Высшее профессиональное образование

БАКАЛАВРИАТ

С. В. СИНИЦЫН, А. В. БАТАЕВ, Н. Ю. НАЛЮТИН

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Учебник

Рекомендовано
Учебно-методическим объединением по образованию
в области прикладной информатики в качестве учебника
для студентов высших учебных заведений, обучающихся
по направлению 230700 «Прикладная информатика»
и другим экономическим и техническим специальностям

3-е издание, стереотипное

ACADEMIA

Москва
Издательский центр «Академия»
2013

УДК 681.3.066(075.8)
ББК 32.973-018.2я73
С384

Рецензенты:
проф. кафедры математического обеспечения и администрирования информационных систем Московского государственного университета экономики, статистики и информатики канд. экон. наук, доцент *В. П. Грибанов*;
проф. кафедры управления и информатики Московского энергетического института (технического университета) д-р техн. наук *О. С. Колосов*

Синицын С. В.

С384 Операционные системы : учебник для студ. учреждений высш. проф. образования / С. В. Синицын, А. В. Батаев, Н. Ю. Налютин. — 3-е изд., стер. — М. : Издательский центр «Академия», 2013. — 304 с. — (Сер. Бакалавриат).

ISBN 978-5-4468-0412-2

Учебник создан в соответствии с Федеральным государственным образовательным стандартом по направлениям подготовки 230100 «Информатика и вычислительная техника», 010400 «Прикладная математика и информатика», 230700 «Прикладная информатика», 090900 «Информационная безопасность» (квалификация «бакалавр»).

Изложены основные принципы организации современных операционных систем (ОС) на примере ОС UNIX и Windows. Рассмотрены методы и языковые средства для работы с основными объектами, находящимися под управлением ОС: файлами, заданиями, пользователями, процессами. Значительное внимание уделено вопросам обеспечения межпроцессного взаимодействия. Текст иллюстрируется многочисленными примерами, содержит контрольные вопросы и задания.

Для студентов учреждений высшего профессионального образования.

УДК 681.3.066(075.8)
ББК 32.973-018.2я73

ISBN 978-5-4468-0412-2

ВВЕДЕНИЕ

Операционная система (ОС) — программный комплекс, предоставляющий пользователю среду для выполнения прикладных программ и управления ими, а прикладным программам средства доступа и управления аппаратными ресурсами.

Каждый пользователь ОС использует в своей деятельности инструменты, предоставляемые либо непосредственно ядром ОС, либо работающими под управлением ОС прикладными программами. Для решения своих задач пользователь формализует описание задачи на некотором входном языке для ОС или программ.

Синтаксис и семантика таких языков различна в зависимости от решаемых при их помощи задач. Эти задачи могут быть разделены на следующие группы:

- расширение функциональности ОС;
- конфигурирование режимов работы ОС;
- разработка прикладных программ;
- решение прикладных задач при помощи готовых программ.

Пользователей ОС, применяющих тот или иной язык общения с ОС, можно подразделить на несколько групп (рис. В1): системные программисты; системные администраторы; прикладные программисты; прикладные пользователи.

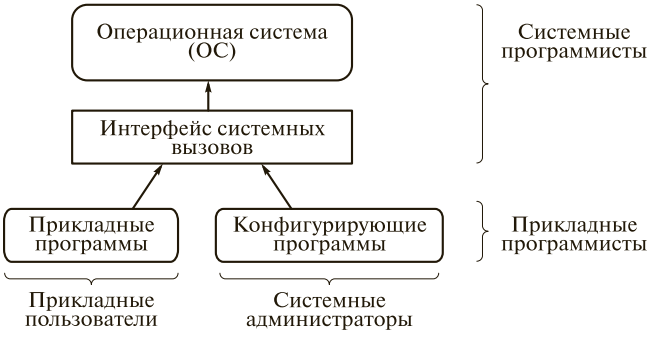

Рис. В1. Группы пользователей ОС

Один и тот же язык может служить для решения различных задач. Если более подробно классифицировать пользователей по используемому ими языку и решаемым задачам, то каждая пара «входной язык — решаемая задача» будет определять роль пользователя при его работе с ОС. В табл. В1 приведены основные типы ролей пользователей ОС.

Таблица В1. **Роли пользователей ОС**

Роль пользователя	Решаемая задача	Входной язык
Системный программист	Расширение функций ОС	Низкоуровневые языки разработки, в том числе Ассемблер
Системный администратор	Конфигурирование ОС и регистрация пользователей	Форматы конфигурационных файлов и языки управления средствами администрирования
Оператор	Текущее администрирование системы, установка/удаление программного обеспечения, его настройка	Форматы конфигурационных файлов инсталляторов и устанавливаемого программного обеспечения
Специалист по аппаратному обеспечению	Обслуживание аппаратуры, ввод ее в эксплуатацию, вывод из эксплуатации	Языки средств настройки оборудования для использования в конкретной ОС
Прикладной программист	Разработка программного обеспечения, предназначенного для решения задач прикладного пользователя	Языки высокого уровня, интерфейс системных вызовов ядра ОС
Администратор данных	Архивирование данных системы, управление информационными ресурсами (базы данных, справочники)	Языки управления и конфигурирования используемых программных средств
Прикладной пользователь	Решение конкретных прикладных задач при помощи готового программного обеспечения	Языки управления заданиями ОС, языки управления используемыми программными средствами

Данное учебное издание в первую очередь ориентировано на студентов, выступающих в роли прикладных программистов и пользователей ОС, но в некоторой мере затрагивает также вопросы администрирования и конфигурирования ОС.

Большая часть материала излагается на примере общих механизмов, существующих в ОС семейств UNIX и Windows. Часть материала применима и для ОС семейства DOS (в частности, раздел, посвященный заданиям в Windows).

Причина того, что речь идет о семействе ОС UNIX, а не об одной ОС, заключается в том, что семейство ОС UNIX развивалось значительное время в различных, зачастую не связанных друг с другом, коллективах разработчиков.

Первый вариант ОС UNIX был создан в 1969 г. несколькими программистами лаборатории Bell Labs фирмы AT&T и работал на компьютере PDP-7. Она использовалась для решения практических задач сотрудников лаборатории, и ее широкое распространение не планировалось. Через некоторое время большая часть ОС была переписана с языка Ассемблер на язык С, что дало возможность перенести ее на большое количество разных платформ. В настоящее время ОС Unix работает на большинстве существующих архитектур, и для многих из них является основной ОС.

Одна из следующих версий UNIX-систем, разработанных фирмой AT&T и их производных, называется System V (пятая версия), сокращенно SysV (иногда используют название «AT&T-версия UNIX»).

В середине 1970-х годов в университете Беркли была создана своя версия UNIX, получившая название BSD UNIX (Berkeley Software Distribution).

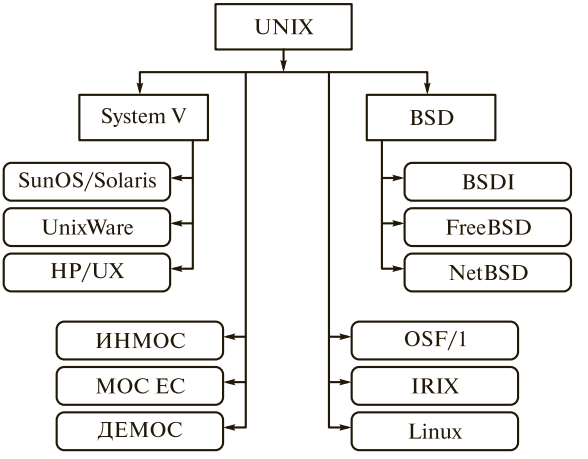

Рис. В2. Генеалогическое древо различных вариантов UNIX

Большинство вариантов ОС UNIX основаны или на System V, или на BSD (рис. В2).

Обе ветви UNIX-систем в той или иной степени удовлетворяют различным стандартам, в частности стандарту POSIX, и в настоящее время вырабатываются единые стандарты. Наиболее современные варианты UNIX, удовлетворяющие требованиям этих стандартов, нельзя четко отнести ни к той, ни к другой ветви. В их число входят IRIX (разработка Silicon Graphics), Digital OSF/1 (разработка DEC) и Linux, ранние версии которой были основаны на MINIX, разработанной Энди Таненбаумом.

Кроме того, к общему генеалогическому древу ОС семейства UNIX следует отнести отечественные разработки: ОС МОС ЕС для ЕС ЭВМ, ДЕМОС для ЕС ЭВМ и СМ ЭВМ, ИНМОС для СМ ЭВМ [6].

В качестве основной ОС рассматривается ОС Linux и Windows XP, однако большая часть материала книги применима для всех основных версий UNIX-систем и Windows семейства NT.

Учебник состоит из 9 глав и приложения:

• в главе 1 даны определения основных понятий и изложены принципы работы ОС, на которые опирается дальнейшее изложение;

• главы 2—6 дают читателю представление об основных объектах, поддерживаемых операционной системой: файлах, пользователях, заданиях и методах работы с ними;

• главы 6—9 углубляют знания читателя о файлах и пользователях. Также в этих главах вводятся в рассмотрение вопросы управления процессами и механизмы взаимодействия между процессами, поддерживаемые операционными системами UNIX и Windows;

• приложения содержат справочную информацию, а также полные исходные тексты и описание структуры данных системы «Контроль знаний», различные фрагменты которой служат примерами в первой части учебника.

Каждая глава учебника завершается контрольными вопросами и заданиями. Контрольные вопросы разделяются на две части — базовые вопросы, направленные на лучшее понимание и закрепление материала раздела, и задания, направленные на выработку практических навыков работы с UNIX- и Windows-системами.

Для понятного изложения материала большая часть приводимых примеров посвящена разработке программного комплекса, автоматизирующего процесс выдачи контрольных работ студентам и сбор выполненных контрольных работ преподавателем. При этом некоторые аспекты взаимодействия преподавателя со студентами намеренно упрощены в соответствии с изложением материала. Явно будут выделены только те моменты, которые хоро-

шо иллюстрируют различные механизмы, предоставляемые ОС пользователю.

В основной части учебника текст, относящийся к описанию программного комплекса «Контроль знаний», будет выделен шрифтом.

Программный комплекс «Контроль знаний» автоматизирует процесс выдачи преподавателем контрольных работ студентам и процесс возврата преподавателю выполненных контрольных работ (рис. В3).

Приложение «Контроль знаний» (см. приложения 1, 2) предназначено для трех основных типов пользователей:

• прикладной программист — выполняет разработку прикладной системы, расширяет ее функциональность;

• преподаватель — поддерживает базу вариантов контрольных работ по различным темам, выдает варианты работ студентам, собирает выполненные работы и проверяет их. Разбор работ и сообщение студентам оценок вынесено за рамки системы;

• студент — просматривает список полученных вариантов контрольных работ, выполняет их и возвращает выполненные работы преподавателю.

Программный комплекс «Контроль знаний» должен предоставлять средства для хранения базы вариантов контрольных работ, сгруппированных по темам. Каждая тема должна иметь уникальный номер, каждый вариант — номер, уникальный в пределах темы.

Для работы студентов им выделяется рабочая область, в которую преподавателем помещаются варианты контрольных работ, предназначенных для выполнения студентом. Внутри рабочей области студента должна быть выделена отдельная область, в которую он помещает выполненные работы. Именно из этой отдельной области преподаватель забирает работы на проверку, помещая их в свою рабочую область.

Таким образом, можно определить основные объекты, которые хранятся в программном комплексе «Контроль знаний», и действия над ними, выполняемые пользователями системы:

• база контрольных работ — основное хранилище информации о доступных вариантах. Основной объект данных — вариант. Варианты сгруппированы по темам, темы составляют общую базу;

• рабочая область студента — хранилище вариантов контрольных работ, выданных студенту. Каждый студент имеет свою собственную рабочую область. Основной объект данных — вариант контрольной работы;

• область готовых работ студента — хранилище выполненных студентом контрольных работ, готовых для проверки;

• рабочая область преподавателя — хранилище контрольных работ, собранных у студентов;

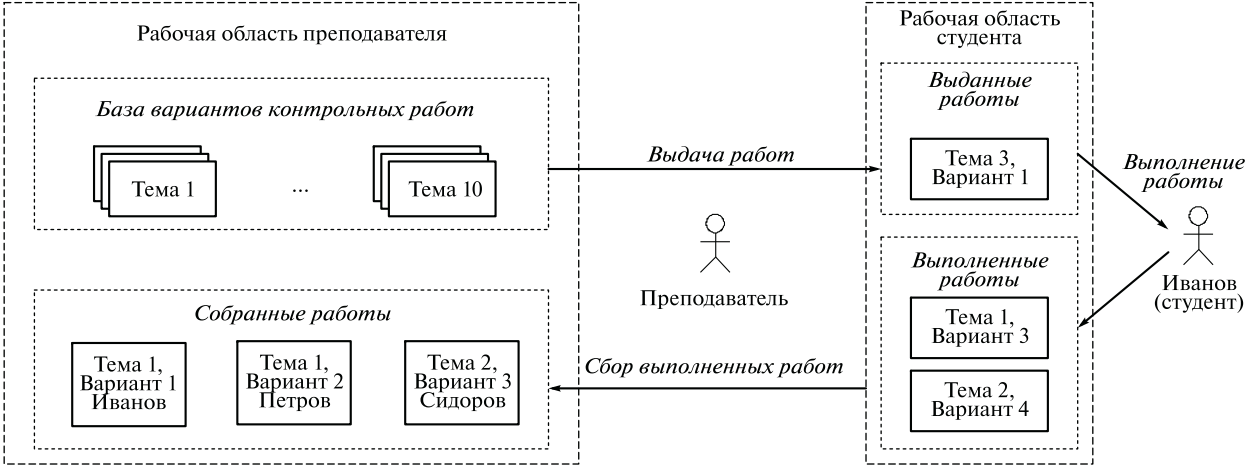

Рис. В3. Схема работы программного комплекса «Контроль знаний»

• вариант контрольной работы — список вопросов и полей, предназначенных для записи ответов студентом;

• выполненный вариант контрольной работы — вариант контрольной работы, в котором студент заполнил поля для ответов.

Программный комплекс должен автоматизировать следующие основные действия пользователя:

преподавателя:

• просмотр количества вариантов по определенной теме,

• выдачу одного варианта по теме конкретному студенту,

• выдачу вариантов по заданной теме всем студентам,

• сбор выполненных работ в свою рабочую область;

студента:

• просмотр полученных вариантов контрольных работ,

• выполнение контрольной работы,

• сдачу контрольной работы преподавателю.

При написании программного комплекса «Контроль знаний» должны учитываться следующие моменты:

• размещение и структуризация информации на дисковых носителях;

• определение прав доступа к различной информации;

• средства автоматизации типичных действий пользователя.

Прочитав внимательно учебник, вдумчивый читатель получит базовые знания о структуре ОС, навыки работы с ОС UNIX и Windows, а также общие представления о вопросах работы распределенных систем (пока только на примере межпроцессного взаимодействия). Однако круг вопросов, связанных с построением, администрированием и использованием операционных систем, гораздо шире.

В частности, в силу ограниченности объема не рассмотрены следующие вопросы:

• генерация ОС (конфигурирование ядра, администрирование) [5];

• графические интерфейсы (X Window System) [2];

• прикладные пакеты программ;

• специфика организации гетерогенных систем и сред [13];

• аспекты разработки дополнительных компонент ядра, драйверов устройств [7];

• программирование в операционных системах реального времени [14];

• микроядерные архитектуры [11];

• разработка сетевых приложений на основе различных протоколов связи [10].

Все эти вопросы заинтересованные читатели могут изучить самостоятельно.

Глава 1

ОБЩАЯ ХАРАКТЕРИСТИКА ОПЕРАЦИОННЫХ СИСТЕМ

1.1. Основные понятия

При решении задач в среде операционной системы (ОС) пользователь должен определить данные и инструментальное (программное) средство для их обработки. В большинстве случаев решение задачи пользователя сводится к последовательному применению нескольких инструментов (например, для ввода данных, сортировки, слияния, вывода).

Операционная система предоставляет пользователю базовый набор инструментов и среду для хранения данных, а также средства задания последовательности использования инструментов. Время, в течение которого пользователь решает последовательно одну или несколько задач, пользуясь при этом средствами, предоставляемыми ОС, называется *сеансом*. В начале любого сеанса пользователь идентифицирует себя, а в конце указывает на необходимость завершения сеанса. Последовательность использования инструментов, записанная на некотором формальном языке, называется *заданием*, сам язык — *языком управления заданиями*.

Выполнение заданий в большинстве ОС производится *командным интерпретатором*, более подробное определение которого будет дано в гл. 3. Обычно пользователю предоставляется некоторый интерфейс общения с командным интерпретатором, команды которого вводятся с клавиатуры, а результат их выполнения выводится на экран. Такой интерфейс ассоциируется с логическим понятием *терминала* — совокупности устройства ввода (обычно клавиатуры) и устройства вывода (дисплея, выводящего текстовую информацию). В настоящее время наиболее употребительным является графический интерфейс пользователя (GUI), рассмотрение которого выходит за рамки данной книги [2].

Операционная система выполняет функции управления аппаратными ресурсами, их распределения между выполняемыми программами пользователя и формирует некоторую среду, содержащую данные, необходимые для выполнения программ. Такая среда в дальнейшем будет называться *информационным окруже-*

нием. В информационное окружение входят все данные и объекты, обрабатываемые ОС, которые оказывают существенное влияние на выполнение программы. Далее будут приведены примеры информационного окружения различного характера.

Программа (в общем случае) — набор инструкций процессора, хранящийся на диске (или другом накопителе информации). Чтобы программа могла быть запущена на выполнение, ОС должна создать среду выполнения — информационное окружение, необходимое для выполнения программы. После этого ОС перемещает исполняемый код и данные программы в оперативную память и инициирует выполнение программы.

Используя понятия программы, данных и информационного окружения, можно определить задачу как совокупность программ и данных, являющихся частью информационного окружения.

Выполняемая программа образует процесс. *Процесс* — совокупность информационного окружения и области памяти, содержащей исполняемый код и данные программы. Обычно в памяти, адресуемой ОС, одновременно может работать большое число процессов. Естественно, что на однопроцессорных компьютерах возможно одновременное выполнение программного кода только одного процесса, поэтому часть процессов находится в режиме ожидания, а один из процессов — в режиме выполнения. Процессы при этом образуют очередь, ОС передает управление первому процессу в очереди, затем следующему и т. д.

Процесс, имеющий потенциальную возможность получить входные данные от пользователя с клавиатуры и вывести результаты своей работы на экран, называется *процессом переднего плана*, процесс, выполняемый без непосредственного взаимодействия с пользователем, — *фоновым процессом*.

В ходе работы процессы используют вычислительную мощность процессора, оперативную память, обращаются к внешним файлам, внутренним данным ядра ОС. Все эти объекты входят в информационное окружение процесса и называются *ресурсами*.

Ресурсом может быть как физический объект, к которому ОС предоставляет доступ — процессор, оперативная память, дисковые накопители, так и логический объект, который существует только в пределах самой ОС, например таблица выполняемых процессов или сетевых подключений. Необходимость в управлении ресурсами со стороны ОС вызвана, в первую очередь, тем, что ресурсы ограничены (по объему, времени использования, количеству обслуживаемых пользователей и т. п.). В этой ситуации ОС либо управляет лимитами ресурсов, предотвращая их исчерпание, либо предоставляет средства обработки ситуаций, связанных с исчерпанием ресурсов. Лимиты многих ресурсов, заданные в ОС по умолчанию, могут изменяться затем администратором системы

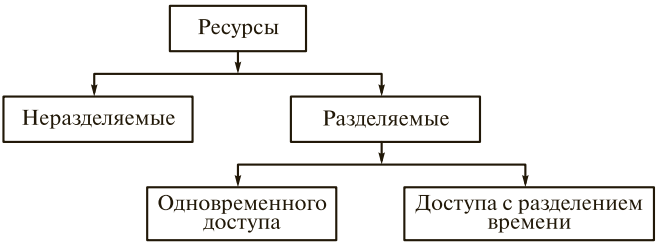

Рис. 1.1. Типы ресурсов ОС

(например, к таким ресурсам можно отнести количество файлов, одновременно открытых пользователем). В случае, если ОС позволяет одновременно использовать ресурсы нескольким процессам, ресурсы такой ОС подразделяют на типы, указанные на рис. 1.1 [1].

Неразделяемые ресурсы могут быть использованы на заданном отрезке времени только одним процессом, при этом другие процессы не имеют доступа к этому ресурсу до полного освобождения ресурса занявшим его процессом. Примером такого ресурса может служить файл, открытый на запись в исключительном режиме — все попытки использовать этот файл другими процессами (даже на чтение) завершаются неудачей.

Разделяемые ресурсы могут использоваться несколькими процессами. При этом к таким ресурсам возможен одновременный доступ процессов (например, к часам, при помощи которых определяется текущее системное время).

Некоторые разделяемые ресурсы не могут обеспечить одновременный доступ, но позволяют использовать их несколькими процессами, не дожидаясь момента полного освобождения ресурса. В этом случае используется квантование моментов использования ресурса по времени. В каждый квант времени один процесс получает полные и исключительные права на использование данного ресурса. При этом величина такого кванта заведомо много меньше полного времени, в течение которого ресурс используется одним процессом, т.е. времени, необходимого процессу для решения задачи пользователя.

Примером ресурса с доступом с разделением времени может служить процессорное время в многозадачных ОС — в каждый квант времени выполняется определенное число инструкций процесса, после чего управление передается следующему процессу и начинается выполнение его инструкций.

Процессы, ожидающие предоставления доступа к разделяемому ресурсу, организуются в очередь с приоритетом. Процессы с одинаковым приоритетом получают доступ к ресурсу последова-

тельными квантами, при этом некоторые процессы имеют более высокий приоритет и получают доступ к ресурсу чаще.

1.2. Типовая структура операционной системы

Обычно в составе ОС выделяют два уровня: ядро системы и вспомогательные системные программные средства, иногда называемые системными утилитами. Ядро выполняет все функции по управлению ресурсами системы — как физическими, так и логическими — и разделяет доступ пользователей (программ пользователей) к этим ресурсам. При помощи системного программного обеспечения пользователь управляет средствами, предоставляемыми ядром.

В ядро типичной ОС входят следующие компоненты: система управления сеансами пользователей, система управления задачами (процессами), файловая система, система ввода/вывода. Интерфейс ядра ОС с прикладными программами осуществляется при помощи программного интерфейса системных вызовов, интерфейс с аппаратным обеспечением — при помощи драйверов (рис. 1.2).

Система управления сеансами пользователей осуществляет регистрацию сеанса пользователя при начале его работы с ОС, хранит оперативную информацию, входящую в информационное окружение сеанса, при помощи системы ввода/вывода поддерживает соответствие пользовательского терминала реальным или

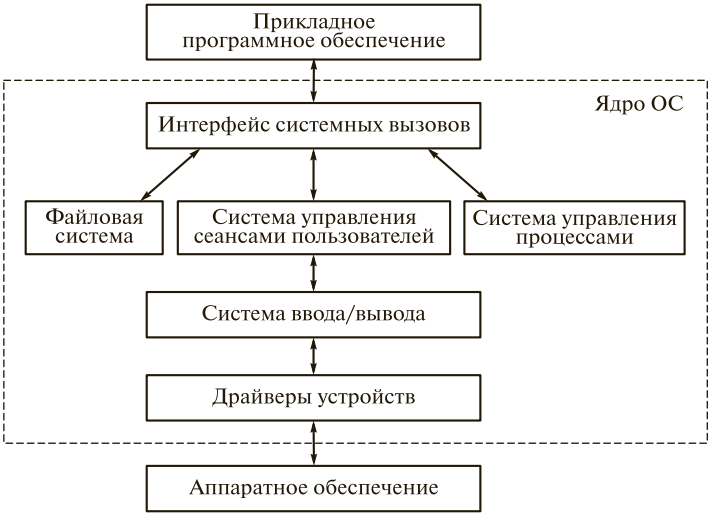

Рис. 1.2. Структура ядра типичной ОС

виртуальным устройствам, корректно завершает сеанс при окончании работы пользователя с системой.

Система управления процессами распределяет ресурсы между выполняемыми задачами (процессами), обеспечивает защиту памяти процессов от модификации ее другими процессами, реализует механизмы межпроцессного взаимодействия.

Файловая система выполняет преобразование данных, хранимых на внешних запоминающих устройствах (например, на дисковых накопителях или на flash-накопителях), в логические объекты — файлы и каталоги. Она также выполняет функции разграничения доступа к файлам и каталогам при обращении к ним со стороны системы управления сеансами или при использовании файловой системы через интерфейс системных вызовов.

Система ввода/вывода обрабатывает запросы всех рассмотренных выше компонентов ядра и преобразовывает их в вызовы логических устройств, поддерживаемых ОС. Каждое такое устройство представляет собой логический объект, обращение к которому происходит стандартными для ОС средствами (например, как к адресу в оперативной памяти либо как к специальному файлу). Логическое устройство может быть чисто виртуальным (целиком функционировать внутри ядра ОС) или представлять логический объект, связанный через драйверы с реальными аппаратными устройствами.

Примером чисто виртуального устройства может служить «черная дыра» */dev/null* в UNIX-системах. Вся информация, записываемая в это устройство, пропадает, т. е. оно может быть использовано для поглощения данных, не существенных для решаемой задачи.

Драйверы устройств преобразуют запросы системы ввода/вывода в последовательности управляющих команд для аппаратных устройств. Драйвер каждого устройства скрывает особенности его аппаратной реализации и предоставляет системе ввода/вывода стандартизированный интерфейс доступа к аппаратному обеспечению системы.

С точки зрения прикладного программиста доступ к компонентам ядра ОС осуществляется при помощи интерфейса системных вызовов — набора библиотек, включающих в себя стандартизованные наборы функций. Каждый такой набор предназначен для решения того или иного класса прикладных задач: доступа к сетевым ресурсам, графическому режиму, реализации межпроцессного взаимодействия и т. п.

1.3. Классификация операционных систем

Сложность составных частей ядра ОС и реализуемые ими функции, в первую очередь, зависят от числа одновременно обслужи-

ваемых ОС пользователей и от числа одновременно выполняемых процессов. В связи с этим разумно провести классификацию ОС по этим двум параметрам и рассмотреть особенности компонентов ядра в каждом из типов ОС.

По числу одновременно обслуживаемых пользователей ОС подразделяют на однопользовательские (одновременно поддерживается не более одного сеанса пользователя) и многопользовательские (одновременно поддерживается множество сеансов пользователя).

Многопользовательские системы, кроме обеспечения защиты данных пользователей от несанкционированного доступа других пользователей, предоставляют средства разделения общих данных между многими пользователями. Рассмотрим особенности этих типов ОС более подробно.

По числу одновременно выполняемых процессов ОС подразделяют на однозадачные (не более одного работающего процесса) и многозадачные (множество работающих процессов). Одним из основных отличий многозадачных систем от однозадачных является наличие средств управления доступом к ресурсам — разделения ресурсов и блокировки используемых ресурсов.

Однопользовательские ОС. Этот тип ОС обеспечивает единовременную поддержку только одного сеанса работы пользователя. Новый сеанс работы пользователя может быть начат только после завершения предыдущего сеанса. При этом новый сеанс пользователя имеет то же самое информационное окружение.

С точки зрения однопользовательской ОС пользователи неразличимы, поэтому если такую ОС начинают использовать несколько пользователей, то каждому из них операционная система предоставляет доступ ко всем ресурсам и, возможно, к одному и тому же информационному окружению. При этом пользователь может работать и со своими уникальными данными, например с данными на съемных дисках. При такой работе информационное окружение каждого сеанса работы пользователя различно.

Система управления сеансами однопользовательских ОС включает в себя только средства инициации и завершения сеанса и средства поддержки информационного окружения пользователя. Причем во многих однопользовательских ОС (например, DOS) момент инициации сеанса пользователя наступает сразу же после загрузки ядра и инициализационных сценариев. Момент завершения сеанса совпадает с моментом выгрузки ядра ОС из памяти (непосредственно перед или вследствие обесточивания оборудования). Таким образом, время жизни сеанса пользователя в однопользовательских ОС приблизительно равно времени жизни работающего ядра системы.

Вследствие неразличимости пользователей система управления сеансами и файловая система в значительной мере упрощаются. Система управления сеансами однопользовательских ОС не включает в себя средств идентификации и аутентификации пользователей, а также средств защиты информационного окружения их сеансов.

Файловая система однопользовательских ОС, как правило, не содержит сложных механизмов разграничения доступа к файлам и каталогам, хотя в файловой системе могут существовать флаги, задающие режимы работы с файлами и каталогами — их атрибуты.

Поддержка ОС только одного сеанса работы пользователя не исключает возможности одновременного выполнения многих задач пользователя. Иными словами, однопользовательская ОС может быть многозадачной.

Многопользовательские ОС. Этот тип ОС обеспечивает одновременную работу большого количества пользователей, что в значительной мере расширяет набор функций, реализуемых системой поддержки сеансов и файловой системой. В несколько меньшей степени поддержка множества пользователей отражается на системе ввода/вывода и системе управления процессами.

Система управления сеансами пользователей должна включать в себя средства идентификации и аутентификации пользователей, обеспечивать связывание каждого сеанса с реальным или виртуальным терминалом, содержать средства инициализации начального информационного окружения сеанса, обеспечивать защиту данных сеанса.

Файловая система многопользовательских ОС обеспечивает разграничение доступа к файлам и каталогам на основании идентификаторов пользователей, полученных от системы управления сеансами. Каждый файл и каталог в файловой системе сопровождается информационным блоком, определяющим права доступа к нему пользователей. Пользователю предоставляется возможность определять права таким образом, чтобы только он имел доступ к данным, содержащимся в файлах и каталогам, а другие пользователи не могли не только изменить эти данные, но даже прочитать их. При необходимости в совместном доступе к одной и той же информации может быть определен доступ на чтение и запись для многих пользователей.

Система ввода/вывода многопользовательских ОС, кроме непосредственного доступа к устройствам и буферизации ввода/вывода, также управляет разделением доступа пользователей к устройствам, т.е. управляет устройствами как разделяемыми ресурсами.

Следует отметить, что многопользовательские ОС обычно являются еще и многозадачными, поскольку они должны обеспечи-

вать одновременное выполнение большого количества программ различных пользователей.

Однозадачные ОС. Такие ОС предназначены для одновременного выполнения только одной задачи. Сразу после старта системы управление передается программе, играющей роль оболочки для работы пользователя. Как правило, одна из функций такой оболочки — запуск других программ.

Перед запуском программы сохраняется информационное окружение оболочки. После запуска программы ее процессу передается полное управление и предоставляется доступ ко всем ресурсам. По завершению программы освобождается память процесса, восстанавливается информационное окружение оболочки, после чего ОС берет управление на себя. Запуск программ в таких ОС чисто последовательный. В случае, если одной из программ требуется вызвать на выполнение другую программу, точно так же сохраняется окружение вызывающей программы и по завершению вызываемой программы окружение восстанавливается.

Система ввода/вывода однозадачных ОС не включает в себя средств разделения доступа к устройствам, поскольку устройство используется одновременно только одним процессом.

Однозадачные ОС могут быть и многопользовательскими. Примером таких систем являются ОС с пакетной обработкой. В таких ОС пользователи формируют очередь заданий на выполнение программ, при этом задания могут принадлежать различным пользователям. Система последовательно выполняет программы разных пользователей, перед сменой пользователя завершается сеанс работы предыдущего пользователя и начинается сеанс нового. Таким образом, при смене задания осуществляется смена информационных окружений каждой программы.

Многозадачные ОС. В многозадачных ОС в один момент времени в системе может быть запущено много программ (процессов). В этом случае система управления процессами включает в себя планировщик процессов, выполняющий следующие функции:

- создание и уничтожение процессов — загрузка программы в память, создание информационного окружения и передача управления процессу при его создании, удаление информационного окружения и выгрузка процесса из памяти при его уничтожении;
- распределение системных ресурсов между процессами — планирование выполнения процессов, формирование очереди процессов и управление приоритетами процессов в очереди;
- межпроцессное взаимодействие — распределение общих данных между процессами или пересылка управляющих воздействий между одновременно выполняемыми процессами;

• синхронизация выполнения процессов — приостановка выполнения процессов до достижения некоторых условий, например, посылки управляющего воздействия одним из процессов.

Система ввода/вывода в таких ОС сложнее, чем в однозадачных, так как любой ресурс (файл или устройство) может использоваться совместно несколькими процессами. Для предотвращения конфликтов доступа используется механизм блокировок, разрешающий доступ к неразделяемому ресурсу только одному процессу в один момент времени.

Операционные системы семейства UNIX относятся к многопользовательским многозадачным ОС. Именно поэтому они подробно рассмотрены в данной книге как среда разработки и эксплуатации прикладного программного обеспечения. Описаны только базовые средства ОС UNIX, при этом оставлены без внимания различные расширения, например графические средства X Window System.

Контрольные вопросы

1. Какие основные функции выполняет ОС?
2. В чем основное различие между разделяемыми ресурсами одновременного и разделяемого доступа?
3. Какие основные функции выполняет планировщик процессов?
4. Для чего в состав ядра ОС включается система ввода/вывода?
5. В чем состоят основные различия однопользовательских и многопользовательских ОС?
6. Может ли существовать ОС, в ядро которой не входит файловая подсистема?
7. Чем может быть ограничено число одновременно работающих в системе процессов?
8. В чем состоит основное усложнение ядра многопользовательской ОС по сравнению с однопользовательской?

Глава 2

ФАЙЛЫ

2.1. Организация хранения данных на диске

В ходе сеанса работы с системой пользователь создает и изменяет собственные данные, например, документы, изображения, тексты программ. В простейшем случае пользователь может набрать текст, распечатать его, получив твердую копию документа, и завершить свой сеанс работы с системой. При этом пользователь получает результат своей работы — распечатку, а набранный текст представляет собой промежуточные данные, которые не нужно хранить.

В более сложных случаях текст имеет большой размер и его нужно сохранять между сеансами работы пользователя — например, за один сеанс работы пользователь может набрать одну главу книги и продолжить редактирование в следующем сеансе.

Для продолжительного хранения данных пользователя используются *накопители данных* — дисковые (винчестеры, CD- и DVD-диски, флоппи-диски), ленточные (стримеры) или твердотельные (флеш-накопители). Взаимодействие с накопителем осуществляет ОС (при помощи файловой системы и драйверов устройств), а единицей пользовательских данных в этом случае является файл.

Файл можно рассматривать как хранимые на диске данные, имеющие уникальное имя, видимое пользователю. Данные файла имеют формат, предназначенный для их использования прикладными программами пользователя. Так, книга может храниться на диске в виде файла с именем `book.txt` в текстовом формате `txt`.

Имена файлов обычно представляют собой символьные строки ограниченной длины (как правило, до 8 или до 255 символов), которые кратко описывают данные, хранимые в файле. Обычно существует ряд символов, запрещенных к использованию в именах файлов. В UNIX-системах это символы `*`, `?`, `/`, `\`, `<`, `>`, `&`, `$`, `|` и др.

Для определения программы, которая работает с файлом, нужно задать его тип. Как правило, тип файла задается при помощи расширения — части имени файла, отделенного точкой. Так, для

файла `book.txt` `book` — имя, а `txt` — расширение. Внутренняя структура файла не зависит от расширения, а типизация файлов может производиться не только при помощи расширений. Например, в UNIX-системах исполняемые файлы программ обычно не имеют расширения, вместо этого им присваивается специальный атрибут «исполняемый файл».

Необходимость в типизации файлов вызвана тем, что в зависимости от программы, работающей с файлом, данные могут представляться совершенно по-разному. Например, текстовый файл, открытый в текстовом редакторе, будет представлять собой последовательность символов — текст, и в этом случае для программ проверки орфографии важны отдельные слова, составляющие этот текст, а для программы резервного копирования этот файл представляет собой просто последовательность блоков по 16 Кбайт. Таким образом, понятие файла ассоциировано с использующей этот файл программой, поскольку структура файла и правила работы с ней задаются именно программой.

Что же представляет собой файл с точки зрения ОС, а точнее, с точки зрения файловой системы как части ядра ОС?

Операционная система предоставляет прикладным программам интерфейс доступа к файлам, но при этом рассматривает данные, из которых состоят файлы, со своей стороны. Представим себе игрушечный конструктор, из которого можно собрать модель автомобиля. Точно так же ОС собирает файлы из отдельных «деталей» — блоков. При этом так же, как в конструкторе, к которому приложена схема сборки, ОС руководствуется правилами сборки отдельных блоков. Такие сконструированные наборы блоков получили название *наборов данных*.

Набор данных — более низкоуровневое представление данных, чем файл. Набор данных имеет уникальное имя, так же, как и файл, содержит все данные, которые содержит файл, но структура этих данных определяется ядром ОС. Кроме того, в набор данных может входить служебная информация, недоступная обычным программам.

Все дисковое пространство, используемое файловой системой, разбивается на отдельные блоки — *кластеры* (обычно имеющие размер 1, 2, 4, 8 или 16 Кбайт). Каждый кластер имеет свой номер и хранит либо данные пользователя, либо служебную информацию. Эта служебная информация используется в том числе и для сборки блоков в наборы данных. Размер кластера устанавливается при создании файловой системы. Например, служебный блок может хранить последовательность номеров блоков (кластеров), входящих в набор данных (рис. 2.1).

Недостатком такого подхода к организации данных является то, что список номеров блоков у больших файлов может занимать

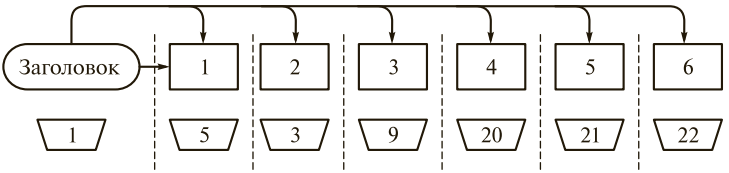

Рис. 2.1. Файловая система со ссылками на все блоки набора данных:
— служебный блок; — блок данных; — кластер

более одного кластера. В результате размер файла получается ограниченным. В UNIX-системах существуют различные методы обхода этого ограничения, например, служебные блоки можно организовать в виде дерева, при этом каждый служебный блок хранит последовательность блоков с данными и служебные блоки следующего уровня дерева (более подробно см. подразд. 6.4).

Второй подход к организации блоков данных состоит в том, что наборы данных размещаются в последовательных кластерах. В этом случае в служебном блоке достаточно хранить номера первого и последнего кластера (рис. 2.2). При частых изменениях данных этот метод организации неудобен — при увеличении размера набора данных нужно или всегда иметь достаточное количество свободных блоков после последнего, или каждый раз перемещать набор данных на свободное место. Тем не менее такой подход положен в основу организации данных на CD-ROM-носителях, а ранее использовались в ОС «РАФОС».

Третий подход заключается в выделении в каждом блоке небольшой служебной области, в которой хранится номер следующего блока в наборе данных (рис. 2.3). Таким образом, набор данных организуется в линейный список, а в служебном блоке достаточно хранить номер первого блока с пользовательскими данными.

Соглашения о структуре наборов данных на носителе являются частью файловой системы, которая состоит из трех компонентов:

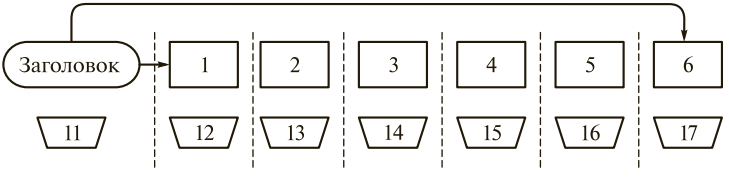

Рис. 2.2. Файловая система с последовательной организацией блоков данных:

— служебный блок; — блок данных; — кластер

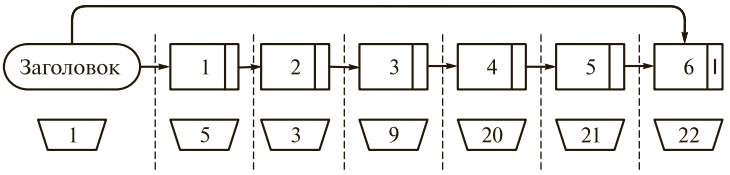

Рис. 2.3. Файловая система со списковой организацией блоков данных:

— служебный блок; — блок данных; — кластер

• соглашения о структуре хранения данных на носителе — определение типов информации, которая может быть размещена на носителе (например, пользовательская/служебная информация), а также набор правил, согласно которым данные размещаются на носителе;

• соглашения о правилах обработки данных — набор правил, согласно которым данные обрабатываются, например, «файл должен быть открыт до того, как в него будут записаны данные»;

• процедуры обработки данных, которые входят в состав ядра ОС и подчиняются определенным выше соглашениям.

Таким образом, говоря о файловой системе, часто нужно уточнять, о каком из ее аспектов идет речь.

В качестве идентификатора набора данных для ОС обычно служит номер кластера служебного блока. Операционная система поддерживает соответствие между идентификаторами наборов данных и именами файлов. Это позволяет скрывать от пользователя механизм обращения к данным — пользователь обращается к файлу по именам, а ОС находит по этому имени идентификатор набора данных.

Соответствие имен файлов и идентификаторов наборов данных хранится в специальной области дискового пространства, называемого *таблицей размещения файлов*. Каждому набору данных может быть сопоставлено несколько имен файлов. Каждое такое соответствие представляет собой отдельную запись в таблице размещения файлов и в UNIX-системах называется *жесткой ссылкой* (рис. 2.4).

Жесткая ссылка позволяет обращаться к одним и тем же данным при помощи различных имен файлов. Это может быть удобно в случае, если данные нужно обрабатывать при помощи нескольких программ, каждая из которых работает только с файлами с определенным расширением.

Поскольку в UNIX-системах в служебном блоке набора данных хранится информация о режиме доступа к данным (права доступа, см. подразд. 4.4), то эта информация будет одинаковой для всех жестких ссылок.

Рис. 2.4. Наборы данных, жесткие и символические ссылки:
— жесткая ссылка; — символическая ссылка

Кроме жестких ссылок существуют *символические ссылки* — файлы специального вида, хранящие имя файла, на который указывает эта ссылка. Таким образом, символическая ссылка указывает не на набор данных, а на файл. Это может быть удобно при частом изменении ссылки или в случае, если на разные ссылки нужно задать разные права доступа.

Структура наборов данных во многом определяет режим доступа к данным. Например, при помещении блоков набора данных в последовательные кластеры или при организации в форме линейного списка легко организовать *последовательный доступ* к данным — все обращения к заданному месту данных возможны только после предварительного перебора всех предыдущих блоков данных. Организация с хранением списка номеров блоков удобна для организации *произвольного доступа* — для получения определенного блока достаточно считать его номер из служебного блока.

Режим доступа к данным определяется не только файловой системой. Если продолжить рассмотрение структуры данных и ее изменений далее, до физической структуры данных на носителе, то очевидно, что данные проходят серию преобразований (рис. 2.5), при этом каждый потребитель этих данных (например, программа пользователя или файловая система) накладывает свои ограничения на режим доступа.

Приведенные выше представления данных задают *логическую структуру* данных — структуру, удобную для использования программным обеспечением и, как правило, не имеющую ничего общего с физическим представлением данных на носителе. Если рассматривать *физическую структуру* данных, то на самом низком уровне данные представлены в виде магнитных доменов (на жестких или гибких дисках), которые объединяются в сектора — минимально адресуемые области диска. Один магнитный домен представляет собой один бит информации, размер сектора обычно составляет 512 байт, хотя это значение может быть изменено при создании дискового раздела. При этом размер может варьироваться в диапазоне от 512 до 4 096 байт. Идентифицируется

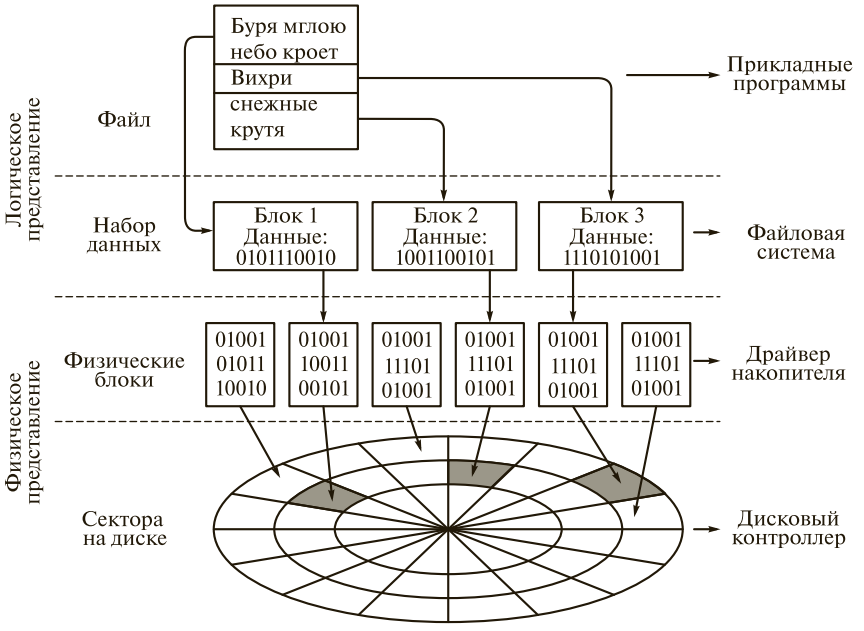

Рис. 2.5. Уровни представления данных

сектор при помощи номера считывающей головки, номера дорожки и номера сектора на дорожке. При считывании или записи секторов они проходят через дисковый контроллер, который управляется драйвером данного дискового накопителя, входящим в ядро ОС.

Интерфейс драйвера с дисковым контроллером определяет режим доступа к устройству: при символьном доступе информация считывается последовательно по одному символу, при блочном — блоками фиксированного размера (обычно кратного размеру сектора). Драйвер представляет собой дисковое пространство в виде неструктурированной последовательности пронумерованных физических блоков[1]. Размер физического блока обычно равен размеру сектора. При переходе от физического представления данных к логическому физические блоки объединяются в кластеры, с которыми работает файловая система.

Таким образом, вся подсистема работы с файлами распределена по четырем основным уровням, два верхних из которых опре-

[1] Существовали ОС, в которых данные на диске рассматривались как структурированные. Примером могут служить индексно-последовательные файлы в ОС ЕС (OS/360).

деляют логическое представление данных, а два нижних — физическое.

Поскольку основная работа прикладного программиста в UNIX-системах идет на уровне файловой системы, перечислим все ее основные функции. Файловая система определяет:

- соглашения об организации данных на носителях информации;
- соглашения о способе доступа к данным — последовательном или произвольном;
- соглашения об именовании файлов;
- соглашения о логической структуре данных;
- набор методов, входящих в состав ядра ОС, предназначенных для работы с данными на носителях по указанным выше соглашениям.

2.2. Каталоги

Чтобы упорядочивать и организовывать файлы, в ОС существует понятие *каталога*. Каталог содержит записи о файлах и других каталогах. Файлы и каталоги, записи о которых содержатся в каком-либо каталоге, считаются содержащимися в этом каталоге. Рекурсивность этого определения позволяет говорить о дереве каталогов — иерархической системе, служащей для организации файлов.

На рис. 2.6 элементы, имена которых начинаются с `dir`, являются каталогами; элементы, имена которых начинаются с `file`, — файлами. Если каталог содержится внутри другого каталога, то внешний каталог называется *родительским* для первого каталога, или *надкаталогом*. Содержащийся внутри каталог называется *подкаталогом*. Например, каталог `dir3` является родительским для `dir5`, а каталог `dir2` — подкаталогом `dir1`. Каталог, находящийся на верхнем уровне иерархии и не имеющий родительского каталога, называется *корневым каталогом*.

Использование каталогов в современных ОС обусловлено тремя факторами:

1) каталоги ускоряют поиск файла ОС. Поиск в дереве при прочих равных обычно происходит быстрее, чем в линейном списке;

2) каталоги позволяют уйти от уникальности имен файлов. Каждый файл должен иметь уникальное имя, но эта уникальность должна быть только в пределах каталога, содержащего файл;

3) каталоги позволяют классифицировать файлы на носителе. Обычно в один каталог помещают файлы, объединенные каким-то общим признаком — например, главы книги или загрузочные файлы операционной системы.

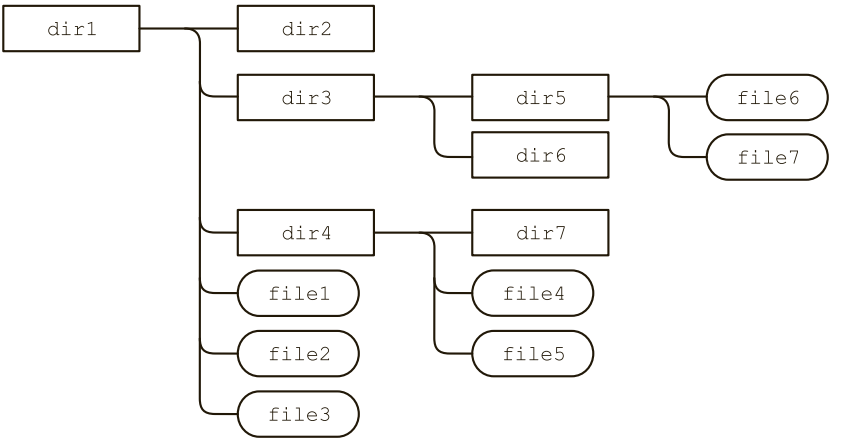

Рис. 2.6. Пример распределения файлов по каталогам

С точки зрения хранения в файловой системе каталог — это файл специального вида, в котором записаны имена и атрибуты содержащихся в нем файлов и каталогов. В отличие от обычных файлов к каталогам невозможен последовательный доступ — работа с каталогами организована несколько иначе, чем с файлами. Работая с каталогом, пользователь работает с отдельными записями в нем, т.е. с информационными блоками, содержащими информацию о находящихся в этом каталоге файлах и каталогах.

Простые ОС могут поддерживать файловые системы, имеющие только один каталог — корневой, в котором хранятся файлы. Более сложные ОС поддерживают работу с деревом каталогов практически неограниченной вложенности. Естественным ограничением глубины вложенности дерева каталогов является длина полных имен файлов, находящихся на нижнем уровне вложенности дерева.

Полное имя файла — текстовая строка, в которой через специальные символы-разделители указаны имена всех каталогов, задающие путь от корневого каталога до самого файла. В качестве разделителя в UNIX-системах используется символ прямой косой черты «/». Например, для файла `file6` полным именем будет `/dir1/dir3/dir5/file6`. Здесь первая косая черта обозначает корневой каталог, т.е. полным именем файла `file1`, находящегося в корневом каталоге, будет `/file1`.

Путь, задаваемый полным именем, называется *абсолютным путем файла или каталога*. Такое название обусловлено тем, что путь задается относительно абсолютной точки отсчета — корневого каталога. Чтобы определить понятие относительного пути,

нужно ввести понятие *текущего каталога*. В информационное окружение сеанса работы пользователя входит информация о каталоге, с которым пользователь работает в данный момент времени.

Имена файлов и каталогов, находящихся в текущем каталоге, могут быть указаны напрямую, без задания полного имени. Например, если текущим каталогом является `dir4`, то к файлам `file4` и `file5` можно обращаться, используя только их собственные имена `file4` и `file5` вместо указания полного имени.

Относительный путь определяется по отношению к текущему каталогу, при этом имя самого каталога обычно не указывается. Тем не менее в UNIX-системах для указания того, что путь считается от текущего каталога, часто используют мнемоническое обозначение «.». Если текущим каталогом является `dir3`, то относительно него относительный путь к файлу `file6` будет задаваться именем `./dir5/file6`.

Для задания в пути родительского каталога используется мнемоника «..». Например, если текущим каталогом является каталог `dir7`, то относительное имя файла `file1` будет выглядеть как `../../file1`. Первые две точки указывают на каталог, родительский для `dir7` — каталог `dir4`, а вторые две точки — на каталог, родительский для `dir4` — каталог `dir1`.

Системные файлы в ОС семейства UNIX распределены по отдельным каталогам. Это позволяет структурировать различные файлы по их назначению. Большинство UNIX-систем имеет стандартную структуру системных каталогов (рис. 2.7). Так, в UNIX-системах обычно выделяются отдельные каталоги для хранения ядра (`/boot`), системных библиотек (`/lib`), системных утилит (`/bin`), системных настроек (`/etc`), пользовательских программ (`/usr`), пользовательских данных (`/home`). Более по-

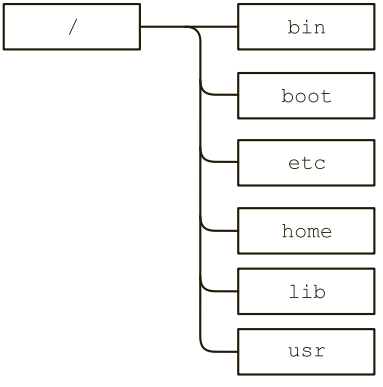

Рис. 2.7. Основные каталоги UNIX-систем

дробно структура стандартных каталогов UNIX-систем рассмотрена в подразд. 6.1.

Для решения нашей прикладной задачи «Контроль знаний» также потребуется определить структуру организации данных. Для этого можно поступить аналогично тому, как распределяются системные файлы в UNIX — распределить данные и программы по отдельным каталогам, а данные, требующие дополнительной структуризации, — по подкаталогам каталога данных.

Пусть каталог, хранящий все данные и программы комплекса «Контроль знаний», будет подкаталогом корневого (рис. 2.8) и имеет имя check.

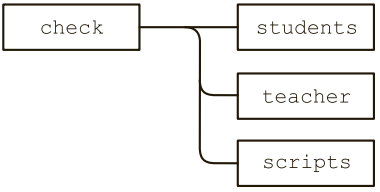

Рис. 2.8. Основные каталоги программного комплекса «Контроль знаний»

Данные приложения будут распределены по двум каталогам — в каталоге check будут содержаться каталоги для хранения рабочих областей студентов (students) и преподавателя (teacher). Задания, предназначенные для выполнения основных действий в системе, будут помещены в каталог scripts. Полные имена каталогов будут иметь следующий вид:

```
/check
/check/students
/check/teacher
/check/scripts
```

Рабочая область каждого студента также будет представлять собой каталог, имя которого будет совпадать с учетным именем студента в системе. В каталоге каждого студента выделяется отдельный подкаталог ready для выполненных работ.

Рабочая область преподавателя будет содержать по одному каталогу для каждой темы, причем темы будут нумероваться, а имена каталогов будут иметь вид theme1, theme2 и т.д. Для собранных у студентов работ в рабочей области преподавателя будет выделен отдельный каталог с именем works.

2.3. Операции над файлами и каталогами

Для доступа к файлам ОС предоставляет набор системных вызовов, реализующих основные функции работы с файлами (табл. 2.1) [11].

Таблица 2.1. **Системные вызовы для работы с файлами**

Системный вызов	Описание
Create	При вызове создается пустой файл, не содержащий данных. Вызов производится перед первой записью в файл
Delete	Удаляет имя файла. Если не существует других имен файлов, связанных с блоком данных, то удаляется сам блок данных и освобождается дисковое пространство
Open	Такой вызов должен быть произведен перед началом работы с любым файлом. Основная цель вызова — считывание параметров файла в оперативную память, выделение части дискового буфера ввода/вывода для файла и назначение файлу временно́го пользовательского идентификатора, использующегося в течение работы программы, открывшей файл
Close	Сбрасывает данные, находящиеся в буфере ввода/вывода в область диска, хранящую набор данных файла, освобождает ресурсы буфера, удаляет временный пользовательский идентификатор
Read	Предназначен для считывания заданного количества байт из файла в заданную область памяти. Чтение начинается с так называемой текущей позиции в файле. После считывания данных текущая позиция перемещается на байт, следующий за считанными данными
Write	Предназначен для записи данных в файл, начиная с текущей позиции. Если текущая позиция совпадает с концом файла, размер набора данных, связанного с файлом, увеличивается. Если текущая позиция находится в середине файла, то данные, содержащиеся в нем, затираются
Append	Вызов представляет собой урезанную форму вызова Write. Единственная его функция — добавление данных в конец файла

Окончание табл. 2.1

Системный вызов	Описание
Seek	Предназначен для перемещения текущей позиции в файле в заданное место. Обычно позиция задается количеством байт с начала файла
Get Attributes	Предназначен для получения атрибутов файла, например, даты его создания. Более подробно об атрибутах см. подразд. 4.4
Set Attributes	Предназначен для установки атрибутов файла
Rename	Предназначен для изменения имени файла. Этот вызов присутствует в ОС не всегда, поскольку практически того же эффекта можно достичь при помощи копирования файла под новым именем и последующего удаления старого файла или при помощи создания жесткой ссылки на набор данных и удаления ненужной более ссылки

Для доступа к каталогам ОС предоставляет набор системных вызовов, реализующих примерно те же функции, что и для работы с файлами (табл. 2.2). Однако существуют значительные отличия, связанные с тем, что содержимое каталогов невозможно прочесть как последовательный текст.

Таблица 2.2. **Системные вызовы для работы с каталогами**

Системный вызов	Описание
Create	Создает пустой каталог, содержащий только элементы «.» и «..»
Delete	Удаляет пустой каталог, не содержащий никаких других файлов и подкаталогов, кроме элементов «.» и «..»
OpenDir	Вызов должен быть произведен перед началом работы с любым каталогом. Основная цель вызова — считывание параметров каталога в оперативную память. В целом вызов аналогичен вызову Open для файла
CloseDir	Высвобождает память, выделенную под параметры каталога, считанные вызовом OpenDir

Окончание табл. 2.2

Системный вызов	Описание
ReadDir	Предназначен для считывания элемента каталога. В отличие от вызова Read для файла единицей информации для этого вызова является структура данных, определяющая свойства файла
Rename	Аналогичен вызову Rename для файла
Link	Создает жесткую ссылку на файл в каталоге
Unlink	Удаляет жесткую ссылку на файл в каталоге. Если удаляется последняя ссылка на файл, то исчезает возможность доступа к набору данных этого файла, а занимаемое набором данных дисковое пространство считается свободным

Для работы с файлами пользователь использует некоторый формальный язык. В зависимости от гибкости языка пользователю либо предоставляется набор синтаксических конструкций языка, соответствующих системным вызовам, указанным выше, либо часть системных вызовов маскируется конструкциями языка.

Одним из типичных способов работы с файлами в UNIX-системах является использование для этой цели командного интерпретатора UNIX и предоставляемого им языка команд. Обзору возможностей командного интерпретатора посвящена следующая глава.

Контрольные вопросы

1. Какие основные функции выполняет файловая система?
2. Каким образом организованы логическая и физическая структуры данных на носителе?
3. Какова структура файловой системы NTFS?
4. Что такое родительский и текущий каталоги?
5. В чем различия абсолютного и относительного имен файла?
6. В чем различия системных вызовов для работы с файлами и каталогами?
7. Может ли относительное имя файла быть длиннее абсолютного?
8. Какой объем дискового пространства будет занимать файл, содержащий строку «Книга»?

Глава 3

ЗАДАНИЯ

3.1. Языки управления заданиями

Командный интерпретатор — программа, работающая в течение всего сеанса работы пользователя с ОС. Ее основная функция — выполнение команд пользователя, записанных на языке данного командного интерпретатора и выполнение этих команд либо непосредственно (встроенными в интерпретатор средствами), либо путем вызова других программ.

Основным способом взаимодействия командного интерпретатора с пользователем является интерфейс командной строки. При таком способе взаимодействия в качестве основного устройства для работы с системой используется *терминал*. Терминал служит для отображения информации и ввода информации пользователем. Физически терминал — это монитор и клавиатура. Логически с точки зрения ОС терминал — набор из двух файлов. Один из этих файлов служит для ввода информации (которая поступает с клавиатуры), другой — для вывода информации (которая выводится на экран).

В некоторых ОС работа с терминалом обеспечивается одним файлом, который используется и для чтения, и для записи. Физическое устройство при этом определяется режимом работы с файлом — записываемые в файл данные отображаются на экране, ввод с клавиатуры наполняет файл данными, которые могут быть прочитаны.

Сигналом для начала работы служит выводимое на экран *приглашение командной строки* или просто *приглашение* — последовательность символов, указывающая на то, что интерпретатор ожидает ввода команды. Типичное приглашение в UNIX-системах имеет следующий вид:

```
$
```

Ввод команды завершается нажатием клавиши Enter, после чего интерпретатор начинает выполнение команды. Например, можно вывести имя файла, соответствующего терминалу. Это осуществляется при помощи команды `tty`:

```
$ tty <Enter>
/dev/console
$
```

Здесь `/dev/console` — имя файла, который используется для вывода данных и считывания ввода пользователя.

Команды пользователя могут подаваться либо в диалоговом режиме с терминала (при помощи интерфейса командной строки), либо в пакетном режиме — без участия пользователя. При работе в пакетном режиме последовательность команд оформляется в виде текстового файла. Последовательность команд определяет *задание*, выполняемое командным интерпретатором. Именно поэтому языки командных интерпретаторов называют также *языками управления заданиями*. Файл, определяющий задание, часто называется *скриптом*.

Язык управления заданиями должен иметь средства:

- определения последовательности выполнения программ в задании и средства определения используемых ресурсов;
- определения типа выполнения программ (основной режим, фоновый режим, определение приоритета и т. п.);
- определения условий выполнения частей задания и ветвления задания;
- проверки состояния ресурсов ОС (файлов и процессов) и их свойств (например, прав доступа).

3.2. Пакетная обработка

Работа с заданиями может выполняться в двух режимах: диалоговом или пакетной обработки.

Работа в *диалоговом режиме* подразумевает постоянный диалог с пользователем во время выполнения задания. В диалоге с пользователем по мере необходимости определяются параметры выполнения команд и программ, входящих в задание.

Если время выполнения задания значительно (часы или дни), то диалоговый режим работы может оказаться неприемлемым, поскольку в течение всего времени выполнения задания потребуется присутствие оператора за терминалом. Один из способов снижения нагрузки на оператора в этом случае — составление задания так, чтобы оно выполнялось в пакетном режиме.

В *пакетном режиме* все данные и параметры, необходимые для выполнения задания, готовятся до начала выполнения задания в виде пакетного файла. Само задание выполняется без участия оператора, а все диагностические сообщения и сообщения об ошибках накапливаются в файлах протокола выполнения. Диагностические сообщения в пакетном режиме также могут выводиться на терминал по мере их появления, но при этом пользова-

телю после завершения выполнения задания будут доступны только последние сообщения, которые были выведены на экран. Если количество диагностических сообщений превышает количество строк терминала, то первые сообщения будут потеряны.

Для передачи заданию данных до выполнения используются *параметры задания*. Если задание обрабатывается командным интерпретатором, предоставляющим интерфейс командной строки, то параметры задания определяются как *параметры командной строки*. Командная строка представляет собой текстовую строку, в которой указано имя вызываемого задания и передаваемые ему параметры. Все параметры отделяются друг от друга символами-разделителями, например пробелами. Параметры — строки, которые могут представлять собой числовые константы, последовательности символов, имена файлов и устройств и т. п. Интерпретация параметров позиционная — параметры различаются по разделителям и поэтому нельзя определить третий параметр, не определив первый и второй. Передаваемые параметры, вообще говоря, не должны содержать символов-разделителей и различных управляющих кодов. Однако возможно использование пробела в тексте параметра, для этого его необходимо заключить в двойные кавычки.

Максимальное количество параметров, которое можно передать команде, определяется максимальной длиной команды, поддерживаемой системой. В разных системах это число варьируется. Согласно стандарту POSIX данное значение не должно быть меньше 4 096. Самый простой способ узнать ограничение на длину строки — это выполнить команду

```
xargs --show-limits
```

Эта команда позволит узнать ограничения, накладываемые вашей системой. Так, например, в программном комплексе «Контроль знаний» присутствует задание give.sh, предназначенное для выдачи определенному студенту варианта контрольной работы по заданной теме. Типичная командная строка для вызова этого задания будет выглядеть следующим образом:

```
give.sh 1 5 vasya
```

Здесь give.sh — имя задания; 1 — первый параметр задания, определяющий номер темы; 5 — второй параметр задания, определяющий номер варианта; vasya — третий параметр, определяющий имя студента, которому выдается задание.

Результат выполнения задания заключается в изменении информационного окружения задания — содержимого и состояния

файлов и каталогов, запуске или останове других заданий и программ пользователя. Отчет о ходе работы и результате задания, как уже говорилось выше, может быть выведен на экран или в файл.

Для облегчения автоматической обработки результата при завершении задания формируется *код возврата* — числовое значение, характеризующее успешность выполнения. Конкретные значения кодов возврата определяются в тексте задания. Доступ к коду возврата можно получить непосредственно после завершения задания при помощи специальных средств командного интерпретатора.

Так, например, в программный комплекс «Контроль знаний» входит задание look.sh, предназначенное для просмотра количества вариантов по заданной теме, которые находятся в рабочей области преподавателя. Для запуска этого задания ему передается номер темы:

```
look.sh 3
```

В результате своего выполнения задание формирует числовой код возврата, равный количеству вариантов по заданной теме, и возвращает его при своем завершении. Этот код возврата может быть считан пользователем или другим заданием.

В настоящее время диалоговый режим более привычен для конечного пользователя — например, большинство программ, работающих под управлением ОС Windows, работает именно в диалоговом режиме. Тем не менее пакетный режим выполнения может быть удобен для автоматизации многих рутинных операций. Поэтому практически во всех ОС существуют командные интерпретаторы или аналогичные им программные средства, позволяющие выполнять задания в пакетном режиме.

3.3. Задания в среде UNIX

3.3.1. Командный интерпретатор BASH

Синтаксис языка управления заданиями определяется используемым командным интерпретатором. В настоящем издании в качестве базового используется командный интерпретатор BASH (Bourne-Again Shell). При работе в диалоговом режиме команды BASH вводятся с клавиатуры в ответ на приглашение командной строки.

Задания, оформляемые в виде файлов, состоят из двух частей — заголовка, определяющего имя командного интерпретатора и путь к нему, и собственно текста задания.

Заголовок начинается с первого символа первой строки файла задания и для интерпретатора BASH выглядит обычно следующим образом:

```
#!/bin/bash
```

Здесь # — символ комментария. Все символы, которые находятся на строке после символа # и не воспринимаются интерпретатором как команды, а также все, что находится после этого символа, игнорируется при выполнении задания.

Заголовок является комментарием особого вида за счет того, что сразу за символом # помещен символ !. Конструкция #!, будучи помещенной в начало файла задания, указывает на то, что после нее записывается полное имя исполняемого файла командного интерпретатора.

После заголовка следует основная часть скрипта — последовательность команд. Одна строка скрипта может содержать одну или более команд, комментарии или быть пустой. Как правило, одна строка скрипта содержит одну команду, при большем количестве команд на одной строке они разделяются точкой с запятой.

Синтаксически вызов большинства команд интерпретатора BASH состоит из двух частей:

```
<имя команды> <параметры>
```

В качестве имени команды может выступать либо внутренняя команда BASH, либо имя файла, содержащего код программы или текст задания.

Краткий справочник по наиболее часто применяемым командам интерпретатора BASH и внешним программам приведен в приложении 3.

Если команда и ее параметры слишком длинны, то можно воспользоваться символом переноса «\». После указания этого символа можно продолжить запись команды на следующей строке, а командный интерпретатор будет воспринимать все, что записано после символа переноса, как продолжение команды.

Для просмотра документации по командам BASH можно воспользоваться встроенной системой помощи `man`. Для вызова страницы помощи BASH необходимо ввести `man bash`. Описание встроенных команд BASH находится на этой странице в разделе BUILT-IN COMMANDS.

Для просмотра страницы помощи по любой команде UNIX необходимо вызвать `man` с параметром — именем команды. Для поиска названий команд и получения краткого их описания мож-

но воспользоваться программой apropos. Так, например, вызов apropos edit выведет на экран список всех страниц помощи по программам, в названии которых встречается подстрока edit.

3.3.2. Переменные

Типы переменных. При написании заданий существует возможность определения и использования переменных (аналогично тому, как это делается при программировании на языках высокого уровня). Существует два типа переменных: внутренние переменные и переменные окружения — т.е. переменные, заданные в специальной области ОС и доступные всем выполняемым программам. Переменные окружения — часть того информационного окружения, которое влияет на выполнение программ пользователя.

Работа со значениями переменных. Присвоение значений переменным осуществляется при помощи конструкций:

```
<имя переменной>=<значение>
```

или

```
let <имя переменной>=<значение>
```

При присвоении значения нужно обратить внимание на то, что пробелы между именем переменной и знаком равенства, а также между знаком равенства и значением не допускаются.

Чтобы при выполнении задания можно было получить доступ к значению переменной, необходимо воспользоваться операцией подстановки $. При прямом указании в тексте задания имени переменной это имя будет воспринято как строка. Если имя предварить операцией подстановки, то при выполнении задания будет произведена подстановка значения переменной с указанным после символа $ именем. Поясним это на примере:

```
#!/bin/bash
variable=Hello
echo variable
echo $variable
```

После выполнения такого задания на экран будет выведено

```
variable
Hello
```

Учитывая все сказанное выше, нетрудно догадаться, что присвоение значения одной переменной другой переменной будет записано как

```
var1=$var2
```

Системные переменные. Кроме переменных, определяемых пользователем, существуют также встроенные системные переменные командного интерпретатора BASH. Эти переменные имеют стандартные имена и фиксированную трактовку, при этом их значения присваиваются самим командным интерпретатором, пользователь может только считать значения встроенных переменных, но не может явно изменить их значение.

В BASH определены следующие встроенные переменные:

1) `$?` — код возврата последней команды. Возврат из скрипта производится командой `exit` с указанием кода возврата (`exit <код возврата>`). Код возврата может служить для управления выполнением задания, например, при последовательном поиске строк в файле. В зависимости от кода возврата программы поиска подстроки в файле `grep` можно либо выдать сообщение об ошибке, если нужная строка не найдена, либо продолжить поиск следующей строки;

2) `$#` — число параметров командной строки вызова задания;

3) `$1,…,$9` — при помощи этих переменных производится подстановка значений параметров, указанных в командной строке, при помощи которой было вызвано задание. Переменной `$1` соответствует первый параметр, переменной `$9` — девятый.

Если необходим доступ к параметрам, следующим за девятым, используется команда `shift`, сдвигающая «окно» из девяти параметров вправо по списку параметров (рис. 3.1). После выполнения команды `shift` второй параметр становится доступным через переменную `$1`, а десятый — через `$9`. Количество выполнений команды `shift` не ограничено, хотя нужно учитывать, что в момент, когда последний параметр командной строки может быть получен подстановкой значения переменной `$1`, при подстановке значений переменных `$2,…,$9` будут получены только пустые строки. Обратного сдвига параметров не предусмотрено.

Часто бывает необходимо записать в задании строку, состоящую из имени переменной и фиксированной части, например, строку вида `$ABC`, в которой `$A` — переменная, а `BC` — фиксированная текстовая строка. При такой форме записи BASH не сможет различить конец имени переменной и начало текстовой строки, поскольку неизвестно, имелась в виду переменная `A` и строка `BC`, переменная `AB` и строка `C` или переменная `ABC`.

Рис. 3.1. Сдвиг окна параметров

Поэтому невозможно обращение более, чем к девяти параметрам командной строки при помощи системных переменных $1,…,$9: при записи $19 неизвестно — имелся в виду 19-й параметр командной строки или первый параметр, за которым следует текстовая строка 9.

Для решения этой проблемы в новых версиях BASH (2.0 и выше) добавлены символы выделения имени переменной — фигурные скобки. Для доступа к переменной используется синтаксис ${имя переменной}.

При помощи символов выделения имени переменной возможен доступ более, чем к девяти параметрам: для этого в качестве имени переменной указывается номер параметра в виде ${n}, где n — любое целое число. При доступе к параметрам при помощи ${n} также учитывается действие команды shift;

4) $* — переменная, в которой хранятся все параметры командной строки, переданные заданию. Все параметры в этой переменной представляют собой единую строку, заключенную в кавычки, т.е. $* = "$1 $2 $3 …";

5) $@ — переменная, в которой хранятся все параметры командной строки, переданные заданию. Каждый переданный параметр в этой переменной заключен в кавычки, т.е. $@ = "$1" "$2" "$3" …;

6) $0 — переменная, в которой хранится имя файла выполняющегося задания. При помощи этой переменной можно организовывать рекурсивный вызов задания, не зависящий от имени его файла, т.е. вызов

```
exec $0
```

всегда рекурсивно вызовет то же самое задание вне зависимости от имени его файла.

В качестве примера использования системных переменных приведем следующее задание на языке BASH:

```
#!/bin/bash
while [ "$1" != "" ]; do
  echo $@
  shift
done
```

Задание в цикле выводит при помощи команды echo все переданные ему параметры. После каждого вывода окно параметров сдвигается при помощи команды shift. Выполнение задания завершается при исчерпании списка параметров, т.е. в случае, когда после последнего выполнения команды shift первый параметр становится равным пустой строке (подробнее об использованных конструкциях и командах см. подразд. 3.3.4, 3.3.6, а также приложение 3).

В результате работы задания на экран будет выведено следующее:

```
$ ./test.sh 1 2 3 4 5
1 2 3 4 5
2 3 4 5
3 4 5
4 5
5
```

Таким образом, на каждой новой итерации цикла значение системной переменной `$@` меняется в соответствии с текущим расположением окна параметров и общее число доступных параметров постепенно уменьшается.

Копирование переменных задания в среду. Переменные, которым в ходе выполнения задания при помощи операции = было присвоено значение, доступны только внутри этого задания и только на время его выполнения. Такие переменные могут рассматриваться как локальные, изолированные от внешней среды выполнения задания.

Чтобы сделать переменные, инициализированные некоторым заданием, доступными другим заданиям, которые будут выполняться в среде того же самого командного интерпретатора, можно воспользоваться командой копирования переменных в среду текущего задания.

Копирование переменных в среду задания выполняется командой `export`. Существуют два формата ее вызова:

`export <имя переменной>` — перемещает уже инициализированную переменную в среду выполнения задания;

`export <имя переменной>=<значение>` — присваивает переменной значение и перемещает ее в область переменных окружения.

Совокупность переменных, объявленных в среде выполнения задания, обычно называется *набором переменных окружения* этого задания. Для просмотра всех объявленных переменных окружения служит команда `set`:

```
$ set
PATH=/bin:/sbin:/usr/sbin
PWD=/home/nick
TTY=/dev/tty6
```

При запуске новой копии командного интерпретатора (например, при помощи явного вызова исполняемого файла `bash`) новый командный интерпретатор наследует все определенные переменные окружения. При этом командный интерпретатор получает новую копию тех же самых переменных — любое изме-

нение значения переменных в среде командного интерпретатора затрагивает только его собственную среду выполнения, но не изменяет среду выполнения вызывающего командного интерпретатора.

Так, например, в следующем диалоге с пользователем определяется переменная окружения MYVAR со значением A, после чего запускается новая копия командного интерпретатора, в которой значение переменной MYVAR наследуется, а затем переопределяется. После завершения нового командного интерпретатора и возврата к предыдущему значение переменной MYVAR восстанавливается. Поскольку в следующем примере рассматривается диалоговый режим работы с пользователем, полужирным шрифтом выделяется вывод системы, подчеркнутым полужирным — ввод пользователя, курсивом в скобках отмечены комментарии к ходу выполнения:

```
$ export MYVAR=A
$ echo $MYVAR
A
$ bash (запуск новой копии командного интерпретатора)
$ echo $MYVAR
A (значение MYVAR унаследовано)
$ export $MYVAR=B
$ echo $MYVAR
B (значение MYVAR переопределено в среде текущего интерпретатора)
$ exit (выход в предыдущий интерпретатор)
$ echo MYVAR
A (значение восстановлено)
```

Некоторые переменные окружения объявляются в среде выполнения основного командного интерпретатора, запускаемого сразу после входа пользователя в систему (см. подразд. 4.1), что позволяет использовать значение этой переменной в любой копии командного интерпретатора, запущенного в ходе сеанса работы пользователя.

Примером такой переменной окружения может служить переменная PATH. Значением этой переменной является список абсолютных имен каталогов, в которых выполняется поиск исполняемых файлов (заданий и программ), если при их запуске было указано только имя файла без указания относительного или абсолютного пути. Порядок задания каталогов определяет *стратегию поиска*, т. е. последовательность просмотра каталогов командным интерпретатором. Последовательность просмотра важна в том случае, если на диске существует несколько исполняемых файлов

с одинаковыми именами — если не будет явно задан его каталог, первым будет запущен исполняемый файл, чей каталог находится ближе к началу списка в переменной PATH.

Пользователь может иметь свой собственный каталог с исполняемыми файлами и добавить этот каталог к списку:

```
export PATH=$PATH:/check/scripts
```

Так, в приведенном примере переменной PATH присваивается ее старое значение, конкатенированное с разделителем «:» и новым именем каталога с исполняемыми файлами /check/scripts. При этом новое переопределенное значение переменной PATH будет находиться уже в среде его командного интерпретатора и будет доступно только в течение его сеанса работы с системой. Все остальные пользователи будут видеть свое значение переменной, определенное в средах их командных интерпретаторов.

Поскольку значения переменных окружения потенциально могут влиять на работу заданий пользователя, например, определять имена каталогов, в которых хранятся файлы пользователя, они будут входить в информационную среду сеанса пользователя. Если какие-либо переменные при этом влияют на выполнение задания — они будут входить в информационное окружение этого задания.

Например, большая часть заданий, входящих в состав системы «Контроль знаний», использует значение переменной BASEDIR в качестве имени каталога, содержащего файлы системы. Если эта переменная не определена, то задание завершает свое выполнение:

```
if [ "${BASEDIR:-DUMMY}" == "DUMMY" ] ; then
  echo "Переменная \$BASEDIR не задана"
  exit 100
fi
```

Доступ к значениям переменных. При обращении к переменной, значение которой не определено, при выполнении задания будет возникать ошибка. Это связано с тем, что в BASH существует разница между отсутствием значения и пустой строкой. Чтобы избежать появления ошибки, существуют различные способы доступа к значению переменной:

1) $var — получение значения переменной var или пустого значения, если она не инициализирована;

2) ${var} — аналогично $var;

3) ${var:-string} — получение значения переменной var, если она определена, или строки string, если переменная var не определена;

4) ${var:=string} — получение значения переменной var, если она определена, или строки string, если переменная var не определена. При этом если значение переменной var не определено, то ей также присваивается значение string;

5) ${var:?string} — получение значения переменной var, если она определена. Если значение переменной var не определено, то выводится строка string и выполнение задания прекращается;

6) ${var:+string} — если переменная var определена, то возвращается значение string, или пустое значение, если переменная var не определена [8].

Использование подстановки некоторого значения string вместо значения переменной может быть удобно, например, в следующем случае — предположим, что задание выводит на экран файл при помощи команды cat. Имя выводимого файла содержится в некоторой переменной окружения. Если переменная окружения не задана, подставляется заранее заданное имя файла:

```
cat ${FILENAME:-/home/sergey/default.txt}
```

Так, если не определена переменная FILENAME, команда cat выведет на экран файл /home/sergey/default.txt.

3.3.3. Запуск задания на исполнение

Чтобы запустить задание на исполнение, его нужно передать командному интерпретатору, который будет анализировать текст задания и трансформировать команды задания в системные вызовы ОС. При этом командный интерпретатор будет поддерживать информационное окружение задания, т. е. хранить все временные данные, необходимые для выполнения задания, например, привязки к текущему терминалу, локальные переменные, файловые дескрипторы. Задание может быть передано на исполнение текущему командному интерпретатору — в этом случае сохраняется текущее информационное окружение (рис. 3.2).

Также возможен запуск средствами текущего командного интерпретатора нового процесса командного интерпретатора (рис. 3.3). Управление при этом передается новому процессу командного интерпретатора, который будет исполнять переданное ему задание. После завершения выполнения задания новый процесс командного интерпретатора будет завершен и передаст управление обратно вызвавшему его командному интерпретатору. Информационное окружение нового командного интерпретатора будет новым, но оно также унаследует некоторые свойства информационного окружения предыдущего интерпретатора, например, значения переменных, содержимое изменяемых заданием файлов и т. п.

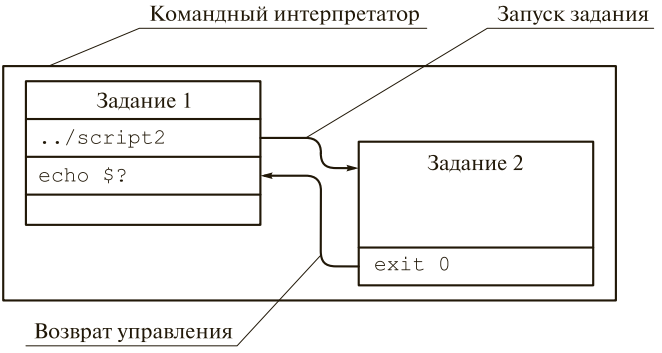

Рис. 3.2. Запуск задания в контексте текущего командного интерпретатора

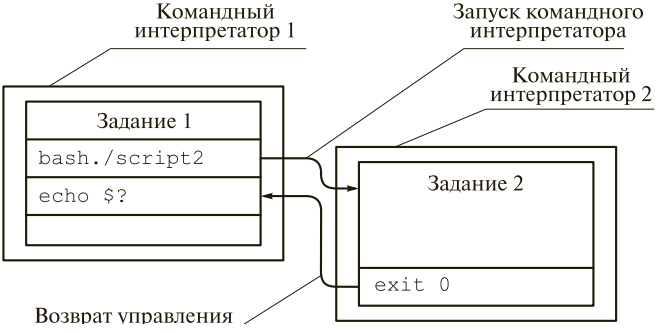

Рис. 3.3. Запуск задания в новом командном интерпретаторе

Задание может быть запущено на исполнение как непосредственно пользователем из командной строки, так и из другого задания. Во втором случае имя запускаемого задания представляет собой команду в тексте запускающего задания. Условимся при этом называть запускающее задание *родительским* по отношению к запускаемому — *дочернему*. Аналогично в случае создания нового командного интерпретатора: создающий интерпретатор будем называть *родительским* по отношению к порождаемому — *дочернему*.

Возможны следующие варианты запуска задания:

• путем указания имени файла задания (возможно с полным или относительным путем к файлу):

```
/check/scripts/teacher/gather.sh
```

Чтобы воспользоваться этим методом запуска, файл с заданием должен иметь атрибут «исполнимый» (см. подразд. 4.4). При таком запуске задания на исполнение запускается новая копия

командного интерпретатора, путь к которому указан в заголовке задания. После завершения задания управление возвращается родительскому командному интерпретатору. В случае если задание было запущено из другого задания, родительское задание продолжает свое выполнение со следующей команды. Если задание было запущено пользователем в командной строке, происходит возврат к приглашению командной строки родительского интерпретатора.

Если в строке запускаемого таким образом задания не указан полный или относительный путь к файлу задания, то поиск файла задания происходит в каталогах, перечисленных в переменной окружения `PATH`, как при запуске обычного исполняемого файла;

• путем запуска командного интерпретатора с указанием имени файла задания в качестве параметра:

```
/bin/bash /check/scripts/teacher/gather.sh
```

При таком запуске файл с заданием необязательно должен иметь атрибут «исполнимый». Также допускается отсутствие заголовка в тексте задания — выбор командного интерпретатора осуществляется пользователем, запускающим задание на выполнение, поэтому при такой форме запуска заголовок игнорируется.

В отличие от предыдущего метода запуска при отсутствии абсолютного или относительного пути к файлу задания его поиск осуществляется в текущем каталоге. Это связано с тем, что в данном случае имя файла передается в качестве параметра командному интерпретатору, а не используется как имя исполняемого файла. В остальном метод запуска аналогичен предыдущему;

• путем запуска при помощи команды `exec`:

```
exec /check/scripts/teacher/gather.sh
```

При данном запуске управление передается дочернему командному интерпретатору безвозвратно — родительский командный интерпретатор полностью заменяется дочерним, который выполняет запущенное задание. Возврата к выполнению родительского задания после завершения дочернего не происходит. Файл задания при такой форме запуска должен иметь атрибут «исполнимый»;

• путем запуска в том же командном интерпретаторе:

```
. /check/scripts/env.sh
```

Если перед именем запускаемого задания указать точку, отделенную от имени пробелом, то выполнение задания продолжится уже запущенной копией командного интерпретатора, при этом для нового задания будет использоваться та же самая среда времени выполнения.

При таком запуске новое задание получит в свое распоряжение те же самые переменные, что и вызвавшее его задание — при

сохранении копии командного интерпретатора сохраняется и его внутреннее состояние, в том числе и переменные. Все переменные, объявленные и инициализированные в вызванном задании, будут иметь те же значения и после его завершения, когда управление будет передано вызвавшему заданию.

3.3.4. Ввод/вывод. Конвейерная обработка

При выполнении задания данные, выводимые на терминал и получаемые с терминала, поступают туда не напрямую. Для передачи и получения данных используются *буферы ввода/вывода* — некоторые промежуточные области оперативной памяти, в которых ОС накапливает выводимые данные перед непосредственной передачей их на терминал или после получения с терминала. Управление буферами ввода/вывода происходит независимо от пользователя подсистемой ввода/вывода, входящей в ядро ОС. Доступ к буферам ввода/вывода происходит практически аналогично доступу к файлам: по умолчанию система ввода/вывода предоставляет прозрачный доступ к трем виртуальным файлам, два из которых служат для вывода данных на экран, а один — для получения данных с терминала. Эти файлы получили название *потоков ввода/вывода*:

`stdout` — поток вывода, по умолчанию — экран терминала;

`stderr` — поток вывода ошибок, по умолчанию — экран терминала;

`stdin` — поток ввода, по умолчанию — клавиатура.

Данные, поступающие в потоки ввода/вывода, обычно направляются в файл, соответствующий устройству терминала в каталоге `/dev`. Ввод/вывод в потоки ввода/вывода может быть перенаправлен в файлы, что позволяет читать из файлов пользовательские данные и/или записывать их в файлы. При этом данные, которые в обычной ситуации выводятся на экран, перенаправляются в указанный файл, а данные, вводимые обычно с клавиатуры — считываются из файла. При включенном перенаправлении вывод данных на экран или ввод их с клавиатуры не происходит, т.е. файлы полностью заменяют устройства терминала. Управление перенаправлением данных производится при помощи команд перенаправления ввода/вывода.

Чтобы перенаправить выводимые программой (в том числе и заданием) данные в файл `file.txt`, необходимо запускать программу с указанием символа перенаправления вывода >:

```
prog > file.txt
```

Если файла `file.txt` не существует, то при таком перенаправлении он будет создан и в него будут записаны все данные, кото-

рые предназначены для вывода на терминал. Если файл `file.txt` уже существует, то все данные, которые находились в нем до запуска перенаправления вывода, будут затерты новыми данными.

При накоплении в одном файле данных, получаемых от нескольких программ, необходимо добавлять данные в конец к уже имеющимся в файле. Для этого можно воспользоваться символом перенаправления вывода с добавлением символа >>:

```
prog >> file.txt
```

При таком вызове данные, выводимые программой, будут добавлены в конец файла `file.txt`. Если файла в момент вызова программы не существует, то он будет создан перед выводом в него данных.

Для перенаправления потока ввода необходимо воспользоваться символом перенаправления ввода <. Для перенаправления ввода из файла `file.txt` вызов имеет вид

```
prog < file.txt
```

Если требуется одновременное перенаправление ввода из файла `infile.txt` и вывода в файл `outfile.txt`, то вызов будет выглядеть следующим образом:

```
prog < infile.txt > outfile.txt
```

Для перенаправления потока вывода одной программы в поток ввода другой можно воспользоваться *конвейером ввода/вывода*. Установка конвейера производится символом |. Так, для перенаправления вывода программы `prog1` на вход программы `prog2` вызов будет иметь следующий вид:

```
prog1 | prog2
```

Конвейер ввода/вывода является типичным примером использования одного из основных принципов, положенных в основу UNIX-систем — принципа декомпозиции. Любая сложная задача может быть разбита на несколько последовательных этапов, на каждом из которых решается своя специфическая подзадача. Применение такого подхода оправдывает себя, поскольку одни и те же действия могут быть применены в совершенно разных заданиях. Например, рассмотрим два задания — одно, которое будет просматривать список полученных вариантов заданий и выводить его в алфавитном порядке, и второе, которое будет просматривать список вариантов по заданной теме и выводить его в алфавитном порядке.

Будучи реализованными в виде монолитной программы, эти два задания дублируют функциональность друг друга. Если же применить всего два действия — вывод текстового файла в поток

вывода и сортировку входного потока с последующей выдачей его в выходной, получим две простые задачи, которые могут быть декомпозированы на отдельные команды.

Первое задание при этом приобретет вид

```
ls /check/students/vasya | sort
```

а второе —

```
ls /check/teacher/theme1 | sort
```

Здесь команда `ls` выводит список файлов в заданном каталоге в поток вывода, а команда `sort` сортирует этот поток и выводит отсортированный результат в поток вывода по умолчанию.

3.3.5. Подстановка

Подстановка вывода программ. При написании заданий часто необходимо сохранить данные, выводимые какой-либо программой. Чтобы в дальнейшем можно было сравнительно просто использовать эти данные в ходе выполнения задания, можно сохранить их в переменной задания. Для этого применяют выражения `$( )` и `` ` ` `` (обратные кавычки). Будучи вставленными в текст задания, при его выполнении они заменяются на данные, выводимые командой.

Форма записи `` `команда` `` поддерживается всеми версиями командного интерпретатора BASH. Так, например, команда

```
var=`ls /check`
```

присвоит переменной `var` значение, соответствующее списку всех файлов, содержащихся в каталоге `/check`.

Новая форма записи `$(команда)` поддерживается только версиями BASH 2.0 и старше и позволяет создавать вложенные друг в друга подстановки.

Так, например, команда

```
var=$(ls /$(ls /check))
```

присвоит переменной `var` значение, соответствующее списку всех файлов, находящихся в подкаталогах, непосредственно содержащихся в каталоге `/check`.

Групповые символы. Задача получения списка всех файлов в каталоге, рассмотренная выше, может быть гораздо сложнее. Пользователю может понадобиться получить список не всех файлов в каталоге, а только файлов, имена которых удовлетворяют определенному критерию — например, начинающиеся с буквы A или содержащие не более восьми символов. Для определения

такого критерия используется *маска имени файла* или просто *маска*. Маска — текстовая строка, на которую накладываются почти те же самые ограничения, что и на имена файлов. Единственное отличие маски состоит в том, что в ее состав могут входить *символы подстановки*, использование которых в именах файлов запрещено. При проверке соответствия имени файла маске символы подстановки заменяют собой один или несколько символов имени.

Наиболее часто используют символы подстановки `*` и `?`. Символ подстановки `*` означает, что вместо него в имени файла может стоять любое количество символов. Так, маске `text*.doc` будут удовлетворять имена файлов `text1.doc`, `text123.doc` и даже `text.doc`.

Символ подстановки `?` означает, что вместо него в имени файла может стоять один символ или ни одного. Так, маске `text?.doc` будут удовлетворять имена файлов `text1.doc`, `text.doc`, но не будет удовлетворять имя `text12.doc`.

Для получения списка файлов, удовлетворяющих маске, маска может быть указана в качестве параметра команды `ls`. Например, команда `ls*~` выведет все файлы текущего каталога, имена которых оканчиваются на тильду (обычно таким образом именуются файлы, содержащие устаревшие данные).

3.3.6. Управление ходом выполнения задания

Последовательности выполнения команд. Самые простые задания заключаются в последовательном выполнении команд пользователя. При этом результат выполнения предыдущих команд никак не влияет на выполнение последующих, также никак не учитывается, как выполняются команды. В простейших заданиях, рассмотренных выше, выполнение команд идет последовательно.

При необходимости определения того, как выполняются команды — последовательно или параллельно — необходимо использовать символы-разделители `;` и `&`.

Параллельное выполнение команд. Для последовательного выполнения нескольких команд они разделяются символом `;`. В результате выполнение очередной команды начинается только после завершения выполнения предыдущей. Если вместо разделителя `;` между командами указывается разделитель `&`, то каждая команда, стоящая до разделителя, выполняется в фоновом режиме, а следующая команда выполняется немедленно после запуска предыдущей. Таким образом, возможен параллельный запуск и выполнение двух или более команд.

Синтаксис выражения для последовательного выполнения двух команд определяется следующим образом:

```
команда1 ; команда2
```

Аналогично для параллельного выполнения:

```
команда1 & команда2
```

Символ-разделитель `;` обычно применяется только в тех случаях, когда требуется разместить несколько команд на одной строке. Того же эффекта можно добиться, если заменить разделители `;` на символы конца строки.

Если при использовании разделителя `&` указать только первую команду (команда `&`), то она будет выполнена в фоновом режиме, а выполнение задания будет продолжено с команды, находящейся на следующей строке. Если запуск команды в фоновом режиме был осуществлен из командной строки, то команда начнет свое выполнение в фоновом режиме, а управление будет сразу передано командному интерпретатору.

Условное выполнение команд. Для выполнения последовательности команд, в которой запуск на исполнение каждой команды зависит от результата выполнения (кода возврата) предыдущих команд, используются разделители `&&` и `||`.

Чтобы выполнить `команду1` и, если она выполнилась удачно, выполнить `команду2`, используется следующая запись:

```
команда1 && команда2
```

Для того, чтобы выполнить `команду1` и, если она выполнилась неудачно, выполнить `команду2`, используется похожая запись:

```
команда1 || команда2
```

Успешность выполнения команды определяется по ее коду возврата. При этом код возврата, равный нулю, означает успешное выполнение команды, код возврата, отличный от нуля — неуспешное.

Командная строка, содержащая разделители `&&` и `||`, может рассматриваться как логическое выражение, значение которого вычисляется командным интерпретатором по мере выполнения команд, а разделители `&&` и `||` — как операции логического умножения и логического сложения соответственно. В качестве аргументов логических функций выступают коды возврата команд, нулевой код возврата соответствует истинному значению, отличный от нуля — ложному. При такой интерпретации разделителей правила их работы могут быть описаны законами логики — если первая из пары команд, разделенных операцией `&&`, завершается с ненулевым кодом возврата (ложное значение), то все значение

логической функции заведомо будет ложным и выполнение второй команды не требуется. Аналогично, если первая из пары команд, разделенных операцией `||`, завершается с нулевым кодом возврата (истинное значение), то все значение логической функции заведомо истинно и выполнение второй команды также не требуется.

Интерпретация операций `&&` и `||` производится слева направо, при этом они имеют одинаковый приоритет. Для изменения приоритета используются круглые скобки. Таким образом, выражение `command1 || command2 && command3` будет соответствовать выражению `(command1 || command2) && command3`, но не `command1 || (command2 && command3)`.

Объединение потоков вывода программ. С точки зрения условного выполнения команд фигурные скобки `{}` являются символами группировки. Однако они могут быть использованы для объединения потоков вывода нескольких программ в один. Чтобы перенаправить ввод/вывод для нескольких команд, вместо

```
команда1 > файл; команда2 >> файл
```

можно написать

```
{команда1 ; команда2;} > файл
```

При этом последовательность команд, заключенная в скобки, будет рассматриваться командным интерпретатором как единая команда, данные от которой поступают в поток вывода.

Области видимости переменных задания. Область применения круглых скобок `()` также шире, чем просто группировка команд. Если последовательность команд поместить в круглые скобки, то после выполнения последовательности команд восстанавливаются значения измененных командами переменных окружения.

Например, следующее задание:

```
var="global"; (var="local"; echo "var is $var"); \
echo "var is $var"
```

выведет

```
var is local
var is global
```

Здесь переменной `var` сначала присваивается значение `global`, потом оно изменяется на `local`. Однако поскольку команда, изменяющая значение переменной `var`, заключена в круглые скобки, то после ее выполнения значение переменной восстанавливается. Таким образом, при помощи круглых скобок можно управлять областями видимости на более тонком уровне — не

только различать локальные переменные задания и глобальные переменные окружения, но и определять области видимости переменных внутри одного задания.

Условные операторы и операторы цикла. Рассмотренных выше способов управления ходом выполнения задания явно недостаточно для написания некоторых заданий, например, таких, в которых стоят задачи отбора файлов по заданным критериям для последующей обработки или организации циклов.

Для отбора объектов в языках управления заданиями используется понятие условного выражения. *Условное выражение* — выражение, определяющее эталон, при сравнении с которым выполняется отбор, объект, для которого проверяется его соответствие эталону и степень соответствия объекта эталону. Например, в выражении $K > 2$ переменная K будет объектом, для которого проверяется соответствие, операция сравнения > будет задавать степень соответствия эталону, а константа 2 — сам эталон. Результатом проверки условного выражения всегда является либо логическая истина — если объект соответствует эталону с заданной степенью точности, или логическая ложь, если объект эталону не соответствует.

Для проверки условных выражений в языке BASH применяется команда `test`. Возможны два формата ее вызова:

```
test <выражение>
```

или

```
[ <выражение> ]
```

В обоих форматах команда вычисляет значение логического выражения, указанного в качестве параметра, и возвращает код возврата 0, если выражение истинно, и 1 — если ложно. Следует обратить внимание на пробелы между выражением и квадратными скобками во втором случае — они обязательны и пропуск пробелов вызывает множество проблем.

Объектами, свойства которых проверяются в выражениях, могут быть файлы, строки и числа. Далее в тексте по мере изложения материала будут приведены дополнительные виды проверок, выполняемых при помощи команды `test`. Формат некоторых выражений для проверки свойств файлов, строк и чисел приведен ниже:

`-z <строка>`	— строка имеет нулевую длину (строка пуста)
`-n <строка>`	— длина строки больше 0 (строка не пуста)
`"строка1" == "строка2"`	— две строки равны друг другу

`"строка1" != "строка2"`	— две строки не равны друг другу
`число1 -eq число2`	— числа равны
`число1 -ne число2`	— числа не равны
`число1 -lt число2`	— число 1 меньше числа 2
`число1 -le число2`	— число 1 меньше или равно числу 2
`число1 -gt число2`	— число 1 больше числа 2
`число1 -ge число2`	— число 1 больше или равно числу 2
`-s <файл>`	— размер файла более 0 (файл не пуст)
`-f <файл>`	— файл существует и является обычным файлом
`-d <файл>`	— файл существует и является каталогом

Перед любым выражением может быть поставлен символ логического отрицания `!`:

`! <выражение>`	— все выражение ложно, когда истинно <выражение>. Все выражение истинно, когда ложно <выражение>

Выражения можно объединять при помощи операций логического И и логического ИЛИ следующим образом:

`<выражение1> -a <выражение2>`	— все выражение истинно, когда истинно <выражение1> И <выражение2>
`<выражение1> -o <выражение2>`	— все выражение истинно, когда истинно <выражение1> ИЛИ <выражение2>

Приведенные выше выражения могут быть использованы при проверке условий в командах ветвления или при задании условия останова в циклах. Например, синтаксис блока ветвления по условию `if` определен следующим образом:

```
if <логическое выражение-1> ; then <команды-1>
elif <логическое выражение-2> ; then <команды-2>
else
  <команды-3>
fi
```

Здесь блок команд <команды-1> будет выполнен при истинном значении выражения <логическое выражение-1>, блок команд <команды-2> будет выполнен при истинном значении выражения <логическое выражение-2>. При этом допускается произвольно большое количество проверок при помощи выражения elif, соответствующего конструкции Else If структурных языков программирования. Если все проверяемые логические выражения ложны, то выполняется блок <команды-3>, следующий после ключевого слова else. Завершается блок ключевым словом fi.

Символ ; используется в последнем фрагменте как разделитель команд, поскольку с точки зрения синтаксиса BASH if и then — это разные команды, которые необходимо разделять либо символом ;, либо размещать их в разных строках файла задания.

В качестве логического выражения для команды ветвления может выступать любая команда, а значение проверяется по ее коду возврата. Например, здесь часто используется рассмотренная выше команда test.

Например, приведенный ниже фрагмент задания проверяет, пуст ли первый параметр командной строки, переданной в задание. В случае, если первый параметр — пустая строка, выводится сообщение о том, что заданию не передано ни одного параметра командной строки, и выполнение задания завершается с кодом возврата 1. Более аккуратной формой такой проверки является проверка количества переданных параметров $# -eq 0, но исходя из того, что передача параметров в задание — позиционная, проверка пустоты первого параметра допустима. Однако такая проверка сработает и при указании пустой строки в качестве первого параметра (при помощи экранирующих символов — двойных кавычек):

```
test.sh "" A B
```

Вторая проверка оперирует с количеством переданных параметров — если их больше трех, то выводится соответствующее сообщение и выполнение задания завершается с кодом возврата 2:

```
if  [ -z $1 ] ; then
  echo "No command line parameters are specified"
  exit 1
elif [ $# -gt 3 ]; then
  echo "Too many parameters are specified"
  exit 2
fi
```

Логические условия могут быть использованы и в качестве ограничителей циклов. Так, в конструкции while … do … done,

синтаксис которой приведен ниже, блок операторов <операторы> выполняется, пока логическое выражение истинно (равно 0):

```
while <логическое выражение> ; do
  <операторы>
done
```

Следующий фрагмент кода выводит все параметры, переданные заданию. При этом используется команда shift, рассмотренная ранее. Выполнение цикла продолжается до тех пор, пока значение первого параметра не пусто. При этом на каждой итерации цикла происходит сдвиг «окна» параметров и их перенумерация (см. подразд. 3.5):

```
while [ ! -z $1 ] ; do
  echo $1
  shift
done
```

Для организации цикла по значениям заданного списка используется конструкция for … in … do … done, общий синтаксис которой имеет вид:

```
for <переменная> in <список> do
  <операторы>
done
```

Список значений <список> представляет собой текстовую строку с разделителями. В роли разделителей выступают пробелы, символы табуляции и символы перевода строки. Переменная, указанная в цикле, последовательно принимает значения элементов списка и может быть задействована в блоке операторов <операторы>. Число итераций этого цикла равно числу элементов списка.

Следующий фрагмент задания выполняет те же действия, что и предыдущий фрагмент — выводит все параметры, переданные заданию. В качестве списка здесь используется встроенная переменная BASH, содержащая все параметры командной строки задания, указанные через разделитель (см. подразд. 3.5):

```
for i in $@ do
  echo $i
done
```

В качестве списка цикла for редко используются фиксированные значения. Обычно список генерируется подстановкой значения в ходе выполнения задания. В последнем примере применяется подстановка значения переменной. Так же можно использовать подстановку вывода какой-либо команды. Например,

следующий фрагмент кода выводит содержимое всех файлов в текущем каталоге, предваряя вывод каждого файла его названием:

```
for i in `ls .` do
  echo === $i ===
  cat $i
done
```

3.4. Задания в Windows

3.4.1. Командный интерпретатор в Windows

Как и в Linux, ОС семейства Windows поддерживают работу с интерфейсом командного интерпретатора. Во многом его свойства были унаследованы от командного интерпретатора ОС MS-DOS и дополнены особенностями, присущими ОС семейства Unix/Linux. Рассмотрим некоторые свойства этой подсистемы в ОС Windows на основе командного интерпретатора, реализованного в системе Windows XP.

Для запуска командного интерпретатора в ОС Windows, основанных на ядре Windows NT, используется программа `cmd.exe`. Для запуска интерфейса командной строки необходимо открыть меню `Start` (`Пуск`), выбрать пункт меню `Run` (`Выполнить`) и запустить программу `cmd.exe`.

После запуска командного интерпретатора появляется окно, содержащее стандартное приглашение к работе. В ОС Windows XP данное приглашение может иметь следующий вид:

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
C:\>
```

Первые две строки информируют пользователя о версии ОС и проприетарных правах фирмы Microsoft. Последняя строка представляет собой стандартное приглашение к вводу команд или приему заданий от пользователя.

Ввод команд в интерпретаторе ОС Windows аналогичен используемому в ОС Linux. Например, можно вывести версию ОС следующим образом:

```
C:\>ver <Enter>
Microsoft Windows XP [Version 5.1.2600]
C:\>
```

3.4.2. Пакетная обработка в Windows

Механизм и средства реализации работы с заданиями в ОС Windows в основном сходны с подходом, используемым в ОС

Linux. Задания также оформляются в виде текстовых файлов, содержащих перечень команд, которые должны быть выполнены командным интерпретатором. Сами файлы должны иметь расширения `.bat` или `.cmd`. А исполнение, механизмы передачи параметров и результат работы командных скриптов в Windows такие же, что и в Linux.

Вызов команд в Windows также состоит из двух частей:

```
<имя команды> <параметры>
```

В качестве имени команды может использоваться либо внутренняя команда командного интерпретатора, либо имя исполняемого файла, содержащего код внешней программы.

Максимально допустимое количество параметров, как и в UNIX-системах, определяется максимально допустимой длиной строки. Для ОС Windows 2000 и Windows NT 4.0 максимально допустимая длина строки составляет 2 047 символов, а для систем Windows XP и более поздних — 8 191 символ.

Для получения полного списка команд, поддерживаемых командным интерпретатором, используется команда `help`. Для получения информации о специфических свойствах какой-либо команды необходимо выполнить следующую команду:

```
help <команда>
```

Все встроенные команды и большинство внешних программ, поддерживающих работу с командной строкой, имеют встроенный параметр `/?`. При вызове команды с этим параметром выводится справка о ее предназначении и параметрах, которые могут быть ей переданы.

3.4.3. Переменные

При создании командных сценариев пользователь имеет возможность оперировать переменными окружения. Переменные могут поступать от нескольких источников. В соответствии с источником определения той или иной переменной окружения их можно подразделить на встроенные системные, встроенные пользовательские и локальные.

Встроенные системные переменные определяются на уровне ОС и доступны всем процессам независимо от пользователя. Встроенные пользовательские переменные определяются на этапе входа пользователя в систему и существуют все время, пока продолжается сеанс работы данного пользователя. Для получения списка всех переменных окружения (как системы, так и пользователя) и их текущего значения используется команда `set`.

В противоположность встроенным переменным локальные переменные определяются на этапе выполнения командного сце-

нария. Для описания новой переменной или изменения ее текущего значения также используется команда set:

```
set <имя переменной>=<значение>
```

Для получения текущего значения переменной недостаточно указать имя переменной. Необходимо явно указать системе, что требуется именно значение переменной. Для этого используются символы % в начале и в конце имени переменной. Без использования этих символов система трактует имя переменной как обычную строчку. Эту особенность работы с переменными можно продемонстрировать на следующем примере, текст которого должен быть записан в файле example1.bat:

```
@set variable=value
@echo variable
@echo %variable%
```

В данном примере команда echo используется для вывода последовательности символов на консоль. Символ @ перед каждой командой сообщает командному интерпретатору о том, что не надо выводить команду на консоль перед ее выполнением. Если же этот символ не указывать, то во время исполнения сценария на консоль будет выводиться каждая команда перед ее выполнением. После выполнения файла example.bat на экране появятся строки

```
C:\>example.bat
variable
value
C:\>
```

Как следует из данного примера, в первом случае последовательность символов variable трактуется системой как обычная символьная строка, а во втором случае — возвращается значение, ассоциированное с соответствующей переменной. Таким образом можно поучить доступ не только к значениям локальных переменных, но и к значениям встроенных переменных.

Не все символы можно напрямую использовать при присвоении некоторого значения переменным, так как ряд символов зарезервированы и трактуются системой как команды или служебные символы. К таким символам относятся @, <, >, &, | и ^. Если необходимо использовать данные символы при означивании переменных, то им должен предшествовать символ ^:

```
set var=login^@e-mail.com
```

В качестве значений переменных могут использоваться не только строки, но и числа, и арифметические выражения. Для при-

своения числовых значений используется конструкция `set /a`. В математических выражениях могут фигурировать только целые числа в диапазоне от -2^{31} до $2^{31}-1$.

В выражениях могут использоваться и арифметические операции, такие как + (сложение), - (вычитание), * (умножение), / (деление), % (остаток от деления). Также существуют и комбинированные операторы присваивания, такие как += (прибавить и присвоить), -= (вычесть и присвоить), *= (умножить и присвоить), /= (разделить и присвоить) и %= (получить остаток от деления и присвоить):

```
set /a var=1
set /a res=%a%+%b%
set /a total+=1
set /a count*=(%amount%+1)
```

Чтобы удалить ранее определенную локальную переменную, используется следующая команда:

```
set <имя переменной>=
```

Локальные переменные доступны только для текущего экземпляра командной оболочки, а также для экземпляров оболочки, порожденных текущим экземпляром. Однако если необходимо некоторые переменные локализовать не только на уровне текущего экземпляра оболочки, а еще и на некотором локальном уровне, то можно использовать пару команд `setlocal` и `endlocal`. Любые изменения в переменных окружения, выполненных пользователем внутри области, которая ограничена данными командами, будут недействительны после выполнения команды `endlocal`. Рассмотрим пример использования команд, которые должны быть занесены в файл `example2.bat`:

```
@set variable=global value
@echo Before setlocal
@echo %variable%
@setlocal
@set variable=local value
@echo After setlocal
@echo %variable%
@endlocal
@echo After endlocal
@echo %variable%
```

В результате работы этого сценария на консоль будут выведены следующие сообщения:

```
Before setlocal
global value
```

```
After setlocal
local value
After endlocal
global value
```

Очевидно, что после выхода из локального блока все изменения, сделанные пользователем, были проигнорированы, а значения переменных были восстановлены.

Так же, как и в Linux, в Windows пользователь имеет доступ не только к встроенным и локальным переменным, но и к особым системным переменным. Эти переменные позволяют осуществлять доступ к параметрам, передаваемым в вызываемый сценарий.

В командном интерпретаторе Windows определены специальные переменные `%0..%9`. Переменная `%0` замещается именем выполняемого файла сценария, а параметры `%1..%9` замещаются первыми девятью параметрами, переданными в сценарий.

Для получения доступа к параметрам, следующим за девятым, используется команда `shift`. Ее поведение похоже на поведение одноименной команды в BASH: после однократного вызова этой команды переменная `%1` сопоставляется с вторым параметром, `%2` — с третьим и т.д.

Для получения списка всех параметров существует специальная встроенная переменная `%*`. Следует отметить, что команда `shift` оказывает влияние не только на значения позиционных переменных `%1`, `%2` и т.д., но и на значение, возвращаемое переменной `%*`.

Параметры командной строки могут использоваться не только напрямую, но к ним могут применяться специальные модификаторы. Эти модификаторы необходимы для выделения из параметров командной строки имен файлов, каталогов, времени создания соответствующих файлов и т.д. Для применения модификаторов служит последовательность `%~`, за которой следует модификатор и позиционный номер параметра, к которому данный модификатор применяется. Список модификаторов параметров командной строки приведен в табл. 3.1.

Таблица 3.1. **Модификаторы параметров командной строки**

Модификатор	Описание
`%~`	Возвращает значение параметра, удаляя окружающие двойные кавычки, если они есть
`%~f`	Возвращает полный путь к файлу, заданному параметром

Окончание табл. 3.1

Модификатор	Описание
`%~d`	Возвращает букву диска, на котором хранится файл, заданный параметром
`%~p`	Возвращает полный путь к каталогу, в котором хранится файл, заданный параметром
`%~n`	Возвращает имя файла, заданного параметром
`%~x`	Возвращает расширение файла, заданного параметром
`%~s`	Возвращает путь к файлу, заданному параметром, но используя имена в формате 8.3
`%~a`	Возвращает атрибуты файла, заданного параметром
`%~t`	Возвращает дату и время файла, заданного параметром
`%~z`	Возвращает размер файла, заданного параметром
`%~$PATH:`	Сканирует список каталогов, определенных в переменной PATH, и ищет в этих каталогах имя файла, заданного параметром. Если такой файл найден, то возвращается полный путь к самому первому найденному файлу. Если такой файл не найден, то возвращается пустая строка

Модификаторы могут использоваться не только по отдельности, но и в комбинации друг с другом. Например, комбинация `%~nx1` разбирает первый параметр и извлекает из него имя файла и расширение, а комбинация `%~ftza1` выводит на консоль первый параметр в том же формате, в котором выводит команда `dir`.

3.4.4. Ввод/вывод. Конвейерная обработка

Организация механизма ввода/вывода информации в консольных приложениях Windows сходна с организацией этого механизма в Linux. Операции чтения с консоли или записи в консоль осуществляются не напрямую, а через некоторые виртуальные файлы устройств.

Так же, как и в Linux, эти файлы имеют стандартные имена и за ними закреплены стандартные файловые дескрипторы, которые автоматически открываются при выполнении любого консольного приложения. Эти виртуальные файлы представляют стандарт-

ный поток ввода (связан с дескриптором 0), стандартный поток вывода (связан с дескриптором 1) и стандартный поток вывода ошибок (связан с дескриптором 2).

Существует возможность перенаправить потоки ввода/вывода, ассоциированные со стандартными потоками, и перенаправить их в файлы. Для этого используются операции перенаправления ввода/вывода. Синтаксис и семантика таких операций аналогичны используемым в Linux. Поэтому не будем подробно останавливаться на их пояснении, а сведем эти операции в единую табл. 3.2.

Таблица 3.2. **Операции перенаправления ввода/вывода**

Операция	Описание
`>`	`команда1 > файл` Поток стандартного вывода команды связывается с файлом или устройством. Если такой файл не существовал, то он создается; если он существовал, то файл перезаписывается
`>>`	`команда1 >> файл` Поток стандартного вывода команды связывается с файлом или устройством. Если такой файл не существовал, то он создается; если он существовал, то данные дописываются в файл
`<`	`команда1 < файл` Поток стандартного ввода команды связывается с файлом
`n >`	`команда1 n> файл` Поток вывода команды с файловым дескриптором n связывается с файлом. Файловые дескрипторы 0-2 связаны со стандартными устройствами ввода/вывода. Если такой файл не существовал, то он создается; если он существовал, то файл перезаписывается
`n >>`	`команда1 n>> файл` Поток вывода команды с файловым дескриптором n связывается с файлом. Если такой файл не существовал, то он создается; если он существовал, то данные дописываются в файл
`n >& m`	`команда1 n>& m` Поток вывода команды с файловым дескриптором n связывается с потоком с файловым дескриптором m

Окончание табл. 3.2

Операция	Описание
`n <& m`	`команда1 n<& m` Поток ввода команды с файловым дескриптором n связывается с потоком с файловым дескриптором m
`|`	`команда1 | команда2` Связывает стандартный поток вывода одного процесса со стандартным потоком ввода второго процесса

Так же, как и в Linux, в Windows эти операции можно комбинировать и использовать сразу несколько операций в команде. Например, чтобы перенаправить все данные, выводимые некоторой командой, как на стандартное устройство вывода, так и на стандартное устройство вывода ошибок, можно использовать следующую команду:

```
команда >> output.log 2>&1
```

Подобным же образом можно связывать и несколько операций организации межпроцессного взаимодействия с помощью каналов:

```
команда1 | команда2 > log.txt
```

В этом случае данные со стандартного вывода `команда1` поступают на стандартный ввод `команда2`, а данные со стандартного вывода `команда2` выводятся в файл `log.txt`.

3.4.5. Управление ходом выполнения заданий

Windows поддерживает несколько различных механизмов управления ходом выполнения программ. Это механизмы последовательного выполнения команд, группировки команд, условного выполнения команд, а также условные операторы выполнения и циклы.

Операции последовательного и условного выполнения команд очень похожи на соответствующие средства, определенные в Linux. Краткое описание этих средств приведено в табл. 3.3.

Если необходимо, эти операции могут быть скомбинированы в одной строке. При этом допускается использование не только одних и тех же операций, но и объединение в одну команду нескольких различных операций. В качестве примера можно рассмотреть следующий вариант использования этих команд:

Таблица 3.3. **Операции последовательного и условного выполнения команд**

Операция	Описание
`&`	`команда1 & команда2` Используется для разделения нескольких команд в одной строке. Сначала исполняется первая команда, по окончании ее работы исполняется вторая
`&&`	`команда1 && команда2` Используется для условного выполнения нескольких команд. Выполняется первая команда, если она завершилась успешно (т.е. вернула 0 в качестве кода возврата), то выполняется вторая
`||`	`команда1 || команда2` Используется для условного выполнения нескольких команд. Выполняется первая команда, если она завершилась неуспешно (т.е. вернула код, отличный от 0), то выполняется вторая
`()`	`(команда1 & команда2)` Используется для группировки последовательности команд
`;` или `,`	`команда1 параметр1;параметр2` Используется для разделения параметров, передаваемых команде в командной строке

```
(((cd c:\logs && dir) > list.txt 2>&1) || echo
Unable to get list of logs) & start list.txt
```

Здесь делается попытка перейти в каталог `c:\logs`. Если эта операция завершается успешно, то выполняется команда `dir` получения списка файлов в текущем каталоге и этот список записывается в файле `list.txt`. Если хотя бы одна из операций завершается неуспешно, на консоль выводится сообщение `Unable to get list of logs`, а в файл `list.txt` записывается системная информация о возникшем сбое. После окончания работы этих команд запускается приложение, ассоциированное с расширением `.txt` (например, `notepad`), и ему в качестве параметра передается файл `list.txt`.

Как и многие языки описания заданий, в командном языке Windows поддерживаются условные операторы. Причем следует отметить, что существует несколько возможных форм задания условных операторов. Общая форма записи условного оператора выглядит следующим образом:

```
if выражение команда1 [else команда2]
```

В этом случае команда1 выполняется, если выражение выполняется. В противном случае выполняется команда2. При этом при использовании в условном операторе `else` конструкции оператор `else` должен находиться на той же строке, что и оператор `if`. Иначе интерпретатор сообщит о наличии ошибки в коде сценария.

В качестве выражение могут использоваться несколько конструкций, в зависимости от целей условного оператора. Список этих выражений приведен в табл. 3.4.

Таблица 3.4. **Формы логических условий**

Логическое условие	Описание
`[not] errorlevel номер`	`if not errorlevel 0 echo Operation is failed` Используется для анализа успешности или неуспешности выполнения предыдущей команды или операции. Код 0 чаще всего означает, что предыдущая команда выполнилась успешно. Код, отличный от 0, чаще всего говорит о том, что выполнение команды завершилось неуспешно

`[not] строка1==строка2`	`if %1==%VAL% (`
	`echo Strings are the same`
	`) else (`
	`Strings are different`
	`)`
	Используется для сравнения двух строк на равенство/неравенство. Если применяется форма с оператором `else`, то команды в теле условного оператора должны группироваться с помощью конструкции `()`, так как все команды обязательно должны завершаться символом конца строки. В противном случае интерпретатор не сможет выполнить эти команды и проигнорирует их

`[not] exist имя_файла`	`if exist temp.txt del temp.txt` Используется для проверки существования/несуществования файла с именем `имя_файла`. `Имя_файла` может задаваться как в относительной форме записи пути, так и в абсолютной

Окончание табл. 3.4

Логическое условие	Описание
`if [/i] стр1 ОпСравнения стр2`	`if /i %2 EQU /a echo Archive option is specified` Используется для сравнения строковых переменных или параметров командной строки. В качестве `ОпСравнения` могут применяться следующие трехбуквенные операции: EQU — равно NEQ — не равно LSS — меньше чем LEQ — меньше или равно GTR — больше чем GEQ — больше или равно Если указан параметр `/i`, то строки сравниваются без учета регистра
`if cmdextversion номер`	`if cmdextversion 2 echo The operation is supported` Используется для проверки текущей версии расширения команд. Возвращает истину, если внутренний номер расширения команд больше или равен заданному номеру. Первая версия имеет внутренний номер 1
`if defined переменная`	`If defined Offset set /a MainAddr+=%Offset%` Используется для проверки факта существования специфицированной переменной

Для организации циклов интерфейс командной оболочки Windows предоставляет единственный оператор — `for`. Однако так же, как и условный оператор, оператор циклов имеет несколько различных форм, позволяющих использовать его для разных целей.

Общая форма записи оператора `for` имеет следующий вид:

```
for [параметр] {%переменная|%%переменная} in (множество) do команда [опции]
```

Параметр `%переменная` или `%%переменная` задает параметр, перебирающий элементы из `множество`. Первый вид записи используется в том случае, если цикл организуется из командной строки, а второй — если цикл является частью сценария.

Список элементов `множество` представляет собой набор элементов, которые и обрабатываются в цикле. Это могут быть спис-

ки файлов, строк, диапазоны значений и т.д. Элементы из `множество` последовательно перебираются в цикле и передаются в качестве опций в команды или программы `команда`.

Тип элементов, включенных в список `множество`, определяется опциональным параметром оператора `for`. Список параметров и типы элементов перечислены в табл. 3.5.

Таблица 3.5. **Элементы цикла `for`**

Цикл	Типы элементов и описание
-	`for %%i in (*.doc) do echo %%i` Используется для обработки списков файлов. Для задания списков файлов могут применяться шаблоны с применением метасимволов * и ?
/D	`for /D %%dir in (*) do echo Directory %%i` Используется для обработки списков каталогов. Для задания списков каталогов могут применяться шаблоны с использованием метасимволов * и ?
/R [корневой_каталог]	`for /R C:\Windows %%dir in (*) do echo Directory %%i` Используется для рекурсивного обхода дерева каталогов, начиная с каталога `корневой_каталог`. Если каталог не указывается, то производится рекурсивный обход каталогов в текущем каталоге. Элементами списка являются файлы, расположенные в каталогах. Если в качестве элементов множества указан элемент «.», то список формируется не из файлов, а из вложенных каталогов
/L	`for /L %%i in (1, 1, 5) do set /a Res+=i` Используется для перебора значений в заданном диапазоне с заданным шагом. Формат списка элементов имеет вид `(начало, шаг, конец)`

/F [“параметры”]	`for /F “tokens=1-3” %%i in (log.info) do @ echo Time: %%i, Date: %%j, Event: %%k`
	`for /F “tokens=1,2*” %%i in (“12.34 05.08.08 ev_3456 - String is too long”) do @echo Time: %%i, Date: %%j, Event: %%k`
	`for /F “tokens=1,2*” %%i in (‘type log.info’) do @echo Time: %%i, Date: %%j, Event: %%k`
	Используется для обработки списка строк, получаемых либо из файла, либо из строки, задаваемой напрямую, либо из вывода команды или программы

К параметрам функции могут применяться модификаторы, используемые для параметров командной строки. Например, в следующем примере на консоль выводится список файлов с расширением .log, расположенных в текущем каталоге, в том же формате, в котором выводится список файлов командой dir:

```
for %%I in (*.log) echo %%~ftzaI
```

Для изменения последовательности команд может применяться оператор безусловного перехода goto:

```
goto метка
```

После выполнения этой команды происходит переход к строке, в которой задана соответствующая метка. Если такая метка не найдена, то выполнение сценария прерывается и выдается сообщение о том, что метка не найдена.

Оператор goto также может использоваться для досрочного прерывания выполнения сценария:

```
goto :EOF
```

Еще одной особенностью оператора goto является то, что он может использоваться в качестве замены оператора выбора:

```
goto lab%1
:lab1
echo Variant 1
goto :EOF
:lab2
echo Variant 2
goto :EOF
:lab3
echo Variant 3
goto :EOF
```

В этом примере принимается параметр из командной строки в диапазоне от 1 до 3 и в зависимости от значения этого параметра выбирается соответствующий вариант.

Для вызова одного командного файла из другого используется команда call. Формат вызова в этом случае выглядит следующим образом:

```
call [диск:][путь]имя_файла [параметры]
```

Здесь диск, путь и имя_файла задают имя командного файла, который должен быть выполнен. Данный файл должен иметь расширение .bat или .cmd. Если необходимо, то командному файлу могут быть переданы параметры:

```
call checkdate.bat file1.txt
```

Команда `call` используется не только для запуска внешних командных файлов, но и для организации вызова функций и процедур. Для этого существует следующая форма данной команды:

```
call :метка [параметры]
```

При организации вызова функции `:метка` должна указывать на начало вызываемой функции. Параметры, передаваемые при вызове, доступны внутри функции при использовании позиционных параметров `%1, …, %9`.

Для возврата из функции предназначена команда `exit` с ключом `/b`:

```
exit /b [код_возврата]
```

Механизм организации функций и процедур достаточно мощен. Он позволяет организовывать рекурсивные вызовы. В качестве примера работы с процедурами можно рассмотреть сценарий `Fibonacci.bat`:

```
@echo off
rem Отключаем режим вывода команд
set N=%1
call :fib %N%
echo %RESULT%
exit /b
rem Функция вычисления чисел Фибоначчи
:fib
if %1 == 1 (
  set RESULT=1
  exit /b
)
if %1 == 2 (
  set RESULT=1
  exit /b
)
rem Вычисляем число N-2
set /a M=%1-2
rem Вычисляем число Фибоначчи номер N-2
call :fib %M%
rem Сохраняем вычисленное число Фибоначчи номер N-2
в стеке
set /a M=%1-2
set RES%M%=%RESULT%
rem Вычисляем число Фибоначчи номер N-1
set /a M=%1-1
call :fib %M%
```

```
rem Извлекаем число Фибоначчи номер N-2 из стека,
складываем с числом Фибоначчи номер N-1 и резуль-
тат записываем в переменную RESULT
set /a M=%1-2
for /F %%i in ('echo %%RES%M%%%') do set /a
RESULT+=%%i
exit /b
```

При вызове этого исполняемого файла с числовым параметром на экран будет выведено число, соответствующее его факториалу:

```
C:\>Fibonacci.bat 8
21
```

К сожалению, при организации рекурсивных вызовов не сохраняются значения переменных. Это приводит к тому, что стеки значений приходится организовывать самому программисту, поэтому приведенный выше пример отличается от классической реализации чисел Фибоначчи.

В рассмотренном примере организуется стек переменных для хранения ранее полученных промежуточных результатов (команда `set RES%M%=%RESULT%`). Число `%M%` задает текущий уровень стека. Затем применяется оператор `for`, позволяющий получить вывод команды и записать его в переменные, для извлечения значений из стека. Эти значения и используются при вычислении чисел Фибоначчи.

В целом следует отметить, что инструменты командной строки ОС Windows уступают по мощности и удобству средствам, предоставляемым различными оболочками в UNIX/Linux-системах. Разработчики фирмы Microsoft регулярно расширяют существующую функциональность командного интерпретатора и добавляют новые средства, но это не может радикально изменить ситуацию с созданием командных сценариев — их разработка до сих пор достаточно тяжела и неудобна.

Поэтому фирма Microsoft приняла решение о создании новой оболочки с интерфейсом командной строки и встроенным языком разработки сценариев, которая получила название PowerShell. Можно сказать, что эта оболочка развивает свойства таких оболочек, как Cmd.exe/Command.com, BASH, WSH, Perl и др. К сожалению, новая оболочка интегрирована с платформой .NET Framework версии 2.0 и выше и может использоваться в ОС Windows XP, Windows Server 2003, Windows Vista и Windows Server 2008.

Контрольные вопросы

1. Перечислите основные свойства языка управления заданиями.
2. Чем различаются пакетный и диалоговый режимы работы?

3. Как можно обратиться более чем к девяти параметрам командной строки задания?

4. Как выполняется копирование переменных задания в среду выполнения задания, какие при этом существуют ограничения и проблемы?

5. Какими способами информация, выводимая программой, может быть использована заданием?

6. Напишите BASH-скрипт, выполняющий программу, имя которой передается в качестве 1-го параметра с параметрами «через один», считая со 2-го параметра, и имеющего следующие параметры запуска:

1-й — имя произвольной программы;

2-й и далее — параметры этой программы.

7. Напишите BASH-скрипт, выполняющий заданные программы, направляя их вывод в текстовый файл с заданным именем. При каждом запуске скрипта файл должен очищаться, вывод программ — добавляться к содержимому. BASH-скрипт должен иметь следующие параметры:

1-й — имя текстового файла;

2-й и далее — имена произвольных программ.

8. Каковы основные свойства языка управления заданиями в Windows?

9. Какие существуют средства перенаправления ввода/вывода?

10. Как пользователь может создавать подпрограммы в языке управления заданиями Windows?

11. Напишите сценарий на языке управления заданиями Windows, выполняющий программу, имя которой передается в качестве 1-го параметра с параметрами «через один», считая со 2-го параметра, и имеющего следующие параметры запуска:

1-й — имя произвольной программы;

2-й и далее — параметры этой программы .

12. Напишите сценарий, рекурсивно вычисляющий факториал заданного числа.

Глава 4

ПРАВА ДОСТУПА

4.1. Вход в систему

В главе 3 были рассмотрены некоторые основные способы взаимодействия с ОС, доступные пользователю. При этом не следует забывать, что ОС UNIX, о которой в основном идет речь — многопользовательская. Далее речь пойдет о том, как именно предоставляется многопользовательская поддержка, т. е. в основном об управлении сеансами и защите данных.

Чтобы начать сеанс работы с UNIX-системой, необходимо ввести свои учетные данные — пользовательское имя и пароль в ответ на *приглашение входа в систему*. Приглашение входа в систему обычно появляется на терминале после окончания загрузки системы. Подключение к серверу, на котором установлена ОС UNIX, может также производиться через сетевой протокол эмуляции терминала `telnet`. При таком сетевом подключении на экран компьютера пользователя передаются данные с экрана одного из логических терминалов, обслуживаемых сервером, а данные, вводимые пользователем с клавиатуры, передаются на сервер.

В обоих случаях приглашение для входа в систему приблизительно имеет следующий вид:

```
login:
```

В ответ на это приглашение требуется ввести свое учетное имя, после чего на экран будет выведен запрос пароля:

```
password:
```

Если учетное имя (логин) и пароль верны, то произойдет вход в систему и будет запущен основной командный интерпретатор, т. е. командный интерпретатор, активный в течение всего сеанса работы пользователя в системе. Завершение работы этого командного интерпретатора завершает сеанс работы пользователя. В большинстве систем по умолчанию для пользователей используется командный интерпретатор BASH.

Все рассмотренное выше справедливо, если пользователем системы является человек. Это верно далеко не всегда — в качестве пользователя системы может выступать любой объект, обладающий правами на использование объектов, предоставляемых ОС. Примером таких прав могут служить права доступа к определенным файлам, запуска на выполнение программ, доступа к устройству печати.

В роли пользователей могут выступать различные запускаемые программы, обладающие определенным набором прав. Такие программы запускаются от имени определенного пользователя, который в таком случае именуется псевдопользователем. Например, типичным псевдопользователем в UNIX-системах является пользователь с учетным именем `mail`, от лица которого происходит управление очередями сообщений электронной почты, доставка сообщений и их пересылка адресатам. Обычный пользователь, как правило, не может войти в систему под именем псевдопользователя — возможность доступа при помощи приглашения входа в систему для псевдопользователей заблокирована.

Кроме обычных пользователей и псевдопользователей в системе еще существует как минимум один суперпользователь, выполняющий роль системного администратора. Этот пользователь имеет исключительное право доступа ко всем ресурсам, предоставляемым ОС. В UNIX-системах этот пользователь обычно имеет учетное имя `root`.

4.2. Домашние каталоги пользователей

Каждому пользователю, работающему в ОС, выделяется каталог для работы и хранения в нем собственных файлов. Такой каталог традиционно называется *домашним каталогом*. Обычно домашние каталоги имеют имя, соответствующее логину пользователя, и находятся в каталоге `/home`. Например, для пользователя `sergey` домашний каталог обычно имеет имя `/home/sergey`.

Для указания домашнего каталога в путевом имени используется мнемоника `~`. Например, чтобы перейти в свой домашний каталог, достаточно ввести команду

```
cd ~
```

Для перехода в подкаталог `dirname` домашнего каталога необходимо ввести команду

```
cd ~/dirname
```

Чтобы перейти в домашний каталог пользователя с известным логином, используется мнемоника `~<login name>`. Например, для перехода в домашний каталог пользователя `nick` требуется ввести

```
cd ~nick
```

При этом переход в чужой домашний каталог будет произведен только в том случае, если достаточно прав доступа к этому каталогу (см. подразд. 4.4).

Кроме этого, имя домашнего каталога по традиции содержится в переменной окружения `$HOME`. Соответственно, в любой момент времени можно перейти в свой домашний каталог при помощи команды

```
cd $HOME
```

Значение этой переменной можно изменить средствами BASH, но делать это не рекомендуется — большинство системных утилит UNIX используют значение этой переменной для определения пути, по которому возможно сохранение файлов с настройками этих программ. Если в переменной `$HOME` будет указано произвольное значение, сохранение настроек окажется невозможным, а некоторые программы станут неработоспособны.

4.3. Идентификация пользователей

Каждый пользователь системы имеет уникальное в пределах данной системы учетное имя (логин). Однако имя присваивается пользователю только для его удобства — ОС различает пользователей по их уникальным идентификаторам — UID (User IDentifier). Идентификатор UID представляет собой целое число, большее или равное нулю, он уникален, как и учетное имя пользователя.

Кроме UID и логина для каждого пользователя определяется набор атрибутов, содержащих системную и справочную информацию: пароль, паспортное имя, путь к домашнему каталогу, полное имя исполняемого файла командного интерпретатора по умолчанию. Эта информация хранится в файле `/etc/passwd`. Информация о каждом пользователе находится в отдельной строке, атрибуты разделены символами двоеточия `:`. Последовательность атрибутов следующая: учетное имя, пароль (в зашифрованном виде), идентификатор пользователя (UID), идентификатор группы (GID), паспортное имя, путь к домашнему каталогу и полное имя командного интерпретатора. Например, пользователь Вася Пупкин, имеющий UID 1001, будет представлен в файле `/etc/passwd` следующей строкой:

```
vasya:Fes8s9xapl:1001:10:Vasya Pupkin:/home/
vasya:/bin/bash
```

Пользователь всегда является членом одной или более групп пользователей. Даже если пользователь является единственным человеком, имеющим доступ к системе, — он является членом как минимум группы «Пользователи системы» (обычно с именем `users`) или группы «Администраторы» (обычно с именем `root`).

Группа пользователей — множество пользователей, задаваемое в виде списка. Объединение пользователей в группы, как правило, происходит по принципу разграничения задач, выполняемых пользователями. Так, в отдельную группу обычно выделяются администраторы системы.

В программном комплексе «Контроль знаний» можно выделить следующие группы пользователей — разработчик системы, студент, преподаватель. Такое разделение по группам связано, в первую очередь, с тем, что пользователи должны иметь разные уровни доступа к системе.

Так, разработчик должен иметь возможность обновлять файлы заданий, составляющих основу системы, и модифицировать структуру каталогов системы, преподаватель должен иметь право обновлять базу контрольных работ и проверять выполненные работы, а студент — просматривать и выполнять полученные работы.

Разработчик системы имеет учетное имя devel, преподаватель — teacher, студенты — произвольные имена, совпадающие с именами их рабочих каталогов в системе. Кроме того, teacher и devel входят в группу teacher, которая используется для ограничения доступа студентов к некоторым каталогам.

Каждая группа имеет уникальное учетное имя группы и уникальный идентификатор группы GID (Group IDentifier).

Информация о группах, определенных в системе, хранится в файле `/etc/group`. Информация о каждой группе находится в отдельной строке, атрибуты группы разделены символами двоеточия `:`. Последовательность атрибутов следующая: Учетное имя группы, флаг состояния группы (обычно — символ `*`), идентификатор группы (GID), список пользователей, входящих в группу, перечисленных через запятую. Пример строки файла `/etc/group` показан ниже:

```
users:*:1001:sergey,nick,alex
```

4.4. Права доступа к файлам и каталогам

4.4.1. Ограничения доступа

Во многих ОС существуют средства ограничения доступа к файлам. В однопользовательских ОС обычно ограничивается доступ к файлам ядра ОС (например, установка атрибута «скрытый» в DOS). В многопользовательских ОС доступ ограничивает-

ся при помощи определения *прав доступа* того или иного пользователя к файлам. Права доступа определяют возможность использования той или иной операции над файлом.

Вообще говоря, термин «права доступа к файлу» не совсем корректен. Права доступа задаются для наборов данных, хранимых на диске. Для всех файлов, связанных с этим набором данных (см. подразд. 2.1), определены те же самые права доступа, что и для набора данных. Для всех файлов, связанных с одним набором данных, эти права одинаковы.

Доступ пользователей к файлам определяется элементами тройки субъект — объект — право доступа. При этом под *субъектом* понимается пользователь, обращающийся к объекту или определенное множество пользователей, под *объектом* — файл или каталог, а под *правом доступа* — разрешение или запрещение на выполнение субъектом той или иной операции над объектом (рис. 4.1).

В UNIX-системах существует три типа субъектов, для которых устанавливаются права: владелец файла, или пользователь-владелец (`u`), владеющая файлом группа пользователей, или группа-владелец (`g`) и все пользователи системы (`o`).

Для каждого типа субъектов можно запретить или разрешить выполнение операций трех типов: чтение (`r`), запись (`w`) и выполнение (`x`).

Под *правом на чтение файла* понимается возможность открыть этот файл и прочитать содержащуюся в нем информацию; под *правом на запись* — возможность изменить содержащиеся в файле данные или удалить файл; под *правом на выполнение* —

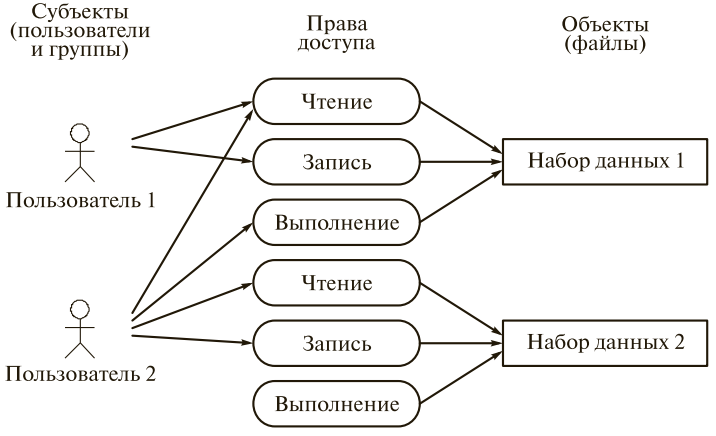

Рис. 4.1. Задание прав доступа при помощи троек субъект — объект — право доступа

возможность запустить на выполнение программу, которая содержится в этом файле (при этом образуется выполняющийся процесс).

Каждый файл в UNIX-системе имеет двух владельцев — пользователя-владельца и группу-владельца. При этом пользователь-владелец не обязан быть членом группы-владельца, что позволяет более гибко управлять доступом к файлам и каталогам для различных пользователей.

Для получения информации о владельцах файлов и каталогов можно использовать ключ `-l` команды `ls`. Команда `ls -l` выводит подробную информацию о файлах в текущем каталоге следующим образом:

```
drwxrwx---  4 nikita users    0 Apr 28 23:25 texmf
-rwxr-x---  1 nikita users  375 Apr 26 04:27 textmode.o
-rwxr-x---  1 nikita users  452 Apr 26 04:27 readmode.o
drwxrwx---  2 nikita users    0 Apr 28 23:22 tk8.4
-rwxr-x---  1 nikita users 2948 Sep  2  2003 tkConfig.sh
```

В третьей колонке выводится учетное имя пользователя-владельца файла, в четвертой — учетное имя группы-владельца.

Первая колонка содержит символьную строку, кодирующую информацию об атрибутах файла, которые определяют права доступа к этому файлу.

Первый символ в первой колонке задает тип файла. Для каталогов первый символ имеет значение `d`, для обычных файлов — прочерк. Существуют и другие типы файлов, о которых подробнее рассказано в подразд. 6.2.

Символы с второго по десятый в первой колонке разбиты на тройки символов (триады), каждая из которых определяет права доступа определенного субъекта. Первая триада определяет права доступа пользователя-владельца файла, вторая триада — группы-владельца, третья триада — всех остальных пользователей системы.

Так, для файла `textmode.o` пользователь-владелец имеет право полного доступа к файлу (т.е. имеет права чтения, записи и выполнения), группа-владелец — на чтение и выполнение, а все остальные пользователи системы не имеют никаких прав доступа к этому файлу.

Действие атрибутов `r`, `w` и `x` на файлы достаточно очевидно — эти атрибуты определяют возможность выполнения системных вызовов Open, Read и Write для файлов (см. подразд. 2.3).

При наличии права на чтение файла программа пользователя может открыть этот файл вызовом Open и считывать данные вызовом Read. Аналогично при наличии права на запись программа пользователя использует вызовы Open и Write. Право на выполне-

ние программы, установленное на исполняемый файл программы, дает возможность пользователю передать управление программе, при этом ОС при помощи вызовов Open и Read перемещает исполняемый код программы в оперативную память и передает ему управление.

Для каталогов атрибуты `r`, `w` и `x` уже не столь очевидны, и связано это, в первую очередь, с отличиями системных вызовов Open, Read и Write для каталогов.

Право доступа `r` для каталога позволяет прочитать имена файлов, хранящихся в каталоге. Возможность доступа к этим файлам определяется уже правами доступа к файлам.

Право доступа `x` для каталога дает возможность получить все атрибуты файлов каталога, хранимые в их блоках метаданных. Если право `x` установлено для каталога и для всех его надкаталогов, то это позволяет перейти в этот каталог, т. е. сделать его текущим.

Право на запись в каталог дает возможность изменения содержимого каталога, т. е. возможность создания и удаления файлов в этом каталоге. Следует отметить, что при наличии права на запись в каталог существует возможность удалить файл, на который нет никаких прав доступа. Такое поведение системы легко объяснить, если вспомнить действие системного вызова Unlink — он удаляет жесткую ссылку на набор данных, не изменяя самого набора данных. Вследствие этого следует быть особенно внимательным при назначении права доступа на запись в каталог.

4.4.2. Задание прав доступа к файлам и каталогам

Права доступа к файлам и каталогам могут меняться с течением времени. Это может быть обусловлено, например, тем, что старый владелец файла передал права на его изменение новому владельцу, или тем, что файл, который раньше был доступен только одному пользователю, теперь доступен всем.

Примером такого изменения прав доступа к файлу с течением времени может служить изменение права доступа к варианту контрольной работы. Пока незаполненный вариант контрольной находится в общей базе преподавателя, он доступен только преподавателю. Как только файл с вариантом передается студенту — он должен стать доступным ему.

Чтобы обеспечить студенту доступ к этому файлу, можно либо сменить владельца, либо открыть полный доступ к этому файлу, например, следующим образом:

```
cp /check/teacher/theme1/var2.txt \
  /check/students/vasya/theme1_var2.txt
chmod 666 /check/students/vasya/theme1_var2.txt
```

Пользователь-владелец файла или группа-владелец могут быть изменены при помощи команды `chown` (CHange OWNer). Изменять владельца файла в большинстве UNIX-систем может только системный администратор.

Первый параметр команды `chown` определяет учетное имя пользователя и, возможно, учетное имя группы, которым передается владение файлом. Второй и последующие параметры содержат список файлов, для которых меняется владелец.

Если первый параметр определяет имя пользователя и имя группы, то они разделяются точкой. Например, команда

```
chown nick file.txt
```

сделает пользователя `nick` пользователем-владельцем файла `file.txt`. А команда

```
chown nick.admins file.txt
```

сделает пользователя `nick` пользователем-владельцем, а группу `admins` — группой-владельцем файла `file.txt`.

Права доступа к файлам и каталогам могут быть изменены при помощи команды `chmod` (CHange MODe). Изменять права доступа может только владелец файла или администратор системы. В качестве аргументов команде `chmod` передаются спецификаторы доступа и имена файлов, для которых определяется доступ.

Спецификатор доступа — строка символов, определяющая права доступа к файлу. Спецификатор состоит из трех следующих друг за другом элементов: субъекта, права доступа и операции над правом доступа. Для задания субъекта указываются символы `u` — пользователь-владелец, `g` — группа-владелец, `o` — все остальные пользователи, `a` — все три предыдущих субъекта. В качестве права доступа указываются символы `r` — чтение, `w` — запись и `x` — выполнение. Операции над правами доступа изменяют набор прав доступа для субъекта. Операция `-` снимает указанное право доступа, `+` — добавляет право доступа, `=` — присваивает субъекту явно заданный набор прав доступа.

Команда `chmod` с простым спецификатором доступа выглядит следующим образом:

```
chmod a+w file.txt # Разрешить всем запись
# в файл file.txt
```

В спецификаторы доступа, передаваемые команде `chmod`, субъекты могут быть перечислены через запятую:

```
chmod u+w,g=r file.txt #Разрешить
#пользователю-владельцу запись.
#Разрешить группе-владельцу
#только чтение из файла file.txt
```

В одном спецификаторе доступа может быть задано сразу несколько субъектов:

```
chmod ug+w file.txt #Разрешить пользователю-владельцу
#и группе-владельцу запись в файл file.txt
```

Также в одном спецификаторе может быть указано несколько операций и прав доступа для субъекта:

```
chmod u+w+r file.txt #Разрешить пользователю-владельцу
#чтение и запись в файл file.txt
```

Существует альтернативный способ задания прав доступа к файлу — числовой. Чтобы пояснить переход от уже рассмотренного способа к числовому, запишем все триады:

```
rwx rwx rwx
```

Примем за двоичную единицу наличие права доступа и за двоичный нуль — его отсутствие. Тогда каждая триада может быть определена двоичным числом от 000 до 111. Так, триада `r-x` будет соответствовать двоичному числу 101. Каждое двоичное число затем может быть переведено в восьмеричную систему счисления.

В процессе такого перевода для триады

```
rwx rw- r—
```

получим

```
111 110 100
```

или

```
764
```

Числовое представление права доступа к файлу также может быть использовано в качестве спецификатора доступа команды `chmod`. Так, например, команда

```
chmod 764 file.txt
```

даст полный доступ пользователю-владельцу, разрешит только запись и чтение группе-владельцу и только чтение — всем остальным пользователям.

В программном комплексе «Контроль знаний» права на каталоги системы необходимо устанавливать таким образом, чтобы исключить возможность повреждения

частей системы ее пользователями. Кроме того, необходимо исключить доступ студентов к общей базе вариантов контрольных работ, а также исключить списывание (доступ студентов к чужим работам). Нужно обеспечить возможность выдачи преподавателем задания (копирования его в каталог студента) и сдачи работы (перемещения его из каталога выполненных работ).

Чтобы ограничить доступ пользователей к основным компонентам системы (скриптам и каталогам), их владельцем назначается пользователь devel (разработчик системы). Чтобы ограничить доступ пользователей к основным компонентам системы (скриптам и каталогам), их владельцем назначается пользователь devel (разработчик системы):

```
chown devel.teacher /check/scripts
```

При этом запись в этот каталог будет ограничиваться установленными правами:

```
chmod 755 /check/scripts
```

т.е. все пользователи получат доступ на чтение и выполнение скриптов, но только разработчик получит доступ на запись.

Пользователь devel является членом группы teacher (преподаватели), в которую входит также и преподаватель (пользователь с логином teacher). Эта группа введена для ограничения доступа студентов к каталогам контрольных работ.

Так, каталог /check/teacher должен иметь права доступа 700 и владельцев teacher.teacher:

```
chown teacher.teacher $TEACHERDIR
chmod 700 $TEACHERDIR
```

Таким образом, никто, кроме преподавателя, не имеет доступа к базе вариантов и сданным работам.

Каждый студент должен являться владельцем своего каталога, но преподаватель должен иметь доступ к этому каталогу на запись (для выдачи заданий) и доступ на чтение и выполнение к подкаталогу ready для сбора выполненных работ. Для этого пользователь-владелец каталога студента — сам студент, а группа-владелец — группа преподавателей. При этом студент имеет полные права на свой каталог, а преподаватель имеет только права на запись в каталог студента и полные права на подкаталог ready:

```
chown $i.teacher /check/students/vasya
chmod 730 /check/students/vasya
chown $i.teacher /check/students/vasya/ready
chmod 770 /check/students/vasya/ready
```

Полная версия задания на языке BASH, устанавливающая права доступа к каталогам системы, приведена в приложении 2.

4.4.3. Проверка прав доступа к файлам и каталогам

Ограничения прав доступа к файлам и каталогам значительно меняют информационное окружение, в котором выполняются задания пользователя. Если пользователь заведомо имеет полный доступ ко всем файлам, влияющим на работу задания, т. е. входящим в информационное окружение задания — беспокоиться не о чем. Однако может возникнуть ситуация, когда необходимые файлы или каталоги не доступны в требуемом режиме доступа. Для этого необходимо предусматривать в задании обработку таких исключительных ситуаций. Как правило, обработчики исключительных ситуаций помещаются в начале текста задания и проверяют возможность использования информационного окружения заданием. В данном случае под возможностью использования понимается наличие необходимых прав доступа.

Для проверки возможности доступа к файлу используется команда `test`, основные параметры которой уже рассматривались в подразд. 3.3.6. Команда `test` проверяет наличие того или иного права доступа у пользователя, от лица которого выполняется задание (текущего пользователя). Для этого используются следующие параметры команды:

`-r <файл>` — текущему пользователю разрешен доступ на чтение файла;

`-w <файл>` — текущему пользователю разрешен доступ на чтение файла;

`-x <файл>` — текущему пользователю разрешен доступ на исполнение файла.

Например, чтобы проверить наличие права на запись у файла `outfile.txt` и права на чтение у файла `infile.txt`, достаточно выполнить следующий фрагмент задания на языке BASH:

```
if [ ! –w outfile.txt –a ! –r infile.txt ] ; then
  echo "Insufficient access rights"
  exit 1
fi
```

В случае нехватки прав доступа задание, в которое будет включен такой фрагмент, выведет на экран сообщение "`Insufficient access rights`" и завершит свое выполнение с кодом возврата 1.

Контрольные вопросы

1. Каким образом идентифицируются пользователи в UNIX-системах?
2. Какие виды ограничений доступа к файлам и каталогам применяются в UNIX-системах, какой эффект они дают?
3. При помощи каких команд и каким образом изменяются права доступа к файлам и каталогам?
4. Каким образом проверяются права доступа к файлам и каталогам?
5. В чем состоят различия прав доступа на чтение и на выполнение у файлов и каталогов?
6. Напишите BASH-скрипт, выводящий только файлы, находящиеся в домашнем каталоге пользователя и недоступные текущему пользователю по чтению.
7. Пересчитайте права доступа к указанным ниже файлам в восьмеричное представление:

```
-rwxr-xrw- 1 nikita users 375 Apr 26 04:27 textmode.o
-rwxr-xr-- 1 nikita users 452 Apr 26 04:27 readmode.o
drwxr-xr-x 2 nikita users 0 Apr 28 23:22 tk8.4
-rw-r-xrw- 1 nikita users 2948 Sep 2 2003 tkConfig.sh
```

Глава 5

ПРИКЛАДНОЕ ПРОГРАММИРОВАНИЕ В СРЕДЕ UNIX

5.1. Задания и прикладные программы

Написание заданий на языке командного интерпретатора расширяет список доступных пользователю команд — каждое задание, оформленное в виде исполняемого файла, может быть запущено как команда. Возможности таких команд обычно ограничены тем фактом, что языки управления заданиями обычно позволяют только комбинировать другие готовые команды (как внутренние команды интерпретатора, так и внешние, существующие в виде исполняемых файлов), но не предоставляют никаких возможностей доступа к функциям ядра ОС— к системным вызовам. В случае необходимости использования системных вызовов используются программы, написанные на одном из языков высокого уровня (реже — на языке ассемблера). Стандартным языком подавляющего большинства UNIX-систем является язык C, который при дальнейшем изложении будет использоваться в качестве основного. При этом будут использоваться стандартные системные вызовы, одинаковые практически во всех вариантах UNIX. Подразумевается, что читатель знаком с программированием на языке C, поэтому в данном разделе приведены только основные особенности, специфичные для UNIX — рассмотрены только заголовочные файлы, определяющие формат системных вызовов, и параметры командной строки компилятора.

5.2. Заголовочные файлы

Стандартные заголовочные файлы, содержащие определения системных вызовов, находятся в каталоге `/usr/include`. Каждый файл содержит определения функций и типов данных. При этом для определений системных вызовов характерен отказ от использования стандартных типов файлов. Например, системный вызов

```
#include <time.h>
clock_t clock(void);
```

возвращающий количество микросекунд, прошедших с момента старта процесса, вызвавшего функцию `clock()`, использует тип `clock_t`. В том же заголовочном файле `time.h` тип `clock_t` обычно определяется как `long`.

Однако, если в каком-нибудь из вариантов UNIX этот тип будет изменен (например, заменен на `long long`), совместимость заголовков функций сохранится.

Названия основных заголовочных файлов, которые содержат определения системных вызовов и библиотечных функций, и краткие комментарии относительно содержимого этих файлов приведены в табл. 5.1. Имена файлов указаны относительно каталога `/usr/include`.

Таблица 5.1. **Основные заголовочные файлы, содержащие системные вызовы и библиотечные функции**

Заголовочный файл	Содержание
`dirent.h`	Функции и типы данных для работы с каталогами — для получения списка файлов в каталоге, для обращения к метаданным файлов и т. п.
`errno.h`	Функции, константы и макроопределения для обработки ошибочных ситуаций, преобразования числовых кодов ошибок в текстовую форму
`fcntl.h`	Функции, типы данных и макроопределения для работы с файлами — для их создания, открытия, изменения атрибутов
`float.h`	Функции и типы данных для работы с числами с плавающей запятой
`limits.h`	Константы, задающие ограничения ОС — максимальную длину имени файла, размеры типов данных, макимальный объем адресуемой памяти и т. д.
`math.h`	Функции для выполнения различных математических операций — тригонометрические, логрифмические функции, возведение в степень и т. д.
`pwd.h`	Типы данных и функции для работы с файлом паролей `/etc/passwd` — функции получения данных о пользователях из файла паролей

Окончание табл. 5.1

Заголовочный файл	Содержание
`regex.h`	Функции и типы данных для работы с регулярными выражениями. Регулярные выражения — расширенный вариант символов подстановки `*` и `?` (более подробно см. [12])
`signal.h`	Функции и типы данных для обработки сигналов (более подробно см. подразд. 9.3)
`stdarg.h`	Функции и типы данных для работы с функциями с переменным числом аргументов
`stddef.h`	Определения стандартных типов
`stdio.h`	Функции и типы данных стандартной библиотеки ввода/вывода
`stdlib.h`	Функции и типы данных стандартной библиотеки
`string.h`	Функции и типы данных для работы со строками: конкатенации, копирования, сравнения и т.д.
`time.h`	Функции и типы данных для работы с системным таймером — установки и считывания системного времени, измерения отрезков времени и т.д.
`unistd.h`	Функции основных системных вызовов и макроопределения для задания различных режимов работы системных вызовов
`sys/ipc.h`	Функции и типы данных для поддержки межпроцессного взаимодействия IPC (см. подразд. 9.3.2 и 9.3.3)
`sys/sem.h`	Функции и типы данных для работы с семафорами (см. подразд. 9.3.3)
`sys/types.h`	Типы данных для работы с системными таблицами ядра

5.3. Компиляция программ в UNIX

Компиляция программ в UNIX проводится в два этапа — на первом этапе из исходных текстов при помощи компилятора `cc` (или `gcc`) формируются объектные файлы (с расширением `.o`),

на втором этапе при помощи сборщика ld формируется исполняемый файл или файл библиотеки.

Управление процессом компиляции и сборки (например, уровень оптимизации кода, формат выходного файла, пути к системным библиотекам) задаются ключами запуска компилятора и сборщика. Для компиляции и сборки простых программ достаточно вызывать только компилятор gcc, который далее автоматически запустит сборщик ld. Например, чтобы получить исполняемый файл myprogram на основе исходного текста программы из файла myprogram.c, достаточно воспользоваться командой

```
gcc -o myprogram myprogram.c
```

Здесь после ключа -o указано имя выходного исполняемого файла. После ключей указывается имя компилируемого файла.

Возможно указание нескольких файлов, содержащих определения функций и типов данных:

```
gcc -o myprogram myprogram1.c myprogram2.c
```

При такой компиляции в файлах исходных текстов не должно присутствовать функций, глобальных переменных или определений типов данных с одинаковыми именами, например, недопустимо несколько раз объявлять функцию main().

Для того, чтобы указать пути, по которым находятся заголовочные файлы и файлы библиотек, используются ключи -I и -L соответственно. Например, при компиляции с приведенными ниже параметрами компилятор будет искать заголовочные файлы в каталоге /usr/local/include, а библиотеки в каталогах /usr/local/lib и /usr/share/lib:

```
gcc -o myprogram -I/usr/local/include -L/usr/local/
lib -L/usr/share/lib myprogram.c
```

Файлы статических библиотек имеют расширение .a, а их имена начинаются со слова lib. Например, libm.a — имя библиотеки, содержащей программный код математических функций, определенных в заголовочном файле math.h.

Для подключения библиотек в процессе сборки исполняемого файла используется ключ -l. Имя подключаемой библиотеки указывается без префикса lib и без расширения .a. Так, например, для компиляции и сборки программы, использующей библиотеку libm.a, необходимо использовать вызов

```
gcc -o myprogram -L/usr/lib -lm myprogram.c
```

Здесь ключ -L задает расположение библиотеки, а -l задает ее имя.

Кроме статических большинство UNIX-систем поддерживает динамически подгружаемые библиотеки. Программный код функций, хранящихся в таких библиотеках, подгружается динамически во время выполнения программы. Файлы динамически подгружаемых библиотек имеют расширение `.so`, имена их файлов также начинаются с префикса `lib`. Например, динамически подгружаемый вариант библиотеки математических функций будет иметь имя `libm.so`.

Чтобы при сборке исполняемого файла в нем были установлены точки вызова функций из динамически подгружаемых библиотек, необходимо указывать ключ компилятора `–dynamic`. Следующая команда выполнит компиляцию и сборку исполняемого файла `myprogram` из исходного текста программы в файле `myprogram.c`, при этом будет подключен динамический вариант математической библиотеки, а поиск библиотечных файлов будет производиться в каталоге `/usr/local/lib`:

```
gcc –dynamic –o myprogram –L/usr/local/lib myprogram.c
```

Динамические библиотеки имеют существенное преимущество над статическими — программный код библиотечных функций находится вне исполняемого файла и может использоваться несколькими программами.

Поскольку в статических библиотеках программный код хранится непосредственно в исполняемом файле, то динамические библиотеки позволяют экономить дисковое пространство за счет устранения избыточности. Однако это преимущество динамических библиотек имеет и обратную сторону. Поскольку динамические библиотеки могут устанавливаться в систему отдельно от использующих их программ, то при отсутствии нужной библиотеки или при несовместимости версий библиотек программы, использующие динамические библиотеки, могут оказаться неработоспособными.

Для выявления причин неработоспособности программ во многих UNIX-системах существует команда `ldd`. После запуска этой программы с параметром — именем исполняемого файла — на экран будет выведен список используемых этим файлом динамических библиотек, пути к файлам библиотек и информация об отсутствии библиотек:

```
$ ldd `which scat`
libnsl.so.1 => No library found
libncurses.so.4 => /usr/lib/libncurses.so.4 (0x40035000)
libc.so.6 => /lib/libc.so.6 (0x40077000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2
(0x40000000)
```

Информация об отсутствии библиотеки — прямое указание на необходимость ее установки. Конфликт библиотек может быть выявлен косвенно по имени и пути к файлу библиотеки. Например, программа может ошибочно искать библиотеку в другом каталоге. В этом случае библиотеку необходимо или перенести в нужный каталог, или установить последовательность поиска библиотек по каталогам в переменной окружения `LD_LIBRARY_PATH`. Формат списка каталогов в этой переменой аналогичен формату списка в переменной PATH.

Переменные окружения могут использоваться и для упрощения работы с компилятором командной строки. Например, если необходимо компилировать программу, используя различные наборы одноименных библиотек, которые находятся в разных каталогах (например, библиотек, содержащих и не содержащих отладочную информацию), то путь к библиотекам можно помещать в переменную окружения (например, `MY_LIB`) и менять значение переменной в зависимости от текущей необходимости. Вызов компилятора при этом будет иметь следующий вид:

```
gcc –dynamic –o myprogram –L${MY_LIB} myprogram.c
```

Во всех рассмотренных выше способах компиляции и сборки программ от пользователя скрыт этап сборки — объектные файлы удаляются сразу после создания исполняемого файла, и пользователь получает исполняемый файл, не видя объектных. Иногда требуется явно выполнять два этапа — компиляцию и сборку, сохраняя объектные файлы. Например, некоторые библиотеки поставляются только в виде объектных файлов, без исходных текстов, и для сборки программ, использующих такие библиотеки, необходимо сначала получить объектные файлы своей программы, а затем собрать все объектные файлы (свои и библиотечные) в исполняемый файл.

Для получения объектных файлов используется ключ `–c` компилятора. Так, команда

```
gcc –c myfile1.c myfile2.c –I/usr/local/include
```

вызовет компиляцию файлов `myfile1.c` и `myfile2.c` и сохранение ее результатов в объектных файлах `myfile1.o` и `myfile2.o`. Поиск заголовочных файлов, необходимых для компиляции, будет производиться в каталоге `/usr/local/include`.

Чтобы получить исполняемый файл, собранный из объектных, достаточно вызвать компилятор, передав ему имена объектных файлов в качестве параметров. Компилятор распознает, что файлы не являются исходными текстами программ, и передаст их на обработку сборщику. Например, команда

```
gcc –shared –o myprogram –L/usr/local/lib –lm
myfile1.o myfile2.o
```

выполняет сборку программы `myprogram` из объектных файлов `myfile1.o` и `myfile2.o`. При этом в исполняемом файле будут проставлены точки вызова функций из динамической библиотеки `libm.so`, поиск которой на этапе сборки будет производиться в каталоге `/usr/local/lib` (при выполнении программы местонахождение библиотеки будет задаваться переменной окружения `LD_LIBRARY_PATH`).

Примером простейшей программы, реализовать которую при помощи задания на языке BASH достаточно трудно, может служить программа, которая возвращает число `x`, равное `sin(y)*100`, где `y` передается программе в качестве параметра. BASH не имеет встроенных тригонометрических функций, однако на языке C такая программа реализуется сравнительно просто:

```
#include <stdio.h>
#include <math.h>
int main(int argc, char **argv)
{
  double res;
  int angle;
  if (argc <= 1)
    return 0;
  atoi(argv[1], angle);
  res = sin(angle)*100;
  return (int)res;
}
```

Для компиляции этой программы можно воспользоваться следующей строкой вызова `gcc`:

```
gcc –lm –o sin –L/usr/lib –I/usr/include sin.c
```

Задание, которое будет использовать такую программу, может выглядеть следующим образом:

```
#!/bin/bash
if [ -z $1 ]; then
  echo "No parameters specified"
  exit 1
fi
if [ ! –z $2 ]; then
  echo "More than one parameter specified"
  exit 2
fi
./sin $1
echo "Hundredths of sine of angle $1 equals " $?
```

Типичный вывод на экран такого задания, запущенного с параметром `120`, имеет вид:

```
Hundredths of sine of angle 120 equals 86
```

Контрольные вопросы

1. В каком каталоге обычно находятся заголовочные файлы с описанием системных вызовов?
2. Какие ключи командной строки компилятора `gcc` предназначены для работы с библиотеками?
3. В чем различие статических и динамических библиотек?
4. Как можно определять причины проблем, связанных с конфликтами динамических библиотек?
5. Как задаются пути поиска динамических библиотек?
6. Напишите программу на языке C, которая выводит количество символов в переданном ей параметре. Параметры: 1-й — произвольная строка.
7. Напишите задание на языке BASH, которое вызывает компилятор `gcc` для компиляции и сборки исполняемого файла, из заданных исходных файлов.

Параметры:

1-й — каталог, в который помещается исполняемый файл;

2-й — имя исполняемого файла;

3-й и далее — имена исходных файлов.

Компиляция должна проводиться с использованием библиотек, имена которых указаны через пробел в переменной окружения `MY_LIB`. Задание должно обрабатывать ситуации отсутствия каталога или пустой переменной `MY_LIB`.

Глава 6

СПЕЦИАЛЬНЫЕ ВОПРОСЫ УПРАВЛЕНИЯ ДАННЫМИ

6.1. Стандартная структура системы каталогов в среде UNIX

Для хранения собственных данных в ОС, подобных UNIX, используется стандартная структура каталогов, показанная на рис. 6.1.

В зависимости от реализации конкретной ОС возможны некоторые отличия от приведенной структуры, например, в ОС SunOS и Solaris присутствует каталог `/opt`.

Каталог `/bin` служит для хранения базовых системных утилит. Как правило, выполнение этих утилит возможно даже при отсутствии основных системных библиотек. В каталоге `/boot` хранятся файлы ядра ОС и данные, необходимые для загрузки. Каталог `/dev` содержит специальные файлы, при помощи которых происходит обращение к устройствам. Каждый файл, находящийся в этом каталоге, соответствует какому-либо устройству. Запись данных в файл вызывает передачу этих данных на устройство, а чтение инициирует чтение данных с устройства. Каталог `/etc` предназначен для хранения конфигурационных файлов и файлов настроек. В каталоге `/home` находятся домашние каталоги пользователей (см. гл. 7). Каталог `/lib` предназначен для хранения системных библиотек, а `/lib/modules` — для хранения модулей ядра ОС, подгружаемых после завершения загрузки ядра драйверов и подсистем. Каталог `/proc` содержит специальные файлы, при помощи которых возможен доступ к системным таблицам ядра ОС и просмотр параметров оборудования. Каталог `/sbin` хранит системные утилиты, предназначенные для администрирования системы. В каталоге `/tmp` содержатся временные файлы.

В каталоге `/usr` находится программное обеспечение, входящее в комплект поставки ОС и не относящееся к системному. Назначение каталогов `/usr/bin`, `/usr/sbin`, `/usr/lib` аналогично назначению соответствующих каталогов верхнего уровня. В каталоге `/usr/doc` хранится документация на программное обеспечение, входящее в комплект поставки ОС, в каталоге `/usr/share` — разделяемые файлы данных. Каталог `/usr/include`, прежде

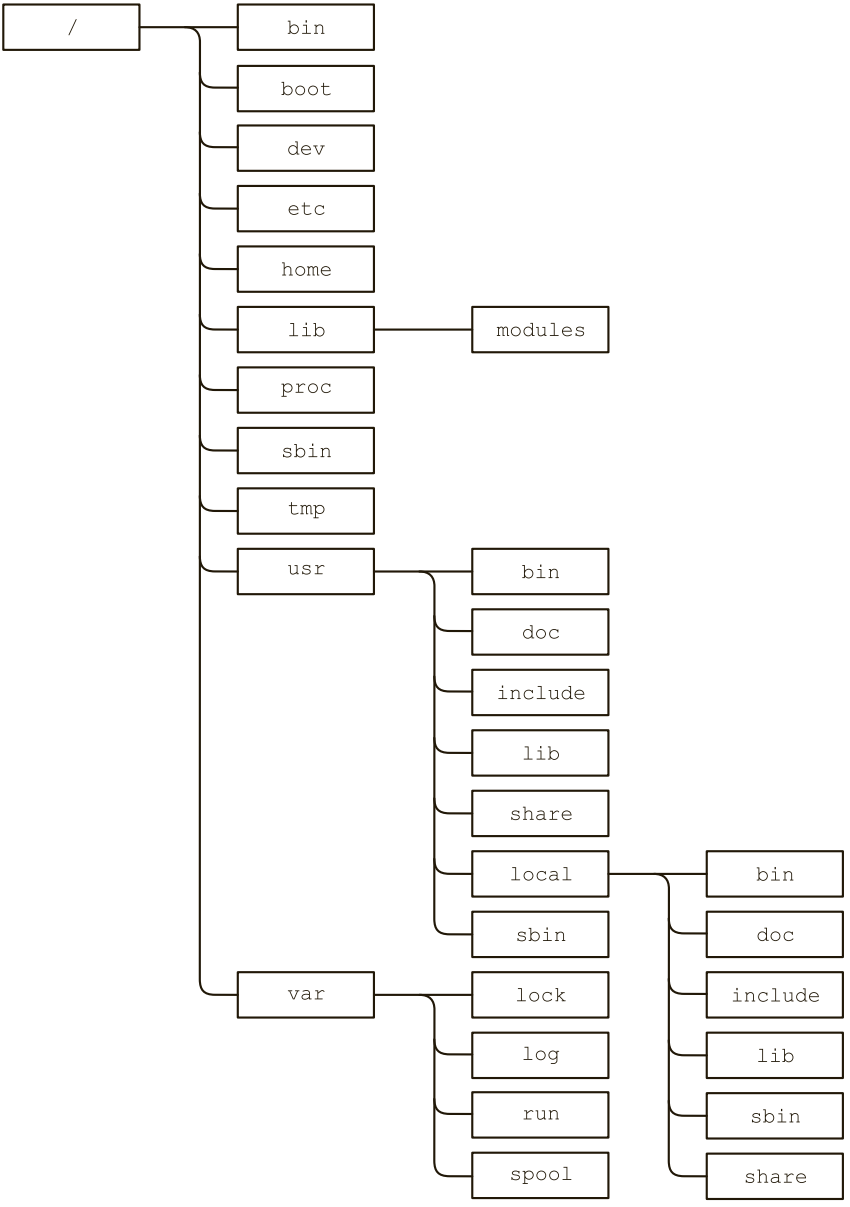

Рис. 6.1. Структура каталогов типичной UNIX-системы

всего, интересен разработчикам — в нем содержатся заголовочные *h*-файлы.

Каталог `/usr/local` предназначен для программ, устанавливаемых пользователем (не входящих в комплект поставки ОС). Его структура аналогична структуре каталога `/usr/`.

В каталоге `/var` хранятся файлы, связанные с текущей работой ОС. Каталог `/var/lock` содержит файлы блокировок, предотвращающие доступ к уже занятым не разделяемым ресурсам. Каталог `/var/log` предназначен для хранения системных журналов, в которые заносится информация обо всех существенных событиях, произошедших во время работы ОС.

В каталоге `/var/run` находятся файлы, указывающие на то, что в данный момент времени запущена та или иная системная утилита. Каталог `/var/spool` содержит буферы почтовых сообщений и заданий на печать.

6.2. Типы файлов

Файловая система может содержать файлы, различные по своему назначению. Один из вариантов определения назначения файлов — это указание для исполняемых файлов права доступа «исполняемый» (`x`). При попытке запуска таких файлов, как программы, ОС предпримет попытку создания процесса и запуска этой программы. Кроме этого, в UNIX-системах определяются атрибуты файлов, которые позволяют задавать другие типы файлов.

После вызова команды `ls -l` атрибут файла выводится в первой позиции первой колонки. Так, в приведенном ниже примере файлы `texmf` и `tk8.4` имеют атрибут `d`, который означает, что этот файл является каталогом, т.е. файлом, содержащим, в свою очередь, другие файлы (о различии между файлами и каталогами см. подразд. 2.2, 2.3):

```
drwxrwx---  4 nikita users    0 Apr 28 23:25 texmf
-rwxr-x---  1 nikita users  375 Apr 26 04:27 textmode.o
-rwxr-x---  1 nikita users  452 Apr 26 04:27 readmode.o
drwxrwx---  2 nikita users    0 Apr 28 23:22 tk8.4
-rwxr-x---  1 nikita users 2948 Sep  2  2003 tkConfig.sh
```

Поскольку кроме файлов и каталогов в большинстве файловых систем, поддерживаемых ОС UNIX, могут создаваться файлы и других типов, представляется разумным классифицировать все возможные типы файлов.

Необходимо отметить, что для обработки файлов различных типов ОС использует различные наборы системных вызовов, пред-

назначенные для работы с каждым поддерживаемым типом файлов. В гл. 2 эти различия были проиллюстрированы на примере системных вызовов для работы с файлами и каталогами.

Обычный файл. Этот тип файлов наиболее распространен. Такие файлы ОС рассматривает как последовательность байт, обработка их содержимого производится прикладными программами.

Каталог. Каталоги — основное средство задания иерархической структуры расположения файлов. С точки зрения внутренней структуры, каталог — это файл специального вида, который содержит имена находящихся в каталоге файлов и ссылки на блоки диска, в котором находятся метаданные соответстующих этим файлам блоков данных.

Символическая ссылка. В гл. 2 рассмотрены жесткие ссылки как способ именования одного набора данных несколькими именами. Каждая жесткая ссылка является элементом каталога, определяющим имя файла, соответствующего набору данных, и ссылку на блок метаданных набора данных. В отличие от жесткой ссылки, символическая ссылка представляет собой файл специального вида, содержащий строку — имя файла (с полным или относительным путем), на который указывает символическая ссылка. Если файл, чье имя хранится в символической ссылке, будет удален, то символическая ссылка будет неверна и будет указывать в «пустоту».

Символьное устройство. При помощи файлов символьных устройств программы пользователя могут обращаться к различным устройствам, поддерживаемым ОС. Чтобы данные, записываемые в файл устройства, были ему переданы, необходимо, чтобы ядро ОС поддерживало работу с устройством этого типа. Если устройство не поддерживается соответствующим драйвером в ядре ОС, файл устройства все равно может существовать на диске, но данные на устройство не попадают.

Обмен данными с устройством, доступным через файл символьного устройства, происходит в последовательном посимвольном режиме — за одну операцию чтения или записи передается на устройство или считывается с него только один символ. Файлы могут представлять собой как физические устройства (например, последовательный порт `/dev/ttyS0`), так и логические, поддерживаемые ОС для удобства работы пользователя или упрощения алгоритмов работы прикладных программ. Примером логических устройств могут служить файлы устройств, предназначенные для поддержки различных сетевых протоколов (`/dev/ppp` для протокола PPP).

Блочное устройство. Файл блочного устройства предназначен для информационного обмена с устройствами, требующими поступления за одну операцию чтения/записи блока данных фик-

сированного размера. Примером таких устройств может служить раздел жесткого диска: данные, записываемые на диск, поступают блоками с размером, кратным сектору диска. В остальном файлы блочных устройств аналогичны файлам символьных устройств.

Именованный канал. Файлы именованных каналов предназначены для обмена данными между запущенными процессами. Именованный канал представляет собой очередь, в которую любым процессом могут быть записаны данные, считываемые другим процессом. Таким образом возможно обеспечивать совместную работу процессов, решающих общую задачу. Объем данных, которые могут быть помещены в именованный канал, ограничен только свободным дисковым пространством, структура данных определяется процессами.

Доменное гнездо. Доменные гнезда также предназначены для обмена данными между различными процессами. Их основное отличие от именованных каналов состоит в том, что при помощи доменных гнезд (называемых также сокетами) возможно организовывать обмен данными между процессами, выполняемыми на различных физических компьютерах, объединенных в сеть. Для обмена данными с использованием доменных гнезд используется стек протоколов TCP/IP, что позволяет обмениваться данными как в пределах одного компьютера, так и в сети.

Чтобы определить тип файла в ходе выполнения задания на языке BASH, можно воспользоваться приведенными ниже ключами команды `test`:

`-f <файл>` — файл существует и является обычным файлом;

`-d <файл>` — файл существует и является каталогом;

`-c <файл>` — файл существует и является символьным устройством;

`-b <файл>` — файл существует и является блочным устройством;

`-p <файл>` — файл существует и является именованным каналом.

6.3. Монтирование дисков

Все файлы и каталоги, доступные пользователю UNIX-системы в ходе его сеанса работы, структурированы при помощи единой иерархической системы каталогов. С точки зрения логической структуры единая структура каталогов может объединять файлы, расположенные на разных дисках, т.е. от пользователя оказываются скрытыми вопросы физического размещения файлов на

накопителях. При этом файлы и каталоги, физически находящиеся на одном диске, представляются в виде поддерева общей системы каталогов. Содержимое физического корневого каталога этого диска представляется в виде содержимого некоторого каталога в общей системе каталогов. Каталог, относительно которого располагаются файлы на определенном физическом носителе, называется *точкой монтирования*, а сам процесс, после которого содержимое диска становится доступным пользователям ОС, — *монтированием диска*.

На рис. 6.2 показаны три дисковых накопителя — `hda1`, `hda2` и `sda1`, смонтированные в единую файловую систему. При этом точками монтирования является корневой каталог, каталог `home` и каталог `usr`, т.е. диск `hda1` содержит основную систему каталогов, а в ней к каталогам `home` и `usr` монтируются каталоги, содержащиеся на дисках `hda2` и `sda1`.

Разные физические диски могут иметь различные файловые системы, но при этом все равно объединяться при этом в единую файловую систему. Например, стандартной файловой системой для UNIX-подобной ОС Linux является система EXT2, а стандартной файловой системой для CD-дисков — ISO9660.

CD-приводы и другие сменные накопители являются основным типом устройств, требующих монтирования в процессе работы. Чтобы получить доступ к файлам, находящимся на CD-ROM-диске, его необходимо не только вставить в привод, но и смонти-

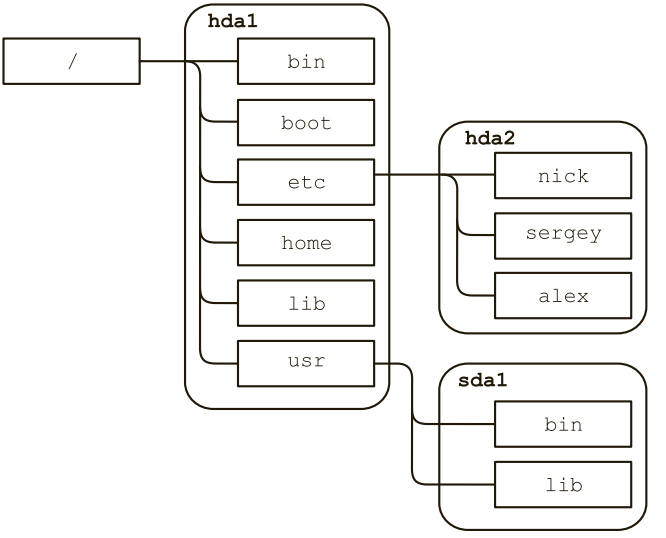

Рис. 6.2. Пример распределения структуры каталогов по физическим носителям

ровать. Только после этого файлы на диске станут доступными для использования.

Для монтирования дисков в UNIX-системах служит команда `mount`. Для ее использования необходимо указать следующие параметры: тип файловой системы, точку монтирования и имя файла устройства дискового накопителя. Файлы устройств накопителя относятся к типу блочных устройств и находятся в каталоге `/dev`.

Правила именования устройств для ОС Linux (без установленных расширений `devfs` и подобных) следующие:

- IDE-накопители имеют имена, начинающиеся на `hd`, SCSI-накопители — начинающиеся на `sd`;
- для IDE-накопителей указывается шина и подключение устройства, т.е. `hda` соответствует диску Primary Master, `hdb` — Primary Slave, `hdc` — Secondary Master, `hdd` — Secondary Slave. Для SCSI-накопителей указывается номер устройства на контроллере, т.е. `sda` — первое устройство на контроллере, `sdb` — второе и т.д.;
- поскольку монтируются обычно не накопители, а разделы на них, то разделы указываются после имени накопителя в виде чисел. При этом числа 1—4 задают первичные разделы, а число 5 и далее — логические диски во вторичных разделах.

Например, файл устройства для второго логического диска на IDE-устройстве, подключенном к порту Primary Master, будет иметь имя `/dev/hda6`. При наличии только одного первичного раздела, доступного из DOS или Windows, это имя будет соответствовать букве диска `E:`.

Для монтирования CD-ROM диска, находящегося в приводе `/dev/hdc`, необходимо выполнить следующую команду:

```
mount –t iso9660 /dev/hdc /mnt/cdrom
```

где `iso9660` — тип файловой системы; `/dev/hdc` — имя устройства; `/mnt/cdrom` — точка монтирования.

После окончания работы с устройством его можно размонтировать, т.е. удалить из файловой системы. Это производится командой `umount`, в качестве параметра которой указывается файл устройства или точка монтирования. Так, чтобы размонтировать CD-ROM, смонтированный в предыдущем примере, необходимо выполнить команду

```
umount /dev/hdc
```

или

```
umount /mnt/cdrom
```

Для автоматического монтирования разделов дисков при загрузке ОС служит файл `/etc/fstab`, в котором определяются

накопители, которые будут смонтированы, и порядок монтирования.

Файл имеет следующий формат:

```
/dev/hda2 /    ext2 defaults 0 1
/dev/hdc  /mnt/cdrom auto noauto,ro 0 0
```

В первой колонке файла указывается имя устройства, во второй — точка монтирования, в третьей — тип файловой системы, в четвертой — параметры монтирования. Параметр `defaults` вызывает монтирование по умолчанию, `noauto` отменяет монтирование файловой системы при загрузке, `ro` запрещает запись на носитель. Последние две колонки управляют режимом проверки поверхности носителя при загрузке.

Если дисковый накопитель указан в файле `/etc/fstab`, то для его монтирования достаточно указать только точку монтирования в качестве параметров команды `mount`, остальные параметры будут считаны из файла.

6.4. Принципы организации файловых систем в среде UNIX

Для хранения наборов данных на диске в UNIX-системах используется следующий метод — каждый набор данных, хранимый на диске, разбивается на блоки (размер одного блока обычно кратен размеру сектора диска). Для обеспечения целостности данных служебные блоки содержат ссылки на блоки, входящие в него. Эти служебные блоки организованы в древовидную структуру, т.е. каждый блок ссылается на другие служебные блоки и блоки данных пользователя (рис. 6.3).

Корнем дерева является индексный блок (`i-node`), в котором хранятся атрибуты файлов (дата модификации, дата последнего обращения, права доступа, тип файла и т.п.) и массив ссылок на блоки данных.

Массив ссылок имеет ограниченный размер и в случае, если количество блоков превосходит размер массива, создается одинарный ссылочный блок, на который и начинает ссылаться один из элементов `i-node`. Сам одинарный ссылочный блок ссылается на блоки данных. Поскольку размер массива ссылок в ссылочном блоке также ограничен, то в случае заполнения всех ссылок в индексном блоке и одинарном ссылочном блоке создаются двойные ссылочные блоки. Ссылки на существующие одинарные блоки переносятся в двойные, а индексный блок начинает ссылаться на двойные блоки. Таким образом растет глубина дерева ссылок. Если и двойных ссылочных блоков не хватает для всех

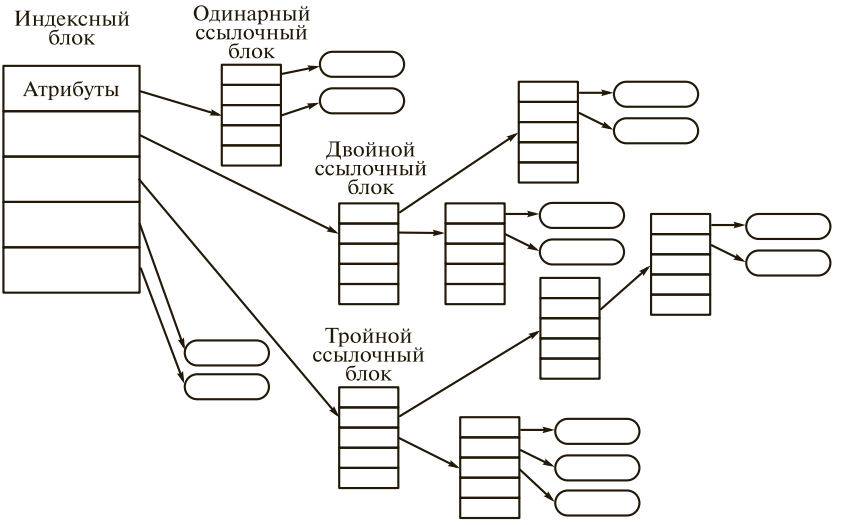

Рис. 6.3. Структура наборов данных в типичной файловой системе UNIX

блоков данных, входящих в набор, то создаются тройные ссылочные блоки. Ссылочных блоков более глубоких уровней вложенности не существует, поскольку временны́е затраты на поиск блоков данных в таких файлах становятся слишком большими [11].

6.5. Файловая система NTFS

Файловая система NTFS была разработана фирмой Microsoft в начале 90-х гг. XX в. Эта система во многом являлась потомком файловой системы HPFS, которая использовалась в новаторской для своего времени операционной системе OS/2. Вначале ОС/2 разрабатывалась совместно фирмами Microsoft и IBM. Но затем фирма Microsoft решила продвигать свою собственную ОС Windows и на основе разработок файловой системы HPFS для нее была спроектирована новая файловая система.

Ранние версии Windows, основанные на ядрах Windows 3.xx и Windows 9x, поддерживали работу только с файловыми системами FAT. Хотя существует множество модификаций FAT и их отличает высокая скорость работы с большим количеством относительно небольших файлов, все они уступают файловым системам, используемым в UNIX/Linux-системах.

Основные проблемы, связанные с файловыми системами семейства FAT, обусловлены тем, что:

• они чрезвычайно подвержены фрагментации данных, снижающей общую производительность файловой системы;

• они перерасходуют дисковое пространство для достаточно больших объемов логических разделов;

• они не поддерживают механизмы восстановления от ошибок, позволяющие восстанавливать работоспособность файловой системы в случае возникновения сбоев или внезапной перезагрузки;

• отсутствие в них поддержки POSIX препятствует использованию ОС Windows в ряде организаций и компаний;

• отсутствие в них средств разделения данных между пользователями и средств управления правами доступа.

Файловая система NTFS разрабатывалась с учетом присущих файловым системам FAT недостатков. Основные свойства, присущие данной файловой системе:

• надежность и восстанавливаемость. NTFS спроектирована таким образом, что операции ввода/вывода представляют собой транзакции. Эти транзакции неделимы, т. е. требуемая операция завершается либо полностью, либо не выполняется вообще. Это позволяет безболезненно совершать операции отката состояния файловой системы в случае возникновения сбоя;

• безопасность и контроль доступа к данным. Файловая система NTFS трактует файлы и каталоги как защищенные объекты в соответствии с общей архитектурой безопасности Windows. Это позволяет ограничивать доступ к данным определенным пользователям или группам пользователей;

• эффективное распределение дискового пространства. Файловая система NTFS поддерживает целый ряд механизмов управления дисковым пространством, такие, как поддержка сжатия каталогов и дисков, поддержка работы с разреженными файлами, наличие специализированного API для дефрагментации данных;

• поддержка POSIX, жестких ссылок, длинных имен, шифрования данных, поддержка работы с логическими томами размером больше 4 Гбайт.

Файловая система NTFS содержит несколько системных областей и областей хранения данных. Общая структура файловой системы NTFS показана на рис. 6.4.

Одной из основных системных областей является область Master File Table (MFT). Эта область содержит информацию обо всех

Рис. 6.4. Структура файловой системы NTFS

файлах в системе, включая и саму MFT. Когда создается новый файл или какой-либо файл удаляется из системы, то соответствующие изменения производятся и в MFT.

Первые 16 записей в таблице MFT зарезервированы для хранения информации о служебных файлах. Первая запись в таблице предназначена для хранения информации о самой MFT. Вторая запись содержит информацию о копии таблицы MFT, которая обычно находится в середине диска и используется для восстановления основной MFT, если та оказалась повреждена. Третья запись в таблице MFT представляет собой системный журнал, используемый для восстановления файлов. Записи с 4-й по 16-ю хранят информацию о прочих служебных файлах, а 17-я запись и далее используются для хранения данных об обычных файлах.

К служебным файлам относятся и так называемые файлы метаданных. Файлы метаданных автоматически создаются при форматировании файловой системы. Основные файлы метаданных и их описание приведены в табл. 6.1.

Таблица 6.1. **Файлы метаданных NTFS**

Файл метаданных	Описание
`$MFT`	Представляет Master File Table
`$MFTmirr`	Копия первых 16 записей MFT, расположенная в середине диска
`$LogFile`	Системный журнал файловых операций для восстановления файловой системы
`$Volume`	Данные о томе данных — метка, версия файловой системы и т. д.
`$AttrDef`	Список стандартных атрибутов файлов
`$.`	Корневой каталог
`$Bitmap`	Карта свободного места в томе
`$Boot`	Загрузочный сектор
`$Quota`	Файл, хранящий данные об ограничениях на дисковое пространство для пользователей
`$Upcase`	Таблица соответствия заглавных и прописных символов в именах файлов. Связана с поддержкой Unicode в именах файлов

Одной из интересных особенностей файловой системы NTFS является поддержка потоков данных в файлах. Причем с файлом может быть ассоциирован не один поток данных, а несколько. Это позволяет хранить в файле не только основные данные, но и приписывать дополнительные данные в альтернативные потоки данных, например информацию об авторе файла, правах на копирование и т.д. Каждый поток, связанный с файлом, может быть открыт независимо. При этом все потоки, связанные с одним и тем же файлом, имеют одни и те атрибуты.

В качестве примера использования потоков рассмотрим следующий фрагмент:

```
C:\>echo data > data.txt
C:\>echo data1 > data.txt:stream1
C:\>echo data2 > data.txt:stream2
C:\>more < data.txt
data
C:\>more < data.txt:stream1
data1
C:\>more < data.txt:stream2
data2
```

В этом примере видно, что хотя основной поток данных файла `data.txt` содержит внутри сообщение `data`, то два альтернативных потока данных содержат свои собственные данные. Однако в файловой системе все эти потоки будут представлять собой один единственный файл.

Автоматическое сжатие файлов, каталогов и отдельных томов файловой системы также является отличительной особенностью файловой системы NTFS. Сжатые файлы могут изменяться/читаться любыми приложениями Windows как если бы не были сжаты, т.е. сжатие и распаковка файлов осуществляются в масштабе реального времени. Однако следует отметить, что процесс работы со сжатыми файлами, каталогами и томами обычно занимает больше времени, чем с не сжатыми.

Также следует упомянуть и еще один механизм, поддержка которого была внедрена в файловой системе NTFS — работу с жесткими ссылками. Этот механизм внедрен как часть общего механизма поддержки стандарта POSIX и позволяет создавать разные файлы, ссылающиеся на одни и те же данные. При этом сами данные не дублируются, что позволяет экономить дисковое пространство. При удалении одной из жестких ссылок не происходит удаления самих данных. Полностью же данные удаляются только в том случае, если удаляется последняя жесткая ссылка, связанная с этими данными.

Для создания жестких ссылок в Windows можно использовать команду `fsutil` с параметрами `hardlink create`. Данная команда позволяет создавать жесткую ссылку для существующего файла.

Общий формат вызова данной команды для создания жесткой ссылки выглядит следующим образом:

```
fsutil hardlink create имя_жесткой_ссылки
имя_существующего_файла
```

В целом файловая система NTFS представляет собой одну из мощных и надежных файловых систем, обладающую целым рядом свойств, упрощающих и повышающих надежность обработки и хранения данных на дисковых накопителях. Корпорация Microsoft и в последующих ОС не собирается отказываться от использования этой файловой системы. Разрабатываемая файловая система WinFS, предназначенная для использования в будущих Windows-системах, не будет являться самостоятельной файловой системой. Она будет представлять собой реляционную базу данных, которая выступает в качестве надстройки над файловой системой NTFS.

Контрольные вопросы

1. В каких каталогах в UNIX-системах обычно размещаются исполняемые файлы?
2. В чем различия файлов символьных и блочных устройств?
3. Что такое точка монтирования?
4. Какие имена присваиваются в Linux IDE-накопителям?
5. Каким образом UNIX-система перемещает ссылки на блоки служебной информации в наборе данных при увеличении его размера?
6. Напишите задание на языке BASH, которое монтирует заданное устройство (1-й параметр) в заданный каталог (2-й параметр) только в том случае, если каталог не содержит файлов и других каталогов. В противном случае задание должно выдавать сообщение об ошибке.
7. Напишите задание на языке BASH, которое подсчитывает количество доменных гнезд в заданном каталоге и всех его подкаталогах.

Глава 7

ПОЛЬЗОВАТЕЛИ

7.1. Создание пользователей и групп

UNIX-системы являются многопользовательскими ОС. Даже в случае, если компьютером с установленной UNIX-системой пользуется один человек, для работы с системой необходима хотя бы одна учетная запись пользователя — учетная запись администратора `root`.

Выполнять обычную работу, используя учетную запись администратора, не рекомендуется по соображениям безопасности — в случае непродуманных действий системе можно нанести значительный урон. Чтобы этого избежать, работать можно из-под учетной записи, имеющей менее широкие права, чем администратор.

Для создания новой учетной записи пользователя используется команда `useradd`, выполнять которую можно только от лица пользователя root. В самой простой форме ее запуска

```
useradd <логин>
```

она создает в файле `/etc/passwd` новую учетную запись с первым незанятым UID и GID, домашним каталогом вида `/home/<логин>` и командным интерпретатором по умолчанию (как правило, `/bin/bash`).

Чтобы явно задать UID, GID, домашний каталог и командный интерпретатор, можно воспользоваться расширенной формой команды:

```
useradd –u <UID> -g <GID> -d <каталог> -s
<интерпретатор> <логин>
```

Например, для создания учетной записи пользователя `vasya` с `UID = 10001`, `GID = 200`, домашним каталогом `/home2/vasya` и командным интерпретатором по умолчанию `/bin/zsh` необходимо выполнить команду

```
useradd –u 10001 –g 200 –d /home2/vasya –s
/bin/zsh vasya
```

Непосредственно после создания учетной записи пользователь не имеет права входа в систему. Чтобы получить это право, для учетной записи необходимо определить пароль при помощи команды `passwd <логин>`. В ответ на вызов этой команды будет выведен запрос на ввод пароля и его повторный ввод для проверки. Будучи запущенной без параметров, команда `passwd` будет изменять пароль текущего пользователя и вначале запросит старый пароль.

Для добавления группы служит команда `groupadd`, которая имеет следующий формат запуска:

```
groupadd –g <GID> <имя группы>
```

т.е. для создания группы с именем `powerusers` и `GID = 200` достаточно выполнить

```
groupadd –g 200 powerusers
```

7.2. Файлы инициализации сеанса пользователя

В начале инициации сеанса пользователя (после его успешного входа в систему, но до момента появления приглашения командной строки) выполняются задания, создающие начальное информационное окружение пользователя. Эти задания могут, например, устанавливать значения переменных окружения, определять режим работы терминала пользователя, монтировать диски.

Имя файла инициализации сеанса зависит от применяемого командного интерпретатора, поэтому будем предполагать, что используется командный интерпретатор BASH.

Существует два файла инициализации сеанса — общесистемный, в котором содержится задание, выполняемое в начале сеанса любого пользователя системы, и пользовательский файл инициализации, содержащий задания, специфичные для каждого отдельного файла.

Общесистемный файл инициализации сеанса имеет имя `/etc/profile` и доступен для чтения всем пользователям. Изменять содержимое этого файла может только администратор системы. Обычно этот файл определяет начальные установки терминала, а также переменные окружения, задающие пути к исполняемым файлам и динамически загружаемым библиотекам.

Пример фрагмента такого файла приведен ниже:

```
export PATH=/bin:/sbin:/usr/bin:/usr/sbin
export LD_LIBRARY_PATH=/lib:/usr/lib:/usr/local/lib
export PS1=”$”
```

В этом файле при помощи установки значения переменной LD_LIBRARY_PATH устанавливаются пути поиска динамических библиотек, а переменная PS1 задает вид приглашения командного интерпретатора.

Пользовательский файл инициализации сеанса имеет имя .profile или .bash_profile и находится в домашнем каталоге пользователя. Этот файл обычно служит для установки путей к программам пользователя в переменной PATH и установки значений других переменных окружения, для определения текстового редактора по умолчанию, для определения алиасов команд — коротких имен команд с наиболее часто используемыми параметрами:

```
if [ -d ~/bin ] ; then
  PATH="~/bin:${PATH}"
fi
ENV=$HOME/.bashrc
USERNAME="admin"
export ENV USERNAME
PATH=$PATH:/usr/local/spice/bin:.
export EDITOR=le
alias ls='ls --color=auto'
```

Так, приведенный выше фрагмент файла .profile добавляет к переменной PATH каталог ~/bin, в который пользователь может поместить свои исполняемые файлы. В случае, если такой каталог отсутствует — добавления не происходит. После этого устанавливаются значения переменных ENV и USERNAME, к переменной PATH добавляется путь /usr/local/spice/bin и текущий каталог ., устанавливается имя текстового редактора по умолчанию. Последней строкой задается алиас для команды ls. После его установки при вводе ls командный интерпретатор будет воспринимать ее так, как будто введена команда ls --color=auto. Параметр --color=auto определяет, что в случае возможности вывода цвета на терминал, имена файлов и каталогов будут выделяться различными цветами в зависимости от их типа.

Контрольные вопросы

1. Почему не рекомендуется выполнять повседневную работу под учетной записью администратора?
2. Кто имеет право создания новых пользователей и групп?
3. Для чего используются файлы инициализации сеанса?
4. Какие файлы инициализации сеанса обычно используются в UNIX-системах?
5. Для чего используются алиасы команд? Как они устанавливаются?

6. Напишите пользовательский файл инициализации сеанса, который после входа пользователя в систему выводит приветствие, выводит значение переменной PATH и архивирует домашний каталог пользователя в файл `home_archive.tar`, находящийся в домашнем каталоге (параметры команды архивирования `tar` — см. приложение 3).

7. Напишите задание очистки домашнего каталога пользователя от временны́х файлов (оканчивающихся на символ ~) и пользовательский файл инициализации сеанса, вызывающий это задание.

Глава 8

ПРОЦЕССЫ

8.1. Основные понятия

В общем случае программа представляет собой набор инструкций процессора, представленный в виде файла. Чтобы программа могла быть запущена на выполнение, ОС должна сначала создать окружение или среду выполнения задачи, включающую в себя ресурсы памяти, возможность доступа к системе ввода/вывода и т. п. Совокупность окружения и области памяти, содержащей код и данные исполняемой программы, называется *процессом*. Процесс в ходе своей работы может находиться в различных состояниях (см. подразд. 8.3), в каждом из которых он особым образом использует ресурсы, предоставляемые ему ОС.

Два основных состояния процесса — это выполнение либо в режиме задачи, либо в режиме ядра. В первом случае происходит выполнение программного кода процесса, а во втором — системных вызовов, находящихся в адресном пространстве ядра.

Для управления процессами ОС использует системные данные, которые существуют в течение всего времени выполнения процесса. Вся совокупность этих данных образует *контекст процесса*, в котором он выполняется. Контекст процесса определяет состояние процесса в заданный момент времени.

С точки зрения структур, поддерживаемых ядром ОС, контекст процесса включает в себя следующие составляющие [15]:

- *пользовательский контекст* — содержимое памяти кода процесса, данных, стека, разделяемой памяти, буферов ввода/вывода;
- *регистровый контекст* — содержимое аппаратных регистров (регистр счетчика команд, регистр состояния процессора, регистр указателя стека и регистры общего назначения);
- *контекст системного уровня* — структуры данных ядра, характеризующие процесс. Контекст системного уровня состоит из статической и динамической части.

В статическую часть входят дескриптор процесса и пользовательская область (U-область).

Дескриптор процесса включает в себя системные данные, используемые ОС для идентификации процесса. Эти данные используются при построении таблицы процессов, содержащей информацию обо всех выполняемых в текущий момент времени процессах.

Дескриптор процесса содержит следующую информацию:

• *расположение и занимаемый процессом объем памяти* — обычно указывается в виде базового адреса и размера при непрерывном распределении процесса в памяти или списка начальных адресов и размеров блоков памяти, если процесс располагается в памяти несколькими фрагментами;

• *идентификатор процесса PID (Process IDentifier)* — уникальное целое число, находящееся обычно в диапазоне от 1 до 65 535, которое присваивается процессу в момент его создания;

• *идентификатор родительского процесса PPID (Parent Process IDentifier)* — идентификатор процесса, породившего данный. Все процессы в UNIX-системах порождаются другими процессами, например, при запуске программы на исполнение из командного интерпретатора ее процесс считается порожденным от процесса командного интерпретатора;

• *приоритет процесса* — число, определяющее относительное количество процессорного времени, которое может использовать процесс. Процессам с более высоким приоритетом управление передается чаще;

• *реальный идентификатор пользователя и группы*, запустивших процесс.

U-область содержит следующую информацию:

• указатель на дескриптор процесса;

• счетчик времени, в течение которого процесс выполнялся (т.е. использовал процессорное время) в режиме пользователя и режиме ядра;

• параметры последнего системного вызова;

• результаты последнего системного вызова;

• таблица дескрипторов открытых файлов;

• максимальные размеры адресного пространства, занимаемого процессом;

• максимальные размеры файлов, которые может создавать процесс.

Динамическая часть контекста системного уровня — один или несколько стеков, которые используются процессом при его выполнении в режиме ядра.

Для просмотра таблицы процессов может быть использована команда `ps`. Запущенная без параметров командной строки, она выведет все процессы, запущенные текущим пользователем. Достаточно полную для практического использования информацию

о таблице процессов можно получить, вызвав ps с параметрами aux, при этом будет выведена информация о процессах всех пользователей (a), часть данных, входящих в дескриптор и контекст процесса (u), а также будут выведены процессы, для которых не определен терминал (x):

```
$ ps aux
USER  PID  %CPU %MEM  VSZ  RSS  TTY      STAT START TIME COMMAND
root    1   0.0  0.0 1324  388  ?          S   Jul06 0:25 init[3]
root    2   0.0  0.0    0    0  ?          SW  Jul06 0:00 [eventd]
lp    594   0.0  0.0 2512  268  ?          S   Jul06 0:00  lpd
nick  891   0.1  0.1 2341  562 /dev/tty1 S   Jul06 0:18  bash
```

В этом примере команда ps aux выводит следующую информацию о процессах: имя пользователя, от лица которого запущен процесс (USER), идентификатор процесса (PID), процент процессорного времени, используемого процессом в данный момент времени (%CPU), процент занимаемой памяти (%MEM), общий объем памяти в килобайтах, занимаемый процессом (VSZ), объем постоянно занимаемой процессом памяти, которая может быть освобождена только по его завершению (RSS), файл терминала (TTY), состояние процесса (см. подразд. 8.3), дату старта процесса (START), количество используемого процессорного времени (TIME), полную строку запуска программы (COMMAND).

При работе процесса ему предоставляется доступ к ресурсам ОС — оперативной памяти, файлам, процессорному времени. Для распределения ресурсов между процессами и управления доступом к ресурсам в состав ядра ОС входит планировщик задач, а также используются механизмы защиты памяти и блокировки файлов и устройств.

Основная функция планировщика задач — балансировка нагрузки на систему между процессами, распределение процессорного времени согласно приоритету процессов.

Механизм защиты памяти запрещает доступ процесса к области оперативной памяти, занятой другими процессами (за исключением случая межпроцессного взаимодействия с использованием общей памяти).

Механизм блокировки файлов и устройств работает по принципу уникального доступа — если какой-либо процесс открывает файл на запись, то на этот файл ставится блокировка, исключающая запись в этот файл данных другим процессом.

8.2. Создание процесса. Наследование свойств

Запуск нового процесса в ОС UNIX возможен только другим, уже выполняющимся процессом. Запущенный процесс при этом

называется *процессом-потомком*, а запускающий процесс — *родительским процессом*. Процесс-потомок хранит информацию о родительском процессе в своем дескрипторе. Единственный процесс, не имеющий родительского процесса — это головной процесс ядра ОС — процесс `init`, имеющий `PID = 1`. Запуск этого процесса происходит при начальной загрузке ядра ОС.

В ОС UNIX существуют механизмы для создания процесса и для запуска новой программы. Для этого используется системный вызов `fork()`, создающий новый процесс, который является почти точной копией родительского:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

Между процессом-потомком и процессом-родителем существуют следующие различия [8, 15]:

- потомку присваивается уникальный идентификатор `PID`, отличный от родительского;
- значение `PPID` процесса-потомка устанавливается в значение `PID` родительского процесса;
- процесс-потомок получает собственную таблицу файловых дескрипторов, т.е. файлы, открытые родителем, не являются таковыми для потомка;
- для процесса-потомка очищаются все ожидающие доставки сигналы (см. подразд. 9.3);
- временна́я статистика выполнения процесса-потомка в таблицах ОС обнуляется;
- блокировки памяти и записи, установленные в родителе, не наследуются.

При этом процесс наследует следующие свойства:

- аргументы командной строки программы;
- переменные окружения;
- реальный (UID) и эффективный (EUID) идентификаторы пользователя, запустившего процесс;
- реальный (GID) и эффективный (EGID) идентификатор группы, запустившей процесс;
- приоритет;
- установки обработчиков сигналов (см. подразд. 9.3).

Функция `fork()` возвращает значение `−1` в случае, если порождение процесса-потомка окончилось неудачей. Причиной этого может служить одна из следующих ситуаций:

- ограничения максимального числа процессов для текущего пользователя, накладываемые командной оболочкой. Текущие ограничения, например, при использовании оболочки BASH можно узнать или изменить с помощью команды `ulimit −u`;

• переполнение максимально допустимого количества дочерних процессов. Узнать ограничения, накладываемые системой на максимальное количество дочерних процессов можно с помощью команды `getconf CHILD_MAX`;

• переполнение максимального числа процессов в системе. Это ограничение накладывается самой системой и узнать его можно с помощью команды `sysctl fs.file-max` или просмотрев содержимое файла `/proc/sys/fs/file-max`;

• исчерпана виртуальная память, определяемая размерами оперативной памяти и раздела подкачки.

В случае успешного выполнения функции `fork()` она возвращает PID процесса-потомка родительскому процессу и 0 процессу-потомку. Для хранения идентификаторов процессов используется тип переменной `pid_t`. В большинстве систем этот тип идентичен типу `int`, но непосредственное использование типа `int` не рекомендуется из соображений обеспечения совместимости с будущими версиями UNIX.

Для получения значения PID и PPID используются функции `getpid()` и `getppid()` соответственно. Тип возвращаемого ими значения — `pid_t`:

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

Поскольку выполнение обоих процессов — и родителя, и потомка — после вызова функции `fork()` продолжается со следующей за ней команды, необходимо разграничивать программный код, выполняемый каждым из этих процессов. Самый очевидный способ для этого — выполнять нужные участки кода в разных ветках условия, проверяющего код возврата функции `fork()`. Ставший уже классическим пример такой проверки приведен ниже:

```
#include <sys/types.h>
#include <unistd.h
int main(void)
{
 pid_t pid;
 switch (pid = fork())
 {
  case -1:
   /* Неудачное выполнение fork() — pid равен -1 */
   perror("Unsuccessful fork() execution\n");
   break;
```

```
   case 0:
    /* Тело дочернего процесса */
    /* pid = 0 — это дочерний процесс. */
    /* В нем значение pid инициализируется нулем */
    sleep(1);
    printf("CHILD: Child spawned with PID = %d\n",
      getpid());
    printf("CHILD: Parent PID = %d\n", getppid());

    /* Завершение работы дочернего процесса */
    _exit(0);
   default:
    /* Тело процесса-родителя */
    /* так как pid>0, выполняется родительский */
    /* процесс, получивший pid потомка */
    printf("PARENT:Child spawned with PID=%d\n",
       pid);
    printf("PARENT:Parent PID=%d\n", getpid());
  }
  /* Тело процесса-родителя после обработки fork() */
  /* Если бы в case 0 не был указан _exit(0), то */
  /* потомок бы тоже выполнил идущие следом команды */
exit(0);
}
```

Программа выведет следующие строки (идентификаторы процессов могут различаться):

```
PARENT:Child spawned with PID=2380
PARENT:Parent PID=2368
CHILD: Child spawned with PID = 2380
CHILD: Parent PID = 1
```

В приведенном примере программы необходимо обратить внимание на следующий момент: если не обеспечить выход из процесса-потомка внутри соответствующего варианта `case`, то потомок, закончив выполнение программного кода внутри `case`, продолжит выполнение кода, находящегося после закрывающей скобки конструкции `switch()`. В большинстве случаев такое поведение необходимо пресекать. Для явного выхода из процесса с заданным кодом возврата используется функция `_exit(<код возврата>)`. Вызов этой функции вызывает уничтожение процесса и выполнение следующих действий:

- отключаются все сигналы, ожидающие доставки (см. подразд. 9.3);
- закрываются все открытые файлы;

• статистика использования ресурсов сохраняется в файловой системе proc;

• извещается процесс-родитель и переставляется PPID у потомков;

• состояние процесса переводится в «зомби» (см. подразд. 8.3).

Другие способы завершения работы процесса рассмотрены в следующих разделах.

Для завершения процесса-потомка рекомендуется использовать именно функцию _exit() вместо функции exit(). Функция _exit() очищает структуры ядра, связанные с процессом — дескриптор и контекст процесса. В отличие от нее функция exit() в дополнение к указанным действиям устанавливает все структуры данных пользователя в начальное состояние. В результате этого может возникнуть ситуация, в которой структуры данных процесса-родителя будут повреждены. Функция exit() может быть вызвана только в функциях, выполняемых процессом-родителем.

В большинстве случаев функция fork() используется совместно с одной из функций семейства exec...(). При последовательном вызове этих функций возможно создание нового процесса и запуск из него новой программы.

В результате выполнения функции exec...() программный код и данные процесса заменяются на программный код запускаемой программы. Неизменными остаются только идентификатор процесса PID и идентификатор родительского процесса PPID.

Если выполнение функции семейства exec...() завершилось неуспешно, это может быть обусловлено следующими причинами:

• путь к исполняемому файлу превышает значение системного параметра PATH_MAX, элемент пути к файлу превышает значение системного параметра NAME_MAX или в процессе разрешения имени файла выяснилось, что число символических ссылок в пути превысило число SYMLOOP_MAX. Узнать значения этих параметров можно с помощью команды getconf –a;

• файл, для которого вызывается функция, не является исполняемым, не является обычным файлом или не существует.

Список параметров, передаваемых в запускаемый процесс, слишком большой. Максимально допустимое количество параметров вычисляется на основе максимально допустимой длины командной строки. Согласно стандарту POSIX данное значение должно быть не меньше 4 096. В реальных системах обычно это число больше на несколько порядков. Чтобы узнать это ограничение для системы, можно использовать команду getconf ARG_MAX. Параметр ARG_MAX и содержит максимально допустимое количество символов в командной строке. В реальности накладываются дополнительные ограничения на максимальную длину

выполняемой команды. Согласно стандарту POSIX, максимально допустимую длину выполняемой команды можно вычислить с помощью команды `expr `getconf ARG_MAX` - `env|wc -c` - 2048`. Кроме того, системой накладываются и ограничения на длину одного параметра.

В приведенном ниже примере процесс-родитель порождает новый процесс, который запускает на выполнение программу `/bin/ls` с параметром `-l`. Поскольку происходит полная замена кода процесса-потомка, то вызывать функцию `_exit()` не обязательно:

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

int main(void)
{
  pid_t pid;
  int status;
  if (fork() == 0)
  {

    /* Тело потомка */
    execl("/bin/ls", "/bin/ls", "-l", 0);
  }
  /* Продолжение тела родителя */
  wait(&status);
  printf("Child return code %d\n", WEXITSTATUS(status));
  return 0;
}
```

Программа выведет следующее:

```
total 17
-rwxr-xr-x  1 nick users 9763 Oct 11 15:41 a.out
-rwxrwxrwx  1 nick users 986 Oct 11 15:20 fork1.c
-rwxrwxrwx  1 nick users 321 Oct 11 15:40 fork2.c
Child return code 0
```

Процесс-родитель после создания потомка ожидает его завершения при помощи функции `wait()`:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

Эта функция ожидает окончания выполнения процесса-потомка (если их было порождено несколько — любого из них). Возвра-

щаемое функцией значение — PID завершившегося процесса. Передаваемое функции по ссылке значение статуса представляет собой число, кодирующее информацию о коде возврата потомка и его состоянии на момент завершения. Для просмотра интересующей информации необходимо воспользоваться одним из макроопределений `WEXIT…()`. Например, макроопределение `WEXISTATUS()` возвращает номер кода возврата, содержащегося в значении статуса.

Если процесс порождает множество потомков и возникает необходимость в ожидании завершения конкретного потомка, используется функция `waitpid()`:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
```

Первый аргумент этой функции — `pid` процесса, завершения которого мы ожидаем, второй — значение статуса. Третий параметр определяет режим работы функции.

Если третий параметр равен 0, то выполнение процесса приостанавливается до тех пор, пока хотя бы один потомок не будет завершен. Если в качестве третьего параметра передается константа `WNOHANG`, то значение статуса присваивается только в том случае, если потомок уже завершил свое выполнение, в противном случае выполнение родителя продолжается. Если указан параметр `WNOHANG` и потомок еще не завершен, то функция `waitpid()` вернет 0:

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
void main(void)
{
  pid_t childPID;
  pid_t retPID = 0;
  int status;

  if ( (childPID = fork()) == 0)
  {
    /* Тело потомка */
    execl("/bin/ls", "/bin/ls", "-l", 0);
  }
  while (!retPID) /* Продолжение тела родителя */
  {
    retPID = waitpid(childPID, &status, WNOHANG);
  }
```

```
  printf("Child return code %d", WEXITSTATUS(status));
}
```

Программа выведет следующее:

```
total 17
-rwxr-xr-x   1 nick users 9763 Oct 11 15:41 a.out
-rwxrwxrwx   1 nick users 986 Oct 11 15:20 fork1.c
-rwxrwxrwx   1 nick users 321 Oct 11 15:40 fork2.c
Child return code 0
```

8.3. Состояния процесса. Жизненный цикл процесса

Между созданием процесса и его завершением процесс находится в различных состояниях в зависимости от наступления некоторых событий в ОС. Сразу же после создания процесса при помощи функции `fork()` он находится в состоянии «Создан» — запись в таблице процессов для него уже существует, но внутренние структуры данных процесса еще не инициализированы. Как только первоначальная инициализация процесса завершается, он переходит в состояние «Готов к запуску». В этом состоянии процессу доступны все необходимые ресурсы, кроме процессорного времени, и он находится в очереди задач, ожидающих выполнения. Как только процесс выбирается планировщиком, он переходит в состояние «Выполняется в режиме ядра», т. е. выполняет программный код ядра ОС, обрабатывающий последнее изменение состояния процесса.

Из этого состояния процесс может перейти в состояние «Выполняется в режиме задачи», в котором он будет выполнять уже свой собственный программный код. При каждом системном вызове процесс будет переходить в состояние «Выполняется в режиме ядра» и обратно (рис. 8.1).

Поскольку системные вызовы могут выполняться для получения доступа к определенным ресурсам, а ресурсы могут оказаться недоступными, то в этом случае из состояния «Выполняется в режиме ядра» процесс переходит в состояние «Ожидание», в котором он освобождает процессорное время и ожидает освобождения ресурса. После того, как ресурс становится доступным процессу, процесс захватывает его и переходит в состояние «Готов к запуску» и опять начинает ожидать выбора планировщиком процессов.

В состояниях «Выполняется в режиме ядра» и «Выполняется в режиме задачи» планировщик может передать управление другому процессу. При этом процесс, у которого забрали управление, будет переведен в состояние «Готов к запуску».

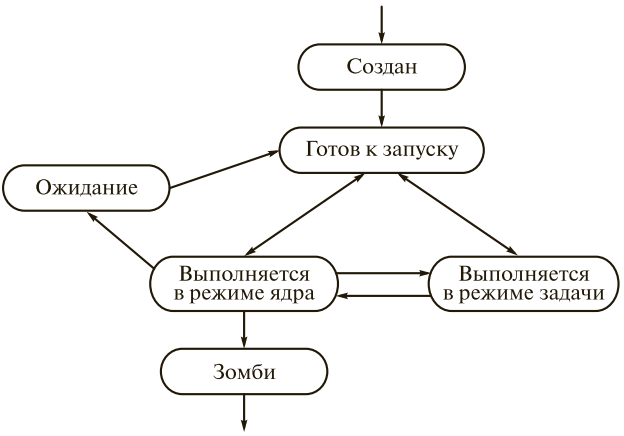

Рис. 8.1. Жизненный цикл процесса

При переключении активного процесса происходит переключение контекста, при котором ядро сохраняет контекст старого процесса и восстанавливает контекст процесса, которому передается управление. После завершения процесса он будет переведен в состояние «Зомби», т.е. память, занимаемая процессом, будет освобождена, а в таблице процессов останется информация о коде возврата процесса. Полностью завершить процесс можно при помощи вызова функции `wait()` процессом-родителем.

8.4. Терминал. Буферизация вывода

По умолчанию системная библиотека ввода/вывода передает данные на устройство терминала не сразу по выполнению системного вызова (например, `printf()`), а по мере накопления в специальной области памяти некоторого количества текста.

Такая область памяти получила название *буфера вывода*, а процесс накопления данных перед непосредственной их передачей на устройство называется *буферизацией*.

Для каждого процесса выделяется свой буфер ввода/вывода. Данные из него передаются на устройство по поступлению команды от ядра ОС или по заполнению буфера. Использование промежуточного буфера позволяет сильно сократить количество обращений к физическому устройству терминала и в целом ускоряет работу с ним. Системная библиотека использует три типа буферизации:

• *полная буферизация* — передача данных на физическое устройство терминала выполняется только после полного заполнения буфера;

• *построчная буферизация*, при которой передача данных на физическое устройство терминала производится после вывода одной строки текста (т.е. последовательности символов, оканчивающейся переводом строки или последовательности символов, длина которой равна ширине терминала);

• *отсутствие буферизации*, при котором данные сразу передаются на устройство, не накапливаясь в буфере.

При параллельном выводе текста несколькими процессами на терминал выводимый ими текст накапливается в отдельном буфере каждого процесса, а момент поступления данных на терминал определяется моментом посылки ОС команды сброса содержимого буферов на устройство. В результате неодинаковой длины выводимых разными процессами текстов и неравномерного предоставления им процессорного времени при включенной буферизации вывод таких параллельно выполняемых процессов может произвольно перемешиваться. То, каким образом данные будут перемешаны, зависит от момента передачи данных из буферов процессов на устройство терминала.

Например, на рис. 8.2 на линии времени показано последовательное изменение состояния буферов ввода/вывода двух процессов, последовательно выводящих на экран арабские цифры от 1 до 9 и латинские буквы от *A* до *F*.

Вывод каждого символа в этих процессах производится отдельным оператором `printf()`. Жирными стрелками вниз на ри-

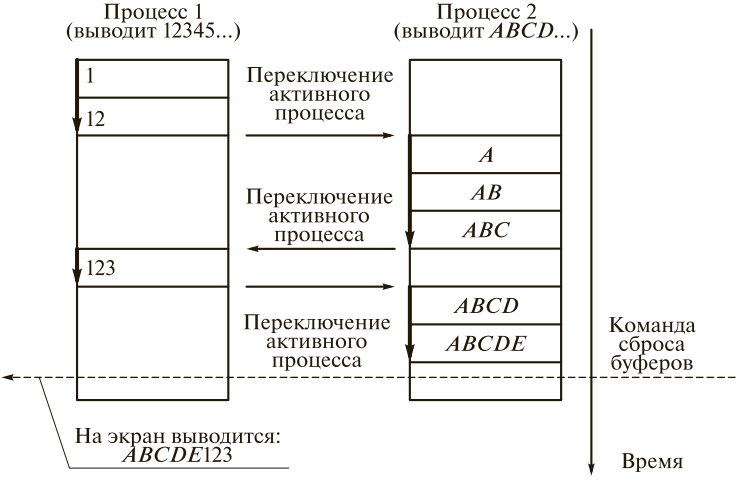

Рис. 8.2. Сброс буферов ввода/вывода параллельно выполняемых процессов:

— время выполнения процесса

сунке показано время, в течение которого процесс активен (использует процессорное время), высота каждого прямоугольника задает общее время жизни процесса.

Поскольку накопление данных в буферах происходит постепенно и между отдельными вызовами функции `printf()` может происходить переключение активного процесса, данные будут накапливаться в буферах неравномерно. Объем текста, выводимого на экран в момент поступления команды сброса буфера, будет зависеть от того, какой объем успел накопиться в буфере, а последовательность вывода фрагментов текста на экран будет определяться последовательностью сброса буферов на экран.

Для того чтобы избежать этой проблемы, можно отключить буферизацию. Это минимизирует задержку между выполнением команды вывода данных и реальным появлением текста на терминале.

Управление буферизацией производится функцией `setvbuf()`:

```
#include <stdio.h>
int setvbuf (FILE *stream, char *buf, int type, size_t
size);
```

Аргумент `stream` задает имя потока ввода/вывода, для которого изменяется режим буферизации, `buf` — задает указатель на область памяти, в которой хранится буфер, `type` — определяет тип буферизации и может принимать следующие значения:

`_IOFBF` — полная буферизация;
`_IOLBF` — построчная буферизация;
`_IONBF` — отсутствие буферизации.

Аргумент `size` задает размер буфера, на который ссылается указатель `buf`.

Для отключения буферизации стандартного потока вывода можно использовать буфер нулевого размера и вызвать функцию `setvbuf()` следующим образом:

```
setvbuf(stdout,(char*)NULL,_IONBF,0);
```

Рекомендуется вызывать эту функцию в начале работы любой программы, порождающей процессы, которые выводят данные на один и тот же терминал.

Контрольные вопросы

1. Как идентифицируется и какими параметрами характеризуется процесс в ОС?
2. Что происходит при выполнении функции `fork()`?
3. В чем различия системных вызовов `wait()` и `waitpid()`?

4. В каких случаях процесс находится в состоянии «Готов к запуску»?

5. Для чего нужно управление буферизацией при выводе данных несколькими процессами на один терминал?

6. Напишите программу, которая порождает 10 процессов-потомков, каждый из которых завершает свое выполнение с кодом возврата, равным номеру своего процесса. После завершения всех потомков программа должна выводить сумму кодов возврата всех потомков.

7. Напишите программу, которая последовательно порождает 10 процессов-потомков (т.е. каждый порожденный потомок порождает следующего потомка). При этом завершение выполнения потомков должно производиться в обратном порядке.

Глава 9

МЕЖПРОЦЕССНОЕ ВЗАИМОДЕЙСТВИЕ

9.1. Виды межпроцессного взаимодействия

Значительная часть сложных программ в настоящее время использует ту или иную форму межпроцессного взаимодействия. Это обусловлено естественной эволюцией подходов к проектированию программных систем, которая последовательно прошла три этапа [9].

1. Монолитные программы, содержащие в своем коде все необходимые для своей работы инструкции. Обмен данными внутри таких программ производится при помощи передачи параметров функций и использования глобальных переменных. При запуске таких программ образуется один процесс, который выполняет всю необходимую работу.

2. Модульные программы, которые состоят из отдельных программных модулей с четко определенными интерфейсами вызовов. Объединение модулей в программу может происходить либо на этапе сборки исполняемого файла (статическая сборка, или *static linking*), либо на этапе выполнения программы (динамическая сборка, или *dynamic linking*). Преимущество модульных программ заключается в достижении некоторого уровня универсальности — один модуль может быть заменен другим. Однако модульная программа все равно представляет собой один процесс, а данные, необходимые для решения задачи, передаются внутри процесса как параметры функций.

3. Программы, использующие межпроцессное взаимодействие. Такие программы образуют программный комплекс, предназначенный для решения общей задачи. Каждая запущенная программа образует один или более процессов. Каждый из процессов использует для решения задачи либо свои собственные данные и обменивается с другими процессами только результатом своей работы, либо работает с общей областью данных, разделяемых между разными процессами. Для решения особо сложных задач процессы могут быть запущены на разных физических компьютерах и взаимодействовать через сеть. Одно из преимуществ ис-

пользования межпроцессного взаимодействия заключается в еще большей универсальности — взаимодействующие процессы могут быть заменены независимо друг от друга при сохранении интерфейса взаимодействия. Другое преимущество состоит в том, что вычислительная нагрузка распределяется между процессами. Это позволяет ОС управлять приоритетами выполнения отдельных частей программного комплекса, выделяя большее или меньшее количество ресурсов ресурсоемким процессам.

Для выполнения многих процессов, решающих общую задачу, ОС необходимо обеспечить средства взаимодействия между ними. Предоставление средств взаимодействия — задача ОС, а не прикладных программ, поскольку необходимо исключить влияние прикладных программ на сами механизмы обмена, а, кроме того, поддержка механизмов обмена может потребовать доступа к ресурсам, недоступным обычным процессам пользователя.

Типичные механизмы взаимодействия между процессами предназначены для решения следующих задач:

- передача данных от одного процесса к другому;
- совместное использование одних и тех же данных несколькими процессами;
- извещения об изменении состояния процессов.

9.2. Механизмы межпроцессного взаимодействия

Современные ОС реализуют семь основных механизмов межпроцессного взаимодействия, перечисленных ниже. В этом подразделе приведена краткая характеристика каждого механизма, некоторые из них более подробно рассматриваются в следующих подразделах.

Прерывания. Изначально механизм прерываний в ОС использовался для оповещения. В определенные моменты времени аппаратное устройство извещает о наступлении некоторого события (например, о готовности к действию, о сбое, о завершении передачи блока данных). Количество вариантов таких событий может быть достаточно большим и все они должны быть различимы ОС. Такое извещение о готовности получило название прерывания, поскольку в момент получения прерывания ОС должна приостановить выполнение текущих задач и среагировать на поступившее прерывание.

Реакция на прерывание, как правило, заключается в выполнении некоторого программного кода, находящегося в памяти, адресуемой ОС. Операционная система поддерживает специальную область памяти, называемую *таблицей прерываний*, в которой каждому прерыванию (как правило, идентифицируемому по но-

меру) ставится в соответствие адрес памяти, по которому находится программный код — обработчик прерывания. Количество прерываний, поддерживаемых ОС, фиксировано. Обычно ОС поддерживает от 16 до 256 прерываний.

Обработчик прерываний независим от выполняемых процессов, однако этот программный код может быть переопределен одним из процессов — таким образом устанавливается пользовательский обработчик прерывания.

После выполнения обработчика ОС возвращает управление задачам или завершает выполнение одной или более задач, которые были активны до поступления прерывания.

Кроме аппаратных прерываний, поступающих от устройств, существуют программные прерывания, которые могут быть инициированы любым процессом. Таким образом, процесс может сообщить ОС о наступлении какого-либо события в ходе его выполнения.

Для каждого системного вызова ОС также активирует прерывание. Например, соответствующее прерывание активируется при выводе текста на экран терминала.

Сигналы. Механизм сигналов имеет много общих черт с механизмом программных прерываний. Он также предназначен для того, чтобы процессы могли быть оповещены о некоторых событиях. Основное отличие сигналов от прерываний состоит в том, что при помощи сигналов один процесс оповещает другие процессы, а не ОС. В отличие от прерывания сигнал должен иметь точку назначения — процесс-получатель. В ответ на получение сигнала процесс-получатель прерывает свое выполнение и начинает выполнять программный код — обработчик сигнала. После выполнения кода обработчика процесс продолжает свое выполнение.

Существенным отличием от прерываний здесь является то, что каждый процесс может иметь свой набор обработчиков сигналов, а память, в которой хранится программный код обработчика — это память процесса. Обычный обработчик сигнала — функция, которая определена в теле программы. В момент получения сигнала происходит вызов этой функции.

Операционная система поддерживает таблицу обработчиков сигналов для каждого процесса. В ней каждому сигналу ставится в соответствие адрес обработчика сигнала. Например, UNIX-подобные ОС поддерживают 16 и более различных сигналов. Подробнее механизм сигналов будет рассмотрен далее.

Сообщения. Механизм сообщений предназначен для обмена данными между процессами. Для этого создается специальная очередь сообщений, поддерживаемая ОС. В очереди накапливаются данные, которые записываются в нее процессами. Накопленные в очереди данные могут быть считаны другими процесса-

ми, таким образом произойдет передача данных от одного процесса к другому.

Операционная система поддерживает специальную область оперативной памяти, в которой хранит очереди. Обычно UNIX-подобная ОС поддерживает до 1024 очередей сообщений.

Первый процесс, использующий сообщения, должен создать очередь, а остальные процессы должны получить доступ к уже созданной очереди. В отличие от взаимодействия при помощи сигналов, в которых выполнение обработчика начинается сразу по получению сигнала, чтение данных из очереди производится процессом-получателем при помощи функции чтения данных из очереди. Процесс-отправитель должен следить за тем, чтобы очередь не переполнилась (например, если процесс-получатель давно не считывал из нее данные), поскольку это может вызвать потерю передаваемых данных.

Именованные каналы. Именованные каналы также предназначены для обмена данными между процессами. Однако в качестве очереди используется не область оперативной памяти, а файл специального вида. Процессы могут записывать данные в этот файл и считывать из него данные. При этом ОС поддерживает равноправный доступ к именованному каналу для всех процессов, использующих его для обмена данными.

Гнезда (сокеты). Механизм межпроцессного взаимодействия при помощи сокетов, в первую очередь, обеспечивает взаимодействие процессов, выполняемых на различных компьютерах. Каждый процесс создает гнездо, в которое может записывать данные и считывать их из него. Гнезда связаны друг с другом через сетевые протоколы.

Процессы, взаимодействующие друг с другом, обмениваются данными через связанные гнезда, а ядро системы передает данные процессов через сеть, как правило, используя стек протоколов TCP/IP и драйверы сетевых адаптеров.

Общая память. Механизм общей памяти заключается в том, что в адресное пространство двух и более процессов отображается один и тот же участок физической памяти, адресуемой ОС. Используя этот общий участок памяти, процессы могут обмениваться данными без участия ядра ОС, что значительно увеличивает скорость работы.

Семафоры. Одна из основных проблем при использовании общей памяти заключается в том, что необходимо предотвращать конфликтные ситуации одновременного доступа. Такая ситуация возникает, например, при одновременной записи данных в один участок памяти несколькими процессами. В этом случае память будет содержать данные последнего записывающего процесса, остальные данные будут потеряны.

Другая конфликтная ситуация возникает в момент чтения одним процессом данных из области памяти, модифицируемой другим процессом. В этом случае считанные данные могут содержать как старую, так и частично обновленную информацию.

Для предотвращения таких ситуаций применяется механизм семафоров — специальных флагов, указывающих на возможность использования того или иного участка памяти. В момент записи в память процесс ставит семафор, что означает, что память «занята», а по окончанию записи снимает его, что означает освобождение памяти.

Другие процессы перед записью или чтением данных должны проверять состояние семафора и специально обрабатывать ситуацию, когда запись невозможна. В этом случае процесс может либо приостановить свое выполнение до снятия семафора, либо накапливать данные в своем внутреннем буфере и записывать данные из буфера в момент снятия семафора.

Механизм семафоров удобен не только при использовании общей памяти, он может применяться при любом совместном использовании ресурсов.

9.3. Межпроцессное взаимодействие в среде UNIX

9.3.1. Сигналы

Основные понятия. Наиболее распространенный механизм межпроцессного взаимодействия, поддерживаемый большинством UNIX-систем — взаимодействие при помощи сигналов. Обмен сигналами между процессами (пользовательскими или системными) позволяет им обмениваться информацией о наступлении тех или иных событий, важных для выполняющегося процесса.

Инициатором посылки сигнала процессу может служить ядро ОС (например, в случае деления на 0 ядро посылает процессу сигнал, сообщающий об этой исключительной ситуации), пользователь (например, при прерывании выполнения процесса нажатием на клавиши Ctrl-C процессу посылается сигнал, соответствующий этой комбинации клавиш) или другой процесс.

В общем случае любой процесс может послать сигнал другому процессу, выполняемому параллельно с ним. Для посылки сигнала процесс-передатчик должен определить два параметра — PID процесса-получателя сигнала и номер передаваемого сигнала. UNIX-системы поддерживают ограниченное число типов сигналов, обычно около 30. В табл. 9.1 приведено краткое описание основных типов сигналов.

Каждому типу сигнала присваивается свой порядковый номер и мнемоническое обозначение вида SIGALRM, где SIG — общее обозначение мнемоник сигналов, ALRM — обозначение события, с возникновением которого обычно связана посылка данного типа сигнала.

Таблица 9.1. **Основные сигналы в UNIX**

Номер сигнала	Мнемоника	Описание	Действие по умолчанию
1	`SIGHUP`	Разрыв связи с текущим терминалом (перевод в режим фонового процесса)	Завершение процесса
2	`SIGINT`	Прерывание процесса (обычно генерируется сочетанием клавиш Ctrl-C)	То же
3	`SIGQUIT`	Выход из процесса (обычно генерируется сочетанием клавиш Ctrl-\)	Завершение процесса и создание файла `core` (содержащего состояние памяти процесса на момент выхода)
6	`SIGABRT`	Аварийное завершение процесса; может генерироваться функцией `abort()`	Завершение процесса и создание файла `core`
9	`SIGKILL`	Уничтожение процесса	Завершение процесса
11	`SIGSEGV`	Ошибка сегментации (обычно возникает при неверной работе с динамической памятью)	Завершение процесса и создание файла `core`
14	`SIGALRM`	Наступление таймаута таймера; может генерироваться функцией `alarm()`	Завершение процесса
15	`SIGTERM`	Завершение процесса	То же

Окончание табл. 9.1

Номер сигнала	Мнемоника	Описание	Действие по умолчанию
20	SIGCHLD	Посылается процессом-потомком при завершении родителю	Игнорируется
30	SIGUSR1	Пользовательский сигнал	Завершение процесса
31	SIGUSR2	То же	То же

Доставка сигнала до процесса-получателя выполняется ОС. В ответ на получение сигнала процесс-получатель может либо проигнорировать сигнал, либо начать выполнять некоторый программный код, связанный с получением сигнала данного типа. Этот код обычно оформляется в виде отдельных функций, вызываемых при получении сигнала определенного типа. Такие функции получили название *функций-обработчиков сигналов* (или просто *обработчиков сигналов*).

Вся совокупность обработчиков сигналов для каждого процесса образует его *таблицу обработчиков сигналов*. Эта таблица уникальна для каждого процесса. Количество строк в таблице обработчиков равно числу поддерживаемых ОС сигналов. В каждой строке таблицы записывается номер сигнала и адрес в памяти, с которого начинается программный код функции-обработчика — указатель на функцию-обработчик, имеющую следующий формат декларации:

```
void handler(int sig_num);
```

При получении сигнала процесс прерывает свое нормальное выполнение и начинает выполнять программный код функции-обработчика. По завершению выполнения обработчика управление возвращается на команду, следующую за той, при выполнении которой был получен сигнал (рис. 9.1).

Таблица обработчиков сигналов может не содержать адресов функций-обработчиков для некоторых (возможно даже для всех) типов сигналов. В этом случае говорят, что для этих типов сигналов не определен пользовательский обработчик. При получении процессом сигнала такого типа начинает выполняться программный код обработчика по умолчанию, предоставляемого операционной системой (рис. 9.2).

Для большинства типов сигналов обработчик по умолчанию завершает выполнение процесса, получившего сигнал (см. табл. 9.1).

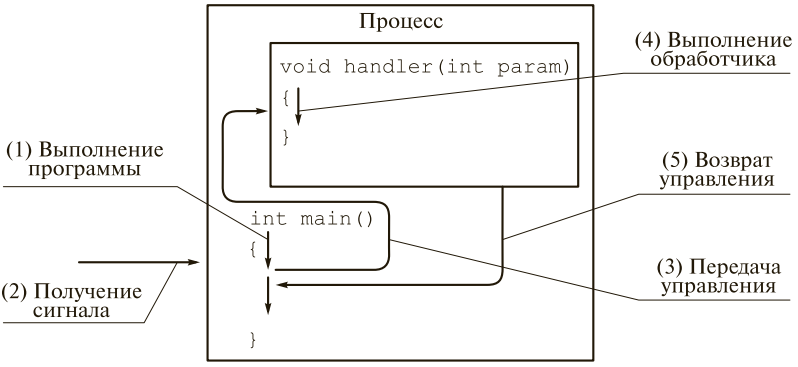

Рис. 9.1. Передача управления обработчику при получении сигнала

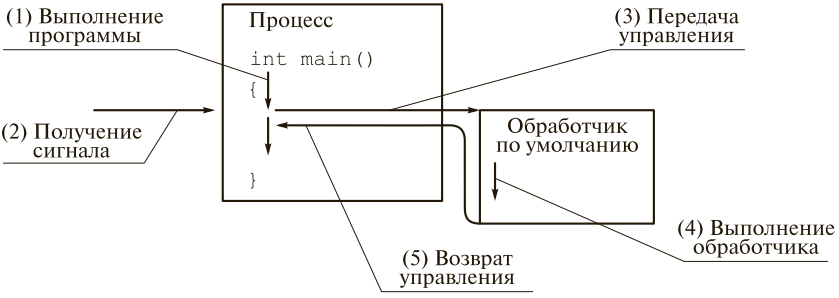

Рис. 9.2. Передача управления обработчику по умолчанию

Для некоторых типов сигналов пользовательский обработчик не может быть определен в принципе. К таким сигналам относятся сигналы, завершающие выполнение процесса: сигнал `SIGSTOP`, вызывающий «мягкое» завершение процесса, во время которого могут быть корректно сброшены все буферы ввода/вывода; и сигнал `SIGKILL`, вызывающий немедленное аварийное завершение работы процесса.

Сигналы в командном интерпретаторе BASH. Для посылки сигнала с известным PID при помощи командного интерпретатора BASH можно воспользоваться командой `kill`, формат вызова которой имеет следующий вид:

```
kill -<мнемоника или номер сигнала> <PID>
```

Например, для посылки сигнала `SIGKILL` процессу с `PID=1046` можно воспользоваться номером сигнала:

```
kill –9 1046
```

Для посылки сигнала SIGINT процессу с PID=1079 можно воспользоваться его мнемоникой:

```
kill –SIGINT 1079
```

При вызове команды kill можно не указывать номер или мнемонику посылаемого сигнала. При этом будет послан сигнал SIGTERM.

Для получения PID последнего запущенного из задания на языке BASH процесса используется системная переменная $!. Если необходимо послать сигнал процессу, PID которого неизвестен, но известно имя программы, в результате запуска которой был порожден процесс, можно воспользоваться командой killall. Эта команда посылает заданный в параметрах сигнал всем процессам, порожденным в результате запуска программы, имя которой также задается в параметрах запуска команды killall. Формат вызова команды следующий:

```
killall -<мнемоника или номер сигнала> <имя программы>
```

Например, для посылки сигнала SIGALRM процессу, порожденному запуском программы timer можно воспользоваться командой killall следующим образом:

```
killall –ALRM timer
```

При вызове команды killall также можно не указывать номер посылаемого сигнала, при этом по умолчанию будет послан сигнал SIGTERM.

Системные вызовы для работы с сигналами. В UNIX-системах определен интерфейс системных вызовов для работы с сигналами. Все прототипы функций и типов данных для работы с сигналами определены в заголовочном файле signal.h.

Для посылки сигналов в программах на языке C используется системный вызов kill():

```
#include <signal.h>
int kill(pid_t pid, int signal_num);
```

Здесь параметр signal_num задает имя посылаемого сигнала. Если параметр PID — положительное число, то сигнал посылается процессу с PID, равным значению параметра. Если PID = 0, то сигнал посылается всем процессам, EGID которых равен EGID процесса, посылающего сигнал. Если параметр PID = –1, то сигнал посылается всем процессам, EUID которых равен EUID процесса, посылающего сигнал. Если при этом EUID = 0, т.е. посылка сигналов происходит от лица суперпользователя root, то сигнал будет послан всем процессам, кроме процессов с PID = 0 и PID = 1. Если параметр PID — отрицательное число, то сигнал

будет послан во все процессы, EGID которых равен абсолютному значению параметра.

Следующая программа иллюстрирует использование функции `kill()` — программа порождает процесс-потомок, который принудительно завершает выполнение процесса-родителя при помощи сигнала `SIGTERM` и выводит сообщение об этом:

```
#include <unistd.h>
#include <process.h> /* В Linux не нужен */
#include <stdio.h>
#include <sys/types.h>
#include <signal.h>
int main()
{
  pid_t pid;
  setvbuf(stdout,(char*)NULL,_IONBF,0);

  switch (pid = fork() )
  {
    case –1:
      perror("Unsuccesful fork");
      _exit(1);
      break;
    case 0:
      printf("Parent process with PID = %d\n",
      getppid());
      kill( getppid(), SIGTERM);
      sleep(1);
      printf("Parent process terminated\n");
      printf("New parent process have PID = %d\n",
      getppid() );
      _exit(0);
      break;
    default:
      while (1); /* Бесконечный цикл */
        /* до получения SIGTERM */
  }
}
```

В этой программе используется функция `sleep()`, приостанавливающая выполнение процесса на число секунд, переданное ей в качестве параметра (в данном случае — на 1 с). Это сделано для того, чтобы у процесса-родителя хватило времени для завершения, только после которого происходит смена PPID у процесса-потомка. Более подробно функция `sleep()` и особенности ее использования рассмотрены далее.

В результате работы процессов на экран будет выведено следующее:

```
Parent process with PID = 1276
Terminated
Parent process terminated
New parent process have PID = 1
```

Вывод слова `Terminated` на экран выполнен обработчиком по умолчанию сигнала `SIGTERM`.

Чтобы задать в программе функцию-обработчик сигнала, используется системный вызов `signal()`:

```
#include <signal.h>
void signal(int signal_num, void *handler_address);
```

Здесь команда `signal_num` задает номер сигнала, при получении которого процесс начинает выполнение функции-обработчика. Адрес функции-обработчика задается параметром `handler_address`.

Как было сказано ранее, формат заголовка функции-обработчика должен быть следующим:

```
void handler(int sig_num);
```

Здесь через параметр `sig_num` функции-обработчику передается номер сигнала, по получению которого функция была вызвана.

Типичная программа, определяющая обработчик сигнала, выглядит следующим образом:

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

void handler( int sig_no )
{
  printf("Signal with number %d caught\n", sig_no);
  _exit( sig_no );
}

int main()
{
  signal( SIGALRM, &handler );
  signal( SIGINT, &handler );
  while (1);
  return 0;
}
```

При нажатии на клавиши Ctrl-C программа выведет

```
Signal with number 2 caught
```

так как нажатие сочетания клавиш Ctrl-C посылает текущему процессу сигнал SIGINT.

В последней программе в функции main() определяется функция-обработчик для сигналов SIGALRM и SIGINT, затем программа встает в бесконечный цикл ожидания прихода сигнала. По получению одного из двух сигналов вызывается функция-обработчик handler(), которая выводит на экран номер полученного сигнала и завершает процесс с кодом возврата, равным номеру полученного сигнала.

Временны́е характеристики обмена сигналами. При анализе последнего примера следует обратить внимание на то, что если сигналы SIGALRM или SIGINT получены процессом до того, как функцией signal() установлены обработчики этих сигналов, то будет вызвана не функция handler(), а обработчик сигнала по умолчанию. Для данного примера эта проблема не слишком существенна, поскольку время, которое пройдет между запуском процесса и установкой сигналов, заведомо меньше того времени, которое пройдет между запуском процесса и посылкой ему первого сигнала. Однако, если бы перед установкой сигналов программа выполняла некоторые действия, требующие значительных вычислительных ресурсов, а значит, и времени на выполнение — мы уже не смогли бы гарантировать того, что обработчики сигналов будут установлены до получения первого сигнала.

Поскольку поступление сигнала в процесс никак не зависит от самого процесса, а зависит только от внешней среды, учет временны́х характеристик обмена сигналами между процессами и вероятностей поступления сигналов в определенный момент времени — один из основных моментов, который должен учитываться при проектировании программ, использующих сигналы.

Чтобы минимизировать вероятность получения сигнала в неурочное время, когда процесс занят другой работой, необходимо, во-первых, четко планировать моменты посылки сигналов другими процессами (когда это возможно), а, во-вторых, приостанавливать выполнение процесса-получателя в те моменты времени, когда получение сигнала наиболее вероятно. Если процесс получит сигнал в приостановленном состоянии, то это гарантированно не прервет никакой вычислительный процесс.

Приостановить выполнение процесса можно двумя способами. Первый способ останавливает выполнение процесса, и выход из состояния останова возможен только по получению процессом сигнала. Для такого останова используется функция pause(), а сам способ останова зачастую называют «установкой процесса в паузу». Второй способ останавливает выполнение процесса на заданный промежуток времени. Продолжение выполнения процесса возможно либо по истечении этого промежутка, либо при

получении сигнала. Для такого останова используется функция sleep(), а сам способ получил название «усыпление процесса».

Функции pause() и sleep() имеют следующие прототипы:

```
#include <signal.h>
void pause(void);
int sleep(int sleep_sec);
```

Функция pause() не возвращает и не принимает никаких параметров. Единственное действие, выполняемое этой функцией — останов выполнения процесса до получения процессом сигнала. После получения сигнала и завершения обработчика управление передается команде, следующей за pause().

Функция sleep() приостанавливает выполнение процесса на промежуток времени, продолжительность которого в секундах равна значению параметра sleep_sec. Если во время останова процесса им получен сигнал, то функция возвращает значение, равное числу целых секунд, оставшихся до истечения заданного промежутка времени. Например, если до истечения промежутка времени осталось 5,8 с, то функция вернет 5.

Если требуется задавать интервал времени более точно, можно воспользоваться функцией usleep():

```
#include <signal.h>
void usleep(long sleep_usec);
```

Функция usleep() отличается от функции sleep() единицей измерения времени — параметр sleep_usec задает время в микросекундах. В отличие от функции sleep() функция usleep() не возвращает времени, оставшегося до истечения интервала времени, если процессом был получен сигнал во время работы функции usleep().

Управление обработчиками сигналов. После получения процессом сигнала соответствующая этому сигналу строка таблицы обработчиков сигналов удаляется. Если обработчик сигнала не будет восстановлен после получения процессом сигнала, то по получению того же сигнала будет вызван обработчик по умолчанию. Для восстановления обработчика сигнала обычно помещают вызов функции signal() непосредственно в тело функции-обработчика сигнала. Например, следующая программа выводит строку «Сигнал получен» при каждом получении сигнала SIGINT, находясь при этом постоянно в ожидании получения сигнала:

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

void handler(int sig_no)
{
```

```
  signal(sig_no, &handler);
  printf("Signal caught\n");
}

int main()
{
  signal( SIGINT, &handler);
  while(1)
  {
    printf("Waiting for signal...\n");
    pause();
  }
  return 0;
}
```

При запуске этой программы (подразумевается, что ее исполняемый файл — `signal3` в текущем каталоге) из следующего задания:

```
#!/bin/bash
./signal3 &
pid=$!
sleep 1
kill -SIGINT $pid
sleep 1
kill -SIGINT $pid
sleep 1
kill -SIGTERM $pid
```

на экран будет выведено:

```
Waiting for signal...
Signal caught
Waiting for signal...
Signal caught
Waiting for signal...
./signal3.sh: line 10:  1044 Terminated
```

Данная программа устанавливает обработчик сигнала `SIGINT` — функцию `handler()`, затем входит в бесконечный цикл, в котором приостанавливает свое выполнение до получения сигнала, о чем выводит соответствующее сообщение. По получению сигнала управление передается функции `handler()`, которая восстанавливает обработчик сигнала и выводит сообщение, информирующее пользователя о получении сигнала. После этого управление передается на команду, следующую за `pause()`, т.е. начинается следующая итерация бесконечного цикла.

Обработчик не обязательно должен восстанавливать таблицу обработчиков так, чтобы следующий сигнал был обработан тем же обработчиком. Восстановление обработчиков можно использовать и в случаях, когда реакция процесса на сигналы должна меняться с течением времени.

Например, следующая программа выводит букву A по получению нечетного по счету сигнала SIGINT (первого, третьего и т.д.) и букву B при получении четного по счету сигнала SIGINT (второго, четвертого и т.д.). Достигается этот эффект последовательной сменой обработчиков сигнала SIGINT:

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

void handler1(int sig_no);
void handler2(int sig_no);

void handler1(int sig_no)
{
  signal(sig_no, &handler2);
  printf("A\n");
}
void handler2(int sig_no)
{
  signal(sig_no, &handler1);
  printf("B\n");
}

int main()
{
  signal(SIGINT, &handler1);
  while (1)
    pause();
  return 0;
}
```

При выполнении этой программы из следующего задания (подразумевается, что имя исполняемого файла программы — signal4)

```
#!/bin/bash
./a &
pid=$!
sleep 1
kill -SIGINT $pid
sleep 1
kill -SIGINT $pid
```

```
sleep 1
kill -SIGINT $pid
sleep 1
kill -SIGINT $pid
sleep 1
kill -SIGTERM $pid
```

на экран будет выведено:

```
A
B
A
B
./signal4.sh: line 14:  2452 Terminated
```

Рассматривавшиеся ранее временны́е ограничения механизма сигналов — вызов обработчика по умолчанию при получении сигнала до установки пользовательского обработчика — действуют не только во время выполнения основных функций программы, но и во время выполнения функций-обработчиков.

Так, если восстанавливать обработчик сигнала первой командой обработчика (как это делалось во всех предыдущих примерах), то при получении сигнала в момент выполнения любой следующей команды обработчика выполнение обработчика будет прервано. Поскольку на момент получения сигнала обработчик уже восстановлен, то он будет запущен заново. Если во время его выполнения не будет получено нового сигнала, то управление вернется обратно в ту же самую функцию-обработчик.

Такое поведение в чем-то сходно с рекурсивным вызовом функций, при этом глубина такой «рекурсии» будет равна числу сигналов, полученных во время выполнения обработчиков. В литературе иногда встречается термин «рекурсивный вызов обработчика». Такое название не совсем корректно, поскольку не имеет практически ничего общего с обычной рекурсией.

Несколько иное поведение будет у процесса, если восстанавливать обработчик, помещая вызов функции `signal()` в последнюю строку обработчика. Таким образом можно избавиться от повторных вызовов обработчика, но если процесс получит сигнал до момента восстановления его обработчика, то будет вызван обработчик по умолчанию, который может и вовсе завершить выполнение процесса.

Чтобы избежать описанных выше проблем, можно воспользоваться одним из трех вариантов:

1) уменьшить объем и сложность программного кода обработчика сигнала, сократив тем самым время его выполнения, а значит, и вероятность получения сигнала во время выполнения обработчика;

2) отложить принятие решения о способе обработки и игнорировать сигнал при помощи константы игнорирования сигналов SIG_IGN. Константа SIG_IGN подставляется в качестве параметра функции signal() вместо адреса функции-обработчика. После установки игнорирования сигнала для них не вызывается никакой обработчик — ни пользовательский, ни обработчик по умолчанию. Таким образом, можно установить режим игнорирования сигнала в начале функции-обработчика, обезопасив себя при этом от повторных вызовов и от вызова обработчика по умолчанию, а восстановить обработчик в конце функции.

Типичный обработчик в этом случае будет иметь следующий вид:

```
void handler(int sig_no)
{
  signal( sig_no, SIG_IGN);
  ...
  ...
  signal( sig_no, &handler);
}
```

3) воспользоваться сигнальными масками, которые подробно рассмотрены в следующем разделе.

Сигнальные маски. Сигнальная маска процесса определяет, какие сигналы, получаемые процессом, игнорируются и не обрабатываются. При создании процесса он наследует сигнальную маску своего родителя, однако в ходе жизненного цикла процесса сигнальная маска может быть изменена.

Для установки или считывания состояния сигнальной маски используется функция sigprocmask():

```
#include <signal.h>
int sigprocmask(int cmd, const sigset_t *new_mask,
  sigset_t *old_mask);
```

Параметр cmd задает действие, выполняемое над маской, и может принимать следующие значения:

- SIG_SETMASK — заменяет сигнальную маску процесса на значение маски, передаваемой в качестве параметра new_mask;
- SIG_BLOCK — добавляет в маску процесса сигналы, указанные в маске, передаваемой в качестве параметра new_mask. Добавленные сигналы начинают игнорироваться процессом;
- SIG_UNBLOCK — удаляет из маски процесса сигналы, указанные в маске, передаваемой в качестве параметра new_mask. Удаленные из маски сигналы более не игнорируются процессом.

Если при изменении значения сигнальной маски необходимо сохранить ее старое значение, то переменная, в которой сохраняется это значение, передается по ссылке как параметр old_mask. Если сохранения старого значения не требуется, то в качестве значения этого параметра передается NULL. Если требуется сохранить значение сигнальной маски процесса в переменной, передаваемой как old_mask, не изменяя его, то в качестве значения параметра new_mask передается NULL.

Для формирования переменных типа sigset_t, определяющих маску процесса, служат три функции — sigemptyset(), sigaddset() и sigdelset().

```
#include <signal.h>
int sigemptyset(sigset_t *sigmask);
int sigaddset(sigset_t *sigmask, const int signal_num);
int sigdelset(sigset_t *sigmask, const int signal_num);
```

Функция sigemptyset() очищает все заблокированные сигналы в переменной, передаваемой как параметр sigmask. Функция sigaddset() добавляет сигнал с номером signal_num в список заблокированных сигналов маски, заданной переменной, переданной как параметр sigmask. Функция sigdelset() выполняет обратное действие — удаляет из списка заблокированных сигналов в маске sigmask сигнал с номером signal_num. В случае успешного выполнения все эти функции возвращают нуль, в случае неуспешного (например, при неверном указателе на sigmask или неверном номере сигнала signal_num) — возвращают –1.

Проиллюстрируем использование сигнальных масок на следующем примере. Процесс ожидает сигнала SIGINT, по его получению блокирует обработку новых сигналов этого типа, выводит сообщение «Таймер запущен», ждет 5 с, выводит сообщение «Таймер остановлен», восстанавливает обработчик сигнала SIGINT, разблокирует его обработку и вновь начинвает ожидать прихода сигнала. Если сигнал SIGINT придет во время 5-секундной паузы, он будет игнорироваться. Блокировка принятого сигнала позволяет в данном случае избежать повторного вызова обработчика или вызова обработчика по умолчанию:

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

sigset_t sigmask;
void handler(int sig_no)
{
  sigemptyset(&sigmask );
  sigaddset(&sigmask, sig_no );
```

```
  sigprocmask(SIG_BLOCK, &sigmask, NULL);
  printf("Timer started\n");
  sleep(5);
  printf("Timer stopped\n");
  signal( sig_no, &handler );
  sigprocmask(SIG_UNBLOCK, &sigmask, NULL);
}
int main()
{
  signal( SIGINT, &handler );
  while (1)
  {
    pause();
  }
}
```

При запуске программы (имя ее исполняемого файла `signal5`) из задания

```
#!/bin/bash
./signal5 &
pid=$!
for i in 1 2 3 4 5 6 ; do
  sleep 1
  kill -SIGINT $pid
done
kill -SIGTERM $pid
```

на экран будет выведено

```
Timer started
Timer stopped
Timer started
./signal5.sh: line 11:  1276 Terminated
```

Время выполнения функций `sigemptyset()` и `sigaddset()` в начале обработчика достаточно мало, однако, если существует вероятность прихода сигнала во время выполнения этих функций, можно переместить вызов этих функций в начало функции `main()` и заранее сформировать нужное значение маски, блокирующей сигнал `SIGINT` при помощи вызова `sigaddset(&sigmask, SIGINT)`. При этом придется пожертвовать универсальностью, поскольку номер сигнала будет задаваться явно, а не через параметр обработчика.

Если изобразить ход выполнения процесса на линии времени, то можно получить картину, изображенную на рис. 9.3 — второй сигнал `SIGINT` игнорируется.

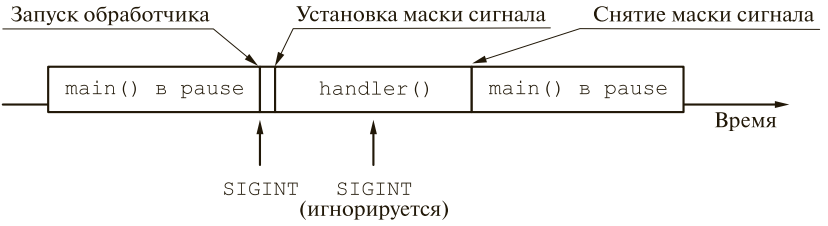

Рис. 9.3. Выполнение процесса при установленной сигнальной маске

Таймер. Все рассмотренные выше примеры программ, основанные на использовании сигналов, имеют один общий недостаток — полное время их выполнения целиком и полностью зависит от внешней среды. Завершить выполнение этих программ могут либо другие процессы, послав сигнал `SIGKILL` или `SIGINT`, либо пользователь, нажавший клавиши Ctrl-C. Если такой процесс будет выполняться в фоновом режиме, без участия пользователя, то завершение процесса будет зависеть только от других процессов. Сам процесс, находясь в состоянии ожидания, не может завершить свое выполнение.

Для решения этой проблемы в UNIX предназначен системный вызов `alarm()`:

```
#include <signal.h>
unsigned int alarm(unsigned int time_interval);
```

После вызова функции `alarm()` по истечении интервала времени `time_interval`, заданного в секундах, ядро пошлет сигнал `SIGALRM` процессу, вызвавшему эту функцию.

Если вызвать функцию `alarm()` до истечения интервала времени, заданного предыдущим вызовом функции, то функция установит новый интервал, заданный параметром `time_interval`, и вернет число секунд, оставшихся до истечения старого интервала. Если значение параметра `time_interval` при этом будет равно нулю, то посылка сигнала будет отменена.

Функцию `alarm()` можно использовать для корректного завершения процесса по истечении заданного промежутка времени. Для этого достаточно определить обработчик сигнала `SIGALRM`, завершающий выполнение процесса, и вызвать функцию `alarm()` в начале выполнения программы. Пример программы, использующей функцию `alarm()` для своего завершения через 20 с после запуска, приведен ниже:

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

void hdlrAlarm(int sig_no)
```

```
{
  printf("Execution terminating\n");
  exit(0);
}
void hdlrInt(int sig_no)
{
  signal( sig_no, &hdlrInt );
  printf("Signal %d caught\n", sig_no);
}
int main()
{
  signal( SIGINT, &hdlrInt);
  signal( SIGALRM, &hdlrAlarm);
  printf("Timer started\n");
  alarm(20);
  while (1)
  {
    printf("Waiting...\n");
    pause();
  }
  return 0;
}
```

При запуске программы (с именем исполняемого файла `signal6`) из задания

```
#!/bin/bash
./a &
pid=$!
for i in 1 2 3 4 5 6 ; do
  sleep 1
  kill -SIGINT $pid
done
wait $pid
```

на экран будет выведено

```
Timer started
Waiting...
Signal 2 caught
Waiting...
Signal 2 caught
Waiting...
Signal 2 caught
Waiting...
Signal 2 caught
Waiting...
```

```
Signal 2 caught
Waiting...
Signal 2 caught
Waiting...
Execution terminating
```

Потери сигналов. Рассматриваемые в этом подразделе сигналы получили название «ненадежные сигналы», поскольку не гарантируется доставка этих сигналов до процесса-получателя и их обработка процессом. Основная причина заключается в том, что после прихода сигнала сбрасывается адрес функции-обработчика в соответствующей строке таблицы обработчиков, а также с тем, что при одновременном приходе нескольких одинаковых сигналов процесс обрабатывает только один из них.

Рассмотрим более подробно структуру таблицы обработчиков сигналов и процесс доставки сигнала в процесс. Таблица обработчиков сигналов — абстрактная структура, которая реализована в UNIX-системах в виде двух отдельных массивов: массива сигнальных флагов и массива спецификаций обработки сигналов (рис. 9.4) [15].

Массив сигнальных флагов хранится в соответствующей процессу строке таблицы процессов, массив спецификаций обработки сигналов — в так называемой U-области процесса — области, содержащей формируемые и используемые во время работы процесса данные. Каждый сигнальный флаг соответствует сигналу одного типа. Когда процессу посылается сигнал, ядро устанавливает соответствующий флаг в его строке таблицы процессов.

Если процесс-получатель находится в состоянии ожидания, ядро переводит процесс в активное состояние и ставит его в очередь ожидающих выполнения процессов. Если процесс-получатель

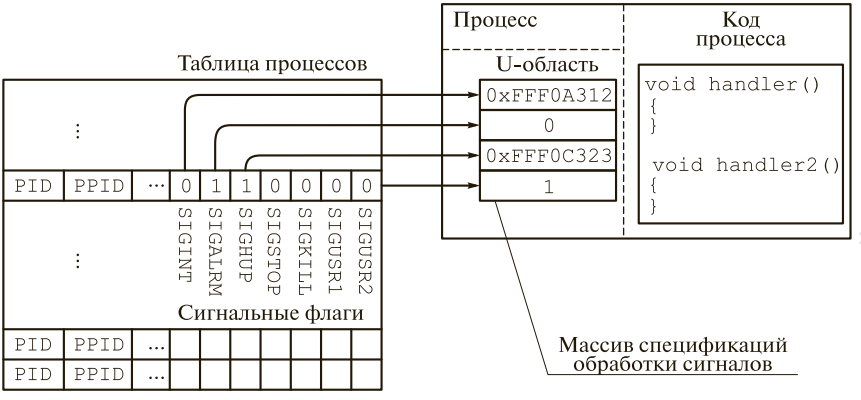

Рис. 9.4. Структура таблицы обработчиков сигналов

активен, то ядро считывает значение элемента массива спецификаций обработки сигнала, который соответствует установленному сигнальному флагу. Если элемент массива содержит нулевое значение, то процесс выполнит обработчик по умолчанию. Если элемент массива содержит единицу, то сигнал будет проигнорирован. Любое другое значение элемента массива используется в качестве адреса функции-обработчика сигнала. В последнем случае ядро передает управление внутри процесса-получателя функции-обработчику.

Перед передачей управления функции-обработчику ядро сбрасывает сигнальный флаг и записывает нулевое значение в соответствующий элемент массива спецификаций обработчиков сигналов. Таким образом, если последовательно посылаются несколько одинаковых сигналов, то заданная пользователем функция-обработчик будет вызвана только для первого сигнала, для остальных будет вызван обработчик по умолчанию.

Как уже упоминалось, чтобы корректно обрабатывать все поступающие последовательно сигналы, необходимо восстанавливать адрес функции-обработчика в соответствующем элементе массива спецификаций обработки. Однако это не даст гарантии того, что процесс обработает все сигналы — при получении сигнала в интервал времени с момента вызова функции-обработчика до момента восстановления ее адреса в массиве спецификаций обработки (этот интервал имеет небольшую, но ненулевую длительность) процесс будет выполнять обработчик по умолчанию.

Если процесс получит сигнал во время восстановления адреса в массиве спецификаций обработки (процесс восстановления также занимает некоторое небольшое время), то возникнет ситуация гонок — неизвестно, какой из обработчиков будет вызван: обработчик по умолчанию или восстановленный пользовательский обработчик (рис. 9.5).

Все эти особенности должны учитываться при разработке приложений, принимающих сигналы, посланные через очень маленькие промежутки времени. Чтобы устранить недостатки ненадежных сигналов, в UNIX-системах используется механизм надежных сигналов, гарантирующих доставку и обработку сигналов процессами-получателями. Рассмотрение этого механизма выходит за рамки данного учебника ввиду ограниченности его объема, ознакомиться с ним можно в книгах [15] и [9].

Синхронизация процессов. Чтобы избежать проблем с потерями сигналов, рассмотренными выше, нужно организовывать обмен сигналами между процессами таким образом, чтобы процесс-передатчик посылал очередной сигнал не ранее, чем процесс-получатель будет в состоянии его корректно обработать.

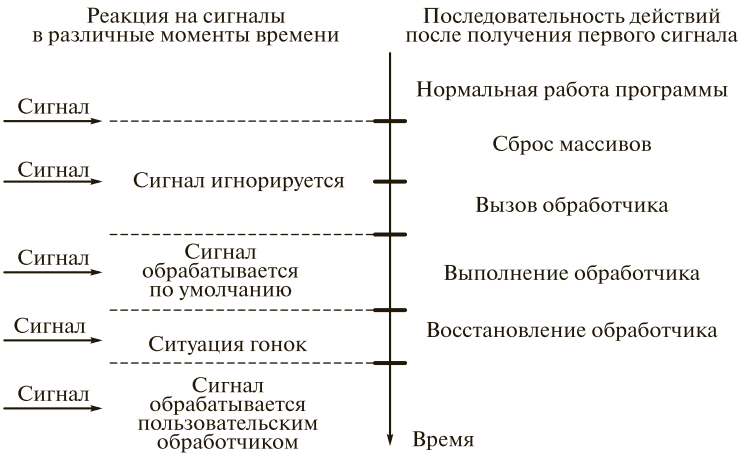

Рис. 9.5. Реакция на сигнал при его поступлении в различные моменты времени

Один из возможных способов реализации такого подхода состоит в том, что процесс-передатчик посылает сигналы через промежутки времени, заведомо большие требуемых процессу-получателю для обработки предыдущего сигнала и восстановления обработчика. Например, следующая программа порождает два процесса — родительский, порождаемый при ее запуске, и дочерний, порождаемый запуском команды `fork()`. Дочерний процесс выступает в роли передатчика и посылает родительскому процессу сигнал `SIGINT` 10 раз с интервалом между посылками в 1 с. Получатель (родительский процесс) в ответ на получение сигнала выводит общее число полученных им сигналов `SIGINT`. После посылки серии из 10-ти сигналов `SIGINT` передатчик посылает получателю сигнал `SIGTERM`, завершая его, а затем завершает свое выполнение:

```
#include <process.h> /* В Linux не нужен */
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>

int num;

void handler(int sig_no)
{
  signal(sig_no, &handler);
  num++;
  printf("%d\n", num);
}
```

```
int main()
{
  pid_t pid;
  int i;
  num = 0;

  setvbuf(stdout,(char*)NULL,_IONBF,0);

  switch (pid=fork())
  {
    case –1:
      perror("Fork error");
      exit(1);
      break;
    case 0:
      sleep(5); /* Даем время родителю поставить*/
                /* обработчик */
      for (i=0; i<10; i++)
      {
        kill( getppid(), SIGINT); /* Шлем SIGINT */
        sleep(1); /* Ждем готовности приемника */
      }
      kill( getppid(), SIGTERM); /*Завершаем приемник*/
      exit(0); /* Завершаем передатчик */
      break;
    default:
      signal( SIGINT, &handler);
      while (1) pause();
      _exit(0);
      break;
  }
  return 0;
}
```

В результате выполнения этой программы на экран будет выведена последовательность чисел от 1 до 10. Если посылать сигналы без перерыва между посылками (убрав вызов `sleep(1)`), то вероятность потери сигналов значительно увеличится, причем эта вероятность будет тем выше, чем больше система загружена выполнением процессов. Потеря сигналов в данном случае будет выражаться в том, что будут выведены числа не от 1 до 10, а до меньшего числа (например, до 8).

Основной недостаток этой программы заключается в том, что время межпроцессного взаимодействия очень велико и значительную часть времени процесс-получатель простаивает в ожидании сигнала. Для уменьшения времени работы и увеличения количест-

ва информации, передаваемой при помощи сигналов за единицу времени, можно синхронизировать выполнение приемника и передатчика, организовав между ними двунаправленную передачу сигналов. При этом передатчик, как и раньше, посылает сигналы приемнику, а приемник посылает сигналы-ответы, сигнализирующие о том, что приемник готов получить и обработать следующий сигнал. Непроизводительные задержки при такой схеме взаимодействия сводятся к минимуму.

В следующем примере процесс-передатчик ждет 5 с перед началом передачи сигналов для того, чтобы дать процессу-приемнику установить обработчики сигналов (реальное время, необходимое для установки обработчиков, много меньше и имеет порядок меньше единиц миллисекунд). Дальнейшее взаимодействие между передатчиком и приемником двунаправленное — передатчик посылает сигнал `SIGINT`, приемник обрабатывает сигнал при помощи функции `hdlrInt()`, восстанавливает обработчик и посылает передатчику сигнал-ответ `SIGALRM`, свидетельствующий о готовности приемника:

```
#include <process.h> /* В Linux не нужен */
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>

int num;
void hdlrInt(int sig_no)
{
  signal(sig_no, &hdlrInt);
  num++;
  printf("%d\n", num);
}

void hdlrAlarm(int sig_no)
{
  signal(sig_no, &hdlrAlarm);
}

int main()
{
  pid_t pid;
  int i;

  setvbuf(stdout,(char*)NULL,_IONBF,0);

  switch ( pid = fork() )
  {
    case -1:
      perror("Fork error\n");
```

```
      _exit(1);
      break;
    case 0:
      signal(SIGALRM, &hdlrAlarm);
      sleep(5); /* Ждем инициализацию приемника */
      for (i=0; i<10; i++)
      {
        kill(getppid(), SIGINT); /* Шлем SIGINT */
        pause(); /* Ждем SIGALRM */
      }
      kill(getppid(), SIGTERM); /*Завершаем приемник */
      _exit(0); /* Завершаем передатчик */
      break;
    default:
      signal(SIGINT, &hdlrInt);
      while (1)
      {
        pause(); /* Ждем SIGINT */
        kill(pid, SIGALRM); /*вернулись из hdlrInt*/
        /* и шлем ответ */
      }
      _exit(0);
      break;
  }
  return 0;
}
```

Приведенный пример намеренно несколько упрощен — следует обратить внимание на то, что и приемник, и передатчик могут быть выведены из состояния паузы (в момент выполнения функции `pause()`) любым сигналом, а не только сигналами `SIGALRM` и `SIGINT`. Для более корректной работы остальные сигналы должны игнорироваться либо при помощи сигнальной маски, либо при помощи константы `SIG_IGN`.

9.3.2. Сообщения

Основные понятия. Механизм сигналов, рассмотренный в подразд. 9.3.1, удобен для организации межпроцессного взаимодействия в случаях, не требующих обмена данными между процессами. Например, сигналы плохо применимы в случаях, когда взаимодействующие процессы обмениваются текстовыми данными.

Ценой потери производительности сигналы могут применяться и в случае обмена текстовыми данными, например, когда каж-

дый символ может кодироваться определенной последовательностью сигналов, начало и конец передачи символа также может обозначаться сигналами определенного типа. Такой способ неудобен тем, что для передачи одного символа требуется передача достаточно большого числа сигналов (для передачи букв латинского алфавита — порядка четырех, включая сигналы, обозначающие начало и конец передачи символа), а это связано с потерей информации в случае потери сигналов. Кроме того, такой метод обмена данными применим только тогда, когда оба обменивающихся данными процесса работают одновременно.

Для обмена данными между процессами (возможно, выполняющимися в разное время) в UNIX-системах существует механизм сообщений. Работа этого механизма напоминает использование почтового ящика общего доступа. Люди могут опускать в такой ящик письма, надписывая на конверте имя адресата, а адресаты, периодически проверяя содержимое почтового ящика, забирают предназначающиеся им сообщения, определяя это по имени адресата. При этом сохраняется последовательность помещения писем в ящик, и если человеку адресовано несколько писем, то вначале он прочитает самое старое, потом перейдет к более новым.

Кроме разделения писем по адресам, возможно также разделение писем по разным ящикам, например, один ящик может использоваться для частной переписки только двух человек.

Аналогично работает механизм сообщений — ОС выделяет специальную область памяти, в которой хранятся последовательности сообщений — так называемые *очереди сообщений*, в которые процессы могут помещать свои сообщения, как в почтовые ящики. Каждое сообщение состоит из числового идентификатора и блока данных, несущего передаваемую информацию. Помещение данных в очередь и их извлечение из очереди идет по принци-

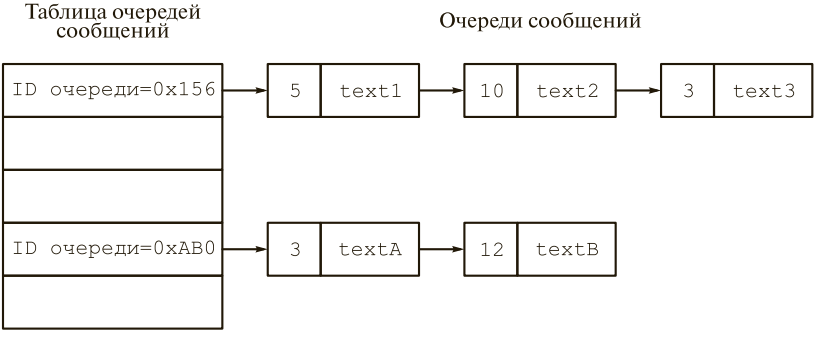

Рис. 9.6. Структура очередей сообщений

пу FIFO (*first in-first out*), т.е. из очереди первым извлекается самое старое сообщение. Однако поскольку каждое сообщение в очереди имеет идентификатор, то может извлекаться и самое старое сообщение с заданным идентификатором, которое совсем не обязательно будет самым старым сообщением среди всех сообщений очереди (рис. 9.6).

Таким образом, ОС гарантирует сохранность сообщений и возможность их обработки в порядке поступления в очередь, однако не может гарантировать того, что сообщение будет извлечено из очереди именно тем процессом, для которого оно предназначалось. Это связано с тем, что идентификатор извлекаемого сообщения задает процесс-получатель.

Очереди сообщений и сообщения в них существуют независимо от использующих их процессов, поэтому возможна ситуация, когда процесс-отправитель помещает сообщение в очередь и завершает свое выполнение. Процесс-получатель получит это сообщение, несмотря на то, что процесс-отправитель уже недоступен.

Каждая очередь сообщений может использоваться несколькими процессами, помещающими сообщения в очередь и извлекающими их из очереди независимо друг от друга. При этом обычно каждый из процессов использует свой набор идентификаторов для сообщений, однако если несколько процессов используют один и тот же идентификатор, могут возникнуть ситуации, в которых сообщения разных отправителей будут перемешиваться.

Возможна ситуация, когда несколько процессов одновременно помещают в очередь два сообщения с одинаковыми идентификаторами. Порядок помещения этих сообщений в очередь будет произвольным и зависит от реализации ядра ОС.

Другая ситуация возникает в случае, когда несколько процессов одновременно пытаются получить из очереди последнее сообщение с заданным идентификатором. Порядок выдачи сообщений процессам также не определен и зависит от реализации ядра ОС.

Обычно такие ситуации не представляют собой серьезной проблемы, поскольку при аккуратном проектировании все пары процессов, обменивающихся данными через одну очередь, используют непересекающиеся множества идентификаторов. Созданная очередь существует с момента ее создания процессом ОС и до момента ее явного уничтожения процессом, имеющим на это право, или до момента завершения работы ОС. Время жизни очереди может быть дольше времени жизни процесса, создавшего ее.

Параметры очередей сообщений определяются параметрами `kernel.msgmni`, `kernel.msgmax` и `kernel.msgmnb`. Параметр `kernel.msgmni` определяет максимальное число идентифика-

торов очередей сообщений, параметр kernel.msgmax определяет максимальный размер сообщения в байтах, а параметр kernel.msgmnb задает максимальный размер очереди в байтах. Эти параметры можно оперативно изменить, воспользовавшись файлами, расположенными в файловой системе /proc. Для этого можно использовать команды

```
echo 2048 > /proc/sys/kernel/msgmax
echo 4096 > /proc/sys/kernel/msgmni
```

Чтобы сохранить эти изменения в системе и после перезагрузки, необходимо вручную занести новые значения этих параметров в файл /etc/sysctl.conf, вставив в этот файл строки вида kernel.msgmnb=8192 или воспользоваться командой sysctl –w <paramer>=<value>, например

```
sysctl –w kernel.msgmnb=8192.
```

Получить информацию о текущем значении параметров очередей сообщений можно с помощью команды ipcs –lq:

```
$ ipcs –lq
------ Messages: Limits --------
max queues system wide = 496                // MSGMNI
max size of message (bytes) = 8192          // MSGMAX
default max size of queue (bytes) = 16384 // MSGMNB
```

Для просмотра всех созданных на данный момент очередей в системе используется команда ipcs. Будучи запущенной без параметров, команда выводит все созданные участки общей памяти, семафоры и очереди. Для просмотра созданных очередей необходимо запустить из командной строки ipcs с параметром –q:

```
$ ipcs –q
------ Message Queues --------
key    msqid owner perms used-bytes messages
0x000001f4 0   admin 644 20   1
0x1cda78da 486 root 600 18920  211
```

Структуры данных для сообщений в ОС UNIX. Вся служебная информация, описывающая существующие в системе очереди сообщений, находится в таблице очередей сообщений, хранящейся в памяти ядра ОС. Каждая запись в такой таблице описывает одну очередь и содержит следующую информацию:

• *идентификатор очереди* — целое число, позволяющее однозначно идентифицировать очередь. Идентификатор присваивается очереди создающим ее процессом; процессы, использующие очередь для обмена данными, могут использовать этот идентификатор для получения доступа к очереди;

• *UID и GID создателя очереди* — процесс, EUID которого совпадает с UID создателя очереди, может управлять очередью: изменять ее параметры или удалить ее;

• PID процесса, который последним поместил сообщение в очередь, и время этого события;

• PID процесса, который последним считал сообщение из очереди, и время этого события;

• указатель на область памяти ядра, в которой хранится линейный список с сообщениями, находящимися в очереди. Каждый элемент такого линейного списка содержит данные сообщения и его идентификатор. Кроме того, элемент списка хранит служебную информацию — размер сообщения в байтах и указатель на следующий элемент линейного списка. Последовательность элементов в списке определяет последовательность сообщений в очереди.

Для ограничения доступа к очередям используется такой же метод задания прав владельца-пользователя и владельца-группы, как и для файлов.

Максимальное количество очередей, максимальный размер очереди и сообщения задается константами, определяемыми ядром ОС.

При помещении сообщения в очередь создается новый элемент линейного списка, в который помещаются данные сообщения и идентификатор. Поскольку данные сообщения и идентификатор копируются из адресного пространства процесса в адресное пространство памяти ядра, процесс-отправитель может завершить свое выполнение в любой момент — сообщение в очереди останется нетронутым.

Системные вызовы для работы с сообщениями. Интерфейс для работы с сообщениями в ОС UNIX очень напоминает интерфейс для работы с файлами — в нем существуют функции для создания объекта, записи и чтения из него данных и управления его состоянием. Для файлов в роли объектов выступают собственно файлы, в роли данных — текстовые или двоичные строки. Состояние файла — права и режим доступа к нему. Для сообщений в роли объектов выступают очереди сообщений, в роли данных — сами сообщения, состояние очереди определяет права и режим доступа к ней.

Соответственно этому существуют четыре основные функции для работы с очередями. Все эти функции требуют включения заголовочных файлов `sys/types.h`, `sys/ipc.h` и `sys/msg.h`:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget( key_t key, int flag);
```

Для создания очереди или для открытия существующей служит функция msgget(). В качестве параметра key этой функции передается уникальный числовой идентификатор очереди (аналог имени файла в аналогичной функции fopen() для работы с файлами). Если задать в качестве параметра key системную константу IPC_PRIVATE, то будет создана очередь, которая сможет использоваться только создавшим ее процессом. Параметр flag задает параметры создания очереди. Если в качестве параметра передана константа IPC_CREAT, то вызов msgget создает очередь с заданным уникальным идентификатором. Для определения прав доступа к очереди следует объединить константу IPC_CREAT с цифровой формой записи прав доступа (см. подразд. 4.4.2), например, следующим образом:

```
m_id = msgget (500, IPC_CREAT | 0644);
```

При этом право r задает возможность считывать сообщения из очереди, право w — помещать сообщения в очередь, а право x — управлять параметрами очереди.

Функция msgget() возвращает дескриптор очереди, который может затем использоваться функциями посылки и приема сообщений (аналогом этого дескриптора для работы с файлами является файловый дескриптор типа FILE*). Если в качестве параметра функции msgget() передать 0, то она откроет очередь с заданным идентификатором и вернет ее дескриптор. Если объединить параметр создания очереди с константой IPC_EXCL и создание или открытие очереди оказалось неудачным, функция возвращает –1.

Сообщения представляются в виде структуры, имеющей следующий формат:

```
#define LENGTH 256
struct message
{
  long m_type;
  char m_text[LENGTH];
}
```

Длинное целое m_type задает идентификатор типа сообщения, массив символов m_text — текст сообщения. Константа LENGTH, задающая максимально возможную длину обрабатываемого программой сообщения (в рассматриваемом примере — 256 символов), должна быть меньше или равна системной константе MSGMAX, которая задает максимально возможную длину сообщения, помещающегося в очередь:

```
int msgsnd (int msgfd, void *msgPtr, int len, int flag);
```

Для посылки сообщения в очередь служит функция `msgsnd()`, которая в качестве параметра `msgfd` принимает дескриптор очереди, полученный в результате вызова `msgget()`, в качестве `msgPtr` — указатель на структуру данных, содержащую сообщение, параметр `len` задает длину элемента `m_text` сообщения. Если в качестве параметра `flag` передается 0, это означает, что выполнение процесса можно приостановить до момента успешного помещения сообщения в очередь. Такая приостановка может быть связана, например, с тем, что очередь сообщений в момент вызова `msgsnd()` переполнена, и необходимо подождать до тех пор, пока из очереди не будет извлечено достаточное количество сообщений, освобождающих место в ней. Если в качестве параметра `flag` указать константу `IPC_NOWAIT`, то в случае необходимости ожидания выполнение функции прерывается, и она возвращает –1, как в случае неуспешного выполнения операции.

Так, следующая программа создает очередь и посылает в нее сообщение

```
#include <stdio.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/types.h>

struct message
{
  long length;
  char text[256];
} my_msg = { 25, "Sample text message" };

int main()
{
  int fd;
  fd = msgget(500, IPC_CREAT | IPC_EXCL |0644 );
  if (fd != -1 )
  {
    msgsnd(fd, &my_msg, strlen(my_msg.text)+1,
    IPC_NOWAIT);
  }
  return 0;
}
```

В приведенном примере введена простая обработка неуспешного кода возврата функции `msgget()` — в случае невозможности создания очереди (например, если переполнена системная область памяти, выделенная для хранения очередей, или если очередь с

таким числовым идентификатором уже существует) сообщение в очередь послано не будет.

Для получения сообщения из очереди служит функция `msgrcv()`:

```
int msgrcv(int msgfd, void *msgPtr, int len, int mtype,
int flag);
```

В качестве параметра `msgfd` ей передается дескриптор очереди, параметр `msgPtr` определяет указатель на область памяти, в которую будет помещено полученное сообщение, параметр `len` задает максимально возможную длину текстовой части принимаемого сообщения. Значение `mtype` задает тип сообщения, которое будет принято. Этот параметр может принимать следующие значения:

- 0 — из очереди будет принято самое старое сообщение любого типа;
- положительное целое число — из очереди будет принято самое старое сообщение с типом, равным этому числу;
- отрицательное целое число — из очереди будет принято самое старое сообщение с типом, равным или меньшим модулю этого числа. Если сообщений, удовлетворяющих этому критерию, в очереди более одного, то будет принято сообщение, имеющее наименьшее значение типа.

В качестве параметра `flag` можно указать 0, в этом случае при отсутствии сообщений в очереди процесс приостановит свое выполнение до получения сообщения. Если в качестве параметра указать константу `IPC_NOWAIT`, то в случае отсутствия доступных для приема сообщений в очереди процесс продолжит свое выполнение.

Функция `msgrcv` возвращает количество принятых байт текстовой части сообщения в случае успешного его получения или –1 в случае неуспешного, например, если длина сообщения превышает длину, указанную параметром `len`. Чтобы предотвратить эту ситуацию и принять хотя бы первые `len` символов, в качестве параметра `flag` можно указать константу `MSG_NOERROR`. Все указанные константы можно объединять при помощи поразрядной операции ИЛИ.

Так, следующая программа принимает сообщение, которое поместила в очередь предыдущая программа, и выводит его текстовую часть на экран:

```
#include <stdio.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

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

struct message
{
  long length;
  char text[256];
} my_msg = { 0, "---------------" };

int main()
{
  int fd, rcv_res;
  fd = msgget(500, IPC_EXCL );
  if (fd != -1 )
  {
    rcv_msg = msgrcv(fd, &my_msg, 1024, MSG_NOERROR);
    if (rcv_msg != 1)
    {
      printf("%s\n", &my_msg.text)
    }
  }
  return 0;
}
```

В случае неуспешного получения сообщения программа ничего не выводит на экран.

Для управления очередью и получения ее параметров служит функция `msgctl()`:

```
int msgctl(int msgfd, int cmd, struct msqid_ds
*mbufPtr);
```

В качестве параметра `msgfd` ей передается дескриптор очереди, параметр `mbufPtr` задает параметры управления очередью, параметр `cmd` задает команду, которая выполняется над очередью. Параметр может принимать следующие значения:

- `IPC_STAT` — копирование управляющих параметров очереди в структуру, указатель на которую передается параметром `mbufPtr`;
- `IPC_SET` — замена параметров очереди теми, которые содержатся в структуре, указатель на которую передается параметром `mbufPtr`. Для успешного выполнения этой операции пользователь должен быть либо пользователем `root`, либо создателем, либо назначенным владельцем очереди;
- `IPC_RMID` — удаление очереди из системы. Для успешного выполнения этой операции пользователь должен быть либо пользователем `root`, либо создателем очереди, либо назначенным владельцем очереди. В качестве параметра `mbufPtr` в этом случае передается `NULL`.

Структура msqid_ds задает следующие параметры очереди сообщений:

```
struct msqid_ds {
    struct ipc_perm msg_perm; /*права доступа к
    очереди*/
    struct msg *msg_first; /* указатель на первое
    сообщение в очереди */
    struct msg *msg_last; /*указатель на последнее
    сообщение в очереди */
    time_t msg_stime; /* время последнего вызова
    msgsnd */
    time_t msg_rtime; /* время последнего вызова
    msgrcv */
    time_t msg_ctime; /* время последнего изменения
    очереди */
    ushort msg_cbytes; /* суммарный размер всех
    сообщений в очереди */
    ushort msg_qnum; /* количество сообщений в
    очереди */
    ushort msg_qbytes; /*максимальный размер очереди
    в байтах */
    ushort msg_lspid; /* PID последнего процесса,
    читавшего значение из очереди*/
    ushort msg_lrpid; /* PID последнего процесса,
    писавшего значение в очередь */
};
```

Следующая программа иллюстрирует обмен данными при помощи сообщений между двумя процессами. Программа порождает два процесса — процесс-родитель и процесс-потомок. Родитель посылает потомку сообщения в виде букв латинского алфавита, тип каждого сообщения — номер буквы. Потомок принимает эти сообщения в обратном порядке. Идентификатор используемой очереди соответствует PID процесса-родителя:

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

struct message
{
  long type;
```

```
  char text[10];
};
int main()
{
  pid_t pid;
  int qId;
  char i;
  int *status;
  struct message my_message;
  setvbuf(stdout,(char*)NULL,_IONBF,0);
  switch (pid = fork() )
  {
    case -1:
      perror("Bad fork\n");
      _exit(1);
      break;
    case 0:
      /* тело потомка */
      /* Ждем, пока родитель заполнит очередь */
      sleep(10);
      /* Открытие существующей очереди */
      qId = msgget(getppid(), 0 );
      if (qId == -1)
      {
        printf("Unable to open queue\n");
        _exit(0);
      }
      printf("r> ");
      for (i=26;i>0;i--)
      {
        /* Прием сообщений в обратном порядке */
        msgrcv(qId,&my_message, 2, i, MSG_NOERROR);
        printf("%d:%s  ", my_message.type,
        &my_message.text);
      }

      printf("\n\n\n");
      /* Удаление очереди */
      msgctl(qId, IPC_RMID, NULL);

      _exit(0);
      break;
    default:
      /* тело родителя */
      printf("Queue ID: %d\n",getpid());
```

```
      /* Создание очереди */
      qId = msgget(getpid(), IPC_CREAT | 0666 );

      if (qId == -1)
      {
        printf("Unable to create queue\n");
        kill(pid, SIGKILL);
        _exit(0);
      }
      printf("s> ");

      for (i='A'; i<='Z'; i++)
      {
        /* Создание сообщения */
        my_message.type = i - 'A' + 1;
        my_message.text[0] = i;
        my_message.text[1] = 0;
        printf("%d:%s  ", my_message.type,
        &my_message.text);
        /* Помещение сообщения в очередь */
        msgsnd(qId, &my_message, 2, 0);
      }

      printf("\n\n\n");
      /* Ожидание завершения потомка */
      wait(&status);
      return 0;
  }
}
```

В результате работы этой программы на экран будет выведен следующий текст:

```
Queue ID: 3952
s>
1:A  2:B  3:C  4:D  5:E  6:F  7:G  8:H  9:I  10:J
11:K  12:L  13:M  14:N  15:O  16:P  17:Q  18:R  19:S
20:T  21:U  22:V  23:W  24:X  25:Y  26:Z

r> 26:Z  25:Y  24:X  23:W  22:V  21:U  20:T  19:S  18:R
17:Q  16:P  15:O  14:N  13:M  12:L  11:K  10:J  9:I  8:H
7:G  6:F  5:E  4:D  3:C  2:B  1:A
```

Со строки s> начинаются сообщения, посылаемые родителем, со строки r> — принимаемые потомком. Далее на экран выводится пара идентификатор-текст в том порядке, в котором сообщение помещается или извлекается из очереди.

9.3.3. Семафоры

Основные понятия. Одна из наиболее важных задач при межпроцессном взаимодействии — задача *синхронизации выполнения процессов*. Под синхронизацией в первую очередь понимается параллельная работа нескольких процессов, при котором выполнение определенных участков кода произойдет не ранее того момента, как все остальные процессы придут в состояние готовности. Например, при параллельном вычислении определителя матрицы по минорам, при котором каждый минор вычисляется своим процессом, переход к минору большего порядка возможен только после того, как все процессы закончат вычисления и получат результат. Состояние готовности других процессов определяется при помощи посылки того или иного уведомления о готовности. Другим примером, требующим синхронизации, является доступ нескольких параллельно работающих процессов к одному неразделяемому ресурсу. В данном случае под синхронизацией понимается управление доступом таким образом, чтобы в один момент времени доступ к ресурсу получал только один процесс.

Один из наиболее часто применяемых механизмов синхронизации — *синхронизация при помощи семафоров*. Простейший семафор представляет собой флаг, значение 0 которого соответствует запрету на доступ, а 1 — разрешению на доступ. Перед доступом к ресурсу процесс должен проверить значение семафора, и если доступ разрешен — установить значение семафора в 0 для того, чтобы заблокировать доступ к ресурсу другим процессам. После окончания использования ресурса процесс устанавливает значение семафора обратно в 0. Такой семафор получил название *бинарного семафора*.

Кроме бинарных семафоров существуют так называемые семафоры-счетчики. Значение этого семафора представляет собой неотрицательное целое число. Над таким семафором определено две операции. Операция *V* (от голл. *verhogen*, часто обозначается *wait*) останавливает выполнение процесса в случае, если его значение меньше либо равно 0, или уменьшает значение семафора на 1 в противном случае. Операция *P* (от голл. *prolagen*, часто обозначается *post*) увеличивает значение семафора на 1 и уведомляет остановленные процессы об увеличении значения семафора, вызывая повторную проверку значения.

Семафоры-счетчики служат для совместного доступа к ограниченному количеству однотипных ресурсов, например, к печатающим устройствам, подключенным к одному компьютеру. Количество доступных устройств записывается в качестве начального значения семафора-счетчика.

Перед тем как занять одно из устройств, процесс выполняет над семафором операцию *V*. В случае, если осталось хотя бы одно доступное устройство — значение счетчика уменьшается на 1, отражая таким образом уменьшение количества доступных устройств. Если не осталось ни одного доступного устройства, то выполнение процесса приостанавливается до тех пор, пока устройство не освободится и значение счетчика не станет положительным. Поскольку ожидающих освобождения устройства процессов может быть несколько, то они организуются в очередь и доступ к устройству при увеличении значения счетчика получает первый в этой очереди процесс.

После освобождения ресурса (в нашем случае — печатающего устройства) процесс выполняет над семафором операцию *P*, тем самым увеличивая значение счетчика на 1.

Для нормальной работы семафоров-счетчиков в UNIX-системах должны выполняться следующие условия:

- значение семафора должно быть доступно различным процессам, поэтому должно находиться в адресном пространстве ядра ОС, а не процесса;
- операция проверки и изменения значения семафора должна быть реализована в виде одной атомарной операции, не прерываемой другими процессами. Иначе возможна ситуация прерывания процесса между проверкой и уменьшением значения семафора, что приведет семафор в непредсказуемое состояние.

Системные вызовы для работы с семафорами. В UNIX-системах существует несколько интерфейсов для работы с семафорами.

Рассмотрим интерфейс, появившийся раньше других и получивший название *интерфейс семафоров System V*. В нем используется понятие набора семафоров-счетчиков, рассматриваемых как единое целое. Каждый элемент набора представляет собой отдельный семафор-счетчик, при этом при помощи единственной операции над набором можно изменить значения входящих в набор семафоров, т.е. одновременно применить несколько операций *P* или *V* или объединить эти операции.

Для создания или открытия существующего набора семафоров в UNIX используется функция `semget()`:

```
#include <sys/sem.h> /*B Linux - #include
<linux/sem.h>*/
int semget (key_t key, int num, int flag);
```

Параметр `key` задает идентификатор набора семафоров, если в качестве него указана константа `IPC_PRIVATE`, набор семафоров может использоваться только процессом, создавшим набор, и его потомками. Параметр `num` задает количество семафоров в наборе,

в большинстве UNIX-подобных ОС этот параметр не должен превышать 25. Параметр `flag` может иметь следующие значения:

- 0 — в случае, если существует набор семафоров с идентификатором `key`, то функция возвращает его дескриптор;
- `IPC_CREAT` — создается новый набор семафоров с идентификатором `key`.

Если константа `IPC_CREAT` объединена с системной константой `IPC_EXCL` операцией логического сложения, то в случае, если набор семафоров с таким идентификатором уже существует, функция возвращает –1. Для задания прав доступа к набору семафоров необходимо объединить их восьмеричное представление с остальными константами.

Право доступа `r` для набора семафоров позволяет считывать его значение, право доступа `w` — изменять значение семафора, а право доступа `x` — изменять параметры набора семафоров.

Так, типичный вызов функции `semget()` будет выглядеть следующим образом:

```
int sem_descr;
sem_descr = semget(ftok("A",1), 1, IPC_CREAT | IPC_EXCL
| 0644);
```

После создания семафора его значение никак не инициализируется, поэтому перед началом использования набора семафоров необходимо присвоить начальное значение каждому семафору набора. Это можно сделать при помощи функции `semctl()`:

```
#include <sys/sem.h>
/*B Linux - #include <linux/sem.h>*/
int semctl (int semfd, int num, int cmd, union semun
arg);
```

Параметр `semfd` задает дескриптор набора семафоров, параметр `num` — номер семафора в наборе, `cmd` задает команду, выполняемую над семафором. Необязательный четвертый параметр `arg` задает параметры команды.

Структура четвертого параметра следующая:

```
union semun
{
  int val; /* используется командой SETVAL */
  struct semid_ds *buf; /*используется командами IPC_SET
  и IPC_STAT*/
  ushort *array; /* используется командами SETALL
  и GETALL */
}
```

Основные команды над семафорами:

• IPC_RMID — удаляет набор семафоров;

• IPC_SET — устанавливает параметры набора семафоров значениями из структуры arg.buf;

• IPC_GET — считывает значения параметров набора семафоров в структуру arg.buf;

• GETALL — считывает все значения семафоров в массив arg.array;

• SETALL — устанавливает значения всех семафоров из массива arg.array;

• GETVAL — возвращает значение семафора с номером num. Аргумент arg не используется;

• SETVAL — устанавливает значение семафора с номером num в arg.val.

В случае неуспешного выполнения функция возвращает –1.

Чтобы установить значения всех семафоров, можно воспользоваться командой SETALL или циклическим вызовом команды SETVAL:

```
#include <unistd.h>
#include <sys/sem.h> /*B Linux - #include <linux/
sem.h>*/

typedef int semnum;
int main()
{
  int i;
  int sem_descr;
  union semnum arg;
  arg.val = 4;

  sem_descr = semget(ftok("A"), 3,
  IPC_CREAT | IPC_EXCL | 0644);
  for (i=0;i<3;i++)
  {
  semctl(sem_descr, i, SETVAL, arg);
  }
}
```

Разделение операций создания и инициализации семафоров может привести к ситуации гонок, когда набор семафоров уже создан, но еще не инициализирован, а другой процесс пытается его использовать. Чтобы предотвратить такую ситуацию, можно принудительно начинать использовать семафоры спустя некоторое время после их создания. Более эффективные, но и более сложные способы избежать такой ситуации приведены в работе [9].

Для просмотра всех созданных на данный момент семафоров в системе используется команда ipcs. Для просмотра созданных семафоров необходимо запустить из командной строки ipcs с параметром -s.

```
$ ipcs –s
------ Message Queues --------
key        semid     owner     perms     nsems
0x00a1684b 0         admin     644       1
0xffffffff 18        nikita    644       1
0xffffffff 24        sergey    644       1
```

Параметры семафоров определяются составным параметром kernel.sem. Этот параметр имеет несколько подпараметров: semmni, semmsl, semmns и semopm. Параметр semmni определяет максимальное количество массивов семафоров, параметр semmsl — максимальное количество семафоров в массиве, параметр semmns задает максимальное число семафоров в системе и вычисляется как произведение параметров semmsl и semmni. Параметр semopm задает максимальное число операций над семафорами, которое может быть выполнено за один раз.

Перечисленные параметры можно оперативно изменить, как и параметры системы очередей сообщений. Для этого можно использовать команду

```
echo 250 256000 32 1024 > /proc/sys/kernel/sem
```

Первое число задает значение параметра semmsl, второе — параметра semmns, третье — параметра semopm и четвертое — параметра semmni.

Чтобы сохранить эти изменения в системе и после перезагрузки, необходимо вручную занести новые значения этих параметров в файл:

```
/etc/sysctl.conf (kernel.sem=250 256000 32 1024)
```

или воспользоваться следующей командой:

```
sysctl –w kernel.sem="250 256000 32 1024".
```

Получить информацию о значении параметров семафоров можно с помощью команды ipcs –ls:

```
$ ipcs –ls
------ Semaphore Limits --------
max number of arrays = 128           // SEMMNI
max semaphores per array = 250       // SEMMSL
max semaphores system wide = 32000   // SEMMNS
max ops per semop call = 32          // SEMOPM
semaphore max value = 32767
```

Для управления значением семафоров в наборе служит функция semop():

```
#include <sys/sem.h>
/*B Linux - #include <linux/sem.h>*/
int semop(int semfd, struct sembuf* opPtr, int len);
```

Параметр semfd задает дескриптор набора семафоров, opPtr — указатель на массив, каждый элемент которого задает одну операцию над семафором в наборе; параметр len определяет, сколько элементов содержится в наборе.

Элементы массива opPtr определены следующим образом:

```
struct sembuf
{
  short sem_num; /* Номер семафора в наборе */
  short sem_op; /* Операция над семафором */
  short sem_flg; /* Флаг операции */
}
```

Операция над семафором sem_op может принимать следующие значения:

• 0 — считывание значения семафора; если при этом оно равно 0, то выполнение процесса приостанавливается до тех пор, пока значение семафора не станет положительным;

• положительное число — увеличить значение семафора на заданную величину;

• отрицательное число — уменьшить значение семафора на заданную величину. Если при этом значение семафора станет отрицательным, то ядро также приостановит выполнение процесса. Выполнение процесса будет продолжено в момент времени, когда значение семафора станет неотрицательным.

Если в качестве флага sem_flg указать константу IPC_NOWAIT, то приостановки процесса не произойдет. Если указать в качестве флага константу SEM_UNDO, то ядро будет отслеживать изменения семафоров. Если процесс, уменьшивший значение семафоров до нуля или отрицательного числа, будет завершен, то сделанные им изменения будут отменены, чтобы не вызвать «вечного» ожидания процессов, ожидающих открытия семафора.

В разных версиях UNIX порядок следования полей в структуре sembuf может различаться, а также могут присутствовать другие поля, поэтому рекомендуется присваивать значения не по порядку, а при помощи имен полей.

В следующем примере два процесса используют семафор для диспетчеризации процесса вывода текста на экран. Два процесса поочередно выводят текстовые строки на экран, при этом при помощи функции sleep() «вывод» процесса-потомка занимает

4 с, родителя — 1 с. Чтобы сохранить очередность вывода, перед тем, как занять ресурс (в данном случае — терминал), каждый из процессов выполняет операцию *V*, а после освобождения — операцию *P*. Все операции над семафором в приведенном ниже примере хранятся в массиве `op`, при этом значение в `op[0]` уменьшает значение семафора на 1 и соответствует операции *V*, `op[1]` проверяет значение семафора на 0 и не используется в примере, `op[1]` увеличивает значение семафора на 1 и соответствует операции *P*:

```
#include <sys/sem.h>
/*B Linux - #include <linux/sem.h>*/
#include <time.h>
#include <unistd.h>
#include <stdio.h>
#include <signal.h>

struct sembuf *op[3]; /*Операции над семафором:
                  op[0] - уменьшает значение
                  op[1] - проверяет
                  op[2] - увеличивает значение*/

int main(void)
{
  int fd, i,j, *status;
  pid_t pid;

  setvbuf(stdout,(char*)NULL,_IONBF,0);
  for (i=0;i<3;i++)
  {
    /* Выделение памяти под операции*/
    p[i]=(struct sembuf*)malloc(sizeof(struct
    sembuf));
  }

  for (i=-1;i<2;i++)
  { /* Заполнение операций */
    op[i+1]->sem_num = 0;
    op[i+1]->sem_op = i;
    op[i+1]->sem_flg = 0;
  }
  /*В результате получаем {0,-1,0},{0,0,0},{0,1,0} */

  witch (pid = fork() )
  {
    case -1:
      perror("Bad fork\n");
```

```
      _exit(1);
      break;

    case 0:
      /* child body */
      sleep(1);
      /*Открытие семафора*/
      fd = semget(ftok("A",1), 1, 0);
      for (i=0;i<10;i++)
      {
        semop(fd, op[0], 1); /* V семафора */
        printf("c%d ",i);
        /*имитация длинного процесса (4 с)*/
        sleep(4);
        semop(fd, op[2], 1); /* P семафора */
      }
      break;

    default:
        /* parent body */
        /*создание семафора*/
        fd = semget(ftok("A",1), 1, IPC_CREAT | 0644);
        *установка начального значения семафора в 1*/
        semctl(fd, 0, SETVAL, 1);
        for (i=0;i<10;i++)
        {
          semop(fd, op[0], 1); /* V семафора */
          printf("p%d ",i);
          /*имитация короткого процесса (1 с)*/
          sleep(1);
          semop(fd, op[2], 1); /* P семафора */
        }
        wait(&status); /*ожидание завершения потомка*/
        semctl(fd, 0, IPC_RMID); /*удаление семафора*/
        break;
  }

    printf("\n");
    /*освобождение памяти операций*/
    for (i=0;i<3;i++) free(op[i]);
    return 0;
}
```

В результате запуска этой программы на экран будет выведена следующая последовательность, из которой видно, что выполнение процессов синхронизовано:

```
c0 p0 c1 p1 c2 p2 c3 p3 c4 p4 c5 p5 c6 p6 c7 p7
c8 p8 c9 p9
```

т. е. процесс-родитель, вывод которого занимает 1 с, ждет 3 с, пока не откроется семафор, установленный более медленным процессом-потомком, вывод которого занимает 4 с. Более медленный процесс ждет 1 с, пока не будет освобожден ресурс.

Если из вышеприведенной программы убрать вызовы функции `semop()`, то вывод программы будет выглядеть следующим образом:

```
c0 p0 p1 p2 p3 c1 p4 p5 p6 p7 c2 p8 p9 c3 c4 c5
c6 c7 c8 c9
```

Из сравнения двух вариантов вызова явно видны различия при отсутствии синхронизации — за то время, пока процесс-потомок успевает вывести одну последовательность символов `cX`, родитель успевает вывести четыре последовательности `pX`.

Если изобразить процесс работы с семафором на линии времени, то можно получить картину, изображенную на рис. 9.7. На рисунке изображено несколько линий времени, каждая из которых соответствует изменению состояния процесса или семафора. Прямоугольники на линиях времени соответствуют периодам активности процесса (первые две линии) и периодам положительного (открытого) значения семафора (третья линия). Нижняя

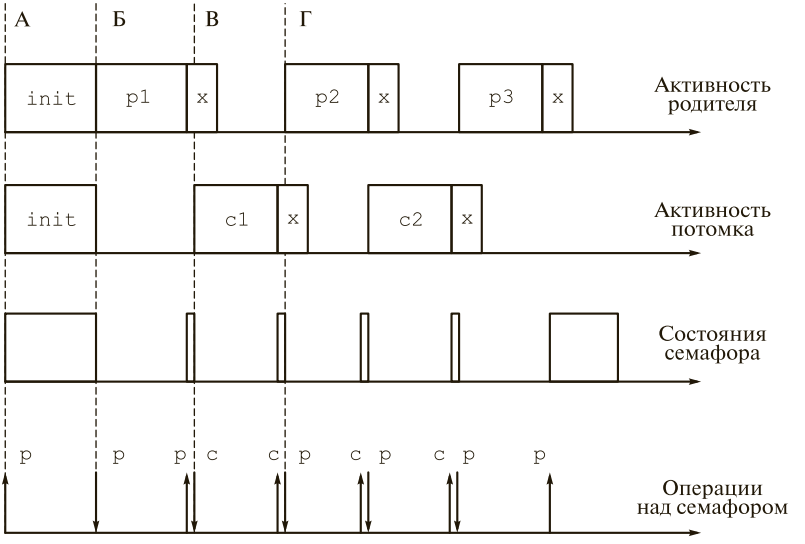

Рис. 9.7. Временна́я диаграмма использования семафора для синхронизации процессов

линия времени отражает операции над семафором, стрелкой вниз отмечена операция уменьшения значения семафора, стрелкой вверх — увеличения значения. Рядом со стрелкой указан процесс, выполнивший операцию. Буквой `p` обозначен процесс-родитель, буквой `c` — процесс-потомок.

Оба процесса стартуют практически одновременно и в начале инициализируют собственные данные, родитель при этом открывает семафор (рис. 9.7, срез А). После завершения инициализации процесс-родитель закрывает семафор, начиная тем самым критическую секцию (рис. 9.7, срез Б и p1).

Процесс-потомок при попытке закрыть семафор останавливается. После завершения своей критической секции родитель открывает семафор и продолжает выполнять программный код, не входящий в критическую секцию (рис. 9.7, `x` на графике). Процесс-потомок в этот момент активизируется, уменьшает значение семафора и начинает свое выполнение в критической секции (рис. 9.7, срез В и `c1` на графике).

Родитель после завершения кода, не входящего в критическую секцию, приостанавливает свое выполнение при попытке закрыть семафор и начинает работу только после того, как сможет вновь уменьшить значение семафора (рис. 9.7, срез Г). Затем процесс повторяется.

9.3.4. Общая память

Основные понятия. Одним из самых быстрых механизмов межпроцессного взаимодействия является механизм на основе *общей памяти* (*shared memory*). Связано это с тем, что при использовании данного механизма несколько процессов одновременно имеют доступ к одной и той же области памяти. При использовании такого механизма не происходит переноса данных, и операция записи выполняется только один раз. Сразу после этого записанные данные становятся доступны другим процессам, которые обращаются с областью общей памяти так, как если бы эта память принадлежала этим процессам (рис. 9.8).

Следует отметить тот факт, что при использовании данного механизма межпроцессного взаимодействия возникают дополнительные проблемы, связанные с одновременным доступом к одному и тому же блоку памяти. ОС не обеспечивает такого рода контроля, т. е. потенциально может возникать ситуация, когда два или более процессов могут записывать данные в один и тот же участок памяти. Поэтому задача разграничения доступа к общей памяти также должна решаться программистом. Эту задачу можно рассматривать как задачу синхронизации процессов. При этом

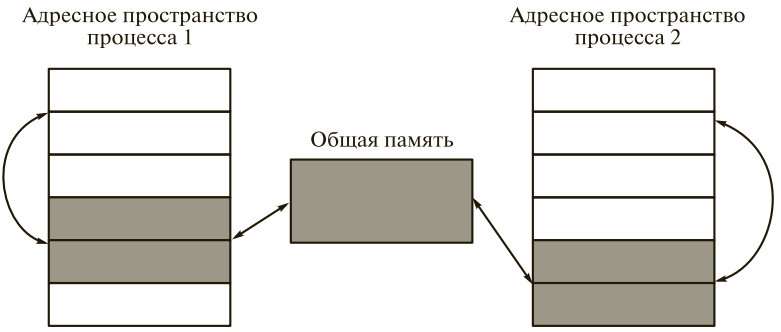

Рис. 9.8. Модель межпроцессного взаимодействия на основе общей памяти

задача синхронизации состоит в том, чтобы в каждый момент времени доступ к области общей памяти имел только один процесс. Поэтому для решения данной задачи часто используют рассмотренный выше механизм синхронизации процессов при помощи семафоров.

Системные вызовы для работы с общей памятью. Для создания новой области общей памяти или присоединения к уже существующей области используется системный вызов `shmget()`:

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int shmflg);
```

Параметр `key` задает ключ, ассоциированный с той областью общей памяти, которая должна быть создана или к которой текущий процесс должен получить доступ, если она уже существует. Параметр `size` задает размер общей памяти, причем система округляет этот размер в большую сторону пропорционально значению константы `PAGE_SIZE`. Третий параметр `shmflg` задает дополнительные параметры для операции выделения области общей памяти. Этот параметр должен представлять собой комбинацию побитово складываемых флагов, которые определяют:

- права доступа пользователей к данной области памяти (младшие 9 битов);
- если указан флаг `IPC_CREAT`, то выделяется новая область общей памяти с заданным идентификатором. Если данный флаг не используется, то делается попытка найти существующую область общей памяти с заданным идентификатором и получить к ней доступ;
- если вместе с флагом `IPC_CREAT` указывается флаг `IPC_EXCL`, то в случае, если область памяти с заданным идентификатором

уже существует, операция создания нового блока памяти завершается неудачей.

Если функции shmget() удается выделить новую область общей памяти или получить доступ к уже существующей области, то возвращается идентификатор этой области. Если операция по какой-либо причине завершается неудачей, то возвращается значение –1.

Типичный вызов функции shmget() выглядит следующим образом:

```
int shm_descr = shmget(ftok("A", 1), 1024, IPC_CREAT |
IPC_EXCL | 0666);
```

Однако прежде чем получить доступ к общей области памяти (созданной или уже существовавшей), необходимо присоединить эту область памяти к адресному пространству процесса. Данная процедура является обязательной для всех процессов в связи с особенностями представления процессов в ОС.

Каждый процесс имеет доступ только к определенному адресному пространству, которое относится исключительно к области действия самого процесса. И прежде чем использовать общую память, ее адресное пространство также должно располагаться в диапазоне адресов, доступных процессу.

Для выполнения операции присоединения общей памяти используется системный вызов shmat():

```
#include <sys/types.h>
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

Первый параметр shmid должен содержать идентификатор области общей памяти, полученный при вызове функции shmget(). Параметр shmaddr должен содержать начальный адрес, к которому будут присоединена область общей памяти. Если этот параметр равен NULL, то в этом случае начальный адрес определяется системой. Параметр shmflg содержит дополнительные флаги для присоединяемой области памяти. Например, если данный флаг содержит значение SHM_RDONLY, то присоединяемая область памяти будет доступна только для чтения. В противном случае область общей памяти доступна и для чтения, и для записи.

Если операция закончилась успешно, то возвращается адрес присоединенной области памяти, в противном случае возвращается значение –1.

Следует заметить, что при выполнении данной операции не выделяется новая область памяти. Вместо этого в виртуальном адресном пространстве процесса выделяется некоторый диапазон

адресов, который затем транслируется системой в реальные физические адреса области общей памяти.

Типичный вызов функции `shmat()` выглядит следующим образом:

```
char* mem = (char*)shmat(shm_descr, NULL, 0);
```

После окончания работы с областью общей памяти ее необходимо отсоединить от адресного пространства вызывающего процесса. Для выполнения данной операции используется функция `shmdt()`:

```
#include <sys/types.h>
#include <sys/shm.h>
int shmdt(const void *shmaddr);
```

В качестве единственного параметра данной функции указывается адрес, к которому была присоединена область общей памяти, возвращаемый функцией `shmat()`. Выполнение этой операции требуется для последующего удаления области общей памяти. Общая память продолжает существовать до тех пор, пока есть хоть один процесс, к которому она присоединена.

В качестве результата данная функция возвращает значение 0, если ранее присоединенную память удалось отсоединить от процесса, и –1 в случае неуспешного завершения операции.

Вызов функции `shmdt()` обычно выглядит следующим образом:

```
if(shmdt((void*)mem) < 0)
{
    perror("Unable to detach shared memory");
    exit(1);
}
```

В данном примере делается попытка отсоединить блок общей памяти от адресного пространства текущего процесса. Если в результате выполнения операции происходит ошибка, то выводится сообщение с описанием этой ошибки и процесс завершается с кодом возврата 1.

Для управления общими сегментами памяти предназначена функция `shmctl()`:

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Она позволяет пользователю получать данные об области общей памяти, устанавливать владельца, группу и права, а также удалять заданный сегмент общей памяти.

Параметр shmid должен содержать дескриптор области общей памяти. Второй параметр cmd задает ту операцию, которую необходимо выполнить над заданной областью общей памяти. В качестве таких операций могут использоваться следующие команды:

- IPC_STAT — для копирования информации об общей памяти в буфер buf;
- IPC_SET — для изменения владельца, группы и прав доступа;
- IPC_RMID — для удаления заданного блока общей памяти.

Параметр buf используется для получения данных об общей памяти или изменения ее параметров. Структура этого параметра имеет следующий вид:

```
struct shmid_ds {
    struct ipc_perm shm_perm; /* права на выполнение
    операции */
    int shm_segsz; /* размер сегмента (в байтах) */
    time_t shm_atime;         /* время последнего
    подключения */
    time_t shm_dtime;         /* время последнего
    отключения */
    time_t shm_ctime;         /* время последнего
    изменения структуры */
    unsigned short shm_cpid; /* идентификатор
    процесса создателя */
    unsigned short shm_lpid; /* идентификатор
    процесса, подключавшегося последним */
    short shm_nattch;         /* количество текущих
    подключений сегмента */
    …
};
```

Структура struct ipc_perm используется для управления правами доступа к области общей памяти:

```
struct ipc_perm {
    key_t key;
    ushort uid;/* идентификатор владельца euid */
    ushort gid;/* идентификатор группы владельца
    egid */
    ushort cuid;/* идентификатор создателя euid */
    ushort cgid;/* идентификатор группы создателя
    egid */
    ushort mode;/* младшие 9 бит прав доступа */
    ushort seq;
};
```

С помощью команды IPC_SET можно изменить параметры uid, gid и mode в данной структуре. При этом пользователь должен быть владельцем, создателем или суперпользователем процесса.

При использовании команды IPC_RMID последний параметр может быть установлен в NULL. После выполнения команды удаления блок памяти будет физически удален только в том случае, если нет ни одного процесса, связанного с данным блоком. В противном случае блок только помечается для удаления и будет удален только после того, как не останется ни одного процесса, связанного с данной памятью.

Для просмотра всех созданных на данный момент областей общей памяти используется команда ipcs. Для этого надо запустить данную команду с ключом –m:

```
$ ipcs -m
------ Shared Memory Segments --------
key   shmid   owner   perms bytes nattch    status
0x01008cdb   1081344 user   666   1024      1
```

Параметры системы управления областями общей памяти определяются параметрами kernel.shmmni, kernel.shmmax и kernel.shmall.

Параметр kernel.shmmni определяет максимально возможное число сегментов памяти, параметр kernel.shmmax — максимальный размер сегмента в килобайтах, а параметр kernel.shmall задает максимальный объем памяти в килобайтах, который может быть выделен системе управления общей памятью.

Эти параметры также можно оперативно изменить, воспользовавшись файлами, расположенными в файловой системе /proc. Для этого можно использовать команды

```
echo 64536 > /proc/sys/kernel/shmmax
echo 3774873 > /proc/sys/kernel/shmall
```

Для того чтобы сохранить эти изменения в системе и после перезагрузки, необходимо вручную занести новые значения этих параметров в файл /etc/sysctl.conf или воспользоваться командой sysctl –w <paramer>=<value>.

Получить информацию о текущем значении параметров системы управления общей памятью можно с помощью команды ipcs –lm:

```
$ ipcs –lm
------ Shared Memory Limits --------
max number of segments = 4096           // SHMMNI
max seg size (kbytes) = 32768           // SHMMAX
max total shared memory (kbytes) = 8388608 // SHMALL
min seg size (bytes) = 1
```

Следующая программа иллюстрирует пример использования межпроцессного взаимодействия на основе общей памяти. В ней используется механизм семафоров для разграничения доступа к области общей памяти. Доступ к области общей памяти сначала получает клиент. Для этого начальное значение семафора устанавливается равным 1. Для освобождения области памяти клиентом и обеспечения доступа к ней сервера клиент устанавливает значение семафора в 2, а затем засыпает до тех пор, пока сервер, в свою очередь, не освободит область общей памяти и не установит значение семафора 1:

Server.c

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <time.h>

int main()
{
    int shm_descr, sem_descr, key;
    char* buf;
    struct sembuf lock[1] = {0, -2, 0};
    struct sembuf unlock[1] = {0, 1, 0};
    setvbuf(stdout, (char*)NULL, _IONBF, 0);
    // Генерируем ключ для общей памяти и для семафора
    if((key = ftok("./server", 1)) < 0)
    {
      perror("Server: Unable to get the key");
      return(1);
    }
    // Создаем область общей памяти
    if((shm_descr = shmget(key, 1024, IPC_CREAT |
    IPC_EXCL | 0666)) < 0)
    {
      perror("Server: Unable to create shared
      memory segment");
      return(1);
    }
    // Присоединяем область общей памяти к адресному
    //пространству сервера
```

```
if((buf = (char*)shmat(shm_descr, NULL, 0)) < 0)
{
  perror("Server: Unable to attach shared
  memory segment");
  return(1);
}
// Создаем семафор для синхронизации доступа к
// общей памяти
if((sem_descr = semget(key, 1, IPC_CREAT |
IPC_EXCL | 0666)) < 0)
{
  perror("Server: Unable to create semaphore");
  return(1);
}
// Устанавливаем начальное значение семафора в 1
if(semctl(sem_descr, 0, SETVAL, 1) < 0)
{
  perror("Server: Unable to initialize semaphore");
  return(1);
}
printf("Server started\n");
printf("Server: r> ");
// Читаем информацию из области общей памяти до
// тех пор, пока не получим команду "disconnect"
for(;;)
{
  // Приостанавливаем процесс, пока клиент не
  // запишет данные в общую память
  if(semop(sem_descr, lock, 1) < 0)
  {
    perror("Server: Unable to lock semaphore");
    return(1);
  }
  // Считываем данные из области общей памяти
  // и выводим на экран
  // Если получена команда "disconnect" —
  // завершаем цикл приема данных
  if(!strcmp(buf, "disconnect")) break;
  printf("%d:%s ", buf[0] - 'A' + 1, buf);
  // Записываем полученные данные обратно
  // в общую память
  buf[0] -= ('A' - 1);
  // Освобождаем доступ к области общей памяти
  // для клиента
  if(semop(sem_descr, unlock, 1) < 0)
```

```
    {
      perror("Server: Unable to unlock
      semaphore");
      return(1);
    }
  }
  printf("\n\n\n");
  // Отсоединяем область общей памяти
  if(shmdt(buf) < 0)
  {
    perror("Server: Unable to detach shared
    memory segment");
    return(1);
  }
  // Удаляем ранее созданную область общей памяти
  if(shmctl(shm_descr, IPC_RMID, NULL) < 0)
  {
    perror("Server: Unable to delete shared
    memory segment");
    return(1);
  }
  // Удаляем семафор
  if(semctl(sem_descr, 0, IPC_RMID) < 0)
  {
    perror("Server: Unable to delete semaphore");
    return(1);
  }
  return(0);
}
```

Client.c

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <time.h>

int main()
{
  int shm_descr, sem_descr, key, i;
  char* buf;
```

```
struct sembuf lock[1] = {0, -1, 0};
struct sembuf unlock[1] = {0, 2, 0};
struct sembuf local_unlock[1] = {0, 1, 0};

setvbuf(stdout, (char*)NULL, _IONBF, 0);
// Получаем ключ для общей памяти и для семафора
if((key = ftok("./server", 1)) < 0)
{
  perror("Client: Unable to get the key");
  return(1);
}
// Получаем дескриптор существующей области
// общей памяти
if((shm_descr = shmget(key, 1024, 0)) < 0)
{
  perror("Client: Unable to get shared memory
  segment");
  return(1);
}
// Присоединяем область общей памяти к адресному
// пространству клиента
if((buf = (char*)shmat(shm_descr, NULL, 0)) < 0)
{
  perror("Client: Unable to attach shared memory
  segment");
  return(1);
}
// Получаем дескриптор существующего семафора
if((sem_descr = semget(key, 1, 0)) < 0)
{
  perror("Client: Unable to get semaphore");
  return(1);
}//if
printf("Client started\n");
printf("Client: s> ");
for(i = 'A'; i <= 'Z'; ++i)
{
  // Блокируем область общей памяти и записываем
  // в нее данные для сервера
  if(semop(sem_descr, lock, 1) < 0)
  {
    perror("Client: Unable to lock semaphore");
    return(1);
  }
  buf[0] = (char)i; buf[1] = '\0';
```

```
    printf("%d:%s ", i - 'A' + 1, buf);
    // Освобождаем область общей памяти для сервера
    if(semop(sem_descr, unlock, 1) < 0)
    {
      perror("Client: Unable to unlock semaphore");
      return(1);
    }
    // Ждем, когда сервер освободит общую память
    // и проверяем ответ сервера
    if(semop(sem_descr, lock, 1) < 0)
    {
      perror("Client: Unable to lock
      semaphore");
      return(1);
    }
    if(buf[0] != i - 'A' + 1)
    {
      printf("Client: Wrong data received from
      server: expected \"%d\",
      received \"%d\"\n", i - 'A' + 1, buf[0]);
        return(1);
      }
      if(semop(sem_descr, local_unlock, 1) < 0)
      {
        perror("Client: Unable to unlock semaphore");
        return(1);
      }
  }
  printf("\n\n\n");
  // Блокируем доступ к области общей памяти и
  // записываем команду "disconnect"
  if(semop(sem_descr, lock, 1) < 0)
  {
    perror("Client: Unable to lock semaphore");
    return(1);
  }
  strcpy(buf, "disconnect");
  // Освобождаем область общей памяти для сервера
  if(semop(sem_descr, unlock, 1) < 0)
  {
    perror("Client: Unable to unlock semaphore");
    return(1);
  }
  // Отсоединяем область общей памяти от адресного
  // пространства клиента
```

```
    if(shmdt(buf) < 0)
    {
      perror("Client: Unable to detach shared
      memory segment");
      return(1);
    }
    return(0);
}
```

Запустим процесс-сервер как фоновый процесс, перенаправив операции вывода информации в файл отчета (`./server > report.txt 2 > &1 &`). Затем запустим процесс-клиент (`./client`). Протокол работы со стороны процесса-клиент в этом случае может выглядеть следующим образом:

```
Client started
Client: s> 1:A 2:B 3:C 4:D 5:E 6:F 7:G 8:H 9:I
10:J 11:K 12:L 13:M 14:N 15:O 16:P 17:Q 18:R 19:S
20:T 21:U 22:V 23:W 24:X 25:Y 26:Z
```

Результат работы процесса-сервера в этом случае, сохраненный в файле `report.txt`, будет иметь вид

```
Server started
Server: r> 1:A 2:B 3:C 4:D 5:E 6:F 7:G 8:H 9:I
10:J 11:K 12:L 13:M 14:N 15:O 16:P 17:Q 18:R 19:S
20:T 21:U 22:V 23:W 24:X 25:Y 26:Z
```

9.3.5. Каналы

Одним из самых простых способов организации межпроцессного взаимодействия являются каналы. Существует два типа каналов — неименованные и именованные (FIFO). Отличие между этими двумя типами каналов заключается в том, что неименованные каналы могут быть использованы только для связи процессов, являющихся потомками одного и того же родителя, который создал неименованный канал.

При этом для обоих видов каналов действуют специальные правила чтения и записи данных. При чтении меньшего числа данных, чем находится в канале, считывается требуемое количество байт, а остальные данные остаются доступными для дальнейших операций чтения. Если же запрашивается больше байт данных, чем содержит канал, то возвращается доступное количество байт. Если процесс пытается считать данные из пустого канала, то он блокируется до тех пор, пока в канале не появятся данные, доступные для считывания.

При записи в канал количества байт меньше, чем емкость канала, гарантируется неделимость данной операции. Это означает, что записываемый процессом блок данных будет записан целиком и не будет смешан с данными от других процессов. Если же в канал записывается больше байт, чем емкость канала, то в этом случае возможно перемешивание данных от разных процессов и неделимость операций не гарантируется.

Механизм организации межпроцессного взаимодействия на основе неименованных каналов часто используется для перенаправления выхода одного процесса на вход другого процесса. Например, для организации постраничного вывода списка процессов в системе можно использовать команду `ps aux | more`. Операция `|` используется именно для создания неименованного канала и перенаправления данных со стандартного потока вывода процесса `ps` в стандартный поток ввода процесса `more`.

Для создания неименованного канала используется системный вызов `pipe()`:

```
#include <unistd.h>
int pipe(int filedes[2]);
```

После вызова создается два файловых дескриптора, ассоциированных с созданным каналом, которые сохраняются в массиве `filedes`. При этом дескриптор `filedes[0]` предназначен для чтения данных из канала, а дескриптор `filedes[1]` — для записи. В случае, если операция создания канала завершается успешно, функция `pipe()` возвращает значение 0, в противном случае — значение –1.

Параметры каналов регулируются параметрами `OPEN_MAX` и `PIPE_BUF`. Параметр `OPEN_MAX` отвечает за максимальное количество дескрипторов файлов, открытых процессом, а параметр `PIPE_BUF` регулирует размер канала. Получить данные об этих параметрах можно из оболочки BASH с помощью команд `ulimit –n` для параметра `OPEN_MAX` и `ulimit –p` для параметра `PIPE_BUF` или с помощью команды `getconf –a`. Кроме того, максимальное число файловых дескрипторов, которое может быть использовано ядром системы, задается системными параметрами `fs.file-max` и `fs.inode-max`. Изменить эти параметры можно с помощью команды `sysctl –w fs.file-max=<value>` и `sysctl –w fs.inode-max=<value>`.

Каким образом два разных, казалось бы, процесса `ps` и `more` получают файловые дескрипторы, ассоциированные с каналом? На самом деле эти процессы не являются независимыми. Они имеют общего родителя, в качестве которого выступает оболочка пользователя. Именно она создает канал, а затем, используя связку `fork-exec`, запускает на выполнение оба процесса. При вы-

полнении системных вызовов `fork()` и `exec()` созданные процессы при прочих равных условиях наследуют файловые дескрипторы, открытые ранее родителем. Таким же образом наследуются и файловые дескрипторы, связанные с концами неименованного канала.

Как процессы догадываются, что они работают не независимо друг от друг, а должны переслать или принять данные от другого процесса? В действительности процессы ничего не знают о том, что они работают в связке. Задача обеспечения «прозрачного» перенаправления данных от одного процесса к другому также решается процессом-родителем. Для этого используется системный вызов `dup2()`:

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
```

Эта функция используется для того, чтобы создать копию старого дескриптора `oldfd` и привязать эту копию к дескриптору `newfd`. При этом дескриптор `newfd` закрывается, если ранее он был открыт. Если процессу удается создать копию дескриптора и связать ее с дескриптором `newfd`, то функция `dup2()` возвращает значение `newfd`. В противном случае, возвращается значение –1.

Задача процесса-родителя состоит в том, чтобы создать новые дескрипторы для стандартных потоков ввода и вывода, которые должны являться копиями дескрипторов, связанных с каналом. В результате процессы по-прежнему читают данные из стандартного потока ввода и записывают данные в стандартный поток вывода, хотя на самом деле осуществляется передача данных от одного процесса к другому через неименованный канал.

Следующий пример демонстрирует принцип работы операции `|`.

Pipe.c

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

  int main()
  {
  int status, filedes[2];
  setvbuf(stdout, (char*)NULL, _IONBF, 0);
  // Создаем неименованный канал
  if(pipe(filedes) < 0)
  {
    perror("Pipe: Unable to create pipe");
    return(1);
  }
  switch(fork())
```

```
{
  case(-1):
      perror("Pipe: Unable to create child
      process");
      return(1);
    case(0):
      // Первый потомок
      printf("Start more\n");
      // Закрываем неиспользуемый дескриптор для
      // записи
      // и создаем копию для дескриптора для
      // чтения,
      // ассоциированную с потоком стандартного
      // ввода
      close(filedes[1]);
      if(dup2(filedes[0], 0) < 0)
      {
        perror("Pipe-parent: Unable to duplicate
        read descriptor");
        return(1);
      }
      // Выполняем команду more
      execlp("more", "more", NULL);
      perror("Pipe-child: Unable to execute
      more");
      return(1);
    default:
      // Родитель
      switch(fork())
      {
        case(-1):
          perror("Pipe-parent: Unable to create
          child process");
          return(1);
        case(0):
          // Второй потомок
          printf("Start ps -aux\n");
          // Закрываем неиспользуемый дескриптор
          // для чтения
          // и создаем копию для дескриптора для
          // записи,
          // ассоциированную с потоком стандарт-
          // ного вывода
          close(filedes[0]);
          if(dup2(filedes[1], 1) < 0)
```

```
            {
              perror("Pipe-child: Unable to
              duplicate write descriptor");
              return(1);
            }
            execlp("ps", "ps", "aux", NULL);
            perror("Pipe-parent-child: Unable to
            execute ps");
            return(1);
          default:
            // Родитель
            // Закрываем оба дескриптора канала
            // и ожидаем завершения потомков
            close(filedes[0]);
            close(filedes[1]);
            wait(&status);
            break;
        }
        wait(&status);
        break;
    }
    return(0);
}
```

В данном примере выполняется команда, аналогичная команде `ps aux | more`. При этом организуется постраничный вывод данных обо всех процессах в системе.

В отличие от неименованных каналов, именованные (или FIFO) каналы могут использоваться процессами, независимо от взаимоотношений между ними. Это связано с тем, что FIFO-каналы представляют собой файлы специального вида и для работы с ними необходимо лишь наличие соответствующих прав доступа.

Для создания FIFO-каналов используется системный вызов `mkfifo()`:

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

Параметр `pathname` задает имя создаваемого FIFO-файла, а параметр `mode` — права доступа пользователей к создаваемому файлу. В случае, если удается создать новый FIFO-файл, то возвращается значение 0, в противном случае возвращается значение –1.

На механизм FIFO-каналов оказывают влияние те же параметры `OPEN_MAX`, `PIPE_BUF`, `fs.file-max` и `fs.inode-max`, что и на механизм неименованных каналов.

Поскольку FIFO-каналы являются файлами, то их список можно вывести с помощью команды `ls –l`:

```
$ ls -l
total 36
-rwxr-xr-x 1 rainman rainman 45 Feb 28 09:09 comp_run
-rwxr-xr-x 1 rainman rainman 42 Feb 28 09:08 comp_run~
prw-r--r-- 1 rainman rainman 0 Feb 28 10:41 fifo
-rwxr-xr-x 1 rainman rainman 10011 Feb 28 09:46 pipe
-rw-r--r-- 1 rainman rainman 1752 Feb 28 09:46 pipe.c
-rw-r--r-- 1 rainman rainman 1748 Feb 28 09:46 pipe.c~
-rw-r--r-- 1 rainman rainman 4121 Feb 28 09:35 tmp.log
```

В этом примере в текущем каталоге существует FIFO-канал с именем `fifo`.

Остальные операции для работы с FIFO-каналом аналогичны тем, что используются для открытия, чтения, записи и закрытия обычных файлов. Однако на FIFO-каналы также распространяются ограничения, связанные с неименованными каналами.

Прежде чем с каналом можно будет производить операции чтения/записи, необходимо, чтобы этот канал был открыт как для чтения, так и для записи. Если какой-либо процесс открывает канал для чтения, то он будет заблокирован до тех пор, пока другой процесс не откроет его на запись. То же самое справедливо и в случае, если процесс пытается открыть FIFO-канал для записи.

Если канал был открыт для записи с флагом `O_NDELAY` (без блокировки), ни один процесс не открыл данный FIFO-канал для чтения, и процесс пытается записать данные в этот канал, то процессу посылается сигнал `SIGPIPE`.

Если канал открывается для чтения с флагом `O_NDELAY` (без блокировки), ни один процесс не открыл данный FIFO-канал для записи, и процесс пытается считать данные из канала, то операция чтения вернет 0 байт.

Рассмотрим пример организации межпроцессного взаимодействия с помощью FIFO-каналов.

Server.c

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
// Файл для приема запросов от клиента
#define FIFOREQ "./fifo_req"
// Файл для отправки ответов клиенту
```

```
#define FIFOANS "./fifo_ans"

int main()
{
  int fr, fw, n;
  char buf[20];

  setvbuf(stdout, (char*)NULL, _IONBF, 0);
  // Создаем FIFO каналы для запросов и ответов
  if(mkfifo(FIFOREQ, 0666) < 0)
  {
    perror("Server: Unable to create fifo");
    return(1);
  }
  if(mkfifo(FIFOANS, 0666) < 0)
  {
    perror("Server: Unable to create fifo");
    return(1);
  }
  // Открываем канал для запросов на чтение и
  // канал ответов для записи
  if((fr = open(FIFOREQ, O_RDONLY)) < 0)
  {
    perror("Server: Unable to open fifo for reading");
    return(1);
  }
  if((fw = open(FIFOANS, O_WRONLY)) < 0)
  {
    perror("Server: Unable to open fifo for writing");
    return(1);
  }
  printf("Server started\n");
  printf("Server: r> ");
  for(;;)
  {
    // Получаем запрос от клиента
    if((n = read(fr, buf, 19)) < 0)
    {
      perror("Server: Unable to read data from fifo");
      return(1);
    }
    buf[n] = '\0';
    // Если запрос — "disconnect", то завершаем
    // работу
    if(!strcmp(buf, "disconnect")) break;
```

```
    printf("%d:%s ", buf[0] - 'A' + 1, buf);
    // Если запрос — не "disconnect", то пересылаем
    // ответ клиенту
    if(write(fw, buf, strlen(buf)) < 0)
    {
      perror("Server: Unable to write data to fifo");
      return(1);
    }
  }
  printf("\n\n\n");
  // Закрываем файловые дескрипторы и удаляем
  // файлы FIFO
  close(fr);
  close(fw);
  if(unlink(FIFOREQ) < 0)
  {
    perror("Server: Unable to delete fifo");
    return(1);
  }
  if(unlink(FIFOANS) < 0)
  {
    perror("Server: Unable to delete fifo");
    return(1);
  }
  return(0);
}
```

Client.c

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
// Файл для отправки запросов серверу
#define FIFOREQ "./fifo_req"
// Файл для приема ответов от сервера
#define FIFOANS "./fifo_ans"

int main()
{
  int fr, fw, i, n;
  char buf[20];

  setvbuf(stdout, (char*)NULL, _IONBF, 0);
```

```
// Открываем каналы для отправки запросов на
// запись и для приема ответов на чтение
if((fw = open(FIFOREQ, O_WRONLY)) < 0)
{
  perror("Client: Unable to open fifo for writing");
  return(1);
}
if((fr = open(FIFOANS, O_RDONLY)) < 0)
{
  perror("Client: Unable to open fifo for reading");
  return(1);
}
printf("Client started\n");
printf("Client: s> ");
for(i = 'A'; i <= 'Z'; ++i)
{
  buf[0] = (char)i; buf[1] = '\0';
  printf("%d:%s ", i - 'A' + 1, buf);
  // Отсылаем запрос серверу
  if(write(fw, buf, strlen(buf)) < 0)
  {
    perror("Client: Unable to write data to fifo");
    return(1);
  }
  // Получаем ответ от сервера
  if((n = read(fr, buf, 19)) < 0)
  {
    perror("Client: Unable to read data from fifo");
    return(1);
  }
  buf[n] = '\0';
  // Проверяем корректность ответа
  if(buf[0] != (char)i || strlen(buf) != 1)
  {
    printf("Client: Wrong data received from
    server: expected \"%c\", received \"%s\"\n",
    (char)i, buf);
    return(1);
  }
}
// Отсылаем команду завершения соединения
strcpy(buf, "disconnect");
if(write(fw, buf, strlen(buf)) < 0)
{
  perror("Client: Unable to write data to fifo");
```

```
    return(1);
  }
  printf("\n\n\n");

  close(fr);
  close(fw);
  return(0);
}
```

Запустим процесс-сервер как фоновый процесс, перенаправив операции вывода информации в файл отчета (`./server > report.txt 2 > &1 &`). Затем запустим процесс-клиент (`./client`). Протокол работы со стороны процесса-клиента в этом случае может выглядеть следующим образом:

```
Client started
Client: s> 1:A 2:B 3:C 4:D 5:E 6:F 7:G 8:H 9:I
10:J 11:K 12:L 13:M 14:N 15:O 16:P 17:Q 18:R 19:S
20:T 21:U 22:V 23:W 24:X 25:Y 26:Z
```

Результат работы процесса-сервера в этом случае, сохраненный в файле `report.txt`, будет иметь следующий вид:

```
Server started
Server: r> 1:A 2:B 3:C 4:D 5:E 6:F 7:G 8:H 9:I
10:J 11:K 12:L 13:M 14:N 15:O 16:P 17:Q 18:R 19:S
20:T 21:U 22:V 23:W 24:X 25:Y 26:Z
```

9.3.6. Сокеты

Основные понятия. Механизм межпроцессного взаимодействия на основе *сокетов* впервые был введен в версии UNIX 4.2 BSD. Этот механизм разрабатывался в целях унификации методов и средств межпроцессного и сетевого взаимодействия с механизмами системы ввода/вывода UNIX на основе файловых дескрипторов.

Как и все остальные механизмы межпроцессного взаимодействия, механизм сокетов определяет *коммуникационные объекты* для организации взаимодействия, *интерфейс* для работы с данными объектами и *протокол* взаимодействия. Коммуникационными объектами в этой модели взаимодействия являются сами сокеты, которые можно трактовать как псевдофайлы, существующие в рамках некоторой виртуальной файловой системы. Интерфейс определяет набор системных вызовов для работы с сокетами. Протокол описывает порядок вызова системных вызовов для корректной организации взаимодействия процессов.

Сокеты реализуют механизм двунаправленной передачи данных типа «точка — точка». Сами сокеты представляют собой конечную точку механизма межпроцессного взаимодействия, которой приписываются имя, тип и один или несколько связанных с ним процессов.

Механизм межпроцессного взаимодействия на основе сокетов основан на модели *клиент-сервер*, согласно которой один процесс, называемый сервером, создает сокет, известный другим процессам-клиентам. Процессы-клиенты взаимодействуют с сервером, соединяясь с именованным сокетом на стороне сервера. Для этого клиент создает неименованный сокет и запрашивает соединение с именованным сокетом сервера. В случае успешной обработки данного запроса возвращается два именованных сокета — один для клиента и один для сервера, которые уже могут использоваться для чтения или записи данных.

Сокеты существуют в рамках *коммуникационных доменов*. Коммуникационный домен представляет собой абстракцию, которая предоставляет структуры данных для адресации сокетов, и транспортные протоколы для передачи данных. Сам коммуникационный домен можно трактовать как некоторую виртуальную файловую систему. Сокеты могут связываться с другими сокетами только в том случае, если они принадлежат одному и тому же коммуникационному домену.

В ОС UNIX поддерживается несколько типов коммуникационных доменов. Каждый домен предоставляет свои собственные способы адресации сокетов и коммуникационные протоколы. Поэтому каждый домен имеет имя, начинающееся с префикса `AF_` (Address Family), или аналогичное имя, но с префиксом `PF_` (Protocol Family) (табл. 9.2).

Таблица 9.2. **Коммуникационные домены в UNIX**

Домен	Описание
`AF_UNIX`, `AF_LOCAL`	Организация межпроцессного взаимодействия в системе
`AF_INET`	Организация сетевого взаимодействия на основе протоколов Internet (TCP/UDP/IPv4)
`AF_IPX`	Организация сетевого взаимодействия на основе протоколов Novell IPX
`AF_PACKET`	Взаимодействие на уровне интерфейсов устройств

Таблица 9.3. **Некоторые типы сокетов**

Тип сокета	Описание
SOCK_STREAM	Обеспечивает надежную, двунаправленную последовательную и неповторяемую передачу данных без разбиения на пакеты, с поддержкой соединений
SOCK_DGRAM	Обеспечивает ненадежные сообщения с ограниченной максимальной длиной, без поддержки установления соединения
SOCK_SEQPACKET	Обеспечивает надежную, двунаправленную последовательную и неповторяемую передачу пакетов данных, с поддержкой соединений
SOCK_RAW	Предоставляет доступ к протоколам низкого уровня

В дальнейшем будет рассматриваться взаимодействие на уровне сокетов на примере коммуникационного домена AF_UNIX, отвечающего за организацию межпроцессного взаимодействия.

Помимо того, что сокет принадлежит некоторому коммуникационному домену, также каждый сокет имеет тип, задающий семантику коммуникации (табл. 9.3).

Коммуникационные домены могут поддерживать не обязательно все типы сокетов, а только некоторые из них.

Системные вызовы для работы с сокетами. Для создания сокетов используется системный вызов socket():

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

В этом системном вызове присутствует третий параметр — protocol, который определяет *коммуникационный протокол* передачи данных между сокетами. Обычно существует только один протокол, ассоциированный с тем или иным типом сокета. Поэтому в большинстве случаев можно предоставить системе выбирать протокол, указав в качестве третьего параметра 0. Однако некоторые типов сокетов поддерживают несколько типов протоколов. В этом случае в явном виде указывается тип протокола. Например, для протокола TCP указывается значение IPPROTO_TCP, а для протокола IP — IPPROTO_IP. Для коммуникационного домена AF_UNIX для типов сокетов SOCK_STREAM и SOCK_DGRAM должно указываться значение 0.

Если системе удается создать сокет, то возвращается дескриптор созданного сокета, в противном случае возвращается –1.

Количество открытых сокетов ограничивается максимальным числом файловых дескрипторов, открытых процессом, и может настраиваться администратором системы.

Созданный сокет еще нельзя использовать для передачи данных. Это связано с тем, что созданный сокет не может быть идентифицирован другими процессами, так как не имеет назначенного имени. Поэтому, прежде всего, необходимо назначить для сокета локальное имя, которое будет использоваться для его адресации.

Процесс связывания сокета с файлом различается для различных типов сокетов данных и для процессов-серверов и процессов-клиентов. Это обусловлено как тем, что разные типы сокетов поддерживают различные модели передачи данных (как, например, сокеты типа `SOCK_STREAM` используются для передачи данных с установлением соединения, а сокеты типа `SOCK_DGRAM` — без установления соединения), так и тем, что серверы и клиенты имеют разную операционную семантику. Модели передачи данных для сокетов с установлением соединений и без установления соединений показаны на рис. 9.9.

Для начального назначения сокету локального адреса используется системный вызов `bind()`:

```
#include <sys/types.h>
#include <sys/socket.h>
int bind(int sockfd, struct sockaddr *my_addr,
socklen_t addrlen);
```

В качестве первого параметра указывается дескриптор сокета, возвращаемый системным вызовом `socket()`. Второй параметр указывает на структуру, хранящую имя сокета, а третий параметр содержит размер данной структуры. Столь сложная система именования сокетов связана с тем, что в различных коммуникационных доменах используется разная система адресации сокетов. Соответственно различаются и размеры структур, предназначенных для задания имен сокетов. Для домена `AF_UNIX` эта структура обычно имеет следующий вид:

```
#include <sys/un.h>
struct sockadr_un{
   short int sun_family;
   char sun_path[108];
};
```

Первое поле данной структуры является общим для различных коммуникационных доменов и используется для идентификации самого домена. В случае организации межпроцессного взаимодействия этому полю должно быть присвоено значение `AF_UNIX`.

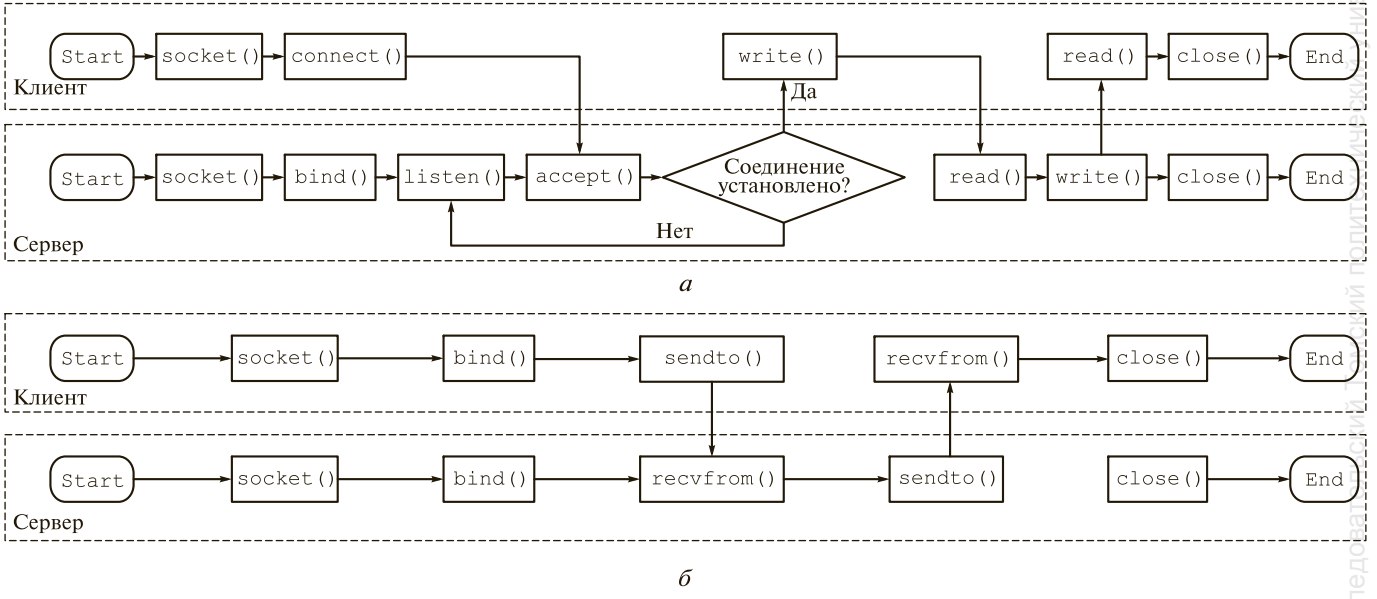

Рис. 9.9. Модели передачи данных:

а — с установлением соединений; *б* — без установления соединений

Второе поле содержит имя файла, ассоциированного с сокетом. Следует заметить, что данный файл автоматически создается при вызове `bind()` как файл типа сокет в рамках файловой системы UNIX. Соответственно данный файл не должен существовать до выполнения операции создания сокета и пользователь должен обладать необходимыми правами для создания ассоциированного с сокетом файла. По окончании работы с сокетом данный файл должен быть удален самим пользователем с помощью системного вызова `unlink()`.

Системный вызов `listen()` используется сервером для прослушивания запросов на установление соединений от процессов-клиентов при использовании передачи данных с установлением соединений:

```
#include <sys/socket.h>
int listen(int s, int backlog);
```

Первый параметр должен представлять собой дескриптор ранее созданного сокета, для которого был назначен локальный адрес. Этот сокет не используется для передачи данных, а служит только для приема запросов от клиентов на соединение.

Второй параметр задает максимальную длину, до которой может расти очередь ожидающих соединений. Если очередь полна и приходит запрос на соединение от клиента, то данный запрос будет отвергнут, а клиент получит сообщение об отказе в соединении.

Если операция постановки сокета на прослушивание запросов на соединение завершилась успешно, то возвращается значение 0. В противном случае возвращается значение –1.

Для приема очередного запроса и установления соединения с клиентом используется функция `accept()`:

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int s, struct sockaddr *addr, socklen_t
*addrlen);
```

Первый параметр должен принимать значение дескриптора сокета, ранее поставленного на прослушивание запросов на установление соединений. Второй параметр должен содержать адрес структуры, в которую будет записан адрес клиента. Формат этой структуры полностью определяется тем коммуникационным доменом, в рамках которого организуется взаимодействие процессов. Для домена `AF_UNIX` адрес должен иметь тип `struct sockadr_un`. Третий параметр содержит адрес переменной, начальное значение которой должно быть равно размеру структуры данных для адреса. Функция `accept()` записывает в эту переменную действительный размер адреса клиентского сокета. Следует заметить, что

второй и третий параметры являются необязательными. Если второй параметр установлен в значение NULL, то адрес не заполняется.

Функция accept() извлекает первый запрос на соединение из очереди запросов и создает новый сокет почти с такими же параметрами, что и у сокета, на который указывает дескриптор s. Единственное отличие этого сокета от оригинального состоит в том, что созданный сокет связан с сокетом на стороне клиента и именно он используется для передачи и приема данных. Если очередь запросов пуста, то функция accept() по умолчанию блокирует процесс-сервер до появления первого запроса на соединение.

Функция accept() возвращает значение –1, если не удалось установить соединение. Если же новый сокет был создан и соединение было установлено, то возвращается дескриптор созданного сокета.

Для установления соединения сокета клиента с сокетом сервера используется функция connect():

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr
*serv_addr, socklen_t addrlen);
```

Первый параметр sockfd должен содержать дескриптор ранее созданного на стороне клиента сокета. Второй позиционный параметр serv_addr должен указывать на адрес сервера, который будет использоваться для приема и передачи данных, а третий параметр — содержать длину удаленного адреса.

Если удается установить соединение с сервером, то функция connect возвращает нулевое значение. Если же установить соединение по той или иной причине не удалось, то возвращается значение –1.

Функция connect() может использоваться не только в случае взаимодействия с установлением соединения (для сокетов SOCK_STREAM), но и для взаимодействия без установления соединения (сокеты SOCK_DGRAM). В этом случае изменяется семантика данного системного вызова и соединение не устанавливается, но адрес сервера, указанный в качестве второго параметра, используется в качестве адреса по умолчанию для приема и передачи данных.

Кроме этого, существует еще одно отличие в случае использования функции connect() для организации межпроцессного взаимодействия. Для сокетов с установлением соединения функция connect() может вызываться только один раз, т.е. каждый сокет типа SOCK_STREAM может связываться с единственным сокетом на стороне сервера.

Для сокетов типа SOCK_DGRAM такой системный вызов может использоваться несколько раз для связи с разными сокетами и разными серверами. Для этого надо лишь вызвать функцию connect() еще раз, но с адресом другого сокета, с которым требуется организовать взаимодействие. Если же необходимо просто прекратить связь с сокетом на стороне сервера без установления новой связи, то для этого достаточно установить поле sa_family в структуре sockaddr в значение AF_UNSPEC и вызвать функцию connect() с этим адресом.

Для передачи и приема данных через сокеты существует целое семейство системных вызовов. В это семейство входят не только системные вызовы, предназначенные для работы исключительно с интерфейсом сокетов, но и системные вызовы для записи и чтения данных с использованием интерфейсов файловых дескрипторов. Подобная совместимость была достигнута за счет того, что механизм работы с файловыми дескрипторами инкапсулирует интерфейс взаимодействия с сокетами. Это позволяет использовать стандартные системные вызовы read(), write(), readv(), writev(), sendfile().

Помимо перечисленных выше функций, прием и передача данных может осуществляться с помощью функций recv(), send(), recvfrom(), sendto():

```
#include <sys/types.h>
#include <sys/socket.h>
int send(int s, const void *msg, size_t len, int
flags);
int sendto(int s, const void *msg, size_t len,
int flags, const struct sockaddr *to, socklen_t tolen);
int recv(int s, void *buf, size_t len, int flags);
int recvfrom(int s, void *buf, size_t len, int
flags, struct sockaddr *from, socklen_t *fromlen);
```

Функции send() и sendto() предназначены для передачи данных. Первый параметр s должен содержать дескриптор того сокета, через который осуществляется взаимодействие процессов. Второй параметр msg должен содержать указатель на данные, которые необходимо передать, а размер этих данных указывается в третьем параметре len. Параметр flags содержит дополнительные атрибуты, отвечающие за управление передачей данных. Так, например, если данный флаг будет установлен в значение MSG_OOB, то передаваемый блок данных будет передан во внеочередном порядке.

Помимо этих параметров функция sendto() имеет еще два дополнительных параметра to и tolen. Данные параметры используются для того, чтобы явно адресовать удаленный сокет, которому должен быть доставлен заданный блок данных.

Если удалось передать данные через сокет, то функции `send()` и `sendto()` возвращают количество отправленных символов. Если же в процессе передачи произошла ошибка, то возвращается значение –1. При этом следует заметить, что эти функции не проверяют тот факт, было ли доставлено отправленное сообщение, т. е. то, что сообщение было передано через сокет, не означает, что переданное сообщение было принято другим сокетом. Таким образом, функции `send()` и `sendto()` обрабатывают только локальные ошибки, произошедшие в связи со сбоем передачи данных на локальном сокете.

Функции `recv()` и `recvfrom()` предназначены для приема данных. Первый параметр `s` должен содержать дескриптор настроенного сокета, второй параметр `buf` — указатель на буфер, в который будут записываться принимаемые данные. Максимальный размер данных, который может быть записан в этот буфер, задается в третьем параметре `len`. Параметр `flags`, так же как и для функций `send()`, содержит дополнительные атрибуты, отвечающие за управление получением данных.

Функция `recvfrom()` имеет два дополнительных параметра `from` и `fromlen`, которые используются для записи адреса того удаленного сокета, от которого было получено сообщение.

Некоторые из перечисленных функций могут применяться только при определенном виде взаимодействия. Такие функции, как `read()`, `write()`, `readv()`, `writev()`, `sendfile()`, `send()` и `recv()` используются только при организации взаимодействия с установлением соединения, а функции `sendto()` и `recvfrom()` могут применяться как для взаимодействия с установлением соединения, так и без.

В случае, если используется взаимодействие с установлением соединения, то пользователь может в любой момент закрыть часть или все дуплексное соединение на сокете. Для этого предназначена функция `shutdown()`:

```
#include <sys/socket.h>
int shutdown(int s, int how);
```

Параметр `s` должен быть установлен в значение дескриптора ранее открытого сокета, для которого установлено соединение. Параметр `how` контролирует, какая часть канала должна быть закрыта. Если этот параметр принимает значение `SHUT_RD`, то запрещается прием данных на сокете, если принимает значение `SHUT_WR`, то запрещается передача, а если `SHUT_RDWR`, то запрещается и прием и передача.

Если операция закрытия части или всего канала завершается успешно, то возвращается значение 0, в случае возникновения ошибки — значение –1.

Для закрытия сокетов используется тот же системный вызов, что и для закрытия дескрипторов файлов `close()`:

```
#include <unistd.h>
int close(int fd);
```

Рассмотрим пример использования приведенных системных вызовов для организации межпроцессного взаимодействия с установлением соединения. В приведенном ниже примере существует два процесса — сервер и клиент. Клиент пересылает серверу сообщения в виде букв латинского алфавита. Сервер ожидает очередного символа от клиента и пересылает его обратно. Процесс продолжается до тех пор, пока клиент не пошлет команду на закрытие соединения `disconnect`:

Server.c

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#define SOCKET_FILE "./socket.file"

int main()
{
  int s, s2, t, len;
  pid_t pid;
  struct sockaddr_un local, remote;
  char str[100];
  // Создаем сокет на стороне сервера
  if((s = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)
  {
    perror("Server: Unable to create socket");
    exit(1);
  }
  // Заполняем адрес сокета сервера
  local.sun_family = AF_UNIX;
  strcpy(local.sun_path, SOCKET_FILE);
  // Удаляем файл сокета, если он существовал ранее
  unlink(local.sun_path);
  len = strlen(local.sun_path) + sizeof(local.sun_
  family);
  // Устанавливаем адрес сокета (связываем его с
  // файлом)
```

```
if(bind(s, (struct sockaddr *)&local, len) < 0)
{
  perror("Server: Unable to bind socket with the
  file");
  exit(1);
}
// Ставим сокет на прослушивание запросов на
// установление соединений
if(listen(s, 5) < 0)
{
  perror("Server: Error while trying to
  listen");
  exit(1);
}
str[0] = '\0';
while(!strcmp(str, "disconnect"))
{
  int done, n;
  printf("Server: Waiting for incoming
  connection...\n");
  t = sizeof(remote);
  // Выбираем запрос из очереди и устанавливаем
  // соединение
  if((s2 = accept(s, (struct sockaddr *)&remote,
  &t)) < 0)
  {
    perror("Server: Error while trying to accept");
    exit(1);
  }
  printf("Server: Connected.\n");
  printf("Server: r> ");
  for(;;)
  {
    // Принимаем данные от клиента
    if(read(s2, str, 100) < 0)
    {
      perror("Server: Unable to receive data
      from client");
      exit(1);
    }
    // Если клиент прислал команду "disconnect"
    // — закрываем соединение
    if(!strcmp(str, "disconnect")) break;
    printf("%d:%s ", str[0] - 'A' + 1, str);
    // Пересылаем строку назад клиенту
```

```
      if(write(s2, str, strlen(str)) < 0)

        perror("Server: Unable to send data to
        client");
        exit(1);
      }
    }
    printf("\n\n\n");
    // Закрываем соединение на стороне сервера
    if(shutdown(s2, SHUT_RDWR) < 0)
    {
      perror("Server: Unable to shutdown the
      connection");
      exit(1);
    }
    // Закрываем дескриптор сокета, предназначенного
    // для передачи данных
    close(s2);
    }
  // Закрываем дескриптор сокета для приема данных
  close(s);
  // Удаляем файл, ассоциированный с сокетом
  if(unlink(SOCK_PATH) < 0)
  {
    perror("Server: Unable to delete socket
    file");
    exit(1);
  }
  return(0);
}
```

Client.c

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#define SOCKET_FILE "./socket.file"

int main()
{
  int s, t, len, i;
  struct sockaddr_un remote;
```

```
char str[100];
// Создаем сокет на стороне клиента
if((s = socket(AF_UNIX, SOCK_STREAM, 0)) == -1)
{
  perror("Client: Unable to create socket");
  exit(1);
}
printf("Client: Trying to connect...\n");
// Заполняем адрес для сокета сервера
remote.sun_family = AF_UNIX;
strcpy(remote.sun_path, SOCKET_FILE);
len = strlen(remote.sun_path) + sizeof(remote.
sun_family);
// Устанавливаем соединение с сокетом сервера
if(connect(s, (struct sockaddr *)&remote, len) == -1)
{
  perror("Client: Unable to connect with the
  server");
  exit(1);
}
printf("Client: Connected.\n");
printf("Client: s> ");
for(i = 'A'; i <= 'Z'; ++i)
{
  str[0] = (char)i; str[1] = '\0';
  printf("%d:%s ", i - 'A' + 1, str);
  // Пытаемся отослать очередной символ серверу
  if(write(s, str, strlen(str)) < 0)
  {
    perror("Client: Unable to send data to
    server");
    exit(1);
  }
  // Принимаем ответ от сервера
  if(read(s, str, 100) < 0)
  {
    perror("Client: Unable to receive data from
    server");
    exit(1);
  }
  // Проверяем, что сервер отослал символ обратно
  if(str[0] != (char)i || strlen(str) != 1)
  {
    printf("Client: Wrong data received from
    server: expected \"%c\", received \"%s\"\n",
```

```
      (char)i, str);
      exit(1);
    }
  }
  printf("\n\n\n");
  // Посылаем серверу команду на закрытие соеди-
  // нения и завершение сервера
  strcpy(str, "disconnect");
  if(write(s, str, strlen(str)) < 0)
  {
    perror("Client: Unable to send data to
    server");
    exit(1);
  }
  // Закрываем соединение на стороне клиента
  if(shutdown(s, SHUT_RDWR) < 0)
  {
    perror("Client: Unable to shutdown the
    connection");
    exit(1);
  }
  // Закрываем дескриптор сокета на стороне клиента
  close(s);
  return(0);
}
```

Скомпилируем и запустим процесс-сервер как фоновый процесс, перенаправив операции вывода информации в файл отчета (`./server > report.txt 2 > &1 &`). Затем запустим процесс-клиент (`./client`). Протокол работы со стороны процесса-клиента в этом случае может выглядеть следующим образом:

```
Client: Trying to connect...
Client: Connected.
Client: s> 1:A 2:B 3:C 4:D 5:E 6:F 7:G 8:H 9:I
10:J 11:K 12:L 13:M 14:N 15:O 16:P 17:Q 18:R 19:S
20:T 21:U 22:V 23:W 24:X 25:Y 26:Z
```

Результат работы процесса-сервера в этом случае, сохраненный в файле `report.txt`, будет иметь следующий вид:

```
Server: Waiting for incoming connection...
Server: Connected.
Server: r> 1:A 2:B 3:C 4:D 5:E 6:F 7:G 8:H 9:I
10:J 11:K 12:L 13:M 14:N 15:O 16:P 17:Q 18:R 19:S
20:T 21:U 22:V 23:W 24:X 25:Y 26:Z
```

9.4. Межпроцессное взаимодействие в Windows

Операционные системы семейства Windows, особенно системы на основе ядра Windows NT (Windows 2000, Windows XP, Windows Vista), предоставляют широкий набор средств для синхронизации процессов и организации межпроцессного взаимодействия. Рассмотрим некоторые из существующих методов организации взаимодействия в ОС семейства Windows.

9.4.1. Процессы и потоки

Как и в других ОС в Windows, *процессы* (*process*) являются основными объектами, позволяющими пользователям решать их задачи. Каждый процесс предоставляет ресурсы, необходимые для выполнения соответствующего приложения, а также имеет ассоциированные с ним виртуальное адресное пространство; исполняемый код; дескрипторы, связанные с открытыми системными объектами; контекст безопасности; уникальный идентификатор процесса; переменные окружения; класс приоритета; минимальный и максимальный размер доступной процессу виртуальной памяти; по крайней мере, один поток управления.

При старте процесс запускает один единственный *поток управления* (*thread*), называемый *первичным потоком* (*primary thread*). При этом каждый поток может создавать новый поток управления. В этом смысле процесс представляет собой контейнер, в котором инкапсулируется множество потоков, выполняющих основную вычислительную работу.

Все потоки процесса делят между собой его виртуальное адресное пространство и системные ресурсы. Кроме этого, каждый поток управления имеет собственные обработчики исключительных ситуаций, приоритет, локальную память потока, уникальный идентификатор потока, и данные о текущем контексте потока. *Контекст потока* (*thread context*) включает текущие значения регистров процессора, стек вызовов ядра, блок окружения потока, содержащий данные о размере стека потока, и пользовательский стек в адресном пространстве родительского процесса. Кроме того, потоки могут иметь собственные контексты безопасности для обработки случаев, когда поток заимствует права, а не наследует их от родительского процесса.

Операционные системы, основанные на ядре Windows NT (Windows NT 3.x-4.0, Windows 2000, Windows XP, Windows Vista и все серверные ОС семейства Windows) и на ядре Windows 9x (Windows 95, Windows 98 и Windows ME), поддерживают так называе-

мую *вытесняющую многозадачность* (*preemptive multitasking*). Она позволяет создавать эффект одновременного выполнения нескольких потоков в нескольких процессах. В более ранних системах Windows (например, семейства Windows 3.x) поддерживалась более простая модель одновременного выполнения потоков, основанная на *невытесняющей многозадачности* (*nonpreemptive multitasking*).

При вытесняющей многозадачности ОС, а точнее специальный системный процесс, называемый *вытесняющим планировщиком* (*preemptive scheduler*), временно прерывает текущий процесс по истечении времени, выделенного процессу для работы, переводя его в приостановленное состояние. Затем планировщик пробуждает один из приостановленных ранее процессов в зависимости от их приоритетов и выделяет квант процессорного времени для этого процесса. Такой механизм носит название *переключения контекста* (*context switching*). При невытесняющей многозадачности в памяти одновременно могут присутствовать несколько процессов, но процессорное время выделяется только основному процессу до тех пор, пока сам процесс или пользователь не освободит процессор.

В многопроцессорных системах ОС на основе ядра Windows NT позволяют одновременно выполнять столько потоков, сколько процессоров (или ядер) установлено в системе. В этом случае реализуется реальная многозадачность, а не ее имитация.

Для порождения новых процессов используется функция `CreateProcess()`:

```
include <windows.h>
BOOL WINAPI CreateProcess(
  LPCTSTR lpApplicationName,
  LPTSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCTSTR lpCurrentDirectory,
  LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation);
```

Процесс, вызвавший функцию `CreateProcess()`, называется процессом-родителем, а созданный в результате вызова этой функции процесс — процессом-потомком. Процесс-потомок полностью независим от породившего его процесса, но процесс-родитель получает возможность следить за порожденным процессом и отслеживать некоторые события, связанные с ним.

Параметр lpApplicationName задает имя программы, которая должна быть выполнена. Если этот параметр равен NULL, то имя запускаемой программы должно задаваться в параметре lpCommandLine. В данном параметре также задаются и параметры, передаваемые запускаемой программе. Если параметр lpCommandLine имеет значение NULL, то запускаемая программа берется из параметра lpApplicationName. Если обе строки не равны NULL, то параметр lpApplicationName задает запускаемую программу, а в параметре lpCommandLine передается список параметров для данной программы, разделенных пробелами.

Параметр lpProcessAttributes используется для задания создаваемому процессу прав доступа, отличных прав по умолчанию. Кроме того, один из элементов структуры, на которую он указывает, используется для обозначения того, что дескриптор процесса, созданного в результате вызова функции CreateProcess(), может быть унаследован процессами-потомками.

Параметр lpThreadAttributes используется для того же, что и параметр lpProcessAttributes. Однако, если параметр lpProcessAttributes предназначен для изменения параметров создаваемого процесса, то данные, передаваемые в параметре lpThreadAttributes, используются для изменения параметров первичного потока создаваемого процесса.

Параметр bInheritHandles используется для указания того, должен ли дочерний процесс наследовать все наследуемые дескрипторы от процесса-родителя (TRUE) или нет (FALSE). При этом наследуются не только дескрипторы открытых файлов, но и дескрипторы созданных процессов, каналов и других системных ресурсов. Унаследованные дескрипторы имеют те же самые значения и те же самые права доступа. Следует отметить, что наследуются не все дескрипторы, а только те, которые помечены как наследуемые. Это свойство дескрипторов очень важно при организации межпроцессного взаимодействия.

Параметр dwCreationFlags предназначен для задания класса приоритета создаваемого процесса, а также используется для управления свойствами процесса. Так, например, если и процесс-родитель, и создаваемый процесс являются консольными приложениями и в параметре передается значение CREATE_NEW_CONSOLE, то созданный процесс будет иметь свое собственное консольное окно. Без указания данного параметра новое окно не создается, а созданный процесс наследует консольное окно процесса-родителя.

Параметр lpEnvironment используется для изменения значений переменных окружения создаваемого процесса и содержит указатель на блок окружения для нового процесса. Этот блок

представляет собой блок, завершающийся нулем, состоящий из строк вида:

```
name=value\0
```

Если параметр `lpEnvironment` имеет значение `NULL`, то окружение наследуется от процесса-родителя.

Параметр `lpCurrentDirectory` используется для задания текущего диска и каталога для создаваемого процесса. Если он равен `NULL`, то текущий диск и каталог наследуются от процесса-родителя.

Параметр `lpStartupInfo` также предназначен для изменения характеристик создаваемого процесса. Он позволяет задавать начальные координаты окна создаваемого процесса, определяет, следует ли создать окно видимым или его нужно скрыть, а также позволяет переопределять дескрипторы стандартных устройств ввода/вывода консольных приложений для перехвата данных, выводимых на консоль процессом-потомком, процессом-родителем или перенаправления этих данных в файл или на устройство. Этот параметр позволяет изменять и другие свойства создаваемого процесса.

Параметр `lpProcessInformation` является возвращаемым параметром. В нем возвращаются дескрипторы и идентификаторы созданного процесса и его первичного потока. Этот параметр возвращает 0, если по какой-либо причине не удалось создать новый процесс, либо значение отличное от 0, если процесс был успешно создан.

Причной отказа системы в создании нового процесса может быть исчерпание виртуальной памяти, переполнение таблицы процессов или тот факт, что запускаемая программа не является программой или скриптом. Кроме того, на возможность порождения нового процесса влияет параметр реестра `HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows\UserProcessHandleQuota`, отвечающий за ограничение на количество дескрипторов для одного процесса. По умолчанию этот параметр в системах Windows XP и Vista установлен в значение 10 000, но администратор системы может изменить это значение и таким образом изменить ограничения на количество запускаемых процессов.

Также накладывается системное ограничение на длину итоговой команды для запуска процесса. Она не должна превышать 32 767 символов в длину.

Для создания новых потоков используется функция `CreateThread()`:

```
include <windows.h>
HANDLE WINAPI CreateThread(
```

```
LPSECURITY_ATTRIBUTES lpThreadAttributes,
SIZE_T dwStackSize,
LPTHREAD_START_ROUTINE lpStartAddress,
LPVOID lpParameter,
DWORD dwCreationFlags,
LPDWORD lpThreadId);
```

Параметр `lpThreadAttributes` используется для задания создаваемому потоку прав доступа, отличных от прав по умолчанию. Также этот параметр используется для указания того, что дескриптор созданного потока может быть унаследован процессами-потомками. Если же параметр установлен в значение `NULL`, то поток получает права по умолчанию, а дескриптор потока не может наследоваться потомками.

Параметр `dwStackSize` предназначен для задания начального размера стека потока в байтах. Если же в параметр передается значение 0, то размер стека определяется размером стека по умолчанию, определенному для приложения.

В параметре `lpStartAddress` передается адрес локальной функции приложения, которая будет выполняться потоком.

Параметр `lpParameter` предназначен для передачи создаваемому потоку входных значений, необходимых для основной функции потока.

Параметр `dwCreationFlags` управляет созданием потока. Например, если он установлен в значение `CREATE_SUSPENDED`, то созданный поток переводится в приостановленное состояние до тех пор, пока головной другой поток не возобновит его.

Возвращаемый параметр `lpThreadId` используется для получения идентификатора созданного потока. Если он равен `NULL`, то идентификатор созданного потока не возвращается. Если вызов функции закончился созданием нового потока, то функция возвращает дескриптор созданного потока. В противном случае возвращается значение `NULL`.

Причиной неуспешной попытки создать новый поток может являться исчерпание ресурсов системы или исчерпание лимита на дескрипторы. Чаще всего это исчерпание виртуальной памяти, если в системе уже существует большое количество потоков.

Для получения дескриптора и идентификатора текущего процесса используются функции `GetCurrentProcess()` и `GetCurrentProcessId()`:

```
include <windows.h>
HANDLE WINAPI GetCurrentProcess(void);
DWORD WINAPI GetCurrentProcessId(void);
```

Аналогичные функции существуют и для получения дескриптора и идентификатора текущего потока:

```
include <windows.h>
HANDLE WINAPI GetCurrentThread(void);
DWORD WINAPI GetCurrentThreadId(void);
```

Рассмотрим пример программы, порождающей новые потоки и процессы:

ProcessAndThread.c

```
#include <windows.h>
#include <stdio.h>
#include <tchar.h>

// Данные, передаваемые создаваемому потоку
typedef struct _Data {
  TCHAR cmd[20];
  int parentThreadID;
} TDATA, *PTDATA;

DWORD WINAPI ThreadProc(LPVOID lpParam);
void ErrorReport(LPTSTR lpszFunction);
int main()
{
  PTDATA pData;
  DWORD dwThreadId;
  HANDLE hThread;

  // Выделяем память для данных, передаваемых
  // создаваемому потоку
  pData = (PTDATA)HeapAlloc(GetProcessHeap(),
  HEAP_ZERO_MEMORY,sizeof(TDATA));
  if(pData == NULL)
  {
    ErrorReport(TEXT("HeapAlloc()"));
    return(1);
  }
  // Заполняем структуру данных результатом выпол-
  // нения команды распечатки переменных окружения
  // и идентификатором текущего потока
  tcscpy(pData->cmd, TEXT("cmd /c set ComSpec"));
  pData->parentThreadID = GetCurrentThreadId();
  // Создаем новый поток
  hThread = CreateThread(
    NULL, // атрибуты безопасности по умолчанию
    0, // размер стека потока по умолчанию
```

```
    ThreadProc, // адрес основной функции потока
    pData, // данные для потока
    0, // флаги создания потока по умолчанию
    &dwThreadId); // содержит идентификатор
    // созданного потока
  // Проверям, что поток был успешно создан
  if (hThread == NULL)
  {
    ErrorReport(TEXT("CreateThread()"));
    return(1);
  }

  // Ждем ожидания завершения потока без таймаута
  WaitForSingleObject(hThread, INFINITE);
  // Закрываем дескриптор потока
  CloseHandle(hThread);

  return(0);
}

// Главная функция создаваемого потока
DWORD WINAPI ThreadProc(LPVOID lpParam)
{
  STARTUPINFO si;
  PROCESS_INFORMATION pi;
  PTDATA pData;

  // Получаем данные, передаваемые создаваемому
  // потоку
  pData = (PTDATA)lpParam;
  // Выводим на консоль данные о потоке-родителе и
  // процессе, который должен быть порожден
  // созданным потоком
  _tprintf(TEXT("New thread is created by thread
  with Id %d and command to execute is \"%s\"\
  n"), pData->parentThreadID, pData->cmd);

  ZeroMemory(&si, sizeof(si));
  si.cb = sizeof(si);
  ZeroMemory(&pi, sizeof(pi));
  // Запускаем процесс-потомок
  if(!CreateProcess(NULL, // Не используем первый
  // параметр
    pData->cmd, // Командная строка
    NULL, // Дескриптор создаваемого процесса не
    // наследуемый
```

```
    NULL, // Дескриптор первичного потока
    // не наследуемый
    FALSE, // Процесс не наследует дескрипторы
    // процесса-родителя
    0,  // Флаги создания процесса по умолчанию
    NULL, // Окружение наследуется от
    // процесса-родителя
    NULL, // Текущий каталог наследуется от
    // процесса-родителя
    &si,  // Указатель на начальные установки для
    // процесса
    &pi) // Получаем дескрипторы процесса и его
    // первичного потока
  )
  {
    ErrorReport(TEXT("CreateProcess()"));
    return(1);
  }

  // Ожидаем завершения выполнения процесса-потомка
  // без таймаута
  WaitForSingleObject(pi.hProcess, INFINITE);

  // Закрываем дескрипторы процесса и его первичного
  // потока
  CloseHandle(pi.hProcess);
  CloseHandle(pi.hThread);

  // Освобождаем память, выделенную для параметров
  // потока
  HeapFree(GetProcessHeap(), 0, pData);

  return(0);
}

void ErrorReport(LPTSTR lpszFunction)
{
  LPVOID lpMsgBuf;
  LPVOID lpDisplayBuf;
  DWORD dw = GetLastError();

  FormatMessage(
    FORMAT_MESSAGE_ALLOCATE_BUFFER |
    FORMAT_MESSAGE_FROM_SYSTEM,
    NULL,
    dw,
    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
    (LPTSTR) &lpMsgBuf,
```

```
    0, NULL );
  lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,
  (lstrlen((LPCTSTR)lpMsgBuf)+lstrlen((LPCTSTR)lpsz
  Function)+40)*sizeof(TCHAR));
    _stprintf((LPTSTR)lpDisplayBuf,
      TEXT("%s failed with error %d: %s"),
      lpszFunction, dw, lpMsgBuf);
    MessageBox(NULL, (LPCTSTR)lpDisplayBuf,
    TEXT("Error"), MB_OK);

    LocalFree(lpMsgBuf);
    LocalFree(lpDisplayBuf);
}
```

В процессе работы приведенной программы создается еще один поток, который принимает два параметра: идентификатор родительского потока и строку, в которой содержится команда. Созданный поток, в свою очередь, порождает новый процесс, используя переданную ему команду (в примере команда просмотра значения переменной окружения `ComSpec – "cmd /c set ComSpec"`).

В результате работы программы на консоль выводится сообщение:

```
C:\>ProcessAndThread.exe
New thread is created by thread with Id 3136 and
command to execute is "cmd /c set ComSpec"
ComSpec=C:\WIN\system32\cmd.exe
C:\>
```

Следует упомянуть еще ряд функций, использованных в этом примере. В первую очередь это функция `WaitForSingleObject()`:

```
include <windows.h>
DWORD WINAPI WaitForSingleObject(
  HANDLE hHandle,
  DWORD dwMilliseconds);
```

Данная функция блокирует текущий процесс до тех пор, пока объект, ассоциированный с переданным дескриптором `hHandle`, не перейдет в сигнальное состояние или не истечет заданный период ожидания `dwMilliseconds` в миллисекундах. Если параметр `dwMilliseconds` установлен в значение `INFINITE`, то период ожидания никогда не истекает и процесс пробуждается только при переходе заданного объекта в сигнальное состояние. Функции семейства `WaitForObjects()` являются одним из элементов механизмов синхронизации, поддерживаемых операционными системами семейства Windows.

В качестве объектов, за чьим состоянием может следить данная функция, могут выступать события, мьютексы, процессы, потоки, семафоры и т. д. Если наблюдаемые объекты являются процессами или потоками, то для них сигнальное состояние наступает в том случае, если они завершаются.

Если функция завершается при переходе наблюдаемого объекта в сигнальное состояние, то возвращается значение WAIT_OBJECT_0., если же функция завершается по истечении периода ожидания, то возвращается значение WAIT_TIMEOUT.

Если необходимо отслеживать одновременно несколько объектов, то используется функция WaitForMultipleObjects():

```
DWORD WINAPI WaitForMultipleObjects(
  DWORD nCount,
  const HANDLE* lpHandles,
  BOOL bWaitAll,
  DWORD dwMilliseconds);
```

В этой функции параметр nCount определяет общее количество дескрипторов объектов, за которыми ведется наблюдение. В параметре lpHandles передается массив дескрипторов объектов, состояние которых отслеживается функцией WaitForMultipleObjects(). Если параметр bWaitAll установлен в TRUE, то функция возвращает управление ожидающему процессу только в том случае, если все объекты из массива были переведены в сигнальное состояние. В противном случае функция ожидает перевода хотя бы одного объекта в сигнальное состояние, а не всех одновременно. Параметр dwMilliseconds, как и для функции WaitForSingleObject(), используется для задания периода ожидания перевода объектов в сигнальное состояние, по истечении которого управление возвращается вызвавшему функцию процессу. Это происходит даже в том случае, если период ожидания истек и ни один объект не был переведен в сигнальное состояние. Данный параметр также может принимать значение INFINITE.

По завершении работы функция WaitForMultipleObjects() возвращает значение WAIT_TIMEOUT, если истекло время ожидания перевода объектов в сигнальное состояние. Если при вызове функции параметр bWaitAll установлен в TRUE и все объекты были переведены в сигнальное состояние до истечения периода ожидания, то функция возвращает значение WAIT_OBJECT_0. Если же параметр bWaitAll установлен в FALSE и какой-либо объект был переведен в сигнальное состояние, то возвращается значение WAIT_OBJECT_0 + n, где n — номер объекта в массиве lpHandles, который был переведен в сигнальное состояние.

По окончании работы с объектами, для которых открываются дескрипторы, дескрипторы должны быть закрыты. Для этого используется функция `CloseHandle()`:

```
include <windows.h>
DWORD WINAPI BOOL CloseHandle(HANDLE hObject);
```

В качестве параметра передается ранее открытый дескриптор. Дескрипторы процессов и потоков должны быть закрыты даже в том случае, если сами процессы и потоки уже завершились.

9.4.2. Синхронизация: события, семафоры, мьютексы

Как и большинство ОС, поддерживающих одновременную работу нескольких конкурирующих процессов, ОС семейства Windows поддерживают ряд механизмов синхронизации процессов и потоков. Это позволяет процессам избегать одновременного использования неразделяемых ресурсов, что чаще всего приводит к краху процессов, а в некоторых случаях и всей системы в целом.

В Windows ОС поддерживают такие механизмы синхронизации, как события, семафоры, мьютексы, критические области, а также ряд методов организации разделения ресурсов между конкурирующими процессами. Рассмотрим работу с такими механизмами синхронизации, как события, семафоры и мьютексы.

События (*events*) представляют собой объекты механизма синхронизации, предназначенные для извещения потоков о наступлении некоторого программно управляемого события. Существует два типа объектов синхронизации — события, сбрасываемые вручную, и события, сбрасываемые автоматически.

Автоматически сбрасываемые события характеризуются тем, что если такое событие переводится в сигнальное состояние, то система автоматически выбирает ожидающий сигнального состояния поток и передает управление ему. При этом такое событие автоматически переводится в несигнальное состояние. При работе с сбрасываемыми вручную событиями ответственность за восстановление несигнального состояния события берет на себя программист. В этом случае он явно должен вызывать функцию `ResetEvent()` каждый раз, когда необходимо перевести событие в несигнальное состояние.

Для создания события используется функция `CreateEvent()`:

```
include <windows.h>
HANDLE WINAPI CreateEvent(
  LPSECURITY_ATTRIBUTES lpEventAttributes,
```

```
  BOOL bManualReset,
  BOOL bInitialState,
  LPCTSTR lpName);
```

Первый параметр `lpEventAttributes` отвечает за то, наследуется или нет дескриптор создаваемого события процессами-потомками, а также за изменение прав доступа создаваемого события. Если этот параметр установлен в значение `NULL`, то дескриптор не наследуется потомками и устанавливаются права доступа по умолчанию.

Параметр `bManualReset` отвечает за то, событие какого типа должно быть создано. Если этот параметр принимает значение `TRUE`, то создается событие, сбрасываемое вручную. Если же передается значение `FALSE`, то создается автоматически сбрасываемое событие.

Параметр `bInitialState` предназначен для задания начального состояния создаваемого события. Если данный параметр равен `TRUE`, то создается событие с сигнальным начальным состоянием. В противном случае начальное состояние события устанавливается в несигнальное.

Параметр `lpName` используется для задания имени создаваемого события. Если этот параметр равен `NULL`, то создается анонимный объект событие. Если же в данном параметре передается имя, то система сначала пытается определить, существует уже событие с таким именем или нет. Если такое событие не существует, то создается новое событие с требуемыми начальными установками. В противном случае новое событие не создается, а открывается дескриптор, ассоциированный с ранее созданным событием с заданным именем.

Количество создаваемых событий ограничивается только системными ресурсами и ограничением количества дескрипторов для процесса, которое может быть изменено администратором системы.

Для явного указания системе, что необходимо получить дескриптор для ранее созданного события, используется функция `OpenEvent()`:

```
include <windows.h>
HANDLE WINAPI OpenEvent(
  DWORD dwDesiredAccess,
  BOOL bInheritHandle,
  LPCTSTR lpName);
```

Параметр `dwDesiredAccess` сообщает системе, какие права доступа требует пользователь для управления событием. Этот параметр может быть установлен в значение `EVENT_ALL_ACCESS`,

позволяющее получить полный доступ к событию, либо в значение EVENT_MODIFY_STATE, которое позволяет пользователю изменять состояние события и в большинстве случаев его достаточно для работы с событием.

Параметр bInheritHandle определяет, может ли наследоваться дочерними процессами создаваемый дескриптор события. Если этот параметр установлен в значение TRUE, то создаваемый дескриптор наследуется процессами-потомками, в противном случае создается ненаследуемый дескриптор.

Параметр lpName задает имя того существующего события, доступ к которому хочет получить пользователь. Таким образом, с помощью данной функции можно получать доступ только к именованным событиям. Анонимные события могут использоваться только процессом, создающим событие, и его потоками.

Перевод события любого типа в сигнальное состояние осуществляется с помощью функции SetEvent():

```
include <windows.h>
DWORD WINAPI SetEvent(HANDLE hEvent);
```

В качестве параметра в эту функцию предается дескриптор того события, которое должно быть переведено в сигнальное состояние.

Для сбрасывания события используется функция ResetEvent():

```
include <windows.h>
DWORD WINAPI ResetEvent(HANDLE hEvent);
```

Параметр hEvent в свою очередь задает дескриптор того события, чье состояние должно быть восстановлено из сигнального состояния в несигнальное.

Как и в случае ожидания завершения работы процессов или потоков, процесс получает уведомление о том, что то или иное событие было переведено в сигнальное состояние с помощью функций WaitForSingleObject() и WaitForMultipleObjects().

В качестве примера использования событий для синхронизации процессов и потоков рассмотрим ранее приведенный пример синхронизации двух процессов, имеющих разное время выполнения.

Events.c

```
#include <windows.h>
#include <stdio.h>
#include <tchar.h>

DWORD WINAPI ThreadProc(LPVOID lpParam);
void ErrorReport(LPTSTR lpszFunction);
```

```
int main()
{
  DWORD dwThreadId;
  HANDLE hThread, hEvent1, hEvent2;
  unsigned i;
  // Создаем автоматически сбрасываемое событие
  // с несигнальным начальным состоянием
  if((hEvent1 = CreateEvent(NULL, FALSE, FALSE,
  "Thread1")) == NULL)
  {
    ErrorReport(TEXT("CreateEvent()"));
    return(1);
  }
  // Создаем поток, который попытаемся синхронизировать
  // с первичным потоком
  hThread = CreateThread(
    NULL,    // права по умолчанию
    0,       // размер стека по умолчанию
    ThreadProc, // функция потока
    NULL,    // аргумент для функции отсутствует
    0,       // флаги по умолчанию
    &dwThreadId);
  if (hThread == NULL)
  {
    ErrorReport(TEXT("CreateThread()"));
    return(1);
  }
  // Ожидаем создания еще одного события вторым
  // потоком
  // После создания второго события он переведет
  // первое событие
  // в сигнальное состояние
  // Функция ResetEvents() не вызывается,
  // так как события сбрасываются автоматически
  WaitForSingleObject(hEvent1, INFINITE);
  // Открываем созданное порожденным потоком событие
  if((hEvent2 = OpenEvent(EVENT_ALL_ACCESS, FALSE,
  "Thread2")) == NULL)
  {
    ErrorReport(TEXT("OpenEvent()"));
    return(1);
  }
  // Передаем управление потоку-потомку, переводя
  // второе событие
```

```
  // в сигнальное состояние
  SetEvent(hEvent2);
  // Основной цикл первичного потока
  for(i = 0; i < 10; ++i)
  {
    // Ожидаем передачи управления от порожденного
    // потока
    WaitForSingleObject(hEvent1, INFINITE);
    // Выводим данные на консоль
    printf("p%d ", i);
    // Блокируем первичный поток на 1 секунду
    Sleep(1000);
    // Передаем управление порожденному потоку
    SetEvent(hEvent2);
  }
  // Ожидаем завершения порожденного потока
  WaitForSingleObject(hThread, INFINITE);
  // Закрываем его дескриптор
  CloseHandle(hThread);
  // Закрываем дескрипторы событий
  CloseHandle(hEvent1);
  CloseHandle(hEvent2);
  printf("\n");
  return(0);
}
DWORD WINAPI ThreadProc(LPVOID lpParam)
{
  HANDLE hEvent1, hEvent2;
  unsigned i;
  // Открываем событие, созданное первичным потоком
  if((hEvent1 = OpenEvent(EVENT_ALL_ACCESS, FALSE,
  "Thread1")) == NULL)
  {
    ErrorReport(TEXT("OpenEvent()"));
    return(1);
  }
  // Создаем свое автоматически сбрасываемое событие
  // с несигнальным начальным состоянием
  if((hEvent2 = CreateEvent(NULL, FALSE, FALSE,
  "Thread2")) == NULL)
  {
    ErrorReport(TEXT("CreateEvent()"));
    return(1);
  }
```

```
  // Передаем управление первичному потоку
  // для того, чтобы он смог открыть событие,
  // созданное в текущем потоке
  SetEvent(hEvent1);
  // Основной цикл дочернего потока
  for(i = 0; i < 10; ++i)
  {
    // Ожидаем передачи управления от первичного
    // потока
    WaitForSingleObject(hEvent2, INFINITE);
    // Выводим данные на консоль
    printf("c%d ", i);
    // Блокируем поток на 4 секунды
    Sleep(4000);
    // Передаем управление первичному потоку
    SetEvent(hEvent1);
  }
  // Закрываем дескрипторы событий
  CloseHandle(hEvent1);
  CloseHandle(hEvent2);
  return(0);
}
void ErrorReport(LPTSTR lpszFunction)
{
  LPVOID lpMsgBuf;
  LPVOID lpDisplayBuf;
  DWORD dw = GetLastError();
  FormatMessage(
    FORMAT_MESSAGE_ALLOCATE_BUFFER |
    FORMAT_MESSAGE_FROM_SYSTEM,
    NULL,
    dw,
    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
    (LPTSTR) &lpMsgBuf,
    0, NULL );
  lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,
(lstrlen((LPCTSTR)lpMsgBuf)+lstrlen((LPCTSTR)lpsz
Function)+40)*sizeof(TCHAR));
  _stprintf((LPTSTR)lpDisplayBuf,
    TEXT("%s failed with error %d: %s"),
    lpszFunction, dw, lpMsgBuf);
MessageBox(NULL, (LPCTSTR)lpDisplayBuf,
TEXT("Error"), MB_OK);
```

```
  LocalFree(lpMsgBuf);
  LocalFree(lpDisplayBuf);
}
```

В результате работы данной программы на консоль будет выведена строка следующего вида:

```
c0 p0 c1 p1 c2 p2 c3 p3 c4 p4 c5 p5 c6 p6 c7 p7
c8 p8 c9 p9
```

Очевидно, что несмотря на разное время блокировки потоков, их вывод строго чередуется. Если же в приведенной программе закомментировать вызовы функций `SetEvent()` и `WaitForSingleObject()` внутри циклов и первичного потока, и потока-потомка, то результат работы программы будет иметь следующий вид:

```
p0 c0 p1 p2 p3 p4 c1 p5 p6 p7 p8 c2 p9 c3 c4 c5
c6 c7 c8 c9
```

Отсюда следует, что первичный поток никак не синхронизирован с потоком-потомком и завершает свою основную работу гораздо раньше, чем дочерний поток.

Второй механизм, используемый для синхронизации в Windows, рассматривался при описании средств синхронизации в Linux — семафоры. В Windows семафоры представляют собой объекты синхронизации, имеющие внутренний счетчик, значение которого может находиться в диапазоне от 0 до заданного максимального значения. Этот счетчик уменьшается на 1 каждый раз, когда одному из потоков возвращается управление из функции ожидания (`WaitForSingleObject()` и `WaitForMultipleObjects()`), и увеличивается на заданное значение каждый раз, когда поток освобождает семафор. Семафор переходит в сигнальное состояние каждый раз, когда значение счетчика не равно 0, и в несигнальное состояние, когда счетчик равен 0.

Для создания семафора используется функция `CreateSemaphore()`:

```
#include <windows.h>
HANDLE WINAPI CreateSemaphore(
  LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
  LONG lInitialCount,
  LONG lMaximumCount,
  LPCTSTR lpName);
```

Параметр `lpSemaphoreAttributes` отвечает за то, наследуется или нет дескриптор создаваемого семафора процессами-потомками, а также за изменение его прав доступа. Если этот пара-

метр установлен в значение NULL, то дескриптор не наследуется потомками и устанавливаются права доступа по умолчанию.

В параметре lInitialCount передается начальное значение внутреннего счетчика семафора. Максимально допустимое значение счетчика передается в параметре lMaximumCount.

Параметр lpName используется для задания имени создаваемого семафора. Если этот параметр равен NULL, то создается анонимный семафор. Если же передается имя, которое уже было использовано при создании системных объектов Windows, таких как события, семафоры, мьютексы и другие, то семафор не создается и функция завершается с ошибкой.

Если функция завершается успешно и удается создать семафор, то функция CreateSemaphore() возвращает дескриптор созданного семафора. В противном случае функция возвращает значение NULL.

Количество созданных семафоров, так же, как и количество событий, ограничивается системными ресурсами и лимитом на количество дескрипторов, которое может создать одиночный процесс.

Для явного указания системе, что необходимо получить дескриптор ранее созданного семафора, используется функция OpenSemaphore():

```
include <windows.h>
HANDLE WINAPI OpenSemaphore(
  DWORD dwDesiredAccess,
  BOOL bInheritHandle,
  LPCTSTR lpName);
```

Параметр dwDesiredAccess сообщает системе, какие права доступа требует пользователь для управления семафором. Он может быть установлен в значение SEMAPHORE_ALL_ACCESS, позволяющий получить полный доступ к объекту, либо SEMAPHORE_MODIFY_STATE.

Параметр bInheritHandle определяет, может ли наследоваться дочерними процессами создаваемый дескриптор семафора. Если этот параметр установлен в значение TRUE, то создаваемый дескриптор наследуется процессами-потомками, в противном случае создается ненаследуемый дескриптор.

Параметр lpName задает имя ранее созданного семафора, доступ к которому необходимо получить. С помощью этого параметра можно получать доступ только к именованным семафорам, а неименованные семафоры могут использоваться только процессом, создающим данный объект, и его потомками.

Для освобождения семафора используется функция ReleaseSemaphore():

```
include <windows.h>
BOOL WINAPI ReleaseSemaphore(
  HANDLE hSemaphore,
  LONG lReleaseCount,
  LPLONG lpPreviousCount);
```

Параметр `hSemaphore` задает дескриптор семафора, который требуется освободить. Параметр `lReleaseCount` задает значение, на которое должен быть увеличен внутренний счетчик семафора. Это значение должно быть больше 0. Параметр `lpPrevious-Count` используется для возвращения предыдущего значения счетчика семафора.

Еще одним средством синхронизации процессов и потоков в Windows являются мьютексы. Мьютексы представляют собой объекты синхронизации, которые переводятся в сигнальное состояние в том случае, если он не принадлежит ни одному потоку. Если же мьютекс принадлежит какому-либо потоку, то он переводится в несигнальное состояние. С этой точки зрения мьютексы удобны при организации взаимно исключающего доступа нескольких потоков к разделяемому ими ресурсу. Примером такого ресурса может являться такой объект межпроцессного взаимодействия, как разделяемая память. Записывать данные в этот объект в каждый момент времени должен только один вычислительный поток. Поэтому задача организации взаимно исключающего доступа к такому механизму межпроцессного взаимодействия является очень важной. В целом же мьютексы можно рассматривать как один из вариантов семафоров.

Для создания мьютексов используется функция `Create-Mutex()`:

```
#include <windows.h>
HANDLE WINAPI CreateMutex(
  LPSECURITY_ATTRIBUTES lpMutexAttributes,
  BOOL bInitialOwner,
  LPCTSTR lpName);
```

Параметр `lpMutexAttributes` отвечает за то, наследуется или нет дескриптор создаваемого мьютекса потомками, а также за изменение его прав доступа. Если этот параметр установлен в значение `NULL`, то дескриптор не наследуется, и устанавливаются права по умолчанию.

Если параметр `bInitialOwner` установлен в `TRUE`, то поток, создавший мьютекс, объявляется его владельцем. Если же данный параметр установлен в `FALSE`, то создающий мьютекс вычислительный поток не получает его во владение.

Параметр `lpName` используется для задания имени мьютекса. Если он равен `NULL`, то создается анонимный объект. Если же

передается имя, которое уже было использовано при создании системных объектов Windows, таких как события, семафоры, мьютексы и другие, то семафор не создается и функция завершается с ошибкой.

Если функция завершается успешно и удается создать семафор, то функция `CreateMutex()` возвращает дескриптор созданного объекта. В противном случае функция возвращает значение `NULL`.

Максимальное количество мьютексов, создаваемых в системе, определяется наличием свободных ресурсов в системе (в особенности, свободной виртуальной памяти) и ограничением на количество дескрипторов в процессе.

Для явного указания системе, что необходимо получить дескриптор ранее созданного мьютекса, используется функция `OpenMutex()`:

```
include <windows.h>
HANDLE WINAPI OpenMutex(
  DWORD dwDesiredAccess,
  BOOL bInheritHandle,
  LPCTSTR lpName);
```

Параметр `dwDesiredAccess` сообщает системе, какие права доступа требует пользователь для управления мьютексом. Этот параметр может быть установлен в значение `MUTEX_ALL_ACCESS`, позволяющий получить полный доступ к объекту, либо `MUTEX_MODIFY_STATE`.

Параметр `bInheritHandle` определяет, может ли наследоваться дочерними процессами создаваемый дескриптор мьютекса. Если данный параметр установлен в значение `TRUE`, то создаваемый дескриптор наследуется дочерними процессами, в противном случае создается ненаследуемый дескриптор.

Параметр `lpName` задает имя мьютекса, доступ к которому необходимо получить. С его помощью можно получать доступ только к именованным семафорам, а неименованные семафоры могут использоваться только процессом, создающим данный объект, и его потоками.

Для освобождения мьютекса используется функция `ReleaseMutex()`:

```
include <windows.h>
BOOL WINAPI ReleaseMutex(HANDLE hMutex);
```

Параметр `hSemaphore` задает дескриптор мьютекса, который требуется освободить.

Рассмотрим пример использования мьютексов и семафоров для синхронизации вычислительных потоков. Для этого модифи-

цируем ранее рассмотренный пример синхронизации двух процессов, имеющих разное время выполнения, и используем оба механизма синхронизации — и семафоры, и мьютексы.

SemaphoreAndMutex.c

```
#include <windows.h>
#include <stdio.h>
#include <tchar.h>

DWORD WINAPI ThreadProc(LPVOID lpParam);
void ErrorReport(LPTSTR lpszFunction);

int main()
{
  DWORD dwThreadId;
  HANDLE hThread, hSem, hMutex;
  unsigned i;

  // Создаем семафор с несигнальным начальным
  // состоянием
  // Данный семафор используется только для
  // синхронизации процесса создания мьютекса
  if((hSem = CreateSemaphore(NULL, 0, 1, "Thread1"))
  == NULL)
  {
    ErrorReport(TEXT("CreateSemaphore()"));
    return(1);
  }
  hThread = CreateThread(
    NULL,       // права по умолчанию
    0,          // размер стека по умолчанию
    ThreadProc, // функция потока
    NULL,       // аргумент для функции отсутствует
    0,          // флаги по умолчанию
    &dwThreadId);
  if(hThread == NULL)
  {
    ErrorReport(TEXT("CreateThread()"));
    return(1);
  }
  // Ожидаем создания мьютекса дочерним
  // вычислительным потоком
  WaitForSingleObject(hSem, INFINITE);
  // Получаем дескриптор на созданный мьютекс
  if((hMutex = OpenMutex(MUTEX_ALL_ACCESS, FALSE,
  "Thread2")) == NULL)
```

```
  {
    ErrorReport(TEXT("OpenMutex()"));
    return(1);
  }
  // Основной цикл первичного потока
  for(i = 0; i < 10; ++i)
  {
    // Ожидаем передачи управления от порожденного
    // потока
    WaitForSingleObject(hMutex, INFINITE);
    // Выводим данные на консоль
    printf("p%d ", i);
    // Блокируем первичный поток на 1 секунду
    Sleep(1000);
    // Передаем управление порожденному потоку
    ReleaseMutex(hMutex);
  }
  // Ожидаем завершения порожденного потока
  WaitForSingleObject(hThread, INFINITE);
  // Закрываем его дескриптор
  CloseHandle(hThread);
  // Закрываем дескрипторы семафора и мьютекса
  CloseHandle(hSem);
  CloseHandle(hMutex);
  printf("\n");
  return(0);
}

DWORD WINAPI ThreadProc(LPVOID lpParam)
{
  HANDLE hSem, hMutex;
  unsigned i;
  // Получаем дескриптор на созданный ранее семафор
  if((hSem = OpenSemaphore(SEMAPHORE_ALL_ACCESS,
  FALSE, "Thread1")) == NULL)
  {
    ErrorReport(TEXT("OpenSemaphore()"));
    return(1);
  }
  // Создаем мьютекс с начальным несигнальным
  // состоянием
  if((hMutex = CreateMutex(NULL, FALSE, "Thread2"))
  == NULL)
  {
    ErrorReport(TEXT("CreateMutex()"));
```

```
    return(1);
  }
  // Переводим семафор в сигнальное состояние,
  // извещая первичный поток
  // о создании мьютекса для синхронизации
  // процесса вывода данных
  ReleaseSemaphore(hSem, 1, NULL);
  // Основной цикл дочернего потока
  for(i = 0; i < 10; ++i)
  {
    // Ожидаем передачи управления от первичного потока
    WaitForSingleObject(hMutex, INFINITE);
    // Выводим данные на консоль
    printf("c%d ", i);
    // Блокируем поток на 4 секунды
    Sleep(4000);
    // Передаем управление первичному потоку
    ReleaseMutex(hMutex);
  }
  // Закрываем дескрипторы семафора и мьютекса
  CloseHandle(hSem);
  CloseHandle(hMutex);
  return(0);
}
void ErrorReport(LPTSTR lpszFunction)
{
  LPVOID lpMsgBuf;
  LPVOID lpDisplayBuf;
  DWORD dw = GetLastError();

  FormatMessage(
    FORMAT_MESSAGE_ALLOCATE_BUFFER |
    FORMAT_MESSAGE_FROM_SYSTEM,
    NULL,
    dw,
    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
    (LPTSTR) &lpMsgBuf,
    0, NULL );

  lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,
  (lstrlen((LPCTSTR)lpMsgBuf)+lstrlen((LPCTSTR)lpsz
  Function)+40)*sizeof(TCHAR));
  _stprintf((LPTSTR)lpDisplayBuf,
    TEXT("%s failed with error %d: %s"),
    lpszFunction, dw, lpMsgBuf);
```

```
  MessageBox(NULL, (LPCTSTR)lpDisplayBuf,
  TEXT("Error"), MB_OK);
  LocalFree(lpMsgBuf);
  LocalFree(lpDisplayBuf);
}
```

В данном примере семафор используется только для первичной синхронизации процессов. С помощью семафора дочерний поток извещает первичный поток о создании мьютекса, который будет использоваться для организации взаимно исключающего доступа первичного и дочернего вычислительных потоков к консоли. В дальнейшем семафор не используется, а используется именно мьютекс.

В результате работы данной программы на консоль будет выведена следующая строка:

```
c0 p0 c1 p1 c2 p2 c3 p3 c4 p4 c5 p5 c6 p6 c7 p7
c8 p8 c9 p9
```

Как и ранее, оба потока синхронизованы, и их вывод строго чередуется. Если же в приведенной выше программе закомментировать вызовы функций `ResetMutex()` и `WaitForSingleObject()` внутри циклов и первичного потока, и потока-потомка, то результат работы программы будет иметь следующий вид:

```
c0 p0 p1 p2 p3 c1 p4 p5 p6 p7 p8 c2 p9 c3 c4 c5
c6 c7 c8 c9
```

Здесь опять наблюдается ситуация, что при отсутствии синхронизации потоков, первичный поток завершает свою работу раньше дочернего потока и вывод данных на консоль двух вычислительных потоков никак не синхронизируется.

9.4.3. Каналы

Каналы являются одним из основных механизмов межпроцессного взаимодействия в Windows. Так же, как ОС семейства Linux, Windows поддерживает два вида каналов — анонимные и именованные. Каналы используются как обычные файлы, для чтения и записи которых существуют стандартные функции `ReadFile()` и `WriteFile()`.

Анонимные каналы (*anonymous pipe*) представляют собой однонаправленный механизм передачи данных между процессами. Чаще всего они используются для передачи данных между процессами-родителями и их потомками. Каждый канал имеет два связанных с ним дескриптора: чтения и записи.

Для создания анонимного канала используется функция `CreatePipe()`:

```
include <windows.h>
BOOL CreatePipe(
  PHANDLE hReadPipe,
  PHANDLE hWritePipe,
  LPSECURITY_ATTRIBUTES lpPipeAttributes,
  DWORD nSize);
```

В параметрах `hReadPipe` и `hWritePipe` возвращаются соответственно дескриптор чтения и дескриптор записи. Эти дескрипторы автоматически создаются при создании неименованного канала. Параметр `lpPipeAttributes` отвечает за то, наследуются ли создаваемые дескрипторы чтения и записи канала процессами-потомками, а также за изменение его прав доступа. Если этот параметр установлен в значение `NULL`, то дескрипторы не наследуются потомками и устанавливаются права доступа по умолчанию. Параметр `nSize` используется для задания размера канала в байтах. Если передается значение 0, то размер создаваемого канала устанавливается по умолчанию.

При создании анонимного канала очень часто указывается, что создаваемые дескрипторы чтения и записи могут быть унаследованы. Это связано с тем, что при работе с анонимными каналами не существует других способов связывания процессов через канал, кроме как наследование процессами-потомками дескрипторов чтения/записи от создавшего их процесса-родителя.

Однако при наследовании дескрипторов, связанных с концами анонимных каналов, возникает проблема определения конца передачи данных по каналу. Дело в том, что система считает, что передача данных закончилась, если закрыты все дескрипторы записи, ассоциированные с каналом. Но эти дескрипторы создаются с признаком наследования и наследуются всеми процессами-потомками, в том числе и тем процессом, который считывает данные из канала. В результате возникает ситуация, когда записывающий процесс закончил передачу данных и закрыл свой дескриптор записи в канал. Процесс, считывающий данные из канала, не получает уведомления о конце операции записи, так как сам владеет еще одним дескриптором записи. В результате это приводит к зависанию процесса, считывающего данные из канала. Чтобы этого не происходило, необходимо каждый раз при создании нового процесса помечать те дескрипторы, которые не должны им наследоваться, как ненаследуемые. Это позволяет корректно обрабатывать ситуацию окончания записи данных в канал.

Для изменения свойств дескриптора используется функция `SetHandleInformation()`:

```
include <windows.h>
BOOL SetHandleInformation(
```

```
  HANDLE hObject,
  DWORD dwMask,
  DWORD dwFlags);
```

Параметр `hObject` задает дескриптор, свойства которого должны быть изменены. Параметр `dwMask` определяет свойство дескриптора, которое должно быть изменено. Этот параметр может быть установлен в значение `HANDLE_FLAG_INHERIT` для изменения свойства наследования или `HANDLE_FLAG_PROTECT_FROM_CLOSE` для защиты дескриптора от явного закрытия с помощью функции `CloseHandle()`. Параметр `dwFlags` определяет, должно ли быть указанное свойство очищено (значение параметра 0) или же оно должно быть установлено (значение должно совпадать с значением параметра `dwMask`).

Рассмотрим пример использования неименованных каналов для организации перенаправления стандартного вывода одной команды в стандартный ввод другой команды. Этот пример демонстрирует реализацию операции | в командной оболочке Windows.

Pipe.c

```
#include <stdio.h>
#include <windows.h>
#include <tchar.h>

BOOL CreateChildProcessPS(HANDLE outHandle,
PROCESS_INFORMATION* pi);
BOOL CreateChildProcessMORE(HANDLE inHandle,
PROCESS_INFORMATION* pi);
void ErrorReport(LPTSTR lpszFunction);

int main(int argc, char *argv[])
{
  HANDLE hChildStdoutRd, hChildStdoutWr;
  SECURITY_ATTRIBUTES saAttr;
  BOOL fSuccess;
  PROCESS_INFORMATION piPSProcInfo,
  piMOREProcInfo;

  // Заполняем структуру saAttr таким образом,
  // чтобы созданные дескрипторы
  // чтения/записи канала были наследуемыми
  saAttr.nLength = sizeof(SECURITY_ATTRIBUTES);
  saAttr.bInheritHandle = TRUE;
  saAttr.lpSecurityDescriptor = NULL;
  // Создаем канал, через который будут
```

```
// передаваться данные со стандартного
// вывода одной команды в стандартный ввод
// другой команды
if(!CreatePipe(&hChildStdoutRd, &hChildStdoutWr,
&saAttr, 0))
{
  ErrorReport(TEXT("CreatePipe()"));
  return(1);
}
// Изменяем атрибут наследуемости для дескриптора
// записи таким образом,
// чтобы он не наследовался процессом, читающим
// данные
SetHandleInformation(hChildStdoutWr, HANDLE_
FLAG_INHERIT, 0);
// Создаем процесс "more.com", организующий
// постраничный вывод данных
fSuccess = CreateChildProcessMORE(hChildStdoutRd,
&piMOREProcInfo);
if (!fSuccess)
  return(1);
// Восстанавливаем свойство наследуемости для
// дескриптора записи
SetHandleInformation(hChildStdoutWr, HANDLE_
FLAG_INHERIT, HANDLE_FLAG_INHERIT);
// Изменяем атрибут наследуемости для дескриптора
// чтения таким образом,
// чтобы он не наследовался процессом,
// записывающим данные
SetHandleInformation(hChildStdoutRd, HANDLE_
FLAG_INHERIT, 0);
// Создаем процесс "qprocess *", выводящий список
// всех процессов в системе
fSuccess = CreateChildProcessPS(hChildStdoutWr,
&piPSProcInfo);
if (!fSuccess)
  return(1);
// Закрываем дескрипторы чтения/записи канала,
// так как текущий процесс
// их больше не будет использовать
CloseHandle(hChildStdoutRd);
CloseHandle(hChildStdoutWr);
// Ожидаем завершения дочерних процессов
WaitForSingleObject(piPSProcInfo.hProcess,
            INFINITE);
```

```
  CloseHandle(piPSProcInfo.hProcess);
  CloseHandle(piPSProcInfo.hThread);
  WaitForSingleObject(piMOREProcInfo.hProcess,
  INFINITE);
  CloseHandle(piMOREProcInfo.hProcess);
  CloseHandle(piMOREProcInfo.hThread);
  return(0);
}

BOOL CreateChildProcessMORE(HANDLE inHandle,
PROCESS_INFORMATION* pi)
{
  // Создаем процесс “more.com”
  TCHAR szCmdline[]=TEXT(“more.com”);
  STARTUPINFO siStartInfo;
  BOOL bFuncRetn = FALSE;
  ZeroMemory(pi, sizeof(PROCESS_INFORMATION));
  // Указываем, что для создаваемого процесса
  // перенаправляется
  // стандартный поток ввода
  ZeroMemory(&siStartInfo, sizeof(STARTUPINFO));
  siStartInfo.cb = sizeof(STARTUPINFO);
  siStartInfo.hStdError = GetStdHandle(STD_ERROR_
  HANDLE);
  siStartInfo.hStdOutput = GetStdHandle(STD_
  OUTPUT_HANDLE);
  siStartInfo.hStdInput = inHandle;
  siStartInfo.dwFlags |= STARTF_USESTDHANDLES;

  // Создаем процесс
  bFuncRetn = CreateProcess(NULL,
    szCmdline, // командная строка
    NULL,      // атрибуты безопасности процесса
    // по умолчанию
    NULL, // атрибуты безопасности первичного
    // потока по умолчанию
    TRUE,      // дескрипторы наследуются
    // от родительского процесса
    0,          // флаги по умолчанию
    NULL,      // окружение наследуется
    // от процесса-родителя
    NULL,      // текущий каталог наследуется
    // от процесса-родителя
    &siStartInfo,  // указываем, что поток ввода
    // перенаправляется
```

```
    pi); // получаем дескрипторы созданного
    // процесса и его первичного потока
  if (bFuncRetn == 0)
  {
    ErrorReport(TEXT("CreateProcess() for MORE"));
    return(FALSE);
  }
  return(bFuncRetn);
}

BOOL CreateChildProcessPS(HANDLE outHandle,
PROCESS_INFORMATION* pi)
{
  // Создаем процесс "qprocess *"
  TCHAR szCmdline[]=TEXT("qprocess *");
  STARTUPINFO siStartInfo;
  BOOL bFuncRetn = FALSE;

  ZeroMemory(pi, sizeof(PROCESS_INFORMATION));
  // Указываем, что для создаваемого процесса
  // перенаправляются
  // стандартные потоки вывода и вывода ошибок
  ZeroMemory(&siStartInfo, sizeof(STARTUPINFO));
  siStartInfo.cb = sizeof(STARTUPINFO);
  siStartInfo.hStdError = outHandle;
  siStartInfo.hStdOutput = outHandle;
  siStartInfo.hStdInput = GetStdHandle(STD_INPUT_
  HANDLE);
  siStartInfo.dwFlags |= STARTF_USESTDHANDLES;

  // Создаем процесс
  bFuncRetn = CreateProcess(NULL,
    szCmdline, // командная строка
    NULL, // атрибуты безопасности процесса
    // по умолчанию
    NULL, // атрибуты безопасности первичного
    // потока по умолчанию
    TRUE, // дескрипторы наследуются
    // от родительского процесса
    0,    // флаги по умолчанию
    NULL, // окружение наследуется
    // от процесса-родителя
    NULL, // текущий каталог наследуется
    // от процесса-родителя
    &siStartInfo, // указываем, что потоки вывода
    // перенаправляются
```

```
    pi);  // получаем дескрипторы созданного
    // процесса и его первичного потока
  if (bFuncRetn == 0)
  {
    ErrorReport(TEXT("CreateProcess() for PS"));
    return(FALSE);
  }
  CloseHandle(pi->hThread);
  return(bFuncRetn);
}
void ErrorReport(LPTSTR lpszFunction)
{
  LPVOID lpMsgBuf;
  LPVOID lpDisplayBuf;
  DWORD dw = GetLastError();

  FormatMessage(
    FORMAT_MESSAGE_ALLOCATE_BUFFER |
    FORMAT_MESSAGE_FROM_SYSTEM,
    NULL,
    dw,
    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
    (LPTSTR) &lpMsgBuf,
    0, NULL );

  lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,
  (lstrlen((LPCTSTR)lpMsgBuf)+lstrlen((LPCTSTR)lpsz
  Function)+40)*sizeof(TCHAR));
  _stprintf((LPTSTR)lpDisplayBuf,
    TEXT("%s failed with error %d: %s"),
    lpszFunction, dw, lpMsgBuf);
  MessageBox(NULL, (LPCTSTR)lpDisplayBuf,
  TEXT("Error"), MB_OK);
  LocalFree(lpMsgBuf);
  LocalFree(lpDisplayBuf);
}
```

В приведенном примере выполняется команда, аналогичная команде `ps aux | more` в ОС Unix. Запускается команда `qprocess * | more.com`, предназначенная для постраничного вывода списка всех процессов, которые исполняются в текущий момент системой.

В отличие от неименованных каналов, именованные каналы могут использоваться не только процессами-потомками и родителями, но и для организации межпроцессного взаимодействия между независимыми процессами. Кроме того, именованные

каналы позволяют организовывать не только однонаправленный канал связи, но и обеспечивать двустороннюю передачу данных. Также достоинством именованных каналов является такое их свойство, как возможность одновременного подключения нескольких клиентов к одному и тому же каналу.

Для создания именованных каналов сервер использует функцию `CreateNamedPipe()`:

```
#include <windows.h>
HANDLE CreateNamedPipe(
  LPCTSTR lpName,
  DWORD dwOpenMode,
  DWORD dwPipeMode,
  DWORD nMaxInstances,
  DWORD nOutBufferSize,
  DWORD nInBufferSize,
  DWORD nDefaultTimeOut,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes);
```

Параметр `lpName` задает имя создаваемого канала. Формат данной строки должен соответствовать следующему шаблону:

```
\\.\pipe\[path]pipename
```

Параметр `dwOpenMode` определяет режим работы создаваемого канала, т.е. является ли канал однонаправленным или двунаправленным. Допустимые значения данного параметра перечислены в табл. 9.4.

Таблица 9.4. **Допустимые значения параметра `dwOpenMode`**

Значение	Описание
`PIPE_ACCESS_DUPLEX`	Создает канал двунаправленной связи. Как сервер, так и клиент могут и писать в канал, и считывать данные из канала
`PIPE_ACCESS_INBOUND`	Создает однонаправленный канал. Сервер может только читать данные, а клиенты — только записывать
`PIPE_ACCESS_OUTBOUND`	Создает однонаправленный канал. Сервер может только записывать данные, а клиенты — только считывать

Параметр `dwPipeMode` определяет свойства создаваемого канала. Допустимые значения данного параметра описаны в табл. 9.5.

Таблица 9.5. **Допустимые значения параметра dwPipeMode**

Значение	Описание
PIPE_TYPE_BYTE	Данные записываются в канал как последовательность байт. Не может использоваться вместе с флагом PIPE_READMODE_MESSAGE
PIPE_TYPE_MESSAGE	Данные записываются в канал как последовательность сообщений. Может использоваться как с флагом PIPE_READMODE_MESSAGE, так и с флагом PIPE_READMODE_BYTE
PIPE_READMODE_BYTE	Данные считываются из канала как последовательность байт. Может использоваться как с флагом PIPE_TYPE_BYTE, так и с флагом PIPE_TYPE_MESSAGE
PIPE_READMODE_MESSAGE	Данные считываются из канала как последовательность сообщений. Может использоваться только с флагом PIPE_TYPE_MESSAGE

Параметр nMaxInstances определяет максимальное количество экземпляров каналов и максимальное количество клиентов, которое может присоединиться к создаваемому каналу.

Параметры nOutBufferSize и nInBufferSize задают размеры выходного и входного буферов именованного канала. Если эти параметры установлены в 0, устанавливаются размеры буферов по умолчанию.

Параметр nDefaultTimeOut задает значение периода ожидание в миллисекундах для функции WaitNamedPipe().

Параметр lpSecurityAttributes определяет, наследуется ли созданный дескриптор канал процессами-потомками, а также определяет атрибуты безопасности создаваемого канала. Если параметр установлен в NULL, то созданный дескриптор не наследуется.

После создания именованного канала сервер может ожидать запроса на соединения с каналом от клиента. Для этого используется функция ConnectNamedPipe():

```
#include <windows.h>
BOOL ConnectNamedPipe(
  HANDLE hNamedPipe,
  LPOVERLAPPED lpOverlapped);
```

Первый параметр hNamedPipe задает дескриптор ранее созданного канала. Если значение параметра lpOverlapped равно NULL,

то функция возвращает управление при подсоединении клиента к каналу. Если же созданный канал рассчитан на работу в асинхронном режиме, то данный параметр должен содержать указатель на структуру, задающую параметры асинхронного режима передачи данных по каналу.

Если клиент успешно подсоединился к каналу, то функция возвращает значение TRUE. Если же операция завершилась неудачей, то возвращается значение FALSE.

После установления соединения с клиентом сервер может читать запросы от клиента с помощью функции ReadFile() и возвращать ответы клиенту с помощью функции WriteFile().

Для завершения сеанса связи сервер должен закрыть соединение с клиентом. Для этого используется функция DisconnectNamedFile():

```
#include <windows.h>
BOOL DisconnectNamedPipe(HANDLE hNamedPipe);
```

При этом все данные, которые были в канале, теряются. Чтобы гарантировать их доставку клиенту, используется функция FlushFileBuffers():

```
#include <windows.h>
BOOL FlushFileBuffers(HANDLE hNamedPipe);
```

Она гарантирует, что по ее завершении все недоставленные ранее данные были доставлены клиенту, и соединение может быть разорвано.

Клиент, прежде чем пытаться открыть существующий канал, может проверить его доступность. Для этого используется функция WaitNamedPipe():

```
#include <windows.h>
BOOL WaitNamedPipe(
  LPCTSTR lpNamedPipeName,
  DWORD nTimeOut);
```

Параметр lpNamedPipeName задает имя канала, доступность которого проверяется клиентом. Параметр nTimeOut задает период ожидания этой функцией доступности заданного канала в миллисекундах. Кроме того, для данного параметра могут использоваться стандартные значения NMPWAIT_USE_DEFAULT_WAIT и NMPWAIT_WAIT_FOREVER. Первое значение говорит о том, что в качестве периода ожидания должно использоваться значение, ассоциированное с каналом при его создании в функции CreateNamedPipe(). Второе значение устанавливает бесконечный период ожидания, т.е. функция будет блокировать процесс до тех пор, пока требуемый канал не будет доступен для соединения.

Если канал с заданным именем не существует или истек период ожидания завершения операции, то функция завершается и возвращает FALSE. В противном случае функция возвращает TRUE.

Для подсоединения к каналу, созданному сервером, клиент должен открыть соответствующий канал и получить его дескриптор. Для этого используется функция CreateFile().

При использовании функции CreateFile() клиент задает имя канала, к которому он хочет получить доступ. При этом указывается соответствующий режим работы с каналом в зависимости от его свойств (т.е. доступен канал только для записи, только для чтения или он поддерживает работу в дуплексном режиме). В качестве имени файла используются строки, формат которых описан в табл. 9.6.

Для изменения параметров некоторого канала используется функция SetNamedPipeHandleState():

```
#include <windows.h>
BOOL SetNamedPipeHandleState(
  HANDLE hNamedPipe,
  LPDWORD lpMode,
  LPDWORD lpMaxCollectionCount,
  LPDWORD lpCollectDataTimeout);
```

Параметр hNamedPipe задает дескриптор того канала, параметры которого должны быть изменены. Новые параметры передаются в параметре lpMode. Этот параметр может включать флаги PIPE_READMODE_BYTE и PIPE_READMODE_MESSAGE, отвечающие за режим передачи данных. Кроме того, он может включать флаги, изменяющие механизм канала на блокирующий (флаг PIPE_WAIT) или неблокирующий (флаг PIPE_NOWAIT). Параметр lpMaxCollectionCount задает рамер блока памяти на стороне клиента, в котором буферизуются данные перед отправкой серверу. Параметр lpCollectDataTimeout определяет период ожидания на стороне клиента в миллисекундах, которое должно

Таблица 9.6. **Формат имен каналов в функциях `CreateFile()`**

Формат	Описание
\\.\pipe\name	Возвращает дескриптор ранее созданного канала с именем name на локальной машине
\\ computername \pipe\name	Возвращает дескриптор ранее созданного канала с именем name на машине с именем computername

пройти, прежде чем клиент отправит данные серверу. Последние два параметра могут быть установлены в NULL, если соответствующие параметры должны сохранить свои начальные значения.

В случае успешного завершения операции возвращается значение TRUE, иначе возвращается значение FALSE.

Для дальнейшего чтения или записи данных клиент, как и сервер, использует стандартные функции ReadFile() и WriteFile(). По окончании сеанса связи клиент закрывает дескриптор канала с помощью функции CloseHandle().

В качестве примера использования именованных каналов рассмотрим ранее приведенный пример пересылки латинского алфавита между сервером и клиентом.

Server.c

```
#include <stdio.h>
#include <windows.h>
#include <tchar.h>

#define BUFSIZE 512
// Имя канала
#define PIPE_NAME TEXT("\\\\.\\pipe\\pipe_example")
void ErrorReport(LPTSTR lpszFunction);

int main(int argc, char *argv[])
{
  TCHAR buf[BUFSIZE];
  DWORD cbBytesRead, cbWritten;
  BOOL fSuccess;
  HANDLE hPipe;
  // Создаем именованный канал
  hPipe = CreateNamedPipe(
    PIPE_NAME,           // имя
    PIPE_ACCESS_DUPLEX, // право на запись/чтение
    PIPE_TYPE_MESSAGE | // канал передачи сообщений
    PIPE_READMODE_MESSAGE | // канал чтения
    // сообщений
    PIPE_WAIT,          // блокирующий канал
    PIPE_UNLIMITED_INSTANCES,
    BUFSIZE,            // размер буфера вывод
    BUFSIZE,            // размер буфера ввод
    NMPWAIT_USE_DEFAULT_WAIT, // время ожидания
    // клиента
    NULL);
  if(hPipe == INVALID_HANDLE_VALUE)
  {
```

```
  ErrorReport(TEXT("CreateNamedPipe()"));
  return(1);
}
// Ожидаем запроса на соединение от клиента
if(!ConnectNamedPipe(hPipe, NULL))
{
  ErrorReport(TEXT("ConnectNamedPipe()"));
  return(1);
}

printf("Server started\n");
printf("Server: r> ");
// Основной цикл сервера
for(;;)
{
  // Принимаем данные от клиента
  fSuccess = ReadFile(
    hPipe, // дескриптор канала
    buf,    // буфер для записи данных из канала
    BUFSIZE*sizeof(TCHAR), // размер буфера
    &cbBytesRead, // реальное число байт,
    полученное из канала
    NULL); // используем синхронный режим
    // передачи
  if(!fSuccess || cbBytesRead == 0)
  {
    ErrorReport(TEXT("ReadFile()"));
    return(1);
  }
  // Если передана команда "disconnect", то
  // завершаем работу сервера
  if(!_tcscmp(buf, TEXT("disconnect"))) break;
  // Выводим полученные данные на консоль
  printf("%d:%s ", buf[0] - 'A' + 1, buf);
  // Пересылаем данные обратно клиенту
  fSuccess = WriteFile(
    hPipe, // дескриптор канала
    buf,    // буфер для передачи данных в канал
    (DWORD)(_tcslen(buf)+1)*sizeof(TCHAR),
    //количество байт для передачи
    &cbWritten, // реальное количество байт,
    // переданное через канал
    NULL); // синхронный режим передачи
  if(!fSuccess || cbWritten != DWORD)
  (_tcslen(buf)+1)*sizeof(TCHAR))
```

```
      {
      ErrorReport(TEXT("WriteFile()"));
      return(1);
    }
   }
   printf("\n\n\n");
   // Очищаем внутренний буфер канала
   FlushFileBuffers(hPipe);
   // Отсоединяем клиента
   DisconnectNamedPipe(hPipe);
   // Закрываем дескриптор канала
   CloseHandle(hPipe);
   return(0);
}

void ErrorReport(LPTSTR lpszFunction)
{
  LPVOID lpMsgBuf;
  LPVOID lpDisplayBuf;
  DWORD dw = GetLastError();

  FormatMessage(
    FORMAT_MESSAGE_ALLOCATE_BUFFER |
    FORMAT_MESSAGE_FROM_SYSTEM,
    NULL,
    dw,
    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
    (LPTSTR) &lpMsgBuf,
    0, NULL );

  lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,

  (lstrlen((LPCTSTR)lpMsgBuf)+lstrlen((LPCTSTR)lpsz
  Function)+40)*sizeof(TCHAR));
  _stprintf((LPTSTR)lpDisplayBuf,
    TEXT("%s failed with error %d: %s"),
    lpszFunction, dw, lpMsgBuf);
  MessageBox(NULL, (LPCTSTR)lpDisplayBuf,
  TEXT("Error"), MB_OK);

  LocalFree(lpMsgBuf);
  LocalFree(lpDisplayBuf);
}
```

Client.c

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

```
#include <tchar.h>

#define BUFSIZE 512
// Имя канала
#define PIPE_NAME TEXT("\\\\.\\pipe\\pipe_example")
void ErrorReport(LPTSTR lpszFunction);
int main()
{
  HANDLE hPipe;
  int i;
  BOOL fSuccess;
  TCHAR buf[BUFSIZE];
  DWORD cbWritten, cbRead, dwMode;
  // Ожидаем доступности именованного канала для
  // соединения
  if(!WaitNamedPipe(PIPE_NAME, NMPWAIT_WAIT_
  FOREVER))
  {
    ErrorReport("WaitNamedPipe()");
    return(1);
  }
  // Открываем канал на чтение/запись
  hPipe = CreateFile(
    PIPE_NAME,     // имя канала
    GENERIC_READ | // права на чтение и запись
    GENERIC_WRITE,
    0,             // нет общего доступа
    NULL,  // атрибуты безопасности по умолчанию
    OPEN_EXISTING, // открываем существующий канал
    0,             // атрибуты по умолчанию
    NULL);
  if(hPipe == INVALID_HANDLE_VALUE)
  {
    ErrorReport("CreateFile()");
    return(1);
  }
  // Изменяем режим чтения данных из канала
  // на режим сообщений
  dwMode = PIPE_READMODE_MESSAGE;
  if(!SetNamedPipeHandleState(hPipe, &dwMode,
  NULL, NULL))
  {
    ErrorReport("SetNamedPipeHandleState()");
    return(1);
  }
```

```
printf("Client started\n");
printf("Client: s> ");
// Основной цикл клиента
for(i = 'A'; i <= 'Z'; ++i)
{
  // Создаем запрос для сервера
  buf[0] = (TCHAR)i; buf[1] = (TCHAR)0;
  printf("%d:%s ", buf[0] - 'A' + 1, buf);
  // Передаем запрос через канал
  fSuccess = WriteFile(
    hPipe,     // дескриптор канала
    buf,       // сообщение
    (DWORD)(_tcslen(buf)+1)*sizeof(TCHAR),
    // длина сообщения
    &cbWritten, // реальное количество
    // переданных байт
    NULL);      // синхронный режим передачи
  if(!fSuccess)
  {
    ErrorReport("WriteFile()");
    return(1);
  }
  // Получаем ответ от сервера
  fSuccess = ReadFile(
    hPipe, // дескриптор канала
    buf,   // буфер для получения данных
    // из канала
    BUFSIZE*sizeof(TCHAR),  // размер буфера
    &cbRead,  // реальное количество байт,
    // полученных из канала
    NULL);    // синхронный режим передачи
  if(!fSuccess)
  {
    ErrorReport("ReadFile()");
    return(1);
  }
  // Проверяем корректность ответа сервера
  if(buf[0] != (TCHAR)i || buf[1] != '\0')
  {
    printf("Client: Wrong data received from the
    server");
    return(1);
  }
}
// Отсылаем команду на завершение работы сервера
```

```
  tcscpy(buf, TEXT("disconnect"));
  fSuccess = WriteFile(
    hPipe,    // дескриптор канала
    buf,      // сообщение
    (DWORD)(lstrlen(buf)+1)*sizeof(TCHAR), // длина
    сообщения
    &cbWritten, // реальное количество переданных байт
    NULL);    // синхронный режим передачи
  if(!fSuccess)
  {
    ErrorReport("WriteFile()");
    return(1);
  }
  printf("\n\n\n");
  // Закрываем канал
  CloseHandle(hPipe);
  return(0);
}

void ErrorReport(LPTSTR lpszFunction)
{
  LPVOID lpMsgBuf;
  LPVOID lpDisplayBuf;
  DWORD dw = GetLastError();

  FormatMessage(
    FORMAT_MESSAGE_ALLOCATE_BUFFER |
    FORMAT_MESSAGE_FROM_SYSTEM,
    NULL,
    dw,
    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
    (LPTSTR) &lpMsgBuf,
    0, NULL );

  lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,

  (lstrlen((LPCTSTR)lpMsgBuf)+lstrlen((LPCTSTR)lpsz
  Function)+40)*sizeof(TCHAR));
  _stprintf((LPTSTR)lpDisplayBuf,
    TEXT("%s failed with error %d: %s"),
    lpszFunction, dw, lpMsgBuf);
  MessageBox(NULL, (LPCTSTR)lpDisplayBuf,
  TEXT("Error"), MB_OK);

  LocalFree(lpMsgBuf);
  LocalFree(lpDisplayBuf);
}
```

После окончания работы клиента и сервера на консоли появятся следующие сообщения, показывая, что данные были успешно переданы между клиентом и сервером.

Результаты работы клиента (client.c):

```
Client started
Client: s> 1:A 2:B 3:C 4:D 5:E 6:F 7:G 8:H 9:I
10:J 11:K 12:L 13:M 14:N 15:O 16:P 17:Q 18:R 19:S
20:T 21:U 22:V 23:W 24:X 25:Y 26:Z
```

Результаты работы сервера (server.c):

```
Server started
Server: r> 1:A 2:B 3:C 4:D 5:E 6:F 7:G 8:H 9:I
10:J 11:K 12:L 13:M 14:N 15:O 16:P 17:Q 18:R 19:S
20:T 21:U 22:V 23:W 24:X 25:Y 26:Z
```

9.4.4. Почтовые ящики

Одним из простейших механизмов организации межпроцессного взаимодействия в Windows является механизм `mailslots` (*почтовые ящики*). Этот механизм позволяет осуществлять одностороннюю связь, которая по своей сути очень напоминает электронную почту: один процесс сохраняет сообщения в почтовом ящике, а второй забирает их оттуда.

Сам по себе почтовый ящик представляет собой псевдофайл, расположенный в памяти. Для работы с этим псевдофайлом используются стандартные функции открытия, записи, чтения и закрытия файлов. Однако в отличие от обычных файлов почтовые ящики являются временными файлами, и как только закрывается последний дескриптор, ссылающийся на некоторый почтовый ящик, то и сам ящик, и все сообщения, находящиеся в нем, удаляются.

Для создания почтового ящика используется функция `CreateMailslot()`:

```
#include <windows.h>
HANDLE CreateMailslot(
  LPCTSTR lpName,
  DWORD nMaxMessageSize,
  DWORD lReadTimeout,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes);
```

Параметр `lpName` задает имя создаваемого почтового ящика. Формат имени почтового ящика должен иметь вид `\\.\mailslot\[path]name`. Поле `name` должно быть уникально для каждого вновь создаваемого почтового ящика.

Параметр nMaxMessageSize задает максимальный размер сообщений, который может быть передан через создаваемый почтовый ящик. Если длина сообщений не ограничена, то данный параметр должен быть установлен в 0.

Параметр lReadTimeout задает время в миллисекундах, в течение которого операция чтения может ожидать доставки сообщения в почтовый ящик. Если данный параметр установлен в 0, то в случае отсутствия сообщения в почтовом ящике операция чтения завершается без какого-либо ожидания. При этом досрочное завершение операции чтения не считается ошибочным. Если же параметр установлен в значение MAILSLOT_WAIT_FOREVER(-1), то операция чтения блокируется до тех пор, пока не появится хотя бы одно сообщение.

Опциональный параметр lpSecurityAttributes задает указатель на структуру атрибутов безопасности, если пользователь хочет управлять данными атрибутами. Если же атрибуты безопасности должны быть унаследованы от прав текущего пользователя, то данный параметр может быть установлен в NULL.

В случае успешного создания нового почтового ящика функция CreateMailslot() возвращает дескриптор вновь созданного ящика, в противном случае возвращается значение INVALID_HANDLE_VALUE.

Следует отметить, что почтовый ящик, созданный функцией CreateMailslot(), доступен только на чтение. В него нельзя записать никакого сообщения. Если попытаться выполнить операцию записи в созданный ящик, то операция завершится с сообщением о недостаточности прав.

Для открытия почтового ящика на запись используется стандартная функция создания или открытия уже существующих файлов CreateFile():

```
#include <windows.h>
HANDLE CreateFile(
  LPCTSTR lpFileName,
  DWORD dwDesiredAccess,
  DWORD dwShareMode,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  DWORD dwCreationDisposition,
  DWORD dwFlagsAndAttributes,
  HANDLE hTemplateFile);
```

При открытии ранее созданного почтового ящика для записи параметр lpFileName должен задавать имя ранее созданного почтового ящика. Формат имени ящика должен соответствовать одному из форматов, приведенных в табл. 9.7.

Таблица 9.7. **Формат имен почтовых ящиков в функции `CreateFile()`**

Формат	Описание
`\\.\mailslot\name`	Возвращает дескриптор ранее созданного почтового ящика с именем `name` на локальной машине
`\\computername\mailslot\name`	Возвращает дескриптор ранее созданного почтового ящика с именем `name` на машине с именем `computername`
`\\domainname\mailslot\name`	Возвращает дескриптор для всех почтовых ящиков с именем `name`, созданных в домене `domainname`
`\\*\mailslot\name`	Возвращает дескриптор для всех почтовых ящиков с именем `name`, созданных в первичном системном домене

Как следует из приведенных в табл. 9.7 описаний, механизм почтовых ящиков может использоваться не только для организации межпроцессного взаимодействия на локальной машине, но и в сети между отдельными машинами или целыми доменами. Однако при передаче сообщений между компьютерами следует учитывать тот факт, что пересылаемые сообщения не должны быть длиннее 424 байт.

При открытии почтового ящика на запись параметр `dwDesiredAccess` должен быть установлен в значение `GENERIC_WRITE`, а параметр `dwShareMode` должен быть установлен в значение `FILE_SHARE_READ`. При этом открываемый на запись почтовый ящик должен быть создан. Если же делается попытка открыть несуществующий почтовый ящик, то функция `CreateFile()` возвращает значение `INVALID_HANDLE_VALUE`.

Для получения информации о существующем почтовом ящике используется функция `GetMailslotInfo()`:

```
#include <windows.h>
BOOL GetMailslotInfo(
  HANDLE hMailslot,
  LPDWORD lpMaxMessageSize,
  LPDWORD lpNextSize,
  LPDWORD lpMessageCount,
  LPDWORD lpReadTimeout);
```

В параметре `hMailslot` передается дескриптор ранее созданного почтового ящика. В параметре `lpMaxMessageSize` возвра-

щается максимально допустимый размер сообщения в байтах, если значение данного параметра не равно NULL. В параметре lpNextSize возвращается размер в байтах следующего сообщения, находящегося в ящике, если параметр не равен NULL. Если нет ни одного сообщения, то возвращается значение MAILSLOT_NO_MESSAGE(-1).

В параметре lpMessage-Count возвращается общее количество сообщений, которые могут быть прочитаны, если данный параметр не равен NULL. В параметре lpReadTimeout возвращается время, на которое может быть блокирован процесс при выполнении операции чтения сообщений, если в ящике нет ни одного сообщения. Это значение возвращается, если параметр не равен NULL.

Для изменения текущего значения таймаута при выполнении операции чтения используется функция SetMailslotInfo():

```
#include <windows.h>
BOOL SetMailslotInfo(
  HANDLE hMailslot,
  DWORD lReadTimeout);
```

В параметре hMailslot передается дескриптор ранее созданного почтового ящика, для которого необходимо изменить значение таймаута. В параметре lReadTimeout передается новое значение времени в миллисекундах, в течение которого операция чтения может ожидать доставки сообщения в почтовый ящик.

Рассмотрим реализацию примера с пересылкой отдельных символов с использованием почтовых ящиков. Клиент и сервер обмениваются сообщениями до тех пор, пока клиент не пошлет команду на закрытие соединения disconnect:

Mailslots.h

```
#include <stdio.h>
#include <windows.h>
#include <tchar.h>
// Имена почтовых ящиков для пересылки данных
// от сервера к клиенту
// и от клиента к серверу
#define ServerSlot TEXT("\\\\.\\mailslot\\from_
server")
#define ClientSlot TEXT("\\\\.\\mailslot\\from_
client")
// Функция сообщений о возникновении исключительных
// ситуаций
void ErrorReport(LPTSTR lpszFunction);
// Функция записи сообщения в заданный почтовый ящик
BOOL WriteSlot(HANDLE hSlot, LPTSTR lpszMessage);
```

```
// Функция чтения сообщений из заданного почтового
// ящика
BOOL ReadSlot(HANDLE hSlot, LPTSTR lpszBuffer);
```

Mailslots.c

```
BOOL WriteSlot(HANDLE hSlot, LPTSTR lpszMessage)
{
  BOOL fResult;
  DWORD cbWritten;
// Используем стандартную функцию записи WriteFile
// для записи сообщений в почтовый ящик
  fResult = WriteFile(hSlot,
    lpszMessage,
    (DWORD) (lstrlen(lpszMessage)+1)*sizeof(TCHAR),
    &cbWritten,
    (LPOVERLAPPED) NULL);
  if(!fResult)
  {
    ErrorReport("WriteFile()");
    return(FALSE);
  }
  return(TRUE);
}

BOOL ReadSlot(HANDLE hSlot, LPTSTR lpszBuffer)
{
  DWORD cbMessage, cMessage, cbRead;
  BOOL fResult;
  DWORD cAllMessages;
  HANDLE hEvent;
  OVERLAPPED ov;

  cbMessage = cMessage = cbRead = 0;
  if((hEvent = CreateEvent(NULL, FALSE, FALSE,
  TEXT("SlotServer"))) == NULL)
  {
    ErrorReport("CreateEvent()");
    return(FALSE);
  }
  ov.Offset = 0;
  ov.OffsetHigh = 0;
  ov.hEvent = hEvent;
// Ожидаем хотя бы одно сообщение в почтовом ящике
  do
  {
    fResult = GetMailslotInfo(hSlot,
```

```
      (LPDWORD)NULL,
      &cbMessage,
      &cMessage,
      (LPDWORD)NULL);
    if(!fResult)
    {
      ErrorReport("GetMailslotInfo()");
      return(FALSE);
    }
    Sleep(1000);
  }
  while(cbMessage == MAILSLOT_NO_MESSAGE);
// Считываем все сообщения из почтового ящика
  cAllMessages = cMessage;
  while(cMessage != 0)
  {
    lpszBuffer[0] = '\0';
    fResult = ReadFile(hSlot,
      lpszBuffer,
      cbMessage,
      &cbRead,
      &ov);
    if(!fResult)
    {
      ErrorReport("ReadFile()");
      GlobalFree((HGLOBAL)lpszBuffer);
      return(FALSE);
    }
// Считываем количество оставшихся сообщений
    fResult = GetMailslotInfo(hSlot,
      (LPDWORD) NULL,
      &cbMessage,
      &cMessage,
      (LPDWORD) NULL);
    if(!fResult)
    {
      ErrorReport("GetMailslotInfo()");
      return(FALSE);
    }
  }
  CloseHandle(hEvent);
  return(TRUE);
}
void ErrorReport(LPTSTR lpszFunction)
{
```

```
  LPVOID lpMsgBuf;
  LPVOID lpDisplayBuf;
  DWORD dw = GetLastError();

  FormatMessage(
    FORMAT_MESSAGE_ALLOCATE_BUFFER |
    FORMAT_MESSAGE_FROM_SYSTEM,
    NULL,
    dw,
    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
    (LPTSTR) &lpMsgBuf,
    0, NULL );

  lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,
  (lstrlen((LPCTSTR)lpMsgBuf)+lstrlen((LPCTSTR)lpsz
  Function)+40)* sizeof(TCHAR));
  _stprintf((LPTSTR)lpDisplayBuf,
    TEXT("%s failed with error %d: %s"),
    lpszFunction, dw, lpMsgBuf);
  MessageBox(NULL, (LPCTSTR)lpDisplayBuf,
      TEXT("Error"), MB_OK);
  LocalFree(lpMsgBuf);
  LocalFree(lpDisplayBuf);
}
```

Server.c

```
#include "Mailslots.h"
int main(int argc, const char *argv[])
{
  HANDLE hSlotServ, hSlotClient;
  int i;
  TCHAR buf[20];
  // Создаем почтовый ящик для чтения сообщений
  // от клиента
  hSlotClient = CreateMailslot(ClientSlot,
    0,
    MAILSLOT_WAIT_FOREVER,
    (LPSECURITY_ATTRIBUTES)NULL);
  if(hSlotClient == INVALID_HANDLE_VALUE)
  {
    ErrorReport("CreateMailslot() for client slot");
    return(1);
  }
  // Пытаемся открыть почтовый ящик для отсылки
  // сообщений клиенту
  do{
```

```
    hSlotServ = CreateFile(ServerSlot,
      GENERIC_WRITE,
      FILE_SHARE_READ,
      (LPSECURITY_ATTRIBUTES)NULL,
      OPEN_EXISTING,
      FILE_ATTRIBUTE_NORMAL,
      (HANDLE)NULL);
    Sleep(1000);
  }
  while(hSlotServ == INVALID_HANDLE_VALUE);
  printf("Server started\n");
  printf("Server: r> ");
  for(;;)
  {
    // Считываем сообщение от клиента
    if(!ReadSlot(hSlotClient, buf))
      return(1);
    // Если сообщение "disconnect", то останавливаем
    // сервер
    if(!_tcscmp(buf, TEXT("disconnect"))) break;
    // Выводим сообщение на консоль и отсылаем
    // обратно клиенту
    printf("%d:%s ", buf[0] - 'A' + 1, buf);
    if(!WriteSlot(hSlotServ, buf))
      return(1);
  }
  printf("\n\n\n");
  return(0);
}
```

Client.c

```
int main(int argc, const char *argv[])
{
  HANDLE hSlotServ, hSlotClient;
  int i;
  TCHAR buf[20];
  // Создаем почтовый ящик для приема сообщений
  // от сервера
  hSlotServ = CreateMailslot(ServerSlot,
    0,
    MAILSLOT_WAIT_FOREVER,
    (LPSECURITY_ATTRIBUTES)NULL);
  if(hSlotServ == INVALID_HANDLE_VALUE)
  {
    ErrorReport("CreateMailslot() for server slot");
```

```
    return(1);
  }
  // Пытаемся открыть почтовый ящик для отсылки
  // сообщений серверу
  do
  {
    hSlotClient = CreateFile(ClientSlot,
      GENERIC_WRITE,
      FILE_SHARE_READ,
      (LPSECURITY_ATTRIBUTES)NULL,
      OPEN_EXISTING,
      FILE_ATTRIBUTE_NORMAL,
      (HANDLE)NULL);
    Sleep(1000);
  }
  while(hSlotServ == INVALID_HANDLE_VALUE);
  printf("Client started\n");
  printf("Client: s> ");
  for(i = 'A'; i <= 'Z'; ++i)
  {
    // Формируем сообщение и отсылаем сообщение
    // серверу
    buf[0] = (TCHAR)i; buf[1] = (TCHAR)0;
    printf("%d:%s ", buf[0] - 'A' + 1, buf);
    if(!WriteSlot(hSlotClient, buf))
      return(1);
    // Считываем ответ от сервера и проверяем его
    if(!ReadSlot(hSlotServ, buf))
      return(1);
    if(buf[0] != (TCHAR)i || buf[1] != '\0')
    {
      printf("Client: Wrong data received from the
      server");
      return(1);
    }
  }
  // Отсылаем серверу команду на останов обработки
  if(!WriteSlot(hSlotClient, TEXT("disconnect")))
    return(1);
  printf("\n\n\n");
  CloseHandle(hSlotServ);
  CloseHandle(hSlotClient);
  return(0);
}
```

Запустив процесс-сервер и процесс-клиент, мы увидим следующий протокол работы данных процессов.

Результат работы клиента

```
Client started
Client: s> 1:A 2:B 3:C 4:D 5:E 6:F 7:G 8:H 9:I
10:J 11:K 12:L 13:M 14:N 15:O 16:P 17:Q 18:R 19:S
20:T 21:U 22:V 23:W 24:X 25:Y 26:Z
```

Результат работы сервера

```
Server started
Server: r> 1:A 2:B 3:C 4:D 5:E 6:F 7:G 8:H 9:I
10:J 11:K 12:L 13:M 14:N 15:O 16:P 17:Q 18:R 19:S
20:T 21:U 22:V 23:W 24:X 25:Y 26:Z
```

9.4.5. Общая память

В отличие от ОС Unix, в Windows отсутствует механизм работы с общей памятью в чистом виде. Вместо этого Windows поддерживает механизм работы с отображаемыми файлами. При использовании такого механизма процесс работает с памятью, принадлежащей процессу, но при этом изменяется реальный файл, расположенный на каком-либо устройстве. Механизм отображения файлов в память по своей сути очень похож на механизм работы с файлом подкачки. Более того, в качестве отображаемого файла можно использовать и непосредственно файл подкачки. При этом полностью имитируется работа с механизмом общей памяти, присутствующим в ОС Unix.

Прежде чем начать работу с отображаемыми файлами, необходимо создать соответствующий объект отображаемых файлов и получить дескриптор для этого объекта. Для этого используется функция `CreateFileMapping()`:

```
#include <windows.h>
HANDLE CreateFileMapping(
  HANDLE hFile,
  LPSECURITY_ATTRIBUTES lpAttributes,
  DWORD flProtect,
  DWORD dwMaximumSizeHigh,
  DWORD dwMaximumSizeLow,
  LPCTSTR lpName);
```

Параметр `hFile` задает дескриптор того файла, который должен быть отображен в память процесса. Если этот параметр установлен в значение `INVALID_HANDLE_VALUE`, то имитируется создание области общей памяти, которая может использоваться для орга-

Таблица 9.8. **Параметры доступа к отображаемым файлам**

Значение	Описание
PAGE_ READONLY	Создается отображаемый файл, доступный только для чтения
PAGE_ READWRITE	Создается отображаемый файл, доступный и для чтения, и для записи
PAGE_ WRITECOPY	Создается отображаемый файл с правами copy-on-write, т.е. при попытке записи в такой файл создается его копия, которая затем и модифицируется
PAGE_ EXECUTE_ READ	Создается отображаемый файл, доступный для чтения и исполнения
PAGE_ EXECUTE_ READWRITE	Создается отображаемый файл, доступный для чтения, записи и исполнения

низации межпроцессного взаимодействия. При этом выделяется область в файле подкачки.

Параметр lpAttributes, как и для других системных объектов, определяет, является ли наследуемым создаваемый дескриптор для отображаемого файла и задает атрибуты безопасности. Если этот параметр равен NULL, то создаваемый дескриптор не является наследуемым.

Параметр flProtect задает параметры доступа к создаваемому отображаемому файлу. Список параметров перечислен в табл. 9.8.

Параметры dwMaximumSizeHigh и dwMaximumSizeLow задают старшие и младшие части максимального размера отображаемого файла. Для работы с общей памятью хотя бы одно из этих чисел должно отличаться от 0.

Параметр lpName задает имя создаваемого отображаемого файла. Если этот параметр равен NULL, то создается анонимный отображаемый файл. Если же он не равен NULL, но объект с заданным именем уже существует, то функция возвращает дескриптор на существующий объект, если его свойства соответствуют таковым, запрошенным в параметре flProtect.

Если функции удается создать новый отображаемый файл или получить доступ к уже существующему отображаемому файлу, то возвращается дескриптор для такого объекта. В противном случае функия возвращает NULL.

Доступ к уже существующему объекту типа отображаемый файл можно также получить с помощью функции `OpenFileMapping()`:

```
#include <windows.h>
HANDLE OpenFileMapping(
  DWORD dwDesiredAccess,
  BOOL bInheritHandle,
  LPCTSTR lpName);
```

Параметр `dwDesiredAccess` устанавливает требуемые права доступа для существующего отображаемого файла. Этот параметр использует тот же набор значений, что и параметр `flProtect` в функции `CreateFileMapping()`. Параметр `bInheritHandle` определяет, будет ли наследоваться процессами-потомками создаваемый дескриптор или нет. Параметр `lpName` задает имя отображаемого файла, который необходимо открыть.

Прежде чем начать работу с отображаемым файлом, его необходимо отобразить на адресное пространство текущего процесса. Этот механизм очень похож на соответствующий в ОС Unix, в котором при работе с общей памятью также требуется отобразить область в адресное пространство процесса, прежде чем получить возможность записывать или считывать данные в/из общей памяти.

Для выполнения операции отображения файла в адресное пространство процесса используется функция `MapViewOfFile()`:

```
#include <windows.h>
LPVOID MapViewOfFile(
  HANDLE hFileMappingObject,
  DWORD dwDesiredAccess,
  DWORD dwFileOffsetHigh,
  DWORD dwFileOffsetLow,
  SIZE_T dwNumberOfBytesToMap);
```

Параметр `hFileMappingObject` задает дескриптор того отображаемого файла, который должен быть отображен на адресное пространство процесса.

Параметр `dwDesiredAccess` задает права для процесса к памяти, на которую отображается файл. Значения, которые может принимать данный параметр, перечислены в табл. 9.9.

Параметры `dwFileOffsetHigh` и `dwFileOffsetLow` задают старшую и младшую часть смещения начала отображаемого участка файла. Если данные параметры равны 0, то файл отображается, начиная с начала.

Параметр `dwNumberOfBytesToMap` задает размер отображаемого участка файла в байтах. Если этот параметр равен 0, то файл отображается, начиная от заданного смещения, и до конца.

Таблица 9.9. **Параметры доступа к памяти для отображаемого файла**

Значение	Описание
FILE_MAP_WRITE	Память доступна и для записи, и для чтения. Сам отображаемый файл также должен быть создан с правами PAGE_READWRITE
FILE_MAP_READ	Память доступна только для чтения. Отображаемый файл должен быть создан с правами PAGE_READWRITE либо PAGE_READONLY
FILE_MAP_COPY	Память обладает правами copy-on-write, т.е. при попытке записи в нее создается новый блок памяти, которая затем и модифицируется. Ассоциированный файл не изменяется. Отображаемый файл должен быть создан с правами PAGE_WRITECOPY
FILE_MAP_EXECUTE	Память обладает правами на исполнение, т.е. там может находиться код, которому можно передать управление. Отображаемый файл должен быть создан с правами PAGE_EXECUTE_READWRITE или PAGE_EXECUTE_READ

В случае успешного завершения данная функция возвращает адрес блока памяти, связанный с отображаемым файлом. В случае неудачи функция возвращает NULL.

После окончания работы с памятью, в которой отображен файл, эта память должна быть освобождена. Для этого используется функция UnmapViewOfFile():

```
#include <windows.h>
BOOL UnmapViewOfFile(LPCVOID lpBaseAddress)
```

Параметр lpBaseAddress должен задавать адрес, возвращаемый функцией MapViewOfFile().

Следует отметить, что так же, как и в ОС Unix, механизм общей памяти не обладает собственными средствами синхронизации процессов для разделения доступа к одному и тому же разделяемому файлу. Поэтому здесь приходится использовать какой-либо специализированный механизм синхронизации процессов.

Рассмотрим пример использования общей памяти в Windows для организации взаимодействия между двумя процессами.

Server.c

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

```
#include <tchar.h>

#define BUFSIZE 512
// Имя отображаемого файла
#define MAPFILE_NAME TEXT("SharedMemory_example")
// Имена семафоров для синхронизации операций
// с общей памятью
#define SEMAPHORE1_NAME TEXT("Semaphore1_example")
#define SEMAPHORE2_NAME TEXT("Semaphore2_example")
void ErrorReport(LPTSTR lpszFunction);

int main(int argc, char *argv[])
{
  LPTSTR buf;
  HANDLE hMapFile, hSem1, hSem2;
  // Создаем отображаемый файл, являющийся
  // именованной общей памятью
  hMapFile = CreateFileMapping(
    INVALID_HANDLE_VALUE, // используем файл под-
    // качки
    NULL,  // атрибуты безопасности по умолчанию
    PAGE_READWRITE, // доступ на запись и чтение
    0,             // старшая часть размера файла
    BUFSIZE,       // младшая часть размера файла
    MAPFILE_NAME); // имя отображаемого файла
  if(hMapFile == NULL)
  {
    ErrorReport(TEXT("CreateMappedFile()"));
    return(1);
  }
  // Отображаем файл в адресное пространство процесса
  buf = (LPTSTR)MapViewOfFile(hMapFile,
  // дескриптор отображаемого файла
    FILE_MAP_ALL_ACCESS,  // полный доступ к памяти
    0,
    0,
    BUFSIZE);
  if(buf == NULL)
  {
    ErrorReport(TEXT("MapViewOfFile()"));
    return(1);
  }
  // Создаем семафор с начальным несигнальным
  // состоянием
  if((hSem1 = CreateSemaphore(NULL, 0, 1,
```

```
  SEMAPHORE1_NAME)) == NULL)
  {
    ErrorReport(TEXT("CreateSemaphore()"));
    return(1);
  }
  // Ожидаем, пока клиент не создаст свой семафор
  WaitForSingleObject(hSem1, INFINITE);
  // Получаем дескриптор на созданный семафор
  if((hSem2 = OpenSemaphore(SEMAPHORE_ALL_ACCESS,
  FALSE, SEMAPHORE2_NAME)) == NULL)
  {
    ErrorReport(TEXT("OpenSemaphore()"));
    return(1);
  }
  // Передаем управление клиенту
  ReleaseSemaphore(hSem2, 1, NULL);
  printf("Server started\n");
  printf("Server: r> ");
  // Основной цикл сервера
  for(;;)
  {
    // Ожидаем, пока клиент не закончит запись в
    // общую память
    WaitForSingleObject(hSem2, INFINITE);
    // Если передана команда "disconnect", то
    // завершаем работу сервера
    if(!_tcscmp(buf, TEXT("disconnect"))) break;
    // Выводим полученные данные на консоль
    printf("%d:%s ", buf[0] - 'A' + 1, buf);
    // Пересылаем данные обратно клиенту
    buf[0] -= ('A' - 1);
    // Передаем управление клиенту
    ReleaseSemaphore(hSem1, 1, NULL);
  }
  printf("\n\n\n");
  // Освобождаем память, ассоциированную
  // с отображаемым файлом
  UnmapViewOfFile(buf);
  // Закрываем дескриптор отображаемого файла и
  // дескрипторы семафоров
  CloseHandle(hMapFile);
  CloseHandle(hSem1);
  CloseHandle(hSem2);
  return(0);
}
```

```
void ErrorReport(LPTSTR lpszFunction)
{
  LPVOID lpMsgBuf;
  LPVOID lpDisplayBuf;
  DWORD dw = GetLastError();

  FormatMessage(
    FORMAT_MESSAGE_ALLOCATE_BUFFER |
    FORMAT_MESSAGE_FROM_SYSTEM,
    NULL,
    dw,
    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
    (LPTSTR) &lpMsgBuf,
    0, NULL );

  lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,
  (lstrlen((LPCTSTR)lpMsgBuf)+lstrlen((LPCTSTR)lpsz
  Function)+40)*sizeof(TCHAR));
  _stprintf((LPTSTR)lpDisplayBuf,
    TEXT("%s failed with error %d: %s"),
    lpszFunction, dw, lpMsgBuf);
  MessageBox(NULL, (LPCTSTR)lpDisplayBuf,
  TEXT("Error"), MB_OK);

  LocalFree(lpMsgBuf);
  LocalFree(lpDisplayBuf);
}
```

Client.c

```
#include <windows.h>
#include <stdio.h>
#include <tchar.h>

#define BUFSIZE 512
// Имя отображаемого файла
#define MAPFILE_NAME TEXT("SharedMemory_example")
// Имена семафоров для синхронизации операций
// с общей памятью
#define SEMAPHORE1_NAME TEXT("Semaphore1_example")
#define SEMAPHORE2_NAME TEXT("Semaphore2_example")
void ErrorReport(LPTSTR lpszFunction);
int main()
{
  LPTSTR buf;
  HANDLE hMapFile, hSem1, hSem2;
  int i;
```

```
// Открываем существующий отображаемый файл
hMapFile = OpenFileMapping(
  FILE_MAP_ALL_ACCESS, // полный доступ к файлу
  FALSE,           // дескриптор не наследуется
  MAPFILE_NAME);   // имя отображаемого файла
if(hMapFile == NULL)
{
  ErrorReport("OpenFileMapping()");
  return(1);
}
// Отображаем файл в адресное пространство процесса
buf = (LPTSTR)MapViewOfFile(hMapFile,
// дескриптор отображаемого файла
  FILE_MAP_ALL_ACCESS, // полный доступ к памяти
  0,
  0,
  BUFSIZE);
if(buf == NULL)
{
  ErrorReport(TEXT("MapViewOfFile()"));
  return(1);
}
// Получаем дескриптор на созданный семафор
if((hSem1 = OpenSemaphore(SEMAPHORE_ALL_ACCESS,
FALSE, SEMAPHORE1_NAME)) == NULL)
{
  ErrorReport(TEXT("OpenSemaphore()"));
  return(1);
}
// Создаем семафор с начальным несигнальным
// состоянием
if((hSem2 = CreateSemaphore(NULL, 0, 1,
SEMAPHORE2_NAME)) == NULL)
{
  ErrorReport(TEXT("CreateSemaphore()"));
  return(1);
}
// Передаем управление серверу
ReleaseSemaphore(hSem1, 1, NULL);
// Ожидаем, пока сервер не передаст ответ
WaitForSingleObject(hSem2, INFINITE);

printf("Client started\n");
printf("Client: s> ");
// Основной цикл клиента
```

```
  for(i = 'A'; i <= 'Z'; ++i)
  {
    // Записываем данные в область общей памяти
    buf[0] = (TCHAR)i; buf[1] = (TCHAR)0;
    printf("%d:%s ", buf[0] - 'A' + 1, buf);
    // Передаем управление серверу
    ReleaseSemaphore(hSem2, 1, NULL);
    // Ожидаем, пока сервер не освободит общую
    // память
    WaitForSingleObject(hSem1, INFINITE);
    if(buf[0] != (TCHAR)(i - 'A' + 1) || buf[1]
    != (TCHAR)'\0')
    {
      printf("Client: Wrong data received from the
      server");
      return(1);
    }
  }
  // Передаем команду серверу на завершение работы
  _tcscpy(buf, TEXT("disconnect"));
  // Передаем управление серверу
  ReleaseSemaphore(hSem2, 1, NULL);
  printf("\n\n\n");
  // Освобождаем память, ассоциированную
  // с отображаемым файлом
  UnmapViewOfFile(buf);
  // Закрываем дескриптор отображаемого файла и
  // дескрипторы семафоров
  CloseHandle(hMapFile);
  CloseHandle(hSem1);
  CloseHandle(hSem2);
  return(0);
}

void ErrorReport(LPTSTR lpszFunction)
{
  LPVOID lpMsgBuf;
  LPVOID lpDisplayBuf;
  DWORD dw = GetLastError();

  FormatMessage(
    FORMAT_MESSAGE_ALLOCATE_BUFFER |
    FORMAT_MESSAGE_FROM_SYSTEM,
    NULL,
    dw,
```

```
    MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
    (LPTSTR) &lpMsgBuf,
    0, NULL );

  lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,
  (lstrlen((LPCTSTR)lpMsgBuf)+lstrlen((LPCTSTR)lpsz
  Function)+40)*sizeof(TCHAR));
  _stprintf((LPTSTR)lpDisplayBuf,
    TEXT("%s failed with error %d: %s"),
    lpszFunction, dw, lpMsgBuf);
  MessageBox(NULL, (LPCTSTR)lpDisplayBuf,
  TEXT("Error"), MB_OK);

  LocalFree(lpMsgBuf);
  LocalFree(lpDisplayBuf);
}
```

В приведенном примере для синхронизации операций чтения/записи в/из общей памяти используются два семафора. Один из семафоров (дескриптор `hSem1`) используется для блокирования памяти при выполнении операций клиентом, а второй (`hSem2`) — для блокирования памяти при выполнении операций сервером. Без этих семафоров возникла бы конфликтная ситуация между двумя процессами, когда два процесса одновременно писали бы в одну и ту же память.

В результате работы клиента на консоль выводится следующее сообщение:

```
Client started
Client: s> 1:A 2:B 3:C 4:D 5:E 6:F 7:G 8:H 9:I
10:J 11:K 12:L 13:M 14:N 15:O 16:P 17:Q 18:R 19:S
20:T 21:U 22:V 23:W 24:X 25:Y 26:Z
```

Результатом работы сервера будут являться следующие сообщения:

```
Server started
Server: r> 1:A 2:B 3:C 4:D 5:E 6:F 7:G 8:H 9:I
10:J 11:K 12:L 13:M 14:N 15:O 16:P 17:Q 18:R 19:S
20:T 21:U 22:V 23:W 24:X 25:Y 26:Z
```

Контрольные вопросы

1. В чем состоят различия механизмов прерываний и сигналов?
2. Каким образом производится обработка полученного сигнала процессом? Какие возможны варианты обработки сигнала?
3. Чем обусловлены потери сигналов?

4. Каким образом принимаются из очереди сообщения, размер которых больше ожидаемого?

5. Как реализованы операции P (post) и V (wait) для работы с семафорами в UNIX-системах? В чем состоят основные различия бинарных семафоров и семафоров-счетчиков?

6. Почему необходимо использовать средства синхронизации процессов при использовании общей памяти?

7. Чем отличаются неименованные каналы от именованных? Как процесс может записать и прочитать данные из канала?

8. Что такое сокет и коммуникационный домен? Какие существуют модели передачи данных при организации межпроцессного взаимодействия и в чем их различия?

9. Что будет выведено на экран в результате выполнения следующей программы:

```
#include <signal.h>
int alarmHandler(int sig)
{
  printf("ABC\n");
}
int main(void)
{
  signal(SIGALRM, &alarmHandler);
  kill(getpid(), SIGTERM);
  pause();
  return 0;
}
```

10. Напишите программу, которая после своего запуска порождает пять процессов: процесс-родитель, время порождения которого совпадает с временем запуска программы, и четыре процесса-потомка. Процесс-родитель должен запустить ALARM-таймер на 20 с, по получению от него сигнала SIGALRM завершить выполнение потомков сигналом SIGKILL и завершить свое выполнение. Родитель должен посылать потомкам сигнал SIGALRM в цикле каждую секунду, а они — выводить в ответ на экран свой PID и количество полученных сигналов SIGALRM.

11. В чем разница между потоком и процессом? Сколько потоков существует при старте процесса?

12. Чем отличаются наследуемые дескрипторы объектов от ненаследуемых? Почему при работе с каналами следует попеременно отключать и затем включать наследование дескрипторов канала?

13. Чем отличаются автоматически сбрасываемые события от событий, сбрасываемых вручную?

14. В чем состоят основные различия семафоров и мьютексов?

15. Как с помощью почтовых ящиков организовать взаимодействие процессов, находящихся на разных машинах? Какие ограничения при этом существуют?

16. Напишите программу, которая запускает процесс `qprocess *` и перенаправляет выводимые данные в файл.

17. Напишите две программы, одна из которых является сервером, а другая — клиентом. Программа-сервер получает в командной строке список строк и организует их в динамический список в общей памяти. Процесс-клиент считывает этот список из общей памяти и последовательно выводит его элементы на консоль.

ПРИЛОЖЕНИЕ 1

Программный комплекс «Контроль знаний». Структура каталогов

Общая постановка задачи разработки программного комплекса «Контроль знаний» приведена в подразд. 1.4. Каталоги и файлы этого комплекса размещаются в каталоге `check` и имеют следующую структуру и права доступа:

```
check        devel/teacher 755 (rwxr-xr-x)
├──scripts     devel/teacher 755 (rwxr-xr-x)
│    ├──students    devel/teacher 755 (rwxr-xr-x)
│    └──teacher    devel/teacher 755 (rwxr-xr-x)
├──students     devel/teacher 755 (rwxr-xr-x)
│    ├──fedya    fedya/teacher 730 (rwx-wx---)
│    │    └──ready   fedya/teacher 770 (rwxrwx---)
│    ├──ivan     ivan/teacher 730 (rwx-wx---)
│    │    └──ready   ivan/teacher 770 (rwxrwx---)
│    ├──kolya    kolya/teacher 730 (rwx-wx---)
│    │    └──ready   kolya/teacher 770 (rwxrwx---)
│    ├──petr     petr/teacher 730 (rwx-wx---)
│    │    └──ready   petr/teacher 770 (rwxrwx---)
└──teacher     teacher/teacher 770 (rwx------)
     ├──theme1    teacher/teacher 770 (rwx------)
     ├──theme2    teacher/teacher 770 (rwx------)
     ├──theme3    teacher/teacher 770 (rwx------)
     ├──theme4    teacher/teacher 770 (rwx------)
     └──works    teacher/teacher 770 (rwx------)
```

В каталоге `students` находятся рабочие области студентов. Каждая область представляет собой каталог с именем, соответствующим имени студента, в котором хранятся работы, выполняемые студентом, и подкаталог `ready`, в котором хранятся уже выполненные работы.

В каталоге `teacher` хранится рабочая область преподавателя с базой вариантов контрольных работ и каталогом с собранными работами. База вариантов контрольных работ состоит из системы каталогов, каждый из которых соответствует одной теме. Имя каталога темы — `theme<N>`, где `N` — номер темы от 1 и далее. Каждый вариант контрольной работы представляет собой файл с именем `var<N>.txt`, где `N` — номер варианта от 1 и далее. Первая строка файла должна содержать номер темы и номер варианта.

Пример файла варианта контрольной работы:

```
Тема 1 «Структуры данных в языке С»
Вариант 1

Вопрос 1: Какой объем памяти занимает переменная типа
signed int?
Ответ: _____

Вопрос 2: Какой объем памяти занимает следующая струк-
тура?
packed struct {
  int       a;
  char[19]  b;
  float     c;
} example_struct;
Ответ: _____
```

Каталог с собранными работами содержит файлы с вариантами контрольных работ, выполненных студентами. Формат имени файла: <имя>theme<номер темы>_var<номер варианта>.txt. Например, первый вариант второй темы, выполненный студентом vasya, будет называться vasyatheme2_var1.txt.

В программном комплексе определены три типа пользователей:

- разработчик — учетное имя devel, член группы teacher;
- преподаватель — учетное имя teacher, член группы teacher;
- студенты — учетные имена произвольные, не входят в группу teacher.

За счет выделения отдельной группы teacher можно предотвратить несанкционированный доступ студентов к каталогу с вариантами контрольных, а разграничение прав доступа сводит к минимуму вероятность списывания (просмотра контрольных у другого студента).

Программный комплекс «Контроль знаний». Исходные тексты

Общая постановка задачи разработки программного комплекса «Контроль знаний» приведена в подразд. 1.4.

Задания, входящие в этот комплекс, написаны на языке командного интерпретатора BASH.

Задания разделены на три группы — задания для работы преподавателя, размещенные в каталоге `check/scripts/teacher`, задания для работы студентов, которые хранятся в каталоге `check/scripts/students`, а также служебные задания, помещенные в каталог `check/scripts`.

Для работы заданий преподавателя и студентов необходимо присвоить значение переменной окружения `BASEDIR`, которая задает полное имя каталога программного комплекса «Контроль». Для работы заданий студента необходимо еще установить значение переменной `NAME`, равное имени каталога, хранящего рабочую область студента. Для задания этих переменных используется задание `scripts/env.sh`.

Типичный сеанс работы с программным комплексом выглядит следующим образом (предположим, что он установлен в каталоге `/check`):

Преподаватель

```
$ . /check/scripts/env.sh
$ give_all.sh 3
Студенту fedya выдан вариант 1 по теме 3
Студенту ivan выдан вариант 2 по теме 3
Студенту kolya выдан вариант 3 по теме 3
Студенту petr выдан вариант 1 по теме 3
Студенту sergey выдан вариант 2 по теме 3
Студенту vasya выдан вариант 3 по теме 3

$ gather.sh
У студента fedya отсутствует каталог ready или он не
доступен!
У студента ivan получен файл с работой theme2_var2.txt
У студента ivan получен файл с работой theme3_var1.txt
У студента kolya получен файл с работой theme4_var1.txt
У студента petr отсутствует каталог ready или он не
доступен!
```

```
У студента sergey отсутствует каталог ready или он не
доступен!
```

Студент

```
$ . /check/scripts/env.sh
$ list.sh
Тема 1 Вариант 3: theme1_var3.txt
Тема 3 Вариант 1: theme3_var1.txt
Тема 4 Вариант 3: theme4_var3.txt
$ do.sh 1 3
..... запуск текстового редактора ....
$ ready.sh 1 3
Вариант 3 темы 1 сдан
```

Исходные тексты заданий

scripts/env.sh — установка переменных окружения

```
#!/bin/bash
# Задание предназначено для установки переменных
# окружения пользователя для корректной
# работы с программным комплексом

if [ "${EDITOR:-DUMMY}" == "DUMMY" ] ; then
  export EDITOR=mcedit
  echo "EDITOR=$EDITOR"
fi

export BASEDIR=/check/
echo "BASEDIR=$BASEDIR"
export NAME=`whoami`
echo "NAME=$NAME"
export PATH=$PATH:$BASEDIR/scripts/teacher: \
$BASEDIR/scripts/students
echo "PATH=$PATH"
```

scripts/rights.sh — установка прав доступа на каталоги системы

```
#!/bin/bash
# Задание предназначено для установки прав доступа к
каталогам программного комплекса
# Установка прав доступа на основные каталоги
chown devel.teacher $BASEDIR
chmod 755 $BASEDIR
chown -R devel.teacher $BASEDIR/scripts
chmod -R 755 $BASEDIR/scripts
chown devel.teacher $BASEDIR/students
chmod 755 $BASEDIR/students
```

```
TEACHERDIR=$BASEDIR/teacher
chown teacher.teacher $TEACHERDIR
chmod 700 $TEACHERDIR

# Установка прав доступа на все подкаталоги каталога
# преподавателя. Доступно только владельцу и группе
# (т.е. только преподавателю и разработчику)
for i in `ls $TEACHERDIR` ; do
  chown teacher.teacher $TEACHERDIR/$i
  chmod 770 $TEACHERDIR/$i
done

# Установка прав доступа на каталоги студентов
# Каждый студент - владелец своего каталога
# При этом член группы teacher может писать (не читать)
# в каталог студента и читать из каталога ready каждого
# студента
for i in `ls $BASEDIR/students` ; do
    STUDENTDIR=$BASEDIR/students/$i
    READYDIR=$STUDENTDIR/ready
    if [ -d $STUDENTDIR ]; then
  chown $i.teacher $STUDENTDIR
  chmod 730 $STUDENTDIR
    fi
    if [ -d $READYDIR ]; then
  chown $i.teacher $STUDENTDIR
  chmod 770 $STUDENTDIR
    fi
done
```

scripts/teacher/makedirs.sh — создание каталогов системы

```
#!/bin/bash

# Задание предназначено для первоначального создания
# всех каталогов программного комплекса "Контроль
знаний"
mkdir -p $BASEDIR
mkdir $BASEDIR/scripts
mkdir $BASEDIR/scripts/students
mkdir $BASEDIR/scripts/teacher
mkdir $BASEDIR/students
mkdir $BASEDIR/students/fedya
mkdir $BASEDIR/students/fedya/ready
mkdir $BASEDIR/students/ivan
mkdir $BASEDIR/students/ivan/ready
mkdir $BASEDIR/students/kolya
```

```
mkdir $BASEDIR/students/kolya/ready
mkdir $BASEDIR/students/petr
mkdir $BASEDIR/students/petr/ready
mkdir $BASEDIR/teacher
mkdir $BASEDIR/teacher/theme1
mkdir $BASEDIR/teacher/theme2
mkdir $BASEDIR/teacher/theme3
mkdir $BASEDIR/teacher/theme4
mkdir $BASEDIR/teacher/works
```

scripts/teacher/give.sh — выдача задания студенту

```
#!/bin/bash

# Задание предназначено для выдачи варианта по заданной
# теме определенному студенту
#
# Параметры вызова :
#   $1 - номер темы
#   $2 - номер варианта
#   $3 - учетное имя студента
# Рабочие переменные окружения:
#      BASEDIR - основной каталог системы

# Начальные проверки
if [ "${BASEDIR:-DUMMY}" == "DUMMY" ] ; then
  echo "Переменная \$BASEDIR не задана"
  exit 100
fi

if [ ! -d $BASEDIR -o ! -r $BASEDIR -o ! -x $BASEDIR ]
then
  echo "$BASEDIR не является каталогом или не доступен"
  exit 101
fi

if [ $# -ne 3 ]; then
  echo "Формат вызова:"
  echo "`basename $0` <N темы> <N варианта> \
  <имя студента>"
  exit 1
fi

THEMEDIR=$BASEDIR/teacher/theme$1
if [ ! -d $THEMEDIR ]; then
  echo "Невозможно открыть каталог с темой $1"
  exit 2
fi
```

```
if [ ! -r $THEMEDIR -o ! -x $THEMEDIR ]; then
  echo "Каталог с темой $1 недоступен по чтению \
  или выполнению"
  exit 3
fi

VARIANTNAME=$THEMEDIR/var$2.txt
if [ ! -f $VARIANTNAME ]; then
  echo "Файл варианта $2 в теме $1 не существует"
  exit 3
fi

if [ ! -r $VARIANTNAME ]; then
  echo "Файл варианта $2 в теме $1 не доступен \
  по чтению"
  exit 4
fi

STUDENTDIR=$BASEDIR/students/$3
if [ ! -d $STUDENTDIR ]; then
  echo "Отсутствует рабочий каталог студента $3"
  exit 5
fi

if [ ! -w $STUDENTDIR ]; then
  echo "Рабочий каталог студента $3 не доступен \
  по записи"
  exit 5
fi

# Основная часть

cp $VARIANTNAME $STUDENTDIR/theme$1_var$2.txt

RETCODE=$?

if [ $RETCODE -ne 0 ]; then
  echo "Не удалось скопировать файл $1\
  варианта $2 темы студенту $3"
  RETCODE=`expr $RETCODE + 100`
  exit $RETCODE
fi

chmod 666 $STUDENTDIR/theme$1_var$2.txt
RETCODE=$?

if [ $RETCODE -ne 0 ]; then
  echo "Не удалось установить права доступа для файла \
  варианта $2 темы $1 студенту $3"
```

```
  RETCODE=`expr $RETCODE + 50`
  exit $RETCODE
fi
```

scripts/teacher/look.sh — просмотр количества вариантов по теме

```
#!/bin/bash

# Задание предназначено для выдачи количества вариантов
# задания по заданной теме
#
# Параметры вызова :
#  $1 - номер темы
# Рабочие переменные окружения:
#     BASEDIR - основной каталог системы

# Начальные проверки

if [ $# -ne 1 ] ; then
  echo "Формат вызова"
  echo "`basename $0` <номер темы>"
  exit 1
fi

if [ "${BASEDIR:-DUMMY}" == "DUMMY" ] ; then
  echo "Переменная \$BASEDIR не задана"
  exit 100
fi

if [ ! -d $BASEDIR -o ! -r $BASEDIR -o ! -x $BASEDIR ]
then
  echo "$BASEDIR не является каталогом или не доступен"
  exit 101
fi

# Основная часть

NUM_VARIANTS=0

for i in `ls $BASEDIR/teacher/theme$1` ; do
  NUM_VARIANTS=`expr $NUM_VARIANTS + 1`   # подсчет
  # количества # вариантов по теме
done

echo $NUM_VARIANTS

exit $NUM_VARIANTS
```

scripts/teacher/give_all.sh — выдача контрольной всем студентам

```
#!/bin/bash
# Задание предназначено для выдачи вариантов заданий
```

```
# по заданной теме всем студентам
#
# Параметры вызова :
#  $1 - номер темы
# Рабочие переменные окружения:
#  BASEDIR - основной каталог системы

# Начальные проверки

if [ $# -ne 1 ] ; then
  echo "Формат вызова"
  echo "`basename $0` <номер темы>"
  exit 1
fi

if [ "${BASEDIR:-DUMMY}" == "DUMMY" ] ; then
  echo "Переменная \$BASEDIR не задана"
  exit 100
fi

if [ ! -d $BASEDIR -o ! -r $BASEDIR -o ! -x $BASEDIR ]
then
  echo "$BASEDIR не является каталогом или не досту-
  пен"
  exit 101
fi

# Основная часть

NUM_VARIANTS=`$BASEDIR/scripts/teacher/look.sh $1`
if [ "${NUM_VARIANTS:-DUMMY}" == "DUMMY" ] ; then
  echo "Невозможно получить общее число вариантов"
  exit 201
fi

i=1 # Счетчик номера варианта
for j in `ls $BASEDIR/students` ; do
  # вывод имени студента и номера варианта
  echo "Студенту $j выдан вариант $i по теме $1"
  # запуск задания give.sh для выдачи задания
  $BASEDIR/scripts/teacher/give.sh $1 $i $j
  i=`expr $i + 1`

  # если счетчик дошел до максимального номера,
  if [ $i -gt $NUM_VARIANTS ] ; then
    i=1 # сбрасываем его
  fi
done
```

scripts/teacher/gather.sh — сбор выполненных контрольных

```
#!/bin/bash

# Задание предназначено для сбора всех выполненных
# контрольных из подкаталога ready рабочего каталога
# студента
#
# Параметры вызова : отсутствуют
# Рабочие переменные окружения:
#     BASEDIR - основной каталог системы

# Начальные проверки
if [ "${BASEDIR:-DUMMY}" == "DUMMY" ] ; then
  echo "Переменная \$BASEDIR не задана"
  exit 100
fi

if [ ! -d $BASEDIR -o ! -r $BASEDIR -o ! -x $BASEDIR ]
then
  echo "$BASEDIR не является каталогом или не доступен"
  exit 101
fi

WORKSDIR=$BASEDIR/teacher/works
if [ ! -d $WORKSDIR -o ! -r $WORKSDIR -o ! \
-x $WORKSDIR ]; then
  echo "$WORKSDIR не является каталогом или \
  не доступен"
  exit 101
fi

# Основная часть

for i in `ls $BASEDIR/students` ; do
    READYDIR=$BASEDIR/students/$i/ready
  if [ ! -d $READYDIR -o \
    ! -x $READYDIR -o \
    ! -r $READYDIR ] ; then
  echo "У студента $i отсутствует или не доступен \
    каталог ready"
  else
    for j in `ls $READYDIR` ; do
      mv $READYDIR/$j $WORKSDIR/$i-$j
      RETCODE=$?
      if [ $RETCODE -ne 0 ] ; then
        echo "У студента $i недоступен файл \
        с работой $j"
      else
```

```
        echo "У студента $i получен файл \
        с работой $j"
        fi
      done
    fi
  done
```

scripts/students/list.sh — вывод всех полученных вариантов

```
#!/bin/bash

# Задание предназначено для вывода имен всех файлов с уже
# полученными, но еще не выполненными контрольными и тем
# контрольных на экран
#
# Параметры вызова : отсутствуют
# Рабочие переменные окружения:
#    BASEDIR - основной каталог системы
#    NAME - учетное имя пользователя

# Начальные проверки

if [ "${BASEDIR:-DUMMY}" == "DUMMY" ] ; then
  echo "Переменная \$BASEDIR не задана"
  exit 100
fi

if [ ! -d $BASEDIR -o ! -r $BASEDIR -o ! -x $BASEDIR ]
then
  echo "$BASEDIR не является каталогом или не доступен"
  exit 101
fi

if [ "${NAME:-DUMMY}" == "DUMMY" ] ; then
  echo "Переменная \$NAME не задана"
  exit 200
fi

STUDENTDIR=$BASEDIR/students/$NAME/
if [ ! -d $STUDENTDIR -o \
    ! -r $STUDENTDIR -o \
    ! -x $STUDENTDIR ] ; then
  echo "$STUDENTDIR не является каталогом или \
  не доступен"
  exit 201
fi

# Основная часть

# Цикл по всем файлам в каталоге студента
for i in `ls $STUDENTDIR` ; do
```

```
  FULLNAME=$STUDENTDIR/$i # Полное имя файла варианта
  # Проверка соответствия имени файла
  # заданному шаблону имени файла контрольной
  echo $i | grep -q -E 'theme[0-9]+_var[0-9]+.txt'
  if [ $? -eq 0 ] ; then
    if [ ! -f $FULLNAME -o \
      ! -r $FULLNAME -o \
      ! -w $FULLNAME ] ; then
      echo "Нет доступа к файлу варианта $FULLNAME"
    else
      # Вывод темы контрольной и имени ее файла
      # в формате Тема : Имя
      echo "`head -n 1 $FULLNAME`: $i"
    fi
  fi
done
```

scripts/students/do.sh — выполнение заданного варианта

```
#!/bin/bash

# Задание предназначено для запуска текстового
# редактора для выполнения варианта контрольной
#
# Параметры вызова :
# $1 - номер темы
# $2 - номер варианта
# Рабочие переменные окружения:
#     BASEDIR - основной каталог системы
#     NAME - учетное имя пользователя
#     EDITOR - имя файла текстового редактора
#                по умолчанию
# Устанавливается в большинстве UNIX-систем

# Начальные проверки

if [ "${BASEDIR:-DUMMY}" == "DUMMY" ] ; then
  echo "Переменная \$BASEDIR не задана"
  exit 100
fi

if [ ! -d $BASEDIR -o ! -r $BASEDIR -o ! -x $BASEDIR ]
then
  echo "$BASEDIR не является каталогом или не доступен"
  exit 101
fi

if [ $# -ne 2 ]; then
```

```
  echo "Формат вызова:"
      echo "`basename $0` <N темы> <N варианта>"
  exit 1
fi

if [ "${NAME:-DUMMY}" == "DUMMY" ] ; then
  echo "Переменная \$NAME не задана"
  exit 200
fi

STUDENTDIR=$BASEDIR/students/$NAME
if [ ! -d $STUDENTDIR -o \
  ! -r $STUDENTDIR -o \
  ! -x $STUDENTDIR ] ; then
  echo "$STUDENTDIR не является каталогом или \
  не доступен"
  exit 201
fi

if [ ! -f $STUDENTDIR/theme$1_var$2.txt ] ; then
  echo "Не существует файла theme$1_var$2.txt"
  exit 1
fi

READYDIR=$STUDENTDIR/ready/
if [ ! -d $READYDIR ]; then
  echo "Невозможно открыть каталог для сделанных \
  работ ready"
  exit 2
fi

if [ ! -r $READYDIR -o ! -x $READYDIR ]; then
  echo "Каталог для сделанных работ ready недоступен \
  по чтению или выполнению"
  exit 3
fi

if [ "${EDITOR:-DUMMY}" == "DUMMY" ] ; then
  echo "Переменная \$EDITOR не задана"
  exit 150
fi

$EDITOR $STUDENTDIR/theme$1_var$2.txt
```

scripts/students/ready.sh — сдача выполненного варианта

```
#!/bin/bash

# Задание предназначено для сдачи выполненной работы
преподавателю
```

```
#
# Параметры вызова :
# $1 - номер темы
# $2 - номер варианта
# Рабочие переменные окружения:
#     BASEDIR - основной каталог системы
#     NAME - учетное имя пользователя

# Начальные проверки

if [ "${BASEDIR:-DUMMY}" == "DUMMY" ] ; then
  echo "Переменная \$BASEDIR не задана"
  exit 100
fi

if [ ! -d $BASEDIR -o ! -r $BASEDIR -o ! -x $BASEDIR ]
then
  echo "$BASEDIR не является каталогом или не доступен"
  exit 101
fi

if [ $# -ne 2 ]; then
  echo "Формат вызова:"
  echo "`basename $0` <N темы> <N варианта>"
  exit 1
fi

if [ "${NAME:-DUMMY}" == "DUMMY" ] ; then
  echo "Переменная \$NAME не задана"
  exit 200
fi

STUDENTDIR=$BASEDIR/students/$NAME
if [ ! -d $STUDENTDIR -o \
  ! -r $STUDENTDIR -o \
  ! -x $STUDENTDIR ] ; then
  echo "$STUDENTDIR не является каталогом или не до-
  ступен"
  exit 201
fi

if [ ! -f $STUDENTDIR/theme$1_var$2.txt ] ; then
  echo "Не существует файла theme$1_var$2.txt"
  exit 1
fi

READYDIR=$STUDENTDIR/ready
if [ ! -d $READYDIR ]; then
  echo "Невозможно открыть каталог для сделанных \
```

```
    работ ready”
  exit 2
fi

if [ ! -r $READYDIR -o ! -x $READYDIR ]; then
  echo “Каталог для сделанных работ ready не доступен \
  для чтения или выполнения”
  exit 3
fi

# Основная часть

mv $STUDENTDIR/theme$1_var$2.txt $READYDIR

RETCODE=$?

if [ $RETCODE -ne 0 ]; then
  echo “Не удалось скопировать файл варианта $2 темы $1”
  RETCODE=`expr $RETCODE + 100`
  exit $RETCODE
else
  echo “Вариант $2 темы $1 сдан”
fi
```

Краткий справочник по командам UNIX

В приложении приведена краткая справочная информация по основным командам UNIX. Более полную информацию можно получить, вызвав страницу помощи `man` для каждой команды. Если команда помечена как внутренняя команда `bash` (под ее названием сделана пометка (`BASH`)), то информацию по ней можно получить, вызвав страницу помощи `bash`.

Условные обозначения:

В таблице в квадратных скобках `[ ]` указаны необязательные конструкции, например, необязательные параметры, в угловых скобках < > — обязательные параметры.

Основные команды UNIX

Команда	Описание

`.(BASH)`	Запуск задания BASH в той же копии командного интерпретатора
	Формат вызова:
	`. <имя задания>`
	Параметры:
	`<имя задания>` — имя запускаемого задания.
	Пример использования
	`. /usr/bin/purge_script.sh`

`:(BASH)`	Пустая команда, код возврата которой всегда равен 0
	Формат вызова:
	`:`
	Параметры:
	отсутствуют.
	Пример использования:
	`while : ; do`
	`echo "Infinite loop"`
	`done`

`break (BASH)`	Выход из цикла `for` или `while` **Формат вызова:** `break [n]`

Команда	Описание

	Параметры:
	n — количество вложенных циклов, из которых происходит выход.
	Пример использования:
	`for i in `ls` ; do`
	`  cat $i`
	`  if [ "$i" == "end" ] ; then`
	`  break`
	`  fi`
	`done`

cat	Вывод содержимого файлов в стандартный поток вывода
	Формат вызова:
	`cat [ параметры ] <имя файла>`
	Параметры:
	-s — заменять несколько пустых строк, идущих подряд в файле, на одну;
	-E — показывать символ $ после конца каждой строки.
	Пример использования:
	`cat –s /home/sergey/file.txt`

cd, chdir (BASH)	Переход в указанный каталог
	Формат вызова:
	`cd [имя каталога]`
	Параметры:
	[имя каталога] — полное или относительное имя каталога, в который осуществляется переход. Если параметр не задан или задано имя ~ — переход осуществляется в домашний каталог пользователя.
	Если в качестве имени задано ~<учетное имя пользователя>, то осуществляется переход в домашний каталог этого пользователя (при наличии достаточных прав).
	Пример использования:
	`cd /usr/local/bin`
	`cd ~alex`

chgrp	Изменение группы-владельца для заданных файлов **Формат вызова:** `chgrp [ параметры ] <группа> <список>`

Команда	Описание
	Параметры: `-R` — рекурсивное изменение владельца во всех подкаталогах каталогов, указанных в списке; `-v` — вывод на экран имени каждого обрабатываемого файла или каталога; `-c` — вывод на экран имени каждого файла или каталога, для которого изменяется группа-владелец; `<группа>` — имя или GID группы-владельца, которая должна быть установлена; `<список>` — список имен файлов или каталогов, для которых устанавливается новая группа-владелец. **Пример использования:** `chgrp –R –v users /home/vasya`

`chown`	Изменение пользователя-владельца для заданных файлов
	Формат вызова:
	`chown [ параметры ] <пользователь> <список>`
	Параметры:
	`-R` — рекурсивное изменение владельца во всех подкаталогах каталогов, указанных в списке;
	`-v` — вывод на экран имени каждого обрабатываемого файла или каталога;
	`-c` — вывод на экран имени каждого файла или каталога, для которого изменяется группа-владелец;
	`<пользователь>` — имя или UID пользователя-владельца, который должен быть установлен;
	`<список>` — список имен файлов или каталогов, для которых устанавливается новый пользователь-владелец.
	Пример использования:
	`chown –R –v vasya /home/vasya`

`cp`	Копирование файлов и каталогов **Формат вызова:** `cp [ параметры ] <файлы> <каталог>` — копирует файлы из списка в каталог `cp [ параметры ] <файл1> <файл2>` — делает копию файла `файл1` под именем `файл2` **Параметры:** `-r` — копировать каталоги рекурсивно; `-v` — выводить имя каждого файла перед его копированием.

Продолжение таблицы

Команда	Описание
	Пример использования: `cp –r file1.txt file2.txt /usr/doc/`

`cut`	Извлечение отдельных полей из форматированных строк файлов
	Формат вызова:
	`cut –f <номер поля> -d<разделитель>`
	Параметры:
	`-f` — задает номер извлекаемого поля;
	`-d` — задает разделитель форматированных строк.
	Пример использования:
	для файла `file.txt` с содержимым
	`Иванов:Иван:Иванович:1978`
	команда
	`cat file.txt | cut –f2 –d:`
	выведет `Иван`

`diff`	Поиск различий между двумя файлами и вывод их в стандартный поток вывода
	Формат вызова:
	`diff [ параметры ] <файл1> <файл2>`
	Параметры:
	`-b` — игнорировать различия в пробелах и символах табуляции;
	`-t` — заменять в выводе символы табуляции пробелами;
	`-u` — использовать унифицированный формат вывода;
	`-n` — вывод в формате RCS-diff.
	Пример использования:
	`diff –nur file1.txt file2.txt`

`echo (BASH)`	Вывод строк текста в стандартный поток вывода
	Формат вызова:
	`echo [ параметры ] <строка текста>`
	Параметры:
	`-n` — не выводить символ перевода строки после вывода строки.
	Пример использования:
	`echo "Hello world"`

`exec (BASH)`	Выполнение программы с заменой на ее процесс процесса текущего командного интерпретатора **Формат вызова:** `exec <имя программы>`

Продолжение таблицы

Команда	Описание
	Параметры: <имя программы> — полное, относительное или краткое путевое имя исполняемого файла. **Пример использования:** exec ls Будучи выполненной в первичном командном интерпретаторе, команда выводит список файлов в текущем каталоге, после чего сеанс завершается (поскольку она заменила собой первичный интерпретатор)
exit (BASH)	Завершение выполнения текущего задания с заданным кодом возврата **Формат вызова:** exit <n> **Параметры:** <n> — код возврата. Неотрицательное число. **Пример использования:** exit 1
export (BASH)	Перемещение переменных, объявленных в задании, во внешнюю среду этого задания **Формат вызова:** export <имя переменной> export <имя переменной>=<значение> **Параметры:** <имя переменной> — имя переменной, которая экспортируется в среду; <значение> — значение, которое может быть присвоено переменной непосредственно перед экспортом в среду. **Пример использования:** export IP_Address=192.168.0.1
grep	Поиск подстроки или регулярного выражения в файлах с последующим выводом найденных строк на экран **Формат вызова:** grep [параметры] <подстрока> <список файлов> **Параметры:** -c — выдает количество строк, содержащих подстроку; -i — игнорирует регистр символов при поиске; -n — выдает перед каждой строкой ее номер в файле; <подстрока> — строка символов или регулярное выражение для поиска. Подробнее о регулярных выражениях — см. книгу [12];

Продолжение таблицы

Команда	Описание

	<список файлов> — список имен файлов, в которых производится поиск. Команда возвращает 0, если подстрока найдена, и 1 — если не найдена.
	Пример использования:
	`grep "Операционные системы" book.txt`
	`if [ $? -ne 0 ] ; then`
	`  echo "Подстрока не найдена"`
	`fi`

gzip	Сжатие файлов с использованием алгоритма LZW. Сжатый файл получает то же имя, что и исходный, но имеет расширение .gz. Для сжатия большого количества файлов в один архив необходимо сначала склеить файлы при помощи команды tar
	Формат вызова:
	`gzip [ параметры ] <имя файла>`
	Параметры:
	<имя файла> — имя сжимаемого файла;
	-c — выводит сжатые данные в стандартный поток вывода, не изменяя исходный файл;
	-t — проверка целостности сжатого файла;
	-d — распаковка сжатого файла;
	-v — вывод имени сжимаемого файла и процент его сжатия.
	Пример использования:
	`gzip -d linux-kernel-2.4.34.tar.gz`

head	Вывод начальных строк файла
	Формат вызова:
	`head [ параметры] <имя файла>`
	Параметры:
	<имя файла> — имя файла, начальные строки которого выводятся;
	-n <число строк> — вывод заданного числа строк из начала файла;
	-c <число байт> — вывод заданного числа байт из начала файла.
	При указании отрицательного числа n строк или байт выводится весь текст, кроме последних n строк или байт.
	Пример использования:
	`head -n 15 file.txt`

Продолжение таблицы

Команда	Описание

`kill`	Посылка сигнала процессу с заданным PID
	Формат вызова:
	`kill [параметры] <PID>`
	Параметры:
	`-l` — выводит список доступных сигналов;
	`–<номер>` или `–<название>` — посылаемый сигнал;
	`<PID>` — PID процесса, которому посылается сигнал.
	Пример использования:
	`kill –SIGHUP 1035`

`killall`	Посылка сигнала всем процессам, созданным в результате запуска программы с известным именем
	Формат вызова:
	`killall -<сигнал> <имя программы>`
	Параметры:
	`–<сигнал>` — посылаемый сигнал, задаваемый номером или названием;
	`<имя программы>` — имя программы, породившей процессы. Может быть прочитана в таблице процессов, выводимой командой `ls`.
	Пример использования:
	`killall –SIGSTOP hasher`

`less`	Постраничный вывод текстового файла на экран с возможностью прокрутки и поиска
	Формат вызова:
	`less <имя файла>`
	Параметры:
	`<имя файла>` — имя выводимого файла.
	Пример использования:
	`less file.txt`

`ln`	Создание ссылок на файлы
	Формат вызова:
	`ln [параметры] <исходный файл> <файл ссылки>`
	Параметры:
	будучи запущенной без параметров, команда создает жесткую ссылку с именем `<файл ссылки>` на тот набор данных, на который указывает имя `<исходный файл>`;
	`-s` — вместо жесткой ссылки создается символическая ссылка.
	Пример использования:
	`ln –s file.txt linkfile.txt`

Продолжение таблицы

Команда	Описание

`ls`	Вывод списка файлов каталога
	Формат вызова:
	`ls [параметры] [список файлов и каталогов]`
	Параметры:
	`-a` — выводить файлы с именами, начинающимися с `.`;
	`-l` — выводить расширенную информацию об атрибутах файла;
	`[список файлов и каталогов]` — список файлов, которые будут выведены командой `ls` и каталогов, содержимое которых будет выведено командой `ls`.
	Пример использования:
	`ls –l /home/nick`

`mkdir`	Создание каталога
	Формат вызова:
	`mkdir [параметры] <имя каталога>`
	Параметры:
	`-p` — если в качестве `<имени каталога>` задано имя, включающее в себя несколько уровней иерархии (например, `/usr/local/share/doc/programs`), то при указании этого ключа будут создаваться все недостающие каталоги, т.е. если не существует каталога `/usr/local/share/doc`, то вначале будет создан он, а затем его подкаталог `programs`.
	Пример использования:
	`mkdir –p /usr/local/share/doc/programs`

`more`	Постраничный вывод текстового файла на экран с возможностью прокрутки
	Формат вызова:
	`more <имя файла>`
	Параметры:
	`<имя файла>` — имя выводимого файла.
	Пример использования:
	`more file.txt`

`mv`	Перемещение (переименование) файлов
	Формат вызова:
	`mv [параметры] <исходный файл> <файл назначения>`
	`mv [параметры] <файл> <каталог назначения>`
	Параметры:
	`-f` — не выдавать никаких запросов на подтверждение операции;

Продолжение таблицы

Команда	Описание

	`-i` — всегда выдавать запрос на подтверждение, если файл назначения существует.
	Первая форма вызова команды переименовывает файл, имя которого задано параметром `<исходный файл>`, присваивая ему имя, заданное параметром `<файл назначения>`. Вторая форма вызова перемещает файл в каталог, заданный параметром `<каталог назначения>`.
	Пример использования:
	`mv file1.txt file2.txt`
	`mv file1.txt dir2/`

`nice`	Запуск программы с измененным приоритетом для планировщика задач. Приоритет определяет частоту выделения процессу процессорного времени
	Формат вызова:
	`nice [ параметры ] <программа> [аргументы программы]`
	Параметры:
	`-n <смещение>` — изменение базового приоритета процесса на величину смещения. Смещение находится в пределах от –20 (наивысший приоритет) до 19 (низший приоритет);
	`<программа>` — имя исполняемого файла запускаемой программы;
	`[аргументы программы]` — параметры, передаваемые программе при ее запуске.
	Пример использования:
	`nice –n 15 alpha_gamma 1057`

`ps`	Вывод информации о запущенных процессах **Формат вызова:** `ps [ параметры ]` **Параметры:** `a` — вывод информации обо всех процессах всех пользователей; `x` — вывод процессов, не привязанных к терминалу (системных процессов и демонов); `l` — длинный формат вывода (максимум данных); `u` — формат вывода, ориентированный на пользователя (самые необходимые данные);

Продолжение таблицы

Команда	Описание
	s — вывод информации об обрабатываемых сигналах. **Пример использования:** ps aux
pwd	Вывод имени текущего каталога **Формат вызова:** pwd **Параметры:** отсутствуют. **Пример использования:** pwd (будет выведено, например, /home/sergey)
read (BASH)	Построчное считывание слов, разделенных пробелами, символами табуляции или переводами строки из стандартного потока ввода, с последующим присвоением значения слов переменным, заданным в параметрах **Формат вызова:** read <список переменных> **Параметры:** <список переменных> — список имен переменных, заданных через запятую. **Пример использования:** echo Иванов Иван Иванович | read name1 name2 name3 вызовет присвоение строк Иванов, Иван и Иванович переменным name1 name2 и name3 соответственно
rm	Удаление файлов или каталогов **Формат вызова:** rm [параметры] <список файлов или каталогов> **Параметры:** -f — не запрашивать подтверждение операции; -i — выводить запрос на подтверждение удаления каждого файла или каталога; -r — рекурсивно удалить все дерево каталогов, начиная с заданного; <список файлов или каталогов> — имена файлов или каталогов, подлежащих удалению. **Пример использования:** rm –rf /home/nick/junk_files/

Продолжение таблицы

Команда	Описание

`rmdir`	Удаление пустых каталогов (не содержащих файлов и подкаталогов)
	Формат вызова:
	`rmdir [параметры] <имя каталога>`
	Параметры:
	`-p` — если имя каталога включает в себя несколько уровней иерархии (например, `/cat1/cat2/cat3`), то при указании этого ключа команда будет удалять каталоги, начиная с нижнего уровня (т.е. в рассматриваемом случае будет аналогична `rmdir /cat/cat2/cat3 ; rmdir /cat1/cat2 ; rmdir /cat1`);
	`<имя каталога>` — имя удаляемого пустого каталога.
	Пример использования:
	`rmdir –p /cat1/cat2/cat3/`

`set (BASH)`	Вывод списка переменных окружения, определенных в среде, или присвоение значений позиционных параметров
	Формат вызова:
	`set [список значений]`
	Параметры:
	при отсутствии параметров выводит полный список определенных переменных и их значений;
	при задании списка значений, разделенных пробелом, последовательно присваивает эти значения встроенным переменным `$1 … ${n}`, где `n` — количество значений.
	Пример использования:
	`set 2 3 4 5`
	`echo $3` (будет выведено 4)

`shift (BASH)`	Сдвиг окна чтения позиционных параметров задания на заданное число позиций
	Формат вызова:
	`shift [n]`
	Параметры:
	`[n]` — число позиций, на которые происходит сдвиг. Если число не указано, то сдвиг осуществляется на одну позицию.
	Пример использования:
	задание запущено с параметрами `a b c d e f`
	`echo $1` (выведет a)
	`shift 4`
	`echo $1` (выведет e)

Продолжение таблицы

Команда	Описание

`sleep`	Приостановка выполнения задания на заданное число секунд
	Формат вызова:
	`sleep <n>`
	Параметры:
	`<n>` — число секунд, на которое приостанавливается выполнение задания.
	Пример использования:
	`sleep 10`

`sort`	Сортировка в алфавитном порядке строк, подаваемых со стандартного потока ввода
	Формат вызова:
	`sort [параметры]`
	Параметры:
	`-r` — сортировка в обратном алфавитном порядке;
	`-b` — игнорирование пробелов в начале строк.
	Пример использования:
	`cat file.txt | sort > sorted.txt`

`tail`	Вывод последних строк файла **Формат вызова:** `tail [ параметры] <имя файла>` **Параметры:** `<имя файла>` — имя файла, последние строки которого выводятся; `-n <число строк>` — вывод заданного числа строк из конца файла; `-c <число байт>` — вывод заданного числа байт из конца файла. При указании отрицательного числа `n` строк или байт выводится весь текст, кроме последних `n` строк или байт. `-f` — не завершать выполнение программы по достижению конца файла, а ждать добавления данных в файл и выводить их по мере поступления. Может быть удобно при выводе новых записей в файлах протоколов. Если при этом не указывать параметры `–n` или `–c`, то команда будет ждать добавления данных в файл, не выводя данные, которые были записаны в файл на момент ее вызова; `-F` — аналогично `–f`, но также отслеживается ситуация переименования файла во время работы команды `tail`.

Продолжение таблицы

Команда	Описание
	Пример использования: `tail –F logfile.txt`

`tar`	Склеивание файлов и каталогов в один файл для подготовки к архивированию
	Формат вызова:
	`tar <параметр><модификатор параметра> <имя архивного файла> <список файлов для архивации>`
	Параметры:
	`-c` — создается новый архив; запись начинается с начала архива, а не после последнего файла;
	`-u` — указанные файлы добавляются в архив, если их там еще нет или если они были изменены с момента последней записи в данный архив;
	`-x` — указанные файлы извлекаются из архива;
	`<имя архивного файла>` — имя файла, в который склеиваются файлы, подлежащие архивации;
	`<список файлов для архивации>` — список имен файлов и каталогов, подлежащих архивации. Каталоги обрабатываются рекурсивно.
	Модификаторы:
	`z` — сразу упаковывает файлы архиватором `gzip` или распаковывает их;
	`v` — вызывает выдачу имени каждого обрабатываемого файла;
	`f` — если указан этот модификатор и в качестве имени архивного файла указывается символ – (минус), то архив считывается или выдается на стандартный поток ввода или вывода.
	Пример использования:
	`cd fromdir; tar -cf - . | (cd todir; tar -xf -)`

`tee`	Конвейер с одним входом (стандартный ввод) и двумя выходами (стандартный вывод и указанный файл)
	Формат вызова:
	`tee [ параметры ] <выходной файл>`
	Параметры:
	`-a` — добавление текста в конец выходного файла;
	`<выходной файл>` — файл, в который производится запись.
	Пример использования:
	`cat infile.txt | tee –a outfile.txt`

Продолжение таблицы

Команда	Описание

`test`	Вычисление значения условного выражения. Более подробно — см. подразд. 3.3.6.
	Формат вызова:
	`test <выражение>`
	Параметры:
	`<выражение>` — проверяемое условное выражение.
	Пример использования:
	`test –f file.txt`

`touch`	Изменение даты модификации указанного файла на текущую или создание нового файла, если указанный файл не существует.
	Формат вызова:
	`touch <имя файла>`
	Параметры:
	`<имя файла>` — имя создаваемого файла или файла с изменяемой датой.
	Пример использования:
	`touch lock.txt`

`trap (BASH)`	Определение команды, которая будет выполнена при получении заданием сигнала с заданным номером.
	Формат вызова:
	`trap <команда> <список сигналов>`
	Параметры:
	`<команда>` — выполняемая команда;
	`<список сигналов>` — список номеров сигналов.
	Пример использования:
	`trap "exit 1" 3 4 7`

`unset (BASH)`	Деинициализация переменной. После выполнения этой команды переменная более не имеет определенного значения.
	Формат вызова:
	`unset <имя переменной>`
	Параметры:
	`<имя переменной>` — имя деинициализируемой переменной.
	Пример использования:
	`var="123" ; echo var ; unset var`
	`echo var` (возникнет ошибка)

Окончание таблицы

Команда	Описание

`wait (BASH)`	Ожидание завершения процесса с заданным PID и возврат его кода возврата.
	Формат вызова:
	`wait <PID>`
	Параметры:
	`<PID>` — PID процесса, завершения которого ожидает команда.
	Пример использования:
	`wait 1078`

`wc`	Вывод количества байт, слов или строк в файле; сначала выводится количество, затем через пробел — имя файла.
	Формат вызова:
	`wc [ параметры ] <имя файла>`
	Параметры:
	`-c` — выводится количество байт в файле;
	`-l` — выводится количество строк в файле;
	`-w` — выводится количество слов в файле.
	Пример использования:
	`wc –w file.txt | cut –f1 –d\<пробел>`

`which`	Вывод полного пути к программе с заданным именем, если этот каталог присутствует в переменной `$PATH`.
	Формат вызова:
	`which <имя программы>`
	Параметры:
	`<имя программы>` — имя исполняемого файла, для которого производится поиск каталога.
	Пример использования:
	`which ls (выведет /bin)`

СПИСОК ЛИТЕРАТУРЫ

1. *Гордеев А. В.* Системное программное обеспечение / А. В. Гордеев, А. Ю. Молчанов. — СПб.: Питер, 2003. — 736 с.

2. Графический стандарт X Window. — М.: ИМВС РАН, 2000. — 316 с.

3. *Дунаев С.* UNIX-сервер / С. Дунаев. В 2-х т. — Т. 1. Общее руководство по системе. — М.: «ДИАЛОГ-МИФИ», 1998. — 304 с.

4. *Краковяк С.* Основы организации и функционирования ОС ЭВМ / С. Краковяк; пер. с франц. — М.: МИР, 1988. — 480 с.

5. *Немет Э.* UNIX. Руководство системного администратора / Э. Немет, Г. Снайдер, Т. Хейн; пер. с англ. — СПб-К.: Питер, BHV, 2002. — 928 с.

6. *Орлов В. Н.* Мобильная операционная система МОС ЕС / В. Н. Орлов, В. Ю. Блажнов, О. А. Барвин. — М.: Финансы и статистика, 1990. — 208 с.

7. *Померанц О.* Ядро Linux. Программирование модулей / О. Померанц; пер. с англ. — М.: КУДИЦ-Образ, 2000. — 112 с.

8. *Робачевский А. М.* Операционная система UNIX / А. М. Робачевский. — СПб.: BHV-Санкт-Петербург, 1999. — 528 с.

9. *Стивенс У.* UNIX. Взаимодействие процессов / У. Стивенс; пер. с англ. — СПб.: Питер, 2002. — 576 с.

10. *Стивенс У.* UNIX. Разработка сетевых приложений / У. Стивенс; пер. с англ. — СПб.: Питер, 2003. — 1 088 с.

11. *Таненбаум Э.* Современные операционные системы / Э. Таненбаум; пер. с англ. — СПб.: Питер, 2002. — 1 040 с.

12. *Фридл Дж.* Регулярные выражения. Библиотека программиста / Дж. Фридл; пер. с англ. — СПб.: Питер, 2001. — 352 с.

13. *Хендриксен Д.* Интеграция UNIX и Windows NT / Д. Хендриксен; пер. с англ. — М.: Диа-Софт, 1999. — 352 с.

14. *Цилюрик О.* QNX/UNIX. Анатомия параллелизма / О. Цилюрик, Е. Горошко. — М.: Символ-Плюс, 2006. — 288 с.

15. *Чан Т.* Системное программирование на C++ для UNIX / Т. Чан; пер. с англ. — Киев.: BHV, 1997. — 592 с.

ОГЛАВЛЕНИЕ

Учебное издание

Синицын Сергей Владимирович,
Батаев Алексей Владимирович,
Налютин Никита Юрьевич

Операционные системы

Учебник

3-е издание, стереотипное

Редактор *Н. Е. Овчеренко*
Технический редактор *Н. И. Горбачева*
Компьютерная верстка: *В. А. Крыжко*
Корректор *А. П. Сизова*

Изд. № 103116479. Подписано в печать 01.08.2013. Формат 60×90/16
Гарнитура «Ньютон». Бумага офс. № 1. Печать офсетная. Усл. печ. л. 19,0.
Тираж 1 000 экз. Заказ №

ООО «Издательский центр «Академия». www.academia-moscow.ru
129085, Москва, пр-т Мира, 101В, стр. 1.
Тел./факс: (495) 648-0507, 616-00-29.
Санитарно-эпидемиологическое заключение № РОСС RU. АЕ51. Н 16476 от 05.04.2013.

Отпечатано с электронных носителей издательства.
ОАО «Тверской полиграфический комбинат», 170024, г. Тверь, пр-т Ленина, 5.
Телефон: (4822) 44-52-03, 44-50-34. Телефон/факс: (4822) 44-42-15.
Home page — www.tverpk.ru Электронная почта (E-mail) — sales@tverpk.ru www.sarpk.ru