



Программирование в стандарте POSIX

Курс лекций. Учебное пособие

В. А. Галатенко

Под редакцией академика РАН В.Б. Бетелина

**Рекомендовано УМО в области прикладной информатики для студентов
высших учебных заведений, обучающихся по специальности 351400
«Прикладная информатика»**

Серия «Основы информационных технологий»

УДК 004.438POSIX(075.8)
Г-15
ББК 32.973-018.1POSIX.я73-2

Г-15 Галатенко Владимир Антонович

Программирование в стандарте POSIX: Курс лекций: Учеб. пособие: Для студентов вузов, обучающихся по специальности 351400 «Прикладная информатика» / В.А. Галатенко; Под ред. В.Б. Бетелина. - М.: Интернет-университет информ. технологий, 2004. - 560 с. - (Серия «Основы информационных технологий»). - ISBN 5-9556-0011-6.

В курсе рассматривается стандарт на мобильный интерфейс операционной системы (POSIX), а также приемы и методы программирования приложений на основе данного стандарта, поясняемые многочисленными примерами. Затрагиваются вопросы программирования многопроцессных систем, взаимодействия приложений в рамках распределенных конфигураций.

Рекомендовано УМО в области прикладной информатики для студентов высших учебных заведений, обучающихся по специальности 351400 «Прикладная информатика»

Библиогр. 15

Издание осуществлено при финансовой и технической поддержке компаний:
Издательство «Открытые Системы», «РМ Телеком», Kraftway Computers и «СДМ-Банк».



**ОТКРЫТЫЕ
СИСТЕМЫ**
Системное Программирование

РМ Телеком



Kraftway



СДМ-БАНК

Курс лекций по информационным технологиям
Интернет-Университета Информационных Технологий
www.intuit.ru

Руководитель проекта — А.В. Шкред

Программирование в стандарте POSIX

Курс лекций. Учебное пособие

Серия «Основы информационных технологий»

Подписано в печать 25.04.2004. Формат 60×90^{1/16}.
Усл. печ. л. 35. Бумага офсетная. Тираж 2000 экз. Заказ № 4004.

ООО «ИНТУИТ.ру» Интернет-Университет Информационных Технологий, www.intuit.ru,
123056, Москва, Электрический пер., 8, стр. 3.

Отпечатано с готовых диапозитивов на ФГУП ордена «Знак Почета» Смоленская областная
типография им. В.И. Смирнова, 214000, г. Смоленск, проспект им. Ю. Гагарина, д. 2.

Полное или частичное воспроизведение или размножение каким-либо способом, в том
числе и публикация в Сети, настоящего издания допускается только с письменного
разрешения Интернет-Университета Информационных Технологий.

(с) Интернет-Университет Информационных Технологий, www.intuit.ru, 2004

ISBN 5-9556-0011-6

О проекте

Интернет-Университет Информационных Технологий — это первое в России высшее учебное заведение, которое предоставляет возможность получить дополнительное образование во Всемирной Сети. Web-сайт университета находится по адресу www.intuit.ru.

Мы рады, что Вы решили расширить свои знания в области компьютерных технологий. Современный мир — это мир компьютеров и информации. Компьютерная индустрия — самый быстрорастущий сектор экономики, и ее рост будет продолжаться еще долгое время. Во времена жесткой конкуренции от уровня развития информационных технологий, достижений научной мысли и перспективных инженерных решений зависит успех не только отдельных людей и компаний, но и целых стран. Вы выбрали самое подходящее время для изучения компьютерных дисциплин. Профессионалы в области информационных технологий сейчас востребованы везде: в науке, экономике, образовании, медицине и других областях, в государственных и частных компаниях, в России и за рубежом. Анализ данных, прогнозы, организация связи, создание программного обеспечения, построение моделей процессов — вот далеко не полный список областей применения знаний для компьютерных специалистов.

Обучение в университете ведется по собственным учебным планам, разработанным ведущими российскими специалистами на основе международных образовательных стандартов Computer Curricula 2001 Computer Science. Изучать учебные курсы можно самостоятельно по учебникам или на сайте Интернет-университета, задания выполняются только на сайте. Для обучения необходимо зарегистрироваться на сайте университета. Удостоверение об окончании учебного курса или специальности выдается при условии выполнения всех заданий к лекциям и успешной сдачи итогового экзамена.

Книга, которую Вы держите в руках, очередная в многотомной серии «Основы информационных технологий», выпускаемой Интернет-Университетом Информационных Технологий. В этой серии будут выпущены учебники по всем базовым областям знаний, связанным с компьютерными дисциплинами.

Добро пожаловать в Интернет-Университет Информационных Технологий!

Анатолий Шкред
anatoli@shkred.ru

Предисловие

Обеспечение мобильности (переносимости, портативности) программного обеспечения (ПО) — задача исключительной важности и сложности; в наше время это обстоятельство едва ли нуждается в пространных обоснованиях. Один из общепринятых способов повышения мобильности ПО — стандартизация окружения приложений: предоставляемых программных интерфейсов, утилит и т.п. На уровне системных сервисов подобное окружение описывает стандарт POSIX (Portable Operating System Interface — мобильный интерфейс операционной системы); название предложил известный специалист, основатель Фонда свободного программного обеспечения Ричард Столмэн.

В курсе рассматривается наиболее современная его версия в редакции 2003 г., которую можно назвать «стандартом втройне», а именно: стандартом IEEE Std 1003.1, Техническим стандартом Open Group и, что для нас важнее всего, международным стандартом ISO/IEC 9945.

Основная задача настоящего курса состоит в осмыслении приемов и методов использования стандартизованных служебных программ и функций. Не ставилась цель пересказать стандарт, осветив все тонкости реализации ОС, все возможные коды ошибок и т.п. Главное, на наш взгляд, — прочувствовать дух стандарта, научиться мобильным образом применять заложенные в нем возможности.

В предположении, что читатель владеет языком C, мы не рассматривали ни его синтаксис, ни хрестоматийные библиотечные функции. Что же касается стандартного командного языка и его интерпретатора, то эта тема изложена довольно подробно, хотя многие практикующие программисты предпочитают пользоваться другими интерпретаторами.

Значительное место — и по объему, и по роли — отведено примерам программ. Многие положения стандарта (связанные, скажем, с обработкой ошибочных ситуаций) излагаются не в основном тексте, а в соответствующих примерах. Последние по возможности компилировались и выполнялись на нескольких аппаратно-программных платформах, в той или иной степени претендующих на соответствие стандарту POSIX. Тем не менее, недосмотры, конечно, возможны. Мы будем признательны за все замечания и предложения, относящиеся как к курсу в целом, так и к отдельным примерам программ.

Об авторе

Галатенко Владимир Антонович

Доктор физико-математических наук, заведующий сектором в отделе информационной безопасности НИИ системных исследований РАН. Автор теоретических и практических разработок, книг, учебных пособий и статей по информационной безопасности и технологии программирования.

Принятые сокращения

АСБ-функция – асинхронно-сигнально-безопасная функция

ОС – операционная система бит

ПДИП – бит переустановки действующего идентификатора пользователя бит

ПДИГ – бит переустановки действующего идентификатора группы сохраненный

ПДП-идентификатор – сохраненный переустановленный действующий идентификатор пользователя сохраненный

ПДГ-идентификатор – сохраненный переустановленный действующий идентификатор группы

Лекции

Лекция 1. Основные понятия и идеи стандарта POSIX	13
Лекция 2. Язык shell	29
Лекция 3. Утилиты и функции, обслуживающие понятие «пользователь»	71
Лекция 4. Организация файловой системы	83
Лекция 5. Файловый ввод/вывод.	131
Лекция 6. Средства обработки структурированных данных.	175
Лекция 7. Процессы	233
Лекция 8. Средства межпроцессного взаимодействия	263
Лекция 9. Общий терминальный интерфейс	333
Лекция 10. Опрос характеристик хостов и их использование в приложениях	355
Лекция 11. Сетевые средства	373
Лекция 12. Время и работа с ним.	441
Лекция 13. Языково-культурная среда	491
Лекция 14. Заключение.	525

Содержание

Лекция 1. Основные понятия и идеи стандарта POSIX	13
История создания и текущий статус стандарта POSIX.....	13
Основные идеи стандарта POSIX	15
Основные понятия стандарта POSIX	16
Основные понятия операционных систем, соответствующих стандарту POSIX.....	19
Среда компиляции POSIX-совместимых приложений	21
Мобильность POSIX-совместимых приложений	22
Лекция 2. Язык shell	29
Основные понятия языка shell	29
Конвейеры и примеры их использования	31
Переменные и аргументы shell-процедур	32
Служебные переменные языка shell	37
Окружение процессов	39
Подстановка результатов выполнения команд	41
Управляющие конструкции	43
Генерация маршрутных имен файлов	48
Перенаправление ввода/вывода	50
Встроенные команды	55
Вызов командного интерпретатора shell	60
Правила формирования и средства разбора командных строк ..	62
Лекция 3. Утилиты и функции, обслуживающие понятие «пользователь»	71
Данные, ассоциированные с пользователем	71
Служебные программы, обслуживающие взаимодействие пользователей	77
Лекция 4. Организация файловой системы	83
Основные понятия	83
Получение информации о файлах и файловых системах	87
Изменение атрибутов файлов и текущей позиции в файловой иерархии	104
Создание, удаление, копирование и перемещение файлов	110
Обход и обработка файловых иерархий	121

Лекция 5. Файловый ввод/вывод.	131
Основные понятия	131
Открытие и закрытие файлов	134
Чтение и запись данных.	141
Управляющие операции с файлами и ассоциированными данными.	155
Лекция 6. Средства обработки структурированных данных.	175
Простейшие средства обработки текстовых файлов.	175
Регулярные выражения	191
Использование регулярных выражений.	165
Аналоги реляционных операций для текстовых файлов	216
Средства обработки каталогов.	221
Лекция 7. Процессы	233
Основные понятия	233
Опрос и изменение атрибутов процессов	237
Создание и завершение процессов.	251
Лекция 8. Средства межпроцессного взаимодействия	263
Каналы	264
Сигналы.	271
Очереди сообщений	289
Семафоры	301
Разделяемые сегменты памяти	322
Лекция 9. Общий терминальный интерфейс	333
Основные понятия и объекты	333
Служебные программы и функции для управления терминалами	339
Лекция 10. Опрос характеристик хостов и их использование в приложениях	355
Опрос идентифицирующих данных хостов	355
Опрос конфигурационных параметров хоста	356
Лекция 11. Сетевые средства	373
Основные понятия и объекты	373
Опрос данных о сети	376
Функции для работы с сокетами.	394
Примеры программ работы с сокетами	405

Лекция 12. Время и работа с ним.	441
Основные понятия и объекты	442
Преобразование данных о времени	453
Опрос показаний часов процессорного времени	459
Приостановка выполнения на заданное время	470
Работа с интервальными таймерами.	475
Лекция 13. Языково-культурная среда	491
Основные понятия и объекты	491
Определение языково-культурной среды.	495
Создание и опрос характеристик языково-культурной среды. . .	504
Средства для работы с каталогами сообщений	519
Лекция 14. Заключение.	525
Основные идеи курса	527
Основные понятия, рассматриваемые в курсе	528
Средства, обслуживающие понятие пользователя	530
Средства, обслуживающие понятие файла	531
Средства обработки структурированных данных	534
Процессы.	535
Средства межпроцессного взаимодействия	537
Общий терминальный интерфейс	540
Опрос характеристик хостов	541
Сетевые средства.	542
Время	545
Языково-культурная среда.	548
Литература	553

Лекция 1. Основные понятия и идеи стандарта POSIX

Излагаются основные сведения о стандарте POSIX, описываются его статус и структура. Выделяются наиболее важные понятия и идеи.

Ключевые слова: мобильность ПО, стандарт, POSIX, системный сервис, прикладной программный интерфейс, командный язык, служебная программа, утилита, мобильность на уровне исходного языка, системный вызов, библиотечная функция, администрирование, суперпользователь, обратная совместимость, средства разработки, сетевые средства, средства реального времени, потоки управления, математические интерфейсы, пакетные сервисы, заголовочные файлы, шифрование, трассировка, соответствие стандарту POSIX, строгое соответствие, профиль, подпрофиль, неспецифицированное поведение, неопределенное поведение, поведение, зависящее от реализации, устаревший, унаследованный, пользователь, файл, процесс, терминал, хост, узел сети, группа, время, межпроцессное взаимодействие, права доступа, соответствующие привилегии, окружение, языково-культурная среда, стандартный ввод, стандартный вывод, стандартный протокол, ОС реального времени, инструментальная платформа, целевая платформа, среда компиляции приложений, заголовочный файл, включаемый файл, прототип функции, символическая константа, макрос, макрос проверки возможностей, имя, префикс имени, порождение процесса, завершение процесса, смена программы процесса, группирование процессов, приоритет процесса, управление заданиями, межпроцессное взаимодействие, сигнал, очередь сообщений, разделяемая память, семафор, управление памятью, блокировка фрагментов файлов, асинхронный ввод/вывод, внешнее устройство, язык shell, конвейер, фильтр, фоновый режим, интерфейс командной строки.

История создания и текущий статус стандарта POSIX

Обеспечение мобильности (переносимости, портативности) программного обеспечения (ПО) – задача исключительной важности и сложности; в наше время это обстоятельство едва ли нуждается в пространственных обоснованиях. Один из общепринятых способов повышения мобильности ПО – стандартизация окружения приложений: предоставляемых программных

интерфейсов, утилит и т.п. На уровне системных сервисов подобное окружение описывает стандарт POSIX (Portable Operating System Interface – мобильный интерфейс операционной системы); название предложено известным специалистом, основателем Фонда свободного программного обеспечения Ричардом Столмэн.

Мы будем рассматривать наиболее современную из доступных версий стандарта POSIX, в редакции 2003 г., которую можно назвать «стандартом втроем», а именно: стандартом IEEE Std 1003.1, Техническим стандартом Open Group и (см. [6], что для нас важнее всего, международным стандартом ISO/IEC 9945 (см. [1 – 4]).

История создания этой версии такова. В начале 1998 г. представители трех организаций – Комитета по стандартам мобильных приложений Института инженеров по электротехнике и электронике, Open Group и рабочей группы 15 подкомитета 22 совместного технического комитета 1 (JTC1/SC22/WG15) Международной организации по стандартизации – начали консультации по вопросу слияния и развития курируемых ими стандартов интерфейсов к системным сервисам: IEEE Std 1003.1, IEEE Std 1003.2, Базовых спецификаций от Open Group, ISO/IEC 9945-1, ISO/IEC 9945-2. В сентябре того же года в городе Остин, штат Техас, в офисе корпорации IBM состоялось организационное заседание группы, сформированной для достижения поставленной цели (см. <http://www.opengroup.org/austin>).

Основополагающим документом для пересмотренного стандарта, первый проект которого был представлен в июле 1999 года, стали Базовые спецификации от Open Group, поскольку они включали положения стандартов IEEE и ISO/IEC. В 2001 году, по завершении подготовительной работы, стандарт содержал следующие четыре части:

- 1) основные определения (термины, концепции и интерфейсы, общие для всех частей);
- 2) описание прикладного программного C-интерфейса к системным сервисам;
- 3) описание интерфейса к системным сервисам на уровне командного языка и служебных программ;
- 4) детальное разъяснение положений стандарта, обоснование принятых решений.

Далее в ISO, IEEE и Open Group с большей или меньшей скоростью (в 2001-2002 гг.) прошло формальное утверждение нового стандарта POSIX. Тем временем накапливались относительно мелкие исправления, учтенные в редакции 2003-го года.

С развитием стандарта расширялась и трактовка термина «POSIX». Первоначально он относился к документу IEEE Std 1003.1-1988, описывавшему прикладной программный интерфейс ОС класса Unix. После

стандартизации интерфейса на уровне командного языка и служебных программ более правильно понимать под словом «POSIX» стандарт в целом, обозначая перечисленные выше части 2 и 3 через POSIX.1 и POSIX.2 в соответствии с нумерацией документов IEEE и ISO/IEC.

Основные идеи стандарта POSIX

Стандарт POSIX описывает множество базовых, системных сервисов, необходимых для функционирования прикладных программ. Доступ к ним предоставляется посредством интерфейса, специфицированного для языка C, командного языка и общеупотребительных служебных программ.

У каждого интерфейса есть две стороны: вызывающая и вызываемая. Стандарт POSIX ориентирован в первую очередь на вызывающую. Его цель — сделать приложения мобильными на уровне исходного языка. Это значит, в частности, что при переносе C-программ на другую операционную платформу потребуется перекомпиляция. О мобильности выполнимых программ и/или объектных файлов речь не идет.

Стандарт POSIX отнюдь не ограничен рамками Unix-среды. Существуют операционные системы (ОС) «не зависимо происхождения» (например, системы реального времени), предоставляющие необходимые сервисы и тем самым поддерживающие выполнение POSIX-совместимых приложений. Можно утверждать, что следование стандарту POSIX облегчает перенос приложений практически на любую сколько-нибудь распространенную операционную платформу. Дополнительные усилия по повышению мобильности, прилагаемые на этапе разработки, безусловно, окупаются.

Определяя интерфейс к системным сервисам, POSIX оставляет за рамками рассмотрения их реализацию. В частности, не различаются системные вызовы и библиотечные функции. Не являются объектом стандартизации средства администрирования, аппаратные ограничения и функции, необходимые только суперпользователю, что еще раз подчеркивает направленность стандарта POSIX на приложения, а не на операционные системы.

POSIX нейтрален по отношению к системной архитектуре и разрядности процессора. Это очень важный аспект мобильности приложений.

Ориентация на международный стандарт языка C определила не только стиль описания функций, но и, до некоторой степени, направление развития спецификаций POSIX в плане синхронизации обоих стандартов. Как известно в утвержденной в 1999 г. редакции спецификаций языка C (см. [5]) узаконен комплексный тип данных, что вызвало соответствующее пополнение POSIX-функций.

В стандарте POSIX проведено разделение на обязательные и дополнительные функции, причем обязательное ядро сделано по возможности компактным. Разумеется, особое внимание уделяется способам реализации стандартизуемых функций как в «классической» Unix-среде, так и на других операционных платформах, в сетевых и распределенных конфигурациях.

Разработчики новой версии стандарта POSIX очень бережно отнеслись и к его предыстории, и к предыстории Unix-систем, и, главное, к приложениям, удовлетворявшим более ранним версиям стандарта. Существующие интерфейсы старались сохранять; в процессе развития соблюдался принцип обратной совместимости; новые интерфейсы добавлялись так, чтобы они не конфликтовали со старыми. Полностью избежать внесения изменений в приложения не удалось по вполне понятным причинам: потребовалось устранить противоречия между разными исходными спецификациями, а также отказаться от поддержки «традиционного» варианта языка C и перейти на его международный стандарт.

Основные понятия стандарта POSIX

Стандарт POSIX в редакции 2003-го года — весьма обширный, многогранный документ, где подробно рассматриваются следующие категории системных компонентов:

- средства разработки;
- сетевые средства;
- средства реального времени;
- потоки управления;
- математические интерфейсы;
- пакетные сервисы;
- заголовочные файлы;
- унаследованные интерфейсы.

Именно такой (на верхнем уровне, далеко не полный) репертуар должна предоставлять операционная система для работы приложения.

Важнейшим является понятие соответствия стандарту POSIX. Мы уже отмечали, что всякий интерфейс располагает двумя сторонами: вызывающей и вызываемой. Две стороны есть и у POSIX-соответствия: соответствие реализации (операционной системы) и приложения.

Реализация (операционная система), соответствующая стандарту POSIX, должна поддерживать все обязательные служебные программы, функции, заголовочные файлы с обеспечением специфицированного в стандарте поведения. Константа `_POSIX_VERSION` имеет значение 200112L.

ОС может предоставлять возможности, помеченные в стандарте в качестве дополнительных, а также содержать нестандартные функции.

Если утверждается, что поддерживается некоторое расширение, это должно производиться непротиворечивым образом, для всех необходимых частей и так, как описано в стандарте.

В заголовочном файле `<unistd.h>` следует определить константы, соответствующие поддерживаемым необязательным возможностям (например, константа `_POSIX2_C_DEV` обслуживает средства разработки на языке C). Анализируя эти константы во время компиляции, приложение выяснит возможности используемой ОС и подстроится под них. Аналогичные действия на этапе выполнения могут быть выполнены с помощью функции `sysconf()` и/или служебной программы `getconf`.

Для минимизации размеров ОС и приложений стандартом POSIX предусмотрена весьма мелкая гранулярность необязательных возможностей (всего их сорок). С другой стороны, проведено объединение взаимосвязанных необязательных возможностей в группы, что во многих случаях избавляет от анализа большого числа опций. Группы эти таковы:

- шифрование;
- средства реального времени;
- продвинутые средства реального времени;
- потоки реального времени;
- продвинутые потоки реального времени;
- трассировка;
- ПОТОКИ;
- унаследованные возможности.

Например, в группу «средства реального времени» (`_XOPEN_REALTIME`) входят возможности четырнадцати видов, в том числе планирование на основе приоритетов, асинхронный ввод/вывод, семафоры, таймеры и т.п.

Версия ОС Linux, на которой готовился текст данного курса, выдавала следующие значения некоторых конфигурационных констант (см. листинг 1.1.).

```
$ getconf _POSIX_VERSION
199506
$ getconf POSIX2_C_DEV
1
$ getconf _XOPEN_REALTIME
1
$ getconf _POSIX_TRACE
undefined
```

Листинг 1.1. Результат применения утилиты `getconf` к одной из версий ОС Linux.

Это значит, что поддерживается устаревшая версия стандарта POSIX, среди прочих присутствуют средства разработки и возможности реального времени; средства трассировки отсутствуют.

В документации на ОС должны быть отражены вопросы соответствия стандарту POSIX, описаны поддерживаемые дополнительные и нестандартные возможности.

Для приложений понятие соответствия стандарту POSIX богаче нюансами. Предусмотрено строгое соответствие, главный отличительный признак которого – ограничение круга используемых возможностей рамками стандарта. Рассматривается и соответствие с применением расширений; в этом случае документация на приложение должна содержать описание требуемых нестандартных возможностей. Желательно, чтобы используемые расширения POSIX-возможностей описывались международными и/или национальными стандартами.

(Отметим, что для реализации понятие строгого POSIX-соответствия бессмысленно хотя бы по той причине, что не бывает операционных систем без средств администрирования, а они не описываются данным стандартом.)

Профилем будем называть набор опций, описывающих необязательные возможности. Соответствие профилю означает соответствие стандарту POSIX и поддержку заданных возможностей. Разумным образом выбранные профили позволяют учитывать потребности представительных классов пользователей и/или приложений.

Допускается существование «подпрофилей», описывающих подмножества стандартных возможностей. Реализация, соответствующая подпрофилю, может функционировать на аппаратных платформах с ограниченными ресурсами и/или обслуживать нужды специфических приложений.

К числу важнейших принадлежат понятия, описывающие поведение реализации в различных ситуациях. Для многих корректных ситуаций поведение бывает неспецифицированным, а значит, мобильное приложение не должно полагаться на совпадение поведения разных реализаций. Для некорректных ситуаций поведение может быть неопределенным; приложению не только не следует полагаться на определенный характер подобного поведения – оно не должно совершать некорректных действий, вызывающих неопределенное поведение.

Еще один близкий термин, «поведение, зависящее от реализации», дополнительно означает, что поведение реализации необходимо документировать.

Стандарт POSIX – это существующий много лет, развивающийся организм, в котором с каждой новой редакцией что-то появляется, а что-то утрачивается. Устаревшими называются возможности, которые еще

поддерживаются различными реализациями, но в будущем они, вероятно, отомрут. Новые приложения не должны их использовать; для каждой из них стандартом предусмотрена адекватная по функциональности современная замена.

Более ограниченный смысл придан термину «унаследованный»: он описывает устаревшие необязательные возможности, которых, разумеется, следует избегать в новых приложениях.

Основные понятия операционных систем, соответствующих стандарту POSIX

Мы рассмотрим следующие основные понятия операционных систем, соответствующих стандарту POSIX:

- пользователь;
- файл;
- процесс;
- терминал;
- хост;
- узел сети;
- время;
- языково-культурная среда.

Это первичные понятия. Их нельзя строго определить, но можно пояснить с помощью других понятий и отношений. Для каждого из выделенных понятий будут описаны присущие им атрибуты и применимые к ним операции.

В тексте стандарта POSIX содержатся следующие пояснения основных понятий вместе со ссылками на атрибуты и операции.

- 1) У пользователя есть имя и числовой идентификатор.
- 2) Файл — объект, допускающий чтение и/или запись и имеющий такие атрибуты, как права доступа и тип. К числу последних относятся обычный файл, символьный и блочный специальные файлы, канал, символьная ссылка, сокет и каталог. Реализация может поддерживать и другие типы файлов.
- 3) Процесс — адресное пространство вместе с выполняемыми в нем потоками управления, а также системными ресурсами, которые этим потокам требуются.
- 4) Терминал (или терминальное устройство) — символьный специальный файл, подчиняющийся спецификациям общего терминального интерфейса.
- 5) Сеть — совокупность взаимосвязанных хостов.
- 6) Языково-культурная среда — часть пользовательского окружения, зависящая от языковых и культурных соглашений.

Для работы с большим числом сущностей всегда предоставляются механизмы группирования и построения иерархий. Существует иерархия файлов, группы пользователей и процессов, подсети и т.п.

Для написания программ, оперирующих с сущностями POSIX-совместимых систем, применяются командный интерпретатор (язык shell) и/или компилируемый язык C. В первом случае приложение может пользоваться служебными программами (утилитами), во втором – функциями. Функциональный интерфейс операционных систем естественно считать первичным, поскольку большинство служебных программ предназначены, по сути, для вызова той или иной функции. По этой причине далее мы будем рассматривать преимущественно уровень функций.

Основными операциями, применимыми к объектам ОС, являются чтение, запись и выполнение. Механизм прав доступа позволяет избирательно разрешать и запрещать осуществление подобных операций. Ранее в стандарте фигурировало понятие суперпользователя, не подверженного контролю прав доступа. В POSIX-2001 выбрана более гибкая формулировка – «имеющий соответствующие привилегии», что отражает прогресс в реализации ОС с расщеплением суперпользовательских возможностей.

В POSIX-совместимых ОС определены объекты, которые можно называть вспомогательными; они помогают организовать взаимодействие между основными сущностями. Особенно широк спектр средств межпроцессного взаимодействия.

Процессы выполняются в определенном окружении, частью которого является языково-культурная среда (Locale), образованная такими категориями, как символы и их свойства, форматы сообщений, дата и время, числовые и денежные величины.

Как правило, с процессом ассоциированы по крайней мере три файла – стандартный ввод, стандартный вывод, стандартный протокол. Обычно стандартный ввод назначается на клавиатуру терминала, а стандартный вывод и стандартный протокол – на экран. Со стандартного ввода читаются команды и (иногда) исходные данные для них. На стандартный вывод поступают результаты выполнения команд. В стандартный протокол помещаются диагностические сообщения.

К операционным системам могут предъявляться качественные требования, например, требование поддержки реального времени: способность обеспечить необходимый сервис в течение заданного отрезка времени.

Среда компиляции POSIX-совместимых приложений

Как правило (хотя это и не всегда осознается), разработка приложений ведется в кросс-режиме, то есть платформа разработки (эквивалентный термин — инструментальная платформа) не совпадает с платформой выполнения (называемой также целевой платформой). На инструментальной платформе создается среда компиляции приложений, так что результат компиляции может быть перенесен для последующего выполнения на целевую платформу.

Важнейшая часть среды компиляции — заголовочные (или включаемые) файлы, содержащие прототипы функций, определения символических констант, макросов, типов данных, структур и т.п. Для каждой описанной в стандарте POSIX функции определено, какие заголовочные файлы должны быть включены использующим ее приложением (обычно требуется один файл).

Выше было указано, что посредством символических констант, определенных в заголовочном файле `<unistd.h>`, операционная система предоставляет приложению информацию о поддерживаемых возможностях. Стандартом POSIX предусмотрен симметричный механизм, называемый механизмом макросов проверки возможностей, он позволяет приложениям объявлять о своем желании получить доступ к определенным прототипам и именам.

Основным макросом проверки возможностей является `_POSIX_C_SOURCE`. Среди требований к приложениям, строго соответствующим стандарту POSIX, фигурирует необходимость определения символической константы `_POSIX_C_SOURCE` со значением 200112L до включения каких-либо заголовочных файлов. Таким образом POSIX-совместимое приложение заявляет, что ему нужны POSIX-имена. Близкую по смыслу роль играет макрос `_XOPEN_SOURCE` (со значением 600).

Примером использования макроса `_POSIX_C_SOURCE` во включаемых файлах ОС Linux может служить фрагмент, приведенный на листинге 1.2.

```
#if defined(_REENTRANT) || (_POSIX_C_SOURCE - 0 >=
    199506L)
#define LIBXML_THREAD_ENABLED
#endif
```

Листинг 1.2. Пример использования макроса проверки возможностей `_POSIX_C_SOURCE`.

Стандартом POSIX предусмотрены некоторые меры для решения важной и трудной проблемы (вызванной в первую очередь необъектным характером языка C), заключающейся в отсутствии пересечений по именам между приложением и операционной системой. Префиксы `_posix_`, `POSIX_` и `_POSIX_` зарезервированы для нужд стандарта.

C подчеркивания, за которым следует еще одно подчеркивание или заглавная латинская буква, могут начинаться только системные (но не прикладные) имена. Для включаемых файлов описаны префиксы используемых в них имен. Например, для операций управления файлами, фигурирующих в `<fcntl.h>`, в качестве префиксов задействованы `F_`, `O_`, `R_`. У средств межпроцессного взаимодействия, описанных в файле `<sys/ipc.h>`, префиксом служит `IPC_`. К сожалению, заголовочных файлов много, а какая-то общая дисциплина именования отсутствует вследствие исторических причин. Так, для манипулирования характеристиками терминалов в файле `<termios.h>` определено множество разнообразных имен: `EXTB`, `VDONR`, `DEFERNO`, `FLUSHO` и т.п. Еще имеется четыреста семнадцать имен типа `_Exit`, `abort`, `abs`, `acos` и т.д., которые могут участвовать в редактировании внешних связей прикладной программы. В результате, прикладной программист может случайно «перебить» системный макрос, внешнюю переменную или функцию, поэтому целесообразно задействовать все диагностические средства среды компиляции и тщательно изучать выдаваемые ими сообщения.

Мобильность POSIX-совместимых приложений

Мобильность приложений, соответствующих стандарту POSIX, принципиально достижима благодаря двум основным факторам. Во-первых — это наличие огромного числа стандартизованных системных сервисов, а во-вторых — возможность динамического выяснения характеристик целевой платформы и подстройки под них приложения. (Естественно, мы имеем в виду мобильность в рамках, регламентируемых стандартом.)

Приложения, соответствующие стандарту POSIX, могут быть одно- и многопроцессными, с возможностью динамической адаптации конфигурации к свойствам целевой платформы. Стандартизованы средства порождения и завершения процессов, смены их программ, опроса и/или изменения разнообразных характеристик. Процессы можно приостанавливать и активизировать в заданное время. Механизм сигналов позволяет извещать о событиях и завершать процессы внешним образом. Для их группирования предусмотрены средства управления заданиями. Приложения снабжены регуляторами для управления планированием и приоритетами процессов. Широкий спектр средств межпроцессного взаимодействия

(очереди сообщений, разделяемая память, семафоры) и управления памятью. Наконец, в пределах процесса можно организовать несколько потоков управления.

Необходимая степень детерминизма выполнения достигается благодаря средствам поддержки реального времени (к ним относятся управление дисциплиной выделения процессоров, сигналы реального времени, удержание страниц в оперативной памяти, таймеры высокого разрешения и т.д.).

Функции для работы с файлами удовлетворяют потребности приложений в чтении и записи долговременных данных, защите таких данных от несанкционированного доступа. Механизм блокировки фрагментов файлов позволяет обеспечить атомарность транзакций. Асинхронный ввод/вывод дает возможность совмещать операции обмена, оптимизируя тем самым приложения. С помощью множества служебных программ можно относительно легко организовать сложную обработку данных.

В стандарте POSIX тщательно проработаны вопросы доступа к внешним устройствам, подсоединенным по последовательным линиям, особенно к терминалам. Возможно, в большей детализации нуждаются средства работы с такими распространенными носителями, как магнитная лента.

Стандартизованный командный язык shell — адекватное средство для написания небольших мобильных процедур и их быстрой интерактивной отладки. Выделим механизм конвейеров, позволяющий объединять команды в цепочки с фильтрацией промежуточных результатов. Служебные программы образуют развитую среду выполнения для shell-процедур. За счет фонового режима можно организовать одновременное выполнение нескольких программ и взаимодействие с ними посредством обычного терминала без многооконных возможностей (впрочем, окна, несомненно, не помешали бы).

POSIX стандартизует интерфейс командной строки. В принципе, он достаточен, в меру удобен и, что важно, создает минимум проблем с точки зрения мобильности. Вероятно, в будущих версиях стандарта будет регламентирован графический интерфейс, но, безусловно, это чревато дополнительными сложностями для разработчиков мобильных приложений.

Языково-культурная среда — одно из важнейших понятий стандарта POSIX с точки зрения мобильности. Приложения способны определять нужную им среду и адаптироваться к потребностям пользователей.

Для многопользовательских систем требуется организация взаимодействия большого числа людей. POSIX решает эту проблему, регламентируя средства непосредственного и почтового обмена информацией.

Стандартом POSIX предусмотрены базовые средства поддержки разработки (в первую очередь — для языка C), что, конечно, не снижает

потребности в специализированных, развитых системах, когда речь идет о работе с действительно большими программными проектами.

Приложениям предоставляются стандартизованные средства для выяснения как «крупноблочных» характеристик целевой системы (например, спектр поддерживаемых необязательных возможностей), так и более мелких характеристик (текущий размер свободного дискового пространства).

Проблема мобильности приложений чрезвычайно сложна, и было бы преувеличением утверждать, что стандарт POSIX-2001 решает ее полностью. Во-первых, за его рамками остаются такие важнейшие вопросы, как графика, многооконный интерфейс и целый ряд других. Во-вторых, в регламентируемых областях присутствуют «белые пятна» неспецифицированного поведения реализаций. Тем не менее, подчеркнем это еще раз, следование стандарту POSIX — обязательный элемент современной дисциплины разработки прикладных систем.

Вариант 1

1. Стандарт POSIX-2001 состоит из

- ☐ двух частей
- ☐ трех частей
- ☐ четырех частей

2. В состав стандарта POSIX-2001 входят

- ☐ основные определения
- ☐ описание математических основ компьютерной арифметики
- ☐ детальное разъяснение положений стандарта, обоснование принятых решений

3. В стандарте POSIX-2001 рассматриваются следующие категории системных компонентов

- ☐ средства администрирования
- ☐ средства обеспечения информационной безопасности
- ☐ средства разработки

Вариант 2

1. Стандарт POSIX-2001 состоит из

- ☐ трех частей
- ☐ четырех частей
- ☐ пяти частей

2. В состав стандарта POSIX-2001 входят

- ☐ описание Java-интерфейса к системным сервисам
- ☐ описание прикладного программного C-интерфейса к системным сервисам
- ☐ описание прикладного программного интерфейса к системным сервисам, не зависящее от языка программирования

3. В стандарте POSIX-2001 рассматриваются следующие категории системных компонентов

- ☐ сетевые средства
- ☐ средства поддержки интеллектуального ввода/вывода
- ☐ средства поддержки многопроцессорных конфигураций

Вариант 3

1. Стандарт POSIX-2001 состоит из

- ☐ четырех частей
- ☐ пяти частей
- ☐ шести частей

2. В состав стандарта POSIX-2001 входят

- ☐ описание интерфейса администрирования ОС
- ☐ описание интерфейса к системным сервисам на уровне командного языка и служебных программ
- ☐ описание интерфейса к системным сервисам, не зависящее от языка программирования

3. В стандарте POSIX-2001 рассматриваются следующие категории системных компонентов

- ☐ потоки управления
- ☐ средства прямого доступа к аппаратуре
- ☐ средства реального времени

Лекция 2. Язык shell

Описывается синтаксис и семантика командного языка shell, способы вызова командного интерпретатора shell, приводятся примеры shell-процедур, анализируются правила формирования и средства разбора командных строк.

Ключевые слова: язык программирования, командный интерпретатор, пробелслово, поле, имя, параметр, команда, простая команда, конвейер, список, командная строка, shell-процедура, необязательная часть, повторение, канал, фильтр, переменная, аргумент, значение, сопоставление с образцом, синхронно выполненная команда, идентификатор процесса, асинхронно запущенный процесс, окружение процесса, домашний каталог пользователя, список поиска, экспорт переменных в окружение, подстановка результатов, вычисление выражения, управляющая конструкция, цикл, условная конструкция, проверка условия, операция отрицания, логическое И, логическое ИЛИ, шаблон, функция, определение функции, имя функции, тело функции, вызов функции, аргумент функции, шаблон имен файлов, префикс, суффикс, тильда, перенаправление ввода/вывода, дескриптор файла, номер дескриптора, вставка, экранирование, встроенная команда, обычная встроенная команда, специальная встроенная команда, трассировка, синоним, интерактивный shell, командная строка, имя программы, опция, имя опции, буквенно-цифровой символ, аргумент опции, операнд, цепочка имен опций.

Основные понятия языка shell

В дальнейшем изложении слово shell будет употребляться в двух смыслах — как имя языка программирования и как название командного интерпретатора.

Для нас shell важен прежде всего как мощный язык мобильного программирования. Предваряя дальнейшее изложение, отметим имеющиеся в нем возможности комбинирования команд с помощью конвейеров, подстановки значений переменных и результатов выполнения команд, генерации имен файлов по шаблонам.

Свойства shell как интерактивного командного интерпретатора, непосредственно взаимодействующего с пользователем, хранение и использование истории сеанса, возможности редактирования командной строки и т.п., на наш взгляд, менее актуальны, поскольку современный пользовательский интерфейс строится на иной основе.

Выделим основные понятия языка shell на лексическом уровне.

Под пробелом в дальнейшем понимается не только собственно пробел, но также и символ табуляции.

Слово — это лексема, отличная от знака операции.

Имя — последовательность букв, цифр, символов подчеркивания, начинающаяся с буквы или подчеркивания.

Параметр — имя, цифра или любой из символов *, @, #, ?, -, \$, !.

Комментарий — лексема, начинающаяся с символа #, а также вся последующая часть строки.

На синтаксическом уровне различаются несколько видов команд.

Простая команда — последовательность полей с разделителями (обычно пробелами) между ними. Первое поле определяет имя команды, которая будет выполняться; оставшиеся поля, за исключением присваиваемых параметрам и перенаправлений ввода/вывода (см. далее), передаются команде в качестве аргументов. Имя команды передается как аргумент ().

Значение простой команды — ее код завершения.

Команда — это либо простая команда, либо одна из управляющих конструкций (см. далее). Кодом завершения команды является код завершения последней выполненной простой команды.

Конвейер — последовательность команд, разделенных знаком |. Стандартный вывод всех команд, кроме последней, направляется на стандартный ввод следующей команды конвейера. Каждая команда выполняется как самостоятельный процесс; shell ожидает завершения последней команды, код завершения которой становится кодом завершения конвейера. Формально будем считать простую команду частным случаем конвейера.

Список — последовательность из одного или нескольких разделенных символами ;, &, && или || конвейеров, она может заканчиваться символами ; или &. Из четырех указанных операций ; и & имеют равные приоритеты, меньшие, чем у && и ||. Приоритеты последних также равны между собой. Символ ; означает, что конвейеры будут выполняться последовательно, а & — параллельно (т. е. shell не ожидает завершения конвейера). Операция && (||) означает, что список, следующий за ней, будет выполняться лишь в том случае, если код завершения предыдущего конвейера нулевой (ненулевой). В списке в качестве разделителя конвейеров вместо символа ; можно использовать символ перевода строки.

Командная строка — строка текста на языке shell.

Shell-процедура — файл, содержащий программу на языке shell.

Для выполнения (почти) каждой простой команды shell порождает отдельный процесс, в рамках которого выполняется программа, хранящаяся в файле, заданном именем команды. Программа может быть выполняемой, т. е. содержать машинные инструкции, или представлять собой shell-процедуру — содержать текст на языке shell.

Далее при описании синтаксиса конструкций языка shell и способа вызова служебных программ будут использоваться следующие соглашения:

- в квадратные скобки заключаются необязательные части конструкций;
- многоточие означает повторение предыдущей части конструкции произвольное число раз.

Конвейеры и примеры их использования

Конвейер — одна из самых красивых конструкций ОС Unix, вошедшая, разумеется, и в стандарт POSIX. Идея его проста, но на редкость продуктивна. С помощью конвейеров удастся комбинировать возможности, предоставляемые разными командами, получая по существу новое качество.

Например, утилита `ls` не подсчитывает число файлов в каталоге, а лишь выдает информацию о них. С другой стороны, служебная программа `wc` способна подсчитать число строк в файле, но не имеет отношения к распечатке содержимого каталогов. Если же построить конвейер из двух упомянутых команд, количество файлов в каталоге легко вычисляется. Например, результатом работы конвейера (см. листинг 2.1) на нашей установке ОС Linux будет число 92 (утилита `wc`, вызванная без аргументов, обрабатывает файл стандартного ввода, который в данном случае является результатом работы команды `ls`). Значит, в каталоге `/bin` 91 файл, если считать и элементы, соответствующие текущему и вышедшему каталогам (первая строка выдачи `ls` содержит суммарное число блоков, занятых файлами каталога).

```
ls -al /bin | wc -l
```

Листинг 2.1. Пример конвейера.

Еще один пример. Пусть необходима информация о файлах текущего каталога, модифицированных в октябре. К цели ведет конвейер, показанный в листинге 2.2.

```
ls -al | grep "Oct "
```

Листинг 2.2. Еще один пример конвейера.

Служебная программа `grep` играет здесь роль фильтра, который пропускает для вывода только часть строк, выдаваемых `ls`.

Можно выстроить и трехступенчатый конвейер, если требуется подсчитать число файлов, модифицированных в октябре (см. листинг 2.3).

```
ls -al | grep "Oct " | wc -l
```

Листинг 2.3. Пример трехступенчатого конвейера.

Здесь утилиту `grep` с еще большим правом можно назвать фильтром.

Приведем еще один пример конвейера, полезного, когда нужна подробная информация о большом каталоге (см. листинг 2.4).

```
ls -Rl /dev | more
```

Листинг 2.4. Конвейер для постраничного просмотра результатов.

Связующее звено между последовательными компонентами конвейера называется каналом. Иными словами, для интерпретации конвейера `shell` создает временный файл типа «канал», с одного конца в него заносят данные, а с другого — читают. С помощью служебной программы `tee` можно организовать ответвление канала, т. е. помещать информацию не только на стандартный вывод, но и в указанные файлы:

```
tee файл ...
```

Например, если нужно не только подсчитать число файлов из текущего каталога, модифицированных в октябре, но и поместить информацию о них в файл для последующего анализа, следует построить четырехступенчатый конвейер (см. листинг 2.5).

```
ls -al | grep "Oct " | tee /tmp/tmpinf | wc -l
```

Листинг 2.5. Четырехступенчатый конвейер.

В результате его выполнения на экране появится сообщение о количестве нужных файлов, а в файле `/tmp/tmpinf` — информация о них.

Переменные и аргументы `shell`-процедур

Переменные обозначаются именами. Значения могут присваиваться им привычным способом, то есть посредством команд вида:

```
имя=значение [имя=значение] ...
```

Все значения в языке `shell` трактуются как текстовые. Подчеркнем, что, в соответствии с этими требованиями, конструкция

имя=значение

должна представлять собой одно слово — в ней не может быть пробелов.

Обычно в языках программирования ясно из контекста, где подразумевается имя переменной, а где значение. Так, в левой части оператора присваивания обычно используется имя, в правой — значение. В shell все не так. Переход от имени переменной к значению помечается посредством явной операции \$. Если в команде встречается конструкция

```
$имя
```

то вместо нее интерпретатор shell подставляет значение переменной с указанным именем. Допускается и запись

```
${имя}
```

с тем же смыслом, если нужно отделить имя от последующего текста.

Рассмотрим пример. После выполнения команд (см. листинг 2.6) — утилита echo (эхо) выдает на стандартный вывод свои аргументы — на экране появится результат вывода значений переменных (см. листинг 2.7).

```
a=value_of_variable
b=1+2
echo a = $a
echo b = $b
```

Листинг 2.6. Присваивание и извлечение значение переменных.

```
a    value_of_variable
b    1+2
```

Листинг 2.7. Результат вывода значений переменных.

Значения формальных аргументов shell-процедур обозначаются как

\$цифра

\$0 — это имя интерпретируемой shell-процедуры. Если заданных при вызове команды фактических аргументов меньше, чем 9, «лишние» формальные аргументы получают пустые значения. О том, как добраться до фактических аргументов с номерами большими, чем 9, будет сказано ниже (см. управляющую конструкцию for и команду shift).

В качестве примера рассмотрим shell-процедуру, которая выдаст на стандартный вывод свое имя и значения трех первых аргументов (см. листинг 2.8).

```
echo Имя команды: $0
echo Значение первого аргумента: $1
echo Значение второго аргумента: $2
echo Значение третьего аргумента: $3
```

Листинг 2.8. Пример shell-процедуры.

Пусть приведенный текст помещен в файл с именем `three_args`. Тогда после выполнения команды (см. листинг 2.9) на экране появится ее результат (см. листинг 2.10).

```
three_args arg1 . - arg4
```

Листинг 2.9. Пример вызова shell-процедуры с аргументами.

```
Имя команды: three_args
Значение первого аргумента: arg1
Значение второго аргумента: .
Значение третьего аргумента: -
```

Листинг 2.10. Результат выполнения shell-процедуры.

Поскольку в тексте shell-процедуры упомянуты только первые 3 аргумента, значения аргументов с большими номерами (даже если они заданы, как в приведенном примере) не влияют на ее работу.

Команда

```
three_args arg1 . .
```

выдаст другой результат (см. листинг 2.11).

```
Имя команды: three_args
Значение первого аргумента: arg1
Значение второго аргумента: --
Значение третьего аргумента:
```

Листинг 2.11. Еще один результат выполнения shell-процедуры.

Значение третьего формального аргумента пусто.

Прежде чем начнется выполнение командной строки, т. е. будет вызвана заданная в ней команда, строку обрабатывает shell, выполняющий в ней некоторые подстановки. С одной из них — значения переменной вместо конструкции `$имя` — мы уже познакомились. Язык shell содержит ряд аналогичных конструкций. Рассмотрим их.

Там, где в командной строке встречается запись вида

```
{имя: слово}
```

вместо нее подставляется значение переменной с указанным именем, если это значение непусто, в противном случае подставляется слово. Например, при наличии в shell-процедуре присваивания

```
initdir=${1:-/}
```

переменная `initdir` получит значение первого аргумента, если оно непусто. Если же процедуру вызвали без аргументов, значением `initdir` станет символ `/`. Подобные конструкции — удобный способ использования подразумеваемых значений.

Запись вида

```
{имя:=слово}
```

влечет присваивание нового значения (слово) переменной с заданным именем, только если ее старое значение не установлено или пусто (удобный способ присваивания переменным подразумеваемых значений). Вместо всей конструкции в командную строку подставляется итоговое (непустое старое или присвоенное новое) значение переменной.

Конструкция

```
{имя?:слово}
```

предназначена для выдачи в стандартный протокол сообщения об ошибке, если значение переменной с заданным именем не установлено или пусто, после чего неинтерактивный shell завершает работу с ненулевым кодом возврата. Непустое значение подставляется в командную строку, и интерпретация командного файла продолжается обычным образом.

Вместо конструкции

подставляется слово, если значение переменной с заданным именем не пусто; в противном случае не подставляется ничего. Такой способ удобен для использования альтернативных значений.

Если в приведенных выше конструкциях опустить двоеточие, будет отменена проверка непустоты значения (то есть останется лишь проверка того, установлено ли значение переменной с заданным именем). В остальном смысл конструкций остается прежним.

Shell содержит базовые средства для обработки цепочек символов. Так, вместо конструкции

```
${#имя}
```

подставляется число символов в значении переменной с заданным именем.

Предоставляется четыре конструкции для сопоставления с образцом:

```
${#имя%слово}  
${#имя%%слово}  
${#имя#слово}  
${#имя##слово}
```

Во всех случаях слово рассматривается как образец (шаблон, см. далее); после сопоставления с ним подставляется значение переменной с заданным именем, из которого удален минимальный (максимальный) сопоставленный суффикс (префикс).

Приведем несколько примеров. Для их понимания достаточно знать, что шаблон `*` сопоставляется с произвольной цепочкой символов, в том числе с пустой.

После присваивания переменной

```
HOME=/home/galat
```

команда

```
echo ${#HOME}
```

выдаст 11.

Заменить в имени файла суффикс `.f` на `.for` можно с помощью команд, показанных в листинге 2.12.

```
f=file.f  
mv $f ${x%.f}.for
```

Листинг 2.12. Пример сопоставления с образцом.

Выдать первый элемент маршрутного имени файла (см. далее) можно с помощью команд, показанных в листинге 2.13.

```
f=маршрутное_имя  
echo ${f%%/*}
```

Листинг 2.13. Второй пример сопоставления с образцом.

Последний элемент выдается командой, приведенной в листинге 2.14.

```
echo ${f##*/}
```

Листинг 2.14. Третий пример сопоставления с образцом.

Служебные переменные языка shell

Значения некоторых переменных устанавливаются самим языком shell. Перечислим эти переменные и опишем их предназначение.

#

Количество фактических аргументов (десятичное).

-

Флаги (однобуквенные опции), указанные при запуске shell или установленных посредством команды `set`. (см. далее).

?

Десятичное значение, возвращенное предыдущей синхронно выполненной командой.

\$

Идентификатор процесса, в рамках которого выполняется shell.

!

Идентификатор последнего асинхронно запущенного процесса.

`*`, `@`

Совокупность всех фактических аргументов (начиная с `$1`), разделенных пробелами.

Напомним: чтобы получить значения этих переменных, перед ними нужно поставить знак `$`.

Значения `$@` и `$*` имеют некоторые тонкие различия, на которых мы, однако, останавливаться не будем.

Несколько усложним процедуру `three_args`, чтобы продемонстрировать только что описанные возможности (см. листинг 2.15).

```
echo Идентификатор текущего процесса: $$
echo Имя команды: $0
echo Число фактических аргументов: $#
echo Совокупность всех аргументов: $@
echo Значение первого аргумента: $1
echo Значение второго аргумента: $2
echo Значение третьего аргумента: $3
```

Листинг 2.15. Усовершенствованная shell-процедура `three_args`.

Теперь, вызывая усовершенствованную процедуру `three_args`, выполним командную строку

```
three_args arg1 . - arg4
```

Тогда на экране появится примерно следующее (см. листинг 2.16).

```
Идентификатор текущего процесса: 3882
Имя команды: three_args
Число фактических аргументов: 4
Совокупность всех аргументов: arg1 . - arg4
Значение первого аргумента: arg1
Значение второго аргумента: .
Значение третьего аргумента: -
```

Листинг 2.16. Результат вызова усовершенствованной процедуры `three_args`.

Окружение процессов

Окружение — это набор пар (имя, значение), который передается выполняемой программе так же, как и обычный список аргументов. Иными словами, порождаемые процессы наследуют окружение процесса-предка. Компонентами окружения являются, помимо прочих, следующие переменные и их значения:

HOME

Подразумеваемый аргумент утилиты смены текущего каталога `cd` — домашний каталог пользователя.

IFS

Цепочка символов, где перечислены разделители полей; обычно включает пробел, табуляцию и перевод строки.

PATH

Список имен каталогов для поиска команд. В дальнейшем подобные списки называются списками поиска. Элементы списка разделяются двоеточием. Пустой элемент означает текущий каталог.

PS1

Основное приглашение интерактивного языка shell (по умолчанию «\$»).

TERM

Тип пользовательского терминала.

TZ

Информация о часовом поясе.

Для отсылки информации об окружении на стандартный вывод следует воспользоваться командой

env

Поясним подробнее смысл некоторых переменных окружения.

Прежде чем выполнить команду, shell ищет файл с соответствующим именем в последовательности каталогов, являющейся значением переменной `PATH`. Назовем подобную последовательность списком поиска. Если, например, значение `$PATH` суть

```
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/galat/bin
```

то нужный файл будет сначала разыскиваться в каталоге `/usr/local/bin`, затем в `/bin` и т.д. Как только файл отыщется, поиск прекратится. Это важно, если в разных каталогах есть одноименные выполнимые файлы.

Значение `$IFS` влияет не только на разбор команд, но и на чтение строк данных с помощью команды `read` (см. далее). Кроме того, первый символ из значения `$IFS` вставляется между фактическими аргументами при выполнении подстановки `$*`.

Переменная `TERM` хранит тип терминала пользователя. Интерактивные утилиты (редакторы или другие программы с экранным интерфейсом, например `talk`) с помощью значения `$TERM` настраиваются на конкретный тип терминала.

Переменная `TZ` задает локальный часовой пояс. Эта информация необходима всегда, когда требуется показать текущее время. Особенно полезна она при почтовом взаимодействии с территориально удаленными (в частности, зарубежными) пользователями.

Для изменения окружения мало присвоить новое значение соответствующей переменной. Дело в том, что по умолчанию переменные считаются локальными по отношению к shell-процедуре, т. е. присваивание изменит локальную переменную, но не затронет одноименную переменную окружения. Таким образом, в окружение новых процессов (порожденных, например, для выполнения последующих команд данной shell-процедуры) войдет переменная со старым значением.

С помощью конструкции

```
export имя[=слово]
```

переменная с указанным именем и значением (последнее может быть опущено) помещается в окружение, т. е. становится глобальной.

Следующая строка иллюстрирует типичное применение команды `export`:

```
export PATH=/local/bin:$PATH
```

Переменная `PATH` изменилась, и новое значение экспортировано в окружение.

Команда

```
export -p
```

выдает на стандартный вывод имена и значения всех экспортированных переменных. Эту выдачу можно использовать для сохранения и последующего восстановления (быть может, с некоторыми модификациями) окружения. Ниже приведен фрагмент возможного результата работы команды `export -p` (см. листинг 2.17).

```
export HISTSIZE="1000"
export HOME="/home/galat"
export LANG="C"
export LESSCHARSET="koi8-r"
export LOGNAME="galat"
export MAIL="/var/spool/mail/galat"
export TTY="/dev/ttyS4"
export USER="galat"
```

Листинг 2.17 Возможные результаты выполнения команды `export -p`.

Мы видим, что выдача устроена так, чтобы вновь быть поданной на вход командного интерпретатора.

Подстановка результатов выполнения команд

Если в командной строке встретилась цепочка символов, заключенная в обратные кавычки (```), она интерпретируется как команда, стандартный вывод которой подставляется вместо упомянутой конструкции. Говорят, что в этом случае производится подстановка результатов выполнения команды, а сами обратные кавычки называют символами подстановки.

Эквивалентной синтаксической формой для подстановки результата выполнения команды является конструкция вида

```
$(команда)
```

Для выполнения заданной команды порождается процесс, в рамках которого работает еще один экземпляр командного интерпретатора.

Опишем несколько употребительных способов использования подстановки результатов. Пусть файл `filelist` содержит список имен других

файлов и над совокупностью последних требуется проделать некоторое действие. Если в командную строку поместить конструкцию

```
... `cat filelist` ...
```

то это аналогично явному указанию в командной строке на все файлы из списка. (Утилита `cat` выдает на стандартный вывод содержимое файлов, заданных в качестве аргументов.) Пример подстановки результатов выполнения команды (выдача информации о файлах):

```
ls -l `cat filelist`
```

Нет нужды в том, чтобы каждая служебная программа умела читать свои аргументы из файла: имеются универсальный механизм подстановки результатов и команда `cat`.

Подстановка результатов выполнения команд может сочетаться с другими видами подстановок, например, значения вместо имени переменной. Так, после обработки конструкции

```
${x:-$(ls)}
```

команда `ls` выполнится только тогда, когда значение `x` не установлено или пусто; в противном случае выполнится команда `$x`, и ее результат займет место в командной строке.

В языке `shell` все значения считаются текстовыми. Значит, для выполнения операций с числами нужны особые средства. Вместо конструкции вида

```
$( (выражение) )
```

`shell` подставит результат вычисления этого выражения, что (с некоторой натяжкой) можно рассматривать как частный случай подстановки результатов выполнения команд. Например, после обработки строки

```
i=$((i+1))
```

значение `i` (если оно было числом) увеличится на единицу.

Стандарт POSIX обязывает поддерживать арифметику длинных целых со знаком; константы (десятичные, восьмеричные, шестнадцатеричные) должны удовлетворять требованиям языка C.

Управляющие конструкции

Среди прочих язык shell содержит следующие управляющие конструкции.

Оператор цикла `for`:

```
for имя [in слово ...]
do список
done
```

При каждой итерации переменная `имя` принимает следующее значение из набора

```
in слово ...
```

Если конструкция `in слово ...` опущена, то список выполняется для каждого формального аргумента.

Условный оператор:

```
if список_1
then список_2
[elif список_3
then список_4]
...
[else список_5]
fi
```

Выполняется `список_1`. Если код его завершения 0, то выполняется `список_2`, в противном случае — `список_3`, и если код его завершения 0, то выполняется `список_4` и т.д. Если же коды завершения всех списков, использованных в качестве условий, оказались ненулевыми, выполняется `else-часть` (`список_5`). Если `else-часть` отсутствует и ни одна `then-часть` не выполнялась, возвращается нулевой код завершения.

Оператор цикла `while` (`until`):

```
while список_1
do список_2
done
```

Пока код завершения последней команды `списка_1` есть 0, выполняются команды `списка_2`. При замене служебного слова `while` на `until` условие продолжения цикла меняется на противоположное. Если

команды из списка_2 не выполнялись вообще, код завершения устанавливается равным нулю.

Оператор выбора:

```
case слово in
  [шаблон [! шаблон] ...) список ;;]
  ...
esac
```

Выполняется список, соответствующий первому из шаблонов, успешно сопоставленным со словом. Формат шаблона аналогичен используемому для генерации имен файлов (см. далее).

Определение функции:

```
имя () открывающая_скобка
  список
закрывающая_скобка
```

Определяется функция с заданным именем. Список является ее телом, которое окружают круглые или фигурные скобки. Для вызова функции используется обычный синтаксис команды:

```
имя [аргумент ...]
```

Если тело функции заключено в фигурные скобки, она выполняется в рамках текущего процесса; в случае круглых скобок порождается отдельный процесс. На время выполнения функции аргументы \$1, \$2, ..., а также служебная переменная # получают новые значения, определенные аргументами команды вызова. Затем восстанавливаются старые значения.

Приведем примеры использования управляющих конструкций. Сначала усовершенствуем процедуру `three_args`, чтобы она выдавала значения всех, а не только первых трех аргументов (см. листинг 2.18).

```
echo Идентификатор текущего процесса: $$
echo Имя команды: $0
echo Число фактических аргументов: $#
echo Совокупность всех аргументов: $@
i=1
for arg
do
  echo Значение аргумента номер ${i}: $arg
```



```
i=$((i+1))  
done
```

Листинг 2.18. Еще одно усовершенствование shell-процедуры `three_args`.

В управляющих конструкциях `if` и `while` часто используется команда `test`, она проверяет некоторое условие и вырабатывает нулевой или ненулевой код завершения в зависимости от того, оказалось ли условие, соответственно, истинным или ложным. Команда записывается любым из двух способов:

```
test условие
```

или при помощи пары квадратных скобок:

```
[ условие ]
```

Условие может включать, помимо прочих, следующие примитивы:

`-r` файл

Истина, если файл существует и доступен для чтения.

`-w` файл

Истина, если файл существует и доступен для записи.

`-x` файл

Истина, если файл существует и является выполнимым.

`-f` файл

Истина, если файл существует и является обычным файлом.

`-d` файл

Истина, если файл существует и является каталогом.

`-s` файл

Истина, если файл существует и является специальным символьным файлом.

`-b` файл

Истина, если файл существует и является специальным блочным файлом.

`-p` файл

Истина, если файл существует и является именованным каналом.

`-s` файл

Истина, если файл существует и имеет ненулевой размер.

`-n s1`

Истина, если цепочка символов `s1` имеет ненулевую длину.

`s1 = s2`

Истина, если цепочки символов `s1` и `s2` равны.

`s1 != s2`

Истина, если цепочки символов `s1` и `s2` не равны.

`n1 -eq n2`

Истина, если целые числа `n1` и `n2` алгебраически равны. На месте `-eq` могут быть также операции сравнения `-ne` (не равно), `-gt` (больше), `-ge` (больше или равно), `-lt` (меньше), `-le` (меньше или равно).

Примитивы комбинируются с помощью круглых скобок, а также следующих операций (в порядке уменьшения приоритета):

`!` — унарная операция отрицания;

`-a` — логическое И;

`-o` — логическое ИЛИ.

Обратим внимание на то, что квадратные скобки, обрамляющие условие, и каждый компонент условия должны быть выделены пробелами.

Приведем пример использования управляющей конструкции `if`. В процессе загрузки практически любой разновидности ОС Unix выполняются строки следующего или близкого вида (см. листинг 2.19):

```
if [ -s ${f} ]
then
    /bin/sh ${f} start
fi
```

Листинг 2.19. Пример условного оператора.

Если файл, имя которого является значением переменной `t`, существует и имеет ненулевой размер, он выполняется с аргументом `start`.

В качестве примера употребления конструкции `case` приведем еще один фрагмент, типичный для процесса загрузки системы (см. листинг 2.20).

```
case "$1" in
start)
    start
    ;;
stop)
    stop
    ;;
reload | restart)
    restart
    ;;
condrestart)
    if [ -f /var/lock/subsys/atd ]
    then
        restart
    fi
    ;;
*)
    echo $"Usage: $0 {start | stop | restart
        | condrestart}"
    exit 1
esac
```

Листинг 2.20. Пример оператора выбора.

Известные значения первого аргумента распознаются, в ответ на все прочие (шаблон `*`) сообщается, как пользоваться данной shell-процедурой.

Следующий пример иллюстрирует определение и вызов функции (см. листинг 2.21).

```
echo $# $1
f ( ) {
    echo $# $1
}
f a b
f b
echo $# $1
```

Листинг 2.21. Пример определения и вызова функции.

Если данный текст хранится в файле `g`, то по окончании выполнения команды (вызов `shell`-процедуры, содержащей функцию)

```
g c d e
```

будет выдан следующий результат:

```
3 c
2 a
1 b
3 c
```

Генерация маршрутных имен файлов

После всех подстановок, прежде чем команда начнет выполняться, в каждом составляющем ее поле осуществляется поиск символов `*`, `?`, и `[`. Если находится хотя бы один из них, то это поле рассматривается как шаблон имен файлов и заменяется именами файлов, удовлетворяющими данному шаблону. Имена подставляются в алфавитном порядке. Если ни одно имя файла не удовлетворяет шаблону, поле остается неизменным. Символ `.` в начале имени файла или непосредственно после `/`, так же как и сам символ `/`, должны быть заданы в шаблоне явно. Трактовка символов `*`, `?` и `[`:

- `*` — сопоставляется с произвольной цепочкой символов, в том числе с пустой.

- `?` — сопоставляется с произвольным символом.

- `[ . . . ]` — сопоставляется с любым из перечисленных в скобках символов. Пара символов, разделенных знаком `-`, обозначает отрезок алфавита, включающий сами указанные символы. Если сразу вслед за `[` идет `!`, шаблону удовлетворяет любой символ, не перечисленный в скобках.

Рассмотрим несколько примеров. Чтобы подсчитать суммарное число строк во всех `C`-файлах текущего каталога, достаточно выполнить команду

```
wc -l *.c
```

В качестве второго примера расширим приведенный выше фрагмент действий при загрузке системы (см. листинг 2.22)

```
for f in /etc/rc$runlevel.d/S*
do
    if [ -s ${f} ]
    then
        /bin/sh ${f} start
    fi
done
```

Листинг 2.22. Пример сочетания управляющих конструкций с генерацией имен файлов.

В цикле будут в алфавитном порядке запускаться все непустые файлы из каталога, соответствующего заданному уровню выполнения (обычно это `/etc/rc3.d`, поскольку `runlevel` имеет значение 3, но в данном случае это не суть важно), имена которых начинаются на `S`.

Как пример использования шаблона имен файлов приведем команду

```
rm -i *.*[Bb]?
```

Она удалит из текущего каталога все файлы, имена которых начинаются с точки, сразу после второй точки стоит одна из букв — `B` или `b`, а затем произвольный символ.

Читателю предлагается рассмотреть еще один способ применения шаблона имен файлов и самостоятельно ответить на вопрос, удалит ли команда

```
rm -f *.*[Bb][AaUu]
```

файл с именем `.bu`.

Наконец, рассмотрим более сложный пример. Пусть требуется так переименовать фортрановские файлы текущего каталога, чтобы окончание `.f` заменилось на `.for`. Служебная программа

```
basename цепочка_символов [суффикс]
```

убирает из цепочки_символов любой оканчивающийся на / префикс и суффикс (если он есть) и выдает результат на стандартный вывод. Применим ее для решения сформулированной задачи (см. листинг 2.23).

```
for f in *.f
do
    mv $f `basename $f .f`.for
done
```

Листинг 2.23. Пример подстановки результатов команды как части слова.

Особенностью приведенного примера является использование подстановки результатов команды как части слова (в данном случае — как части нового имени файла).

Еще одним видом генерации маршрутных имен файлов можно было бы считать обработку символа тильды '~', однако это действие открывает, а не завершает подстановки в командной строке. Префикс слов, от тильды до ближайшего слэша (или конца слова, если слэш отсутствует), заменяется на имя домашнего каталога пользователя, входное имя которого задано префиксом. Если префикс пустой, вместо него подставляется значение переменной окружения HOME. В присваиваниях тильда распознается и обрабатывается не только в начале слова, но и после знаков равенства и двоеточия, что позволяет, в частности, естественным образом обращаться со значением переменной окружения PATH.

Так, при подстановке результатом работы команды

```
echo~
```

может быть

```
/home/galat
```

Перенаправление ввода/вывода

Командный интерпретатор shell ассоциирует с каждым открытым файлом так называемый дескриптор. Дескрипторы нумеруются десятичными целыми числами, начиная с нуля. Верхняя граница зависит от реализации, но, согласно стандарту POSIX, должны поддерживаться по крайней мере десять одновременно открытых файлов (с номерами дескрипторов от 0 до 9 включительно).

Дескрипторы с номерами 0, 1 и 2 имеют специальный смысл. Они соответствуют стандартному вводу, стандартному выводу и стандартному протоколу.

Перед выполнением команды ее ввод и вывод могут быть перенаправлены, для чего используется специальная нотация, которую интерпретирует shell. Описанные ниже конструкции располагаются в любом месте простой команды, предшествуют команде или завершают ее и не передаются в качестве аргументов команды; если они завершают определение функции, то воздействуют на все команды тела, не производящие собственных перенаправлений.

Перенаправление ввода:

```
<слово
```

Использовать файл слово для стандартного ввода (дескриптор файла 0).

Перенаправление вывода с перезаписью:

```
>слово  
>| слово
```

Использовать файл слово для стандартного вывода (дескриптор файла 1). Имеющийся файл опустошается, а если файла нет, он создается. (При использовании первой разновидности перенаправления вывода в некоторых случаях попытка перезаписи существующего файла является ошибкой; см. далее описание опции -C команды set.)

Перенаправление вывода с дозаписью:

```
>>слово
```

Использовать файл слово для стандартного вывода. Если файл существует, то выводимая информация добавляется в конец (сначала производится поиск конца файла); в противном случае файл создается.

Если любой из этих конструкций предшествует цифра, она определяет дескриптор (вместо подразумеваемых дескрипторов 0 или 1), который будет ассоциирован с файлом, указанным в конструкции. Например, строка

```
... 2>protocol
```

перенаправляет стандартный протокол (дескриптор 2) в файл по имени protocol.

Ввод и вывод могут перенаправляться не только в файл с заданным именем, но и в открытый файл с заданным дескриптором. Для этого в описанных выше конструкциях в качестве слова следует употребить связку &номер_дескриптора:

```
<&номер_дескриптора  
>&номер_дескриптора
```

Например, чтобы перенаправить стандартный протокол туда же, куда назначен стандартный вывод, следует употребить конструкцию

```
... 2>&1
```

Если в качестве номера_дескриптора указан минус ('-'), соответствующий файл (стандартный ввод, вывод или явно заданный своим дескриптором файл) закрывается.

Shell позволяет открыть файл одновременно на чтение и запись при помощи конструкции

```
<>слово
```

Ей, как всегда, может предшествовать номер дескриптора для открываемого файла. По умолчанию используется 0, то есть стандартный ввод.

Приведем два примера. Пусть нужно измерить время выполнения некоторой команды, направив ее результаты в файл `cmd.res`, а данные о времени — в файл `cmd.time`. К цели ведет строка, приведенная в листинге 2.24.

```
time команда >cmd.res 2>cmd.time
```

Листинг 2.24. Пример перенаправления стандартного вывода и стандартного протокола.

Второй пример. Рассмотрим цикл, описанный в листинге 2.25.

```
i=0  
while [ $i -lt 40 ]  
do  
  > lost+found/g$i  
  i=$((i+1))  
done  
rm lost+found/*
```

Листинг 2.25. Пример перенаправления стандартного вывода пустой команды.

С его помощью создается 40 файлов в каталоге `lost+found`, которые затем удаляются. Отметим, что перенаправляется стандартный вывод пустой команды, а в результате создается пустой файл.

Поясним смысл приведенного фрагмента. При проверке и коррекции файловой системы утилитой `fsck` в каталог `/lost+found` помещаются непустые файлы, на которые нет ссылок. Сложность в том, что пока утилита `fsck` работает, ни один файл не должен расширяться, т. е. в каталоге `/lost+found` должны быть заранее заготовленные пустые места.

Shell даст возможность размещать исходные данные в виде вставки в интерпретируемый командный файл, для чего употребляются многострочные конструкции:

```
<<слово
    вставка
слово
```

и

```
<<-слово
    вставка
слово
```

Их можно считать разновидностями перенаправления ввода. На вход команды подается вставка, представляющая собой череду строк вплоть до слова, указанного при перенаправлении. Второй вариант отличается тем, что из строк вставки удаляются начальные символы табуляции.

В качестве примера использования вставок приведем файл ОС Linux `/etc/tripwire/twinstall.sh`, который содержит следующие фрагменты (см. листинг 2.26 и листинг 2.27):

```
cat << END_OF_TEXT
A clear-text version of the Tripwire configuration
file
$TXT_CFG
has been preserved for your inspection. It is
recommended
that you delete this file manually after you have
examined it.
END_OF_TEXT
```

Листинг 2.26. Пример использования вставки.

```
cat <<eof1; cat <<eof2
Вставка 1
eof1
Вставка 2
eof2
```

Листинг 2.27. Пример использования двух последовательных вставок.

В результате выполнения второго фрагмента будут выданы две строки:

```
Вставка 1
Вставка 2
Экранирование
```

Под экранированием имеется в виду защита символов от интерпретации со стороны языка shell. Следующие символы shell трактует по-особому; не будучи экранированными, они завершают предшествующее им слово:

```
| & ; < > ( ) $ ` \ " ' пробел перевод_строки
```

Не являясь разделителями слов, особую роль играют для shell и символы

```
* ? [ | # ~ = %
```

Каждый из перечисленных выше символов может быть экранирован, т. е. представлять самого себя, если перед ним стоит `\` либо он расположен между одиночными или двойными кавычками (`' '` или `" "`).

Экранирование с помощью двойных кавычек имеет некоторые особенности. Во-первых, оно не отменяет особой трактовки символов `$` и ```. Иными словами, внутри двойных кавычек обычным образом выполняется подстановка значений переменных, результатов выполнения команд и результатов вычисления арифметических выражений, только результат подстановки расценивается как одно слово. Во-вторых, особая трактовка символа `\` отменяется внутри двойных кавычек лишь частично: символ `\` сохраняет экранирующую роль, если после него следуют:

```
$ ` " \ перевод_строки
```

Приведем несколько примеров использования двойных кавычек для экранирования. В первом команда

```
echo "$(echo *)"
```

выдаст список файлов текущего каталога, а во втором результатом выполнения строки

```
echo "$(echo " * ") "
```

станет звездочка. Если же аргумент `"$(echo *)"` предложить усовершенствованной процедуре `three_args`, то число фактических аргументов окажется равным единице, а список файлов текущего каталога станет значением первого аргумента.

Рассмотрим третий пример. Пусть нужно выдать информацию о всех файлах текущего каталога, которые модифицировались последний раз 3 октября. Конвейер

```
ls -al | grep "Oct 3"
```

сделает то, что нужно (служебная программа `grep` выполняет сопоставление заданного шаблона со строками заданных файлов или стандартного ввода). Кавычки использованы для экранирования двух пробелов в разыскиваемой цепочке символов. Без экранирования пробелы превращаются в разделители аргументов, и команда `grep` попыталась бы найти текст `Oct` в файле с цифровым именем 3.

Еще один пример. Запись

```
${($i\*5)}
```

синтаксически неверна, поскольку в арифметических выражениях `shell` рассматривает звездочку как операцию умножения, а не как элемент шаблона имен файлов; соответственно, она не нуждается в экранировании, и символ `\` становится лишним, нарушая синтаксис выражения. Таким образом, лишнее экранирование может быть вредным.

Встроенные команды

Мы уже отмечали, что подавляющее большинство команд разыскивается в списке поиска и выполняется языком `shell` в рамках порожденных процессов. Однако для некоторых команд — либо по соображениям эффективности, либо потому, что требуемую функциональность трудно достичь в рамках отдельного процесса — целесообразно сделать исключение и реализовать их как встроенные, их выполняет сам `shell`. Очевидным примером встроенной команды служит `export` — пометить переменную как глобальную, входящую в окружение процесса проще внутренними методами.

Встроенные команды подразделяются на обычные и специальные. С точки зрения приложений первые ничем (кроме, быть может, эффективности обработки) не отличаются от невстроенных утилит. В частности, они могут фигурировать в качестве программы процесса. Специальные встроенные команды этим свойством обладать не обязаны. Кроме того, у них есть еще два отличия:

- синтаксическая ошибка в специальной встроенной команде способна привести к аварийному завершению shell;
 - присваивание переменным, заданное посредством специальной встроенной команды, остается в силе и после ее завершения.
- Перечислим некоторые из специальных встроенных команд.

`.` файл

Shell читает и выполняет команды из файла, затем возобновляется чтение со стандартного ввода; при поиске файла используется значение переменной `PATH`. Иными словами, специальная команда `.` позволяет выполнить любую команду в рамках текущего процесса.

`eval [аргумент ...]`

Выполнить команду, заданную аргументами `eval`.

`exec [аргумент ...]`

Сменить программу процесса: в рамках текущего процесса команда, заданная аргументами `exec`, заменяет shell. В качестве аргументов могут указываться спецификации ввода/вывода, и, если нет никаких других аргументов, будет лишь перенаправлен ввод/вывод текущего процесса shell.

`exit [код_завершения]`

Завершить выполнение shell с указанным кодом. При отсутствии аргумента код завершения определяется последней выполненной командой. Чтение символа конца файла также приводит к завершению shell.

`shift [n]`

Формальные аргументы, начиная с $(n+1)$ -го, переименовываются в $\$1$ и т.д. По умолчанию $n=1$.

К числу обычных встроенных команд относятся:

```
cd [каталог]
```

Сделать текущим заданный каталог. Если каталог не указан, используется значение переменной окружения HOME.

```
pwd
```

Выводит имя текущего каталога.

```
read [переменная ...]
```

Со стандартного ввода читается одна строка и делится на поля; первое поле присваивается первой переменной, второе — второй и т.д., причем последовательность всех оставшихся полей присваивается последней переменной. Исходная строка имеет продолжение, если в конце ее стоит последовательность `\перевод_строки`. Символы, отличные от перевода строки, также могут быть экранированы с помощью `\`, который удаляется перед присваиванием полей. Возвращается нулевой код завершения, если только не встретился конец файла.

Некоторые пояснения. С помощью команды `exec` осуществляется смена программы текущего процесса. Например, если пользователь привык работать в командном интерпретаторе `bash`, он может перейти туда по команде

```
exec bash
```

При этом обычный shell бесследно исчезнет.

Аргументы специальной встроенной команды `eval` обрабатываются дважды: когда исходную строку интерпретирует `shell` и когда работает сама `eval`. Так, после выполнения строк

```
a=1
b=a
eval c='$'$b
echo $c
```

на стандартный вывод будет выдана единица, поскольку аргументом `eval` будет оператор присваивания `c=$a`.

Если файл `data` содержит строку

```
1 2 3 4 5
```

то после выполнения специальной встроенной команды `read`

```
read a b c < data
echo $c $b $a
```

на стандартный вывод будет выдан следующий результат:

```
3 4 5 2 1
```

Текст «3 4 5» (остаток строки) стал значением переменной `c`.

Команды `shift` используют при последовательной обработке аргументов `shell`-процедур. Например, если требуется выдать аргументы, которые являются именами существующих обычных файлов, употребляют цикл вида (см. листинг 2.28):

```
while [ "$1" ]
do
    [ -f $1 ] && echo $1
    shift
done
```

Листинг 2.28. Пример использования специальной встроенной команды `shift`.

Важную роль играет еще одна специальная встроенная команда — `set`. Она используется для изменения режима работы `shell` и для присваивания новых значений формальным аргументам. Сразу же отметим, что наивное присваивание

```
цифра=слово
```

`shell` трактует как запуск программы с именем `цифра=слово`, т. е. оно не позволяет изменить значение формального аргумента.

Команда `set` записывается в виде

```
set [опция ...] [аргумент ...]
```

Из ее опций упомянем следующие:

```
a
```

Экспортировать в окружение все переменные, которым производится присваивание.

-C

Защищать существующие файлы от перезаписи при перенаправлении вывода посредством конструкции «>>» (см. выше раздел «Перенаправление ввода/вывода»).

+O

Выдать на стандартный вывод текущие опции языка shell в виде команд, которые устанавливают те же значения и которые можно подать ему на вход.

+x

Устанавливает режим трассировки после выполнения всех подстановок в командных строках, но до выполнения они будут выдаваться в стандартный протокол. Опция +x выключает трассировку. Стандарт не специфицирует, будет ли трассироваться команда `set +x`.

Если в команде `set` заданы аргументы, они станут новыми значениями \$1, \$2 и т.д. Типичный пример использования команды `set` содержится в одном из файлов ОС Linux (/etc/rc.d/rc) выполняемых при загрузке системы (см. листинг 2.29).

```
argv1="$1"
set ` /sbin/runlevel `
runlevel=$2
previous=$1
```

Листинг 2.29. Пример использования специальной встроенной команды `set`. (административная утилита `runlevel` выдает предыдущий и текущий уровни выполнения системы).

Возможностью, не обязательной с точки зрения стандарта POSIX-2001 (отнесенной к расширению «Мобильность пользователей», UP), но весьма полезной на практике, является механизм синонимов для имен простых команд. Синонимы задаются с помощью обычной встроенной команды `alias`:

```
alias [имя[=синоним] ...]
```

Когда shell выделяет в командной строке слово, расположенное вместо имени простой команды, он проверяет, не является ли оно именем

синонима, и при положительном ответе заменяет его значением, заданным командой `alias` и представляющим собой цепочку символов.

Например, если хочется, чтобы команда `ls` по умолчанию выдавала информацию о файлах, имена которых начинаются с точки, можно воспользоваться спецификацией для определения синонима:

```
alias ls="ls -a"
```

Если подставленный синоним оканчивается пробелом, то анализируется следующее слово командной строки; если и оно оказывается именем синонима, подстановки продолжаются. Например, если синоним для `ls` задать в виде строки с завершающим пробелом

```
alias ls="ls -a "
```

и дополнительно ввести еще одно определение синонима

```
alias l="-Rl"
```

то команда, использующая синоним с завершающим пробелом,

```
ls l
```

будет раскрыта в строку

```
ls -a -Rl
```

и выдаст в длинном формате информацию о всех файлах текущего каталога и его подкаталогов.

Команда `alias` без аргументов выдает список имеющихся синонимов.

Вызов командного интерпретатора shell

Вызов командного интерпретатора shell осуществляется командой

```
sh [опция...] [командный_файл [аргумент ...]]
```

или

```
sh -c [опция...] командная_цепочка [имя_команды  
{аргумент ...}]
```


или

```
sh -s [опция...] [аргумент ...]
```

В первом случае интерпретируется заданный командный файл (или содержимое стандартного ввода, если файл не указан), во втором – цепочка символов, в третьем команды читаются со стандартного ввода.

Большинство опций команд `sh` и `set` (см. выше) совпадают. Им может предшествовать не только знак минус, но и плюс, что означает инвертирование их смысла. Из специфических опций команды `sh` выделим `-i`, предписывающую считать shell интерактивным. Shell будет интерактивным и тогда, когда команды читаются со стандартного ввода, направленного, как и стандартный протокол, на терминал.

Заданные в командной строке аргументы становятся значениями фактических аргументов `$1`, `$2` и т.д. Если при наличии опции `-c` задано имя_команды, то в результате интерпретации командной цепочки оно становится значением `$0`.

Пример. Команда

```
sh -c 'echo $0 $1 $2' a b c
```

выдаст на стандартный вывод

```
a b c
```

Читателю предлагается самостоятельно определить, что выдаст на стандартный вывод похожая команда

```
sh -c "echo $0 $1 $2" a b c
```

(вместо одиночных кавычек для экранирования пробелов использованы двойные), и объяснить полученный результат.

Командный интерпретатор можно вызвать и из программы на языке C, воспользовавшись функциями `system()` или `popen()` (см. листинг 2.30).

```
#include <stdlib.h>
int system (const char *command);
#include <stdio.h>
FILE *popen (const char *command, const char *mode);
```

Листинг 2.30. Описание функций `system()` и `popen()`.

Аргумент `command` трактуется языком `shell` как командная цепочка в вызове

```
sh -c command
```

и может содержать имя и аргументы любой выполнимой программы. При обращении к `system()` вызвавшая программа ожидает завершения выполнения переданной команды, а затем продолжает выполнение со следующего выполняемого оператора. Возвращаемое функцией `system()` значение — код завершения `shell`. Пример вызова функции `system()`:

```
code = system ("cd /usr/bin; ./ls > /tmp/1st");
```

Функция `popen()`, как и `system()`, вызывает выполнение указанной команды. Отличие в том, что при использовании функции `popen()` создается канал между вызвавшей ее программой и командой. Аргумент `mode` определяет, что программа будет делать с этим каналом: читать ("r") или писать ("w").

Правила формирования и средства разбора командных строк

К теме командного языка и его интерпретатора логически примыкают правила формирования и средства разбора командных строк.

В общем случае командная строка состоит из имени служебной программы (утилиты), опций, аргументов этих опций и, наконец, операндов команды.

Разработчики служебных программ должны руководствоваться следующими правилами.

- 1) Имя утилиты состоит не менее чем из двух и не более чем из девяти малых латинских букв и/или цифр.
- 2) Имя опции — это один буквенно-цифровой символ. Опциям предшествует знак минус. После одного минуса могут располагаться несколько опций без аргументов.
- 3) Опции отделены от своих аргументов.
- 4) У опций нет необязательных аргументов.
- 5) Если у опции несколько аргументов, они представляются одним словом и отделяются друг от друга запятыми или экранированными пробелами.
- 6) Все опции располагаются в командной строке перед операндами.

- 7) Специальный элемент командной строки `--`, который не является ни опцией, ни операндом, обозначает конец опций. Все последующие слова трактуются как операнды, даже если они начинаются со знака минус.
- 8) Порядок разных опций в командной строке не имеет значения. Если повторяется одна опция с аргументами, последние должны интерпретироваться в порядке, указанном в командной строке.
- 9) Порядок интерпретации операндов может зависеть от утилиты.
- 10) Если операнд задает читаемый или записываемый файл, то знак минус на его месте используется только для обозначения стандартного ввода (или стандартного вывода, если из контекста ясно, что специфицируется выходной файл).

Стандартным средством разбора опций и их аргументов, заданных в командной строке, является служебная программа `getopts` (чаще всего реализуемая как обычная встроенная команда shell):

```
getopts цепочка_имен_опций переменная [аргумент ...]
```

Как правило, shell-процедура, запущенная разбираемой командной строкой, вызывает утилиту `getopts` многократно (в цикле). При каждом таком вызове `getopts` помещает имя очередной выделенной ею опции в качестве значения заданной переменной. Место продолжения разбора (индекс в командной строке) запоминается в shell-переменной `OPTIND`, начальным значением которой служит единица.

Если у опции должен быть аргумент (что обозначается двоеточием после имени опции в цепочке имен опций), `getopts` выделяет его из командной строки и помещает в shell-переменную `OPTARG`.

Заданную при вызове `getopts` переменную, а также `OPTIND` и `OPTARG` следует использовать в качестве локальных, они не должны экспортироваться в окружение.

Если при вызове `getopts` указаны аргументы, разбираются они, а не элементы упомянутой выше командной строки.

Когда в командной строке обнаруживается опция, отсутствующая в цепочке имен опций, значением заданной переменной становится знак вопроса. Если первый символ цепочки имен опций — двоеточие, то в `OPTARG` помещается обнаруженный символ, иначе в стандартный протокол выдается диагностическое сообщение. С точки зрения shell-процедуры, вызвавшей `getopts`, это должно считаться ошибкой; с точки зрения `getopts` — нет.

По достижении конца опций утилита `getopts` возвращает положительный код завершения, а в `OPTIND` помещается номер первого элемента командной строки, не являющегося опцией.

Если переменной `OPTIND` присваивается единица, осуществляется (новый) разбор текущей командной строки (сформированной, быть мо-

жет, с помощью специальной встроенной команды `set`) или текущих значений заданных при вызове `getopts` аргументов.

В качестве примера использования служебной программы `getopts` рассмотрим фрагмент `shell`-процедуры `cmd`, где производится разбор заданных при ее вызове опций (см. листинг 2.31).

```
while getopts :abo: c
do
    case $c in
        a | b) FLAG=$c;;
        o)    OARG=$OPTARG;;
        ?)    printf \
                "Использование: %s: [-a | -b]"
                "[ -o выходной_файл] [аргумент ...]\n" $0
                exit 1;;
    esac
done
shift $((OPTARG - 1))
printf "Заданный флаг: %s\n" $FLAG
printf "Аргумент опции o: %s\n" $OARG
printf "Остаток командной строки: %s\n" "$*
```

Листинг 2.31. Пример использования служебной программы `getopts`.

Если вызвать `shell`-процедуру `cmd` любым из способов, показанных в листинге 2.32, будет выдан результат, приведенный в листинге 2.33.

```
cmd -a -o f1.o,f2.o file1 -
cmd -ba -o f1.o,f2.o -- file1 -
cmd -o f1.o,f2.o -b -a file1 -
```

Листинг 2.32. Возможные варианты вызова `shell`-процедуры, использующей служебную программу `getopts`.

```
Заданный флаг: a
Аргумент опции o: f1.o,f2.o
Остаток командной строки: file1 -
```

Листинг 2.33. Возможный результат работы `shell`-процедуры, использующей служебную программу `getopts`.

Для разбора опций и их аргументов средствами языка C служит функция `getopt()` и ассоциированные с ней внешние переменные (см. листинг 2.34).

```
#include <unistd.h>
int getopt (int argc, char *const argv[],
            const char *optstring);
extern char *optarg;
extern int optind, opterr, optopt;
```

Листинг 2.34. Описание функции `getopt()` и ассоциированных с ней внешних переменных.

В общем и целом логика ее работа та же, что и у служебной программы `getopts` (с соответствующим переименованием используемых переменных). Аргументы `argc` и `argv` задают командную строку в том виде, как она передается функции `main()`, `optstring` представляет собой цепочку имен опций. Переменная `optind` (с начальным значением 1) служит индексом в массиве `argv []`. Стандарт POSIX-2001 не специфицирует, как именно `getopt()` разбирает несколько расположенных поочередно (после одного знака минус) имен опций и определяет, какие опции уже обработаны.

В итоге функция `getopt()` возвращает имя очередной опции из числа перечисленных в цепочке `optstring` (если таковое удалось выделить). При наличии у опции аргумента указатель на него помещается в переменную `optarg` с соответствующим увеличением значения `optind`.

Если у опции аргумент отсутствует, а на первом месте в `optstring` задано двоеточие, оно и служит результатом.

Если встретилось имя опции, не перечисленное в `optstring`, или у опции нет аргумента, а на первом месте в `optstring` задано не двоеточие, то результатом станет знак вопроса.

В любой из перечисленных выше ошибочных ситуаций в переменную `optopt` помещается имя «проблемной» опции. Кроме того, в стандартный протокол выдается диагностическое сообщение по образу и подобию утилиты `getopts`. Для подавления выдачи следует присвоить переменной `opterr` нулевое значение.

Наконец, если при вызове `getopt()` указатель `argv[optind]` не отмечает начало опции (например, он пуст или первый символ указуемой цепочки отличен от знака минус), результат равен `-1` как признак того, что разбор опций закончен.

Следующий пример программы (см. листинг 2.35) возвращает нас к рассмотренной выше shell-процедуре `cmd`. Он показывает, как можно

обработать командную строку вызова утилиты, допускающей взаимоисключающие опции а и b, а также опцию о, которая должна иметь аргумент.

```

/* * * * * * * * * * * * * * * * * * * * * * * * * */
/* Программа разбирает опции вызвавшей ее           */
/* командной строки                                   */
/* * * * * * * * * * * * * * * * * * * * * * * * * */
#include <unistd.h>
#include <stdio.h>
int main (int argc, char *argv []) {
    int c;          /* Имя анализируемой опции */
    int aflag = 0;  /* Признак того, что задана */
                    /* опция а */
    int bflag = 0;  /* Признак того, что задана */
                    /* опция b */
    int errflag = 0; /* Флаг наличия ошибки в */
                    /* командной строке */
    int flg = '?';  /* Флаг (а или b), заданный */
                    /* в командной строке */
    char *ofile = NULL; /* Указатель на аргумент */
                        /* опции о */

    /* Подавим стандартную диагностику */
    /* независимо от первого символа */
    /* цепочки имен опций */
    opterr = 0;
    while ((c = getopt (argc, argv, ":abo:")) != -1) {
        switch (c) {
            case 'a':
                aflag++;
                flg = c;
                if (bflag) {
                    fprintf (stderr,
                        "Опции а и b несовместимы\n");
                    errflag++;
                }
                break;
            case 'b':
                bflag++;
                flg = c;
                if (aflag) {
                    fprintf (stderr,
                        "Опции а и b несовместимы\n");

```

```
        errflg++;
    }
    break;
case 'o':
    ofile = optarg;
    break;
case ':':
    fprintf (stderr, "Отсутствует аргумент"
              "опции -%c\n", optopt);
    errflg++;
    break;
case '?':
    fprintf (stderr,
            "Недопустимая опция -%c\n", optopt);
    errflg++;
    break;
}
}
if (errflg) {
    (void) fprintf (stderr, "Использование: %s:"
                  "[-a | -b] [-o выходной_файл] "
                  "[аргумент ...]\n", argv [0]);
    return (1);
}
printf ("Заданный флаг: %c\n", flg);
printf ("Аргумент опции o: %s\n", ofile);
printf ("Остаток командной строки:");
for (; optind < argc; optind++) {
    printf (" %s", argv [optind]);
}
printf ("\n");
return 0;
}
```

Листинг 2.35. Пример использования функции `getopt()`.

Приведенная программа отличается от shell-процедуры `cmd` более тщательным анализом ошибочных ситуаций и детальной диагностикой.

Вариант 1

1. После выполнения строк

a=2*2

echo \$a

на стандартный вывод будет выдано

- ☐ 4
- ☐ 2*2
- ☐ 2<имена файлов текущего каталога>2

2. Выберите корректные конструкции, изменяющие значение первого аргумента shell-процедуры

- ☐ 1=a
- ☐ \${1:=a}
- ☐ set a

3. Значением служебной переменной # служит

- ☐ количество фактических аргументов shell-процедуры
- ☐ совокупность фактических аргументов shell-процедуры
- ☐ это не служебная переменная, а символ начала комментария

Вариант 2

1. После выполнения строк

a=3+3

echo \$a

на стандартный вывод будет выдано

- ☐ 6
- ☐ 3+3
- ☐ не будет выдано ничего, а в стандартный протокол поступит сообщение об ошибке

2. Командную строку 1=a shell трактует как

- ☐ оператор присваивания первому аргументу
- ☐ имя команды
- ☐ синтаксически неверная

3. Значением служебной переменной \$ служит

- ☐ количество фактических аргументов shell-процедуры
- ☐ идентификатор процесса, в рамках которого выполняется shell
- ☐ это не служебная переменная, а символ подстановки значения переменной

Вариант 3

1. После выполнения строк

a=3#3

echo \$a

на стандартный вывод будет выдано

- ☐ 3
- ☐ 3#3
- ☐ не будет выдано ничего, а в стандартный протокол поступит сообщение об ошибке

2. Командную строку #=1 shell трактует как

- ☐ имя команды
- ☐ комментарий
- ☐ синтаксически неверная

3. Совокупность фактических аргументов shell-процедуры является значением служебной переменной

- ☐ *
- ☐ @
- ☐ #

Лекция 3. Утилиты и функции, обслуживающие понятие «пользователь»

Описываются утилиты и функции, предназначенные для получения информации о пользователях и их взаимодействия.

Ключевые слова: пользователь, база данных пользователей, имя пользователя, идентификатор пользователя, начальный рабочий каталог, начальная программа, группа, член группы, начальная группа, дополнительная группа, база данных групп, имя группы, идентификатор группы, смена группы, пароль группы, суперпользователь, сеанс, поиск в базе данных, активный пользователь, терминал, экранный интерфейс, статус терминала, почтовое взаимодействие, почтовый ящик, письмо.

Данные, ассоциированные с пользователем

Операционная система, соответствующая стандарту POSIX, должна поддерживать базу данных пользователей, в которой о каждом из них хранится по крайней мере следующая информация:

- имя пользователя;
- числовой идентификатор пользователя;
- числовой идентификатор начальной группы;
- начальный рабочий каталог;
- начальная программа пользователя.

Поясним смысл перечисленных элементов данных.

Каждый зарегистрированный пользователь ОС имеет имя, которое он указывает для целей идентификации при входе в систему. После проведения идентификации и, как правило, аутентификации пользователя, с ним ассоциируются (неотрицательные) числовые идентификаторы пользователя и начальной группы. В отличие от имен, ОС оперирует ими во всех случаях, кроме первоначальной идентификации. Затем запускается начальная программа пользователя (например, командный интерпретатор shell) с указанным начальным рабочим каталогом.

Поля начального рабочего каталога и начальной программы пользователя могут быть пустыми; в таком случае их трактовка зависит от реализации. Обычно в системе определена подразумеваемая начальная программа, в качестве которой обычно используется `/bin/sh`.

Пользователи объединяются в группы; каждый пользователь является членом хотя бы одной из них. Для групп также существует база данных, ее записи содержат по крайней мере следующие поля:

- имя группы;
- числовой идентификатор группы;
- список пользователей, которым разрешено становиться членами данной группы.

В базе данных пользователей указывается идентификатор начальной группы, в нее пользователь попадает сразу после входа в систему. В процессе работы возможен переход в другую группу (см. далее описание утилиты `newgrp`), однако на эти переходы наложены ограничения в виде списка возможных членов группы.

С объектно-ориентированной точки зрения можно считать, что класс «пользователь» предоставляет один метод — начальную программу. Его можно применить для программирования определенных услуг. Например, если нужно дать возможность любому человеку (не обязательно зарегистрированному пользователю), оказавшемуся рядом со свободным терминалом, узнать текущие дату и время, заводят пользователя `date` с начальной программой `/bin/date`.

Опросить ассоциированные с пользователем данные (точнее, идентификаторы и имена пользователя и начальной группы, а также всех групп, членами которых ему разрешено быть, — так называемых дополнительных групп) позволяет служебная программа

```
id [имя_пользователя]
```

Если `имя_пользователя` опущено, выдается информация о текущем пользователе и его текущей группе.

Пример использования служебной программы `id`. Команда

```
id root
```

может выдать на стандартный вывод следующий результат:

```
uid=0(root) gid=0(root) groups=0(root),1(bin),  
2(daemon),3(sys),4(adm),6(disk),10(wheel)
```

Числовой идентификатор 0 характеризует суперпользователя, который в большинстве Unix-систем не подвержен контролю прав доступа.

Входное имя текущего пользователя можно узнать также с помощью утилиты

```
logname
```

и функции `getlogin()`:

```
#include <unistd.h>
char *getlogin (void);
```

Отметим, что одному идентификатору пользователя может соответствовать несколько записей в базе данных пользователей, различающихся входными именами и, возможно, другими атрибутами; `logname` и `getlogin()` выдают имя, под которым начат текущий сеанс работы с ОС.

Над базой данных пользователей определены операции поиска по идентификатору или имени пользователя, реализуемые, соответственно, функциями `getpuid()` и `getpwnam()` (см. листинг 3.1).

```
#include <pwd.h>
struct passwd *getpuid (uid_t uid);
#include <pwd.h>
struct passwd *getpwnam (const char *name);
```

Листинг 3.1. Описание функций `getpuid()` и `getpwnam()`.

По стандарту структура `passwd` должна содержать по крайней мере следующие поля, соответствующие описанным выше обязательным элементам базы данных пользователей:

```
char *pw_name; /* Имя пользователя */
uid_t pw_uid; /* Числовой идентификатор */
/* пользователя */
gid_t pw_gid; /* Числовой идентификатор */
/* начальной группы */
char *pw_dir; /* Начальный рабочий каталог */
char *pw_shell; /* Начальная программа */
/* пользователя */
```

Типы `uid_t` и `gid_t` определяются в заголовочном файле `<sys/types.h>`.

Приведем пример выдачи информации о текущем пользователе и пользователе `root` с идентификатором 0 (см. листинг 3.2).

```
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>
#include <stdio.h>
/* Печать элемента базы данных пользователей */
static void print_pwent (const struct passwd *pw) {
```

```
printf ("Имя пользователя: %s\n", pw->pw_name);
printf ("Идентификатор пользователя: %d\n",
        pw->pw_uid);
printf ("Идентификатор группы: %d\n", pw->pw_gid);
printf ("Начальный каталог: %s\n", pw->pw_dir);
printf ("Начальная программа: %s\n",
        pw->pw_shell);
}
int main (void) {
    char *lgnm;          /* Имя текущего пользователя */
    struct passwd *pw;    /* Данные о текущем */
                        /* пользователе */
    /* Поиск и печать информации о текущем */
    /*пользователе */
    if ((lgnm = getlogin ()) == NULL ||
        (pw = getpwnam (lgnm)) == NULL) {
        fprintf (stderr, "\nНе удалось найти"
                "информацию о текущем пользователе\n");
        return 1;
    }
    printf ("\nИнформация о текущем пользователе\n");
    print_pwent (pw);
    /* То же для пользователя root */
    if ((pw = getpwuid ((uid_t) 0)) == NULL) {
        fprintf (stderr, "\nНе удалось найти информацию"
                "о пользователе root\n");
        return 1;
    }
    printf ("\nИнформация о пользователе root\n");
    print_pwent (pw);
    return 0;
}
```

Листинг 3.2. Пример работы с базой данных пользователей.

По окончании работы этой программы может быть получен следующий результат (см. листинг 3.3):

```
Информация о текущем пользователе
Имя пользователя: galat
Идентификатор пользователя: 108
Идентификатор группы: 3
```

```
Начальный каталог: /home/galat
Начальная программа: /bin/sh
Информация о пользователе root
Имя пользователя: root
Идентификатор пользователя: 0
Идентификатор группы: 0
Начальный каталог: /root
Начальная программа: /bin/bash
```

Листинг 3.3. Возможный результат работы с базой данных пользователей.

Аналогичные функции имеются для поиска в базе данных групп — `getgrgid()` и `getgrnam()` (см. листинг 3.4).

```
#include <grp.h>
struct group *getgrgid (gid_t gid);
#include <grp.h>
struct group *getgrnam (const char *name);
```

Листинг 3.4. Описание функций `getgrgid()` и `getgrnam()`.

Структура `group` обязана содержать поля

```
char *gr_name; /* Имя группы */
gid_t gr_gid; /* Числовой идентификатор группы */
char **gr_mem; /* Указатель на ограниченный пустым
                указателем массив символьных указателей на
                имена пользователей, которым разрешено
                становиться членами данной группы */
```

Применение функции `getgrgid()` отражено в листинге 3.5.

```
#include <sys/types.h>
#include <grp.h>
#include <stdio.h>
/* Печать списка пользователей, включенных в */
/* группу с заданным идентификатором */
static int print_gr_mem (const gid_t gid) {
    struct group *grp; /* Данные о группе */
    char **c_gr_mem; /* Текущий указатель на */
                    /* имя члена группы */
```

```
char *c_gr_mem_name; /* Текущее имя члена группы */
if ((grp = getgrgid (gid)) == NULL) {
    fprintf (stderr, "\nНе удалось найти информацию"
             "о группе с идентификатором %d\n", gid);
    return 1;
}
printf ("\nПользователи, включенные в группу "
        "с идентификатором %d:\n", gid);
c_gr_mem = grp->gr_mem;
while ((c_gr_mem_name = *c_gr_mem++) != NULL) {
    printf(" %-8.8s", c_gr_mem_name);
}
printf ("\n");
return 0;
}
int main (void) {
    return print_gr_mem (1);
}
```

Листинг 3.5. Пример работы с базой данных групп.

Приведенная в качестве примера программа может привести к результату, показанному в листинге 3.6.

```
Пользователи, включенные в группу с идентификатором 1:
root    bin    daemon
```

Листинг 3.6. Возможный результат работы с базой данных групп.

Для смены текущей группы пользователя предназначена служебная программа `newgrp` (стандарт POSIX-2001 относит ее к числу необязательных, входящих в расширение «Мобильность пользователей», UP):

```
newgrp [-l] [группа]
```

Группа, в которую осуществляется переход, задается именем или числовым идентификатором. Будучи вызванной без аргументов, утилита `newgrp` возвращает пользователя в его начальную группу, заданную в базе данных пользователей.

При смене группы порождается новый процесс, в рамках которого запускается новый экземпляр командного интерпретатора, наследующий у своего предшественника текущий каталог и значения переменных

окружения. При указании опции `-l` окружение формируется так, будто пользователь заново вошел в систему.

Если пользователь не входит в список возможных членов новой группы, при переходе в нее может запрашиваться пароль, однако в стандарте POSIX-2001 этот аспект считается зависящим от реализации. Более того, отмечается, что в базе данных групп нет удобных способов задания паролей, поэтому их использование в приложениях не приветствуется, а со временем пароли групп могут стать ненужными.

Служебные программы, обслуживающие взаимодействие пользователей

Активными мы будем называть пользователей, работающих в системе в некоторый момент времени.

Чтобы узнать, какие пользователи активны и за какими терминалами они работают, можно воспользоваться служебной программой

```
who
```

(заметим, что стандарт POSIX-2001 трактует ее как необязательную, входящую в расширение «Мобильность пользователей»).

Выдача утилиты `who` может выглядеть, например, так (правый столбец означает время входа в систему):

```
galat  ttyS4  Aug 22 12:41
kost   ttyS6  Aug 22 10:09
```

К той же дополнительной категории, что и `who`, принадлежат утилиты `write`, `talk` и `mesg` (а также описанная выше служебная программа `newgrp`).

После установления соединения утилита

```
write имя_пользователя [терминал]
```

позволяет построчно пересылать стандартный ввод отправителя на терминал пользователя-получателя. Аргументы `имя_пользователя` и `терминал` задаются в том виде, как их выводит служебная программа `who`. Необязательный аргумент `[терминал]` нужен в тех случаях, когда пользователь-получатель вошел в систему с нескольких терминалов.

Утилиту

```
talk имя_пользователя [терминал]
```

можно рассматривать как более современный аналог `write`, поскольку она имеет экранный интерфейс и поддерживает двустороннее взаимодействие активных пользователей.

С помощью служебной программы

```
mesg [y|n]
```

пользователь может разрешить или запретить установление соединений со своим терминалом. При вызове без аргументов `mesg` выдает текущий статус терминала.

Приведем пример употребления описанных служебных программ. Рассылку сообщения-аргумента всем активным пользователям можно реализовать посредством `shell`-процедуры (см. листинг 3.7).

```
who | while read a b c
do
    write $a $b << !
    $1 !
done
```

Листинг 3.7. Пример использования утилит `who` и `write`.

Обратим внимание на использование во вставке значения аргумента `shell`-процедуры.

Базовым средством обеспечения почтового взаимодействия между пользователями, согласно стандарту POSIX-2001, является служебная программа `mailx`. Она позволяет и отправлять, и получать письма. В первом случае ее следует вызывать командной строкой

```
mailx {-s тема} адрес ...
```

во втором —

```
mailx -e
```

или

```
mailx [опция ...]
```

или

```
mailx -f [опция ...] [почтовый_ящик]
```

В процессе отправки текст письма читается со стандартного ввода. При получении всеми указанными выше способами, кроме последнего, проверяется системный почтовый ящик, выделенный текущему пользователю, а при наличии опции `-f` в качестве почтового ящика используется явно заданный файл. Опция `-e` предписывает только проверить почтовый ящик и, если он не пуст, вернуть код успеха.

В режиме получения можно отправлять письма и управлять содержимым почтового ящика (для чего утилита `mailx` предоставляет весьма богатый набор команд), однако поддержку этого режима стандарт POSIX-2001 относит к числу необязательных возможностей.

При отправке писем `shell`-процедурами часто пользуются вставками (см. листинг 3.8).

```
address=...
. . .
mailx $address << !
. . .
текст письма
. . .
!
```

Листинг 3.8. Пример использования вставки для формирования письма.

Разумеется, интерактивные пользователи редко применяют утилиту `mailx` напрямую; для работы с почтой практически во всех ОС имеются средства с более дружелюбным интерфейсом.

Вариант 1

1. **Согласно стандарту POSIX, записи базы данных пользователей включают следующие поля**
 - ☐ имя пользователя
 - ☐ зашифрованный пароль пользователя
 - ☐ полное имя пользователя
2. **Согласно стандарту POSIX, записи базы данных групп включают следующие поля**
 - ☐ имя группы
 - ☐ зашифрованный пароль группы
 - ☐ время жизни пароля группы
3. **Согласно стандарту POSIX, с помощью утилиты `logname` можно**
 - ☐ опросить входное имя текущего пользователя
 - ☐ сменить входное имя текущего пользователя
 - ☐ протоколировать имена пользователей, входящих в систему

Вариант 2

1. **Согласно стандарту POSIX, записи базы данных пользователей включают следующие поля**
 - ☐ числовой идентификатор пользователя
 - ☐ числовой идентификатор начальной группы
 - ☐ начальный рабочий каталог
2. **Согласно стандарту POSIX, записи базы данных групп включают следующие поля**
 - ☐ число пользователей в группе
 - ☐ числовой идентификатор группы
 - ☐ число групп в базе
3. **Согласно стандарту POSIX, к числу обязательных принадлежат функции**
 - ☐ получения входного имени текущего пользователя
 - ☐ получения пароля текущего пользователя в открытом виде
 - ☐ получения зашифрованного пароля текущего пользователя

Вариант 3

1. Согласно стандарту POSIX, записи базы данных пользователей включают следующие поля

- ☐ начальная программа пользователя
- ☐ рабочая программа пользователя
- ☐ конечная программа пользователя

2. Согласно стандарту POSIX, записи базы данных групп включают следующие поля

- ☐ список групп, в чей состав входит данный пользователь
- ☐ список числовых идентификаторов пользователей, которым разрешено становиться членами данной группы
- ☐ список имен пользователей, которым разрешено становиться членами данной группы

3. Согласно стандарту POSIX, служебная программа id

- ☐ осуществляет тождественное преобразование
- ☐ выдает информацию о пользователе
- ☐ выдает информацию о текущей группе пользователя

Лекция 4. Организация файловой системы

Определяются основные понятия, ассоциированные с файлами. Описывается структура файловой системы, а также утилиты и функции для опроса и изменения атрибутов файлов и файловых систем, создания, удаления, копирования и перемещения файлов, обхода и обработки файловых иерархий.

Ключевые слова: файл, чтение, запись, тип файла, обычный файл, каталог, канал, символьный специальный файл, блочный специальный файл, символьная ссылка, раскрытие символьной ссылки, сокет, элемент каталога, имя файла, простое имя, описатель файла, файловая система, корневой каталог, уникальный идентификатор файла, порядковый номер файла, маршрутное имя, составное имя, абсолютное маршрутное имя, текущий каталог, вышележащий каталог, относительное маршрутное имя, маршрутный префикс, разрешение маршрутных имен, режим файла, биты режима, режим доступа к файлу, биты режима доступа к файлу, владелец файла, владеющая группа, прочие пользователи, право на чтение, право на запись, право на выполнение, право на поиск в каталоге, дополнительный механизм контроля доступа, альтернативный механизм контроля доступа, открытый файл, файловый дескриптор, описание открытого файла, монтирование, точка монтирования, объединенная файловая система, жесткая ссылка, рекурсивный обход, идентификатор устройства, старший номер устройства, младший номер устройства, пространство, использованное пространство, свободное пространство, процент занятости, смена текущего каталога, изменение атрибутов файлов, побитные операции, файл-замок, создание файлов, удаление файлов, копирование файлов, перемещение файлов, переименование файлов, код ошибки, запрос подтверждения, логическое выражение, дисциплина обхода иерархии, операция отрицания, логическое И, логическое ИЛИ, архивирование, архивный формат, извлечение файлов из архива.

Основные понятия

В трактовке стандарта POSIX понятие файла охватывает все, что может содержать, потреблять и/или поставлять информацию. В объектно-ориентированной терминологии файл должен предоставлять методы для чтения и/или записи и иметь такие атрибуты, как тип, имя и режим.

В стандарте зафиксированы следующие типы файлов:

- обычный файл;
- каталог;
- канал;
- символьный специальный файл;
- блочный специальный файл;
- символьная ссылка;
- сокет.

Обычный файл представляет собой последовательность байт с возможностью случайного доступа и без какой-либо дополнительной структуры, наложенной операционной системой.

Каталог состоит из элементов (ссылок), ассоциирующих имена с файлами. Одинаковые имена не могут фигурировать в разных элементах одного каталога, но разные элементы могут ссылаться на один и тот же файл.

Канал можно представлять себе в виде транспортера, с одной стороны которого находится поставщик (процесс, пишущий в канал), а с другой – потребитель (процесс, читающий из канала). Данные читаются в том же порядке, в каком производилась запись, т. е. с точки зрения структур данных канал – это очередь.

Специальные файлы соответствуют аппаратным компонентам компьютера. Обычно при использовании символьных специальных файлов остаются видимыми аппаратные характеристики соответствующих устройств, а при доступе к устройствам посредством блочных специальных файлов аппаратные характеристики, как правило, остаются скрытыми. Типичный пример устройства, которому соответствует символьный специальный файл, – терминал.

Символьная ссылка – это файл, хранящий цепочку символов. Когда подобный файл является компонентом маршрутного имени (см. далее), хранящая цепочка символов влияет на результирующий маршрут. Обычно ссылка раскрывается (прозрачным для приложений образом), т. е. вместо ее имени подставляется содержимое (цепочка символов).

Сокет – это оконечная точка межпроцессных коммуникаций.

Файлы вместе со служебной информацией, хранящейся в объектах, которые называются описателями файлов, объединяются в иерархическую структуру (направленный граф), именуемую файловой системой. Все неконцевые вершины графа (т. е. вершины, откуда выходит хотя бы одна дуга) являются каталогами; все концевые имеют другие типы.

Согласно стандарту POSIX-2001, для каждого процесса определен корневой каталог с именем `/` – вершина графа, из которой осуществляется доступ к прочим файлам данной файловой системы. «Абсолютного», единого для всех корня может и не существовать.

В пределах файловой системы каждый файл имеет уникальный идентификатор (порядковый номер — он же номер описателя файла).

Имена, ассоциируемые с файлами (точнее, с их порядковыми номерами) посредством каталогов, называются простыми. С формальной точки зрения они могут включать любые символы, кроме слэша (/), однако для повышения мобильности приложений стандарт рекомендует ограничиться латинскими буквами, цифрами, а также символами ., _ , — (точка, подчеркивание, минус), причем минус не должен быть первым символом. В соответствующей стандарту операционной системе сохраняется различие между большими и малыми буквами.

(При использовании в именах файлов некоторых других символов довольно часто возникают проблемы. Укажем лишь на часть из них.

Символы национальных алфавитов создают трудности с отображением имен. Символы, имеющие специальный смысл для командного интерпретатора, могут вызвать неожиданные и нежелательные модификации имен. Символ : (двоеточие) в имени каталога способен нарушить нормальную интерпретацию значения переменной окружения PATH. Минус в качестве первого символа придает имени файла вид комбинации опций, нарушая процесс разбора аргументов утилит. И т.д. и т.п.).

В каждом каталоге должны присутствовать имена . (точка) и . . (по стандарту читается как точка-точка), которые трактуются как ссылки на текущий и вышележащий каталоги. Для корневого каталога имя . . может также ссылаться на корень.

Под маршрутным именем понимается цепочка символов, идентифицирующая файл и состоящая из нескольких (в том числе из нуля) простых имен файлов, разделенных символами / и называемых компонентами маршрута. Маршрутные имена, начинающиеся символом /, называются абсолютными, они задают маршрут от корня файловой системы. (Отметим, что стандарт POSIX-2001 определяет трактовку маршрутных имен, начинающихся с двух символов /, как зависящую от реализации. В частности, таким способом в некоторых ОС задаются имена файлов на других хостах сети.)

Прочие имена называются относительными, маршрут в них задан относительно текущего каталога.

Маршрутным префиксом называется маршрутное имя, ссылающееся на каталог. Если маршрутное имя оканчивается символом /, оно обязано быть маршрутным префиксом.

Маршрутные имена называют также составными именами файлов. Примеры составных имен:

```
./autofsck  
/usr/lib/tcl
```

```
../susv3  
./l4.txt
```

Первые два имени являются абсолютными. Файл `.autofsck` лежит в корневом каталоге, файл `tel` — в подкаталоге `lib` подкаталога `usr` корневого каталога. Третье и четвертое — относительные имена, специфицирующие, соответственно, файл `susv3`, расположенный в иерархии на один уровень выше файлов текущего каталога, и файл `l4.txt` из текущего каталога. Последнее маршрутное имя можно было задать и проще:

```
l4.txt
```

Стандарт POSIX определяет процесс разрешения маршрутных имен, компонентами которых являются символьные ссылки. Если такая ссылка — последний компонент, а затребованная функция относится к самой ссылке, имя остается без изменений. В противном случае ссылка раскрывается: вместо соответствующего компонента подставляется хранимая цепочка символов, и процесс разрешения маршрутного имени продолжается. Если он заикливается или нарушает какие-либо ограничения (например, на длину маршрутного имени или количество раскрытых ссылок), операционная система должна зафиксировать ошибку.

Согласно стандарту, с каждым файлом ассоциирована по крайней мере следующая служебная информация:

- режим — объект, содержащий биты режима и тип файла;
- числовой идентификатор владельца-пользователя;
- числовой идентификатор владеющей группы.

Типы файлов мы рассматривали выше. Биты режима подразделяются на биты режима доступа, бит переустановки идентификатора пользователя (этот и следующий биты будут описаны при рассмотрении процессов), бит переустановки идентификатора группы, а также бит ограничения на удаление элементов каталогов (последняя возможность относится к числу необязательных).

Биты режима доступа, вместе с другими атрибутами, определяют, имеет ли процесс, действующий от имени некоторого пользователя, право на соответствующие операции с файлом.

По отношению к конкретному файлу все пользователи делятся на три категории:

- владелец файла;
- члены владеющей группы;
- прочие пользователи.

Для каждой из них режим доступа определяет права на операции с файлом, а именно:

- право на чтение;
- право на запись;
- право на выполнение (для каталогов — право на поиск).

Указанных видов прав достаточно, чтобы определить допустимость любой операции с файлами. Например, для удаления файла необходимо иметь право на запись в соответствующий каталог (если не используется бит ограничения на удаление элементов каталогов).

Стандарт POSIX предусматривает реализацию дополнительных и/или альтернативных механизмов контроля доступа в операционной системе. Дополнительные механизмы ужесточают стандартный контроль, альтернативные заменяют его.

Выполнение большинства операций с файлами требует их открытия. Открытому файлу соответствует файловый дескриптор — неотрицательное целое число, уникальное в пределах процесса и используемое для целей доступа к файлу. Дескриптор является ссылкой на описание открытого файла, хранящее смещение в файле, его статус, режимы доступа. Разные дескрипторы могут ссылаться на одно описание.

Вероятно, все версии ОС Unix поддерживают понятие монтирования — отождествления корня дополнительной файловой системы с некоторым каталогом в существующей иерархии (этот каталог называется точкой монтирования) и формирования таким образом объединенной файловой системы. В стандарте POSIX-2001 понятие монтирования, равно как и необходимые для его поддержки служебные программы и функции, отсутствуют. Причина — зависимость от реализации.

Получение информации о файлах и файловых системах

Чтобы узнать абсолютное маршрутное имя текущего каталога, приложение может воспользоваться упоминавшейся ранее обычной встроенной командой языка shell

```
pwd [-L | -P]
```

или функцией `getcwd()` (см. листинг 4.1).

```
#include <unistd.h>
char *getcwd (char *buf, size_t size);
```

Листинг 4.1. Описание функции `getcwd()`.

Команда `pwd` с (подразумеваемой) опцией `-L` извлекает имя текущего каталога из переменной окружения `PWD`, если это возможно. Посредством опции `-P` выполняется раскрытие символьных ссылок: в выдаваемом на стандартный вывод абсолютном маршрутном имени вместо символьных ссылок подставляется их содержимое.

Функция `getcwd()` помещает абсолютное маршрутное имя текущего каталога (без символьных ссылок) в массив `buf` длины `size`, который и возвращается в качестве результата (при ошибке результат равен `NULL`).

Приведем пример программы, использующей функцию `getcwd()` (см. листинг 4.2).

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
int main (void) {
    size_t size;
    char *buf;
    char *apath;
    /* Выясним, каким должен быть размер буфера */
    /* для абсолютного маршрутного имени текущего */
    /* каталога */
    size = (size_t) pathconf (".", _PC_PATH_MAX);
    if ((buf = (char *) malloc (size)) == NULL) {
        fprintf (stderr, "\nНе удалось выделить буфер"
                "размера %d\n", size);
        return 1;
    }
    if ((apath = getcwd (buf, size)) == NULL) {
        fprintf (stderr, "\nНе удалось определить"
                "абсолютное маршрутное имя текущего каталога\n");
        return 1;
    }
    printf ("\nАбсолютное маршрутное имя текущего"
            "каталога: %s\n", apath);
    return 0;
}
```

Листинг 4.2. Пример использования функции `getcwd()`.

Обратим внимание на применение функции `pathconf()`, возвращающей текущее значение одного из конфигурируемых параметров целевой системы – в данном случае максимальную длину маршрутного имени. Подобный прием характерен для мобильных приложений.

Для выдачи на стандартный вывод информации о файлах всех типов служит утилита

```
ls [опция ...] [файл ...]
```

(напомним, что в квадратные скобки заключаются необязательные аргументы). Как правило, если в качестве файла задано имя каталога, то выводится информация обо всех содержащихся в нем файлах.

Опции управляют порядком и степенью подробности выдаваемой информации о файлах. Если опции не заданы, выводятся только имена файлов. Если не заданы файлы, выдается информация о файлах текущего каталога. Опция `-l` предписывает выводить подробную информацию. Например, при использовании служебной программы `ls` по команде

```
ls -l /
```

может быть выдано следующее (см. листинг 4.3):

```
total 338
drwxrwxr-x 2 root root 4096 Jul 6 13:31 a
drwxr-xr-x 2 root root 4096 Jul 8 13:32 bin
drwxr-xr-x 4 root root 1024 Jul 6 15:06 boot
drwxrwxr-x 2 root root 4096 Jul 6 13:31 cdrom
drwxr-xr-x 18 root root 86016 Sep 2 10:42 dev
drwxr-xr-x 6 root root 4096 Feb 23 2003 dss
drwxr-xr-x 60 root root 4096 Sep 2 10:42 etc
drwxr-xr-x 20 root root 4096 Jul 15 18:00 home
drwxr-xr-x 2 root root 4096 Jun 12 2001 initrd
drwxr-xr-x 8 root root 4096 Aug 8 13:16 lib
drwx----- 2 root root 16384 Jul 6 14:15 lost+found
drwxr-xr-x 2 root root 4096 Apr 1 2002 misc
drwxr-xr-x 5 root root 4096 Jul 6 11:24 mnt
drwxr-xr-x 7 root root 4096 Jul 23 13:09 opt
dr-xr-xr-x 84 root root 0 Sep 2 2003 proc
drwxr-x--- 5 root root 4096 Sep 1 17:10 root
drwxr-xr-x 2 root root 4096 Jul 11 17:38/sbin
drwxrwxrwt 23 root root 8192 Sep 2 12:43 tmp
drwxr-xr-x 14 root root 4096 Jul 15 12:50 usr
drwxr-xr-x 16 root root 4096 Jul 9 16:22 var
```

Листинг 4.3. Возможный результат использования служебной программы `ls`.

Число в первой строке есть суммарный размер (в блоках по 512 байт) всех файлов каталога, информация о которых выдана. Далее следуют строки с информацией об отдельных файлах. Первый символ в этих строках задает тип файла:

- d - каталог;
- b - блочный специальный файл;
- c - символьный специальный файл;
- l - символьная ссылка;
- p - канал;
- (минус) - обычный файл.

Отметим, что стандартом POSIX-2001 не предусмотрено обозначение для сокетов, указано только, что реализации могут вводить свои типы файлов и обозначения для них.

Девять последующих символов отражают режим доступа к файлу: первые три символа - права доступа его владельца, следующие три - владеющей группы, последние три - права доступа прочих пользователей. Наличие буквы r (чтение), w (запись) или x (выполнение/поиск) означает, что соответствующее право имеется; знак минус свидетельствует об отсутствии права. Например, файл /usr является каталогом, куда может писать только пользователь root, а читать и искать информацию - все остальные.

Следом идет число ссылок на файл (считаются только так называемые жесткие ссылки, т. е. количество вхождений имени файла в каталоги). Для каталога оно заведомо не меньше двух (обоснование этого факта предоставляется читателю).

Следующие два столбца - имена владельца файла и владеющей группы, после чего идет размер файла в байтах. Для специальных файлов вместо размера нередко выдается зависящая от реализации информация о соответствующем устройстве.

Наконец, следуют дата и время последнего изменения и имя файла. По умолчанию имена файлов сортируются в алфавитном порядке.

Опишем еще несколько употребительных опций служебной программы ls.

-l

Вывести список всех файлов (обычно не выводятся данные о файлах, имена которых начинаются с точки).

-C

Выдавать имена файлов в несколько столбцов (с сортировкой по столбцам). Отметим, что если в командной строке встречаются пары

взаимоисключающих опций (например, `--lC`), то действует та, что указана последней.

`-d`

Трактовать каталоги наравне с файлами других типов. Часто используется с опцией `-l` для получения сведений о состоянии каталога.

`-F`

Выводить символ `/` после имен каталогов, `*` — после выполнимых файлов, `|` — после каналов, `@` — после символьных ссылок.

`-i`

Выдавать порядковый номер файла в файловой системе (см. выше).

`-R`

Рекурсивно обойти встретившиеся подкаталоги.

`-r`

Изменить порядок сортировки файлов на обратный.

`-t`

Использовать в качестве первичного ключа сортировки время последнего изменения (сначала идут самые свежие файлы); имя служит вторичным ключом.

Обычно символьные ссылки трактуются утилитой `ls` наравне с файлами других типов, только на месте имени выводится комбинация

`имя_файла-ссылки -> содержимое_файла-ссылки`

Для получения информации о файле, на который указывает символьная ссылка, следует воспользоваться одной из следующих опций.

`-L`

Если символьная ссылка является аргументом утилиты `ls` или встретилась в процессе обхода файловой иерархии, выдавать информацию об

указуемом файле, а не о ссылке. На месте имени выводится имя файла-ссылки (а не указуемого файла).

-H

Аналогична -L, но воздействует только на символьные ссылки, заданные в командной строке и указывающие на каталог.

Приведем еще несколько примеров употребления утилиты `ls`. После выполнения команд, показанных в листинге 4.4, наша Linux-система выдает информацию определенного вида (см. листинг 4.5):

```
cd /usr/share/zoneinfo/posix
ls -il UTC Universal Zulu
```

Листинг 4.4. Еще один пример использования служебной программы `ls`.

```
2506763 -rw-r--r-- 6 root root 56 Apr 15 2002 UTC
2506763 -rw-r--r-- 6 root root 56 Apr 15 2002 Universal
2506763 -rw-r--r-- 6 root root 56 Apr 15 2002 Zulu
```

Листинг 4.5. Возможный результат использования служебной программы `ls`.

Как и предписывает опция `-i`, выданы порядковые номера (уникальные идентификаторы) файлов. Мы видим, что для всех трех имен они одинаковы (2506763), т. е. эти имена ссылаются на один и тот же файл. Поскольку выданное утилитой `ls` число ссылок равно шести, в файловой системе имеются еще три (жесткие) ссылки на данный файл.

Полезно сопоставить команды и результаты, показанные в листингах 4.4 и 4.5, с теми, что приведены в листингах 4.6 и 4.7.

```
cd /bin
ls -il cx rvi rview vi view
```

Листинг 4.6. Применение служебной программы `ls` для выдачи информации о символьных ссылках.

```
4849722 lrwxrwxrwx 1 root root 2 Jul 6 14:17 ex -> vi
4849723 lrwxrwxrwx 1 root root 2 Jul 6 14:17 rvi -> vi
4849724 lrwxrwxrwx 1 root root 2 Jul 6 14:17 rview -> vi
```



```
4849725 -rwxr-xr-x 1 root root 386120 Mar 8 2002 vi
4849726 lrwxrwxrwx 1 root root 2 Jul 6 14:17 view -> vi
```

Листинг 4.7. Возможный результат использования служебной программы `ls` при выдаче информации о символьных ссылках.

Все пять файлов – аргументов командной строки (четыре символьные ссылки и один обычный файл) имеют разные порядковые номера и, следовательно, являются разными файлами (на каждый имеется лишь по одной жесткой ссылке). Для символьных ссылок выдана информация о них, а не об указываемых файлах. Если же добавить опцию `-L`, мы получим сведения об указываемом файле (`vi`) в пяти экземплярах, но на месте имен будут фигурировать имена символьных ссылок (см. листинг 4.8).

```
4849725 -rwxr-xr-x 1 root root 386120 Mar 8 2002 ex
4849725 -rwxr-xr-x 1 root root 386120 Mar 8 2002 rvi
4849725 -rwxr-xr-x 1 root root 386120 Mar 8 2002 rview
4849725 -rwxr-xr-x 1 root root 386120 Mar 8 2002 vi
4849725 -rwxr-xr-x 1 root root 386120 Mar 8 2002 view
```

Листинг 4.8. Возможный результат использования служебной программы `ls` при выдаче информации о символьных ссылках и наличии опции `-L`.

Данные о специальных файлах (см. листинг 4.9) могут выглядеть так, как показано в листинге 4.10.

```
ls -l /dev/tty[0-3] /dev/fd[2-5]
```

Листинг 4.9. Пример использования служебной программы `ls` для выдачи информации о специальных файлах.

```
brw-rw---- 1 root floppy 2, 2 Apr 1 2002 /dev/fd2
brw-rw---- 1 root floppy 2, 3 Apr 1 2002 /dev/fd3
brw-rw---- 1 root floppy 2, 128 Apr 1 2002 /dev/fd4
brw-rw---- 1 root floppy 2, 129 Apr 1 2002 /dev/fd5
crw-rw-rw- 1 root root 3, 0 Sep 2 17:21 /dev/tty0
crw-rw-rw- 1 root tty 3, 1 Apr 1 2002 /dev/tty1
crw-rw-rw- 1 root tty 3, 2 Apr 1 2002 /dev/tty2
crw-rw-rw- 1 root tty 3, 3 Apr 1 2002 /dev/tty3
```

Листинг 4.10. Возможный результат использования служебной программы `ls` при выдаче информации о специальных файлах.

Для специальных файлов вместо размера выдаются так называемые старший и младший номера, однозначно определяющие соответствующее устройство; наличие такой возможности, разумеется, зависит от реализации.

Сведения о файлах-каналах могут выглядеть так, как показано в листинге 4.11.

```
total 0
prw--w---- 1 root      root 0 Sep 2 10:42 xdmctl
prw--w---- 1 shmyrev  root 0 Sep 2 10:42 xdmctl-:0
```

Листинг 4.11. Возможный результат использования служебной программы `ls` при выдаче информации о каналах.

В листинге 4.12 приведен пример вывода информации о сокетах.

```
srwx----- 1 shmyrev  root 0 Sep 2 10:42 /dev/gpmctl
srw-rw-rw- 1 root      root 0 Sep 2 10:42 /dev/log
```

Листинг 4.12. Возможный результат использования служебной программы `ls` при выдаче информации о сокетах.

Опция `-t` позволяет увидеть сначала те файлы, которые изменялись позже других. Например, командная строка служебной программы `ls`

```
ls -tl /
```

может выдать следующую информацию (см. листинг 4.13):

```
total 338
drwxrwxrwt 23 root root 8192 Sep 2 16:26 tmp
dr-xr-xr-x 143 root root 0 Sep 2 14:42 proc
drwxr-xr-x 60 root root 4096 Sep 2 10:42 etc
drwxr-xr-x 18 root root 86016 Sep 2 10:42 dev
drwxr-x--- 5 root root 4096 Sep 1 17:10 root
drwxr-xr-x 8 root root 4096 Aug 8 13:16 lib
drwxr-xr-x 7 root root 4096 Jul 23 13:09 opt
drwxr-xr-x 20 root root 4096 Jul 15 18:00 home
drwxr-xr-x 14 root root 4096 Jul 15 12:50 usr
drwxr-xr-x 2 root root 4096 Jul 11 17:38/sbin
drwxr-xr-x 16 root root 4096 Jul 9 16:22 var
drwxr-xr-x 2 root root 4096 Jul 8 13:32 bin
drwxr-xr-x 4 root root 1024 Jul 6 15:06 boot
```

```

drwx----- 2 root    root 16384 Jul  6 14:15  lost+found
drwxrwxr-x  2 root    root  4096 Jul  6 13:31  cdrom
drwxrwxr-x  2 root    root  4096 Jul  6 13:31  a
drwxr-xr-x  5 root    root  4096 Jul  6 11:24  mnt
drwxr-xr-x  6 root    root  4096 Feb 23 2003  dss
drwxr-xr-x  2 root    root  4096 Apr  1 2002  misc
drwxr-xr-x  2 root    root  4096 Jun 12 2001  initrd

```

Листинг 4.13. Возможный результат использования служебной программы `ls` с сортировкой файлов по времени последнего изменения.

Видеть в первую очередь самые свежие файлы полезно в тех случаях, когда программная система перестала работать и нужно найти причину поломки, а для этого требуется прежде всего точно узнать, что же, собственно, изменилось.

При программировании на языке C информацию о файлах получают с помощью функций семейства `stat()` (см. листинг 4.14).

```

#include <sys/stat.h>
int stat (const char *restrict path,
          struct stat *restrict buf);
#include <sys/stat.h>
int fstat (int fildes, struct stat *buf);
#include <sys/stat.h>
int lstat (const char *restrict path,
           struct stat *restrict buf);

```

Листинг 4.14. Описание функций семейства `stat()`.

Обратим внимание на использование в описании функций `stat()` и `lstat()` ключевого слова `restrict` из репертуара C99. Оно означает, что по указателям-аргументам доступ осуществляется к непересекающимся областям памяти. Подобная спецификация расширяет оптимизационные возможности компилятора.

Функция `stat()` предоставляет информацию о поименованном файле: аргумент `path` указывает на маршрутное имя файла. Чтобы получить эти сведения, достаточно иметь право на поиск для всех компонентов маршрутного префикса. Функция `fstat()` сообщает данные об открытом файле, задаваемом дескриптором файла `fildes`. Функция `lstat()` эквивалентна `stat()` за одним исключением: если аргумент `path` задает символьную ссылку, `lstat()` возвращает информацию о ней, а `stat()` — о файле, на который эта ссылка указывает.

В случае нормального завершения результат функций семейства `stat()` равен 0.

Аргумент `buf` является указателем на структуру типа `stat`, в которую помещается информация о файле. Согласно стандарту POSIX-2001, в ней содержатся по крайней мере следующие поля:

```
dev_t      st_dev;    /* Идентификатор устройства, */
              /* содержащего файл */
ino_t      st_ino;    /* Порядковый номер файла в */
              /* файловой системе */
mode_t      st_mode;  /* Режим файла */
nlink_t     st_nlink; /* Число жестких ссылок на */
              /* файл */
uid_t      st_uid;    /* Идентификатор владельца */
              /* файла */
gid_t      st_gid;    /* Идентификатор владеющей */
              /* группы */
off_t      st_size;   /* Для обычных файлов и */
              /* символьных ссылок - размер в байтах */
              /* Для файлов других типов значение */
              /* этого поля неспецифицировано */
time_t      st_atime;  /* Время последнего доступа */
time_t      st_mtime;  /* Время последнего */
              /* изменения файла */
time_t      st_ctime;  /* Время последнего */
              /* изменения статуса файла */
```

Некоторые пояснения. Комбинация значений (`st_dev`, `st_ino`) должна однозначно определять файл в пределах объединенной (в том числе сетевой) файловой системы. Статус файла меняется, когда модифицируются его атрибуты (например, режим), а не содержимое.

В файле `<sys/stat.h>` определена не только структура `stat`, но и константы, полезные для работы с битами режима. Так, `S_IFMT` выделяет тип файла, `S_IFREG` обозначает обычные файлы, `S_IRWXU` – биты режима доступа владельца и т.д.

Приведем пример использования функций `stat()` и `lstat()` (см. листинг 4.15).

```
/* Программа выдает информацию о файлах - */
/* аргументах командной строки */
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <stdio.h>

/* Функция возвращает односимвольное      */
/* обозначение типа файла                  */
/* Для неизвестного типа возвращается 'u' */
static char my_filetype (const mode_t mode) {
    switch (mode & S_IFMT) {
        case S_IFDIR:
            return ('d');
        case S_IFBLK:
            return ('b');
        case S_IFCHR:
            return ('c');
        case S_IFLNK:
            return ('l');
        case S_IFIFO:
            return ('p');
        case S_IFREG:
            return ('-');
        case S_IFSOCK:
            return ('s');
        default:
            return ('u');
    }
}

int main (int argc, char *argv[]) {
    struct stat buf;
    int i;

    for (i = 1; i < argc; i++) {
        if (stat (argv [i], &buf)) {
            fprintf (stderr, "\nstat: не удалось получить"
                    "информацию о файле %s\n", argv [i]);
            return (-1);
        }

        printf ("\nstat-информация о файле %s:\n",
                argv [i]);
        printf ("Тип: %c\n",
                my_filetype (buf.st_mode));
        printf ("Размер: %ld\n", buf.st_size);
    }
}
```

```
if (lstat (argv [i], &buf)) {
    fprintf (stderr, "\nlstat: не удалось получить"
            "информацию о файле %s\n", argv [i]);
    return (-1);
}

printf ("\nlstat-информация о файле %s:\n",
        argv [i]);
printf ("Тип: %c\n",
        my_filetype (buf.st_mode));
printf ("Размер: %ld\n", buf.st_size);
}
return 0;
}
```

Листинг 4.15. Пример использования функций `stat()` и `lstat()`.

Возможный результат работы этой программы с аргументами `/bin/view` и `/bin/vi` показан в листинге 4.16.

```
stat-информация о файле /bin/view:
Тип: -
Размер: 386120
lstat-информация о файле /bin/view:
Тип: l
Размер: 2
stat-информация о файле /bin/vi:
Тип: -
Размер: 386120
lstat-информация о файле /bin/vi:
Тип: -
Размер: 386120
```

Листинг 4.16. Возможный результат работы программы, использующей функции `stat()` и `lstat()`.

Для получения интегральной информации о файловых системах, содержащих заданные файлы, следует воспользоваться служебной программой

```
df [опция ...] [файл ...]
```

или функциями `fstatvfs()` или `statvfs()` (см. листинг 4.17).

```
#include <sys/statvfs.h>
int fstatvfs (int fildes, struct statvfs *buf);
int statvfs (const char *restrict path,
             struct statvfs *restrict buf);
```

Листинг 4.17. Описание функций `fstatvfs()` и `statvfs()`.

Утилита `df` выдает на стандартный вывод информацию об имеющемся пространстве и, возможно, некоторые другие данные. Если файлы-аргументы не указаны, поступают сообщения обо всех компонентах объединенной файловой системы.

По умолчанию пространство измеряется в 512-байтных блоках; опция `-k` предписывает использовать блоки размером в килобайт. Подразумываемый формат вывода не специфицирован, но должны становиться известными по крайней мере имя файловой системы и объем свободного пространства. Если задана опция `-P`, употребляется так называемый мобильный формат вывода с заголовком

```
Filesystem 512-blocks Used Available Capacity Mounted on
```

и следующей построчной информацией о каждой файловой системе:

- имя файловой системы;
- общее пространство;
- занятое пространство;
- свободное пространство;
- процент занятости;
- корень файловой системы (точка монтирования в объединенной файловой системе).

Возможный результат выполнения служебной программы `df` с опциями `-kP` приведен в листинге 4.18.

```
Filesystem 1024-blocks Used      Available Capacity Mounted on
/dev/sda3  75822432  43456504  28514348   61%      /
/dev/sda1  46636     21129    23099      48%      /boot
none       257212    0         257212     0%       /dev/shm
/dev/fd0   1424      392      1032       28%      /a
```

Листинг 4.18. Возможный результат выполнения командной строки `df -kP`.

Пара функций `statvfs()` и `fstatvfs()` по интерфейсу аналогична функциям `stat()` и `fstat()`, только в выходную структуру (типа `statvfs`) помещается информация не о файлах, а о файловых системах, содержащих эти файлы. Согласно стандарту POSIX-2001, в структуре `statvfs` должны присутствовать по крайней мере следующие поля:

```
unsigned long f_bsize; /* Размер блока файловой */
                      /* системы */
unsigned long f_frsize; /* Базовый размер блока */
                      /* файловой системы */
fsblkcnt_t    f_blocks; /* Общее число блоков */
                      /* базового размера в файловой системе */
fsblkcnt_t    f_bfree; /* Общее число свободных */
                      /* блоков */
fsblkcnt_t    f_bavail; /* Число свободных блоков, */
                      /* доступных непривилегированным процессам */
fsfilcnt_t    f_files; /* Общее число описателей */
                      /* файлов */
fsfilcnt_t    f_ffree; /* Общее число свободных */
                      /* описателей файлов */
fsfilcnt_t    f_favail; /* Число описателей файлов, */
                      /* доступных непривилегированным процессам */
unsigned long f_fsid; /* Идентификатор файловой */
                      /* системы */
unsigned long f_flag; /* Битная шкала флагов */
unsigned long f_namemax; /* Максимальная длина */
                      /* имени файла */
```

Среди флагов, входящих в шкалу `f_flag`, выделим `ST_RDONLY` — признак того, что файловая система смонтирована только на чтение.

В листинге 4.19 приведен исходный текст программы, которая выдает сведения о файловых системах, содержащих заданные в командной строке файлы-аргументы.

```
#include <sys/statvfs.h>
#include <stdio.h>

int main (int argc, char *argv[]) {
    struct statvfs buf;
    int i;

    for (i = 1; i < argc; i++) {
```



```
if (statvfs (argv [i], &buf)) {
    fprintf (stderr, "\nstatvfs: не удалось"
        "получить информацию о файловой системе,"
        "содержащей файл %s\n", argv [i]);
    return (-1);

    printf ("\nstatvfs-информация о файловой"
        "системе, содержащей файл %s:\n", argv [i]);
    printf ("Размер блока файловой системы:"
        "%ld\n", buf.f_bsize);
    printf ("Базовый размер блока файловой"
        "системы: %ld\n", buf.f_frsize);
    printf ("Общее число блоков базового размера"
        "в файловой системе: %ld\n", buf.f_blocks);
    printf ("Общее число свободных блоков: %ld\n",
        buf.f_bfree);
    printf ("Число свободных блоков, доступных"
        "непривилегированным процессам: %ld\n",
        buf.f_bavail);
    printf ("Общее число описателей файлов:"
        "%ld\n", buf.f_files);
    printf ("Общее число свободных описателей"
        "файлов: %ld\n", buf.f_ffree);
    printf ("Число описателей файлов, доступных"
        "непривилегированным процессам: %ld\n",
        buf.f_favail);
    printf ("Идентификатор файловой системы:"
        "%ld\n", buf.f_fsid);
    printf ("Битная шкала флагов: %lx\n",
        buf.f_flag);
    printf ("Максимальная длина имени файла:"
        "%ld\n", buf.f_namemax);
}
return 0;
}
```

Листинг 4.19. Пример использования функции `statvfs()`.

Если смонтировать файловую систему FAT на каталог `/a` с опцией «только на чтение», то результат выполнения приведенной программы с аргументами `/i` и `/a` может выглядеть так, как показано в листинге 4.20.

statvfs-информация о файловой системе, содержащей
файл /:

Размер блока файловой системы: 4096

Базовый размер блока файловой системы: 4096

Общее число блоков базового размера в файловой
системе: 18955608

Общее число свободных блоков: 7990010

Число свободных блоков, доступных
непривилегированным процессам: 7027115

Общее число описателей файлов: 9633792

Общее число свободных описателей файлов: 8259049

Число описателей файлов, доступных
непривилегированным процессам: 8259049

Идентификатор файловой системы: 0

Битная шкала флагов: 0

Максимальная длина имени файла: 255

statvfs-информация о файловой системе, содержащей
файл /a:

Размер блока файловой системы: 512

Базовый размер блока файловой системы: 512

Общее число блоков базового размера в файловой
системе: 2847

Общее число свободных блоков: 1960

Число свободных блоков, доступных
непривилегированным процессам: 1960

Общее число описателей файлов: 0

Общее число свободных описателей файлов: 0

Число описателей файлов, доступных
непривилегированным процессам: 0

Идентификатор файловой системы: 0

Битная шкала флагов: f

Максимальная длина имени файла: 260

Листинг 4.20. Возможный результат работы программы,
использующей функцию `statvfs()`.

Интерпретация полученных результатов предоставляется читателю.
Служебная программа

```
du [опция ...] [файл ...]
```

выдает информацию о суммарном объеме пространства (измеряемого аналогично `df`), занятого иерархиями файлов с указанными в командной строке корнями, предваряя ее аналогичными сведениями для каждого из подкаталогов, входящих в иерархии. При отсутствии аргументов выводятся сведения о текущем каталоге. Файлы, на которые есть несколько жестких ссылок, учитываются только один раз. Символьные ссылки трактуются по сути так же, как и в служебной программе `ls`, включая смысл опций `-H` и `-L`.

Выделим несколько других опций.

`-a`

В дополнение к подразумеваемому выводу сообщать размеры файлов, входящих в обрабатываемые иерархии и не являющихся каталогами.

`--s`

Вместо подразумеваемого вывода информировать только о суммарном объеме занятого пространства для каждой заданной в командной строке иерархии.

Пример. В ответ на команду

```
du -k /usr/local/man /bin/vi /bin/view
```

может быть выдано следующее (см. листинг 4.21):

```
1428 /usr/local/man/man1
12   /usr/local/man/man5
64   /usr/local/man/man7
36   /usr/local/man/man8
1544 /usr/local/man
384  /bin/vi
0    /bin/view
```

Листинг 4.21. Возможный результат использования утилиты `du`.

Таким образом, иерархия файлов с корнем в `/usr/local/man` занимает около 1.5 Мб (почти все приходится на долю файлов подкаталога `man1`), обычный файл `/bin/vi` – 384 Кб (читателю предлагается сопоставить эту величину и приведенный выше размер данного файла в байтах), символьная ссылка лишнего места не занимает.

Изменение атрибутов файлов и текущей позиции в файловой иерархии

Для смены текущего каталога (т. е. начальной точки маршрутов, начинающихся символом /) служат упоминавшаяся ранее обычная встроенная команда языка shell

```
cd [-L | -P] [целевой_каталог]
cd -
```

и функция `chdir()`:

```
#include <unistd.h>
int chdir (const char *path);
```

Команда `cd` без аргументов осуществляет переход в домашний каталог пользователя, заданный переменной окружения `HOME`. Если аргументом является минус, выполняются действия, показанные в листинге 4.22, осуществляется переход в каталог, ранее бывший текущим, и в случае успеха выводится его абсолютное маршрутное имя.

```
cd "$OLDPWD" && pwd
```

Листинг 4.22. Действия, выполняемые по команде `cd -`.

Алгоритм работы команды `cd`, как ни странно, довольно сложен и содержит целый ряд тонкостей. Сначала, если целевой каталог задан относительным маршрутным именем, выполняется преобразование к абсолютному формату. При этом:

- если имя начинается с точки или точки-точки, перед ним подставляется значение переменной окружения `$PWD` и /;
- в других случаях вместо `$PWD` подставляются элементы списка, являющегося значением переменной окружения `$CDEPATH` и устроенного аналогично `$PATH`; процесс продолжается до тех пор, пока не получится существующий каталог; в случае необходимости в последнюю очередь используется значение `$PWD`.

Затем выполняется раскрытие символьных ссылок и устранение имен «точка» и «точка-точка» («точка» уничтожается вместе со следующим за ней символом /, «точка-точка» — вместе с предыдущим компонентом, отличным от «точки-точки», и символом / между ними). Опции команды `cd` влияют на порядок выполняемых действий. По опции `-P` сначала раскрываются символьные ссылки. Это значит, что «точка-точка»

трактуются как физический надкаталог (каталог, вышележащий по отношению к указуемому файлу). При наличии (подразумеваемой) опции `-L` порядок действий обратный; в результате «точка-точка» обозначает логический надкаталог (каталог, вышележащий по отношению к символьной ссылке, а не к указуемому файлу).

Наконец, выполняется переход по результирующему маршруту.

В случае успешной смены текущего каталога соответственно изменяются значения переменных окружения `$OLDPWD` (текущий каталог непосредственно перед выполнением команды `cd`) и `$PWD` (текущий каталог после выполнения команды `cd`).

Рассмотрим пример выполнения команды `cd` с разными опциями (см. листинг 4.23). Возможный результат показан в листинге 4.24.

```
ls -dl /usr/tmp /var/tmp
cd /usr/tmp
pwd
pwd -P
cd ..
pwd
cd -
cd -P ..
pwd
```

Листинг 4.23. Пример выполнения команды `cd` с разными опциями.

```
lrwxrwxrwx 1 root root 10 Jul 6 14:16 /usr/tmp ->
    ../var/tmp
drwxrwxrwt 5 root root 4096 Sep 4 18:05 /var/tmp
/usr/tmp
/var/tmp
/usr
/usr/tmp
/var
```

Листинг 4.24. Возможный результат выполнения команд `cd` с разными опциями.

Можно видеть, что `/usr/tmp` является символьной ссылкой на каталог `/var/tmp`. При варьировании опций команды `pwd` каталог `/usr/tmp` по-разному отображается в качестве текущего (напомним, опция `-P` команды `pwd` вызывает раскрытие символьных ссылок). По-разному

срабатывает и команда `cd` с целевым каталогом «точка-точка», опцией `-P` и без таковой. В результате видно различие между физическим и логическим надкаталогами символьной ссылки.

Для изменения атрибутов файлов служат утилиты

```
chown [-R] [-H | -L | -P ] владелец[:группа] файл ...  
(смена владельца и, быть может, владеющей группы файла) и  
chmod [-R] изменение_режима файл ...
```

(модификация режима файла), а также аналогичные им по назначению и именам функции (см. листинг 4.25).

```
#include <unistd.h>  
int chown (const char *path, uid_t owner, gid_t group);  
#include <unistd.h>  
int fchown (int fd, uid_t owner, gid_t group);  
#include <sys/stat.h>  
int chmod (const char *path, mode_t mode);  
#include <sys/stat.h>  
int fchmod (int fd, mode_t mode);
```

Листинг 4.25. Описание функций `chown()`, `fchown()`, `chmod()` и `fchmod()`.

При обращении к утилите `chown` владелец и группа задаются именами или числовыми идентификаторами. Реализации должны сначала произвести поиск заданных аргументов как имен в базах данных пользователей и, если нужно, групп и извлечь оттуда соответствующие числовые идентификаторы; если поиск окажется неудачным, аргументы рассматриваются как идентификаторы.

Изменить владельца может только нынешний владелец файла или пользователь, «имеющий соответствующие привилегии» (см. выше раздел «Основные понятия и идеи стандарта POSIX»); некоторые реализации предоставляют подобное право только привилегированным пользователям. Обычно приходится изменять владельца и/или группу после переноса файлов с другого компьютера с иным соответствием числовых идентификаторов и имен.

Опции `-R`, `-H` и `-L` имеют в целом тот же смысл, что и для утилиты `ls` (см. выше): первая предписывает рекурсивный обход встретившихся подкаталогов, две другие — выборочное или полное раскрытие символьных ссылок, указывающих на каталоги. Опция `-P` означает, что изменения относятся к самим символьным ссылкам.

Если при использовании функций `chown()` и `fchown()` меняется лишь владелец, то аргумент `group` задается равным `(gid_t) (-1)`; при смене только группы идентификатор владельца следует задать как `(uid_t) (-1)`.

Задавая аргумент изменение_режима служебной программы `chmod` владельца файла обозначают буквой `u`, владеющую группу — буквой `g`, прочих пользователей — `o`. Добавлению прав соответствует знак `+`, их удаление пометается знаком `-`. Знак `=` обозначает буквальное задание прав (для указанной категории пользователей устанавливается указанный режим доступа). После букв, определяющих категорию пользователей, и знака операции следуют сами добавляемые (удаляемые, устанавливаемые) режимы доступа — обычно `r`, `w` и/или `x`; можно указать несколько подобных связей, разделяя их запятыми (без пробелов).

Пусть, например, все могут читать и изменять файл `myfile`. Чтобы отнять право на запись у владеющей группы и прочих пользователей и одновременно добавить себе право на выполнение, владельцу следует воспользоваться командой, показанной в листинге 4.26.

```
chmod go-w,u+x myfile
```

Листинг 4.26. Пример использования служебной программы `chmod`.

Как указывалось выше, для удобства программирования аналогичных действий на языке C в заголовочном файле `<sys/stat.h>` определены константы, соответствующие битам режима файлов. Например, `S_IRUSR` соответствует праву на чтение для владельца, `S_IRWXG` охватывает все три бита режима доступа для владеющей группы и т.п. С помощью побитных операций с текущим режимом файла и этими константами можно сформировать требуемое значение аргумента `mode` функции `chmod()` (см. листинг 4.27).

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

/* Программа добавляет права на выполнение для */
/* файлов - аргументов командной строки          */
int main (int argc, char *argv[]) {
    struct stat buf;
    int i;
    for (i = 1; i < argc; i++) {
        if (stat (argv [i], &buf)) {
```

```
fprintf (stderr, "\nstat: не удалось получить"
        "информацию о файле %s\n", argv [i]);
return (-1);
}

if (chmod (argv [i], buf.st_mode | S_IXUSR |
        S_IXGRP | S_IXOTH)) {
    fprintf (stderr, "\nchmod: не удалось изменить"
            "режим доступа к файлу %s\n", argv [i]);
    return (-1);
}
}

return 0;
}
```

Листинг 4.27. Пример использования функций `stat()` и `chmod()`.

За счет рационального выбора режимов доступа можно сделать невозможными большинство ошибочных операций с файлами (например, их удаление или изменение посторонними пользователями), не ограничивая свободы осмысленных действий.

В число атрибутов файла входят показатели времени последнего доступа и/или изменения (см. выше описание структуры `stat`). Обычно их модификация осуществляется как побочный эффект операций с файлами, но они устанавливаются и явным образом с помощью служебной программы `touch`:

```
touch [-am] [-r эталонный_файл | -t время] [-c]
      файл ...
```

Опция `-a` предписывает модифицировать время последнего доступа, опция `-m` — время последнего изменения; если не задана ни одна из них, выполняется и то, и другое.

Устанавливаемое время может быть текущим (если не заданы опции `-r` и `-t`), наследоваться у эталонного_файла или задаваться явно как аргумент опции `-t`. В последнем случае оно должно иметь вид

```
{[CC]YY}MMDDhhmm[.SS]
```


Пары цифр указывают, соответственно, на две старшие цифры года, две младшие цифры года, месяц в году, номер дня в месяце, час в дне, минуту в часе и секунду в минуте.

(Любопытны правила формирования старших цифр года, если они опущены. Когда младшие цифры лежат в диапазоне от 69 до 99, подразумевается 19; в противном случае – 20. Несомненно, в будущих версиях стандарта данное соглашение изменится.)

Если файл-аргумент не существует, он создается по умолчанию служебной программой `touch`. Опция `-c` запрещает делать это.

Приведем пример совместного использования служебных программ `touch`, `chown` и `chmod`. В процессе загрузки ОС Linux выполняются действия по инициализации файлов с информацией о пользовательских сеансах, подобные тем, что приведены в листинге 4.28.

```
> /var/run/utmp  
touch /var/log/wtmp  
chown :utmp /var/run/utmp /var/log/wtmp  
chmod ug=rw,o=r /var/run/utmp /var/log/wtmp
```

Листинг 4.28. Пример совместного использования утилит `touch`, `chown` и `chmod`.

Здесь полезны обе возможности `touch` – и модификация атрибутов существующих файлов, и создание новых.

Часто `touch` применяют при работе с файлами-замками (см. листинг 4.29).

```
start () {  
    echo -n "Starting cupsd: "  
    daemon cupsd  
    RETVAL=$?  
    echo  
    [ $RETVAL = 0 ] && touch /var/lock/subsys/cups  
    return $RETVAL  
}
```

Листинг 4.29. Пример использования утилиты `touch` для работы с файлами-замками при загрузке ОС Linux.

Создание, удаление, копирование и перемещение файлов

Стандарт POSIX-2001 не требует наличия утилиты для создания обычных файлов. Они появляются по мере необходимости как побочный продукт многочисленных служебных программ (например, утилиты копирования), поэтому сама постановка задачи — создать файл «просто так» — является отчасти надуманной. В то же время, если новый файл все-таки нужен, полезно иметь в виду возможность перенаправления вывода пустой команды, которая имеется в языке shell.

При программировании на языке C для создания обычных файлов можно воспользоваться функцией `creat()` (см. листинг 4.30).

```
#include <fcntl.h>
int creat (const char *path, mode_t mode);
```

Листинг 4.30. Описание функции `creat()`.

Функция `creat()` имеет два аргумента: маршрутное имя вновь образуемого файла и устанавливаемый режим доступа (идентификаторы владельца и владеющей группы наследуются у текущего пользователя). Результатом служит файловый дескриптор (который, напомним, представляется неотрицательным целым числом), т. е. функция `creat()` не только создает файл, но и открывает его.

Если файл, который пытаются создать при помощи `creat()`, уже существует, он опустошается (размер становится равным 0), а режим доступа и владелец не изменяются.

В случае неудачи результат `creat()` равен -1, а внешней переменной `errno` присваивается код ошибки, позволяющий определить причину ее (ошибки) возникновения. Переменная `errno`, а также мнемоники для кодов ошибок определены в заголовочном файле `<errno.h>`. Для формирования системного сообщения об ошибке можно воспользоваться функцией `perror()` (см. листинг 4.31), которая, опираясь на значение `errno`, помещает в стандартный протокол описание последней ошибки.

```
#include <stdio.h>
void perror (const char *s);
```

Листинг 4.31. Описание функции `perror()`.

Например, при первом выполнении программы, приведенной в листинге 4.32, в стандартный протокол может быть выдан соответствующий результат (см. листинг 4.33).

```
#include <fcntl.h>
#include <errno.h>
#include <stdio.h>
#include <limits.h>
/* Программа пытается создавать в текущем */
/* каталоге файлы с именами g1, g2, ..., */
/* пока эти попытки не закончатся неудачей */
int main (void) {
    int n = 0;
    char name [PATH_MAX];
    do
        sprintf (name, "g%d", ++n);
    while (creat (name, (mode_t) 0) >= 0);
    perror ("CREAT failed");
    fprintf (stderr, "errno = %d\n", errno);
    fprintf (stderr, "Неудача на файле номер %d\n", n);
    return 0;
}
```

Листинг 4.32. Пример программы, использующей функции `creat()` и `perror()`.

```
CREAT failed: Too many open files
errno = 24
Неудача на файле номер 1022
```

Листинг 4.33. Возможный результат первого выполнения программы, использующей функции `creat()` и `perror()`.

Результат второго запуска той же программы показан в листинге 4.34.

```
CREAT failed: Permission denied
errno = 22
Неудача на файле номер 1
```

Листинг 4.34. Результат повторного выполнения программы, использующей функции `creat()` и `perror()`.

При первом запуске причина неудачи — превышение максимально допустимого числа одновременно открытых файлов (с учетом стандартных ввода, вывода и протокола), при повторном — попытка создать существующий файл вопреки отсутствию права на запись в него (файлы создавались с нулевым режимом доступа).

Перечислим несколько других условий, способных привести к неудачному завершению вызова `creat()`:

- компонент маршрутного имени не существует или не является каталогом;
- у компонента маршрута отсутствует право на поиск;
- создание файла требует записи в каталог, права на запись в который нет;
- файл существует и является каталогом.

Рассмотренный в примере стиль уведомления о неудачном завершении общий для большинства функций. Неудача определяется возвращением результата, невозможного в другом случае (почти всегда это `-1` или пустой указатель `NULL`); код ошибки заносится в переменную `errno`. Разумеется, в реальных программах `errno` не выводят, а анализируют. В данном случае уместно было бы сравнивать `errno` с константами `EACCES`, `EINVAL`, `EMFILE` и т.д. (см. `<errno.h>`).

Файл не обязательно создавать в текущем каталоге; в качестве аргумента `creat()` может быть передано составное имя. Пример, когда устанавливаются все биты режима доступа, приведен в листинге 4.35.

```
df = creat ("/tmp/sample", S_IRWXU | S_IRWXG | S_IRWXO);
```

Листинг 4.35. Пример вызова функции `creat()`.

Для создания (пустых) каталогов служит утилита

```
mkdir [-p] [-m режим_доступа] каталог ...
```

и функция `mkdir()`

```
#include <sys/stat.h>
int mkdir (const char *path, mode_t mode);
```

С помощью одной команды `mkdir` можно сформировать несколько каталогов. В каждом из них появятся ссылки `.` и `..`, которые другими способами создать нельзя. По умолчанию устанавливаются все биты режима доступа; режим можно задать и явно с помощью опции `-m`.

Если нужен не один, а целая цепочка каталогов, следует воспользоваться опцией `-p`. Соответствующий пример приведен в листинге 4.36.

```
mkdir -p work/tmp/save
```

Листинг 4.36. Пример использования служебной программы `mkdir` для создания цепочки каталогов.

По этой команде в текущем каталоге появится подкаталог `work`, в нем — каталог `tmp`, а уже в нем — каталог `save`.

Смысл аргументов функции `mkdir()` тот же, что и для `creat()`, в случае неудачи возвращается `-1`, но при нормальном завершении результат равен `0` (`mkdir()` не оставляет новый каталог открытым). Перечисленные выше условия неудачного завершения `creat()` (кроме превышения максимального числа одновременно открытых файлов) применимы и к `mkdir()`.

Интерфейс, сходный с `mkdir`, имеют утилита `mkfifo` для создания каналов:

```
mkfifo [-m режим_доступа] канал ...
```

Заметим, что здесь отсутствует опция `-p` и одноименная функция (см. листинг 4.37).

```
#include <sys/stat.h>
int mkfifo (const char *path, mode_t mode);
```

Листинг 4.37. Описание функции `mkfifo()`.

Создать новые ссылки (жесткие или символичные) на файл позволяют служебная программа `ln`:

```
ln [-fs] исходный_файл целевой_файл
ln [-fs] исходный_файл ... целевой_каталог
```

а также функции `link()` и `symlink()` (см. листинг 4.38).

```
#include <unistd.h>
int link (const char *source_path,
          const char *target_path);
#include <unistd.h>

int symlink (const char *link_contents,
             const char *link_name);
```

Листинг 4.38. Описание функций `link()` и `symlink()`.

При помощи указанных утилиты и функций существующему файлу можно дать любое имя — полезная возможность, когда какая-либо программа обращается к нужному ей файлу по фиксированному имени.

Утилита `ln` без опций создает новые жесткие ссылки на заданные исходные файлы. В первой форме — одну ссылку с именем целевой_файл, во второй — несколько новых ссылок в существующем целевом каталоге, имена которых задаются последними компонентами маршрутных имен исходных файлов. Мобильное приложение должно считать, что исходные и целевой файлы обязаны принадлежать одной файловой системе и создать новую жесткую ссылку на каталог или символьную ссылку нельзя.

Функция `link()` аналогична служебной программе `ln` в первой форме, без опций. Ее нормальным результатом служит 0; в случае ошибки возвращается -1. Важно отметить, что с точки зрения файловой системы образование нового элемента каталога и увеличение счетчика жестких ссылок на исходный файл осуществляется функцией `link()` как неделимое действие.

Утилита `ln` с опцией `-s` и функция `symlink()` создают новые символьные ссылки. Исходные файлы не обязаны существовать; соответствующие аргументы трактуются как цепочки символов и задают содержимое ссылок.

Опция `-f` позволяет замещать существующие элементы каталогов новыми (по умолчанию задание существующего файла в качестве целевого считается ошибкой).

Примером смены содержимого символьной ссылки посредством служебной программы `ln` с опциями `-s` и `-f` может служить фрагмент действий при загрузке ОС Linux, приведенный в листинге 4.39.

```
if [ -n "$mver" ]; then
    ln -sf /lib/modules/$mver /lib/modules/default
fi
```

Листинг 4.39. Использование утилиты `ln` для формирования содержимого символьной ссылки.

Для удаления файлов служат утилиты

```
rm [-fiRr] файл ...
```

и

```
rmdir [-p] файл ...
```

а также функции `unlink()`, `rmdir()` и `remove()` (см. листинг 4.40).

```
#include <unistd.h>
int unlink (const char *path);
#include <unistd.h>
int rmdir (const char *path);
#include <stdio.h>
int remove (const char *path);
```

Листинг 4.40. Описание функций `unlink()`, `rmdir()` и `remove()`.

Строго говоря, перечисленные утилиты и функции удаляют не файлы, а указанные ссылки на них, содержащиеся в каталогах — маршрутных префиксах, и, соответственно, уменьшают на единицу число жестких ссылок на эти файлы. Файл удаляется и занятое им пространство освобождается, только если других ссылок на него не остается. Мы, однако, для краткости будем употреблять не совсем точное словосочетание «удаление файлов».

Опишем опции утилиты `rm`. Опция `-i` означает удаление файлов с предварительным запросом подтверждения.

Опция `-f` предписывает не запрашивать подтверждений и не выдавать диагностических сообщений, если удаляемый файл не существует. Опция полезна, когда нет уверенности в существовании удаляемого файла и его отсутствие не является ошибкой.

Для удаления файла желательно иметь право на запись в него (обязательным является наличие права на запись в каталог, содержащий удаляемую ссылку). Если такого права нет, то утилита `rm` запрашивает подтверждение.

В общем случае подтверждение на удаление запрашивается при выполнении следующего условия:

```
(не задана опция -f) И
(((отсутствует право на запись в файл) И
 (стандартный ввод назначен на терминал)) ИЛИ
(задана опция -i)))
```

Наконец, две эквивалентные «сверхмощные» опции `-R` и `-r` (первая предусмотрена стандартом POSIX, вторая поддерживается по историческим причинам) позволяют удалять каталоги со всем их содержимым (если, конечно, хватает прав). Так, в процессе загрузки некоторых Unix-систем выполняется команда, которая удаляет каталог `/tmp` и все содержащиеся в нем файлы и подкаталоги:

```
rm -rf /tmp
```

Разумеется, пользоваться опцией `-R` следует крайне осторожно, но без нее удалить каталог с помощью утилиты `rm` невозможно (по умолчанию удаляемый файл не может быть каталогом). Отметим также, что `rm` удаляет символичные ссылки, а не указуемые файлы.

Для удаления каталогов (правда, только пустых) рекомендуется использовать служебную программу `rmdir` (или одноименную функцию). Опция `-p` предписывает удалять всю цепочку каталогов, составляющих заданное маршрутное имя. Например, если в текущем каталоге есть подкаталог `a`, содержащий только пустой подкаталог `b`, то команда, показанная в листинге 4.41, удалит и `b`, и `a`.

```
rm -p a/b
```

Листинг 4.41. Пример команды удаления цепочки каталогов.

Функция `unlink()` предназначена для удаления файлов, не являющихся каталогами, а функция `remove()`, имеющая в стандарте C99 [5], объединяет возможности функций `rmdir()` и `unlink()`.

В листинге 4.42 приведен пример программы, которая с помощью функций `link()` и `unlink()` осуществляет ответственную обработку файлов с сохранением копии текущей версии.

```
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <sys/stat.h>
/* Программа выполняет обработку с осторожным */
/* замещением основного файла рабочим          */
#define MAIN_FILE "/home/galat/garb/temp/mfile"
#define OLD_FILE  "/home/galat/garb/temp/ofile"
#define WORK_FILE "/home/galat/garb/temp/wfile"
int main (void) {
    /* Необходимые описания */
    /* . . .                  */
    int work_success = 1;

    /* Выполнение операций над рабочим файлом */
    /* . . .                                     */

    /* В случае неудачи выдадим диагностическое */
    /* сообщение и удалим рабочий файл           */
}
```



```
if (! work_success) {
    fprintf (stderr, "\nНеудачное завершение"
             "операций над рабочим файлом %s\n", WORK_FILE);
    unlink (WORK_FILE);
    return (-1);
}

/* Установим режим доступа к рабочему файлу, */
/* подходящие для последующего использования */
/* Пусть, например, это будет доступ для всех */
/* только на чтение */
chmod (WORK_FILE, S_IRUSR | S_IRGRP | S_IROTH);
/* Удалим ранее сохраненную старую версию */
/* основного файла */
unlink (OLD_FILE);
/* Сохраним текущую версию основного файла */
if (link (MAIN_FILE, OLD_FILE)) {
    perror ("Не удалось сохранить текущую версию"
            "основного файла");
    return (-1);
}
/* Удалим текущую версию основного файла */
unlink (MAIN_FILE);
/* Сделаем рабочий файл основным */
if (link (WORK_FILE, MAIN_FILE)) {
    perror ("Не удалось сделать рабочий файл"
            "основным");
    /* Восстановим основной файл */
    link (OLD_FILE, MAIN_FILE);
    return (-1);
}

/* Удалим рабочий файл */
unlink (WORK_FILE);

return 0;
}
```

Листинг 4.42. Пример программы, использующей функции `link()` и `unlink()`.

Выше мы отмечали, что, как правило, файлы создаются по мере необходимости при выполнении определенных операций. Одной из таких операций является копирование файлов, выполняемое служебной программой `cp`:

```
cp [-fip] исходный_файл целевой_файл
cp [-fip] исходный_файл ... целевой_каталог
cp -R [-H | -L | -P] [-fip] исходный_файл ...
    целевой_каталог
```

В двух первых формах утилита `cp` напоминает `ln`, только наряду с возможным созданием новых ссылок создаются и новые файлы и, в любом случае, копируется содержимое исходных (или указуемых исходными, если последние представляют собой символичные ссылки) файлов.

При наличии опции `-R` выполняется рекурсивное копирование исходных файловых иерархий в целевой каталог со стандартной трактовкой символических ссылок, управляемой опциями `-H`, `-L` и `-P` (см. выше, например, описание утилиты `showln`). Если копирование производится в существующий каталог, имена копий образуются как конкатенация целевого_каталога, символа `/` и маршрута относительно каталога, содержащего исходный_файл. Если в командной строке заданы два файла и целевой не существует, он создается и становится корнем иерархии-копии, то есть исходный_файл копируется непосредственно в него, а маршруты файлов нижележащих уровней считаются относительно исходного_файла.

Если в командной строке задано более двух файлов и целевой не существует или не является каталогом, это считается ошибкой.

Опция `-i` влечет запрос подтверждения перед попыткой копирования в существующий файл.

По опции `-f`, если не удастся открыть целевой файл, делается попытка сначала удалить его, а затем продолжить действия по копированию.

Опция `-p` вызывает копирование не только содержимого, но и атрибутов исходных файлов (режима, идентификаторов владельца и владющей группы, времени последнего доступа и изменения).

Синтаксис, с точностью до опций аналогичный двум первым формам `cp`, имеет вызов служебной программы перемещения файлов `mv`:

```
mv [-fi] исходный_файл целевой_файл
mv [-fi] исходный_файл ... существующий_целевой_каталог
```

Формально можно считать, что она выполняет копирование иерархий с корнями в исходных_файлах (без раскрытия символических ссылок

и с копированием атрибутов файлов) и уничтожает оригиналы. Отметим, что в первой форме `mv` целевой_файл может не существовать, а исходный — являться каталогом; в любом случае на месте целевого каталога допускается использование символьной ссылки на него.

На самом деле утилита `mv`, как правило, применяется для переименования файлов, и тогда ее работа сводится к созданию новых элементов каталогов и уничтожению старых, а реальное копирование может потребоваться только при перемещении файлов между файловыми системами (например, со съемного носителя на постоянный).

По опции `-i` запрашивается подтверждение перед замещением существующего файла, опция `-f` влечет отсутствие подобных запросов (по поводу логики запроса подтверждений см. выше описание команды `rm`).

Приведем пример употребления утилиты `mv`. Пусть текущий каталог содержит только файлы `x` и `y` и пустой каталог `d1` (см. листинг 4.43). Тогда после выполнения команд, показанных в листинге 4.44, будет создан каталог `d2`, где окажутся файлы `x` и `y`, а каталог `d1` исчезнет (см. листинг 4.45).

```
.: d1/ x y
./d1:
```

Листинг 4.43. Состояние текущего каталога перед перемещением файлов.

```
ls -RF
mv x y d1
mv d1 d2
ls -RF
```

Листинг 4.44. Использование утилиты `mv` для перемещения файлов и файловых иерархий.

```
.:
d2/
./d2:
x y
```

Листинг 4.45. Состояние текущего каталога после перемещения файлов.

Продолжим этот пример двумя одинаковыми командами копирования (см. листинг 4.46).

```
cp -R d2 d1
cp -R d2 d1
ls -RF
```

Листинг 4.46. Применение утилиты `cp` для копирования файловых иерархий.

Первая команда скопирует иерархию с корнем `d2` во вновь созданный каталог `d1`, вторая — под `d1` (с сохранением имени `d2` для корня копии). Результат показан в листинге 4.47.

```
.:
d1/ d2/
./d1:
d2/ x y
./d1/d2:
x y
./d2:
x y
```

Листинг 4.47. Результат использования утилиты `cp` для копирования файловых иерархий.

Для перемещения (переименования) одного файла служит функция `rename()` из репертуара C99 (см. листинг 4.48).

```
#include <stdio.h>
int rename (const char *old_path, const char *new_path);
```

Листинг 4.48. Описание функции `rename()`.

Функция `rename()` обладает частью функциональности служебной программы `mv`. Она не перемещает файловых иерархий и не раскрывает символьных ссылок, если их имена заданы в качестве аргументов. С другой стороны, `rename()` поддерживает переименование каталогов. В этом случае аргумент `new_path` не должен задавать файл другого типа, а если он задает существующий каталог, тот, по понятным причинам, должен быть пустым.

В продолжение двух предыдущих примеров продемонстрируем переименование каталогов (см. листинг 4.49). Итоговое состояние текущего каталога показано в листинге 4.50.

```
#include <stdlib.h>
#include <stdio.h>
#define OLD_DIR "d1/d2"
#define NEW_DIR "d2"

int main (void) {
    system ("rm -f " NEW_DIR "/*");
    if (rename (OLD_DIR, NEW_DIR)) {
        perror ("RENAME failed");
        return (-1);
    }
    system ("ls -RF");
    return 0;
}
```

Листинг 4.49. Пример использования функции `rename()` для переименования каталогов.

```
.:
d1/ d2/
./d1:
x y
./d2:
x y
```

Листинг 4.50. Состояние текущего каталога после переименования каталогов с помощью функции `rename()`.

Читателю предлагается оценить последствия повторного запуска приведенной программы.

Обход и обработка файловых иерархий

Для обхода файловой иерархии и систематической обработки ее элементов служит утилита `find`:

```
find [-H | -L] файл ... [выражение]
```

Она рекурсивно обходит каждую из иерархий с корнями в заданных файлах (разумеется, обычно в этом качестве указываются каталоги), отыскивая файлы, которые удовлетворяют логическому выражению, построенному с помощью описанных ниже средств. Опции `-H` и `-L` стандартным образом влияют на трактовку символьных ссылок. Если указуемый файл не существует, `find` оперирует с самой ссылкой.

Перечислим элементарные логические выражения и правила их вычисления. (Далее n обозначает целое десятичное число; вместо него могут указываться также комбинации $+n$ и $-n$, что означает, соответственно, «больше, чем n » и «меньше, чем n ».)

`-name шаблон_файлов`

Истина, если имя текущего файла удовлетворяет шаблону_файлов. Символы шаблона, имеющие для shell специальный смысл, должны быть экранированы.

`-type тип_файла`

Истина, если файл имеет заданный тип: `b`, `c`, `d`, `f`, `p` или `s` — является, соответственно, блочным или символьным специальным файлом, каталогом, обычным файлом, каналом или сокетом.

`-size n[c]`

Истина, если файл занимает n блоков по 512 байт. Когда указана буква `c`, размер файла задается в символах. Напомним, что с помощью комбинаций $+n$ и $-n$ можно проверять размер (и три указанные ниже характеристики) не только на равенство, но и на неравенство.

`-atime n`

Истина, если последний доступ к файлу производился n дней назад (в данном контексте день — это промежуток времени в 86400 секунд).

`-mtime n`

Истина, если файл последний раз модифицировался n дней назад.

`-ctime n`

Истина, если атрибуты файла последний раз изменялись n дней назад.

`-perm [-]режим`

Истина, если режим файла соответствует заданному. При наличии знака минус соответствие понимается как включение (все заданные биты должны присутствовать в режиме доступа к файлу); если минус отсутствует,

требуется точное совпадение. Режим задается аналогично утилите `chmod`.

`-links n`

Истина, если на файл имеется `n` жестких ссылок.

`-user имя_пользователя`

Истина, если владельцем файла является заданный пользователь. Когда в качестве имени задано десятичное число и пользователя с таким именем нет, число трактуется как идентификатор пользователя.

`nouser`

Истина, если идентификатор владельца файла отсутствует в базе данных пользователей.

`--group владеющая_группа`

Истина, если файлом владеет заданная группа.

`-nogroup`

Истина, если идентификатор владеющей группы файла отсутствует в базе данных групп.

`-depth`

Всегда истина; задает дисциплину обхода иерархии вглубь: сначала обрабатываются все элементы каталога, потом — сам каталог (по умолчанию `find` в первую очередь обрабатывает каталог и лишь после этого — его элементы).

`--xdev`

Всегда истина; предписывает не спускаться в каталоги, имеющие другой идентификатор устройства (`st_dev`, см. выше описание структуры `stat`).

`-prune`

Всегда истина; предписывает не обрабатывать текущий файл, если он является каталогом.

-exes команда

Истина, если после выполнения команды возвращается нулевой код завершения. Запись команды должна заканчиваться экранированной точкой с запятой. Аргумент команды, заданный в виде пары фигурных скобок {}, заменяется маршрутным именем обрабатываемого файла.

-ok команда

Эквивалентно -exes за исключением того, что перед выполнением команды запрашивается подтверждение (в виде сгенерированной командной строки со знаком вопроса в конце), и она выполняется только при ответе у.

-print

Всегда истина; вызывает выдачу маршрутного имени обрабатываемого файла на стандартный вывод. Если в командной строке find не задано выражение, то подразумевается -print. Если выражение не содержит ни -exes, ни -ok, ни -print, вместо него используется конструкция

```
{ выражение } -print  
-newer файл
```

Истина, если текущий файл был модифицирован позднее указанного файла.

(выражение)

Истина, если истинно заключенное в скобки выражение (скобки должны быть экранированы от интерпретации языком shell).

Элементарные логические выражения могут комбинироваться с помощью следующих операций (в порядке уменьшения приоритета):

- унарная операция отрицания, обозначается !.
- логическое И, обозначается пробелом или -a. Если значением первого операнда оказалась ложь, второй не вычисляется. Таким образом, последовательность разделенных пробелами выражений-операндов можно рассматривать как составной фильтр, через который пропускается текущий файл: если значением очередного операнда оказалась ложь, обработка прекращается; в противном случае файл передается следующему компоненту фильтра.

- логическое ИЛИ, обозначается -o. Если значением первого операнда оказалась истина, второй не вычисляется.

Приведем несколько примеров. Пусть нужно подсчитать число C-файлов в текущем каталоге и его подкаталогах. Воспользуемся следующим конвейером:

```
find . -name \*.c -print | wc -l
```

Обратим внимание на то, что команда `find` — одна из немногих, интерпретирующая шаблоны имен файлов самостоятельно. Собственно, у нее нет другого выхода, поскольку она должна рекурсивно обойти указанные иерархии и в каждой из них искать файлы, имена которых удовлетворяют заданному шаблону. Нетрудно заметить, что стандартный shell-механизм генерации имен файлов в данном случае не годится.

Рассмотрим более сложный пример. Нужно подсчитать суммарное число строк во всех C-файлах текущего каталога и его подкаталогов. Решений может быть несколько, рассмотрим три из них. Во-первых, можно организовать конвейер:

```
cat `find . -name \*.c -print` | wc -l
```

Утилита `find` отберет имена нужных файлов, `cat` выдаст их совокупное содержимое на стандартный вывод, а команда `wc -l` подсчитает общее число строк. По существу ту же идею можно выразить в другой форме:

```
find . -name \*.c -exec cat {} \; | wc -l
```

Здесь содержимое нужных файлов будет выдаваться на стандартный вывод по мере их обнаружения.

Наконец, можно поступить совсем просто, сделав нужные файлы аргументами команды `wc`. Правда, при этом будет выдаваться еще и число строк в каждом из C-файлов:

```
wc -l `find . -name \*.c -print`
```

Напишем теперь команду, позволяющую удалить из текущего каталога и его подкаталогов все файлы нулевого размера, а также объектные файлы, к которым не было доступа более месяца, запрашивая подтверждение (см. листинг 4.51). Обратим внимание на пробелы, окружающие экранированные скобки и экранированную точку с запятой.

```
find . \( -size 0c -o -name \*.o -a -atime +30 \)
      -ok rm {} \;
```

Листинг 4.51. Еще один пример использования утилиты `find`.

Пусть на каталог `/mnt` смонтирован съемный носитель. Требуется выявить на нем файлы, владелец или группа которых на данной системе неизвестны, и изменить их на `nobody`. К цели ведет команда, показанная в листинге 4.52 (предполагается, что тот, кто ее выполняет, обладает соответствующими привилегиями).

```
find /mnt \( -nouser -o -nogroup \) -exec chown
      nobody:nobody {} \;
```

Листинг 4.52. Пример выявления и обработки файлов с неизвестными владельцем или владеющей группой.

Если нужно обработать корневую файловую систему без учета смонтированных файловых систем, располагающихся на других устройствах, целесообразно воспользоваться командой вида `find / -xdev ...`.

Две команды, показанные в листинге 4.53, почти эквивалентны: обе не заходят в подкаталоги с именем `skip` текущего каталога, но только вторая выводит их имена.

```
find . -name skip -prune -o -print find . -print
      -name skip -prune
```

Листинг 4.53. Пример использования элементарного выражения `-prune`.

Одной из форм обхода и обработка файловой иерархии можно считать архивирование. Стандарт POSIX предусматривает для этого служебную программу `rax`:

```
rax [опция ...] [шаблон ...]
rax -r [опция ...] [шаблон ...]
rax -w [опция ...] [файл ...]
rax -r -w [опция ...] [файл ...] каталог
```

Утилита `rax` поддерживает несколько архивных форматов, в число которых входят «родной» `rax` и унаследованные `cpio` и `tar`. В зависимости

от заданной комбинации опций `-r` и `-w` `рх` осуществляет одну из четырех операций: выводит оглавление архива (первая из приведенных выше форм команды `рх`), читает архив и извлекает из него файлы по заданным шаблонам имен (вторая форма), записывает заданные файлы в архив (третья форма) или копирует иерархии файлов (четвертая форма).

Как правило, архив читается со стандартного ввода, а записывается на стандартный вывод; если встречается файл типа «каталог», обработке подвергается вся иерархия файлов с корнем в этом каталоге.

При извлечении файлов из архива разрешение маршрутных имен выполняется относительно того каталога, из которого утилита `рх` была вызвана.

Режим копирования можно представлять себе как комбинацию архивирования с последующим извлечением файлов в заданный каталог.

В листинге 4.54 представлены примеры использования команды `рх`. Первая архивирует содержимое текущего каталога, последняя копирует каталог `old_dir` в `new_dir`.

```
рх -w .  
mkdir new_dir  
рх -rw old_dir new_dir
```

Листинг 4.54. Пример использования служебной программы `рх`.

Вариант 1

1. Согласно стандарту POSIX, в файловой системе

- ☐ определен единый для всех процессов корень
- ☐ для каждого процесса определен свой корень
- ☐ для каждого пользователя определен свой корень

2. Согласно стандарту POSIX, в каждом каталоге должны присутствовать следующие имена

- ☐ .
- ☐ ..
- ☐ /

3. Согласно стандарту POSIX, для получения информации о файловых системах предназначены следующие функции

- ☐ stat()
- ☐ statfs()
- ☐ statvfs()

Вариант 2

1. **Согласно стандарту POSIX, в файловую систему объединяются**
 - ☐ только файлы
 - ☐ только служебная информация о файлах
 - ☐ файлы вместе со служебной информацией

2. **Согласно стандарту POSIX, для повышения мобильности в имена файлов, помимо латинских букв и цифр, рекомендуется включать только следующие символы**
 - ☐ .
 - ☐ +
 - ☐ -

3. **Согласно стандарту POSIX, для получения информации о файлах предназначены следующие функции**
 - ☐ stat()
 - ☐ fstat()
 - ☐ statfs()

Вариант 3

1. Согласно стандарту POSIX, файловая система является

- ☐ деревом
- ☐ лесом деревьев
- ☐ направленным графом

2. Согласно стандарту POSIX, минус не рекомендуется делать первым символом в имени файла, поскольку

- ☐ такое имя может быть неотличимо от опции в командной строке
- ☐ такое имя текстуально не является мобильным
- ☐ такое имя может быть воспринято как указание на удаление файла

3. Согласно стандарту POSIX, для получения информации о текущем каталоге предназначены следующие функции

- ☐ `pwd();`
- ☐ `getcwd();`
- ☐ `getpwid();`

Лекция 5. Файловый ввод/вывод

Рассматриваются основные понятия, ассоциированные с вводом/выводом, описываются функции, предназначенные для открытия и закрытия файлов, выполнения операций ввода/вывода и сопутствующих действий.

Ключевые слова: ввод, вывод, ввод/вывод нижнего уровня, буферизованный ввод/вывод, управляемая буферизация, форматный ввод/вывод, файловый дескриптор, поток, описание открытого файла, открытие файла, операция ввода/вывода, индикатор текущей позиции, предопределенный поток, стандартный ввод, стандартный вывод, стандартный протокол, чтение, запись, модификация, добавление, позиционирование, полностью буферизованный поток, построчно буферизованный поток, небуферизованный поток, блок, байтный символ, широкий символ, ориентация потока, широкая ориентация, байтная ориентация, закрытие файла, синхронный ввод/вывод, асинхронный ввод/вывод, синхронизированный ввод/вывод, операция синхронизированного ввода/вывода, завершение операции синхронизированного ввода/вывода, целостность данных, целостность файлов, флаг статуса файла, допустимые виды операций ввода/вывода, опустошение файла, канал, временный файл, временное имя, индикатор ошибки, индикатор конца файла, построчный ввод, построчный вывод, порядок байт в слове, обработка ошибочных ситуаций, точка отсчета, управляющая команда, дублирование дескриптора открытого файла, флаг, ассоциированный с файловым дескриптором, блокировка, синхронизация доступа к данным, рекомендательная блокировка, постановка блокировки, снятие блокировки, сегмент файла, тип блокировки, блокировка на чтение, блокировка на запись, характеристики блокировки, управление буферизацией, выталкивание буферов.

Основные понятия

В стандарте POSIX-2001 предусмотрены две основные группы функций, обслуживающие операции ввода/вывода:

- функции нижнего уровня, использующие упоминавшиеся ранее целочисленные файловые дескрипторы (эти функции и ассоциированные объекты определены в заголовочном файле `<fcntl.h>`);
- функции более высокого уровня, осуществляющие буферизованный ввод/вывод с применением потоков (см. `<stdio.h>`).

Применение потоков не расширяет базовых возможностей ввода/вывода, поскольку в конечном счете используются функции нижнего уровня, но в отличие от них позволяет приложениям получить дополнительный сервис в виде управляемой буферизации и форматного ввода/вывода.

Согласно стандарту POSIX-2001, поток — это объект, служащий для доступа к файлам как к упорядоченной последовательности символов.

Поток представляется структурой типа `FILE`, с которой ассоциирован соответствующий дескриптор открытого файла. Несколько дескрипторов и/или потоков могут ссылаться на одно описание открытого файла.

Существенны не только содержимое, но и адрес объекта типа `FILE`; копия подобного объекта не обязательно является корректным представлением потока.

И файловые дескрипторы, и потоки формируются в результате выполнения функций открытия файлов, которые должны предшествовать операциям ввода/вывода. Имеется, однако, три предопределенных потока: стандартный ввод, стандартный вывод и стандартный протокол, открываемые окружением времени выполнения еще перед началом работы C-программ. Для обращения к ним служат указатели на объекты типа `FILE` с именами, соответственно, `stdin`, `stdout` и `stderr`.

При открытии файлов указывается вид последующих операций ввода/вывода: чтение, запись, модификация (чтение и запись), добавление (запись в конец). Вид операций должен быть согласован с правами доступа к файлу; в противном случае открытие закончится неудачей.

Если файл поддерживает запросы на позиционирование (таковы обычные файлы в противоположность, например, символьным специальным, соответствующим терминалам), то после открытия индикатор текущей позиции устанавливается в начало (на нулевой байт) при условии, что файл не открывали на добавление; в этом случае от реализации зависит, будет ли индикатор указывать на начало или конец файла.

В дальнейшем индикатор текущей позиции смещается под воздействием операций чтения, записи и позиционирования, чтобы упростить последовательное продвижение по файлу.

Потоки могут быть полностью буферизованными, буферизованными построчно и небуферизованными. В первом случае передача байт из файла/в файл осуществляется преимущественно блоками, когда буфер оказывается заполненным. При построчной буферизации передача данных также осуществляется блоками, по достижении символа перевода строки или заполнении буфера. При отсутствии буферизации байты передаются по возможности без задержек.

Стандартом C99 [5] предусмотрены байтные и широкие символы. Соответственно, в стандарте POSIX-2001 введено понятие ориентации потока, которая может быть широкой или байтной. Задаёт ориентацию

первая после открытия файла операция ввода/вывода. Если вначале применяется функция ввода/вывода широких символов, поток получает широкую ориентацию; в противном случае – байтную. Сменить ориентацию можно только повторным открытием файла; применяемые к потоку функции ввода/вывода должны соответствовать его ориентации.

После завершения работы с файлом его необходимо закрыть. При этом не только разрывается связь между файлами с одной стороны и дескрипторами и потоками с другой, но и обеспечивается передача данных, оставшихся буферизованными.

Стандарт POSIX-2001 предусматривает как синхронный, так и асинхронный ввод/вывод.

Операция асинхронного ввода/вывода сама по себе не способна привести к приостановке процесса (потока управления) и лишить его возможности использовать процессор. Это означает, что процесс и асинхронная операция ввода/вывода могут выполняться параллельно.

При синхронном вводе/выводе процесс (поток управления) приостанавливается до завершения запрошенной операции обмена данными.

С одним файлом могут параллельно работать несколько процессов, выполняющихся на разных хостах (узлах сети) и использующих разные буфера. Для корректного обслуживания подобной ситуации в стандарте POSIX-2001 определено (в качестве необязательной возможности) понятие синхронизированного ввода/вывода – так называют механизм, повышающий детерминированность и устойчивость средств обмена данными, поэтому приложение может быть уверено, что данные, которыми оно манипулирует, физически присутствуют на устройствах вторичной (массовой, стабильной) памяти и наряду с файлами находятся в целостном состоянии.

Операция синхронизированного ввода/вывода считается завершенной, когда данные успешно переданы или она диагностирована как неудачная. Различают завершение с целостностью данных и с целостностью файлов.

Синхронизированное чтение с целостностью данных считается завершенным, когда образ данных успешно передан запрашивающему процессу. Если в момент запроса операции чтения оставались незавершенные операции записи, пересекающиеся с чтением по данным, сначала должны быть обслужены запросы на запись.

Синхронизированная запись с целостностью данных завершена, когда данные успешно переданы и, кроме того, модифицирована и передана в стабильную память информация о файловой системе, необходимая для последующей выборки данных (такая, например, как размер файла).

Синхронизированная операция с целостностью файлов отличается только тем, что к моменту ее завершения дополнительно должны быть модифицированы и переданы в стабильную память все атрибуты файлов,

относящиеся к вводу/выводу (включая время последнего доступа, изменения файлов и их статуса, не влияющие на выборку данных).

Открытие и закрытие файлов

Как уже указывалось, открытие файла должно предшествовать операциям ввода/вывода, поскольку оно возвращает дескриптор файла или поток, которые используют подобные операции. Для открытия файлов и формирования новых описаний открытых файлов, файловых дескрипторов и потоков служат функции нижнего уровня `open()` и `pipe()` (см. листинг 5.1), а также функции буферизованного ввода/вывода, показанные в листинге 5.2.

```
#include <fcntl.h>
int open (const char *path, int oflag, ...);
#include <unistd.h>
int pipe (int fildes [2]);
```

Листинг 5.1. Описание функций `open()` и `pipe()`.

```
#include <stdio.h>
FILE *fopen (const char *restrict path,
             const char *restrict mode);
#include <stdio.h>
FILE *fdopen (int fildes, const char *mode);
#include <stdio.h>
FILE *freopen (const char *restrict path,
               const char *restrict mode, FILE *restrict stream);
```

Листинг 5.2. Описание функций `fopen()`, `fdopen()`, `freopen()`.

Функция `open()` открывает файл с заданным маршрутным именем (первый аргумент, `path`), создавая для него описание — новое и, следовательно, не разделяемое с другими процессами. Возвращаемый в качестве результата файловый дескриптор является минимальным из числа не используемых в данный момент текущим процессом (при неудаче возвращается `-1`).

Второй аргумент, `oflag`, устанавливает флаги статуса файла и определяет допустимые виды операций ввода/вывода. Его значение формируется как побитное ИЛИ перечисленных ниже флагов. Из первых трех флагов должен быть задан ровно один.

`O_RDONLY`

Открыть файл только на чтение.

`O_WRONLY`

Открыть файл только на запись.

`O_RDWR`

Открыть файл на чтение и запись.

Следующие флаги могут комбинироваться произвольным образом.

`O_APPEND`

Перед каждой записью устанавливать индикатор текущей позиции на конец файла.

`O_CREAT`

Если файл существует, данный флаг принимается во внимание только при наличии описываемого далее флага `O_EXCL`. Если файла нет, он создается от имени текущего пользователя. Обратим внимание, что функция `open()` имеет переменное число аргументов. При создании файла предполагается, что в качестве третьего аргумента типа `mode_t` задается режим доступа.

`O_EXCL`

Если установлены флаги `O_CREAT` и `O_EXCL`, а файл существует (хотя бы и в виде символической ссылки на несуществующий файл), вызов `open()` завершится неудачей. Проверка существования файла и его создание представляют собой атомарное действие по отношению к попыткам других процессов (потоков управления) выполнить аналогичный запрос.

`O_TRUNC`

Если файл существует, является обычным и успешно открывается с флагами `O_RDWR` или `O_WRONLY`, он опустошается (размер устанавливается равным нулю).

Отметим, что рассмотренная ранее функция `creat (path, mode)` по определению эквивалентна вызову `open (path, O_WRONLY | O_CREAT | O_TRUNC, mode)`.

Следующий флаг относится к асинхронному вводу/выводу.

`O_NONBLOCK`

Обычно, если канал открывается только на чтение (или только на запись), процесс (поток управления) блокируется, пока этот же канал не откроют на запись (чтение). При установленном флаге `O_NONBLOCK` открытие на чтение завершается без задержки, а открытие на запись заканчивается неудачей, если канал еще не открыт кем-либо на чтение.

При открытии специального файла вызов `open()` завершается только после того, как устройство оказывается в состоянии готовности. Флаг `O_NONBLOCK` отменяет эту задержку, однако последующее поведение устройства зависит от реализации.

Следующая группа флагов обслуживает синхронизированный ввод/вывод.

`O_DSYNC`

Операции записи с возвращаемым файловым дескриптором должны завершаться с обеспечением целостности данных.

`O_SYNC`

Операции записи должны завершаться с обеспечением целостности файла.

`O_RSYNC`

Операции чтения должны завершаться на уровне целостности, заданном флагами `O_DSYNC` или `O_SYNC`.

Смысл последнего из стандартизованных флагов будет пояснен при рассмотрении процессов.

`O_NOCTTY`

Если открывается специальный файл, соответствующий терминалу, последний не должен становиться управляющим терминалом процесса.

Для создания и открытия канала предназначена функция `pipe()`. В массиве `fildev` она возвращает сразу два дескриптора: `fildev [0]`

служит для чтения, `fildes [1]` — для записи. Данные читаются в том же порядке, в каком были записаны.

При успешном завершении `pipe()` возвращает 0, при неудаче — -1.

Функция `fopen()` из группы буферизованного ввода/вывода по сути аналогична `open()`, только вместо файлового дескриптора в качестве результата возвращается указатель на объект, служащий для управления сформированным потоком (в случае неудачи результат равен пустому указателю `NULL`).

Второй аргумент, `mode`, определяющий допустимые виды операций ввода/вывода, задается как цепочка символов. Он может принимать следующие значения.

"r"

Открыть файл на чтение.

"w"

Опустошить или создать файл, открыв его на запись.

"a"

Открыть или создать файл на запись в конец.

"r+"

Открыть файл на изменение (чтение и запись).

"w+"

Опустошить или создать файл, открыв его на изменение.

"a+"

Открыть или создать файл на изменение с записью в конец.

(Стандарт языка C позволяет приписывать к перечисленным цепочкам символ 'b', который, впрочем, ни на что не влияет.)

Если открытый файл не соответствует интерактивному устройству, ассоциированный с ним поток полностью буферизуется.

Функция `fdopen()` формирует поток, ассоциированный с дескриптором ранее открытого файла.

Второй аргумент, `mode`, может принимать те же значения, что и для `fopen()`, но их трактовка по понятным причинам отличается: существующие файлы не опустошаются, а новые не создаются.

Функция `freopen()` предназначена для ассоциирования существующего потока (третий аргумент, `stream`) с заданным файлом (первый аргумент, `path`) и разрешенными видами операций ввода/вывода (второй аргумент, `mode`).

В первую очередь `freopen()` пытается вытолкнуть буфера потока `stream` и закрыть ассоциированный с ним файл. Неудача данного действия ни на что не влияет.

Затем, аналогично `fopen()`, открывается заданный файл, только без формирования нового потока; результатом служит `stream` (лишенный, правда, ориентации) или `NULL` (в случае неудачи).

Если в качестве первого аргумента функции `freopen()` задан пустой указатель, делается попытка изменить виды операций, разрешенные для потока `stream`. (Формально можно считать, что первым аргументом `freopen()` служит имя файла, ассоциированного с этим потоком.) От реализации зависит, разрешены ли подобные действия вообще и при каких условиях они завершаются успешно.

Для закрытия файлов (точнее, файловых дескрипторов или потоков) предназначены функции `close()` и `fclose()` (см. листинг 5.3).

```
#include <unistd.h>
int close (int fildes);
#include <stdio.h>
int fclose (FILE *stream);
```

Листинг 5.3. Описание функций `close()` и `fclose()`.

Функция `close()` освобождает файловый дескриптор `fildes`, который становится доступным для последующего использования при открытии файлов.

Когда закрывается последний дескриптор, ссылающийся на описание открытого файла, оно освобождается.

Если число жестких ссылок на файл равно нулю и закрывается последний ассоциированный с ним дескриптор, файл перестает быть доступным, а занимавшееся им пространство освобождается.

Когда закрывается последний дескриптор, ассоциированный с каналом, все оставшиеся непрочитанными данные теряются.

Функция `close()` возвращает 0 в случае успешного завершения и -1 при неудаче.

Функция `fclose()` по сути аналогична, только она освобождает поток, выталкивая при этом буфера. Признаком успешного завершения также служит 0, признаком неудачи — константа `EOF`.

Приведем примеры использования описанных функций. Сочетание флагов `O_CREAT` и `O_EXCL` функции `open()` позволяет организовать проверку и создание файлов-замков, для которых важен факт существования в одном экземпляре, а не содержимое (см. листинг 5.4).

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

#define LOCK_FILE "my_lock"

/* Функция пытается создать файл-замок          */
/* Результат равен 0 в случае успеха,            */
/* 1, если файл уже существует,                  */
/* -1 в случае прочих ошибок                     */
static int gate (const char *lock_name) {
    int fd;

    if ((fd = open (lock_name, O_WRONLY | O_CREAT |
                     O_EXCL, (mode_t) 0)) < 0) {
        if (errno == EEXIST) {
            return (1);
        }
        return (-1);
    }

    return (close (fd));
}

int main (void) {
    int res;

    if ((res = gate (LOCK_FILE)) < 0) {
        perror ("Ошибка при создании файла-замка "
                LOCK_FILE);
    } else if (res == 1) {
        fprintf (stderr, "Файл-замок " LOCK_FILE " уже"
                "существует\n");
    }
}
```

```
    }  
  
    return (res);  
}
```

Листинг 5.4. Пример программы, использующей функции `open()` и `close()`.

Читателю предлагается выполнить приведенную программу дважды.

Следующая программа иллюстрирует перенаправление стандартного вывода в файл (см. листинг 5.5). Ее тоже полезно выполнить дважды и затем самостоятельно осмыслить результаты.

```
#include <stdio.h>  
#define LOGFILE "my_logfile"  
int main (void) {  
    FILE *fp;  
    printf ("До перенаправления стандартного вывода"  
           " в файл " LOGFILE "\n");  
    if ((fp = freopen (LOGFILE, "a", stdout)) ==  
        NULL) {  
        perror ("Не удалось перенаправить стандартный"  
               " вывод в файл " LOGFILE);  
        return (-1);  
    }  
    printf ("После перенаправления стандартного"  
           " вывода в файл " LOGFILE "\n");  
    if (fclose (fp) == EOF) {  
        perror ("Не удалось закрыть файл " LOGFILE);  
        return (-1);  
    }  
    printf ("После закрытия файла " LOGFILE "\n");  
    return (0);  
}
```

Листинг 5.5. Перенаправление стандартного вывода с помощью функции `freopen()`.

Весьма полезной с практической точки зрения является функция создания и открытия временных файлов `tmpfile()` (см. листинг 5.6).


```
#include <stdio.h>
FILE *tmpfile (void);
```

Листинг 5.6. Описание функции tmpfile().

Временный файл открывается на изменение (w+) и автоматически удаляется после закрытия всех ссылок на него.

Использование функции tmpfile() предпочтительнее генерации «временного» имени с помощью функции tmpnam() и последующего создания файла с этим именем, поскольку в промежутке какой-либо другой процесс может создать одноименный файл.

Чтение и запись данных

Чтение данных из файла выполняют функции read() и fread() (см. листинг 5.7).

```
#include <unistd.h>
ssize_t read (int fd, void *buf, size_t nbyte);
#include <stdio.h>
size_t fread (void *restrict buf, size_t size,
              size_t nitems, FILE *restrict stream);
```

Листинг 5.7. Описание функций read() и fread().

Функция read() пытается прочитать nbyte байт из файла, ассоциированного с дескриптором fd, и поместить их в буфер buf.

Для файлов, допускающих позиционирование, read() выполняет чтение, начиная со значения индикатора текущей позиции, ассоциированного с дескриптором fd. После завершения операции этот индикатор увеличивается на количество прочитанных байт. Для устройств, не поддерживающих позиционирования (таких, например, как терминал), значение упомянутого индикатора не определено, а чтение выполняется с текущей позиции устройства.

При успешном завершении read() возвращает количество байт, реально прочитанных и помещенных в буфер; это значение может оказаться меньше значения аргумента nbyte, если до конца файла оставалось меньше, чем nbyte байт. Например, если текущая позиция совпадала с концом файла, результат будет равен 0. В случае ошибки возвращается -1.

Функция буферизованного ввода/вывода fread() во многом аналогична read(), но число читаемых байт задается как произведение размера одного элемента (аргумент size) на число элементов (аргумент

nitems), а результатом служит количество успешно прочитанных элементов. В стандарте оговаривается, что элементы читаются побайтно.

Число элементов, успешно прочитанных функцией `fread()`, может быть меньше затребованного, только если достигнут конец файла или произошла ошибка чтения. В таком случае `fread()` устанавливает для потока индикатор ошибки или конца файла, проверить которые позволяют функции `feof()` и `ferror()`, соответственно (см. листинг 5.8), возвращая при установленном индикаторе ненулевой результат.

```
#include <stdio.h>
int feof (FILE *stream);
#include <stdio.h>
int ferror (FILE *stream);
```

Листинг 5.8. Описание функций `feof()` и `ferror()`.

Отметим, что использование функции бинарного ввода `fread()` ограничивает мобильность приложений, так как результат зависит от размера элементов и порядка байт, поддерживаемого процессором.

Обратим также внимание на некоторые нюансы синхронного и асинхронного ввода с помощью функции `read()`. При попытке чтения из пустого канала, не открытого кем-либо на запись, результат равен 0 (как признак конца файла). Если пустой канал открыт кем-либо на запись, при установленном флаге `O_NONBLOCK` возвращается -1 (как признак ошибки `EAGAIN`); при отсутствии флага `O_NONBLOCK` процесс (поток управления) блокируется до появления данных в канале. Аналогичным образом устроен ввод из файлов других типов, поддерживающих чтение в асинхронном режиме.

Содержимое символьных ссылок приходится читать особым образом (хотя бы потому, что обычно функция `open()` раскрывает их, т. е. открывает указуемый файл). Для этого служит функция `readlink()` (см. листинг 5.9). Она помещает содержимое ссылки с именем `link_name` в буфер `buf` длины `buf_size` (если буфер мал, остаток содержимого отбрасывается). Результат равен числу помещенных в буфер байт или -1 в случае неудачи.

```
#include <unistd.h>
ssize_t readlink (const char *restrict link_name,
                  char *restrict buf, size_t buf_size);
```

Листинг 5.9. Описание функции `readlink()`.

Следующая программа (см. листинг 5.10) переправляет недлинные сообщения с управляющего терминала процесса (ему соответствует специальный файл `/dev/tty`) на стандартный вывод до тех пор, пока не будет введен символ конца файла.

```
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#define C_TERM "/dev/tty"

int main (void) {
    char buf [BUFSIZ];
    int fd;
    ssize_t line_len;
    /* Открытие на чтение специального файла,      */
    /* ассоциированного с управляющим терминалом */
    if ((fd = open (C_TERM, O_RDONLY)) < 0) {
        perror ("OPEN");
        return (-1);
    }
    /* Ввод с терминала */
    printf ("Вводите строки\n");
    while ((line_len = read (fd, buf, BUFSIZ - 1))
           > 0) {
        buf [line_len] = '\0';
        printf ("Вы ввели: %s", buf);
    }
    if (line_len == -1) {
        perror ("READ");
        close (fd);
        return (-1);
    }
    return (close (fd));
}
```

Листинг 5.10. Пример чтения из файла.

В качестве примера мобильного использования функции `fread()`, а также функций `feof()` и `ferror()`, рассмотрим программу, подсчитывающую число символов, слов и строк в файле – аргументе командной строки (см. листинг 5.11).

```
/* * * * * * * * * * * * * * * * * * * * * */
/* Подсчет символов, слов и строк в файле */
/* * * * * * * * * * * * * * * * * * * * * */

#include <stdio.h>
int main (int argc, char *argv[]) {
    long nwords = 0; /* Счетчик слов */
    long nlines = 0; /* Счетчик строк */
    long nchars = 0; /* Счетчик символов */
    FILE *fp = stdin; /* Если файл не задан, */
                      /* читается стандартный ввод */
    unsigned char buf [BUFSIZ]; /* Буфер для */
                                /* чтения файла */
    unsigned char *pl; /* Указатель на */
                      /* обрабатываемую часть буфера */
    size_t nbytes = 0; /* Количество прочитанных, */
                      /* но не обработанных байт */
    register int c; /* Обрабатываемый символ */
    int inword = 0; /* Признак - находимся ли мы */
                   /* внутри слова */

    if (argc > 2) {
        fprintf (stderr, "Использование: %s [файл]\n",
            argv [0]);
        return (2);
    }

    if (argc > 1 && (fp = fopen (argv [1], "r")) ==
        NULL) {
        perror ("OPEN");
        fprintf (stderr, "%s: Не могу открыть файл"
            "%s\n", argv [0], argv[1]);
        return (2);
    }

    pl = buf;

    for (;;) { /* Посимвольная обработка файла */
        if (nbytes == 0) { /* Нужно прочитать новую */
                           /* порцию */
            if (feof (fp)) {
                /* При предыдущем чтении дошли до конца файла */
                break;
            }
        }
    }
}
```

```
    }
    nbytes = fread (p1 = buf, (size_t) 1,
        sizeof (buf), fp);
    if (ferror (fp)) {
        perror ("READ");
        fprintf (stderr, "%s: Ошибка чтения из"
            "файла %s:", argv [0],
            argc == 1 ? "стандартного ввода" : argv [1]);
        break;
    }
    if (nbytes == 0) {
/* В файле не оставалось непрочитанных символов */
break;
    }
    nchars += nbytes;

    c = *p1++; /* Обработка одного символа */
    nbytes--;
    if (c > ' ') {
        if (!inword) {
            nwords++;
            inword = 1;
        }
        continue;
    }
    if (c == '\n') {
        nlines++;
    } else if (c != ' ' && c != '\t') {
        continue;
    }
    inword = 0;
}

if (argc > 1) {
    fclose (fp);
}

printf ("Файл %s: строк: %ld, слов: %ld,"
    "символов: %ld\n",
    argc == 1 ? "стандартного ввода" : argv [1],
    nlines, nwords, nchars);
```

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

Листинг 5.11. Программа, подсчитывающая число строк, слов и символов в файле.

Читателю предлагается убрать из цикла проверку feof (fp) и оценить, как изменится обработка интерактивного стандартного ввода.

Для иллюстрации использования функции readlink() напомним программу, выдающую на стандартный вывод содержимое символьных ссылок, имена которых заданы в командной строке (см. листинг 5.12).

```
#include <unistd.h>  
#include <stdio.h>  
/* Программа выдает на стандартный вывод содержимое */  
/* символьных ссылок - аргументов командной строки */  
int main (int argc, char *argv[]) {  
    char buf [BUFSIZ];  
    ssize_t link_len;  
    int err = 0;  
    int i;  
  
    for (i = 1; i < argc; i++) {  
        if ((link_len = readlink (argv [i], buf,  
                                sizeof (buf) - 1)) < 0) {  
            perror ("READLINK");  
            fprintf (stderr, "%s: Не удалось прочитать"  
                    "содержимое символьной ссылки %s\n", argv  
                    [0], argv [i]);  
            err = -1;  
            continue;  
        }  
        buf [link_len] = '\0';  
        printf ("Содержимое символьной ссылки %s ->"  
                "%s\n", argv [i], buf);  
    }  
    return (err);  
}
```

Листинг 5.12. Пример программы, читающей содержимое символьных ссылок.

Запись данных в файл выполняют функции `write()` и `fwrite()` (см. листинг 5.13).

```
#include <unistd.h>
ssize_t write (int fildes, const void *buf,
               size_t nbyte);
#include <stdio.h>
size_t fwrite (const void *restrict buf, size_t size,
               size_t nitems, FILE *restrict stream);
```

Листинг 5.13. Описание функций `write()` и `fwrite()`.

Вероятно, лишь несколько моментов, связанных с функциями `write()` и `fwrite()`, нуждаются в пояснениях. При записи размер файла может увеличиться. Если файл открыт на добавление, запись производится в его конец.

При записи в канал, если флаг `O_NONBLOCK` не установлен, процесс (поток управления) может быть отложен, но после нормального завершения функция `write()` вернет `nbyte`. При установленном флаге `O_NONBLOCK` поведение зависит от значения `nbyte` и наличия свободного места в канале. Если `nbyte` не превосходит константы `PIPE_BUF`, запишется все или ничего (в последнем случае результат будет равен `-1`). При попытке записать порцию данных большего размера запишется сколько можно или ничего.

Приведем несколько примеров. Следующая программа (см. листинг 5.14) выводит приветствие на управляющий терминал.

```
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#define C_TERM "/dev/tty"
char msg [] = "HELLO !!!\n";
int main (void) {
    int fd;
    /* Открытие на запись специального файла,      */
    /* ассоциированного с управляющим терминалом */
    if ((fd = open (C_TERM, O_WRONLY)) < 0) {
        perror ("OPEN");
        return (-1);
    }

    /* Вывод на терминал */
```

```
    if (write (fd, msg, sizeof (msg)) != (ssize_t)
        sizeof (msg)) {
        perror ("WRITE");
        return (1);
    }

    return (close (fd));
}
```

Листинг 5.14. Пример программы, использующей функцию `write()`.

Программа `prnmyself` (см. листинг 5.15) выводит свой собственный исходный текст. При этом применяется следующий прием: данные фиксированными порциями читаются из файла и выводятся на терминал; процесс повторяется до тех пор, пока число реально прочитанных байт совпадает с указанным (до обнаружения конца файла).

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

#define SOURCE_FILE "prnmyself.c"
#define C_TERM "/dev/tty"

int main (void) {
    unsigned char buf [BUFSIZ];
    int fdr, fdw; /* Дескрипторы для чтения и записи */
    ssize_t nb;

    if (((fdr = open (SOURCE_FILE, O_RDONLY)) < 0) ||
        ((fdw = open (C_TERM, O_WRONLY)) < 0)) {
        perror ("OPEN " SOURCE_FILE " or " C_TERM);
        return (1);
    }

    do {
        if ((nb = read (fdr, buf, BUFSIZ)) < 0) {
            perror ("READ");
            break;
        }
        if (write (fdw, buf, nb) != nb) {
```



```
        perror ("WRITE");
        break;
    }
} while (nb == BUFSIZ);

(void) close (fdw);
(void) close (fdr);

return (0);
}
```

Листинг 5.15. Пример программы, использующей функции `read()` и `write()`.

Для буферизованного ввода/вывода байт служат функции `fgetc()` и `fputc()`, строки рекомендуется вводить, вызывая функцию `fgets()`, а выводить с помощью функций `fputs()` и `puts()` (см. листинг 5.16).

```
#include <stdio.h>
int fgetc (FILE *stream);
#include <stdio.h>
int fputc (int c, FILE *stream);
#include <stdio.h>
char *fgets (char *restrict s, int n,
             FILE *restrict stream);
#include <stdio.h>
int fputs (const char *restrict s,
           FILE *restrict stream);
#include <stdio.h> int puts (const char *s);
```

Листинг 5.16. Описание функций `fgetc()`, `fputc()`, `fgets()`, `fputs()`, `puts()`.

Описание аналогичных функций для широких символов приведено в листинге 5.17.

```
#include <stdio.h>
#include <wchar.h>
wint_t fgetwc (FILE *stream);

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

```
wint_t fputwc (wchar_t wc, FILE *stream);

#include <stdio.h>
#include <wchar.h>
wchar_t *fgetws (wchar_t *restrict ws, int n,
    FILE *restrict stream);

#include <stdio.h>
#include <wchar.h>
int fputws (const wchar_t *restrict ws,
    FILE *restrict stream);
```

Листинг 5.17. Описание функций `fgetwc()`, `fputwc()`, `fgetws()`, `fputws()`.

Функция `fgetc()` пытается прочитать из заданного потока один байт, преобразовать его из типа `unsigned char` в `int` и вернуть в качестве результата. В случае ошибки или при достижении конца файла возвращается константа `EOF`.

Функция `fgets()` читает из заданного потока и помещает в буфер с адресом `s` (`n - 1`) байт или строку, включая символ перевода строки, или байты, оставшиеся до конца файла, если длина строки или число байт до конца меньше (`n - 1`). После прочитанных добавляется нулевой байт.

При нормальном завершении `fgets()` возвращает `s`. В случае ошибки или при достижении конца файла возвращается пустой указатель.

Функция `fputc()` помещает в поток значение `c`, предварительно преобразовав его к типу `unsigned char`. Результатом служит `c` или `EOF`.

Функция `fputs()` выводит в поток цепочку символов (без завершающего нулевого байта) и возвращает неотрицательное целое число или `EOF`. Функция `puts()` делает то же для потока `stdout`, завершая вывод переводом строки.

Функции для работы с потоками широкой ориентации устроены аналогично с точностью до замены `int` на `wint_t`, `char` на `wchar_t` и `EOF` на `WEOF`.

Программа, показанная в листинге 5.18, иллюстрирует использование функций `fgets()` и `fputs()`. Читателю предлагается сравнить ее с листингом 5.10.

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

```
/* Программа копирует строки со стандартного */
/* ввода на стандартный вывод */
int main (void) {
    char line [LINE_MAX];

    fputs ("Вводите строки\n", stdout);
    while (fgets (line, sizeof (line), stdin) !=
           NULL) {
        if ((fputs ("Вы ввели: ", stdout) == EOF) ||
            (fputs (line, stdout) == EOF)) {
            break;
        }
    }

    return (ferror (stdin) || ferror (stdout));
}
```

Листинг 5.18. Пример использования функций `fgets()` и `fputs()`.

Использование функций `fgetc()` и `fputc()` иллюстрируется программой, написанной С.В. Самборским (см. листинг 5.19). Она выполняет раскодировку файлов формата `base64`, применяемого, например, в электронной почте.

```
#include <stdio.h>
#include <assert.h>
#include <string.h>
#include <endian.h>

FILE *input=NULL, *output=NULL;
const char str []
    ="ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
    "uvwxyz0123456789+/";
unsigned int Prm [256]; /* Таблица перекодировки */
const int WHITE = 100, ERR = 101, END = 102;

static void usage (char argv0 []) {
    fprintf (stderr, "Программа раскодирует файлы"
             "формата base64\n");
    fprintf (stderr, "Использование:\n%s входной_файл"
             "выходной_файл\n", argv0);
}
```

```
fprintf (stderr, "Файл должен начинаться с"  
    "первого символа в кодировке base64.\n");  
}  
  
int main (int argc, char *argv []) {  
    int n;  
    union {  
        unsigned long l;  
        char c[4];  
    } a;  
  
    {  
        int i;  
        for (i = 0; i < 256; i++) Prm [i] = ERR;  
        Prm [' ' ] = WHITE; Prm ['\t'] = WHITE;  
        Prm ['\n'] = WHITE; Prm ['\r'] = WHITE;  
        Prm ['='] = END;  
        for (i = 0; i < 64; i++) Prm [(int) str [i]]  
            = i;  
    }  
  
    if (argc != 3) {  
        usage (argv [0]);  
        return (1);  
    }  
  
    assert (NULL != (input = fopen (argv [1], "r")));  
    assert (NULL != (output = fopen (argv [2], "w")));  
  
    for (a.l = 0, n = 0; ; ) { /* Цикл обработки */  
                                /* входного файла */  
        int c, b, shift;  
  
        assert (EOF != (c = fgetc (input)));  
        b = Prm [c];  
        if (WHITE == b) continue;  
        if (END == b) break;  
        if (ERR == b) {  
            fprintf (stderr, "Символ номер %d: %d не"  
                "входит в кодировку base64\n", n, c);  
            return (1);  
        }  
    }
```

```
n++;
assert (b < 64);
shift = 6 * (4 - n % 4);
if (shift != 24) b = b << shift;
a.l += b;

    if (0 == n % 4) {
#ifdef __BYTE_ORDER == __BIG_ENDIAN
        fputc (a.c[1], output); fputc (a.c[2], output);
        fputc (a.c[3], output);
#elif __BYTE_ORDER == __LITTLE_ENDIAN
        fputc (a.c[2], output); fputc (a.c[1], output);
        fputc (a.c[0], output);
#elif __BYTE_ORDER == __PDP_ENDIAN
        fputc (a.c[0], output); fputc (a.c[3], output);
        fputc (a.c[2], output);
#else
#error "Unknown endian"
#endif
        a.l = 0;
    }
}

{ /* Обработка остатка входного файла */
    int t1 = ((n - 1) % 4) * 6 + 7) / 8;

    if (t1 == 3) {
#ifdef __BYTE_ORDER == __BIG_ENDIAN
        fputc (a.c[1], output); fputc (a.c[2], output);
        fputc (a.c[3], output);
#elif __BYTE_ORDER == __LITTLE_ENDIAN
        fputc (a.c[2], output); fputc (a.c[1], output);
        fputc (a.c[0], output);
#elif __BYTE_ORDER == __PDP_ENDIAN
        fputc (a.c[0], output); fputc (a.c[3],
            output); fputc (a.c[2], output);
#else
#error "Unknown endian"
#endif
    }

    if (t1 == 2) {
```

```
#if __BYTE_ORDER == __BIG_ENDIAN
    fputc (a.c[1], output);
    fputc (a.c[2], output);
#elif __BYTE_ORDER == __LITTLE_ENDIAN
    fputc (a.c[2], output);
    fputc (a.c[1], output);
#elif __BYTE_ORDER == __PDP_ENDIAN
    fputc (a.c[0], output);
    fputc (a.c[3], output);
#else
#error "Unknown endian"
#endif
    }

    if (t1 == 1) {
#if __BYTE_ORDER == __BIG_ENDIAN
    fputc (a.c[1], output);
#elif __BYTE_ORDER == __LITTLE_ENDIAN
    fputc (a.c[2], output);
#elif __BYTE_ORDER == __PDP_ENDIAN
    fputc (a.c[0], output);
#else
#error "Unknown endian"
#endif
    }
}

fclose (input);
fclose (output);

return (0);
}
```

Листинг 5.19. Пример использования функций `fgetc()` и `fputc()`.

Приведенный пример показывает, что написание мобильных программ даже для сравнительно простых задач требует заметных усилий. В данном случае пришлось воспользоваться нестандартной возможностью – включаемым файлом `<endian.h>`, содержащим препроцессорные константы, которые описывают порядок байт в машинном слове.

Отметим также стиль обработки ошибочных ситуаций, основанный на применении макроса `assert`. Для авторов программ такой стиль, безусловно, упрощает жизнь, исходный текст получается более компактным, однако для пользователей сообщение вида

```
decode64: decode64.c:39: main: Assertion `((void *)0)
!= (input = fopen (argv [1], "r"))' failed.
```

может оказаться менее информативным, чем выдача функции `perror()`.

Управляющие операции с файлами и ассоциированными данными

К числу управляющих операций с файлами мы отнесем прежде всего позиционирование. Индикатор текущей позиции может быть опрошен или передвинут при помощи функции нижнего уровня `lseek()`, а также функций буферизованного ввода/вывода `fseek()`, `ftell()`, `ftello()`, `fgetpos()`, `fsetpos()`, `rewind()` (см. листинг 5.20).

```
#include <unistd.h>
off_t lseek (int fildes, off_t offset, int whence);
#include <stdio.h>
int fseek (FILE *stream, long offset, int whence);
long ftell (FILE *stream);
off_t ftello (FILE *stream);
int fgetpos (FILE *restrict stream,
             fpos_t *restrict pos);
int fsetpos (FILE *stream, const fpos_t *pos);
void rewind (FILE *stream);
```

Листинг 5.20. Описание функций `lseek()`, `fseek()`, `ftell()`, `ftello()`, `fgetpos()`, `fsetpos()`, `rewind()`.

Функция `lseek()` устанавливает индикатор текущей позиции следующим образом. Сначала, в зависимости от значения третьего аргумента, `whence`, выбирается точка отсчета: 0, если это значение равно `SEEK_SET`, текущая позиция для `SEEK_CUR` и размер файла для `SEEK_END`. Затем к точке отсчета прибавляется смещение `offset` (второй аргумент).

Индикатор текущей позиции можно сместить за конец файла (причем его размер не изменится). Если с этой позиции будет произведена

запись, в файле образуется дыра, чтение из которой выдает нулевые байты.

Результатом функции `lseek()` служит новое значение индикатора текущей позиции, отсчитанное в байтах от начала файла. В случае ошибки возвращается `(off_t) (-1)`, а текущая позиция остается прежней.

Функция `fseek()` по сути аналогична, только в случае нормального завершения возвращается 0. Кроме того, если поток имеет широкую ориентацию, значение аргумента `whence` должно равняться `SEEK_SET`, а значение `offset` — нулю или результату вызова функции `ftell()` для того же потока.

Функция `ftell()` возвращает значение индикатора текущей позиции для заданного потока (в случае ошибки — `(long) (-1)`). Функция `ftello()` эквивалентна `ftell()` с точностью до типа результата; в новых приложениях рекомендуется использовать `ftello()`, так как эта функция применима к большим файлам.

Функции `fgetpos()` и `fsetpos()` являются парными. Первая из них заполняет структуру, на которую указывает аргумент `pos`, а вторая применяет ее для установки индикатора текущей позиции. Нормальным результатом служит 0.

Функция `rewind()`, если пренебречь некоторыми тонкостями, сводится к вызову

```
(void) fseek (stream, 0L, SEEK_SET).
```

Приведем несколько примеров. В листинге 5.21 показана установка индикатора текущей позиции в начало и конец файла, а также его (индикатора) приращение.

```
(void) lseek (fildes, (off_t) 0, SEEK_SET);  
(void) lseek (fildes, (off_t) 0, SEEK_END);  
(void) lseek (fildes, inc, SEEK_CUR);
```

Листинг 5.21. Примеры вызова функции `lseek()`.

Применение функций буферизованного ввода/вывода иллюстрируется программой, показанной в листинге 5.22. Отметим, что она не является мобильной относительно смены порядка байт, поддерживаемого процессором.

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * */  
/* Набор функций для занесения текстов в файл, */
```



```

/* который составляется из двух частей: */
/* таблицы длин и смещений текстов от начала файла */
/* и собственно текстов */
/* */
/* В начало файла помещается специальный */
/* элемент таблицы с магическим числом */
/* и общим количеством текстов */
/* * * * * * * * * * * * * * * * * * * * * */

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

/* Магическое число файла с текстами */
#define G_TXT_MAGIC    0x1993

/* Элемент таблицы длин и смещений */
typedef struct {
    unsigned int l_txt;      /* Длина текста (без */
                           /* (нулевого байта) */
    unsigned long off_txt;   /* Смещение текста от */
                           /* начала файла */
} txt_table_elem;

static FILE *fp = NULL; /* Указатель на поток */
                           /* файла с текстами */
static unsigned long max_n_txt; /* Общее число */
                           /* текстов в файле */

/* * * * * * * * * * * * * * * * * * * * * */
/* Функция для инициализации набора. */
/* n_txts - максимальное число добавляемых текстов */
/* * * * * * * * * * * * * * * * * * * * * */
int g_init_add_txt (const int argc, char *argv [],
    const unsigned long n_txts) {
    char *path; /* Имя файла, куда нужно поместить */
               /* тексты */
    int magic; /* Магическое число файла с текстами */
    txt_table_elem tte;
    unsigned int i;

    if (argc != 2) {
        fprintf (stderr, "Использование: %s"

```

```
    "файл_для_текстов\n", argv [0]);
    return (-1);
}

path = argv [1];

/* Аккуратно откроем файл с текстами */
/* Если он уже есть и в нем правильное */
/* магическое число, */
/* будем добавлять тексты. */
/* В противном случае создадим и */
/* инициализируем файл */

if (((fp = fopen (path, "r+")) != NULL) &&
    (fread (&magic, sizeof (unsigned int), 1, fp)
     == 1) &&
    (magic == G_TXT_MAGIC)) {
    /* Перед нами - наш файл */
    /* Проверим, не превышает ли заказанная */
    /* верхняя граница существующую */
    if (fread (&max_n_txt, sizeof (unsigned long),
              1, fp) != 1) {
        fprintf (stderr, "Не удастся прочитать"
                  "информацию из файла %s\n", path);
        return (-1);
    }
    if (n_txts > max_n_txt) {
        fprintf (stderr, "***** Новая верхняя"
                  "граница числа сообщений %lu больше"
                  "существующей %lu\n", n_txts, max_n_txt);
    }
} else {
    /* Файла нет или он не наш */
    (void) fclose (fp);
    if ((fp = fopen (path, "w+")) == NULL) {
        fprintf (stderr, "Не удастся открыть файл"
                  "%s\n", path);
        return (-1);
    }
    tte.l_txt = magic = G_TXT_MAGIC;
    tte.off_txt = max_n_txt = n_txts;
    if (fwrite (&tte, sizeof (txt_table_elem), 1,
```

```
fp) != 1) {
    fprintf (stderr, "Не удастся записать"
             "информацию в файл %s\n", path);
    return (-1);
}

/* Пропишем нулями индексную таблицу */
/* Заодно конец файла окажется в будущем */
/* начале текстов */
tte.l_txt = 0;
tte.off_txt = 0;
for (i = 0; i < max_n_txt; i++) {
    if (fwrite (&tte, sizeof (txt_table_elem),
               1, fp) != 1) {
        fprintf (stderr, "Не удастся записать информацию"
                 "в файл %s\n", path);
        return (-1);
    }
}
} /* if - существует ли файл с текстами */

return 0;
}

/* * * * * * * * * * * * * * * * * * * * * * * * * */
/* Функция для добавления одного текста */
/* * * * * * * * * * * * * * * * * * * * * * * * */
int g_add_txt (const unsigned long n_t, const char
               *txt) {
    unsigned int l; /* Длина текста txt */
    txt_table_elem tte;

    if (n_t >= max_n_txt) {
        fprintf (stderr, "Номер текста: %lu должен"
                 "быть меньше: %lu\n", n_t, max_n_txt);
        return (-1);
    }

    l = strlen (txt);
    tte.l_txt = l;
    if (fseek (fp, 0L, SEEK_END)) {
        fprintf (stderr, "Ошибка позиционирования при"
```

```

        "добавлении текста номер %lu\n", n_t);
    return (-1);
}
tte.off_txt = ftell (fp);

if (fseek (fp, (n_t + 1) * sizeof
    (txt_table_elem), SEEK_SET)) {
    fprintf (stderr, "Ошибка позиционирования при"
        "добавлении текста номер %lu\n", n_t);
    return (-1);
}
if (fwrite (&tte, sizeof (tte), 1, fp) != 1) {
    fprintf (stderr, "Ошибка записи при добавлении"
        "текста номер %lu\n", n_t);
    return (-1);
}
if (fseek (fp, tte.off_txt, SEEK_SET)) {
    fprintf (stderr, "Ошибка позиционирования при"
        "добавлении текста номер %lu\n", n_t);
    return (-1);
}
if (fwrite (txt, sizeof (char), 1, fp) != 1) {
    fprintf (stderr, "Ошибка записи при добавлении"
        "текста номер %lu\n", n_t);
    return (-1);
}
return 0;
}

/* * * * * * * * * * * * * * * * * * * * * * * */
/* Функция для завершения добавления текстов      */
/* * * * * * * * * * * * * * * * * * * * * * * */
int g_term_add_txt () {
    return (fclose (fp));
}

/* * * * * * * * * * * * * * * * * * * * * * * */
/* Главная программа, вызывающая определенные     */
/* выше функции                                     */
/* * * * * * * * * * * * * * * * * * * * * * * */
#define MAX_TXTS 10240
int main (int argc, char *argv[]) {

```

```

if (g_init_add_txt (argc, argv, MAX_TXTS) ||
    g_add_txt (0, "Reference to section number"
               "%d in %s\n") ||
    g_add_txt (1, "Data .init section in"
               "%s\n")) {
    (void) g_term_add_txt ();
    return (-1);
}
return (g_term_add_txt ());
}

```

Листинг 5.22. Пример использования функций буферизованного ввода/вывода.

Функция `fcntl()` предназначена для выполнения разнообразных управляющих операций над открытым файлом (см. листинг 5.23).

```

#include <fcntl.h>
int fcntl (int fildes, int cmd, ...);

```

Листинг 5.23. Описание функции `fcntl()`.

Аргумент `fildes` задает дескриптор открытого файла, `cmd` — управляющую команду, дополнительные данные для которой могут быть переданы в качестве третьего (необязательного) аргумента `arg` (обычно имеющего тип `int`). В случае успешного завершения возвращаемое значение естественным образом зависит от команды; при неудаче всегда возвращается `-1`.

Допустимые команды (значения аргумента `cmd`) определены в файле `<fcntl.h>`. Перечислим сначала наиболее употребительные из них.

`F_DUPFD`

Дублирование дескриптора открытого файла: вернуть минимальный среди бывших незанятыми файловый дескриптор (не меньший, чем `arg`) и ассоциировать его с тем же описанием открытого файла, что и `fildes`.

`F_GETFL`

Вернуть флаги статуса и режим доступа к файлу (см. выше описание функции `open()`). Для выделения режима доступа из возвращаемого значения предоставляется маска `O_ACCMODE`.

F_SETFL

Установить флаги статуса файла в соответствии со значением третьего аргумента `arg`.

Приведем пример использования описанных команд функции `fcntl()`. Перенаправить стандартный вывод в файл, а затем стандартный протокол на стандартный вывод позволяет программа, показанная в листинге 5.24 (читателю рекомендуется сопоставить ее с программой перенаправления стандартного вывода с помощью функции `freopen()`, см. выше листинг 5.5). Кроме того, в ней демонстрируется изменение набора флагов статуса файла, ассоциированного с дескриптором.

```
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <assert.h>
#define LOGFILE "my_logfile"

int main (void) {
    int fd;
    int flags;

    assert ((fd = open (LOGFILE, O_WRONLY | O_CREAT
        | O_APPEND, S_IRWXU)) > 2);

    printf ("До перенаправления стандартного вывода"
        " в файл " LOGFILE "\n");
    close (1);
    assert (fcntl (fd, F_DUPFD, 1) == 1);
    close (fd);
    printf ("После перенаправления стандартного"
        " вывода в файл " LOGFILE "\n");

    /* Добавим флаг обеспечения целостности файла */
    /* при записи */
    assert ((flags = fcntl (1, F_GETFL, 0)) != -1);
    assert (fcntl (1, F_SETFL, flags | O_SYNC) !=
        -1);

    fprintf (stderr, "До перенаправления стандартного"
        " протокола на стандартный вывод\n");
```

```
close (2);
assert (fcntl (1, F_DUPFD, 2) == 2);
fprintf (stderr, "После перенаправления"
"стандартного протокола на стандартный вывод\n");

close (1);
close (2);

return (0);
}
```

Листинг 5.24. Пример перенаправления стандартного вывода в файл, а стандартного протокола – на стандартный вывод.

Команды `F_GETFD` и `F_SETFD` функции `fcntl()` предназначены, соответственно, для опроса и изменения флагов, ассоциированных с заданным файловым дескриптором (а не с файлом). В стандарте упомянут один такой флаг – `FD_CLOEXEC`, предписывающий закрывать дескриптор при смене программы процесса.

Особый класс управляющих операций с файлами, имеющих свою систему понятий, составляют блокировки, хотя и они оформляются как команды функции `fcntl()`.

Назначение механизма блокировок – дать возможность процессам (потокам управления), одновременно обрабатывающим одни и те же данные, синхронизировать свою работу. В стандарте POSIX-2001 представлены только так называемые рекомендательные блокировки, которые взаимодействуют исключительно с другими блокировками и не препятствуют операциям ввода/вывода. Это значит, что для достижения синхронизации процессы должны окружать операции ввода/вывода разделяемых данных операциями установки и снятия соответствующей блокировки.

Блокировка связывается с сегментом (произвольным последовательным участком) файла. Различают блокировку на запись и на чтение. Блокировка на запись носит монопольный характер. Когда сегмент заблокирован на запись, никакие другие процессы не имеют возможности заблокировать на чтение или запись этот же или пересекающийся с ним сегмент.

Блокировка на чтение используется для того, чтобы ограничить доступ к сегментам: если сегмент заблокирован на чтение, то другие процессы также могут заблокировать на чтение весь сегмент или его часть, однако никакие сегменты, заблокированные на запись, с ним не пересекаются.

Для установки блокировки на чтение требуется, чтобы файл был открыт как минимум на чтение. Соответственно, для блокировки на запись необходим доступ на запись.

Блокировка сегмента описывается структурой `flock`, которая, согласно стандарту, должна содержать по крайней мере следующие поля:

```
short l_type /* Тип блокировки: */
              /* F_RDLCK (на чтение) */
              /* F_WRLCK (на запись), */
              /* F_UNLCK (снятие блокировки) */
short l_whence /* Точка отсчета для смещения */
              /* l_start: SEEK_SET, SEEK_CUR */
              /* или SEEK_END */
off_t l_start /* Смещение в байтах начала */
              /* блокируемого сегмента */
              /* относительно точки отсчета */
off_t l_len /* Размер блокируемого сегмента. */
            /* 0 означает блокировку до конца файла. */
pid_t l_pid /* Идентификатор процесса, */
            /* установившего блокировку; */
            /* (возвращается по команде F_GETLK, */
            /* см. далее) */
```

Отметим, что блокируемый сегмент может выходить за конец, но не за начало файла. Длину сегмента разрешается задавать отрицательным числом, если тип `off_t` допускает такую возможность; в таком случае смещение начала блокируемого сегмента относительно точки отсчета равно `l_start+l_len`, а смещение конца — `l_start-1`.

Приведем несколько примеров заполнения структуры `flock` (см. листинг 5.25).

```
#include <fcntl.h>
struct flock lck;

. . .

lck.l_type = F_RDLCK; /* Блокировка на чтение */
                    /* всего файла */
lck.l_whence = SEEK_SET;
lck.l_start = 0;
lck.l_len = 0;
```



```
    . . .  
    lck.l_type = F_WRLCK; /* Блокировка на запись */  
                        /* сегмента файла */  
    lck.l_whence = SEEK_CUR;  
    lck.l_start = inc;  
    lck.l_len = sizeof (struct rcrd);
```

Листинг 5.25. Примеры заполнения структуры flock.

Установка блокировки осуществляется управляющими командами `F_SETLK` и `F_SETLKW` функции `fcntl()` (см. листинг 5.26).

```
if (fcntl (fd, F_SETLK, &lck) != -1) ...  
if (fcntl (fd, F_SETLKW, &lck) != -1) ...
```

Листинг 5.26. Примеры вызова функции 2 для установки блокировок.

Если блокировка не может быть установлена, выполнение команды `F_SETLK` немедленно завершается, и возвращается `-1`. Команда `F_SETLKW` отличается только тем, что в аналогичной ситуации процесс переходит в состояние ожидания до тех пор, пока нужный сегмент файла не будет разблокирован.

Для снятия блокировки можно воспользоваться командами `F_SETLK` или `F_SETLKW`. Для этого значение поля `l_type` должно быть установлено равным `F_UNLCK`.

Получить характеристики блокировки, мешающей установить новую, позволяет управляющая команда `F_GETLK` (новая блокировка задается структурой, на которую при обращении к `fcntl()` указывает третий аргумент `arg`). Если запрашиваемую блокировку установить нельзя, информация о первой помещается в ту же структуру; в частности, будет задано значение поля `l_pid` — оно идентифицирует процесс, установивший блокировку. (Естественно, значение поля `l_whence` будет установлено равным `SEEK_SET`.) Если нет помех для создания нужной блокировки, полю `l_type` присваивается значение `F_UNLCK`, а остальные поля в структуре не изменяются.

При закрытии файлового дескриптора все блокировки, установленные в файле текущим процессом, снимаются.

Приведем пример двух программ, первая из которых (назовем ее `set_locks`) устанавливает блокировки нескольких сегментов файла и засыпает на некоторое время (см. листинг 5.27), а вторая (`test_locks`, см. листинг 5.28), выполняющаяся, разумеется, параллельно, в рамках

другого процесса, получает и выводит информацию о заблокированных сегментах того же файла (см. листинг 5.29).

```
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <assert.h>
#define LOCKFILE "my_lockfile"

/* Программа устанавливает несколько блокировок */
/* на файл LOCKFILE */
int main (void) {
    int fd;
    struct flock lck;

    assert ((fd = open (LOCKFILE, O_RDWR | O_CREAT |
        O_TRUNC, S_IRWXU)) != -1);

    /* Установим блокировку на запись на весь файл */
    lck.l_type = F_WRLCK;
    lck.l_whence = SEEK_SET;
    lck.l_start = (off_t) 0;
    lck.l_len = (off_t) 0;
    if (fcntl (fd, F_SETLK, &lck) == -1) {
        perror ("FCNTL-F_SETLK-1");
        close (fd);
        return (-1);
    }

    /* Сделаем размер файла ненулевым */
    if (lseek (fd, (off_t) 1024, SEEK_SET) == -1) {
        perror ("LSEEK");
        close (fd);
        return (-1);
    }
    if (write (fd, &lck, sizeof (lck)) != sizeof
        (lck)) {
        perror ("WRITE");
        close (fd);
        return (-1);
    }
}
```

```
/* Снимем блокировку в середине файла */
lck.l_type = F_UNLCK;
lck.l_whence = SEEK_SET;

lck.l_start = (off_t) 512;
lck.l_len = (off_t) sizeof (lck);
if (fcntl (fd, F_SETLK, &lck) == -1) {
    perror ("FCNTL-F_SETLK-2");
    close (fd);
    return (-1);
}

/* Установим блокировку на чтение в конце файла */
lck.l_type = F_RDLCK;
lck.l_whence = SEEK_END;
lck.l_start = (off_t) -sizeof (lck);
lck.l_len = (off_t) sizeof (lck);
if (fcntl (fd, F_SETLK, &lck) == -1) {
    perror ("FCNTL-F_SETLK-2");
    close (fd);
    return (-1);
}

sleep (10);

return (close (fd));
}
```

Листинг 5.27. Пример программы `set_locks`, устанавливающей блокировки файла.

```
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <assert.h>
#define LOCKFILE "my_lockfile"

/* Программа выявляет блокировки, установленные */
/* на файл LOCKFILE */
int main (void) {
    int fd;
    struct flock lck;
```

```
assert ((fd = open (LOCKFILE, O_WRONLY)) != -1);
(void) printf ("ид-р проц. тип начало длина\n");
/* Начнем с попытки установить блокировку на */
/* весь файл */
lck.l_whence = SEEK_SET;
lck.l_start = 0;
lck.l_len = 0;

do {
    lck.l_type = F_WRLCK;
    (void) fcntl (fd, F_GETLK, &lck);
    if (lck.l_type != F_UNLCK) {
        (void) printf ("%9d %3c %7ld %5ld\n",
            lck.l_pid,
            (lck.l_type == F_WRLCK) ? 'W' : 'R',
            lck.l_start, lck.l_len);
        /* Если эта блокировка покрывает остаток файла, */
        /* нет нужды выявлять другие блокировки */
        if (lck.l_len == 0) break;
        /* Иначе поищем новую блокировку после найденной */
        lck.l_start += lck.l_len;
    }
    while (lck.l_type != F_UNLCK);
return (close (fd));
}
```

Листинг 5.28. Пример программы `test_locks`, выявляющей блокировки файла.

ид-р проц.	тип	начало	длина
31174	W	0	512
31174	W	528	496
31174	R	1024	16
31174	W	1040	0

Листинг 5.29. Возможный результат выполнения командной строки `set_locks & test_locks`.

Отметим, что блокировка на чтение расщепила блокировку на запись, первоначально покрывавшую весь файл.

Читателю предлагается выполнить командную строку `set_locks & set_locks` и объяснить полученный результат.

Функции `setbuf()`, `setvbuf()` и `fflush()` выполняют управляющие операции с буферами потоков (см. листинг 5.30).

```
#include <stdio.h>
void setbuf (FILE *restrict stream,
             char *restrict buf);
#include <stdio.h>
int setvbuf (FILE *restrict stream,
             char *restrict buf, int type, size_t size);
#include <stdio.h>
int fflush (FILE *stream);
```

Листинг 5.30. Описание функций `setbuf()`, `setvbuf()` и `fflush()`.

Функция `setvbuf()`, которую можно использовать после открытия файла, но до первой операции ввода/вывода, устанавливает режим буферизации в соответствии со значением своего третьего аргумента, `type`:

- `_IOFBF` – полная буферизация;
- `_IOLBF` – построчная буферизация;
- `_IONBF` – отсутствие буферизации.

Функция `setvbuf()` сама резервирует буфера заданного размера, но если аргумент `buf` не является пустым указателем, может использоваться и пользовательский буфер. В случае нормального завершения функция возвращает 0.

Функцию `setbuf()` можно считать частным случаем `setvbuf()`. С точностью до возвращаемого значения вызов `setbuf (stream, NULL)` эквивалентен `setvbuf (stream, NULL, _IONBF, BUFSIZ)` (отмена буферизации); если же значение `buf` не равно `NULL`, вызов `setbuf (stream, buf)` сводится к `setvbuf (stream, buf, _IOFBF, BUFSIZ)` (полная буферизация).

Функцию `setbuf()` чаще всего применяют для отмены буферизации стандартного вывода и/или стандартного протокола, выполняя вызовы `setbuf (stdout, NULL)` и/или `setbuf (stderr, NULL)`.

Использование функций `setbuf()` и `setvbuf()` требует известной аккуратности. Типичная ошибка – указание в качестве аргумента `buf` автоматического массива, определенного внутри блока, и продолжение работы с потоком после выхода из этого блока.

Функция `fflush()` выталкивает буфера, помещая в файл предназначенные для записи, но еще не записанные данные. Если в качестве аргумента `stream` задан пустой указатель, выталкиваются все буфера всех потоков.

Вызов `fflush()` – необходимый элемент завершения транзакций, но он полезен и применительно к стандартному выводу (протоколу), если нужно выдать приглашение для пользовательского ввода на экран, а не в буфер (см. листинг 5.31).

```
char name [LINE_MAX];  
(void) printf ("Введите Ваше имя: ");  
    (void) fflush (stdout);  
    (void) fgets (name, sizeof (name), stdin);
```

Листинг 5.31. Пример использования функции `fflush()`.

Вариант 1

1. Согласно стандарту POSIX-2001, операция асинхронного ввода/вывода

- ☐ всегда приводит к приостановке процесса
- ☐ может привести к приостановке процесса
- ☐ не приводит к приостановке процесса

2. open() имеет

- ☐ два аргумента
- ☐ три аргумента
- ☐ переменное число аргументов

3. В стандарте POSIX-2001 представлены

- ☐ только рекомендательные блокировки
- ☐ только принудительные блокировки
- ☐ и рекомендательные, и принудительные блокировки

Вариант 2

1. Согласно стандарту POSIX-2001, операция синхронного ввода/вывода

- ☐ всегда приводит к приостановке процесса
- ☐ может привести к приостановке процесса
- ☐ не приводит к приостановке процесса

2. Функция `fcntl()` имеет

- ☐ два аргумента
- ☐ три аргумента
- ☐ переменное число аргументов

3. При использовании механизма блокировок тупики

- ☐ невозможны
- ☐ возможны
- ☐ неизбежны

Вариант 3

1. **Согласно стандарту POSIX-2001, для операций синхронизированного ввода/вывода различают следующие виды целостности**
 - ☐ целостность данных
 - ☐ целостность файлов
 - ☐ целостность кэшей

2. **Необязательный третий аргумент функции `fcntl()`**
 - ☐ всегда имеет тип `int`
 - ☐ может иметь тип `int`
 - ☐ не может иметь тип `int`

3. **Блокировка файла на запись**
 - ☐ может пересекаться с другой блокировкой на запись
 - ☐ может пересекаться с другой блокировкой на чтение
 - ☐ не может пересекаться с другой блокировкой

Лекция 6. Средства обработки структурированных данных

Описываются служебные программы, осуществляющие обработку текстовых файлов. Приводятся примеры их комбинирования. Определяется понятие регулярного выражения. Рассматриваются средства обработки каталогов.

Ключевые слова: конкатенация файлов, просмотр файлов, фильтр, страница, заголовок, хвостовик, столбец, отступ, сортировка, ключ сортировки, слияние, именованный символ, слово, контроль целостности, контрольная сумма, регулярное выражение, цепочка символов, подцепочка, сопоставление, базовое регулярное выражение, расширенное регулярное выражение, конкатенация регулярных выражений, полное регулярное выражение, односимвольное регулярное выражение, многосимвольное регулярное выражение, специальный символ, класс символов, элемент алфавитного сравнения, подвыражение, обратная ссылка, интервальное выражение, фиксация границ, фиксатор начала, фиксатор конца, повторитель, операция логического ИЛИ, потоковый редактор, сценарий, действие, запись, разделитель записей, поле, разделитель полей, оператор, переменная, выражение, встроенная функция, специальная переменная, компиляция регулярного выражения, трансляция символов, сжатие последовательности символов, сжатие последовательности строк, реляционная операция, проекция, горизонтальное соединение, соединение отношений, каталог, элемент каталога, открытие каталога, чтение элемента каталога, позиционирование, шаблон имен файлов, сопоставление, генерация маршрутных имен, выделение имени каталога.

Простейшие средства обработки текстовых файлов

Согласно определению, данному в стандарте POSIX-2001, текстовым называется файл, символы которого объединены в строки длиной не более `LINE_MAX`, не содержащие нулевого символа.

Вероятно, простейшей операцией с файлами (не обязательно текстовыми) является их выдача на стандартный вывод, что обеспечивается посредством утилиты

```
cat [-u] [файл ...]
```

Несмотря на внешнюю простоту, применение служебной программы `cat` сопряжено с рядом тонкостей. Во-первых, опция `-u` предписывает передавать на стандартный вывод байты из входных файлов без задержек, это означает, в частности, отмену буферизации стандартного вывода. Во-вторых, если не указаны исходные файлы или в качестве аргумента задан знак минус, используется стандартный ввод. Минус может фигурировать в командной строке несколько раз, стандартный ввод не закрывается и не переоткрывается. Например, допустимы команды, показанные в листинге 6.1, причем первая обеспечит ввод двух интерактивных вставок, а вторая эквивалентна третьей (читателю предлагается самостоятельно объяснить данный факт).

```
cat f1.txt - f2.txt - f3.txt
cat g1.txt - g2.txt - g3.txt < input.txt
cat g1.txt - g2.txt /dev/null g3.txt < input.txt
cat f1.txt f2.txt > f1.txt
```

Листинг 6.1. Пример использования утилиты `cat`.

В-третьих, нужно учитывать особенности перенаправления вывода командным интерпретатором `shell`. В общем случае утилита `cat` выдает результат конкатенации содержимого файлов — аргументов, но выполнение четвертой команды, показанной в листинге 6.1, приведет к утрате содержимого файла `f1.txt`.

Для просмотра больших текстовых файлов предпочтительнее пользоваться служебной программой

```
more [опция ...] [файл ...]
```

Она разбивает вывод на страницы (экраны) и выдает их по явному указанию пользователя (например, по нажатию клавиши пробела).

Программа `more` — довольно мощный просмотрщик файлов со своей системой команд, которую мы, однако, описывать не будем. Отметим лишь возможности поиска и перемещения по файлам. Трактовка аргументов аналогична `cat`, а из опций выделим три.

`-n` число

Задаёт число строк, выдаваемых на экран.

`-p` команды `more`

Команды `more` выполняются после входа в очередной файл и вывода его первого экрана. Это может быть, к примеру, команда поиска.

8

Сжимать последовательные пустые строки в одну.

Утилита `more` часто используется как заключительное звено конвейеров для удобного просмотра результатов.

Для просмотра нетекстовых файлов рекомендуется служебная программа

```
od [опция ...] [файл ...]
```

Она выдает на стандартный вывод содержимое исходных файлов в формате, заданном пользователем, а также в виде блоков, каждый из которых в простейшем (и наиболее употребительном) случае состоит из шестнадцати байт и занимает при выводе одну строку. В начале подобной строки располагается смещение блока от начала файла, затем следует содержимое блока. С помощью аргумента опции `-A` можно задать основание системы счисления для смещений (`d` – десятичное, `o` – восьмеричное, `x` – шестнадцатеричное, `n` – не выдавать смещение). Формат выдачи содержимого определяется аргументом опции `-t`. Помимо перечисленных типов могут быть заданы `a` (именованные символы), `c` (символы), `f` (вещественные числа), `u` (беззнаковые десятичные). За спецификациями `d`, `f`, `o`, `u` и `x` может следовать десятичное число – количество байт в одном значении заданного типа; за спецификацией `f` – символы `F`, `D` или `L`, указывающие тип вещественных чисел (`float`, `double` или `long double`, соответственно), а за спецификациями `d`, `o`, `u`, `x` – спецификаторы целочисленного типа `C` (`char`), `S` (`short`), `I` (`int`) или `L` (`long`).

Рассмотрим пример. Пусть файл `ascii.tab` содержит байты от 0 до 127 в порядке возрастания. Тогда результат выполнения команды, показанной в листинге 6.2, будет таким, как в листинге 6.3.

```
od -A x -t a ascii.tab
```

Листинг 6.2. Пример использования утилиты `od`.

```
000000 nul soh stx etx eot enq ack bel bs ht nl
      vt ff cr so si
000010 dle dc1 dc2 dc3 dc4 nak syn etb can em sub
      esc fs gs rs us
```

```
000020 sp ! " # $ % & ' ( ) * + , - . /
000030 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
000040 @ A B C D E F G H I J K L M N O
000050 P Q R S T U V W X Y Z [ \ ] ^ _
000060 ` a b c d e f g h i j k l m n o
000070 p q r s t u v w x y z { | } ~ del
000080
```

Листинг 6.3. Возможный результат использования утилиты `od`.

Полезная возможность — взглянуть на начало файла, чтобы понять, нуждается ли он в более детальном изучении. Эта возможность реализуется служебной программой `head`:

```
head [-n число] [файл ...]
```

Утилита `head` копирует на стандартный вывод указанное число (по умолчанию — 10) начальных строк исходных файлов (или весь файл, если он слишком короткий).

«Симметричная» служебная программа — `tail`

```
tail [-f] [-c число | -n число] [файл]
```

выдает на стандартный вывод конец исходного файла, начиная с указанного места. Если задана опция `-c`, единицей измерения является байт, если `-n` — строка. Числовой аргумент опции может быть задан со знаком; знак `+` означает, что отсчет байт (строк) идет от начала файла.

Опция `-f` полезна для слежения за растущими файлами, в которые пишет какой-либо другой процесс. При наличии этой опции утилита `tail` не завершается по достижении конца файла, но ждет появления новых данных и выводит их.

Рассмотрим примеры. Команда

```
tail -f f1
```

выдаст последние десять строк файла `f1`, за которыми последуют строки, добавленные к файлу `f1` в промежуток времени между запуском и терминованием утилиты `tail`.

В качестве примера совместного использования служебных программ `head` и `tail` приведем фрагмент одного из командных файлов ОС Linux (см. листинг 6.4).

```

echo "Reloading automounter: checking for changes ... "
TMP=/var/run/autofs.tmp
getmounts > $TMP
for i in /var/run/autofs.*.pid
do
    pid=`head -n 1 $i 2>/dev/null`
    [ "$pid" = "" ] && continue
    command=`tail -n +2 $i`
    if ! grep -q "^$command" $TMP
    then
        echo "Stopping automounter: $command"
        kill -USR2 $pid
    fi
done
rm -f $TMP

```

Листинг 6.4. Пример использования служебных программ `head` и `tail`.

Отметим, что здесь утилиты `head` и `tail` делят исходный файл «по-братски»: первой достается первая строка, второй — все остальные.

Служебная программа `pr`

```

pr [+номер_страницы] [-число_столбцов] [-adFmrt]
    [-e{символ}[число]]
    [-h заголовок] [-i{символ}[число]] [-l
        число_строк] [-n{символ}[ширина]]
    [-o отступ] [-s{символ}] [-w ширина] [файл ...]

```

является фильтром для печати и оформления страниц. По умолчанию задача разбивается на страницы, каждая из которых содержит в пятистрочном заголовке свой номер, дату, время и имя файла. Длина страницы 66 строк, включая пять пустых строк хвостовика.

Утилита `pr` обрабатывает следующие аргументы командной строки.

`+номер_страницы`

Начать вывод со страницы с заданным номером (по умолчанию с первой).

-число_столбцов

Вывод в заданное число столбцов (по умолчанию в один). При выводе в несколько столбцов автоматически действуют опции -e и -i. Этот аргумент несовместим с опцией -m.

-a

Выдача в несколько столбцов с упорядочением по строкам. Например, если заказана печать в два столбца, первая исходная строка попадает в первый столбец, вторая — во второй, третья — снова в первый и т.п.

-d

Выдача через строку.

-e [символ] [число]

Установка табуляции через заданное число позиций, начиная с первой. Символы табуляции во входном файле разворачиваются в соответствующее количество пробелов. Если задан любой нецифровой символ, он трактуется как символ табуляции во входном файле.

-F

Применять символы перехода к новой странице (по умолчанию переход на новую страницу осуществляется при помощи последовательности переводов строк).

-h заголовок

Использовать заданный заголовок (по умолчанию в качестве заголовка выступает имя исходного файла).

-i [символ] [число]

При выводе, где это возможно, последовательности пробельных символов заменяются символами табуляции, при условии, что табуляции установлены через заданное число позиций, начиная с первой. Если задан любой нецифровой символ, он выводится в качестве символа табуляции.

-l число_строк

Установка длины страницы (по умолчанию 66 строк).

-m

Слияние и печать всех файлов параллельно, по одному в столбце. Реализация должна поддерживать слияние по крайней мере девяти файлов.

-n [символ] [ширина]

Производится нумерация строк. Под номер отводится поле заданной ширины (по умолчанию – 5). Если задан любой нецифровой символ, то он присоединяется к номеру строки, отделяя ее от последующего текста (подразумеваемым значением является символ табуляции).

-o отступ

Установка отступа слева на заданное число позиций. Количество позиций в каждой строке есть сумма ширины и отступа.

-p

Пауза перед выдачей каждой страницы, если вывод назначен на терминал (prg включит звонок и будет ждать нажатия клавиши возврата каретки).

-r

Подавление диагностических сообщений при ошибках открытия файлов.

-s [символ]

Разделение столбцов единственным символом (по умолчанию – табуляцией), а не соответствующим количеством пробелов.

-t

Подавление выдачи пятистрочного идентифицирующего заголовка и такого же размера хвостовика, сопровождающих обычно каждую страницу.

-w ширина

Установка ширины строки (по умолчанию – 72 позиции). Эта опция действует только при печати в несколько столбцов.

В качестве примера использования служебной программы `pr` приведем конвейер, позволяющий получить пронумерованный список всех файлов текущего каталога:

```
ls -a | pr -2 -a -l 24 -n' '3 -h  
"Список файлов каталога $(pwd) "
```

Начальный фрагмент результата может выглядеть так, как показано в листинге 6.5.

```
2003-10-17 17:24  Список файлов каталога /var/tmp Page 1  
1 .                2 ..  
3 Blank.ReS        4 Make45.ReS  
5 Make46.ReS        6 from_cvs  
7 gcc-20032204      8 gcc-3.4-16-jun-2003  
9 htdocs            10 rpm-tmp.29785  
. . .
```

Листинг 6.5. Начальный фрагмент возможного результата работы служебной программы `pr`.

Для подсчета числа символов, слов и строк в файлах служит утилита `wc`:

```
wc [-c ' ' -m] [-w] [-l] файл ...
```

Опция `-c` предписывает подсчитывать байты, `-m` – символы, `-w` – слова, `-l` – строки. По умолчанию подсчитываются символы, слова и строки. Если в командной строке задано несколько файлов, помимо результатов для каждого из них выводятся также суммарные показатели.

Рассмотрим пример. Предположим, что результаты, показанные в листинге 6.3, сохранены в файле `ascii.od`. Тогда команда, приведенная в листинге 6.6, выдаст результат, показанный в листинге 6.7.

```
wc ascii.od
```

Листинг 6.6. Пример использования утилиты `wc`.

```
9 137 575 ascii.od
```

Листинг 6.7. Возможный результат использования утилиты `wc`.

Иными словами, в указанном файле 575 символов, 137 слов, расположенных на 9 строках.

Служебная программа `sort`

```
sort [-m] [-o выходной_файл] [-bdfinru] [-t символ]
    [-k определение_ключа] ... [файл ...]
sort -c [-bdfinru] [-t символ] [-k определение_ключа]
    [файл]
```

в зависимости от заданных опций выполняет одно из трех возможных действий:

- сортировку строк всех исходных файлов с записью результата в выходной файл;
- слияние всех исходных (предварительно отсортированных) файлов с записью результата в выходной файл;
- проверку того, что единственный исходный файл действительно отсортирован.

При упорядочении используется один или несколько ключей сортировки, выделяемых из каждой вводимой строки. По умолчанию ключ сортировки один — вся строка, а порядок является лексикографическим.

Следующие опции управляют порядком работы утилиты `sort`.

`-c`

Проверить, является ли (единственный) исходный файл уже отсортированным. В выходной файл ничего не записывается, результат работы определяется по коду завершения (0 — успешное завершение, 1 — данные в файле не упорядочены должным образом, больше единицы — зафиксирована ошибка).

`-m`

Слияние исходных файлов, которые предполагаются отсортированными.

`-o выходной_файл`

Результат направляется не на стандартный вывод, а в выходной_файл, который может совпадать с одним из исходных.

-u

Опция уникальности: из всех совпадающих строк выводить только одну, при наличии опции -c контролировать отсутствие строк с совпадающими ключами сортировки.

Следующие опции изменяют подразумеваемый способ сравнения. Если они употреблены независимо от определения ключей сортировки, то диктуемые ими правила действуют глобально, на все ключи.

-d

«Словарный» порядок: при сравнении являются значимыми только буквы, цифры и пробельные символы.

-f

При сравнении преобразовывать малые буквы в большие.

-i

При сравнении игнорировать непечатные символы.

-n

Числовое сравнение. Ограничить ключ сортировки начальной числовой цепочкой, которая может содержать пробельные символы, знак минус, цифры, символ основания системы счисления и разделители тысяч.

-r

Заменить результат сравнения на противоположный.

Следующие опции изменяют трактовку разделителей полей.

-b

Игнорировать начальные пробельные символы при определении начала и конца ключей сортировки. Если опция -b указана перед первой опцией -k, она воздействует на все ключи, снабженные опцией -k.

-t символ

Использовать заданный символ как разделитель полей. Разделитель не является частью поля (хотя и может входить в ключ сортировки). Каж-

дое вхождение разделителя значимо, т. е. два рядом стоящих разделителя ограничивают пустое поле. В отсутствие опции `-t` подразумеваемыми разделителями становятся пробельные символы, точнее, максимальная непустая последовательность таких символов, а в поле включаются предшествующие ему разделители.

Ключи сортировки определяются с помощью опции `-k`. Определение ключа имеет следующий формат:

```
начало_поля [тип] [ , конец_поля [тип] ]
```

Ключевое поле охватывает все позиции — от начальной до конечной, которые задаются, соответственно, как

```
номер_поля [ . первый_символ ]  
номер_поля [ . последний_символ ]
```

Тип — это один из модификаторов `b, d, f, i, n, r`. Модификатор `b` ведет себя аналогично опции `-b`, но применяется только к той границе ключа, после которой он указан. Остальные модификаторы также аналогичны одноименным опциям, но воздействуют на ключ в целом.

Если указано несколько ключей сортировки (согласно стандарту, реализацией должно поддерживаться по крайней мере девять вхождений опции `-k`), то более поздние используются только в случае равенства более ранних. Если значения ключей сортировки двух строк совпадают, строки упорядочиваются с учетом всех символов.

Приведем примеры использования служебной программы `sort`. Автор хранит данные о собственном кровяном давлении в виде файла `pp.txt`, содержащего пять столбцов: дата, время, верхнее давление, нижнее давление, пульс (см. листинг 6.8). Естественно, строки добавляются к файлу в хронологическом порядке. Чтобы упорядочить их по убыванию верхнего давления (третий столбец), можно воспользоваться командой, показанной в листинге 6.9. Возможный результат приведен в листинге 6.10.

12.05.2000	17:30	200	125	120
15.05.2000	17:00	130	80	70
17.05.2000	10:30	150	90	70
17.05.2000	21:45	154	99	74
19.05.2000	10:05	158	83	89
21.05.2000	21:00	161	104	64
22.05.2000	21:00	147	104	69

Листинг 6.8. Пример исходных данных для служебной программы `sort`.

```
sort -r -k 3,3 pp.txt
```

Листинг 6.9. Пример использования служебной программы `sort`.

```
12.05.2000 17:30 200      125    120
18.11.2000 19:30 172      107     68
04.07.2002 09:00 170       98     85
30.10.2001 13:00 168       94     88
27.12.2000 20:30 166       98     69
23.05.2002 10:00 166      104     56
22.05.2002 10:00 166      103     57
23.10.2001 11:00 165       88     88
```

* * *

Листинг 6.10. Возможный результат работы служебной программы `sort`.

Если ключи сортировки равны, строки упорядочиваются с учетом всех байт. В рассматриваемом примере это не очень удобно, поскольку они начинаются с номера дня в месяце. Более естественно либо расширить ключ сортировки путем охвата полей нижнего давления и пульса, либо сделать эти поля дополнительными ключами. Оба варианта показаны в листинге 6.11, а результат сортировки – в листинге 6.12. Отметим, что второй вариант предпочтительнее, поскольку он устойчив к ошибкам выравнивания столбцов.

- # Первый вариант – ключ сортировки покрывает несколько полей `sort -r -k 3,5 pp.txt`
- # Второй вариант – используется несколько ключей сортировки `sort -n -r -k 3,3 -k 4,4 -k 5,5 pp.txt`

Листинг 6.11. Два варианта использования служебной программы `sort`.

```
12.05.2000 17:30 200      125    120
18.11.2000 19:30 172      107     68
04.07.2002 09:00 170       98     85
30.10.2001 13:00 168       94     88
23.05.2002 10:00 166      104     56
22.05.2002 10:00 166      103     57
27.12.2000 20:30 166       98     69
23.10.2001 11:00 165       88     88
```

* * *

Листинг 6.12. Результат работы служебной программы `sort` с несколькими ключами сортировки.

Предположим теперь, что данные о давлении и пульсе разбиты по годам и в уже отсортированном виде хранятся в файлах `pp2000.sorted`, `pp2001.sorted` и т.д. Для их слияния можно воспользоваться командной строкой или циклом из листинга 6.13. Опять-таки второй вариант предпочтительнее по причине его устойчивости к количеству исходных файлов.

```
# Первый вариант слияния файлов sort -m -o
  pp.sorted -n -r -k 3,3 -k 4,4 -k 5,5 pp2*.sorted
# Второй вариант слияния файлов
> pp.sorted
for f in pp2*.sorted
do
    sort -m -o pp.sorted -n -r -k 3,3 -k 4,4 -k 5,5
    $f pp.sorted
done
```

Листинг 6.13. Два варианта слияния файлов с помощью служебной программы `sort`.

Проверить правильность слияния можно с помощью команд, показанных в листинге 6.14.

```
name=pp.sorted
if sort -c -r -n -k 3,3 -k 4,4 -k 5,5 $name
then
    echo Данные в файле $name отсортированы верно
elif
    echo Данные в файле $name отсортированы неверно
fi
```

Листинг 6.14. Проверка правильности упорядоченности строк в файле с помощью служебной программы `sort`.

Содержимое файлов `f1` и `f2` сортирует команда (см. листинг 6.15), применяющая в качестве ключа второй непробельный символ второго поля.

```
sort -k 2.2b,2.2b f1 f2
```

Листинг 6.15. Пример использования служебной программы `sort` с модификаторами в определении ключей.

Отсортировать базу данных пользователей по числовым значениям идентификаторов пользователей (третье поле, поля разделяются символом :) можно посредством команды, показанной в листинге 6.16.

```
sort -t ':' -k 3,3n /etc/passwd
```

Листинг 6.16. Пример использования служебной программы `sort` с опцией `-t`.

Выдать содержимое уже отсортированного файла `f.sorted`, выводя лишь первую из строк с одинаковыми третьими полями, поможет комбинация опций `-m` и `-u` (см. листинг 6.17).

```
sort -um -k 3.1,3 f.sorted
```

Листинг 6.17. Пример использования служебной программы `sort` с опциями `-m` и `-u`.

В ситуациях, когда есть две версии текста — «правильная» и «неправильная» (например, исходные тексты работающей и неработающей программ), полезно понять, чем они отличаются. Для этого предназначена служебная программа `diff`:

```
diff [-c | -e | -f | -C число] [-br] файл1 файл2
```

Утилита `diff` сравнивает содержимое исходных файлов и выдает на стандартный вывод список изменений, которые необходимо произвести, чтобы преобразовать `файл1` в `файл2`. Она поддерживает следующие опции.

`-b`

Игнорировать пробельные символы в конце строк; остальные цепочки пробельных символов считать равными.

`-c`

Производить вывод в формате, обеспечивающем три строки контекста.

`-C число`

Производить вывод в формате, обеспечивающем заданное число строк контекста.

-e

Производить вывод в формате, пригодном для подачи на вход редактора ed и преобразования файла1 в файл2.

-f

Производить вывод в альтернативном формате, напоминающем -e, но в обратном порядке и не предназначенном для подачи на вход редактора ed.

-r

Если файл1 и файл2 представляют собой каталоги, применять утилиту diff рекурсивно к одноименным и одностипным обычным файлам и каталогам. Если только один из файлов является каталогом, в нем сравнивается «тезка» другого исходного файла.

В качестве примера использования служебной программы diff сравним две версии бинарных утилит (см. листинг 6.18). Начальный фрагмент результата показан в листинге 6.19. Видно, что различия, по сути, сводятся к добавлению новых файлов.

```
diff -r binutils-2_14 binutils-2_14-branch
```

Листинг 6.18. Пример использования служебной программы diff.

```
diff -r binutils-2_14/bfd/version.h binutils-2_14-branch/bfd/version.h
1c1
< #define BFD_VERSION_DATE 20030612
---
> #define BFD_VERSION_DATE 20031007
Only in binutils-2_14-branch/binutils: ChangeLog
Only in binutils-2_14-branch/binutils: arlex.c
Only in binutils-2_14-branch/binutils: deflex.c
Only in binutils-2_14-branch/binutils: rclex.c
Only in binutils-2_14-branch/binutils: syslex.c
Only in binutils-2_14-branch: config.guess
Only in binutils-2_14-branch/gas: ChangeLog
Only in binutils-2_14-branch/gas/config: tc-ns32k.c
Only in binutils-2_14-branch/gas: configure
Only in binutils-2_14-branch/gas: configure.in
```

```
Only in binutils-2_14-branch/gas: itbl-lex.c
. . .
```

Листинг 6.19. Фрагмент возможного результата работы служебной программы `diff`.

Если нужно проверить два файла на совпадение, предпочтительнее воспользоваться не утилитой `diff`, а более простой и быстрой служебной программой `cmp`:

```
cmp [-l | -s] файл1 файл2
```

По умолчанию `cmp` выдает на стандартный вывод номера байта и строки, где встретилось первое отличие. Опция `-l` предписывает выдавать все отличия в виде номеров и содержимого (осьмеричного) байт. По опции `-s` отличия не выдаются, возвращается лишь соответствующий код завершения (0 — файлы совпадают, 1 — файлы различаются, больше единицы — зафиксирована ошибка).

Если в продолжение примера с утилитой `diff` сравнить две версии файла `version.h` посредством `cmp` (см. листинг 6.20), результат будет выглядеть так, как показано в листинге 6.21.

```
cmp -l binutils-2_14/bfd/version.h binutils-2_14-
branch/bfd/version.h
```

Листинг 6.20. Пример использования служебной программы `cmp`.

```
30    60    61
31    66    60
32    61    60
33    62    67
```

Листинг 6.21. Возможный результат работы служебной программы `cmp`.

Еще одно средство выявления различий (и, кроме того, совпадений) текстовых файлов — служебная программа `comm`:

```
comm [-123] файл1 файл2
```

Она читает файл1 и файл2 (их содержимое должно быть отсортировано в лексикографическом порядке) и выводит результаты в три столбца:

- 1) строки, входящие только в файл1;
- 2) строки, входящие только в файл2;
- 3) строки, входящие в оба файла.

Опции -1, -2 или -3 подавляют вывод соответствующих столбцов. Так, `comm -12` выводит строки, общие для обоих файлов, `comm -23` — имеющиеся только в файле1, `comm -123` не выводит ничего.

В стандарте POSIX-2001 приведен следующий пример использования служебной программы `comm`. Пусть файл `xcu` содержит отсортированный список утилит, вошедших в данный стандарт, файл `xpg3` — аналогичный список для спецификации X/Open Portability Guide, Issue 3, а файл `svid89` представляет собой отсортированный список утилит из System V Interface Definition Third Edition. Тогда конвейер, показанный в листинге 6.22, выдаст список служебных программ, фигурирующих в спецификациях XPG3 и SVID, но не вошедших в стандарт IEEE Std 1003.1-2001.

```
comm -12 xpg3 svid89 | comm -23 - xcu
```

Листинг 6.22. Пример использования служебной программы `comm`.

Для контроля целостности файлов (не являющегося, впрочем, криптостойким), предназначена служебная программа

```
cksum [файл ...]
```

Для каждого файла — аргумента она выдает на стандартный вывод подсчитанную по стандарту ISO/IEC 8802-3:1996 (Ethernet) контрольную сумму, количество байт и имя.

Применение утилиты `cksum` — полезный элемент оформления поставляемых приложений.

Регулярные выражения

Понятие регулярного выражения (РВ) — одно из важнейших для программ обработки текстовых файлов. Согласно стандарту POSIX-2001, регулярное выражение — это шаблон, служащий для выборки определенных цепочек символов из множества подобных цепочек. Говорят, что выбранные цепочки удовлетворяют РВ (успешно с ним сопоставляются).

Когда хотят подчеркнуть составной характер РВ, используют термин «полное регулярное выражение». Под этим понимается шаблон, заданный как конкатенация одного или нескольких элементарных РВ.

Стандарт накладывает на сложность (полных) РВ единственное ограничение: реализация должна поддерживать любое РВ, длина которого не превышает 256 байт.

Оговаривается, что при наличии альтернатив выбирается самая левая из возможных цепочек, имеющая к тому же максимальную длину.

Различают базовые (БРВ) и расширенные регулярные выражения (РРВ). В большинстве случаев используются БРВ, они и будут описаны в первую очередь.

Минимальным элементом БРВ являются односимвольные БРВ, т. е. БРВ, которым удовлетворяют цепочки из одного символа. Односимвольные БРВ строятся по следующим правилам.

- 1) Обычный символ (не входящий в перечень из следующего пункта) — это односимвольное БРВ, которое успешно сопоставляется с указанным символом.
- 2) Если за символом \ следует любой специальный символ, то последний теряет свой специальный смысл. Данная комбинация определяет односимвольное БРВ, успешно сопоставляющееся со специальным символом. Специальными символами являются:

. [\

Эти символы имеют специальный смысл, за исключением случая, когда они заключены в квадратные скобки.

*

Символ имеет специальный смысл, если он не заключен в квадратные скобки или не является первым символом БРВ.

^

Символ имеет специальный смысл в начале полного БРВ или в квадратных скобках сразу после открывающей скобки.

\$

Символ имеет специальный смысл в конце полного БРВ.

- 3) Точка (.) — односимвольное БРВ, успешно сопоставляющееся с любым символом.
- 4) Непустая цепочка символов, заключенная в квадратные скобки [] представляет собой односимвольное БРВ, которое успешно сопоставляется с любым символом из этой цепочки. Если цепочка начинается

с символа `^`, то БРВ успешно сопоставляется с любым не входящим в данный набор символом.

Внутри квадратных скобок можно использовать также конструкции, описываемые ниже.

Диапазон символов можно задать с помощью знака `-`. Например, БРВ `[0-9]` эквивалентно `[0123456789]`. Минус теряет свой специальный смысл, если он стоит в начале (допустимо после `^`) или в конце цепочки символов в квадратных скобках либо использован в качестве границы диапазона. Так, БРВ `[%-]` успешно сопоставляется со всеми символами от процента до минуса включительно.

Закрывающая квадратная скобка не рассматривается как окончание цепочки символов, если она стоит вначале (быть может, после `^`). Таким образом, БРВ `[ ]a-f]` определяет либо закрывающую квадратную скобку, либо любой символ от `a` до `f` включительно.

Стандарт POSIX-2001 предусматривает поддержку классов символов (таких, например, как буквы, цифры, знаки пунктуации, пробельные символы, управляющие символы и т.д.). Класс символов задается своим именем, заключенным в квадратные скобки с двоеточиями. Должны поддерживаться по крайней мере следующие классы:

<code>[:alnum:]</code>	<code>[:cntrl:]</code>	<code>[:lower:]</code>	<code>[:space:]</code>
<code>[:alpha:]</code>	<code>[:digit:]</code>	<code>[:print:]</code>	<code>[:upper:]</code>
<code>[:blank:]</code>	<code>[:graph:]</code>	<code>[:punct:]</code>	<code>[:xdigit:]</code>

Например, БРВ `[^[:alnum:]]` удовлетворяют символы, отличные от букв и цифр.

В некоторых языках элемент алфавитного сравнения может задаваться несколькими символами. Такие элементы следует заключать в квадратные скобки с точками: `[.ch.]`.

Если определены классы элементов, эквивалентных при алфавитном сравнении, то подобный класс можно задать, заключив любой его элемент в квадратные скобки со знаками равенства: `{[= и =]}`.

Таковы правила построения заключенных в квадратные скобки односимвольных БРВ.

Построение многосимвольных БРВ из односимвольных регламентируется следующим образом.

- 1) В состав БРВ могут входить подвыражения (в том числе вложенные), заключенные в экранированные круглые скобки: `\( ( и \)`. Подвыражение в скобках успешно сопоставляется с теми же цепочками, что и без скобок.
- 2) На подвыражения допускаются обратные ссылки вида `\n` (где `n` — цифра от 1 до 9). Обратной ссылке удовлетворяет такая же цепочка

символов, что была успешно сопоставлена подвыражением, открывающимся экранированной скобкой номер n , считая от начала полного БРВ. Например, БРВ $\backslash(.\backslash)^+1$ удовлетворяют пары одинаковых символов.

- 3) За односимвольными БРВ, подвыражениями и обратными ссылками может следовать звездочка. Такая конструкция успешно сопоставляется с любым (в частности, с нулевым) числом последовательных вхождений упомянутых БРВ. Например, шаблону $\backslash(.*)\backslash1$ удовлетворяют пары стоящих рядом одинаковых цепочек символов.
 - 4) За односимвольными БРВ, подвыражениями и обратными ссылками может следовать запись вида $\backslash\{m\}$, $\backslash\{m, \backslash\}$ или $\backslash\{m, n\}$, называемая интервальным выражением. Такая конструкция успешно сопоставляется с определенным числом вхождений упомянутых БРВ. Значения m и n должны удовлетворять неравенствам $0 \leq m < n \leq \text{RE\_DUP\_MAX}$, где RE_DUP_MAX – конфигурационная константа. Запись вида $\backslash\{m\}$ означает ровно m вхождений; $\backslash\{m, \backslash\}$ – по крайней мере m вхождений; $\backslash\{m, n\}$ – любое число вхождений от m до n включительно. Например, в цепочке «aaaabbbbaaaa» БРВ $a\backslash\{1, 3\}$ успешно сопоставится с первой тройкой букв a .
 - 5) Конкатенация компонентных БРВ есть БРВ, которое успешно сопоставляется с конкатенацией цепочек, удовлетворяющих каждому из компонентных БРВ.
 - 6) На БРВ можно наложить ограничение, чтобы успешно сопоставленная подцепочка символов примыкала к одной или обоим границам анализируемой цепочки (произвести фиксацию границ). Если в начале БРВ стоит символ $\wedge$ (фиксатор начала), то сопоставление должно выполняться с начала цепочки. Если в конце БРВ стоит символ $\$$ (фиксатор конца), то сопоставление должно выполняться до конца цепочки. Например, БРВ $\wedge\backslash(.*)\backslash1\$$ успешно сопоставляется с цепочками символов, состоящими из двух одинаковых подцепочек.
- Операции построения многосимвольных БРВ описаны выше в порядке убывания приоритетов. Наибольшим приоритетом обладают подвыражения и обратные ссылки, далее следуют повторители – звездочка и интервальные выражения, затем конкатенация и, наконец, фиксация границ.

Опишем отличия расширенных регулярных выражений (РРВ) от базовых.

- 1) В число специальных символов дополнительно входят круглые скобки, открывающая фигурная скобка, а также символы $+$, $?$, $|$.
- 2) Для группирования (выделения подвыражений) используются неэкранированные круглые скобки.
- 3) Отсутствуют обратные ссылки.

- 4) Добавлены два повторителя: плюс обозначает любое ненулевое число последовательных вхождений; знак вопроса — ноль или одно вхождение.
- 5) Для выделения интервальных выражений используются незэкранированные фигурные скобки.
- 6) Введена операция логического ИЛИ, которая обозначается символом `|` и имеет низший приоритет. Результат сопоставляется с цепочками, удовлетворяющими хотя бы одному из операндов. Например, `РРВ а ((bc) | d)` успешно сопоставляется и с цепочкой символов «abc», и с «ad». Односимвольные РРВ, разделенные символом `|` и заключенные в круглые скобки, трактуются как односимвольные.

Таковы правила построения и обработки регулярных выражений, зафиксированные в стандарте POSIX-2001. Отметим, что базовые регулярные выражения не являются подмножеством расширенных, хотя число специфических особенностей БРВ невелико.

Использование регулярных выражений

Мы приступаем к описанию наиболее употребительных служебных программ и функций, использующих механизм регулярных выражений. Безусловно, на первое место следует поставить утилиту `grep`:

```
grep [-E | -F] [-c | -l | -q] [-insvx] -e
    список_шаблонов ... [-f файл_шаблонов] ...
    [файл ...]
grep [-E | -F] [-c | -l | -q] [-insvx] [-e
    список_шаблонов ...] -f файл_шаблонов ...
    [файл ...]
grep [-E | -F] [-c | -l | -q] [-insvx]
    список_шаблонов [файл ...]
```

Она служит для выборки строк исходных файлов, удовлетворяющих хотя бы одному шаблону из заданного списка.

Три приведенные выше варианта вызова служебной программы `grep` отличаются способом задания списка шаблонов. В первом случае он является аргументом опции `-e`, во втором извлекается из файла шаблонов — аргумента опции `-f`, в третьем указывается как самостоятельный аргумент командной строки, но отдельные элементы этого списка всегда разделяются символами перевода строки. Опция `-e` полезна в ситуациях, когда нужно задать шаблон, начинающийся со знака минус.

По умолчанию шаблоны трактуются как базовые регулярные выражения. Опция `-E` предписывает переход к расширенным РВ, а опция `-F`

означает, что в качестве шаблонов выступают цепочки символов (и, следовательно, процесс сопоставления существенно упрощается).

Опции `-c`, `-l`, `-q` и `-n` влияют на выдачу результатов. По умолчанию на стандартный вывод выдаются строки исходных файлов, в которых присутствуют успешно сопоставленные цепочки символов. Опция `-c` предписывает выдавать только общее число подобных строк, `-l` — только имена файлов, где имеются успешно сопоставленные строки, `-q` — только код завершения (0 — есть успешно сопоставленные строки), `-n` требует ставить ее номер в исходном файле перед каждой выводимой строкой (нумерация строк начинается с 1).

Опции `-i`, `-v` и `-x` воздействуют на процесс сопоставления: `-i` предписывает не различать при сопоставлении большие и малые буквы, `-v` — выбирать строки, не удовлетворяющие ни одному из заданных шаблонов, `-x` — рассматривать только строки, все символы которых участвуют в успешном сопоставлении с одним из шаблонов.

Опция `-s` подавляет выдачу диагностических сообщений о том, что исходный файл не существует или не доступен на чтение.

Отметим, что опции `-F` и `-q`, каждая по-своему, ускоряют работу служебной программы `grep`: `-F` упрощает сопоставление, `-q` позволяет завершить действие после первого успешного сопоставления (и не обрабатывать оставшиеся строки и/или файлы). Сочетание опций `-q` и `-s` позволяет также более свободно задавать исходные файлы, не заботясь об их существовании и доступности.

Рассмотрим примеры использования утилиты `grep`. Для выборки пустых строк из файла стандартного ввода пригодны два шаблона:

```
grep ^$  
grep -v .
```

Если нужно выбрать строки, имеющие вид `abc` или `def`, можно воспользоваться одной из трех команд:

```
grep -E '^abc$|^def$'  
grep -F -x 'abc  
def'
```

И наконец, пусть в файлах с исходными текстами Фортран-программ требуется найти все строки, содержащие вызовы подпрограмм и не являющиеся комментариями. Для такой выборки «в первом приближении» (учитывая нерегулярный синтаксис Фортрана) подойдет следующая команда:

```
grep -i '^[^C].* CALL '*,for
```


В командных файлах для обработки текстов часто используется потоковый редактор `sed`:

```
sed [-n] сценарий [файл ...]  
sed [-n] [-e сценарий] ... [-f файл_сценария] ...  
[файл ...]
```

Редактор `sed` читает указанные текстовые файлы (по умолчанию — стандартный ввод), выполняет редактирование в соответствии с командами сценария и записывает результат на стандартный вывод. Смысл опций `-e` и `-f` аналогичен утилите `grep`. Опция `-n` подавляет подразумеваемый вывод и предписывает выдавать только явно отобранные строки.

Сценарий для `sed` состоит из редактирующих команд (каждая на отдельной строке), имеющих следующий формат:

```
[адрес [, адрес]] функция [аргумент ...]
```

Функция имеет здесь однобуквенное обозначение.

В нормальном режиме `sed` циклически выполняет следующие действия:

- 1) Добавляет входную строку в буфер. Обычно буфер пуст, если только предыдущий цикл не завершился командой `D`.
- 2) Применяет к буферу последовательно все команды сценария, адреса в которых позволяют их применить.
- 3) Если не указана опция `-n`, копирует буфер на стандартный вывод, добавив в конце перевод строки. Очищает буфер.

Некоторые команды используют хранилище, чтобы запомнить весь буфер или его часть для последующего применения.

Адрес в редактирующей команде `sed` — это либо десятичное число, означающее номер входной строки в совокупности входных файлов, либо символ `$`, который обозначает последнюю входную строку, либо контекстный адрес, имеющий вид `/базовое_регулярное_выражение/`. Контекстный адрес задает первую (начиная с текущей) из строк, успешно сопоставленных с БРВ при движении вперед.

Командная строка без адреса применима к любому буферу, командная строка с одним адресом — к буферу с соответствующим адресом, командная строка с двумя адресами — к буферам с адресами в диапазоне от первого до второго включительно; затем процесс повторяется, начиная с первой строки вслед за выбранным диапазоном.

Перечислим команды редактора `sed`. В скобках указывается максимальное число допустимых адресов для каждой функции.

Аргумент текст состоит из одной или более строк. Все строки, кроме последней, заканчиваются на \, чтобы экранировать символ перевода строки.

```
(2) { функция  
    функция  
    ...  
}
```

Выполнить заданную последовательность функций.

```
(1) a\  
текст
```

Добавить. Вывести текст перед чтением следующей входной строки.

```
(2) b [метка]
```

Перейти к команде :, содержащей метку. Если метка пуста, перейти на конец сценария.

```
(2) c\ текст
```

Заменить. Удалить содержимое буфера. При 0 или 1 адресе или в конце двухадресного диапазона вывести текст. Начать новый цикл.

```
(2) d
```

Удалить содержимое буфера. Начать новый цикл.

```
(2) D
```

Удалить начало буфера до первого перевода строки. Начать новый цикл.

```
(2) g
```

Заменить содержимое буфера содержимым хранилища.

```
(2) G
```

Добавить к содержимому буфера содержимое хранилища.

(2) h

Заменить содержимое хранилища содержимым буфера.

(2) n

Добавить к содержимому хранилища содержимое буфера.

(1) i \

текст

Вставить. Вывести текст.

(2) l

Вывести буфер, заменяя непечатные символы на пары символов ASCII и разбивая длинные строки.

(2) n

Скопировать буфер на стандартный вывод, если подразумеваемый вывод не подавлен. Заменить содержимое буфера на следующую входную строку. Если таковой не оказалось, завершить выполнение сценария.

(2) N

Добавить к буферу следующую входную строку, вставив перед ней символ перевода строки. (Текущий номер строки изменяется.) Если входных строк больше нет, завершить выполнение сценария.

(2) p

Скопировать буфер на стандартный вывод.

(2) P

Скопировать начальный сегмент буфера (до первого перевода строки) на стандартный вывод.

(1) q

Выйти. Перейти на конец сценария. Нового цикла не начинать.

(2) r ч_файл

Прочитать содержимое ч_файла. Поместить его на стандартный вывод перед чтением следующей входной строки.

(2) s/БРВ/замена/флаги

Подставить замену вместо фрагментов буфера, отождествленных с БРВ. Флаги могут быть опущены или иметь следующие значения:

число n — заменить n-е вхождение БРВ;

g — заменить все вхождения БРВ, а не только первое;

p — если замена произошла, вывести содержимое буфера;

w з_файл — если замена произошла, добавить содержимое буфера к з_файлу.

Вместо символа & в замене подставляется цепочка, отождествленная с БРВ.

(2) t [метка]

Проверить. Перейти к команде :, содержащей метку, если со времени последнего чтения входной строки или последнего выполнения команды t в буфере производились подстановки. Если метка пуста, перейти на конец сценария.

(2) w з_файл

Записать. Добавить содержимое буфера к з_файлу.

(2) x

Обменять содержимое буфера и хранилища.

(2) y/цепочка1/цепочка2/

Заменить все символы буфера, содержащиеся в цепочке1, на соответствующие символы цепочки2. Длины цепочек должны совпадать.

(2) ! функция

Отрицание. Применить функцию (или группу, если функция начинается с {) только к строкам, не соответствующим адресам.

(0) : метка

Не делает ничего. Содержит лишь метку, на которую может быть осуществлен переход командами `t` или `b`.

(1) =

Вывести в качестве отдельной строки номер текущей строки.

(0)

Пустая команда.

(0) #

Управляющий комментарий. Если сценарий начинается с символов `#n`, подразумеваемый вывод подавляется (что эквивалентно опции `-n` в командной строке). В остальных случаях игнорировать `#` и остаток строки.

Последовательность символов `\n` успешно сопоставляется с переводом строки. Явный символ перевода строки не должен использоваться в БРВ контекстных адресов и функции замены.

Приведем примеры использования потокового редактора `sed`. В процессе загрузки ОС Linux выполняются командные строки, аналогичные показанным в листинге 6.23.

```
map=`basename $map | sed -e s/^auto_home/auto.home/  
-e s/^auto_mnt/auto.mnt/  
cat /etc/auto.master | grep -v '^[^+]' | sed  
-e '/^#/d' -e '/^$/d'
```

Листинг 6.23. Пример использования редактора `sed`.

Первая из них заменяет подчеркивание на точку в именах файлов, обслуживающих автомонтирование файловых систем, вторая отсеивает строки файла `auto.master`, начинающиеся с символа `+` (это делает `grep -v`), комментарии (строки, начинающиеся символом `#`) и пустые строки.

Следующий вызов `sed` (см. листинг 6.24) сжимает несколько идущих подряд пустых строк в одну.

```
# Текущая строка - пустая.  
# Добавляем следующие строки к буферу, пока он  
# остается пустым.  
# Тем самым игнорируются "лишние" пустые строки.  
:Empty  
    n  
    /^$/ b Empty  
# Добавленная строка оказалась непустой. Выведем ее.  
    p  
}  
,
```

Листинг 6.24. Сжатие пустых строк средствами редактора sed.

Любопытно сопоставить приведенный нами сценарий с примером, включенным в текст стандарта POSIX-2001 (см. листинг 6.25). Наш вариант явно проще и короче.

```
sed -n '  
# Выведем непустые строки  
/./ {  
    p  
    d  
}  
# Выведем одну пустую строку, затем проанализируем  
# следующие.  
/^$/ p  
# Прочитаем следующую строку, отбросим оставшийся  
# перевод строки (пустую строку) и вернемся к  
# проверке пустой строки.  
:Empty  
/^$/ {  
    N  
    s /././.  
    b Empty  
}  
# Выведем непустую строку, затем вернемся к поиску  
# первой пустой.  
p  
,
```

Листинг 6.25. «Стандартный» вариант сжатия пустых строк средствами редактора sed.

Еще одно популярное средство обработки текстовых файлов — служебная программа `awk`:

```
awk [-F PPB] [-v присваивание] ... программа  
    {аргумент ...}  
awk [-F PPB] -f программный_файл ...  
    [-v присваивание] ... {аргумент ...}
```

Утилита `awk` выполняет программы, написанные на одноименном языке программирования, специально предназначенном для обработки текстов. Программа на языке `awk` представляет собой последовательность шаблонов и соответствующих действий, которые выполняются при чтении входных данных, успешно сопоставляющихся с шаблоном. Оператор шаблон-действие имеет вид

```
шаблон { действие }
```

Ввод для `awk` делится на записи, разделяемые специальным символом. По умолчанию это перевод строки; в таком случае `awk` обрабатывает ввод построчно. Разделитель записей можно изменить, переопределив переменную `RS`. Каждая запись делится на поля, ограниченные разделителями полей (по умолчанию — пробелами или табуляциями). Любой из них можно изменить, переопределив переменную `FS` или указав опцию `-F` с аргументом — расширенным регулярным выражением (PPB). Поля исходных строк доступны по именам `$1`, `$2`, ... ; `$0` — вся входная строка.

Каждая исходная строка сопоставляется с каждым из шаблонов; в случае успеха выполняются указанные действия. После сопоставления со всеми шаблонами вводится следующая строка и процесс сопоставления повторяется. Может быть опущен либо шаблон, либо действие, но не оба вместе. Если для данного шаблона не указаны действия, то строка просто копируется на стандартный вывод. Если для действия не определен шаблон, то оно будет выполняться для каждой входной строки. Строки, которые не удалось сопоставить ни одному шаблону, игнорируются.

Действие есть последовательность операторов. Поскольку шаблоны и действия могут быть опущены, то, чтобы различать их в программе, последние надо брать в фигурные скобки. Оператор — это одна из конструкций:

- 1) `if ( условие ) оператор [ else оператор ]`
- 2) `while ( условие ) оператор`
- 3) `for ( выражение; условие; выражение ) оператор`
- 4) `break`
- 5) `continue`

- 6) { [оператор] ... }
- 7) переменная = выражение # оператор присваивания
- 8) print [список_выражений] [> выражение]
- 9) printf формат [, список_выражений] [> выражение]
- 10) next # пропустить оставшиеся шаблоны и перейти к следующей строке
- 11) exit # пропустить оставшиеся строки

Операторы завершаются точкой с запятой, переводом строки или правой скобкой. Пустой список_выражений означает всю строку. Выражения строятся из цепочек символов и чисел с помощью операций +, -, *, /, %, ^ (возведение в степень) и конкатенации (обозначается пробелом). В них также можно использовать операции из языка C: ++, --, +++, --, *=, /=, %=, ^=, ? : (условное выражение). Переменные инициализируются пустыми цепочками, могут быть скалярами, элементами массива (обозначается `x[i]`) или полями. Индексами массива служат любые (не обязательно числовые) цепочки символов, что позволяет реализовать разновидность ассоциативной памяти. Цепочки символов заключаются в двойные кавычки ("").

Оператор `print` выдает свои аргументы на стандартный вывод (или в файл, если присутствует часть `>выражение`), разделяя их текущим разделителем полей и завершая каждую запись выходным разделителем записей. Оператор `printf` делает то же, но под управлением формата.

Язык `awk` содержит большое число встроенных функций. Кратко опишем их.

Математические функции `atan2 (y, x)`, `cos (x)`, `sin (x)`, `exp (x)`, `log (x)`, `sqrt (x)` не нуждаются в пояснениях. Функция `int (x)` отбрасывает дробную часть своего аргумента, `rand ()` возвращает псевдослучайное число в диапазоне от 0 до 1, `srand ([выражение])` устанавливает затравочное значение для `rand` (если выражение опущено, используется текущее время).

В число функций, оперирующих цепочками символов, входят `gsub (PPB, замена[, цепочка])` и `sub (PPB, замена[, цепочка])` – соответственно, глобальная и однократная замена вхождений PPB в \$0 или цепочку, по аналогии с командой редактора `sed` и ее флагом `g`; `index (цепочка, подцепочка)` – поиск подцепочки в цепочке; `length [[цепочка]]` – вычисление длины цепочки-аргумента или \$; `match (цепочка, PPB)` – поиск вхождения PPB в цепочку с установкой значений переменных `RSTART` и `RLENGTH` (см. далее); `split (цепочка, массив[, PPB-разделитель])` – расщепление цепочки по полям в элементы массива; `sprintf (формат, выражение, выражение, ...)` – формирование цепочки символов средствами форматного вывода; `substr (цепочка, m[, n])` – выделение n-символьной подцепочки, начинающейся с позиции m; `tolower (цепочка)`

– приведение к строчным буквам; `tolower` (цепочка) – приведение к прописным буквам.

В языке `awk` имеются также группа функций ввода/вывода и функции общего назначения. Функция `close` (выражение) закрывает файл или канал, поименованный заданным выражением, `getline` [переменная] обеспечивает чтение записи из текущего входного файла (возможно использование конвейера вида выражение | `getline` [переменная] и перенаправление ввода `getline` [переменная] < выражение), `system` (выражение) — выполнение команды, заданной выражением.)

Язык `awk` допускает определение пользовательских функций, для чего служит конструкция

```
function имя_функции ([аргумент, ...]) { операторы }
```

Шаблон в языке `awk` — это произвольная логическая комбинация, составленная с помощью операций `!`, `||`, `&&` и скобок из расширенных регулярных выражений и выражений сравнения. РРВ обрамляются символами `/`. Отдельное РРВ в шаблоне сопоставляется со всей строкой. РРВ допускаются и в выражениях сравнения. Шаблон может состоять из двух шаблонов, разделенных запятой; указанные действия выполняются для всех строк между строкой, удовлетворяющей первому шаблону, и строкой, удовлетворяющей второму.

Выражение сравнения — одна из следующих конструкций:

```
выражение опер_сопост РРВ  
выражение опер_сравн выражение
```

Здесь `опер_сравн` — любая из шести операций сравнения языка `C`, `опер_сопост` — это `~` (успешно сопоставляется) или `!~` (не сопоставляется).

Условие — арифметическое выражение, выражение сравнения или их логическая комбинация.

Для выполнения каких-либо действий перед чтением первой или после чтения последней исходной строки определены специальные шаблоны `BEGIN` и `END`. Шаблон `BEGIN` следует указывать первым, `END` — последним.

Присваивания, заданные в командной строке с помощью опции `-v`, выполняются до начала интерпретации `awk`-программы (в частности, до действий, ассоциированных с шаблоном `BEGIN`). Например, для использования символа `c` в качестве разделителя полей можно указать в командной строке `-v 'FS = c'`.

В командной строке можно указать также аргументы двух типов — имена файлов с исходными данными и присваивания. Последние выполняются непосредственно перед чтением указанного следующим исходного файла. В частности, присваивания, заданные перед первым аргументом-файлом выполняются после действий, ассоциированных с шаблоном BEGIN, а те, что расположены в конце командной строки, — перед действиями, ассоциированными с шаблоном END. Если в командной строке нет аргументов-файлов, присваивания выполняются перед обработкой стандартного ввода.

Перечислим специальные переменные `awk`.

ARGC

Число элементов в массиве ARGV.

ARGV

Массив аргументов командной строки `awk`, исключая опции и программы.

CONVFMT

Формат для преобразования чисел в цепочки символов (кроме операторов вывода, где используется переменная OFMT, см. далее). По умолчанию — `% . 6g`.

ENVIRON

Массив, представляющий окружение. Индексами служат цепочки символов, совпадающих с именами переменных окружения.

FILENAME

Имя файла, из которого в данный момент производится ввод.

FNR

Порядковый номер текущей записи в текущем исходном файле.

FS

RPB-разделитель полей во входных данных, по умолчанию — пробел.

NF

Количество полей в текущей записи.

NR

Порядковый номер текущей записи, считая от начала обработки исходных данных.

OFMT

Формат вывода чисел, по умолчанию % . 6g.

OFS

Разделитель полей при выводе, по умолчанию — пробел.

ORS

Разделитель записей при выводе, по умолчанию — перевод строки.

RLENGTH

Длина успешно сопоставленной функцией `match()` цепочки символов.

RS

Первым символом цепочки, представляющей собой значение переменной `RS`, является разделитель исходных записей (по умолчанию — перевод строки). Если значение `RS` пусто, между записями может располагаться несколько пустых строк.

RSTART

Начальная позиция успешно сопоставленной функцией `match()` цепочки символов (считая от 1).

SUBSEP

Цепочка символов — разделитель индексов многомерных массивов; подразумеваемое значение зависит от реализации.

Приведем примеры использования утилиты `awk`. Сложить числа, стоящие в первом столбце исходного файла, вывести сумму и среднее арифметическое позволяет `awk`-программа, показанная в листинге 6.26.

```
{ s += $1 } END { print "Сумма:", s,  
    " Среднее арифметическое:", s/NR }
```

Листинг 6.26. Пример `awk`-программы, оперирующей с числами.

Командная строка из листинга 6.27, служит для вывода тех строк файла `f1.txt`, у которых первое поле не совпадает с первым полем предыдущей строки.

```
awk '$1 != prev { print; prev = $1 }' f1.txt
```

Листинг 6.27. Пример `awk`-программы, заданной в командной строке.

Чтобы распечатать файл `f2.txt`, вставляя после слова «Page» номера страниц (начиная с первой), можно воспользоваться `awk`-программой (предполагается, что она помещена в файл `prog.awk`) и командной строкой, представленными, соответственно, в листингах 6.28 и 6.29.

```
/Page/ { $2 = n++ }      { print }
```

Листинг 6.28. Пример `awk`-программы, использующей шаблоны.

```
awk -f prog.awk -v 'n=1' f2.txt
```

Листинг 6.29. Пример вызова `awk`-программы, использующей шаблоны.

Программа, показанная в листинге 6.30, выводит поля входных записей, по одному на строке.

```
{ for (i = NF; i > 0; --i) print $i }
```

Листинг 6.30. Пример `awk`-программы, использующей оператор цикла.

Промоделировать работу утилиты `echo` можно с помощью `awk`-программы (см. листинг 6.31).

```
BEGIN {
    for (i = 1; i < ARGV; ++i)
        printf ("%s%s", ARGV [i], i == ARGV - 1 ? "\n"
            : " ")
}
```

Листинг 6.31. Пример awk-программы, использующей оператор цикла и специальные переменные awk.

Следующая awk-программа (см. листинг 6.32) позволяет разложить список поиска, хранящийся в переменной окружения PATH, по элементам массива.

```
BEGIN {
    n = split (ENVIRON ["PATH"], path, ":")
    for (i = 1; i <= n; ++i)
        print path [i]
}
```

Листинг 6.32. Пример awk-программы, использующей встроенную функцию split().

В листинге 6.33 приведен фрагмент командного файла, выполняемого при выключении системы. Здесь можно обратить внимание на разные виды экранирования. (Третье поле в выдаче mount — это точка монтирования.)

```
# Перемонтируем на чтение все, что еще остается
# смонтированным.
mount | awk '/( \| |^\/dev\/root)/ { print $3 }'
| while read line; do
    mount -n -o ro,remount $line
done
```

Листинг 6.33. Пример использования утилиты awk в системном командном файле.

Отметим, что в POSIX-2001 стандартизована весьма развитая версия awk, входной язык которой приближен к языку C.

На уровне функций работа с регулярными выражениями поддерживается семейством regex (см. листинг 6.34).

```
#include <regex.h>
int regcomp (regex_t *restrict preg,
    const char *restrict pattern, int cflags);
int regexec (const regex_t *restrict preg,
    const char *restrict string, size_t nmatch,
    regmatch_t pmatch [restrict], int eflags);
void regfree (regex_t *preg);
size_t regerror (int errcode,
    const regex_t *restrict preg, char *restrict errbuf,
    size_t errbuf_size);
```

Листинг 6.34. Описание функций семейства `regex()`.

Первый член этого семейства, функция `regcomp()`, компилирует регулярное выражение, заданное аргументом `pattern`, и помещает результат компиляции в структуру типа `regex_t`, на которую указывает аргумент `preg`. Эта структура, описанная в заголовочном файле `<regex.h>`, должна содержать про крайней мере поле

```
size_t re_nsub; /* Число заключенных */
                /* в скобки подвыражений */
```

Третий аргумент функции `regcomp()`, `cflags`, задается как побитное ИЛИ следующих флагов:

`REG_EXTENDED`

Использовать расширенные регулярные выражения (подразумеваемый тип регулярных выражений — базовые).

`REG_ICASE`

При сопоставлении не различать большие и малые буквы.

`REG_NOSUB`

В `regexec()` сообщать только об успехе/неудаче сопоставления (и не устанавливать значения поля `re_nsub` структуры `regex_t`).

`REG_NEWLINE`

Изменить трактовку переводов строк (мы не будем на этом останавливаться).

Функция `regexec()` сопоставляет цепочку символов `string` со скомпилированным шаблоном, заданным аргументом `preg`. При успешном выполнении результат равен нулю; в противном случае возвращается ненулевое значение, свидетельствующее о неудаче сопоставления или ошибке. Аргумент `eflags` – побитное ИЛИ флагов `REG_NOTBOL` и `REG_NOTEOL` – определяет, являются ли границы цепочки границами строки, что важно для обработки фиксаторов `^` и `$`. Если значение аргумента `nmatch` равно нулю или при вызове `regcomp()` был задан флаг `REG_NOSUB`, аргумент `pmatch` функции `regexec()` игнорируется. В противном случае он должен указывать на массив не менее чем из `nmatch` элементов, который будет заполнен смещениями подцепочек, сопоставленных с заключенными в скобки подвыражениями шаблона (`pmatch[0]` соответствует всему регулярному выражению, в неиспользуемые элементы помещается 1).

Структурный тип `regmatch_t` должен включать по крайней мере следующие поля:

```
regoff_t rm_so; /* Смещение в байтах начала */
                /* подцепочки от начала цепочки */
regoff_t rm_eo; /* Смещение в байтах первого */
                /* символа за концом подцепочки от начала цепочки */
```

Тип `regoff_t` определяется как целое со знаком, способное вместить любое значение типов `off_t` и `ssize_t`.

Функция `regfree()` освобождает память, запрошенную вызовом `regcomp()` с тем же значением аргумента `preg`, которое после этого нельзя использовать как указатель на скомпилированное регулярное выражение.

В файле `<regex.h>` определены константы, возвращаемые функциями семейства `regex()` в случае ошибки. Например, значение `REG_NOMATCH` возвращается функцией `regexec()` при неудаче сопоставления, `REG_BADPAT` обозначает некорректное регулярное выражение, `REG_ESPACE` – нехватку памяти и т.д. Функция `regerror()` отображает эти константы в неспецифицируемые стандартом цепочки печатных символов и помещает их в буфер `errbuf`. Приложение, вызывая `regerror()`, должно перелать в качестве аргумента `errcode` последнее ненулевое значение, возвращенное функциями `regcomp()` или `regexec()` с заданным значением аргумента `preg`.

Приведем пример использования функций семейства `regex()` (см. листинг 6.35). Обратим внимание на задание флага `REG_NOTBOL` при повторных обращениях к `regexec()`.

```
#include <stdio.h>
#include <limits.h>
#include <regex.h>

/* Программа ищет все вхождения заданного шаблона */
/* во всех входных строках */
/* и выводит успешно сопоставленные подцепочки */

#define PATTERN "[A-Za-z][A-Za-z0-9]{0,31}"

int main (void) {
    char line [LINE_MAX]; /* Буфер для входных строк */
    char *pline; /* Указатель на начало */
                    /* сопоставляемой части строки */
    regex_t cere; /* Скомпилированное расширенное */
                    /* регулярное выражение */
    regmatch_t pm; /* Структура для запоминания */
                    /* границ сопоставленной подцепочки */
    int reerrcode; /* Код ошибки от regcomp или */
                    /* regexec */
    char reerrbuf [LINE_MAX]; /* Буфер для строк */
                    /* с сообщениями об ошибках */

    int i;
    if ((reerrcode = regcomp (&cere, PATTERN,
        REG_EXTENDED)) != 0) {
        (void) regerror (reerrcode, &cere, reerrbuf,
            sizeof (reerrbuf));
        fputs (reerrbuf, stderr);
        fputc ('\n', stderr);
        regfree (&cere);
        return (reerrcode);
    }
    fputs ("Вводите строки, сопоставляемые"
        "с шаблоном " PATTERN "\n", stdout);
    while (fgets (line, sizeof (line), stdin) !=
        NULL) {
        /* Произведем первое сопоставление с */
        /* прочитанной строкой. */
        /* Оно отличается от остальных при наличии */
        /* в шаблоне фиксатора начала */
        reerrcode = regexec (&cere, pline = line, 1,
            &pm, 0);
```



```
while (reerrcode == 0) {
    /* Повторяем, пока сопоставления с остатком */
    /* строки успешны */
    fputs ("Сопоставленная подцепочка: ", stdout);
    for (pline += pm.rm_so, i = pm.rm_eo -
        pm.rm_so; i-- > 0; ) {
        fputc (*pline++, stdout);
    }
    fputc ('\n', stdout);
    reerrcode = regexexec (&cere, pline, 1, &pm,
        REG_NOTBOL);
}
}
regfree (&cere);
return (ferror (stdin) || ferror (stdout));
}
```

Листинг 6.35. Пример использования функций семейства `regex()`.

Читателю рекомендуется поэкспериментировать с шаблоном PATTERN, пробуя различные варианты, в том числе некорректные.

Идейно простым, но весьма мощным и полезным средством обработки текстовых файлов является служебная программа преобразования символов `tr`:

```
tr [-c | -C] [-s] цепочка1 цепочка2
tr -s [-c | -C] цепочка1
tr -d [-c | -C] цепочка1
tr -ds [-c | -C] цепочка1 цепочка2
```

Она не применяет регулярных выражений как таковых, но в цепочках – аргументах можно употреблять конструкции, аналогичные некоторым односимвольным РВ.

Утилита `tr` копирует стандартный ввод на стандартный вывод с заменой либо удалением выбранных символов. При отсутствии опций введенные символы, найденные в цепочке1, заменяются на соответствующие (стоящие на тех же относительных позициях) символы из цепочки2. Опции `-c` и `-C` предписывают использовать вместо цепочки1 ее дополнение до множества всех символов; в первом случае дополнение упорядочивается в соответствии с кодировкой, во втором – по алфавиту. По опции `-d` будут удалены все входные символы, заданные цепочкой1. Опция `-s` задает сжатие (до одного) последовательностей одинаковых символов,

специфицированных последней из цепочек, указанных в командной строке (сжатие производится после каждой замены и/или удаления).

Чтобы задавать в цепочках — аргументах утилиты `tr` группы символов, можно воспользоваться следующими конструкциями.

`c1 c2`

Обозначает цепочку символов, лежащих в диапазоне от `c1` до `c2` включительно.

`[:класс_символов:]`

Обозначает цепочку символов, принадлежащих указанному классу (см. выше описание базовых регулярных выражений).

`[-класс_эквивалентности=]`

Обозначает цепочку символов, принадлежащих указанному классу эквивалентности при алфавитном сравнении.

`[c*n]`

Обозначает символ `c`, повторенный `n` раз. Может использоваться только в цепочке2. Если первая цифра в `n` есть 0, `n` рассматривается как восьмеричное число; иначе — как десятичное. Нулевое или отсутствующее `n` воспринимается как «очень много»; эта возможность полезна при дополнении цепочки2 до длины цепочки1.

Обратный слэш можно использовать для задания управляющих символов (`'\\'`, `'\a'`, `'\b'`, `'\f'`, `'\n'`, `'\r'`, `'\t'`, `'\v'`). Кроме того, `\` обозначает код символа, если за ним идут одна, две или три восьмеричные цифры.

Следующая команда (см. листинг 6.36) помещает список всех слов из файла `f1`, по одному на строку, в файл `f2` (под словом понимается максимальная последовательность букв).

```
tr -cs '[:alpha:]' '[\n*]' < f1 > f2
```

Листинг 6.36. Пример использования служебной программы `tr`.

Команда, показанная в листинге 6.37, переводит большие буквы в малые, попутно сжимая последовательности одинаковых (без учета регистра) букв.

```
tr -s '[:upper:]' '[:lower:]'
```

Листинг 6.37. Пример трансляции и сжатия последовательностей символов с использованием служебной программы `tr`.

Служебная программа `uniq`

```
uniq [-c | -d | -u] [-f число] [-s число]  
    [входной_файл [выходной_файл]]
```

позволяет сократить до одной подряд идущие одинаковые строки (сделать одинаковые строки файла смежными можно с помощью утилиты `sort`). Опции предоставляют дополнительный сервис.

`-c`

Перед каждой выходной строкой помещать ее кратность во входном файле.

`-d`

Подавить вывод неповторяющихся строк.

`-f число`

При сравнении строк игнорировать заданное число начальных полей. Поле определяется как максимальная цепочка символов, успешно сопоставляемая с базовым регулярным выражением `[[:blank:]]*[^[:blank:]]*`.

`-s число`

При сравнении строк игнорировать заданное число начальных символов. При совместном использовании опций `-f` и `-s` игнорируется указанное число символов, идущих после заданного числа полей.

`-u`

Подавить вывод строк, повторявшихся во входном файле.

В качестве примера употребления утилиты `uniq` приведем конвейер, позволяющий найти десять самых употребительных заголовочных файлов среди включаемых в стандартные заголовочные файлы, расположенные в

каталоге /usr/include и его подкаталогах (см. листинг 6.38). Результат работы конвейера может выглядеть так, как показано в листинге 6.39.

```
find /usr/include -name '*.h' -exec cat {} \; | tr
-d '[:blank:]' | \
grep -E -e '^#include(<.*>|\".*\")' | sort | uniq
-dc | sort -r | head
```

Листинг 6.38. Пример использования служебной программы `uniq`.

```
977 #include "nsISupports.h"
315 #include <glib.h>
201 #include <gdk/gdk.h>
167 #include <glibmm.h>
160 #include <features.h>
154 #include <glib-object.h>
144 #include "nsCOMPtr.h"
139 #include <sys/types.h>
139 #include <glibmm/class.h>
135 #include "nscore.h"
```

Листинг 6.39. Возможный результат работы конвейера, показанного в листинге 6.38.

Аналоги реляционных операций для текстовых файлов

Служебная программа `cut`

```
cut -b список [-n] [файл ...]
cut -c список [файл ...]
cut -f список [-d разделитель] [-s] [файл ...]
```

используется для выборки байт (опция `-b`), символов (`-c`) либо ограниченных разделителями полей (опция `-f`) с заданными списком номерами из строк исходных файлов, их конкатенации и выдачи на стандартный вывод. Если применить терминологию реляционных баз данных, `cut` выполняет операцию проекции отношения.

Список, являющийся аргументом опций `-b`, `-c` и `-f`, задается как последовательность разделенных запятыми или пробелами положительных чисел или диапазонов (с естественными умолчаниями, когда границы опущены). Опция `-d` определяет разделитель полей (по умолчанию

это символ табуляции), `-n` предписывает не разрывать (многобайтные) символы, опция `-s` в сочетании с `-f` подавляет вывод строк, в которых не оказалось символов-разделителей (по умолчанию такие строки копируются на стандартный вывод без изменений).

Пример. Чтобы выделить из базы данных пользователей входные имена и идентификаторы, можно воспользоваться командой, показанной в листинге 6.40. Начальный фрагмент возможного результата приведен в листинге 6.41.

```
cut -d : -f 1,3 /etc/passwd
```

Листинг 6.40. Пример использования служебной программы `cut`.

```
root:0
bin:1
daemon:2
adm:3
lp:4
sync:5
. . .
```

Листинг 6.41. Начальный фрагмент возможного результата работы служебной программы `cut`.

Служебная программа `paste`

```
paste [-s] [-d список] файл ...
```

в терминологии реляционных баз данных осуществляет горизонтальное соединение, конкатенируя соответственные строки исходных файлов и помещая результат на стандартный вывод. При наличии опции `-s` конкатенируются строки каждого из исходных файлов. Во всех случаях строки склеиваются посредством символа табуляции (по умолчанию) или символов из списка – аргумента опции `-d`. Список этих символов рассматривается как кольцевой, т. е. будучи исчерпан, он используется повторно. Сочетание `\0` в нем трактуется как пустая цепочка, а не как нулевой символ.

В качестве имени исходного файла может быть задан минус, что означает стандартный ввод. Если минус употреблен многократно, то стандартный ввод читается построчно, циклически по именам – минусам. Любопытно отметить, что стандарт POSIX-2001 предписывает реализациям поддерживать обработку не менее двенадцати исходных файлов.

Рассмотрим несколько примеров. Для выдачи в четыре столбца имен файлов текущего каталога можно воспользоваться конвейером, показанным в листинге 6.42.

```
ls | paste - - - -
```

Листинг 6.42. Пример использования служебной программы `paste`.

Для попарного слияния последовательных строк файла целесообразно «сыграть» на том, что список разделителей — кольцевой (см. листинг 6.43).

```
paste -s -d "\0\n" f.txt
```

Листинг 6.43. Пример использования служебной программы `paste` с кольцевым списком разделителей.

Если файл не подпадает под определение текстового из-за чрезмерно длинных строк, утилитой `cut` можно выделить начальные байты, поместив «хвосты» строк в другой файл. В дальнейшем из двух полученных файлов с помощью служебной программы `paste` можно воссоздать исходный (см. листинг 6.44).

```
cut -b 1-80 -n f > f1.txt
cut -b 81- -n f > f2
. . .
paste -d '\0' f1.txt f2 > f3
```

Листинг 6.44. Пример использования служебных программ `cut` и `paste`.

Еще один оператор реляционной СУБД — `join` — выдает на стандартный вывод результат соединения двух отношений, задаваемых строками файла1 и файла2:

```
join [-а номер_файла | -v номер_файла] [-е
      цепочка] [-о список] [-t символ]
      [-1 номер_поля] [-2 номер_поля] файл1 файл2
```

Отношения файл1 и файл2 должны быть отсортированы в возрастающем лексикографическом порядке по полям соединения (обычно по первому полю в каждой строке).

Результат операции составляют строки, по одной для каждой пары из отношений файл1 и файл2, имеющих одинаковые поля соединения. Обычно строка результата состоит из общего поля, затем остатка строки отношения файл1, затем остатка строки отношения файл2.

Как правило, поля разделяются символами пробела, табуляции или перевода строки. В этом случае несколько последовательных разделителей интерпретируются как один и начальные разделители отбрасываются. Подразумеваемым разделителем при выводе является пробел.

Допустимы следующие опции.

-a номер_файла

В дополнение к обычному выводу выдать строку для каждой непарной строки из файла с указанным номером (1 или 2).

-c цепочка

Заменить пустые поля вывода из списка опции -o заданной цепочкой символов.

-o список

Составлять выходные строки из полей, заданных в списке. Элемент списка имеет вид номер_файла.номер_поля или 0 (обозначает поле соединения). Поля, отсутствующие в исходном файле, трактуются при выводе как пустые.

-t символ

Использовать символ в качестве разделителя. Каждое вхождение символа в строку значимо. Указанный символ используется как разделитель и при вводе, и при выводе.

-v номер_файла

Вместо подразумеваемого вывода выдавать только непарные строки из файла с указанным номером (1 или 2).

-l номер_поля

Производить соединение по полю файла1 с заданным номером.

-2 номер_поля

Производить соединение по полю файла2 с заданным номером.

Рассмотрим примеры. Командная строка (см. листинг 6.45) выполняет соединение баз данных пользователей и групп, отсортированных в порядке возрастания идентификаторов групп. На стандартный вывод поступают входные имена, имена групп и основные каталоги (см. листинг 6.46).

```
join -1 4 -2 3 -o 1.1,2.1,1.6 -t : passwd.sorted  
group.sorted
```

Листинг 6.45. Пример использования служебной программы `join`.

```
halt:root:/sbin  
operator:root:/root  
root:root:/root  
shutdown:root:/sbin  
sync:root:/sbin  
bin:bin:/bin  
daemon:daemon:/sbin  
.  
.  
.
```

Листинг 6.46. Начальный фрагмент возможного результата работы служебной программы `join`.

Пусть имеется два упорядоченных по алфавиту справочника: номера телефонов и адреса электронной почты (см. листинг 6.47). Предполагается, что в качестве разделителя полей применяется символ табуляции. Тогда командная строка, показанная в листинге 6.48, позволит получить объединенный справочник (см. листинг 6.49).

Имя	Номер телефона
Иван	123-4567
Петр	123-5678
Яков	123-6789

Имя	Адрес электронной почты
Иван	ivan123@mail.ru
Олег	oleg@yahoo.com
Яков	yak@yandex.ru

Листинг 6.47. Возможное содержимое двух справочников с информацией о телефонных номерах и об адресах электронной почты.


```
join -t '<tab>' -a 1 -a 2 -e '-----' -o  
0,1.2,2.2 phone.txt email.txt
```

Листинг 6.48. Еще один пример использования служебной программы `join`.

Имя	Номер телефона	Адрес электронной почты
Иван	123-45-67	ivan123@mail.ru
Олег	-----	oleg@yahoo.com
Петр	123-56-78	-----
Яков	123-67-89	yak@yandex.ru

Листинг 6.49. Возможный результат работы служебной программы `join`.

Читателю предлагается самостоятельно выбрать правильный способ задания символа табуляции в качестве аргумента опции `-t` служебной программы `join`, а также объяснить, зачем нужен пробел в начале заголовка каждого из справочников.

Средства обработки каталогов

Обработка каталогов, как и обычных файлов, начинается с их открытия. Для этого предназначена функция `opendir()` (см. листинг 6.50).

```
#include <dirent.h>  
DIR *opendir (const char *dirname);
```

Листинг 6.50. Описание функции `opendir()`.

После открытия текущим становится первый элемент каталога. Если в дальнейшем понадобится вновь позиционироваться на первый элемент, можно воспользоваться функцией `rewinddir()` (см. листинг 6.51).

```
#include <dirent.h>  
void rewinddir (DIR *dirp);
```

Листинг 6.51. Описание функции `rewinddir()`.

Чтение элементов каталога выполняет функция `readdir()` (см. листинг 6.52), которая возвращает указатель на структуру, представляющую текущий элемент каталога; после ее завершения текущим становится

следующий элемент. По достижении конца каталога и в случае ошибки возвращается пустой указатель. Следовательно, если приложению необходимо различать обе ситуации, оно должно обнулить значение переменной `errno` перед вызовом `readdir()`, а затем, если результат равен `NULL`, проанализировать это значение.

```
#include <dirent.h>
struct dirent *readdir (DIR *dirp);
```

Листинг 6.52. Описание функции `readdir()`.

Согласно стандарту POSIX-2001, структура `dirent` содержит по крайней мере одно поле:

```
char d_name []; /* Имя файла */
```

В качестве необязательного описано еще одно поле:

```
ino_t d_ino; /* Порядковый номер файла */
```

Если элемент каталога представляет символическую ссылку, значение этого поля не определено.

Следует учитывать, что указатель, возвращаемый функцией `readdir()`, может ссылаться на область памяти, перезаписываемую другими обращениями к `readdir()` с тем же значением аргумента `dirp`. Кроме того, нужно помнить и о том, что содержимое читаемого каталога асинхронно изменяют другие процессы (потoki управления), создающие и удаляющие файлы.

После завершения работы с каталогом его следует закрыть с помощью функции `closedir()` (см. листинг 6.53), возвращающей 0 при нормальном завершении и -1 в случае ошибки.

```
#include <dirent.h>
int closedir (DIR *dirp);
```

Листинг 6.53. Описание функции `closedir()`.

Нередко чтение элементов каталога и сопоставление с шаблоном имен файлов сочетаются (см. выше раздел «Генерация маршрутных имен файлов»). Подобное сопоставление реализует функция `fnmatch()` (см. листинг 6.54).

```
#include <fnmatch.h>
int fnmatch (const char *file_pattern,
             const char *file_name, int flags);
```

Листинг 6.54. Описание функции `fnmatch()`.

На процесс сопоставления имени `file_name` с шаблоном `file_pattern` влияют следующие флаги:

`FNM_PATHNAME`

Трактовать имя `file_name` как маршрутное. Символу `/` в имени должен явным образом сопоставляться этот же символ в шаблоне (а не `*`, `?` или выражение в квадратных скобках). Если флаг не установлен, символ `/` трактуется наравне с другими.

`FNM_NOESCAPE`

При наличии этого флага символ `\` трактуется наравне с другими. В противном случае он играет экранирующую роль.

`FNM_PERIOD`

Если этот флаг установлен, точка в начале имени файла должна сопоставляться с точкой в шаблоне. Иначе точка в начале имени сопоставляется на общих основаниях.

В случае успешного сопоставления функция `fnmatch()` возвращает ноль, при неудаче результат равен `FNM_NOMATCH`, в случае ошибки возвращается другое ненулевое значение.

В качестве примера употребления описанных функций рассмотрим программу, которая выводит имена файлов текущего каталога, удовлетворяющие заданным в командной строке шаблонам (см. листинг 6.55).

```
#include <dirent.h>
#include <fnmatch.h>
#include <errno.h>
#include <stdio.h>

/* Программа сопоставляет имена файлов текущего */
/* каталога с заданными шаблонами */

#define SEARCH_DIR "."
```

```
static void match_names (DIR *dirp,
    const char *pattern) {
    struct dirent *dp;

    rewinddir (dirp);

    while (errno = 0, (dp = readdir (dirp)) != NULL) {
        if (fnmatch (pattern, dp->d_name, FNM_PERIOD)
            == 0) {
            (void) printf (" %s\n", dp->d_name);
        }
    }

    if (errno != 0) {
        perror ("Ошибка при чтении каталога "
            SEARCH_DIR);
    }
}

int main (int argc, char *argv []) {
    DIR *dirp;
    int i;

    if ((dirp = opendir (SEARCH_DIR)) == NULL) {
        perror ("Ошибка при открытии каталога "
            SEARCH_DIR);
        return (-1);
    }

    for (i = 1; i < argc; i++) {
        (void) printf ("Файлы каталога " SEARCH_DIR " ,"
            "удовлетворяющие шаблону %s\n", argv [i]);
        match_names (dirp, argv [i]);
    }
    return (closedir (dirp));
}
```

Листинг 6.55. Пример использования функций, обрабатывающих каталоги.

Справедливости ради нужно отметить, что приведенный пример носит искусственный характер, поскольку для генерации маршрутных

имен, удовлетворяющих заданному шаблону, стандартом POSIX-2001 предусмотрена функция `glob()` (см. листинг 6.56).

```
#include <glob.h>
int glob (const char *restrict file_pattern,
         int flags), int (*errfunc) (const char *epath,
         int eerrno), glob_t *restrict pglob);
void globfree (glob_t *pglob);
```

Листинг 6.56. Описание функций `glob()` и `globfree()`.

Функция `glob()` сопоставляет все доступные маршрутные имена с шаблоном `file_pattern` и генерирует список успешно сопоставленных имен. Результаты своей работы функция `glob()` возвращает в структуре типа `glob_t`, содержащей, согласно стандарту, по крайней мере следующие поля:

```
size_t gl_pathc; /* Число успешно сопоставленных */
                /* маршрутных имен */
char **gl_pathv; /* Массив указателей на успешно */
                /* сопоставленные маршрутные имена */
size_t gl_offs; /* Число элементов, которые */
                /* нужно зарезервировать в начале массива gl_pathv */
```

Структура, на которую указывает аргумент `pglob`, должна создаваться приложением. Другие области памяти, в том числе массив `gl_pathv`, резервируются функцией `glob()` по мере необходимости и освобождаются после обращения к `globfree()` с тем же значением `pglob`.

Работой функции `glob()` можно управлять с помощью следующих флагов.

`GLOB_APPEND`

Добавлять вновь сгенерированные маршрутные имена к предыдущим результатам функции `glob()`.

`GLOB_DOOFFS`

Принимать во внимание поле `gl_offs`. Если этот флаг установлен, то `pglob->gl_pathv` будет указывать на `pglob->gl_offs` пустых указателей, за которыми следуют `pglob->gl_pathc` указателей на маршрутные имена и завершающий пустой указатель.

GLOB_ERR

Завершать работу по достижении каталога, который не удастся открыть или прочитать. Обычно в таких случаях функция `glob()` продолжает процесс генерации маршрутных имен.

GLOB_MARK

Добавлять символ `/` после имен успешно сопоставленных файлов, являющихся каталогами.

GLOB_NOCHECK

При отсутствии успешно сопоставленных имен возвращать шаблон.

GLOB_NOESCAPE

Запретить экранирование символом `\`.

GLOB_NOSORT

Не сортировать сгенерированные имена по алфавиту.

Если в процессе своей работы функция `glob()` достигает каталога, который не удастся открыть или прочитать, а аргумент `errfunc` отличен от пустого указателя, то `glob()` вызывает `(*errfunc())` с двумя аргументами: «проблемным» маршрутным именем и соответствующим значением `errno`. Если этот вызов возвращает значение, отличное от нуля, или если установлен флаг `GLOB_ERR`, генерация маршрутных имен завершается, значения полей `pglob->gl_pathc` и `pglob->gl_pathv` устанавливаются в соответствии с уже обработанными маршрутами, а результат вызова функции `glob()` полагается равным `GLOB_ABORTED` (при нормальном завершении возвращается 0).

Перепишем программу, которая выводит имена файлов, удовлетворяющие заданному в командной строке шаблону, воспользовавшись функцией `glob()` (см. листинг 6.57).

```
#include <glob.h>
#include <errno.h>
#include <stdio.h>

/* Программа выводит маршрутные имена, */
/* сгенерированные по заданным шаблонам */
```

```
static int errfunc (const char *epath, int eerrno) {
    fprintf (stderr, "Ошибка при обработке каталога "
             "%s: ", epath);
    errno = eerrno;
    perror (NULL);
    return (0);
}

int main (int argc, char *argv []) {
    glob_t gl_buf;
    int i;

    for (i = 1; i < argc; i++) {
        (void) glob (argv [i], ((i == 1) ? 0 :
                           GLOB_APPEND) | GLOB_NOSORT, errfunc, &gl_buf);
    }

    (void) printf ("Маршрутные имена, сгенерированные"
                  "по заданным шаблонам:\n");
    for (i = 0; (unsigned) i < gl_buf.gl_pathc; i++) {
        (void) printf ("%s\n", gl_buf.gl_pathv [i]);
    }

    globfree (&gl_buf);
    return (0);
}
```

Листинг 6.57. Пример программы, использующей функции `glob()` и `globfree()`.

Выше, при описании языка `shell`, мы рассматривали служебную программу `basename`. Ее логическим дополнением является утилита `dirname`:

`dirname цепочка_символов`

Она выдает на стандартный вывод имя каталога, в котором находится поименованный цепочкой символов файл. Отметим, что цепочка символов трактуется служебной программой `dirname` как маршрутное имя только синтаксически: никаких проверок существования компонентов маршрутного имени не производится.

В качестве примера использования утилиты `dirname` приведем фрагмент командного файла, выполняемого при загрузке ОС Linux (см. листинг 6.58).

```
if [ "`dirname $RAW`" = "/dev/raw" -a -f /dev/raw
]; then
echo $" Please correct your"
    "/etc/sysconfig/rawdevices:"
echo $" rawdevices are now located in the"
    "directory /dev/raw/ "
echo $" If the command 'raw' still refers to"
    "/dev/raw as a file."
echo $" you'll have to upgrade your util-linux"
    "package"
exit 0
fi
```

Листинг 6.58. Пример использования служебной программы `dirname`.

Совместное использование утилит `basename` и `dirname` иллюстрирует листинг 6.59. Это командный файл с одним аргументом — маршрутным именем компилируемого C-файла, которое может быть задано как с расширением `.c`, так и без него.

```
gcc -Wall -W -pedantic -o ${basename "$1" .c}
    ${dirname "$1"/}${basename "$1" .c}.c
```

Листинг 6.59. Пример совместного использования служебных программ `basename` и `dirname`.

Вариант 1

1. **Согласно стандарту POSIX-2001, опция -u утилиты cat предписывает**
 - ☐ не выдавать повторяющиеся символы
 - ☐ из группы одинаковых строк выдавать только первую
 - ☐ передавать на стандартный вывод байты из входных файлов без задержек
2. **Согласно стандарту POSIX-2001, опция -i утилиты sort предписывает**
 - ☐ при сравнении игнорировать пробельные символы
 - ☐ при сравнении игнорировать непечатные символы
 - ☐ при сравнении игнорировать непечатные слова
3. **Согласно стандарту POSIX-2001, в расширенных регулярных выражениях могут использоваться**
 - ☐ обратные ссылки
 - ☐ прямые ссылки
 - ☐ сегментные ссылки

Вариант 2

1. Согласно стандарту POSIX-2001, опция -s утилиты more предписывает

- ☐ сжимать последовательные пустые строки в одну
- ☐ конкатенировать строки исходных файлов
- ☐ подавлять вывод строк, в которых не оказалось символов-разделителей

2. Согласно стандарту POSIX-2001, опция -d утилиты sort предписывает

- ☐ при сравнении использовать словарный порядок
- ☐ сравнивать строки через одну
- ☐ сжимать последовательности повторяющихся строк в одну

3. Согласно стандарту POSIX-2001, в расширенных регулярных выражениях для группирования подвыражений используются:

- ☐ незранированные круглые скобки
- ☐ экранированные круглые скобки
- ☐ экранированные фигурные скобки

Вариант 3

1. Согласно стандарту POSIX-2001, опция -A утилиты od задает

- ☐ основание системы счисления для содержимого
- ☐ основание системы счисления для смещений
- ☐ формат вывода символов

2. Согласно стандарту POSIX-2001, опция -r утилиты sort предписывает

- ☐ подавлять диагностические сообщения при ошибках открытия файлов
- ☐ заменить результат сравнения на противоположный
- ☐ применять утилиту sort рекурсивно

3. Согласно стандарту POSIX-2001, в расширенных регулярных выражениях для выделения интервальных выражений используются

- ☐ незранированные круглые скобки
- ☐ незранированные фигурные скобки
- ☐ экранированные фигурные скобки

Лекция 7. Процессы

Определяются основные понятия, ассоциированные с процессами. Описываются служебные программы и функции для опроса и изменения атрибутов процессов, а также для создания и завершения процессов.

Ключевые слова: процесс, атрибут процесса, идентификатор процесса, создание процесса, завершение процесса, нормальное завершение, аварийное завершение, зомби-процесс, время жизни процесса, порожденный процесс, активный процесс, родительский процесс, идентификатор родительского процесса, группа процессов, идентификатор группы процессов, время жизни группы процессов, задание, управление заданиями, сеанс, время жизни сеанса, лидер сеанса, управляющий терминал, управляющий процесс, приоритетный процесс, приоритетная группа процессов, приоритетное задание, фоновый процесс, фоновая группа процессов, фоновое задание, программа, реальный идентификатор пользователя процесса, реальный идентификатор группы процессов, действующий идентификатор пользователя процесса, действующий идентификатор группы процесса, переустановка действующего идентификатора пользователя, переустановка действующего идентификатора группы, сохраненный ПДП-идентификатор, сохраненный ПДГ-идентификатор, идентификаторы дополнительных групп процесса, маска режима создания файлов, поток управления, идентификатор потока управления, потоковобезопасная функция, активный поток управления, готовый к выполнению поток управления, вытесненный поток управления, блокированный поток управления, список потоков управления, планирование, приоритет, политика планирования, область планирования размещения, область планирования конкуренции, опрос атрибута процесса, изменение атрибута процесса, блокировка файла, сигнал, обработка сигналов, образ процесса, файл образа процесса, статус завершения процесса, остановленный процесс, регистрация функции, терминирование процесса.

Основные понятия

Напомним данное в стандарте POSIX-2001 определение процесса. Процесс – это адресное пространство вместе с выполняемыми в нем потоками управления, а также системными ресурсами, которые этим потокам требуются.

Каждый процесс обладает целым рядом атрибутов. Важнейшим среди них является идентификатор процесса – положительное целое число, однозначно идентифицирующее процесс в течение времени его жизни.

Процессы могут создаваться и завершаться. Время жизни процесса – это период от его создания до возврата идентификатора операционной системе.

После того как процесс создан с помощью функции `fork()`, он считается активным. До завершения процесса в его рамках существуют по крайней мере один поток управления и адресное пространство.

Процесс может перейти в неактивное состояние, и тогда некоторые из его ресурсов (но не идентификатор) могут быть возвращены системе. Когда по отношению к неактивному процессу выполняют функцию семейства `wait()`, системе возвращаются остальные ресурсы. Последний из них – идентификатор процесса, и на этом время жизни процесса заканчивается.

Завершение процесса может быть нормальным или аварийным. Нормальное завершение происходит, в частности, при возврате из функции `main()`.

Зомби-процесс – завершившийся процесс, подлежащий ликвидации после того, как код его завершения будет передан ожидающему этого другому процессу.

Процесс, создавший данный, называется родительским, в течение времени жизни которого существует идентификатор родительского процесса. По завершении времени жизни указанного процесса родительским становится определяемый реализацией системный процесс.

Группа – совокупность процессов, допускающая согласованную доставку сигналов. У каждой группы имеется уникальный положительный целочисленный идентификатор, представляющий ее в течение времени ее жизни. В такой роли выступает идентификатор процесса, именуемого лидером группы.

Временем жизни группы процессов называют период от создания группы до момента, когда ее покидает последний процесс (по причине завершения или смены группы).

Задание – это набор процессов, составляющих конвейер, а также порожденных ими процессов, входящих в одну группу.

Под управлением заданиями подразумеваются предоставленные пользователям средства выборочно (при)останавливать и затем продолжать (возобновлять) выполнение процессов. На отдельные задания ссылаются с помощью идентификаторов.

Сеансом называется множество групп процессов, сформированное для целей управления заданиями. Каждая группа принадлежит некоторому сеансу; считается, что все процессы группы принадлежит тому же сеансу.

Вновь созданный процесс присоединяется к сеансу своего создателя; в дальнейшем принадлежность сеансу может быть изменена.

Время жизни сеанса представляет собой период от создания сеанса до истечения времени жизни всех групп процессов, принадлежавших сеансу.

Лидер сеанса – процесс, создавший данный сеанс.

Управляющим терминалом называется терминал, ассоциированный с сеансом. У сеанса может быть не более одного управляющего терминала, а тот, в свою очередь, ассоциируется ровно с одним сеансом. Некоторые последовательности символов, вводимые с управляющего терминала, вызывают посылку сигналов всем процессам группы, ассоциированной с данным управляющим терминалом.

Управляющий процесс – это лидер сеанса, установивший соединение с управляющим терминалом. Если в дальнейшем терминал перестанет быть управляющим для сеанса, лидер сеанса утратит статус управляющего процесса.

Задания, группы процессов и процессы подразделяются на приоритетные (переднего плана) и фоновые. Процессы переднего плана, в отличие от фоновых, имеют некоторые привилегии при доступе к управляющему терминалу. В сеансе, установившем соединение с управляющим терминалом, может быть не более одной группы процессов, приоритетной по отношению к данному управляющему терминалу.

С каждым процессом ассоциируется идентификатор создавшего его пользователя. Этот атрибут называется реальным идентификатором пользователя процесса.

В момент создания процесса пользователь входил в некоторую группу, идентификатор которой называется реальным идентификатором группы процесса.

Для определения прав процесса (в том числе прав доступа к файлам) применяются действующие идентификаторы пользователя и группы, которые в общем случае могут отличаться от реальных.

Поведение процесса определяется исполняемой в его рамках программой – последовательностью инструкций для решения определенной задачи. Программы хранятся в файлах. В режим файлов входят биты переустановки действующих идентификаторов пользователя (ПДИП) и группы (ПДИГ). Если эти биты взведены, то при запуске программы на выполнение действующие идентификаторы пользователя и группы процесса могут наследоваться у файла.

Стандартом POSIX-2001 предусмотрены также сохраненные переустановленные действующие идентификаторы пользователя (сохраненные ПДИП-идентификаторы) и группы (сохраненные ПДИГ-идентификаторы) процесса. Эти атрибуты расширяют возможности, связанные со сменой действующих идентификаторов пользователя и группы процесса.

При определении прав доступа к файлам наравне с действующим идентификатором группы процесса используются идентификаторы дополнительных групп.

С процессом ассоциируются маска режима создания файлов, влияющая на устанавливаемый режим доступа к новым файлам, и ограничение на размер записываемых файлов.

Текущий и корневой каталоги и набор дескрипторов открытых файлов также относятся к числу атрибутов процессов.

Все перечисленные выше атрибуты разделяются существующими в рамках процесса потоками управления. К числу индивидуальных атрибутов относятся идентификатор, приоритет и политика планирования, значение переменной `errno`, ассоциированные с потоком управления пары ключ/значение, а также системные ресурсы, требующиеся для поддержки потока управления.

Всем потокам управления одного процесса доступны все объекты, адреса которых могут быть определены потоком. В число таких объектов входят статические переменные, полученные от функции `malloc()` области динамической памяти, унаследованная от системно-зависимых функций прямоадресуемая память, автоматические переменные и т.д.

По отношению к потокам управления вводится понятие безопасных функций, которые можно вызывать параллельно в нескольких потоках без нарушения корректности их функционирования. К числу безопасных принадлежат «чистые» функции, а также функции, обеспечивающие взаимное исключение перед доступом к разделяемым объектам. Если в стандарте явно не оговорено противное, функция считается потоковобезопасной.

Выполняющимся (активным) называется поток управления, обрабатываемый в данный момент процессором. В многопроцессорных конфигурациях может одновременно выполняться несколько потоков.

Поток управления считается готовым к выполнению, если он способен стать активным, но не может этого сделать из-за отсутствия свободного процессора.

Поток управления называют вытесненным, когда его выполнение приостановлено из-за того, что другой поток с более высоким приоритетом уже готов к выполнению.

Процесс (поток управления) считается блокированным, если для продолжения его выполнения должно стать истинным некоторое условие, отличное от доступности процессора.

Список потоков управления — это упорядоченный набор готовых к выполнению равноприоритетных потоков, очередность которых определяется политикой планирования. Множество наборов включает все потоки в системе, готовые к выполнению.

Планированием, согласно стандарту POSIX-2001, называется применение политики изменения списков потоков управления, а также политики выбора процесса или готового к выполнению потока управления для перевода как того, так и другого в число активных.

Под политикой планирования понимается набор правил, используемых для определения порядка выполнения процессов или потоков управления при достижении некоторой цели.

Политика планирования воздействует на порядок процессов (потоков управления) по крайней мере в следующих ситуациях:

- когда активный процесс (поток управления) блокируется или вытесняется;
- когда блокированный процесс (поток управления) становится готовым к выполнению.

Область планирования размещения — это набор процессоров, по отношению к которым в какой-то момент времени планируется поток управления.

Областью планирования конкуренции называется свойство потока управления, определяющее набор потоков, с которыми он конкурирует за ресурсы (например, за процессор). В стандарте POSIX-2001 предусмотрены две подобные области — `PTHREAD_SCOPE_SYSTEM` и `PTHREAD_SCOPE_PROCESS`.

Данные выше определения будут поясняться и раскрываться по мере описания служебных программ и функций, обслуживающих понятие процесса.

Опрос и изменение атрибутов процессов

Для выдачи информации о процессах служит утилита `ps`:

```
ps [-aA] [-defl] [-G список_групп] [-o формат] ...  
    [-p список_процессов]  
    [-t список_терминалов] [-U список_пользователей]  
    [-g список_групп]  
    [-n список_имен] [-u список_пользователей]
```

По умолчанию информация выдается обо всех процессах, имеющих тот же действующий идентификатор и тот же управляющий терминал, что и у текущего пользователя. При этом выводятся идентификатор процесса, имя управляющего терминала, истраченное к данному моменту процессорное время и имя программы (команды), выполняемой в рамках процесса. Например, выдача команды `ps` может выглядеть так, как показано в листинге 7.1.

PID	TTY	TIME	CMD
1594	ttyS4	00:00:02	sh
1645	ttyS4	00:00:00	sh
1654	ttyS4	00:02:45	rk.20.01
18356	ttyS4	00:00:00	prconecm
18357	ttyS4	00:00:00	sh
18358	ttyS4	00:00:00	ps

Листинг 7.1. Возможный результат использования утилиты `ps`.

Если нужна более подробная информация о более широком наборе процессов, следует пользоваться опциями. Перечислим наиболее употребительные из них.

-a

Выдать информацию обо всех процессах, ассоциированных с терминалами. Заметим, однако, что, во-первых, при получении информации о процессах контролируются права доступа (например, пользователю будут видны только порожденные им процессы), а во-вторых, по стандарту реализация может не включать в выдаваемый список лидеров сеансов.

-A

Выдать информацию обо всех процессах.

-G список_групп

Выдать информацию о процессах с заданными реальными идентификаторами групп.

-o формат

Выдать информацию о процессах в заданном формате.

-p список_процессов

Выдать информацию о процессах с заданными идентификаторами.

-t список_терминалов

Выдать информацию о процессах, ассоциированных с заданными терминалами. Способ задания терминалов зависит от реализации. Обычно указывается имя специального файла, например, `ttyS4`, или, если имя начинается с `tty`, просто `S4`.

`-U` список_пользователей

Выдать информацию о процессах с заданными реальными идентификаторами пользователей (они могут указываться и в виде входных имен).

Все перечисленные опции, кроме `-o`, ведут отбором процессов. Если задан целый ряд подобных опций, выводится информация обо всех специфицированных ими процессах.

Опции `-o` (их в командной строке может быть несколько) позволяют задать выходной формат информации о процессах. Указываются выводимые поля и, если нужно, отличные от подразумеваемых тексты соответствующих им заголовков, отделяющиеся от имени поля знаком равенства и продолжающиеся до конца аргумента опции `-o`.

Перечислим имена полей, которые могут указываться в выходном формате, и соответствующие им подразумеваемые заголовки.

`ruser (RUSER)`

Выдавать реальный идентификатор пользователя процесса (в символической или числовой форме).

`user (USER)`

Действующий идентификатор пользователя процесса.

`rgroup (RGROUP)`

Реальный идентификатор группы процесса.

`group (GROUP)`

Действующий идентификатор группы процесса.

`pid (PID)`

Идентификатор процесса.

`ppid` (PPID)

Идентификатор родительского процесса.

`pgid` (PGID)

Идентификатор группы процессов.

`pcpu` (%CPU)

Процент процессорного времени, потребляемый процессом.

`vsz` (VSZ)

Размер процесса в (виртуальной) памяти (в килобайтных блоках).

`nice` (NI)

Число, используемое как рекомендация системе при планировании процессов. Меньшие значения соответствуют более приоритетным процессам.

`etime` (ELAPSED)

Астрономическое время, прошедшее с момента запуска процесса.

`time` (TIME)

Процессорное время, потребленное процессом.

`tty` (TT)

Имя управляющего терминала.

`comm` (COMMAND)

Имя выполняемой команды (`argv [0]`).

`args` (COMMAND)

Выполняемая командная строка.

Например, командная строка, показанная в листинге 7.2, породит выдачу, фрагмент которой приведен в листинге 7.3

```
ps -A -o ruser,user,pid,ppid,TTY -o
    nice,vsz,args
```

Листинг 7.2. Пример использования утилиты ps.

```
RUSER  USER    PID PPID TTY  NI  VSZ  COMMAND
root   root      1    0    ?    0  1372  init [5]
root   root      4    1    ?    19  0    [ksoftirqd_CPU0]
root   root     555  1    ?    0  1428  syslogd -m 0
root   root     560  1    ?    0  1364  klogd -x
rpc     rpc       580  1    ?    0  1508  portmap
rpcuser rpcuser 608  1    ?    0  1560  rpc.statd
root   root     743  1    ?    0  2620  /usr/sbin/sshd
root   root     776  1    ?    0  2200  xinetd -stayalive -reuse -pidfi
root   root     805  1    ?    0  1500  rpc.rquotad
root   root     810  1    ?    0  1504  rpc.mountd
root   root     897  1    ?    0  3236  /usr/libexec/postfix/master
postfix postfix 906  897  ?    0  3384  nmgr -l -n qmgr -t fifo -u -c
root   root     918  1    ?    0  1400  gpm -t ps/2 -m /dev/mouse
root   root     936  1    ?    0  1548  crond
xfs     xfs       968  1    ?    0  4432  xfs -droppriv ч-daemon
nobody nobody 987  1    ?    0  36488  dictd 1.9.7: 0/0
root   daemon 1022  1    ?    0  1404  /usr/sbin/atd
root   root     1057 1    ?    0  5768  cupsd
root   root     1064 1    tty1 0  1344  /sbin/mingetty tty1
root   root     1070 1    ttyS2 0  1352  /sbin/agetty -i -L ttyS2 38400
root   root     1072 1    ?    0  2300  login -- galat
galat   galat    1086 1072 ttyS4 0  2260  -sh
root   root     1124 1085 ?    0  16900 /usr/bin/kdm_greet
postfix postfix 1826 897 ?    0  3304  pickup -l -t fifo -u -c
galat   galat    2013 1171 ttyS4 0  1940  /bin/sh -c ps -A -o user,pid,pp
galat   galat    2014 2013 ttyS4 0  2584  ps -A -o user,pid,ppid,TTY
```

Листинг 7.3. Фрагмент возможного результата использования утилиты ps.

Для опроса идентификаторов процесса, родительского процесса и группы процессов предусмотрены функции `getpid()` и `getppid()` `getpgrp()` (см. листинг 7.4).

```
#include <unistd.h>
pid_t getpid (void);

#include <unistd.h>
pid_t getppid (void);

#include <unistd.h>
pid_t getpgrp (void);
```

Листинг 7.4. Описание функций `getpid()`, `getppid()` и `getpgrp()`.

По стандарту эти функции всегда завершаются успешно, поэтому ошибочных кодов возврата не предусмотрено.

Для установки идентификатора группы процессов в целях управления заданиями предназначена функция `setpgid()` (см. листинг 7.5).

```
#include <unistd.h>
int setpgid (pid_t pid, pid_t pgid);
```

Листинг 7.5. Описание функции `setpgid()`.

Выполнение функции `setpgid()` влечет либо присоединение к существующей группе процессов, либо создание новой группы в рамках сеанса, в который входит вызывающий процесс. Процесс может установить идентификатор группы для себя или для порожденного процесса. Нельзя изменить идентификатор группы процессов лидера сеанса.

В случае успешного завершения функции `setpgid()` (результат при этом равен нулю) идентификатор группы процессов устанавливается равным `pgid` для заданного аргументом `pid` процесса. Если значение `pid` равно нулю, установка производится для вызывающего процесса. А если значение `pgid` равно нулю, то в качестве идентификатора группы процессов используется идентификатор процесса, заданного аргументом `pid`.

Для создания сеанса и установки идентификатора группы процессов служит функция `setsid()` (см. листинг 7.6).

```
#include <unistd.h>
pid_t setsid (void);
```

Листинг 7.6. Описание функции `setsid()`.

Если вызывающий процесс не является лидером группы, в результате выполнения функции `setsid()` будет создан новый сеанс, причем вызывающий процесс станет лидером этого сеанса, равно как и лидером новой группы процессов (без управляющего терминала и без других процессов в группе и сеансе).

Программа, показанная в листинге 7.7, служит примером использования (в том числе некорректного) описанных функций. Возможный результат работы этой программы приведен в листинге 7.8.

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

int main (void) {
    pid_t ppid;
    pid_t pgid;

    /* Отменим буферизацию стандартного вывода */
    setbuf (stdout, NULL);

    printf ("Атрибуты текущего процесса: pid: %d,"
           "ppid: %d, pgid: %d\n",
           getpid (), ppid = getppid (),
           pgid = getpgrp ());

    /* Выделимся в новую группу */
    if (setpgid (0, 0) != 0) {
        perror ("setpgid (0, 0)");
    }
    printf ("Новая группа текущего процесса: %d\n",
           getpgrp ());

    /* Попробуем создать новый сеанс */
    if (setsid () == (pid_t) (-1)) {
        perror ("setsid от имени лидера группы");
    }

    /* Вернемся в прежнюю группу */
    if (setpgid (0, pgid) != 0) {
        perror ("setpgid (0, pgid)");
    }
}
```

```
printf ("Группа текущего процесса после"
        "повторной смены: %d\n", getpgrp ());

/* Повторим попытку создать новый сеанс */
if (setsid () == (pid_t) (-1)) {
    perror ("setsid от имени не-лидера группы");
}
printf ("Группа текущего процесса после создания"
        "нового сеанса: %d\n", getpgrp ());

/* Попробуем сменить группу родительского
   процесса */
if (setpgid (ppid, 0) != 0) {
    perror ("setpgid (ppid, 0)");
}

/* Попробуем установить несуществующий */
/* идентификатор группы процессов */
if (setpgid (0, ppid) != 0) {
    perror ("setpgid (0, ppid)");
}

return (0);
}
```

Листинг 7.7. Пример программы, использующей функции getpid(), getppid(), getpgrp(), setpgid(), setsid().

```
Атрибуты текущего процесса: pid: 11726, ppid:
11725, pgid: 1153
Новая группа текущего процесса: 11726
setsid от имени лидера группы: Operation not
permitted
Группа текущего процесса после повторной смены:
1153
Группа текущего процесса после создания нового
сеанса: 11726
setpgid (ppid, 0): No such process
setpgid (0, ppid): Operation not permitted
```

Листинг 7.8. Возможный результат работы программы, показанной в листинге 7.7.

Опрос реальных и действующих идентификаторов пользователя и группы вызывающего процесса осуществляется с помощью функций `getuid()`, `geteuid()`, `getgid()`, `getegid()` (см. листинг 7.9). Как и `getpid()`, они всегда завершаются успешно.

```
#include <unistd.h>
uid_t getuid (void);

#include <unistd.h>
uid_t geteuid (void);

#include <unistd.h>
gid_t getgid (void);

#include <unistd.h>
gid_t getegid (void);
```

Листинг 7.9. Описание функций `getuid()`, `geteuid()`, `getgid()`, `getegid()`.

Более сложный интерфейс имеет функция `getgroups()`, предназначенная для получения идентификаторов дополнительных групп вызывающего процесса (см. листинг 7.10). Эти идентификаторы (в их число может входить и действующий идентификатор группы процесса) помещаются в массив `grouplist`.

```
#include <unistd.h>
int getgroups (int gidsetsize, gid_t grouplist []);
```

Листинг 7.10. Описание функции `getgroups()`.

Аргумент `gidsetsize` задает число элементов в массиве `grouplist`, а реальное количество записанных идентификаторов групп возвращается в виде результата функции. Если в качестве значения `gidsetsize` задать нуль, `getgroups()` выдаст количество дополнительных групп, не модифицируя массив `grouplist`.

Переустановить действующий идентификатор пользователя вызывающего процесса позволяют функции `setuid()` и `seteuid()` (см. листинг 7.11). Операция разрешена, если реальный или сохраненный ПДП-идентификатор пользователя совпадает со значением аргумента `uid`. Помимо этого, обладающие соответствующими привилегиями процессы с помощью

функции `setuid()` могут установить по значению `uid` все три идентификатора пользователя процесса – реальный, действующий и сохраненный.

```
#include <unistd.h>
int setuid (uid_t uid);

#include <unistd.h>
int seteuid (uid_t uid);
```

Листинг 7.11. Описание функций `setuid()` и `seteuid()`.

Для непривилегированных процессов по соображениям мобильности рекомендуется использование функции `seteuid()`.

Аналогичные функции для переустановки идентификаторов группы процесса показаны в листинге 7.12.

```
#include <unistd.h>
int setgid (gid_t gid);

#include <unistd.h>
int setegid (gid_t gid);
```

Листинг 7.12. Описание функций `setgid()` и `setegid()`.

Отметим, что функция для изменения списка дополнительных групп `setgroups()` относится к числу привилегированных и, следовательно, остается за рамками стандарта POSIX.

Проиллюстрируем использование описанных функций с помощью программы, показанной в листинге 7.13.

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

int main (void) {
    uid_t uid;
    int nsupgroups;
    gid_t *supgroupslst;
    int i;
```

```
/* Отменим буферизацию стандартного вывода */
setbuf (stdout, NULL);

printf ("Идентификаторы пользователя текущего"
        "процесса:\n" "реальный: %d, действующий:"
        "%d\n", uid = getuid (), geteuid ());

printf ("Идентификаторы группы текущего"
        "процесса:\n" "реальный: %d, действующий:"
        "%d\n", getgid (), getegid ());

printf ("Количество дополнительных групп"
        "текущего процесса: %d\n",
        nsupgroups = getgroups (0, supgroupslst));

if (nsupgroups > 0) {
if ((supgroupslst = (gid_t *) malloc
    (nsupgroups * sizeof (gid_t))) == NULL) {
    perror ("MALLOC");
} else if (getgroups (nsupgroups,
    supgroupslst) == (-1)) {
    perror ("GETGROUPS");
} else {
    /* Выдадим идентификаторы дополнительных */
    /* групп процесса */
    printf ("Идентификаторы дополнительных"
            "групп текущего процесса:\n");
    for (i = 0; i < nsupgroups; i++) {
        printf (" %d", supgroupslst [i]);
    }
    printf ("\n");
}
}

/* Попробуем переустановить идентификатор */
/* пользователя процесса */
if (setuid ((uid_t) 1) != 0) {
    perror ("setuid (1)");
}

printf ("Идентификаторы пользователя текущего"
        "процесса после первой смены:\n"
        "реальный: %d, действующий: %d\n",
        getuid (), geteuid ());
```

```
/* Попробуем вернуть прежний идентификатор */
/* пользователя процесса */
if (setuid (uid) != 0) {
    perror ("setuid (uid)");
}
printf ("Идентификаторы пользователя текущего"
        "процесса после второй смены:\n"
        "реальный: %d, действующий: %d\n",
        getuid (), geteuid ());

/* Попробуем сменить действующий идентификатор */
/* с помощью функции seteuid() */
if (seteuid ((uid_t) 1) != 0) {
    perror ("seteuid (1)");
}
printf ("Идентификаторы пользователя текущего"
        "процесса после третьей смены:\n"
        "реальный: %d, действующий: %d\n",
        getuid (), geteuid ());
return (0);
}
```

Листинг 7.13. Пример использования функций опроса и изменения идентификаторов пользователя процесса.

Если эту программу запустить от имени обычного пользователя, результат может выглядеть так, как показано в листинге 7.14.

```
Идентификаторы пользователя текущего процесса:
    реальный: 108, действующий: 108
Идентификаторы группы текущего процесса:
    реальный: 3, действующий: 3
Количество дополнительных групп текущего процесса: 1
Идентификаторы дополнительных групп текущего
процесса: 3
setuid (1): Operation not permitted
Идентификаторы пользователя текущего процесса после
первой смены:
    реальный: 108, действующий: 108
Идентификаторы пользователя текущего процесса после
второй смены:
    реальный: 108, действующий: 108
```

```
seteuid (1): Operation not permitted
Идентификаторы пользователя текущего процесса после
третьей смены:
реальный: 108, действующий: 108
```

Листинг 7.14. Возможный результат работы программы, показанной в листинге 7.13 и запущенной от имени обычного пользователя.

После запуска той же программы от имени суперпользователя может получиться результат, показанный в листинге 7.15.

```
Идентификаторы пользователя текущего процесса:
реальный: 0, действующий: 0
Идентификаторы группы текущего процесса:
реальный: 0, действующий: 0
Количество дополнительных групп текущего процесса: 7
Идентификаторы дополнительных групп текущего
процесса:
0 1 2 3 4 6 10
Идентификаторы пользователя текущего процесса после
первой смены:
реальный: 1, действующий: 1
setuid (uid): Operation not permitted
Идентификаторы пользователя текущего процесса после
второй смены:
реальный: 1, действующий: 1
Идентификаторы пользователя текущего процесса после
третьей смены:
реальный: 1, действующий: 1
```

Листинг 7.15. Возможный результат работы программы, показанной в листинге 7.13 и запущенной от имени суперпользователя.

Утратить статус суперпользователя легко, а вернуть трудно...

Наконец, сделаем владельцем выполняемого файла рассматриваемой программы пользователя с идентификатором 1, то же сделаем с владеющей группой, взведем в режиме этого файла биты ПДИП и ПДИГ(на ОС Linux можно воспользоваться командой `chmod ug+s`) и вновь запустим его от имени обычного пользователя (см. листинг 7.16).

```
Идентификаторы пользователя текущего процесса:
реальный: 108, действующий: 1
```

Идентификаторы группы текущего процесса:

реальный: 3, действующий: 1

Количество дополнительных групп текущего процесса: 1

Идентификаторы дополнительных групп текущего процесса:

3

Идентификаторы пользователя текущего процесса после первой смены:

реальный: 108, действующий: 1

Идентификаторы пользователя текущего процесса после второй смены:

реальный: 108, действующий: 108

Идентификаторы пользователя текущего процесса после третьей смены:

реальный: 108, действующий: 1

Листинг 7.16. Возможный результат работы программы, показанной в листинге 7.13 и запущенной от имени обычного пользователя после взведения в режиме выполнимого файла бита ПДИП.

Видно, что сохраненный ПДП-идентификатор помогает непривилегированному процессу переустанавливать действующий идентификатор пользователя.

Отметим, что если функция `getgroups()` применяется в программе несколько раз, то память под массив `supgroupslst` лучше отвести по максимуму, воспользовавшись выражением вида `nsupgroups = sysconf (_SC_NGROUPS_MAX) + 1;`

Для опроса и/или изменения маски режима создания файлов вызывающего процесса предназначены служебная программа `umask`:

```
umask [-S] [маска]
```

и одноименная функция (см. листинг 7.17).

```
#include <sys/stat.h> mode_t umask (mode_t cmask);
```

Листинг 7.17. Описание функции `umask()`.

Согласно стандарту, в маске могут фигурировать только биты режима доступа к файлам; трактовка прочих бит зависит от реализации.

Для служебной программы `umask` маска может задаваться и выводиться в символьной и восьмеричной формах. Маска в символьной форме является «позитивной» в том смысле, что заданные в ней биты режима доступа

будут сохранены при создании файла. Восьмеричная форма — «негативная», заданные в ней биты будут «вычитаться» (очищаться) из режима доступа, заданного при создании. Синтаксис маски в `umask` аналогичен утилите `chmod`.

Опция `-S` предписывает выдавать маску в символьной форме.

Функция `umask()` устанавливает новую («негативную») маску и возвращает старую.

Пример. Последовательность команд, показанная в листинге 7.18, выдаст результат, воспроизведенный в листинге 7.19.

```
umask 0
umask -S
umask -- -x
umask
```

Листинг 7.18. Пример использования служебной программы `umask`.

```
u=rwx,g=rwx,o=rwx
0111
```

Листинг 7.19. Возможный результат использования служебной программы `umask`.

Обратим внимание на употребление признака конца опций `--`. Он позволяет задавать аргументы, начинающиеся с минуса. Отметим также, что результаты, выдаваемые `umask`, могут в дальнейшем использоваться как аргументы данной утилиты (и в символьной, и в восьмеричных формах).

Создание и завершение процессов

Новые процессы создаются при помощи функции `fork()` (см. листинг 7.20).

```
#include <unistd.h>
pid_t fork (void);
```

Листинг 7.20. Описание функции `fork()`.

Новый (порожденный) процесс является точной копией процесса, вызвавшего `fork()` (родительского), за исключением следующих моментов.

- 1) У порожденного процесса свой идентификатор, равно как и идентификатор родительского процесса.
- 2) У порожденного процесса собственная копия файловых дескрипторов, ссылающихся на те же описания открытых файлов, что и соответствующие дескрипторы родительского процесса.
- 3) Порожденный процесс не наследует блокировки файлов, установленные родительским процессом.
- 4) Порожденный процесс создается с одним потоком управления — копией того, что вызвал `fork()`.
- 5) Имеются также некоторые тонкости, связанные с обработкой сигналов, на которых мы, однако, останавливаться не будем.

В случае успешного завершения функция `fork()` возвращает порожденному процессу 0, а родительскому процессу — идентификатор порожденного процесса. После этого оба процесса начинают независимо выполнять инструкции, расположенные за обращением к `fork()`. При неудаче родительскому процессу возвращается -1, новый процесс не создается.

Поскольку возвращаемые функцией `fork()` значения различны для обеих копий, родительский и порожденный процессы могут далее выполняться по-разному. Например, процесс-предок переходит в состояние ожидания завершения процесса-потомка либо, если процесс-потомок запущен асинхронно, продолжает выполнение параллельно с ним. Процесс-потомок при помощи функции семейства `exec()` подменяет программу, которая определяет поведение процесса, и передает ей управление и список аргументов.

Напомним, что заголовок функции `main()` С-программы выглядит в общем случае так, как показано в листинге 7.21.

```
int main (int argc, char *argv []);
```

Листинг 7.21. Заголовок функции `main()` С-программы.

Значение `argc` равно количеству аргументов; `argv` — это массив указателей собственно на аргументы, которые определяются исходя из командной строки, запускающей С-программу. В соответствии с принятым соглашением, значение `argc` не меньше единицы, а первый элемент массива `argv` указывает на цепочку символов, содержащую имя выполняемого файла.

Аналогичный смысл имеют аргументы функций семейства `exec()` (см. листинг 7.22).


```
#include <unistd.h>
extern char **environ;
int execl (const char *path, const char *arg0, ...
    /*, (char *) 0 */);
int execv (const char *path, char *const argv []);
int execlp (const char *path, const char *arg0,
    ... /*, (char *) 0, char *const envp [] */);
int execve (const char *path, char *const argv [],
    char *const envp []);
int execlp (const char *file, const char *arg0,
    ... /*, (char *) 0 */);
int execvp (const char *file, char *const argv []);
```

Листинг 7.22. Описание функций семейства `exec()`.

Функции семейства `exec()` заменяют текущий образ процесса новым (и, следовательно, в случае успешного завершения возврат в вызывающий процесс невозможен). Новый образ создается на основе выполняемого файла, называемого файлом образа процесса.

Переменная `environ` инициализируется как указатель на массив указателей на составляющие окружение цепочки символов. Массивы `argv` и `environ` завершаются пустым указателем.

Аргумент `path` указывает на маршрутное имя файла с новым образом процесса.

Аргумент `file` имеет аналогичный смысл, однако, если он задан как простое имя, то производится поиск в каталогах, заданных переменной окружения `PATH`.

Аргументы `arg0, ...,` являются указателями на цепочки символов, составляющие список аргументов нового образа процесса. Последним в списке располагается пустой указатель, а аргумент `arg0` должен указывать на имя файла-образа.

Аргумент `envp` имеет тот же смысл и назначение, что и переменная `environ`.

Файловые дескрипторы остаются открытыми в новом образе, если только они не были снабжены флагом `FD_CLOEXEC`.

Если у файла с новым образом процесса взведен бит ПДИП, действующий идентификатор пользователя процесса переустанавливается равным идентификатору владельца файла (аналогично для группы).

Следующие атрибуты процесса остаются неизменными:

- идентификатор процесса;
- идентификатор родительского процесса;

- идентификатор группы процессов;
- членство в сеансе;
- реальные идентификаторы пользователя и группы процесса;
- идентификаторы дополнительных групп;
- текущий и корневой каталоги;
- маска режима создания файлов;
- атрибуты, связанные с обработкой сигналов.

Родительский процесс реализует ожидание завершения процессов-потомков и получает информацию о его (завершения) статусе с помощью функций семейства `wait()` (см. листинг 7.23). Если информация о статусе завершения была доступна до вызова `wait()`, родительский процесс не приостанавливается, возврат из `wait()` происходит немедленно.

```
#include <sys/wait.h>
pid_t wait (int *stat_loc);
pid_t waitpid (pid_t pid, int *stat_loc, int
               options);
```

Листинг 7.23. Описание функций семейства `wait()`.

Функция `waitpid()` эквивалентна `wait()`, если аргумент `pid` равен (`pid_t`) `(-1)`, а аргумент `options` имеет нулевое значение. Аргумент `pid` задает набор порожденных процессов, статус завершения которых запрашивается. Значение (`pid_t`) `(-1)` представляет произвольный элемент множества порожденных процессов. Если `pid > 0`, имеется в виду один процесс с данным идентификатором. Нулевое значение специфицирует любого потомка из той же группы процессов, что и вызывающий. Наконец, при `pid < (pid_t) (-1)` запрашивается статус завершения любого порожденного процесса из группы, идентификатор которой равен абсолютной величине `pid`.

Аргумент `options` задается как побитное ИЛИ следующих флагов, определенных в заголовочном файле `<sys/wait.h>`.

`WNOHANG`

Функция `waitpid()` не приостанавливает выполнение вызывающего потока управления, если запрашиваемый статус не доступен немедленно.

`WUNTRACED`

Наряду со статусом завершения запрашивать статус остановленных, но еще не опрошенных процессов.

Если запрос статуса порожденного процесса завершился успешно, функции `wait()` и `waitpid()` возвращают идентификатор этого процесса и размещают по указателю `stat_loc` (если он отличен от `NULL`) значение, которое будет нулевым тогда и только тогда, когда выдан статус порожденного процесса, завершившегося по одной из трех причин:

- произошел возврат из функции `main` с нулевым результатом;
- порожденный процесс вызвал функцию `_exit()` или `exit()` с нулевым аргументом;
- завершились все потоки управления порожденного процесса.

Для анализа целочисленного значения `stat_val`, на которое указывает аргумент `stat_loc`, в файле `<sys/wait.h>` определен набор макросов. Например, значение `WIFEXITED (stat_val)` будет ненулевым в случае нормального завершения порожденного процесса; при этом `WEXITSTATUS (stat_val)` возвращает младший байт статуса. Макрос `WIFSTOPPED (stat_val)` возвращает ненулевое значение, если получен статус остановленного процесса.

Процесс может вызвать собственное завершение, обратившись к функциям семейства `exit()` (см. листинг 7.24).

```
#include <stdlib.h>
void exit (int status);
void _Exit (int status);
#include <unistd.h>
void _exit (int status);
```

Листинг 7.24. Описание функций семейства `exit()`.

Значением аргумента `status` могут служить константы `0`, `EXIT_SUCCESS`, `EXIT_FAILURE` или любое другое значение, хотя ожидающему родительскому процессу будет доступен только младший байт статуса (`status & 0377`).

Функция `exit()`, в отличие от `_Exit()` и `_exit()`, производит аккуратное завершение: выполняет зарегистрированные функции (см. ниже описание функции `atexit()`), выталкивает буфера, закрывает потоки, удаляет временные файлы и т.д. Функции `_Exit()` и `_exit()` эквивалентны. Все функции семейства `exit()` закрывают файловые дескрипторы, открытые вызывающим процессом.

Если родительский процесс выполняет функции `wait()` или `waitpid()`, ему передается младший байт значения `status`. Если родительский процесс этого не делает, функции семейства `exit()` переводят вызывающий процесс в состояние зомби.

Если у завершающегося процесса были потомки, родительским для них становится системный процесс, определяемый реализацией.

Функция `atexit()` (см. листинг 7.25) позволяет зарегистрировать функции, которые будут вызываться, если процесс завершается, обращаясь к `exit()` или возвращаясь из `main()`.

```
#include <stdlib.h>
int atexit (void (*func) (void));
```

Листинг 7.25. Описание функции `atexit()`.

Реализация должна поддерживать регистрацию по крайней мере тридцати двух функций и вызывать их в обратном порядке.

В листинге 7.26 приведен пример использования функций порождения и завершения процессов.

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

static void atefunc (void) {

    /* Перед завершением выдадим информацию о */
    /* процессах */
    printf ("Ситуация перед завершением "
           "родительского процесса\n");
    (void) system ("ps -o pid,ppid,vsz,args");
}

int main (void) {
    int pid;
    int stat;

    /* Отменим буферизацию стандартного вывода */
    setbuf (stdout, NULL);
    /* Зарегистрируем обработчик завершения процесса */
    if (atexit (atefunc) != 0) {
        perror ("ATEXIT");
        exit (1);
    }

    /* Создадим новый процесс */
    if ((pid = fork ()) < 0) {
```

```

    perror ("FORK");
    exit (2);
} else if (pid == 0) {
    /* Порожденный процесс */
    /* Выполним служебную программу ps */
    printf ("Ситуация с точки зрения порожденного"
           "процесса\n");
    (void) execl ("/bin/ps", "ps", "-o",
                 "pid,ppid,args", (char *) 0);
    perror ("EXEC");
    exit (3); /* execl() завершился неудачей */
} else {
    /* Родительский процесс */
    sleep (1);
    /* Вероятно, порожденный процесс уже завершился */
    /* Посмотрим на текущую ситуацию */
    printf ("Ситуация перед вызовом waitpid() в "
           "родительском процессе\n");
    (void) system ("ps -o pid,ppid,vsz,args");

    (void) waitpid (pid, &stat, 0);
    printf ("Статус завершения порожденного"
           "процесса с идентификатором %d: %d\n", pid,
           stat);
}
return 0;
}

```

Листинг 7.26. Пример использования функций порождения и завершения процессов.

Результат работы этой программы может выглядеть так, как показано в листинге 7.27 (выполнимый файл программы имеет имя prog30).

```

Ситуация с точки зрения порожденного процесса
PID PPID COMMAND
6123 1072 -sh
29568 6123 prog30
29569 29568 ps -o pid,ppid,args
Ситуация перед вызовом waitpid() в родительском процессе
PID PPID VSZ COMMAND
6123 1072 2260 -sh

```

```
29568 6123 1340 prog30
29569 29568    0 [ps <defunct>]
29570 29568 2584 ps -o pid,ppid,vsz,args
Статус завершения порожденного процесса с
идентификатором 29569: 0
Ситуация перед завершением родительского процесса
PID PPID VSZ COMMAND
6123 1072 2260 -sh
29568 6123 1340 prog30
29571 29568 2584 ps -o pid,ppid,vsz,args
```

Листинг 7.27. Возможный результат работы программы, использующей функции порождения и завершения процессов.

Обратим внимание на информацию о зомби-процессе (с пометкой `<defunct>`), а также на то, что функция, зарегистрированная с помощью `atexit()`, вызвана в результате возврата из `main()`.

Для терминирования процессов извне предназначена служебная программа `kill`, вызванная следующим образом:

```
kill -s TERM идентификатор_процесса ...
```

Если это не помогло, можно применить более сильный вариант:

```
kill -s KILL идентификатор_процесса ...
```

Разумеется, попытки терминирования процессов подвержены контролю прав доступа.

Вариант 1

1. **Согласно стандарту POSIX-2001, лидер сеанса – это**
 - ☐ процесс, создавший данный сеанс
 - ☐ процесс, потребляющий максимальный процент процессорного времени
 - ☐ процесс, завершившийся раньше других членов сеанса
2. **Согласно стандарту POSIX-2001, непривилегированный процесс с помощью функции `setuid()` может переустановить**
 - ☐ реальный идентификатор пользователя процесса
 - ☐ действующий идентификатор пользователя процесса
 - ☐ сохраненный ПДП-идентификатор процесса
3. **Согласно стандарту POSIX-2001, с помощью функции `exit` можно вернуть значение статуса завершения размером в**
 - ☐ восемь бит
 - ☐ целое
 - ☐ длинное целое

Вариант 2

1. **Согласно стандарту POSIX-2001, у сеанса может быть**
 - ☐ два управляющих терминала
 - ☐ один управляющий терминал
 - ☐ ни одного управляющего терминала

2. **Согласно стандарту POSIX-2001, процесс, обладающий соответствующими привилегиями, с помощью функции `setuid()` может переустановить**
 - ☐ реальный идентификатор пользователя процесса
 - ☐ действующий идентификатор пользователя процесса
 - ☐ сохраненный ПДП-идентификатор процесса

3. **Согласно стандарту POSIX-2001, функции, зарегистрированные с помощью `atexit()`, вызываются при выполнении**
 - ☐ возврата из функции `main()`
 - ☐ выполнения функции `exit()`
 - ☐ выполнения функции `_Exit()`

Вариант 3

1. Согласно стандарту POSIX-2001, управляющий процесс – это

- ☐ системный процесс, контролирующий сеанс
- ☐ системный процесс, контролирующий доступ к управляющему терминалу
- ☐ лидер сеанса, установивший соединение с управляющим терминалом

2. Согласно стандарту POSIX-2001, непривилегированный процесс с помощью функции `setuid()` может установить действующий идентификатор пользователя процесса равным

- ☐ произвольному неотрицательному целочисленному значению
- ☐ реальному идентификатору пользователя процесса
- ☐ сохраненному ПДП-идентификатору процесса

3. Согласно стандарту POSIX-2001, функции, зарегистрированные с помощью `atexit()`, выполняются

- ☐ в прямом порядке
- ☐ в обратном порядке
- ☐ в порядке, зависящем от реализации

Лекция 8. Средства межпроцессного взаимодействия

Рассматриваются средства локального межпроцессного взаимодействия — каналы, сигналы, очереди сообщений, семафоры, разделяемые сегменты памяти.

Ключевые слова: межпроцессное взаимодействие, локальное межпроцессное взаимодействие, канал, файловый дескриптор, ввод/вывод нижнего уровня, буферизованный ввод/вывод, поток, синхронный режим ввода/вывода, канал с командным интерпретатором, сигнал, событие, исключительная ситуация, генерация сигнала, посылка сигнала, доставка сигнала, прием сигнала, ждущий сигнал, блокировка сигнала, маска сигналов, способ обработки сигнала, подразумеваемый способ, игнорирование сигнала, функция обработки сигнала, асинхронно-сигнально-безопасная функция, АСБ-функция, ожидание доставки сигнала, набор сигналов, X/Open-расширение системного интерфейса, идентификатор средства межпроцессного взаимодействия, ключ средства межпроцессного взаимодействия, создатель средства межпроцессного взаимодействия, активное средство межпроцессного взаимодействия, операции с блокировкой, операции без блокировки, очередь сообщений, создание очереди сообщения, идентификатор очереди сообщения, прием сообщения, отправка сообщения, тип сообщения, текст сообщения, управление очередью сообщений, семафор, значение семафора, набор семафоров, создание набора семафоров, идентификатор набора семафоров, групповые операции с семафорами, увеличение значения семафора, уменьшение значения семафора, ожидание обнуления значения семафора, управляющие действия с семафорами, задача об обходящих философах, сегмент, разделяемый сегмент памяти, адресное пространство процесса, виртуальная память, создание разделяемого сегмента, идентификатор разделяемого сегмента, присоединение разделяемого сегмента, отсоединение разделяемого сегмента, управляющие действия над разделяемыми сегментами, критический интервал.

Каналы

Средства локального межпроцессного взаимодействия реализуют высокопроизводительную, детерминированную передачу данных между процессами в пределах одной системы.

К числу наиболее простых и в то же время самых употребительных средств межпроцессного взаимодействия принадлежат каналы, представляемые файлами соответствующего типа. Стандарт POSIX-2001 различает именованные и безымянные каналы. Напомним, что первые создаются функцией `mkfifo()` и одноименной служебной программой, а вторые — функцией `pipe()`. Именованным каналам соответствуют элементы файловой системы, ко вторым можно обращаться только посредством файловых дескрипторов. В остальном эти разновидности каналов эквивалентны.

Взаимодействие между процессами через канал может быть установлено следующим образом: один из процессов создает канал и передает другому соответствующий открытый файловый дескриптор. После этого процессы обмениваются данными через канал при помощи функций `read()` и `write()`. Примером подобного взаимодействия служит программа, показанная в листинге 8.1.

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

/* Программа копирует строки со стандартного ввода */
/* на стандартный вывод, "прокачивая" их через канал. */
/* Используются функции ввода/вывода нижнего уровня */

#define MY_PROMPT "Вводите строки\n"
#define MY_MSG    "Вы ввели: "

int main (void) {
    int fd [2];
    char buf [1];
    int new_line = 1; /* Признак того, что надо */
    /* выдать сообщение MY_MSG перед отображением */
    /* очередной строки */

    /* Создадим безымянный канал */
    if (pipe (fd) < 0) {
        perror ("PIPE");
```

```
    exit (1);
}

switch (fork ()) {
    case -1:
        perror ("FORK");
        exit (2);
    case 0:
        /* Чтение из канала и выдачу на стандартный */
        /* вывод реализуем в порожденном процессе. */
        /* Необходимо закрыть дескриптор, */
        /* предназначенный для записи в канал, */
        /* иначе чтение не завершится по концу файла */
        close (fd [1]);
        while (read (fd [0], buf, 1) == 1) {
            if (write (1, buf, 1) != 1) {
                perror ("WRITE TO STDOUT");
                break;
            }
        }
        exit (0);
}

/* Чтение со стандартного ввода и запись в канал */
/* возложим на родительский процесс. */
/* Из соображений симметрии закроем дескриптор, */
/* предназначенный для чтения из канала */
close (fd [0]);
if (write (fd [1], MY_PROMPT, sizeof (MY_PROMPT)
    - 1) !=
    sizeof (MY_PROMPT) - 1) {
    perror ("WRITE TO PIPE-1");
}

while (read (0, buf, 1) == 1) {
    /* Перед отображением очередной строки */
    /* нужно выдать сообщение MY_MSG */
    if (new_line) {
        if (write (fd [1], MY_MSG, sizeof (MY_MSG)
            - 1) != sizeof (MY_MSG) - 1) {
            perror ("WRITE TO PIPE-2");
            break;
        }
    }
}
```

```
    }  
}  
if (write (fd [1], buf, 1) != 1) {  
    perror ("WRITE TO PIPE-3");  
    break;  
}  
new_line = (buf [0] == '\n');  
}  
close (fd [1]);  
  
(void) wait (NULL);  
return (0);  
}
```

Листинг 8.1. Пример взаимодействия между процессами через канал с помощью функций ввода/вывода нижнего уровня.

Решение той же задачи, но с использованием функций буферизованного ввода/вывода, показано в листинге 8.2.

```
#include <unistd.h>  
#include <stdlib.h>  
#include <stdio.h>  
#include <limits.h>  
#include <sys/wait.h>  
#include <assert.h>  
  
/* Программа копирует строки со стандартного ввода */  
/* на стандартный вывод, "прокачивая" их через */  
/* канал. */  
/* Используются функции буферизованного */  
/* ввода/вывода /  
  
int main (void) {  
    int fd [2];  
    FILE *fp [2];  
    char line [LINE_MAX];  
  
    /* Создадим безымянный канал */  
    if (pipe (fd) < 0) {  
        perror ("PIPE");  
        exit (1);  
    }
```

```
    }

    /* Сформируем потоки по файловым дескрипторам */
    /* канала */
    assert ((fp [0] = fdopen (fd [0], "r")) != NULL);
    assert ((fp [1] = fdopen (fd [1], "w")) != NULL);

    /* Отменим буферизацию вывода */
    setbuf (stdout, NULL);
    setbuf (fp [1], NULL);

    switch (fork ()) {
        case -1:
            perror ("FORK");
            exit (2);
        case 0:
            /* Чтение из канала и выдачу на стандартный */
            /* вывод реализуем в порожденном процессе. */
            /* Необходимо закрыть поток, предназначенный */
            /* для записи в канал, иначе чтение не */
            /* завершится по концу файла */
            fclose (fp [1]);
            while (fgets (line, sizeof (line), fp [0])
!= NULL) {
                if (fputs (line, stdout) == EOF) {
                    break;
                }
            }

            exit (0);
        }

    /* Чтение со стандартного ввода и запись в канал */
    /* возложим на родительский процесс. */
    /* Из соображений симметрии закроем поток, */
    /* предназначенный для чтения из канала */
    fclose (fp [0]);
    fputs ("Вводите строки\n", fp [1]);
    while (fgets (line, sizeof (line),
        stdin) != NULL) {
        if ((fputs ("Вы ввели: ", fp [1]) == EOF) ||
            (fputs (line, fp [1]) == EOF)) {
```

```
        break;
    }
}
fclose (fp [1]);
(void) wait (NULL);
return (0);
}
```

Листинг 8.2. Пример взаимодействия между процессами через канал с помощью функций буферизованного ввода/вывода.

Если не указано противное, обмен данными через канал происходит в синхронном режиме: процесс, пытающийся читать из пустого канала, открытого кем-либо на запись, приостанавливается до тех пор, пока данные не будут в него записаны; с другой стороны, запись в полный канал задерживается до освобождения необходимого для записи места. Чтобы отменить подобный режим взаимодействия, надо связать с дескрипторами канала флаг статуса `O_NONBLOCK` (это может быть сделано при помощи функции `fcntl()`). В таком случае чтение или запись, которые невозможно выполнить немедленно, завершаются неудачей.

Подчеркнем, что при попытке чтения из пустого канала результат равен 0 (как признак конца файла), только если канал не открыт кем-либо на запись. Под «кем-либо» понимается и сам читающий процесс; по этой причине в приведенной выше программе потребовалось закрыть все экземпляры файлового дескриптора `fd [1]`, возвращенного функцией `pipe()` как дескриптор для записи в канал.

Функция `popen()`, описанная выше, при рассмотрении командного интерпретатора, является более высокоуровневой по сравнению с `pipe()`. Она делает сразу несколько вещей: порождает процесс, обеспечивает выполнение в его рамках заданной команды, организует канал между вызывающим и порожденным процессами и формирует необходимые потоки для этого канала. Если при обращении к `popen()` задан режим «w», то стандартный ввод команды, выполняющейся в рамках порожденного процесса, перенаправляется на конец канала, предназначенный для чтения; если задан режим «r», то в канал перенаправляется стандартный вывод.

После вызова `popen()` процесс может писать в канал или читать из него посредством функций буферизованного ввода/вывода, используя сформированный поток. Канал остается открытым до момента вызова функции `pclose()` (см. листинг 8.3).


```
#include <stdio.h>
int pclose (FILE *stream);
```

Листинг 8.3. Описание функции pclose().

Функция pclose() не только закрывает поток, сформированный popen(), но и дожидается завершения порожденного процесса, возвращая его статус.

Типичное применение popen() – организация канала для выдачи динамически порождаемых данных на устройство печати командой lp (см. листинг 8.4).

```
#include <stdio.h>
/* Программа печатает несколько первых строк */
/* треугольника Паскаля */
#define T_SIZE 16
int main (void) {
    FILE *outptr;
    long tp [T_SIZE]; /* Массив для хранения */
                        /* текущей строки треугольника */
    int i, j;

    /* Инициализируем массив, чтобы далее все */
    /* элементы */
    /* можно было считать и выводить единообразно */
    tp [0] = 1;
    for (i = 1; i < T_SIZE; i++) {
        tp [i] = 0;
    }

    /* Создадим канал с командой */
    if ((outptr = popen ("lp", "w")) == NULL) {
        perror ("POPEN");
        return (-1);
    }

    (void) fprintf (outptr, "\nТреугольник "
        "Паскаля:\n");

    for (i = 0; i < T_SIZE; i++) {
        /* Элементы очередной строки нужно считать */
        /* от конца к началу */
```

```
/* Элемент tp [0] пересчитывать не нужно */
for (j = i; j > 0; j--) {
    tp [j] += tp [j - 1];
}
/* Вывод строки треугольника в канал */
for (j = 0; j <= i; j++) {
    (void) fprintf (outptr, " %ld", tp [j]);
}
(void) fprintf (outptr, "\n");
}

return (pclose (outptr));
}
```

Листинг 8.4. Пример создания и использования канала для вывода данных.

Сходным образом можно организовать канал для чтения результатов выполнения команды (см. листинг 8.5).

```
#include <stdio.h>
#include <limits.h>
#include <assert.h>

#define MY_CMD "ls -l *.c"

int main (void) {
    FILE *inptr;
    char line [LINE_MAX];

    assert ((inptr = popen (MY_CMD, "r")) != NULL);

    while (fgets (line, sizeof (line), inptr) !=
           NULL) {
        fputs (line, stdout);
    }

    return (pclose (inptr));
}
```

Листинг 8.5. Пример создания и использования канала для ввода данных.

Сигналы

Как и каналы, сигналы являются внешне простым и весьма употребительным средством локального межпроцессного взаимодействия, но связанные с ними идеи существенно сложнее, а понятия — многочисленнее.

Согласно стандарту POSIX-2001, под сигналом понимается механизм, с помощью которого процесс или поток управления уведомляют о некотором событии, произошедшем в системе, или подвергают воздействию этого события. Примерами подобных событий могут служить аппаратные исключительные ситуации и специфические действия процессов. Термин «сигнал» используется также для обозначения самого события.

Говорят, что сигнал генерируется (или посылается) для процесса (потока управления), когда происходит вызвавшее его событие (например, выявлен аппаратный сбой, отработал таймер, пользователь ввел с терминала специфическую последовательность символов, другой процесс обратился к функции `kill()` и т.п.). Иногда по одному событию генерируются сигналы для нескольких процессов (например, для группы процессов, ассоциированных с некоторым управляющим терминалом).

В момент генерации сигнала определяется, посылается ли он процессу или конкретному потоку управления в процессе. Сигналы, сгенерированные в результате действий, приписываемых отдельному потоку управления (таких, например, как возникновение аппаратной исключительной ситуации), посылаются этому потоку. Сигналы, генерация которых ассоциирована с идентификатором процесса или группы процессов, а также с асинхронным событием (к примеру, пользовательский ввод с терминала) посылаются процессу.

В каждом процессе определены действия, предпринимаемые в ответ на все предусмотренные системой сигналы. Говорят, что сигнал доставлен процессу, когда взято для выполнения действие, соответствующее данным процессу и сигналу. Сигнал принят процессом, когда он выбран и возвращен одной из функций `sigwait()`.

В интервале от генерации до доставки или принятия сигнал называется ждущим. Обычно он невидим для приложений, однако доставку сигнала потоку управления можно блокировать. Если действие, ассоциированное с заблокированным сигналом, отлично от игнорирования, он будет ждать разблокирования.

У каждого потока управления есть маска сигналов, определяющая набор блокируемых сигналов. Обычно она достается в наследство от родительского потока.

С сигналом могут быть ассоциированы действия одного из трех типов.

SIG_DFL

Подразумеваемые действия, зависящие от сигнала. Они описаны в заголовочном файле `<signal.h>`.

SIG_IGN

Игнорировать сигнал. Доставка сигнала не оказывает воздействия на процесс.

указатель на функцию

Обработать сигнал, выполнив при его доставке заданную функцию. После завершения функции обработки процесс возобновляет выполнение с точки прерывания. Обычно функция обработки вызывается в соответствии со следующим C-заголовком: `void func (int signo);` где `signo` – номер доставленного сигнала.

Первоначально, до входа в функцию `main()`, реакция на все сигналы установлена как `SIG_DFL` или `SIG_IGN`.

Функция называется асинхронно-сигнально-безопасной (АСБ), если ее можно вызывать без каких-либо ограничений при обработке сигналов. В стандарте POSIX-2001 имеется список функций, которые должны быть либо повторно входимыми, либо непрерываемыми сигналами, что превращает их в АСБ-функции. В этот список включены 117 функций, в том числе почти все из рассматриваемых нами.

Если сигнал доставляется потоку, а реакция заключается в завершении, остановке или продолжении, весь процесс должен завершиться, остановиться или продолжиться.

Перейдем к изложению возможностей по генерации сигналов. Выше была кратко рассмотрена служебная программа `kill` как средство терминирования процессов извне. На самом деле она посылает заданный сигнал; то же делает и одноименная функция (см. листинг 8.6).

```
#include <signal.h>
int kill (pid_t pid, int sig);
```

Листинг 8.6. Описание функции `kill()`.

Сигнал задается аргументом `sig`, значение которого может быть нулевым; в этом случае действия функции `kill()` сводятся к проверке допустимости значения `pid` (нулевой результат – признак успешного завершения `kill()`).

Если `pid > 0`, это значение трактуется как идентификатор процесса. При нулевом значении `pid` сигнал посылается всем процессам из той же группы, что и вызывающий. Если значение `pid` равно `-1`, адресатами являются все процессы, которым вызывающий имеет право посылать сигналы. При прочих отрицательных значениях `pid` сигнал посылается группе процессов, чей идентификатор равен абсолютной величине `pid`.

Процесс имеет право послать сигнал адресату, заданному аргументом `pid`, если он (процесс) имеет соответствующие привилегии или его реальный или действующий идентификатор пользователя совпадает с реальным или сохраненным ПДП-идентификатором адресата.

У служебной программы `kill` имеется полезная опция `-l`, позволяющая увидеть соответствие между номерами сигналов и их мнемоническими именами. Результат выполнения команды

```
kill -l
```

может выглядеть так, как показано в листинге 8.7.

1) SIGHUP	2) SIGINT	3) SIGQUIT
4) SIGILL	5) SIGTRAP	6) SIGABRT
7) SIGBUS	8) SIGFPE	9) SIGKILL
10) SIGUSR1	11) SIGSEGV	12) SIGUSR2
13) SIGPIPE	14) SIGALRM	15) SIGTERM
17) SIGCHLD	18) SIGCONT	19) SIGSTOP
20) SIGTSTP	21) SIGTTIN	22) SIGTTOU
23) SIGURG	24) SIGXCPU	25) SIGXFSZ
26) SIGVTALRM	27) SIGPROF	28) SIGWINCH
29) SIGIO	30) SIGPWR	31) SIGSYS
32) SIGRTMIN	33) SIGRTMIN+1	34) SIGRTMIN+2
35) SIGRTMIN+3	36) SIGRTMIN+4	37) SIGRTMIN+5
38) SIGRTMIN+6	39) SIGRTMIN+7	40) SIGRTMIN+8
41) SIGRTMIN+9	42) SIGRTMIN+10	43) SIGRTMIN+11
44) SIGRTMIN+12	45) SIGRTMIN+13	46) SIGRTMIN+14
47) SIGRTMIN+15	48) SIGRTMAX 15	49) SIGRTMAX-14
50) SIGRTMAX-13	51) SIGRTMAX-12	52) SIGRTMAX-11
53) SIGRTMAX-10	54) SIGRTMAX-9	55) SIGRTMAX-8
56) SIGRTMAX-7	57) SIGRTMAX-6	58) SIGRTMAX-5
59) SIGRTMAX-4	60) SIGRTMAX-3	61) SIGRTMAX-2
62) SIGRTMAX-1	63) SIGRTMAX	

Листинг 8.7. Возможный результат выполнения команды `kill -l`.

Мы не будем пояснять назначение всех представленных в листинге сигналов, ограничившись кратким описанием тех, что фигурируют в стандарте POSIX-2001 как обязательные для реализации. Попутно отметим, что, согласно стандарту языка C, должны быть определены имена всего шести сигналов: SIGABRT, SIGFPE, SIGILL, SIGINT, SIGSEGV и SIGTERM.

SIGABRT

Сигнал аварийного завершения процесса. Подразумеваемая реакция предусматривает, помимо аварийного завершения, создание файла с образом памяти процесса.

SIGALRM

Срабатывание будильника. Подразумеваемая реакция — аварийное завершение процесса.

SIGBUS

Ошибка системной шины как следствие обращения к неопределенной области памяти. Подразумеваемая реакция — аварийное завершение и создание файла с образом памяти процесса.

SIGCHLD

Завершение, остановка или продолжение порожденного процесса. Подразумеваемая реакция — игнорирование.

SIGCONT

Продолжение процесса, если он был остановлен. Подразумеваемая реакция — продолжение выполнения или игнорирование (если процесс не был остановлен).

SIGFPE

Некорректная арифметическая операция. Подразумеваемая реакция — аварийное завершение и создание файла с образом памяти процесса.

SIGHUP

Сигнал разъединения. Подразумеваемая реакция – аварийное завершение процесса.

SIGILL

Некорректная команда. Подразумеваемая реакция – аварийное завершение и создание файла с образом памяти процесса.

SIGINT

Сигнал прерывания, поступивший с терминала. Подразумеваемая реакция – аварийное завершение процесса.

SIGKILL

Уничтожение процесса (этот сигнал нельзя перехватить для обработки или проигнорировать). Подразумеваемая реакция – аварийное завершение процесса.

SIGPIPE

Попытка записи в канал, из которого никто не читает. Подразумеваемая реакция – аварийное завершение процесса.

SIGQUIT

Сигнал выхода, поступивший с терминала. Подразумеваемая реакция – аварийное завершение и создание файла с образом памяти процесса.

SIGSEGV

Некорректное обращение к памяти. Подразумеваемая реакция – аварийное завершение и создание файла с образом памяти процесса.

SIGSTOP

Остановка выполнения (этот сигнал нельзя перехватить для обработки или проигнорировать). Подразумеваемая реакция – остановка процесса.

SIGTERM

Сигнал терминирования. Подразумеваемая реакция — аварийное завершение процесса.

SIGTSTP

Сигнал остановки, поступивший с терминала. Подразумеваемая реакция — остановка процесса.

SIGTTIN

Попытка чтения из фонового процесса. Подразумеваемая реакция — остановка процесса.

SIGTTOU

Попытка записи из фонового процесса. Подразумеваемая реакция — остановка процесса.

SIGUSR1, SIGUSR2

Определяемые пользователем сигналы. Подразумеваемая реакция — аварийное завершение процесса.

SIGPOLL

Опрашиваемое событие. Подразумеваемая реакция — аварийное завершение процесса.

SIGPROF

Срабатывание таймера профилирования. Подразумеваемая реакция — аварийное завершение процесса.

SIGSYS

Некорректный системный вызов. Подразумеваемая реакция — аварийное завершение и создание файла с образом памяти процесса.

SIGTRAP

Попадание в точку трассировки/прерывания. Подразумеваемая реакция – аварийное завершение и создание файла с образом памяти процесса.

SIGURG

Высокоскоростное поступление данных в сокет. Подразумеваемая реакция – игнорирование.

SIGVTALRM

Срабатывание виртуального таймера. Подразумеваемая реакция – аварийное завершение процесса.

SIGXCPU

Исчерпан лимит процессорного времени. Подразумеваемая реакция – аварийное завершение и создание файла с образом памяти процесса.

SIGXFSZ

Превышено ограничение на размер файлов. Подразумеваемая реакция – аварийное завершение и создание файла с образом памяти процесса.

Процесс (поток управления) может послать сигнал самому себе с помощью функции `raise()` (см. листинг 8.8). Для процесса вызов `raise()` эквивалентен `kill(getpid(), sig)`;

```
#include <signal.h>
int raise (int sig);
```

Листинг 8.8. Описание функции `raise()`.

Посылка сигнала самому себе использована в функции `abort()` (см. листинг 8.9), вызывающей аварийное завершение процесса. (Заметим, что этого не произойдет, если функция обработки сигнала `SIGABRT` не возвращает управления. С другой стороны, `abort()` отменяет блокирование или игнорирование `SIGABRT`).

```
#include <stdlib.h>
void abort (void);
```

Листинг 8.9. Описание функции `abort()`.

Опросить и изменить способ обработки сигналов позволяет функция `sigaction()` (см. листинг 8.10).

```
#include <signal.h>
int sigaction (int sig, const struct sigaction
               *restrict act, struct sigaction *restrict oact);
```

Листинг 8.10. Описание функции `sigaction()`.

Для описания способа обработки сигнала используется структура `sigaction`, которая должна содержать по крайней мере следующие поля:

```
void (*sa_handler) (int); /* Указатель на функцию */
/* обработки сигнала или */
/* один из макросов SIG_DFL или SIG_IGN */
sigset_t sa_mask; /* Дополнительный набор */
/* сигналов, блокируемых на время выполнения */
/* функции обработки */
int sa_flags; /* Флаги, влияющие на поведение */
/* сигнала */
void (*sa_sigaction) (int, siginfo_t *, void *);
/* Указатель на функцию обработки сигнала */
```

Приложение, соответствующее стандарту, не должно одновременно использовать поля обработчиков `sa_handler` и `sa_sigaction`.

Тип `sigset_t` может быть целочисленным или структурным и представлять набор сигналов (см. далее).

Тип `siginfo_t` должен быть структурным по крайней мере со следующими полями:

```
int si_signo; /* Номер сигнала */
int si_errno; /* Значение переменной errno, */
/* ассоциированное с данным сигналом */
int si_code; /* Код, идентифицирующий причину */
/* сигнала */
pid_t si_pid; /* Идентификатор процесса, */
/* пославшего сигнал */
uid_t si_uid; /* Реальный идентификатор */
/* пользователя процесса, пославшего сигнал */
void *si_addr; /* Адрес, вызвавший генерацию */
/* сигнала */
```

```
int si_status; /* Статус завершения порожденного */  
                /* процесса */  
long si_band;  /* Событие, связанное с сигналом */  
                /* SIGPOLL */
```

В заголовочном файле `<signal.h>` определены именованные константы, предназначенные для работы с полем `si_code`, значения которого могут быть как специфичными для конкретного сигнала, так и универсальными. К числу универсальных кодов относятся:

`SI_USER`

Сигнал послан функцией `kill()`.

`SI_QUEUE`

Сигнал послан функцией `sigqueue()`.

`SI_TIMER`

Сигнал сгенерирован в результате срабатывания таймера, установленного функцией `timer_settime()`.

`SI_ASYNCIO`

Сигнал вызван завершением асинхронной операции ввода/вывода

`SI_MSGQ`

Сигнал вызван приходом сообщения в пустую очередь сообщений.

Из кодов, специфичных для конкретных сигналов, мы упомянем лишь несколько, чтобы дать представление о степени детализации диагностики, предусмотренной стандартом POSIX-2001. (Из имени константы ясно, к какому сигналу она относится.)

`ILL_ILLOPC`

Некорректный код операции.

`ILL_COPROC`

Ошибка сопроцессора.

FPE_INTDIV

Целочисленное деление на нуль.

FPE_FLTOVF

Переполнение при выполнении операции вещественной арифметики.

FPE_FLTSUB

Индекс вне диапазона.

SEGV_MAPERR

Адрес не отображен на объект.

BUS_ADRALN

Некорректное выравнивание адреса.

BUS_ADRERR

Несуществующий физический адрес.

TRAP_BRKPT

Процесс достиг точки прерывания.

TRAP_TRACE

Срабатывание трассировки процесса.

CLD_EXITED

Завершение порожденного процесса.

CLD_STOPPED

Остановка порожденного процесса.

`POLL_PRI`

Поступили высокоприоритетные данные.

Вернемся непосредственно к описанию функции `sigaction()`. Если аргумент `act` отличен от `NULL`, он указывает на структуру, специфицирующую действия, которые будут ассоциированы с сигналом `sig`. По адресу `oact` (если он не `NULL`) возвращаются сведения о прежних действиях. Если значение `act` есть `NULL`, обработка сигнала остается неизменной; подобный вызов можно использовать для опроса способа обработки сигналов.

Следующие флаги в поле `sa_flags` влияют на поведение сигнала `sig`.

`SA_NOCLDSTOP`

Не генерировать сигнал `SIGCHLD` при остановке или продолжении порожденного процесса (значение аргумента `sig` должно равняться `SIGCHLD`).

`SA_RESETHAND`

При входе в функцию обработки сигнала `sig` установить подразумеваемую реакцию `SIG_DFL` и очистить флаг `SA_SIGINFO` (см. далее).

`SA_SIGINFO`

Если этот флаг не установлен и определена функция обработки сигнала `sig`, она вызывается с одним целочисленным аргументом — номером сигнала. Соответственно, в приложении следует использовать поле `sa_handler` структуры `sigaction`. При установленном флаге `SA_SIGINFO` функция обработки вызывается с двумя дополнительными аргументами, как `void func(int sig, siginfo_t *info, void *context)`; второй аргумент указывает на данные, поясняющие причину генерации сигнала, а третий может быть преобразован к указателю на тип `ucontext_t` — контекст процесса, прерванного доставкой сигнала. В этом случае приложение должно использовать поле `sa_sigaction` и поля структуры типа `siginfo_t`. В частности, если значение `si_code` неположительно, сигнал был сгенерирован процессом с идентификатором `si_pid` и реальным идентификатором пользователя `si_uid`.

SA_NODEFER

По умолчанию обрабатываемый сигнал добавляется к маске сигналов процесса при входе в функцию обработки; флаг SA_NODEFER предписывает не делать этого, если только sig не фигурирует явным образом в sa_mask.

Опросить и изменить способ обработки сигналов можно и на уровне командного интерпретатора, посредством специальной встроенной команды trap:

```
trap [действие условие ...]
```

Аргумент «условие» может задаваться как EXIT (завершение командного интерпретатора) или как имя доставленного сигнала (без префикса SIG). При задании аргумента «действие» минус обозначает подразумеваемую реакцию, пустая цепочка (" ") — игнорирование. Если в качестве действия задана команда, то при наступлении условия она обрабатывается как eval действие.

Команда trap без аргументов выдает на стандартный вывод список команд, ассоциированных с каждым из условий. Выдача имеет формат, пригодный для восстановления способа обработки сигналов (см. листинг 8.11).

```
save_traps=$(trap)
. . .
eval "$save_traps"
```

Листинг 8.11. Пример сохранения и восстановления способа обработки сигналов посредством специальной встроенной команды trap.

Обеспечить выполнение утилиты logout из домашнего каталога пользователя во время завершения командного интерпретатора можно с помощью команды, показанной в листинге 8.12.

```
trap '$HOME/logout' EXIT
```

Листинг 8.12. Пример использования специальной встроенной команды trap.

При перенаправлении вывода в файл приходится считаться с возможностью возникновения ошибок, специфичных для каналов. Чтобы защитить от них процедуры начальной загрузки, в ОС Linux применяются

связки из игнорирования и последующего восстановления подразумеваемой реакции на сигнал SIGPIPE (см. листинг 8.13).

```
trap "" PIPE
echo "$INITLOG_ARGS -n $0 -s \"$1\" -e 1" >&21
trap - PIPE
```

Листинг 8.13. Пример использования специальной встроенной команды `trap` для защиты от ошибок, специфичных для каналов.

К техническим аспектам можно отнести работу с наборами сигналов, которая выполняется посредством функций, показанных в листинге 8.14. Функции `sigemptyset()` и `sigfillset()` инициализируют набор, делая его, соответственно, пустым или «полным». Функция `sigaddset()` добавляет сигнал `signo` к набору `set`, `sigdelset()` удаляет сигнал, а `sigismember()` проверяет вхождение в набор. Обычно признаком завершения является нулевой результат, в случае ошибки возвращается `-1`. Только `sigismember()` выдает `1`, если сигнал `signo` входит в набор `set`.

```
#include <signal.h>
int sigemptyset (sigset_t *set);
int sigfillset (sigset_t *set);
int sigaddset (sigset_t *set, int signo);
int sigdelset (sigset_t *set, int signo);
int sigismember (const sigset_t *set, int signo);
```

Листинг 8.14. Описание функций для работы с наборами сигналов.

Функция `sigprocmask()` (см. листинг 8.15) предназначена для опроса и/или изменения маски сигналов процесса, определяющей набор блокируемых сигналов.

```
#include <signal.h>
int sigprocmask (int how,
    const sigset_t *restrict set, sigset_t *restrict oset);
```

Листинг 8.15. Описание функции `sigprocmask()`.

Если аргумент `set` отличен от `NULL`, он указывает на набор, используемый для изменения текущей маски сигналов. Аргумент `how` определяет способ изменения; он может принимать одно из трех значений:

SIG_BLOCK (результатирующая маска получается в результате объединения текущей и заданной аргументом `set`), SIG_SETMASK (результатирующая маска устанавливается равной `set`) и SIG_UNBLOCK (маска `set` вычитается из текущей).

По адресу `oset` (если он не `NULL`) возвращается прежняя маска. Если значение `set` есть `NULL`, набор блокируемых сигналов остается неизменным; подобный вызов можно использовать для опроса текущей маски сигналов процесса.

Если к моменту завершения `sigprocmask()` будут существовать ждущие неблокированные сигналы, по крайней мере один из них должен быть доставлен до возврата из `sigprocmask()`.

Нельзя блокировать сигналы, не допускающие игнорирования.

Функция `sigpending()` (см. листинг 8.16) позволяет выяснить набор заблокированных сигналов, ожидающих доставки вызывающему процессу (поток управления). Дождаться появления подобного сигнала можно с помощью функции `sigwait()` (см. листинг 8.17).

```
#include <signal.h>
int sigpending (sigset_t *set);
```

Листинг 8.16. Описание функции `sigpending()`.

```
#include <signal.h>
int sigwait (const sigset_t *restrict set,
             int *restrict sig);
```

Листинг 8.17. Описание функции `sigwait()`.

Функция `sigwait()` выбирает ждущий сигнал из заданного набора (он должен включать только заблокированные сигналы), удаляет его из системного набора ждущих сигналов и помещает его номер по адресу, заданному аргументом `sig`. Если в момент вызова `sigwait()` нужного сигнала нет, процесс (поток управления) приостанавливается до появления такового.

Отметим, что стандарт POSIX-2001 не специфицирует воздействие функции `sigwait()` на обработку сигналов, включенных в набор `set`. Чтобы дождаться доставки обрабатываемого или терминирующего процесс сигнала, можно воспользоваться функцией `pause()` (см. листинг 8.18).

```
#include <unistd.h>
int pause (void);
```

Листинг 8.18. Описание функции `pause()`.

Функция `pause()` может ждать доставки сигнала неопределенно долго. Возврат из `pause()` осуществляется после возврата из функции обработки сигнала (результат при этом равен `-1`). Если прием сигнала вызывает завершение процесса, возврата из функции `pause()`, естественно, не происходит.

Несмотря на внешнюю простоту, использование функции `pause()` сопряжено с рядом тонкостей. При наивном подходе сначала проверяют некоторое условие, связанное с сигналом, и, если оно не выполнено (сигнал отсутствует), вызывают `pause()`. К сожалению, сигнал может быть доставлен в промежутке между проверкой и вызовом `pause()`, что нарушает логику работы процесса и способно привести к его зависанию. Решить подобную проблему позволяет функция `sigsuspend()` (см. листинг 8.19) в сочетании с рассмотренной выше функцией `sigprocmask()`.

```
#include <signal.h>
int sigsuspend (const sigset_t *sigmask);
```

Листинг 8.19. Описание функции `sigsuspend()`.

Функция `sigsuspend()` заменяет текущую маску сигналов вызывающего процесса на набор, заданный аргументом `sigmask`, а затем переходит в состояние ожидания, аналогичное функции `pause()`. После возврата из `sigsuspend()` (если таковой произойдет) восстанавливается прежняя маска сигналов.

Обычно парой функций `sigprocmask()` и `sigsuspend()` обрамляют критические интервалы. Перед входом в критический интервал посредством `sigprocmask()` блокируют некоторые сигналы, а на выходе вызывают `sigsuspend()` с маской, которую возвратила `sigprocmask()`, восстанавливая тем самым набор заблокированных сигналов и дожидаясь их доставки.

В качестве примера использования описанных выше функций работы с сигналами рассмотрим упрощенную реализацию функции `abort()` (см. листинг 8.20).

```
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
void abort (void) {
    struct sigaction sact;
    sigset_t sset;

    /* Вытолкнем буфера */
    (void) fflush (NULL);
```

```
/* Снимем блокировку сигнала SIGABRT */
if ((sigemptyset (&sset) == 0) &&
    (sigaddset (&sset, SIGABRT) == 0)) {
    (void) sigprocmask (SIG_UNBLOCK, &sset,
        (sigset_t *) NULL);
}
/* Пошлем себе сигнал SIGABRT. */
/* Возможно, его перехватит функция обработки, */
/* и тогда вызывающий процесс может не завершиться */
raise (SIGABRT);

/* Установим подразумеваемую реакцию на сигнал */
/* SIGABRT */
sact.sa_handler = SIG_DFL;
sigfillset (&sact.sa_mask);
sact.sa_flags = 0;
(void) sigaction (SIGABRT, &sact, NULL);

/* Снова пошлем себе сигнал SIGABRT */
raise (SIGABRT);

/* Если сигнал снова не помог, попробуем еще */
/* одно средство завершения */
_exit (127);
}

int main (void) {
    printf ("Перед вызовом abort()\n");
    abort ();
    printf ("После вызова abort()\n");
    return 0;
}
```

Листинг 8.20. Упрощенная реализация функции `abort()` как пример использования функций работы с сигналами.

В качестве нюанса, характерного для работы с сигналами, отметим, что до первого обращения к `raise()` нельзя закрыть потоки (можно только вытолкнуть буфера), поскольку функция обработки сигнала `SIGABRT`, возможно, осуществляет вывод.

Еще одним примером использования механизма сигналов может служить приведенная в листинге 8.21 упрощенная реализация функции

sleep(), предназначенной для «засыпания» на заданное число секунд. (Можно надеяться, что не описанные пока средства работы с временем интуитивно понятны.)

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

/* Функция обработки сигнала SIGALRM. Она ничего */
/* не делает, но игнорировать сигнал нельзя */
static void signal_handler (int sig) {
    /* В демонстрационных целях распечатаем номер */
    /* обрабатываемого сигнала */
    printf ("Принят сигнал %d\n", sig);
}

/* Функция для "засыпания" на заданное число */
/* секунд */
/* Результат равен разности между заказанной и */
/* фактической продолжительностью "сна" */
unsigned int sleep (unsigned int seconds) {
    time_t before, after;
    unsigned int slept;
    sigset_t set, oset;
    struct sigaction act, oact;

    if (seconds == 0) {
        return 0;
    }

    /* Установим будильник на заданное время, */
    /* но перед этим блокируем сигнал SIGALRM */
    /* и зададим свою функцию обработки для него */
    if ((sigemptyset (&set) < 0) ||
        (sigaddset (&set, SIGALRM) < 0) ||
        sigprocmask (SIG_BLOCK, &set, &oset)) {
        return seconds;
    }

    act.sa_handler = signal_handler;
    act.sa_flags = 0;
```

```
act.sa_mask = oset;
if (sigaction (SIGALRM, &act, &oact) < 0) {
    return seconds;
}

before = time ((time_t *) NULL);
(void) alarm (seconds);

/* Как атомарное действие восстановим старую */
/* маску сигналов (в надежде, что она не */
/* блокирует SIGALRM) и станем ждать доставки */
/* обрабатываемого сигнала */
(void) sigsuspend (&oact);
/* Сигнал доставлен и обработан */

after = time ((time_t *) NULL);

/* Восстановим прежний способ обработки */
/* сигнала SIGALRM */
(void) sigaction (SIGALRM, &oact,
    (struct *) NULL);

/* Восстановим первоначальную маску сигналов */
(void) sigprocmask (SIG_SETMASK, &oact,
    (sigset_t *) NULL);

return ((slept = after - before)
    > seconds ? 0 : (seconds - slept));
}

int main (void) {
    struct sigaction act;

    /* В демонстрационных целях установим обработку */
    /* прерывания с клавиатуры */
    act.sa_handler = signal_handler;
    (void) sigemptyset (&act.sa_mask);
    act.sa_flags = 0;
    (void) sigaction (SIGINT, &act,
        (struct sigaction *) NULL);

    printf ("Заснем на 10 секунд\n");
```

```
printf ("Проснулись, не дослав %d секунд\n",  
       sleep (10));  
return (0);  
}
```

Листинг 8.21. Упрощенная реализация функции `sleep()` как пример использования механизма сигналов.

Обратим внимание на применение функции `sigsuspend()`, которая реализует (неделимую) транзакцию снятия блокировки сигналов и перехода в режим ожидания. Отметим также, что по умолчанию при входе в функцию обработки к маске добавляется принятый сигнал для защиты от бесконечной рекурсии. Наконец, если происходит возврат из функции `sigsuspend()` (после возврата из функции обработки), то автоматически восстанавливается маска сигналов, существовавшая до вызова `sigsuspend()`. В данном случае в этой маске заблокирован сигнал `SIGALRM`, и потому можно спокойно менять способ его обработки.

Вызвать «недосыпание» приведенной программы можно, пошлав ей сигнал `SIGALRM` (например, посредством команды `kill -s SIGALRM идентификатор_процесса`) или `SIGINT` (путем нажатия на клавиатуре терминала комбинации клавиш `CTRL+C`).

Очереди сообщений

Мы переходим к рассмотрению средств локального межпроцессного взаимодействия, относящихся к необязательной части стандарта POSIX-2001, именуемой «X/Open-расширение системного интерфейса» (XSI). Будут описаны очереди сообщений, семафоры и разделяемые сегменты памяти.

Остановимся сначала на понятиях и структурах, общих для всех трех упомянутых средств.

Каждая очередь сообщений, набор семафоров и разделяемый сегмент однозначно идентифицируются положительным целым числом, которое обычно обозначается, соответственно, как `msgid`, `semid` и `shmid` и возвращается в качестве результатов функций `msgget()`, `semget()` и `shmget()`.

При получении идентификаторов средств межпроцессного взаимодействия используется еще одна сущность — ключ, а для его генерации предназначена функция `ftok()` (см. листинг 8.22). Аргумент `path` должен задавать маршрутное имя существующего файла, к которому вызывающий процесс может применить функцию `stat()`. В качестве значения аргумента `id`, по соображениям мобильности, рекомендуется использовать однобайтный символ. Гарантируется, что функция `ftok()` сгенерирует один и

тот же ключ для заданной пары (файл, символ) и разные ключи для разных пар.

```
#include <sys/ipc.h>
key_t ftok (const char *path, int id);
```

Листинг 8.22. Описание функции `ftok()`.

С идентификатором средства межпроцессного взаимодействия ассоциирована структура данных, содержащая информацию о допустимых и выполненных операциях. Соответствующие декларации сосредоточены в заголовочных файлах `<sys/msg.h>`, `<sys/sem.h>` и `<sys/shm.h>`.

В упомянутую структуру входит подструктура `ipc_perm` с данными о владельцах и режиме доступа, описанная в файле `<sys/ipc.h>` и содержащая по крайней мере следующие поля.

```
uid_t uid;      /* Идентификатор владельца */
gid_t gid;      /* Идентификатор владеющей группы */
uid_t cuid;     /* Идентификатор пользователя, */
                /* создавшего данное средство */
                /* межпроцессного взаимодействия */
gid_t cgid;     /* Идентификатор создавшей группы */
mode_t mode;    /* Режим доступа на чтение/запись */
```

Управление доступом к описываемым средствам межпроцессного взаимодействия осуществляется аналогично файловому, только наряду (и наравне) с владельцами (пользователем и группой) рассматриваются те, кто эти средства создал (создатели).

Опросить статус присутствующих в данный момент в системе (т. е. активных) средств межпроцессного взаимодействия позволяет служебная программа `ipcs`:

```
ipcs [-qms] [-a | -bcopt]
```

По умолчанию выдается краткая информация обо всех средствах — очередях сообщений, семафорах и разделяемых сегментах памяти. Если нужно ограничиться их отдельными видами, следует воспользоваться опциями `-q`, `-s` и/или `-m`, соответственно.

Следующие опции управляют форматом выдачи. Задание опции `-a` равносильно указанию всех опций формата. Опция `-b` предписывает выдавать лимиты на размер (максимальное количество байт в сообщениях очереди и т.п.), `-c` — имена пользователя и группы создателя средства,

-o — информацию об использовании (количество сообщений в очереди, их суммарный размер и т.п.), -p — информацию о процессах (идентификаторы последнего отправителя, получателя и т.п.), -t — информацию о времени (последняя управляющая операция, последняя отправка сообщения и т.п.).

Для удаления из системы активных средств межпроцессного взаимодействия предназначена служебная программа `ipcrm` (разумеется, подверженная контролю прав доступа). Удаляемые средства могут задаваться идентификаторами или ключами:

```
ipcrm [-q msgid | -Q msgkey | -s semid | -S semkey  
      | -m shmid | -M shmkey | ...
```

На этом мы завершаем изложение общих вопросов, относящихся к средствам межпроцессного взаимодействия, и переходим к рассмотрению специфических возможностей каждого из них.

Механизм очередей сообщений позволяет процессам взаимодействовать, обмениваясь данными. Данные передаются между процессами дискретными порциями, называемыми сообщениями. Процессы выполняют над сообщениями две основные операции — прием и отправку. Процессы, отправляющие или принимающие сообщение, могут приостанавливаться, если требуемую операцию невозможно выполнить немедленно. В частности, могут быть отложены попытки отправить сообщение в заполненную до отказа очередь, получить сообщение из пустой очереди и т.п. («операции с блокировкой»). Если же указано, что приостанавливать процесс нельзя, «операции без блокировки» либо выполняются немедленно, либо завершаются неудачей.

Прежде чем процессы смогут обмениваться сообщениями, один из них должен создать очередь. Одновременно определяются первоначальные права на выполнение операций для различных процессов, в том числе соответствующих управляющих действий над очередями.

Для работы с очередями сообщений стандарт POSIX-2001 предусматривает следующие функции (см. листинг 8.23): `msgget()` (получение идентификатора очереди сообщений), `msgctl()` (управление очередью сообщений), `msgsnd()` (отправка сообщения) и `msgrcv()` (прием сообщения).

```
#include <sys/msg.h>  
int msgget (key_t key, int msgflg);  
int msgsnd (int msqid, const void *msgp,  
            size_t msgsz, int msgflg);  
ssize_t msgrcv (int msqid, void *msgp,  
                size_t msgsz, long msgtyp, int msgflg);
```

```
int msgctl (int msqid, int cmd,  
            struct msqid_ds *buf);
```

Листинг 8.23. Описание функций для работы с очередями сообщений.

Структура `msqid_ds`, ассоциированная с идентификатором очереди сообщений, должна содержать по крайней мере следующие поля.

```
struct ipc_perm msg_perm; /* Данные о правах */  
                        /* доступа к очереди сообщений */  
msgqnum_t msg_qnum; /* Текущее количество */  
                        /* сообщений в очереди */  
msglen_t msg_qbytes; /* Максимально допустимый */  
                        /* суммарный размер сообщений в очереди */  
pid_t msg_lspid; /* Идентификатор процесса, */  
                        /* отправившего последнее сообщение */  
pid_t msg_lrpid; /* Идентификатор процесса, */  
                        /* принявшего последнее сообщение */  
time_t msg_stime; /* Время последней отправки */  
time_t msg_rtime; /* Время последнего приема */  
time_t msg_ctime; /* Время последнего */  
                        /* изменения посредством msgctl() */
```

Перейдем к детальному рассмотрению функций для работы с очередями сообщений.

Функция `msgget()` возвращает идентификатор очереди сообщений, ассоциированный с ключом `key`. Новая очередь, ее идентификатор и соответствующая структура `msqid_ds` создаются для заданного ключа, если значение аргумента `key` равно `IPC_PRIVATE` или очередь еще не ассоциирована с ключом, а в числе флагов `msgflg` задан `IPC_CREAT`.

Если необходима уверенность в том, что очередь с указанным ключом создается заново, в дополнение к флагу `IPC_CREAT` следует установить `IPC_EXCL`. Тогда попытка получить идентификатор уже существующий очереди завершится неудачей.

Структура `msqid_ds` для новой очереди инициализируется следующим образом.

- 1) Значения полей `msg_perm.cuid`, `msg_perm.uid`, `msg_perm.cgid` и `msg_perm.gid` устанавливаются равными действующим идентификаторам пользователя и группы вызывающего процесса.
- 2) Младшие девять бит поля `msg_perm.mode` устанавливаются равными младшим девяти битам значения `msgflg`.

- 3) Поля `msg_qnum`, `msg_lspid`, `msg_lrpid`, `msg_stime` и `msg_rtime` обнуляются.
- 4) В поле `msg_ctime` помещается текущее время, а в поле `msg_qbytes` – определенный в системе лимит.

Один из тонких вопросов, связанных с созданием очереди сообщений, заключается в выборе ключа. Всем процессам, которые намереваются работать с общей очередью сообщений, для получения идентификатора `msqid` необходимо знать ключ очереди. Задание ключа одинаковым константным значением во всех этих программах небезопасно, поскольку может оказаться так, что тот же ключ будет случайно задействован и другими программами. Как одно из возможных решений рекомендуется использование функции `ftok()`, вычисляющей действительно «уникальный» ключ.

В листинге 8.24 приведен простейший пример программы, где создается очередь сообщений с правами доступа, указанными в командной строке.

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

/* Программа создает очередь сообщений. */
/* В командной строке задаются имя файла */
/* для ftok() и режим доступа к очереди сообщений */

#define FTOK_CHAR    'G'

int main (int argc, char *argv []) {
    key_t key;
    int msqid;
    int mode = 0;

    if (argc != 3) {
        fprintf (stderr, "Использование: %s"
                 "маршрутное_имя режим доступа\n", argv [0]);
        return (1);
    }

    if ((key = ftok (argv [1],
                    FTOK_CHAR)) == (key_t) (-1)) {
        perror ("FTOK");
        return (2);
    }
}
```

```
    }  
    (void) sscanf (argv [2], "%o",  
        (unsigned int *) &mode);  
  
    if ((msqid = msgget (key, IPC_CREAT | mode)) < 0) {  
        perror ("MSGGET");  
        return (3);  
    }  
  
    return 0;  
}
```

Листинг 8.24. Пример программы, создающей очередь сообщений.

Если после выполнения этой программы воспользоваться командой `ipcs -q`, то результат может выглядеть так, как показано в листинге 8.25.

```
----- Message Queues -----  
key          msqid  owner   perms  used-bytes  messages  
0x47034bac   163840 galat   644    0              0
```

Листинг 8.25. Возможный результат опроса статуса очередей сообщений.

Удалить созданную очередь из системы, соответствующей стандарту POSIX-2001, можно командой `ipcrm -q 163840`.

Операции отправки/приема сообщений выполняют функции `msgsnd()` и `msgrcv()`; `msgsnd()` помещает сообщения в очередь, а `msgrcv()` читает и «достает» их оттуда.

В обоих случаях первый аргумент задает идентификатор очереди; второй является указателем на содержащую сообщение структуру. Сообщение состоит из двух частей: текста (последовательности байт) и так называемого типа (положительного целого числа). Тип, указанный во время отправки, используется впоследствии при выборе сообщения из очереди. Аргумент `msgsz` определяет длину сообщения; аргумент `msgflg` задает флаги.

В зависимости от значения, указанного в качестве аргумента `msgtyp` функции `msgrcv()`, из очереди выбирается то или иное сообщение. Если значение аргумента равно нулю, запрашивается первое сообщение в очереди, если больше нуля — первое сообщение типа `msgtyp`, а если меньше нуля — первое сообщение наименьшего из типов, не превышающих абсолютную величину аргумента `msgtyp`. Пусть, например, в очередь последовательно помещены сообщения с типами 5, 3 и 2. Тогда вызов `msgrcv (msqid, msgp, size, 0, flags)` выберет из очереди сообщение

с типом 5, поскольку оно отправлено первым; вызов `msgrcv (msqid, msgp, size, -4, flags)` – последнее сообщение, так как 2 – это наименьший из возможных типов в указанном диапазоне; наконец, вызов `msgrcv (msqid, msgp, size, 3, flags)` – сообщение с типом 3.

Во многих приложениях взаимодействующим посредством очереди сообщений процессам требуется синхронизировать свое выполнение. Например, процесс-получатель, пытавшийся прочитать сообщение и обнаруживший, что очередь пуста (либо сообщение указанного типа отсутствует), должен иметь возможность подождать, пока процесс-отправитель не поместит в очередь требуемое сообщение. Аналогичным образом, процесс, желающий отправить сообщение в очередь, в которой нет достаточного для него места, может ожидать его освобождения в результате чтения сообщений другими процессами. Процесс, вызвавший подобного рода «операцию с блокировкой», приостанавливается до тех пор, пока либо станет возможным выполнение операции, либо будет ликвидирована очередь. С другой стороны, имеются приложения, где подобные ситуации должны приводить к немедленному (и неудачному) завершению вызова функции.

Если не указано противное, функции `msgsnd()` и `msgrcv()` выполняют операции с блокировкой, например: `msgsnd (msqid, msgp, size, 0); msgrcv (msqid, msgp, size, type, 0)`. Чтобы выполнить операцию без блокировки, необходимо установить флаг `IPC_NOWAIT`: `msgsnd (msqid, msgp, size, IPC_NOWAIT); msgrcv (msqid, msgp, size, type, IPC_NOWAIT)`.

Аргумент `msgp` указывает на значение структурного типа, в котором представлены тип и тело сообщения (см. листинг 8.26).

```
struct msgbuf {  
    long mtype;          /* Тип сообщения */  
    char mtext [1];      /* Текст сообщения */  
};
```

Листинг 8.26. Описание структурного типа для представления сообщений.

Для хранения реальных сообщений в прикладной программе следует определить аналогичную структуру, указав желаемый размер сообщения, например, так, как это сделано в листинге 8.27.

```
#define MAXSZTMSG 8192  
  
struct mymsgbuf {  
    long mtype;          /* Тип сообщения */
```

```
    char mtext [MAXSZTMSG]; /* Текст сообщения */
};
struct mymsgbuf msgbuf;
```

Листинг 8.27. Описание структуры для хранения сообщений.

В качестве аргумента `msgsz` обычно указывается размер текстового буфера, например: `sizeof (msgbuf.text)`.

Если не указано противное, в случае, когда длина выбранного сообщения больше, чем `msgsz`, вызов `msgrcv()` завершается неудачей. Если же установить флаг `MSG_NOERROR`, длинное сообщение обрезается до `msgsz` байт. Отброшенная часть пропадает, а вызывающий процесс не получает никакого уведомления о том, что сообщение обрезано.

При успешном завершении `msgsnd()` возвращает 0, а `msgrcv()` – значение, равное числу реально полученных байт; при неудаче возвращается -1.

Процессы, обладающие достаточными правами доступа, посредством функции `msgctl()` могут получать информацию о состоянии очереди, изменять ряд характеристик, удалять очередь.

Управляющее действие определяется значением аргумента `cmd`. Допустимых значений три: `IPC_STAT` – получить информацию о состоянии очереди, `IPC_SET` – переустановить характеристики очереди, `IPC_RMID` – удалить очередь.

Команды `IPC_STAT` и `IPC_SET` для хранения информации об очереди используют имеющуюся в прикладной программе структуру типа `msgid_ds`, указатель на которую содержит аргумент `buf`: `IPC_STAT` копирует в нее ассоциированную с очередью структуру данных, а `IPC_SET`, наоборот, в соответствии с ней обновляет ассоциированную структуру. Команда `IPC_SET` позволяет переустановить значения идентификаторов владельца (`msg_perm.uid`) и владеющей группы (`msg_perm.gid`), режима доступа (`msg_perm.mode`), максимально допустимый суммарный размер сообщений в очереди (`msg_qbytes`). Увеличить значение `msg_qbytes` может только процесс, обладающий соответствующими привилегиями.

В листинге 8.28 приведена программа, изменяющая максимально допустимый суммарный размер сообщений в очереди. Предполагается, что очередь сообщений уже создана, а ее идентификатор известен. Читателю предлагается выполнить эту программу с разными значениями максимально допустимого суммарного размера (как меньше, так и больше текущего), действуя от имени обычного и привилегированного пользователя.

```
#include <stdio.h>
#include <sys/msg.h>
```

```
int main (int argc, char *argv []) {
    int msqid;
    struct msqid_ds msqid_ds;

    if (argc != 3) {
        fprintf (stderr, "Использование: %s"
            "идентификатор_очереди максимальный_размер\n",
            argv [0]);
        return (1);
    }

    (void) sscanf (argv [1], "%d", &msqid);

    /* Получим исходное значение структуры данных */
    if (msgctl (msqid, IPC_STAT, &msqid_ds) == -1) {
        perror ("IPC_STAT-1");
        return (2);
    }
    printf ("Максимальный размер очереди до"
        "изменения: %ld\n", msqid_ds.msg_qbytes);

    (void) sscanf (argv [2], "%d", (int *)
        &msqid_ds.msg_qbytes);

    /* Попробуем внести изменения */
    if (msgctl (msqid, IPC_SET, &msqid_ds) == -1) {
        perror ("IPC_SET");
    }

    /* Получим новое значение структуры данных */
    if (msgctl (msqid, IPC_STAT, &msqid_ds) == -1) {
        perror ("IPC_STAT-2");
        return (3);
    }
    printf ("Максимальный размер очереди после"
        "изменения: %ld\n", msqid_ds.msg_qbytes);

    return 0;
}
```

Листинг 8.28. Пример программы управления очередями сообщений.

Две программы, показанные в листингах 8.29 и 8.30, демонстрируют полный цикл работы с очередями сообщений — от создания до удаления. Программа из листинга 8.29 представляет собой родительский процесс, читающий строки со стандартного ввода и отправляющий их в виде сообщений процессу-потомку (листинг 8.30). Последний принимает сообщения и выдает их тела на стандартный вывод. Предполагается, что программа этого процесса находится в файле `msq_child` текущего каталога.

```
#include <unistd.h>
#include <stdio.h>
#include <limits.h>
#include <string.h>
#include <sys/wait.h>
#include <sys/msg.h>

/* Программа копирует строки со стандартного */
/* ввода на стандартный вывод, "прокачивая" их */
/* через очередь сообщений      */

#define FTOK_FILE      "/home/galat"
#define FTOK_CHAR      "G"
#define MSGQ_MODE      0644
#define MY_PROMPT      "Вводите строки\n"
#define MY_MSG          "Вы ввели: "

int main (void) {
    key_t key;
    int msqid;
    struct mymsgbuf {
        long mtype;
        char mtext [LINE_MAX];
    } line_buf, msgbuf;

    switch (fork ()) {
        case -1:
            perror ("FORK");
            return (1);
        case 0:
            /* Чтение из очереди и выдачу на стандартный */
            /* вывод */
            /* реализуем в порожденном процессе.      */
            void) execl ("./msq_child", "msq_child",
```

```
        FTOK_FILE, FTOK_CHAR, (char *) 0);
    perror ("EXEC");
    return (2); /* execl() завершился неудачей */
}

/* Чтение со стандартного ввода и запись */
/* в очередь возложим на родительский процесс */
/* Выработаем ключ для очереди сообщений */
if ((key = ftok (FTOK_FILE, FTOK_CHAR [0])) !=
    (key_t) (-1)) {
    perror ("FTOK");
    return (3);
}

/* Получим идентификатор очереди сообщений */
if ((msqid = msgget (key, IPC_CREAT | MSGQ_MODE))
    < 0) {
    perror ("MSGGET");
    return (4);
}

/* Приступим к отправке сообщений в очередь */
msgbuf.mtype = line_buf.mtype = 1;
strncpy (msgbuf.mtext, MY_PROMPT,
    sizeof (msgbuf.mtext));
if (msgsnd (msqid, (void *) &msgbuf,
    strlen (msgbuf.mtext) + 1, 0) != 0) {
    perror ("MSGSEND-1");
}

strncpy (msgbuf.mtext, MY_MSG,
    sizeof (msgbuf.mtext));

while (fgets (line_buf.mtext,
    sizeof (line_buf.mtext), stdin) != NULL) {
    if (msgsnd (msqid, (void *) &msgbuf,
        strlen (msgbuf.mtext) + 1, 0) != 0) {
        perror ("MSGSEND-2");
        break;
    }
    if (msgsnd (msqid, (void *) &line_buf,
        strlen (line_buf.mtext) + 1, 0) != 0) {
```

```
        perror ("MSGSEND-3");
        break;
    }
}

/* Удалим очередь */
if (msgctl (msqid, IPC_RMID, NULL) == -1) {
    perror ("MSGCTL-IPC_RMID");
    return (6);
}

return (0);
}
```

Листинг 8.29. Передающая часть программы работы с очередями сообщений.

```
#include <stdio.h>
#include <limits.h>
#include <sys/msg.h>

/* Программа получает сообщения из очереди */
/* и копирует их тела на стандартный вывод */

#define MSGQ_MODE    0644

int main (int argc, char *argv []) {
    key_t key;
    int msqid;
    struct mymsgbuf {
        long mtype;
        char mtext [LINE_MAX];
    } msgbuf;

    if (argc != 3) {
        fprintf (stderr, "Использование: %s имя_файла"
                 "цепочка_символов\n", argv [0]);
        return (1);
    }

    /* Выработаем ключ для очереди сообщений */
    if ((key = ftok (argv [1],
```



```
    *argv [2])) == (key_t) (-1)) {
    perror ("CHILD FTOK");
    return (2);
}

/* Получим идентификатор очереди сообщений */
if ((msqid = msgget (key,
    IPC_CREAT | MSGQ_MODE)) < 0) {
    perror ("CHILD MSGGET");
    return (3);
}

/* Цикл приема сообщений и выдачи строк */
while (msgrcv (msqid, (void *) &msgbuf,
    sizeof (msgbuf.mtext), 0, 0) > 0) {
    if (fputs (msgbuf.mtext, stdout) == EOF) {
        break;
    }
}

return 0;
}
```

Листинг 8.30. Приемная часть программы работы с очередями сообщений.

Обратим внимание на способ выработки согласованного ключа, а также на то, что, вообще говоря, неизвестно, какой из процессов – родительский или порожденный – создаст очередь, а какой получит уже ассоциированный с ключом идентификатор (вызовы `msgget()` в обоих процессах одинаковы), но на корректность работы программы это не влияет.

Семафоры

Согласно определению стандарта POSIX-2001, семафор – это минимальный примитив синхронизации, служащий основой для более сложных механизмов синхронизации, определенных в прикладной программе.

У семафора есть значение, которое представляется целым числом в диапазоне от 0 до 32767.

Семафоры создаются (функцией `semget()`) и обрабатываются (функцией `semop()`) наборами (массивами), причем операции над наборо-

рами с точки зрения приложений являются атомарными. В рамках групповых операций для любого семафора из набора можно сделать следующее: увеличить значение, уменьшить значение, дождаться обнуления.

Процессы, обладающие соответствующими правами, также могут выполнять различные управляющие действия над семафорами. Для этого служит функция `semctl()`.

Описание перечисленных функций представлено в листинге 8.31.

```
#include <sys/sem.h>
int semget (key_t key, int nsems, int semflg);
int semop (int semid, struct sembuf *sops,
           size_t nsops);
int semctl (int semid, int semnum, int cmd, ...);
```

Листинг 8.31. Описание функций для работы с семафорами.

Структура `semid_ds`, ассоциированная с идентификатором набора семафоров, должна содержать по крайней мере следующие поля.

```
struct ipc_perm sem_perm; /* Данные о правах */
                           /* доступа к набору семафоров */
unsigned short sem_nsems; /* Число семафоров в */
                           /* наборе */
time_t         sem_otime; /* Время последней операции
                           /* semop() */
time_t         sem_ctime; /* Время последнего */
                           /* изменения посредством semctl() */
```

Отдельные семафоры из набора представляются безымянной структурой, состоящей по крайней мере из следующих полей.

```
unsigned short semval; /* Значение семафора */
pid_t         sempid; /* Идентификатор процесса, */
/* выполнившего последнюю операцию над семафором */
unsigned short semncnt; /* Число процессов, */
/* ожидающих увеличения текущего значения семафора */
unsigned short semzcnt; /* Число процессов, */
/* ожидающих обнуления значения семафора */
```

Функция `semget()` аналогична `msgget()`; аргумент `nsems` задает число семафоров в наборе. Структура `semid_ds` инициализируется так же, как `msqid_ds`. Безымянные структуры, соответствующие отдельным семафорам, функцией `semget()` не инициализируются.

Операции, выполняемые посредством функции `semop()`, задаются массивом `sops` с числом элементов `nsops`, состоящим из структур типа `sembuf`, каждая из которых содержит по крайней мере следующие поля.

```
unsigned short sem_num; /* Номер семафора в наборе */
                        /* (нумерация с нуля) */
short         sem_op; /* Запрашиваемая операция над */
                        /* семафором */
short         sem_flg; /* Флаги операции */
```

Операция над семафором определяется значением поля `sem_op`: положительное значение предписывает увеличить значение семафора на указанную величину, отрицательное – уменьшить, нулевое – сравнить с нулем. Вторая операция не может быть успешно выполнена, если в результате значение семафора становится отрицательным, а третья – если значение семафора ненулевое.

Выполнение массива операций с точки зрения пользовательского процесса является неделимым действием. Это значит, во-первых, что если операции выполняются, то только все вместе и, во-вторых, никакой другой процесс не может получить доступ к промежуточному состоянию набора семафоров, когда часть операций из массива уже выполнялась, а другая еще не успела. Операционная система, разумеется, выполняет операции из массива по очереди, причем порядок не оговаривается. Если очередная операция не может быть выполнена, то эффект предыдущих аннулируется, а вызов функции `semop()` приостанавливается (операция с блокировкой) или немедленно завершается неудачей (операция без блокировки). Подчеркнем, что в случае неудачного завершения вызова `semop()` значения всех семафоров в наборе останутся неизменными.

Приведенный в листинге 8.32 массив операций задает уменьшение (с блокировкой) семафора 1 при условии, что значение семафора 0 равно нулю.

```
sembuf [0].sem_num = 1;
sembuf [0].sem_flg = 0;
sembuf [0].sem_op = -2;

sembuf [1].sem_num = 0;
sembuf [1].sem_flg = IPC_NOWAIT;
sembuf [1].sem_op = 0;
```

Листинг 8.32. Пример задания массива операций над семафорами.

Обращаясь к функции `semctl()`, процессы могут получать информацию о состоянии набора семафоров, изменить ряд его характеристик, удалить набор.

Аргументы `semid` (идентификатор набора семафоров) и `semnum` (номер семафора в наборе) определяют объект, над которым выполняется управляющее действие, задаваемое значением аргумента `cmd`. Если объектом является набор, значение `semnum` игнорируется.

Для некоторых действий задействован четвертый аргумент (см. листинг 8.33).

```
union semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
} arg;
```

Листинг 8.33. Описание четвертого (дополнительного) аргумента функции `semctl()`.

Среди допустимых действий – `GETVAL` (получить значение семафора и выдать его в качестве результата) и `SETVAL` (установить значение семафора равным `arg.val`). Имеются и аналогичные групповые действия – `GETALL` (прочитать значения всех семафоров набора и поместить их в массив `arg.array`) и `SETALL` (установить значения всех семафоров набора равными значениям элементов массива). Предусмотрены действия, позволяющие выяснить идентификатор процесса, выполнившего последнюю операцию над семафором (`GETPID`), а также число процессов, ожидающих увеличения/обнуления (`GETNCNT/GETZCNT`) значения семафора (информация о процессах выдается в качестве результата, см. листинг 8.34).

```
val = semctl (semid, semnum, GETVAL);

arg.val = ...;
if (semctl (semid, semnum, SETVAL, arg) == -1)
    ...;

arg.array = (unsigned short *) malloc (nsems *
    sizeof (unsigned short));
err = semctl (semid, 0, GETALL, arg);

for (i = 0; i < nsems; i++) arg.array [i] = ...;
err = semctl (semid, 0, SETALL, arg);
```

```
lpid = semctl (semid, semnum, GETPID);  
ncnt = semctl (semid, semnum, GETNCNT);  
zcnt = semctl (semid, semnum, GETZCNT);
```

Листинг 8.34. Примеры управляющих действий над семафорами.

Наконец, для семафоров, как и для очередей сообщений, определены управляющие команды `IPC_STAT` (получить информацию о состоянии набора семафоров), `IPC_SET` (переустановить характеристики), `IPC_RMID` (удалить набор семафоров), представленные в листинге 8.35.

```
arg.buf = (struct semid_ds *) malloc (sizeof  
    (struct semid_ds);  
err = semctl (semid, 0, IPC_STAT, arg);  
arg.buf->sem_perm.mode = 0644;  
err = semctl (semid, 0, IPC_SET, arg);
```

Листинг 8.35. Дополнительные примеры управляющих действий над семафорами.

В качестве примера использования семафоров рассмотрим известную задачу об обедающих философях. За круглым столом сидит несколько философов. В каждый момент времени каждый из них либо беседует, либо ест. Для еды одновременно требуется две вилки. Поэтому, прежде чем в очередной раз перейти от беседы к приему пищи, философу надо дожидаться, пока освободятся обе вилки — слева и справа от него, и взять их в руки. Немного поев, философ кладет вилки на стол и вновь присоединяется к беседе. Требуется разработать программную модель обеда философов. Главное в этой задаче — корректная дисциплина захвата и освобождения вилок. В самом деле, если, например, каждый из философов одновременно с другими возьмется за вилку, лежащую слева от него, и будет ждать освобождения правой, обед не завершится никогда.

Предлагаемое решение состоит из двух программ. Первая (см. листинг 8.36) реализует процесс-монитор, который порождает набор семафоров (по одному семафору на каждую вилку), устанавливает начальные значения семафоров (занятой вилке будет соответствовать значение 0, свободной — 1), запускает несколько процессов, представляющих философов, указывая место за столом (в качестве одного из аргументов передается число от 1 до `QPN`), ожидает, пока все процессы завершатся (все философы съедят свой обед), и удаляет набор семафоров. Предполагается (для нужд функции `ftok()`), что исходный текст программы находится в файле `phdin.c` (точнее, что такой файл существует).

```
#include <unistd.h>
#include <stdio.h>
#include <sys/sem.h>
#include <sys/wait.h>

/* Программа-монитор обеда философов */

#define QPH 5

#define ARG_SIZE    20
int main (void) {
    int key;      /* Ключ набора семафоров */
    int semid;    /* Идентификатор набора семафоров */
    int no;       /* Номер философа и/или вилки */
    char ssemid [ARG_SIZE], sno [ARG_SIZE],
        sqph [ARG_SIZE];

    /* Создание и инициализация набора семафоров */
    /* (по семафору на вилку) */
    key = ftok ("phdin.c", 'C');
    if ((semid = semget (key, QPH, 0600 | IPC_CREAT))
        < 0) {
        perror ("SEMGET");
        return (1);
    }
    for (no = 0; no < QPH; no++) {
        if (semctl (semid, no, SETVAL, 1) < 0) {
            perror ("SETVAL");
            return (2);
        }
    }

    sprintf (ssemid, "%d", semid);
    sprintf (sqph, "%d", QPH);

    /* Все - к столу */
    for (no = 1; no <= QPH; no++) {
        switch (fork ()) {
            case -1:
                perror ("FORK");
                return (3);
```

```

        case 0:
            sprintf (sno, "%d", no);
            execl ("./phil", "phil", ssemid, sqph,
                sno, (char *) 0);
            perror ("EXEC");
            return (4);
        }
    }

    /* Ожидание завершения обеда */
    for (no = 1; no <= QPH; no++) {
        (void) wait (NULL);
    }

    /* Удаление набора семафоров */
    if (semctl (semid, 0, IPC_RMID) < 0) {
        perror ("SEMCTL");
        return (5);
    }

    return 0;
}

```

Листинг 8.36. Процесс-монитор для обеда философов.

Вторая программа (см. листинг 8.37) описывает обед каждого философа. Философ какое-то время беседует (случайное значение `trnd`), затем пытается взять вилки слева и справа от себя, когда ему это удастся, некоторое время ест (случайное значение `ernd`), после чего освобождает вилки. Так продолжается до тех пор, пока не будет съеден весь обед. Предполагается, что выполнимый файл программы называется `phil`.

```

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/sem.h>

/* Процесс обеда одного философа */

#define ernd (rand () % 3 + 1)
#define trnd (rand () % 5 + 1)
#define FO 15

```

```
int main (int argc, char *argv []) {
    int semid; /* Идентификатор набора семафоров */
    int qph;   /* Число философов */
    int no;    /* Номер философа */
    int t;     /* Время очередного отрезка еды */
               /* или беседы */
    int fo;    /* Время до конца обеда */
    struct sembuf sembuf [2];

    if (argc != 4) {
        fprintf (stderr, "Использование: %s "
            идентификатор_набора_семафоров число_философов"
            "номер_философа \n", argv [0]);
        return (1);
    }

    fo = FO;
    sscanf (argv [1], "%d", &semid);
    sscanf (argv [2], "%d", &qph);
    sscanf (argv [3], "%d", &no);

    /* Выбор вилок */
    sembuf [0].sem_num = no - 1; /* Левая */
    sembuf [0].sem_flg = 0;
    sembuf [1].sem_num = no % qph; /* Правая */
    sembuf [1].sem_flg = 0;

    while (fo > 0) { /* Обед */

        /* Философ говорит */
        printf ("Философ %d беседует\n", no);
        t = trnd; sleep (t); fo -= t;
        /* Пытается взять вилки */
        sembuf [0].sem_op = -1;
        sembuf [1].sem_op = -1;
        if (semop (semid, sembuf, 2) < 0) {
            perror ("SEMOP");
            return (1);
        }

        /* Ест */
    }
}
```



```
printf ("Философ %d ест\n", no);
t = ernd; sleep (t); fo -= t;
/* Отдает вилки */
sembuf [0].sem_op = 1;
sembuf [1].sem_op = 1;
if (semop (semid, sembuf, 2) < 0) {
    perror ("SEMOP");
    return (2);
}
}
printf ("Философ %d закончил обед\n", no);
return 0;
}
```

Листинг 8.37. Программа, описывающая обед одного философа.

Отметим, что возможность выполнения групповых операций над семафорами предельно упростила решение, сделав его прямолинейным, по большому счету нечестным, но зато очевидным образом гарантирующим отсутствие тупиков.

В листинге 8.38 приведен второй вариант решения задачи, предложенный С.В. Самборским. В нем реализованы четыре стратегии захвата вилок, которые сравниваются по результатам моделирования поведения философов в течение нескольких минут. Все стратегии гарантируют отсутствие тупиков, но только две из них, соответствующие опциям `-a` и `-p`, заведомо не позволят ни одному философу умереть от голода из-за невозможности получить обе вилки сразу. (Это свойство «стратегий `-a` и `-p`» является следствием упорядоченности ресурсов.)

```
/* Обедающие философы. Запуск:
mudrecProc [-a | -p | -I -V] [-t число_секунд]
            имя_философа ...
Опции:
-t число_секунд - сколько секунд моделируется
Стратегии захвата вилок:
-a - сначала захватывается вилка с меньшим
    номером;
-I - некорректная (но эффективная) интеллигентная
    стратегия: во время ожидания уже захваченная
    вилка кладется;
-p - сначала захватывается нечетная вилка;
-V - использован групповой захват семафоров.
```

Пример запуска: mudrecProc -p -t 600 A B C D E F G
H I J K L M N O P Q R S T U V W X Y Z

*/

```
static char rcsid[] __attribute__((unused)) = \
"$Id: mudrecProc.c,v 1.7 2003/11/11 13:14:07"
"sambor Exp $";
```

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <fcntl.h>
#include <time.h>
#include <limits.h>
#include <errno.h>
```

```
#include <sys/sem.h>
#include <sys/msg.h>
```

```
union semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
} arg;
```

```
#define max(a,b) ((a)>(b)?(a):(b))
#define min(a,b) ((a)>(b)?(b):(a))
```

```
struct mudrec {
    long num;
    char *name;
    int left_fork, right_fork;
    int eat_time, wait_time, think_time, max_wait_time;
    int count;
};
```

```
int Stop; /* Семафор для синхронизации выхода */
```

```
/* Различные дескрипторы */
```

```
int protokol [2] = {-1, -1};
```

```
#define pFdIn (protokol [1])
#define pFdOut (protokol [0])

int semFork; /* Вилки */
int from_fil; /* Очередь для возврата результатов */

/* Разные алгоритмы захвата вилок */
static void get_forks_simple (struct mudrec *this);
static void get_forks_parity (struct mudrec *this);
static void get_forks_maybe_infinitt_time
    (struct mudrec *this);
static void get_forks_use_groups
    (struct mudrec *this);

/* Используемый метод захвата вилок */
void (*get_forks) (struct mudrec *this) =
    get_forks_simple;

/* Возвращение вилок */
static void put_forks (struct mudrec *this);

/*
 * Философы
 */
void filosof (struct mudrec this) {
    char buffer [LINE_MAX];
    int bytes;

    if (fork ()) return;

    srandom (getpid ()); /* Очень важно для */
        /*процессов, иначе получим одно и то же! */
    random (); random (); random (); random (); random ();
    random (); random (); random (); random (); random ();

    /* Пока семафор Stop не поднят */
    while (!semctl (Stop, 0, GETVAL)) {
        /* Пора подкрепиться */
        {
            int wait_time, tm;

            sprintf (buffer, "%s: хочет есть\n", this.name);
```

```
bytes = write (pFdIn, buffer, strlen (buffer));
tm = time (0);

(*get_forks) (&this);

wait_time = time (0) - tm; /* Сколько времени */
                          /* получали вилки */
this.wait_time += wait_time;
this.max_wait_time = max (wait_time,
    this.max_wait_time);

sprintf (buffer, "%s: ждал вилок %d сек\n",
    this.name, wait_time);
bytes = write (pFdIn, buffer,
    strlen (buffer));
}

/* Может, обед уже закончился? */
if (semctl (Stop, 0, GETVAL)) {
    put_forks (&this);
    break;
}

/* Едим */
{
    int eat_time = random () % 20 + 1;

    sleep (eat_time);

    this.eat_time += eat_time;
    this.count++;
    sprintf (buffer, "%s: ел %d сек\n",
        this.name, eat_time);
    bytes = write (pFdIn, buffer,
        strlen (buffer));
}

/* Отдаем вилки */
put_forks (&this);

if (semctl (Stop, 0, GETVAL)) break;
```

```
        /* Размышляем */
    {
        int think_time = random () % 10 + 1;
        sleep (think_time);

        this.think_time += think_time;
    }
}

sprintf (buffer,"%s: уходит\n", this.name);
bytes = write (pFdIn, buffer, strlen (buffer));

msgsnd (from_fil, &this, sizeof (this), 0);
/* Отослали статистику своего обеда */

_exit (0); /* ВАЖНО (_): Нам не нужны */
        /* преждевременные вызовы cleanup_ipc */
}

/* Кладем вилки одну за другой */
static void put_forks (struct mudrec *this) {
    struct sembuf tmp_buf;

    tmp_buf.sem_flg = 0;
    tmp_buf.sem_op = 1;
    tmp_buf.sem_num = this->left_fork - 1;
    semop (semFork, &tmp_buf, 1);

    tmp_buf.sem_flg = 0;
    tmp_buf.sem_op = 1;
    tmp_buf.sem_num = this->right_fork - 1;
    semop (semFork, &tmp_buf, 1);
}

/* Берем вилки по очереди в порядке номеров */
static void get_forks_simple (struct mudrec *this) {
    struct sembuf tmp_buf;

    int first = min (this->left_fork,
        this->right_fork);
    int last = max (this->left_fork,
        this->right_fork);
```

```
tmp_buf.sem_flg = SEM_UNDO;
tmp_buf.sem_op = -1;
tmp_buf.sem_num = first - 1;
semop (semFork, &tmp_buf, 1);

tmp_buf.sem_flg = SEM_UNDO;
tmp_buf.sem_op = -1;
tmp_buf.sem_num = last - 1;
semop (semFork, &tmp_buf, 1);
}

/* Берем сначала нечетную вилку (если обе */
/* нечетные - то с большим номером) */
static void get_forks_parity (struct mudrec *this) {
    struct sembuf tmp_buf;

    int left = this->left_fork,
        right = this->right_fork;
    int first = max ((left & 1) * 1000 + left,
        (right & 1) * 1000 + right) % 1000;
    int last = min ((left & 1) * 1000 + left,
        (right & 1) * 1000 + right) % 1000;

    tmp_buf.sem_flg = SEM_UNDO;
    tmp_buf.sem_op = -1;
    tmp_buf.sem_num = first - 1;
    semop (semFork, &tmp_buf, 1);

    tmp_buf.sem_flg = SEM_UNDO;
    tmp_buf.sem_op = -1;
    tmp_buf.sem_num = last - 1;
    semop (semFork, &tmp_buf, 1);
}

/* Берем вилки по очереди, в произвольном порядке.
 * Но если вторая вилка не берется сразу, то кладем
 * первую.
 * То есть философ не расходует вилочное время впустую.
 */
static void get_forks_maybe_infinite_time
    (struct mudrec *this) {
```

```
struct sembuf tmp_buf;
int left = this->left_fork,
    right = this->right_fork;

for (;;) {
    tmp_buf.sem_flg = SEM_UNDO; /* Первую вилку */
                                /* берем с ожиданием */
    tmp_buf.sem_op = -1;
    tmp_buf.sem_num = left - 1;
    semop (semFork, &tmp_buf, 1);

    tmp_buf.sem_flg = SEM_UNDO | IPC_NOWAIT;
    /* Вторую - без ожидания */
    tmp_buf.sem_op = -1;
    tmp_buf.sem_num = right - 1;

    if (0 == semop (semFork, &tmp_buf, 1)) return;
    /* Успех */

    tmp_buf.sem_flg = 0; /* Неуспех: возвращаем */
                        /* первую вилку */
    tmp_buf.sem_op = 1;
    tmp_buf.sem_num = left - 1;
    semop(semFork,&tmp_buf,1);

    tmp_buf.sem_flg = SEM_UNDO; /* Отдав первую, */
                                /* ждем вторую */
    tmp_buf.sem_op = -1;
    tmp_buf.sem_num = right - 1;
    semop (semFork, &tmp_buf, 1);

    tmp_buf.sem_flg = SEM_UNDO | IPC_NOWAIT;
    /* Берем первую вилку без ожидания */
    tmp_buf.sem_op = -1;
    tmp_buf.sem_num = left - 1;

    if (0 == semop (semFork, &tmp_buf, 1)) return;
    /* Успех */

    tmp_buf.sem_flg = 0; /* Неуспех: отдаем */
                        /* вторую вилку, */
    tmp_buf.sem_op = 1; /* чтобы ждать первую */
}
```

```
        tmp_buf.sem_num = right - 1;
        semop (semFork, &tmp_buf, 1);
    }
}

/* Хватаем обе вилки сразу, используя групповые */
/* операции */
static void get_forks_use_groups (struct mudrec
    *this) {
    struct sembuf tmp_buf [2];

    tmp_buf[0].sem_flg = SEM_UNDO;
    tmp_buf[0].sem_op = -1;
    tmp_buf[0].sem_num = this->left_fork - 1;
    tmp_buf[1].sem_flg = SEM_UNDO;
    tmp_buf[1].sem_op = -1;
    tmp_buf[1].sem_num = this->right_fork - 1;
    semop (semFork, tmp_buf, 2);
}

/*
 * Мелкие служебные функции.
 */
static void stop (int dummy) {
    struct sembuf tmp_buf;

    tmp_buf.sem_flg = 0;
    tmp_buf.sem_op = 1;
    tmp_buf.sem_num = 0;
    semop (Stop, &tmp_buf, 1);
}

void cleanup_ipc (void) {
    /*
     * Уничтожение семафоров.
     */
    semctl (semFork, 1, IPC_RMID);
    semctl (Stop, 1, IPC_RMID);

    /* То же с очередью */
    msgctl (from_fil, IPC_RMID, NULL);
}
```



```
}
static void usage (char name []) {
    fprintf (stderr, "Использование: %s [-a | -p | -I | -V]"
        " [-t число_секунд] имя_философа ...\n", name);
    exit (1);
}

/*
 * Точка входа демонстрационной программы.
 */
int main (int argc, char *argv[]) {
    char buffer [LINE_MAX], *p;
    int i, n, c;
    int open_room_time = 300;
    union semun tmp_arg;
    int nMudr;
    struct sigaction sact;

    while ((c = getopt (argc, argv, "apIVt:")) != -1) {
        switch (c) {
            case 'a': get_forks = get_forks_simple;
                       break;
            case 'p': get_forks = get_forks_parity;
                       break;
            case 'I': get_forks =
                get_forks_maybe_infinitt_time; break;
            case 'V': get_forks = get_forks_use_groups;
                       break;
            case 't': open_room_time = strtol (optarg,
                &p, 0);
            if (optarg [0] == 0 || *p != 0) usage
                (argv [0]);
            break;
            default: usage (argv [0]);
        }
    }

    nMudr = argc - optind;
    if (nMudr < 2) usage (argv [0]); /* Меньше двух */
                                     /* философов неинтересно ... */

    /* Создание канала для протокола обработки событий */
```

```
pipe (protokol);
/* Создадим семафоры для охраны вилок */

semFork = semget (ftok (argv [0], 2), nMudr,
    IPC_CREAT | 0777);
tmp_arg.val = 1;
for (i=1; i <= nMudr; i++)
    semctl (semFork, i - 1, SETVAL, tmp_arg);
/* Начальное значение 1 */

/* Прежде чем впускать философов, обеспечим */
/* окончание обеда */
Stop = semget (ftok (argv [0], 3), 1,
    IPC_CREAT | 0777);
tmp_arg.val = 0;
semctl (Stop, 0, SETVAL, tmp_arg); /* Начальное */
/* значение 0 */

/* Очередь для возврата результатов */
from_fil = msgget (ftok (argv [0], 4),
    IPC_CREAT | 0777);

atexit (cleanup_ipc); /* Запланировали */
/* уничтожение семафоров и других средств */
/* межпроцессного взаимодействия */
/*
* Философы входят в столовую
*/
for (i = 0; i < nMudr; i++, optind++) {
    struct mudrec next;

    memset (&next, 0, sizeof (next));

    next.num = i + 1;          /* Номер */
    next.name = argv [optind]; /* Имя */

    /* Указали, какими вилками пользоваться */
    next.left_fork = i + 1;
    next.right_fork = i + 2;
    if (i == nMudr - 1)
        next.right_fork = 1; /* Последний пользуется */
        /* вилкой первого */
}
```

```
    filosof (next);
}

/* Зададим реакцию на сигналы и установим */
/* будильник на конец обеда */
sact.sa_handler = stop;
(void) sigemptyset (&sact.sa_mask);
sact.sa_flags = 0;
(void) sigaction (SIGINT, &sact,
    (struct sigaction *) NULL);
(void) sigaction (SIGALRM, &sact,
    (struct sigaction *) NULL);

alarm (open_room_time);

/*
* Выдача сообщений на стандартный вывод и выход
* после окончания обеда.
*/
close (pFdIn); /* Сами должны закрыть, иначе */
               /* из цикла не выйдем! */
for (;;) {
    n = read (pFdOut, buffer, LINE_MAX);
    if ((n == 0) || ((n == -1) &&
        (errno != EINTR))) break;
    for (i = 0; i < n; i++) putchar (buffer [i]);
}
close (pFdOut);

/* Распечатали сводную информацию */
{
    int full_eating_time = 0;
    int full_waiting_time = 0;
    int full_thinking_time = 0;
    for (i = 1; i <= nMudr; i++) {
        struct mudrec this;
        /* Получили статистику обеда */
        msgrcv (from_fil, &this, sizeof (this), i, 0);
        /* За счет i получаем строго по порядку */
        full_eating_time += this.eat_time;
        full_waiting_time += this.wait_time;
        full_thinking_time += this.think_time;
    }
}
```

```
if (this.count > 0) {
    float count = this.count;
    float think_time = this.think_time / count;
    float eat_time = this.eat_time / count;
    float wait_time = this.wait_time / count;

    printf ("%s: ел %d раз в среднем:"
           " думал=%.1f ел=%.1f ждал=%.1f"
           "(максимум %d)\n", this.name,
           this.count, think_time, eat_time,
           wait_time, this.max_wait_time);
}
else
    printf ("%s: не поел\n", this.name);
}
{
    float total_time = (full_eating_time +
                       full_waiting_time + full_thinking_time)
                       / (float)nMudr;
    printf (" Среднее число одновременно"
           " едящих = %.3f\n Среднее число"
           " одновременно ждущих = %.3f\n",
           full_eating_time / total_time,
           full_waiting_time / total_time);
}
}

/* Сообщим об окончании работы */
printf ("Конец обеда\n");

return 0;
}
```

Листинг 8.38. Второй вариант решения задачи об обедающих философях.

Получит ли в конце концов философ вилки при групповых операциях (опция -V), зависит от реализации. Может случиться так, что хотя бы одна из них в каждый момент времени будет в руках у одного из соседей. То же верно и для «интеллигентной» стратегии (опция -I). Тем не менее, результаты моделирования показывают, что на практике две последние стратегии эффективнее в смысле минимизации времени ожидания вилок.

Отметим небольшие терминологические различия в двух приведенных вариантах решения задачи об обедающих философах. Во втором варианте явно выделены начальные и конечные моделируемые события — вход философов в столовую и выход из нее (в первом варианте они просто сидят за столом).

С методической точки зрения второй вариант интересен тем, что в нем использованы все рассмотренные нами средства межпроцессного взаимодействия — каналы, сигналы, очереди сообщений и, конечно, семафоры. (Тонкость: флаг `SEM_UNDO` обеспечивает корректировку значения семафора при завершении процесса.)

В листинге 8.39 приведена статистика поведения пяти философов для всех четырех стратегий при времени моделирования 100 секунд. Эти результаты говорят в пользу групповых операций над семафорами.

-а:

A: ел 2 раза	в среднем:	думал=3.5	ел=11.5
		ждал=36.5	(максимум 73)
B: ел 3 раза	в среднем:	думал=5.7	ел=7.7
		ждал=20.0	(максимум 41)
C: ел 3 раза	в среднем:	думал=5.7	ел=11.3
		ждал=17.0	(максимум 33)
D: ел 3 раза	в среднем:	думал=1.7	ел=16.7
		ждал=15.7	(максимум 19)
E: ел 1 раз	в среднем:	думал=10.0	ел=20.0
		ждал=73.0	(максимум 41)

Среднее число одновременно едящих = 1.471

Среднее число одновременно ждущих = 2.980

-р:

A: ел 3 раза	в среднем:	думал=3.7	ел=15.3
		ждал=16.0	(максимум 34)
B: ел 4 раза	в среднем:	думал=5.0	ел=13.8
		ждал=8.2	(максимум 15)
C: ел 3 раза	в среднем:	думал=6.7	ел=3.7
		ждал=25.7	(максимум 27)
D: ел 4 раза	в среднем:	думал=5.8	ел=8.5
		ждал=13.8	(максимум 28)
E: ел 3 раза	в среднем:	думал=5.3	ел=15.3
		ждал=16.7	(максимум 29)

Среднее число одновременно едящих = 1.761

Среднее число одновременно ждущих = 2.413

-I:

A: ел 5 раз	в среднем:	думал=4.2	ел=9.4
		ждал=6.6	(максимум 15)
B: ел 3 раза	в среднем:	думал=6.3	ел=10.3
		ждал=17.0	(максимум 31)
C: ел 4 раза	в среднем:	думал=6.8	ел=7.0
		ждал=12.2	(максимум 45)
D: ел 3 раза	в среднем:	думал=4.3	ел=16.0
		ждал=13.0	(максимум 16)
E: ел 4 раза	в среднем:	думал=5.8	ел=8.5
		ждал=10.8	(максимум 22)

Среднее число одновременно едящих = 1.858

Среднее число одновременно ждущих = 2.125

-V:

A: ел 5 раз	в среднем:	думал=5.6	ел=5.6
		ждал=8.8	(максимум 17)
B: ел 3 раза	в среднем:	думал=6.3	ел=10.3
		ждал=16.7	(максимум 20)
C: ел 4 раза	в среднем:	думал=4.8	ел=11.0
		ждал=9.8	(максимум 18)
D: ел 4 раза	в среднем:	думал=5.2	ел=12.0
		ждал=8.8	(максимум 15)
E: ел 4 раза	в среднем:	думал=5.2	ел=10.5
		ждал=10.2	(максимум 20)

Среднее число одновременно едящих = 1.892

Среднее число одновременно ждущих = 2.049

Листинг 8.39. Результаты моделирования поведения философов.

Разделяемые сегменты памяти

В стандарте POSIX-2001 разделяемый объект памяти определяется как объект, представляющий собой память, которая может быть параллельно отображена в адресное пространство более чем одного процесса.

Таким образом, процессы могут иметь общие области виртуальной памяти и разделять содержащиеся в них данные. Единицей разделяемой памяти являются сегменты. Разделение памяти обеспечивает наиболее быстрый обмен данными между процессами.

Работа с разделяемой памятью начинается с того, что один из взаимодействующих процессов посредством функции `shmget()` создает разделяемый сегмент, специфицируя первоначальные права доступа к нему и его размер в байтах.

Чтобы получить доступ к разделяемому сегменту, его нужно присоединить (для этого служит функция `shmat()`), т. е. разместить сегмент в виртуальном пространстве процесса. После присоединения, в соответствии с правами доступа, процессы могут читать данные из сегмента и записывать их (быть может, синхронизируя свои действия с помощью семафоров). Когда разделяемый сегмент становится ненужным, его следует отсоединить с помощью функции `shmdt()`.

Предусмотрена возможность выполнения управляющих действий над разделяемыми сегментами (функция `shmctl()`).

Описание перечисленных функций представлено в листинге 8.40.

```
#include <sys/shm.h>
int shmget (key_t key, size_t size, int shmflg);
void *shmat (int shmid, const void *shmaddr,
             int shmflg);
int shmdt (const void *shmaddr);
int shmctl (int shmid, int cmd,
            struct shmid_ds *buf);
```

Листинг 8.40. Описание функций для работы с разделяемыми сегментами памяти.

Структура `shmid_ds`, ассоциированная с идентификатором разделяемого сегмента памяти, должна содержать по крайней мере следующие поля.

```
struct ipc_perm shm_perm; /* Данные о правах */
                          /* доступа к разделяемому сегменту */
size_t    shm_segsz; /* Размер сегмента в байтах */
pid_t     shm_lpid; /* Идентификатор процесса, */
                          /* выполнившего последнюю операцию */
                          /* над разделяемым сегментом */
pid_t     shm_cpid; /* Идентификатор процесса, */
                          /* создавшего разделяемый сегмент */
shmatt_t  shm_nattch; /* Текущее число */
                          /* присоединений сегмента */
time_t    shm_atime; /* Время последнего */
                          /* присоединения */
time_t    shm_dtime; /* Время последнего */
                          /* отсоединения */
time_t    shm_ctime; /* Время последнего изменения */
                          /* посредством shmctl() */
```

Функция `shmget()` аналогична `msgget()` и `semget()`; аргумент `size` задает нижнюю границу размера сегмента в байтах; реализация, учитывающая, например, правила выравнивания, имеет право создать разделяемый сегмент большего размера.

Структура `shmid_ds` инициализируется в соответствии с общими для средств межпроцессного взаимодействия правилами. Поле `shm_segsz` устанавливается равным значению аргумента `size`.

Число уникальных идентификаторов разделяемых сегментов памяти ограничено; попытка его превышения ведет к неудачному завершению `shmget()` (возвращается `-1`). Вызов `shmget()` завершится неудачей и тогда, когда значение аргумента `size` меньше минимально допустимого либо больше максимально допустимого размера разделяемого сегмента.

Чтобы присоединить разделяемый сегмент, используется функция `shmat()`. Аргумент `shmid` задает идентификатор разделяемого сегмента; аргумент `shmataddr` — адрес, по которому сегмент должен быть присоединен, т. е. тот адрес в виртуальном пространстве процесса, который получит начало сегмента. Поскольку свойства сегментов зависят от аппаратных особенностей управления памятью, не всякий адрес является приемлемым. Если установлен флаг `SHM_RND`, адрес присоединения округляется до величины, кратной константе `SHMLBA`.

Если `shmataddr` задан как пустой указатель, реализация выбирает адрес присоединения по своему усмотрению.

По умолчанию присоединяемый сегмент будет доступен и на чтение, и на запись (если процесс обладает необходимыми правами). Флаг `SHM_RDONLY` предписывает присоединить сегмент только для чтения.

При успешном завершении функции `shmat()` результат равен адресу, который получил присоединенный сегмент; в случае неудачи возвращается `-1`. (Разумеется, для использования результата `shmat()` в качестве указателя его нужно преобразовать к требуемому типу.)

Отсоединение сегментов производится функцией `shmdt()`; аргумент `shmataddr` задает начальный адрес отсоединяемого сегмента.

Управление разделяемыми сегментами осуществляется при помощи функции `shmctl()`, аналогичной `msgctl()`. Как и для очередей сообщений, для разделяемых сегментов определены управляющие команды `IPC_STAT` (получить информацию о состоянии разделяемого сегмента), `IPC_SET` (переустановить характеристики), `IPC_RMID` (удалить разделяемый сегмент). Удалять сегмент нужно после того, как от него отсоединились все процессы.

Аппарат разделяемых сегментов предоставляет нескольким процессам возможность одновременного доступа к общей области памяти. Обеспечивая корректность доступа, процессы тем или иным способом должны синхронизировать свои действия. В качестве средства синхронизации

удобно использовать семафор. В листинге 8.41 показана реализация так называемого критического интервала — механизма, обеспечивающего взаимное исключение разделяющих общие данные процессов.

Для «создания» подобного механизма необходимо породить разделяемый сегмент памяти, присоединить его во всех процессах, которым предоставляется доступ к разделяемым данным, а также породить и проинициализировать простейший семафор. После этого монопольный доступ к разделяемой структуре обеспечивается применением P- и V-операций.

```
#include <unistd.h>
#include <stdio.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <sys/wait.h>

int main (void) {
    struct region {
        pid_t fpid;
    } *shm_ptr;

    struct sembuf P = {0, -1, 0};
    struct sembuf V = {0, 1, 0};

    int shmid;
    int semid;

    shmid = shmget (IPC_PRIVATE, sizeof
        (struct region), 0777);
    semid = semget (IPC_PRIVATE, 1, 0777);
    (void) semctl (semid, 0, SETVAL, 1);

    switch (fork ()) {
        case -1:
            perror ("FORK");
            return (1);
        case 0:
            if ((int) (shm_ptr = (struct region *)
                shmatt (shmid, NULL, 0)) == (-1)) {
                perror ("CHILD-SHMAT");
                return (2);
            }
    }
```

```
if (semop (semid, &P, 1) != 0) {
    perror ("CHILD-SEMOP-P");
    return (3);
}
printf ("Процесс-потомок вошел в критический"
        "интервал\n");
shm_ptr->fpid = getpid (); /* Монопольный */
                          /* доступ */
printf ("Процесс-потомок перед выходом из"
        "критического интервала\n");
if (semop (semid, &V, 1) != 0) {
    perror ("CHILD-SEMOP-V");
    return (4);
}
(void) shmdt (shm_ptr);
return 0;
}
if ((int) (shm_ptr = (struct region *) shmat
          (shmid, NULL, 0)) == (-1)) {
    perror ("PARENT-SHMAT");
    return (2);
}
if (semop (semid, &P, 1) != 0) {
    perror ("PARENT-SEMOP-P");
    return (3);
}
printf ("Родительский процесс вошел в"
        "критический интервал\n");

shm_ptr->fpid = getpid (); /* Монопольный доступ */

printf ("Родительский процесс перед выходом из"
        "критического интервала\n");
if (semop (semid, &V, 1) != 0) {
    perror ("PARENT-SEMOP-V");
    return (4);
}

(void) wait (NULL);

printf ("Идентификатор родительского процесса:"
        "%d\n", getpid ());
```

```

printf ("Идентификатор процесса в разделяемой"
        "структуре: %d\n", shm_ptr->fpid);
(void) shmdt (shm_ptr);

(void) semctl (semid, 1, IPC_RMID);
(void) shmctl (shmid, IPC_RMID, NULL);

return 0;
}

```

Листинг 8.41. Пример работы с разделяемыми сегментами памяти.

Результат работы приведенной программы может выглядеть так, как показано в листинге 8.42.

```

Родительский процесс вошел в критический интервал
Родительский процесс перед выходом из критического
интервала
Процесс-потомок вошел в критический интервал
Процесс-потомок перед выходом из критического
интервала
Идентификатор родительского процесса: 2161
Идентификатор процесса в разделяемой структуре: 2162

```

Листинг 8.42. Возможный результат синхронизации доступа к разделяемым данным.

В листинге 8.43 представлен пример использования разделяемых сегментов памяти в сочетании с обработкой сигнала SIGSEGV, который посылается процессу при некорректном обращении к памяти. Идея в том, чтобы создавать разделяемые сегменты, «накрывающие» запрашиваемые адреса. При некотором воображении пример можно считать основой программной реализации виртуальной памяти.

```

/* * * * * * * * * * * * * * * * * * * * * * * */
/* Реализация "виртуальной" памяти из одного */
/* сегмента. Используются разделяемые сегменты */
/* памяти и обработка сигнала SIGSEGV */
/* * * * * * * * * * * * * * * * * * * * * * */
#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>

```

```
#include <sys/shm.h>
#include <signal.h>

/* Константа, зависящая от реализации */
#define SHM_BASE_ADDR 0x40014000
static int shm_id = -1;
static void *shm_addr;

/* Реакция на сигнал SIGSEGV. */
/* Создаем и присоединяем на чтение разделяемый */
/* сегмент, */
/* накрывающий переданный адрес. */
/* Если это не помогло, переприсоединяем сегмент */
/* на запись */
static void sigsegv_sigaction (int sig,
    siginfo_t *sig_info, void *addr) {
    struct shmid_ds shmid_ds;
    if (shm_id == -1) {
        /* Сегмента еще нет. Создадим */
        if ((shm_id = shmget (IPC_PRIVATE, SHMLBA,
            S_IRUSR)) == -1) {
            perror ("SHMGET");
            exit (1);
        }
        /* Присоединим сегмент на чтение */
        if ((int) (shm_addr = shmat (shm_id,
            sig_info->si_addr, SHM_RDONLY | SHM_RND))
            == (-1)) {
            perror ("SHMAT-RDONLY");
            exit (2);
        }
        return;
    } else {
        /* Сегмент уже есть, но обращение по адресу */
        /* вызвало сигнал SIGSEGV. */
        /* Значит, это была попытка записи, и сегмент */
        /* нужно переприсоединить на запись, поменяв */
        /* соответственно режим доступа */
        if (shmctl (shm_id, IPC_STAT, &shmid_ds) == -1) {
            perror ("SHMCTL-IPC_STAT");
            exit (3);
        }
    }
}
```

```
    shmid_ds.shm_perm.mode |= S_IWUSR;
    if (shmctl (shm_id, IPC_SET, &shmid_ds) == -1) {
        perror ("SHMCTL-IPC_SET");
        exit (4);
    }
    (void) shmdt (shm_addr);
    if (shmat (shm_id, shm_addr, 0) != shm_addr) {
        perror ("SHMAT-RDWD");
        exit (5);
    }
}

int main (void) {
    char *test_ptr;
    struct sigaction sact;

    /* Установим реакцию на сигнал SIGSEGV */
    (void) sigemptyset (&sact.sa_mask);
    sact.sa_flags = SA_SIGINFO;
    sact.sa_sigaction = sigsegv_sigaction;
    (void) sigaction (SIGSEGV, &sact,
        (struct sigaction *) NULL);
    /* Убедимся, что разделяемые сегменты */
    /* инициализируются нулями */
    test_ptr = (char *) (SHM_BASE_ADDR + 3);
    printf ("Результат попытки чтения до записи:"
        "%x\n", *test_ptr);
    /* Попробуем записать */
    *test_ptr = 'A';
    printf ("Результат попытки чтения после записи:"
        "%x\n", *test_ptr);
    return (shmctl (shm_id, IPC_RMID, NULL));
}
```

Листинг 8.43. Пример работы с разделяемыми сегментами памяти и сигналами.

Обратим внимание на использование флагов округления адреса присоединения разделяемого сегмента (SHM_RND) и присоединения только на чтение (SHM_RDONLY), а также обработчика сигналов, задаваемого полем sa_sigaction структуры типа sigaction (в сочетании с флагом SA_SIGINFO) и имеющего доступ к расширенной информации о сигнале и его причинах.

Вариант 1

1. **Согласно стандарту POSIX-2001, функция `pipe()` выполняет следующие действия**
 - ☐ порождает процесс
 - ☐ обеспечивает выполнение заданной команды
 - ☐ создает канал
2. **Согласно стандарту POSIX-2001, при приеме функцией `msgrcv()` сообщения выбираются из очереди**
 - ☐ в порядке, зависящем от реализации
 - ☐ в порядке поступления в очередь
 - ☐ в порядке поступления с учетом типа
3. **Согласно стандарту POSIX-2001, управляющие действия, выполняемые функцией `semctl()`, позволяют**
 - ☐ создать набор семафоров
 - ☐ изменить число семафоров в наборе
 - ☐ удалить набор семафоров

Вариант 2

1. **Согласно стандарту POSIX-2001, функция `pipe()` выполняет следующие действия**
 - ☐ порождает процесс
 - ☐ обеспечивает выполнение заданной команды
 - ☐ создает канал

2. **Согласно стандарту POSIX-2001, функция `msgctl()` позволяет изменить**
 - ☐ идентификатор владельца
 - ☐ идентификатор создателя
 - ☐ максимальный размер сообщения

3. **Согласно стандарту POSIX-2001, процесс может увеличить максимально допустимый суммарный размер сообщений в очереди, если**
 - ☐ действующий идентификатор процесса совпадает с идентификатором владельца
 - ☐ действующий идентификатор процесса совпадает с идентификатором создателя
 - ☐ процесс обладает соответствующими привилегиями

Вариант 3

1. **Согласно стандарту POSIX-2001, функция `system()` выполняет следующие действия**
 - ☐ порождает процесс
 - ☐ обеспечивает выполнение заданной команды
 - ☐ создает канал

2. **Согласно стандарту POSIX-2001, тип сообщения может быть**
 - ☐ отрицательным
 - ☐ нулевым
 - ☐ положительным

3. **Согласно стандарту POSIX-2001, процесс может увеличить число семафоров в наборе, если**
 - ☐ действующий идентификатор процесса совпадает с идентификатором владельца
 - ☐ действующий идентификатор процесса совпадает с идентификатором создателя
 - ☐ процесс обладает соответствующими привилегиями

Лекция 9. Общий терминальный интерфейс

Анализируются основные понятия и объекты общего терминального интерфейса, служебные программы и функции для управления терминалами, особенности терминального ввода/вывода, псевдотерминалы.

Ключевые слова: терминал, терминальное устройство, полнодуплексный режим, псевдотерминал, главное устройство, подчиненное устройство, буферизация, очередь ввода, очередь вывода, режимы ввода, канонический режим ввода, неканонический режим ввода, режимы вывода, управляющие режимы, локальные режимы, специальный управляющий символ, скорость передачи, эхоотображение, контроль четности, сигнал прерывания, перевод строки, возврат каретки, забой, уничтожение строки, старт/стопное управление вводом, старт/стопное управление выводом, разрыв соединения, имя терминала, характеристика терминала, мультиплексирование ввода/вывода, управляющий терминал.

Основные понятия и объекты

В стандарте POSIX-2001 терминал или терминальное устройство определяется как символьный специальный файл, удовлетворяющий спецификациям общего терминального интерфейса.

Наряду с физическими устройствами в стандарте рассматриваются псевдотерминалы – сущности, поддерживающие интерфейс, идентичный терминальному. Псевдотерминал состоит из двух «устройств»: главного и подчиненного. Подчиненное «устройство» предоставляет процессам терминальный интерфейс, не опирающийся на прямую аппаратную поддержку. Данные, которые записываются на главное устройство, становятся входными для подчиненного и наоборот.

Обычно терминальное устройство работает в полнодуплексном режиме, когда ввод и вывод могут совмещаться во времени.

С каждым терминальным устройством ассоциирована очередь ввода, куда система помещает входные данные до того, как их прочитают прикладные процессы. На размер этой очереди (в байтах) может быть наложено ограничение `MAX_INPUT`. Поддерживается также очередь вывода, где хранятся записанные прикладными процессами, но еще не выведенные на терминал символы.

Ввод может происходить в каноническом и неканоническом режимах. Канонический режим означает построчную буферизацию ввода системой,

т. е. запрос на чтение из прикладной программы будет удовлетворен лишь после того, как с клавиатуры поступит символ перевода строки или конца файла, а прочитает программа заведомо не больше одной строки, независимо от того, сколько байт она запросила. На размер строки может быть наложено ограничение `{MAX_CANON}`. Канонический режим подразумевает также естественную обработку системой символов забота и уничтожения строки.

(Отметим, что в каноническом режиме приложение не обязано сразу прочитать всю буферизованную строку. Можно запросить любое количество байт (даже один), и данные не будут потеряны.)

В неканоническом режиме входные данные не подвергаются пре-процессированию системой, а обработка запроса на чтение зависит от двух параметров — `MIN` и `TIME`. Запрос на чтение не будет удовлетворен, пока не поступит по крайней мере `MIN` байт или не истечет время задержки `TIME` (время задается в десятых долях секунды). Нулевое значение `TIME` трактуется как бесконечная задержка.

Более точно, если `MIN > 0`, `TIME` трактуется как задержка между поступлениями байт; следовательно, отсчет времени начинается после прихода очередного байта. Если `MIN = 0`, `TIME` означает общее время обслуживания запроса на чтение. Такой подход позволяет эффективно читать во время всплеск активности ввода и не препятствует побайтному вводу.

Помимо режима, канонического или нет, на передачу данных читающему процессу оказывает влияние флаг `O_NONBLOCK`, устанавливаемый функциями `open()` или `fcntl()`, а на обработку входных символов — режимы ввода и локальные режимы. Подобная обработка может включать, например, эхоотображение вводимых символов.

Подвергаются обработке системой (в соответствии с режимами вывода) и выводимые прикладной программой символы (в частности, они могут буферизоваться).

Ряд управляющих символов играет специальную роль при вводе и/или выводе. Кратко опишем эти функции, не уточняя их соответствия с нажатиями на клавиатуре, поскольку оно зависит от реализации. Отметим, что за некоторыми очевидными исключениями (например, перевод строки), специальные управляющие символы не передаются читающему процессу.

`INTR`

Генерирует сигнал прерывания (`SIGINT`), посылаемый всем процессам, для которых данный терминал является управляющим.

QUIT

Генерирует сигнал выхода.

ERASE

При каноническом режиме ввода устраняет предыдущий символ, но не далее начала строки.

KILL

При каноническом режиме ввода уничтожает всю строку.

EOF

При каноническом режиме ввода при получении этого символа все буферизованные байты передаются процессу, а сам символ EOF отбрасывается. Таким образом, если буферизованных байт нет, т. е. EOF встретился в начале строки, в процесс будет передано нуль байт, что и является стандартным обозначением конца файла.

NL

Стандартный разделитель строк (перевод строки) при каноническом режиме ввода. Его нельзя изменить.

EOL

Дополнительный разделитель строк, аналогичный NL, при каноническом режиме ввода. Обычно не используется.

SUSP

Генерирует сигнал остановки.

STOP

Специальный символ как при вводе, так и при выводе, распознаваемый в случае наличия флагов IXON (управление выводом) или IXOFF (управление вводом). Обычно используется для временной приостановки вывода, когда нужно прочитать текст на экране терминала.

START

Употребляется для возобновления вывода, приостановленного с помощью символа STOP.

CR

При каноническом режиме ввода и выполнении некоторых дополнительных условий — эквивалент перевода строки.

Центральную роль в управлении терминалами играет структура `termios`, определенная во включаемом файле `<termios.h>`. Она должна содержать по крайней мере следующие поля.

```
tcflag_t c_iflag;      /* Режимы ввода */
tcflag_t c_oflag;      /* Режимы вывода */
tcflag_t c_cflag;      /* Управляющие режимы */
tcflag_t c_lflag;      /* Локальные режимы */
cc_t     c_cc [NCCS];  /* Специальные управляющие */
                        /* символы */
```

Типы `tcflag_t`, `cc_t` и фигурирующий далее `speed_t` должны определяться реализацией посредством `typedef` как беззнаковые целые.

Обращение к элементам массива `c_cc`, хранящего специальные управляющие символы, которые могут быть изменены, выполняется с помощью индексов с именами, полученными вставкой буквы V перед названием символа: VEOF, VEOL, VERASE, VINTR, VKILL, VQUIT, VSTART, VSTOP, VSUSP. Кроме того, еще два индекса, VMIN и VTIME, используются для работы со значениями MIN и TIME и могут совпадать с VEOF и VEOL, соответственно (поскольку символы EOF и EOL нужны только в каноническом режиме, а значения MIN и TIME — только в неканоническом).

Поле `c_iflag` структуры `termios` описывает основные параметры терминального ввода.

BRKINT

При разрыве соединения (когда в течение времени, превышающего длительность передачи байта, поступают нулевые биты) генерировать сигнал прерывания и сбрасывать очереди ввода/вывода.

ICRNL

Преобразовывать возврат каретки в перевод строки.

IGNBRK

Игнорировать разрыв соединения.

IGNCR

Игнорировать возврат каретки.

IGNPAR

Игнорировать символы с ошибками четности.

INLCR

Преобразовывать перевод строки в возврат каретки.

INPCK

Разрешить контроль четности.

ISTRIP

Отбрасывать старший бит, обрезая байты до семи бит.

IXOFF

Разрешить старт/стопное управление вводом.

IXON

Разрешить старт/стопное управление выводом.

PARMRK

Отмечать ошибки четности.

Поле `c_oflag` определяет системную обработку вывода. К числу обязательных для поддержки стандарт POSIX-2001 относит только один флаг — `OPOST` (постпроцессировать вывод). В расширение XSI входят флаги, определяющие характер постпроцессирования: `ONLCR` (преобразовывать перевод строки в пару — перевод строки, возврат каретки), `OCRNL` (преобразовывать возврат каретки в перевод строки), `NLDLY` (выбрать задержку для перевода строки) и т.п.

Поле управляющих режимов `c_cflag` описывает аппаратные характеристики линии и терминала: размер символа в битах (`CSIZE`: от `CS5` — 5 бит до `CS8` — 8 бит), число стоп-бит (`CSTOPB`: два стоп-бита), освобождение линии при закрытии последнего файлового дескриптора, ассоциированного с терминалом (`HUPCL`), контроль четности (`PARENB`: контроль включен; `PARODD`: проверка на нечетность) и т.п.

Стандарт осторожен в части представления скорости передачи. Оговаривается только, что скорость хранится в структуре `termios` как значение типа `speed_t`, но не утверждается, что она представлена как часть поля `c_cflag` (хотя в исторически сложившихся реализациях это так). Допустимые значения скорости задаются именованными константами: от `B0` — нулевой, означающей разрыв соединения, до `B38400` — 38400 бит/сек.

Поле локальных режимов `c_lflag` структуры `termios` используется для управления различными характеристиками терминала. В их число входят:

`ECNO`

Включить эхоотображение.

`ECNOE`

Отображать символ забоя как тройку (возврат на шаг, пробел, возврат на шаг).

`ECNOK`

Отображать символ уничтожения строки (по крайней мере путем выдачи перевода строки).

`ECNONL`

Включить эхоотображение перевода строки.

`ICANON`

Канонический режим ввода.

`ISIG`

Разрешить сигналы. Если установлен флаг `ISIG`, то каждый вводимый символ сравнивается со специальными управляющими символами

INTR, QUIT и SUSP. В случае совпадения выполняется ассоциированная функция.

Служебные программы и функции для управления терминалами

Служебная программа

```
tty
```

позволяет узнать имя пользовательского терминала. Более точно, она выведет на стандартный вывод имя терминала, открытого в качестве стандартного ввода.

Если для примера перенаправить стандартный ввод (см. листинг 9.1), можно получить результат, показанный в листинге 9.2.

```
tty > /tmp/tty.res  
tty < /tmp/tty.res >> /tmp/tty.res
```

Листинг 9.1. Пример использования служебной программы `tty`.

```
/dev/ttyS4  
not a tty
```

Листинг 9.2. Возможный результат использования служебной программы `tty`.

Узнать, ассоциирован ли открытый файловый дескриптор с терминальным устройством, а также получить имя этого устройства можно с помощью функций `isatty()` и `ttyname()` (см. листинг 9.3).

```
#include <unistd.h>  
int isatty (int fildes);  
char *ttyname (int fildes);
```

Листинг 9.3. Описание функций `isatty()` и `ttyname()`.

Если с дескриптором ассоциирован терминал, функция `isatty()` возвращает единицу, а `ttyname()` — указатель на цепочку символов (располагающуюся, быть может, в перезаписываемой каждым вызовом статической области). В противном случае возвращаются, соответственно, нуль и пустой указатель.

Каждый терминал обладает рядом характеристик, которые можно опросить и/или изменить. Для этого служит утилита `stty`:

```
stty [ -a | -g ]
stty характеристика ...
```

Будучи вызванной без опций и операндов, она выдает значения основных характеристик терминала, ассоциированного со стандартным вводом. Смысл опций таков:

`a`

Выдать значение всех установленных характеристик.

`-g`

Выдать текущие установки в формате, который может быть использован в качестве аргумента другой команды `stty`.

В частности, выдача команды

```
stty -a
```

может выглядеть так, как показано в листинге 9.4. Здесь присутствуют как стандартные, так и специфичные для ОС Linux характеристики (минус перед именем характеристики означает, разумеется, что соответствующий флаг не установлен). Например, для опрашиваемого терминала задан канонический режим ввода и постпроцессирование вывода, но отсутствует контроль четности. Можно видеть также, какие символы назначены на роли специальных управляющих.

```
speed 19200 baud; rows 0; columns 0; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U;
eof = ^D; eol = <undef>;
col2 = <undef>; start = ^Q; stop = ^S;
susp = <undef>; rprint = ^R; werase = ^W;
lnext = ^V; flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 hupcl -cstopb cread clocal
-crtscts
-ignbrk -brkint -ignpar -parmrk -inpck -istrip
-inlcr -igncr -icrnl ixon ixoff
-iuclc -ixany -imaxbel
opost -olcuc -ocrnl -onlcr -onocr -onlret -ofill
```



```
-ofdel nl0 cr0 tab0 bs0 vt0
ff0
isig icanon -iexten echo echoe echok -echonl
-noflsh -xcase -tostop -echoprnt
-echoctl echoke
```

Листинг 9.4. Возможный результат команды `stty -a`.

Чтобы переустановить какую-либо характеристику, следует указать ее имя и, если нужно, новое значение. Например, после выполнения команды

```
stty kill '^k'
```

для отмены строки придется нажимать `CTRL+K`. Вслед за выполнением команды

```
stty -echo
```

на экране перестанут отображаться символы, вводимые пользователем, что, правда, не повлияет на выдачу результатов работы команд. Подобный режим применяется для обеспечения секретности, например, во время ввода пароля. Для восстановления эхоотображения советуем воспользоваться командой

```
stty echo
```

Сохранение и восстановление характеристик терминала можно реализовать так, как показано в листинге 9.5.

```
saved="$ (stty -g) "
stty новые_характеристики
. . .
stty $saved
```

Листинг 9.5. Пример сохранения и восстановления характеристик терминала.

Некоторые, хотя и весьма ограниченные, возможности управления терминалами предоставляет служебная программа `tput`:

```
tput [-T тип_терминала] действие
```

Стандарт POSIX-2001 предусматривает всего три действия: `clear` (очистка экрана), `init` (инициализация терминала), `reset` (переустановка терминала). Действие осуществляется путем отправки на терминал соответствующей управляющей последовательности. Стандарт не специфицирует, как эта последовательность определяется, каковы все результаты ее отправки и чем инициализация отличается от переустановки.

Служебная программа `tput` дает хорошие примеры немобильности. Во-первых, она входит в необязательную часть стандарта POSIX-2001. Во-вторых, в общем случае управляющие действия с терминалом могут реализовываться не только путем отправки на него некоторой последовательности символов, но и путем обращения к системным вызовам, поэтому присваивание

```
ClearScreen=`tput clear`
```

может и не достичь цели (`tput` ничего не выдаст). В-третьих, стандартизованный перечень действий слишком ограничен, так что реально можно рекомендовать лишь включение строки вида

```
tput init
```

в разного рода инициализационные файлы. Тем не менее, показанный в листинге 9.6 фрагмент командного файла, выполняющегося при загрузке ОС Linux, демонстрирует поучительные способы борьбы с немобильностью.

```
if [ -x /usr/bin/tput ]; then
    if [ "x`tput kbs`" != "x" ]; then
        stty erase `tput kbs`
    elif [ -x /usr/bin/wc ]; then
        if [ "`tput kbs` | wc -c "` -gt 0 ]; then
            stty erase `tput kbs`
        fi
    fi
fi
..
```

Листинг 9.6. Пример совместного использования утилит `stty` и `tput`.

Здесь задается управляющая последовательность для выполнения функции заоя. Впрочем, несмотря на все проверки, немобильность все-таки осталась, поскольку действие `kbs` не входит в число стандартных.

На уровне функций опрос и изменение характеристик терминала разбиты на два семейства: `tc*()` (см. листинг 9.7) и `cf*()` (см. листинг 9.8), которые в стандарте POSIX введены в качестве замены исторически сложившейся, универсальной, но чрезмерно эклектичной, не вписывающейся ни в какие синтаксические рамки функции `ioctl()`.

```
#include <termios.h>
int tcgetattr (int fildes, struct termios
    *termios_p);
int tcsetattr (int fildes, int optional_actions,
    const struct termios *termios_p);
int tcflow (int fildes, int action);
int tcflush (int fildes, int queue_selector);
int tcdrain (int fildes);
int tcsendbreak (int fildes, int duration);
Листинг 9.7 Описание функций семейства tc*().
#include <termios.h>
speed_t cfgetispeed (const struct termios
    *termios_p);
speed_t cfgetospeed (const struct termios
    *termios_p);
int cfsetispeed (struct termios *termios_p,
    speed_t speed);
int cfsetospeed (struct termios *termios_p,
    speed_t speed);
```

Листинг 9.8. Описание функций семейства `cf*()`.

Пара функций `tcgetattr()/tcsetattr()` позволяет стандартным образом опросить/изменить характеристики терминала, ассоциированного с открытым файловым дескриптором `fildes`. Значения характеристик помещаются в структуру, на которую указывает аргумент `termios_p`. Нюансами изменения ведаёт аргумент `optional_actions`. Если его значение равно `TCSANOW`, изменение выполняется немедленно. Значение `TCSADRAIN` предписывает сначала дождаться передачи на терминальное устройство всех записанных по дескриптору `fildes` данных; это полезно, если изменения затрагивают режимы вывода. Значение `TCSAFLUSH` дополнительно означает предварительный сброс введенных устройством, но не прочитанных прикладными процессами символов.

Отметим, что вызов `tcsetattr()` завершается успешно (и возвращает нулевое значение), если удалось осуществить хотя бы одно из запрошенных изменений характеристик терминала. Это сделано по соображениям

мобильности, поскольку разные реализации могут поддерживать разные подмножества характеристик. После `tcsetattr()` целесообразно снова вызвать `tcgetattr()` и узнать реальное состояние терминального устройства.

Приложение, изменяющее характеристики терминала, обязано позаботиться об их последующем восстановлении (например, при завершении процесса). Завершение должно выполняться аккуратно, с обработкой сигналов.

Функция `tcflow()` позволяет приостановить или возобновить терминальный ввод/вывод. Если значение аргумента `action` равно `TCOOFF`, приостанавливается вывод; по значению `TCOON` он возобновляется. Значение `TCIOFF` вызывает передачу на терминал символа `STOP`, значение `TCION` — символа `START`, что оказывает соответствующее воздействие на ввод.

С помощью функции `tcflush()` можно сбросить очередь ввода (значение `TCIFLUSH` аргумента `queue_selector`), вывода (`TCOFLUSH`) или совместить их (значение `TCIOFLUSH`).

Функция `tcdrain()` действует мягче. Она блокирует вызывающий процесс, пока все данные, записанные по дескриптору `files`, не будут переданы на терминальное устройство, дожидаясь тем самым физического окончания вывода.

Функция `tcsendbreak()` носит технический характер, позволяя организовать разрыв соединения путем послышки на терминал нулевых бит в течение заданного интервала времени. Если значение аргумента `fraction` равно нулю, послышка продолжается от четверти до половины секунды.

Поскольку стандарт POSIX-2001 не специфицирует способ представления в структуре `termios` данных о скорости передачи, для выборки/записи данной характеристики из/в `termios` потребовалось ввести семейство функций `cf*()`, что, конечно, хорошо, ибо это проявление (возможно, невольное) объектно-ориентированного подхода. Кроме того, в общем случае терминальное устройство может поддерживать для ввода и вывода разные скорости, поэтому определено четыре функции, названия которых говорят сами за себя. Разумеется, для реального опроса/изменения скорости обмена с терминалом нужно воспользоваться функциями `tcgetattr()/tcsetattr()`. Отметим также, что скорость должна задаваться не в виде целого числа, а с помощью определенных в заголовочном файле `<termios.h>` констант `B0, ..., B38400`.

При работе с (псевдо)терминалами (а также с каналами и сокетами) полезна функция `poll()` (см. листинг 9.9), входящая в расширение `XSI` и позволяющая мультиплексировать ввод/вывод в пределах набора файловых дескрипторов.

```
#include <poll.h>
int poll (struct pollfd fds [], nfds_t nfds,
          int timeout);
```

Листинг 9.9. Описание функции `poll()`.

Для каждого элемента массива `fds` (с числом элементов `nfds`) функция проверяет, наступили ли заданные события; в частности, возможны ли чтение или запись без блокирования процесса.

Согласно стандарту, структура `pollfd` содержит по крайней мере следующие поля.

```
int fd; /* Опрашиваемый файловый дескриптор */
short events; /* Флаги опрашиваемых событий */
short revents; /* Флаги произошедших событий */
```

Среди событий, обрабатываемых функцией `poll()`, выделим `POLLIN` (информирует о том, что можно без блокирования прочитать данные, не являющиеся высокоприоритетными), `POLLRDNORM` (можно прочитать обычные данные), `POLLOUT` (без блокирования можно записать обычные данные), `POLLERR` (выходной флаг — зафиксирована ошибка устройства или потока).

Разделение данных на обычные, приоритетные и высокоприоритетные зависит от реализации.

Если запрашиваемые события не наступают, `poll()` ждет по крайней мере `timeout` миллисекунд. Значение `-1` задает бесконечную задержку.

В качестве примера описанных функций приведем упрощенную версию программы, запускающей командный интерпретатор на псевдотерминале (см. листинг 9.10). Программа использует библиотеку `curses` для высокоуровневой работы с терминалами, которую мы не описываем, однако понять смысл фигурирующих в ней функций довольно просто. Отметим лишь, что функция `endwin()` восстанавливает характеристики терминала.

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Программа запускает shell на псевдотерминале */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * */

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

```
#include <fcntl.h>
#include <termios.h>
#include <signal.h>
#include <poll.h>
#include <sys/resource.h>
#include <curses.h>

/* Действия при завершении процесса */
static void termination (int errcode) {
    endwin ();
    exit (errcode);
}

/* Функция обработки сигнала SIGCHLD */
static void chldied (int dummy) {
    /* Просто завершимся */
    termination (34);
}

int main (void) {
    WINDOW *win1, *win2;
    /* win1 - окно только для рамки */
    /* win2 - окно для shell */

    int pty, tty;      /* Дескрипторы обеих сторон */
                      /* псевдотерминала */
    int fr;            /* Результат fork'a */
    unsigned char ch;  /* Прочитанный символ */
    struct termios pt; /* Структура для смены */
                      /* характеристик псевдотерминала */
    struct pollfd fds [2]; /* Массив параметров */
                      /* для вызова poll */
    char ptybuf [L_ctermid]; /* Массив для хранения */
                      /* имени псевдотерминала */
    char *s, *t;       /* Указатели для перебора */
                      /* компонентов имени псевдотерминала */
    int w2lines, w2cols; /* Размер создаваемого окна */

    int x, y;          /* Координаты в окне */
    struct sigaction sact;
    int i;
```

```
initscr ();
cbreak ();
noecho ();

win1 = newwin (LINES, COLS, 0, 0);
box (win1, 0, 0);
wrefresh (win1);

w2lines = LINES - 2;
w2cols = COLS - 4;
win2 = newwin (w2lines, w2cols, 1, 2);
scrollok (win2, TRUE);

/* Откроем первый свободный псевдотерминал */
for (s = "pqrs"; *s; s++) {
    for (t = "0123456789abcdef"; *t; t++) {
        sprintf (ptybuf, "/dev/pty%c%c", *s, *t);
        if ((pty = open (ptybuf, O_RDWR)) >= 0) {
            goto findpty;
        }
    }
}

fprintf (stderr, "Не удалось найти свободный"
        "псевдотерминал\n");
termination (-1);

findpty:
ptybuf [5] = 't';
if ((tty = open (ptybuf, O_RDWR)) < 0) {
    perror ("TTY OPEN ERROR");
    termination (-1);
}

/* Установим подходящие характеристики */
/* псевдотерминала */
if (tcgetattr (pty, &pt) < 0) {
    perror ("PTY TERMIOS GET ERROR");
    return (1);
}
pt.c_iflag = 0;
pt.c_oflag = ONLCR;
```

```
pt.c_cflag = CS8 | HUPCL;
pt.c_lflag = ISIG | ICANON | ECHO | ECHOE
    | ECHOK;
pt.c_cc [VINTR] = 3; /* CTRL+C */
pt.c_cc [VEOF] = 4; /* CTRL+D */
if (tcsetattr (pty, TCSADRAIN, &pt) < 0) {
    perror ("PTY TERMIOS SET ERROR");
    return (2);
}

/* То же - для стандартного ввода */
(void) tcgetattr (0, &pt);
pt.c_lflag &= ~ISIG;
(void) tcsetattr (0, TCSADRAIN, &pt);

/* Установим обработку сигнала о завершении */
/* потомка */
sact.sa_handler = chldied;
(void) sigemptyset (&sact.sa_mask);
sact.sa_flags = 0;
(void) sigaction (SIGCHLD, &sact,
    (struct sigaction *) NULL);

/* Раздвоимся на процесс чтения с клавиатуры */
/* и вывода на экран */
/* и на процесс, в рамках которого запустим shell */

if ((fr = fork ()) < 0) {
    perror ("FORK1 ERROR");
    termination (-1);
} else if (fr) {
    /* Это процесс, читающий с клавиатуры */
    /* и выводящий на экран */
    close (tty);

    /* Будем ждать ввода с клавиатуры или */
    /* псевдотерминала */
    fds [0].fd = 0;
    fds [0].events = POLLIN;
    fds [1].fd = pty;
    fds [1].events = POLLIN;
```



```
while (1) {
    if (poll (fds, 2, -1) < 0) {
        perror ("POLL ERROR");
        termination (0);
    }
    if (fds [0].revents & POLLIN) {
        /* Пришел символ со стандартного ввода */
        read (0, &ch, 1);
        write (pty, &ch, 1);
    }
    if (fds [1].revents & POLLIN) {
        /* Пришел символ с псевдотерминала */
        read (pty, &ch, 1);
        switch (ch) {
            case '\n': {
                /* Проинтерпретируем перевод строки */
                getyx (win2, y, x);
                if (y == (w2lines - 1)) {
                    wmove (win2, y, w2cols - 1);
                    waddch (win2, (chtype) ch);
                } else {
                    wmove (win2, y + 1, 0);
                }
                break;
            }
            default: {
                /* Символ не интерпретируется */
                waddch (win2, (chtype) ch);
                break;
            }
        }
        wrefresh (win2);
    }
}
/* Просто завершимся */
termination (0);

} else { /* Порожденный процесс - запустим в */
        /* нем shell */
        /* Закроем все файлы, кроме псевдотерминала */
        for (i = 0; i < RLIMIT_NOFILE; i++) {
            if (i != tty) {
```

```
        (void) close (i);
    }
}

/* Сделаем процесс лидером сеанса */
(void) setsid ();

/* Свяжем стандартные ввод, вывод и протокол */
/* с псевдотерминалом */
(void) fcntl (tty, F_DUPFD, 0);
(void) fcntl (tty, F_DUPFD, 0);
(void) fcntl (tty, F_DUPFD, 0);
close (tty);

/* Сделаем псевдотерминал управляющим */
if ((tty = open (ptybuf, O_RDWR)) < 0) {
    perror ("TTY OPEN ERROR");
    exit (-1);
}
close (tty);

/* Поместим в окружение параметры псевдотерминала */
{
    char lnbuf [20];
    char clbuf [20];
    sprintf (lnbuf, "LINES=%2d", w2lines);
    sprintf (clbuf, "COLUMNS=%2d", w2cols);
    putenv (lnbuf);
    putenv (clbuf);
}
if (execl ("/bin/sh", "sh", (char *) NULL) < 0) {
    perror ("EXECL ERROR");
    exit (-1);
}
}

return 0;
}
```

Листинг 9.10. Пример программы, использующей псевдотерминалы.

В соответствии с определением, выводимые на псевдотерминал символы являются входными для порожденного процесса. Они читаются по

дескриптору `pty`, а затем интерпретируются и выводятся на физический терминал. Обратим также внимание на манипуляции с флагом `ISIG`: сигналы разрешены только для псевдотерминала (но не для стандартного ввода).

В целях полноты изложения укажем, что в (частично) описанное выше семейство `tc*()` входят еще три функции, предназначенные для работы с управляющими терминалами (см. листинг 9.11).

```
#include <unistd.h>
pid_t tcgetpgrp (int fildes);

#include <unistd.h>
int tcsetpgrp (int fildes, pid_t pgid_id);

#include <termios.h>
pid_t tcgetsid (int fildes);
```

Листинг 9.11. Описание функций семейства `tc*()` для работы с управляющими терминалами.

Функция `tcgetpgrp()` позволяет узнать идентификатор группы процессов переднего плана, ассоциированной с терминалом, заданным открытым файловым дескриптором `fildes`. Для установки идентификатора группы служит функция `tcsetpgrp()`. Требуется, чтобы аргумент `fildes` соответствовал управляющему терминалу вызывающего процесса.

Функция `tcgetsid()`, входящая в расширение XSI, по сути аналогична `tcgetpgrp()`. Она позволяет узнать идентификатор группы процессов лидера сеанса, для которого терминал, ассоциированный с файловым дескриптором `fildes`, является управляющим.

Для выполнения в известном смысле обратной операции – получения маршрутного имени управляющего терминала для вызывающего процесса – служит функция `ctermid()` (см. листинг 9.12).

```
#include <stdio.h>
char *ctermid (char *s);
```

Листинг 9.12. Описание функции `ctermid()`.

Если значение аргумента `s` отлично от `NULL`, маршрутное имя управляющего терминала записывается по этому адресу; предполагается, что массив `s` имеет длину не менее `L_ctermid` байт. В противном случае имя помещается в область статической памяти, которая перезаписывается каждым вызовом.

Вариант 1

1. **Согласно стандарту POSIX-2001, канонический режим ввода включает**
 - ☐ построчную буферизацию ввода
 - ☐ обработку символа забора
 - ☐ обработку символа прерывания

2. **Согласно стандарту POSIX-2001, флаг ICANON входит в поле**
 - ☐ режимов ввода
 - ☐ режимов вывода
 - ☐ локальных режимов

3. **Согласно стандарту POSIX-2001, в неканоническом режиме ввода при $\text{MIN} > 0$ и $\text{TIME} > 0$ значение TIME трактуется как**
 - ☐ максимальное время ожидания прихода первого байта
 - ☐ общее время ожидания запрошенного числа байт
 - ☐ максимальное время ожидания между приходом последовательных байт

Вариант 2

1. **Согласно стандарту POSIX-2001, канонический режим ввода подразумевает**
 - ☐ отсутствие буферизации ввода
 - ☐ обработку символа уничтожения строки
 - ☐ эхо отображение ввода

2. **Согласно стандарту POSIX-2001, флаг PARENB входит в поле**
 - ☐ режимов ввода
 - ☐ режимов вывода
 - ☐ управляющих режимов

3. **Согласно стандарту POSIX-2001, в неканоническом режиме ввода при $\text{MIN} = 0$ и $\text{TIME} > 0$ значение TIME трактуется как**
 - ☐ максимальное время ожидания прихода первого байта
 - ☐ общее время ожидания запрошенного числа байт
 - ☐ максимальное время ожидания в интервале между приходом последовательных байт

Вариант 3

1. Согласно стандарту POSIX-2001, в неканоническом режиме входные данные

- ☐ всегда подвергаются препроцессированию
- ☐ не подвергаются препроцессированию
- ☐ подвергаются препроцессированию в зависимости от режимов ввода

2. Согласно стандарту POSIX-2001, значение B38400 входит в поле

- ☐ режимов ввода
- ☐ режимов вывода
- ☐ управляющих режимов

3. Согласно стандарту POSIX-2001, в неканоническом режиме ввода при $MIN > 0$ и $TIME = 0$ запрос на чтение завершится

- ☐ без ожидания
- ☐ после прихода запрошенного числа байт
- ☐ после прихода MIN байт

Лекция 10. Опрос характеристик хостов и их использование в приложениях

Рассматриваются характеристики хостов (систем), важнейшими из которых являются конфигурационные параметры, а также служебные программы, функции для их опроса и способы настройки приложений на целевые конфигурации.

Ключевые слова: имя хоста, имя узла, имя системы, конфигурационная константа, конфигурационный параметр, значение конфигурационного параметра, условная компиляция, лимитирующая конфигурационная константа, минимально допустимое значение, целевая конфигурация, среда компиляции.

Опрос идентифицирующих данных хостов

Самую общую информацию о характеристиках хоста позволяют получить служебная программа `uname`

```
uname [-snrvma]
```

и одноименная функция (см. листинг 10.1).

```
#include <sys/utsname.h>
int uname (struct utsname *name);
```

Листинг 10.1. Описание функции `uname()`.

Функция `uname()` помещает информацию, идентифицирующую опрашиваемую систему, в структуру типа `utsname`, которая должна содержать по крайней мере следующие поля.

```
char sysname []; /* Имя реализации ОС */
char nodename []; /* Имя хоста как узла */
/* коммуникационной сети */
char release []; /* Номер выпуска ОС */
char version []; /* Номер версии ОС */
char machine []; /* Название аппаратной */
/* конфигурации */
```

Размер массивов не специфицируется, оговаривается только, что хранящиеся в них цепочки символов завершаются нулевым байтом.

Аналогичные данные выдает на стандартный вывод служебная программа `uname`, опции которой соответствуют полям структуры `utsname` (`-m` — выдать название аппаратной конфигурации, `-n` — имя узла, `-s`, `-r` и `-v` — имя реализации ОС, номера выпуска и версии, соответственно). Опция `-a` предписывает выдавать все; без опций выдается имя реализации.

Результат выполнения команды

```
uname -a
```

может выглядеть так, как показано в листинге 10.2. Правда, здесь фигурирует нестандартная составляющая, выдаваемая по опции `-r`, — тип процессора (в данном случае `unknown`).

```
Linux t94 2.4.18-3 #1 Thu Apr 18 07:37:53 EDT 2002
i686 unknown
```

Листинг 10.2. Возможный результат выполнения команды `uname -a`.

Функция `gethostname()` (см. листинг 10.3) возвращает в массиве `name` длины `namelen` имя хоста. Подходящее значение для `namelen` — `HOST_NAME_MAX + 1` (см. следующий раздел).

```
#include <unistd.h>
int gethostname (char *name, size_t namelen);
```

Листинг 10.3. Описание функции `gethostname()`.

Опрос конфигурационных параметров хоста

Значения конфигурационных параметров — важнейшая характеристика хоста, а настройка на конфигурацию целевой системы — обязательный элемент мобильного программирования приложений. Подобная настройка может выполняться статически, средствами условной компиляции, а также динамически, путем опроса значений конфигурационных параметров во время выполнения.

Основная часть статической конфигурационной информации сосредоточена в упоминавшемся ранее заголовочном файле `<unistd.h>`. Кратко опишем наиболее важные характеристики.

`_POSIX_VERSION`

Поддерживаемая версия системного интерфейса для языка C стандарта POSIX. Для POSIX-2001 значение этой константы должно равняться 200112L. Предыдущей версии (1996 года) соответствует значение 199506L.

`_POSIX2_VERSION`

Поддерживаемая версия интерфейса к системным сервисам на уровне командного языка и служебных программ; для реализаций, соответствующих стандарту POSIX-2001, значение этой константы должно равняться 200112L.

В листинге 10.4 показан пример использования значения `_POSIX_VERSION` для условной компиляции.

```
#if _POSIX_VERSION >= 200112L
/* Используем новую функцию, применимую к */
/* большим файлам */
off_t fpos = ftello (fp);
#else
/* Либо реализация поддерживает только старую */
/* версию стандарта POSIX, либо */
/* константа _POSIX_VERSION вообще не определена. */
/* Используем старую, традиционную функцию */
/* опроса текущей позиции в файле */
long fpos = ftell (fp);
#endif
```

Листинг 10.4. Пример условной компиляции с использованием конфигурационной константы `_POSIX_VERSION`.

Следующая группа констант описывает поддерживаемые необязательные возможности стандарта POSIX-2001. Значения этих констант могут равняться -1 (необязательная возможность не поддерживается), 0 (в системе присутствуют все необходимые описания, но наличие поддержки нужно опросить во время выполнения) или быть положительными (возможность полностью поддерживается реализацией).

`_POSIX_CHOWN_RESTRICTED`

Функции `chown()` и `fchown()` могут вызываться для смены владельца только процессами, обладающими соответствующими привилегиями.

`_POSIX_IPV6`

Реализация поддерживает IPv6. Положительное значение этой константы должно равняться 200112L.

`_POSIX_JOB_CONTROL`

Реализация поддерживает управление заданиями.

`_POSIX_NO_TRUNC`

Компоненты маршрутных имен, имеющие длину более NAME_MAX, вызывают сообщения об ошибках.

`_POSIX_REGEX`

Реализация поддерживает обработку регулярных выражений.

`_POSIX_SAVED_IDS`

Каждый процесс имеет сохраненный ПДП-идентификатор и сохраненный ПДГ-идентификатор.

`_POSIX_SHELL`

Реализация поддерживает стандартный командный интерпретатор.

`_POSIX_VDISABLE`

Значением этой константы должен быть символ, отключающий обработку соответствующего специального управляющего символа терминала, если его (символ `_POSIX_VDISABLE`) сделать значением элемента массива `c_cc[]`.

`_POSIX2_C_BIND`

Реализация поддерживает системный интерфейс для языка C. Значение этой константы должно равняться 200112L.

`_POSIX2_CHAR_TERM`

Реализация поддерживает по крайней мере один тип терминалов.

`_POSIX2_LOCALEDEF`

Реализация поддерживает создание языково-культурных сред. Положительное значение должно равняться 200112L.

`_POSIX_V6_ILP32_OFF32`

Реализация предоставляет среду компиляции С-программ с 32-битными типами `int`, `long`, `off_t` и такими же указателями.

`_POSIX_V6_ILP32_OFFBIG`

Реализация предоставляет среду компиляции С-программ с 32-битными типами `int`, `long` и такими же указателями; размер значений типа `off_t` составляет не менее 64 бит.

`_POSIX_V6_LP64_OFF64`

Реализация предоставляет среду компиляции С-программ с 32-битным типом `int` и 64-битными указателями и типами `long` и `off_t`.

`_POSIX_V6_LPBIG_OFFBIG`

Реализация предоставляет среду компиляции С-программ с не менее чем 32-битными значениями типа `int` и не менее чем 64-битными указателями и значениями типов `long` и `off_t`.

`_POSIX__ASYNC_IO`

Поддерживается асинхронный ввод/вывод.

`_POSIX__PRIO_IO`

Поддерживается приоритетный ввод/вывод.

`_POSIX__SYNC_IO`

Поддерживается синхронизированный ввод/вывод.

Три константы задают номера файловых дескрипторов для стандартных ввода (`STDIN_FILENO` со значением 0), вывода (`STDOUT_FILENO` – 1) и протокола (`STDERR_FILENO` – 2). Их нужно не анализировать, а использовать вместо явных числовых значений.

Разного рода лимиты на количество и размеры заданы в заголовочном файле `<limits.h>`. Стандарт определяет минимально допустимые значения для подобных лимитов, которые поддерживаются реализациями и предоставляются приложениям.

Мобильные приложения не должны полагаться на какие-то конкретные значения лимитов и требовать больше ресурсов, чем предусмотрено минимально допустимыми значениями.

Перечислим наиболее важные из числа лимитирующих конфигурационных констант. В скобках после имени константы указаны имя минимально допустимого значения и само это значение (если, конечно, они определены в стандарте).

```
ARG_MAX  (_POSIX_ARG_MAX = 4096)
```

Максимально допустимая длина аргумента функций семейства `exec*()` (включая данные окружения).

```
ATEXIT_MAX (32)
```

Максимально допустимое число функций, которые можно зарегистрировать посредством `atexit()`.

```
CHARCLASS_NAME_MAX (_POSIX2_CHARCLASS_NAME_MAX = 14)
```

Максимально допустимое число байт в имени класса символов.

```
CHILD_MAX  (_POSIX_CHILD_MAX = 25)
```

Максимально допустимое число одновременно существующих процессов с одним реальным идентификатором пользователя.

```
HOST_NAME_MAX (_POSIX_HOST_NAME_MAX = 255)
```

Максимально допустимая длина имени хоста (без завершающего нулевого байта).

```
LINE_MAX  (_POSIX2_LINE_MAX = 2048)
```

Максимально допустимое число байт в строке текстового файла.

```
LOGIN_NAME_MAX (_POSIX_LOGIN_NAME_MAX = 9)
```

Максимально допустимая длина входного имени (с учетом завершающего нулевого байта). Таким образом, по соображениям мобильности входное имя должно состоять не более чем из восьми байт.

NGROUPS_MAX (_POSIX_NGROUPS_MAX = 8)

Максимально допустимое число дополнительных групп для процесса.

OPEN_MAX (_POSIX_OPEN_MAX = 20)

Максимально допустимое число файлов, одновременно открытых в одном процессе.

PAGESIZE, PAGE_SIZE (1)

Размер страницы оперативной памяти в байтах.

RE_DUP_MAX (_POSIX2_RE_DUP_MAX = 255)

Максимально допустимое число повторных вхождений регулярного выражения при использовании интервальной нотации $\{m, n\}$.

STREAM_MAX (_POSIX_STREAM_MAX = 8)

Максимально допустимое число потоков, одновременно открытых в одном процессе.

SYMLOOP_MAX (_POSIX_SYMLINK_MAX = 8)

Максимально допустимое число символьных ссылок, которые могут быть обработаны в случае разрешения маршрутных имен файлов при отсутствии циклов.

TTY_NAME_MAX (_POSIX_TTY_NAME_MAX = 9)

Максимально допустимая длина имени терминального устройства (с учетом завершающего нулевого байта).

TZNAME_MAX (_POSIX_TZNAME_MAX = 6)

Максимально допустимая длина имени часового пояса.

Следующие конфигурационные параметры, строго говоря, относятся не к хосту, а к файловой системе или даже ее части; тем не менее, их естественно рассмотреть наряду с общесистемными характеристиками.

FILESIZEBITS (32)

Минимальное число бит, необходимое для представления размера обычного файла в виде целочисленного значения со знаком.

LINK_MAX (_POSIX_LINK_MAX = 8)

Максимально допустимое число жестких ссылок на файл.

MAX_CANON (_POSIX_MAX_CANON = 255)

Максимально допустимое число байт во входной строке при каноническом вводе с терминала.

MAX_INPUT (_POSIX_MAX_INPUT = 255)

Максимально допустимое число байт во входной терминальной очереди.

NAME_MAX (_POSIX_NAME_MAX = 14)

Максимально допустимая длина простого имени файла (без завершающего нулевого байта).

PATH_MAX (_POSIX_PATH_MAX = 256)

Максимально допустимая длина маршрутного имени файла (с учетом завершающего нулевого байта).

PIPE_BUF (_POSIX_PIPE_BUF = 512)

Максимально допустимое число байт, которое можно неделимым образом записать в канал.

SYMLINK_MAX (_POSIX_SYMLINK_MAX = 255)

Максимально допустимое число байт в символьной ссылке.

Следующие конфигурационные константы определяют границы числовых значений различных типов для целевой конфигурации, которые могут стать предметом проверок в директивах условной компиляции.

`CHAR_BIT` (8)

Число бит в значении типа `char`.

`CHAR_MAX`

Максимальное значение типа `char`. Совпадает с `UCHAR_MAX` или `SCHAR_MAX`.

`CHAR_MIN`

Минимальное значение типа `char` (`SCHAR_MIN` или 0).

`INT_MAX` (2147483647).

Максимальное значение типа `int`.

`LONG_BIT` (32)

Число бит в значении типа `long`.

`LONG_MAX` (2147483647)

Максимальное значение типа `long`.

`SCHAR_MAX` (127)

Максимальное значение типа `signed char`.

`SHRT_MAX` (32767)

Максимальное значение типа `short`.

`SSIZE_MAX` (`_POSIX_SSIZE_MAX` = 32767)

Максимальное значение типа `ssize_t`.

UCHAR_MAX (255)

Максимальное значение типа `unsigned char`.

UINT_MAX (4294967295)

Максимальное значение типа `unsigned int`.

ULONG_MAX (4294967295)

Максимальное значение типа `unsigned long`.

USHRT_MAX (65535)

Максимальное значение типа `unsigned short`.

WORD_BIT (16)

Число бит в слове или значении типа `int`.

INT_MIN (-2147483647)

Минимальное значение типа `int`.

LONG_MIN (-2147483647)

Минимальное значение типа `long`.

SCHAR_MIN (-128)

Минимальное значение типа `signed char`.

SHRT_MIN (-32767)

Минимальное значение типа `short`.

LLONG_MIN (-9223372036854775807)

Минимальное значение типа `long long`.

LLONG_MAX (9223372036854775807)

Максимальное значение типа `long long`.

`ULLONG_MAX` (18446744073709551615)

Максимальное значение типа `unsigned long long`.

Некоторые полезные конфигурационные константы определены в заголовочном файле `<stdio.h>`.

`BUFSIZ`

Размер буферов ввода/вывода.

`L_ctermid`

Максимальный размер символьного массива для хранения имени управляющего терминала - результата функции `ctermid()`.

`L_tmpnam`

Максимальный размер символьного массива для хранения имени временного файла — результата функции `tmpnam()`.

`FILENAME_MAX`

Максимально допустимое число байт в имени открываемого файла.

`TMP_MAX` (25) .

Максимально допустимое число обращений к функции `tmpnam()`, при котором гарантируется уникальность генерируемых имен.

Упомянутая ранее служебная программа `getconf`

```
getconf [-v спецификация] конфигурационный_параметр
getconf [-v спецификация] конфигурационный_параметр
маршрутное имя
```

а также функции `sysconf()`, `confstr()`, `fpathconf()` и `pathconf()` (см. листинг 10.5) предназначены для опроса значений системных параметров во время выполнения. Первая из перечисленных функций опрашивает лимитирующие конфигурационные параметры, имеющие числовые значения, вторая возвращает конфигурационные цепочки символов, а две последние — конфигурационные значения, относящиеся к файлам.

```
#include <unistd.h>
long sysconf (int name);
size_t confstr (int name, char *buf, size_t len);
long fpathconf (int fildes, int name);
long pathconf (const char *path, int name);
```

Листинг 10.5. Описание функций `sysconf()`, `confstr()`, `fpathconf()` и `pathconf()`.

Утилита `getconf` выдает на стандартный вывод значение конфигурационного параметра, которое во второй форме может быть специфичным для файловой системы или даже для файла. В первом приближении можно считать, что по набору опрашиваемых параметров она эквивалентна совокупности функций, представленных в листинге 10.5, только в именах, обслуживаемых функцией `confstr()`, нужно удалять префикс `_CS_`.

Опция

`-v` спецификация

задает опрашиваемую конфигурацию и версию. По умолчанию используется подразумеваемое для данной реализации окружение.

Например, если определен конфигурационный параметр `__POSIX_V6_ILP32_OFF32`, то команда вида

```
getconf -v __POSIX_V6_ILP32_OFF32 ...
```

опрашивает значения, соответствующие 32-битной среде компиляции.

Приведем несколько примеров использования служебной программы `getconf`. Команда

```
getconf LINE_MAX
может выдать 2048, а команда
getconf PATH_MAX /tmp
```

в используемой автором версии ОС Linux выдает 4096.

Обращение со служебной программой `getconf` требует известной аккуратности. Нужно помнить, что конфигурационный параметр может быть неопределенным, а указанное имя не обслуживаться вообще; в последнем случае на стандартный вывод ничего не выдается. В листинге 10.6 показан фрагмент командного файла, где сделана попытка учесть эти нюансы.

```

if unum=$(getconf "$@" ); then
    if [ "$unum" = "undefined" ]; then
        echo Значение "$@" не определено
    else
        echo Значение "$@" равно $unum.
    fi
else
    echo Ошибка при выполнении getconf
fi

```

Листинг 10.6. Пример аккуратной обработки результатов утилиты `getconf`.

Функция `sysconf()` опрашивает описанные выше конфигурационные параметры с числовыми значениями, определенные в заголовочных файлах `<unistd.h>` и `<limits.h>`; аргумент `name` задает имя опрашиваемого параметра.

Функция `confstr()` возвращает значения для следующих конфигурационных параметров, определенных в заголовочном файле `<unistd.h>` и заданных аргументом `name`:

`_CS_PATH`

Значение переменной окружения `PATH`, позволяющее найти все стандартные служебные программы.

`_CS_POSIX_V6_ILP32_OFF32_CFLAGS`
`_CS_POSIX_V6_ILP32_OFF32_LDFLAGS`
`_CS_POSIX_V6_ILP32_OFF32_LIBS`

Опции и библиотеки для компиляции и редактирования внешних связей приложений, использующих 32-битную программную модель.

`_CS_POSIX_V6_ILP32_OFFBIG_CFLAGS`
`_CS_POSIX_V6_ILP32_OFFBIG_LDFLAGS`
`_CS_POSIX_V6_ILP32_OFFBIG_LIBS`
`_CS_POSIX_V6_LP64_OFF64_CFLAGS`
`_CS_POSIX_V6_LP64_OFF64_LDFLAGS`
`_CS_POSIX_V6_LP64_OFF64_LIBS`
`_CS_POSIX_V6_LPBIG_OFFBIG_CFLAGS`
`_CS_POSIX_V6_LPBIG_OFFBIG_LDFLAGS`
`_CS_POSIX_V6_LPBIG_OFFBIG_LIBS`

Опции и библиотеки для других программных моделей.

`_CS_POSIX_V6_WIDTH_RESTRICTED_ENVS`

Список имен поддерживаемых реализацией программных моделей, в которых размеры типов `blksize_t`, `cc_t`, `mode_t`, `nfds_t`, `pid_t`, `ptrdiff_t`, `size_t`, `speed_t`, `ssize_t`, `suseconds_t`, `tcflag_t`, `useconds_t`, `wchar_t` и `wint_t` не больше, чем `long`.

Значение конфигурационного параметра возвращается функцией `confstr()` в буфере `buf` длины `len` (при необходимости значение обрывается). Результатом вызова `confstr()` является минимальный размер буфера, где можно разместить опрашиваемое значение вместе с завершающим нулевым байтом. Если в качестве `len` задан нуль, в буфер ничего не помещается, но требуемый размер выдается. В листинге 10.7 показана программа, динамически резервирующая буфер необходимого размера и получающая в нем стандартное значение списка поиска.

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

int main (void) {
    size_t buf_len;
    char *buf;

    if ((buf_len = confstr (_CS_PATH, (char *) NULL,
        (size_t) 0)) > 0) {
        if ((buf = (char *) malloc (buf_len)) != NULL) {
            (void) confstr (_CS_PATH, buf, buf_len);
            printf ("Standard PATH: %s\n", buf);
        }
    }

    return 0;
}
```

Листинг 10.7. Пример аккуратного опроса конфигурационной цепочки символов.

В результате выполнения этой программы может быть выдано:

Standard PATH: /bin:/usr/bin

Функции `fpathconf()` и `pathconf()` служат для опроса текущих значений лимитирующих конфигурационных параметров или наличия необязательных возможностей, ассоциированных с файлами или каталогами, которые могут задаваться открытыми дескрипторами (аргумент `fd` функции `fpathconf()`) или маршрутными именами (аргумент `path` функции `pathconf()`).

Аргумент `name` задает имя опрашиваемого параметра. Выше мы уже описывали подобные конфигурационные параметры: `FILESIZEBITS`, `LINK_MAX`, `MAX_CANON`, `MAX_INPUT`, `NAME_MAX`, `PATH_MAX`, `PIPE_BUF`, `SYMLINK_MAX`, `_POSIX_CHOWN_RESTRICTED`, `_POSIX_NO_TRUNC`, `_POSIX_VDISABLE`, `_POSIX_ASYNC_IO`, `_POSIX_PRIO_IO`, `_POSIX_SYNC_IO`. Допустимые значения для `name` получаются путем приписывания к имени параметра префикса `_PC_` (например, `_PC_PATH_MAX`) или замены префикса `_POSIX_` на `_PC_` (например, `_PC_SYNC_IO`).

Отметим, что во время работы приложения результаты, возвращаемые функциями `fpathconf()` и `pathconf()`, могут меняться, поскольку существует вероятность монтирования и/или размонтирования файловых систем, поддерживающих различные необязательные возможности и разные лимиты. В этом смысле результаты функций `sysconf()` и `confstr()` более стабильны.

Вариант 1

1. Согласно стандарту POSIX-2001, результат функции `uname()` имеет тип

- ☐ `int`
- ☐ `long`
- ☐ `unsigned int`

2. Согласно стандарту POSIX-2001, конфигурационный параметр `_POSIX_SHELL` хранит

- ☐ маршрутное имя стандартного командного интерпретатора
- ☐ признак того, что реализация поддерживает стандартный командный интерпретатор
- ☐ простое имя стандартного командного интерпретатора

3. Согласно стандарту POSIX-2001, результат функции `confstr()` имеет тип

- ☐ `char *`
- ☐ `long`
- ☐ `size_t`

Вариант 2

1. Согласно стандарту POSIX-2001, результат функции `uname()` имеет тип

- ☐ `char *`
- ☐ `struct utsname *`
- ☐ `void *`

2. Согласно стандарту POSIX-2001, конфигурационный параметр `_POSIX_VDISABLE` хранит

- ☐ признак того, что работа с виртуальной памятью не поддерживается
- ☐ признак того, что работа с видеотерминалами не поддерживается
- ☐ символ, отключающий обработку специального управляющего символа терминала

3. Согласно стандарту POSIX-2001, функция `pathconf()` имеет

- ☐ один аргумент
- ☐ два аргумента
- ☐ три аргумента

Вариант 3

1. Согласно стандарту POSIX-2001, в число полей структуры типа `utsname` входят

- ☐ имя хоста как узла коммуникационной сети
- ☐ сетевой адрес хоста
- ☐ тип процессора

2. Согласно стандарту POSIX-2001, конфигурационный параметр `_POSIX2_CHAR_TERM` хранит

- ☐ байт, завершающий в данной реализации цепочки символов
- ☐ имя консольного терминала
- ☐ признак того, что реализация поддерживает по крайней мере один тип терминалов

3. Согласно стандарту POSIX-2001, первым аргументом функции `fpathconf()` является

- ☐ имя опрашиваемого конфигурационного параметра
- ☐ маршрутное имя файла
- ☐ открытый файловый дескриптор

Лекция 11. Сетевые средства

Представлены основные понятия и объекты, ассоциированные с сетевыми средствами. Описываются функции для опроса данных о сети и для работы с сокетами. Приводятся многочисленные примеры использования сетевых средств.

Ключевые слова: сеть, сетевой адрес, узел сети, сетевая база, связывание, освобождение, оконечная точка, маршрутизация, база данных маршрутизации, октет, последовательность октетов, порядок байт, сетевой порядок байт, хостовый порядок байт, сокет, адрес сокета, адресное семейство, тип сокета, протокол, режим с установлением соединения, режим без установления соединения, база данных о хостах как узлах сети, соединение с базой, сервис, порт, номер порта, база данных сетей, база данных сетевых протоколов, база данных сетевых сервисов, создание сокета, пара соединенных безымянных сокетов, опрос локального адреса сокета, имя сокета, слушающий сокет, очередь соединений, прием соединения, установление соединения, асинхронное установление соединения, мультиплексирование ввода/вывода, время ожидания, опция сокета, значение опции, протокольный уровень опции, статус ошибок, отладочный режим, отправка данных, прием данных, экстренные данные, сообщение, разнесение сообщений, сборка сообщений, вспомогательные данные, флаг сообщения, граница записи, завершение операций, датаграмма, исходный адрес, серверный процесс, порожденный процесс, зомби-процесс, мультиплексирование ввода.

Основные понятия и объекты

Стандарт POSIX-2001 определяет сеть как совокупность взаимосвязанных хостов. Тем самым предполагается, что сетевая инфраструктура остается скрытой от приложений, которым предоставляются высокоуровневые средства взаимодействия в распределенной среде.

Под сетевым адресом понимается видимый в пределах сети идентификатор, используемый для обозначения оконечных точек сети. Адреса есть у определенных оконечных точек хостов, могут они быть и у хостов в целом.

Данные о хостах как узлах сети хранятся в сетевой базе, допускающей и последовательный, и случайный доступ с возможностью поиска по именам и адресам хостов.

Процесс присвоения сетевого адреса оконечной точке называется связыванием, или привязкой, а обратное действие – освобождением, или отменой привязки.

Обычно оконечной точкой служит аппаратный сетевой интерфейс, посредством которого данные передаются и принимаются, однако с таким интерфейсом, как шлейфовый (loopback), никакой аппаратуры не ассоциировано.

Поддерживается база данных маршрутизации, используемая при выборе сетевого интерфейса для передачи порции данных (сетевого пакета).

Данные передаются по сети в виде последовательности октетов (восьмибитных беззнаковых величин). Если некоторый элемент данных (например, адрес или номер порта) состоит более чем из восьми бит, для его передачи и хранения требуется несколько октетов. Сетевым называется порядок октетов (байт), при котором первый (с наименьшим адресом) октет содержит старшие (наиболее значимые) биты.

Последовательности октетов – неудобный объект обработки на хостах, где предпочтительнее аппаратно поддерживаемые типы, в особенности целочисленные. Значения этих типов обычно хранятся с другим порядком байт, называемым хостовым, поэтому вполне возможно, что старшего бита не окажется в первом байте и вообще будет использоваться некое неочевидное распределение бит по байтам.

Для преобразования значений типов `uint16_t` и `uint32_t` из хостового порядка байт в сетевой служат функции `htons()` и `htonl()`; функции `ntohs()` и `ntohl()` осуществляют обратную операцию.

При взаимодействии процессов оконечными точками служат сокеты, они трактуются стандартом POSIX-2001 как отдельный тип файлов.

Под адресом сокета как (удаленной) оконечной точки понимается структура, включающая идентификатор адресного семейства и адресную информацию, специфичную для данного семейства. Последняя может состоять из нескольких компонентов, в том числе сетевого адреса хоста и идентификатора конкретной оконечной точки.

Основные описания, относящиеся к сокетам, сосредоточены в заголовочном файле `<sys/socket.h>`. Фигурирует в нем и упомянутая выше структура `sockaddr` для адреса сокета, которая должна содержать по крайней мере следующие поля.

```
sa_family_t sa_family; /* Адресное семейство */
char sa_data[];        /* Адрес сокета (данные */
                        /* переменной длины) */
```

Адресное семейство соответствует конкретной среде взаимодействия. Стандарт POSIX-2001 определяет три таких семейства.

AF_UNIX

Адресное семейство UNIX поддерживает межпроцессное взаимодействие в пределах одной системы. Формально это можно считать вырожденным случаем сетевого взаимодействия. Описания, специфичные для данного семейства, содержатся в заголовочном файле `<sys/un.h>`.

AF_INET

Адресное семейство, поддерживающее взаимодействие по протоколам IPv4. Специфичные для него описания располагаются в заголовочном файле `<netinet/in.h>`.

AF_INET6

Взаимодействие по протоколам IPv6 (необязательная возможность). За счет поддержки адресов IPv6, отображенных на IPv4, обеспечивается совместимость с приложениями, использующими IPv4. Применяемые этим адресным семейством описания распределены по заголовочным файлам `<netinet/in.h>`, `<arpa/inet.h>` и `<netdb.h>`.

В пределах каждого адресного семейства могут существовать сокеты нескольких типов. В стандарте POSIX-2001 их четыре.

SOCK_STREAM

Сокеты данного типа поддерживают надежные, упорядоченные, полнодуплексные потоки октетов в режиме с установлением соединения.

SOCK_SEQPACKET

Аналог SOCK_STREAM с дополнительным сохранением границ между записями.

SOCK_DGRAM

Передача данных в виде датаграмм в режиме без установления соединения.

SOCK_RAW

(Необязательная возможность). Аналог SOCK_DGRAM с дополнительной возможностью доступа к протокольным заголовкам и другой

информации нижнего уровня. Создавать сокеты этого типа могут лишь процессы с соответствующими привилегиями.

Для каждого адресного семейства каждый тип сокета может поддерживаться одним или несколькими протоколами. В частности, в адресном семействе `AF_INET` для сокетов типа `SOCK_STREAM` подразумеваемым является протокол с именем `IPPROTO_TCP`, а для типа `SOCK_DGRAM` — `IPPROTO_UDP`; посредством «прозрачных» сокетов (`SOCK_RAW`) можно воспользоваться протоколом `ICMP`, задав имя `IPPROTO_ICMP`, и т.д.

Общая логика работы с сокетами состоит в следующем. Сокеты создаются с помощью функции `socket()`, которой в качестве аргументов передают адресное семейство, тип сокета и протокол, а в результате получают открытый файловый дескриптор. Затем посредством функции `bind()` сокету присваивают локальный адрес. Если сокет ориентирован на режим с установлением соединения, то, прибегнув к функции `listen()`, его следует пометить как готового принимать соединения. Реальный прием соединений выполняет функция `accept()`, создающая для каждого из них новый сокет по образу и подобию «слушающего». В свою очередь, потенциальный партнер по взаимодействию инициирует соединение, применяя функцию `connect()`. (В режиме без установления соединения функция `connect()` позволяет специфицировать адрес отправляемых через сокет датаграмм.)

Для приема поступивших в сокет данных можно воспользоваться универсальной функцией низкоуровневого ввода/вывода `read()` или специализированным семейством функций `recv*()`, а для передачи — функцией `write()` или семейством `send*()`. Кроме того, функции `select()` и/или `poll()` помогут проверить наличие данных для приема или возможность отправки очередной порции данных.

Обращение к функции `shutdown()` завершает взаимодействие между партнерами.

Опрос данных о сети

Данные о хостах как узлах сети хранятся в сетевой базе, последовательный доступ к которой обслуживается функциями `sethostent()`, `gethostent()` и `endhostent()` (см. листинг 11.1).

```
#include <netdb.h>
void sethostent (int stayopen);
struct hostent *gethostent (void);
void endhostent (void);
```

Листинг 11.1. Описание функций последовательного доступа к сетевой базе данных о хостах — узлах сети.

Функция `sethostent()` устанавливает соединение с базой, остающееся открытым после вызова `gethostent()`, если значение аргумента `stayopen` отлично от нуля. Функция `gethostent()` последовательно читает элементы базы, возвращая результат в структуре типа `hostent`, содержащей по крайней мере следующие поля.

```
char *h_name; /* Официальное имя хоста */
char **h_aliases; /* Массив указателей на */
    /* альтернативные имена хоста, завершаемый */
    /* пустым указателем */
int h_addrtype; /* Тип адреса хоста */
int h_length; /* Длина в байтах адреса данного */
    /* типа */
char **h_addr_list; /* Массив указателей на сетевые */
    /* адреса хоста, завершаемый пустым указателем */
```

Функция `endhostent()` закрывает соединение с базой.

В листинге 11.2 показана программа, осуществляющая последовательный просмотр сетевой базы данных о хостах — узлах сети, а в листинге 11.3 приведен фрагмент ее возможной выдачи.

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

int main (void) {
    struct hostent *pht;
    char *pct;
    int i, j;

    sethostent (1);

    while ((pht = gethostent ()) != NULL) {
        printf ("Официальное имя хоста: %s\n",
            pht->h_name);
        printf ("Альтернативные имена:\n");

        for (i = 0; (pct = pht->h_aliases [i]) != NULL;
            i++) {
            printf (" %s\n", pct);
        }
        printf ("Тип адреса хоста: %d\n",
            pht->h_addrtype);
    }
```

```

    printf ("Длина адреса хоста: %d\n",
           pht->h_length);
    printf ("Сетевые адреса хоста:\n");
    for (i = 0; (pct = pht->h_addr_list [i]) !=
         NULL; i++) {
        for (j = 0; j < pht->h_length; j++) {
            printf (" %d", (unsigned char) pct [j]);
        }
        printf ("\n");
    }
}

endhostent ();
return 0;
}

```

Листинг 11.2. Пример программы, осуществляющей последовательный доступ к сетевой базе данных о хостах — узлах сети.

```

Официальное имя хоста: localhost
Альтернативные имена:
Тип адреса хоста: 2
Длина адреса хоста: 4
Сетевые адреса хоста:
127 0 0 1

```

```

. . .
Официальное имя хоста: t01
Альтернативные имена:
niisi.msk.ru
t01.niisi.msk.ru
mail
mailhost
loghost
server
server3
server3.systud.msk.su
www
www.systud.msk.su
t01.systud.msk.su
Тип адреса хоста: 2
Длина адреса хоста: 4

```

Сетевые адреса хоста:

193 232 173 1

. . .

Официальное имя хоста: t17

Альтернативные имена:

galatenko

Тип адреса хоста: 2

Длина адреса хоста: 4

Сетевые адреса хоста:

193 232 173 17

. . .

Листинг 11.3. Фрагмент возможных результатов работы программы, осуществляющей последовательный доступ к сетевой базе данных о хостах – узлах сети.

К рассматриваемой базе возможен и случайный доступ по ключам – именам и адресам хостов с помощью функций `gethostbyname()` и `gethostbyaddr()`, однако они считаются устаревшими и из новой версии стандарта POSIX могут быть исключены. Вместо них предлагается использовать функции `getnameinfo()` и `getaddrinfo()` (см. листинг 11.4).

```
#include <sys/socket.h>
#include <netdb.h>

void freeaddrinfo (struct addrinfo *ai);

int getaddrinfo (const char *restrict nodename,
                 const char *restrict servname,
                 const struct addrinfo *restrict hints,
                 struct addrinfo **restrict res);

int getnameinfo (const struct sockaddr *restrict sa,
                 socklen_t salen, char *restrict node,
                 socklen_t nodelen, char *restrict service,
                 socklen_t servicelen, int flags);
```

Листинг 11.4. Описание функций `freeaddrinfo()`, `getaddrinfo()`, `getnameinfo()`.

Функция `getaddrinfo()` позволяет по имени узла сети (хоста) (аргумент `nodename`) и/или имени сетевого сервиса (`servname`) получить набор адресов сокетов и ассоциированную информацию, что дает возможность создать сокет для обращения к заданному сервису.

Если аргумент `nodename` отличен от пустого указателя, он способен задавать описательное имя или адресную цепочку. Для адресных семейств `AF_INET` и `AF_UNSPEC` (см. ниже описание аргумента `hints`) именем может служить имя хоста, а адресной цепочкой – стандартные для Internet адреса в точечных обозначениях (например, `193.232.173.17`).

При пустом значении `nodename` подразумевается хост, локальный для вызывающего процесса.

Аргумент `servname` может задавать имя сервиса или (для адресных семейств `AF_INET` и `AF_UNSPEC`) десятичный номер порта. Пустое значение `servname` означает запрос сетевого адреса.

Аргумент `hints` позволяет передать дополнительную информацию об опрашиваемом сервисе – адресное семейство, тип сокета, протокол, флаги. Согласно стандарту, структура `addrinfo`, описанная в заголовочном файле `<netdb.h>`, должна содержать по крайней мере следующие поля.

```
int          ai_flags; /* Входные флаги */
int          ai_family; /* Адресное семейство */
                /* сокета */
int          ai_socktype; /* Тип сокета */
int          ai_protocol; /* Протокол сокета */
socklen_t    ai_addrlen; /* Длина адреса сокета */
struct sockaddr *ai_addr; /* Адрес сокета */
char         *ai_canonname; /* Официальное имя */
                /* узла сети */
struct addrinfo *ai_next; /* Указатель на */
                /* следующий элемент списка */
```

При обращении к функции `getaddrinfo()` все поля структуры `addrinfo`, на которую указывает аргумент `hints`, кроме первых четырех (`ai_flags`, `ai_family`, `ai_socktype`, `ai_protocol`), должны быть нулевыми или равными `NULL`. Значение `AF_UNSPEC` в поле `ai_family` подразумевает, что вызывающего устройт любое адресное семейство. Аналогичный смысл имеют нулевые значения полей `ai_socktype` и `ai_protocol`. При `hints`, равном `NULL`, подразумевается `AF_UNSPEC` для `ai_family` и нулевые значения для других полей.

Из флагов, которые могут быть установлены в поле `ai_flags`, упомянем следующие.

AI_PASSIVE

Если значение аргумента `nodename` равно `NULL`, этот флаг игнорируется. В противном случае, если он указан, будет возвращен адрес сокета, предназначенного для принятия входящих соединений.

AI_CANONNAME

Данный флаг предписывает выяснить официальное имя узла сети.

AI_NUMERICHOST

Флаг означает, что хост задан адресной цепочкой, и не допускает использования какого-либо сервиса имен.

AI_NUMERICSERV

Флаг помечает, что сервис (аргумент `servname`) задан номером порта, и налагает запрет на обращение к какому-либо сервису имен.

Признаком успешного завершения функции `getaddrinfo()` является нулевой результат. В таком случае выходной аргумент `res` будет ссылаться на указатель на список структур типа `addrinfo` (связанных полем `ai_next`) – принадлежащие им значения полей `ai_family`, `ai_socktype`, `ai_protocol` пригодны для создания подходящих сокетов с помощью функции `socket()`, а значения `ai_addr` и `ai_addrlen`, в зависимости от флага `AI_PASSIVE`, могут служить аргументами функций `connect()` или `bind()`, применяемых к созданному сокету.

Функция `freeaddrinfo()` позволяет освободить память, занятую списком структур типа `addrinfo` и ассоциированными данными.

Функцию `getnameinfo()` можно считать обратной по отношению к `getaddrinfo()`. Аргумент `sa` – входной, он задает транслируемый адрес сокета (`salen` – размер структуры `sockaddr`). Аргументы `node` и `service` – выходные, задающие адреса областей памяти, куда помещаются, соответственно, имя узла и сервиса; размеры этих областей ограничены значениями `nodelen` и `servicelen`.

По умолчанию предполагается, что сокет имеет тип `SOCK_STREAM`, а кроме того, возвращается полное доменное имя хоста. Аргумент `flags` позволяет изменить подразумеваемое поведение. Если задан флаг `NI_DGRAM`, сокет считается датаграммным. При установленном флаге `NI_NOFQDN` возвращается короткое имя узла. Флаги `NI_NUMERICHOST` и `NI_NUMERICSERV` предписывают возвращать числовые цепочки для адресов хоста и сервиса, соответственно.

Обратим внимание на несколько технических деталей. При работе с адресами сокетов вместо родовой структуры типа `sockaddr` обычно используются более специализированные (описанные в заголовочном файле `<netinet/in.h>`) – `sockaddr_in` для адресного семейства `AF_INET` (IPv4) и `sockaddr_in6` для `AF_INET6` (IPv6). Первая из них, согласно стандарту POSIX-2001, должна содержать по крайней мере следующие поля (все с сетевым порядком байт).

```
sa_family_t  sin_family; /* AF_INET */
in_port_t    sin_port;    /* Номер порта */
struct in_addr sin_addr; /* IP-адрес */
```

Структура типа `sockaddr_in6` устроена несколько сложнее; мы не будем ее рассматривать.

Структуры типов `sockaddr` и `sockaddr_in` (или `sockaddr_in6`) мысленно накладываются друг на друга, а преобразование типов между ними выполняется по мере необходимости. Отметим также, что тип `in_port_t` эквивалентен `uint16_t`, структура типа `in_addr` содержит по крайней мере одно поле:

```
in_addr_t s_addr;
```

тип `in_addr_t` эквивалентен `uint32_t`.

Техническую роль играют и функции преобразования IP-адресов из текстового представления в числовое и наоборот (см. листинг 11.5).

```
#include <arpa/inet.h>
in_addr_t inet_addr (const char *cp);
char *inet_ntoa (struct in_addr in);
int inet_pton (int af, const char *restrict src,
               void *restrict dst);
const char *inet_ntop (int af, const void
                       *restrict src, char *restrict dst, socklen_t size);
```

Листинг 11.5. Описание функций преобразования IP-адресов из текстового представления в числовое и наоборот.

Первые две функции манипулируют только адресами IPv4: `inet_addr()` преобразует текстовую цепочку `cp` (адрес в стандартных точечных обозначениях) в пригодное для использования в качестве IP-адреса

целочисленное значение, `inet_ntoa()` выполняет обратное преобразование.

Вторая пара функций по сути аналогична первой, но имеет чуть более общий характер, так как способна преобразовывать адреса в формате IPv6. Первый аргумент этих функций, `af`, задает адресное семейство: `AF_INET` для IPv4 и `AF_INET6` для IPv6. Буфер, на который указывает аргумент `dst` функции `inet_pton()` (в него помещается результат преобразования – IP-адрес в числовой двоичной форме с сетевым порядком байт), должен иметь длину не менее 32 бит для адресов IPv4 и 128 бит для IPv6.

Аргумент `src` функции `inet_ntop()`, возвращающей текстовое представление, указывает на буфер с IP-адресом в числовой форме с сетевым порядком байт. Аргумент `size` задает длину выходного буфера, на него указывает аргумент `dst`. Подходящими значениями, в зависимости от адресного семейства, могут служить `INET_ADDRSTRLEN` или `INET6_ADDRSTRLEN`.

Выше было отмечено, что преобразование значений типов `uint16_t` и `uint32_t` из хостового порядка байт в сетевой выполняется посредством функций `htons()` и `htonl()`; функции `ntohs()` и `ntohl()` осуществляют обратную операцию (см. листинг 11.6).

```
#include <arpa/inet.h>
uint32_t htonl (uint32_t hostlong);
uint16_t htons (uint16_t hostshort);
uint32_t ntohl (uint32_t netlong);
uint16_t ntohs (uint16_t netshort);
```

Листинг 11.6. Описание функций преобразования целочисленных значений из хостового порядка байт в сетевой и наоборот.

Использование функции `getaddrinfo()` вместе с сопутствующими техническими деталями проиллюстрируем программой, показанной в листинге 11.7. Возможные результаты ее выполнения приведены в листинге 11.8.

```
#include <stdio.h>
#include <netdb.h>
#include <arpa/inet.h>
int main (void) {
    struct addrinfo hints = {AI_CANONNAME, AF_INET,
        SOCK_STREAM, IPPROTO_TCP, 0, NULL, NULL, NULL};
    struct addrinfo *addr_res;
    if (getaddrinfo ("www", "http", &hints,
```

```
&addr_res) != 0) {
    perror ("GETADDRINFO");
} else {
    printf ("Результаты для сервиса http\n");
/* Пройдем по списку возвращенных структур */
do {
    printf ("Адрес сокета: Порт: %d IP-адрес:"
           "%s\n", ntohs (((struct sockaddr_in *)
           addr_res->ai_addr)->sin_port),
           inet_ntoa (((struct sockaddr_in *)
           addr_res->ai_addr)->sin_addr));
    printf ("Официальное имя хоста: %s\n",
           addr_res->ai_canonname);
} while ((addr_res = addr_res->ai_next)
        != NULL);
}

return 0;
}
```

Листинг 11.7. Пример программы, использующей функцию `etaddrinfo()`.

```
Результаты для сервиса http
Адрес сокета: Порт: 80 IP-адрес: 193.232.173.1
Официальное имя хоста: t01
```

Листинг 11.8. Возможный результат работы программы, использующей функцию `getaddrinfo()`.

Завершая изложение серии технических моментов, укажем, что полезным дополнением к функциям `getaddrinfo()` и `getnameinfo()` является функция `gai_strerror()` (см. листинг 11.9). Она возвращает текстовую цепочку, расшифровывающую коды ошибок, перечисленные в заголовочном файле `<netdb.h>`. Стандарт POSIX-2001 специфицирует следующие коды ошибок, имена которых говорят сами за себя: `EAI_AGAIN`, `EAI_BADFLAGS`, `EAI_FAIL`, `EAI_FAMILY`, `EAI_MEMORY`, `EAI_NONAME`, `EAI_OVERFLOW`, `EAI_SERVICE`, `EAI_SOCKTYPE`, `EAI_SYSTEM`.

```
#include <netdb.h> const char *gai_strerror (int
    ecode);
```

Листинг 11.9. Описание функции `gai_strerror()`.

Если немного модифицировать приведенную выше программу (в листинге 11.10 показан измененный фрагмент, где имя сервиса – «HTTP» – задано большими буквами), то в стандартный протокол с помощью функции `gai_strerror()` будет выдано содержательное диагностическое сообщение (см. листинг 11.11). Отметим, что выдача функции `error()` в данном случае неважна.

```
    . . .  
    int res;  
  
    if ((res = getaddrinfo ("www", "HTTP", &hints,  
        &addr_res)) != 0) {  
        perror ("GETADDRINFO");  
        fprintf (stderr, "GETADDRINFO: %s\n",  
            gai_strerror (res));  
    } else {  
        printf ("Результаты для сервиса HTTP\n");  
        . . .  
    }
```

Листинг 11.10. Модифицированный фрагмент программы, использующей функции `getaddrinfo()` и `gai_strerror()`.

```
GETADDRINFO: No such file or directory  
GETADDRINFO: Servname not supported for ai_socktype
```

Листинг 11.11. Диагностические сообщения от функций `perror()` и `gai_strerror()`, выданные по результатам работы функции `getaddrinfo()`.

Наряду с базой данных хостов (узлов сети) поддерживается база данных сетей с аналогичной логикой работы и набором функций (см. листинг 11.12).

```
#include <netdb.h>  
void setnetent (int stayopen);  
struct netent *getnetent (void);  
struct netent *getnetbyaddr (uint32_t net, int type);  
struct netent *getnetbyname (const char *name);  
void endnetent (void);
```

Листинг 11.12. Описание функций доступа к базе данных сетей.

Функция `getnetent()` обслуживает последовательный доступ к базе, `getnetbyaddr()` осуществляет поиск по адресному семейству (аргумент `type`) и номеру `net` сети, а `getnetbyname()` выбирает сеть с заданным (официальным) именем. Структура типа `netent`, указатель на которую возвращается в качестве результата этих функций, согласно стандарту POSIX-2001, должна содержать по крайней мере следующие поля.

```
char    *n_name;           /* Официальное имя сети */
char    **n_aliases;       /* Массив указателей на */
                           /* альтернативные имена сети, */
                           /* завершаемый пустым указателем */
int      n_addrtype;        /* Адресное семейство (тип */
                           /* адресов) сети */
uint32_t n_net;             /* Номер сети (в хостовом */
                           /* порядке байт) */
```

Точно такой же программный интерфейс предоставляет база данных сетевых протоколов (см. листинг 11.13).

```
#include <netdb.h>
void setprotoent (int stayopen);
struct protoent *getprotoent (void);
struct protoent *getprotobyname (const char *name);
struct protoent *getprotobynumber (int proto);
void endprotoent (void);
```

Листинг 11.13. Описание функций доступа к базе данных сетевых протоколов.

Структура типа `protoent` содержит по крайней мере следующие поля.

```
char    *p_name;           /* Официальное имя протокола */
char    **p_aliases;       /* Массив указателей на */
                           /* альтернативные имена протокола, */
                           /* завершаемый пустым указателем */
int      p_proto;          /* Номер протокола */
```

В листинге 11.14 показан пример программы, осуществляющей последовательный и случайный доступ к базе данных сетевых протоколов. Листинг 11.15 содержит фрагмент возможных результатов работы этой программы.

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

int main (void) {
    struct protoent *pht;
    char *pct;
    int i;

    setprotoent (1);

    while ((pht = getprotoent ()) != NULL) {
        printf ("Официальное имя протокола: %s\n",
            pht->p_name);
        printf ("Альтернативные имена:\n");
        for (i = 0; (pct = pht->p_aliases [i]) != NULL;
            i++) {
            printf (" %s\n", pct);
        }
        printf ("Номер протокола: %d\n\n",
            pht->p_proto);
    }
    if ((pht = getprotobyname ("ipv6")) != NULL) {
        printf ("Номер протокола ipv6: %d\n\n",
            pht->p_proto);
    } else {
        fprintf (stderr, "Протокол ip в базе"
            "не найден\n");
    }

    if ((pht = getprotobyname ("IPV6")) != NULL) {
        printf ("Номер протокола IPV6: %d\n\n",
            pht->p_proto);
    } else {
        fprintf (stderr, "Протокол IPV6 в базе"
            "не найден\n");
    }
    endprotoent ();
    return 0;
}
```

Листинг 11.14. Пример программы, осуществляющей последовательный и случайный доступ к базе данных сетевых протоколов.

Официальное имя протокола: ip

Альтернативные имена:

IP

Номер протокола: 0

Официальное имя протокола: icmp

Альтернативные имена:

ICMP

Номер протокола: 1

. . .

Официальное имя протокола: tcp

Альтернативные имена:

TCP

Номер протокола: 6

. . .

Официальное имя протокола: udp

Альтернативные имена:

UDP

Номер протокола: 17

. . .

Официальное имя протокола: ipv6

Альтернативные имена:

IPv6

Номер протокола: 41

. . .

Официальное имя протокола: ipv6-crypt

Альтернативные имена:

IPv6-Crypt

Номер протокола: 50

. . .

Официальное имя протокола: visa

Альтернативные имена:

VISA

Номер протокола: 70

. . .

Официальное имя протокола: iso-ip

Альтернативные имена:

ISO-IP

Номер протокола: 80

. . .

Официальное имя протокола: sprite-rpc

Альтернативные имена:

Sprite-RPC

Номер протокола: 90

. . .

Официальное имя протокола: ipx-in-ip

Альтернативные имена:

IPX-in-IP

Номер протокола: 111

. . .

Официальное имя протокола: fc

Альтернативные имена:

FC

Номер протокола: 133

Номер протокола ipv6: 41

Протокол IPV6 в базе не найден

Листинг 11.15. Фрагмент возможных результатов работы программы, осуществляющей последовательный и случайный доступ к базе данных сетевых протоколов.

Еще одно проявление той же логики работы — база данных сетевых сервисов (см. листинг 11.16).

```
#include <netdb.h>
void setservent (int stayopen);
struct servent *getservent (void);
struct servent *getservbyname (const char *name,
    const char *proto);
struct servent *getservbyport (int port,
    const char *proto);
void endservent (void);
```

Листинг 11.16. Описание функций доступа к базе данных сетевых сервисов.

Обратим внимание на то, что в данном случае можно указывать второй аргумент поиска – имя протокола. Впрочем, значение аргумента `proto` может быть пустым указателем, и тогда поиск производится только по имени сервиса (функция `getservbyname()`) или номеру порта (`getservbyport()`), который должен быть задан с сетевым порядком байт.

Структура типа `servent` содержит по крайней мере следующие поля.

```
char    *s_name;      /* Официальное имя сервиса */
char    **s_aliases; /* Массив указателей на */
    /* альтернативные имена сервиса, завершаемый */
    /* пустым указателем */
int     s_port; /* Номер порта, соответствующий */
    /* сервису (в сетевом порядке байт) */
char    *s_proto; /* Имя протокола для */
    /* взаимодействия с сервисом */
```

В листинге 11.17 приведен пример программы, использующей функции доступа к базе данных сервисов, а также функции преобразования между хостовым и сетевым порядками байт. В листинге 11.18 показан фрагмент возможных результатов работы этой программы.

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

int main (void) {
    struct servent *pht;
    char *pct;
    int i;

    setservent (1);
```

```
while ((pht = getservent ()) != NULL) {
    printf ("Официальное имя сервиса: %s\n",
        pht->s_name);
    printf ("Альтернативные имена:\n");
    for (i = 0; (pct = pht->s_aliases [i]) != NULL;
        i++) {
        printf (" %s\n", pct);
    }
    printf ("Номер порта: %d\n",
        ntohs ((in_port_t) pht->s_port));
    printf ("Имя протокола: %s\n\n",
        pht->s_proto);
}

if ((pht = getservbyport (htons ((in_port_t) 21),
    "udp")) != NULL) {
    printf ("Официальное имя сервиса: %s\n",
        pht->s_name);
    printf ("Альтернативные имена:\n");
    for (i = 0; (pct = pht->s_aliases [i]) != NULL;
        i++) {
        printf (" %s\n", pct);
    }
    printf ("Номер порта: %d\n",
        ntohs ((in_port_t) pht->s_port));
    printf ("Имя протокола: %s\n\n",
        pht->s_proto);
} else {
    perror ("GETSERVBYPOR");
}

if ((pht = getservbyport (htons ((in_port_t) 21),
    (char *) NULL)) != NULL) {
    printf ("Официальное имя сервиса: %s\n",
        pht->s_name);
    printf ("Альтернативные имена:\n");
    for (i = 0; (pct = pht->s_aliases [i]) != NULL;
        i++) {
        printf (" %s\n", pct);
    }
    printf ("Номер порта: %d\n",
        ntohs ((in_port_t) pht->s_port));
```

```
    printf ("Имя протокола: %s\n\n",
           pht->s_proto);
} else {
    perror ("GETSERVBYPORT");
}

endservent ();

return 0;
}
```

Листинг 11.17. Пример программы, использующей функции доступа к базе данных сервисов, а также функции преобразования между хостовым и сетевым порядками байт.

. . . .

Официальное имя сервиса: ftp-data
Альтернативные имена:
Номер порта: 20
Имя протокола: tcp

Официальное имя сервиса: ftp-data
Альтернативные имена:
Номер порта: 20
Имя протокола: udp

Официальное имя сервиса: ftp
Альтернативные имена:
Номер порта: 21
Имя протокола: tcp

Официальное имя сервиса: ftp
Альтернативные имена:
 fsp
 fspd
Номер порта: 21
Имя протокола: udp

. . . .

Официальное имя сервиса: kerberos

Альтернативные имена:

kerberos5

krb5

Номер порта: 88

Имя протокола: tcp

Официальное имя сервиса: kerberos

Альтернативные имена:

kerberos5

krb5

Номер порта: 88

Имя протокола: udp

. . .

Официальное имя сервиса: auth

Альтернативные имена:

authentication

tap

ident

Номер порта: 113

Имя протокола: tcp

Официальное имя сервиса: auth

Альтернативные имена:

authentication

tap

ident

Номер порта: 113

Имя протокола: udp

. . .

Официальное имя сервиса: printer

Альтернативные имена:

spooler

Номер порта: 515

Имя протокола: tcp

Официальное имя сервиса: printer

Альтернативные имена:

spooler

Номер порта: 515
Имя протокола: udr

. . .

Официальное имя сервиса: fido
Альтернативные имена:
Номер порта: 60179
Имя протокола: tcp

Официальное имя сервиса: fido
Альтернативные имена:
Номер порта: 60179
Имя протокола: udr

Официальное имя сервиса: ftp
Альтернативные имена:
 fsp
 fspd
Номер порта: 21
Имя протокола: udr

Официальное имя сервиса: ftp
Альтернативные имена:
Номер порта: 21
Имя протокола: tcp

Листинг 11.18. Фрагмент возможных результатов работы программы, использующей функции доступа к базе данных сервисов, а также функции преобразования между хостовым и сетевым порядками байт.

Отметим, что при поиске по ключу возвращается первый подходящий элемент базы данных. По этой причине, когда не был задан протокол (второе обращение к функции `getservbyport()`), в качестве результата был возвращен элемент с протоколом `tcp`.

Функции для работы с сокетами

Работа с сокетами как оконечными точками при взаимодействии процессов начинается с их (сокетов) создания посредством функции `socket()` (см. листинг 11.19). Она возвращает открытый файловый дескриптор,

который может быть использован в последующих вызовах функций, оперирующих с сокетами. В том же листинге показано описание функции `socketpair()`, создающей пару сокетов с установленным между ними соединением.

```
#include <sys/socket.h>
int socket (int af, int type, int protocol);
int socketpair (int af, int type, int protocol,
               int sds [2]);
```

Листинг 11.19. Описание функций `socket()` и `socketpair()`.

Аргумент `af` задает адресное семейство создаваемого сокета, аргумент `type` – тип, аргумент `protocol` – конкретный протокол (0 обозначает неспецифицированный подразумеваемый протокол, пригодный для запрошенного типа). Напомним, что подходящие значения для аргументов `af`, `type` и `protocol` можно получить с помощью описанной ранее функции `getaddrinfo()`.

Функция `socketpair()` по назначению аналогична `pipe()`, только организуется не безымянный канал, а пара соединенных, безымянных (не привязанных к каким-либо адресам), идентичных сокетов, открытые файловые дескрипторы которых помещаются в массив `sds`. Обычно она используется для адресного семейства `AF_UNIX`; поддержка для других семейств не гарантируется.

После того как посредством функции `bind()` (см. листинг 11.20) создан сокет, идентифицируемый дескриптором `sd`, ему присваивают локальный адрес, заданный аргументом `address` (`address_len` – длина структуры `sockaddr`, на которую указывает `address`). Источником локальных адресов для сокетов может служить вышеупомянутая функция `getaddrinfo()`.

```
#include <sys/socket.h>
int bind (int sd, const struct sockaddr *address,
         socklen_t address_len);
```

Листинг 11.20. Описание функции `bind()`.

Опросить присвоенный локальный адрес (его иногда называют именем сокета) можно с помощью функции `getsockname()` (см. листинг 11.21): она помещает его в структуру `sockaddr`, на которую указывает аргумент `address`, а длину адреса записывает по указателю `address_len`.

```
#include <sys/socket.h>
int getsockname (int sd, struct sockaddr *restrict
    address, socklen_t *restrict address_len);
```

Листинг 11.21. Описание функции `getsockname()`.

Если сокет ориентирован на режим с установлением соединения (имеет тип `SOCK_STREAM`), то, воспользовавшись функцией `listen()` (см. листинг 11.22), его следует пометить как готового принимать соединения («слушающего»).

```
#include <sys/socket.h>
int listen (int sd, int backlog);
```

Листинг 11.22. Описание функции `listen()`.

Аргумент `backlog` сообщает операционной системе рекомендуемую длину очереди соединений, ожидающих обработки слушающим сокетом. Реализация должна поддерживать значения аргумента `backlog` вплоть до конфигурационной константы `SOMAXCONN`, определенной в заголовочном файле `<sys/socket.h>`. ОС имеет право установить длину очереди меньше рекомендуемой. При неположительном значении `backlog` очередь будет иметь зависящую от реализации минимальную длину.

Прием соединений выполняет функция `accept()` (см. листинг 11.23). Она выбирает первое соединение из очереди, ассоциированной с заданным дескриптором `sd` слушающим сокетом, создает новый сокет с теми же адресным семейством, типом и протоколом и возвращает в качестве результата его файловый дескриптор.

```
#include <sys/socket.h>
int accept (int sd, struct sockaddr *restrict
    address, socklen_t *restrict address_len);
```

Листинг 11.23. Описание функции `accept()`.

Если значение аргумента `address` отлично от пустого указателя, то в структуру `sockaddr`, на которую указывает `address`, помещается адрес сокета, пытающегося установить соединение. По указателю `address_len` при обращении к функции `accept()` должна быть задана длина переданной структуры `sockaddr`, а на выходе туда записывается длина адреса партнера по взаимодействию.

Если очередь соединений, ожидающих обработки слушающим сокетом, пуста и для дескриптора `sd` не установлен флаг `O_NONBLOCK`, то вызвавший функцию `accept()` процесс (поток управления) блокируется до появления подобного соединения. При непустой очереди функция `select()` сообщит о готовности дескриптора `sd` к чтению.

Другая сторона, т. е. потенциальный партнер по взаимодействию, инициирует соединение с помощью функции `connect()` (см. листинг 11.24). Аргументы `address` и `address_len` стандартным образом задают адрес сокета (как правило, слушающего), с которым необходимо установить соединение.

```
#include <sys/socket.h>
int connect (int sd, const struct sockaddr *address,
             socklen_t address_len);
```

Листинг 11.24. Описание функции `connect()`.

Если для сокета, заданного аргументом `sd` (запрашивающего установление соединения), еще не выполнена привязка к локальному адресу, функция `connect()` сама осуществит связывание со свободным локальным адресом (правда, лишь при условии, что адресное семейство сокета отлично от `AF_UNIX`).

Функция `connect()` ограничится фиксацией адреса сокета, взаимодействующего с заданным аргументом `sd`, если тип сокета не требует установления соединения. В частности, для сокетов типа `SOCK_DGRAM` таким способом можно специфицировать адреса отправляемых (с помощью функции `send()`) и принимаемых (посредством обращения к функции `recv()`) датаграмм.

Когда в качестве аргумента `address` передается пустой указатель, адрес взаимодействующего сокета сбрасывается.

Попытка установить соединение блокирует вызывающий процесс на неспецифицируемый промежуток времени (в случае наличия флага `O_NONBLOCK`). Если по истечении этого промежутка соединение установить не удалось, вызов `connect()`, равно как и попытка установления соединения, завершаются неудачей. Если ожидание прерывается обрабатываемым сигналом, вызов `connect()` завершается неудачей (переменной `errno` присваивается значение `EINTR`), но установление соединения продолжается и будет завершено асинхронно.

Если для дескриптора `sd` задан флаг `O_NONBLOCK`, а соединение не может быть установлено немедленно, то вызов `connect()` завершается неудачей со значением `errno`, равным `EINPROGRESS`, но установление соединения продолжается и будет завершено асинхронно. Последующие

обращения к функции `connect()` с тем же сокетом, выполненные до установления соединения, завершаются неудачей (`EALREADY`).

Сокет оказывается в неспецифицированном состоянии, если функция `connect()` завершается неудачей по другим причинам. Приложения, соответствующие стандарту POSIX-2001, должны закрыть файловый дескриптор `sd` и создать новый сокет для продолжения попыток установить соединение.

После асинхронного установления соединения функции `select()` и `poll()` сообщают, что файловый дескриптор `sd` готов к записи.

Функция `poll()`, позволяющая мультиплексировать ввод/вывод в пределах набора файловых дескрипторов, была описана нами выше. Имеющая сходную направленность (но входящая в обязательную часть стандарта POSIX-2001) функция `select()` и ее недавно введенный аналог `pselect()` представлены в листинге 11.25.

```
#include <sys/select.h>
int pselect (int nfd, fd_set *restrict readfds,
             fd_set *restrict writefds, fd_set *restrict errorfds,
             const struct timespec *restrict timeout,
             const sigset_t *restrict sigmask);
int select (int nfd, fd_set *restrict readfds,
            fd_set *restrict writefds, fd_set *restrict errorfds,
            struct timeval *restrict timeout);
void FD_CLR (int fd, fd_set *fdset);
int FD_ISSET (int fd, fd_set *fdset);
void FD_SET (int fd, fd_set *fdset);
void FD_ZERO (fd_set *fdset);
```

Листинг 11.25. Описание функций семейства `select*()`.

Если значение аргумента `readfds` (`writefds`, `errorfds`) функции `pselect()` отлично от `NULL`, оно ссылается на объект типа `fd_set`, который на входе специфицирует набор файловых дескрипторов, проверяемых на готовность к чтению (или к записи, или к обработке исключительных ситуаций), а на выходе указывает, какие из них успешно прошли проверку. Аргумент `nfd` задает границу проверяемых дескрипторов (они являются небольшими целыми числами): дескриптор подлежит проверке, если его значение находится в пределах от 0 до (`nfd - 1`) включительно.

Стандарт POSIX-2001 определяет тип `fd_set` как абстрактный. Для работы со значениями этого типа служат макросы `FD_ZERO()` (сделать набор пустым), `FD_SET()` (добавить дескриптор к набору), `FD_CLR()` (удалить дескриптор из набора), `FD_ISSET()` (проверить принадлежность

дескриптора набору). Значение именованной константы `FD_SETSIZE` равно максимально допустимому числу дескрипторов в наборе.

В качестве результата `pselect()` возвращается общее количество дескрипторов во всех трех наборах, успешно прошедших проверку.

При отсутствии готовых дескрипторов вызывающий процесс блокируется, пока таковые не появятся, или пока не истечет заданное аргументом `timeout` время ожидания, или пока ожидание не будет прервано сигналом. Пустой указатель в качестве значения `timeout` означает бесконечное ожидание, а нулевые значения полей структуры `timespec` – отсутствие блокировки. Реализация может накладывать ограничение на максимальное время ожидания, но оно не должно быть меньше тридцати одного дня.

Если значение аргумента `sigmask` отлично от пустого указателя, функция `pselect()` на входе заменяет маску сигналов процесса на заданную, а на выходе восстанавливает прежнюю маску.

Функция `pselect()` поддерживает обычные файлы, терминалы и псевдотерминалы, каналы и сокеты.

Функция `select()` эквивалентна `pselect()` со следующими оговорками.

- 1) Для функции `select()` время ожидания задается в секундах и микросекундах в виде структуры типа `timeval`, а для `pselect()` – в секундах и наносекундах как аргумент типа `struct timespec`.
- 2) У функции `select()` нет аргумента – маски сигналов, что эквивалентно пустому указателю в качестве значения аргумента `sigmask` функции `pselect()`.
- 3) В случае успешного завершения функция `select()` может модифицировать структуру, на которую указывает аргумент `timeout`.

С сокетами могут быть ассоциированы опции, влияющие на их функционирование. Значения этих опций можно опросить или изменить с помощью функций `getsockopt()` и `setsockopt()` (см. листинг 11.26).

```
#include <sys/socket.h>
int getsockopt (int sd, int level,
               int option_name, void *restrict option_value,
               socklen_t *restrict option_len);
int setsockopt (int sd, int level, int
               option_name, const void *option_value,
               socklen_t option_len);
```

Листинг 11.26. Описание функций `getsockopt()` и `setsockopt()`.

Опции задаются именованными константами (аргумент `option_name`), которые определены в заголовочном файле `<sys/socket.h>`. Выделим среди них наиболее употребительные и разделим на несколько групп. К первой отнесем опции с целочисленными значениями, описывающими характеристики или состояние сокета.

`SO_ERROR`

Статус ошибок (после опроса очищается).

`SO_TYPE`

Тип сокета.

Ко второй группе отнесем булевы опции, представленные целочисленными значениями (0 означает ложь).

`SO_DEBUG`

Сообщает, записывается ли отладочная информация о работе сокета.

`SO_ACCEPTCONN`

Указывает, является ли сокет слушающим.

В третью группу включены опции с целочисленными значениями, определяющими количественные характеристики объектов, ассоциированных с сокетом.

`SO_SNDBUF`

Размер буфера для передаваемых данных (выходного буфера).

`SO_RCVBUF`

Размер входного буфера.

`SO_RCVLOWAT`

Минимальное число байт, обрабатываемых при вводе. Подразумеваемое значение равно единице. Большие значения могут вызвать блокировку принимающего процесса до поступления в сокет необходимого объема данных.

SO_SNDLOWAT

Минимальное число байт, обрабатываемых при выводе.
В четвертую группу входят опции со структурными значениями.

SO_LINGER

Определяет, блокировать ли процесс при закрытии дескриптора `sd` до передачи буферизованных данных, и если блокировать, то на какой срок. Значением опции является структура `linger`, определенная в заголовочном файле `<sys/socket.h>` и содержащая, согласно стандарту POSIX-2001, по крайней мере следующие поля.

```
int l_onoff; /* Признак, включена ли опция */
             /* блокирования при закрытии */
int l_linger; /* Длительность блокирования */
             /* в секундах */
```

SO_RCVTIMEO

Длительность ожидания поступления данных при вводе. Значение — упомянутая выше структура типа `timeval`. Подразумеваемая длительность равна нулю. Если в течение специфицированного промежутка времени новых данных не поступило, операция ввода вернет число байт, меньше запрошенного, или завершится ошибкой `EAGAIN`.

SO_SNDTIMEO

Длительность ожидания отправки данных при выводе.

Не все из перечисленных опций могут быть переустановлены функцией `setsockopt()`. Исключение по понятным причинам составляют `SO_ERROR`, `SO_TYPE`, `SO_ACCEPTCONN`.

Аргумент `level` задает протокольный уровень опции. Уровню сокетов соответствует значение `SOL_SOCKET`, уровню TCP — `IPPROTO_TCP`.

Функция `getpeername()` (см. листинг 11.27), во многом аналогичная рассмотренной выше функции `getsockname()`, позволяет опросить еще одну характеристику — адрес (имя) сокета, с которым установлено соединение.

```
#include <sys/socket.h>
int getpeername(int sd, struct sockaddr *restrict address,
                socklen_t *restrict address_len);
```

Листинг 11.27. Описание функции `getpeername()`.

После привязки сокета к локальному адресу, а также установления, при необходимости, соединения и задания значений опций, можно приступать к отправке и/или приему данных через сокет. Для этого служат функции, описания которых показаны в листинге 11.28.

```
#include <sys/socket.h>
ssize_t recvfrom (int sd, void *restrict buffer,
    size_t length, int flags,
    struct sockaddr *restrict address,
    socklen_t *restrict address_len);
ssize_t recv (int sd, void *buffer, size_t length,
    int flags);
ssize_t recvmsg (int sd, struct msghdr *message,
    int flags);
ssize_t sendto (int sd, const void *message,
    size_t length, int flags,
    const struct sockaddr *dest_addr,
    socklen_t dest_len);
ssize_t send (int sd, const void *buffer,
    size_t length, int flags);
ssize_t sendmsg (int sd,
    const struct msghdr *message, int flags);
```

Листинг 11.28. Описание функций обмена данными через сокет.

Функция `recvfrom()` позволяет прочитать данные (в рассматриваемом контексте называемые также сообщением) из сокета с дескриптором `sd` и поместить их в буфер `buffer` длины `length`. Обычно функцию `recvfrom()` применяют к сокетам, ориентированным на режим без установления соединения, поскольку она выдает исходный адрес в структуре типа `sockaddr`.

В число возможных составляющих значения аргумента `flags` входят следующие флаги.

`MSG_PEEK`

Не удалять прочитанные данные. Следующий вызов `recvfrom()` или другой функции ввода снова вернет их.

`MSG_WAITALL`

Для сокетов типа `SOCK_STREAM` флаг означает, что вызывающий процесс блокируется до получения всего запрошенного объема данных (а не до прихода первого сообщения).

`MSG_OOB`

Запрашиваются экстренные данные. Трактовка этого понятия зависит от протокола.

Как и положено функции ввода, в результате `recvfrom()` возвращает количество прочитанных и помещенных в буфер данных. Для сокетов, учитывающих границы сообщений (`SOCK_RAW`, `SOCK_DGRAM`, `SOCK_SEQPACKET`), за одно обращение читается все сообщение; если оно не помещается в буфер, лишние байты (в отсутствие флага `MSG_PEEK`) отбрасываются.

Формально можно считать, что функция `recv()` эквивалентна `recvfrom()` с нулевым значением аргумента `address_len`. Поскольку она не позволяет узнать исходный адрес, ее обычно используют для сокетов, установивших соединение. (Можно провести и еще одну аналогию: если пренебречь флагами, функция `recv()` эквивалентна `read()`).

По сравнению с `recv()`, более содержательным аналогом функции `recvfrom()` является `recvmsg()`. Отличия в данном случае носят скорее синтаксический характер и по сути сводятся к способу передачи входных значений и возврата результатов: для минимизации числа аргументов функция `recvmsg()` использует структуру типа `msghdr`, которая, согласно стандарту POSIX-2001, должна содержать по крайней мере следующие поля.

```
void      *msg_name; /* Указатель на буфер для */
              /* адреса (исходного или целевого) */
socklen_t  msg_namelen; /* Размер буфера для */
              /* адреса */
struct iovec *msg_iov; /* Массив для */
              /* разнесения/сборки сообщений */
int        msg_iovlen; /* Число элементов в */
/* массиве для разнесения/сборки сообщений */
void      *msg_control; /* Указатель на буфер */
              /* для вспомогательных данных */
socklen_t  msg_controllen; /* Размер буфера для */
              /* вспомогательных данных */
int msg_flags; /* Флаги принятого сообщения */
```

Если для сокета, специфицированного дескриптором `sd`, соединение не установлено, в заданный указателем `msg_name` буфер длины `msg_namelen` помещается исходный адрес сообщения; если этого не нужно, указатель `msg_name` можно сделать пустым. При установлении соединения оба поля игнорируются.

Массив `msg_iov` совместно с полем `msg_iovlen` задает набор буферов для размещения принимаемых данных. Структура типа `iovec` определяется в заголовочном файле `<sys/uio.h>` и содержит по крайней мере два поля.

```
void *iov_base; /* Адрес области памяти (буфера) */
size_t iov_len; /* Размер области памяти (в байтах) */
```

Заданные полем `msg_iov` области памяти по очереди заполняются поступающими данными, пока не будут размещены все принятые данные или не заполнятся все буфера.

Трактовка вспомогательных данных (поля `msg_control` и `msg_controllen` структуры типа `msghdr`) в стандарте POSIX-2001 весьма туманна. Описаны лишь средства доступа к ним. Мы, однако, не будем на этом останавливаться.

В случае успешного завершения функции `recvmsg()` в поле `msg_flags` могут быть установлены следующие флаги.

MSG_EOR

Достигнута граница записи (если это понятие поддерживается протоколом).

MSG_OOB

Получены экстренные данные.

MSG_TRUNC

Пришлось урезать обычные данные.

MSG_CTRUNC

Пришлось урезать управляющие данные.

Подобно тому, как `write()` составляет пару `read()`, функцию `sendto()` можно считать парной по отношению к `recvfrom()`. Она предназначена для отправки через сокет `sd` сообщения, заданного аргументами

message и length, по адресу dest_addr длины dest_len. Если для сокета установлено соединение, целевой адрес dest_addr игнорируется.

В число возможных составляющих значения аргумента flags входят следующие флаги.

MSG_EOR

Обозначает границу записи (если это понятие поддерживается протоколом).

MSG_OOB

Отправляются экстренные данные.

Разумеется, успешное завершение функции sendto() не гарантирует доставку сообщения. Результат, равный -1, указывает только на локально выявляемые ошибки.

Функция send() — пара для recv() со всеми следующими из этого обстоятельства эквивалентностями и аналогиями.

Для функции sendmsg() структура типа msghdr, на которую указывает аргумент message, является входной (что показывает спецификатор const). В поле msg_name задается целевой адрес. Поле msg_flags игнорируется.

Для завершения операций приема и/или отправки данных через сокет служит функция shutdown() (см. листинг 11.29).

```
#include <sys/socket.h>
int shutdown (int sd, int how);
```

Листинг 11.29. Описание функции 2.

Значение аргумента how показывает, что именно завершается: SHUT_RD прекращает прием, SHUT_WR — отpravku, SHUT_RDWR — и то, и другое.

Примеры программ работы с сокетами

Приводимые далее примеры строятся вокруг задачи копирования данных в распределенной программной конфигурации со стандартного ввода одного процесса на стандартный вывод другого, удаленного. Однако начнем мы с локального случая, обслуживаемого сокетами адресного семейства AF_UNIX (см. листинг 11.30).

```
/* * * * * * * * * * * * * * * * * * * * * * * * */
/* Программа копирует строки со стандартного      */
/* ввода на стандартный вывод, "прокачивая" их      */
/* через пару сокетов.                             */
/* Используются функции ввода/вывода нижнего уровня */
/* * * * * * * * * * * * * * * * * * * * * * * * */

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/socket.h>
#include <sys/wait.h>

#define MY_PROMPT    "Вводите строки\n"
#define MY_MSG       "Вы ввели: "

int main (void) {
    int sds [2];
    char buf [1];
    int new_line = 1; /* Признак того, что надо */
    /* выдать сообщение MY_MSG перед отображением */
    /* очередной строки */

    /* Создадим пару соединенных безымянных сокетов */
    if (socketpair (AF_UNIX, SOCK_STREAM, 0, sds)
        < 0) {
        perror ("SOCKETPAIR");
        exit (1);
    }

    switch (fork ()) {
        case -1:
            perror ("FORK");
            exit (2);
        case 0:
            /* Чтение из сокета sds [0] и выдачу на */
            /* стандартный вывод реализуем в порожденном */
            /* процессе */
            while (read (sds [0], buf, 1) == 1) {
                if (write (STDOUT_FILENO, buf, 1) != 1) {
                    perror ("WRITE TO STDOUT");
                    break;
                }
            }
        }
    }
```

```
        }
    }
    exit (0);
}

/* Чтение со стандартного ввода и запись в */
/* сокет sds [1] */
/* возложим на родительский процесс      */
if (write (sds [1], MY_PROMPT,
    sizeof (MY_PROMPT) - 1) !=
    sizeof (MY_PROMPT) - 1) {
    perror ("WRITE TO SOCKET-1");
}

while (read (STDIN_FILENO, buf, 1) == 1) {
    /* Перед отображением очередной строки */
    /* нужно выдать сообщение MY_MSG      */
    if (new_line) {
        if (write (sds [1], MY_MSG,
            sizeof (MY_MSG) - 1) != sizeof (MY_MSG) - 1) {
            perror ("WRITE TO SOCKET-2");
            break;
        }
    }
    if (write (sds [1], buf, 1) != 1) {
        perror ("WRITE TO SOCKET-3");
        break;
    }
    new_line = (buf [0] == '\n');
}
shutdown (sds [1], SHUT_WR);

(void) wait (NULL);
return (0);
}
```

Листинг 11.30. Пример программы, использующей сокеты адресного семейства AF_UNIX.

Обратим внимание на употребление в процессе, осуществляющем запись в сокет, функции `shutdown()`, без чего читающий процесс не определит конец файла и не выйдет из цикла.

Решим теперь ту же задачу для действительно распределенной конфигурации. Наше приложение будет состоять из двух процессов, выполняющихся на разных хостах. Первый процесс (см. листинг 11.31) читает строки со стандартного ввода и отправляет их в виде датаграмм другому, который принимает датаграммы и выдает их содержимое на стандартный вывод (см. листинг 11.32).

```

/* * * * * * * * * * * * * * * * * * * * * * * * * */
/* Программа процесса, читающего строки со          */
/* стандартного ввода и посылающего их в виде        */
/* датаграмм другому процессу (будем называть его    */
/* сервером), который должно выдать их на свой      */
/* стандартный вывод. Имя серверного хоста          */
/* аргумент командной строки                        */
/* * * * * * * * * * * * * * * * * * * * * * * * */

#include <stdio.h>
#include <netdb.h>
#include <sys/socket.h>
#include <string.h>

#define MY_PROMPT    "Вводите строки\n"

int main (int argc, char *argv []) {
    int sd;          /* Дескриптор передающего сокета */
    char line [LINE_MAX]; /* Буфер для копируемых */
                        /* строк */
    /* Структура - аргумент sendmsg */
    struct msghdr msg = {NULL, 0, NULL, 0, NULL, 0,
        0};
    struct iovec iovbuf; /* Структура для сборки */
                        /* отправляемых данных */
    /* Структура - входной аргумент getaddrinfo */
    struct addrinfo hints = {0, AF_INET, SOCK_DGRAM,
        IPPROTO_UDP, 0, NULL, NULL, NULL};
    /* Указатель - выходной аргумент getaddrinfo */
    struct addrinfo *addr_res;
    int res;          /* Результат getaddrinfo */
    int msg_len;      /* Длина очередной введенной */
                        /* строки, включая завершающий нулевой байт */
    if (argc != 2) {
        fprintf (stderr, "Использование: %s"

```

```
        "имя_серверного_хоста\n", argv [0]);
return (1);
}
/* Создадим сокет, через который будем отправлять */
/* прочитанные строки */
if ((sd = socket (AF_INET, SOCK_DGRAM,
    IPPROTO_UDP)) < 0) {
    perror ("SOCKET");
    return (2);
}
/* Выясним целевой адрес для датаграмм */
/* Воспользуемся портом для сервиса spooler */
if ((res = getaddrinfo (argv [1], "spooler",
    &hints, &addr_res)) != 0) {
    fprintf (stderr, "GETADDRINFO: %s\n",
        gai_strerror (res));
    return (3);
}
/* Заполним структуру msghdr */
msg.msg_name = addr_res->ai_addr;
msg.msg_namelen = addr_res->ai_addrlen;
msg.msg_iov = &iobuf;
msg.msg_iovlen = 1;
iobuf.iov_base = line;
/* Цикл чтения строк со стандартного ввода */
/* и отправки их через сокет в виде датаграмм */
fputs (MY_PROMPT, stdout);
while (fgets (line, sizeof (line), stdin) !=
    NULL) {
    msg_len = strlen (line) + 1;
    iobuf.iov_len = msg_len;
    if (sendmsg (sd, &msg, 0) != msg_len) {
        perror ("SENDMSG");
        break;
    }
}

return (0);
}
```

Листинг 11.31. Пример программы, использующей сокеты адресного семейства AF_INET для отправки датаграмм.

```
/* * * * * * * * * * * * * * * * * * * * * * * * * */
/* Программа процесса (будем называть его          */
/* сервером), читающего сообщения (строки) из      */
/* датаграммного сокета и выдающего их на          */
/* стандартный вывод                                */
/* * * * * * * * * * * * * * * * * * * * * * * * */

#include <stdio.h>
#include <netdb.h>
#include <sys/socket.h>

#define MY_MSG      "Вы ввели: "

int main (void) {
    int sd;      /* Дескриптор приемного сокета */
    char line [LINE_MAX]; /* Буфер для копируемых */
                        /* строк */
    /* Структура - аргумент recvmsg */
    struct msghdr msg = {NULL, 0, NULL, 0, NULL,
        0, 0};
    struct iovec iovbuf; /* Структура для разнесения */
                        /* принимаемых данных */
    /* Структура - входной аргумент getaddrinfo */
    struct addrinfo hints = {AI_PASSIVE, AF_INET,
        SOCK_DGRAM, IPPROTO_UDP, 0, NULL, NULL, NULL};
    /* Указатель - выходной аргумент getaddrinfo */
    struct addrinfo *addr_res;
    int res;      /* Результат getaddrinfo */

    /* Создадим сокет, через который будем принимать */
    /* прочитанные строки */
    if ((sd = socket (AF_INET, SOCK_DGRAM,
        IPPROTO_UDP)) < 0) {
        perror ("SOCKET");
        return (1);
    }

    /* Привяжем этот сокет к адресу сервиса spooler */
    /* на локальном хосте */
    if ((res = getaddrinfo (NULL, "spooler", &hints,
        &addr_res)) != 0) {
        fprintf (stderr, "GETADDRINFO: %s\n",
```

```
        gai_strerror (res));
    return (2);
}
if (bind (sd, addr_res->ai_addr,
        addr_res->ai_addrlen) < 0) {
    perror ("BIND");
    return (3);
}

/* Можно освободить память, которую запрашивала */
/* функция getaddrinfo() */
freeaddrinfo (addr_res);

/* Заполним структуру msghdr */
msg.msg_iov = &iobuf;
msg.msg_iovlen = 1;
iobuf.iov_base = line;
iobuf.iov_len = sizeof (line);

/* Цикл приема и выдачи строк */
while (1) {
    if (recvmsg (sd, &msg, 0) < 0) {
        perror ("RCVMSG");
        break;
    }
    fputs (MY_MSG, stdout);
    fputs (line, stdout);
}

return (0);
}
```

Листинг 11.32. Пример программы, использующей сокеты адресного семейства AF_INET для приема датаграмм.

Обратим внимание на несколько моментов. Адреса сокетов (целевого и приемного) получены при помощи функции `getaddrinfo()` с сервисом «spooler» в качестве аргумента. (Если на хосте этот сервис реально используется, для данного примера придется подыскать другой, свободный порт.) С практической точки зрения более правильным было бы пополнить базу данных сетевых сервисов новым элементом, специально предназначенным для представленного приложения, однако подобные

административные действия находятся вне рамок стандарта POSIX и, следовательно, нами не рассматриваются. (Менее правильно, на наш взгляд, формировать адрес сокета покомпонентно, выбирая порт, по сути, случайным образом, поскольку это ведет к неявному, бессистемному, неконтролируемому пополнению базы сервисов.)

Для успешного вызова функции `bind()` процесс должен обладать соответствующими привилегиями. Это может оказаться существенным для запуска программы, показанной в листинге 11.32.

Чтобы получить от `getaddrinfo()` адрес, пригодный для использования в качестве аргумента ориентированных на прием функций (`listen()`, `bind()` для приемного сокета), необходимо указать флаг `AI_PASSIVE` во входной для `getaddrinfo()` структуре `addrinfo` (аргумент `hints` в описании функции `getaddrinfo()`). В таком случае для IP-адреса будет выдано значение `INADDR_ANY`, которое трактуется в адресном семействе `AF_INET` как адрес локального хоста, успешно сопоставляющийся как со шлейфовым (127.0.0.1), так и с реальным сетевыми адресами. В результате через этот сокет можно будет принимать датаграммы, посланные и с локального, и с удаленного хостов.

Цикл приема данных сервером сделан бесконечным, поскольку у последовательности датаграмм нет естественного признака конца.

Для передающего сокета не выполнялась привязка к локальному адресу, а в серверном процессе не запрашивался исходный адрес поступающих датаграмм — подход, возможно, слишком экономный, но непротиворечивый. Заметим, однако, что у представленного приложения есть неиспользованный идейный запас, состоящий в том, что сервер способен принимать датаграммы не только из одного, но и из нескольких источников. Чтобы задействовать этот потенциал, внесем в приложения модификации двух видов.

В программе, посылающей датаграммы, выполним привязку сокета к свободному локальному адресу, воспользовавшись для этого функцией `connect()` и, для оправдания затраченных усилий, изменим способ отправки: вместо универсальной `sendmsg()` задействуем даже не более простую функцию `send()`, а ее вполне привычный аналог `write()`, скрытый под личиной `fputs()`.

Какой режим буферизации выбрать для потока, сформированного по открытому файловому дескриптору сокета, — дело вкуса. Читателю рекомендуется попробовать разные варианты, каждый из которых имеет свои «за» и «против».

Второй вид модификаций касается серверного процесса. Поскольку теперь у датаграмм появился исходный адрес, его можно выяснить и выдать, а заодно опросить выходной флаг `MSG_TRUNC`, чтобы убедиться, что при пересылке в виде датаграмм копируемые строки не были урезаны.

Кроме того, изменение способа формирования отправляемых датаграмм привело к тому, что теперь их содержимое не завершается нулевым байтом; следовательно, способ вывода принимаемых данных также нуждается в модификации (в нашем случае – в дописывании нулевого байта).

Модифицированные варианты программ представлены в листингах 11.33 и 11.34.

```

/* * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Программа процесса, читающего строки со          */
/* стандартного ввода и посылающего их в виде        */
/* датаграмм другому процессу (будем называть его    */
/* сервером), который должно выдать их на свой      */
/* стандартный вывод. Имя серверного хоста -        */
/* аргумент командной строки                         */
/* * * * * * * * * * * * * * * * * * * * * * * * * * */

#include <stdio.h>
#include <netdb.h>
#include <sys/socket.h>

#define MY_PROMPT    "Вводите строки\n"

int main (int argc, char *argv []) {
    int sd;          /* Дескриптор передающего сокета */
    FILE *ss;        /* Поток, соответствующий */
                      /* передающему сокету */
    char line [LINE_MAX]; /* Буфер для копируемых */
                      /* строк */
    /* Структура - входной аргумент getaddrinfo */

    struct addrinfo hints = {0, AF_INET, SOCK_DGRAM,
        IPPROTO_UDP, 0, NULL, NULL, NULL};
    /* Указатель - выходной аргумент getaddrinfo */
    struct addrinfo *addr_res;
    int res;          /* Результат getaddrinfo */

    if (argc != 2) {
        fprintf (stderr, "Использование: %s"
            "имя_серверного_хоста\n", argv [0]);
        return (1);
    }
}

```

```
/* Создадим сокет, через который будем отправлять */
/* прочитанные строки */
if ((sd = socket (AF_INET, SOCK_DGRAM,
    IPPROTO_UDP)) < 0) {
    perror ("SOCKET");
    return (2);
}

/* Выясним целевой адрес для датаграмм */
/* Воспользуемся портом для сервиса spooler */
if ((res = getaddrinfo (argv [1], "spooler",
    &hints, &addr_res)) != 0) {
    fprintf (stderr, "GETADDRINFO: %s\n",
        gai_strerror (res));
    return (3);
}

/* Воспользуемся функцией connect() для достижения */
/* двух целей: фиксации целевого адреса и привязки */
/* к некоему локальному */
if (connect (sd, addr_res->ai_addr,
    addr_res->ai_addrlen) < 0) {
    perror ("CONNECT");
    return (4);
}

/* Сформирует поток по дескриптору сокета */
if ((ss = fdopen (sd, "w")) == NULL) {
    perror ("FDOPEN");
    return (5);
}

/* Отменим буферизацию для этого потока */
setbuf (ss, NULL);

/* Цикл чтения строк со стандартного ввода */
/* и отправки их через сокет в виде датаграмм */
fputs (MY_PROMPT, stdout);
while (fgets (line, sizeof (line),
    stdin) != NULL) {
    fputs (line, ss);
}
```

```

    return (0);
}

```

Листинг 11.33. Модифицированный вариант программы, использующей сокеты адресного семейства AF_INET для отправки датаграмм.

```

/* * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Программа процесса (будем называть его */
/* сервером), читающего сообщения из датаграммного */
/* сокета и копирующего их на стандартный вывод с */
/* указанием адреса, откуда они поступили */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * */

#include <stdio.h>
#include <netdb.h>
#include <sys/socket.h>
#include <arpa/inet.h>

int main (void) {
    int sd; /*Descriptor приемного сокета */
    /* Буфер для принимаемых сообщений */
    /* Оставлено место для вставки нулевого байта */
    char lbuf [BUFSIZ + 1];
        /* Структура - аргумент recvmsg */
    struct msghdr msg = {NULL, 0, NULL, 0, NULL, 0,
        0};
    struct iovec iovbuf; /* Структура для */
        /* разнесения принимаемых данных */
    /* Структура - входной аргумент getaddrinfo */
    struct addrinfo hints = {AI_PASSIVE, AF_INET,
        SOCK_DGRAM, IPPROTO_UDP, 0, NULL, NULL, NULL};
    /* Указатель - выходной аргумент getaddrinfo */
    struct addrinfo *addr_res;
    int res; /* Результат getaddrinfo */
        /* Структура для исходного адреса датаграмм */
    struct sockaddr_in sai;
    ssize_t lmsg; /* Длина принятой датаграммы */

    /* Создадим сокет, через который будем принимать */
    /* прочитанные строки */
    if ((sd = socket (AF_INET, SOCK_DGRAM,
        IPPROTO_UDP)) < 0) {

```

```
    perror ("SOCKET");
    return (1);
}

/* Привяжем этот сокет к адресу сервиса spooler */
/* на локальном хосте */
if ((res = getaddrinfo (NULL, "spooler", &hints,
    &addr_res)) != 0) {
    fprintf (stderr, "GETADDRINFO: %s\n",
        gai_strerror (res));
    return (1);
}
if (bind (sd, addr_res->ai_addr,
    addr_res->ai_addrlen) < 0) {
    perror ("BIND");
    return (2);
}

/* Можно освободить память, которую запрашивала */
/* функция getaddrinfo() */
freeaddrinfo (addr_res);

/* Заполним структуру msghdr */
msg.msg_name = &sai;
msg.msg_namelen = sizeof (struct sockaddr_in);
msg.msg_iov = &iobuf;
msg.msg_iovlen = 1;
iobuf.iov_base = lbuf;
/* Оставим место для вставки нулевого байта */
iobuf.iov_len = sizeof (lbuf) - 1;

/* Цикл приема и выдачи строк */
while (1) {
    if ((lmsg = recvmsg (sd, &msg, 0)) < 0) {
        perror ("RCVMSG");
        break;
    }
    printf ("Вы ввели и отправили с адреса %s,"
        "порт %d :", inet_ntoa (((struct sockaddr_in *)
        msg.msg_name)->sin_addr), ntohs
        (((struct sockaddr_in *)
        msg.msg_name)->sin_port));
```

```

    if (msg.msg_flags & MSG_TRUNC) {
        printf ("Датagramма была урезана\n");
    }
    lbuf [lmsg] = 0;
    fputs (lbuf, stdout);
}

return (0);
}

```

```
#define MY_PROMPT      "Вводите строки\n"

int main (int argc, char *argv []) {
    int sd;           /* Дескриптор передающего сокета */
    FILE *ss;         /* Поток, соответствующий */
                        /* передающему сокету */
    char line [LINE_MAX]; /* Буфер для копируемых */
                        /* строк */
    /* Структура - входной аргумент getaddrinfo */
    struct addrinfo hints = {0, AF_INET,
        SOCK_STREAM, IPPROTO_TCP, 0, NULL, NULL,
        NULL};
    /* Указатель - выходной аргумент getaddrinfo */
    struct addrinfo *addr_res;
    int res;           /* Результат getaddrinfo */

    if (argc != 2) {
        fprintf (stderr, "Использование: %s"
            "имя_серверного_хоста\n", argv [0]);
        return (1);
    }

    /* Создадим сокет, через который будем отправлять */
    /* прочитанные строки */
    if ((sd = socket (AF_INET, SOCK_STREAM,
        IPPROTO_TCP)) < 0) {
        perror ("SOCKET");
        return (2);
    }

    /* Выясним целевой адрес для соединения */
    /* Воспользуемся портом для сервиса spooler */
    if ((res = getaddrinfo (argv [1], "spooler",
        &hints, &addr_res)) != 0) {
        fprintf (stderr, "GETADDRINFO: %s\n",
            gai_strerror (res));
        return (3);
    }

    /* Воспользуемся функцией connect() для */
    /* достижения двух целей: установления соединения */
    /* и привязки к некоему локальному адресу */
}
```

```

    if (connect (sd, addr_res->ai_addr,
        addr_res->ai_addrlen) < 0) {
        perror ("CONNECT");
        return (4);
    }

    /* Сформируем поток по дескриптору сокета */
    if ((ss = fdopen (sd, "w")) != NULL) {
        perror ("FDOPEN");
        return (5);
    }

    /* Отменим буферизацию для этого потока */
    setbuf (ss, NULL);

    /* Цикл чтения строк со стандартного ввода */
    /* и отправки их через сокет в виде потока */
    fputs (MY_PROMPT, stdout);
    while (fgets (line, sizeof (line),
        stdin) != NULL) {
        fputs (line, ss);
    }
    shutdown (sd, SHUT_WR);

    return (0);
}

```

Листинг 11.35. Пример программы, использующей режим с установлением соединения и сокеты типа SOCK_STREAM для пересылки строк.

```

/* * * * * * * * * * * * * * * * * * * * * * * * * */
/* Программа процесса (будем называть его          */
/* демоном), принимающего запросы на установления  */
/* соединения и запускающего процессы для их       */
/* обслуживания                                     */
/* * * * * * * * * * * * * * * * * * * * * * * * * */

#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <netdb.h>
#include <sys/socket.h>

```

```
#include <sys/wait.h>

int main (void) {
    int sd;      /* Дескриптор слушающего сокета */
    int ad;      /* Дескриптор приемного сокета */
    /* Буфер для принимаемых строк */
    struct addrinfo hints = {AI_PASSIVE, AF_INET,
        SOCK_STREAM, IPPROTO_TCP, 0, NULL, NULL,
        NULL};
    /* Указатель - выходной аргумент getaddrinfo */
    struct addrinfo *addr_res;
    int res;     /* Результат getaddrinfo */
    /* Создадим слушающий сокет */
    if ((sd = socket (AF_INET, SOCK_STREAM,
        IPPROTO_TCP)) < 0) {
        perror ("SOCKET");
        return (1);
    }
    /* Привяжем этот сокет к адресу сервиса spooler */
    * на локальном хосте */
    if ((res = getaddrinfo (NULL, "spooler", &hints,
        &addr_res)) != 0) {
        fprintf (stderr, "GETADDRINFO: %s\n",
            gai_strerror (res));
        return (1);
    }
    if (bind (sd, addr_res->ai_addr,
        addr_res->ai_addrlen) < 0) {
        perror ("BIND");
        return (2);
    }
    /* Можно освободить память, которую запрашивала */
    /* функция getaddrinfo() */
    freeaddrinfo (addr_res);

    /* Пометим сокет как слушающий */
    if (listen (sd, SOMAXCONN) < 0) {
        perror ("LISTEN");
        return (3);
    }
    /* Цикл приема соединений и запуска обслуживающих */
    /* процессов */
}
```



```
while (1) {
    /* Примем соединение. */
    /* Адрес партнера по общению нас в данном */
    /* случае не интересует */
    if ((ad = accept (sd, NULL, NULL)) < 0) {
        perror ("ACCEPT");
        return (4);
    }
    /* Запустим обслуживающий процесс */
    switch (fork ()) {
    case -1:
        perror ("FORK");
        return (5);
    case 0:
        /* Сделаем сокет ad стандартным вводом */
        (void) close (STDIN_FILENO);
        (void) fcntl (ad, F_DUPFD, STDIN_FILENO);
        (void) close (ad);
        /* Сменим программу процесса на */
        /* обслуживающую */
        if (execl ("./gsce", "gsce",
            (char *) ) < 0) {
            perror ("EXECL");
            return (6);
        }
    }

    /* В родительском процессе дескриптор принятого */
    /* соединения нужно закрыть */
    (void) close (ad);

    /* Обслужим завершившиеся порожденные процессы */
    while (waitpid ((pid_t) (-1), NULL, WNOHANG)
        > 0) ;
}

return (0);
}
```

Листинг 11.36. Пример программы, принимающей запросы на установление соединения.

```

/* * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Программа обслуживающего процесса, выдающего */
/* принимаемые строки на стандартный вывод. */
/* Дескриптор приемного сокета передан в */
/* качестве стандартного ввода */
/* * * * * * * * * * * * * * * * * * * * * * * * * * */

#include <unistd.h>
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>

int main (void) {
    char line [LINE_MAX]; /* Буфер для принимаемых */
                          /* строк */
    /* Структура для записи адреса */
    struct sockaddr_in sai;
    /* Длина адреса */
    socklen_t sai_len = sizeof (struct sockaddr_in);
    /* Опросим адрес партнера по общению */
    /* (передающего сокета) */
    if (getpeername (STDIN_FILENO,
        (struct sockaddr *) &sai, &sai_len) < 0) {
        perror ("GETPEERNAME");
        return (1);
    }
    /* Цикл чтения строк из сокета */
    /* и выдачи их на стандартный вывод */
    while (fgets (line, sizeof (line),
        stdin) != NULL) {
        printf ("Вы ввели и отправили с адреса %s,"
            "порт %d :",
            inet_ntoa (sai.sin_addr),
            ntohs (sai.sin_port));
        fputs (line, stdout);
    }
    /* Закрывание соединения */
    shutdown (STDIN_FILENO, SHUT_RD);
    return (0);
}

```

Листинг 11.37. Пример программы, обрабатывающей данные, поступающие через сокет типа SOCK_STREAM.

Обратим внимание на два технических момента. Во-первых, дескриптор приемного сокета передается в обслуживающий процесс под видом стандартного ввода. В результате обслуживающая программа по сути свелась к циклу копирования строк со стандартного ввода на стандартный вывод. Мы вернулись к тому, с чего начинали рассмотрение средств буферизованного ввода/вывода, но на новом витке спирали. Во-вторых, в иллюстративных целях для обслуживания завершившихся порожденных процессов использована функция `waitpid()` со значением аргумента `pid`, равным `-1` (выражающим готовность обслужить любой завершившийся процесс-потомок), и флагом `WNOHANG`, означающим отсутствие ожидания. В данном случае подобный прием не гарантирует отсутствия зомби-процессов, но препятствует их накоплению.

Чтобы проиллюстрировать еще один типичный прием, связанный с обработкой запросов на установление соединения, изменим способ обслуживания завершившихся порожденных процессов. Определим функцию обработки сигнала `SIGCHLD` и поместим в нее вызов `waitpid()`. Модифицированный вариант программы процесса-демона, аккуратно ликвидирующий все зомби-процессы, показан в листинге 11.38.

```
/* * * * * * * * * * * * * * * * * * * * * * * * * */
/* Программа процесса (будем называть его          */
/* демоном), принимающего запросы на установления  */
/* соединения и запускающего процессы для их       */
/* обслуживания                                     */
/* * * * * * * * * * * * * * * * * * * * * * * * */

#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <signal.h>
#include <errno.h>

/* Функция обработки сигнала SIGCHLD */
static void chldied (int dummy) {
/* Вдруг число завершившихся потомков отлично */
/* от единицы? */
    while (waitpid ((pid_t) (-1), NULL, WNOHANG) > 0) ;
}
```

```
int main (void) {
    int sd;      /* Дескриптор слушающего сокета */
    int ad;      /* Дескриптор приемного сокета */
    /* Структура - входной аргумент getaddrinfo */
    struct addrinfo hints = {AI_PASSIVE, AF_INET,
        SOCK_STREAM, IPPROTO_TCP, 0, NULL, NULL, NULL};
    /* Указатель - выходной аргумент getaddrinfo */
    struct addrinfo *addr_res;
    int res;     /* Результат getaddrinfo */
    /* Структура для задания реакции на сигнал SIGCHLD */
    struct sigaction sact;

    /* Создадим слушающий сокет */
    if ((sd = socket (AF_INET, SOCK_STREAM,
        IPPROTO_TCP)) < 0) {
        perror ("SOCKET");
        return (1);
    }

    /* Привяжем этот сокет к адресу сервиса spooler */
    /* на локальном хосте */
    if ((res = getaddrinfo (NULL, "spooler", &hints,
        &addr_res)) != 0) {
        fprintf (stderr, "GETADDRINFO: %s\n",
            gai_strerror (res));
        return (1);
    }
    if (bind (sd, addr_res->ai_addr,
        addr_res->ai_addrlen) < 0) {
        perror ("BIND");
        return (2);
    }

    /* Можно освободить память, которую запрашивала */
    /* функция getaddrinfo() */
    freeaddrinfo (addr_res);

    /* Пометим сокет как слушающий */
    if (listen (sd, SOMAXCONN) < 0) {
        perror ("LISTEN");
        return (3);
    }
}
```

```
/* Установим обработку сигнала о завершении */
/* потомка */
sact.sa_handler = chldied;
(void) sigemptyset (&sact.sa_mask);
sact.sa_flags = 0;
(void) sigaction (SIGCHLD, &sact,
    (struct *) NULL);

/* Цикл приема соединений и запуска */
/* обслуживающих процессов */
while (1) {
/* Примем соединение с учетом того, что ожидание */
/* может быть прервано доставкой обрабатываемого */
/* сигнала. */
/* Адрес партнера по общению в данном случае */
/* нас не интересует */
    while ((ad = accept (sd, NULL, NULL)) < 0) {
        if (errno != EINTR) {
            perror ("ACCEPT");
            return (4);
        }
    }

/* Запустим обслуживающий процесс */
    switch (fork ()) {
        case -1:
            perror ("FORK");
            return (5);
        case 0:
            /* Сделаем сокет ad стандартным вводом */
            (void) close (STDIN_FILENO);
            (void) fcntl (ad, F_DUPFD, STDIN_FILENO);
            (void) close (ad);
            /* Сменим программу процесса на */
            /* обслуживающую */
            if (execl ("./gsce", "gsce",
                (char *) NULL) < 0) {
                perror ("EXECL");
                return (6);
            }
        }
    }
}
```



```
#include <arpa/inet.h>
#include <sys/select.h>

/* Структура для хранения дескрипторов приемных */
/* сокетов и ассоциированной информации */
#define MY_MSG_LN    128
struct sads {
    int ad;
    FILE *sd;
    char my_msg [MY_MSG_LN];
};

/*Максимальное число параллельно обслуживаемых */
/*запросов */
#define MY_MAX_QS    FD_SETSIZE

/* Функция для освобождения элемента массива */
/* дескрипторов приемных сокетов */
static void free_elem (struct sads *sads) {
    shutdown (sads->ad, SHUT_RD);
    close (sads->ad);
    sads->ad = -1;
}

int main (void) {
    int sd; /* Дескриптор слушающего сокета */
    /* Структура - входной аргумент getaddrinfo */
    struct addrinfo hints = {AI_PASSIVE, AF_INET,
        SOCK_STREAM, IPPROTO_TCP, 0, NULL, NULL,
        NULL};
    /* Указатель - выходной аргумент getaddrinfo */
    struct addrinfo *addr_res;
    int res; /* Результат getaddrinfo */
    fd_set rdset; /* Набор дескрипторов для чтения */
    /* Структура для записи адреса */
    struct sockaddr_in sai;
    socklen_t sai_len; /* Длина адреса */
    char line [LINE_MAX]; /* Буфер для принимаемых */
    /* строка */
    /* Массив для хранения дескрипторов */
    /* приемных сокетов и ассоциированной */
    /* информации. */
}
```

```
        /* Последний элемент - фиктивный, */
        /* нужен для упрощения поиска свободного */
    struct sads sadsarr [MY_MAX_QS + 1];
    int sads_max; /* Верхняя граница дескрипторов, */
                  /* проверяемых select() */

    int i;
    /* Создадим слушающий сокет */
    if ((sd = socket (AF_INET, SOCK_STREAM,
        IPPROTO_TCP)) < 0) {
        perror ("SOCKET");
        return (1);
    }

    /* Привяжем этот сокет к адресу сервиса spooler */
    /* на локальном хосте */
    if ((res = getaddrinfo (NULL, "spooler", &hints,
        &addr_res)) != 0) {
        fprintf (stderr, "GETADDRINFO: %s\n",
            gai_strerror (res));
        return (1);
    }
    if (bind (sd, addr_res->ai_addr,
        addr_res->ai_addrlen) < 0) {
        perror ("BIND");
        return (2);
    }

    /* Можно освободить память, которую запрашивала */
    /* функция getaddrinfo() */
    freeaddrinfo (addr_res);

    /* Пометим сокет как слушающий */
    if (listen (sd, SOMAXCONN) < 0) {
        perror ("LISTEN");
        return (3);
    }

    /* Инициализируем массив sadsarr. */
    /* -1 в поле ad означает, что элемент свободен */
    for (i = 0; i < (MY_MAX_QS + 1); i++) {
        sadsarr [i].ad = -1;
    }
}
```



```
/* Цикл приема соединений и обслуживания запросов */
while (1) {
/* Подготовим наборы дескрипторов для select() */
    FD_ZERO (&rdset);
    FD_SET (sd, &rdset);
    sads_max = sd + 1;
    for (i = 0; i < MY_MAX_QS; i++) {
        if (sadsarr [i].ad >= 0) {
            FD_SET (sadsarr [i].ad, &rdset);
            if ((sadsarr [i].ad + 1) > sads_max) {
                sads_max = sadsarr [i].ad + 1;
            }
        }
    }

/* Подождем запроса на установление соединения */
/* или готовности данных для чтения. */
/* Время ожидания зададим как бесконечное */
    if (select (sads_max, &rdset, NULL, NULL,
        NULL) == -1) {
        perror ("PSELECT");
        return (4);
    }

/* Посмотрим, есть ли запросы, ждущие */
/* установления соединения */
    if (FD_ISSET (sd, &rdset)) {
/* Примем запрос на установление соединения, */
/* если есть свободный элемент массива */
/* дескрипторов. */
/* Последний элемент считаем фиктивным; */
/* он всегда свободен */
        i = -1;
        while (sadsarr [++i].ad >= 0) ;
        if (i < MY_MAX_QS) {
            /* Свободный элемент найден */
            sai_len = sizeof (struct sockaddr_in);
            if ((sadsarr [i].ad = accept (sd,
                (struct sockaddr *) &sai,
                &sai_len)) == -1) {
                perror ("ACCEPT");
                continue;
            }
        }
    }
}
```

```

        /* Сформируем сообщение, выдаваемое */
        /* перед принятой строкой */
        (void) sprintf (sadsarr [i].my_msg,
            "Вы ввели и отправили с адреса %s,"
            "порт %d :", inet_ntoa (sai.sin_addr),
            ntohs (sai.sin_port));
    /* Сформируем поток по дескриптору сокета */
    /* и отменим буферизацию для этого потока */
    if ((sadsarr [i].sd = fdopen
        (sadsarr [i].ad, "r")) == NULL) {
        perror ("FDOPEN");
        free_elem (&sadsarr [i]);
        continue;
    }
    setbuf (sadsarr [i].sd, NULL);
}
}

/* Посмотрим, есть ли данные, готовые для чтения */
for (i = 0; i < MY_MAX_QS; i++) {
    if ((sadsarr [i].ad >= 0) && FD_ISSET
        (sadsarr [i].ad, &rdset)) {
        /* Есть данные, готовые для чтения, */
        /* или установлен признак конца файла */
        if (fgets (line, sizeof (line),
            sadsarr [i].sd) != NULL) {
            /* Выведем полученную строку */
            fputs (sadsarr [i].my_msg, stdout);
            fputs (line, stdout);
        } else {
            /* Отправитель закрыл соединение. */
            /* Закроем его и мы */
            fclose (sadsarr [i].sd);
            free_elem (&sadsarr [i]);
        }
    }
}
}
return (0);
}

```

Листинг 11.39. Пример программы, мультиплексирующей ввод через сокеты с помощью функции `select()`.

```
/* Инициализируем массив для хранения текущей */
/* строки треугольника Паскаля, чтобы далее все */
/* элементы можно было считать и передавать */
/* единообразно */
tp [0] = 1;
for (i = 1; i < T_SIZE; i++) {
    tp [i] = 0;
}

/* Передадим заголовок */
mtl = T_UCHAR;
if (write (sd, &mtl, 1) != 1) {
    perror ("WRITE-1");
    return (5);
}
mtl = sizeof (T_HDR) - 1;
if (write (sd, &mtl, 1) != 1) {
    perror ("WRITE-2");
    return (6);
}
if (write (sd, T_HDR, mtl) != mtl) {
    perror ("WRITE-3");
    return (7);
}

/* Вычислим и передадим строки треугольника */
/* Паскаля */
for (i = 0; i < T_SIZE; i++) {
    /* Элементы очередной строки нужно считать */
    /* от конца к началу */
    /* Элемент tp [0] пересчитывать не нужно */
    for (j = i; j > 0; j--) {
        tp [j] += tp [j - 1];
    }
    /* Вывод строки треугольника в сокет */
    mtl = T_UINT32;
    if (write (sd, &mtl, 1) != 1) {
        perror ("WRITE-4");
        return (8);
    }
    mtl = i + 1;
    if (write (sd, &mtl, 1) != 1) {
```

```

        perror ("WRITE-5");
        return (9);
    }
    /* Преобразование элементов строки к */
    /* сетевому порядку байт и вывод */
    for (j = 0; j <= i; j++) {
        ntelem = htonl (tp [j]);
        if (write (sd, &ntelem,
            sizeof (ntelem)) != (ntelem)) {
            perror ("WRITE-6");
            return (10);
        }
    }
}
shutdown (sd, SHUT_WR);

return (close (sd));
}

```

Листинг 11.40. Пример программы, передающей через сокеты целочисленные данные.

```

/* * * * * * * * * * * * * * * * * * * * * */
/* Программа процесса (будем называть его */
/* наивным сервером), принимающего запросы на */
/* установления соединения и осуществляющего вывод */
/* поступающих числовых данных без порождения */
/* процессов и мультиплексирования */
/* * * * * * * * * * * * * * * * * * * * * */

#include <unistd.h>
#include <stdio.h>
#include <netdb.h>
#include <sys/socket.h>
#include <arpa/inet.h>

/* Маркеры типов передаваемых данных */
#define T_UCHAR      1
#define T_UINT16     2
#define T_UINT32     4

int main (void) {

```

```
int sd; /* Дескриптор слушающего сокета */
int ad; /* Дескриптор приемного сокета */
/* Структура - входной аргумент getaddrinfo */
struct addrinfo hints = {AI_PASSIVE, AF_INET,
    SOCK_STREAM, IPPROTO_TCP, 0, NULL, NULL,
    NULL};
/* Указатель - выходной аргумент getaddrinfo */
struct addrinfo *addr_res;
int res; /* Результат getaddrinfo */
/* Структура для записи адреса */
struct sockaddr_in sai;
socklen_t sai_len; /* Длина адреса */
char mt; /* Маркер типа поступающих данных */
char ml; /* Маркер кратности поступающих данных */
char bufc; /* Буфер для приема символьных */
/* данных */
uint16_t buf16; /* Буфер для приема */
/* 16-разрядных целых */
uint32_t buf32; /* Буфер для приема */
/* 16-разрядных целых */

/* Создадим слушающий сокет */
if ((sd = socket (AF_INET, SOCK_STREAM,
    IPPROTO_TCP)) < 0) {
    perror ("SOCKET");
    return (1);
}

/* Привяжем этот сокет к адресу сервиса spooler */
/* на локальном хосте */
if ((res = getaddrinfo (NULL, "spooler", &hints,
    &addr_res)) != 0) {
    fprintf (stderr, "GETADDRINFO: %s\n",
        gai_strerror (res));
    return (1);
}
if (bind (sd, addr_res->ai_addr,
    addr_res->ai_addrlen) < 0) {
    perror ("BIND");
    return (2);
}
```

```
/* Можно освободить память, которую запрашивала */
/* функция getaddrinfo() */
freeaddrinfo (addr_res);

/* Пометим сокет как слушающий */
if (listen (sd, SOMAXCONN) < 0) {
    perror ("LISTEN");
    return (3);
}

/* Цикл приема соединений и обслуживания */
/* запросов на вывод */
while (1) {
    /* Примем соединение */
    sai_len = sizeof (struct sockaddr_in);
    if ((ad = accept (sd, (struct sockaddr *) &sai,
        &sai_len)) < 0) {
        perror ("ACCEPT");
        continue;
    }

    /* Цикл приема поступающих данных и их вывода */
    printf ("Получено с адреса %s, порт %d :",
        inet_ntoa (sai.sin_addr), ntohs
            (sai.sin_port));
    while (read (ad, &mt, 1) == 1) {
        /* Есть очередная порция данных, */
        /* начинающаяся с типа. */
        /* Прочитаем кратность, потом сами данные */
        if (read (ad, &ml, 1) != 1) {
            perror ("READ-1");
            return (4);
        }
        while (ml-- > 0) {
            switch (mt) {
                case T_UCHAR:
                    if (read (ad, &bufc,
                        sizeof (bufc)) != sizeof (bufc)) {
                        perror ("READ-2");
                        return (5);
                    }
                    printf ("%c", bufc);
            }
        }
    }
}
```

```
        continue;
    case T_UINT16:
        if (read (ad, &buf16,
            sizeof (buf16)) != sizeof (buf16)) {
            perror ("READ-3");
            return (6);
        }
        printf (" %d", ntohs (buf16));
        continue;
    case T_UINT32:
        if (read (ad, &buf32,
            sizeof (buf32)) != sizeof (buf32)) {
            perror ("READ-4");
            return (7);
        }
        printf (" %d", ntohl (buf32));
        continue;
    }
}

/* Вывод порции завершим переводом строки */
printf ("\n");
}

/* Конец обслуживания соединения */
(void) close (ad);
}

return (0);
}
```

Листинг 11.41. Пример программы, принимающей через сокеты целочисленные данные.

Вариант 1

1. **Адресное семейство AF_UNIX поддерживает межпроцессное взаимодействие**
 - ☐ в пределах одной системы
 - ☐ между системами с одной версией ОС Unix
 - ☐ между произвольными Unix-системами
2. **Согласно стандарту POSIX-2001, функция sethostent()**
 - ☐ устанавливает входной хост сети
 - ☐ формирует элемент базы данных о хостах как узлах сети
 - ☐ устанавливает соединение с базой данных о хостах как узлах сети
3. **Согласно стандарту POSIX-2001, для получения сетевого адреса хоста по его имени рекомендуется использовать функции**
 - ☐ gethostbyname()
 - ☐ getaddrinfo()
 - ☐ gethostent()

Вариант 2

1. **Адресное семейство AF_INET поддерживает взаимодействие по протоколам**

- ☐ IPv4
- ☐ IPv6
- ☐ произвольной версии IP

2. **Согласно стандарту POSIX-2001, функция gethostent()**

- ☐ опрашивает входной хост сети
- ☐ читает элемент базы данных о хостах как узлах сети
- ☐ опрашивает характеристики соединения с базой данных о хостах как узлах сети

3. **Согласно стандарту POSIX-2001, для получения имени хоста по его сетевому адресу рекомендуется использовать функции**

- ☐ gethostbyaddr()
- ☐ getnameinfo()
- ☐ gethostent()

Вариант 3

1. **Адресное семейство AF_INET6 поддерживает взаимодействие по протоколам**
 - ☐ IPv4
 - ☐ IPv6
 - ☐ произвольной версии IP

2. **Согласно стандарту POSIX-2001, функция endhostent()**
 - ☐ закрывает входной хост сети
 - ☐ закрывает элемент базы данных о хостах как узлах сети
 - ☐ закрывает соединение с базой данных о хостах как узлах сети

3. **Согласно стандарту POSIX-2001, для получения адреса сетевого сервиса по его имени рекомендуется использовать функцию**
 - ☐ getservbyname()
 - ☐ getaddrinfo()
 - ☐ getservent()

Лекция 12. Время и работа с ним

Рассматриваются служебные программы и функции, предназначенные для работы с часами и таймерами реального и процессорного времени, средства приостановки выполнения процессов на заданное время, а также опрос и изменение данных о времени, ассоциированных с файлами.

Ключевые слова: время, реальное время, астрономическое время, всемирное время, начало отсчета, момент времени, секунда, минута, час, день, месяц, год, високосный год, часовой пояс, поясное время, летнее время, часы, разрешающая способность часов, скачок часов, монотонные часы, такт часов, время выполнения, процессорное время, часы процессорного времени, мониторинг времени выполнения, виртуальное время процесса, таймер, таймер процессорного времени, взвести таймер, снять таймер со взвода, избыточное срабатывание таймера, показания часов, опрос показаний часов, спецификатор преобразования данных о времени, установка показаний часов, проблема 2038-го года, идентификатор часов, общесистемные часы, представление данных о времени, преобразование представлений данных о времени, стандарт ISO 8601:2000, время выполнения простой команды, процессорное время, затраченное процессом, процессорное время, затраченное системой на обслуживание процесса, процессорное время, затраченное порожденными процессами, идентификатор часов процессорного времени, манипулирование часами процессорного времени, опрос данных о файле, изменение ассоциированных с файлом данных о времени, время последнего доступа, время последнего изменения, время последнего изменения статуса, текущее время, ожидание наступления событий, активное ожидание, пассивное ожидание, приостановка выполнения, приостановка выполнения на заданное время, приостановка выполнения процесса, приостановка выполнения потока управления, интервальный таймер, таймер реального времени, таймер виртуального времени процесса, таймер профилирования, интервал таймера, текущие показания таймера, обратный отсчет, идентификатор таймера, скрытый канал по времени, управление ходом выполнения программы.

Основные понятия и объекты

Согласно стандарту POSIX, за начало отсчета (астрономического) времени принимается ноль часов, ноль минут, ноль секунд первого января 1970-го года универсального координированного времени (далее для краткости именуемого также всемирным).

Всемирным называют поясное время нулевого часового пояса, которое представляет собой местное среднее солнечное время гринвичского меридиана.

Московское время опережает всемирное на три часа. Имеется в виду так называемое декретное московское время, которое не следует путать с поясным.

Поясное время Москвы, лежащей во втором часовом поясе, опережает всемирное на два часа.

Московское летнее время опережает всемирное на 4 часа.

За стандартную единицу измерения астрономического времени в POSIX-2001 принята секунда. Пусть некоторый момент времени задан в терминах секунд (значение `tm_sec`), минут (`tm_min`), часов (`tm_hour`), а также номером дня в году (`tm_yday`) и номером года минус 1900 (`tm_year`). Тогда число секунд всемирного времени, прошедшее на этот момент от начала отсчета, вычисляется по формуле

$$\begin{aligned} & \text{tm\_sec} + \text{tm\_min} * 60 + \text{tm\_hour} * 3600 + \text{tm\_yday} * 86400 \\ & + (\text{tm\_year} - 70) * 31536000 + ((\text{tm\_year} - 69) / 4) * 86400 - \\ & ((\text{tm\_year} - 1) / 100) * 86400 + ((\text{tm\_year} + 299) / 400) * 86400 \end{aligned}$$

(три последних слагаемых предназначены для учета високосных лет).

Учет разного рода поправок к всемирному времени (таких, например, как вставка дополнительной секунды в начале 1996 года) зависит от реализации.

Для представления данных о моментах времени в программах на языке C обычно используется структура `tm`, описанная во включаемом файле `<time.h>` и содержащая, согласно стандарту POSIX-2001, по крайней мере следующие поля.

```
int tm_sec;      /* Секунды [0, 60] */
int tm_min;      /* Минуты [0, 59] */
int tm_hour;     /* Часы [0, 23] */
int tm_mday;     /* Номер дня в месяце [1, 31] */
int tm_mon;      /* Номер месяца в году [0, 11] */
int tm_year;     /* Номер года минус 1900 */
```

```
int tm_wday; /* Номер дня недели [0, 6] */  
              /* (воскресенью соответствует ноль) */  
int tm_yday; /* Номер дня в году [0, 365] */  
int tm_isdst; /* Признак учета летнего времени */
```

Диапазон [0, 60] для поля `tm_sec` позволяет справляться с производимой время от времени вставкой дополнительной секунды.

Значение поля `tm_year` задается со знаком, что позволяет представлять годы до 1900.

Значение признака `tm_isdst` должно быть положительным, если действует летнее время, нулевым, если летнее время не действует, или отрицательным, если про летнее время ничего не известно.

Часами называется программный или аппаратный объект, предназначенный для измерения видимого или истинного хода времени.

Показания часов можно опросить и установить (в допустимых для часов пределах).

Разрешающей способностью часов называется минимальный промежуток времени, который может быть этими часами измерен.

Под скачком часов понимается разность между двумя последовательными, различными с точки зрения приложения (использующего стандартные средства) показаниями часов.

Часы называются монотонными, если их нельзя установить стандартными средствами и они не могут иметь отрицательных скачков.

Такт часов – это зависящие от реализации промежутки времени, на которые дробится каждая секунда.

Реальным (или астрономическим) называется время, измеренное по системным часам безотносительно к тому, какой процесс (поток управления) выполняется.

Под временем выполнения (процессорным временем) понимается время, затрачиваемое на выполнение процесса (потока управления), включая работающие от его имени системные сервисы.

Время выполнения конкретного процесса или потока управления измеряется часами процессорного времени.

Под мониторингом времени выполнения понимается оперативное отслеживание процессорного времени, затрачиваемого процессом (поток управления).

Виртуальное время процесса – время, измеряемое системными часами, пока процесс выполняется.

Таймер – это механизм, способный известить процесс (поток управления) об истечении заданного временного промежутка (подобный таймер называется интервальным) или о достижении (или превышении) часами

заданных показаний (абсолютный таймер). Соответствующее событие называется срабатыванием таймера.

Таймером процессорного времени называется таймер, ассоциированный с часами процессорного времени.

Взвести (зарядить) — значит запустить таймер, измеряющий ход времени и позволяющий уведомить процесс о наступлении заданного момента.

Таймер снимается со взвода (разряжается), когда он перестает измерять ход времени, отключая тем самым будущие уведомления процесса до нового взведения таймера.

Таймеры можно подразделить на одноразовые и периодические.

Одноразовые таймеры снимаются со взвода после первого (и единственного) срабатывания.

Периодические таймеры после очередного срабатывания запускаются вновь в соответствии с однажды заданной спецификацией следующего срабатывания.

Под избыточным срабатыванием таймера понимается ситуация, когда таймер срабатывает до того, как процесс обработал сигнал о предыдущем срабатывании.

Все, что касается часов процессорного времени и таймеров, относится к необязательным частям стандарта POSIX-2001. Тем не менее, мы детально рассмотрим соответствующие средства ввиду их важности и употребительности.

Опрос и установка показаний часов реального времени

Простейшим средством опроса и/или изменения текущих даты и времени является служебная программа `date`:

```
date [-u] [+формат]
date [-u] ммддччмм[ [вв]гг]
```

В первой форме утилита `date` выдает на стандартный вывод дату и время (по умолчанию — текущие), в соответствии с заданным форматом. Во второй форме `date` позволяет установить системные дату и время.

При наличии опции `-u` работа ведется без учета часового пояса, во всемирном времени (как если бы значением переменной окружения `TZ` была цепочка «UTC0»). Без этой опции учитывается значение `TZ` или, если таковое не установлено или пусто, принимается во внимание системное подразумеваемое значение.

Если в командной строке `date` задан аргумент, начинающийся со знака `+`, форматом вывода даты и времени управляет пользователь. Формат включает спецификаторы преобразований и прочие символы (последние

выводятся без изменений). Спецификаторы преобразований начинаются со знака % и замещаются при выводе соответствующим значением.

%a

Сокращенное название дня недели.

%A

Полное название дня недели.

%b

Сокращенное название месяца.

%B

Полное название месяца.

%c

Принятое в данной языково-культурной среде представление даты и времени.

%C

Две первые цифры четырехзначного номера года [00, 99].

%d

Номер дня в месяце [01, 31].

%D

Дата в формате мм/дд/гг.

%e

Номер дня в месяце [1, 31] в двухсимвольном поле с дополнением, при необходимости, пробела.

%h

То же, что %b.

%H

Номер часа [00, 23].

%I

Номер часа [01, 12].

%j

Номер дня в году [001, 366].

%m

Номер месяца [01, 12].

%M

Минуты [00, 59].

%n

Перевод строки

%S

Секунды [00, 60].

%t

Табуляция

%T

Время в формате чч:мм:сс.

%u

Номер дня недели [1, 7] (единица соответствует понедельнику).

%U

Номер недели в году [00, 53]. Первым днем недели считается воскресенье. Все дни нового года, предшествующие первому воскресенью, относятся к нулевой неделе.

%V

Номер недели в году [01, 53]. Первым днем недели считается понедельник. Если первое января приходится на пятницу, субботу или воскресенье, конец недели формально относят к прошлому году, а первой считают следующую неделю.

%w

Номер дня недели [0, 6] (ноль соответствует воскресенью).

%W

Номер недели в году [00, 53]. Первым днем недели считается понедельник. Все дни нового года, предшествующие первому понедельнику, относятся к нулевой неделе.

%x

Принятое в данной языково-культурной среде представление даты.

%X

Принятое в данной языково-культурной среде представление времени.

%y

Две младшие цифры номера года [00, 99].

%Y

Четырехзначный номер года.

%Z

Имя часового пояса.

%%

Знак процента.

Чтобы установить текущие дату и время, необходимо, чтобы система допускала данное действие, а у процесса были соответствующие привилегии.

Во второй форме вызова утилиты `date` первая пара букв `m` обозначает номер месяца, `dd` — номер дня в месяце, `hh` — часы, `mm` — минуты, `vvvv` — год. Если две первые цифры года опущены, то диапазон `[69, 99]` трактуется как `[1969, 1999]`, а `[00, 68]` — как `[2000, 2068]`. Если год не указан, имеется в виду текущий.

Приведем несколько примеров употребления служебной программы `date`.

Вывод в подразумеваемом формате, используемом командой

```
date
```

без аргументов, может выглядеть так:

```
Fri Dec 26 17:48:02 MSK 2003
```

Опция `-u` повлияет на вывод следующим образом:

```
Fri Dec 26 14:48:05 UTC 2003
```

Команда

```
date '+Дата: %d.%m.%Y\nВремя: %H:%M:%S'
```

выдаст примерно следующее:

```
Дата: 26.12.2003
```

```
Время: 17:49:15
```

Для сдвига показаний системных часов на один час вперед можно воспользоваться `shell`-процедурой, показанной в листинге 12.1 (предполагается, что ее выполняют днем при наличии соответствующих привилегий).

```
moda=`date +%m%d`  
ho=$((`date +%H` + 1))  
mi=`date +%M`  
date ${moda}${ho}${mi}
```

Листинг 12.1. Пример использования служебной программы `date`.

На уровне языка C для опроса текущего времени (в секундах от начала отсчета) служит функция `time()` (см. листинг 12.2).

```
#include <time.h>  
time_t time (time_t *tloc);
```

Листинг 12.2. Описание функции `time()`.

Если значение аргумента `tloc` отлично от пустого указателя, то текущее время не только возвращается в качестве результата, но еще и записывается по заданному адресу.

Отметим, что если тип `time_t` реализован как 32-разрядное целое со знаком, то в 2038-м году наступит переполнение (так называемая проблема 2038-го года). Стандарт POSIX-2001 не справляется с проблемой в полной мере, но облегчает ее решение за счет использования более или менее абстрактного типа `time_t` вместо какой-либо конкретной разновидности `int`. Вероятно, в последующие версии стандарта POSIX будут включены требования к способности значений типа `time_t` представлять время в далеком будущем (например, вплоть до 10000-го года) или явное требование не менее чем 64-разрядности.

С функцией `time()` ассоциирована функция `difftime()`, вычисляющая (в виде значения типа `double`) разность в секундах между двумя моментами времени (`time1 - time0`, см. листинг 12.3).

```
#include <time.h>  
double difftime (time_t time1, time_t time0);
```

Листинг 12.3. Описание функции `difftime()`.

Узнать текущее время с большей точностью позволяет функция `gettimeofday()` (см. листинг 12.4).

```
#include <sys/time.h>
int gettimeofday (struct timeval *restrict tp,
    void *restrict tzp);
```

Листинг 12.4. Описание функции `gettimeofday()`.

Согласно стандарту POSIX-2001, описанная в заголовочном файле `<sys/time.h>` структура `timeval` содержит по крайней мере следующие поля.

```
time_t    tv_sec;    /* Секунды */
suseconds_t tv_usec; /* Микросекунды */
```

Функция `gettimeofday()` записывает текущее время, выраженное в секундах и микросекундах от начала отсчета, в структуру типа `timeval`, на которую указывает аргумент `tp`. Указатель `tzp` должен быть пустым.

Впрочем, микросекунды — это еще на предел точности, а опрос системных часов — не предел общности. В необязательную часть стандарта POSIX-2001, регламентирующую работу с таймерами, входят функции, позволяющие опросить и установить показания заданных часов, а также узнать их разрешающую способность (см. листинг 12.5).

```
#include <time.h>
int clock_getres (clockid_t clock_id,
    struct timespec *res);
int clock_gettime (clockid_t clock_id,
    struct timespec *tp);
int clock_settime (clockid_t clock_id,
    const struct timespec *tp);
```

Листинг 12.5. Описание функций опроса характеристик и установки часов.

Структура `timespec` отличается от `timeval` тем, что вместо микросекунд хранит наносекунды:

```
time_t tv_sec; /* Секунды */
long tv_nsec; /* Наносекунды */
```

Функция `clock_getres()` позволяет опросить разрешающую способность часов с заданным идентификатором и поместить результат в структуру, на которую указывает аргумент `res`.

Функция `clock_gettime()` записывает в аналогичную структуру текущие показания заданных часов.

Наконец, функция `clock_settime()` устанавливает заданные часы на основании значений полей структуры типа `timespec`, на которую указывает аргумент `tr`.

Часы могут быть общесистемными, т. е. видимыми для всех процессов, или принадлежащими процессу (измеряют время, имеющее смысл только для конкретного процесса). Все реализации должны поддерживать общесистемные часы реального времени, измеряющие время от начала отсчета и имеющие идентификатор `CLOCK_REALTIME`. Интерпретация показаний других часов (если таковые имеются) стандартом не специфицируется.

Установка новых показаний для часов `CLOCK_REALTIME` влияет на срабатывание таймеров, взведенных на абсолютное время, но не затрагивает интервальные таймеры.

Если в системе поддерживаются монотонные часы (с идентификатором `CLOCK_MONOTONIC`), они измеряют время от некоторого неспецифицированного момента в прошлом (например, от загрузки системы или начала отсчета). Монотонные часы предназначены для измерения относительного времени. По определению, их показания нельзя установить с помощью функции `clock_settime()`.

Применение описанных выше функций иллюстрируется программой, показанной в листинге 12.6. Возможно, при редактировании ее внешних связей придется указать дополнительные библиотеки (например, `libc`), что является следствием использования необязательных компонентов стандарта POSIX.

```
#include <stdio.h>
#include <time.h>
#include <sys/time.h>

int main (void) {
    struct timespec tmsp;
    struct timeval tmvl;
    time_t st;
    double s = 0;
    double d = 1;
    int i;
```

```
if (clock_getres (CLOCK_REALTIME, &tmsp) == -1) {
    perror ("CLOCK_GETRES");
    return (1);
}
printf ("Разрешающая способность общесистемных"
        "часов: %ld нсек.\n",
        tmsp.tv_nsec);

if (clock_gettime (CLOCK_REALTIME, &tmsp) == -1) {
    perror ("CLOCK_GETTIME");
    return (2);
}
printf ("Текущее время по общесистемным часам:"
        "%ld сек. %ld нсек.\n",
        tmsp.tv_sec, tmsp.tv_nsec);

(void) gettimeofday (&tmvl, NULL);
printf ("Текущее время, функция gettimeofday():"
        "%ld сек. %ld мсек.\n",
        tmvl.tv_sec, tmvl.tv_usec);

(void) time (&st);
for (i = 1; i <= 100000000; i++) {
    s += d / i;
    d = -d;
}
printf ("Время выполнения цикла: %g сек.\n",
        difftime (time (NULL), st));

return (0);
}
```

Листинг 12.6. Пример программы, использующей функции опроса показаний часов реального времени.

Результаты работы этой программы могут выглядеть так, как показано в листинге 12.7.

```
Разрешающая способность общесистемных часов:
100000000 нсек.
Текущее время по общесистемным часам: 1072534678
сек. 296598000 нсек.
```

Текущее время, функция `gettimeofday()`: 1072534678 сек. 296637 мксек.

Время выполнения цикла: 15 сек.

Листинг 12.7. Возможные результаты работы программы, использующей функции опроса показаний часов реального времени.

Преобразование данных о времени

Стандартом POSIX-2001 предусмотрено несколько способов представления данных о времени. Выше были описаны структура `tm` и тип `time_t`. Кроме того, время может записываться в виде цепочки символов. (Есть еще структуры `timeval` и `timespec`, но они в данном контексте играют подчиненную роль, лишь уточняя значения типа `time_t`.)

Для выполнения преобразований между разными представлениями данных о времени служат описываемые далее функции (см. также рис. 12.1).

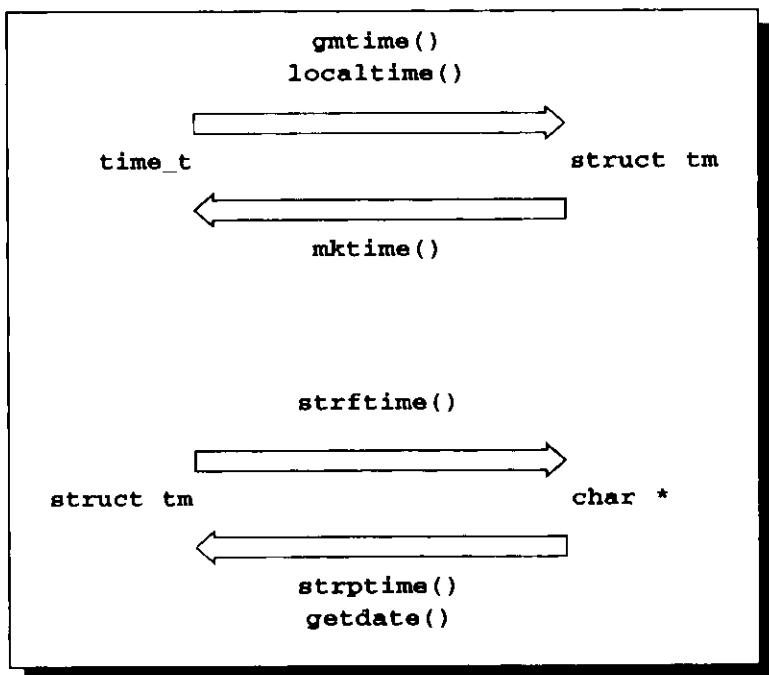


Рис. 12.1. Функции для выполнения преобразований между разными представлениями данных о времени.

Функции `gmtime()` и `localtime()` (см. листинг 12.8) преобразуют значения типа `time_t` в структуру типа `tm`. Соотношение между временем в секундах от начала отсчета и значениями полей структуры типа `tm` дается в приведенной выше формуле. Кроме того, функция `localtime()` учитывает данные о часовом поясе и сезонных поправках.

```
#include <time.h>
struct tm *gmtime (const time_t *tloc);
struct tm *localtime (const time_t *tloc);
```

Листинг 12.8. Описание функций `gmtime()` и `localtime()`.

Для учета данных о часовом поясе и сезонных поправках используются внешние переменные `tzname`, `timezone`, `daylight`, значения которых функция `tzset()` устанавливает по переменной окружения `TZ` (см. листинг 12.9).

```
#include <time.h>
extern char *tzname[2];
extern long timezone;
extern int daylight;
void tzset (void);
```

Листинг 12.9. Описание функции `tzset()` и ассоциированных внешних переменных.

Элементам массива `tzname` присваиваются имена местного часового пояса в стандартном (`tzname [0]`) и «летнем» (`tzname [1]`) вариантах. Значение переменной `timezone` устанавливается равным разности в секундах между всемирным и местным поясным временем. Переменной `daylight` присваивается отличное от нуля значение, если для местного часового пояса предусмотрен переход на летнее время.

Отметим, что в общем случае значение переменной окружения `TZ` устроено довольно сложным образом:

```
Станд_поясСмещение[Лет_пояс[Смещение]
[ , Нач_лет[/Время] , Кон_лет[/Время] ]]
```

Здесь `Станд_пояс` и `Лет_пояс` — имена, присваиваемые элементам массива `tzname []`, `Смещение` — разность между всемирным и местным поясным временем (в виде `чч[:мм[:сс]]`), `Нач_лет` и `Кон_лет`, соответственно, даты начала и окончания действия летнего времени (обычно

их задают в виде Ммм.н.д — месяц, неделя, день), Время — время перехода (по умолчанию — два часа ночи). Можно видеть, что данных для вычисления местного времени оказывается вполне достаточно.

Функцию `mktime()` (см. листинг 12.10) можно считать обратной по отношению к `localtime()`. Она преобразует местное время, заданное в виде структуры типа `tm`, в значение типа `time_t`, т. е. в секунды от начала отсчета по всемирному времени.

```
#include <time.h>
time_t mktime (struct tm *tmptr);
```

Листинг 12.10. Описание функции `mktime()`.

При входе в функцию `mktime()` значения полей `tm_wday` и `tm_yday` структуры `tm`, на которую указывает аргумент `tmptr`, игнорируются; при выходе они устанавливаются должным образом. Значения других полей также приводятся к стандартным для них диапазонам (при входе это условие может не выполняться).

Другим весьма мощным средством преобразования местного времени из структурного в текстовое представление является функция `strftime()` (см. листинг 12.11). Как и служебная программа `date`, она преобразует дату и время в соответствии с заданным форматом, только исходными данными служит не текущий момент времени, а структура типа `tm`, на которую указывает аргумент `tmptr`, и результат направляется не на стандартный вывод, а в буфер, заданный указателем `s` и длиной `maxsize`.

```
#include <time.h>
size_t strftime (char *restrict s, size_t maxsize,
    const char *restrict format, const struct tm
    *restrict tmptr);
```

Листинг 12.11. Описание функции `strftime()`.

По сравнению с `date`, функция `strftime()` содержит несколько дополнительных спецификаторов форматных преобразований. Перечислим наиболее употребительные из них.

%F

Представление даты в соответствии со стандартом ISO 8601:2000: эквивалент последовательности спецификаторов `%Y-%m-%d`.

%R

Часы и минуты (%H: %M) в 24-часовом представлении.

%z

Смещение относительно всемирного времени, представленное по стандарту ISO 8601:2000: +ччммм или -ччммм (положительные значения соответствуют часовым поясам к востоку от Гринвича).

Функция `strptime()` возвращает число байт, помещенное в буфер (без учета завершающего нулевого байта). Если буфер оказался мал, возвращается ноль.

На роль обратной по отношению к `strptime()` могут претендовать сразу две функции: `strptime()` и `getdate()` (см. листинг 12.12).

```
#include <time.h>
char *strptime(const char *restrict s, const char
               *restrict format, struct tm *restrict tmpr);
struct tm *getdate(const char *s);
```

Листинг 12.12. Описание функций `strptime()` и `getdate()`.

Функция `strptime()` напоминает `scanf()`: она сканирует цепочку символов, на которую указывает аргумент `s`, в соответствии с заданным форматом, включающим описанные выше спецификаторы преобразования, а также, быть может, пробельные и обычные символы, и помещает извлеченные значения в структуру типа `tm` по указателю `tmpr`. В качестве результата возвращается указатель на первый несканированный символ или `NULL` в случае неудачи.

Функция `getdate()`, по сути аналогичная `strptime()`, использует для разбора входной цепочки `s` форматы, содержащиеся в файле, чье полное маршрутное имя задано переменной окружения `DATETIME` (для интерпретации выбирается первый подходящий формат). Если дата и время специфицированы не полностью (например, задан только день недели), как исходные берутся данные о первом подходящем моменте времени, начиная с текущего. Если в формате присутствует спецификатор `%Z`, выходная структура инициализируется текущим временем в сканируемом часовом поясе. В противном случае применяется местное время.

С помощью внешней переменной (или макроса) `getdate_err` функция `getdate()` возвращает коды ошибок.

Приведем пример использования описанных функций преобразования данных о времени (см. листинг 12.13).

```
#include <stdio.h>
#include <time.h>
#include <limits.h>

int main (void) {
    char dtbuf [LINE_MAX]; /* Буфер для данных о */
                           /* времени */

    time_t st;
    struct tm stm;

    (void) time (&st);

    (void) strftime (dtbuf, sizeof (dtbuf), "%c",
        gmtime (&st));
    printf ("Текущее всемирное время: %s\n", dtbuf);

    (void) strftime (dtbuf, sizeof (dtbuf), "%c",
        localtime (&st));
    printf ("Текущее местное время: %s\n", dtbuf);

    /* Узнаем, каким днем недели будет 01 января */
    /* 2038 года */
    stm.tm_year = 2038 - 1900;
    stm.tm_mon = 1 - 1;
    stm.tm_mday = 1;
    stm.tm_hour = 0;
    stm.tm_min = 0;
    stm.tm_sec = 0;
    stm.tm_isdst = -1;
    if ((st = mktime (&stm)) == (time_t) (-1)) {
        perror ("MKTIME");
    } else {
        (void) strftime (dtbuf, sizeof (dtbuf), "%A",
            &stm);
        printf ("День недели 01 января 2038 года:"
            "%s\n", dtbuf);
        printf ("Число секунд от начала отсчета в"
            "начале 2038 года (шест.): %x\n",
            (unsigned int) st);
    }
}
```

```
/* Узнаем, когда наступит переполнение значений */
/* типа time_t, представленных как 32-разрядное */
/* целое со знаком */
st = (time_t) 0xffffffff;
(void) strftime (dbuf, sizeof (dbuf), "%c",
    gmtime (&st));
printf ("Всемирное время конца 32-разрядного"
    "отсчета: %s\n", dbuf);

/* Преобразуем эту дату в формат ISO 8601:2000 */
if (strptime (dbuf, "%c", &stm) == NULL) {
    perror ("STRPTIME");
} else {
    (void) strftime (dbuf, sizeof (dbuf), "%F",
        &stm);
    printf ("Дата конца 32-разрядного отсчета в"
        "формате ISO 8601:2000: %s\n",
            dbuf);
}

return (0);
}
```

Листинг 12.13. Пример программы, использующей функции преобразования данных о времени.

Возможные результаты выполнения этой программы показаны в листинге 12.14.

```
Текущее всемирное время: Sat Jan 3 13:54:02 2004
Текущее местное время: Sat Jan 3 16:54:02 2004
День недели 01 января 2038 года: Friday
Число секунд от начала отсчета в начале 2038 года
(шест.): 7fe7ed50
Всемирное время конца 32-разрядного отсчета: Tue
Jan 19 03:14:07 2038
Дата конца 32-разрядного отсчета в формате ISO
8601:2000: 2038-01-19
```

Листинг 12.14. Возможные результаты работы программы, использующей функции преобразования данных о времени.

Последовательность инструкций языка C

```
char dtbuf [];  
time_t st;  
(void) strftime (dtbuf, sizeof (dtbuf), "%c",  
    localtime (&st));
```

является стандартной для получения текущего времени в текстовом виде. (Разумеется, вместо «%c» допустим другой спецификатор преобразования.) Нужно помнить только, что функции `gmtime()` и `localtime()` возвращают указатели на статические структуры, содержимое которых, возможно, перезаписывается при каждом вызове, поэтому, если оно понадобится в дальнейшем, его следует скопировать в собственные объекты.

Опрос показаний часов процессорного времени

Базовым средством для работы с часами процессорного времени является функция `clock()` (см. листинг 12.15). Она возвращает в качестве результата процессорное время, затраченное процессом с некоего момента, зависящего от реализации и связанного только с его (процесса) запуском. В случае ошибки результат равен `(clock_t) (-1)`.

```
#include <time.h>  
clock_t clock (void);
```

Листинг 12.15. Описание функции `clock()`.

Чтобы перевести время, возвращаемое функцией `clock()`, в секунды, его следует поделить на константу `CLOCKS_PER_SEC`, которая определена в заголовочном файле `<time.h>` равной одному миллиону, хотя разрешающая способность часов процессорного времени в конкретной реализации может и отличаться от микросекунды.

Отметим, что если значения типа `clock_t` представлены как 32-разрядные целые со знаком, то менее чем через 36 минут (процессорного времени) наступит переполнение.

В листинге 12.16 показан пример использования функции `clock()`. Поскольку начало отсчета для часов процессорного времени не специфицировано, их показания опрашиваются в начале и конце измеряемого промежутка, а результатом измерения затраченного процессорного времени служит разность этих показаний.

```
#include <stdio.h>
#include <time.h>
int main (void) {
    time_t st;
    clock_t ct;
    double s = 0;
    double d = 1;
    int i;
    fprintf (stderr, "Начало выполнения цикла\n");
    (void) time (&st);
    ct = clock ();
    for (i = 1; i <= 100000000; i++) {
        s += d / i;
        d = -d;
    }
    fprintf (stderr, "Процессорное время выполнения"
            "цикла: %g сек.\n",
            (double) (clock () - ct) / CLOCKS_PER_SEC);
    fprintf (stderr, "Астрономическое время"
            "выполнения цикла: %g сек.\n",
            difftime (time (NULL), st));
    return (0);
}
```

Листинг 12.16. Пример программы, использующей функции опроса показаний часов реального и процессорного времени.

Возможные результаты работы этой программы показаны в листинге 12.17. Процессорное время получилось больше астрономического из-за ошибок округления.

```
Начало выполнения цикла
Процессорное время выполнения цикла: 15.19 сек.
Астрономическое время выполнения цикла: 15 сек.
```

Листинг 12.17. Возможные результаты работы программы, использующей функции опроса показаний часов реального и процессорного времени.

Для измерения времени выполнения простой команды можно воспользоваться служебной программой `time` (не входящей, правда, в обязательную часть стандарта POSIX-2001):

```
time [-p] команда [аргумент ...]
```

Она выдает в стандартный протокол астрономическое и процессорное время, прошедшее от запуска до завершения команды с указанными аргументами. Если задана опция `-p`, время выдается в секундах в виде вещественных чисел. Процессорное время подразделяется на пользовательское (затраченное самой командой) и системное (ушедшее на оказание системных услуг команде). Если предположить, что выполнимый файл приведенной выше программы называется `my_tcex`, то командная строка

```
time -p my_tcex
```

выдаст в стандартный протокол примерно следующее (см. листинг 12.18):

```
Начало выполнения цикла
Процессорное время выполнения цикла: 15.2 сек.
Астрономическое время выполнения цикла: 15 сек.
real 15.20
user 15.20
sys 0.00
```

Листинг 12.18. Возможные результаты измерения времени работы программы, использующей функции опроса показаний часов реального и процессорного времени.

Если нужно измерить время выполнения конвейера или списка команд, их можно поместить в файл, превратив тем самым в простую команду, или воспользоваться конструкцией вида

```
time sh -c 'составная команда'
```

Оружием более крупного калибра является специальная встроенная в shell команда

```
times
```

Она выдает на стандартный вывод процессорное время, затраченное командным интерпретатором и порожденными им процессами. Например, после выполнения команд

```
my_tcex & my_tcex
times
```

на стандартный вывод может быть выдано:

```
0m0.010s 0m0.000s
0m30.410s 0m0.000s
```

Можно видеть, что сам командный интерпретатор практически не занимал процессор, а два порожденных им процесса в сумме потребили чуть меньше тридцати с половиной секунд; время системных услуг в обоих случаях оказалось пренебрежимо малым.

Реализация описанных выше утилит `time` и `times` опирается на функцию `times()` (см. листинг 12.19). Она опрашивает данные о времени выполнения вызывающего процесса и порожденных процессов, завершение которых ожидалось с помощью функций `wait()` или `waitpid()`. В отличие от `clock()`, функция `times()` измеряет время в тактах часов, и переполнение 32-разрядного представления значений типа `clock_t` происходит не через полчаса, а примерно в течение года (если секунда делится на 60 – 100 тактов). Соответственно, для перевода результатов работы `times()` в секунды их нужно делить на `sysconf(_SC_CLK_TCK)`, а не на `CLOCKS_PER_SEC`.

```
#include <sys/times.h>
clock_t times (struct tms *buffer);
```

Листинг 12.19. Описание функции `times()`.

В качестве результата функция `times()` возвращает астрономическое время, прошедшее от некоторого момента в прошлом (например, от загрузки системы), но основные, содержательные данные, относящиеся к вызывающему процессу и его потомкам, помещаются в структуру типа `tms`, которая описана в заголовочном файле `<sys/times.h>` и, согласно стандарту POSIX-2001, должна содержать по крайней мере следующие поля:

```
clock_t tms_utime; /* Процессорное время, */
/* затраченное вызывающим процессом */
clock_t tms_stime; /* Процессорное время, */
/* затраченное системой на обслуживание */
/* вызывающего процесса */
clock_t tms_cutime; /* Процессорное время, */
/* затраченное завершившимися порожденными */
/* процессами */
clock_t tms_cstime; /* Процессорное время, */
```



```
/* затраченное системой на обслуживание */
/* завершившихся порожденных процессов */
```

Значения времени завершившихся порожденных процессов (`tms_utime` и `tms_cutime`, а также `tms_stime` и `tms_cstime`) добавляются, соответственно, к элементам `tms_cutime` и `tms_cstime` структуры родительского процесса при возврате из функций `wait()` или `waitpid()`. Это происходит рекурсивно: если порожденный процесс ожидал завершения своих потомков, то в упомянутых элементах структуры накопятся данные о времени выполнения поддерева процессов.

В качестве примера использования функции `times()` измерим время работы рассматривавшейся ранее программы, копирующей строки со стандартного ввода на стандартный вывод через сокеты адресного семейства `AF_UNIX` с помощью порожденного процесса (см. листинг 12.20).

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Программа копирует строки со стандартного          */
/* ввода на стандартный вывод, "прокачивая" их         */
/* через пару сокетов. Используются функции           */
/* ввода/вывода нижнего уровня.                      */
/* Измеряется астрономическое и процессорное время   */
/* * * * * * * * * * * * * * * * * * * * * * * * * * */

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <sys/times.h>

#define MY_PROMPT    "Вводите строки\n"
#define MY_MSG       "Вы ввели: "

int main (void) {
    int sds [2];
    char buf [1];
    int new_line = 1; /* Признак того, что надо */
    /* выдать сообщение MY_MSG перед отображением */
    /* очередной строки */
    clock_t st_ct; /* Данные об астрономическом */
    /* и процессорном времени */
    struct tms st_cpt; /* в начале работы программы */
```

```
clock_t en_ct; /* Данные об астрономическом */
               /* и процессорном времени */
struct tms en_cpt; /* в конце работы программы */
long tck_p_sec; /* Количество тактов часов */
               /* в секунде */

/* Опросим данные о времени начала выполнения */
if ((st_ct = times (&st_cpt)) == (clock_t)
    (-1)) {
    perror ("TIMES-1");
    return (1);
}

/* Создадим пару соединенных безымянных сокетов */
if (socketpair (AF_UNIX, SOCK_STREAM, 0, sds)
    < 0) {
    perror ("SOCKETPAIR");
    return (2);
}
switch (fork ()) {
    case -1:
        perror ("FORK");
        return (3);
    case 0:
        /* Чтение из сокета sds [0] и выдачу на */
        /* стандартный вывод реализуем в */
        /* порожденном процессе */
        while (read (sds [0], buf, 1) == 1) {
            if (write (STDOUT_FILENO, buf, 1) != 1) {
                perror ("WRITE TO STDOUT");
                break;
            }
        }
        exit (0);
}

/* Чтение со стандартного ввода и запись в */
/* сокет sds [1] возложим на родительский процесс */
if (write (sds [1], MY_PROMPT,
    sizeof (MY_PROMPT) - 1) != sizeof (MY_PROMPT) - 1) {
    perror ("WRITE TO SOCKET-1");
}
```

```
while (read (STDIN_FILENO, buf, 1) == 1) {
    /* Перед отображением очередной строки */
    /* нужно выдать сообщение MY_MSG */
    if (new_line) {
        if (write (sds [1], MY_MSG,
            sizeof (MY_MSG) - 1) != sizeof (MY_MSG) - 1) {
            perror ("WRITE TO SOCKET-2");
            break;
        }
    }
    if (write (sds [1], buf, 1) != 1) {
        perror ("WRITE TO SOCKET-3");
        break;
    }
    new_line = (buf [0] == '\n');
}
shutdown (sds [1], SHUT_WR);

(void) wait (NULL);

/* Опросим данные о времени конца выполнения, */
/* вычислим и выдадим результаты измерений */
if ((en_ct = times (&en_cpt)) == (clock_t) (-1)) {
    perror ("TIMES-2");
    return (4);
}

tck_p_sec = sysconf (_SC_CLK_TCK);
fprintf (stderr, "Тактов в секунде: %ld\n",
    tck_p_sec);
fprintf (stderr, "Астрономическое время работы"
    " программы: %g сек.\n",
    (double) (en_ct - st_ct) / tck_p_sec);
fprintf (stderr, "Процессорное время,"
    " затраченное процессом: %g сек.\n",
    (double) (en_cpt.tms_utime - st_cpt.tms_utime) /
    tck_p_sec);
fprintf (stderr, "Процессорное время,"
    " затраченное системой: %g сек.\n", (double)
    (en_cpt.tms_stime - st_cpt.tms_stime) /
    tck_p_sec);
fprintf (stderr, "Аналогичные данные для"
```

```
    "порожденных процессов:\n");  
    fprintf (stderr, "%g сек.\n%g сек.\n",  
            (double) (en_cpt.tms_cutime -  
            st_cpt.tms_cutime) / tck_p_sec,  
            (double) (en_cpt.tms_cstime -  
            st_cpt.tms_cstime) / tck_p_sec);  
    return (0);  
}
```

Листинг 12.20. Пример программы, использующей функцию `times()`.

Если перенаправить стандартный ввод в какой-либо текстовый файл заметных размеров, а стандартный вывод — в другой файл, можно получить результаты, подобные тем, что показаны в листинге 12.21.

```
Тактов в секунде: 100  
Астрономическое время работы программы: 1.19 сек.  
Процессорное время, затраченное процессом: 0.02 сек.  
Процессорное время, затраченное системой: 0.08 сек.  
Аналогичные данные для порожденных процессов:  
0.09 сек.  
1 сек.
```

Листинг 12.21. Возможные результаты работы программы, использующей функцию `times()`.

Прямое манипулирование часами процессорного времени возможно после обращения к функции `clock_getcpuclockid()` (см. листинг 12.22), позволяющей выяснить их идентификатор. Оговоримся, однако, что эта функция не входит в обязательную часть стандарта POSIX-2001 (она относится к продвинутым средствам реального времени).

```
#include <time.h>  
int clock_getcpuclockid (pid_t pid,  
                        clockid_t *clock_id);
```

Листинг 12.22. Описание функции `clock_getcpuclockid()`.

Идентификатор часов процессорного времени, обслуживающих заданный аргументом `pid` процесс, возвращается по адресу, специфицированному аргументом `clock_id`. Если значение `pid` равно нулю, имеется

в виду вызывающий процесс. Признаком успешного завершения является нулевой результат; в противном случае возвращается код ошибки.

В листинге 12.23 показан пример употребления функции `clock_getcpuclockid()` и манипулирования часами процессорного времени. Возможные результаты выполнения этой программы — в листинге 12.24.

```
#include <stdio.h>
#include <time.h>
#include <sys/types.h>
#include <errno.h>

int main (void) {
    clockid_t clk_id; /* Идентификатор часов */
                        /* процессорного времени */
    struct timespec tmsp;
    double s = 0;
    double d = 1;
    int i;

    if ((errno = clock_getcpuclockid ((pid_t) 0,
        &clk_id)) != 0) {
        perror ("CLOCK_GETCPULOCKID");
        return (1);
    }

    if (clock_getres (clk_id, &tmsp) == -1) {
        perror ("CLOCK_GETRES");
        return (2);
    }
    printf ("Разрешающая способность часов"
        "процессорного времени: %ld нсек.\n",
        tmsp.tv_nsec);

    /* Измерим процессорное время выполнения цикла. */
    fprintf (stderr, "Начало выполнения цикла\n");
    tmsp.tv_sec = tmsp.tv_nsec = 0;
    if (clock_settime (clk_id, &tmsp) == -1) {
        perror ("CLOCK_SETTIME");
        return (3);
    }
    for (i = 1; i <= 1000000000; i++) {
```

```

        s += d / i;
        d = -d;
    }
    if (clock_gettime (clk_id, &tmsp) != 0) {
        perror ("CLOCK_GETTIME");
        return (4);
    }
    fprintf (stderr, "Процессорное время выполнения"
            "цикла: %ld сек. %ld нсек. и",
            tmsp.tv_sec, tmsp.tv_nsec);
    return (0);
}

```

Листинг 12.23. Пример программы, манипулирующей часами процессорного времени.

Разрешающая способность часов процессорного времени: 2 нсек.
 Начало выполнения цикла
 Процессорное время выполнения цикла: 15 сек.
 350313054 нсек.

Листинг 12.24. Возможные результаты работы программы, манипулирующей часами процессорного времени.

Обратим внимание на два момента. Перед началом измеряемого участка программы на часах процессорного времени были установлены нулевые показания, что, разумеется, не влияет на приоритет процесса и его планирование. Конечно, можно было обойтись без переустановки, запомнив показания часов в начале и конце измеряемого промежутка и произведя затем их вычитание, но технически это чуть сложнее; кроме того, нам хотелось применить функцию `clock_gettime()`. Второй заслуживающий отдельного упоминания момент состоит в отсутствии необходимости выяснять идентификатор часов процессорного времени вызывающего процесса с помощью функции `clock_getcpuclockid()`: можно воспользоваться именованной константой `CLOCK_PROCESS_CPUTIME_ID`.

Опрос и изменение данных о времени, ассоциированных с файлами

Информацию о файле, в том числе данные о времени последнего доступа и изменения, можно получить с помощью описанных ранее функций семейства `stat()`.

Для изменения ассоциированных с файлами данных о времени служит функция `utime()` (см. листинг 12.25), оперирующая в терминах секунд от начала отсчета.

```
#include <time.h>
int utime(const char *path,
          struct timeval *times);
```

Листинг 12.25. Описание функции `utime()`.

Если значение аргумента `times` равно пустому указателю, используется текущее время. В этом случае для успешного изменения данных о времени действующий идентификатор процесса должен соответствовать владельцу файла либо процесс должен иметь права на запись в файл или соответствующие привилегии.

Если значение `times` отлично от `NULL`, оно трактуется как указатель на структуру типа `struct timeval`, которая, согласно стандарту POSIX-2001, содержит по крайней мере следующие поля:

```
time_t lastime; /* Время последнего доступа */
time_t modtime; /* Время последнего изменения */
```

При этом данные о времени устанавливаются по указуемой структуре, а вызывающий процесс должен обладать соответствующими привилегиями или действующим идентификатором, совпадающим с идентификатором владельца файла (наличия права на запись недостаточно).

В случае успешного завершения функция `utime()` возвращает нулевой результат и, кроме того, переустанавливает время последнего изменения статуса файла.

Приведем пример программы, изменяющей время последнего доступа к файлу (см. листинг 12.26).

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Программа устанавливает время последнего                */
/* доступа к файлу - аргументу командной строки,          */
/* равное текущему времени:                                 */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * */

#include <stdio.h>
#include <sys/stat.h>
#include <time.h>
#include <utime.h>
```

```
int main (int argc, char *argv []) {
    struct stat st_buf; /* Буфер для опроса данных */
                        /* о файле */
    struct utimbuf ut_buf; /* Буфер для формирования */
                          /* изменяемых данных о файле */
    if (argc != 2) {
        fprintf (stderr, "Использование: %s файл\n",
            argv [0]);
        return (1);
    }
    if (stat (argv [1], &st_buf) != 0) {
        perror ("STAT");
        return (2);
    }
    ut_buf.actime = time (NULL);
    ut_buf.modtime = st_buf.st_mtime;
    if (utime (argv [1], &ut_buf) != 0) {
        perror ("UTIME");
        return (3);
    }
    return (0);
}
```

Листинг 12.26. Пример программы, изменяющей время последнего доступа к файлу.

Чтобы изменить только одно из двух ассоциированных с файлом времен, необходимо с помощью функции `stat()` получить информацию о файле, перенести неизменяемое значение из структуры типа `stat` в `utimbuf`, установить требуемым образом второй элемент структуры типа `utimbuf` и только после этого обратиться к функции `utime()`.

Читателю предлагается убедиться, что приведенная программа работает должным образом и что опция `-u` служебной программы `ls` предписывает оперировать со временем последнего доступа вместо времени последнего изменения.

Приостановка выполнения на заданное время

Обычным средством ожидания наступления каких-либо событий, не прерывающих выполнения процесса (потока управления), является циклическое чередование приостановки выполнения на заданное время и опроса статуса события.

Функция `sleep()` (см. листинг 12.27) позволяет приостановить выполнение процесса (потока управления) на заданное число секунд. Если в это время будет доставлен сигнал, который не игнорируется, вызов `sleep()` завершится досрочно, а результат будет равен «недоспанному» числу секунд. (В случае нормального завершения результат равен нулю, что формально можно считать частным случаем предыдущего.)

```
#include <unistd.h>
unsigned sleep (unsigned seconds);
```

Листинг 12.27. Описание функции `sleep()`.

Обратим внимание на то, что аргумент и результат функции `sleep()` описаны как `unsigned`. Это значит, что приложения, строго соответствующие стандарту POSIX-2001, не должны передавать `sleep()` величины, превышающие минимально допустимое значение конфигурационной константы `UINT_MAX`, которое в стандарте ISO C [5] полагается равным 65535. Использование больших величин, вообще говоря, ограничивает мобильность приложений.

Если вместо функции `sleep()` воспользоваться ее более современным аналогом `nanosleep()` (см. листинг 12.28), можно убить сразу двух зайцев: избавиться от обременительного ограничения на максимальную длительность приостановки выполнения и получить возможность задавать эту длительность с существенно более высокой точностью, ограниченной только разрешающей способностью часов реального времени.

```
#include <time.h>
int nanosleep (const struct timespec *rqtp,
               struct timespec *rmtp);
```

Листинг 12.28. Описание функции `nanosleep()`.

Аргумент `rqtp` функции `nanosleep()` задает запрашиваемую длительность приостановки. По указателю `rmtp`, если он не пуст, возвращается «недоспанное» время. В отличие от `sleep()`, результат `nanosleep()` равен `-1` — как в результате ошибки, так и при «недоспании», вызванном доставкой сигнала.

В общем случае ожидание наступления какого-либо события может быть активным или пассивным. Активное ожидание предполагает постоянный, повторяющийся в цикле опрос ассоциированных с событием значений и, следовательно, вызывает расход процессорного времени. При пассивном ожидании процесс приостанавливается (например, на семафоре) до

наступления события, не занимая процессор. Функции `sleep()` и `nanosleep()` позволяют выбрать компромиссный способ ожидания, когда процессорного времени расходуется немного и оперативность реагирования на события оказывается удовлетворительной.

В листинге 12.29 показан пример использования функции `sleep()` для «полупассивного» ожидания изменения файла. Разумеется, обработка файла может быть более содержательной, однако представленная схема действий носит весьма общий характер и используется, например, служебными программами `cron`, `tail` и др.

```

/* * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Программа отслеживает изменение размера           */
/* файла, заданного как аргумент командной строки     */
/* * * * * * * * * * * * * * * * * * * * * * * * * * */

#include <unistd.h>
#include <stdio.h>
#include <sys/stat.h>
#include <time.h>
#include <limits.h>

static time_t lt_mod = 0; /* Время последнего */
                          /* изменения файла */
static struct stat st_buf; /* Данные о файле */

/* Функция начальной обработки файла */
static void proc_init (const char *path) {
    fprintf (stderr, "Данные о размере файла %s\n",
            path);
    fprintf (stderr, "Время изменения Размер\n");
}

/* Функция проверки, нужно ли обрабатывать файл */
/* Результат > 0 - нужно */
/*      0 - не нужно */
/*      < 0 - ошибка */
static int check_mod (const char *path) {
    if (stat (path, &st_buf) != 0) {
        perror ("STAT");
        return (-1);
    }
    if (st_buf.st_mtime != lt_mod) {

```

```
/* Файл изменился и, следовательно, нуждается */
/* в обработке */
    lt_mod = st_buf.st_mtime;
    return (1);
}
return 0;
}

/* Функция обработки файла. */
/* Выведем время последнего изменения и */
/* текущий размер */
static void proc_mod (void) {
    char dtbuf [LINE_MAX]; /* Буфер для данных */
                               /* о времени */

    (void) strftime (dtbuf, sizeof (dtbuf), "%c",
        localtime (&lt_mod));
    fprintf (stderr, "%s %ld\n", dtbuf,
        st_buf.st_size);
}

int main (int argc, char *argv []) {
    int res;
    if (argc != 2) {
        fprintf (stderr, "Использование: %s файл\n",
            argv [0]);
        return (1);
    }
    proc_init (argv [1]);
    while (1) {
        if ((res = check_mod (argv [1])) > 0) {
            proc_mod ();
        } else if (res < 0) {
            return (res);
        }
        sleep (1);
    }
    return 0;
}
```

Листинг 12.29. Пример программы, использующей функцию `sleep()`.

Возможные результаты работы этой программы показаны в листинге 12.30.

Данные о размере файла /var/log/cron	
Время изменения	Размер
Tue Jan 6 12:50:00 2004	11191
Tue Jan 6 13:01:00 2004	11263
Tue Jan 6 13:10:00 2004	11409
Tue Jan 6 13:20:00 2004	11481
. . .	
Tue Jan 6 13:40:00 2004	11624
. . .	

Листинг 12.30. Возможные результаты работы программы, использующей функцию `sleep()`.

Функция `nanosleep()` позволяет до некоторой степени промоделировать работу функций `poll()` и `select()`, реализующих пассивное ожидание готовности данных, в тех ситуациях, когда данные поступают из разнородных источников, которые `poll()` и `select()` не обслуживают (не только из файлов, но и из очередей сообщений, как результат опроса значений семафоров и т.п.). В листинге 12.31 показан цикл «полуактивного» (с короткими приостановками) ожидания поступления данных из канала и обнуления значения семафора.

```
#define N          ...
#define MY_BUFSIZ  ...

struct timespec tmsp = {0, 20000000};
int fds [2]; /* Файловые дескрипторы канала */
int semid; /* Идентификатор набора семафоров */
struct sembuf sembuf [1]; /* Массив для задания */
/* операций над семафором */
char buf [MY_BUFSIZ]; /* Буфер для чтения данных */
/* из канала */
int brd; /* Число прочитанных байт */
int i;
. . .
fcntl (fds [0], F_SETFL, O_NONBLOCK);
sembuf [0].sem_num = 0;
sembuf [0].sem_flg = IPC_NOWAIT;
sembuf [0].sem_op = 0;
```

```
for (i = 0; i < N; i++) {
    nanosleep (&tmsp, NULL);
    if ((brd = read (fds [0], buf, MY_BUFSIZ) > 0) {
        /* Обработка прочитанных данных. */
        /* Возможно, поступление данных на этом */
        /* не закончилось */
        . . .
        continue;
    }

    if (semop (semid, sembuf, 1) == 0) {
        /* Значение семафора обнулено, ждать */
        /* больше нечего */
        . . .
        if (brd <= 0) {
            close (fds [0]);
            break;
        }
    }
}
. . .
```

Листинг 12.31. Пример реализации полуактивного ожидания наступления нескольких событий с помощью функции `nanosleep()`.

Подобные действия характерны для функций аккуратного завершения многопроцессных приложений, когда значение семафора играет роль счетчика активных порожденных процессов, от которых через канал поступают данные.

Работа с интервальными таймерами

Интервальным таймером называется механизм, способный известить процесс (поток управления) об истечении заданного промежутка времени.

Описываемые далее средства для работы с интервальными таймерами входят в необязательную часть стандарта POSIX-2001, именуемую «X/Open-расширение системного интерфейса» (XSI).

Реализация должна предоставлять каждому процессу по крайней мере три интервальных таймера, обозначаемых следующими идентификаторами.

ITIMER_REAL

Таймер реального времени. Он ассоциирован с часами `CLOCK_REALTIME` и, следовательно, его показания уменьшаются в реальном масштабе времени. Когда он срабатывает, процессу доставляется сигнал `SIGALRM`.

ITIMER_VIRTUAL

Таймер виртуального времени процесса. Его показания уменьшаются в соответствии с течением виртуального времени процесса, т. е. только тогда, когда процесс выполняется. При срабатывании таймера процессу доставляется сигнал `SIGVTALRM`.

ITIMER_PROF

Таймер профилирования. Показания этого таймера уменьшаются с течением виртуального времени процесса, а также тогда, когда работают запрошенные им системные сервисы. Он предназначен для построения статистическими методами профилей расходования процессом процессорного времени. При его срабатывании процессу доставляется сигнал `SIGPROF`.

Согласно стандарту POSIX-2001, интервальные таймеры обслуживаются функциями `getitimer()` и `setitimer()` (см. листинг 12.32).

```
#include <sys/time.h>
int getitimer (int timer_id,
               struct itimerval *ovalue);
int setitimer (int timer_id,
               const struct itimerval *restrict nvalue,
               struct itimerval *restrict ovalue);
```

Листинг 12.32. Описание функций `getitimer()` и `setitimer()`.

Поддерживаемые стандартом значения аргумента `timer_id` перечислены выше. Другие характеристики интервальных таймеров задаются в структуре типа `itimerval`, которая должна содержать по крайней мере следующие поля:

```
struct timeval it_interval; /* Интервал таймера */
struct timeval it_value;   /* Текущие показания */
/* таймера (ведется обратный отсчет) */
```

(Напомним, что структура типа `timeval` была описана выше. Она содержит по крайней мере два поля, задающие время, соответственно, в секундах и микросекундах.)

Значение поля `it_value` (если оно отлично от нуля) показывает время, оставшееся до срабатывания таймера. После срабатывания таймер запускается вновь с начальным значением поля `it_value`, равным `it_interval` (если последнее отлично от нуля).

Установка нулевого значения в поле `it_value`, независимо от величины `it_interval`, снимает таймер со взвода. Если сделать нулевым значение `it_interval`, таймер будет разряжен после очередного срабатывания. Таким способом можно реализовать таймер с ограниченной периодичностью (в частности, одноразовый).

Функция `getitimer()` запоминает текущие характеристики таймера с заданным идентификатором в структуре, на которую указывает аргумент `ostruct`. Функция `setitimer()` — ее можно назвать многоцелевой — взводит или снимает таймер со взвода, устанавливает новые характеристики, пользуясь значением аргумента `nvalue`, и сохраняет старые по указателю `ovalue` (если он отличен от `NULL`).

Признаком успешного завершения обеих функций является нулевой результат.

Здесь уместно акцентировать внимание на одной тонкости, которую до сих пор мы старательно замалчивали. Поскольку разрешающая способность любых часов конечна, для показаний часов и таймеров можно установить только те значения, которые кратны этой разрешающей способности. Если соответствующие аргументы функций `clock_gettime()`, `clock_settime()` или `setitimer()` не удовлетворяют данному условию, производится округление. Для часов используется наилучшее приближение снизу, для длительности приостановки выполнения и таймеров — наилучшее приближение сверху (стандарт POSIX-2001 требует, чтобы длительности приостановки выполнения и промежутка времени до срабатывания таймера были не меньше заказанных).

Последнее обстоятельство позволяет выяснить размер такта часов реального времени довольно неожиданным образом (впрочем, вполне стандартным и мобильным с точки зрения спецификаций POSIX-2001) (см. листинг 12.33).

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* Программа выясняет размер такта часов                */
/* реального времени                                     */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * */

#include <stdio.h>
```

```
#include <sys/time.h>

int main (void) {
    struct itimerval itvl;
    itvl.it_interval.tv_sec = 0;
    itvl.it_interval.tv_usec = 1;
    itvl.it_value.tv_sec = 0;
    itvl.it_value.tv_usec = 0;

    if (setitimer (ITIMER_REAL, &itvl, NULL) < 0) {
        perror ("SETITIMER");
        return (1);
    }
    if (getitimer (ITIMER_REAL, &itvl) < 0) {
        perror ("GETITIMER");
        return (2);
    }
    if (itvl.it_interval.tv_usec < 2) {
        printf ("Не удалось выяснить размер такта"
               "часов реального времени\n");
    } else {
        printf ("Размер такта часов реального времени:"
               "%ld мксек\n", itvl.it_interval.tv_usec);
    }
    return 0;
}
```

Листинг 12.33. Пример программы, использующей функции `getitimer()` и `setitimer()`.

Здесь интервальный таймер не взводится (поскольку значение `it_value` – нулевое) и, соответственно, не срабатывает. Он нужен лишь для того, чтобы операционная система подкорректировала должным образом (с учетом разрешающей способности часов реального времени) значение поля `it_interval.tv_usec`. В результате корректировки первоначально присвоенная этому полю единица может превратиться, например, в 10000 (микросекунд), что дает тактовую частоту часов реального времени 100 Гц.

Следующая программа (см. листинг 12.34) сравнивает ход реального и виртуального времени процесса, определяя, сколько реального времени прошло за один квант виртуального (в данном случае квант – это 100000 мксек, т. е. 0.1 сек). Колебания показаний, естественно, вызываются

нагрузкой на процессор, индуцируемой другими, параллельно работающими процессами. В принципе, целенаправленная организация подобных колебаний и их анализ могут быть использованы для создания так называемых скрытых каналов по времени (см. [7]).

```

* * * * *
/* Программа сравнивает ход реального и */
/* виртуального времени процесса */
* * * * *

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <signal.h>

#define TM_BUF_SIZE      128

/* Массив для хранения показаний часов */
/* реального времени в момент срабатывания */
/* таймера виртуального времени процесса */
static struct timeval rt_vls [TM_BUF_SIZE];
/* Указатель на текущий элемент */
/* массива rt_vls */
static struct timeval *prt_vls = rt_vls;

/* Функция обработки срабатывания таймеров */
/* (сигналы SIGALRM и SIGVTALRM) */
static void proc_itexprtn (int signo) {
    struct itimerval itmvl = {{0, 0}, {0, 0}};

    /* Запомним текущее астрономическое время */
    (void) gettimeofday (prt_vls++, NULL);
    if (signo == SIGALRM) {
        /* Сработал таймер реального времени. */
        /* Разрядим таймер виртуального времени */
        /* и завершимся (с выдачей результатов) */
        (void) setitimer (ITIMER_VIRTUAL, &itmvl, NULL);
        exit (0);
    }
}

```

```
/* Функция выдачи данных о ходе реального времени */
/* на фоне равномерного течения виртуального */
/* времени процесса */
static void print_rt_data (void) {
    struct timeval *tpvt_vls = rt_vls;
    int i = 0;

    printf ("Прошедшее реальное время за один квант"
           " виртуального\n");
    while (++tpvt_vls != pvt_vls) {
        printf (" %3d %10ld мксек\n", ++i,
               (tpvt_vls->tv_sec - (pvt_vls - 1)->tv_sec)
               * 1000000 + (tpvt_vls->tv_usec -
               (pvt_vls - 1)->tv_usec));
    }
}

int main (void) {
    struct itimerval itvl;
    struct sigaction sact;
    sigset_t sset;

    /* Установим реакцию на сигналы SIGALRM и */
    /* SIGVTALRM. */
    /* Позаботимся, чтобы функция обработки не */
    /* могла быть прервана срабатыванием */
    /* другого таймера */
    if ((sigemptyset (&sset) < 0) ||
        (sigaddset (&sset, SIGALRM) < 0) ||
        (sigaddset (&sset, SIGVTALRM) < 0)) {
        perror ("SIGEMPTYSET or SIGADDDSET");
        return (1);
    }

    sact.sa_handler = proc_itexprtn;
    sact.sa_flags = 0;
    sact.sa_mask = sset;
    if (sigaction (SIGALRM, &sact, NULL) < 0) {
        perror ("SIGACTION-SIGALRM");
        return (2);
    }
    if (sigaction (SIGVTALRM, &sact, NULL) < 0) {
```

```
    perror ("SIGACTION-SIGVTALRM");
    return (3);
}

/* Зарегистрируем функцию print_rt_data() в */
/* atexit() */
if (atexit (print_rt_data) != 0) {
    perror ("ATEXIT");
    return (4);
}

/* Взведем таймер реального времени как */
/* однократный */
itvl.it_interval.tv_sec = 0;
itvl.it_interval.tv_usec = 0;
itvl.it_value.tv_sec = 10;
itvl.it_value.tv_usec = 0;
if (setitimer (ITIMER_REAL, &itvl, NULL) < 0) {
    perror ("SETITIMER-REAL");
    return (5);
}

/* Установим начало отсчета для данных о */
/* реальном времени */
(void) gettimeofday (&pvt_vls, NULL);

/* Таймер виртуального времени сделаем */
/* периодическим */
itvl.it_interval.tv_sec = 0;
itvl.it_interval.tv_usec = 100000; /* 0.1 сек */
itvl.it_value.tv_sec = 0;
itvl.it_value.tv_usec = 100000;
if (setitimer (ITIMER_VIRTUAL, &itvl, NULL)
    < 0) {
    perror ("SETITIMER-VIRTUAL");
    return (6);
}

/* Убедимся, что пока процесс приостановлен, */
/* его виртуальное время также стоит */
sleep (5);
```

```
/* Нагрузим процессор вычислениями, которые */
/* не должны завершиться до срабатывания */
/* таймера реального времени */
{
    double s = 0;
    double d = 1;
    int i;

    for (i = 1; i <= 1000000000; i++) {
        s += d / i;
        d = -d;
    }
}
return 0;
}
```

Листинг 12.34. Пример программы, использующей интервальные таймеры реального и виртуального времени.

Результаты работы программы могут выглядеть так, как показано в листинге 12.35. Можно видеть, что, во-первых, доля процессорного времени, которая достается процессу, может существенно колебаться (см., например, отсчеты 4, 6 и 8) и, во-вторых, пока выполнение процесса приостановлено, его виртуальное время также стоит (см. инструкцию `sleep(5)` в листинге 12.34, отсчет 1 в результатах).

Прошедшее реальное время за один квант виртуального 1 5255378 мксек

2	249993 мксек
3	250003 мксек
4	100000 мксек
5	250177 мксек
6	859831 мксек
7	100054 мксек
8	729943 мксек
9	580003 мксек
10	99989 мксек
11	580129 мксек
12	429881 мксек
13	99990 мксек

Листинг 12.35. Возможные результаты работы программы, использующей интервальные таймеры реального и виртуального времени.

Интервальные таймеры полезны как средство управления ходом выполнения программы. В частности, они позволяют ограничить время работы отдельных фрагментов, что особенно важно для приложений реального времени. В программе, показанной в листинге 12.36, фигурируют две функции обработки данных; каждой из них отводится не более заданного константой IT_PERIOD числа секунд.

```

* * * * *
* Программа вызывает функции обработки *
* и контролирует время их выполнения *
* с помощью интервального таймера *
/* * * * * */

#include <stdio.h>
#include <sys/time.h>
#include <signal.h>
#include <setjmp.h>

/* Период интервального таймера (в секундах) */
#define IT_PERIOD 1

static jmp_buf buf_env; /* Буфер для функций */
/* setjmp и longjmp */

static double s; /* Результат функций обработки */
/* данных */

/* Функция обработки срабатывания таймера */
/* реального времени (сигнал SIGALRM) */
static void proc_sigalrm (int dummy) {
    printf ("Текущий результат текущей функции"
           "обработки данных: %g\n", s);
    longjmp (buf_env, 1);
}

/* Первая функция обработки данных */
/* (вычисляет ln (2)) */
static void proc_data_1 (void) {
    double d = 1;
    int i;

    s = 0;

```

```
    for (i = 1; i <= 1000000000; i++) {
        s += d / i;
        d = -d;
    }
}
/* Вторая функция обработки данных */
/* (вычисляет sqrt (2)) */
static void proc_data_2 (void) {
    s = 1;
    do {
        s = (s + 2 / s) * 0.5;
    } while ((s * s - 2) > 0.000000001);
}

int main (void) {
/* Массив указателей на функции */
    /* обработки данных */
    void (*fptrs []) (void) = {proc_data_1,
        proc_data_2, NULL};
    /* Указатель на указатель на текущую */
    /* функцию обработки данных */
    void (**tfptr) (void);
    /* Вспомогательная временная переменная */
    void (*tmpfptr) (void);
    struct itimerval itvl;
    struct sigaction sact;
    sigset_t sset;
    int i;

    /* Установим реакцию на сигнал SIGALRM */
    if (sigemptyset (&sset) < 0) {
        perror ("SIGEMPTYSET");
        return (1);
    }
    sact.sa_handler = proc_sigalrm;
    sact.sa_flags = SA_NODEFER; /* Не блокировать */
    /* SIGALRM в функции обработки этого сигнала */
    sact.sa_mask = sset;
    if (sigaction (SIGALRM, &sact, NULL) < 0) {
        perror ("SIGACTION");
        return (2);
    }
}
```

```
/* На всякий случай сделаем таймер реального */
/* времени периодическим */
itvl.it_interval.tv_sec = IT_PERIOD;
itvl.it_interval.tv_usec = 0;

/* Цикл вызова функций обработки данных. */
/* Выполним его дважды */
for (i = 0; i < 2; i++) {
    tfptr = fptrs;
    (void) setjmp (buf_env);
    /* Сюда вернется управление после */
    /* срабатывания таймера */
    while ((tmpfptr = *tfptr++) != NULL) {
        /* Даже если предыдущая функция обработки */
        /* данных закончилась до того, как */
        /* сработал таймер, обеспечим текущей */
        /* функции полный */
        /* интервал */
        itvl.it_value.tv_sec = IT_PERIOD;
        itvl.it_value.tv_usec = 0;
        if (setitimer (ITIMER_REAL, &itvl, NULL)
            < 0) {
            perror ("SETITIMER");
            return (3);
        }
        (*tmpfptr) ();
        printf ("Результат текущей функции обработки"
            "данных: %g\n", s);
    }
}
return 0;
}
```

Листинг 12.36. Пример программы, использующей интервальные таймеры реального времени для управления ходом выполнения программы.

Результаты работы приведенной программы могут выглядеть так, как показано в листинге 12.37. Видно, что первая функция обработки данных не уложилась в отведенное время, поэтому ее выполнение было прервано срабатыванием таймера; второй функции этого времени оказалось достаточно.

Текущий результат текущей функции обработки данных:
0.693147

Результат текущей функции обработки данных: 1.41421

Текущий результат текущей функции обработки данных:
0.693147

Результат текущей функции обработки данных: 1.41421

Листинг 12.37. Возможные результаты работы программы, использующей интервальные таймеры реального времени для управления ходом выполнения программы.

Комбинация нелокальных переходов и асинхронной обработки сигналов представляет собой взрывоопасную смесь, обращаться с которой следует крайне осторожно.

Несмотря на кажущуюся простоту, в пояснениях нуждается целый ряд фрагментов приведенной выше программы.

Во-первых, обратим внимание на использование флага `SA_NODEFER`, предписывающего не блокировать сигнал при входе в функцию его (сигнала) обработки. Этот флаг необходим, поскольку не происходит нормального выхода из функции `proc_sigalrm()`, а значит, не происходит и восстановления маски сигналов процесса. В результате, после первого входа в функцию обработки сигнала `SIGALRM` он рискует остаться заблокированным, вследствие чего процесс не будет извещен о последующих срабатываниях интервального таймера реального времени.

Во-вторых, следует исходить из возможности приостановки выполнения процесса в произвольном месте, в частности, внутри вычисления выражений. Если оформить вызов функции обработки данных в виде инструкции `(**t_fptr++)()`, то нет гарантии, что компилятор не запланирует увеличение указателя `t_fptr` после возврата из функции, выполнение которой может быть прервано срабатыванием таймера. В таком случае управление вернется в начало внутреннего цикла, а указатель не увеличится, что приведет к повторному вызову той же функции обработки данных. Чтобы справиться с этой проблемой, пришлось задействовать вспомогательную временную переменную `tmp_fptr`.

В-третьих, процессу может выделяться сколь угодно малая доля процессорного времени, а потому соотношение между виртуальным и реальным временем оказывается недетерминированным. Если, например, расположить вызов `setjmp()` после изведения таймера (см. листинг 12.38), то таймер реального времени может сработать до инициализации буфера `buf_env`, после чего нелокальный переход из функции обработки сигнала приведет к полному хаосу.


```
if (setitimer (ITIMER_REAL, &itvl, NULL) < 0) {  
    . . .  
}  
  
. . .  
  
(void) setjmp (buf_env);
```

Листинг 12.38. Пример некорректной последовательности взведения таймера и инициализации структуры для нелокального перехода.

Проще (и, вероятно, лучше) всего бороться с подобными проблемами, манипулируя маской сигналов процесса, попеременно блокируя и разрешая обработку сигнала SIGALRM. Мы, однако, попытались обойтись без этого и будем признательны, если читатель обнаружит ошибки в нашем варианте программы и сообщит о них.

Еще один любопытный момент технического характера состоит в том, что обычно проверяют значение, возвращаемое `setjmp()`, чтобы понять, как управление оказалось в данной точке — после естественного возврата или нелокального перехода. Но поскольку в приведенной программе это ни на что не влияет, проверку можно опустить.

К теме интервальных таймеров логически примыкает функция `alarm()` (см. листинг 12.39), позволяющая вызывающему процессу заказать доставку сигнала SIGALRM через заданное число секунд реального времени. Если в момент обращения к `alarm()` предыдущий заказ существовал и еще не был выполнен, он отменяется, а если значение аргумента `seconds` отлично от нуля, его место занимает новый.

```
#include <unistd.h>  
unsigned alarm (unsigned seconds);
```

Листинг 12.39. Описание функции `alarm()`.

При наличии предыдущего заказа функция `alarm()` возвращает число секунд, оставшееся до его выполнения; в противном случае результат равен нулю.

Примеры использования функции `alarm()` были приведены при рассмотрении средств межпроцессного взаимодействия.

Вариант 1

1. Согласно стандарту POSIX-2001, результат функции `time()` имеет тип

- ☐ `clock_t`
- ☐ `double`
- ☐ `time_t`

2. Согласно стандарту POSIX-2001, функция `gmtime()` позволяет узнать

- ☐ всемирное время
- ☐ местное время с учетом сезонных поправок
- ☐ данные о времени, ассоциированные с файлами

3. Согласно стандарту POSIX-2001, для перевода в секунды времени, возвращаемого функцией `clock()`, его следует разделить на

- ☐ `CLOCKS_PER_SEC`
- ☐ `sysconf (_SC_CLK_TCK)`
- ☐ `sizeof (clock_t)`

Вариант 2

1. Согласно стандарту POSIX-2001, результат функции `times()` имеет тип

- ☐ `clock_t`
- ☐ `double`
- ☐ `time_t`

2. Согласно стандарту POSIX-2001, функция `localtime()` позволяет узнать

- ☐ всемирное время
- ☐ местное время с учетом сезонных поправок
- ☐ данные о времени, ассоциированные с файлами

3. Согласно стандарту POSIX-2001, для перевода в секунды времени, возвращаемого функцией `times()`, его следует поделить на

- ☐ `CLOCKS_PER_SEC`
- ☐ `sysconf (_SC_CLK_TCK)`
- ☐ `sizeof (clock_t)`

Вариант 3

1. Согласно стандарту POSIX-2001, результат функции `difftime()` имеет тип
 - ☐ `clock_t`
 - ☐ `double`
 - ☐ `time_t`
2. Согласно стандарту POSIX-2001, функция `utime()` позволяет узнать
 - ☐ всемирное время
 - ☐ местное время с учетом сезонных поправок
 - ☐ данные о времени, ассоциированные с файлами
3. Согласно стандарту POSIX-2001, для перевода в секунды времени, возвращаемого функцией `time()`, его следует поделить на
 - ☐ `CLOCKS_PER_SEC`
 - ☐ `sysconf (_SC_CLK_TCK)`
 - ☐ `sizeof (time_t)`

Лекция 13. Языково-культурная среда

Описываются основные понятия и объекты, ассоциированные с языково-культурной средой, служебные программы и функции, позволяющие опрашивать и устанавливать характеристики среды, а также способы написания программ, настраиваемых на различные языково-культурные среды.

Ключевые слова: языково-культурная среда, языковые соглашения, культурные соглашения, пользовательское окружение, настройка на языково-культурную среду, локализация, интернационализация, категория, элемент, ключевое слово, классификация символов, преобразование регистра символов, порядок алфавитного сравнения символов, форматирование денежных величин, форматирование числовых величин, форматы даты и времени, форматы информационных и диагностических сообщений, а также интерактивных ответов, POSIX-среда, подразумеваемая языково-культурная среда, файл отображения символов, файл определения среды, определение категории, заголовок, тело, хвостовик, комментарий, управляющий символ, создание языково-культурной среды, исходный формат определения языково-культурной среды, целевой формат определения языково-культурной среды, общедоступная языково-культурная среда, языково-культурная среда ограниченного доступа, опрос характеристик языково-культурной среды, зарезервированное слово, текущая языково-культурная среда, преобразование денежных величин, спецификатор преобразования, номер ошибки, диагностическое сообщение, каталог сообщений, исходное описание каталога сообщений, идентификатор набора сообщений, идентификатор сообщения, дескриптор каталога сообщений.

Основные понятия и объекты

Согласно стандарту POSIX-2001, языково-культурная среда — это часть пользовательского окружения, зависящая от языковых и культурных соглашений.

Настройка на определенную языково-культурную среду представляет собой процесс формирования в системе совокупности данных, специфичных для поддержки конкретных естественных языков, местных настроек и кодировок символов. Иногда подобный процесс называют также локализацией, в противоположность интернационализации — процессу подготовки приложений, способных настраиваться на различные языково-культурные среды.

Языково-культурная среда формируется из данных, принадлежащих нескольким именованным категориям. Каждая категория управляет определенными аспектами поведения компонентов системы. Имена и назначение категорий соответствуют следующим переменным окружения.

LC_STYPE

Классификация символов, преобразование регистра (верхний/нижний) символов.

LC_COLLATE

Порядок алфавитного сравнения символов.

LC_MONETARY

Форматирование денежных величин.

LC_NUMERIC

Форматирование числовых (но не денежных) величин.

LC_TIME

Форматы даты и времени.

LC_MESSAGES

Форматы информационных и диагностических сообщений и интерактивных ответов.

Категории подразделяются на более мелкие элементы, средством именованья которых служат ключевые слова. Например, в категорию LC_STYPE входят элементы с именами alpha (буквы), digits (цифры) и т.п.

Как правило, значениями перечисленных переменных окружения являются абсолютные маршрутные имена файлов, содержащих соответствующие определения. Другой вариант (если значение не начинается с символа /) – имя среды.

Расширение XSI стандарта POSIX-2001 уточняет один из возможных форматов имени языково-культурной среды:

язык[_территория] [. кодировка] [@модификатор]

Примеры: ru_RU.koi8r, ru-UA.

Модификатор позволяет выбрать конкретный вариант данных о среде в пределах отдельной категории (скажем, словарный порядок сравнения в категории LC_COLLATE). Пример: LC_COLLATE=De_DE@dict.

В каждой реализации определены одна или несколько языково-культурных сред. Поддержка POSIX-среды с именами-синонимами «POSIX» и «C» является обязательной. Кроме того, возможно создание новых сред, если в системе определен конфигурационный параметр _POSIX2_LOCALEDEF.

Стандартом POSIX-2001 предусмотрены две «собираательные» переменные окружения, определяющие сразу все (или почти все) категории языково-культурной среды.

LC_ALL

Значение этой переменной окружения учитывается в первую очередь.

LANG

Значение этой переменной окружения учитывается в последнюю очередь, если не определены значения переменных вида LC_*.

В любой реализации некоторая среда (POSIX или иная) должна быть определена как подразумеваемая, т. е. используемая тогда, когда перечисленные выше переменные окружения не определены или имеют пустые значения. Это можно сделать, например, путем присваивания LANG=POSIX или LANG=C, после чего подразумеваемой станет POSIX-среда.

В структуре lconv, описанной во включаемом файле <locale.h>, содержатся поля, определяющие форматы денежных и числовых величин. Даже беглый взгляд на перечень этих полей позволяет составить представление о богатстве возможностей определения языково-культурной среды.

```
char *currency_symbol; /* Местное обозначение */
/* денежной величины */
char *decimal_point; /* Символ, отделяющий */
/* целую часть числа от дробной */
char frac_digits; /* Количество цифр в дробной */
/* части местных денежных величин */
char *grouping; /* Размеры групп цифр при */
/* форматировании чисел */
char *int_curr_symbol; /* Международное */
/* обозначение денежной величины */
```

```
char int_frac_digits; /* Количество цифр в */
/* дробной части международных денежных величин */
char int_n_cs_precedes; /* Признак того, что */
/* международное обозначение предшествует */
/* отрицательной денежной величине, а не следует */
/* за ней */
char int_n_sep_by_space; /* Признак того, что */
/* международное обозначение отделяется от */
/* отрицательной денежной величины пробелом */
char int_n_sign_posn; /* Позиция знака минус */
/* в международных обозначениях отрицательных */
/* денежных величин */
char int_p_cs_precedes; /* Признак того, что */
/* международное обозначение предшествует */
/* неотрицательной денежной величине, а не */
/* следует за ней */
char int_p_sep_by_space; /* Признак того, что */
/* международное обозначение отделяется от */
/* неотрицательной денежной величины пробелом */
char int_p_sign_posn; /* Позиция знака плюс в */
/* международных обозначениях положительных */
/* денежных величин */
char *mon_decimal_point; /* Символ, отделяющий */
/* целую часть денежной величины от дробной */
char *mon_grouping; /* Размеры групп цифр при */
/* форматировании денежных величин */
char *mon_thousands_sep; /* Символ, разделяющий */
/* группы цифр в денежных величинах */
char *negative_sign; /* Знак минус для */
/* отрицательных денежных величин */
char n_cs_precedes; /* Признак того, что */
/* местное обозначение предшествует отрицательной */
/* денежной величине, а не следует за ней */
char n_sep_by_space; /* Признак того, что */
/* местное обозначение отделяется от отрицательной */
/* денежной величины пробелом */
char n_sign_posn; /* Позиция знака минус в */
/* местных обозначениях отрицательных денежных */
/* величин */
char *positive_sign; /* Знак плюс для */
/* неотрицательных денежных величин */
char p_cs_precedes; /* Признак того, что местное */
/* обозначение предшествует положительной */
/* денежной величине, а не следует за ней */
char p_sep_by_space; /* Признак того, что местное */
/* обозначение отделяется от положительной */
/* денежной величины пробелом */
char p_sign_posn; /* Позиция знака плюс в */
/* местных обозначениях положительных денежных */
/* величин */
```



```
/* обозначение предшествует неотрицательной */  
/* денежной величине, а не следует за ней */  
char p_sep_by_space; /* Признак того, что местное */  
/* обозначение отделяется от неотрицательной */  
/* денежной величины пробелом */  
char p_sign_posn; /* Позиция знака плюс в местных */  
/* обозначениях положительных денежных величин */  
char *thousands_sep; /* Символ, разделяющий */  
/* группы цифр в числовых величинах */
```

Стандарт POSIX-2001 регламентирует средства опроса и установки характеристик языково-культурной среды, а также опроса доступного набора таких сред.

Общая логика использования средств интернационализации/локализации такова: пользователь, присваивая соответствующие значения описанным выше переменным окружения, идентифицирует свою языково-культурную среду; приложения могут опросить данные о целевой среде с помощью функции `setlocale()` и произвести настройку интернационализированных компонентов.

Определение языково-культурной среды

Определение языково-культурной среды помещается в файлы, к описанию формата которых мы и приступаем. Можно считать, что в подобных файлах, называемых далее файлами определения среды, содержится «исходный текст» определения, поскольку, чтобы возыметь действие, они должны быть обработаны утилитой `localedef` или эквивалентным средством.

Файл определения среды должен содержать определение одной или нескольких категорий, которое может сводиться к директиве `сору` (см. далее).

Определение категории состоит из заголовка, тела и хвостовика. Заголовок именует категорию; он должен начинаться с символов `LC_`. Хвостовик строится из слова `END` и имени, употребленного в качестве заголовка.

Первому заголовку могут предшествовать строки, переопределяющие символ комментария (по умолчанию — `#`) и управляющий символ (`\`).

Тело категории состоит из одной или нескольких строк, содержащих идентификатор и, возможно, следующие за ним операнды. В качестве идентификатора может выступать как ключевое слово, именующее отдельный элемент языково-культурной среды, так и элемент алфавитного сравнения символов. Ключевые слова должны быть уникальными в пределах среды и не начинаться с префикса `LC_`.

Операндами могут быть символы, элементы сравнения и цепочки символов. Цепочки символов заключаются в двойные кавычки, операнды разделяются точками с запятой и, возможно, пробелами.

Символы и элементы сравнения представляются определенными ниже именами. Кроме того, символы могут обозначаться естественным образом или кодами; правда, при этом определение языково-культурной среды теряет в мобильности.

Имена символов заключаются в угловые скобки (например, `<A>`, `<z>` и т.п.). Такая конструкция должна в точности совпадать с именем, определенным в файле отображения символов, который специфицирует опция `-f` утилиты `localedef`.

Реализация должна поддерживать по крайней мере один набор символов и соответствующий файл отображения. Любой набор должен включать в себя все элементы из так называемого мобильного набора символов, определяемого стандартом POSIX-2001 (их имена перечислены в листинге 13.2).

Файл отображения символов имеет довольно простую структуру. Если отвлечься от несущественных деталей, он должен состоять из директив вида

```
<имя_символа> <код> <комментарий>
```

Эти директивы заключаются между строками

```
CHARMAP
```

и

```
END CHARMAP
```

Перед строкой `CHARMAP` может располагаться определение имени набора символов, которое задается директивой

```
<code_set_name> имя
```

Фрагмент возможного описания набора символов KOI8-R показан в листинге 13.1.

```
<code_set_name> KOI8-R
<comment_char> %
<escape_char> /
% version: 1.0
% source: RFC1489 via Gabor Kiss <kissg@sztaki.hu>
```

```
% and Andrey A. Chernov <ache@astral.msk.su>
CHARMAP
<U0000> /x00  NULL (NUL)
<U0001> /x01  START OF HEADING (SOH)
<U0002> /x02  START OF TEXT (STX)
. . .
<U0020> /x20  SPACE
<U0021> /x21  EXCLAMATION MARK
. . .
<U0030> /x30  DIGIT ZERO
<U0031> /x31  DIGIT ONE
. . .
<U0041> /x41  LATIN CAPITAL LETTER A
<U0042> /x42  LATIN CAPITAL LETTER B
. . .
<U0061> /x61  LATIN SMALL LETTER A
<U0062> /x62  LATIN SMALL LETTER B
. . .
<U044E> /xc0  CYRILLIC SMALL LETTER YU
<U0430> /xc1  CYRILLIC SMALL LETTER A
<U0431> /xc2  CYRILLIC SMALL LETTER BE
. . .
<U0429> /xfd  CYRILLIC CAPITAL LETTER SHCHA
<U0427> /xfe  CYRILLIC CAPITAL LETTER CHE
<U042A> /xff  CYRILLIC CAPITAL LETTER HARD SIGN
END CHARMAP
```

Листинг 13.1. Фрагмент файла отображения символов для кодировки KOI8-R.

Опишем теперь формат определения категорий языково-культурной среды.

Категория LC_CTYPE определяет классификацию символов, преобразование регистра и другие атрибуты. К ней относятся следующие ключевые слова: `upper` (прописные буквы; для POSIX-среды – 26 латинских букв верхнего регистра), `lower` (строчные буквы), `alpha` (буквы), `digit` (цифры), `alnum` (буквы и цифры), `space` (пробельные символы; для POSIX-среды в их число, помимо пробела, входят перевод строки, табуляция и т.п.), `cntrl` (управляющие символы), `punct` (знаки пунктуации), `graph` (печатные символы за исключением пробела), `print` (печатные символы), `xdigit` (шестнадцатеричные цифры), `blank` (для POSIX-среды – пробел и табуляция).

С ключевым словом `сору` задается имя существующей среды, у которой наследуется определение категории. При наличии данной директивы другие ключевые слова недопустимы.

Ключевое слово `charclass` позволяет задать имена специфичных для определяемой среды классов символов. Каждый такой класс далее должен быть описан в определении категории `LC_CTYPE` по образцу и подобию перечисленных выше стандартных классов.

Ключевое слово `toupper` открывает определение отображения малых букв в большие. Операнд этого слова представляет собой пару символов, заключенных в скобки и разделенных запятой. Аналогичным образом устроен элемент `tolower`.

В листинге 13.2 показано определение категории `LC_CTYPE` для POSIX-среды.

```
LC_CTYPE
# POSIX-среда, категория LC_CTYPE
# "alpha" по определению есть объединение элементов
# "upper" и "lower"
# "alnum" по определению есть объединение элементов
# "alpha" и "digit"
# "print" по определению есть объединение элементов
# "alnum", "punct" и <space>
# "graph" по определению есть объединение элементов
# "alnum" и "punct"
#
upper <A>;<B>;<C>;<D>;<E>;<F>;<G>;<H>;<I>;<J>;<K>;\
    <L>;<M>;<N>;<O>;<P>;<Q>;<R>;<S>;<T>;<U>;<V>;<W>;\
    <X>;<Y>;<Z>
#
lower <a>;<b>;<c>;<d>;<e>;<f>;<g>;<h>;<i>;<j>;<k>;\
    <l>;<m>;<n>;<o>;<p>;<q>;<r>;<s>;<t>;<u>;<v>;<w>;\
    <x>;<y>;<z>
#
digit <zero>;<one>;<two>;<three>;<four>;<five>;\
    <six>;<seven>;<eight>;<nine>
#
space <tab>;<newline>;<vertical-tab>;<form-feed>;\
    <carriage-return>;<space>
#
cntrl <alert>;<backspace>;<tab>;<newline>;\
    <vertical-tab>;<form-feed>;<carriage-return>;\
    <NUL>;<SOH>;<STX>;<ETX>;<EOT>;<ENQ>;<ACK>;<SO>;\
```

```

<SI>;<DLE>;<DC1>;<DC2>;<DC3>;<DC4>;<NAK>;<SYN>;\
<ETB>;<CAN>;<EM>;<SUB>;<ESC>;<IS4>;<IS3>;<IS2>;\
<IS1>;<DEL>

#
punct <exclamation-mark>;<quotation-mark>;\
<number-sign>;<dollar-sign>;<percent-sign>;\
<ampersand>;<apostrophe>;<left-parenthesis>;\
<right-parenthesis>;<asterisk>;<plus-sign>;\
<comma>;<hyphen>;<period>;<slash>;<colon>;\
<semicolon>;<less-than-sign>;<equals-sign>;\
<greater-than-sign>;<question-mark>;\
<commercial-at>;<left-square-bracket>;\
<backslash>;<right-square-bracket>;\
<circumflex>;<underscore>;<grave-accent>;\
<left-curly-bracket>;<vertical-line>;\
<right-curly-bracket>;<tilde>

#
xdigit <zero>;<one>;<two>;<three>;<four>;<five>;\
<six>;<seven>;<eight>;<nine>;<A>;<B>;<C>;<D>;\
<E>;<F>;<a>;<b>;<c>;<d>;<e>;<f>

#
blank <space>;<tab>

#
toupper (<a>,<A>);(<b>,<B>);(<c>,<C>);(<d>,<D>);\
(<e>,<E>);(<f>,<F>);(<g>,<G>);(<h>,<H>);\
(<i>,<I>);(<j>,<J>);(<k>,<K>);(<l>,<L>);(<m>,<M>);\
(<n>,<N>);(<o>,<O>);(<p>,<P>);(<q>,<Q>);(<r>,<R>);\
(<s>,<S>);(<t>,<T>);(<u>,<U>);(<v>,<V>);(<w>,<W>);\
(<x>,<X>);(<y>,<Y>);(<z>,<Z>)

#
tolower (<A>,<a>);(<B>,<b>);(<C>,<c>);(<D>,<d>);\
(<E>,<e>);(<F>,<f>);(<G>,<g>);(<H>,<h>);\
(<I>,<i>);(<J>,<j>);(<K>,<k>);(<L>,<l>);\
(<M>,<m>);(<N>,<n>);(<O>,<o>);(<P>,<p>);\
(<Q>,<q>);(<R>,<r>);(<S>,<s>);(<T>,<t>);\
(<U>,<u>);(<V>,<v>);(<W>,<w>);(<X>,<x>);\
(<Y>,<y>);(<Z>,<z>)

END LC_CTYPE

```

Листинг 13.2. Определение категории LC_CTYPE для POSIX-среды.

Категория `LC_COLLATE` определяет порядок алфавитного сравнения символов для многочисленных служебных программ (`sort`, `uniq` и т.д.), регулярных выражений, а также функций `strcoll()`, `strxfrm()` и других.

В алфавитном сравнении могут участвовать односимвольные и многосимвольные элементы. Их порядок задается весами, которых у каждого элемента может быть несколько (не более `COLL_WEIGHTS_MAX`). Первый вес называется основным. Если при сравнении элементов основные веса совпали, используются соответственные пары дополнительных — до тех пор, пока либо не будет обнаружено неравенство, либо не кончатся веса.

Допускается формирование классов эквивалентности — присвоение одного основного веса нескольким элементам сравнения. Поддерживаются также отображения один-ко-многим (один символ отображается в последовательность элементов сравнения).

Разумеется, сравнение цепочек символов начинается с разбиения на элементы сравнения.

Правила алфавитного сравнения и директивы присвоения весов открываются ключевым словом `order_start` и завершаются директивой `order_end`. Подразумеваемым правилом является `forward` — сравнение от начала к концу цепочки. Возможен и противоположный порядок (`backward`).

После строки `order_start` элементы сравнения перечисляются в порядке возрастания. После элемента может быть задана последовательность его весов. Веса задаются как относительные, в виде имен других элементов и определяются позицией этих элементов при перечислении. Специальный вес `IGNORE` означает, что данный элемент следует игнорировать при сравнении цепочек. Многоточие в качестве элемента сравнения обозначает диапазон от предыдущего до последующего элементов. Элемент `UNDEFINED` представляет все, что не было указано явно.

В POSIX-среде алфавитный порядок совпадает с упорядоченностью символов в кодировке ASCII.

Категория `LC_MONETARY` определяет формат денежных величин. Входящие в нее элементы именуются аналогично полям приведенной выше структуры `lconv`. Например, после ключевого слова `currency_symbol` задается цепочка символов, обозначающая местные денежные величины. Если ключевое слово данной категории отсутствует в файле определения среды либо его значением является пустая цепочка или `-1`, значит, соответствующий элемент в языково-культурной среде неспецифицирован.

В POSIX-среде все элементы этой категории остаются неспецифицированными.

В категории `LC_NUMERIC`, во многом аналогичной `LC_MONETARY`, определяются правила и обозначения, используемые применительно к числовым данным. Здесь также действует параллель со структурой `lconv`.

В листинге 13.3 показано определение категории LC_NUMERIC для POSIX-среды. Собственно, специфицируется только десятичная точка.

```
LC_NUMERIC
# POSIX-среда, категория LC_NUMERIC
#
decimal_point "<period>"
thousands_sep ""
grouping      -1
#
END LC_NUMERIC
```

Листинг 13.3. Определение категории LC_NUMERIC для POSIX-среды.

Элементы категории LC_TIME определяют интерпретацию спецификаторов преобразований и, тем самым, поведение служебной программы `date`, а также функций `strftime()`, `strptime()` и некоторых других. В число поддерживаемых ключевых слов входят `abday` (сокращенные названия дней недели, начиная с воскресенья; элемент соответствует спецификатору `%a`), `day` (полные названия дней недели, `%A`), `abmon` (сокращенные названия месяцев, `%b`), `mon` (полные названия месяцев, `%B`), `d_t_fmt` (принятое в данной языково-культурной среде представление даты и времени, `%c`), `d_fmt` (принятое в данной среде представление даты, `%x`), `t_fmt` (принятое в данной среде представление времени, `%X`).

Особым образом устроен элемент `era`, определяющий способ отсчета и отображения лет для каждой эры, поддерживаемой языково-культурной средой. Операнд ключевого слова `era` состоит из цепочек символов, каждая из которых описывает конкретную эру и имеет следующий формат:

```
направление:смещение:начальная_дата:конечная_дата:
имя_эры:формат_эры
```

Здесь `направление` — знак плюс или минус, `смещение` — ближайший к начальной дате номер года, `начальная_дата` — цепочка вида `гггг/мм/дд` (год, месяц и число начала эры), `конечная_дата` — год, месяц и число конца эры либо цепочки `<-*` (конечной датой служит начало отсчета времени) или `<+*` (конечной датой служит конец отсчета времени), `формат_эры` — цепочка, используемая при форматировании номеров года в эре.

Отметим, что точка отсчета может быть как самой ранней, так и самой поздней в эре. Примерами служат две христианские эры — после и до Рождества Христова, соответственно.

В листинге 13.4 показано определение категории LC_TIME для POSIX-среды.

```
LC_TIME
# POSIX-среда, категория LC_TIME
#
# Сокращенные названия дней недели (%a)
abday  "<S><u><n>"; "<M><o><n>"; "<T><u><e>"; \
        "<W><e><d>"; "<T><h><u>"; "<F><r><i>"; "<S><a><t>"
#
# Полные названия дней недели (%A)
day    "<S><u><n><d><a><y>"; \
        "<M><o><n><d><a><y>"; \
        "<T><u><e><s><d><a><y>"; \
        "<W><e><d><n><e><s><d><a><y>"; \
        "<T><h><u><r><s><d><a><y>"; \
        "<F><r><i><d><a>\<y>"; \
        "<S><a><t><u><r><d><a><y>"
#
# Сокращенные названия месяцев (%b)
abmon  "<J><a><n>"; "<F><e><b>"; "<M><a><r>"; \
        "<A><p><r>"; "<M><a><y>"; "<J><u><n>"; \
        "<J><u><l>"; "<A><u><g>"; "<S><e><p>"; \
        "<O><c><t>"; "<N><o><v>"; "<D><e><c>"
#
# Полные названия месяцев (%B)
mon    "<J><a><n><u><a><r><y>" \
        "<F><e><b><r><u><a><r><y>"; \
        "<M><a><r><c><h>"; "<A><p><r><i><l>"; \
        "<M><a><y>"; "<J><u><n><e>"; \
        "<J><u><l><y>"; "<A><u><g><u><s><t>"; \
        "<S><e><p><t><e><m><b><e><r>"; \
        "<O><c><t><o><b><e><r>"; \
        "<N><o><v><e><m><b><e><r>"; \
        "<D><e><c><e><m><b><e><r>"
#
# Эквивалент AM/PM (%p)          "AM"; "PM"
am_pm  "<A><M>"; "<P><M>"
#
```



```

# Принятое в POSIX-среде представление даты и
# времени (%c)
# "%a %b %e %H:%M:%S %Y"
d_t_fmt    "<percent-sign><a><space><percent-sign>\ "
           "<b><space><percent-sign><e><space>\ "
           "<percent-sign>H<colon><percent-sign>\ "
           "<M><colon><percent-sign><S><space>\ "
           "<percent-sign><Y>"

#
# Принятое в POSIX-среде представление даты
# (%x) "%m/%d/%y"
d_fmt      "<percent-sign><m><slash><percent-sign><d>\ "
           "<slash><percent-sign><y>"

#
# Принятое в POSIX-среде представление времени
# (%X) "%H:%M:%S"
t_fmt      "<percent-sign>H<colon><percent-sign>M>\ "
           "<colon><percent-sign><S>"

#
# Принятое в POSIX-среде 12-часовое представление
# времени (%r) "%I:%M:%S %p"
t_fmt_ampm "<percent-sign><I><colon><percent-sign>\ "
           "<M><colon><percent-sign><S><space>\ "
           "<percent-sign><p>"

#
END LC_TIME

```

Листинг 13.4. Определение категории LC_TIME для POSIX-среды.

Категория LC_MESSAGES играет весьма ограниченную роль, определяя положительные и отрицательные ответы. Соответственно, она содержит два элемента — yesexpr и noexpr, значениями которых служат расширенные регулярные выражения. Для POSIX-среды данная категория описывается так, как показано в листинге 13.5.

```

LC_MESSAGES
# POSIX-среда, категория LC_MESSAGES
#
yesexpr    "<circumflex><left-square-bracket>\ "
           "<y><Y><right-square-bracket>"

#
noexpr     "<circumflex><left-square-bracket>\ "

```

```
"<n><N><right-square-bracket>"  
#  
END LC_MESSAGES
```

Листинг 13.5. Определение категории LC_MESSAGES для POSIX-среды.

Создание и опрос характеристик языково-культурной среды

Для создания целевой языково-культурной среды служит утилита `localedef`:

```
localedef [-c] [-f файл_отображения_символов]  
          [-i исходный_файл] [-u имя_набора_символов]  
          имя_целевой_среды
```

Утилита `localedef` преобразует рассмотренное выше определение категорий языково-культурной среды из исходного формата в целевой, пригодный для использования функциями и служебными программами, поведение которых зависит от значений переменных окружения.

Исходные данные для `localedef` поступают из файла с именем операнда опции `-i` или со стандартного ввода.

Операнд `имя_целевой_среды` определяет имя создаваемой среды. Она может быть как общедоступной, так и приватной, с ограниченным доступом.

Аргументом опции `-f` является маршрутное имя файла отображения символов. После опции `-u` можно задать имя нестандартного набора символов.

Опция `-c` предписывает генерировать выходной файл даже при наличии предупреждений.

Аргумент `имя_целевой_среды` идентифицирует созданную языково-культурную среду. Если это имя начинается с символа `/`, оно интерпретируется как маршрутное имя, под которым сохраняется сгенерированное определение целевой среды (формат результатов работы `localedef` стандартом не специфицируется); в противном случае интерпретация имени зависит от реализации, а созданная среда становится общедоступной.

Для получения информации о языково-культурных средах предназначена служебная программа `locale`:

```
locale [-a | -m]  
locale [-ck] имя ...
```

Будучи вызванной без аргументов, утилита `locale` выдает на стандартный вывод сводную информацию о среде, которая состоит из значений соответствующих переменных окружения (описанных выше, а также дополнительных, определенных, быть может, целевой системой). Результат может выглядеть так, как показано в листинге 13.6. Кавычки вокруг значений всех переменных окружения, кроме `LANG`, означают, что они не были установлены явным образом, а определены по значению `$LANG`.

```
LANG=ru_RU.koi8r
LC_CTYPE="ru_RU.koi8r"
LC_NUMERIC="ru_RU.koi8r"
LC_TIME="ru_RU.koi8r"
LC_COLLATE="ru_RU.koi8r"
LC_MONETARY="ru_RU.koi8r"
LC_MESSAGES="ru_RU.koi8r"
LC_PAPER="ru_RU.koi8r"
LC_NAME="ru_RU.koi8r"
LC_ADDRESS="ru_RU.koi8r"
LC_TELEPHONE="ru_RU.koi8r"
LC_MEASUREMENT="ru_RU.koi8r"
LC_IDENTIFICATION="ru_RU.koi8r"
LC_ALL=
```

Листинг 13.6. Возможный результат работы служебной программы `locale`.

При наличии аргументов выдается информация о некоторых (если задано ключевое слово) или обо всех (если указано имя категории) элементах категорий.

Опциям служебной программы `locale` приписан следующий смысл.

-a

Выдать информацию обо всех общедоступных языково-культурных средах (включая POSIX-среду).

-m

Выдать имена доступных файлов отображения символов.

-c

Выдавать имена выбранных (прямо или косвенно, посредством ключевых слов) категорий, что делает результаты более читабельными, когда они охватывают несколько категорий.

-k

Выдавать имена и значения заданных элементов категорий.

Операнд имя может быть именем категории, элемента категории (ключевым словом) или зарезервированным словом `charmap`. В двух первых случаях выдается информация о поименованной категории или ее элементе, в последнем — имя файла отображения символов, который был использован при создании языково-культурной среды.

Приведем еще два примера употребления утилиты `locale`. Данные о категории `LC_TIME`, выдаваемые по команде

```
locale -ck LC_TIME
```

могут выглядеть так, как показано в листинге 13.7.

```
LC_TIME
abday="Вск;Пнд;Втр;Срд;Чтв;Птн;Сбт"
day="Воскресенье;Понедельник;Вторник;Среда;"
    "Четверг;Пятница;Суббота"
abmon="Янв;Фев;Мар;Апр;Май;Июн;Июл;Авг;Сен;Окт;"
    "Ноя;Дек"
mon="Января;Февраля;Марта;Апреля;Мая;Июня;Июля;"
    "Августа;Сентября;Октября;Ноября;Декабря"
d_t_fmt="%a %d %b %Y %T"
d_fmt="%d.%m.%Y"
t_fmt="%T"
. . .

first_weekday=1
first_workday=1
cal_direction=1
date_fmt="%a %b %e %H:%M:%S %Z %Y"
time-codeset="KOI8-R"
```

Листинг 13.7. Фрагмент возможного результата выполнения команды `locale -ck LC_TIME`.

Второй пример демонстрирует фрагмент интернационализированного варианта shell-процедуры, анализирующей интерактивный ответ пользователя. Предполагается, что ответ является значением переменной `response`.

```
if echo "$response" | grep -Eq "${locale yesexpr}"
then
    echo "Ответ положительный"
else
    echo "Ответ отрицательный"
fi
```

Листинг 13.8. Фрагмент интернационализированного варианта shell-процедуры.

Программная установка и опрос характеристик языково-культурной среды

Мы приступаем к рассмотрению функций, предназначенных для работы с языково-культурной средой.

Функция `setlocale()` (см. листинг 13.9) служит для установки и/или опроса всей языково-культурной среды вызывающего процесса или отдельных категорий.

```
#include <locale.h>
char *setlocale (int category, const char *locale);
```

Листинг 13.9. Описание функции `setlocale()`.

Аргумент `category` задает категорию (`LC_COLLATE`, `LC_CTYPE`, `LC_MESSAGES`, `LC_MONETARY`, `LC_NUMERIC`, `LC_TIME`), а если его значение равно `LC_ALL`, то и всю среду.

Аргумент `locale` указывает на цепочку символов, содержащую требуемые установки для выбранной категории (категорий); в общем случае ее формат зависит от реализации. Имеется, однако, несколько предопределенных, стандартизованных цепочек.

"POSIX" или "C"

Специфицирует минимальную языково-культурную среду (POSIX-среду) для C-программ. При отсутствии явных обращений к функции `setlocale()` такая среда устанавливается по умолчанию при входе в `main()`.

" "

Специфицирует зависящую от реализации местную языково-культурную среду, определяемую значениями переменных окружения `LC_*` и `LANG`. Интернационализированная программа должна выполнять вызов вида

```
setlocale (LC_ALL, "");
```

для настройки на местную среду выполнения.

Если значением аргумента `locale` служит пустой указатель, функция `setlocale()` опрашивает текущее окружение и возвращает имя используемой языково-культурной среды или цепочку символов, ассоциированную с выбранной категорией. В противном случае в качестве результата возвращаются те же характеристики, но для новой среды. Результирующая цепочка (скопированная в отведенную прикладной программой область памяти) в сочетании с заданной категорией может быть использована в последующих вызовах функции `setlocale()` для восстановления соответствующей части языково-культурной среды.

Разумеется, в случае неудачного завершения вызова `setlocale()` результатом служит пустой указатель.

Получить детальную информацию о категориях `LC_MONETARY` и `LC_NUMERIC` текущей языково-культурной среды можно с помощью функции `localeconv()` (см. листинг 13.10).

```
#include <locale.h>
struct lconv *localeconv (void);
```

Листинг 13.10. Описание функции `localeconv()`.

Структура типа `lconv` была описана выше. Здесь мы отметим лишь два момента.

Поля, имеющие тип `char *`, указывают на цепочки символов, каждая из которых (за исключением `*decimal_point`) может иметь вид " " как признак того, что данное значение в текущей среде недоступно или имеет нулевую длину.

Поля типа `char` содержат неотрицательные числа; `CHAR_MAX` указывает на то, что данное значение в текущей среде недоступно.

Вызов `localeconv()` не может кончиться неудачей, однако следует учитывать, что возвращаемая структура может быть перезаписана последующими обращениями не только к `localeconv()`, но и к `setlocale()`.

Следующая программа (см. листинг 13.11) демонстрирует использование функций `setlocale()` и `localeconv()`.

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

/* Функция опрашивает и выводит цепочки символов, */
/* ассоциированные с текущей средой и ее */
/* категориями */
void print_curr_locale_data (void) {
    /* Массив катерорий для setlocale() */
    int ctgrs [] = {LC_ALL, LC_COLLATE, LC_CTYPE,
        LC_MESSAGES, LC_MONETARY, LC_NUMERIC, LC_TIME};
    /* Массив имен катерорий */
    char *ctgrnms [] = {"LC_ALL", "LC_COLLATE",
        "LC_CTYPE", "LC_MESSAGES", "LC_MONETARY",
        "LC_NUMERIC", "LC_TIME"};
    unsigned int i;

    printf ("Цепочки символов, ассоциированные"
        "в текущей среде c\n");
    for (i = 0; i < sizeof (ctgrs) / sizeof (int);
        i++) {
        printf (" %s: %s\n", ctgrnms [i],
            setlocale (ctgrs [i], NULL));
    }
}

int main (void) {
    struct lconv *plc;

    /* Убедимся, что подразумеваемой является */
    /* POSIX-среда */
    printf ("Текущая языково-культурная среда:"
        "подразумеваемая\n");
    print_curr_locale_data();
    /* Установим местную среду */
    (void) setlocale (LC_ALL, "");
    /* Опросим и выдадим ее характеристики */
    printf ("Текущая языково-культурная среда:"
        "местная\n");
    print_curr_locale_data();
}
```

```
plc = localeconv ();
printf ("Некоторые элементы категорий "
        LC_MONETARY и LC_NUMERIC\n");
printf ("int_curr_symbol: %s\n",
        plc->int_curr_symbol);
printf ("currency_symbol: %s\n",
        plc->currency_symbol);
printf ("mon_decimal_point: %s\n",
        plc->mon_decimal_point);
printf ("decimal_point: %s\n",
        plc->decimal_point);
printf ("thousands_sep: %s\n",
        plc->thousands_sep);

/* Сделаем "денежную" категорию украинской */
if (setlocale (LC_MONETARY, "ru-UA") == NULL) {
    perror ("SETLOCALE");
    return (1);
}
printf ("Категория LC_MONETARY переустановлена "
        "для Украины\n");
print_curr_locale_data();
plc = localeconv ();
printf ("Некоторые элементы категории "
        "LC_MONETARY\n");
printf ("int_curr_symbol: %s\n",
        plc->int_curr_symbol);
printf ("currency_symbol: %s\n",
        plc->currency_symbol);

return 0;
}
```

Листинг 13.11. Пример использования функций `setlocale()` и `localeconv()`.

Возможный результат работы этой программы показан в листинге 13.12.

Текущая языково-культурная среда: подразумеваемая
Цепочки символов, ассоциированные в текущей среде с
LC_ALL: C
LC_COLLATE: C


```
LC_CTYPE: C
LC_MESSAGES: C
LC_MONETARY: C
LC_NUMERIC: C
LC_TIME: C
```

Текущая языково-культурная среда: местная

Цепочки символов, ассоциированные в текущей среде с

```
LC_ALL: ru_RU.koi8r
LC_COLLATE: ru_RU.koi8r
LC_CTYPE: ru_RU.koi8r
LC_MESSAGES: ru_RU.koi8r
LC_MONETARY: ru_RU.koi8r
LC_NUMERIC: ru_RU.koi8r
LC_TIME: ru_RU.koi8r
```

Некоторые элементы категорий LC_MONETARY и LC_NUMERIC

```
int_curr_symbol: RUR
currency_symbol: PYB
mon_decimal_point: .
decimal_point: ,
thousands_sep: .
```

Категория LC_MONETARY переустановлена для Украины

Цепочки символов, ассоциированные в текущей среде с

```
LC_ALL: LC_CTYPE=ru_RU.koi8r;LC_NUMERIC=ru_RU.koi8r;
LC_TIME=ru_RU.koi8r;LC_COLLATE=ru_RU.koi8r;
LC_MONETARY=ru_UA;LC_MESSAGES=ru_RU.koi8r;
LC_PAPER=ru_RU.koi8r;LC_NAME=ru_RU.koi8r;
LC_ADDRESS=ru_RU.koi8r;LC_TELEPHONE=ru_RU.koi8r;
LC_MEASUREMENT=ru_RU.koi8r;
LC_IDENTIFICATION=ru_RU.koi8r
LC_COLLATE: ru_RU.koi8r
LC_CTYPE: ru_RU.koi8r
LC_MESSAGES: ru_RU.koi8r
LC_MONETARY: ru_UA
LC_NUMERIC: ru_RU.koi8r
LC_TIME: ru_RU.koi8r
```

Некоторые элементы категории LC_MONETARY

```
int_curr_symbol: UAH
currency_symbol: ГР
```

Листинг 13.12. Возможный результат работы программы, использующей функции `setlocale()` и `localeconv()`.

Для преобразования денежных величин в цепочку символов в соответствии с настройками текущей языково-культурной среды можно воспользоваться функцией `strfmon()` (см. листинг 13.13), входящей в XSI-расширение стандарта POSIX-2001.

```
#include <monetary.h>
ssize_t strfmon (char *restrict s, size_t maxsize,
    const char *restrict format, ...);
```

Листинг 13.13. Описание функции `strfmon()`.

Результат преобразования (длиной не более `maxsize` байт) помещается в буфер, на который указывает аргумент `s`. Само преобразование выполняется под управлением аргумента `format`. В формат могут входить обычные символы, просто копируемые в выходную цепочку, а также спецификаторы преобразований – под их воздействием производится извлечение и вывод дополнительных аргументов.

Спецификатор преобразования в общем случае состоит из следующих компонентов:

- символ `%`;
- флаги;
- ширина поля;
- число символов в целой части;
- число символов в дробной части;
- символ, задающий преобразование.

Обязательными являются только первый и последний компоненты. Возможных преобразований два:

i

Преобразовать значение типа `double` в денежную величину международного формата.

n

Преобразовать значение типа `double` в денежную величину местного формата.

В качестве результата функция `strfmon()` возвращает количество байт, помещенное в массив `s`.

Может показаться, что в деле интернационализации/локализации категориям `LC_MONETARY` и `LC_NUMERIC` повезло больше других, поскольку для них определена многокомпонентная структура `lconv`, содержащая

значения всех ключевых слов этих категорий, и функция `localeconv()`, позволяющая данную структуру получить. Однако на практике дело обстоит прямо противоположным образом. Для других категорий настройка приложений на местную языково-культурную среду сводится к вызову

```
setlocale (LC_ALL, "");
```

Всю остальную специфику среды учитывают функции `isalpha()`, `tolower()`, `strcoll()`, `strftime()` и т.д., которыми приложения могут пользоваться стандартными способами.

В листинге 13.14 показан пример программы, использующей упомянутые функции. Видно, как меняется их поведение при модификации языково-культурной среды (см. листинг 13.15).

```
#include <stdio.h>
#include <ctype.h>
#include <locale.h>
#include <limits.h>
#include <monetary.h>

#define MY_VALUE      1234.567

int main (void) {
    char sbuf [LINE_MAX];

    printf ("Текущая языково-культурная среда: "
           "подразумеваемая\n");
    printf ("isalpha ('B'): %d\n", isalpha ('B'));
    printf ("tolower ('B'): '%c'\n", tolower ('B'));

    printf ("Результат преобразования в денежную "
           "величину числа 1234.567\n");
    if (strfmon (sbuf, sizeof (sbuf),
                "Международное обозначение: %i\n", MY_VALUE)
        == -1) {
        perror ("STRFMON");
        return (1);
    }
    fputs (sbuf, stdout);
    (void) strfmon (sbuf, sizeof (sbuf),
                "Местное обозначение: %n\n", MY_VALUE);
    fputs (sbuf, stdout);
```

```
(void) setlocale (LC_ALL, "");
printf ("Текущая языково-культурная среда:"
       "местная\n");
printf ("isalpha ('Б'): %d\n", isalpha ('Б'));
printf ("tolower ('Б'): '%c'\n", tolower ('Б'));
printf ("Результат преобразования в денежную"
       "величину числа 1234.567\n");
(void) strfmon (sbuf, sizeof (sbuf),
               "Международное обозначение: %i\n", MY_VALUE);
fputs (sbuf, stdout);
(void) strfmon (sbuf, sizeof (sbuf),
               "Местное обозначение: %n\n", MY_VALUE);
fputs (sbuf, stdout);
(void) setlocale (LC_MONETARY, "ru-UA");
printf ("Категория LC_MONETARY переустановлена"
       "для Украины\n");
printf ("Результат преобразования в денежную"
       "величину числа 1234.567\n");
(void) strfmon (sbuf, sizeof (sbuf),
               "Международное обозначение: %i\n", MY_VALUE);
fputs (sbuf, stdout);
(void) strfmon (sbuf, sizeof (sbuf),
               "Местное обозначение: %n\n", MY_VALUE);
fputs (sbuf, stdout);
return 0;
}
```

Листинг 13.14. Пример программы, работающей в нескольких языково-культурных средах.

Текущая языково-культурная среда: подразумеваемая

isalpha ('Б'): 0

tolower ('Б'): 'Б'

Результат преобразования в денежную величину числа
1234.567

Международное обозначение: 1234.57

Местное обозначение: 1234.57

Текущая языково-культурная среда: местная

isalpha ('Б'): 1024

tolower ('Б'): 'б'

Результат преобразования в денежную величину числа

1234.567

Международное обозначение: 1 234.57 RUR

Местное обозначение: 1 234.57 руб

Категория LC_MONETARY переустановлена для Украины

Результат преобразования в денежную величину числа
1234.567

Международное обозначение: 1 234.57 UAH

Местное обозначение: 1 234.57 ГР

Листинг 13.15. Возможные результаты программы, работающей в нескольких языково-культурных средах.

Если требуется получить детальную информацию обо всех аспектах языково-культурной среды, можно воспользоваться функцией `nl_langinfo()` (см. листинг 13.16), отнесенной стандартом POSIX-2001 к расширению XSI.

```
#include <langinfo.h>
char *nl_langinfo (nl_item item);
```

Листинг 13.16. Описание функции `nl_langinfo()`.

Аргументом функции `nl_langinfo()` могут служить именованные константы, определенные в заголовочном файле `<langinfo.h>`. В общем и целом они соответствуют описанным выше ключевым словам. В качестве результата `nl_langinfo()` возвращает указатель на цепочку символов, содержащую данные о выбранном элементе среды. Если в текущей среде этот элемент отсутствует, выдаются данные для POSIX-среды.

Пример программы, использующей функцию `nl_langinfo()`, показан в листинге 13.17, возможные результаты ее работы – в листинге 13.18.

```
#include <stdio.h>
#include <locale.h>
#include <langinfo.h>
#include <regex.h>
#include <limits.h>

int main (void) {
    regex_t cere; /* Скомпилированные расширенные */
                  /* регулярные выражения */
    regex_t ceren;
    int reerrcode; /* Код ошибки от regcomp или */
```

```
/* regexes */
char reerrbuf [LINE_MAX]; /* Буфер для строк с */
                          /* сообщениями об ошибках */
char response [LINE_MAX]; /* Буфер для ответа */
                          /* пользователя */

printf ("Текущая языково-культурная среда:"
        "подразумеваемая\n");
printf ("Элемент YESEXPR категории LC_MESSAGES:"
        "%s\n", nl_langinfo (YESEXPR));
printf ("Элемент MON_1 категории LC_TIME: %s\n",
        nl_langinfo (MON_1));

(void) setlocale (LC_ALL, "");
printf ("Текущая языково-культурная среда:"
        "местная\n");
printf ("Элемент YESEXPR категории LC_MESSAGES:"
        "%s\n", nl_langinfo (YESEXPR));
printf ("Элемент MON_1 категории LC_TIME: %s\n",
        nl_langinfo (MON_1));

/* Скомпилируем расширенное регулярное */
/* выражение для утвердительного ответа */
if ((reerrcode = regcomp (&cere, nl_langinfo
    (YESEXPR), REG_EXTENDED | REG_ICASE |
    REG_NOSUB )) != 0) {
    (void) regerror (reerrcode, &cere, reerrbuf,
        sizeof (reerrbuf));
    fputs (reerrbuf, stderr);
    fputc ('\n', stderr);
    regfree (&cere);
    return (reerrcode);
}

/* То же для отрицательного ответа */
if ((reerrcode = regcomp (&ceren, nl_langinfo
    (NOEXPR), REG_EXTENDED | REG_ICASE |
    REG_NOSUB )) != 0) {
    (void) regerror (reerrcode, &ceren, reerrbuf,
        sizeof (reerrbuf));
    fputs (reerrbuf, stderr);
    fputc ('\n', stderr);
}
```

```

    regfree (&ceren);
    return (reerrcode);
}
fputs ("Вы поддерживаете идею стандартизации"
      "программных интерфейсов? ", stdout);
fgets (response, sizeof (response), stdin);
if (regexec (&cere, response, 0, NULL, 0) == 0) {
    fputs ("Ответ положительный\n", stdout);
} else if (regexec (&ceren, response, 0, NULL,
    0) == 0) {
    fputs ("Ответ отрицательный\n", stdout);
} else {
    fputs ("Ответ уклончивый\n", stdout);
}

regfree (&cere);
regfree (&ceren);
return 0;
}

```

Листинг 13.17. Пример программы, использующей функцию `nl_langinfo()`.

```

Текущая языково-культурная среда: подразумеваемая
Элемент YESEXPR категории LC_MESSAGES: ^[yY]
Элемент MON_1 категории LC_TIME: January
Текущая языково-культурная среда: местная
Элемент YESEXPR категории LC_MESSAGES: ^[ДдYy].*
Элемент MON_1 категории LC_TIME: Января
Вы поддерживаете идею стандартизации программных
интерфейсов? Да
Ответ положительный

```

Листинг 13.18. Возможные результаты работы программы, использующей функцию `nl_langinfo()`.

К сожалению, обязательная часть стандарта POSIX-2001 содержит лишь минимум средств для настройки на целевую среду выдаваемых диагностических сообщений. По сути он сводится к функции `strerror()` (см. листинг 13.19), которая отображает номера (коды) ошибок в сообщении, зависящие от языково-культурной среды, точнее, от установки категории `LC_MESSAGES`.

```
#include <string.h>
char *strerror (int errnum);
```

Листинг 13.19. Описание функции `strerror()`.

Обычно в качестве аргумента этой функции используют `errno`, но номера ошибок могут поступать и из произвольного источника.

В листинге 13.20 показан пример программы, выдающей диагностические сообщения в разных средах и разными средствами – с помощью функций `perror()` и `strerror()`.

```
#include <stdio.h>
#include <locale.h>
#include <string.h>
#include <errno.h>

int main (void) {
    fprintf (stderr, "Текущая языково-культурная "
              "среда: подразумеваемая\n");
    errno = 1;
    perror ("PERROR, сообщение номер 1");
    fprintf (stderr, "STRERROR, сообщение номер 1:"
              "%s\n", strerror (1));
    (void) setlocale (LC_ALL, "");
    fprintf (stderr, "Текущая языково-культурная "
              "среда: местная\n");
    errno = 1;
    perror ("PERROR, сообщение номер 1");
    fprintf (stderr, "STRERROR, сообщение номер 1:"
              "%s\n", strerror (1));
    (void) setlocale (LC_MESSAGES, "ru_UA");
    fprintf (stderr, "Категория LC_MESSAGES "
              "переустановлена для Украины\n");
    errno = 1;
    perror ("PERROR, сообщение номер 1");
    fprintf (stderr, "STRERROR, сообщение номер 1:"
              "%s\n", strerror (1));
    return 0;
}
```

Листинг 13.20. Пример программы, выдающей диагностические сообщения в разных языково-культурных средах и разными средствами.

Средства для работы с каталогами сообщений

Развитые средства для работы с каталогами сообщений (диагностических, информационных), выдаваемых приложениями, вынесены в расширение XSI стандарта POSIX-2001. Идея этих средств состоит в том, чтобы в интернационализированных программах фигурировали не сами сообщения, а их идентификаторы в каталоге, который для каждой языково-культурной среды может быть своим.

Стандарт не специфицирует формат каталогов сообщений, но предлагает служебную программу `gencat` для их генерации по исходному описанию:

```
gencat каталог файл_сообщений ...
```

Аргумент `каталог` задает маршрутное имя, под которым будет сохранен результат генерации.

Исходное описание может состоять из нескольких файлов сообщений, строки которых, помимо прочих, содержат следующие директивы.

```
$set идентификатор_набора_сообщений комментарий
```

Задает идентификатор набора для дальнейших сообщений. Идентификатором служит целое число в диапазоне `[1, NL_SETMAX]`. В пределах одного исходного файла сообщений идентификаторы должны задаваться в порядке возрастания. Подразумеваемым является набор `NL_SETD` (см. `<nl_types.h>`).

```
идентификатор_сообщения текст_сообщения
```

Идентификатором сообщения (в пределах набора) служит целое число в диапазоне `[1, NL_MSGMAX]`. В пределах одного набора идентификаторы сообщений задаются в порядке возрастания. Длина текста сообщения должна лежать в диапазоне `[0, NL_TEXTMAX]`.

Отметим, что генерируемые каталоги сообщений могут иметь двоичный формат и, следовательно, нуждаться в регенерации при переносе на другие системы. Иными словами, и здесь (как и везде в стандарте POSIX) речь идет о мобильности на уровне исходных текстов.

В прикладной программе работа с каталогами сообщений осуществляется посредством функций `catopen()`, `catgets()` и `catclose()` (см. листинг 13.21).

```
#include <nl_types.h>
nl_catd catopen (const char *name, int oflag);
char *catgets (nl_catd catd, int set_id,
               int msg_id, const char *s);
int catclose (nl_catd catd);
```

Листинг 13.21. Описание функций `catopen()`, `catgets()` и `catclose()`.

Функция `catopen()` открывает каталог сообщений и возвращает его дескриптор. Когда цепочка символов, на которую указывает аргумент `name`, содержит `/`, она трактуется как полное маршрутное имя. В противном случае (если пренебречь деталями) используется маршрут, зависящий от имени и реализации, а также от значения переменной окружения `LANG` (если аргумент `oflag` равен нулю) или `LC_MESSAGES` (если `oflag` равен `NL_CAT_LOCALE`).

К сожалению, рекомендации по расположению каталогов сообщений на момент написания данного текста еще не были выработаны. Стандарт POSIX-2001 ограничился замечанием о том, что авторам приложений следует соблюдать осторожность, дабы избежать конфликтов с другими приложениями и стандартными служебными программами.

Дескриптор, выданный функцией `catopen()`, используется в вызове `catgets()` для чтения сообщения под номером `msg_id` из набора с номером `set_id`. Аргумент `s` указывает на подразумеваемую цепочку символов, которая будет возвращена `catgets()` в качестве результата, если запрошенное сообщение не удастся прочитать из каталога.

Функция `catclose()` закрывает каталог сообщений.

Вариант 1

1. Стандартом POSIX-2001 предусмотрены следующие элементы категории LC_STYPE

- ☐ lower
- ☐ medium
- ☐ upper

2. Согласно стандарту POSIX-2001, функция setlocale() служит

- ☐ только для установки языково-культурной среды
- ☐ только для опроса языково-культурной среды
- ☐ и для установки, и для опроса языково-культурной среды

3. Согласно стандарту POSIX-2001, функция catopen()

- ☐ открывает текстовый файл и выдает его содержимое на стандартный вывод
- ☐ открывает каталог и выдает его содержимое на стандартный вывод
- ☐ открывает каталог сообщений и возвращает его дескриптор

Вариант 2

1. **Стандартом POSIX-2001 предусмотрены следующие элементы категории LC_MONETARY**
 - ☐ currency_symbol
 - ☐ decimal_point
 - ☐ thousands_sep

2. **Согласно стандарту POSIX-2001, функция localeconv() в случае неудачи возвращает в качестве результата**
 - ☐ (struct lconv *) NULL
 - ☐ -1
 - ☐ она не может кончиться неудачей

3. **Согласно стандарту POSIX-2001, третьим аргументом функции catgets() служит**
 - ☐ идентификатор очереди сообщений
 - ☐ номер сообщения в наборе
 - ☐ номер сообщения в очереди

Вариант 3

1. Стандартом POSIX-2001 предусмотрены следующие элементы категории LC_TIME

- ☐ day
- ☐ mon
- ☐ year

2. Согласно стандарту POSIX-2001, функция strerror()

- ☐ выдает в стандартный протокол сообщение, соответствующее номеру ошибки
- ☐ возвращает указатель на текст сообщения, соответствующего коду ошибки
- ☐ преобразует номер ошибки в текстовое представление

3. Согласно стандарту POSIX-2001, четвертым аргументом функции catgets() служит

- ☐ указатель на подразумеваемое имя каталога сообщений
- ☐ указатель на подразумеваемое имя языково-культурной среды
- ☐ указатель на подразумеваемый текст сообщения

Лекция 14. Заключение

Подводится итог курса, кратко суммируются полученные знания.

Ключевые слова: стандарт, POSIX, программное обеспечение, приложение, прикладной программный интерфейс, системный сервис, обязательный системный сервис, дополнительный системный сервис, аппаратно-программная платформа, операционная платформа, операционная система, системная архитектура, разрядность, мобильность программного обеспечения, мобильность на уровне исходного языка, командный язык, язык C, служебная программа, Unix-среда, соответствие стандарту POSIX, соответствие реализации, соответствие приложения, строгое соответствие, соответствие с использованием расширений, профиль, подпрофиль, неспецифицированное поведение, неопределенное поведение, поведение, зависящее от реализации, пользователь, файл, процесс, терминал, хост, узел сети, время, межпроцессное взаимодействие, права доступа, окружение, языково-культурная среда, пользователь, база данных пользователей, имя пользователя, идентификатор пользователя, начальный рабочий каталог, начальная программа, группа, член группы, начальная группа, база данных групп, имя группы, идентификатор группы, смена группы, поиск в базе данных, активный пользователь, почтовое взаимодействие, файл, тип файла, обычный файл, каталог, канал, символьный специальный файл, блочный специальный файл, символьная ссылка, сокет, файловая система, уникальный идентификатор файла, описатель файла, номер описателя файла, режим файла, владелец файла, владеющая группа, прочие пользователи, право на чтение, право на запись, право на выполнение, право на поиск, открытие файла, файловый дескриптор, получение информации о файлах, получение информации о файловых системах, смена атрибутов файлов, создание обычных файлов, создание каталогов, удаление файлов, копирование файлов, перемещение файлов, обход файловой иерархии, архивирование, операции ввода/вывода, функции нижнего уровня, буферизованный ввод/вывод, поток, закрытие файлов, временный файл, чтение данных, запись данных, позиционирование, текущая позиция, индикатор текущей позиции файла, управляющая операция, блокировка, управляющие операции с буферами, структурированные данные, текстовый файл, просмотр файла, символ, слово, строка, сортировка, слияние, сравнение содержимого, контроль целостности, регулярное выражение, базовое

регулярное выражение, расширенное регулярное выражение, преобразование символов, сжатие файлов, аналог реляционной операции для текстовых файлов, проекция отношения, горизонтальное соединение, соединение отношений, обработка каталогов, генерация маршрутных имен, удовлетворяющих заданному шаблону, маршрутный префикс, процесс, адресное пространство, поток управления, ресурс, идентификатор процесса, родительский процесс, порожденный процесс, реальный идентификатор пользователя процесса, действующий идентификатор пользователя процесса, программа процесса, сеанс, группа процессов, дополнительные группы процесса, маска режима создания файлов, создание процесса, смена программы процесса, ожидание завершения процесса-потомка, регистрация функции, терминирование процесса, межпроцессное взаимодействие, локальное межпроцессное взаимодействие, канал, именованный канал, безымянный канал, сигнал, генерация сигнала, доставка сигнала, реакция на сигнал, подразумеваемая реакция на сигнал, маска сигналов, блокирование сигнала, игнорирование сигнала, обработка сигнала, набор сигналов, очередь сообщений, идентификатор очереди сообщений, ключ, прием сообщения, отправка сообщения, семафор, значение семафора, набор семафоров, разделяемый сегмент памяти, присоединение сегмента, отсоединение сегмента, терминал, терминальное устройство, очередь ввода, очередь вывода, канонический режим ввода, неканонический режим ввода, построчная буферизация, препроцессирование ввода, характеристика терминала, управление терминалом, хост, характеристика хостов, имя хоста, конфигурационный параметр, настройка на конфигурацию целевой системы, статическая конфигурационная информация, лимит, минимально допустимое значение, сеть, сетевой адрес, связывание, привязка, освобождение, отмена привязки, оконечная точка, сетевой интерфейс, октет, сетевой порядок байт, хостовый порядок байт, сокет, адрес сокета, адресное семейство, тип сокета, протокол, режим с установлением соединения, режим без установления соединения, прием данных, отправка данных, база данных о хостах как узлах сети, база данных сетей, база данных сетевых протоколов, база данных сетевых сервисов, опция сокета, начало отсчета времени, всемирное время, часовой пояс, сезонная поправка, секунда, часы, показания часов, разрешающая способность часов, такт часов, реальное время, астрономическое время, общесистемные часы, время выполнения, процессорное время, часы процессорного времени, виртуальное время процесса, таймер, таймер реального времени, таймер процессорного времени, таймер виртуального времени процесса, таймер профилирования, интервальный таймер, абсолютный таймер, срабатывание таймера,

взвод таймера, снятие таймера со взвода, приостановка выполнения процесса, языково-культурная среда, настройка на языково-культурную среду, локализация, интернационализация, категория, элемент, ключевое слово, каталог сообщений.

Основные идеи курса

Один из общепринятых способов обеспечения мобильности ПО — стандартизация окружения приложений: предоставляемых программных интерфейсов, утилит и т.п. На уровне системных сервисов подобное окружение описывает стандарт POSIX.

Стандарт POSIX-2001 включает в себя четыре части:

- 1) основные определения (общие для всех частей термины, концепции и интерфейсы);
- 2) описание прикладного программного C-интерфейса к системным сервисам;
- 3) описание интерфейса к системным сервисам на уровне командного языка и служебных программ;
- 4) детальное разъяснение положений стандарта, обоснование принятых решений.

Стандарт POSIX описывает множество базовых, системных сервисов, необходимых для функционирования прикладных программ. Доступ к ним предоставляется посредством интерфейса, специфицированного для языка C, командного языка и общеупотребительных служебных программ.

Основная цель стандарта POSIX — сделать приложения мобильными на уровне исходного языка по отношению к смене аппаратно-программной платформы. О мобильности выполнимых программ и/или объектных файлов речь не идет.

POSIX нейтрален по отношению к системной архитектуре и разрядности процессора. Это очень важный аспект мобильности приложений.

Стандарт POSIX не ограничен рамками Unix-среды. Существуют и другие операционные системы (например, системы реального времени), предоставляющие необходимые сервисы и тем самым поддерживающие выполнение POSIX-совместимых приложений. Можно утверждать, что следование стандарту POSIX облегчает перенос приложений практически на любую сколько-нибудь распространенную операционную платформу.

В стандарте POSIX проведено разделение на обязательные и дополнительные системные сервисы, причем обязательное ядро сделано по возможности компактным. Для рассмотрения в данном курсе были отобраны в основном обязательные возможности; другие (среди которых выбирались давно устоявшиеся, присутствующие во многих исторически

сложившихся реализациях) привлекались только тогда, когда явно не хватало функциональности для создания приложений.

Основная задача курса – рассмотрение приемов и методов использования стандартизованных служебных программ и функций. Не ставилась цель пересказать стандарт, осветив все тонкости реализации ОС, все возможные коды ошибок и т.п. Главное, на наш взгляд, прочувствовать дух стандарта, научиться мобильным образом применять заложенные в нем возможности.

Предполагается, что читатель владеет языком C; мы не рассматривали ни синтаксис, ни хрестоматийные библиотечные функции этого языка. По отношению к стандартному командному языку и его интерпретатору аналогичного предположения не делалось. Эта тема изложена довольно подробно, хотя многие практикующие программисты предпочитают пользоваться другими интерпретаторами.

Мы не противопоставляли программирование на уровне командного языка и языка C. И то, и другое, порознь или совместно, может служить адекватным инструментом реализации приложений. Соответственно, рассматриваемые интерфейсы группировались по их предметной направленности, а не по языку реализации.

Значительное место в курсе и по объему, и по роли отведено примерам программ. Многие положения стандарта (связанные, скажем, с обработкой ошибочных ситуаций) излагаются не в основном тексте, а в этих примерах. По возможности они компилировались и выполнялись на нескольких аппаратно-программных платформах, в той или иной степени претендующих на соответствие стандарту POSIX. Тем не менее, недосмотры, конечно, возможны. Мы будем признательны за все замечания и предложения, относящиеся как к курсу в целом, так и к отдельным примерам программ.

Основные понятия, рассматриваемые в курсе

Важнейшим является понятие соответствия стандарту POSIX. У него есть две стороны: соответствие реализации (операционной системы) и соответствие приложения.

Реализация (операционная система), соответствующая стандарту POSIX, должна поддерживать все обязательные служебные программы, функции, заголовочные файлы с обеспечением специфицированного в стандарте поведения.

Для приложений предусмотрено как строгое соответствие (его главный отличительный признак – ограничение круга используемых возможностей рамками стандарта), так и соответствие с использованием расширений.

Профилем называется набор опций, описывающих необязательные возможности. Соответствие профилю означает соответствие стандарту POSIX и поддержку заданных возможностей. Разумным образом выбранные профили позволяют учитывать потребности представительных классов пользователей и/или приложений.

Допускается существование «подпрофилей», описывающих подмножества стандартных возможностей. Реализация, соответствующая подпрофилю, может функционировать на аппаратных платформах с ограниченными ресурсами и/или обслуживать нужды специфических приложений.

К числу важнейших принадлежат понятия, описывающие поведение реализации в различных ситуациях. Для многих корректных ситуаций поведение может быть неспецифицированным; это значит, что мобильное приложение не должно полагаться на совпадение поведения разных реализаций. Для некорректных ситуаций возможно неопределенное поведение; приложение не только не должно полагаться на определенный характер подобного поведения — оно не должно совершать некорректных действий, вызывающих неопределенное поведение.

Еще один близкий термин — «поведение, зависящее от реализации» — дополнительно означает, что поведение реализации необходимо документировать.

Стандарт POSIX в редакции от 2003 года — весьма обширный, многогранный документ. Для настоящего курса были отобраны следующие основные понятия операционных систем:

- пользователь;
- файл;
- процесс;
- терминал;
- хост;
- узел сети;
- время;
- языково-культурная среда.

За пределами рассмотрения оставлены такие важные и сложные темы:

- средства разработки;
- средства реального времени;
- потоки управления;
- математические интерфейсы;
- пакетные сервисы;
- унаследованные интерфейсы.

В тексте стандарта POSIX можно найти следующие пояснения рассматриваемых в курсе основных понятий вместе со ссылками на атрибуты и операции.

1) У пользователя есть имя и числовой идентификатор.

- 2) Файл – объект, допускающий чтение и/или запись и имеющий такие атрибуты, как права доступа и тип. В число типов входят обычный файл, символьный и блочный специальные файлы, канал, символьная ссылка, сокет и каталог. Реализация может поддерживать и другие типы файлов.
- 3) Процесс – адресное пространство вместе с выполняемыми в нем потоками управления и необходимыми для этих потоков системными ресурсами.
- 4) Терминал (или Терминальное устройство) – символьный специальный файл, подчиняющийся спецификациям общего терминального интерфейса.
- 5) Сеть – совокупность взаимосвязанных хостов.
- 6) Языково-культурная среда – часть пользовательского окружения, зависящая от языковых и культурных соглашений.

Основными операциями, применимыми к объектам ОС, являются чтение, запись и выполнение. Механизм прав доступа позволяет избирательно разрешать и запрещать выполнение подобных операций.

В POSIX-совместимых ОС определены объекты, которые можно называть вспомогательными; они помогают организовать взаимодействие между основными сущностями. Особенно широк спектр средств межпроцессного взаимодействия.

Средства, обслуживающие понятие пользователя

Операционная система, соответствующая стандарту POSIX, должна поддерживать базу данных пользователей, в которой о каждом из них хранится по крайней мере следующая информация:

- имя пользователя;
- числовой идентификатор пользователя;
- числовой идентификатор начальной группы;
- начальный рабочий каталог;
- начальная программа пользователя.

Над базой данных пользователей определены операции поиска по идентификатору или имени пользователя, реализуемые, соответственно, функциями `getpwuid()` и `getpwnam()`.

Пользователи объединяются в группы; каждый является членом по крайней мере одной группы. Для групп, как и для пользователей, существует база данных, записи которой содержат по крайней мере следующие поля:

- имя группы;
- числовой идентификатор группы;

- список пользователей, которым разрешено становиться членами данной группы.

Имеются функции для поиска в базе данных групп — `getgrgid()` и `getgrnam()`.

Опросить ассоциированные с пользователем данные позволяет служебная программа `id`.

Входное имя текущего пользователя можно узнать также с помощью утилиты `login` и функции `getlogin()`.

Для смены текущей группы пользователя предназначена служебная программа `newgrp`.

Чтобы узнать, какие пользователи в данный момент активны и за какими терминалами они работают в системе, можно воспользоваться служебной программой `who`.

Утилиты `writed`, `talk` и `mesg` позволяют в ограниченной форме организовать взаимодействие между пользователями.

Базовым средством обеспечения почтового взаимодействия, согласно стандарту POSIX-2001, является служебная программа `mailx`.

Средства, обслуживающие понятие файла

В трактовке стандарта POSIX понятие файла охватывает все, что может содержать, потреблять и/или поставлять информацию. Файл имеет такие атрибуты, как тип, имя и режим.

В стандарте зафиксированы следующие типы файлов:

- обычный файл;
- каталог;
- канал;
- символьный специальный файл;
- блочный специальный файл;
- символьная ссылка;
- сокет.

Файлы вместе со служебной информацией, хранящейся в объектах, которые называются описателями файлов, объединяются в иерархическую структуру, именуемую файловой системой.

В пределах файловой системы каждый файл имеет уникальный идентификатор (порядковый номер — он же номер описателя файла).

Согласно стандарту, с каждым файлом ассоциирована по крайней мере следующая служебная информация:

- режим — объект, содержащий биты режима и тип файла;
- числовой идентификатор владельца-пользователя;
- числовой идентификатор владеющей группы.

По отношению к конкретному файлу все пользователи делятся на три категории:

- владелец файла;
- члены владеющей группы;
- прочие пользователи.

Для каждой из этих категорий режим доступа определяет права на операции с файлом, а именно:

- право на чтение;
- право на запись;
- право на выполнение (для каталогов — право на поиск).

Для выполнения большинства операций с файлами их необходимо открыть. Открытому файлу соответствует файловый дескриптор — неотрицательное целое число, уникальное в пределах процесса и используемое для целей доступа к файлу. Дескриптор является ссылкой на описание открытого файла, хранящее смещение в файле, его статус и режимы доступа.

Для получения информации о файлах и файловых системах, а также для смены их атрибутов предназначены следующие служебные программы и функции:

- утилита `pwd` и функция `getcwd()` позволяют опросить абсолютное маршрутное имя текущего каталога;
- утилита `ls` и функции семейства `stat()` (`stat()`, `fstat()`, `lstat()`) выдают информацию о файлах;
- утилита `df` и функции `fstatvfs()` и `statvfs()` обеспечивают получение интегральной информации о файловых системах;
- служебная программа `du` выдает информацию о суммарном объеме пространства, занятого иерархиями файлов;
- утилита `cd` и функция `chdir()` позволяют изменить текущий каталог;
- утилиты `chown` и `chmod`, функции `chown()`, `fchown()`, `chmod()`, `fchmod()` служат для изменения таких атрибутов файлов, как владелец и режим доступа;
- утилита `touch` модифицирует время последнего доступа и/или изменения файла.

Создание обычных файлов обслуживается функцией `creat()`, создание каталогов — утилитой `mkdir` и одноименной функцией, создание каналов — утилитой `mkfifo` и одноименной функцией.

Новые ссылки на файл (жесткие или символичные) создают служебная программа `ln`, а также функции `link()` и `symlink()`.

Для удаления файлов служат утилиты `rm` и `rmdir`, функции `unlink()`, `rmdir()` и `remove()`.

Копирование файлов выполняется служебной программой `cp`, перемещение — программой `mv` или функцией `rename()`.

Обход файловой иерархии и систематическую обработку ее элементов осуществляет утилита `find`.

Одной из форм обхода и обработки файловой иерархии можно считать архивирование. Стандарт POSIX предусматривает для этого служебную программу `rar`.

В стандарте POSIX-2001 выделены две основные группы функций, обслуживающие операции ввода/вывода:

- функции нижнего уровня, использующие целочисленные файловые дескрипторы;
- функции более высокого уровня, осуществляющие буферизованный ввод/вывод с применением потоков.

Для открытия файлов и формирования новых описаний открытых файлов, файловых дескрипторов и потоков служат функции нижнего уровня `open()` и `pipe()`, а также функции буферизованного ввода/вывода `fopen()`, `fdopen()`, `freopen()`.

Для закрытия файлов предназначены функции `close()` и `fclose()`.

Весьма полезной с практической точки зрения является функция создания и открытия временных файлов `tmpfile()`.

Чтение данных из файла выполняют функции `read()` и `fread()`, запись - функции `write()` и `fwrite()`.

Содержимое символьных ссылок приходится читать особым образом, посредством функции `readlink()`.

Для буферизованного ввода/вывода байт предназначены функции `fgetc()` и `fputc()`, строки рекомендуется вводить, вызывая функцию `fgets()`, а выводить с помощью функций `fputs()` и `puts()`.

Индикатор текущей позиции файла может быть опрошен или передвинут посредством функции нижнего уровня `lseek()`, а также функций буферизованного ввода/вывода `fseek()`, `ftell()`, `ftello()`, `fgetpos()`, `fsetpos()`, `rewind()`.

Для выполнения разнообразных управляющих операций над открытым файлом предназначена функция `fcntl()`.

Особый класс управляющих операций с файлами со своей системой понятий составляют блокировки, хотя они также оформляются как команды функции `fcntl()`.

Функции `setbuf()`, `setvbuf()` и `fflush()` выполняют управляющие операции с буферами потоков.

Средства обработки структурированных данных

Простейшая разновидность структурированных данных — текстовые файлы. В свою очередь, простейшей операцией с файлами является их выдача на стандартный вывод. Для этого служит утилита `cat`.

Просмотр больших текстовых файлов предпочтительнее осуществлять при помощи служебной программы `more`.

Для просмотра нетекстовых файлов рекомендуется служебная программа `od`.

Полезная возможность — взглянуть на начало файла, она реализуется служебной программой `head`.

«Симметричная» служебная программа — `tail` — выдает на стандартный вывод конец исходного файла.

Служебная программа `pr` является фильтром для печати и оформления страниц.

Для подсчета числа символов, слов и строк в файлах служит утилита `wc`.

Служебная программа `sort` в зависимости от заданных опций выполняет одно из трех возможных действий:

- сортирует строки всех исходных файлов с записью результата в выходной файл;
- производит слияние всех исходных (предварительно отсортированных) файлов и записывает результат в выходной файл;
- проверяет, действительно ли отсортирован единственный исходный файл.

Утилита `diff` сравнивает содержимое исходных файлов и выдает на стандартный вывод список изменений, которые необходимо произвести, чтобы преобразовать один файл в другой.

Если нужно проверить на совпадение два файла, предпочтительнее воспользоваться более простой и быстрой служебной программой `cmp`.

Еще одним средством выявления различий (и совпадений) текстовых файлов является служебная программа `comm`.

Для контроля целостности файлов предназначена служебная программа `cksum`.

Понятие регулярного выражения (РВ) — одно из важнейших для программ обработки текстовых файлов. Согласно стандарту POSIX-2001, регулярное выражение — это шаблон, служащий для выборки определенных цепочек символов из множества подобных цепочек. Говорят, что выбранные цепочки удовлетворяют (успешно сопоставляются с) РВ.

Различают базовые (БРВ) и расширенные (РРВ) регулярные выражения.

Наиболее употребительной служебной программой, использующей механизм регулярных выражений, является `grep`.

В командных файлах обработка текстов часто производится потоковым редактором `sed`.

Еще одно популярное средство обработки текстовых файлов — служебная программа `awk`. Она выполняет программы, написанные на одноименном языке программирования.

На уровне функций работа с регулярными выражениями поддержана семейством `regex`: `regcomp()`, `regexec()`, `regfree()`, `regerror()`.

Идейно простое, но весьма мощное и полезное средство обработки текстовых файлов — служебная программа преобразования символов `tr`.

Служебная программа `uniq` позволяет ужать подряд идущие одинаковые строки до одной.

Стандартом POSIX-2001 предусмотрены аналоги реляционных операций для текстовых файлов:

- `cut` — выполняет операцию проекции отношения;
- `paste` — осуществляет горизонтальное соединение;
- `join` — выдает на стандартный вывод результат соединения двух отношений.

Каталог также можно рассматривать как набор структурированных данных. Обработка каталогов начинается с их открытия. Для этого предназначена функция `opendir()`.

После открытия текущим становится первый элемент каталога. Если в дальнейшем понадобится вновь позиционироваться на первый элемент, можно воспользоваться функцией `rewinddir()`.

Чтение элементов каталога выполняет функция `readdir()`.

После завершения работы с каталогом его следует закрыть с помощью функции `closedir()`.

Нередко сочетаются чтение элементов каталога и сопоставление с шаблоном имен файлов. Подобное сопоставление реализует функция `fnmatch()`.

Для генерации маршрутных имен, удовлетворяющих заданному шаблону, стандартом POSIX-2001 предусмотрена функция `glob()`.

Выделение из маршрутного имени файла простого имени выполняет утилита `basename`. Маршрутный префикс выделяется служебной программой `dirname`.

Процессы

Согласно стандарту POSIX-2001, процесс — это адресное пространство вместе с выполняемыми в нем потоками управления, а также системными ресурсами, которые этим потокам требуются.

Каждый процесс обладает целым рядом атрибутов. Важнейшим среди них является идентификатор процесса — положительное целое число, однозначно идентифицирующее процесс в течение времени его жизни.

Процесс, создавший данный, называется родительским.

С каждым процессом ассоциируется идентификатор создавшего его пользователя. Этот атрибут называется реальным идентификатором пользователя процесса.

Для определения прав процесса (в том числе прав доступа к файлам) используются действующие идентификаторы пользователя и группы, которые в общем случае могут отличаться от реальных.

Поведение процесса определяется исполняемой в его рамках программой.

Для выдачи информации о процессах служит утилита `ps`.

Опрос идентификаторов процесса, родительского процесса и группы процессов выполняется посредством функций `getpid()` и `getppid()` `getpgrp()`.

Для установки идентификатора группы процессов предназначена функция `setpgid()`.

За создание сеанса и установку идентификатора группы процессов отвечает функция `setsid()`.

Опрос реальных и действующих идентификаторов пользователя и группы вызывающего процесса осуществляется с помощью функций `getuid()`, `geteuid()`, `getgid()`, `getegid()`.

Функция `getgroups()` предназначена для получения идентификаторов дополнительных групп вызывающего процесса.

Переустановить действующий идентификатор пользователя вызывающего процесса позволяют функции `setuid()` и `seteuid()`. Аналогичные функции для переустановки идентификаторов группы процесса называются `setgid()` и `setegid()`.

Опрос и/или изменение маски режима создания файлов вызывающего процесса осуществляет служебная программа `umask` и одноименная функция.

Новые процессы создаются при помощи функции `fork()`.

Обычно процесс-потомок, используя функцию семейства `exec()`, подменяет программу, которая определяет поведение процесса, и передает ей управление и список аргументов. К числу функций этого семейства принадлежат `execl()`, `execv()`, `execle()`, `execve()`, `execlp()`, `execvp()`.

Родительский процесс реализует ожидание завершения процессов-потомков и получает информацию о статусе завершения от функций семейства `wait()` — `wait()` и `waitpid()`.

Процесс может вызвать собственное завершение, обратившись к функциям семейства `exit()` — `exit()`, `_Exit()`, `_exit()`.

Функция `atexit()` позволяет зарегистрировать функции, которые будут вызываться, если процесс завершается, обращаясь к `exit()` или возвращаясь из `main()`.

Для терминирования процессов извне предназначена служебная программа `kill`.

Средства межпроцессного взаимодействия

Средства локального межпроцессного взаимодействия реализуют высокопроизводительную, детерминированную передачу данных между процессами в пределах одной системы.

К числу наиболее простых и в то же время самых употребительных средств межпроцессного взаимодействия принадлежат каналы, представляемые файлами соответствующего типа. Стандарт POSIX-2001 различает именованные и безымянные каналы.

Взаимодействие между процессами через канал может быть установлено следующим образом: один из процессов с помощью функций `open()` или `pipe()` создает канал и передает другому соответствующий открытый файловый дескриптор. После этого процессы обмениваются данными через канал посредством функций `read()` и `write()`.

По сравнению с `pipe()` функция `open()` является более высокоуровневой. Она делает сразу несколько вещей: порождает процесс, обеспечивает выполнение заданной команды в его рамках, организует канал между вызывающим и порожденным процессами и формирует необходимые потоки для этого канала.

Канал остается открытым до тех пор, пока не будет вызвана функция `pclose()`.

Согласно стандарту POSIX-2001, под сигналом понимается механизм, с помощью которого процесс или поток управления уведомляют о некотором событии, произошедшем в системе, или подвергаются воздействию этого события.

В каждом процессе определены действия, предпринимаемые в ответ на все предусмотренные системой сигналы.

У каждого потока управления есть маска сигналов, определяющая набор блокируемых сигналов.

С сигналом могут быть ассоциированы действия одного из трех типов.

`SIG_DFL`

Подразумеваемые действия, зависящие от сигнала. Они описаны в заголовочном файле `<signal.h>`.

SIG_IGN

Игнорировать сигнал. Доставка сигнала не оказывает воздействия на процесс.

указатель на функцию

Обработать сигнал, выполнив при его доставке заданную функцию. После завершения функции обработки процесс возобновляет выполнение с точки прерывания.

К средствам генерации сигналов относятся служебная программа `kill` и одноименная функция.

Процесс (поток управления) может послать сигнал самому себе с помощью функции `raise()`.

Функция `abort()` вызывает аварийное завершение процесса.

Опросить и изменить способ обработки сигналов позволяет функция `sigaction()`.

Опросить и изменить способ обработки сигналов можно и на уровне командного интерпретатора, посредством специальной встроенной команды `trap`.

К техническим аспектам можно отнести работу с наборами сигналов. Ее выполняют инициализирующие набор функции `sigemptyset()` и `sigfillset()`, добавляющая сигнал `signo` к набору `set` функция `sigaddset()`, удаляющая сигнал функция `sigdelset()` и проверяющая вхождение в набор функция `sigismember()`.

Функция `sigprocmask()` предназначена для опроса и/или изменения маски сигналов процесса, определяющей набор блокируемых сигналов.

Функция `sigpending()` позволяет выяснить набор заблокированных сигналов, ожидающих доставки вызывающему процессу (потoku управления). Дождаться появления подобного сигнала можно с помощью функции `sigwait()`.

Функция `pause()` поможет дождаться доставки обрабатываемого или терминирующего процесс сигнала.

Функция `sigsuspend()` заменяет текущую маску сигналов вызывающего процесса, а затем переходит в состояние ожидания. Обычно парой функций `sigprocmask()` и `sigsuspend()` обрамляют критические интервалы.

Очереди сообщений, семафоры и разделяемые сегменты памяти отнесены к необязательной части стандарта POSIX-2001, именуемой «X/Open-расширение системного интерфейса» (XSI).

Каждую очередь сообщений, набор семафоров и разделяемый сегмент однозначно идентифицирует положительное целое число, которое

возвращается в качестве результатов функций `msgget()`, `semget()` и `shmget()`.

При получении идентификаторов средств межпроцессного взаимодействия используется еще одна сущность — ключ, для его генерации предназначена функция `ftok()`.

Опросить статус присутствующих в данный момент в системе (активных) средств межпроцессного взаимодействия позволяет служебная программа `ipcrm`.

Для удаления из системы активных средств межпроцессного взаимодействия предназначена служебная программа `ipcrm`.

Механизм очередей сообщений позволяет процессам взаимодействовать, обмениваясь данными, которые передаются между процессами дискретными порциями, называемыми сообщениями. Процессы выполняют над сообщениями две основные операции — прием и отправку.

Для работы с очередями сообщений в стандарте POSIX-2001 предусмотрены следующие функции: `msgget()` — получение идентификатора очереди сообщений, `msgctl()` — управление очередью сообщений, `msgsnd()` — отправка сообщения, `msgrcv()` — прием сообщения.

Согласно определению стандарта POSIX-2001, семафор — это минимальный примитив синхронизации, служащий основой для более сложных механизмов синхронизации, определенных в прикладной программе.

У семафора есть значение, которое представляется целым числом в диапазоне от 0 до 32767.

Семафоры создаются функцией `semget()` и обрабатываются функцией `semop()` определенными наборами (массивами), причем операции над наборами с точки зрения приложений являются атомарными. В рамках групповых операций для любого семафора из набора можно сделать следующее: увеличить значение, уменьшить значение, дождаться обнуления.

Процессы, обладающие соответствующими правами, могут выполнять также различные управляющие действия над семафорами. Для этого служит функция `semctl()`.

В стандарте POSIX-2001 разделяемый объект памяти определяется как объект, представляющий собой память, которая может быть параллельно отображена в адресное пространство более чем одного процесса.

Работа с разделяемой памятью начинается с того, что один из взаимодействующих процессов посредством функции `shmget()` создает разделяемый сегмент, специфицируя первоначальные права доступа к нему и его размер в байтах.

Для получения доступа к разделяемому сегменту его нужно присоединить посредством функции `shmat()`, т. е. разместить сегмент в виртуальном пространстве процесса. После присоединения, в соответствии с правами доступа, процессы могут читать данные из сегмента и записывать их

(быть может, синхронизируя свои действия с помощью семафоров). Когда разделяемый сегмент становится ненужным, его следует отсоединить с помощью функции `shmdt()`.

Предусмотрена возможность выполнения управляющих действий над разделяемыми сегментами (функция `shmctl()`).

Общий терминальный интерфейс

В стандарте POSIX-2001 терминал или терминальное устройство определяется как символьный специальный файл, удовлетворяющий спецификациям общего терминального интерфейса.

С каждым терминальным устройством ассоциированы очереди ввода и вывода.

Ввод может происходить в каноническом и неканоническом режимах. Канонический режим означает построчную буферизацию ввода системой, а также естественную обработку символов забора и уничтожения строки.

В неканоническом режиме входные данные не подвергаются пре-процессированию системой, а обработка запроса на чтение зависит от двух параметров — `MIN` и `TIME`.

Некоторые символы играют специальную роль при вводе и/или выводе. Например, символ `INTR` генерирует сигнал прерывания (`SIGINT`), посылаемый всем процессам, для которых данный терминал является управляющим.

Служебная программа `tty` выдает на стандартный вывод имя терминала, открытого в качестве стандартного ввода.

Узнать, ассоциирован ли открытый файловый дескриптор с терминальным устройством, а также получить имя этого устройства можно с помощью функций `isatty()` и `ttyname()`.

Каждый терминал обладает рядом характеристик, которые можно опросить и/или Изменить, используя утилиту `stty`.

Некоторые, хотя и весьма ограниченные, возможности управления терминалами предоставляет служебная программа `tput`.

На уровне функций опрос и изменение характеристик терминала разбиты на два семейства: `tc*()` и `cf*()`. В первое входят функции `tcgetattr()` — опрос, `tcsetattr()` — изменение, `tcflow()` — приостановка или возобновление терминального ввода/вывода, `tcflush()` — сброс очереди ввода, `tcdrain()` — ожидание физического окончания вывода, `tcsendbreak()` — разрыв соединения, `tcgetpgrp()` — получение идентификатора ассоциированной с терминалом группы процессов переднего плана, `tcsetpgrp()` — установка идентификатора группы,

`tcgetsid()` – опрос идентификатора группы процессов лидера сеанса, для которого терминал является управляющим.

Функции семейства `cf*` – `cfgetispeed()`, `cfgetospeed()`, `cfsetispeed()`, `cfsetospeed()` – служат для выборки/изменения данных о скорости терминального ввода/вывода.

Для получения маршрутного имени управляющего терминала служит функция `ctermid()`.

Опрос характеристик хостов

Самую общую информацию о характеристиках хоста позволяют получить служебная программа `uname` и одноименная функция.

Функция `gethostname()` возвращает имя хоста.

Значения конфигурационных параметров – важнейшая характеристика хоста, а настройка на конфигурацию целевой системы – обязательный элемент мобильного программирования приложений.

Основная часть статической конфигурационной информации сосредоточена в заголовочном файле `<unistd.h>`. К числу наиболее важных характеристик принадлежат `_POSIX_VERSION` (поддерживаемая версия системного интерфейса для языка C стандарта POSIX) и `_POSIX2_VERSION` (поддерживаемая версия интерфейса к системным сервисам на уровне командного языка и служебных программ).

Отдельная группа констант описывает поддерживаемые необязательные возможности стандарта POSIX-2001. Среди них `_POSIX_IPV6` (реализация поддерживает IPv6), `_POSIX_REGEX` (реализация поддерживает обработку регулярных выражений), `_POSIX_SHELL` (реализация поддерживает стандартный командный интерпретатор), `_POSIX_V6_ILP32_OFF32` (реализация предоставляет среду компиляции C-программ с 32-битными типами `int`, `long`, `off_t` и такими же указателями) и т.д.

Три константы задают номера файловых дескрипторов для стандартных ввода (`STDIN_FILENO` со значением 0), вывода (`STDOUT_FILENO` – 1) и протокола (`STDERR_FILENO` – 2).

Разного рода лимиты на количество и размеры заданы в заголовочном файле `<limits.h>`. Стандарт определяет минимально допустимые значения для подобных лимитов, которые должны поддерживаться реализациями и предоставляться приложениям.

Мобильным приложениям не следует полагаться на какие-то конкретные значения лимитов и требовать больше ресурсов, чем предусмотрено минимально допустимыми значениями.

Для опроса значений системных параметров во время выполнения предназначены служебная программа `getconf`, а также функции

`sysconf()`, `confstr()`, `fpathconf()` и `pathconf()`, первая из которых опрашивает лимитирующие конфигурационные параметры, имеющие числовые значения, вторая возвращает конфигурационные цепочки символов, а две последние — конфигурационные значения, относящиеся к файлам.

Сетевые средства

Стандарт POSIX-2001 определяет сеть как совокупность взаимосвязанных хостов.

Под сетевым адресом понимается видимый в пределах сети идентификатор, используемый для обозначения оконечных точек сети.

Процесс присвоения сетевого адреса оконечной точке называется связыванием, или привязкой, а обратное действие — освобождением, или отменой привязки.

Обычно оконечной точкой служит аппаратный сетевой интерфейс, посредством которого данные передаются и принимаются, однако с таким интерфейсом, как шлейфовый (`loopback`), никакой аппаратуры не ассоциировано.

Данные передаются по сети в виде последовательности октетов (восьмибитных беззнаковых величин). Если некоторый элемент данных (например, адрес или номер порта) состоит более чем из восьми бит, для его передачи и хранения требуется несколько октетов. Сетевым называется порядок октетов (байт), при котором первый (с наименьшим адресом) октет содержит старшие (наиболее значимые) биты.

При взаимодействии процессов оконечными точками служат сокеты, стандарт POSIX-2001 трактует их как отдельный тип файлов.

Под адресом сокета как (удаленной) оконечной точки понимается структура, включающая идентификатор адресного семейства и специфичную для данного семейства адресную информацию.

Адресное семейство соответствует определенной среде взаимодействия. Стандарт POSIX-2001 определяет три таких семейства: `AF_UNIX` (межпроцессное взаимодействие в пределах одной системы), `AF_INET` (взаимодействие по протоколу IPv4), `AF_INET6` (взаимодействие по протоколу IPv6 — необязательная возможность).

В пределах каждого адресного семейства могут существовать сокеты нескольких типов. Стандартом POSIX-2001 предусмотрено четыре типа: `SOCK_STREAM` (надежные, упорядоченные, полнодуплексные потоки октетов в режиме с установлением соединения), `SOCK_SEQPACKET` (налог `SOCK_STREAM` с дополнительным сохранением границ между записями), `SOCK_DGRAM` (передача данных в виде датаграмм в режиме без установления соединения), `SOCK_RAW` (аналог `SOCK_DGRAM` с дополнительной

возможностью доступа к протокольным заголовкам и другой информации нижнего уровня – необязательная возможность).

Для каждого адресного семейства каждый тип сокета может поддерживаться одним или несколькими протоколами. В частности, в адресном семействе `AF_INET` для сокетов типа `SOCK_STREAM` подразумеваемым является протокол с именем `IPPROTO_TCP`, а для типа `SOCK_DGRAM` – `IPPROTO_UDP`.

Общая логика работы с сокетами состоит в следующем. Сокеты создаются с помощью функции `socket()`, которой в качестве аргументов передаются адресное семейство, тип сокета и протокол, а в качестве результата получают открытый файловый дескриптор. Затем, посредством функции `bind()`, сокету присваивают локальный адрес. Если сокет ориентирован на режим с установлением соединения, то его следует помечать как готового принимать соединения, для чего понадобится функция `listen()`. Реальный прием соединений выполняет функция `accept()`, создающая для каждого из них новый сокет по образу и подобию «слушающего». В свою очередь, потенциальный партнер по взаимодействию инициирует соединение, прибегнув к функции `connect()`. (В режиме без установления соединения функция `connect()` позволяет специфицировать адрес отправляемых через сокет датаграмм.)

Для приема данных, поступивших в сокет, можно воспользоваться универсальной функцией низкоуровневого ввода/вывода `read()` или специализированным семейством функций `recv*()`, а для передачи – функцией `write()` или семейством `send*()`. Кроме того, посредством функций `select()` и/или `poll()` можно опросить наличие данных для приема или возможность отправки очередной порции данных.

Завершается взаимодействие между партнерами обращением к функции `shutdown()`.

Данные о хостах как узлах сети хранятся в сетевой базе, последовательный доступ к которой обслуживается функциями `sethostent()`, `gethostent()` и `endhostent()`.

Функция `getaddrinfo()` позволяет по имени узла сети (хоста) и/или имени сетевого сервиса получить набор адресов сокетов и ассоциированную информацию, что дает возможность создать сокет для обращения к заданному сервису.

Функция `freeaddrinfo()` носит технический характер: благодаря ей можно освободить память, зарезервированную функцией `getaddrinfo()`.

Функцию `getnameinfo()` можно считать обратной по отношению к `getaddrinfo()`. Она позволяет по адресу сокета узнать имя узла и сервиса.

Техническую роль играют и функции преобразования IP-адресов из текстового представления в числовое и наоборот: `inet_addr()`, `inet_ntoa()`, `inet_pton()`, `inet_ntop()`.

Первые две манипулируют только адресами IPv4: `inet_addr()` преобразует текстовую цепочку в целочисленное значение, пригодное для использования в качестве IP-адреса, `inet_ntoa()` выполняет обратное преобразование.

Вторая пара функций по сути аналогична первой, но имеет чуть более общий характер, так как способна преобразовывать адреса в формате IPv6.

Для преобразования значений типов `uint16_t` и `uint32_t` из хостового порядка байт в сетевой служат функции `htons()` и `htonl()`; функции `ntohs()` и `ntohl()` осуществляют обратную операцию.

Полезным дополнением к функциям `getaddrinfo()` и `getnameinfo()` является функция `gai_strerror()`, возвращающая текстовую цепочку, которая расшифровывает коды ошибок, перечисленные в заголовочном файле `<netdb.h>`.

Наряду с базой данных хостов (узлов сети), поддерживается база данных сетей с аналогичной логикой работы и набором функций: `setnetent()`, `getnetent()`, `getnetbyaddr()`, `getnetbyname()`, `endnetent()`.

Функция `getnetent()` обслуживает последовательный доступ к базе, `getnetbyaddr()` осуществляет поиск по адресному семейству и номеру сети, а `getnetbyname()` выбирает сеть с заданным (официальным) именем.

Точно такой же программный интерфейс предоставляет база данных сетевых протоколов: `setprotoent()`, `getprotoent()`, `getprotobyname()`, `getprotobynumber()`, `endprotoent()`.

Еще одно проявление той же логики работы – база данных сетевых сервисов: `setservent()`, `getservent()`, `getservbyname()`, `getservbyport()`, `endservent()`.

Для создания сокетов, помимо функции `socket()`, может быть использована функция `socketpair()`, создающая пару сокетов с установленным между ними соединением. Она обычно используется для адресного семейства `AF_UNIX`; поддержка других семейств не гарантируется.

Опросить присвоенный локальный адрес (его иногда называют именем сокета) можно с помощью функции `getsockname()`.

С сокетами могут быть ассоциированы опции, влияющие на их функционирование. Опросить или изменить значения этих опций помогут функции `getsockopt()` и `setsockopt()`.

Функция `getpeername()` позволяет опросить еще одну характеристику – адрес (имя) сокета, с которым установлено соединение.

После привязки сокета к локальному адресу и, возможно, установления соединения и задания значений опций, следует приступить к отправке и/или приему данных через сокет. Для этого служат функции `recvfrom()`, `recv()`, `recvmsg()`, `sendto()`, `send()`, `sendmsg()`.

Время

Согласно стандарту POSIX, за начало отсчета времени принимается ноль часов, ноль минут, ноль секунд первого января 1970-го года всемирного времени.

Всемирным называют поясное время нулевого часового пояса, которое представляет собой местное среднее солнечное время гринвичского меридиана.

За стандартную единицу измерения астрономического времени в POSIX-2001 принята секунда.

Часами называется программный или аппаратный объект, который может быть использован для измерения видимого или истинного хода времени.

Показания часов можно опросить и, возможно, установить (в допустимых для часов пределах).

Разрешающей способностью часов называется минимальный промежуток времени, измеряемый этими часами.

Тактом часов называются зависящие от реализации промежутки времени, на которые дробится каждая секунда.

Реальным (или астрономическим) называется время, измеренное по системным часам безотносительно к тому, какой процесс (поток управления) выполняется.

Под временем выполнения (процессорным временем) понимается время, затрачиваемое на выполнение процесса (потока управления), включая работающие от его имени системные сервисы.

Часами процессорного времени называются часы, измеряющие время выполнения конкретного процесса или потока управления.

Под виртуальным временем процесса понимается время, измеряемое системными часами, пока процесс выполняется.

Таймер — это механизм, способный известить процесс (поток управления) об истечении заданного промежутка времени (интервальный таймер) или о достижении (превышении) часами заданных показаний (абсолютный таймер). Соответствующее событие называется срабатыванием таймера.

Таймером процессорного времени называется таймер, ассоциированный с часами процессорного времени.

Взвести (зарядить) — значит запустить таймер, измеряющий ход времени и позволяющий уведомить процесс о наступлении заданного момента.

Таймер снимается со взвода (разряжается), когда он перестает измерять ход времени.

Простейшим средством опроса и/или изменения текущих даты и времени является служебная программа `date`.

На уровне языка C опрос текущего времени (в секундах от начала отсчета) выполняет функция `time()`.

Если тип `time_t` реализован как 32-разрядное целое со знаком, то в 2038-м году наступит переполнение (так называемая проблема 2038-го года).

С функцией `time()` ассоциирована функция `difftime()`, вычисляющая (в виде значения типа `double`) разность в секундах между двумя моментами времени.

Узнать текущее время с большей точностью позволяет функция `gettimeofday()`.

В необязательную часть стандарта POSIX-2001, регламентирующую работу с таймерами, входят функции, позволяющие опросить и установить показания заданных часов, а также узнать их разрешающую способность: `clock_gettime()`, `clock_settime()`, `clock_getres()`.

Часы могут быть общесистемными, т. е. видимыми для всех процессов, или принадлежащими процессу. Все реализации должны поддерживать общесистемные часы реального времени, измеряющие время от начала отсчета и имеющие идентификатор `CLOCK_REALTIME`.

Стандартом POSIX-2001 предусмотрено несколько способов представления данных о времени. Для выполнения преобразований между разными представлениями данных о времени служат функции `gmtime()`, `localtime()`, `mktime()`, `strftime()`, `strptime()`, `getdate()`.

Для учета данных о часовом поясе и сезонных поправках используются внешние переменные `tzname`, `timezone`, `daylight`, значения которых устанавливает по переменной окружения TZ функция `tzset()`.

Базовым средством для работы с часами процессорного времени является функция `clock()`. Она возвращает в качестве результата процессорное время, затраченное процессом с некоего момента, зависящего от реализации и связанного только с его (процесса) запуском.

Чтобы перевести время, возвращаемое функцией `clock()`, в секунды, его следует поделить на константу `CLOCKS_PER_SEC`, которая определена в заголовочном файле `<time.h>` равной одному миллиону.

Для измерения времени выполнения простой команды можно воспользоваться служебной программой `time`.

Более развитыми возможностями обладает специальная встроенная в shell команда `times`, она выдает на стандартный вывод процессорные времена, затраченные командным интерпретатором и порожденными им процессами.

Реализация утилит `time` и `times` опирается на функцию `times()`, опрашивающую данные о времени выполнения вызывающего процесса и порожденных процессов. Функция `times()` измеряет все времена в тактах часов. Соответственно, для перевода результатов работы `times()` в секунды их нужно делить на `sysconf(_SC_CLK_TCK)`, а не на `CLOCKS_PER_SEC`.

Прямое манипулирование часами процессорного времени возможно после обращения к функции `clock_gettime(clockid())`, позволяющей выяснить их идентификатор.

Для изменения ассоциированных с файлами данных о времени служит функция `utime()`, оперирующая в терминах секунд от начала отсчета.

Функция `sleep()` позволяет приостановить выполнение процесса (потока управления) на заданное число секунд.

Более современный аналог этой функции, `nanosleep()`, позволяет избавиться от ограничения на максимальную длительность приостановки выполнения и получить возможность задавать эту длительность с существенно более высокой точностью.

Описываемые далее средства для работы с интервальными таймерами входят в необязательную часть стандарта POSIX-2001, именуемую «X/Open-расширение системного интерфейса» (XSI).

Реализация должна предоставлять каждому процессу по крайней мере три интервальных таймера, обозначаемых следующими идентификаторами: `ITIMER_REAL` (таймер реального времени, генерирует сигнал `SIGALRM`), `ITIMER_VIRTUAL` (таймер виртуального времени процесса, сигнал `SIGVTALRM`), `ITIMER_PROF` (таймер профилирования, сигнал `SIGPROF`).

Согласно стандарту POSIX-2001, интервальные таймеры обслуживаются функциями `getitimer()` и `setitimer()`. Функция `getitimer()` запоминает текущие характеристики таймера, а `setitimer()` взводит или снимает таймер со звонка, устанавливая новые характеристики и запоминая старые.

К теме интервальных таймеров логически примыкает функция `alarm()`, позволяющая вызывающему процессу заказать доставку сигнала `SIGALRM` через заданное число секунд реального времени.

Языково-культурная среда

Согласно стандарту POSIX-2001, языково-культурная среда — это часть пользовательского окружения, зависящая от языковых и культурных соглашений.

Под настройкой на языково-культурную среду понимается процесс формирования данных, специфичных для поддержки конкретных естественных языков, местных настроек и кодировок символов. Иногда подобный процесс называют локализацией, в противоположность интернационализации — процессу подготовки приложений, способных настраиваться на различные языково-культурные среды.

Языково-культурная среда формируется из данных нескольких именованных категорий. Каждая управляет определенными аспектами поведения компонентов системы. Имена и назначение категорий соответствуют следующим переменным окружения: LC_TYPE (классификация символов, преобразование регистра), LC_COLLATE (порядок алфавитного сравнения символов), LC_MONETARY (форматирование денежных величин), LC_NUMERIC (форматирование числовых величин), LC_TIME (форматы даты и времени), LC_MESSAGES (форматы сообщений и интерактивных ответов).

Категории подразделяются на более мелкие элементы, средством именования которых служат ключевые слова.

В каждой реализации определены одна или несколько языково-культурных сред. Поддержка POSIX-среды с именами-синонимами «POSIX» и «C» является обязательной.

Стандартом POSIX-2001 предусмотрены две переменные окружения, определяющие все категории языково-культурной среды: LC_ALL (значение этой переменной учитывается в первую очередь), LANG (ее значение учитывается в последнюю очередь).

Общая логика использования средств интернационализации/локализации состоит в следующем. Пользователь, присваивая соответствующие значения описанным выше переменным окружения, идентифицирует свою языково-культурную среду. Приложения могут опросить данные о целевой среде с помощью функции `set locale()` и произвести настройку интернационализированных компонентов.

При создании новой языково-культурной среды ее определение в исходном формате помещается в файлы, которые должны быть обработаны утилитой `localedef`.

Для получения информации о языково-культурных средах предназначена служебная программа `locale`.

Функция `setlocale()` служит для установки и/или опроса всей языково-культурной среды вызывающего процесса или отдельных категорий.

Получить детальную информацию о категориях `LC_MONETARY` и `LC_NUMERIC` текущей языково-культурной среды поможет функция `localeconv()`.

Для преобразования денежных величин в цепочку символов в соответствии с настройками текущей языково-культурной среды можно воспользоваться функцией `strfmon()`, входящей в XSI-расширение стандарта POSIX-2001.

Если требуется получить детальную информацию обо всех аспектах языково-культурной среды, следует прибегнуть к функции `nl_langinfo()`, также отнесенной стандартом POSIX-2001 к расширению XSI.

Функция `strerror()` отображает номера (коды) ошибок в сообщении, зависящие от языково-культурной среды.

Развитые средства для работы с каталогами сообщений (диагностических, информационных), выдаваемых приложениями, вынесены в расширение XSI стандарта POSIX-2001. Идея состоит в том, чтобы в интернационализированных программах фигурировали не сами сообщения, а их идентификаторы в каталоге, который для каждой языково-культурной среды может быть своим.

Стандарт не специфицирует формат каталогов сообщений, но предлагает служебную программу `genscat` для их генерации по исходному описанию.

В прикладной программе работа с каталогами сообщений осуществляется посредством функций `catopen()`, `catgets()` и `catclose()`.

Функция `catopen()` открывает каталог сообщений и возвращает его дескриптор, который затем используется в вызове `catgets()` для чтения сообщения.

Функция `catclose()` закрывает каталог сообщений.

На этом мы завершаем описание базовых средств программирования приложений в стандарте POSIX.

Вариант 1

1. Основная задача настоящего курса состоит в

- ☐ детальном изложении стандарта POSIX-2001
- ☐ освещении тонкостей реализации операционных систем
- ☐ рассмотрении приемов и методов использования стандартизованных служебных программ и функций

2. Стандартом POSIX-2001 предусмотрены следующие группы функций, обслуживающие операции ввода/вывода

- ☐ функции физического уровня, непосредственно взаимодействующие с устройствами
- ☐ функции нижнего уровня, использующие целочисленные файловые дескрипторы
- ☐ функции более высокого уровня, осуществляющие буферизованный ввод/вывод с применением потоков

3. Для определения прав процесса используются

- ☐ действующий идентификатор пользователя процесса
- ☐ идентификатор процесса
- ☐ реальный идентификатор пользователя процесса

Вариант 2

1. В настоящем курсе рассматриваются

- ☐ библиотечные функции языка C
- ☐ синтаксис языка C
- ☐ средства разработки для языка C

2. В зависимости от заданных опций служебная программа `sort` выполняет следующие действия

- ☐ сортирует строки всех исходных файлов с записью результата в выходной файл
- ☐ производит слияние всех исходных (предварительно отсортированных) файлов и записывает результат в выходной файл
- ☐ проверяет, действительно ли отсортирован единственный исходный файл

3. Функции семейства `hexv*()` имеют

- ☐ два аргумента
- ☐ переменное число аргументов
- ☐ фиксированное число аргументов

Вариант 3

1. В настоящем курсе рассматриваются

- ☐ интерпретатор стандартного командного языка
- ☐ средства разбора опций командной строки
- ☐ стандартный командный язык

2. Согласно стандарту POSIX-2001, различают следующие виды регулярных выражений

- ☐ базовые
- ☐ полные
- ☐ расширенные

3. Согласно стандарту POSIX-2001, под сигналом понимается

- ☐ механизм, с помощью которого процесс уведомляют о некотором событии
- ☐ особый вид прерывания
- ☐ сообщение фиксированного формата

Обязательная литература

1. Информационная технология – интерфейс мобильной операционной системы (POSIX) – Часть I: Интерфейс прикладных программ (API) [Язык программирования C]. - Международный стандарт ISO/IEC 9945-1. ANSI/IEEE Std 1003.1. Вторая редакция 1996-07-12. - Том 1. Введение и нормативная часть. - М.: НИИСИ РАН, 1999, 423 с.
2. Информационная технология – интерфейс мобильной операционной системы (POSIX) – Часть I: Интерфейс прикладных программ (API) [Язык программирования C]. - Международный стандарт ISO/IEC 9945-1. ANSI/IEEE Std 1003.1. Вторая редакция 1996-07-12. - Том 2. Информативная часть (Приложения). - М.: НИИСИ РАН, 1999, 375 с.

Дополнительная литература

1. Богачев К.Ю.
Основы параллельного программирования.
М.: БИНОМ. Лаборатория знаний, 2003, 343 с.

Подготовительная литература

1. Керниган Б.В., Пайк Р.
UNIX – универсальная среда программирования (пер. с англ.)
М.: Финансы и статистика, 1992, 304 с.
2. Керниган Б., Ритчи Д.
Язык программирования Си (пер. с англ.)
М.: Финансы и статистика, 1992, 272 с.

Статьи по теме курса

- 1) Information technology – Portable Operating System Interface (POSIX) – Part 1: Base Definitions. – ISO/IEC 9945-1.2003.
- 2) Information technology – Portable Operating System Interface (POSIX) – Part 2: System Interface. – ISO/IEC 9945-2.2003.
- 3) Information technology – Portable Operating System Interface (POSIX) – Part 3: Shell and Utilities. – ISO/IEC 9945-3.2003.
- 4) Information technology – Portable Operating System Interface (POSIX) – Part 4: Rationale. – ISO/IEC 9945-4.2003.
- 5) Information technology – Programming languages – C. – ISO/IEC 9899:1999.
- 6) The Open Group Base Specifications Issue 6. IEEE Std 1003.1, 2003 Edition. – The IEEE and The Open Group, 2001-2003.
- 7) Галатенко А.В. О скрытых каналах и не только. – Jet Info, 2002, 11.

Сайты по теме курса

1. <http://www.opengroup.org/>
Web-сервер Open Group. Эта группа возглавляла процесс разработки стандарта POSIX-2001.
2. <http://www.ieee.org/>
Сервер Института инженеров по электротехнике и электронике, который разделяет с Open Group права на стандарт POSIX-2001.
3. <http://www.iso.ch/>
Сервер Международной организации по стандартизации.



Серия «Основы информационных технологий»

Интернет-университет информационных технологий представляет серию учебников «Основы информационных технологий». В рамках серии планируется выпуск более 120 книг по различным темам в области современных компьютерных систем и связанным с ними вопросам. Все книги серии выходят в едином оформлении и одинаковы по структуре организации материала. Каждая книга организована в виде курса лекций, который рассчитан на полугодовой или годовой учебный план и содержит несколько вариантов тестовых заданий к каждой лекции. Учебники проходят обязательную сертификацию учебно-методическими объединениями вузов и имеют гриф «рекомендовано» для соответствующих специальностей. В основу методик и принципов организации учебных планов положены стандарты документа Computing Curricula 2001 Computer Science (CC2001). Авторы курсов – ведущие российские специалисты в области информационных технологий, профессора и преподаватели вузов.

Книги издательства Интернет-Университета
Информационных Технологий всегда можно заказать на
сайте: **shop.intuit.ru**
или в интернет-магазине «Болеро»: **www.bolero.ru**

По вопросам оптовых закупок звоните
(095) 253-9312.

Наш адрес: Россия, Москва 123056, Электрический пер.,
дом 8, строение 3.

В серии изданы

1. Основы Web-технологий,

П.Б. Храмцов и др., 2003, 512 стр., ISBN 5-9556-0001-9

2. Основы сетей передачи данных,

В.Г. Олифер, Н.А. Олифер, 2003, 248 стр., ISBN 5-9556-0002-7

3. Основы информационной безопасности, 2-е издание,

В.А. Галатенко, 2004, 264 стр., ISBN 5-9556-0015-9

4. Основы микропроцессорной техники, 2-е издание,

Ю.В. Новиков, П.К. Скоробогатов, 2004, 440 стр.,
ISBN 5-9556-0004-3

5. Язык программирования Си++,

А.Л. Фридман, 2003, 288 стр., ISBN 5-9556-0005-1

6. Программирование на Java,

Н.А. Вязовик, 2003, 592 стр., ISBN 5-9556-0006-X

7. Стандарты информационной безопасности,

В.А. Галатенко, 2004, 328 стр., ISBN 5-9556-0007-8

8. Основы функционального программирования,

Л.В. Городняя, 2004, 280 стр., ISBN 5-9556-0008-6

9. Программирование в стандарте POSIX,

В.А. Галатенко, 2004, 560 стр., ISBN 5-9556-0011-6

10. Введение в теорию программирования,

С.В. Зыков, 2004, 400 стр., ISBN 5-9556-0009-4

Выйдут в первой половине 2004 года

11. Основы операционных систем,

В.Е. Карпов, К.А. Коньков, 2004, ISBN 5-9556-0012-4

12. Основы менеджмента программных проектов,

И.Н. Скопин, 2004, ISBN 5-9556-0013-2

13. Основы SQL,

Л.Н. Полякова, 2004, ISBN 5-9556-0014-0

Вниманию авторов!

Интернет-Университет Информационных Технологий приглашает авторов и творческие коллективы к участию в конкурсе на создание учебных курсов по компьютерным технологиям.

Интерес представляют любые курсы компьютерной тематики для учебных программ высшего и послевузовского образования. Наиболее приоритетными являются курсы по фундаментальным основам информационных технологий.

Все принятые курсы будут опубликованы на сайте Интернет-Университета информационных технологий, а также по ним будут опубликованы учебники в серии "Основы информационных технологий".

Стоимость работ по созданию учебного курса оценивает администрация в зависимости от его сложности и актуальности из расчета 60-300 рублей за 1000 печатных знаков. Дополнительно выплачивается авторский гонорар от реализации учебников в размере 10% от стоимости.

Внимание! Договор на публикацию и издание курсов в Интернет-университете информационных технологий не требует исключительных прав и допускает издание курса и его частей в других издательствах.

Подробная информация об условиях конкурса размещена на сайте Интернет-Университета информационных технологий [**www.intuit.ru/konkurs/**](http://www.intuit.ru/konkurs/).

По всем вопросам можно обращаться по адресу электронной почты: [*admin@intuit.ru*](mailto:admin@intuit.ru).

Серия «Основы информационных технологий»

В.А. Галатенко

Под редакцией академика РАН В.Б. Бетелина

Программирование в стандарте POSIX.

Курс лекций. Учебное пособие

Литературный редактор Т. Качинская

Компьютерная верстка Ю. Волшмид

Обложка М. Автономова

Подписано в печать 05.05.2004. Формат 60×90¹/₁₆.

Усл. печ. л. 35. Бумага офсетная. Тираж 2000 экз. Заказ № 4004.

Санитарно-эпидемиологическое заключение о соответствии санитарным
правилам №77.99.02.953.Д.006052.08.03 от 12.08.2003

ООО «ИНТУИТ.ру»

Интернет-Университет Информационных Технологий, www.intuit.ru
123056, Москва, Электрический пер., 8, стр.3.

Отпечатано с готовых диапозитивов на ФГУП ордена «Знак Почета»

Смоленская областная типография им. В.И.Смирнова.

Адрес: 214000, г.Смоленск, проспект им.Ю.Гагарина, д.2.

(с) Интернет-Университет Информационных Технологий
www.intuit.ru, 2004

