.
Если она не завершилась успешно, произошел разрыв
соединения, поэтому завершаем работу. */
if (bytes_read == -1) {
close(connection_fd);
return;
}
/* Проверка поля версии. Сервер понимает протокол HTTP
версий 1.0 и 1.1. */
if (strcmp(protocol, "HTTP/1.0") &&
strcmp(protocol, "HTTP/1.1")) {
/* Протокол не поддерживается. */
write(connection_fd, bad_request_response,
sizeof(bad_request_response));
} else if (strcmp(method, "GET")) {
/* Сервер реализует только метод GET, а клиент указал
другой метод. */
char response[1024];
snprintf(response, sizeof(response),
bad_method_response_template, method);
write(connection_fd, response, strlen(response));
} else
/* Корректный запрос. Обрабатываем его. */
handle_get(connection_fd, url);
} else if (bytes_read == 0)
/* Клиент разорвал соединение, не успев отправить данные.
Ничего не предпринимаем */
;

```

```

else
/* Операция чтения завершилась ошибкой. */
system_error("read");
}

void server_run(struct in_addr local_address, uint16_t port) {
struct sockaddr_in socket_address;
int rval;
struct sigaction sigchld_action;
int server_socket;

/* Устанавливаем обработчик сигнала SIGCHLD, который будет
удалять завершившиеся дочерние процессы. */
memset(&sigchld_action, 0, sizeof(sigchld_action));
sigchld_action.sa_handler = &clean_up_child_process;
sigaction(SIGCHLD, &sigchld_action, NULL);

/* Создание TCP-сокета */
server_socket = socket(PF_INET, SOCK_STREAM, 0);
if (server_socket == -1) system_error("socket");
/* Создание адресной структуры, определяющей адрес
для приема запросов. */
memset(&socket_address, 0, sizeof(socket_address));
socket_address.sin_family = AF_INET;
socket_address.sin_port = port;
socket_address.sin_addr = local_address;
/* Привязка сокета к этому адресу. */
rval =
bind(server_socket, &socket_address,
sizeof(socket_address));
if (rval != 0)
system_error("bind");
/* Перевод сокета в режим приема запросов. */
rval = listen(server_socket, 10);
if (rval != 0)
system_error("listen");

if (verbose) {
/* В режиме развернутых сообщений отображаем адрес и порт,
с которыми работает сервер. */
socklen_t address_length;

/* Нахождение адреса сокета. */
address_length = sizeof(socket_address);
rval =
getsockname(server_socket, &socket_address, &address_length);
assert(rval == 0);
/* Вывод сообщения. Номер порта должен быть преобразован
из сетевого (обратного) порядка следования байтов
в серверный (прямой). */
printf("server listening on %s:%d\n",
inet_ntoa(socket_address.sin_addr),
(int)ntohs(socket_address.sin_port));
}

/* Бесконечный цикл обработки запросов. */
while (1) {
struct sockaddr_in remote_address;

```



```

socklen_t address_length;
int connection;
pid_t child_pid;

/* Прием запроса. Эта функция блокируется до тех пор, пока
не поступит запрос. */
address_length = sizeof(remote_address);
connection = accept(server_socket, &remote_address,
&address_length);
if (connection == -1) {
/* Функция завершилась неудачно. */
if (errno == EINTR)
/* Функция была прервана сигналом. Повторная попытка. */
continue;
else
/* Что-то случилось. */
system_error("accept");
}

/* Соединение установлено. Вывод сообщения, если сервер
работает в режиме развернутых сообщений. */
if (verbose) {
socklen_t address_length;
/* Получение адреса клиента. */
address_length = sizeof(socket_address);
rval =
getpeername(connection, &socket_address, &address_length);
assert(rval == 0);
/* Вывод сообщения. */
printf("connection accepted from %s\n",
inet_ntoa(socket_address.sin_addr));
}

/* Создание дочернего процесса для обработки запроса. */
child_pid = fork();
if (child_pid == 0) {
/* Это дочерний процесс. Потоки stdin и stdout ему не нужны,
поэтому закрываем их. */
close(STDIN_FILENO);
close(STDOUT_FILENO);
/* Дочерний процесс не должен работать с серверным сокетом,
поэтому закрываем его дескриптор. */
close(server_socket);
/* Обработка запроса. */
handle_connection(connection);
/* Обработка завершена. Закрываем соединение и завершаем
дочерний процесс. */
close(connection);
exit(0);
} else if (child_pid > 0) {
/* Это родительский процесс. Дескриптор клиентского сокета
ему не нужен. Переход к приему следующего запроса. */
close(connection);
} else
/* Вызов функции fork() завершился неудачей. */
system_error("fork");
}
}

```

В файле `server.c` определены следующие функции.

■ Функция `server_run()` является телом сервера. Она запускает сервер и начинает принимать запросы на подключение, не завершаясь до тех пор, пока не произойдет серьезная ошибка. Сервер создает потоковый TCP-сокет (см. раздел 5.5.3, "Серверы").

Первый аргумент функции `server_run` определяет локальный адрес, по которому принимаются запросы. У компьютера может быть несколько адресов, каждый из которых соответствует определённому сетевому интерфейсу.<sup>[37]</sup> Данный аргумент ограничивает работу сервера конкретным интерфейсом или разрешает принимать запросы отовсюду, если равен `INADDR_ANY`.

Второй аргумент функции `server_run()` это номер порта сервера. Если порт уже используется или является привилегированным, работа сервера завершится. Когда номер порта задан равным нулю, ОС Linux автоматически выберет неиспользуемый порт.

Для обработки каждого клиентского запроса сервер создает дочерний процесс с помощью функции `fork()` (см. раздел 3.2.2. "Функции `fork()` и `exec()`"), в то время как родительский процесс продолжает принимать новые запросы. Дочерний процесс вызывает функцию `handle_connection()`, после чего закрывает соединение и завершается.

■ Функция `handle_connection()` обрабатывает отдельный клиентский запрос, принимая в качестве аргумента дескриптор сокета. Функция читает данные из сокета и пытается интерпретировать их как HTTP-запрос на получение страницы.

Сервер обрабатывает только запросы протокола HTTP версий 1.0 и 1.1. Столкнувшись с иными протоколом или версией сервер возвращает HTTP-код 400 и сообщение `bad_request_response`. Сервер понимает только HTTP-метод GET. Если клиент запрашивает какой-то другой метод, сервер возвращает HTTP-код 501 и сообщение `bad_method_response_template`.

■ Если клиент послал правильно сформированный запрос GET, функция `handle_connection()` вызывает функцию `handle_get()`, которая обрабатывает запрос. Эта функция пытается загрузить серверный модуль, имя которого генерируется на основании имени запрашиваемой страницы. Например, когда клиент запрашивает страницу с именем "information", делается попытка загрузить модуль `information.so`. Если модуль не может быть загружен, функция `handle_get()` возвращает HTTP-код 404 и сообщение `not_found_response_template`.

В случае обращения к верной странице функция `handle_get()` возвращает клиенту HTTP-код 200, указывающий на успешную обработку запроса, и вызывает функцию `module_generate()`, содержащуюся в модуле. Последняя генерирует HTML-код Web-страницы и посылает его клиенту.

■ Функция `server_run()` регистрирует функцию `clean_up_child_process()` в качестве обработчика сигнала `SIGCHLD`. Обработчик просто очищает ресурсы завершившегося дочернего процесса (см. раздел 3.4.4. "Асинхронное удаление дочерних процессов").

#### 11.2.4. Основная программа

В файле `main.c` (листинг 11.5) содержится функция `main()` сервера. Она отвечает за анализ аргументов командной строки и обнаружение ошибок в них, а также за конфигурирование и запуск сервера.

**Листинг 11.5. (`main.c`) Главная серверная функция, выполняющая анализ аргументов командной строки**

```

#include <assert.h>
#include <getopt.h>
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <unistd.h>
#include "server.h"

/* Описание длинных опций для функции getopt_long(). */
static const struct option long_options[] = {
    { "address", 1, NULL, 'a' },
    { "help", 0, NULL, 'h' },
    { "module-dir", 1, NULL, 'm' },
    { "port", 1, NULL, 'p' },
    { "verbose", 0, NULL, 'v' },
};

/* Описание коротких опций для функции getopt_long(). */
static const char* const short_options = "a:hm:p:v";

/* Сообщение о том, как правильно использовать программу. */
static const char* const usage_template =
    "Usage: %s { options }\n"
    " -a, --addressADDR Bind to local address (by default, bind\n"
    " to all local addresses).\n"
    " -h, --help Print this information.\n"
    " -m, --module-dir DIR Load modules from specified directory\n"
    " (by default, use executable directory).\n"
    " -p, --portPORT Bind to specified port.\n"
    " -v, --verbose Print verbose messages.\n";

/* Вывод сообщения о правильном использовании программы
и завершение работы. Если аргумент IS_ERROR не равен нулю,
сообщение записывается в поток stderr и возвращается
признак ошибки, в противном случае сообщение выводится в
поток stdout и возвращается обычный нулевой код. */
static void print_usage(int is_error) {
    fprintf(is_error ? stderr : stdout, usage_template,
        program_name);
    exit(is_error ? 1 : 0);
}

int main(int argc, char* const argv[]) {
    struct in_addr local_address;
    uint16_t port;
    int next_option;

    /* Сохранение имени программы для отображения в сообщениях
    об ошибке. */
    program_name = argv[0];

    /* Назначение стандартных установок. По умолчанию сервер
    связан со всеми локальными адресами, и ему автоматически
    назначается неиспользуемый порт. */
    local_address.s_addr = INADDR_ANY;
    port = 0;

```

```

/* Не отображать развернутые сообщения. */
verbose = 0;
/* Загружать модули из каталога, в котором содержится
исполняемый файл. */
module_dir = get_self_executable_directory();
assert(module_dir != NULL);

/* Анализ опций. */
do {
next_option =
getopt_long(argc, argv, short_options,
long_options, NULL);
switch (next_option) {
case 'a':
/* Пользователь ввел -a или --address. */
{
struct hostent* local_host_name;

/* Поиск заданного адреса. */
local_host_name = gethostbyname(optarg);
if (local_host_name == NULL ||
local_host_name->h_length == 0)
/* Не удалось распознать имя. */
error(optarg, "invalid host name");
else
/* Введено правильное имя */
local_address.s_addr =
*((int*)(local_host_name->h_addr_list[0]));
}
break;
case 'h':
/* Пользователь ввёл -h или --help. */
print_usage(0);
case 'm':
/* Пользователь ввел -m или --module-dir. */
{
struct stat dir_info;

/* Проверка существования каталога */
if (access(optarg, F_OK) != 0)
error(optarg, "module directory does not exist");
/* Проверка доступности каталога. */
if (access(optarg, R_OK | X_OK) != 0)
error(optarg, "module directory is not accessible");
/* Проверка того, что это каталог. */
if (stat(optarg, &dir_info) != 0 || !S_ISDIR(dir_info.st_mode))
error(optarg, "not a directory");
/* Все правильно. */
module_dir = strdup(optarg);
}
break;
case 'p':
/* Пользователь ввел -p или --port. */
{
long value;
char* end;

value = strtol(optarg, &end, 10);
if (*end != '\0')

```

```

/* В номере порта указаны не только цифры. */
print_usage(1);
/* Преобразуем номер порта в число с сетевым (обратным)
порядком следования байтов. */
port = (uint16_t)htons(value);
}
break;
case 'v':
/* Пользователь ввел -v или --verbose. */
verbose = 1;
break;
case '?':
/* Пользователь ввел непонятную опцию. */
print_usage(1);
case -1:
/* Обработка опций завершена. */
break;
default:
abort();
}
} while (next_option != -1);

/* Программа не принимает никаких дополнительных аргументов.
Если они есть, выдается сообщение об ошибке. */
if (optind != argc)
print_usage(1);

/* Отображение имени каталога, если программа работает в режиме
развернутых сообщений. */
if (verbose)
printf("modules will be loaded from %s\n", module_dir);

/* Запуск сервера. */
server_run(local_address, port);
return 0;
}

```

Файл `main.c` содержит следующие функции.

■ Функция `getopt_long()` (см. раздел 21.3, "Функция `getopt_long()`") вызывается для анализа опций командной строки. Опции могут задаваться в двух форматах: длинном и коротком. Описание длинных опций приведено в массиве `long_options`, а коротких в массиве `short_options`.

По умолчанию серверный порт имеет номер 0, а локальный адрес задан в виде константы `INADDR_ANY`. Эти установки можно переопределить с помощью опций `--port (-p)` и `--address (-a)` соответственно. Если пользователь ввел адрес, вызывается библиотечная функция `gethostbyname()`, преобразующая его в числовой Internet-адрес. [\[38\]](#)

По умолчанию серверные модули загружаются из каталога, где находится исполняемый файл. Этот каталог определяется с помощью функции `get_self_executable_directory()`. Данную установку можно переопределить с помощью опции `--module (-m)`. В таком случае проверяется, является ли указанный каталог доступным.

По умолчанию развернутые сообщения не отображаются, если не указать опцию `--verbose (-v)`.

■ Если пользователь ввел опцию `--help (-h)` или указал неправильную опцию, вызывается функция `print_usage()`, которая отображает сообщение о правильном использовании программы и завершает работу.

## 11.3. Модули

В дополнение к основной программе созданы четыре модуля, в которых реализованы функции сервера. Чтобы создать собственный модуль, достаточно определить функцию `module_generate()`, которая будет возвращать HTML-код.

### 11.3.1. Отображение текущего времени

Модуль `time.so` (исходный текст приведен в листинге 11.6) генерирует простую страницу, где отображается текущее время на сервере. В функции `module_generate()` вызывается функция `gettimeofday()`, возвращающая значение текущего времени (см. раздел 8.7, "Функция `gettimeofday()`: системные часы"), после чего функции `localtime()` и `strftime()` преобразуют это значение в текстовый формат. Полученная строка встраивается в шаблон HTML-страницы `page_template`.

#### *Листинг 11.6. (time.c) серверный модуль, отображающий текущее время*

```
#include <assert.h>
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include "server.h"

/* шаблон HTML-страницы, генерируемой данным модулем. */
static char* page_template =
"<html>\n"
" <head>\n"
" <meta http-equiv=\"refresh\" content=\"5\">\n"
" </head>\n"
" <body>\n"
" <p>The current time is %s </p>\n"
" </body>\n"
"</html>\n";

void module_generate(int fd) {
    struct timeval tv;
    struct tm* ptm;
    char time_string[40];
    FILE* fp;

    /* Определение времени суток и заполнение структуры типа tm. */
    gettimeofday(&tv, NULL);
    ptm = localtime(&tv.tv_sec);

    /* Получение строкового представления времени с точностью
    до секунды. */
    strftime(time_string, sizeof(time_string), "%H:%M:%S", ptm);

    /* Создание файлового потока, соответствующего дескриптору
    клиентского сокета. */
    fp = fdopen(fd, "w");
    assert(fp != NULL);
```

```

/* Запись HTML-страницы. */
fprintf(fp, page_template, time_string);
/* Очистка буфера потока */
fflush(fp);
}

```

Для удобства в этом модуле используются стандартные библиотечные функции ввода-вывода. Функция `fdopen()` возвращает указатель потока (`FILE*`), соответствующий дескриптору клиентского сокета (подробнее об этом рассказывается в приложении Б, "Низкоуровневый ввод-вывод"). Для отправки страницы клиенту вызывается обычная функция `fprintf()`, а функция `fflush()` предотвращает потерю данных в случае закрытия сокета.

HTML-страница, возвращаемая модулем `time.so`, содержит в заголовке тэг `<meta>`, который служит клиенту указанием перезагружать страницу каждые 5 секунд. Благодаря этому клиент всегда будет знать точное время.

### 11.3.2. Отображение версии Linux

Модуль `issue.so` (исходный текст приведен в листинге 11.7) выводит информацию о дистрибутиве Linux, с которым работает сервер. Традиционно эта информация хранится в файле `/etc/issue`. Модель посылает клиенту Web-страницу с содержимым файла, заключенным в тэге `<pre></pre>`.

#### *Листинг 11.7. (issue.c) Серверный модуль, отображающий информацию о дистрибутиве Linux*

```

#include <fcntl.h>
#include <string.h>
#include <sys/sendfile.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include "server.h"

/* HTML-код начала генерируемой страницы. */
static char* page_start =
"<html>\n"
" <body>\n"
" <pre>\n";

/* HTML-код конца генерируемой страницы. */

static char* page_end =
" </pre>\n"
" </body>\n"
"</html>\n";

/* HTML-код страницы, сообщающей о том, что
при открытии файла /etc/issue произошла ошибка. */
static char* error_page =
"<html>\n"
" <body>\n"
" <p>Error: Could not open /etc/issue.</p>\n"

```

```

" </body>\n"
"</html>\n";

/* Сообщение об ошибке. */
static char* error_message =
"Error reading /etc/issue.";

void module_generate(int fd) {
int input_fd;
struct stat file_info;
int rval;

/* Открытие файла /etc/issue */
input_fd = open("/etc/issue", O_RDONLY);
if (input_fd == -1)
system_error("open");
/* Получение информации о файле. */
rval = fstat(input_fd, &file_info);
if (rval == -1)
/* не удалось открыть файл или прочитать данные из него. */
write(fd, error_page, strlen(error_page));
else {
int rval;
off_t offset = 0;

/* Запись начала страницы */
write(fd, page_start, strlen(page_start));
/* Копирование данных из файла /etc/issue
в клиентский сокет. */
rval = sendfile(fd, input_fd, &offset, file_info.st_size);
if (rval == -1)
/* При отправке файла /etc/issue произошла ошибка.
Выводим соответствующее сообщение. */
write(fd, error_message, strlen(error_message));
/* Конец страницы. */
write(fd, page_end, strlen(page_end));
}
close(input_fd);
}

```

Сначала модуль пытается открыть файл /etc/issue. Если это не удалось, клиенту возвращается сообщение об ошибке. В противном случае посылается начальный код HTML-страницы, содержащийся в переменной page\_start, затем содержимое файла /etc/issue (это делается с помощью функции sendfile(), о которой рассказывалось в разделе 8.12. "Функция sendfile(): быстрая передача данных") и, наконец конечный код HTML-страницы, содержащийся в переменной page\_end.

Этот модуль можно легко настроить на отправку любого другого файла. Если файл содержит HTML-страницу, переменные page\_start и page\_end будут не нужны.

### 11.3.3. Отображение объема свободного дискового пространства

Модуль diskfree.so (исходный текст приведен в листинге 11.8) генерирует страницу с информацией о свободном дисковом пространстве в файловых системах, смонтированных на серверном компьютере. Эта информация берется из выходных данных команды `df -h`. Как и в модуле issue.so, выходные данные заключаются в тэги `<pre></pre>`.



## Листинг 11.8. (diskfree.c) Серверный модуль, отображающий информацию о свободном дисковом пространстве

```
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include "server.h"

/* HTML-код начала генерируемой страницы. */
static char* page_start =
"<html>\n"
" <body>\n"
" <pre>\n";

/* HTML-код конца генерируемой страницы. */
static char* page_end =
" </pre>\n"
" </body>\n"
"</html>\n";

void module_generate(int fd) {
    pid_t child_pid;
    int rval;

    /* Запись начала страницы. */
    write(fd, page_start, strlen(page_start));
    /* Создание дочернего процесса. */
    child_pid = fork();
    if (child_pid == 0) {
        /* Это дочерний процесс. */
        /* Подготовка списка аргументов команды df. */
        char* argv[] = { "/bin/df", "-h", NULL };

        /* Дублирование потоков stdout и stderr для записи данных
        в клиентский сокет. */
        rval = dup2(fd, STDOUT_FILENO);
        if (rval == -1)
            system_error("dup2");
        rval = dup2(fd, STDERR_FILENO);
        if (rval == -1)
            system_error("dup2");
        /* Запуск команды df, отображающей объем свободного
        пространства в смонтированных файловых системах. */
        execv(argv[0], argv);
        /* Функция execv() возвращает управление в программу только
        при возникновении ошибки. */
        system_error("execv");
    } else if (child_pid > 0) {
        /* Это родительский процесс, ожидаем завершения дочернего
        процесса. */
        rval = waitpid(child_pid, NULL, 0);
        if (rval == -1)
            system_error("waitpid");
    } else
        /* Вызов функции fork() завершился неудачей. */
        system_error("fork");
}
```

```

/* запись конца страницы. */
write(fd, page_end, strlen(page_end));
}

```

В то время как модуль `issue.so` посылает содержимое файла с помощью функции `sendfile()`, данный модуль должен вызвать внешнюю команду и перенаправить результаты ее работы клиенту. Для этого модуль придерживается такой последовательности действий.

1. Сначала с помощью функции `fork()` создается дочерний процесс (см. раздел 3.2.2. "Функции `fork()` и `exec()`").

2. Дочерний процесс копирует дескриптор сокета в дескрипторы `STDOUT_FILENO` и `STDERR_FILENO`, соответствующие стандартным потокам вывода и ошибок (см. раздел 2.1.4, "Стандартный ввод-вывод"). Это копирование осуществляется с помощью системного вызова `dup2()` (см. раздел 5.4 3. "Перенаправление стандартных потоков ввода, вывода и ошибок"). Все последующие данные, записываемые в эти потоки в рамках дочернего процесса, будут направляться в сокет.

3. Дочерний процесс с помощью функции `execv()` вызывает команду `df -h`.

4. Родительский процесс дожидается завершения дочернего процесса, вызывая функцию `waitpid()` (см. раздел 5.4 2. "Системные вызовы `wait()`").

Этот модуль можно легко настроить на вызов другой системной команды.

#### 11.3.4. Статистика выполняющихся процессов

Модуль `processes.so` (исходный текст приведен в листинге 11.9) сложнее остальных модулей. Он генерирует страницу, в которой содержится таблица процессов, выполняющихся в данный момент на сервере. Каждому процессу отводится в таблице одна строка. В этой строке указан идентификатор процесса, имя исполняемого файла, имена владельца и группы, которым принадлежит процесс, а также размер резидентной части процесса.

#### *Листинг 11.9. (processes.c) Серверный модуль, отображающий таблицу процессов*

```

#include <assert.h>
#include <dirent.h>
#include <fcntl.h>
#include <grp.h>
#include <pwd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
#include "server.h"

```

```

/* Эта функция записывает в аргументы UID и GID
идентификаторы пользователя и группы, которым
принадлежит процесс с указанным идентификатором,
в случае успешного завершения возвращается нуль,
иначе -- ненулевое значение. */
static int get_uid_gid(pid_t pid, uid_t* uid, gid_t* gid) {
char dir_name[64];
struct stat dir_info;

```

```

int rval;

/* Формирование имени каталога процесса
в файловой системе /proc. */
snprintf(dir_name, sizeof(dir_name), "/proc/%d", (int)pid);

/* Получение информации о каталоге. */
rval = stat(dir_name, &dir_info);
if (rval != 0)
/* Каталог не найден. Возможно, процесс больше
не существует. */
return 1;
/* Убеждаемся в том, что это действительно каталог. */
assert(S_ISDIR(dir_info.st_mode));

/* Определяем интересующие нас идентификаторы. */
*uid = dir_info.st_uid;
*gid = dir_info.st_gid;
return 0;
}

/* Эта функция находит имя пользователя,
соответствующее заданному идентификатору.
Возвращаемый буфер должен быть удален
в вызывающей функции. */
static char* get_user_name(uid_t uid) {
struct passwd* entry;
entry = getpwuid(uid);
if (entry == NULL)
system_error("getpwuid");
return xstrdup(entry->pw_name);
}

/* Эта функция находит имя группы, соответствующее
заданному идентификатору, возвращаемый буфер
должен быть удален в вызывающей функции. */
static char* get_group_name(gid_t gid) {
struct group* entry;
entry = getgrgid(gid);
if (entry == NULL)
system_error("getgrgid");
return xstrdup(entry->gr_name);
}

/* Эта функция находит имя программы, которую выполняет
процесс с заданным идентификатором. Возвращаемый буфер
должен быть удален в вызывающей функции. */
static char* get_program_name(pid_t pid) {
char file_name[64];
char status_info[256];
int fd;
int rval;
char* open_paren;
char* close_paren;
char* result;

/* Генерируем имя файла "stat", находящегося в каталоге
данного процесса в файловой системе /proc,

```

```

и открываем этот файл. */
snprintf(file_name, sizeof(file_name), "/proc/%d/stat",
(int)pid);
fd = open(file_name, O_RDONLY);
if (fd == 1)
/* Файл не удалось открыть. Возможно, процесс
больше не существует. */
return NULL;
/* Чтение содержимого файла
rval = read(fd, status_info, sizeof(status_info) 1);
close(fd);
if (rval <= 0)
/* По какой-то причине файл не удалось прочитать, завершаем
работу. */
return NULL;
/* Завершаем прочитанный текст нулевым символом. */
status_info[rval] = '\0';

/* Имя программы -- это второй элемент файла, заключенный в
круглые скобки. Находим местоположение скобок. */
open_paren = strchr(status_info, '(');
close_paren = strchr(status_info, ')');
if (open_paren == NULL ||
close_paren == NULL || close_paren < open_paren)
/* Не удалось найти скобки, завершаем работу. */
return NULL;
/* Выделение памяти для результирующей строки */
result = (char*)xmalloc(close_paren - open_paren);
/* Копирование имени программы в буфер. */
strncpy(result, open_paren + 1, close_paren - open_paren 1);
/* Функция strncpy() не завершает строку нулевым символом,
приходится это делать самостоятельно. */
result[close_paren - open_paren - 1] = '\0';
/* Конец работы. */
return result;
}

/* Эта функция определяет размер (в килобайтах) резидентной
части процесса с заданным идентификатором.
В случае ошибки возвращается -1. */
static int get_rss(pid_t pid) {
char file_name[64];
int fd;
char mem_info[128];
int rval;
int rss;

/* Генерируем имя файла "statm", находящегося в каталоге
данного процесса в файловой системе proc. */
snprintf(file_name, sizeof(file_name), "/proc/%d/statm",
(int)pid);
/* Открытие файла. */
fd = open(file_name, O_RDONLY);
if (fd == -1)
/* Файл не удалось открыть. Возможно, процесс больше не
существует. */
return -1;
/* Чтение содержимого файла. */
rval = read(fd, mem_info, sizeof(mem_info) 1);

```

```

close(fd);
if (rval <= 0)
/* Файл не удалось прочитать, завершаем работу. */
return -1;
/* Завершаем прочитанный текст нулевым символом. */
mem_info[rval] = '\0';
/* Определяем размер резидентной части процесса. Это второй
элемент файла. */
rval = sscanf(mem_info, "%*d %d", &rss);
if (rval != 1)
/* Содержимое файла statm отформатировано непонятным
образом. */
return -1;

/* Значения в файле statm приведены в единицах, кратных размеру
системной страницы. Преобразуем в килобайты. */
return rss * getpagesize() / 1024;
}

/* Эта функция генерирует строку таблицы для процесса
с заданным идентификатором. Возвращаемый буфер должен
удаляться в вызывающей функции, в случае ошибки
возвращается NULL. */
static char* format_process_info(pid_t pid) {
int rval;
uid_t uid;
gid_t gid;
char* user_name;
char* group_name;
int rss;
char* program_name;
size_t result_length;
char* result;

/* Определяем идентификаторы пользователя и группы, которым
принадлежит процесс. */
rval = get_uid_gid(pid, &uid, &gid);
if (rval != 0)
return NULL;
/* Определяем размер резидентной части процесса. */
rss = get_rss(pid);
if (rss == -1)
return NULL;
/* Определяем имя исполняемого файла процесса. */
program_name = get_program_name(pid);
if (program_name == NULL)
return NULL;
/* Преобразуем идентификаторы пользователя и группы в имена. */
user_name = get_user_name(uid);
group_name = get_group_name(gid);
/* Вычисляем длину строки, в которую будет помещен результат,
и выделяем для нее буфер. */
result_length =
strlen(program_name) + strlen(user_name) +
strlen(group_name) + 128;
result = (char*)xmalloc(result_length);
/* Форматирование результата. */
snprintf(result, result_length,
"<tr><td align=\" right\">%d</td><td><tt>%s</tt></td><td>%s</td>"

```

```

"<td>%s</td><td align= \"right\">%d</td></tr>\n",
(int)pid, program_name, user_name, group_name, rss);
/* Очистка памяти. */
free(program_name);
free(user_name);
free(group_name);
/* Конец работы. */
return result;
}

/* HTML-код начала страницы, содержащей таблицу процессов. */
static char* page_start =
"<html>\n"
" <body>\n"
" <table cellpadding=\"4\" cellspacing=\"0\" border=\"1\">\n"
" <thead>\n"
" <tr>\n"
" <th>PID</th>\n"
" <th>Program</th>\n"
" <th>User</th>\n"
" <th>Group</th>\n"
" <th>RSS&nbsp;(KB)</th>\n"
" </tr>\n"
" </thead>\n"
" <tbody>\n";

/* HTML-код конца страницы, содержащей таблицу процессов. */
static char* page_end =
" </tbody>\n"
" </table>\n"
" </body>\n"
"</html>\n";
void module_generate(int fd) {
size_t i;
DIR* proc_listing;

/* Создание массива iovec. В этот массив помещается выходная
информации, причем массив может увеличиваться динамически. */

/* Число используемых элементов массива */
size_t vec_length = 0;
/* выделенный размер массива */
size_t vec_size = 16;
/* Массив элементов iovec. */
struct iovec* vec =
(struct iovec*)xmalloc(vec_size *
sizeof(struct iovec));

/* Сначала в массив записывается HTML-код начала страницы. */
vec[vec_length].iov_base = page_start;
vec[vec_length].iov_len = strlen(page_start);
++vec_length;

/* Получаем список каталогов в файловой системе /proc. */
proc_listing = opendir("/proc");
if (proc_listing == NULL)
system_error("opendir");

```

```

/* Просматриваем список каталогов. */
while (1) {
    struct dirent* proc_entry;
    const char* name;
    pid_t pid;
    char* process_info;

    /* Переходим к очередному элементу списка. */
    proc_entry = readdir(proc_listing);
    if (proc_entry == NULL)
        /* Достигнут конец списка. */
        break;

    /* Если имя каталога не состоит из одних цифр, то это не
    каталог процесса; пропускаем его. */
    name = proc_entry->d_name;
    if (strspn(name, "0123456789") != strlen(name))
        continue;
    /* Именем каталога является идентификатор процесса. */
    pid = (pid_t)atoi(name);

    /* генерируем HTML-код для строки таблицы, содержащей
    описание данного процесса. */
    process_info = format_process_info(pid);
    if (process_info == NULL)
        /* Произошла какая-то ошибка. Возможно, процесс уже
        завершился. Создаем строку-заглушку. */
        process_info =
            "<tr><td colspan=\\\"5\\\">ERROR</td></tr>";
    /* Убеждаемся в том, что в массиве iovec достаточно места
    для записи буфера (один элемент будет добавлен в массив
    по окончании обработки списка процессов). Если места
    не хватает, удваиваем размер массива. */
    if (vec_length == vec_size - 1) {
        vec_size *= 2;
        vec = xrealloc(vec, vec_size - sizeof(struct iovec));
    }
    /* Сохраняем в массиве информацию о процессе. */
    vec[vec_length].iov_base = process_info;
    vec[vec_length].iov_len = strlen(process_info);
    ++vec_length;
}

/* Конец обработки списка каталогов */
closedir(proc_listing);

/* Добавляем HTML-код конца страницы. */
vec[vec_length].iov_base = page_end;
vec[vec_length].iov_len = strlen(page_end);
++vec_length;

/* Передаем всю страницу клиенту. */
writev(fd, vec, vec_length);
/* Удаляем выделенные буферы. Первый и последний буферы
являются статическими, поэтому не должны удаляться. */
for (i = 1; i < vec_length - 1; ++i)
    free(vec[i].iov_base);
/* Удаляем массив iovec. */

```

```
free(vec);  
}
```

Задача сбора информации о процессах и представления ее в виде HTML-таблицы разбивается на ряд более простых операций.

■ Функция `get_uid_gid()` возвращает идентификатор пользователя и группы, которым принадлежит процесс. Для этого вызывается функция `stat()` (описана в приложении Б, "Низкоуровневый ввод-вывод"), берущая информацию из каталога процесса в файловой системе `/proc`.

■ Функция `get_user_name()` возвращает имя пользователя, соответствующее заданному идентификатору. Она просто вызывает библиотечную функцию `getpwuid()`, которая обращается к файлу `/etc/passwd` и возвращает копию строки из него. Функция `get_group_name()` находит имя группы по заданному идентификатору. Она вызывает функцию `getgrgid()`.

■ Функция `get_program_name()` возвращает имя программы, соответствующей заданному процессу. Эта информация извлекается из файла `stat`, находящегося в каталоге процесса в файловой системе `/proc` (см. раздел 7.2, "Каталоги процессов"). Мы поступаем так, а не проверяем символические ссылки `exe` или `cmdline`, поскольку последние недоступны, если серверный процесс не принадлежит тому же пользователю, что и проверяемый процесс.

■ Функция `get_rss()` определяет объем резидентной части процесса. Эта информация содержится во втором элементе файла `statm` (см. раздел 7.2.6, "Статистика использования процессом памяти"), находящегося в каталоге процесса в файловой системе `/proc`.

■ Функция `format_process_info()` генерирует набор HTML-тэгов для строки таблицы, представляющей заданный процесс. Здесь вызываются все вышеперечисленные функции.

■ Функция `module_generate()` генерирует HTML-страницу с таблицей процессов. Выводная информация включает начальный HTML-блок (переменная `page_start`), строки с информацией о процессах (создаются функцией `format_process_info()`) и конечный HTML-блок (переменная `page_end`).

Функция `module_generate()` определяет идентификаторы процессов, проверяя содержимое файловой системы `/proc`. Для получения и анализа списка каталогов вызываются функции `opendir()` и `readdir()` (описаны в приложении Б, "Низкоуровневый ввод-вывод"). Из данного списка отбираются элементы, имена которых состоят из одних цифр: это каталоги процессов.

Поскольку в таблице может содержаться достаточно большое число строк, последовательная запись их в сокет с помощью функции `write()` приведет к ненужному повышению трафика. Для оптимизации числа передаваемых пакетов используется функция `writenv()` (описана в приложении Б, "Низкоуровневый ввод-вывод"). Для нее создается массив `vec`, состоящий из элементов типа `iovec`. Так как число процессов не известно заранее приходится начинать с маленького массива и увеличивать его по мере необходимости. В переменной `vec_length` содержится число используемых элементов массива `vec`, а в переменной `vec_size` число выделенных элементов. Когда эти переменные становятся почти равными друг другу, размер массива удваивается с помощью функции `xrealloc()`. По окончании работы с массивом удаляются все адресуемые в нем строки, а также сам массив.

## 11.4. Работа с сервером

Если бы демонстрационную программу нужно было распространять в виде исходных текстов, сопровождать и переносить на другие платформы, потребовалось бы упаковать ее с помощью GNU-утилит `Automake` и `Autosconf`. Но их рассмотрение выходит за рамки нашей книги.



### 11.4.1. Файл Makefile

Вместо утилиты Autoconf мы воспользуемся простым файлом Makefile, совместимым с GNU-утилитой Make.<sup>[39]</sup> Этот файл упростит компиляцию и компоновку сервера и его модулей. Содержимое файла показано в листинге 11.10.

*Листинг 11.10. (Makefile) Файл конфигурации сервера*

```
### Конфигурация. #####

# Стандартные параметры компилятора языка C.
CFLAGS = -Wall -g
# Исходные файлы сервера.
SOURCES = server.c module.c common.c main.c
# Соответствующие объектные файлы.
OBJECTS = $(SOURCES:.c=.o)
# Совместно используемые библиотеки серверных модулей.
MODULES = diskfree.so issue.so processes.so time.so

### Правила. #####

# Служебный целевой модуль.
.PHONY: all clean

# Стандартный целевой модуль: компиляция всех файлов.
all: server $(MODULES)

# Удаление всех компонентов.
clean:
rm -f $(OBJECTS) $(MODULES) server

# Главная серверная программа, должна компоноваться с флагами
# -Wl,-export-dynamic, чтобы динамически загружаемые модули могли
# находить в программе символические константы. Подключается также
# библиотека libdl, в которой находятся функции динамической
# загрузки.
server: $(OBJECTS)
$(CC) $(CFLAGS) Wl,-export-dynamic -o $@ $^ -ldl

# Все объектные файлы сервера зависят от файла server.h.
# Используем стандартное правило создания объектных файлов из
# исходных файлов.
$(OBJECTS): server.h

# Правило создания совместно используемых библиотек из
# соответствующих исходных файлов, компилируем с флагом -fPIC и
# генерируем совместно используемый объектный файл.
$(MODULES): \
%.so: %.c server.h
$(CC) $(CFLAGS) -fPIC -shared -o $@ $<

В файле Makefile есть следующие целевые модули.
■Модуль all (используется по умолчанию при вызове файла Makefile без аргументов, так
как стоит первым) содержит исполняемый файл server и все серверные модули. Последние
```

перечислены в переменной `MODULES`.

- Модуль `clean` предназначен для удаления всех скомпилированных компонентов.

- Модуль `server` подключает к проекту исполняемый файл сервера. Компилируются и компонуются исходные файлы, перечисленные в переменной `SOURCES`.

- Последнее правило представляет собой шаблон компиляции совместно используемых файлов серверных модулей.

Обратите внимание на то, что исходные файлы серверных модулей компилируются с флагом `-fPIC`, так как они включаются в совместно используемые библиотеки (см. раздел 2.3.2, "Совместно используемые библиотеки").

Исполняемый файл `server` компонуется с флагом `-Wl,-export-dynamic`. Благодаря этому файл будет экспортировать свои символические константы, что позволит динамически загружаемым модулям ссылаться на функции, находящиеся в файле `common.c`.

## 11.4.2. Создание сервера

Построить исполняемый файл несложно. Перейдите в каталог, содержащий исходные файлы, и вызовите команду `make`:

```
% make
cc -Wall -g -c -o server.o server.c
cc -Wall -g -c -o module.o module.c
cc -Wall -g -c -o common.o common.c
cc -Wall -g -c -o main.o main.c
cc -Wall -g -Wl,-export-dynamic -o server server.o module.o
common.o main.o -ldl
cc -Wall -g -fPIC -shared -o diskfree.so diskfree.c
cc -Wall -g -fPIC -shared -o issue.so issue.c
cc -Wall -g -fPIC -shared -o processes.so processes.c
cc -Wall -g -fPIC -shared -o time.so time.c
```

В результате будут созданы программа `server` и совместно используемые библиотеки серверных модулей:

```
% ls -l server *.so
-rwxr-xr-x 1 samuel samuel 25769 Mar 11 01:15 diskfree.so
-rwxr-xr-x 1 samuel samuel 31184 Mar 11 01:15 issue.so
-rwxr-xr-x 1 samuel samuel 41579 Mar 11 01:15 processes.so
-rwxr-xr-x 1 samuel samuel 71758 Mar 11 01:15 server
-rwxr-xr-x 1 samuel samuel 13980 Mar 11 01:15 time.so
```

## 11.4.3. Запуск сервера

Для запуска сервера достаточно ввести в командной строке имя `server`. Если не задать номер порта с помощью опции `--port (-p)`, ОС Linux самостоятельно выберет порт. При указании опции `--verbose (-v)` сервер покажет, какой порт ему назначен.

Если не назначить серверу адрес с помощью опции `--address (-a)`, сервер будет принимать запросы по всем имеющимся адресам. Для подключенного к сети компьютера это означает, что любой пользователь сети, зная номер порта сервера и имя страницы, сможет обратиться к серверу. Из соображений безопасности рекомендуем указывать адрес `localhost`, пока вы не убедитесь в правильной работе сервера. В этом случае сервер будет связан с локальным сетевым устройством (обозначается как `lo`) и к нему смогут обращаться только программы, работающие на том же самом компьютере.

```
% ./server --address localhost --port 4000
```

Теперь сервер работает. Откройте окно браузера и попытайтесь обратиться к серверу по номеру порта. Запросите страницу, имя которой совпадает с именем модуля. Вот как, например, вызывается модуль `diskfree.so`:

```
http://localhost:4000/diskfree
```

Вместо 4000 можно указать любой другой номер порта, который был выбран. Чтобы завершить работу сервера, нажмите `<Ctrl+C>`.

Если сервер принимает запросы по сети, к нему можно подключиться с помощью браузера, работающего на другом компьютере, например:

```
http://host.domain.com:4000/diskfree
```

Если задать опцию `--verbose (-v)`, сервер при запуске отобразит свою конфигурационную информацию, а затем будет показывать IP-адрес каждого подключающегося к нему клиента. Если подключаться через интерфейс `localhost`, клиентский адрес всегда будет равен `127.0.0.1`.

С помощью опции `--module-dir (-m)` можно указать другой каталог размещения серверных модулей. По умолчанию они находятся там же, где и программа `server`.

Те, кто забыли или не знают синтаксис опций командной строки, могут вызвать программу `server` с опцией `--help (-h)`:

```
% ./server --help
Usage: ./server [ options ]
-a, --address ADDR Bind to local address (by default, bind
to all local addresses).
-h, --help Print this information.
-m, --module-dir DIR Load modules from specified directory
(by default, use executable directory).
-p, --port PORT Bind to specified port.
-v, --verbose Print verbose messages.
```

## 11.5. Вместо эпилога

Планируя распространять программу в Internet, не забудьте написать для нее документацию. Многие люди не осознают, что создать качественную документацию так же трудно и долго, как и написать хорошую программу. Правда, вопрос подготовки документации тема отдельной книги, поэтому мы лишь укажем, где можно получить информацию по данной теме.

Вероятнее всего, для программы потребуется создать `man`-страницу. Это первое место, куда пользователи обращаются за информацией о программе. Страницы интерактивной документации форматируются с помощью классической программы `troff`. Чтобы узнать формат `troff`-файлов, введите такую команду:

```
% man troff
```

Чтобы узнать, как ОС Linux ищет `man`-страницы, просмотрите справочную информацию о самой команде `man`:

```
% man man
```

Можно также подготовить документацию в формате GNU-системы `Info`. Для получения информации об этой системе выполните команду

```
% info info
```

Для многих Linux-программ имеется также документация в формате простого текста и HTML.

*Удачного программирования!*

Часть III

Приложения

# Приложение А

## Вспомогательные инструменты разработки

Разработка безошибочных и быстрых Linux-программ требует не только понимания операционной системы Linux и ее системных вызовов. В этом приложении будут рассмотрены методики, позволяющие находить ошибки периода выполнения (например, неправильное использование оперативной памяти) и определять, какие компоненты программы требуют наибольших вычислительных ресурсов. Анализ программного кода дает лишь часть этой информации; чтобы получить остальную часть, необходимо запустить программу и воспользоваться описанными ниже инструментами.

### А.1. Статический анализ программы

Некоторые программные ошибки можно выявить, воспользовавшись средствами статического анализа исходных текстов. Если вызвать компилятор gcc с флагами `-Wall` и `-pedantic`, он выдаст предупреждения о рискованных и потенциально ошибочных программных конструкциях. Исправив эти конструкции, вы снизите вероятность появления в программе скрытых ошибок, а также упростите компиляцию программы в других вариантах Linux или даже в других операционных системах.

С помощью различных флагов командной строки можно заставить компилятор gcc выдавать предупреждения о множестве спорных программных конструкций. Большинство проверок включается флагом `-Wall`. Например, компилятор будет сообщать о комментариях, начинающемся в другом комментарии, о неправильном типе возвращаемого значения в функции `main()`, о функциях, в которых пропущена инструкция `return`, и т.д. При наличии флага `-pedantic` компилятор будет выдавать предупреждения о несоответствии стандарту ANSI. В частности, будет сообщаться о наличии функции `asm()` и других GNU-расширений языка. В документации к компилятору не рекомендуется использовать этот флаг. Мы же советуем избегать большинства GNU-расширений, так как они имеют тенденцию меняться со временем и плохо поддаются оптимизации.

Попробуем скомпилировать программу "Hello, World", представленную в листинге А.1.

#### *Листинг А.1. (hello.c) Простейшая программа*

```
main() {  
    printf("Hello, world.\n");  
}
```

Будучи вызванным без флагов, компилятор не выдаст никаких предупреждений, хотя программа не соответствует стандарту ANSI. Если же включить флаги `-Wall` и `-pedantic`, то обнаружатся три спорные конструкции:

```
% gcc -Wall -pedantic hello.c  
hello.c:2: warning: return type defaults to 'int'  
hello.c: In function 'main':  
hello.c:3: warning: implicit declaration of function 'printf'  
hello.c:4: warning: control reaches end of non-void function
```

Компилятор сообщает о следующих проблемах:

- не указан тип возвращаемого значения функции `main()`;

- функция `printf()` не объявлена, так как файл `<stdio.h>` не включен в программу;
- функция `main()`, которая неявно возвращает значение типа `int`, не содержит инструкцию `return`.

Анализ исходных текстов программы не позволяет выявить все возможные ошибки и неэффективные конструкции. В следующем разделе описываются четыре средства поиска ошибок при работе с динамической памятью. В конце приложения будет рассказано о том, как анализировать время работы программы с помощью утилиты-профайлера `gprof`.

## **А.2. Поиск ошибок в динамической памяти**

При написании программы зачастую неизвестно, сколько памяти потребуется ей во время выполнения. Например, строка, читаемая из файла, может иметь любую длину. Работа с динамической памятью осуществляется посредством функций `malloc()`, `free()` и их вариантов. Следует придерживаться таких правил:

- число запросов на выделение памяти (вызовов функции `malloc()`) должно в точности совпадать с числом запросов на освобождение памяти (вызовов функции `free()`);
- операции чтения и записи динамической памяти должны выполняться в рамках выделенной области, не выходя за ее пределы;
- к выделенной области нельзя обращаться после того, как она была освобождена.

Выделение и освобождение динамической памяти происходят на этапе выполнения программы, поэтому статический анализ исходных текстов редко позволяет выявить недочеты. Утилиты проверки памяти сначала загружают программу, а затем определяют, нарушаются ли перечисленные выше правила. Выявляются следующие ошибки:

- чтение памяти до того, как она была выделена;
- запись в память до того, как она была выделена;
- чтение данных по адресу, предшествующему началу выделенной области;
- запись данных по адресу, предшествующему началу выделенной области;
- чтение данных по адресу, стоящее после выделенной области;
- запись данных по адресу, стоящему после выделенной области;
- чтение памяти после того, как она была освобождена;
- запись в память после того, как она была освобождена;
- неудачная попытка освободить выделенную память;
- попытка повторно освободить ту же самую область памяти;
- попытка освободить память, которая не была выделена.

Полезно также предупреждать о выделениях областей размером 0 байтов, так как это обычно свидетельствует об ошибке программиста.

В табл. А.1 описаны возможности четырех диагностических средств. К сожалению, ни одно из них не выявляет все возможные ошибки. Кроме того, ни одно средство не позволяет обнаруживать попытки чтения или записи памяти до того, как она была выделена, хотя такая попытка наверняка приведет к нарушению сегментации. Обнаруживаются те ошибки, которые действительно происходят в процессе работы программы. Если передать программе такие входные данные, что выделять память не понадобится, ошибки обращения к памяти не будут найдены. Для максимально тщательной проверки программы рекомендуется передавать ей самые разные входные данные, чтобы протестировать все возможные пути ее выполнения. Желательно также тестировать программу всеми имеющимися средствами.

**Таблица А.1. Возможности средств проверки динамической памяти (X обнаружение,**

О обнаружение в некоторых случаях):

Ошибка	Проверка функции malloc()	Утилита mtrace	Библиотека csmalloc	Библиотека Electric Fence
Чтение памяти до того, как она была выделена				
Запись в память до того, как она была выделена				
Чтение данных по адресу, предшествующему началу выделенной области				X
Запись данных по адресу, предшествующему началу выделенной области	O		O	X
Чтение данных по адресу, стоящему после выделенной области				X
Запись данных по адресу, стоящему после выделенной области			X	X
Чтение памяти после того, как она была освобождена				X
Запись в память после того, как она была освобождена				X
Неудачная попытка освободить выделенную память		X	X	
Попытка повторно освободить ту же самую область памяти	X		X	
Попытка освободить память, которая не была выделена		X	X	
Выделение памяти нулевого размера			X	X

A.2.1. Программа для тестирования динамической памяти

Программа malloc-use, приведенная в листинге A.2, позволяет тестировать операции выделения, освобождения и обращения к памяти. Единственный аргумент командной строки задает максимальное число выделяемых буферов. Например, по команде malloc-use 12 будет создан массив A из двенадцати пустых указателей. Программа принимает пять разных команд.

- Если ввести a i b, для элемента массива A[i] будет выделено b байтов. Индекс i должен быть неотрицательным числом, меньшим, чем аргумент командной строки. Число байтов также должно быть неотрицательным.
- Если ввести d i, будет удален буфер A[i].
- Если ввести r i p, из буфера A[i] будет прочитан p-й символ (A[i][p]). Значение p должно быть целым.
- Если ввести w i p, в позицию p буфера A[i] будет записан символ.
- Для завершения работы программы введите q.

Прежде чем привести исходный текст программы, опишем, как работать с ней.

### A.2.2. Проверка функции malloc()

Функции выделения и освобождения памяти, имеющиеся в GNU-библиотеке языка C, способны обнаруживать факт записи в память до начала выделенной области, а также попытку освободить одну и ту же область дважды. Если задать переменную среды `MALLOC_CHECK_` равной 2, программа `malloc-use` аварийно завершит работу в случае выявления такого рода ошибки. Подобное изменение поведения не требует перекомпиляции программы.

Вот что произойдет, если записать символ перед началом массива;

```
% export MALLOC_CHECK_=2
% ./malloc-use 12
Please enter a command: a 0 10
Please enter a command: w 0 -1
Please enter a command: d 0
Aborted (core dumped)
```

Команда `export` включила проверку функции `malloc()`, а значение 2 заставило программу завершиться сразу после обнаружения ошибки.

Проверка функции `malloc()` очень полезна, потому что программу не нужно перекомпилировать, однако возможности этой проверки весьма ограничены. В основном определяется, не были ли повреждены выделенные структуры данных. Таким образом, сразу же обнаруживаются попытки повторно удалить ту же самую область. Кроме того, выявляется факт записи данных непосредственно перед началом выделенного блока, поскольку его размер хранится именно там. К сожалению, проверка выполняется только тогда, когда программа вызывает функцию `malloc()` или `free()`, а не когда происходит обращение к памяти. То есть до обнаружения ошибки может произойти множество неправильных операций чтения и записи. В частности, в предыдущем примере ошибка записи была выявлена лишь при попытке освободить выделенную область.

### A.2.3. Поиск потерянных блоков памяти с помощью утилиты mtrace

Утилита `mtrace` позволяет выявить наиболее распространенную ошибку при работе с динамической памятью: несоответствие числа операций выделения и освобождения памяти. Алгоритм применения утилиты таков.

1. Включите в программу файл `<mcheck.h>` и разместите в самом начале программы вызов функции `mtrace()`. Эта функция активизирует трассировку операций выделения и освобождения памяти.

2. Задайте имя файла, в котором будет сохраняться трассировочная информация. Это делается следующим образом:

```
% export MALLOC_TRACE=memory.log
```

3. Запустите программу. Все операции выделения и освобождения памяти будут зарегистрированы в журнальном файле.

4. Вызовите утилиту `mtrace`, которая проверит, совпадает ли число выделенных блоков памяти с числом освобожденных блоков.

```
% mtrace my_program $MALLOC_TRACE
```

Сообщения, выдаваемые утилитой `mtrace`, достаточно понятны. Например, в случае программы `malloc-use` будет получена такая информация:

```
- 000000000000 Free 3 was never alloc'd malloc-use.c:39
```



```
Memory not freed:
```

```
-----
```

```
Address Size Caller
```

```
0x08049d48 0xc at malloc-use.c:30
```

Эти сообщения говорят о том, что в строке 39 файла `malloc-use.c` делается попытка освободить память, которая никогда не была выделена, а память, выделенная в строке 30, так и не была освобождена.

Функция `malloc()` заставляет программу фиксировать все операции выделения и освобождения памяти в файле, указанном в переменной среды `MALLOC_TRACE`. Чтобы данные были записаны в файл, программа должна завершиться нормальным образом. Утилита `mtrace` анализирует этот файл и находит в нем непарные записи.

## A.2.4. Библиотека `ccmalloc`

Библиотека `ccmalloc` замещает функции `malloc()` и `free()` кодом трассировки. Если программа завершается успешно, создается отчет о потерянных блоках памяти и прочих ошибках. Библиотеку `ccmalloc` написал Армин Бир (Armin Biere).

Код библиотеки требуется загрузить и установить самостоятельно. Дистрибутив можно найти по адресу <http://www.inf.ethz.ch/personal/biere/projects/ccmalloc>. Распакуйте дистрибутив и запустите сценарий `configure`. Далее выполните команды `make` и `make install`, скопируйте файл `ccmalloc.cfg` в каталог, из которого будет запускаться проверяемая программа, и переименуйте копию в `.ccmalloc`.

К объектным файлам программы необходимо подключить библиотеку `ccmalloc` и библиотеку функций динамической компоновки. Вот как это делается:

```
% gcc -g -Wall -pedantic malloc-use.o -o ccmalloc-use -lccmalloc -ldl
```

Запустите программу, чтобы получить отчет. Например, если попросить программу `malloc-use` выделить память и забыть ее освободить, будут выданы следующие результаты:

```
% ./ccmalloc-use 12
file-name=a.out does not contain valid symbols
trying to find executable in current directory ...
using symbols from 'ccmalloc-use'
(to speed up this search specify 'file ccmalloc-use'
in the startup file '.ccmalloc')
Please enter a command: a 0 12
Please enter a command: q

.------.
| ccmalloc report |
=====
| total # of | allocated | deallocated | garbage |
+-----+-----+-----+-----+
| bytes | 60 | 48 | 12 |
+-----+-----+-----+-----+
| allocations | 2 | 1 | 1 |
+-----+-----+-----+-----+
| number of checks: 1 |
| number of counts: 3 |
| retrieving function names for addresses ... done. |
| reading file info from gdb ... done. |
| sorting by number of not reclaimed bytes ... done. |
| number of call chains: 1 |
| number of ignored call chains: 0 |
| number of reported call chains: 1 |
```

```
| number of internal call chains: 1 |
| number of library call chains: 0 |
=====
|
*100.0% = 12 Bytes of garbage allocated in 1 allocation
| |
| | 0x400389cb in <??>
| | 0x08045198 in <main>
| | at malloc-use.c:89
| |
| | 0x06048fdc in <allocate>
| | at malloc-use.c:30
| |
| | -----> 0x08049647 in <malloc>
| | at src/wrapper.c:284
| -----
```

В последних нескольких строках показана цепочка вызовов функций, в которых была выделена, но не освобождена память.

Если необходимо, чтобы библиотека `ccmalloc` отслеживала операции записи в память вне выделенной области, придется модифицировать файл `.ccmalloc`, расположенный в текущем каталоге. Этот файл проверяется при запуске программы.

## A.2.5. Библиотека Electric Fence

Библиотека Electric Fence, написанная Брюсом Перензом (Bruce Perens), останавливает выполнение программы в той строке, где происходит обращение к памяти за пределами выделенной области. Это единственное средство, позволяющее выявить неправильные операции чтения. Библиотека входит в большинство дистрибутивов Linux, а ее исходные коды можно найти по адресу <http://www.perens.com/FreeSoftware>.

Как и в случае библиотеки `ccmalloc`, к объектным файлам программы необходимо подключить код библиотеки Electric Fence:

```
% gcc -g -wall -pedantic malloc-use.o -o emalloc-use -lefence
```

После запуска программы библиотека проверяет правильность обращений к выделенной памяти. В случае нарушения возникает ошибка сегментации:

```
% ./emalloc-use 12
Electric Fence 2.0.5 Copyright (C) 1987-1998 Bruce Perens.
Please enter a command a 0 12
Please enter a command r 0 12
Segmentation fault
```

Контекст неправильной операции можно определить с помощью отладчика.

По умолчанию библиотека Electric Fence выявляет только обращения к памяти после выделенной области. Если необходимо, чтобы она находила только обращения к памяти по адресам, предшествующим началу выделенной области, введите такую команду:

```
% export EF_PROTECT_BELOW=1
```

Чтобы библиотека отслеживала доступ к освобожденным областям, задайте переменную `EF_PROTECT_FREE` равной 1. Дополнительные возможности описаны на `man`-странице `libefence`.

С целью выявления ошибок доступа библиотека Electric Fence запрашивает для каждой выделенной области как минимум две страницы памяти. По умолчанию конец области приходится на конец первой страницы. Выход за пределы области, т.е. обращение ко второй странице, вызывает ошибку сегментации. Если переменная `EF_PROTECT_BELOW` равна 1, начало области выравнивается по началу второй страницы. В связи с тем, что за один вызов функции

malloc() выделяется не менее двух страниц памяти, библиотека Electric Fence способна потреблять достаточно много памяти, поэтому ее рекомендуется использовать только при отладке.

### **A.2.6. Выбор средств отладки**

Мы рассмотрели четыре разных, несовместимых друг с другом средства диагностирования неправильных случаев использования динамической памяти. Ни одно из средств не гарантирует нахождение всех ошибок, но это лучше, чем полное отсутствие проверок. Чтобы облегчить поиск ошибок, выделите код, в котором происходит работа с динамической памятью. Если программа пишется на C++, создайте класс, обрабатывающий все обращения к динамической памяти. При написании программы на языке C постарайтесь минимизировать число функций, в которых выделяется и освобождается память. Тестируя программу, не забывайте о том, что одновременно должно использоваться только одно средство отладки памяти, так как эти средства несовместимы.

Какое же из четырех средств выбрать? Поскольку чаще всего забывают согласовать число операций выделения и освобождения памяти, на начальных этапах разработки лучше применять утилиту mtrace. Она доступна во всех Linux-системах и хорошо себя зарекомендовала. Пройдя данную фазу тестирования, воспользуйтесь утилитой Electric Fence для нахождения неправильных обращений к памяти. Связка двух этих утилит позволяет найти практически все ошибки, связанные с использованием динамической памяти.

### **A.2.7. Исходный текст программы, работающей с динамической памятью**

В листинге A.2 показан исходный текст программы, на примере которой иллюстрируется выделение, освобождение и использование динамической памяти. Описание программы было дано в разделе A.2.1, "Программа для тестирования динамической памяти".

#### ***Листинг A.2. (malloc-use.c) Пример работы с динамической памятью***

```
/* Использование функций работы с динамической памятью. */
```

```
/* Программе передается один аргумент, определяющий  
размер массива. Этот массив состоит из указателей  
на (возможно) выделенные буферы памяти.
```

В процессе работы программы ей можно задавать  
следующие команды:

```
выделение памяти -- a <индекс> <размер_буфера>  
освобождение памяти -- d <индекс>  
чтение памяти -- r <индекс> <смещение>  
запись в память -- w <индекс> <смещение>  
выход -- q
```

```
Ответственность за соблюдение правил доступа  
к динамической памяти лежит на пользователе. */
```

```

#ifdef MTRACE
#include <mcheck.h>
#endif /* MTRACE */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

/* Выделение памяти указанного размера. */
void allocate(char** array, size_t size) {
    *array = malloc(size);
}

/* Освобождение памяти. */
void deallocate(char** array) {
    free((void*)*array);
}

/* Чтение указанной ячейки памяти. */
void read_from_memory(char* array, int position) {
    volatile char character = array[position];
}

/* Запись в указанную ячейку памяти. */
void write_to_memory(char* array, int position) {
    array[position] = 'a';
}

int main(int argc, char* argv[]) {
    char** array;
    unsigned array_size;
    char command[32];
    unsigned array_index;
    char command_letter;
    int size_or_position;
    int error = 0;

#ifdef MTRACE
    mtrace();
#endif /* MTRACE */

    if (argc != 2) {
        fprintf(stderr, "%s: array-size\n", argv[0]);
        return 1;
    }

    array_size = strtoul(argv[1], 0, 0);
    array = (char**)calloc(array_size, sizeof(char*));
    assert(array != 0);

    /* Выполнение вводимых пользователем команд. */
    while (!error) {
        printf("Please enter a command: ");
        command_letter = getchar();
        assert(command_letter != EOF);
        switch (command_letter) {

case 'a':

```

```

fgets(command, sizeof(command), stdin);
if (sscanf(command, "%u %i", &array_index,
&size_or_position) == 2 &&
array_index < array_size)
allocate(&(array[array_index]), size_or_position);
else
error = 1;
break;

case 'd':
fgets(command, sizeof(command), stdin);
if (sscanf(command, "%u", &array_index) == 1 &&
array_index < array_size)
deallocate(&(array[array_index]));
else
error = 1;
break;

case 'r':
fgets(command, sizeof(command), stdin);
if (sscanf(command, "%u %i", &array_index,
&size_or_position) == 2 &&
array_index < array_size)
read_from_memory(array[array_index], size_or_position);
else
error = 1;
break;

case 'w':
fgets(command, sizeof(command), stdin);
if (sscanf(command, "%u %i", &array_index,
&size_or_position) == 2 &&
array_index < array_size)
write_to_memory(array[array_index], size_or_position);
else
error = 1;
break;

case 'q':
free((void*)array);
return 0;

default:
error = 1;
}

free((void*)array);
return 1;
}

```

### A.3. Профилирование

Теперь, когда мы знаем, как искать ошибки в программах, настало время разобраться, как ускорить выполнение программы. Профайлер `gprof` позволяет определить, какие функции требуют наибольших вычислительных ресурсов и тем самым являются кандидатами на

оптимизацию. Профилирование полезно также при отладке, поскольку с помощью этого метода можно установить, какие функции вызываются чаще, чем нужно.

Для получения профильной информации необходимо следовать такому алгоритму.

- 1.Скомпилируйте и скомпонуйте программу с опциями профилирования.
- 2.Запустите программу, чтобы сгенерировать профильные данные.
- 3.Вызовите утилиту `gprof` для отображения и анализа профильных данных.

### A.3.1. Простейший калькулятор

Для иллюстрации методики профилирования мы напишем простейшую программу-калькулятор. Чтобы программа выполнялась нетривиальным образом, заставим ее работать с унарными числами, чего не встречается в реальных калькуляторах. Код программы приведен в конце приложения.

Значение *унарного числа* представляется аналогичным количеством символов. Например, число 1 это "x", 2 "xx", 3 "xxx" и т.д. Вместо символов "x" программа использует связный список, количество элементов которого соответствует значению числа. В файле `number.c` содержатся функции, позволяющие создавать число 0, добавлять единицу к числу, вычитать единицу из числа, а также складывать, вычитать и умножать числа. Есть функция, которая преобразует строку, содержащую неотрицательное десятичное число, в унарное число. Другая функция преобразует унарное число в значение типа `int`. Сложение реализуется путем последовательного добавления единицы, вычитание путем последовательного отнимания единицы, а умножение путем многократного сложения. Функции `even()` и `odd()` возвращают унарный эквивалент единицы тогда и только тогда, когда их единственный операнд является соответственно четным или нечетным числом. В противном случае возвращается унарный эквивалент нуля. Обе функции взаимно рекурсивны. Например, число является четным, если оно равно нулю или если число, на единицу меньше, является нечетным.

Калькулятор принимает однострочные постфиксные выражения [\[40\]](#) и отображает значение каждого выражения:

```
% ./calculator
Please enter a postfix expression:
2 3 +
5
Please enter a postfix expression:
2 3 + 4 -
1
```

Калькулятор, реализованный в файле `calculator.c`, читает каждое выражение и сохраняет промежуточные результаты в стеке унарных чисел, реализованном в файле `stack.c`. Унарные числа представляются в стеке в виде связных списков.

### A.3.2. Сбор профильной информации

Первый этап профилирования заключается в настройке исполняемого файла на сбор профильной информации. Для этого при компиляции и компоновке объектных файлов необходимо указывать флаг `-pg`. Рассмотрим, к примеру, такую последовательность команд:

```
% gcc -pg -c -o calculator.o calculator.c
% gcc -pg -c -o stack.o stack.c
% gcc -pg -c -o number.o number.c
% gcc -pg calculator.o stack.o number.o -o calculator
```

Здесь разрешается сбор информации о вызовах функций и времени их выполнения. Чтобы получать сведения о каждой выполняемой строке программы, укажите флаг -g. При наличии флага -a будет подсчитываться количество итераций циклов.

На втором этапе требуется запустить программу. В процессе ее выполнения профильные данные накапливаются в файле gmon.out. Исследуются только те участки программы, которые действительно выполняются. Чтобы профильный файл был записан, программа должна завершиться нормальным образом.

### А.3.3. Отображение профильных данных

Получив имя исполняемого файла, утилита gprof проверяет файл gmon.out и отображает информацию о том, сколько времени заняло выполнение каждой функции. Давайте проанализируем ход выполнения операции  $1787 \times 13 - 1918$  в нашей программе-калькуляторе, создав *простой профиль*.

```
Flat profile:
Each sample counts as 0.01 seconds.
% cumulative self self total
time seconds seconds calls ms/call ms/call name
26.07 1.76 1.76 20795463 0.00 0.00 decrement_number
24.44 3.41 1.65 1787 0.92 1.72 add
19.85 4.75 1.34 62413059 0.00 0.00 zerop
15.11 5.77 1.02 1792 0.57 2.05 destroy_number
14.37 6.74 0.97 20795463 0.00 0.00 add_one
0.15 6.75 0.01 1788 0.01 0.01 copy_number
0.00 6.75 0.00 1792 0.00 0.00 make_zero
0.00 6.75 0.00 11 0.00 0.00 empty_stack
```

Вычисление функции decrement\_number() и всех вызываемых в ней функций заняло 26,07% общего времени выполнения программы. Эта функция вызывалась 20795463 раза. Каждый вызов выполнялся 0,00 с, т.е. столь малое время, что его не удалось замерить. Функция add() вызывалась 1787 раз, очевидно для вычисления произведения. Каждый проход по функции занимал 0,92 секунды. Функция copy\_number() вызывалась почти столько же раз 1788, но на ее выполнение ушло всего 0.15% общего времени работы программы. Иногда в отчете присутствуют функции mcount() и profil(), используемые профайлером.

В простом профиле отражается время, затраченное на выполнение каждой функции. Утилита gprof умеет также создавать схему вызовов, где показывается время, проведенное не только в каждой функции, но и во всех вызываемых в ее контексте дочерних функциях.

```
index % time self children called name
<spontaneous>
[1] 100.0 0.00 6.75 main [1]
0.006.75 2/2 apply_binary_function [2]
0.00 0.00 1/1792 destroy_number [4]
0.00 0.00 1/1 number_to_unsigned_int [10]
0.00 0.00 3/3 string_to_number [12]
0.00 0.00 3/5 push_stack [16]
0.00 0.00 1/1 create_stack [16]
0.00 0.00 1/11 empty_stack [14]
0.00 0.00 1/5 pop_stack [15]
0.00 0.00 1/1 clear_stack [17]
-----
0.00 6.752/2 main [1]
[2] 100.0 0.00 6.75 2 apply_binary_function [2]
0.00 6.74 1/1 product [3]
0.00 0.01 4/1792 destroy_number [4]
```

```

0.00 0.00 1/1 subtract [11]
0.00 0.00 4/11 empty_stack [14]
0.00 0.00 4/5 pop_stack [15]
0.00 0.00 2/5 push_stack [16]
-----
0.00 6.74 1/1 apply_binary_function [2]
[3] 99.6 0.00 6.74 1 product [3]
1.02 2.65 1767/1792 destroy_number [4]
1.65 1.43 1767/1767 add [5]
0.00 0.00 1760/62413059 zerop [7]
0.00 0.00 1/1792 make_zero [13]

```

В первой секции сообщается о том, что на выполнение функции `main()` и всех ее дочерних функций ушло 100% времени (6.75 секунд). Функцию `main()` вызвал некто `<spontaneous>`: это означает, что профайлер не смог определить, как был осуществлен вызов. В функции `main()` дважды вызывалась функция `apply_binary_function()` (всего таких вызовов в программе было тоже два). В третьей секции сообщается о том, что выполнение функции `product()` и ее дочерних функций заняло 98% времени. Эта функция вызывалась только один раз из функции `apply_binary_function()`.

По схеме вызовов несложно определить время работы той или иной функции. Однако рекурсивные функции требуют особого подхода. Например, функция `even()` вызывает функцию `odd()`, а та снова функцию `even()`. Самому длинному из таких циклов присваивается номер и выделяется отдельная секция отчета. Следующий фрагмент профильных данных получен в результате проверки того, является ли результат операции  $1787 \times 13 \times 3$  четным:

```

-----
0.00 0.02 1/1 main [1]
[9] 0.1 0.00 0.02 1 apply_unary_function [9]
0.01 0.00 1/1 even <cycle 1> [13]
0.00 0.00 1/1806 destroy_number [5]
0.00 0.00 1/13 empty_stack [17]
0.00 0.00 1/6 pop_stack [16]
0.00 0.00 1/6 push_stack [19]
-----
[10] 0.1 0.01 0.00 1+69993 <cycle 1 as a whole> [10]
0.00 0.00 34647 even <cycle 1> [13]
-----
34847 even <cycle 1> [13]
[11] 0.1 0.01 0.00 34847 odd <cycle 1> [11]
0.00 0.00 34847/186997954 zerop [7]
0.00 0.00 1/1806 make_zero [16]
34846 even <cycle 1> [13]

```

Выражение `1+69693` в секции 10 сообщает о том что цикл 1 выполнялся один раз и в нем насчитывается 69693 обращений к функциям. Первой в цикле вызывалась функция `even()`, а из нее функция `odd`. Обе функции вызывались по 34847 раз.

Утилита `gprof` располагает рядом полезных опций.

- При задании опции `-s` будут суммироваться результаты нескольких запусков программы.

- С помощью опции `-c` можно узнать, какие дочерние функции могли быть, но так и не были вызваны

- При задании опции `-l` отображается построчная профильная информация.

- При задании опции `-A` будет отображен исходный текст программы, сопровождаемый процентными показателями времени выполнения.

### **A.3.4. Как работает утилита `gprof`**



Схема работы утилиты `gprof` выглядит следующим образом. Когда в ходе выполнения программы происходит вызов функции, счётчик обращений к функции увеличивается на единицу. Утилита периодически прерывает программу, чтобы выяснить, какая функция выполняется в данный момент. На основании этих "выборок" и определяется время выполнения. В Linux тактовые импульсы генерируются с интервалом 0,01 с, следовательно, это наименьший промежуток между прерываниями. Таким образом, профильные данные о слишком быстро выполняющихся функциях могут оказаться неточными. Во избежание погрешностей рекомендуется запускать программу на длительные периоды времени или суммировать профильные данные по результатам нескольких запусков (это делается с помощью опции `-s`).

### **A.3.5. Исходные тексты программы-калькулятора**

В листинге A.3 показан текст программы, вычисляющей значения постфиксных выражений.

#### ***Листинг A.3. (calculator.c) Основная часть программы-калькулятора***

```
/* Вычисления в унарном формате. */

/* На вход программы подаются однострочные выражения
в обратной польской (постфиксной) записи, например:
602 7 5 - 3 * +
Вводимые числа должны быть неотрицательными
десятичными числами. Поддерживаются операторы
"+", "-" и "*". Унарные операторы "even" и "odd"
возвращают значение 1 в том случае, когда операнд
является четным или нечетным соответственно.
Лексемы разделяются пробелами. Отрицательные числа
не поддерживаются. */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "definitions.h"

/* Эта функция выполняет указанную бинарную операцию над
операндами, извлекаемыми из стека, помещая результат
обратно в стек, в случае успеха возвращается
ненулевое значение. */
int apply_binary_function(number (*function)(number, number),
Stack* stack) {
    number operand1, operand2;
    if (empty_stack(*stack))
        return 0;
    operand2 = pop_stack(stack);
    if (empty_stack(*stack))
        return 0;
    operand1 = pop_stack(stack);
    push_stack(stack, (*function)(operand1, operand2));
    destroy_number(operand1);
    destroy_number(operand2);
    return 1;
}
```

```

/* Эта функция выполняет указанную унарную операцию над
операндом, извлекаемым из стека, помещая результат
обратно в стек. В случае успеха возвращается
ненулевое значение. */
int apply_unary_function(number (*function)(number), Stack* stack) {
    number operand;
    if (empty_stack(*stack))
        return 0;
    operand = pop_stack(stack);
    push_stack(stack, (*function)(operand));
    destroy_number(operand);
    return 1;
}

```

```

int main() {
    char command_line[1000];
    char* command_to_parse;
    char* token;
    Stack number_stack = create_stack();
    while (1) {
        printf("Please enter a postfix expression:\n");
        command_to_parse =
            fgets(command_line, sizeof (command_line), stdin);
        if (command_to_parse == NULL)
            return 0;

        token = strtok(command_to_parse, " \t\n");
        command_to_parse = 0;
        while (token != 0) {
            if (isdigit(token[0]))
                push_stack(&number_stack, string_to_number(token));
            else if (((strcmp(token, "+ ") == 0) &&
                !apply_binary_function(&add, &number_stack)) ||
                ((strcmp(token, "- ") == 0) &&
                !apply_binary_function(&subtract, &number_stack)) ||
                ((strcmp(token, "* ") == 0) &&
                !apply_binary_function(&product, &number_stack)) ||
                ((strcmp(token, "even") == 0) &&
                !apply_unary_function(&even, &number_stack)) ||
                ((strcmp(token, "odd") == 0) &&
                !apply_unary_function(&odd, &number_stack)))
                return 1;
            token = strtok(command_to_parse, " \t\n");
        }
        if (empty_stack(number_stack))
            return 1;
        else {
            number answer = pop_stack(number_stack);
            printf("%u\n", number_to_unsigned_int(answer));
            destroy_number(answer);
            clear_stack(&number_stack);
        }
    }
    return 0;
}

```

Функции, приведенные в листинге А.4 выполняют операции над унарными числами, представленными в виде связанных списков.

```
/* Операции над унарными числами */

#include <assert.h>
#include <stdlib.h>
#include <limits.h>
#include "definitions.h"

/* Создание числа, равного нулю. */
number make_zero() {
    return 0;
}

/* Эта функция возвращает ненулевое значение,
если аргумент равен нулю. */
int zerop(number n) {
    return n == 0;
}

/* Уменьшение числа на единицу. */
number decrement_number(number n) {
    number answer;
    assert(!zerop(n));
    answer = n->one_less_;
    free(n);
    return answer;
}

/* Добавление единицы к числу. */
number add_one(number n) {
    number answer = malloc(sizeof(struct LinkedListNumber));
    answer->one_less_ = n;
    return answer;
}

/* Удаление числа. */
void destroy_number(number n) {
    while (!zerop(n))
        n = decrement_number(n);
}

/* Копирование числа. Эта функция необходима для того,
чтобы при временных вычислениях не искажались
исходные операнды. */
number copy_number(number n) {
    number answer = make_zero();
    while (!zerop(n)) {
        answer = add_one(answer);
        n = n->one_less_;
    }
    return answer;
}

/* Сложение двух чисел. */
number add(number n1, number n2) {
    number answer = copy_number(n2);
```

```

number addend = n1;
while(!zerop(addend)) {
    answer = add_one(answer);
    addend = addend->one_less_;
}
return answer;
}

/* Вычитание одного числа из другого. */
number subtract(number n1, number n2) {
    number answer = copy_number(n1);
    number subtrahend = n2;
    while(!zerop(subtrahend)) {
        assert(!zerop(answer));
        answer = decrement_number(answer);
        subtrahend = subtrahend->one_less_;
    }
    return answer;
}

/* Умножение двух чисел. */
number product(number n1, number n2) {
    number answer = make_zero();
    number multiplicand = n1;
    while (!zerop(multiplicand)) {
        number answer2 = add(answer, n2);
        destroy_number(answer);
        answer = answer2;
        multiplicand = multiplicand->one_less_;
    }
    return answer;
}

/* Эта функция возвращает ненулевое значение, если
ее аргумент является четным числом. */
number even(number n) {
    if (zerop(n))
        return add_one(make_zero());
    else
        return odd(n->one_less_);
}

/* Эта функция возвращает ненулевое значение, если
ее аргумент является нечетным числом. */
number odd (number n) {
    if (zerop(n))
        return make_zero();
    else
        return even(n->one_less_);
}

/* Приведение строки, содержащей десятичное целое,
к типу "number". */
number string_to_number(char* char_number) {
    number answer = make_zero();
    int num = strtoul(char_number, (char **)0, 0);
    while (num != 0) {
        answer = add_one(answer);
        --num;
    }
}

```

```

}
return answer;
}

/* Приведение значения типа "number"
к типу "unsigned int". */
unsigned number_to_unsigned_int (number n) {
    unsigned answer = 0;
    while (!zerop(n)) {
        n = n->one_less_;
        ++answer;
    }
    return answer;
}

```

Функции, приведенные в листинге А.5, реализуют стек унарных чисел, представленных в виде СВЯЗНЫХ СПИСКОВ.

### *Листинг А.5. (stack.c) Стек унарных чисел*

```

/* Реализация стека значений типа "number". */

#include <assert.h>
#include <stdlib.h>
#include "definitions.h"

/* Создание пустого стека. */
Stack create_stack() {
    return 0;
}

/* Эта функция возвращает ненулевое значение,
если стек пуст. */
int empty_stack(Stack stack) {
    return stack == 0;
}

/* Удаление числа, находящегося на вершине стека.
Если стек пуст, программа аварийно завершается. */
number pop_stack(Stack* stack) {
    number answer;
    Stack rest_of_stack;

    assert(!empty_stack(*stack));
    answer = (*stack)->element_;
    rest_of_stack = (*stack)->next_;
    free(*stack);
    *stack = rest_of_stack;
    return answer;
}

/* Добавление числа в начало стека. */
void push_stack(Stack* stack, number n) {
    Stack new_stack =
        malloc(sizeof(struct StackElement));
    new_stack->element_ = n;
}

```

```

new_stack->next_ = *stack;
*stack = new_stack;
}

/* Очистка стека. */
void clear_stack(Stack* stack) {
while(!empty_stack(*stack)) {
number top = pop_stack (stack);
destroy_number(top);
}
}

```

В листинге А.6 показаны объявления типов данных и функций работы со стеком и унарными числами.

#### ***Листинг А.6. (definitions.h) Файл заголовков для файлов number.c и stack.c***

```

#ifndef DEFINITIONS_H
#define DEFINITIONS_H 1

/* Представление числа в виде связанного списка. */
struct LinkedListNumber {
struct LinkedListNumber* one_less_;
};

typedef struct LinkedListNumber* number;

/* Реализация стека чисел, представленных в виде
связных списков. Значение 0 соответствует
пустому стеку. */
struct StackElement {
number element_;
struct StackElement* next_;
};
typedef struct StackElement* Stack;

/* Операции над стеком. */
Stack create_stack();
int empty_stack(Stack stack);
number pop_stack Stack* stack);
void push_stack(Stack* stack, number n);
void clear_stack(Stack* stack);

/* Операции над числами */
number make_zero();
void destroy_number(number n);
number add(number n1, number n2);
number subtract(number n1, number n2);
number product(number n1, number n2);
number even(number n);
number odd(number n);
number string_to_number(char* char_number);
unsigned number_to_unsigned_int(number n);

#endif /* DEFINITIONS_H */

```

# Приложение Б

## Низкоуровневый ввод-вывод

Программисты, пишущие Linux-программы на языке C, имеют в своем распоряжении два набора функций ввода-вывода. Один из них включен в стандартную библиотеку языка C: `printf()`, `fopen()` и т.д. <sup>[41]</sup> Мы предполагаем, что читатели уже знакомы с языком C и знают, как использовать эти функции ввода-вывода, поэтому не будем их подробно описывать.

Ядро Linux предоставляет собственные операции ввода-вывода, работающие на более низком уровне. В основном они имеют вид системных вызовов и обеспечивают самый непосредственный доступ к файловой системе. По сути, стандартные библиотечные функции реализованы на их основе. Низкоуровневые вызовы обеспечивают наибольшую эффективность операций ввода-вывода.

### Б.1. Чтение и запись данных

Первая функция ввода-вывода, с которой сталкиваются те, кто начинают изучать язык C, называется `printf()`. Она форматирует текстовую строку и записывает ее в стандартный выходной поток. Обобщенная ее версия `fprintf()` записывает текст в заданный поток. Поток данных представляется в программе указателем типа `FILE*`. Чтобы получить этот указатель, необходимо открыть файл с помощью функции `fopen()`. По завершении работы с файлом его необходимо закрыть с помощью функции `fclose()`. Помимо функции `fprintf()` существуют также функции `fputc()`, `fputs()` и `fwrite()`, записывающие данные в поток. Функции `fscanf()`, `fgetc()`, `fgets()` и `fread()` читают данные из потока.

В низкоуровневых операциях ввода-вывода участвуют не файловые указатели, а дескрипторы. Дескриптор представляет собой целое число, обозначающее конкретный экземпляр файла, открытого в одном процессе. Файл можно открыть для чтения, записи, а также одновременно для чтения и записи. Файловому дескриптору не обязательно соответствует файл: это может быть другой системный компонент, способный передавать или принимать данные (аппаратное устройство, сокет, противоположный конец канала).

Для работы с описанными ниже низкоуровневыми функциями необходимо включить в программу файлы `<fcntl.h>`, `<sys/types.h>`, `<sys/stat.h>` и `<unistd.h>`.

#### Б.1.1. Открытие файла

Чтобы открыть файл и получить дескриптор для работы с ним, необходимо вызвать функцию `open()`. В качестве аргументов она принимает строку с путевым именем файла и флаги, определяющие способ открытия. С помощью функции `open()` можно также создать новый файл. Для этого ей нужно передать третий аргумент, определяющий права доступа к файлу.

Если второй аргумент равен `O_RDONLY`, файл открывается только для чтения. При попытке записи в такой файл будет выдана ошибка. Точно так же флаг `O_WRONLY` объявляет файл доступным только для записи. В случае флага `O_RDWR` файл открывается и для чтения, и для записи. Не всякий файл можно открыть в любом из трех режимов. Например, существующие права доступа к файлу могут не позволить конкретному процессу открывать файл для чтения или записи. Файл, находящийся в устройстве, запись в которое невозможна (скажем, компакт-

диск), тем более нельзя открыть для записи.

Существуют и другие флаги, определяющие режим открытия файла. Все они могут объединяться с помощью операции побитового ИЛИ. Перечислим наиболее распространенные флаги.

■ `O_TRUNC` приводит к очистке существующего файла. Данные, записываемые в файл, замещают предыдущее содержимое файла.

■ `O_APPEND` приводит к открытию файла в режиме добавления. Данные, записываемые в файл, добавляются в его конец.

■ `O_CREAT` означает создание нового файла. Если указанное имя соответствует несуществующему файлу, он будет создан при условии, что заданный каталог существует и процесс имеет разрешение создавать в нем файлы. Если файл уже существует, он будет открыт. При наличии дополнительного флага `O_EXCL` функция `open()` откажется открывать существующий файл.

Когда в функции `open()` задан флаг `O_CREAT`, должен присутствовать третий аргумент, определяющий права доступа к создаваемому файлу. О режиме доступа к файлу и битах режима рассказывалось в разделе 10.3, "Права доступа к файлам".

Программа, представленная в листинге Б.1, создает файл, имя которого задано в командной строке. Функции `open()` передается флаг `O_EXCL`, поэтому в случае указания существующего файла возникнет ошибка. Владельцу и группе нового файла предоставляются права чтения и записи, остальным пользователям только право чтения (если для пользователя, которому принадлежит программа, установлено значение `umask`, права доступа к файлу могут оказаться более жесткими).

### ***Значения `umask`***

При создании файла с помощью функции `open()` некоторые из указываемых битов режима могут отключаться. Это следствие того, что значение `umask` не равно нулю. Данное значение определяет биты, которые отнимаются от кода режима всех файлов, создаваемых пользователем. Правило определения режима доступа к файлу таково, значение `umask` подвергается инверсии, а затем побитово умножается на заданный код режима. Полученное значение становится новым кодом режима.

Для изменения значения `umask` предназначена одноименная команда, принимающая восьмеричный аргумент. Если требуется изменить значение `umask` работающего процесса, вызовите функцию `umask()`.

Например, функция  
`umask(S_IRWXO | S_IWGRP);`

и команда  
`% umask 027`

означают, что право записи для группы а также права чтения, записи и выполнения для остальных пользователей будут всегда отниматься от прав доступа к создаваемым файлам.

### ***Листинг Б.1. (`createfile.c`) Создание файла***

```
#include <fcntl.h>
#include <stdio.h>
```



```

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

int main(int argc, char* argv[]) {
/* Путь к новому файлу */
char* path = argv[1];
/* Права доступа к файлу. */
mode_t mode =
S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH;

/* Создание файла. */
int fd = open(path, O_WRONLY | O_EXCL | O_CREAT, mode);
if (fd == -1) {
/* Произошла ошибка. Выводим сообщение и завершаем работу. */
perror("open");
return 1;
}
return 0;
}

```

Результаты работы программы будут такими:

```

% ./create-file testfile
% ls -l testfile
-rw-rw-r-- 1 samuel users 0 Feb 1 22:47 testfile
% ./create-file testfile
open: File exists

```

Обратите внимание на то, что длина файла равна нулю, так как программа не записывала в него никакие данные.

### Б.1.2. Закрывание файла

По окончании работы с файлом его следует закрыть с помощью функции `close()`. В ряде случаев, например в программе, показанной в листинге Б.1, нет необходимости вызывать данную функцию явно, так как ОС Linux автоматически закрывает все открытые файлы по завершении программы. Естественно, после того как файл был закрыт, обращаться к нему нельзя.

Закрывание файла вызывает разную реакцию операционной системы, в зависимости от природы файла. Например, когда закрывается сокет, происходит разрыв сетевого соединения между двумя компьютерами, взаимодействующими через сокет.

Linux ограничивает число файлов, которые могут быть открыты процессом в определенный момент времени. Дескрипторы открытых файлов занимают ресурсы ядра, поэтому желательно вовремя закрывать файлы, чтобы дескрипторы удалялись из системных таблиц. Обычно процессам назначается лимит в 1024 дескриптора. Изменить это значение позволяет системный вызов `setrlimit()` (см. раздел 8.5, "Функции `getrlimit()` и `setrlimit()`: лимиты ресурсов").

### Б.1.3. Запись данных

Для записи данных в файл предназначена функция `write()`. Она принимает дескриптор файла, указатель на буфер данных и число записываемых байтов. Файл должен быть открыт для записи. Функция `write()` работает не только с текстовыми данными, но и с произвольными байтами.

В листинге Б.2 показана программа, которая записывает в указанный файл значение текущего времени. Если файл не существует, он создается. Для получения и форматирования значения времени программа использует функции `time()`, `localtime()` и `asctime()`.

### ***Листинг Б.2. (timestamp.c) Запись в файл метки времени***

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>

/* Эта строка возвращает строку, содержащую значение
текущих даты и времени. */
char* get_timestamp() {
    time_t now = time(NULL);
    return asctime(localtime(&now));
}

int main(int argc, char* argv[]) {
    /* Файл, в который записывается метка времени. */
    char* filename = argv[1];
    /* Получение метки времени. */
    char* timestamp = get_timestamp();
    /* Открытие файла для записи. Если файл существует, он
открывается в режиме добавления; в противном случае
файл создается. */
    int fd =
open(filename, O_WRONLY | O_CREAT | O_APPEND, 0666);
    /* Вычисление длины строки с меткой времени. */
    size_t length = strlen(timestamp);
    /* Запись метки времени в файл. */
    write(fd, timestamp, length);
    /* Конец работы. */
    close(fd);
    return 0;
}
```

Вот как работает программа:

```
% ./timestamp tsfile
% cat tsfile
The Feb 1 23:25:20 2001
% ./timestamp tsfile
% cat tsfile
Thu Feb 1 23:25:20 2001
Thu Feb 1 23:25:47 2001
```

Обратите внимание на то, что при первом вызове программы `timestamp` файл был создан, а при втором вызове дополнен.

Функция `write()` возвращает число записанных байтов или -1, если произошла ошибка. Для некоторых типов файлов чисто фактически записанных байтов может оказаться меньше требуемого. Программа должна выявлять подобные случаи и вызывать функцию `write()` повторно, чтобы передать оставшуюся часть данных. Этот прием продемонстрирован в листинге Б.3. Но иногда даже таких методов недостаточно. Например, если показанная функция будет

записывать данные в сокет, в нее придется добавить код проверки того, не произошел ли в ходе операции записи разрыв соединения.

### *Листинг Б.3. (write-all.c) Запись буфера*

```
/* Запись указанного числа байтов (COUNT) из буфера BUFFER
в файл FD. В случае ошибки возвращается -1,
иначе -- число записанных байтов. */

ssize_t write_all(int fd, const void* buffer, size_t count) {
    size_t left_to_write = count;
    while (left_to_write > 0) {
        size_t written = write(fd, buffer, count);
        if (written == -1)
            /* Произошла ошибка, завершаем работу. */
            return -1;
        else
            /* подсчитываем число оставшихся байтов. */
            left_to_write -= written;
    }
    /* Нельзя записать больше, чем COUNT байтов! */
    assert(left_to_write == 0);
    /* Число записанных байтов равно COUNT. */
    return count;
}
```

## **Б.1.4. Чтение данных**

Функция, осуществляющая чтение данных из файла, называется `read()`. Подобно функции `write()`, она принимает дескриптор файла, указатель на буфер и счетчик числа извлекаемых байтов. Функция возвращает число прочитанных байтов или `-1` в случае ошибки. Иногда читается меньше байтов, чем требовалось, если, например, в файле содержится недостаточно байтов.

### *Чтение текстовых файлов DOS/Windows*

В Linux-программах нередко приходится читать файлы, созданные в DOS или Windows. Важно понимать разницу между тем, как структурируются текстовые файлы в Linux и в DOS/Windows.

В Linux каждая строка текстового файла оканчивается символом новой строки. Он представляется символьной константой `'\n'`, ASCII-код которой равен 10. В Windows строки разделяются двухсимвольной комбинацией символ возврата каретки (константа `'\r'`, ASCII-код 13), за которым идет символ новой строки.

Некоторые текстовые редакторы Linux при отображении текстовых файлов Windows ставят в конце каждой строки обозначение `^M` символ возврата каретки. В Emacs такие файлы отображаются правильно, но в строке режима появляется запись (dos). Многие Windows-редакторы, например Notepad (Блокнот), показывают содержимое текстовых файлов Linux в виде одной длинной строки, так как предполагают наличие в конце строки символа возврата каретки.

Если программа читает текстовые файлы, сгенерированные Windows-программами, желательно менять последовательность '\r\n' одним символом новой строки. Точно так же при записи текстовых файлов, которые будут читаться Windows-программами, нужно менять одиночные символы новой строки комбинациями '\r\n'.

В листинге Б.4 демонстрируется применение функции `read()`. Программа отображает шестнадцатиричный дамп файла, заданного в командной строке. В каждой строке показано смещение от начала файла, а затем следующие 16 байтов.

#### ***Листинг Б.4. (hexdump.c) Отображение шестнадцатеричного дампа файла***

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

int main(int argc, char* argv[]) {
    unsigned char buffer[16];
    size_t offset = 0;
    size_t bytes_read;
    int i;

    /* Открытие файла для чтения. */
    int fd = open(argv[1], O_RDONLY);

    /* Чтение данных из файла по одному блоку за раз. Чтение
    продолжается до тех пор, пока размер очередной порции байтов
    не окажется меньше размера буфера. Это свидетельствует
    о достижении конца буфера. */
    do {
        /* чтение следующей строки байтов. */
        bytes_read = read(fd, buffer, sizeof(buffer));
        /* Отображение смещения, а затем самих байтов. */
        printf("0x%06x : ", offset);
        for (i = 0; i < bytes_read; ++i)
            printf("%02x ", buffer[i]);
        printf("\n");
        /* Вычисление позиции в файле. */
        offset += bytes_read;
    }
    while (bytes_read == sizeof(buffer));

    /* Конец работы. */
    close(fd);
    return 0;
}
```

Ниже показаны результаты работы программы. Она выводит дамп самой себя.

```
% ./hexdump hexdump
0x0000000 : 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
0x0000010 : 02 00 03 00 01 00 00 00 c0 b3 04 0b 34 00 00 00
0x0000020 : e8 23 00 00 00 00 00 00 34 00 20 00 06 00 28 00
0x0000030 : 1d 00 1a 00 06 00 00 00 34 00 00 00 34 80 04 08
...
```

Эти результаты могут быть разными в зависимости от того, какой компилятор применялся и какие флаги компиляции были установлены.

### **Б.1.5. Перемещение по файлу**

В файловом дескрипторе запоминается текущая позиция в файле. При чтении или записи данных указатель текущей позиции перемещается на то количество байтов, которое было прочитано или записано. Но иногда нужно осуществлять простое перемещение по файлу (позиционирование). Например, может потребоваться вернуться в начало файла и прочитать его заново, не открывая повторно.

Позиционирование указателя текущей позиции файла осуществляет функция `lseek()`. Она принимает дескриптор файла и два дополнительных аргумента, определяющих новую позицию указателя.

■ Если третий аргумент равен `SEEK_SET`, функция `lseek()` интерпретирует второй аргумент как смещение (в байтах) от начала файла.

■ Если третий аргумент равен `SEEK_CUR`, функция `lseek()` интерпретирует второй аргумент как смещение (положительное или отрицательное) от текущей позиции.

■ Если третий аргумент равен `SEEK_END`, функция `lseek()` интерпретирует второй аргумент как смещение (в байтах) от конца файла.

Функция `lseek()` возвращает смещение новой позиции от начала файла. Тип этого значения `off_t`. В случае ошибки возвращается `-1`. Функция неприменима к файлам некоторых типов, например к сокетам.

Если требуется узнать текущую позицию файла, задайте смещение 0:

```
off_t position = lseek(file_descriptor, 0, SEEK_CUR);
```

ОС Linux позволяет перемещать указатель текущей позиции за пределы файла. Обычно, если текущая позиция находится за концом файла и выполняется операция записи, операционная система автоматически увеличивает файл, чтобы вместить в него новые данные. "Промежуток" между старым признаком конца файла и указателем текущей позиции не записывается на диск. Linux лишь помечает его длину. Если впоследствии попытаться прочесть файл, окажется, что данный промежуток заполнен нулевыми байтами.

Благодаря данной особенности функции `lseek()` можно создавать файлы огромного размера, практически не занимающие места на диске. Это продемонстрировано в листинге Б.5. В качестве аргументов командной строки программа принимает имя файла и требуемый размер в мегабайтах. Программа создает файл, перемещается с помощью функции `lseek()` на нужное расстояние и записывает нулевой байт, после чего закрывает файл.

#### ***Листинг Б.5. (lseek-huge.c) Создание огромных файлов с помощью функции lseek()***

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

int main (int argc, char* argv[]) {
    int zero = 0;
    const int megabyte = 1024 * 1024;
    char* filename = argv[1];
```

```
size_t length = (size_t)atoi(argv[2]) * megabyte;
```

```
/* Создание нового файла. */  
int fd = open(filename, O_WRONLY | O_CREAT | O_EXCL, 0666);  
/* Перемещение в точку, где должен быть записан последний байт  
файла. */  
lseek(fd, length - 1, SEEK_SET);  
/* Запись нулевого байта. */  
write(fd, &zero, 1);  
/* Конец работы. */  
close(fd);  
return 0;  
}
```

Давайте теперь создадим файл размером 1 Гбайт. Обратите внимание на объем свободного места на диске до и после выполнения программы.

```
% df -h .  
Filesystem Size Used Avail Use% Mounted on  
/dev/hda5 2.9G 2.1G 655M 76% /  
% ./lseek-huge bigfile 1024 % ls -l bigfile  
-rw-r----- 1 samuel samuel 1073741824 Feb 5 16:29 bigfile  
% df -h .  
Filesystem Size Used Avail Use% Mounted on  
/dev/hda5 2.9G 2.1G 655M 76% /
```

Как видите, файл практически не занимает место на диске, несмотря на свой огромный размер. Но если открыть его и попытаться прочитывать данные, окажется, что в нем находится 1 Гбайт нулей. Давайте, к примеру, проверим это с помощью программы hexdump:

```
% ./hexdump bigfile / head -10  
0x0000000 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
0x0000010 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
0x0000020 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
0x0000030 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
0x0000040 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
0x0000050 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
...
```

Чтобы не наблюдать, как по экрану проносятся  $2^{30}$  нулей, нажмите <Ctrl+C>.

"Волшебные промежутки" в файлах являются особенностью файловых систем типа ext2, обычно создаваемых на жестких дисках Linux. Если попытаться с помощью программы lseek-huge создать файл в файловой системе типа fat или vfat, то он займет весь указанный объем диска.

ОС Linux не позволяет функции lseek() ставить указатель текущей позиции перед началом файла.

## Б.2. Функция stat()

Функция read() позволяет прочесть только содержимое файла. Но как насчет остальной информации? Например, команда ls -l сообщает такие сведения о файлах в текущем каталоге, как размер, время последнего обновления, права доступа, владелец и пр. Аналогичную информацию об отдельном файле можно получить с помощью функции stat(). Ей необходимо передать путевое имя файла и указатель на структуру типа stat. В случае успешного завершения функция возвращает 0 и заполняет поля структуры данными о файле, иначе возвращается -1.

Перечислим наиболее полезные поля структуры stat.

■ В поле st\_mode содержится код доступа к файлу. О правах доступа к файлам рассказывалось в разделе 10.3. "Права доступа к файлам". В старшем бите поля закодирован тип

файла. Об этом пойдет речь ниже.

■ В полях `st_uid` и `st_gid` содержатся идентификаторы соответственно пользователя и группы, которым принадлежит файл. Назначение идентификатора описывалось в разделе 10.1, "Пользователи и группы".

■ В поле `st_size` хранится размер файла в байтах.

■ В поле `st_atime` записано время последнего обращения к файлу (для чтения или записи).

■ В поле `st_mtime` записано время последней модификации файла.

Следующие макросы проверяют поле `st_mode`, чтобы определить, для файла какого типа была вызвана функция `stat`. Макросы возвращают ненулевое значение, если их догадка о типе файла подтвердилась.

■ `S_ISBLK(код доступа)` блочное устройство;

■ `S_ISCHR(код доступа)` символьное устройство;

■ `S_ISDIR(код доступа)` каталог;

■ `S_ISFIFO(код доступа)` FIFO-файл (именованный канал);

■ `S_ISLNK(код доступа)` символическая ссылка.

■ `S_ISREG(код доступа)` обычный файл;

■ `S_ISSOCK(код доступа)` сокет.

В поле `st_dev` содержатся старший и младший номера аппаратного устройства, в котором расположен файл (о номерах устройств рассказывалось в главе 6, "Устройства"). Старший номер находится в старшем байте поля, а младший в младшем. В поле `st_info` содержится номер индексного дескриптора файла, определяющий местоположение файла в файловой системе.

Если вызвать функцию `stat()` для символической ссылки, функция проследит, куда указывает ссылка, и вернет информацию о том файле, а не о самой ссылке. Таким образом, в случае функции `stat()` макрос `S_ISLNK()` всегда будет возвращать значение 0. Есть другая функция, `lstat()`, которая не пытается отслеживать символические ссылки. Во всем остальном она эквивалентна функции `stat()`. Если вызвать функцию `stat()` для поврежденной ссылки (которая указывает на несуществующий или недоступный файл), возникнет ошибка, тогда как функция `lstat()` в подобной ситуации выполнится успешно.

Если файл уже открыт для чтения или записи, лучше пользоваться функцией `fstat()`. В качестве первого аргумента она принимает не путевое имя, а дескриптор.

В листинге Б.6 показана функция которая создает буфер достаточного размера и загружает в него содержимое указанного файла. Размер файла определяется с помощью функции `fstat()`. Она же позволяет проверить, соответствует ли заданное имя обычному файлу.

### ***Листинг Б.6. (read-file.c) Загрузка файла в буфер***

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

/* Загрузка содержимого файла FILENAME в память.
Размер буфера записывается в аргумент LENGTH.
Создаваемый буфер должен удаляться в вызывающей функции.
Если аргумент FILENAME не соответствует обычному файлу,
возвращается NULL. */
char* read_file(const char* filename, size_t* length) {
```

```

int fd;
struct stat file_info;
char* buffer;

/* Открытие файла. */
fd = open(filename, O_RDONLY);

/* Получение информации о файле. */
fstat(fd, &file_info);
*length = file_info.st_size;
/* Проверка того, что это обычный файл. */
if (!S_ISREG(file_info.st_mode)) {
/* Этот тип файла не поддерживается. */
close(fd);
return NULL;
}

/* выделение буфера достаточного размера. */
buffer = (char*)malloc(*length);
/* Загрузка файла в буфер. */
read(fd, buffer, *length);

/* Конец работы. */
close(fd);
return buffer;
}

```

### Б.3. Векторные чтение и запись

Аргументами функции `write()` являются указатель на буфер и длина буфера. Эта функция записывает в файл непрерывный блок данных, хранящихся в памяти. Но программам часто требуется записывать группы блоков, хранящихся по разным адресам. Если использовать функцию `write()`, придется либо предварительно объединять блоки в памяти, что неэффективно, либо многократно вызывать функцию. Последнее тоже не всегда приемлемо. Например, при записи в сокет два вызова функции `write()` приведут к отправке в сеть двух пакетов, тогда как те же самые данные можно перестать в одном пакете.

Функция `writenv()` позволяет записать в файл несколько несвязанных буферов одновременно. Это называется *векторной записью*. Сложность применения функции `writenv()` заключается в создании структуры, задающей начало и конец каждого буфера. Эта структура представляет собой массив элементов типа `struct iovec`. Каждый элемент описывает одну область памяти. В поле `iov_base` указывается адрес начала области, а в поле `iov_len` ее длина. Если число буферов известно заранее, можно просто объявить массив типа `struct iovec`. В противном случае придется выделять массив динамически.

Функции `writenv()` передается дескриптор записываемого файла, массив структур `iovec` и размер массива. Функция возвращает общее число записанных байтов.

Программа, показанная в листинге Б.7, записывает свои аргументы командной строки в файл с помощью одной-единственной функции `writenv()`. Первый аргумент это имя файла, в котором сохраняются все последующие аргументы, каждый в отдельной строке. Число элементов в массиве структур `iovec` в два раза превышает число аргументов командной строки, так как после каждого аргумента записывается символ новой строки. Поскольку количество аргументов неизвестно заранее, массив создается с помощью функции `malloc()`.



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

int main(int argc, char* argv[]) {
    int fd;
    struct iovec* vec;
    struct iovec* vec_next;
    int i;

    /* Символ новой строки хранится в обычной переменной
    типа char. */
    char newline = '\n';
    /* Первый аргумент командной строки -- это имя выходного
    файла. */
    char* filename = argv[1];
    /* Пропускаем первые два элемента списка аргументов.
    Элемент номер 0 -- это имя самой программы,
    а элемент номер 1 -- это имя выходного файла */
    argc -= 2;
    argv += 2;

    /* Выделяем массив элементов типа iovec каждому аргументу
    командной строки соответствует два элемента массива:
    один -- для самого аргумента,
    а другой -- для символа новой строки. */
    vec =
    (struct iovec*)malloc(2 * argc * sizeof(struct iovec));

    /* Просмотр списка аргументов и создание массива. */
    vec_next = vec;
    for (i = 0; i < argc; ++i) {
        /* первый элемент -- это текст аргумента */
        vec_next->iiov_base = argv[i];
        vec_next->iiov_len = strlen(argv[i]);
        ++vec_next;
        /* Второй элемент -- это символ новой строки, допускается,
        чтобы несколько элементов массива указывали на одну и
        ту же область памяти. */
        vec_next->iiov_base = &newline;
        vec_next->iiov_len = 1;
        ++vec_next;
    }

    /* Запись аргументов в файл. */
    fd = open(filename, O_WRONLY | O_CREAT);
    writew(fd, vec, 2 * argc);
    close(fd);
    free(vec);
    return 0;
}
```

Вот пример работы программы:

```
% ./write-args outputfile "first arg" "second arg" "third arg"
% cat outputfile
first arg
second arg
third arg
```

В Linux имеется также функция `readv()`, которая загружает содержимое файла в несколько несвязанных областей памяти. Как и в функции `writev()`, массив структур типа `iovec` определяет начало и размер каждой области.

## Б.4. Взаимосвязь с библиотечными функциями ввода-вывода

Выше уже говорилось о том, что функции ввода-вывода стандартной библиотеки языка C реализованы на основе низкоуровневых функций. Иногда удобнее работать с одними, иногда с другими.

Если файл был открыт с помощью функции `fopen()`, то узнать его дескриптор позволяет функция `fileno()`. Она принимает аргумент типа `FILE*` и возвращает соответствующий ему дескриптор. Например, можно открыть файл с помощью библиотечной функции `fopen()`, но осуществить в него запись посредством функции `writev()`:

```
FILE* stream = fopen(filename, "w");
int file_descriptor = fileno(stream);
writev(file_descriptor, vector, vector_length);
```

Учтите, что переменные `stream` и `file_descriptor` соответствуют одному и тому же открытому файлу. Если выполнить следующую функцию, дескриптор `file_descriptor` станет недействительным:

```
fclose(stream);
```

Аналогичным образом следующая функция делает недействительным файловый указатель `stream`:

```
close(file_descriptor);
```

Чтобы получить файловый указатель, соответствующий дескриптору, воспользуйтесь функцией `fdopen()`. Ее аргументами является дескриптор и строка, определяющая режим создания файлового потока. Синтаксис строки аналогичен синтаксису второго аргумента функции `fopen()`, а задаваемый режим должен быть совместим с режимом открытия файла. Например, файлу, открытому для чтения, соответствует режим `r`, а файлу, открытому для записи, режим `w`. Как и в случае функции `fileno()`, файловый указатель и дескриптор ссылаются на один и тот же файл, поэтому закрытие одного сделает недействительным другой.

## Б.5. Другие низкоуровневые операции

Есть ряд других полезных функций для работы с файлами и каталогами.

- Функция `getcwd()` возвращает имя текущего каталога. Она принимает два аргумента: указатель на буфер и длину буфера и копирует имя каталога в буфер.

- Функция `chdir()` делает текущим заданный каталог.

- Функция `mkdir()` создает новый каталог. Ее первым аргументом является путевое имя каталога. Второй аргумент задает права доступа к каталогу. Интерпретация этого аргумента такая же, как и третьего аргумента функции `open()`. На итоговый код доступа влияет значение `umask` процесса.

- Функция `rmdir()` удаляет указанный каталог.

- Функция `unlink()` удаляет файл. Ее аргументом является путевое имя файла. С помощью этой функции можно удалять и другие объекты файловой системы, например именованные

каналы и файлы устройств.

В действительности функция `unlink()` не обязательно удаляет содержимое файла. Как подсказывает ее имя, она удаляет из каталога ссылку на файл. Файл не будет больше фигурировать в списке содержимого каталога, но если какой-то процесс владеет открытым дескриптором этого файла, то содержимое файла не удаляется с диска. Это произойдет только тогда, когда не останется открытых дескрипторов файла. Так что если один процесс откроет файл для чтения или записи, а второй процесс в это время удалит ссылку на файл и создаст новый файл с таким же именем, первый процесс продолжит работать со старым содержимым файла. Чтобы получить доступ к новому содержимому первому процессу придется закрыть и повторно открыть файл.

■ Функция `rename()` переименовывает или перемещает файл. Двумя ее аргументами являются старое и новое путевые имена. Если путевые имена ссылаются на разные каталоги, функция перемещает файл (при условии, что он остается в той же файловой системе). С помощью функции `rename()` можно перемещать также каталоги и другие объекты файловой системы.

## Б.6. Чтение содержимого каталога

В Linux имеются функции, предназначенные для чтения содержимого каталога. И хотя они не относятся к низкоуровневым функциям, мы все же решили их описать, так как они широко применяются в программах.

При чтении содержимого каталога необходимо придерживаться такой последовательности действий.

1. Вызовите функцию `opendir()`, передав ей путевое имя требуемого каталога. Эта функция возвращает дескриптор типа `DIR*`, который можно использовать для доступа к содержимому каталога. В случае ошибки возвращается `NULL`.

2. Последовательно вызывайте функцию `readdir()`, передавая ей дескриптор, полученный от функции `opendir()`. Всякий раз функция `readdir()` будет возвращать указатель на структуру типа `dirent`, содержащую информацию о следующем элементе каталога. По достижении конца каталога будет получено значение `NULL`. У структуры `dirent` есть поле `d_name`, где содержится имя элемента каталога.

3. Вызовите функцию `closedir()`, передав ей имеющийся дескриптор, чтобы завершить сеанс работы с каталогом.

Для использования перечисленных функций необходимо включить в программу файлы `<sys/types.h>` и `<dirent.h>`. Ответственность за сортировку содержимого каталога возлагается на программу.

В листинге Б.8 показана программа отображающая список содержимого каталога. Имя каталога задается в командной строке. Если этого не сделать, будет проанализирован текущий каталог. Для каждого элемента каталога отображается его тип и путевое имя. Функция `get_file_type()` определяет тип объекта файловой системы с помощью функции `lstat()`.

### Листинг Б.8. (`listdir.c`) Вывод содержимого каталога

```
#include <assert.h>
#include <dirent.h>
#include <stdio.h>
#include <string.h>
```

```

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

/* Эта функция возвращает строку с описанием типа объекта
файловой системы, заданного в аргументе PATH. */
const char* get_file_type(const char* path) {
    struct stat st;
    lstat(path, &st);
    if (S_ISLNK(st.st_mode))
        return "symbolic link";
    else if (S_ISDIR(st.st_mode))
        return "directory";
    else if (S_ISCHR(st.st_mode))
        return "character device";
    else if (S_ISBLK(st.st_mode))
        return "block device";
    else if (S_ISFIFO(st.st_mode))
        return "fifo";
    else if (S_ISSOCK(st.st_mode))
        return "socket";
    else if (S_ISREG(st.st_mode))
        return "regular file";
    else
        /* Нераспознанный тип. */
        assert(0);
}

int main(int argc, char* argv[]) {
    char* dir_path;
    DIR* dir;
    struct dirent* entry;
    char entry_path[PATH_MAX + 1];
    size_t path_len;

    if (argc >= 2)
        /* Если каталог указан в командной строке, анализируем его. */
        dir_path = argv[1];
    else
        /* В противном случае анализируем текущий каталог. */
        dir_path = ".";
    /* Копируем имя каталога в переменную entry_path. */
    strncpy(entry_path, dir_path, sizeof(entry_path));
    path_len = strlen(dir_path);
    /* Если имя каталога не заканчивается косой чертой,
    добавляем ее. */
    if (entry_path[path_len - 1] != '/') {
        entry_path[path_len] = '/';
        entry_path[path_len + 1] = '\0';
        ++path_len;
    }

    /* Начинаем обрабатывать список содержимого каталога. */
    dir = opendir(dir_path);
    /* просматриваем все элементы каталога. */
    while ((entry = readdir(dir)) != NULL) {
        const char* type;
        /* Формируем полное путьевое имя элемента каталога. */
        strncpy(entry_path + path_len, entry->d_name,

```

```

sizeof(entry_path) path_len);
/* Определяем тип элемента. */
type = get_file_type(entry_path);
/* Отображаем собранную информацию. */
printf("%-18s: %s\n", type, entry_path);
}
/* Конец работы. */
closedir(dir);
return 0;
}

```

Приведем несколько строк листинга полученного в каталоге /dev (в разных системах могут быть выданы разные результаты)

```

% ./listdir /dev
directory : /dev/.
directory : /dev/..
socket : /dev/log
character device : /dev/null
regular file : /dev/MAKEDEV
fifo : /dev/initctl
character device : /dev/agpgart
...

```

Для проверки этих данных можно воспользоваться командой ls. Флаг -U отменяет сортировку списка, а флаг -a заставляет включить в список записи текущего (.) и родительского (..) каталогов.

```

% ls -lua /dev total 124
drwxr-xr-x 7 root root 36864 Feb 1 15:14 .
drwxr-xr-x 22 root root 4096 Oct 11 16:39 ..
srw-rw-rw- 1 root root 0 Dec 18 01:31 log
crw-rw-rw- 1 root root 1, 3 May 5 1998 null
-rwxr-xr-x 1 root root 26689 Mar 2 2000 MAKEDEV
prw----- 1 root root 0 Dec 11 18:37 initctl
crw-rw-r-- 1 root root 10, 175 Feb 3 2000 agpgart

```

Первый символ каждой строки определяет тип элемента каталога.

# Приложение В

## Таблица сигналов

В табл. В.1 перечислены сигналы, которые чаще всего приходится обрабатывать в Linux-программах. Некоторые сигналы имеют разные интерпретации в зависимости от того, где они были получены.

Указанные имена сигналов определены в виде макроконстант препроцессора. Чтобы иметь возможность сослаться на них в программе необходимо подключить файл `<signal.h>`. Реальное определение сигналов дано в файле `/usr/sys/unistd.h`, который подключается к файлу `<signal.h>`.

Для получения полного списка сигналов, поддерживаемых в Linux, необходимо выполнить такую команду:

```
% man 7 signal
```

**Таблица В.1. Сигналы ОС Linux**

Название	Описание
----------	----------

SIGHUP	Linux посылает этот сигнал, когда происходит отключение от терминала. Многие программы применяют этот сигнал в совершенно иных целях: он служит указанием программе повторно прочитать свой файл конфигурации
SIGINT	Linux посылает процессу этот сигнал, когда пользователь пытается завершить процесс нажатием клавиш <code>&lt;Ctrl+C&gt;</code>
SIGILL	Процесс получает этот сигнал при попытке выполнить недопустимую инструкцию. Это может означать повреждение стека программы
SIGABRT	Этот сигнал посылается функцией <code>abort()</code>
SIGFPE	По течение этого сигнала означает, что процесс выполнил недопустимую операцию с плавающей запятой. В зависимости от конфигурации центрального процессора результатом операции может стать специальное нечисловое значение, например <code>inf</code> (бесконечность) или <code>NaN</code> (не число), а не сигнал SIGFPE
SIGKILL	Этот сигнал приводит к немедленному завершению процесса и не может быть перехвачен
SIGUSR1	Этот сигнал зарезервирован для прикладного использования
SIGUSR2	Этот сигнал зарезервирован для прикладного использования
SIGSEGV	Этот сигнал означает, что программа выполнила недопустимое обращение к памяти. Возможно, указанный адрес находится за пределами адресного пространства процесса или процессу запрещен доступ к этому участку памяти
SIGPIPE	Этот сигнал означает, что программа обратилась к разрушенному потоку данных, например к сокету, который был закрыт на противоположной стороне
SIGALRM	Доставка этого сигнала планируется функциями <code>alarm()</code> и <code>setitimer()</code> (см. раздел 8.13 "Функция <code>setitimer()</code> : задание интервальных таймеров")
SIGTERM	Этот сигнал является запросом на завершение процесса и посылается командой <code>kill</code> по умолчанию
SIGCHLD	Linux посылает процессу этот сигнал при завершении одного из дочерних процессов (см. раздел 3.4.4, "Асинхронное удаление дочерних процессов") Linux посылает процессу этот сигнал в случае превышения разрешенного времени

`SIGXCPU`    доступа к центральному процессору (см. раздел8.5, "Функции `getrlimit()` и `setrlimit()`: лимиты ресурсов")

`SIGVTALRM` Доставка этого сигнала планируется функцией `setitimer( )` (см. раздел8.13, "Функция `setitimer()`: задание интервальных таймеров")

# Приложение Г

## Internet-ресурсы

В этом приложении перечислен ряд Web-узлов, где можно найти информацию о программировании Linux-систем.

### Г.1. Общая информация

■<http://www.advancedlinuxprogramming.com>. Это Web-узел данной книги. Здесь можно загрузить текст книги в электронном виде вместе с исходными текстами программ, найти ссылки на другие ресурсы и получить дополнительную информацию о программировании в Linux.

■<http://www.linuxdoc.org>. Это Web-узел проекта Linux Documentation Project. Здесь находится хранилище всевозможной документации, а также FAQ-архивов.

### Г.2. Информация о программном обеспечении GNU/Linux

■<http://www.gnu.org>. Это Web-узел проекта GNU Project. Здесь можно загрузить всевозможные бесплатно распространяемые программы. Среди них и GNU-библиотека языка C, содержащая многие из описанных в данной книге функций. Здесь же приведена информация о том, как внести свой вклад в развитие системы GNU/Linux, написав программный код и документацию либо используя бесплатное программное обеспечение.

■<http://www.kernel.org>. Это основной Web-узел для распространения исходных кодов ядра Linux и лучшее место для поиска ответов на самые сложные вопросы о том, как работает Linux. В разделе "Documentation" приведена информация о структуре ядра системы.

■<http://www.linuxhq.com>. Здесь также распространяются исходные коды ядра Linux наряду с "заплатами" и прочей информацией.

■<http://gcc.gnu.org>. Это Web-узел коллекции GNU-компиляторов (GCC). В нее входят компиляторы языков C, C++, Objective C, Java, Chill и Fortran.

■<http://www.gnome.org> и <http://www.kde.org>. Это Web-узлы двух наиболее популярных графических оболочек Linux: Gnome и KDE. Они понадобятся тем, кому необходимо разрабатывать приложения с пользовательским интерфейсом.

### Г.3. Другие ресурсы

■<http://developer.intel.com>. Здесь содержится информация о процессорах Intel, включая архитектуру x86 (IA32). Отметим очень полезные справочники встроенных ассемблерных инструкций.

■<http://www.amd.com>. Здесь представлена аналогичная информация о процессорах AMD.

■<http://freshmeat.net>. Здесь находится список программ с открытыми кодами, в основном для платформы GNU/Linux. Это одно из лучших мест, где можно оперативно узнавать о программных новинках для Linux, начиная от базовых системных компонентов и заканчивая специализированными приложениями.

■<http://www.linuxsecurity.com>. Здесь содержится информация о программах, обеспечивающих безопасность Linux-систем. Этот Web-узел будет интересен пользователям,



системным администраторам и разработчикам.

# Приложение Д

## Лицензия на публикацию программ с открытыми кодами, версия 1.0

### I. Требования к модифицированной и немодифицированной версиям

Материалы, для которых действует лицензия на публикацию программ с открытыми кодами (далее "Лицензия"), могут публиковаться и распространяться как целиком, так и по частям, в любой среде, физической или электронной, при условии соблюдения требований Лицензии и включения самой Лицензии либо ссылки на нее (с любыми дополнительными ограничениями, выбранными автором/авторами и/или издателем) в публикацию.

Правильный способ включения ссылки таков:

<имя автора или правообладателя: <год>. Представленные материалы могут распространяться только на условиях Лицензии на публикацию программ с открытыми кодами, версии X.Y или более поздней (самая последняя версия в настоящий момент доступна по адресу <http://www.opencontent.org/openpub/>).

Сразу за ссылкой могут быть указаны дополнительные ограничения, выбранные автором/авторами или издателем документа (см раздел VI. "Предусмотренные ограничения").

Коммерческое воспроизведение материалов, для которых действует Лицензия, запрещено.

Любая публикация в стандартном (книжном) виде должна сопровождаться ссылкой на исходного издателя или автора материалов. Имя издателя или автора должно указываться на всех сторонах обложки книги. На любой из сторон обложки имя исходного издателя должно быть набрано таким же шрифтом, как и название книги, и стоять в притяжательном падеже по отношению к названию книги.

### II. Авторские права

Право на каждую копию материала, распространяемого на условиях Лицензии, принадлежит автору/авторам либо лицу, которое указано как таковое.

### III. Область действия Лицензии

Следующие условия применимы ко всем материалам, распространяемым на условиях Лицензии, если в документе явно не указано обратное.

Простое объединение материалов, распространяемых на условиях Лицензии, или части таких материалов с другими материалами или программами на одном носителе не означает, что Лицензия переносится и на эти дополнительные материалы. Объединенный продукт должен содержать примечание, которое указывает на включение материалов, распространяемых на условиях Лицензии, а также соответствующее примечание, касающееся авторских прав.

■ **Частичное нарушение лицензии.** Если какая-либо часть Лицензии оказывается неприменимой в той или иной правовой сфере, оставшаяся часть Лицензии остается в силе.

■ **Отсутствие гарантии.** Материалы, для которых действует Лицензия, предоставляются "как есть" без какой-либо гарантии явной или подразумеваемой, в том числе (но не только)

подразумеваемой гарантии годности к продаже и пригодности к конкретному применению, а также гарантии на ненарушение прав.

## **IV. Требования к модифицированным материалам**

Все модифицированные версии документов подпадающих под действие Лицензии, включая переводы, сборники, компиляции и частичные публикации, должны соответствовать следующим требованиям.

- 1.Модифицированная версия должна быть помечена как таковая.
- 2.Должно быть указано лицо, вносящее модификации, а также дата модификации.
- 3.Ссылка на исходного автора и издателя, если имеется, должна быть сохранена в соответствии с обычными правилами цитирования академической литературы.
- 4.Должно быть указано местоположение исходной, немодифицированной версии документа.
- 5.Имя (имена) исходного автора (авторов) нельзя использовать для указания авторства полученного документа без разрешения исходного автора (авторов).

## **V. Рекомендации**

Помимо требований, выдвигаемых в Лицензии, распространителю предлагается и настоятельно рекомендуется следующее.

1.Если вы собираетесь распространять материалы, для которых действует Лицензия, в печатном виде или на компакт-диске, сообщите авторам по электронной почте о намерении распространять материалы за 30 дней до того, как манускрипт или носитель будут изготовлены, чтобы дать авторам возможность предоставить обновленные версии документов. В сообщении должны быть описаны вносимые в документ изменения, если таковые имеются.

2.Все существенные изменения (включая удаления) должны быть либо явно помечены в документе, либо иным образом описаны в приложении к документу

3.Наконец, хоть это и не обязательно, считается хорошим тоном предоставление бесплатного экземпляра печатной копии или компакт-диска с опубликованным материалом его автору (авторам).

## **VI. Предусмотренные ограничения**

Автор (авторы) или издатель документа, распространяемого на условиях Лицензии, может устанавливать определенные ограничения, добавляя соответствующую формулировку к копии Лицензии или ссылке на нее. Эти ограничения считаются частью экземпляра Лицензии и должны включаться в Лицензию (или ссылку на нее) в производных материалах-

А.Можно запретить распространение существенно модифицированных версий без явного разрешения автора (авторов). "Существенная модификация" определяется как изменение семантического содержания документа и не подразумевает изменение формата или типографские изменения.

Чтобы добиться этого, необходимо включить фразу "Распространение существенно модифицированных версий данного документа запрещено без явного разрешения владельца авторских прав" в копию Лицензии или ссылку на нее.

Б.Можно запретить публикацию данного материала или его производных целиком или по

частям в стандартном (печатном) виде а коммерческих целях без получения соответствующего разрешения у владельца авторских прав.

Чтобы добиться этого, необходимо включить фразу "Распространение данного материала или его производных целиком или по частям в стандартном (печатном) виде запрещено без получения соответствующего разрешения у владельца авторских прав" в копию Лицензии или ссылку на нее

## Дополнение, касающееся политики публикации

(Не является частью Лицензии.)

Материалы, распространяемые на условиях Лицензии, доступны в исходном формате на Web-узле <http://works.opencontent.org>.

Авторы, которые хотят сопроводить своей собственной лицензией материалы, распространяемые на условиях Лицензии, могут это сделать при условии, что требования новой лицензии не являются более жесткими, чем существующие.

Если у вас есть вопросы по Лицензии, обращайтесь к Дэвиду Уайли (David Wiley) или в список рассылки Open Publication Authors ([opal@opencontent.org](mailto:opal@opencontent.org)) по электронной почте.

Чтобы подписаться на список рассылки Open Publication Authors, пошлите электронное сообщение по адресу [opal-request@opencontent.org](mailto:opal-request@opencontent.org) со словом "subscribe" в теле сообщения.

Чтобы опубликовать сообщение в списке рассылки Open Publication Authors, пошлите электронное сообщение по адресу [opal@opencontent.org](mailto:opal@opencontent.org) или просто ответьте на предыдущее сообщение.

Чтобы отменить подписку на список рассылки Open Publication Authors, пошлите электронное сообщение по адресу [opal-request@opencontent.org](mailto:opal-request@opencontent.org) со словом "unsubscribe" в теле сообщения.

# Приложение Е

## Общая лицензия GNU<sup>[42]</sup>

Версия 2, июнь 1991 года

Copyright 1989, 1991 Free Software Foundation Inc.

59 Temple Place - Suite 330, Boston, MA 02111-1307 USA

Разрешается копирование и распространение копии этого документа, но запрещается внесение каких бы то ни было изменений в его текст.

### Преамбула

Для большинства программных продуктов лицензии разрабатываются с целью запрещения их свободного распространения и внесения в них изменений. Данная общая лицензия GNU (GNU General Public License), наоборот, призвана гарантировать всем пользователям свободное обладание и изменение свободно распространяемых программных продуктов. Эта лицензия применяется к большинству программных продуктов организации FSF и любым другим программам, авторы которых берут на себя обязательство придерживаться ее. (Существует также ряд программных продуктов FSF, которые подчиняются правилам "библиотечной" лицензии GNU (GNU Library General Public License).) Вы можете применять ее к своим собственным программам.

Когда мы говорим о свободно распространяемом программном продукте, мы имеем в виду свободу, а не цену. Наши общие лицензии разрабатываются для того, чтобы предоставить вам свободу в распространении копий свободно распространяемых программных продуктов (при желании вы можете назначить цену за этот сервис); чтобы предоставить вам гарантию или возможность получения исходных кодов; чтобы вы могли вносить изменения в продукт или использовать его фрагменты для создания новых свободно распространяемых программ; а также для того, чтобы вы знали о возможности реализации всех перечисленных выше действий.

Для защиты ваших прав нам необходимо очертить круг ограничений, которые запрещают кому бы то ни было отказать вам в этих правах или просить вас отказаться от своих прав. Эти ограничения налагают на вас определенные обязательства, если вы распространяете копии некоторой программы или модифицируете ее.

Например, при распространении копии такой программы (бесплатно или за некоторую плату) вы должны передать получателям все права, которыми обладаете сами. Вы должны убедиться в том, что они также получают или могут получить исходный код. Кроме того, вы должны показать им текст данной лицензии, чтобы они тоже знали свои права.

Мы защищаем ваши права двумя способами: обеспечивая авторское право на программный продукт и предлагая вам эту лицензию, которая дает вам легальное разрешение на копирование, распространение и/или модификацию программы.

Кроме того, в целях защиты всех авторов (и нас в том числе) мы хотим, чтобы все четко понимали, что свободно распространяемый программный продукт свободен от каких бы то ни было гарантий. Иными словами, если программа модифицируется другими людьми и передается для последующих модификаций, то те, кому она попадает в руки, должны знать, что они получили не оригинал. Поэтому любые изменения, внесенные другими пользователями, не должны отразиться на репутации авторов

И последнее. Любая свободно распространяемая программа постоянно находится под угрозой получения патента. Мы хотим избежать ситуации, при которой распространители могли

бы приобрести лицензию на такую программу, став ее владельцем. Во избежание этого мы подчеркиваем, что любой патент должен быть лицензирован для беспрепятственного использования всеми желающими иди не лицензирован вообще.

Ниже приводится подробное описание условий для копирования, распространения и модификации программных продуктов.

## **Условия копирования, распространения и модификации программных продуктов**

0. Данная лицензия применяется к любой программе или другому продукту, который содержит замечание, внесенное владельцем авторских прав, где указано, что данный продукт может распространяться только на условиях общей лицензии GNU. Термин "Программа" относится к любой такой программе или продукту, а фраза "продукт, основанный на Программе" означает либо программу, либо любой производный продукт, для которого соблюдается авторское право, т.е. продукт, содержащий Программу или ее часть, в неизменном виде либо с модификациями и/или в переводе на другие языки (в дальнейшем возможность перевода подразумевается в термине "модификация"). Ко всем владельцам лицензий используется обращение во втором лице: вы, вам, ваш и т.д.

Данная лицензия не распространяется на действия, отличные от копирования, распространения и модификации. На запуск программы ограничения не накладываются, а результатов ее работы они касаются только в том случае, если эти результаты представляют собой продукт, основанный на Программе, Справедливость этого положения зависит от того, что делает Программа.

1. Вы можете копировать или распространять точные копии полученного вами исходного кода программы на любом носителе с нанесением на каждую копию соответствующего замечания об авторских правах и отказе от предоставления гарантии при условии, что вы сохраните в точности все замечания, в которых дается ссылка на эту лицензию и на отсутствие гарантии, а также передадите другим лицам копию этой лицензии вместе с Программой.

Вы можете назначить цену за услугу по передаче копии, а также на ваше усмотрение предложить гарантийную защиту в обмен на определенную сумму.

Вы имеете право модифицировать свою копию (копии) Программы или любую ее часть, создавая таким образом продукт, основанный на Программе, а также копировать и распространять такие модификации или продукт при соблюдении условий п. 1, выполняя при этом следующие требования.

а) Вы должны внести в модифицированные файлы замечания, которые нельзя не заметить и в которых сообщается о том, что вы изменили файлы, с обязательным указанием даты каждого изменения.

б) Согласно условиям лицензии, вы должны бесплатно лицензировать любой продукт, который вы распространяете или публикуете и который содержит Программу целиком или частично либо является продуктом, производным от Программы или от одной из ее частей.

с) Если при работе модифицированной программы обычно выполняется чтение команд в интерактивной режиме, вы должны сразу после запуска программы распечатать или отобразить на экране объявление, содержащее соответствующее замечание об авторских правах и отсутствии гарантии (или другое сообщение, если вы предоставляете какую-то гарантию) извещение о том, что пользователи могут распространять данную программу при соблюдении этих условий, а также указание, разъясняющее, каким образом пользователь может получить

копию данной лицензии. (Исключение: если Программа сама по себе работает в интерактивном режиме, но обычно не выводит подобные объявления, ваш продукт, основанный на Программе, не должен в обязательном порядке выводить указанное объявление.)

Эти требования относятся в целом к модифицированному продукту. Если в этом продукте есть разделы которые не являются производными от Программы и могут рассматриваться как независимые продукты работающие отдельно от других частей, то данная лицензия и ее условия не относятся к таким разделам, если вы распространяете их как отдельные продукты. Но если вы распространяете те же самые разделы в качестве составных частей одного целого которые образуют продукт, основанный на Программе, распространение такого "целого" должно происходить на условиях данной лицензии, ограничения которой в этом случае расширяются на все составные части продукта, независимо от того, кто является их автором.

Цель этого раздела лицензии состоит не в том, чтобы оспаривать ваши права на продукт, полностью написанный вашими руками, а в заявлении права на осуществление контроля над распространением унаследованных или коллективных продуктов, основанных на Программе.

Необходимо также отметить, что простое объединение других продуктов, не основанных на Программе, с Программой (или с продуктами, основанными на Программе) на одном носителе информации или распространение такого носителя не вносит другие продукты в область действия данной лицензии.

3)Вы можете копировать или распространять Программу (или продукт, основанный на Программе, при соблюдении требований п. 2) в объектном коде или исполняемой форме при соблюдении условий пп. 1 и 2, следуя одному из нижеперечисленных требований.

а)Приложите к Программе соответствующий исходный код, который должен распространяться при соблюдении условий пп. 1 и 2 на носителях, обычно используемых для обмена программными продуктами.

б)Приложите к Программе записанное предложение (действительное в течение по крайней мере трех лет) передать любым независимым исполнителям за плату, не превышающую стоимость физического выполнения операции доставки, полной машинной копии соответствующего исходного кода, распространяемого при соблюдении условий пп. 1 и 2 на носителях, обычно используемых для обмена программными продуктами.

с)Приложите к Программе информацию, полученную вами в качестве предложения распространять соответствующий исходный код. (Эта альтернатива разрешена только для некоммерческой поставки и только в том случае, если вы получили программу в объектном коде или в исполняемой форме с таким предложением в соответствии со вторым подпунктом.)

Исходный код программы является предпочтительной формой для внесения изменений. Для исполняемого продукта полный исходный код означает наличие исходного кода всех модулей, содержащихся в продукте, плюс соответствующие интерфейсные файлы и сценарии, используемые для управления компиляцией и установкой исполняемого файла. Но в виде исключения поставляемый исходный код не требует включения никаких компонентов, которые обычно распространяются (в исходном либо двоичном коде) с основными компонентами (компилятор, ядро и т.д.) операционной системы, где должен работать продукт, если только какой-то компонент сам не является обязательным приложением к исполняемому продукту. Если поставка исполняемого или объектного кода выполнена в виде предложения обратиться к копии в обозначенном месте то предложение эквивалентного доступа к копии исходного кода из того же самого источника расценивается как поставка исходного кода, даже если получателя не заставляют копировать исходный код вместе с объектным кодом.

4.Вы не можете копировать, модифицировать, лицензировать или распространять Программу в обход этой лицензии. Любые попытки поступить иначе с целью копирования,



модификации, лицензирования или распространения Программы пресекаются с автоматическим аннулированием ваших прав, предусматриваемых лицензией. Но лица, получившие от вас копии или права в соответствии с лицензией, не лишаются своих лицензий до тех пор, пока они полностью соблюдают оговоренные лицензией условия.

5. От вас не требуется принимать условия лицензии, пока вы не подпишете ее. Но имейте в виду, что ничто иное не даст вам разрешение на модификацию или распространение Программы или ее производных продуктов. Эти действия запрещены законом, если вы не примете данную лицензию. Следовательно, модифицируя или распространяя Программу (или любой продукт, основанный на Программе), вы гарантируете свое принятие этой лицензии и всех ее условий и требований, которые необходимо соблюдать при копировании, распространении или модификации Программы или продуктов, основанных на ней.

6. Каждый раз, когда вы передаете Программу (или любой продукт, основанный на Программе) третьему лицу, оно автоматически получает от вас лицензию на право копировать, распространять или модифицировать Программу на указанных условиях. Вы не можете налагать какие бы то ни было дальнейшие ограничения на реализацию прав, данных получателю. Вы не несете ответственность за соблюдение другими лицами условий данной лицензии.

7. Если вследствие решения суда, или заявления о нарушении прав, или по какой-то другой причине (не связанной с вопросами авторских прав) поставленные перед вами условия (по постановлению суда, соглашению и т.п.) противоречат условиям данной лицензии, это не освобождает вас от ответственности за несоблюдение правил, предусматриваемых лицензией. Если вы не можете распространять Программу так, чтобы одновременно удовлетворять условиям, предусмотренным этой лицензией, и соблюдать другие обязательства, значит, вы вообще не можете заниматься распространением Программы. Например, если согласно некоторой лицензии не разрешается безгонорарное распространение Программы всеми, кто прямо или косвенно получает от вас копии, то единственный способ удовлетворить условия обеих лицензий заключается в полном отказе от распространения Программы. Если какая-нибудь часть этого раздела не согласуется или не соблюдается при определенных обстоятельствах, то предполагается, что весь раздел в целом применяется при других обстоятельствах.

Целью этого раздела отнюдь не является склонять вас к нарушению других правовых обязательств или оспаривать их обоснованность. Единственная цель этого раздела состоит в защите целостности системы распространения бесплатных программных продуктов, которая реализуется с помощью общих лицензий. Многие люди внесли огромный вклад в программное обеспечение, распространяемое через эту систему, будучи уверенными в ее постоянном применении. И только от решения самого автора зависит, будет ли он (или она) распространять программный продукт через какую-либо другую систему, и лицензия не может повлиять на этот выбор.

Предполагается, что в этом разделе внесена ясность в то, что считается следствием остальной части лицензии.

8. Если распространение и/или использование Программы ограничивается в определенных странах либо патентами, либо интерфейсами защищенными авторскими правами, исходный владелец авторских прав, который вводит свою Программу под защиту этой лицензии может добавить в явном виде ограничение на географическое распространение: перечислив страны, исключаемые из области распространения (это значит, что распространение разрешено только среди тех стран, которые не входят в этот список). В этом случае лицензия включает ограничение, как если бы оно было записано в теле самой лицензии.

9. Организация FSF время от времени может публиковать модифицированную и/или новую

версию общей лицензии GNU. Новые версии будут содержать ту же идею, что и настоящая версия, но могут отличаться в деталях, связанных с новыми проблемами или концепциями.

Каждой версии присваивается отличительный номер. Если в Программе указан номер версии лицензии, который применяется к этой и "любым последующим версиям", у вас есть возможность следовать требованиям либо данной версии, либо любой из последующих версий, опубликованных организацией FSF. Если в Программе не указан номер версии лицензии, вы можете выбрать любую версию, когда-либо опубликованную организацией FSF.

10.Если вы хотите объединить части Программы с другими свободно распространяемыми программами, условия распространения которых отличаются от описываемых, обратитесь к автору с просьбой о разрешении. Относительно программных продуктов, которые защищаются авторскими правами FSF, обращайтесь непосредственно в организацию FSF. Иногда мы делаем исключения. На наше решение влияет желание достичь двух целей: сохранить свободный статус всех продуктов, производных от наших свободно распространяемых программ, и продвинуть идеи совместного применения и многократного использования программ.

### ***Гарантия отсутствует***

11.Поскольку программа лицензируется бесплатно, для нее не существует никаких гарантий (до степени, разрешенной действующим законодательством). За исключением случаев, специально оговоренных в письменном виде, владельцы авторских прав и/или другие лица предоставляют программу "как есть" без какой-либо гарантии, явной или подразумеваемой, в том числе (но не только) подразумеваемой гарантии годности к продаже и пригодности к конкретному применению. Весь риск, связанный с качеством и выполнением программы, ложится на вас. В случае дефектности программы вам следует взять на себя стоимость всех необходимых доработок, поиска неисправностей и корректировки.

12.Ни при каких обстоятельствах, если того не требует закон или не указано в письменной форме, владелец авторских прав либо какое-то другое лицо, которое имело право модифицировать и/или распространять программу в соответствии с приведенными выше разрешениями, не несет перед вами ответственность за нанесенный программой ущерб, включающий любые повреждения общего, специального, случайного или косвенного характера, являющиеся следствием использования или невозможности использования программы в том числе, но не только: потеря данных вами или другими лицами, неправильное представление данных или неспособность программы работать совместно с другими программами), даже если владелец или другое лицо было уведомлено о возможности подобного ущерба.

### **Конец условий**

### **Как применить эти требования к новым программным продуктам**

Если вы написали новую программу и хотите чтобы любой человек смог ею свободно воспользоваться, лучше всего присвоить ей статус бесплатно распространяемого программного продукта. Тогда любой желающий в рамках приведенной выше лицензии сможет ее свободно распространять и модифицировать.

Для этого включите приведенные ниже строки в дистрибутивный пакет программы. Лучше

всего поместить их в качестве комментария в начало каждого исходного файла, чтобы были хорошо заметны фразы об отсутствии гарантий. В каждый исходный файл должна быть, как минимум, включена строка "copyright", а также ссылка на то, где можно прочитать полный текст лицензионного соглашения.

*В первой строке необходимо указать фамилию автора программы и идею ее создания.*

*Copyright год создания, фамилия автора*

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details. You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston. MA 02111-1307, USA

Кроме того, обязательно поместите в дистрибутивный пакет свои координаты: почтовый адрес и/или адрес электронной почты.

Если программа работает в интерактивном режиме сделайте так, чтобы при запуске выводилось короткое сообщение наподобие приведенного ниже.

*Gnomovision version 69, Copyright год создания, фамилия автора*

Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

Выше были указаны две гипотетические команды 'show w' и 'show c', с помощью которых пользователь может просмотреть соответствующие разделы общей лицензии GNU. Конечно, имена команд могут быть другими. Более того, эти команды могут вызываться из меню или в результате щелчка мышью это зависит от типа вашей программы.

При необходимости укажите также координаты работодателя (если вы работаете программистом) или учебного заведения (если вы студент), которые смогут подтвердить полный отказ от всех авторских прав на программу. Ниже приведен пример, где можно вставить в текст реальные имена

Yoyodyne. Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (which makes passes at compilers) written by James Hacker.

*Личная подпись, дата*

*Должность*

Данное лицензионное соглашение не предусматривает использование программы или ее частей в коммерческих проектах. Если ваше программное обеспечение представляет собой библиотеку подпрограмм, вы можете разрешить подключать ее компоненты к коммерческим программам. При этом вместо общей лицензии GNU используйте библиотечную лицензию.

Вопросы, касающиеся FSF и GNU, направляйте по адресу [gnu@gnu.org](mailto:gnu@gnu.org).

Комментарии к данному тексту посылайте по адресу [webmasters@www.gnu.org](mailto:webmasters@www.gnu.org).

Сообщение об авторских правах приведено выше

Free Software Foundation Inc. 59 Temple Place - Suite 530 Boston, MA 02111-1307, USA

Обновлено: 31 июня 2000 г.

Данная книга была скачана с сайта [Librs.net](#).

---

**notes**

# Примечания

GNU это рекурсивный акроним, который расшифровывается как GNU's Not UNIX (GNU это не UNIX).

Пользователи, не работающие в X Window, должны нажать <F10>, чтобы получить доступ к меню.



Введите команду M-x dunnet, если хотите поиграть в старомодную приключенческую игру.

Дополнительную информацию о GCC можно получить по адресу <http://gcc.gnu.org>.

В Windows исполняемые файлы обычно имеют расширение `.exe`, а в Linux - вообще не имеют его. Поэтому в Windows эквивалент данной программы будет, скорее всего, называться `reciprocal.exe`.

В C++ аналогичное различие существует между потоками `cout` и `cerr`. Манипулятор `endl` добавляет в конец потока символ новой строки и вызывает "выталкивание" буфера. Если состояние буфера временно менять не нужно (из соображений производительности, например), воспользуйтесь вместо манипулятора константой `'\n'`.

В целях обеспечения безопасной работы потоков переменная `errno` реализована в виде макроса, но к ней можно обращаться как к глобальной переменной.

Имеются и другие флаги, предназначенные для удаления файлов из архива и выполнения других операций над ним. Все они описаны на map-странице, посвященной команде `ar`.

Иногда в документации упоминается переменная `LD_RUN_PATH`. Не верьте прочитанному! Данная переменная никак не используется в Linux.

Команда `kill` позволяет посылать процессам и другие сигналы. Об этом рассказывается в разделе 3.4, "Завершение процесса".



Способ синхронизации двух процессов представлен в разделе. 3 4.1, "Ожидание завершения процесса".

В чём между ними разница! Сигнал SIGTERM является запросом на завершение; процесс может его проигнорировать и продолжить свое выполнение.. Сигнал SIGKILL вызывает немедленное безусловное уничтожение процесса и не может быть обработан.

Данный способ не является стандартным. В обязанности программиста входит убедиться, что в процессе подобных преобразований не произойдет потеря значащих разрядов.

Ненулевое значение определяет семафор, совместно используемый несколькими процессами, но в Linux такой вариант семафоров не поддерживается (семафоры процессов создаются по-другому, а в данном случае речь идет о потоковых семафорах).

Эти же константы используются при работе с файлами. Они описываются в разделе 10.3. "Права доступа к файлам".

Команда `sort` читает строки текста из стандартного входного потока, сортирует их в алфавитном порядке и записывает в стандартный выходной поток.

Именованный канал можно создать только в Windows NT. В Windows 9x программы могут устанавливать только клиентские соединения.

Принтер может требовать, чтобы в конце каждой строки стоял символ возврата каретки (ASCII-код 14), а в конце каждой страницы символ подачи листа (ASCII-код 12).



В большинстве Linux-систем можно переключиться на первый виртуальный терминал, нажав <Ctrl+Alt+F1>. Чтобы перейти на второй виртуальный терминал, следует нажать <Ctrl+Alt+F2> и т.д.

Мы могли бы использовать программу `hexdump` вместо команды `od`, так как они делают, по сути, одно и то же. Но когда входной поток исчерпывается, программа `hexdump` завершается, а команда `od` переходит в режим ожидания. Опция `-t x1` сообщает команде `od` о том, что содержимое файла должно отображаться в шестнадцатеричном формате.

В случае повреждения файловой системы данные, которые были восстановлены, но не связаны с каким-либо файлом, помещаются в каталог `lost+found`.

В некоторых UNIX-системах эти идентификаторы дополняются нулями, в Linux нет.

В главе 9, "Встроенный ассемблерный код", рассказывается о том, как задействовать ассемблерные инструкции в Linux-программах.

В DOS и Windows нумерация последовательных портов начинается с единицы, поэтому порту COM1 соответствует последовательный порт с номером 0 в Linux.

Если ядро Linux сконфигурировано правильно, оно поддерживает дополнительные IDE-контроллеры, нумеруемые последовательно начиная от `ide2`.

Команда `hostname`, вызванная без флагов, отображает имя компьютера.



В Linux семейство функции `exec ( )` реализовано на основе системного вызова `execve ( )`.

NFS (Network File System) популярная технология совместного использования файлов в сети.

Режим копирования при записи означает, что Linux создает для процесса частную копию страницы только тогда, когда процесс записывает в нее какие-то данные.

Выражение `sin(angle)` обычно преобразуется в вызов функции библиотеки `libm`, но если задать флаг `-O1` (или включить более сильную оптимизацию), компилятор `gcc` заменит вызов функции простой ассемблерной инструкцией `fsin`.

Тот факт, что в системе есть всего один специальный пользователь, послужил причиной появления названия UNIX. Более ранняя операционная система, в которой было несколько специальных пользователей, называлась MULTICS.

Ядро может отклонить запрос на доступ к файлу, если один из каталогов на пути к нему недоступен данному пользователю. Например, если процессу не разрешено обращаться к каталогу `/tmp/private`, то он не сможет получить доступ к файлу `/tmp/private/data`.

Название бита является анахронизмом. Оно возникло в те далекие времена, когда наличие этого бита означало запрет на выгрузку программы из памяти по окончании выполнения.

Существует также бит смены идентификатора группы (SGID, set group identifier). Программа с установленным битом SGID при запуске примет эффективный идентификатор группы, которой принадлежит файл.



Монтирование данного каталога по сети ошибка системного администратора.

Наиболее популярный Web-сервер с открытым кодом сервер Apache (доступен на Web-узле [www.apache.org](http://www.apache.org)).

В системе могут присутствовать такие интерфейсы, как `eth0` (Ethernet-плата), `lo` (интерфейс обратной связи), `ppp0` (коммутируемое соединение).

При необходимости функция `gethostbyname()` осуществляет поиск имен в DNS.

Эта утилита входит в состав Linux.

В постфиксной записи бинарный оператор ставится после операндов, а не между ними. Например, чтобы умножить 6 на 8, нужно записать  $6\ 8\ *$ . Чтобы умножить 6 на 8, а затем добавить 5, следует записать  $6\ 8\ * 5\ +$ .

В стандартной библиотеке языка C++ аналогичным целям служат *потоки ввода-вывода*.

Исходный текст данной лицензии можно найти по адресу  
<http://www.gnu.org/copyleft/gpl.html>.