Профессиональное образование

ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА

А. В. Батаев, Н. Ю. Налютин,
С. В. Синицын

ОПЕРАЦИОННЫЕ СИСТЕМЫ И СРЕДЫ

Учебник

ACADEMIA

ПРОФЕССИОНАЛЬНОЕ ОБРАЗОВАНИЕ

А.В.БАТАЕВ, Н.Ю.НАЛЮТИН,
С.В.СИНИЦЫН

ОПЕРАЦИОННЫЕ СИСТЕМЫ И СРЕДЫ

Учебник

Рекомендовано
Федеральным государственным автономным учреждением
«Федеральный институт развития образования» (ФГАУ «ФИРО»)
в качестве учебника для использования в учебном процессе
образовательных учреждений, реализующих программы
среднего профессионального образования
по укрупненной группе специальностей
«Информатика и вычислительная техника»

Регистрационный номер рецензии 559
от 20 декабря 2013 г. ФГАУ «ФИРО»

2-е издание, стереотипное

ACADEMIA

Москва
Издательский центр «Академия»
2015

УДК 681.3.066(075.32)
ББК 32.973-018.2я723
Б28

Рецензенты:
декан факультета кибернетики Московского государственного технического университета радиотехники, электроники и автоматики, д-р техн. наук, проф. *М. П. Романов*;
профессор кафедры кибернетики Московского института электроники и математики НИУ ВШЭ, д-р техн. наук *Б. И. Прокопов*

Батаев А. В.

Б28 Операционные системы и среды : учебник для студ. учреждений сред. проф. образования / А. В. Батаев, Н. Ю. Налютин, С. В. Синицын. — 2-е изд., стер. — М. : Издательский центр «Академия», 2015. — 272 с.

ISBN 978-5-4468-2474-8

Учебник создан в соответствии с Федеральным государственным образовательным стандартом среднего профессионального образования по специальностям «Компьютерные сети», ОП.04 «Операционные системы», «Компьютерные системы и комплексы», ОП.07 «Операционные системы и среды», «Программирование в компьютерных системах», ОП.01 «Операционные системы», «Информационные системы (по отраслям)», ОП.02 «Операционные системы», «Прикладная информатика (по отраслям)», ОП.07 «Операционные системы и среды».

Изложены основные сведения о базовых объектах, находящихся под управлением ОС, — файлах, пользователях и задачах. Рассмотрены задания операционной системы, определяющие логическую последовательность выполнения задач пользователя. Особое внимание уделяется обеспечению работы множества пользователей в ОС UNIX и WINDOWS — рассмотрены вопросы идентификации пользователей, размещения их личных данных, управление доступом пользователей к файлам и каталогам, определены языковые средства BASH для работы с правами доступа. Описаны методы управления учетными записями пользователей, а также методика перенаправления сеансов пользователей при помощи файлов инициализации сеанса в системах UNIX. Дан краткий обзор методов построения прикладных программ на языке C в UNIX-подобных операционных системах и операционных системах WINDOWS.

Для студентов учреждений среднего профессионального образования.

УДК 681.3.066(075.32)
ББК 32.973-018.2я723

ISBN 978-5-4468-2474-8

Уважаемый читатель!

Данный учебник является частью учебно-методического комплекта для всех специальностей укрупненной группы «Информатика и вычислительная техника».

Учебник предназначен для изучения общепрофессиональной дисциплины ОП.04 для специальности «Компьютерные сети», ОП.07 для специальности «Компьютерные системы и комплексы», ОП.01 для специальности «Программирование в компьютерных системах», ОП.02 для специальности «Информационные системы (по отраслям)», ОП.07 для специальности «Прикладная информатика (по отраслям)».

Учебно-методические комплекты нового поколения включают в себя традиционные и инновационные учебные материалы, позволяющие обеспечить изучение общеобразовательных и общепрофессиональных дисциплин и профессиональных модулей. Каждый комплект содержит учебники и учебные пособия, средства обучения и контроля, необходимые для освоения общих и профессиональных компетенций, в том числе и с учетом требований работодателя.

Учебные издания дополняются электронными образовательными ресурсами. Электронные ресурсы содержат теоретические и практические модули с интерактивными упражнениями и тренажерами, мультимедийные объекты, ссылки на дополнительные материалы и ресурсы в Интернете. В них включены терминологический словарь и электронный журнал, в котором фиксируются основные параметры учебного процесса: время работы, результат выполнения контрольных и практических заданий. Электронные ресурсы легко встраиваются в учебный процесс и могут быть адаптированы к различным учебным программам.

Введение

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

Каждый пользователь ОС применяет 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, предоставляемом ОС, или пользуется программами, написанными другими пользователями — программистами.

Синтаксис и семантика таких языков различны в зависимости от решаемых с их помощью задач. Сами задачи могут быть условно разделены на следующие группы:

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

Пользователи ОС, применяющие тот или иной язык общения с операционной системой, могут быть, в свою очередь, классифицированы следующим образом (рис. В.1):

- системные программисты;
- системные администраторы;
- прикладные программисты;
- прикладные пользователи.

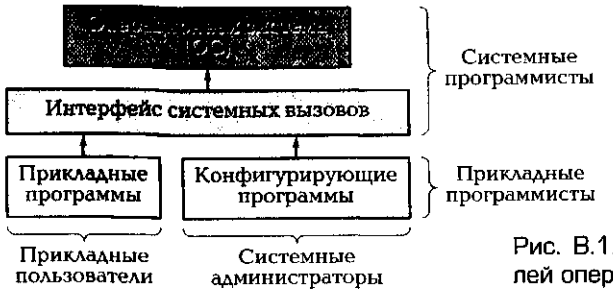

Рис. В.1. Группы пользователей операционной системы

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 «язык общения» — «решаемая задача» будет определять роль пользователя при его работе с операционной системой.

Табл. В.1 содержит основные типы ролей пользователей операционной системы.

Таблица В.1. Роли пользователей

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

Данный учебник в первую очередь ориентирован на студентов средних учебных заведений, прикладных программистов и пользователей операционной системы, но в некоторой мере затрагивает также вопросы системного программирования и администрирования в среде ОС.

Бóльшая часть материала, излагаемого в учебнике, дается на примере общих механизмов, существующих в операционных системах семейства UNIX и Windows. Часть материала применима и для операционных систем семейства DOS (в частности, раздел, посвященный заданиям в Windows).

В качестве основной учебной операционной системы рассматривается свободно распространяемая ОС Linux и операционная система Windows 7, однако бóльшая часть материала книги применима для всех основных версий UNIX-систем и Windows семейства NT.

Учебник состоит из трех частей и приложений:

- терминологическое введение содержит определения основных понятий и принципов работы операционных систем, на которые опирается дальнейшее изложение;
- главы 1 — 6 дают читателю представление об основных этапах истории развития, объектах, поддерживаемых операционной системой: памяти, файлах, пользователях, заданиях и методах работы с ними;
- главы 7 — 10 углубляют знания читателя о файлах и пользователях, затрагивают вопросы администрирования ОС. 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 по структуре каталогов системы «Контроль знаний» и по основным командам операционной системы UNIX.

Каждая глава учебника завершается контрольными вопросами для самопроверки.

В силу ограниченности объема книги в ней не рассмотрены следующие вопросы:

- генерация ОС (конфигурирование ядра, администрирование) [3];
- графические интерфейсы (X Window System) [2];
- часто используемые прикладные пакеты программ;
- специфика организации гетерогенных систем и сред [17];
- разработка дополнительных компонент ядра, драйверов устройств [16];

- специфика программирования в операционных системах реального времени [9];
- микроядерные архитектуры [15];
- разработка сетевых приложений на основе различных протоколов связи [10].

Все они предлагаются заинтересованному читателю для самостоятельного изучения.

Глава 1

ТЕРМИНОЛОГИЧЕСКОЕ ВВЕДЕНИЕ

1.1. ОСНОВНЫЕ ПОНЯТИЯ

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

Операционная система предоставляет пользователю базовый набор инструментов и среду для хранения данных, а также средства управления последовательностью использования инструментов. Интервал времени, в течение которого пользователь решает одну или несколько последовательных задач, пользуясь при этом средствами, предоставляемыми ОС, называется *сеансом*.

В начале любого сеанса пользователь идентифицирует себя, в конце указывает, что сеанс закончен. Описание последовательности использования программных инструментов, записанное на некотором формальном языке, называется *заданием*, а сам язык — *языком управления заданиями*.

Выполнение заданий в большинстве операционных систем производится командным интерпретатором, более подробное определение которого будет дано далее.

Обычно пользователю предоставляется некоторый *интерфейс* общения с командным интерпретатором, при использовании которого команды 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 более употребительным является графический интерфейс пользователя (GUI), рассмотрение которого выходит за рамки данного учебника [2].

Программа (в общем случае) — набор инструкций процессора, хранящийся на диске (или другом накопителе информации). Для того чтобы программа могла быть запущена на выполнение, операционная система должна создать среду выполнения — *информационное окружение* решаемой задачи. После этого операционная система перемещает исполняемый код и данные программы в оперативную память и инициирует выполнение программы.

Операционная система выполняет функции управления аппаратными ресурсами, их распределения между выполняемыми программами пользователя и формирует среду исполнения, которая содержит все данные, необходимые для программы. Такая среда в дальнейшем и будет называться информационным окружением. В информационное окружение входят данные и объекты, обрабатываемые операционной системой, которые влияют на выполнение программы, т. е. на решение задачи пользователя. В ходе дальнейшего изложения будут приведены примеры информационного окружения различного характера.

Используя понятия программы, данных и информационного окружения, можно определить понятие *задачи* в среде ОС как совокупность программы и данных, являющихся частью информационного окружения.

Выполняемая программа образует процесс. *Процесс* представляет собой совокупность информационного окружения и области памяти, содержащей исполняемый код и данные программы. Обычно в памяти, контролируемой операционной системой, может одновременно работать большое число процессов.

Вполне естественно, что на однопроцессорных компьютерах возможно одновременное выполнение программного кода только одного процесса, поэтому часть процессов находятся в режиме ожидания, а один из процессов — в режиме выполнения. Процессы при этом образуют очередь, операционная система 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 своей работы процессы используют вычислительную мощность процессора, оперативную память, обращаются к внешним файлам, внутренним данным ядра операционной системы. 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.). Операционная система в данной ситуации либо управляет ограничениями 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. Примером такого ресурса может служить максимальное количество файлов, одновременно открытых пользователем.

В случае, если операционная система позволяет одновременно использовать ресурсы нескольким процессам, ее ресурсы можно подразделить на типы, указанные на рис. 1.1 [12].

Неразделяемые ресурсы могут быть использованы на заданном отрезке времени только одним процессом, при этом другие процессы не имеют доступа к такому 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.

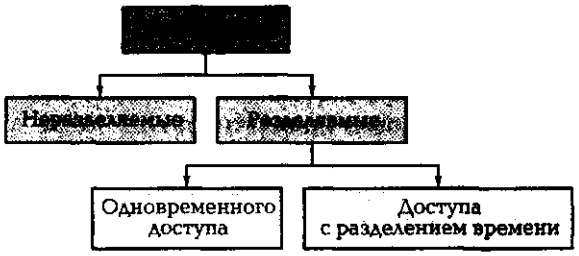

Рис. 1.1. Типы ресурсов, доступных в операционной системе

В этом случае используется квантование моментов использования ресурса по времени.

В каждый квант времени один процесс получает полные и исключительные права на владение данным ресурсом. При этом значение такого кванта много меньше полного времени, в течение которого ресурс используется этим процессом, т. е. интервала 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 одного 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 последовательными квантами, при этом некоторые процессы имеют более высокий приоритет и получают доступ к ресурсу чаще.

1.1.1. Типовая структура операционной системы

Обычно в составе операционной системы 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 операционной системы входят следующие компоненты: система управления сеансами пользователей, файловая система, система управления задачами (процессами), 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 виртуальным устройствам, корректно завершает сеанс при окончании работы пользователя с системой.

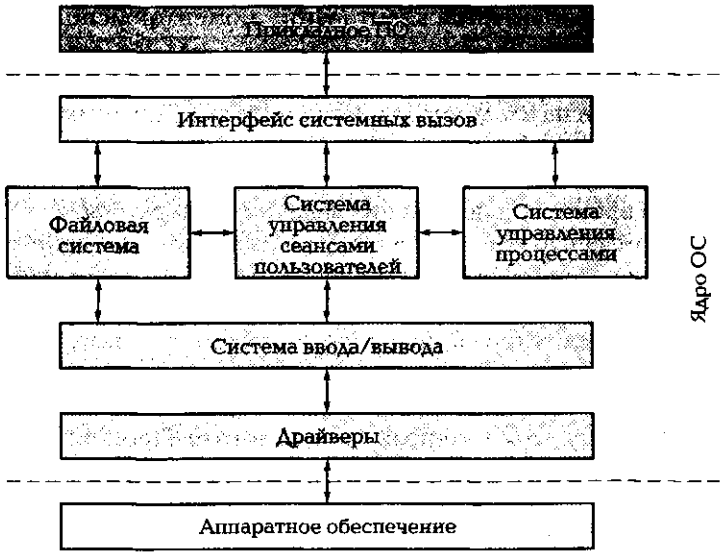

Рис. 1.2. Структура ядра типичной операционной системы

Файловая система выполняет преобразование данных, хранимых на внешних запоминающих устройствах (например, на дисковых накопителях или на flash-накопителях), в логические объекты — файлы и каталоги. Также файловая система 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 управления процессами распределяет ресурсы между выполняемыми задачами (процессами), обеспечивает защиту памяти процессов от модификации ее другими процессами, реализует механизмы межпроцессного взаимодействия.

Система ввода/вывода обрабатывает запросы всех рассмотренных выше компонент ядра и преобразует 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, записывае-

мая в данное устройство, пропадает, т. е. оно может быть использовано для поглощения данных, несущественных для решаемой задачи.

Драйверы устройств — это системные программы, которые 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.

1.1.2. Классификация операционных систем

Сложность составных частей ядра и реализуемые ими функции в первую очередь зависят от числа одновременно обслуживаемых ОС пользователей и от числа одновременно выполняемых процессов. В связи с этим разумно провести классификацию ОС по этим двум параметрам и рассмотреть особенности компонент ядра в каждом из типов ОС.

По количеству одновременно обслуживаемых пользователей операционные системы разделяются на однопользовательские (одновременно поддерживается не более одного сеанса пользователя) и многопользовательские (одновременно поддерживается множество сеансов пользователей). 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 количеству одновременно выполняемых процессов операционные системы делятся 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. Данный тип ОС обеспечивает единовременную поддержку только одного сеанса работы пользователя. Новый сеанс работы пользователя может быть начат только

после завершения предыдущего сеанса. При этом в новом сеансе пользователя сохраняется то же самое информационное окружение.

С точки зрения однопользовательской ОС пользователи неразличимы, поэтому если с такой операционной системой начинают работать несколько пользователей, то каждому из них она предоставляет доступ ко всем ресурсам и, возможно, к тому же самому информационному окружению. При этом пользователь может работать и со своими уникальными данными, например с данными на съемных дисках. При такой работе информационное окружение сеансов работы пользователей в системе становится различным.

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 завершением работы ОС 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 включает в себя сложные механизмы разграничения доступа к файлам и каталогам, хотя в файловой системе могут существовать флаги, задающие режимы работы с файлами и каталогами, их атрибуты.

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

Многопользовательские ОС. Данный тип ОС обеспечивает одновременную работу большого количества пользователей, что в значительной мере расширяет набор функций, реализуемых файловой системой и системой поддержки сеансов. 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. Она обеспечивает связывание каждого сеанса с реальным или виртуальным терминалом, содержит средства инициализации начального информационного окружения сеанса и обеспечивает 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 сопровождаются информационным блоком, определяющим права доступа к ним 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. Однако в случае появления необходимости в совместном доступе к одной и той же информации в файловой системе может быть определен доступ на чтение и запись к одному набору данных 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 обеспечивать одновременное выполнение большого количества программ разных пользователей.

Однозадачные ОС. Данный класс операционных систем предназначен 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 завершении программы освобождается память процесса, восстанавливается информационное окружение оболочки и управление передается операционной системой ей обратно.

Запуск программ в таких ОС последовательный. В случае, если одной из программ требуется вызвать на выполнение другую программу, точно так же сохраняется окружение вызывающей про-

граммы и по завершении 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 могут быть ОС с пакетной обработкой. В таких ОС пользователи формируют очередь заданий на выполнение программ, при этом задания могут принадлежать разным пользователям. Система последовательно выполняет программы пользователей, при этом 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 также усложняется, поскольку 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.

ОС семейства UNIX относятся к многопользовательским многозадачным операционным системам. Именно поэтому этот класс операционных систем был взят за основу данного учебника.

Поскольку, как уже говорилось, учебник ориентирован на прикладных программистов и пользователей, операционная система

в нем рассматривается как среда разработки и эксплуатации прикладного программного обеспечения.

В учебнике рассматриваются только базовые средства ОС UNIX, при этом не уделяется никакого внимания различным расширениям, например графическим средствам X Window System.

1.2. УНИВЕРСАЛЬНЫЕ И СПЕЦИАЛИЗИРОВАННЫЕ ОПЕРАЦИОННЫЕ СИСТЕМЫ. ОПЕРАЦИОННЫЕ СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ

Помимо классификаций на основе количества пользователей в системе и на основе количества одновременно выполняющихся процессов можно ввести еще один вид классификации операционных систем: операционные системы общего назначения и операционные системы специального назначения.

К классу *операционных систем общего назначения* относят операционные системы, которые могут являться как однопроцессными, так и многопроцессными, как многопользовательскими, так и однопользовательскими. Это операционные системы, которые работают в составе обычных настольных систем. Их основное предназначение состоит в том, чтобы предоставить пользователю системы удобный и понятный механизм управления аппаратными средствами вычислительной системы, обрабатывать запросы пользователя, максимально изолируя его от низкоуровневых операций и интерфейсов. Такие операционные системы ориентируются в первую очередь на простоту применения, поскольку их основными пользователями обычно являются не программисты, а пользователи средней или низкой квалификации. Зачастую такие пользователи могут даже не представлять, что за красивой картинкой анимированных обоев и разнообразием всевозможных красот графических интерфейсов скрывается мощнейший программный комплекс, управляющий всем аппаратным комплексом компьютера. К подобным операционным системам относятся настольные версии операционных систем семейства Windows, Linux, Apple iOS и т.д. Иными словами, это те операционные системы, с которыми пользователь может иметь дело как на работе для выполнения каких-то необходимых задач, так и дома для развлечения.

В противоположность операционным системам общего назначения *ОС специального назначения* относительно редко использу-

ются в повседневной жизни. Основные пользователи таких операционных систем — квалифицированные разработчики. Подобные операционные системы предназначены для управления ресурсами специальных вычислительных систем. Зачастую такие системы являются встраиваемыми, т. е. системами, которые должны работать, будучи встроенными непосредственно в устройство, которым они управляют. К ним можно отнести такие операционные системы, как Android, iOS, Windows CE и т. д.

Одним из подмножеств операционных систем специального назначения являются *операционные системы реального времени*. Многие встраиваемые системы требуют, чтобы операционная система, работающая в составе такого программно-аппаратного комплекса, реагировала на внешние события и входные данные в течение некоторого, зачастую очень малого, промежутка времени. Иначе говоря, операционные системы реального времени — это системы, которые способны обеспечить требуемый уровень сервиса в определенный промежуток времени.

Операционные системы реального времени можно разделить на два класса: системы жесткого реального времени и системы мягкого реального времени. Операционные системы, которые способны выполнять действия, не превышая требуемое время выполнения, относят к операционным системам жесткого реального времени. Если же операционная система способна лишь в среднем обеспечивать требуемое время выполнения, без строгого соблюдения верхнего временного предела, то такую систему относят к классу операционных систем мягкого реального времени. Но при этом для обоих классов ОС реального времени критическим требованием является детерминизм такой системы, т. е. предсказуемость ее поведения.

Системы реального времени требуются там, где задержка реакции системы может приводить к аварийным ситуациям, грозящим потерями материальных средств, катастрофами, гибелью людей и т. д. В таких системах часто обходятся вообще без операционных систем.

Использование операционных систем реального времени позволяет сократить сроки разработки управляющего ПО и повысить предсказуемость его поведения в следующих случаях:

- если разработанное управляющее ПО получается достаточно большим по объему;
- если в процессе его выполнения требуется нескольких вычислительных потоков;
- если решаемая задача содержит сложные требования по синхронизации доступа к ресурсам и т. д.

Для управления процессами в таких операционных системах используется один из двух подходов:

1) управление на основе приоритетов событий. При использовании данной стратегии управление передается тому процессу, который связан с обработкой наиболее приоритетного события;

2) управление на основе разделения времени. В этом случае переключение процессов осуществляется на основе регулярных прерываний по заданным интервалам времени и в случае наступления событий.

Во многих таких системах состав процессов является неизменным. Они запускаются при старте операционной системы и существуют до момента завершения ее работы. Неиспользуемые процессы могут переходить в пассивное состояние, когда они не нужны, и выходить из него, если происходит, например, событие, обработка которого требует активности соответствующего процесса. Такая система запуска и завершения процессов также вызвана требованием предсказуемости поведения операционной системы и параметров по времени реакции. Поэтому во многих таких системах запуск новых процессов просто не допускается, все они должны быть предопределены заранее.

В настоящее время существует достаточно большое количество операционных систем реального времени: LynxOS, RTLinux, VxWorks и т. д. Одной из наиболее известных операционных систем реального времени является QNX, которая иногда используется и в качестве настольной. Помимо этих коммерчески распространяемых продуктов существует еще и некоторое количество операционных систем, разработанных самостоятельно компаниями, но не распространяемых на коммерческой основе, а используемых только в составе уже готовых программно-аппаратных комплексов.

1.3. ФУНКЦИИ ОПЕРАЦИОННЫХ СИСТЕМ И ЭТАПЫ ИХ РАЗВИТИЯ

Исторически функции операционной системы развивались совместно с развитием самой вычислительной техники. В свое время было принято выделять поколения вычислительных машин, опираясь на элементную базу их реализации: электронные лампы, полупроводниковые транзисторы, микросхемы и т. д. Однако не сами эти свойства влияли на появление новых системных программных средств.

Основное влияние на развитие системного программного обеспечения общего назначения оказали четыре фактора: унификация архитектуры вычислительных машин, объем оперативной памяти, быстродействие процессора и состав периферийных устройств. Так, для первых машин, в работе с которыми участвовали в основном их конструкторы, ни о какой унификации не было и речи. В каждом коллективе проводились эксперименты как с составом команд, так и с величиной разрядной сетки вычислений. На время решения задачи программист, он же и оператор, становился полновластным хозяином вычислительной установки. Он сам загружал программу, сам вводил данные и оценивал результаты.

Библиотеки стандартных программ. После появления серийно выпускаемых машин стала возможна стандартизация в подходах к разработке программного обеспечения и произошло выделение программистов в отдельный класс пользователей. Естественным средством обобщения и накопления опыта программирования в этот период стали библиотеки стандартных программ, реализующих типовые функции: тригонометрические (sin, tg, arcsin), математические (log, exp, sqrt), ввод данных с внешних носителей, вывод, преобразования из одной системы счисления в другую и т. п. Типичным примером подобной библиотеки могла служить библиотека стандартных программ ИС-2 (интерпретирующая система) для машин типа М-20. При этом следует заметить, что М-20 была серийно выпускаемой машиной с трехадресной системой команд; подобной ей системой команд обладали машины БЭСМ-3, БЭСМ-4, а позднее и М-220.

Комплекс программ ИС-2 программист загружал в оперативную память до загрузки основной функциональной программы. Использовался стандартный интерфейс, позволяющий исполняемой программе обратиться к одной из функций системной библиотеки. Для этого нужно было указать код нужной функции и область памяти для взаимодействия, например область памяти для загрузки данных.

Позднее бóльшая часть вычислительных функций перекочевала в библиотеки языков программирования и даже стандартные процедуры перевода данных из одной формы представления в другую, а вот типовые действия, связанные с вводом и выводом, стали неотъемлемой частью операционных систем. В настоящее время в программистской среде для подобных функций операционной среды используют термин API (Application Programming Interface), подчеркивая тем самым прикладной характер подобных системных средств.

По мере того как архитектура вычислительных машин типизировалась и их быстродействие росло, все острее становилась проблема потерь времени при подготовке программы к выполнению: загрузке в оперативную память, установке носителей с данными, т.е. инициализации информационного окружения задачи. Действительно, когда процессор может выполнять десятки тысяч операций в секунду, затраты полутора-двух минут на установку колоды перфокарт с программой и ввод ее приводят к потере возможности выполнения 300 000 — 500 000 команд.

Пакетные мониторы. Решение нашлось в создании системных программ — мониторов, которые позволяли быстро загружать заранее подготовленные на магнитных носителях (в то время это чаще всего были магнитные ленты) пакеты программ и данных. Для внешнего обслуживания выполнения пакетов появилась особая группа специалистов — операторы. Они загружали на магнитные носители пакеты, подготовленные программистами. Потом запускали мониторную программу, которая сама последовательно считывала очередную задачу в оперативную память, создавала необходимое информационное окружение и запускала программу на выполнение.

Для ускорения выполнения программ результаты не печатались сразу, а выводились на накопительное устройство — часто такую же магнитную ленту. И лишь потом, когда все расчеты были закончены, полученные данные постепенно распечатывались на бумагу. Это позволяло сократить простои процессора, скорость работы которого в тысячи раз превышала скорость печати.

Этот период развития системных средств характеризовался включением в состав операционной системы программы системного ввода для подготовки пакетов, программы системного монитора (иногда ее называли диспетчером, управляющей программой или программой управления сеансами пользователей) и программы системного вывода для разгрузки полученных при расчетах пакета данных на выходные устройства, например на печать.

Что очень важно, вместе с пакетным монитором появился язык, на котором программисты могли оформлять свои задания. Сначала он назывался языком управления заданиями. Позднее стали чаще использовать термин «язык команд операционной системы», потому что его составной частью были команды, выполняемые командным интерпретатором ОС.

По мере роста емкости оперативного запоминающего устройства у операторов появилась возможность постоянно держать в памяти вычислительной установки все эти три программных средства

(ввода, мониторинга и вывода). В результате во все время работы можно было вводить новые задания и по мере готовности распечатывать результаты. Вод и вывод делались в низкоприоритетном режиме, а основное время процессор использовался для последовательного решения задач пакета заданий.

Наличие языка управления заданиями позволило программисту не присутствовать при решении его задачи. Ему достаточно было описать различные ситуации, которые могли бы возникнуть при счете, и предусмотреть соответствующую реакцию операционной системы. Остальное обслуживание вычислений ложилось на плечи оператора. К этому же времени следует отнести явное отделение программистов, подготавливающих программы расчетов на различных языках программирования, от пользователей, которые с помощью этих программ решали прикладные задачи, соединяя в пакетах программы и данные с помощью языка управления заданиями.

Мультипрограммирование, многозадачность. Дальнейший рост скорости работы процессора и объема оперативной памяти привел к тому, что потенциально на машине могли одновременно выполняться программы нескольких пользователей. Это вызвало необходимость решения двух проблем — обеспечения независимости параллельно выполняемых программ и защиты данных различных пользователей, программы которых одновременно находятся в работе.

Для решения первой задачи стали развивать функции диспетчера, который теперь должен был не только планировать работу по смене программ в пакете, но и принимать решение о том, какой из активных программ можно сейчас передать ресурсы центрального процессора. Одновременно на него легла задача распределения памяти между параллельно выполняемыми программами.

В свою очередь, аппаратные средства стали следить за тем, чтобы одна задача не могла помешать работе другой независимой задачи. Это обеспечивалось за счет различных средств защиты памяти, которые приводили к немедленному останову (прерыванию) программы, если она обращалась к адресному пространству чужой задачи.

Как мы видели, уже при пакетной обработке появилось мультипрограммирование — параллельно могли выполняться программы системного ввода, вывода и монитора пакета пользователя. Однако системные программы могли «договориться» друг с другом и не мешать взаимной работе. Теперь же на вычислительной установке параллельно работали программы различных пользователей, которые

либо умышленно, либо случайно могли испортить данные в информационном окружении других пользователей.

Управление данными. Необходимо было придумать такие правила работы с данными на внешних носителях, которые предотвращали бы несанкционированный доступ к ним программ посторонних пользователей. Для этой цели записи данных стали организовывать на носителях в наборы данных (Data set) и снабжать дополнительной информацией, которая позволяла выяснить, когда, кем был создан это набор, и определяла правила доступа к нему.

Было очевидно, что программа пользователя, имеющая непосредственный доступ к устройству (носителю информации), могла бы нарушить соглашения об организации наборов данных. Поэтому все операции по обмену данными между внешними устройствами и программой пользователя стали теперь принадлежностью операционной системы, расширив тем самым состав системных библиотек.

Если программе надо было прочитать данные с внешнего устройства, она обращалась к соответствующему API операционной системы и получала в свою область памяти очередной блок данных. Такая организация обмена создала еще одно полезное свойство — независимость программ от физической организации данных. Действительно, раз с устройством работала только операционная система, программа пользователя становилась в большой степени независимой от специфики носителей данных.

Кроме того, операционная система брала на себя функцию проверки правомочности доступа. Пользователь — владелец набора данных — должен был только указать, кому он разрешает работу со своим набором и в каком режиме (только читать, читать и писать, удалять и т. п.). В ряде случаев операционная система обеспечивала даже шифрование данных по заданному пользователем ключу.

Поддержание определенной организации данных (файловой системы) на носителях потребовало включения в состав операционной системы еще и специальных программ (утилит), которые выполняли предварительную подготовку магнитных носителей. При этом на носитель записывалась специальная разметка, обеспечивающая в дальнейшем размещение на нем данных пользователей.

Системы разделения времени. На следующем этапе роста объемов памяти и производительности стало понятно, что крупная вычислительная установка может обслуживать не только нескольких пользователей одной организации, но и целый регион. Конечно, при этом возникает проблема обеспечения удаленного доступа пользователей к системе. Задачу доступности удалось решить за счет

использования относительно дешевых терминальных устройств на основе телетайпов.

Терминал (клавиатура и печатающее устройство) подключался к вычислительной установке по обычной телефонной паре (проводу с двумя проводниками). Операционная система разделения времени успевала поддерживать одновременное обслуживание (ввод/вывод) нескольких сотен терминалов, поскольку скорость ввода и вывода на телетайпе не превышает нескольких десятков символов в секунду. У пользователя появлялась иллюзия, что он имеет непосредственный доступ к собственной вычислительной машине. Он перестал нуждаться в посредничестве оператора.

Конечно, разделение времени процессора использовалось уже на этапе обеспечения мультипрограммирования. Но появление большого числа пользователей, подключенных через относительно медленные устройства, привело еще и к возможности разделения времени при использовании памяти машины. Пока пользователь читал сообщение на терминале и вводил ответ, его программа простаивала. Поэтому операционная система вытесняла такую программу, находящуюся в режиме ожидания, во внешнюю память (на магнитный диск или магнитный барабан). Когда от пользователя поступали данные для дальнейшей работы, программа вновь восстанавливалась в оперативной памяти. Такой режим работы получил название свопинга (от англ. *swop* — менять местами).

Появлению систем разделения времени способствовало то, что операционная система уже взяла на себя функцию обмена с внешними устройствами и существовала достаточно дешевая в эксплуатации развитая телефонная сеть, позволявшая подключить пользовательский терминал практически в любой квартире. Проблема персонификации данных разных пользователей уже была решена на предшествующем этапе.

Надежность механических телетайпов оставляла желать лучшего, поэтому достаточно быстро появились терминалы на основе электронно-лучевой трубки, обеспечивающие как большую скорость вывода, так и большую надежность. Обычно их дополняли компактные печатающие устройства (принтеры) для получения так называемой твердой копии выведенных данных.

Подобная конфигурация терминалов широко использовалась в Москве в 1980 г. для информационного обслуживания Олимпиады-80 и в 1985 г. при проведении Фестиваля молодежи и студентов. Весьма оригинально были построены учебные терминалы фирмы CDC, в которых вместо электронно-лучевой трубки была установлена плазменная панель. Это обеспечивало очень высокую стабиль-

ность изображения символов, а кроме того, плоскость экрана давала возможность проецировать на него картинки микрофильмов, дополняя таким образом выводимую по линии связи текстовую информацию.

Очень скоро стало понятно, что терминалу можно поручить и предварительную обработку данных: редактирование символьных строк, хранение небольших объемов данных, управление печатью. Такой терминал значительно снимал нагрузку как с линии связи, так и с центральной вычислительной установки. Стали говорить об интеллектуальном терминале.

Персональные компьютеры и сети. Следующим шагом стало превращение интеллектуального терминала в самостоятельное вычислительное устройство со своим центральным процессором, своей оперативной памятью и своим набором внешних устройств. Быстрое развитие компактных устройств хранения данных на основе магнитных носителей с прямым доступом привело к тому, что их емкости и скорости доступа вполне хватало для размещения персональных данных пользователя. И в какой-то момент показалось, что все проблемы пользователей решены. Они получили в монопольное владение собственный вычислитель — персональный компьютер (ПК, Personal Computer — PC).

Первые операционные системы ПК были лишены и развитых средств мультипрограммирования, и средств защиты данных. Это были однозадачные однопользовательские операционные системы, но продолжалось так недолго. Стремление к получению доступа к централизованным архивам данных привело к новому развитию сетей связи. Да в ряде случаев и решение задач пользователя не могло быть обеспечено ресурсами только одного ПК. Рост объемов памяти и быстродействия микропроцессоров ПК дал возможность перейти к мультипрограммированию и многозадачности.

Телефонные сети, получившие развитие на предшествующем этапе, уже не могли справиться с передачей необходимых объемов данных. Скорости в тысячи и десятки тысяч битов в секунду, которые были вполне достаточны при передаче строковой информации объемом в сотни байтов от центрального вычислителя к терминалу, стало не хватать. Действительно, для передачи простой фотографии размером 1 Мбайт потребуется полторы-две минуты.

Поэтому развитие ПК, как ни странно это казалось на первом этапе их появления, резко стимулировало развитие высокоскоростных сетей связи. А в составе операционных систем стало непременным присутствие программных функций сетевого обмена. По сути, эти функции можно рассматривать как дальнейшее раз-

витие библиотек стандартных функций. Но с другой стороны — это часть современной системы управления данными со своими соглашениями об организации и правилах сетевого доступа.

1.4. ОПЕРАЦИОННЫЕ СИСТЕМЫ СЕМЕЙСТВ UNIX И WINDOWS

Самыми известными и популярными операционными системами общего назначения в настоящее время являются операционные системы семейств Windows и UNIX, а также их ответвления. Операционные системы семейства UNIX являются одной из старейших ветвей операционных систем, при этом до сих пор UNIX-подобные операционные системы широко применяются и не теряют своей актуальности.

Причина того, что речь идет о семействе операционных систем UNIX, а не об одной операционной системе, заключается в том, что UNIX-семейство 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. Операционная система использовалась для решения практических задач сотрудников лаборатории, и широкое ее распространение не планировалось. Через некоторое время бóльшая часть операционной системы была переписана с языка ассемблера на язык C, что дало возможность перенести ее на большое количество разных платформ. В настоящее время UNIX работает на большинстве существующих архитектур и для многих из них является основной ОС.

Дальнейшее развитие UNIX-систем, разработанных в AT&T, и их производных называется System V (пятая версия), сокращенно SysV (иногда используется название AT&T-версия UNIX).

В середине 1970-х гг. исходный код UNIX попал в университет Беркли, где была создана своя версия UNIX, получившая название BSD UNIX (Berkeley Software Distribution).

В настоящее время большинство вариантов UNIX основаны или на System V, или на BSD (рис. 1.3).

Обе ветви в той или иной степени удовлетворяют различным стандартам, в частности стандарту POSIX, и в последнее 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 (раз-

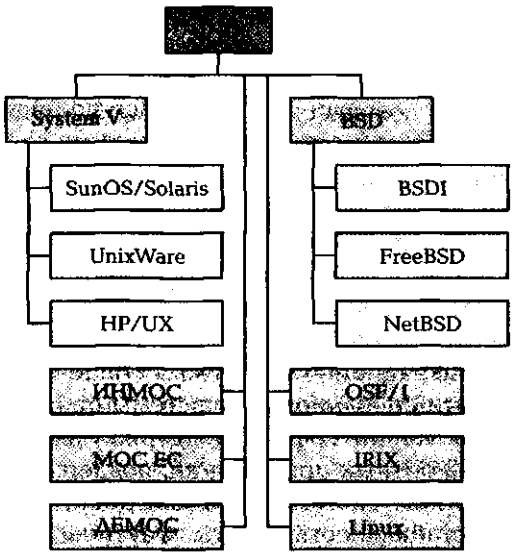

Рис. 1.3. Генеалогическое древо различных вариантов UNIX

работка 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: операционные системы МОС ЕС для ЕС ЭВМ, ДЕМОС для ЕС ЭВМ и СМ ЭВМ, ИНМОС для СМ ЭВМ [5].

Совершенно другая ситуация наблюдается, когда речь заходит об операционных системах семейства Windows. Данные операционные системы разрабатывались и разрабатываются одной-единственной компанией — Microsoft — на протяжении вот уже более 20 лет.

История операционных систем Windows начинает свой отсчет с 1985 г., когда была разработана ОС Windows 1.0. Эта система не являлась полноценной ОС, а была по большей части лишь графической оболочкой для операционной системы MS DOS. Она реализовывала ограниченную поддержку многозадачного режима для программ под ОС MS DOS. В это же самое время Microsoft совместно с фирмой IBM разрабатывала полноценную многозадачную операционную систему с графическим интерфейсом — OS/2. Изначально именно операционная система OS/2 была призвана заменить устаревающую MS DOS, но в начале 1990-х гг. пути Microsoft

и IBM разошлись и Microsoft сконцентрировалась исключительно на развитии своей собственной операционной системы на основе Windows.

В 1990 г. на свет появилась ОС Windows 3.0, которая стала существенным шагом в развитии операционных систем семейства Windows. В 1992 г. вышла значительно переработанная версия ветки Windows 3.x — Windows 3.1, в которой появились поддержка шрифтов TrueType, печать в режиме WYSIWYG, была повышена стабильность работы, добавлены мультимедийные свойства и т. д.

Следующим этапом в развитии ветви операционных систем Windows является семейство Windows 9x. В это семейство входят три версии операционной системы: Windows 95, Windows 98 и Windows ME. Эти операционные системы привнесли обновленный пользовательский интерфейс, поддержку длинных имен файлов, поддержку 32-битных приложений, поддержку стека TCP/IP. Данные операционные системы вызывали достаточно большие нарекания в плане стабильности работы. Фактически ядро этих операционных систем поддерживало так называемую вытесняющую многозадачность (свопинг), позволяющую изолировать процессы друг от друга и в идеале исключить зависания системы из-за зависания одного-единственного процесса. Но из-за того, что реализация части ядра мало отличалась от ядра ОС Windows 3.x, зависание, например, единственного 16-битного приложения в системе приводило к зависанию всей операционной системы.

Совершенно на других принципах создавались системы на основе ветви Windows NT (New Technology). Разработка этой ветви операционных систем семейства Windows началась в 1988 г. Это семейство оказалось наиболее удачной ветвью в развитии операционных систем Windows.

Первая версия, Windows NT 3.1, появилась в 1993 г. Затем с разницей в год последовательно выходили версии Windows NT 3.5, 3.51, 4.0. Версия Windows NT 4.0 имела интерфейс в стиле Windows 95, но страдала отсутствием поддержки DirectX — библиотеки для программирования игровых приложений.

В 2000 г. вышла новая версия операционной системы на основе ядра Windows NT — Windows 2000. Эта система включала новый пользовательский интерфейс, поддерживала файловую систему NTFS 3.0, а также содержала другие существенные изменения.

В 2001 г. появилась операционная система Windows XP. Эта версия была призвана заменить операционные системы Windows 2000, а также предлагалась в качестве замены операционных систем ветви Windows 9x, разработка которой к тому времени была пре-

кращена. Эта операционная система получила новый графический интерфейс, широкую поддержку всевозможных мультимедийных функций, функции восстановления системы, улучшенную совместимость со старыми программами и играми.

В 2006 г. была выпущена в свет операционная система Windows Vista. Эта система поддерживала новый пользовательский интерфейс Windows Aero на основе DirectX, появилась система User Account Control (UAC) для контроля за действиями, выполняемыми пользователями, применялись технологии предотвращения использования эксплойтов Data Execution Prevention и Address Space Layout Randomization (ASLR) и т. д.

Несмотря на все нововведения данная версия подвергалась широкой критике и не получила достаточно широкого распространения в мире. В качестве замены этой версии в 2009 г. была выпущена операционная система Windows 7. Эта система получила новую, 11-ю, версию DirectX, улучшилась совместимость со старыми приложениями (по сравнению с Windows Vista), был снова изменен пользовательский интерфейс системы и т. д.

В 2012 г. фирма Microsoft подготовила к выпуску операционную систему Windows 8. Эта система содержит дополнительные улучшения в механизмах распознавания голоса, улучшенную систему защиты от внедрения вредоносных программ, измененный интерфейс, напоминающий интерфейс мобильной версии Windows Phone 7, и т. д.

Помимо пользовательских версий операционных систем семейство Windows содержит и специальные версии для серверов, больших центров обработки данных и т. п. Фактически выход каждой новой версии пользовательской операционной системы Windows, особенно на основе ядра Windows NT, сопровождается выходом (практически на том же ядре) серверной версии операционной системы. Так, были выпущены серверные версии Windows Server 2003, Windows Server 2008, Windows Small Business 2008, Windows Server 2008 R2, Windows Server 8.

1.5. ПОСТАНОВКА ЗАДАЧИ «КОНТРОЛЬ ЗНАНИЙ»

Для того чтобы сделать изложение материала более понятным, бóльшая часть приводимых в учебнике примеров посвящена постепенной разработке приложения, автоматизирующего выдачу контрольных заданий студентам и сбор выполненных работ пре-

подавателем. Условное название приложения — «Контроль знаний». При этом некоторые аспекты взаимодействия преподавателя со студентами будут намеренно упрощены для сокращения изложения материала. Явно будут выделены только те моменты, которые иллюстрируют применение и работу различных механизмов, предоставляемых операционной системой. Сводный пример для данного приложения приведен в приложении 1.

В основной части учебника текст, относящийся к описанию приложения «Контроль знаний», выделен шрифтом без засечек и имеет больший левый отступ, чем основной текст (как следующий абзац).

Приложение «Контроль знаний» автоматизирует процесс выдачи преподавателем контрольных работ студентам и процесс возврата преподавателю выполненных контрольных работ (рис. 1.4).

Данное приложение 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 написанные работы и проверяет их. Разбор работ и сообщение студентам оценок вынесены за рамки системы;
- студент — просматривает список полученных вариантов контрольных работ, выполняет их и возвращает выполненные работы преподавателю.

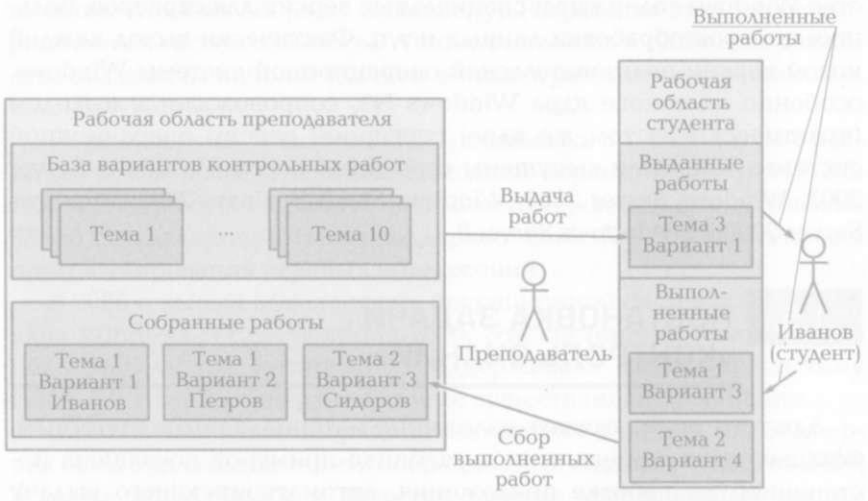

Рис. 1.4. Схема работы системы «Контроль знаний»

Приложение «Контроль знаний» должно 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 выделяется рабочая область, в которую преподаватель помещает варианты контрольных работ, предназначенных для выполнения.

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, хранимые в приложении «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;
- 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.

Система должна автоматизировать основные действия пользователя.

Преподаватель осуществляет:

- 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.

Студент осуществляет:

- просмотр полученных вариантов контрольных работ;
- выполнение контрольной работы;
- сдачу контрольной работы преподавателю.

При написании приложения «Контроль знаний» должны учитываться следующие моменты:

- размещение и структуризация информации на дисковых носителях;
- определение прав доступа к различной информации;
- средства автоматизации типичных действий пользователя.

Интернет-источники

https://secure.wikimedia.org/wikipedia/ru/wiki/Операционная_система_реального_времени
http://www.rtsoft-training.ru/?p=600067&PHPSESSID=9f9bebc48282ab1cc30cd35794ccb576
https://secure.wikimedia.org/wikipedia/en/wiki/Real-time_operating_system
http://www.osp.ru/os/1997/05/179265/

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие основные функции выполняет операционная система?
2. В чем основное различие функций ядра многопользовательской и однопользовательской ОС?
3. Для чего в состав ядра включаются функции ввода/вывода?
4. Может ли существовать ОС, не включающая в свой состав файловую систему?
5. Чем различаются средства общения с операционной системой прикладного программиста и прикладного пользователя?
6. Какие функции операционной системы обеспечивают независимость прикладной программы от физической организации данных на носителях?
7. Может ли существовать ОС, в ядре которой отсутствуют функции поддержки файловой системы?

Глава 2

ФАЙЛОВЫЕ СИСТЕМЫ

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 ряде случаев пользователь может набрать текст, распечатать его, получив твердую копию документа, и завершить свой сеанс работы. При этом пользователь получает результат своей работы — распечатку, а набранный текст представляет собой промежуточные данные, которые хранить не требуется. В более общих случаях текст может иметь большой размер и пользователь не может набрать его за один сеанс работы. Тогда его нужно сохранять между обращениями пользователя к системе. Текст превращается в хранимые данные.

Для продолжительного хранения данных пользователя используются накопители данных — дисковые (жесткие диски, CD- и DVD-диски, флоппи-диски), ленточные (стримеры) или твердотельные (flash-накопители). Взаимодействие с накопителем берет на себя операционная система, а единицей пользовательских данных в этом случае является файл.

Файл (иногда его еще называют набором данных) 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 представляют собой символьные строки ограниченной длины (обычно до 8 или до 255 символов), которые служат для идентификации данных, хранимых в файле. Как правило, существует ряд символов, запрещенных к использованию в именах файлов. Например, в UNIX-системах это символы *, ?, /, \, <, >, &, $, | и ряд других.

Для упрощения сопоставления программы, которая работает с файлом данного формата, можно задать его тип. Как правило, тип файла задается при помощи расширения — части имени файла, отделенного точкой. Так, для файла с именем book.txt book — имя, а txt — расширение. Внутренняя структура файла не зависит от расширения, а типизация файлов может производиться не только при помощи расширений, как это сделано в Windows системах. Например, в UNIX-системах исполняемые файлы программ обычно не имеют расширения, вместо этого им присваивается специальный атрибут «исполняемый файл».

Указание типа в имени файла не определяет формат его обработки. В зависимости от программы, работающей с файлом, данные могут представляться совершенно по-разному. Например, текстовый файл (с расширением txt), открытый в текстовом редакторе, будет представлять собой последовательность символов — текст, и в данном случае для программ проверки орфографии важны отдельные слова, составляющие этот текст. Тот же файл для программы резервного копирования представляет собой просто последовательность блоков данных по 16 Кбайт. Таким образом, структура файла и правила работы с ним задаются именно программой.

2.2. ФАЙЛОВЫЕ СИСТЕМЫ

Что же представляет собой файл с точки зрения операционной системы, 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, по-своему. Представим себе игрушечный конструктор, из которого можно собрать модель автомобиля. Точно так же операционная система 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 — это более низкоуровневое представление данных, чем файл. Его организация во многом определяется свойствами носителя данных.

Набор данных имеет уникальное имя, так же как и файл, содержит все данные, которые содержит файл, но структура этих данных определяется ядром операционной системы. Кроме того, в набор

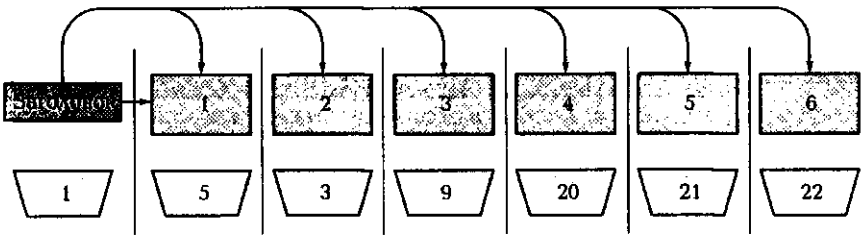

Рис. 2.1. Файловая система со ссылками на все блоки набора данных

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 метода организации данных является то, что список номеров блоков у больших файлов может занимать более одного кластера. В результате размер файла получается ограниченным. В UNIX-системах существуют различные методы обхода этого ограничения, например служебные блоки можно организовать в дерево, при этом каждый служебный блок хранит последовательность блоков с данными и служебные блоки следующего уровня дерева.

Другой 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 кластеров (рис. 2.2). При частых изменениях данных этот

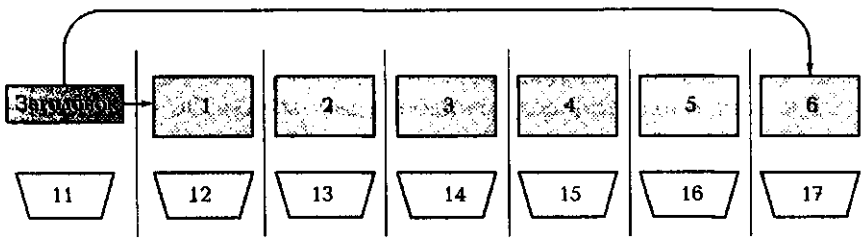

Рис. 2.2. Файловая система с последовательной организацией блоков данных

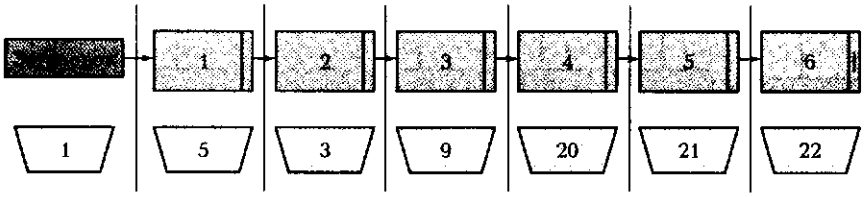

Рис. 2.3. Файловая система со списковой организацией блоков данных

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 сходные принципы положены в основу организации данных на CD-ROM-носителях, а ранее использовались в ОС РАФОС.

Еще один способ организации заключается в выделении в каждом блоке небольшой служебной области, в которой хранится номер следующего блока набора 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 компонент:

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

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

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

Поэтому, 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 имени, а операционная система находит по этому имени идентификатор набора данных.

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

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. 6.4), то эта информация будет одинаковой для всех жестких ссылок.

Кроме жестких ссылок существуют символические ссылки — файлы специального вида, хранящие имя файла, на который указывает эта ссылка (рис. 2.4). 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 продолжать рассмотрение структуры данных и ее изменений далее, до физической структуры данных на носителе, то можно увидеть, что данные проходят серию преобразований (рис. 2.5), при этом каждый потребитель данных (например, про-

Рис. 2.4. Наборы данных, жесткие и символические ссылки

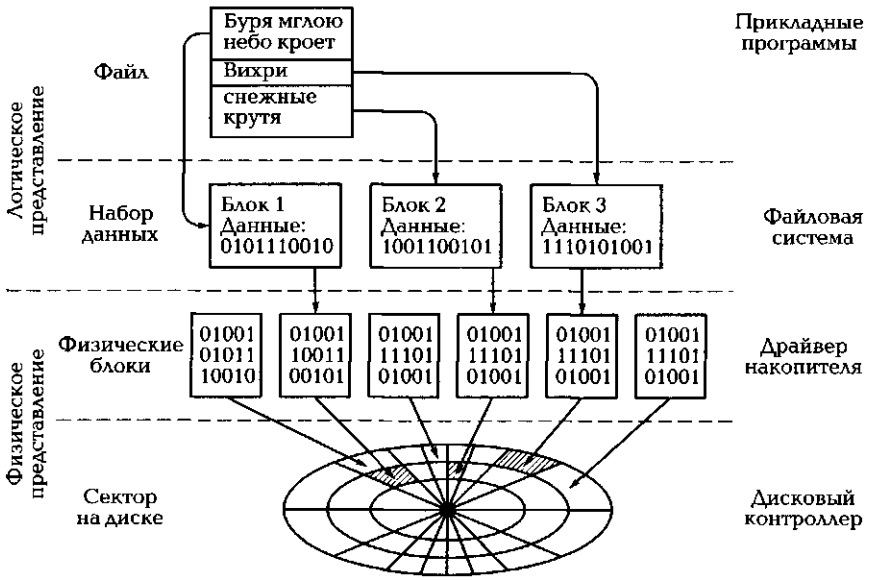

Рис. 2.5. Уровни представления данных

грамма пользователя или файловая система) накладывает свои ограничения на режим доступа.

Рассмотренные 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 равен 512 байт, хотя это значение может быть изменено при создании дискового раздела. При этом размер может варьироваться в диапазоне от 512 до 4096 байт. Идентифицируется сектор при помощи номера считывающей головки, номера дорожки и номера сектора на дорожке. При считывании или записи сектора 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 дисковое пространство в виде неструктурированной последовательности пронумерованных физических блоков. Размер физического блока обычно равен размеру сектора. Далее, 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 определяют логическое представление данных, а два нижних — физическое.

Поскольку основная работа прикладного программиста в UNIX-системах идет на уровне файловой системы, необходимо объединить все перечисленные выше функции файловой системы:

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

2.3. КАТАЛОГИ

Для того чтобы упорядочивать и организовывать файлы, в операционных системах 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 (рис. 2.6).

На приведенном рисунке элементы, имена которых начинаются с dir, являются каталогами, а те, чьи имена 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.

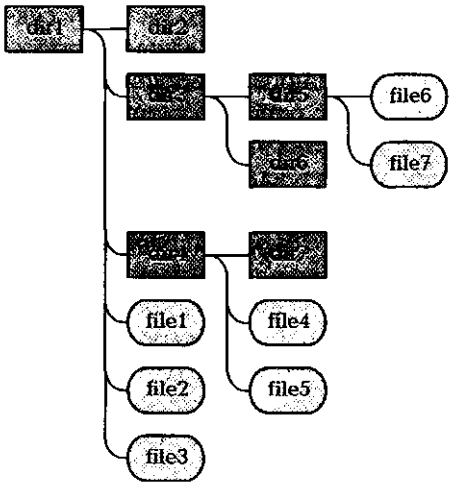

Рис. 2.6. Пример распределения файлов по каталогам

Использование каталогов в современных операционных системах обусловлено тремя факторами:

1) каталоги ускоряют поиск файла операционной системой. Поиск в дереве при прочих равных условиях 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.

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, мы работаем с отдельными записями в нем, т. е. с информационными блоками, содержащими информацию о находящихся в этом каталоге файлах и каталогах.

Простые операционные системы могут работать с файловыми системами, поддерживающими только один каталог — корневой, в котором хранятся файлы. Более сложные операционные системы поддерживают работу с деревом каталогов практически неограниченной вложенности. Естественным ограничением глубины вло-

женности 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 косой черты «/». Например, для файла file6 полным именем будет /dir1/dir3/dir5/file6.

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 связано с тем, что путь задается относительно абсолютной точки отсчета — корневого каталога.

Для того 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 UNIX-подобных операционных системах распределены по отдельным каталогам. Это позволяет структурировать различные файлы по их назначению. Большинство UNIX-систем имеют 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 про-

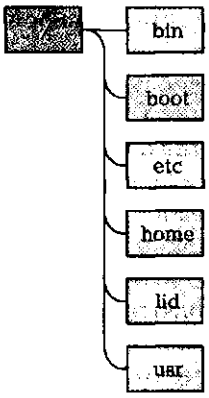

Рис. 2.7. Основные каталоги UNIX-систем

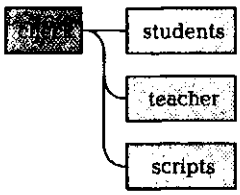

Рис. 2.8. Основные каталоги системы «Контроль знаний»

грамм (/usr), пользовательских данных (/home). Более подробно структура стандартных каталогов UNIX-систем рассмотрена в подразд. 7.1.

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 приложения «Контроль знаний», будет подкаталогом корневого (рис. 2.8) и будет носить имя check.

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 выглядеть как

```
/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 works.

2.4. ОПЕРАЦИИ НАД ФАЙЛАМИ И КАТАЛОГАМИ

Для доступа к файлам операционная система предоставляет набор системных вызовов, реализующих основные функции работы с файлами (табл. 2.1) [16].

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	Данный 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 временного пользовательского идентификатора, используемого 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 байтов 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

Окончание табл. 2.1

Системный вызов	Описание
Append	Данный системный вызов представляет собой урезанную форму вызова Write. Единственная его функция — добавление данных в конец файла
Seek	Вызов предназначен для перемещения текущей позиции в файле в заданное место. Обычно позиция задается количеством байтов с начала файла
Get Attributes	Вызов предназначен для получения атрибутов файла, например даты его создания
Set Attributes	Вызов предназначен для установки атрибутов файла
Rename	Вызов предназначен для изменения имени файла. Данный 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 операционная система 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	Создается пустой каталог, содержащий только элементы «.» и «..»
Delete	Удаляется пустой каталог, не содержащий никаких других файлов и подкаталогов, кроме элементов «.» и «..»
OpenDir	Данный 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

Окончание табл. 2.2

Системный вызов	Описание
CloseDir	Высвобождает память, выделенную под параметры каталога, считанные вызовом OpenDir
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	Аналогично вызову Rename для файла
Link	Создание жесткой ссылки на файл в каталоге
Unlink	Удаление жесткой ссылки 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 либо пользователю 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.

2.5. ПРИНЦИПЫ ОРГАНИЗАЦИИ ФАЙЛОВЫХ СИСТЕМ UNIX И WINDOWS

2.5.1. Принципы организации файловых систем UNIX

Рассмотрим организацию файловых систем ОС UNIX на примере классической файловой системы s5fs.

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 того чтобы обеспечить целостность дан-

ных, 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 данны пользователя (рис. 2.9).

Корнем дерева является индексный блок (i-node), в котором хра нятся атрибуты файлов (дата модификации, дата последнего обра щения, права доступа, тип файла и т. п.) и массив ссылок на блок данных.

Массив ссылок имеет ограниченный размер. В том случае, есл количество блоков превосходит размер массива, создается одинар ный ссылочный блок, на который и начинает ссылаться один из эле ментов i-node. Сам одинарный ссылочный блок ссылается на блок данных.

Поскольку размер массива ссылок в ссылочном блоке такж ограничен, в случае заполнения всех ссылок в индексном блок и одинарном ссылочном блоке создаются двойные ссылочные бло ки. Ссылки на существующие одинарные блоки переносятся в двой ные, а индексный блок начинает ссылаться на двойные блоки.

Таким образом, растет глубина дерева ссылок. Если и двойны ссылочных блоков не хватает для всех блоков данных, входящи

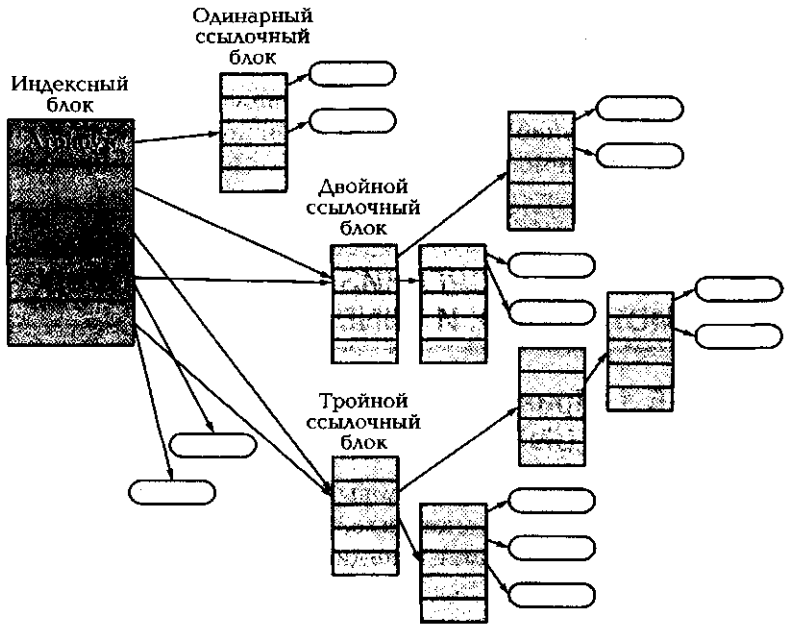

Рис. 2.9. Структура наборов данных в типичной файловой системе UNIX

в набор, то создаются тройные ссылочные блоки. Ссылочных блоков более глубоких уровней вложенности не существует, поскольку временные затраты на поиск блоков данных в таких файлах становятся слишком большими [16].

123	.
4356	..
12	File.txt
4356	Sudbir2

Рис. 2.10. Пример организации каталога

Каталоги в этих файловых системах представляют собой специальный файл, в котором содержится список файлов и подкаталогов (рис. 2.10). В этом файле находятся записи размером 16 байт каждая, содержащие информацию о файлах: в первых двух байтах содержится номер i-node файла, а в следующих 14 байтах — имя файла. Таким образом, в этих файловых системах длина имени файла была ограничена всего лишь 14 символами.

Из-за того что под номер i-node отводится всего 2 байт, максимальное количество файлов в каталоге не могло превышать число 65 535. Более того, минимум три номера в этой таблице резервировались и использовались для описания самого каталога и его родительского каталога, а также для обозначения удаленного файла (значение 0), тем самым сокращая общее количество файлов в каталогах.

Каждый файл имеет свой собственный i-node. В нем содержится вся служебная информация о файле, его размере, правах доступа и т. д. Общая структура i-node приведена в табл. 2.3.

Поле di_mode представляет собой набор битовых полей, определяющих тип и права доступа к файлу (рис. 2.11). Тип файла зада-

Таблица 2.3. Элементы структуры i-node

Поле	Размер, байты	Описание
di_mode	2	Тип файла и права доступа
di_nlinks	2	Количество жестких ссылок на файл
di_uid	2	Идентификатор владельца файла
di_gid	2	Идентификатор группы владельца
di_size	4	Размер файла в байтах
di_addr	39	Индексный блок с адресами блоков памяти
di_atime	4	Время последнего доступа к файлу
di_mtime	4	Время последней модификации файла
di_ctime	4	Время последнего изменения индексного дескриптора

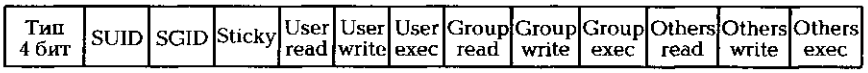

Рис. 2.11. Поле di_mode

ется первыми четырьмя битами и определяет, является файл регулярным, каталогом, файлом блочного устройства и т.д. Остальные биты определяют права доступа к файлу пользователей системы, а также хранят данные о наличии или отсутствии установленных специальных прав.

Эта файловая система отличается простотой. Однако это свойство является обратной стороной определенных проблем с надежностью, производительностью и функциональностью. Использование косвенной адресации для блоков памяти привело к тому, что в случае сбоев в этой файловой системе произойдет разрушение данных или структуры файловой системы. Существенным недостатком выглядят также ограничения на максимальную длину имени файла и максимальное количество файлов в каталоге.

В операционных системах семейства Linux основной файловой системой до последнего времени являлась ext2fs. Она пришла на замену файловой системы операционной системы Minix, которая использовалась в ранних версиях операционных систем Linux. Она была очень похожа на файловую систему s5fs и к 1990-м годам устарела, так как начали сказываться ограничения ее 16-битной архитектуры. Поэтому сначала была разработана расширенная файловая система extfs, а вскоре за ней последовала и ext2fs.

Файловая система ext2fs расширила ограничения на размер файла до 2 Гбайт и увеличила длину имени файла до 255 символов. Кроме того, в этой файловой системе были реализованы некоторые элементы стандарта POSIX. В основе же своей она очень похожа на классическую файловую систему s5fs.

Развитием этой файловой системы была ext3fs. Основным отличием ее от предшественницы была поддержка механизма журналирования.

Журналирование — процесс записи некоторых служебных данных в специальный файл (журнал), позволяющих произвести восстановление структуры файловой системы или даже ее данных в случае возникновения сбоя. В большинстве своем современные файловые системы являются журналируемыми и обеспечивают достаточно высокую надежность хранения данных.

Существует три вида журналирования: 1) writeback, при котором сохраняется только информация об изменениях в структуре

файловой системы, причем запись в журнал не синхронизирована с записью данных; 2) ordered, которое похоже на writeback, но запись в журнал происходит заведомо после того, как происходят изменения в файле; 3) journal, при котором сохраняется информация не только об изменениях в структуре файловой системы, но и об изменении данных. Самым надежным и самым медленным видом журналирования является journal, а наименее надежным и самым быстрым — writeback.

Файловая система ext3fs поддерживала все эти виды журналирования и являлась более надежной, чем ext2fs. Но ее существенным недостатком была более низкая скорость работы, чем у ext2fs.

Для решения проблем со скоростью работы файловой системы, а также для улучшения других параметров была разработана файловая система ext4fs. Эта файловая система также является журналируемой, кроме того, она поддерживает существенно большие по размеру разделы. К тому же в нее включена поддержка экстентов, т. е. возможность выделять память для файлов не только поблочно фиксированного размера, но выделять сразу большую область памяти, объединяющую несколько блоков, для описания которой используется один-единственный дескриптор. Данная файловая система позволила создавать в системе более 32 000 подкаталогов, что было одним из ограничений в файловой системе ext3fs и более ранних.

2.5.2. Принципы организации файловых систем Windows

Одной из старейших и популярнейших файловых систем является FAT. Разработанная изначально для операционной системы MS DOS, данная файловая система получила широкое распространение благодаря простоте, быстроте и эффективности.

Данная файловая структура имела очень простую структуру (рис. 2.12). В начале располагался загрузочный сектор, затем — две таблицы размещения файлов (File Allocation Table — FAT, которые и дали название файловой системе, причем вторая FAT является полной копией первой и хранится на случай ее повреждения), корневой каталог и файлы.

Рис. 2.12. Структура файловой системы FAT

Все дисковое пространство в этой файловой системе разбивается на кластеры, а таблицы FAT содержат служебные ячейки для каждого кластера. Информация о файле представляет собой список. Если ячейка соответствует последнему кластеру в файле, то она содержит значение FFFF. Неиспользуемые кластеры помечены 0000, а удаленные — FFF7. При удалении файла делается только запись в каталоге, а цепочка кластеров не разрушается. Это позволяет достаточно просто восстанавливать файлы, если не была нарушена целостность цепочки ячеек в таблице.

Эта файловая система обладает хорошей устойчивостью. Данные о файлах и свободных кластерах хранятся в одной таблице, поэтому запись файла, обычно выполняющаяся в две операции, в FAT выполняется в одно действие.

Файловая система FAT пережила несколько операционных систем и несколько инкарнаций. В настоящее время существует четыре версии FAT: FAT12, FAT16, FAT32 и exFAT. До настоящего времени этот формат файловой системы активно используется, особенно для переносимых носителей данных, таких как флэш-накопители.

Файловая система NTFS была разработана фирмой Microsoft в начале 1990-х годов. Данная система во многом являлась потомком файловой системы HPFS, которая использовалась в новаторской для своего времени операционной системе OS/2. Эта операционная система разрабатывалась совместно фирмами Microsoft и IBM. Но затем фирма Microsoft решила продвигать свою собственную операционную систему 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, состоят в следующем:

- файловые системы FAT чрезвычайно подвержены фрагментации данных, снижающей общую производительность файловой системы;
- перерасходуют дисковое пространство для достаточно больших объемов логических разделов;
- не поддерживают механизмы восстановления от ошибок, позволяющие восстанавливать работоспособность файловой системы в случае возникновения сбоев или внезапной перезагрузки;

- не имеют поддержки POSIX, что препятствовало использованию операционных систем Windows в ряде организаций и компаний;
- не имеют 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, следующие:

- надежность и восстанавливаемость. NTFS спроектирована таким образом, что операции ввода/вывода представляют собой транзакции. Эти транзакции неделимы, т. е. требуемая операция либо завершается полностью, либо не выполняется вообще. Это позволяет безболезненно совершать операции отката состояния файловой системы в случае 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, таких 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. 2.13.

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

Загрузочный сектор	Мастер File Table	Резерв MFT	Системные файлы	Обычные файлы	Копия первых 16 записей MFT	Обычные файлы

Рис. 2.13. Структура файловой системы NTFS

предназначена для хранения информации о самой MFT. Вторая запись содержит информацию о копии таблицы MFT. Эта копия 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 по 16 хранят информацию о прочих служебных файлах, а записи 17 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. 2.4.

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 не только хранить в файле основные данные, но и приписывать дополнительные данные в альтернативные потоки данных,

Таблица 2.4. 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

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 же атрибуты.

В качестве примера использования потоков можно рассмотреть следующий листинг:

```
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
```

Видно, что, хотя основной поток данных файла data.txt содержит внутри сообщение data, два альтернативных потока данных содержат свои собственные данные. Но в файловой системе все эти потоки будут представлять собой один-единственный файл.

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

Также следует упомянуть и еще один механизм, поддержка которого была внедрена в файловой системе NTFS, — работу с жесткими ссылками. Данный 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 ними.

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

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие основные функции выполняет файловая система?
2. В чем основное различие логической и физической структур данных на носителе?
3. Какова структура данных NTFS?
4. Что такое родительский каталог?
5. Что такое текущий каталог?
6. Чем различаются абсолютное и относительное имена файла?
7. Какие функции операционной системы обеспечивают взаимосвязь прикладной программы и физической организации данных на носителях?
8. Может ли относительное имя файла быть длиннее абсолютного?

Глава 3

УПРАВЛЕНИЕ ПАМЯТЬЮ В ОПЕРАЦИОННЫХ СИСТЕМАХ

3.1. ОБЩИЕ ПОНЯТИЯ

Память, чаще называемая оперативной памятью, является одним из важнейших ресурсов любой вычислительной системы. Поэтому подсистема управления памятью является важнейшей функцией операционной системы. Для того чтобы программы могли быть выполнены процессором вычислительной системы, необходимо, чтобы исполняемый код программы был загружен в основную память.

Исторически скорость обработки данных в основной памяти существенно быстрее скорости обработки данных на внешних носителях, на которых обычно хранятся программы. Поэтому при разработке операционной системы стараются оптимизировать процесс загрузки и выполнения программ таким образом, чтобы каждому выполняющемуся процессу было выделено достаточное количество основной памяти. Это позволяет, во-первых, ускорить выполнение текущего процесса и, во-вторых, не замедлять работу остальных процессов и подсистем операционной системы.

Проблемы возникают тогда, когда операционная система должна распределить ограниченный объем оперативной памяти между несколькими задачами (процессами). При этом каждый получает только такую часть, которая достаточна для выполнения текущих вычислений, но меньше всей памяти, необходимой для работы программы. Тогда неактивная часть задачи находится во внешней памяти и подкачивается оттуда по мере необходимости.

Отсюда следуют основные цели, которых необходимо достичь при разработке подсистемы управления памятью в операционной системе:

1) сократить время доступа к основной памяти;

2) увеличить размер доступной процессам основной памяти;

3) увеличить эффективность доступа процессов к основной памяти.

Основные задачи, которые решает подсистема управления памятью, следующие:

1) выделять процессам основную память;

2) отображать адресное пространство процесса на выделенную область основной памяти;

3) минимизировать время доступа данным, используя доступный объем основной памяти.

Для решения этих задач в условиях, когда происходит динамическое выделение памяти, менеджер памяти должен поддерживать список всех блоков памяти, которые могут быть выделены процессам. Этот список может поддерживаться с помощью практически любой структуры данных, имеющих в основе связанный список. В самом простом случае такая структура может представлять собой список дескрипторов блоков памяти, где каждый дескриптор содержит указатель на соответствующий блок памяти и его размер. Менеджер памяти поддерживает целостность данной структуры и вставляет в список или удаляет из него элементы в определенном порядке, в соответствии с принятой стратегией управления памятью. Существует несколько стратегий распределения памяти между процессами, конкурирующими между собой за право владеть некоторым объемом памяти:

1) распределение на основе стратегии «наиболее подходящий участок»;

2) распределение на основе стратегии «наименее подходящий участок»;

3) распределение на основе стратегии «первый подходящий участок»;

4) распределение на основе стратегии «следующий подходящий участок».

При использовании первой стратегии менеджер выделяет процессу наименьший доступный блок свободной памяти, превышающий по размеру требуемый объем. Данная стратегия требует достаточно затратной операции просмотра всего списка свободных блоков памяти для поиска блока, подходящего по размеру. Решением может служить сортировка списка блоков памяти по размеру.

Существует более важная проблема, связанная с методом выделения памяти по принципу наибольшего подобия. При использовании данной стратегии возникает множество мелких областей памяти, которые слишком малы для использования и остаются незадействованными. Эта ситуация называется фрагментацией памяти и ведет к неоправданным потерям этого ресурса. Проблема фрагментации памяти присуща всем методам управления памятью,

но в наибольшей степени проявляется именно в стратегии «наиболее подходящий участок».

Для того чтобы избежать фрагментации, можно выделять больший объем памяти, чем запрашивается. Например, можно выравнивать размер каждого запрашиваемого блока на определенную величину (обычно кратную степени двойки) и выделять больший объем памяти. При этом проблема фрагментации памяти не решается, данный подход позволяет лишь скрыть фрагментацию.

Как ни странно, уменьшить фрагментацию памяти позволяет стратегия «наименее подходящий участок». Суть данного метода состоит в том, что всегда ищется самый большой свободный участок памяти и именно он выделяется для использования. Основная идея такой стратегии заключается в следующем. Образующиеся неиспользуемые области памяти, приводящие в первом случае к фрагментации, имеют больший размер, чем при использовании первой стратегии. Если в случае стратегии «наиболее подходящий участок» образующиеся участки памяти («дыры») имеют слишком малый размер для повторного использования и вызывают фрагментацию памяти, то «дыры», образующиеся при использовании второго подхода, могут использоваться при последующих запросах на выделение памяти, тем самым уменьшая возможную фрагментацию.

При использовании стратегии «первый подходящий участок» менеджер памяти сканирует список свободных блоков в поиске первого блока, подходящего по размеру. Несмотря на свою простоту такой алгоритм выбора свободных блоков в среднем показывает себя лучше, чем стратегия «наиболее подходящий участок», так как при его использовании меньше фрагментируется память. Основная проблема данной стратегии состоит в том, что большая часть «дырок» памяти концентрируется в области начала списка свободных блоков, заставляя менеджер памяти с течением времени просматривать список свободных блоков все дальше и дальше от начала списка.

Для решения основной проблемы стратегии «первый подходящий участок» используется стратегия «следующий подходящий участок». Для того чтобы исключить проблему концентрации небольших фрагментов памяти в начале списка, используется кольцевой список свободных блоков. При этом для выбора блоков памяти используется стратегия «первый подходящий участок», но каждый раз после выбора блока голова списка начинает указывать на элемент в списке, следующий за выбранным. При этом фрагменты более-менее равномерно распределяются по всему списку свободных блоков.

Как бы ни был хитроумен алгоритм выбора свободных блоков памяти, невозможно придумать абсолютную стратегию, которая бы *обеспечивала полное отсутствие фрагментации памяти. Рано или* поздно количество «дыр» в памяти начинает превышать некоторое критическое значение, и такая ситуация может требовать дополнительной обработки. Такая обработка называется уплотнением или дефрагментацией памяти.

Процедура дефрагментации решает проблему «дыр» за счет сдвига всех используемых блоков памяти к одному концу памяти, сливая «дыры» в один большой свободный блок. Данная операция является очень затратной, так как требует приостановки всех процессов, вычисления требуемых сдвигов в адресах, физического перемещения данных и т. д. Несмотря на это данный подход используется в некоторых операционных системах, не предъявляющих критических требований ко времени отклика процессов, но упрощающих работу систем программирования и самих программ.

3.2. ВИРТУАЛЬНАЯ И ФИЗИЧЕСКАЯ ПАМЯТЬ

Физическая память является одной из составных частей аппаратных средств любой вычислительной системы, которая используется для загрузки операционной системы и исполнения пользовательских и системных процессов. Сегодня эта память представляет собой несколько модулей памяти, в свою очередь, состоящих из нескольких чипов RAM. Эта память является достаточно быстрой (по скорости она уступает лишь регистровой и кэш-памяти процессора), но при этом достаточно дорогой, и максимальный объем этого вида памяти обычно сильно ограничен.

Из-за этих ограничений физическая память может быть очень быстро исчерпана, если одновременно выполняется большое количество процессов. Для решения данной проблемы используется так называемая виртуальная память. Виртуальная память дает возможность объединить несколько видов памяти (память с произвольным доступом RAM, более медленную дисковую память и т. д.), позволяя разрабатывать достаточно большие программы. Вся память для этих программ представляется в виде единого массива виртуальной памяти, которая ведет себя как обычная физическая память, но большого объема. Таким образом, виртуальная память позволяет отделить логическую структуру памяти от ее аппаратной организации.

Использование виртуальной памяти позволяет решить проблему фрагментации физической памяти за счет использования вир-

туальной адресации. При применении данного механизма каждому процессу выделяется собственная независимая область виртуальной памяти, что позволяет защитить процессы от взаимного влияния (рис. 3.1). Также использование виртуальной памяти и логической адресации позволяет перемещать процессы в памяти за счет смены их базовых адресов.

Виртуальная память позволяет выполняться процессам в случае, если только часть их адресного пространства отображена в основную память. В течение всего жизненного цикла требуемые области памяти подгружаются в основную память, а участки, которое долгое время не были востребованы, сбрасываются в область более медленной дисковой или иной дополнительной памяти. Этот механизм позволяет выполнять программы, которые имеют больший размер, чем физический размер основной памяти. Он также позволяет размещать в физической памяти одновременно несколько процессов (или, по крайней мере, их важные части) для поддержки многопроцессного режима работы операционной системы. В качестве основных элементов управления процессами перемещения виртуальной памяти из основной памяти в дополнительную применяются механизмы подкачки, а для определения размеров перемещаемых блоков используется либо перемещение процессов в основную память и из нее, либо сегментная или страничная подкачка виртуальной памяти.

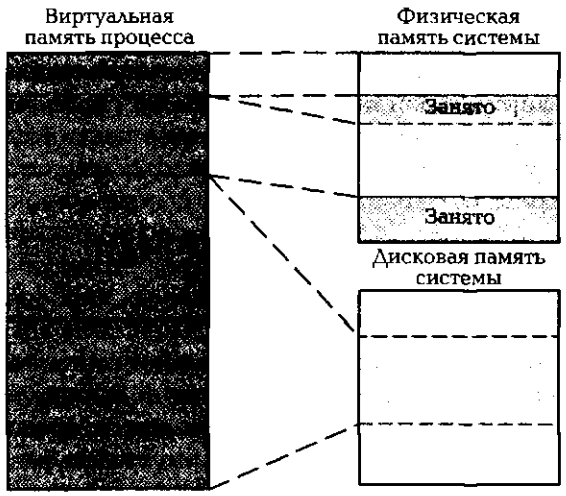

Рис. 3.1. Отображение виртуальной памяти процесса на физическую память

При перемещении процессов заблокированные процессы могут быть полностью перенесены из основной памяти в дополнительную. При этом переносятся полностью все данные процесса и его код, освобождая основную память, а сам процесс переводится в неактивное состояние до тех пор, пока не возникнет необходимость в его активации. В этом случае код процесса и все его данные копируются обратно из дополнительной памяти в основную, возможно, за счет того, что какой-то другой процесс переносится из основной памяти в дополнительную. После этого процесс переводится в активное состояние и продолжает выполнение с той инструкции, в момент исполнения которой он был приостановлен.

Данный подход позволяет одновременно выполнять несколько процессов в основной памяти, причем для этих процессов отсутствуют задержки, связанные с подгрузкой данных, находящихся в дополнительной памяти. Все данные активных процессов всегда располагаются в оперативной памяти, позволяя повысить скорость их реакции. Но при этом операция переключения процессов из активного состояния в неактивное является достаточно долгой, так как приходится полностью загружать/выгружать все данные процесса. Кроме того, такой подход накладывает дополнительные ограничения на размер процесса и его данных: суммарно для одного процесса размер кода и данных не должен превышать доступный объем оперативной памяти.

Данный подход к использованию виртуальной памяти характерен для самых ранних реализаций механизма виртуальной памяти. В дальнейшем наибольшее распространение получили алгоритмы управления виртуальной памятью на основе сегментов, а в настоящее время главным образом используется подход к управлению на основе страниц памяти.

3.3. СЕГМЕНТНАЯ И СТРАНИЧНАЯ ОРГАНИЗАЦИЯ ПАМЯТИ

Сегментная организация виртуальной памяти основана на следующем подходе: все адресное пространство памяти исполняемой программы разбивается на непересекающиеся области, которые называются сегментами. Эти сегменты могут иметь разный размер. Адресация при такой организации виртуальной памяти осуществляется с помощью двух значений, первое из которых задает номер сегмента, а второе — смещение внутри сегмента.

Такие сегменты могут выбираться как самим разработчиком программы (т. е. вручную), так и автоматически линкером при сборке исполняемой программы. Например, в качестве таких сегментов могут использоваться сегменты кода, данных, констант и т. д. Такие сегменты могут выгружаться из основной памяти в дополнительную, если они не нужны в настоящий момент времени, и загружаться из дополнительной памяти в основную, если они требуются для работы процесса.

Такая модель организации виртуальной памяти является очень удобной с точки зрения программиста, поскольку она сама по себе предлагает модульную структуру программ. С каждым сегментом ассоциирован его дескриптор, содержащий информацию о размере и атрибутах сегмента. Процесс может обратиться к какой-то области памяти, только если такая операция разрешена атрибутами сегмента, в котором находится область памяти, а смещение не превосходит размер сегмента. В противном случае генерируется исключение о нарушении границ сегмента.

Кроме того, данный дескриптор содержит информацию о том, где именно в памяти располагается сегмент, а также флаг присутствия, сообщающий системе о наличии или отсутствии сегмента в основной памяти. Если происходит попытка обращения к сегменту, который не загружен в основную память, то генерируется исключение и операционная система подгружает требуемый сегмент в основную память, настраивает параметры в дескрипторе сегмента и выполняет операцию обращения к памяти в сегменте.

Рассмотренный вид организации виртуальной памяти уменьшает фрагментацию. Кроме того, он позволяет экономить основную память за счет выгрузки неиспользуемых сегментов, а также с помощью сегментов решать проблему разделяемых программных модулей. Такие модули загружаются в сегмент памяти, а в таблицы дескрипторов используемых сегментов в процессах заносятся ссылки на этот единственный сегмент.

Недостатки такой организации виртуальной памяти связаны с тем, что операция адресации в этой модели является достаточно медленной, так как сначала надо получить доступ к дескриптору сегмента, считать адрес размещения сегмента в памяти, а потом вычислить конечный адрес, используя смещение. Кроме того, память и процессорное время тратятся на операции с дескрипторами сегментов и таблицами дескрипторов, их размещением и обновлением адресов при операциях подгрузки сегментов.

Для решения этих проблем в современных операционных системах используется страничный метод организации виртуальной

памяти. При страничной организации все пространство виртуальной памяти делится на блоки фиксированного размера, которые называются страницами памяти. Каждый раз, когда требуется выделение памяти определенного размера, этот размер выравнивается по размеру страницы в бóльшую сторону и выделяется целое количество страниц памяти. Если часть какой-либо страницы не используется, то данная память дальше не распределяется и теряется для дальнейшего использования. В этом случае говорят о внутренней фрагментации.

При такой организации виртуальной памяти основная память делится на страницы того же размера, что и страницы виртуальной памяти, которые носят название «фреймы страниц памяти». Такие фреймы являются как бы рамками, в которые должны укладываться страницы виртуальной памяти. Такие страницы могут независимо друг от друга сбрасываться в дополнительную память и по мере необходимости подгружаться в доступный фрейм в основной памяти.

Достоинство данного метода организации виртуальной памяти состоит в том, что так же, как и сегментный метод, этот метод позволяет увеличить доступный объем памяти за счет использования дополнительной памяти. Для одного процесса в каждый момент времени требуется загружать не все принадлежащие ему страницы памяти, а только те, которые реально используются. Остальные могут быть выгружены в дополнительную память, тем самым освобождая место в основной памяти для других процессов и их данных.

Если происходит обращение к странице памяти, которая в настоящий момент выгружена из основной памяти в дополнительную, то генерируется ошибка страницы. При получении такой ошибки менеджер памяти подгружает требуемую страницу в свободный фрейм основной памяти. При необходимости такой фрейм освобождается путем выгрузки неиспользуемых страниц. Частота таких операций зависит от размера страниц и общего количества страниц, которые используются процессом. На практике существует минимальное количество страниц для процесса, которые всегда должны находиться в основной памяти. Это количество носит название «минимальное рабочее множество страниц» и определяется в зависимости от максимального размера памяти, используемой процессом.

Для определения того, какая страница памяти должна быть выгружена из основной памяти в дополнительную, используется специальный алгоритм, называемый «стратегия замещения страниц».

Могут использоваться следующие алгоритмы для реализации таких стратегий.

Алгоритм LRU (Least recently used — дольше всего неиспользуемая страница). При использовании данного алгоритма из основной памяти в дополнительную выгружается страница, которая дольше всех не использовалась. Для реализации данного алгоритма каждой странице сопоставляется метка времени, которая обновляется каждый раз, когда к странице памяти идет обращение. В этом случае операционная система должна просканировать все страницы в поисках той, для которой метка времени покажет, что эта страница дольше всех оставалась неиспользованной. Альтернативным вариантом решения данной проблемы может быть список страниц, отсортированный по значениям меток времени.

Алгоритм NRU (Not recently used — страница, неиспользуемая некоторое время). Данный алгоритм очень похож на алгоритм LRU, но требует гораздо меньше накладных расходов. Каждая страница памяти снабжается специальным битом, который принимает значение 1, если к странице произошло обращение. С некоторой периодичностью операционная система сбрасывает этот бит у всех страниц в 0. Каждая страница, у которой для данного бита установлено значение 0, может быть выгружена в дополнительную память, чтобы освободить место в основной памяти.

Алгоритм FIFO (First in, first out — очередь страниц). При использовании данного алгоритма выгружается в дополнительную память страница, которая наибольшее время находилась в основной памяти, на основании этого полагая, что данная страница вряд ли понадобится в ближайшее время. Для реализации данной стратегии все страницы памяти организованы в упорядоченный по времени нахождения в основной памяти список. Страница, находящаяся в голове списка, будет перемещена из основной памяти самой первой, а страница, перемещенная в основную память, помещается в конец этого списка. Однако данный метод может приводить к низкой производительности подсистемы памяти, так как при его использовании не учитывается частота обращения к страницам. Это может приводить к тому, что часто используемые страницы памяти будут выгружаться на общих основаниях, и вызывать большое количество генерации исключений ошибок страниц и необходимость постоянной подкачки часто используемых страниц из дополнительной памяти.

Кольцевой алгоритм. Метод является одним из вариантов стратегии FIFO, за исключением того, что все страницы организованы в кольцевой список. С каждой страницей ассоциирован ссылочный

бит, для которого устанавливается значение 1, когда к странице идет обращение. Каждый раз, когда необходимо переместить страницу из основной памяти, менеджер памяти просматривает список страниц, пока не встретится страница со ссылочным битом со значением 0. При этом для каждой просмотренной страницы этот ссылочный бит сбрасывается в 0. Как только встретилась страница с ссылочным битом 0, такая страница перемещается в дополнительную память.

Есть и другие алгоритмы управления процессом выгрузки страниц. Например существует частотный алгоритм, при котором учитывается частота обращений к страницам памяти. Также может быть применен случайный выбор страниц для выгрузки.

Стоит отметить, что не все страницы могут быть перемещены из основной памяти в дополнительную. Существует целый ряд страниц, которые всегда должны присутствовать в памяти. Например, механизм прерываний основан на использовании массива указателей на их обработчики, например для обработки ошибок страниц или завершения операций ввода/вывода. Эти обработчики должны всегда находиться в памяти. Поэтому страницы памяти, в которых они располагаются, помечаются как неперемещаемые.

Некоторые страницы могут быть неперемещаемыми в течение всего времени работы операционной системы, другие могут быть неперемещаемыми лишь некоторый период времени. Такими страницами могут являться страницы памяти, содержащие буферы, связанные с внешними устройствами или для операций ввода/вывода. Такие страницы могут быть помечены как неперемещаемые, пока не будет завершена та или иная операция.

3.4. МЕХАНИЗМЫ УПРАВЛЕНИЯ ПАМЯТЬЮ В UNIX- И WINDOWS-СИСТЕМАХ

Первые версии операционных систем семейства UNIX появились еще в 1970-е гг. Эти операционные системы предназначались для обслуживания машин класса PDP-11, которые имели 16-разрядную архитектуру и могли адресовать не более 64 Кбайт памяти. Из-за ограничений в вычислительных ресурсах разработчикам ОС UNIX пришлось использовать подход к управлению памяти на основе программных оверлеев. Оверлеи позволяют многократно использовать память за счет затирания неиспользуемой памяти и загрузки в нее новых блоков кода и данных. Это давало возможность в процессе работы системы, например, затирать код, выпол-

нявшийся на этапе загрузки и инициализации операционной системы и не нужный на этапе работы системы, и использовать высвободившуюся память для других программ.

Данный подход накладывает дополнительные требования на процесс разработки ПО, так как от разработчика требуется явное указание того, когда и какие части кода следует выгрузить или, наоборот, загрузить в основную память. Такие программы были очень плохо переносимыми, поскольку зачастую в них использовались аппаратные особенности тех систем, под которые они были разработаны.

Следующим этапом развития подсистем управления памятью в операционных системах семейства UNIX стало внедрение стратегии на основе полной выгрузки программ в дополнительную память. При использовании этой схемы программы по очереди загружались в оперативную память. Если вдруг выяснялось, что для очередного процесса не хватает памяти, то операционная система выгружала один из процессов из основной памяти в специальный раздел для выгрузки. При старте процесса в этом разделе резервировалось необходимое пространство, что гарантировало наличие необходимого пространства для выгрузки процесса из основной памяти в дополнительную.

В конце 1970-х гг., после появления машин класса VAX-11, в операционные системы UNIX стал внедряться подход по управлению памятью на основе загрузки страниц памяти по запросу. Впоследствии данный подход полностью вытеснил метод управления памятью на основе полной выгрузки процессов. В некоторых реализациях был использован подход на основе сегментной структуры памяти, но в основных реализациях UNIX систем этот подход не прижился и в дальнейшем почти все реализации использовали страничный метод управления памятью.

Подход по управлению памятью на основе страниц используется и в операционных системах семейства Linux. В 32-битных версиях доступно только 4 Гбайт памяти. При стандартных настройках из этих 4 Гбайт 3 отводятся под пользовательскую память, 1 Гбайт отводится под системную памяти. При использовании 64-битных версий операционных систем эти ограничения увеличиваются для пользовательской памяти до 512 Гбайт и более, а оставшееся адресное пространство отводится под системную память (рис. 3.2).

В Linux, как и большинстве версий операционных систем семейства UNIX, в качестве дополнительной памяти используется специальный дисковый раздел. Для определения того, какая

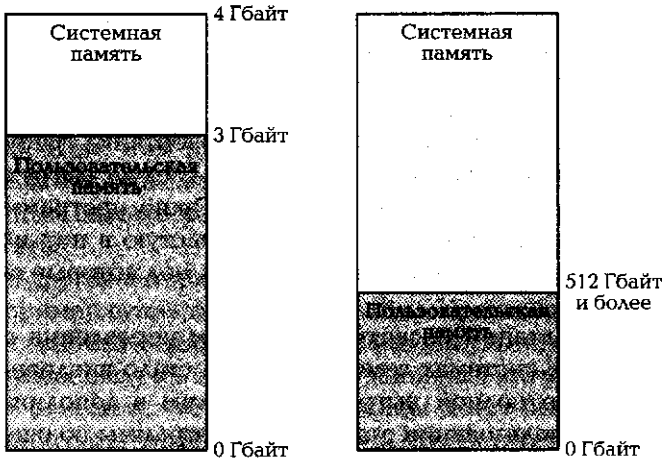

Рис. 3.2. Распределение памяти в 32- (слева) и 64-битных (справа) системах Linux

страница должна быть выгружена из основной памяти в дополнительную, в Linux используется алгоритм LRU, т. е. выгружается та страница памяти, которая дольше всех не использовалась. При этом если для выгружаемой страницы известно, что она является частью исполняемого файла или файла с данными и не была модифицирована, то такая страница может быть вообще удалена из памяти, а не выгружена в дополнительную память. Если впоследствии эта страница будет нужна, то она будет напрямую считана из соответствующего файла. Если же в странице были модифицированы данные, то такая страница будет принудительно выгружена в дополнительную память до тех пор, пока соответствующая страница не понадобится какому-либо процессу или операционной системе.

Операционные системы семейства Windows также относятся к классу систем, использующих виртуальную память. Все процессы работают с виртуальными адресами, только ядро системы использует прямую адресацию физической памяти. Менеджер памяти поддерживает таблицы страниц, которые используются процессором для отображения виртуальных адресов в физические.

Любой процесс, исполняемый под управлением 32-битной версии операционной системы семейства Windows, получает доступ к пространству виртуальной памяти объемом 4 Гбайт. Это ограничение соблюдается всегда, несмотря на объем установленной в систему оперативной памяти. Вся виртуальная память делится на две части: пользовательская память и системная память.

Пользовательская память представляет собой часть виртуальной памяти, которую операционная система применяет для загрузки пользовательских процессов, данных этих процессов и пользовательских библиотек. Каждый процесс имеет свой собственный контекст, т. е. код и данные, используемые этим процессом. Пока процесс выполняется, часть контекста процесса, называемая рабочим множеством, постоянно находится в памяти.

Системная память — часть адресного пространства, в которой располагается код операционной системы и драйверы уровня ядра. Эта память доступна только для кода, выполняемого на уровне ядра, пользовательские процессы не могут получить доступ к данной памяти. Такое разделение поддерживает целостность всей системы и защищает важные части операционной системы от неосторожного влияния пользовательских процессов. Драйвера уровня ядра являются доверенными модулями операционной системы и могут получить доступ и к пользовательской, и к системной памяти.

В 32-битной версии операционных систем младшие 2 Гбайт виртуальной памяти относятся к пользовательской памяти, а старшие 2 Гбайт — к системной. У администратора системы есть возможность изменить эти ограничения с помощью специального ключа /3GB в файле Boot.ini. В этом случае под пользовательскую память отводится 3 Гбайт младших адресов памяти, а под системную — только 1 Гбайт (рис. 3.3).

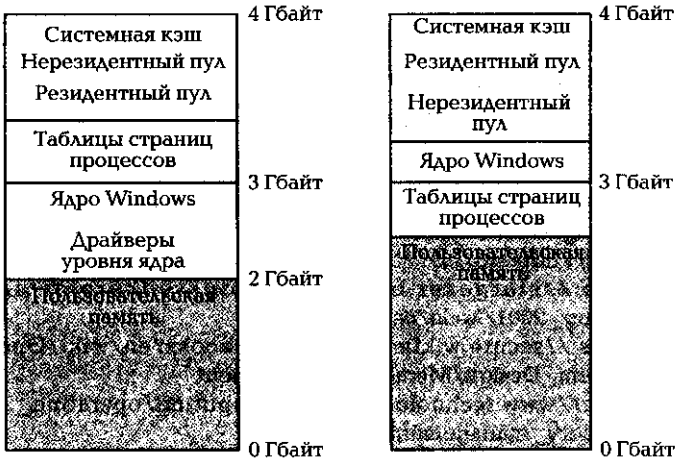

Рис. 3.3. Распределение памяти в 32-битных системах Windows по умолчанию (слева) и с ключом /3GB (справа)

В 64-битных версиях операционных систем семейства Windows виртуальное адресное пространство имеет объем 16 Тбайт, из которых 8 Тбайт отводится под пользовательскую память и столько же — под системную. Расположение основных секций сходно с таковым в 32-битных версиях. Размеры же секций зависят от версий операционных систем и могут изменяться.

Для увеличения объема доступной виртуальной памяти в операционных системах семейства Windows используется дополнительная память, располагающаяся в разделах дисковых накопителей. Для этого в Windows имеется специальный файл подкачки страниц памяти pagefile.sys. Windows использует страничную организацию виртуальной памяти. Обычно размер страниц составляет 4 Кбайт, но на серверных системах с процессорами Itanium используются страницы размером 8 Кбайт. По умолчанию данный файл располагается в корневом каталоге раздела, в который установлена операционная система. При этом администратор может использовать свободное место в любом разделе для файла подкачки.

По умолчанию файл подкачки настраивается так, что его размер может изменяться динамически в зависимости от количества используемой виртуальной памяти. Если изменение размера файла подкачки происходит достаточно часто, он становится очень сильно фрагментированным, что сильно снижает скорость операций с файлом подкачки и производительность системы в целом. Поэтому рекомендуется настроить файл подкачки так, чтобы он имел постоянный фиксированный размер. При этом следует установить размер файла подкачки не менее чем в 2,5 раза больше, чем размер установленной в системе оперативной памяти.

Интернет-источники

https://secure.wikimedia.org/wikipedia/en/wiki/Operating_system
https://secure.wikimedia.org/wikipedia/en/wiki/Virtual_memory
https://secure.wikimedia.org/wikipedia/en/wiki/Memory_management
http://tldp.org/LDP/tlk/mm/memory.html
http://stargazer.bridgeport.edu/sed/projects/cs503/Spring_2001/kode/os/memory.htm
https://secure.wikimedia.org/wikibooks/en/wiki/Operating_System_Design/Memory_Management
http://www.technologyuk.net/computing/operating_systems/memory_management.shtml
http://msdn.microsoft.com/en-us/windows/hardware/gg487427
http://download.microsoft.com/download/7/E/7/7E7662CF-CBEA-470B-A97E-CE7CE0D98DC2/mmwin7.pptx

http://members.shaw.ca/bsanders/WindowsGeneralWeb/RAMVirtualMemoryPageFileEtc.htm
https://secure.wikimedia.org/wikipedia/en/wiki/Paging#Windows_NT

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие модели памяти используются в операционных системах?
2. Чем отличается виртуальная память от физической?
3. Что такое страничная модель памяти?
4. Где располагается отображение виртуальной памяти на внешние носители?
5. Как распределяется память в разных Windows-системах?

Глава 4

ПРОЦЕССЫ

4.1. ОБЩИЕ ПОНЯТИЯ

Программа в общем случае — набор инструкций процессора, представленный в виде файла. Для того 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, в каждом из которых он особым образом использует ресурсы, предоставляемые ему операционной системой.

Два основных состояния: выполнение процесса в режиме задачи, когда выполняется собственный программный код, и выполнение в режиме ядра, когда процесс выполняет системные программы, находящиеся в адресном пространстве ядра операционной системы.

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

С точки зрения структур, поддерживаемых ядром ОС, контекст процесса включает в себя следующие составляющие [19]:

- 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 частей.

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

Дескриптор процесса включает в себя системные данные, используемые операционной системой для идентификации процесса. Эти данные применяются при построении таблицы процессов, содержащей информацию обо всех выполняемых в текущий момент времени процессах. Дескриптор процесса включает в себя 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 пределах 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 — это один или несколько стеков, которые используются процессом при его выполнении в режиме ядра.

Для просмотра таблицы процессов может применяться команда ps. Будучи выполненной 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 приведенном выше 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 завершении (RSS), файл терминала (TTY), состояние процесса (STAT), дата старта процесса (START), количество используемого процессорного времени (TIME), полная строка запуска программы (COMMAND).

При работе процессу предоставляется доступ к ресурсам ОС — оперативной памяти, файлам, процессорному времени.

Для распределения ресурсов между процессами и управления доступом к ресурсам используется планировщик задач, входящий в состав ядра операционной системы, а также применяются 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.

4.2. 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 — родительским процессом. Процесс-потомок хранит информацию о родительском процессе в своем дескрипторе. Единственный процесс, не имеющий родительского процесса, — это головной процесс ядра операционной системы — init, имеющий PID, равный 1. Запуск этого процесса происходит при начальной загрузке ядра операционной системы.

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:

- потомку присваивается уникальный идентификатор PID, отличный от родительского;
- значение PPID процесса-потомка принимает 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. 10.3);
- временная 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) идентификаторы группы, запустившей процесс;
- приоритет;
- установки обработчиков сигналов (см. подразд. 10.3.5).

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

- переполнение максимального числа процессов для текущего пользователя. 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. Данное 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 функция fork() 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 */
/* pid>0. Значит, 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 тоже выполнил бы 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. 10.3);
- 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. 4.3).

Другие способы завершения работы процесса будут рассмотрены в следующих подразделах.

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 использована только в функциях, выполняемых процессом-родителем.

В большинстве случаев функция fork() используется совместно с одной из функцией семейства exec...(). При таком совместном использовании 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, то это может быть вызвано следующими причинами:

1. 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.

2. Файл, для которого вызывается функция, не является ни исполняемым, ни обычным файлом или не существует.

3. 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 не должно быть меньше, чем 4096. В реальных системах обычно это число больше на несколько порядков. Для того чтобы узнать это ограничение для системы, можно использовать команду getconf ARG_MAX. Параметр ARG_MAX 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() необязательно:

```
#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). Возвращаемое 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 за-

вершен. 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
```

4.3. СОСТОЯНИЯ ПРОЦЕССА. ЖИЗНЕННЫЙ ЦИКЛ ПРОЦЕССА

На промежутке времени между созданием и завершением процесс находится в различных состояниях в зависимости от наступления некоторых событий в операционной системе.

Сразу же после создания при помощи функции fork() процесс 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 операционной системы, обрабатывающий последнее изменение состояния процесса. Из этого состояния он может перейти в состояние «Выполняется в режиме задачи», в котором будет выполнять уже свой собственный программный код. При каждом системном вызове процесс будет переходить в состояние «Выполняется в режиме ядра» и обратно (рис. 4.1).

Системные вызовы могут выполняться для получения доступа к определенным ресурсам, а ресурсы могут оказаться недоступными — 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, процесс захватывает его, 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».

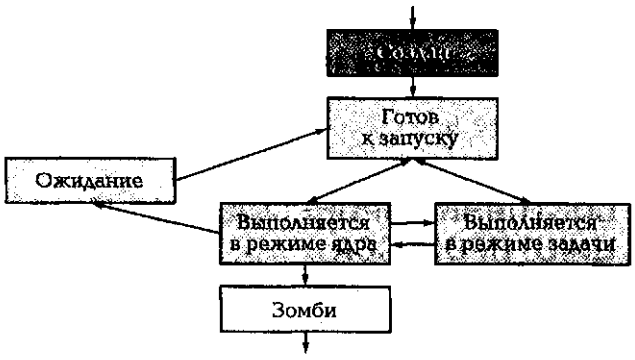

Рис. 4.1. Жизненный цикл процесса

При переключении активного процесса происходит переключение контекста, при этом ядро сохраняет контекст старого процесса и восстанавливает контекст процесса, которому передается управление.

После завершения процесс 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.

4.4. ТЕРМИНАЛ. БУФЕРИЗАЦИЯ ВЫВОДА

По умолчанию системная библиотека ввода/вывода передает данные на устройство терминала не сразу по выполнении 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 поступлении команды от ядра операционной системы или по заполнении 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:

1) полная буферизация — передача данных на физическое устройство терминала выполняется только после полного заполнения буфера;

2) 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);

3) отсутствие буферизации, при этом данные сразу передаются на устройство, без предварительного накопления в буфере вывода.

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. 4.2 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 на рисунке 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 нака-

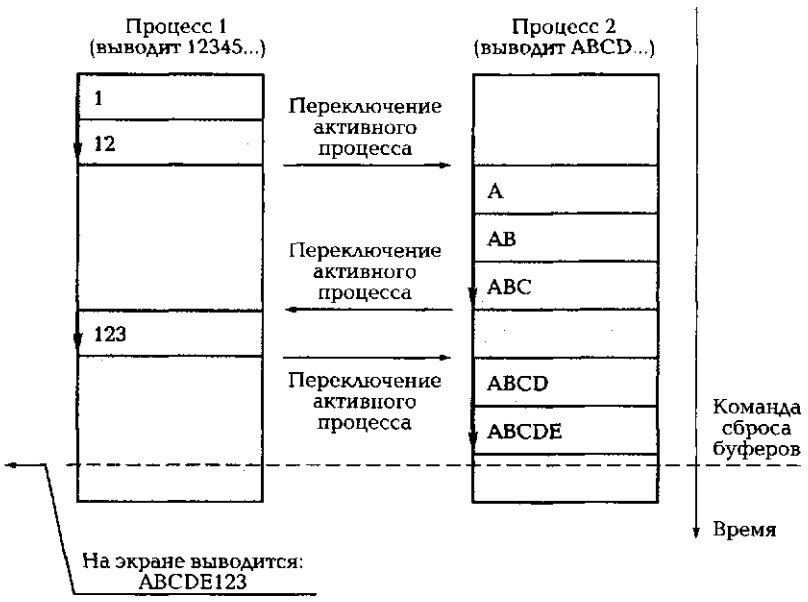

Рис. 4.2. Сброс буферов ввода/вывода параллельно выполняемых процессов

пливаться 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.

Кроме того, существует функция setbuf(), которая позволяет сделать то же самое, что и функция setvbuf(), если требуется отключить буфер или задать новый буфер без изменения типа буферизации. Эта функции имеет следующий интерфейс:

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
```

Соответственно, отключить буферизацию можно следующим образом:

```
setbuf(stdout,(char*)NULL);
```

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие параметры характеризуют процесс в операционной системе?
2. Какие состояния проходит процесс за время своей жизни?
3. Какая команда создает новый процесс?
4. Что происходит при выполнении функции fork()?
5. Какую роль играет системный вызов wait()?
6. Напишите программу, порождающую 10 своих потомков, которые завершаются с кодом возврата, равным номеру своего процесса.

Глава 5

ЗАДАНИЯ

5.1. ЯЗЫКИ УПРАВЛЕНИЯ ЗАДАНИЯМИ

Командный интерпретатор — это программа, запущенная 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 операционной системы, терминал — это 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 операционных системах терминал образован одним файлом, который используется и для чтения, и для записи. Физическое устройство при этом определяется режимом работы с файлом — записываемые в файл данные попадают на экран, 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 выглядит как

```
$
```

Ввод команды завершается нажатием клавиши [Enter], после чего интерпретатор начинает выполнение команды. Например, можно вывести имя файла, соответствующего терминалу. Это делается 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 сценарием.

Язык управления заданиями должен обладать следующими свойствами:

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

5.2. ПАКЕТНАЯ ОБРАБОТКА

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

Работа в диалоговом режиме подразумевает постоянный диалог с пользователем во время выполнения задания. В диалоге с пользователем по мере возникновения 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, по-

скольку при выполнении задания потребуется постоянное присутствие оператора за терминалом. Один из способов снижения нагрузки на оператора в данном случае — составление задания так, чтобы оно выполнялось в пакетном режиме.

В пакетном режиме все необходимые данные и параметры готовятся до начала выполнения задания в виде пакетного файла. Выполнение самого задания производится 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 возникновения, но при этом пользователю 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 — это 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 символы-разделители и различные управляющие коды. Однако можно использовать пробел в тексте параметра: его заключают в двойные кавычки.

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

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

Данная команда позволит узнать ограничения, накладываемые вашей системой. Так, в системе «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 результате выполнения задания, как уже говорилось выше, может быть выведен на экран или в файл. Для облегчения автоматической обработки результата выполнения задания в конце его выполнения формируется код возврата — числовое значение, характеризующее успешность выполнения. Конкретные значения кодов возврата определяются в тексте задания. Доступ к коду возврата задания можно получить непосредственно после завершения при помощи специальных средств командного интерпретатора.

Так, в систему «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, запускаемых под управлением операционной системы Windows, работают именно в диалоговом режиме). Тем не менее пакетный режим выполнения может быть удобен для автоматизации многих рутинных операций. Поэтому практически во всех операционных системах существуют командные интерпретаторы или аналогичные им программные средства, позволяющие выполнять задания в пакетном режиме.

5.3. ОБЩИЕ ПРИНЦИПЫ ЯЗЫКА ИНТЕРПРЕТАТОРА BASH

Синтаксис языка управления заданиями определяется используемым командным интерпретатором. В данном учебнике 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, игнорируются при выполнении задания.

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

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

Синтаксически вызов большинства команд 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 BASH и внешним программам приведен в приложении 2.

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 справочной системой man. Для вызова страницы помощи BASH необходимо ввести «man bash». Описание встроенных команд BASH находится в разделе BUILT-IN COMMANDS.

Для просмотра справочной страницы по любой команде UNIX необходимо вызвать man с параметром — именем команды. Для поиска названий команд и получения краткого их описания можно воспользоваться программой apropos. Например, вызов apropos edit выведет на экран список всех справочных страниц по программам, в названии которых встречается подстрока edit.

5.4. ПЕРЕМЕННЫЕ

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.

5.4.1. Работа со значениями 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. Для того чтобы избежать появления ошибки, существуют различные способы доступа к значению переменной:

- $var — получение значения переменной var или пустого значения, если она не инициализирована;
- ${var} — аналогично $var;
- ${var:-string} — получение значения переменной var, если она определена, или строки string, если переменная var не определена;
- ${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;
- ${var:?string} — получение значения переменной var, если она определена. Если значение переменной var не определено, то выводится строка string и выполнение задания прекращается;

- ${var:+string} — получается значение string, если переменная var определена, или пустое значение, если переменная var не определена [8].

Использование подстановки 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 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.
```

5.4.2. 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 командным интерпретатором, пользователь может только считать их, но не может явно изменить.

В BASH определены следующие встроенные переменные:

1. $? — код возврата последней команды. Возврат из сценария 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;

$1, ..., $9 — при помощи этих переменных производится подстановка значений параметров, указанных в командной строке, вызвавшей задание. Переменной $1 соответствует первый параметр, переменной $9 — девятый.

Если существует необходимость в доступе к параметрам, следующим за девятым, используется команда shift, сдвигающая «окно» из девяти параметров вправо по списку параметров (рис. 5.1). После

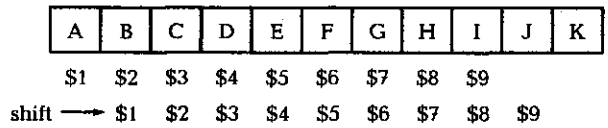

Рис. 5.1. Сдвиг окна параметров

выполнения команды shift второй параметр становится доступным через переменную $1, а десятый — через $9. Количество выполнений команды shift не ограничено. Однако 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.

Именно в связи с этой проблемой невозможно обращение более чем к 9 параметрам командной строки при помощи системных переменных $1, ..., $9: при записи $19 неизвестно, имелся в виду 19-й параметр командной строки или первый, за которым следует текстовая строка «9».

Для решения этой проблемы в новых версиях BASH (2.0 и старше) 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 9 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:

- $* — в этой переменной хранятся все параметры командной строки, переданные заданию. Все параметры в данной переменной представляют собой единую строку, заключенную в кавычки, т. е. $* = "$1 $2 $3 ...";
- $@ — в этой переменной хранятся все параметры командной строки, переданные заданию. Каждый переданный параметр в данной переменной заключен в кавычки, т. е. $@ = "$1" "$2" "$3" ...;
- $0 — в этой переменной хранится имя файла выполняющегося задания. С ее помощью можно организовывать рекурсивный

вызов задания, не зависящий от имени его файла. Иными словами, 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 командах см. приложение 2).

В результате выполнения 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
```

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

5.4.3. 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.

Для того 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 производится командой export. Существует два формата ее вызова:

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

2) 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) наследуются все определенные переменные окружения. При этом командный интерпретатор получает новую копию тех же самых переменных — любое изменение значения переменных в среде командного интерпретатора затрагивает только его собственную среду выполнения, но не изменяет среду выполнения вызывающего командного интерпретатора.

Так, 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, полужирным курсивом — 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. 8.2). Это позволяет использовать значение переменной в любой копии командного интерпретатора, запущенного в ходе сеанса работы пользователя.

Примером такой переменной окружения может служить PATH. Значением этой переменной является список абсолютных имен каталогов, в которых производится 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 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 командного интерпретатора пользователя 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
```

5.4.4. Доступ к значениям переменных

Присвоение значений переменным производится 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.

Для того чтобы при выполнении задания можно было получить доступ к значению переменной, необходимо воспользоваться операцией подстановки «$». При прямом указании в тексте задания имени переменной это имя будет воспринято как строка. Если имя предварить операцией подстановки, то при выполнении задания будет произведена подстановка значения переменной с указанным после «$» 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 вышесказанное, нетрудно догадаться, что присвоение значения одной переменной другой переменной будет записано как

```
var1=$var2
```

5.5. ЗАПУСК ЗАДАНИЯ НА ИСПОЛНЕНИЕ

Для того чтобы запустить задание на исполнение, его нужно передать командному интерпретатору, который будет анализировать текст задания и трансформировать команды задания в системные вызовы операционной системы. 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 перемененные, файловые дескрипторы. Задание может быть передано на исполнение текущему командному интерпретатору — в этом случае сохраняется текущее информационное окружение (рис. 5.2).

Также возможен запуск средствами текущего командного интерпретатора нового процесса командного интерпретатора (рис. 5.3).

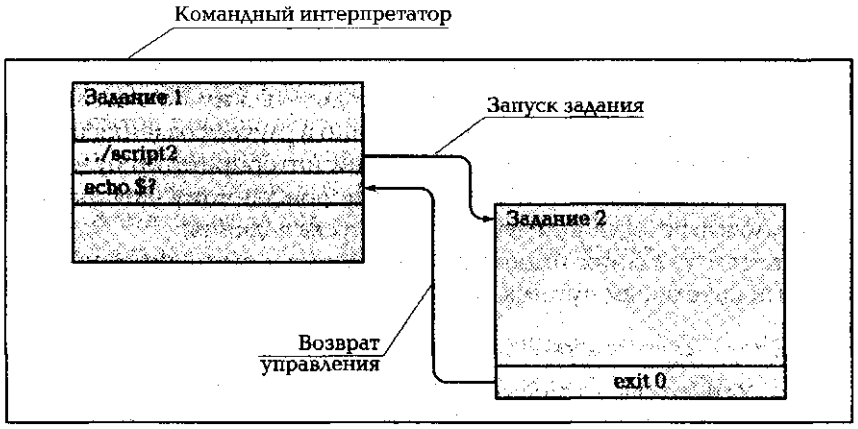

Рис. 5.2. Запуск задания в контексте текущего командного интерпретатора

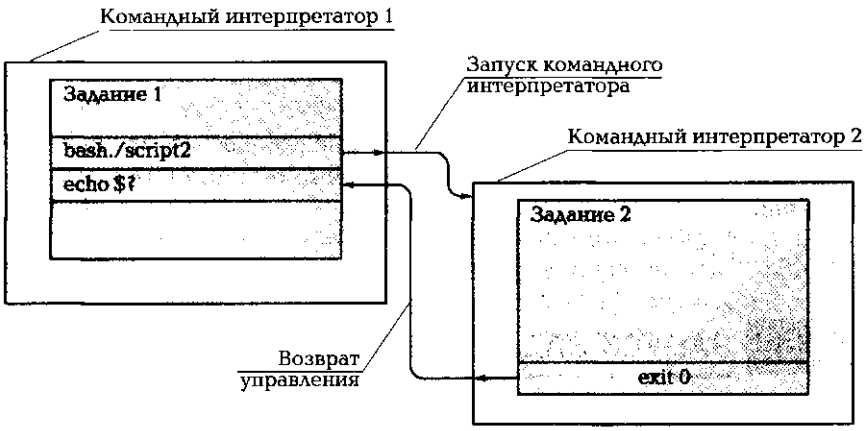

Рис. 5.3. Запуск задания в новом командном интерпретаторе

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.

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:

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

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

Для того чтобы воспользоваться таким методом запуска, файл с заданием должен иметь атрибут «исполнимый».

При таком запуске задания на исполнение запускается новая копия командного интерпретатора, путь к которому указан в заголов-

ке 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;

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

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

При таком методе запуска файл с заданием необязательно должен иметь атрибут «исполнимый». Также допускается отсутствие заголовка в тексте задания — выбор командного интерпретатора осуществляется пользователем, запускающим задание на исполнение, 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;

3) путем запуска при помощи команды exec:

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

При таком 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»;

4) путем запуска в том же командном интерпретаторе:

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

Если перед именем запускаемого задания добавить 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.

Несложно догадаться, что 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.

5.6. 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 — 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 операционной системы. 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. Для того чтобы в ходе выполнения задания выводить пользовательские данные в файл или получать входные данные из файла, ввод/вывод может быть перенаправлен. 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 вывода данных на экран или получения их с клавиатуры не происходит, т. е. файлы полностью заменяют устройства терминала. Управление перенаправлением данных производится при помощи команд перенаправления ввода/вывода.

Для того 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, которые 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, требуется добавлять данные в конец к уже имеющимся в файле. Для этого можно воспользоваться символом перенаправления вывода с добавлением «>>»:

```
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 будет выглядеть следующим образом:

```
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 выглядеть следующим образом:

```
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 потому, что одни и те же действия могут быть применены в совершенно разных заданиях. Например, возьмем два задания — первое, 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 мы применим всего два действия — вывод текстового файла в поток вывода и сортировку входного потока с последующей выдачей его в выходной, мы 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.

5.7. ПОДСТАНОВКА

5.7.1. Подстановка вывода программ

При написании заданий часто возникает необходимость в сохранении данных, выводимых какой-либо программой. Для того чтобы потом можно было сравнительно просто использовать эти данные в ходе выполнения задания, можно сохранить их в переменной задания. Для этого применяются 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. 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.

Например, команда

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

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

5.7.2. Групповые символы

Задача получения списка всех файлов в каталоге, которая была рассмотрена в предыдущем подразделе, может быть более сложной. Пользователю может понадобиться получить список не всех файлов в каталоге, а только файлов, имена которых удовлетворяют определенному критерию, например начинающиеся с буквы А или содержащие не более восьми символов. Для определения такого критерия используется маска имени файла, или просто маска. *Маска* представляет собой текстовую строку, 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 используются 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, последняя 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).

5.8. УПРАВЛЕНИЕ ХОДОМ ВЫПОЛНЕНИЯ ЗАДАНИЯ

5.8.1. 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.

В случае необходимости определения того, как выполняются команды — последовательно или параллельно, — необходимо использовать символы-разделители «;» и «&».

5.8.2. 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 поступаем и в случае параллельного выполнения:

```
команда1 & команда2
```

Необходимо отметить, что символ-разделитель «;» применяется обычно 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 команда будет выполнена в фоновом режиме, а выполнение будет передано команде задания, находящейся на следующей строке. Если запуск команды в фоновом режиме был произведен из командной строки, то команда начнет свое выполнение в фоновом режиме, а управление будет сразу передано командному интерпретатору.

5.8.3. 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 ||.

Для того 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. Разделители && и || могут рассматриваться 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 выполнения 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 выполнения второй команды также не требуется.

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

5.8.4. Объединение потоков вывода программ

С точки зрения условного выполнения команд фигурные скобки { } являются символами группировки. Однако они могут быть использованы для того, чтобы объединить потоки 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.

5.8.5. 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". Но 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.

5.8.6. 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* — вы-

ражение, определяющее эталон, при сравнении с которым выполняется отбор, объект, проверяемый на соответствие эталону, и степень соответствия объекта эталону.

Например, в выражении К > 2 переменная К 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, либо 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 вариантах 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. Формат некоторых выражений для проверки свойств файлов, строк и чисел приведен ниже. Далее в тексте по мере изложения материала будут приведены дополнительные виды проверок, выполняемых при помощи команды test.

- -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 — число1 меньше числа2;
- число1 -le число2 — число1 меньше или равно числу2;
- число1 -gt число2 — число1 больше числа2;
- число1 -ge число2 — число1 больше или равно числу2;
- -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 могут объединяться 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 приведенном фрагменте как разделитель команд, поскольку с точки зрения синтаксиса BASH if и then — это разные команды, которые необходимо либо разделять символом «;», либо размещать в разных строках файла задания.

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

Например, следующий 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 поскольку передача параметров в задание — позиционная, то 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 как раз 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 происходят сдвиг окна параметров и их перенумерация (см. подразд. 5.1):

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

Для организации цикла по значениям заданного списка используется конструкция for ... in ... do ... done, общий синтаксис которой приведен ниже:

```
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, — выводит все параметры, переданные заданию. В качестве списка здесь используется встроенная переменная BASH, содержащая все параметры командной строки задания, указанные через разделитель (см. подразд. 4.2):

```
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 приведенном только что примере применяется подстановка значения переменной. Точно так же можно использовать постановку 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
```

5.9. Языки управления заданиями в операционных системах семейства Windows

5.9.1. Командный интерпретатор в Windows

Так же как и Linux, операционные системы семейства Windows поддерживают работу с интерфейсом командного интерпретатора. Во многом его свойства были унаследованы от командного интер-

претатора операционной системы MS DOS и дополнены особенностями, присущими операционным системам семейства UNIX/Linux. В данном подразделе мы рассмотрим некоторые свойства этой подсистемы в операционных системах Windows на основе командного интерпретатора, реализованного в системе Windows XP.

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

После запуска командного интерпретатора появляется окно, содержащее стандартное приглашение к работе. В операционных системах 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 операционной системы и проприетарных правах фирмы Microsoft. Последняя строка представляет собой стандартное приглашение к вводу команд или приему заданий от пользователя.

Ввод команд в данном интерпретаторе аналогичен данному в операционных системах Linux. Например, можно вывести версию операционной системы следующим образом:

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

5.9.2. Пакетная обработка в Windows

Механизм и средства реализации работы с заданиями в основном сходны с подходом, используемым в операционных системах Linux. Задания также оформляются в виде текстовых файлов, содержащих перечень команд, которые должны быть выполнены командным интерпретатором. Сами файлы должны иметь расширение .bat или .cmd. А исполнение, механизмы передачи параметров и результат работы командных сценариев в Windows такой же, как 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-подобных системах, определяется максимально допустимой длиной строки. Для операционных систем Windows 2000 и Windows NT 4.0 максимально допустимая длина строки составляет 2 047 символов, а для систем Windows XP и более поздних данное значение составляет 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 данным параметром выводится справка о ее предназначении и параметрах, которые могут быть переданы данной команде.

5.9.3. 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 операционной системы 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 сценария. Для определения новой переменной или изменения ее текущего значения также используется команда set:

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

Для получения текущего значения переменной недостаточно указать просто имя переменной. Помимо этого, 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:

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 следующие строки:

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

Как видно 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 получить 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 ним 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 присвое-

ния числовых значений используется конструкция set /a. В математических выражениях могут использоваться только целые числа в диапазоне от –231 до 231 – 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)
```

Для того 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. Но 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, ограниченной данными командами, будут недействительны после выполнения команды endlocal. Рассмотрим пример использования этих команд:

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 данного сценария на консоль будут выведены следующие сообщения:

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

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

Так же как и в Linux, 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 данной команды переменная %1 сопоставляется со вторым параметром, %2 — с третьим и т.д.

Для получения списка всех параметров используется 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 служат для выделения из параметров командной строки имен файлов, каталогов, времени создания соответствующих файлов и т.д. Для применения модификаторов используется последовательность %~, за ней следует модификатор и позиционный номер параметра, к которому данный модификатор применяется. Список модификаторов параметров командной строки приведен в табл. 5.1.

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.

Таблица 5.1. Модификаторы параметров командной строки

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

5.9.4. Ввод/вывод. Конвейерная обработка

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

Так же как и в Linux, эти файлы имеют стандартные имена и за ними зафиксированы стандартные файловые дескрипторы, которые автоматически открываются при выполнении любого консольного приложения. Эти виртуальные файлы представляют стандартный 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 в файлы потоки ввода/вывода, ассоциированные со стандартными потоками. Для этого ис-

пользуются операции перенаправления ввода/вывода. Синтаксис и семантика данных операций аналогичны используемым в Linux. Поэтому не будем подробно останавливаться на их пояснении, а сведем эти операции в единую табл. 5.2.

Так же как и в Linux, эти операции можно комбинировать и использовать сразу несколько операций в команде. Например, для того чтобы перенаправить все данные, выводимые некоторой ко-

Таблица 5.2. Операции перенаправления ввода/вывода

Операция	Описание
>	*команда1 > файл* Поток стандартного вывода команды связывается с файлом или устройством. Если данный файл не существовал, то он создается. Если он существовал, то файл перезаписывается
>>	*команда1 >> файл* Поток стандартного вывода команды связывается с файлом или устройством. Если данный 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 данный 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 данный 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
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

мандой, 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.

5.9.5. 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.

Средства последовательного и условного выполнения команд очень похожи на соответствующие средства, определенные в Linux. Краткое описание этих средств приведено в табл. 5.3.

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

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

В данном примере 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, то на консоль выводится сообщение «Unable to get list of logs», а в файл list.txt записывается системная информация о возникшем сбое. После окончания работы данных команд запускается приложение, ассоциированное с расширением .txt (например, notepad), и ему передается файл list.txt в качестве параметра.

Таблица 5.3. Механизмы 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. Исполняется первая команда, если она завершилась успешно (т. е. вернула 0 в качестве кода возврата), то исполняется вторая
||	*команда1 || команда2* Используется для условного выполнения нескольких команд. Исполняется первая команда, если она завершилась неуспешно (т. е. вернула код, отличный от 0), то исполняется вторая
()	(*команда1 & команда2*) Используется для группировки последовательности команд
; или ,	*команда1 параметр1;параметр2* Используется для разделения параметров, передаваемых команде в командной строке

Как и во многих языках 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 только в том случае, 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. В противном случае интерпретатор сообщит о наличии ошибки в коде сценария.

В качестве выражения могут использоваться несколько конструкций в зависимости от целей условного оператора. Список этих выражений приведен в табл. 5.4.

Для организации циклов интерфейс командной оболочки Windows предоставляет единственный оператор — for. Но так же,

Таблица 5.4. 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 неуспехом
[not] *строка1==строка2*	if %1==%VAL% (echo Strings are the same) else (Strings are different) Используется для сравнения двух строк на равенство/неравенство. Если используется 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
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 использоваться 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

Окончание табл. 5.4

Логическое условие	Описание
if cmdextversion *номер*	if cmdextversion 2 echo The operation is supported Используется для проверки текущей версии расширения команд. Возвращает значение истины, 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 проверки факта существования специфицированной переменной

как и условный оператор, оператор циклов имеет несколько различных форм, позволяющих использовать его для разных целей.

Общая форма записи оператора for выглядит следующим образом:

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

Тип элементов, включенных в список *множество*, определяется опциональным параметром оператора for. Список параметров и типы элементов перечислены в табл. 5.5.

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
```

Таблица 5.5. 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 использованием 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

После выполнения данной 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 применяться для досрочного прерывания выполнения сценария. Для этого используется следующая форма записи:

```
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 данном 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]
```

В этом случае *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
```

Но команда call используется не только для запуска внешних командных файлов, но и для организации вызова функций и процедур. Для этого применяется 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 используется 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 следующий сценарий:

Fibonacci.bat:

```
@echo off
rem Отключим 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 вычисленной 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 результат 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 данного 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. По этой причине приведенный выше пример отличается от классической реализации чисел Фибоначчи.

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

5.9.6. Командная оболочка PowerShell

Следует отметить, что инструменты командной строки операционной системы 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.

По этой причине фирма Microsoft приняла решение о разработке новой оболочки с интерфейсом командной строки и встроенным языком разработки сценариев. Данная оболочка получила название PowerShell. Можно сказать, что данная оболочка развивает свойства таких оболочек, как Cmd.exe/Command.com, BASH, WSH, Perl и др. Эта оболочка интегрирована с платформой .NET Framework и может использоваться в операционных системах Windows XP SP3, Windows Server 2003 SP2, Windows Vista SP1, Windows Server 2008, Windows 7 и Windows 8.

PowerShell предоставляет полный доступ к COM и WMI, позволяя администраторам управлять как локальной, так и удаленной системой.

Каждая команда в PowerShell представляет собой так называемый *командлет*, который представляет собой специализированный класс .NET, реализующий заданную операцию. Несколько командлетов можно объединить в сценарий или в независимо выполняемую бинарную программу.

Windows PowerShell также предоставляет специальный механизм, позволяющий сторонним приложениям использовать командлеты PowerShell. Например, Microsoft Exchange Server 2007 использует этот механизм для того, чтобы предоставить администраторам свой механизм управления в среде PowerShell. При этом у администратора системы появляется возможность управлять состоянием базы данных через интерфейс командлетов.

PowerShell может выполнять четыре вида команд:

1) командлет, который представляет собой специальные .NET библиотеки, загружаемые и выполняемые ядром оболочки;

2) сценарии PowerShell (имеют расширение .ps1);

3) функции PowerShell;

4) бинарные программы.

В качестве примера команд PowerShell можно рассмотреть следующие примеры.

Чтобы просмотреть все предопределенные командлеты, достаточно ввести команду Get-Command. Для получения справки по PowerShell необходимо ввести команду Get-Help.

Для того чтобы завершить все процессы, начинающиеся с символа p, достаточно ввести следующую команду:

```
PS> Get-Process p* | Stop-Process
```

Чтобы проверить, какой процесс завершился, или приостановить выполнение сценария до тех пор, пока процесс не остановится, вводят следующее:

```
PS> $processToWatch = Get-Process Notepad
PS> $processToWatch.WaitForExit()
```

Используя механизм синонимов, этот сценарий можно написать гораздо короче:

```
PS> (ps notepad).WaitForExit()
```

У командлета Get-Process существует синоним ps. Используя этот синоним, можно упрощать сценарии и делать их более похожими на сценарии BASH, так как многие синонимы полностью совпадают с соответствующими командами в этой оболочке. Для получения списка всех синонимов достаточно выполнить командлет Get-Alias.

Поскольку PowerShell построена на основе .NET Framework и наследует многие его свойства, то становится возможным использовать и классы и методы из .NET. Например, если в сценарии необходимо вычислить корень из некоторого числа, то для этого можно вызвать статический метод Sqrt():

```
PS> [System.Math]::Sqrt(16)
4
```

В целом можно сказать, что данная оболочка существенно расширяет возможности администраторов, позволяя писать достаточно простые сценарии с использованием всей мощи .NET и с доступом ко всем свойствам системы без использования графических оболочек, что очень удобно при автоматизации рутинных действий.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие свойства присущи языку описания заданий в операционной системе?
2. Чем отличаются пакетный и диалоговый режимы работы в операционной системе?
3. Как передаются параметры командной строки решаемой задаче?
4. Как можно получить доступ к более чем девяти параметрам командной строки?
5. Какие средства перенаправления ввода/вывода обеспечиваются операционной системой?
6. Как выполняется копирование переменных задания в среду выполнения задачи? Какие ограничения при этом существуют?
7. Как пользователь может создавать подпрограммы в языке управления заданиями?

Глава 6

ПОЛЬЗОВАТЕЛИ СИСТЕМЫ

6.1. ВХОД В СИСТЕМУ

В предыдущей главе были показаны основные способы взаимодействия с операционной системой, доступные пользователю. При этом не следует забывать о том, что операционная система UNIX, о которой в основном идет речь в этом учебнике, — многопользовательская. В этой главе речь пойдет именно о том, как именно предоставляется многопользовательская поддержка, т. е. в основном про управление сеансами и защиту данных.

Для того 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 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 выглядит приблизительно таким образом:

```
login:
```

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

```
password:
```

Если введенные пользователем 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 применяется командный интерпретатор BASH.

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

В роли пользователей могут выступать запускаемые программы с определенным набором прав. Такие программы запускаются от имени конкретного пользователя, который в таком случае именуется *псевдопользователем*. Например, типичный псевдопользователь в UNIX-системах — пользователь с учетным именем mail. От лица этого пользователя происходят управление очередями сообщений электронной почты, доставка сообщений и пересылка их адресатам. Обычный пользователь, как правило, не может войти в систему под именем псевдопользователя, поскольку 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 функцию системного администратора. Этот пользователь имеет исключительное право доступа ко всем ресурсам, предоставляемым операционной системой. В UNIX-системах этот пользователь обычно имеет учетное имя root.

6.2. ДОМАШНИЕ КАТАЛОГИ ПОЛЬЗОВАТЕЛЕЙ

Каждому пользователю, работающему в операционной системе, 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, для того чтобы перейти в свой домашний каталог, достаточно ввести команду

```
cd ~
```

Для перехода в подкаталог dirname домашнего каталога необходимо ввести команду

```
cd ~/dirname
```

Для того 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. 6.4).

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 крайне 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.

6.3. ИДЕНТИФИКАЦИЯ ПОЛЬЗОВАТЕЛЕЙ

Каждый пользователь системы имеет уникальное в пределах данной системы учетное имя (логин). Однако имя присваивается пользователю только для его удобства — операционная система различает пользователей по их *уникальным идентификаторам* — UID (User IDentifier). Идентификатор UID представляет собой целое число, большее или равное нулю. UID 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 двоеточием «:». Последовательность атрибутов следу-

ющая: 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 нескольких 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 — это множество пользователей, задаваемое в виде списка. Объединение пользователей в группы обычно происходит по принципу разграничения задач, выполняемых пользователями. Так, в отдельную группу обычно выделяются администраторы системы.

> В системе «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
```

6.4. ПРАВА ДОСТУПА К ФАЙЛАМ И КАТАЛОГАМ

Во многих операционных системах существуют средства ограничения доступа к файлам. В однопользовательских ОС обычно ограничивается доступ к файлам ядра операционной системы (например, установка атрибута «скрытый» в DOS).

В многопользовательских ОС доступ ограничивается при помощи определения прав доступа того или иного пользователя к файлам. Права доступа определяют возможность выполнения той или иной операции над файлом.

Вообще говоря, термин «права доступа к файлу» не совсем корректен. Права доступа определяются для наборов данных, хранимых на диске. Для всех файлов, связанных с этим набором данных, 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. 6.1).

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 подразумевается 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.

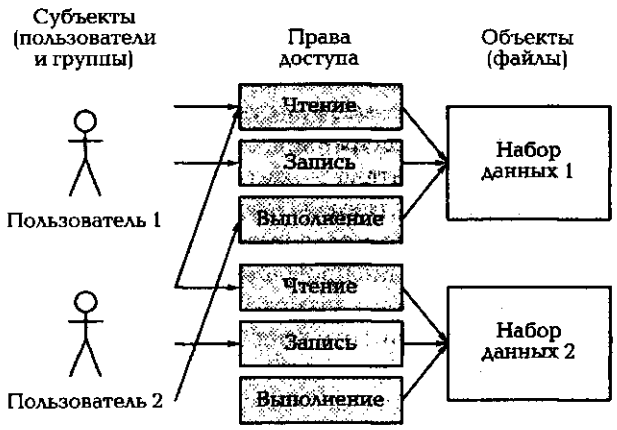

Рис. 6.1. Задание прав доступа при помощи троек субъект—объект—право доступа

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 20:03 tkConfig.sh
```

В третьей колонке выводится учетное имя пользователя-владельца файла, в четвертой — учетное имя группы-владельца. Первая колонка содержит информацию об атрибутах файла, задающих права доступа к файлу.

Первый символ в колонке задает тип файла. Для каталогов первый символ имеет значение d, для обычных файлов — прочерк. Существуют и другие типы файлов.

Символы со второго по десятый 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 имеет права 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. 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 применит вызовы Open и Write. Право на выполнение программы, установленное на исполняемый файл программы, дает возможность пользователю передать управление программе, при этом операционная система 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 даст возможность перейти в данный 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. Здесь нужно 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.

6.4.1. Задание прав доступа к файлам и каталогам

Права доступа к файлам и каталогам могут меняться с течением времени. Это может быть связано, например, с тем, что старый владелец файла передал права на его изменение новому владельцу, или с 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 работы находится в общей базе преподавателя, он доступен только преподавателю; как только файл с вариантом передается студенту, он должен стать доступным ему.

Для того чтобы дать 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 или группа-владелец файла 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 будет выглядеть 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 следующим образом:

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

Существует альтернативный способ задания прав доступа к файлу — числовой. Для того 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, команда

```
chmod 764 file.txt
```

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

В системе «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).

Для того чтобы ограничить доступ пользователей к основным компонентам системы (сценариям и каталогам), их владельцем назначается пользователь devel (разработчик системы):

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

При этом запись в данный каталог будет ограничиваться установленными правами:

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

Иначе говоря, все пользователи получат доступ на чтение и выполнение сценариев, 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 того, чтобы ограничить доступ студентов к каталогам контрольных работ.

Так, каталог /check/teacher должен иметь код доступа 700 и владельца 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
```

6.4.2. 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 недоступны в требуемом режиме доступа. Поэтому 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 главе 5. Команда test проверяет наличие того или иного права доступа у пользователя, от лица которого выполняется задание (текущего пользователя). Для этого применяются следующие параметры команды:

- -r <файл> — текущему пользователю разрешен доступ на чтение файла;
- -w <файл> — текущему пользователю разрешен доступ на запись в файл;

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

Например, для того 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. Какие параметры характеризуют пользователя в операционной системе?
2. Как начинается сеанс работы пользователя? Почему важно корректно заканчивать сеанс работы?
3. Какие виды ограничений доступа к файлам и каталогам предоставляются операционной системой?
4. Как задаются, изменяются и проверяются права доступа пользователя по отношению к файлам и каталогам?
5. В чем различия прав доступа на чтение у файлов и каталогов?
6. Напишите командный файл, который выводит список файлов, находящихся в домашнем каталоге пользователя и недоступных ему по чтению.

Глава 7

ФАЙЛЫ ПОЛЬЗОВАТЕЛЕЙ

7.1. СТАНДАРТНАЯ СТРУКТУРА СИСТЕМЫ КАТАЛОГОВ UNIX И WINDOWS

Для хранения собственных данных в UNIX-подобных операционных системах используется стандартная структура каталогов, показанная на рис. 7.1.

В зависимости от реализации конкретной операционной системы возможны некоторые отличия от приведенной структуры, например в операционных системах SunOS и Solaris присутствует каталог /opt.

Каталог /bin служит для хранения базовых системных утилит. Как правило, выполнение этих утилит возможно даже в случае отсутствия основных системных библиотек. В каталоге /boot хранятся файлы ядра операционной системы и данные, необходимые для загрузки. Каталог /dev содержит специальные файлы, при помощи которых происходит обращение к устройствам (см. подразд. 7.2). 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. следующую главу). Каталог /lib предназначен для хранения системных библиотек, а /lib/modules — для хранения модулей ядра операционной системы, 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 содержится программное обеспечение, входящее в комплект поставки операционной системы и не относящееся к системному. Назначение каталогов /usr/bin, /usr/sbin, /usr/lib аналогич-

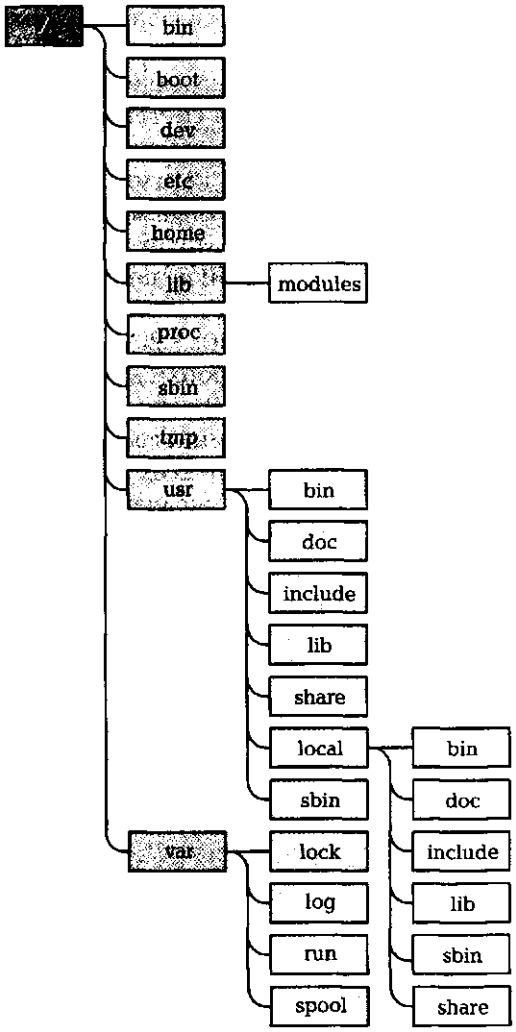

Рис. 7.1. Структура каталогов типичной UNIX-системы

но назначению соответствующих каталогов верхнего уровня. В каталоге /usr/doc хранится документация на программное обеспечение, входящее в комплект поставки ОС, в каталоге /usr/share хранятся разделяемые файлы данных. Каталог /usr/include прежде 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 неразделяемым 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 содержатся 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.

В операционных системах семейства Windows XP структура каталогов имеет вид, показанный на рис. 7.2.

В каталоге Profiles хранятся настройки для каждого пользователя, зарегистрированного в системе. В папке Administrator хранятся настройки для пользователя — администратора системы, а в папке All Users — настройки, общие для всех пользователей. В папке Default User хранятся настройки по умолчанию, которые применяются для всех вновь создаваемых пользователей.

В каталоге Program Files обычно хранятся программы, устанавливаемые администраторами или пользователями системы.

В каталоге Windows хранятся основные настройки операционной системы, а также основные исполняемые файлы операционной системы. В подкаталоге Fonts имеются все шрифты, установленные в системе, в папка Help — файлы справочной системы. В папках system и system32 хранятся основные системные утилиты, апплеты

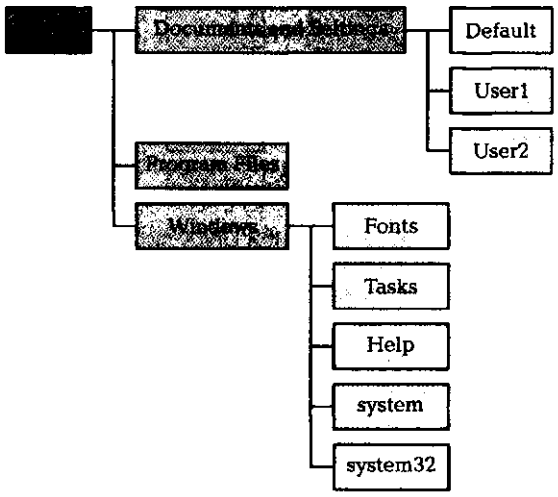

Рис. 7.2. Структура каталогов типичной системы Windows XP

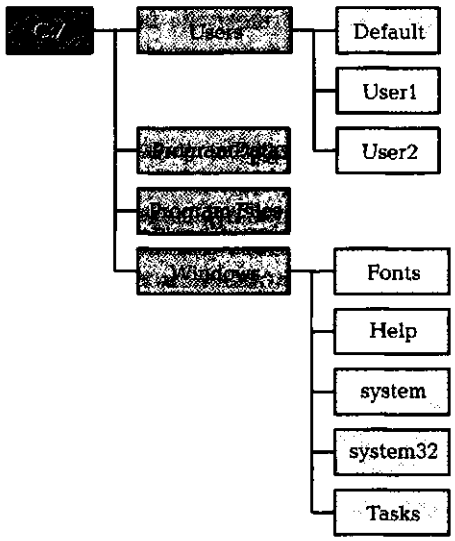

Рис. 7.3. Структура каталогов типичной системы Windows Vista и Windows 7

панели управления, предустановленные с операционной системой утилиты (калькулятор, просмотр символов и т. д.) и прочие утилиты и программы.

В операционных системах семейства Windows Vista и Windows 7 структура каталогов изменилась по сравнению с Windows XP (рис. 7.3).

Вместо каталога Profiles используется каталог Users. Папка All Users пропала, вместо нее в корневом каталоге загрузочного диска появилась папка ProgramData. А в каталоге Users для сохранения совместимости с предыдущими версиями появилась символическая ссылка с именем All Users, указывающая на папку ProgramData. Папка Default теперь используется вместо папки Default User, а для совместимости папке Users создана ссылка Default User на папку Default.

В остальном структура файловой системы не претерпела существенных изменений и сохранила свою прежнюю организацию.

7.2. ТИПЫ ФАЙЛОВ

Файловая система может содержать файлы, различные по своему назначению. Один из вариантов определения назначения файлов был рассмотрен выше — это определение для исполняемых

файлов права доступа «исполняемый» (x). При запуске таких файлов, как программы, операционная система 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 texmf и tk8.4 имеют атрибут d, который означает, что этот файл является каталогом (о различии между файлами и каталогами см. главу 2):

```
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 20:03 tkConfig.sh
```

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

Необходимо заметить, что для обработки файлов разных типов ОС использует разные наборы системных вызовов, предназначенные для работы с каждым поддерживаемым типом файлов. В главе 2 эти различия были проиллюстрированы на примере различий между системными вызовами для работы с файлами и каталогами.

Обычный файл. Данный тип файлов наиболее распространен. Операционная система рассматривает эти файлы как последовательность байтов, вся обработка их содержимого производится прикладными программами.

Каталог. Каталоги — это основное средство задания иерархической структуры расположения файлов. С точки зрения внутренней структуры каталог — это файл специального вида, в котором перечислены имена содержащихся в нем файлов и ссылки на блоки диска, содержащего метаданные соответствующих этим файлам блоков данных.

Символическая ссылка. В главе 2 учебника уже рассматривались жесткие ссылки как способ именования одного набора данных несколькими именами. Каждая жесткая ссылка является элементом каталога, определяющим имя файла, который соответствует набору данных, и ссылку на блок метаданных набора данных.

В отличие от жесткой ссылки символическая ссылка представляет собой файл специального вида, в котором хранится строка — имя файла (с полным или относительным путем), на который указывает

символическая ссылка. Если удалить файл, имя которого хранится в символической ссылке, то символическая ссылка будет неверна и станет указывать в «пустоту».

Символьное устройство. При помощи файлов символьных устройств программы пользователя могут обращаться к различным устройствам, поддерживаемым операционной системой. Для того чтобы данные, записываемые в файл устройства, были ему реально переданы, необходимо, чтобы ядро операционной системы поддерживало работу с устройством этого типа. Если устройство не поддерживается соответствующим драйвером в ядре операционной системы, файл устройства все равно может существовать на диске. Однако 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 операционной системой 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 фиксированного размера. Примером таких устройств может служить раздел жесткого диска: данные, записываемые на диск, поступают блоками 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 потом 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 заключается в том, что при помо-

щи 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.

Для того 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.

7.3. МОНТИРОВАНИЕ ФАЙЛОВЫХ СИСТЕМ

Все файлы и каталоги, доступные пользователю UNIX-системы в ходе его сеанса работы, структурированы при помощи единой иерархической системы каталогов. Если рассматривать эту систему с точки зрения ее логической структуры — от пользователя оказываются скрытыми вопросы физического размещения файлов на накопителях, — единая структура каталогов может объединять файлы, находящиеся на разных дисках. 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, хранящиеся на определенном физическом носителе, называется *точкой монтирования*, а сам процесс, после которого содержимое диска становится доступным пользователям операционной системы, — *монтированием диска*.

Так, на рис. 7.4 показаны три дисковых накопителя — hda1, hda2 и sda1, которые смонтированы 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 разные файловые системы, но при этом все равно объединяться в единую файловую систему. Например, стандартной файловой системой для UNIX-подобной ОС Linux является система ext2, а стандартной файловой системой для CD-дисков служит ISO 9660.

CD-приводы и другие сменные накопители — основные типы устройств, требующие монтирования в процессе работы. Для того чтобы получить доступ к файлам, находящимся на CD-ROM, его необходимо не только вставить в привод, но и смонтировать. Только после этого файлы на диске станут доступными для использования.

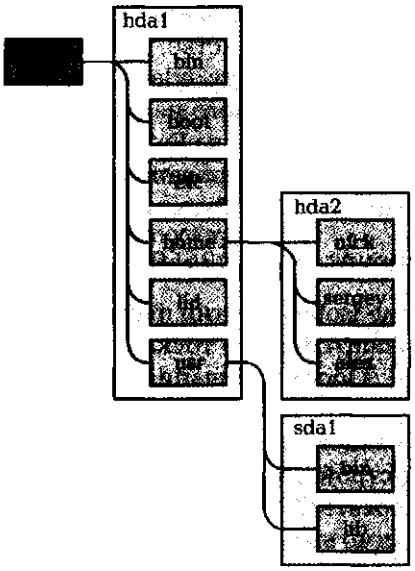

Рис. 7.4. Пример распределения структуры каталогов по физическим носителям

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 операционной системы Linux (без установленных расширений devfs и подобных) следующие:

- IDE-накопители имеют имена, начинающиеся на hd; SCSI-накопители — начинающиеся на sd;
- для IDE-накопителей указываются 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. Иначе говоря, 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 1 — 4 задают первичные разделы, а 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 диска CD-ROM, 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, для того 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 операционной системы 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.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие каталоги UNIX обычно хранят исполняемые файлы?
2. Чем различаются структуры файловых систем Windows XP и Vista?
3. В чем различие файлов символьных и блочных устройств?
4. Что такое точка монтирования?
5. Что такое именованный канал? Как он используется?

Глава 8

УПРАВЛЕНИЕ ПОЛЬЗОВАТЕЛЯМИ

8.1. 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. Для того 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 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 (обычно /bin/bash).

Для того 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. Для того 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
```

Современные операционные системы Windows также являются многопользовательскими, а значит, в них можно создать более одной учетной записи. Для создания, просмотра или удаления пользователей в Windows можно применять одну из команд net user:

```
net user [логин {пароль | *} /add [ключи] [/domain]]
net user [логин [пароль | *] [ключи]] [/domain]
net user [логин [/delete] [/domain]]
```

Первая команда предназначена для добавления нового пользователя в систему, вторая — для изменения пароля или свойств уже существующего пользователя, а третья — для удаления уже существующей учетной записи.

Если при вводе из этих команд вместо пароля ввести символ «*», то перед выполнением команды появится приглашение с просьбой ввода пароля. Причем пароль, набираемый в этом режиме, не отображается на экране, в отличие от ситуации, когда пароль вводится как один из аргументов командной строки команды net user.

В качестве дополнительных опций данных операций могут применяться следующие ключи:

- /active:{no | yes} — включает (yes) или отключает (no) указанную учетную запись;
- /expires:{{дата} | never} — устанавливает дату истечения срока активности заданной учетной записи или сообщает, что данная учетная запись никогда не истекает;

- /homedir:{путь} — устанавливает в качестве домашнего каталога заданный путь, причем этот путь должен существовать;
- /passwordchg:{yes | no} — разрешает (yes) или запрещает (no) пользователю самому менять пароль в своей учетной записи;
- /passwordreq:{yes | no} — требует (yes) использовать пароль для заданной учетной записи или разрешает (no) использовать учетную запись без пароля;
- /times:{время | all} — устанавливает время, в течение которого пользователь может работать с системой.

Если изменяется пароль учетной записи, то минимальная длина вновь устанавливаемого пароля должна быть не меньше, чем длина, установленная с помощью команды net accounts /minpwlen.

8.2. ФАЙЛЫ ИНИЦИАЛИЗАЦИИ СЕАНСА ПОЛЬЗОВАТЕЛЯ

В UNIX и Linux 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 используемого 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, который содержит задания, специфичные для каждого отдельного файла.

Общесистемный файл инициализации сеанса имеет имя /etc/profile. Он 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:/usr/X11R6/bin
```

```
export LD_LIBRARY_PATH=/lib:/usr/lib:/usr/local/lib
export PS1="$"
```

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

Пользовательский файл инициализации сеанса имеет имя .profile или .bash_profile, 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.

В операционных системах Windows для пользователей обычно не устанавливаются файлы инициализации сеанса. Но есть возможность задать такие файлы для отдельных пользователей. Для этого с командой net user нужно использовать ключ /scriptpath:<путь>. Здесь <путь> — путь к файлу инициализации. При этом на такие

пути накладываются определенные ограничения. Например, этот путь должен быть не абсолютным путем, а путем относительно каталога %systemroot%\System32\Repl\Import\Scripts. Можно создать новую учетную запись и установить для нее файл инициализации сеанса следующим образом:

```
NET USER user password /ADD /HOMEDIR:\\Server_05\
/scriptpath:logon.cmd /DOMAIN
```

При этом будет создан пользователь user с паролем password, домашним каталогом \\Server_05\, и пользователь будет создан в том домене, в который входит компьютер.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие основные задачи решаются в процессе управления записями о пользователях операционной системы?
2. Какую роль играет объединение пользователей в группы?
3. Какие действия выполняются в командном файле инициализации сеанса работы пользователя?
4. В чем заключается специфика файлов инициализации системы Windows?

Глава 9

ПРИКЛАДНОЕ ПРОГРАММИРОВАНИЕ ПОД UNIX И WINDOWS

9.1. ЗАГОЛОВОЧНЫЕ ФАЙЛЫ

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 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 операционной системы — к системным вызовам. В случае необходимости применения 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 применяться 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 лишь заголовочные файлы, определяющие формат системных вызовов, и параметры командной строки компилятора.

В UNIX и Linux 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, которые вызвал 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.

В табл. 9.1 перечислены названия основных заголовочных файлов в операционных системах UNIX и Linux, которые содержат определения системных вызовов и библиотечных функций, и крат-

Таблица 9.1. Основные заголовочные файлы, содержащие системные вызовы и библиотечные функции в UNIX/Linux

Заголовочный файл	Содержание заголовочного файла
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 операционной системы — 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
regex.h	Функции и типы данных для работы с регулярными выражениями. Регулярные выражения — расширенный вариант символов подстановки * и ? (более подробно см. главу 5)

Окончание табл. 9.1

Заголовочный файл	Содержание заголовочного файла
signal.h	Функции и типы данных для обработки сигналов (более подробно см. главу 10)
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. главу 10)
sys/sem.h	Функции и типы данных для работы с семафорами (см. главу 10)
sys/types.h	Типы данных для работы с системными таблицами ядра

кие комментарии относительно содержимого этих файлов. Имена файлов указаны относительно каталога /usr/include.

В операционных системах Windows, в отличие от операционных систем UNIX или Linux, система разработки на языке C недоступна по умолчанию, она требует дополнительной установки. Существует целый ряд систем разработки под ОС семейства Windows, в частности, есть семейство систем разработки Microsoft Visual Studio, включающих в себя компилятор языка C. Также существует несколько разновидностей системы компиляции gcc, распространенной в Linux, адаптированной для Windows, например идущие в составе MinGW или cygwin.

Из-за таких особенностей систем компиляции в Windows стандартные заголовочные файлы недоступны изначально, они устанав-

ливаются вместе с выбранной системой разработки. Расположение этих файлов также зависит от того, в какой каталог была установлена выбранная система разработки.

Кроме того, для операционных систем Windows имеется специальная библиотека Windows SDK (Software Development Kit). Эта библиотека разрабатывается и поддерживается в фирмой Microsoft и призвана поддерживать в актуальном состоянии все системные библиотеки и заголовочные файлы, необходимые для разработки приложений под Windows. На момент написания этих строк последней актуальной версией SDK была версия Microsoft Windows SDK for Windows 7 and .NET Framework 4 версии 7.1. В составе Windows SDK идет полноценная среда разработки Microsoft Visual Studio 2010.

В табл. 9.2 перечислены названия основных заголовочных файлов в операционных системах Windows, которые содержат определения системных вызовов и библиотечных функций и краткие комментарии относительно содержимого этих файлов.

Таблица 9.2. Основные заголовочные файлы, содержащие вызовы WinAPI в Windows

Заголовочный файл	Содержание заголовочного файла
windows.h	Главный заголовочный файл, содержащий объявления всех функций Windows API, определения всех специфичных для Windows макросов, типов данных и т. д.
excpt.h	Файл, содержащий декларации для обработки исключительных ситуаций
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 операционной системы — 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.

Окончание табл. 9.2

Заголовочный файл	Содержание заголовочного файла
windef.h	Различные макросы и типы
signal.h	Функции и типы данных для обработки сигналов
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.
WinSock.h	Функции и типы данных для работы с сокетами

9.2. 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) задается ключами запуска компилятора и сборщика.

Для компиляции и сборки простых программ достаточно вызывать только компилятор gcc, который далее автоматически запустит сборщик ld.

Например, для того 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 должны присутствовать функции, глобальные переменные или определения 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 со следующими 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, их имена начинаются со слова lib. Например, libm.a — имя библиотеки, которая содержит 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. 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 — ее имя.

Кроме статических большинство UNIX-систем поддерживают 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. Так, динамически подгружаемый вариант библиотеки математических функций будет иметь имя libm.so.

Для того чтобы при сборке в исполняемом файле были установлены точки вызова функций из динамически подгружаемых библиотек, необходимо указывать ключ компилятора -dynamic. Следу-

ющая 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
```

Использование динамических библиотек имеет существенное преимущество над статическими — программный код библиотечных функций находится вне исполняемого файла и может использоваться несколькими программами. Поскольку при использовании статических библиотек программный код хранится непосредственно в исполняемом файле, динамические библиотеки позволяют экономить дисковое пространство за счет избавления от избыточности. Однако это преимущество динамических библиотек имеет и обратную сторону. Динамические библиотеки могут устанавливаться в систему отдельно от использующих их программ, в случае отсутствия 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 именем исполняемого файла на экран будут выведены 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 ее установку. Конфликт библиотек может быть выявлен косвенно по имени и по пути к файлу библиотеки. Например, программа может ошибочно искать библиотеку в другом каталоге. В этом случае необходимо или библиотеку 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. Так, если возникает необходимость компилировать программу с использованием наборов одноименных библиотек, находящихся 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 выглядеть как

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

Все рассмотренные выше способы компиляции и сборки программ скрывали от пользователя этап сборки — объектные файлы удалялись сразу после создания исполняемого файла и пользователь получал исполняемый файл, не видя объектных.

В некоторых случаях требуется выполнять два этапа — компиляцию и сборку, сохраняя объектные файлы. Например, некоторые библиотеки поставляются только в виде объектных файлов без исходных текстов. 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.

Для того 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. Так, команда

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

вызовет 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 станет производиться в каталоге /usr/local/lib

(при выполнении программы местонахождение библиотеки будет задаваться переменной окружения LD_LIBRARY_PATH).

Приведем пример простейшей программы, реализовать которую при помощи задания на языке BASH достаточно затруднительно. Это программа, возвращающая целое число, которое равно значению синуса от угла, переданного в качестве параметра, умноженному на 100. BASH не имеет встроенных тригонометрических функций, однако на 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, будет выглядеть как

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

9.3. КОМПИЛЯЦИЯ ПРОГРАММ В WINDOWS

Рассмотрим компиляцию с использованием среды разработки Microsoft Visual Studio 2010. Эта среда разработки включает развитую IDE, которая существенно облегчает работу не только начинающих, но и опытных программистов. Очень часто разработчику даже не надо заботиться о дополнительных настройках, достаточно использовать автоматические настройки среды. Но иметь представление об общем ходе процесса компиляции программ необходимо, так как иногда все-таки приходится изменять настройки по умолчанию.

Компиляция программ в Windows также проводится в два этапа. На первом этапе из исходных текстов при помощи компилятора cl.exe формируются объектные файлы (с расширением .obj). На втором этапе при помощи сборщика link.exe формируется исполняемый файл или файл библиотеки.

Так же как и в компиляторах UNIX, управлять ходом компиляции можно с помощью специальных ключей компилятора. Например, для того чтобы получить исполняемый файл myprogram.exe на основе исходного текста программы из файла myprogram.c, достаточно воспользоваться командой

```
cl.exe myprogram.c
```

С помощью данной команды автоматически скомпилируется программа и будет сгенерирован объектный файл, после чего автоматически будет вызван линкер и на основе полученного объектного файла и стандартных библиотек будет сгенерирован исполняемый файл.

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

```
cl.exe myprogram1.c myprogram2.c
```

При таком вызове сначала будут скомпилированы два объектных файла, после чего будет вызван линкер и сгенерирован исполняемый файл. Имя исполняемого файла по умолчанию определяется именем первого файла, переданного для компиляции. В данном примере имя программы будет myprogram1.exe.

Для того чтобы указать пути нахождения заголовочных файлов, используется ключ /I. Например, при компиляции со следующими параметрами компилятор будет искать заголовочные файлы в каталоге ..\include:

```
cl.exe /I"..\include" myprogram.c
```

Если необходимо изменить имя генерируемого выходного файла, то следует передать линкеру специальный ключ /OUT:, позволяющий переопределять имя выходного файла. При этом компилятор необходимо известить, что опция передается линкеру, а не компилятору. Для этого используется ключ /link, например

```
cl.exe myprogram1.c myprogram2.c /link
/OUT:myprog.exe
```

Можно воспользоваться и ключом самого компилятора:

```
cl.exe /Femyprog.exe myprogram1.c myprogram2.c
```

Но обычно для таких процедур процесс компиляции разбивают на два шага — собственно компиляция и линковка. Для того чтобы приказать компилятору просто компилировать исходные файлы без линковки, используется опция /c:

```
cl.exe /c myprogram1.c myprogram2.c
```

После этой команды будут сгенерированы два файла — myprogram1.obj и myprogram2.obj. Чтобы слинковать эти файлы и получить исполняемую программу, необходимо задать следующую команду:

```
link.exe /OUT:myprog.exe myprogram1.obj
myprogram2.obj somelib.lib
```

В результате получится программа myprog.exe, которая будет собрана из двух объектных файлов — myprogram1.obj и myprogram2.obj — и одной библиотеки — somelib.lib. Файлы статических библиотек имеют расширение .lib.

Для того чтобы указать пути нахождения файлов библиотек, используется ключ /LIBPATH:. Например, чтобы собрать программу из приведенного выше примера, но при этом задать дополнительный каталог поиска библиотек, можно вызвать следующую команду:

```
link.exe /OUT:myprog.exe /LIBPATH:"..\libs"
myprogram1.obj myprogram2.obj somelib.lib
```

Рассмотрим ранее приведенный пример объединения сценария на BASH и программы на C на платформе Windows. Только используем для этого компилятор Microsoft Visual Studio и командный интерфейс cmd.exe:

```
#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 cl.exe:

```
cl.exe /Fesin.exe sin.c
```

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

```
@echo off
rem Вычисление синуса
if "%1" == "" (
echo No parameters specified
exit 1
)

if not "%2" == "" (
echo More than one parameter specified
exit 2
)

.\sin %1
echo Hundredths of sine of angle %1 equals
%ERRORLEVEL%
```

Результат работы этого командного файла будет таким же, как и в аналогичном примере в Linux.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие основные компоненты операционной системы входят в состав среды разработки прикладного программного обеспечения?
2. Какую роль играет язык программирования С в среде операционной системы UNIX?
3. Какие действия выполняются при подготовке исполняемого модуля, написанного на языке высокого уровня?
4. В чем специфика подготовки исполняемых программных файлов в системе Windows?

Глава 10

МЕЖПРОЦЕССНОЕ ВЗАИМОДЕЙСТВИЕ

10.1. ВИДЫ МЕЖПРОЦЕССНОГО ВЗАИМОДЕЙСТВИЯ

В значительной части сложных программ в настоящее время присутствует та или иная форма межпроцессного взаимодействия. Это обусловлено естественной эволюцией подходов к проектированию программных систем, которая последовательно прошла следующие этапы:

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, выполняющий всю необходимую работу;

2) модульные программы, состоящие 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 процесс или несколько процессов. Каждый из процессов для решения задачи либо использует 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. Преимущество межпроцессного взаимодействия заключается в еще большей универсальности, т. е. 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 операционной системе управлять приоритетами выполнения отдельных частей программного комплекса, выделяя большее или меньшее количество ресурсов ресурсоемким процессам.

Для реализации процессов, решающих общую задачу, операционная система должна быть обеспечена средствами взаимодействия между ними. Предоставление средств взаимодействия — задача операционной системы, 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, не доступным 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.

10.2. МЕХАНИЗМЫ МЕЖПРОЦЕССНОГО ВЗАИМОДЕЙСТВИЯ

Современные операционные системы реализуют семь основных механизмов межпроцессного взаимодействия, перечисленных далее.

Прерывания. Изначально механизм прерываний использовался в операционных системах для оповещения. В определенные моменты времени аппаратное устройство извещает о наступлении некоторого события (готовности к действию, сбое, завершении передачи блока данных). Количество вариантов таких событий может быть достаточно большим, и все они должны быть различимы операционной системой. Такое извещение о готовности получило название прерывания, поскольку в момент получения прерывания операционная система должна приостановить выполнение текущих задач и среагировать на поступившее прерывание.

Реакция на прерывание, как правило, заключается в выполнении программного кода, который находится в памяти, адресуемой операционной системой. Операционная система поддерживает специальную область памяти (таблица прерываний), где каждому прерыванию (как правило, идентифицируемому по номеру) ставится в соответствие адрес памяти, по которому находится программный код (обработчик прерывания). Количество прерываний, поддерживаемых системой, фиксированно. Операционная система поддерживает обычно 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 операционная система возвращает управление задачам либо 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 операционной системе 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 операционную систему. 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, — память процесса. Обычный обработчик сигнала — функция, определенная в теле программы. В момент получения сигнала происходит вызов этой функции.

Операционная система поддерживает таблицу обработчиков сигналов для каждого процесса. В ней каждому сигналу ставит-

ся в соответствие адрес обработчика сигнала. Например, UNIX-подобные операционные системы поддерживают 16 и более различных сигналов. Более подробно механизм сигналов будет рассмотрен далее.

Сообщения. Механизм сообщений предназначен для обмена данными между процессами. Для этого создается специальная очередь сообщений, поддерживаемая операционной системой. В очереди накапливаются данные, которые записываются в нее процессами. Накопленные в очереди данные могут быть «считаны» другими процессами, таким образом произойдет передача данных от одного процесса к другому.

Операционная система поддерживает специальную область оперативной памяти, в которой хранятся очереди. Обычно UNIX-подобная операционная система поддерживает до 1 024 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 котором выполнение обработчика начинается сразу по получении 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 в этот файл и считывать из него данные. При этом операционная система 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 при взаимодействии 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 процессов отображается один и тот же

участок физической памяти, адресуемой операционной системой. Используя этот общий участок памяти, процессы могут обмениваться данными без участия ядра операционной системы, 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 может возникать при одновременной записи данных в один участок памяти несколькими процессами. В этом случае память будет содержать данные последнего записывающего процесса, остальные 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 конфликтных ситуаций применяется механизм семафоров — специальных флагов, указывающих на возможность использования того или иного участка памяти. В момент записи в память процесс ставит семафор — это означает, что память «занята»; по окончании 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 приостанавливать 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.

10.3. СИГНАЛЫ

10.3.1. Общие 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 операционной системы (например, в случае деления на 0 ядро по-

сылает 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, выполняющемуся 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. 10.1 содержится краткое описание основных типов сигналов.

Каждому типу сигнала присваиваются 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.

Доставка сигнала до процесса-получателя выполняется операционной системой. В ответ на получение сигнала процесс-получатель может либо проигнорировать сигнал, либо начать выполнять программный код, связанный с получением сигнала данного типа. Код обычно оформляется в виде отдельных функций, вызываемых при получении сигнала определенного типа. Такие функции получили название «функции-обработчики сигналов» (или просто «обработчики сигналов»).

Вся совокупность обработчиков сигналов для каждого процесса образует таблицу обработчиков сигналов. Таблица уникальна для каждого процесса. Количество строк в таблице обработчиков равно числу поддерживаемых операционной системой сигналов. В каждой строке таблицы записываются 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 завершении выполнения функции-обработчика управление возвращается командой, следующей за той, при выполнении которой был получен сигнал (рис. 10.1).

Таблица обработчиков сигналов может не содержать адресов функций-обработчиков для некоторых (возможно, даже для всех)

Таблица 10.1. Основные сигналы в UNIX

Номер сигнала	Мнемоника сигнала	Описание сигнала	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 некорректной работе с динамической памятью)	Завершение процесса и создание файла core
14	SIGALRM	Наступление тайм-аута таймера; может генерироваться функцией alarm()	Завершение процесса
15	SIGTERM	Завершение процесса	То же
20	SIGCHLD	Посылается процессом-потомком при завершении родителю	Игнорируется
30	SIGUSR1	Пользовательский сигнал	Завершение процесса
31	SIGUSR2	То же	То же

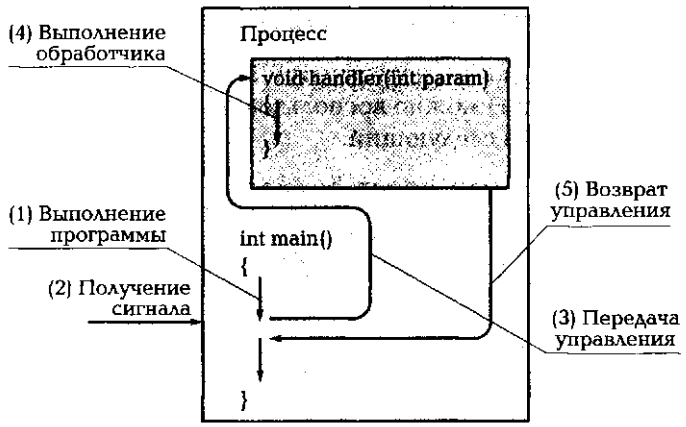

Рис. 10.1. Передача управления обработчику при получении сигнала

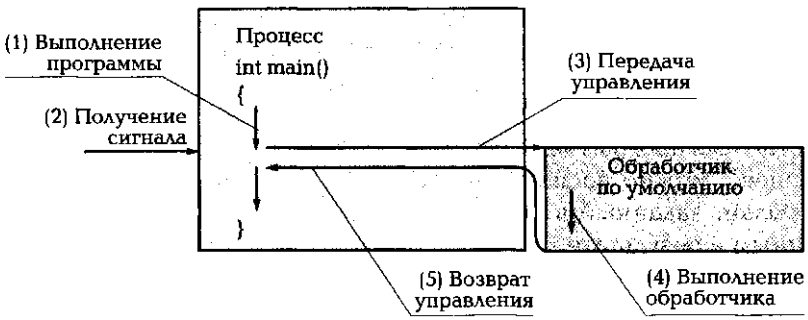

Рис. 10.2. Передача управления обработчику по умолчанию

типов сигналов. В таком случае говорят, что для этих типов сигналов не определен пользовательский обработчик. При получении процессом сигнала такого типа начинает выполняться программный код функции-обработчика по умолчанию, предоставляемого операционной системой (рис. 10.2).

Для большинства типов сигналов функция-обработчик по умолчанию завершает выполнение процесса, получившего сигнал (см. табл. 10.1). Для некоторых типов сигналов пользовательский обработчик не может быть определен в принципе. К ним относятся сигналы, завершающие выполнение процесса: сигнал SIGSTOP, вызывающий «мягкое» завершение процесса, во время которого могут быть корректно сброшены все буферы ввода/вывода; сигнал SIGKILL, вызывающий немедленное аварийное завершение работы процесса.

10.3.2. Сигналы в BASH

Для посылки сигнала с известным PID при помощи командного интерпретатора BASH можно воспользоваться командой kill.

Формат ее вызова следующий:

```
kill -<мнемоника или номер сигнала> <PID>
```

Например, для посылки сигнала SIGKILL процессу с PID = 1046 можно воспользоваться номером сигнала:

```
kill -9 1046
```

А 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. Команда killall 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.

10.3.3. Системные вызовы для работы с сигналами

В UNIX-системах определен интерфейс системных вызовов для работы с сигналами. Все прототипы функций и типов дан-

ных для работы с сигналами определены в заголовочном файле signal.h.

Для посылки сигналов в С-программах 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, сигнал посылается процессу с PID, равным значению параметра. Если параметр pid равен нулю, сигнал посылается всем процессам, EGID которых равен EGID процесса, посылающего сигнал. Если параметр pid равен –1, 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, 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 приведенной выше программе была использована 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, 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.

Для того чтобы задать в программе функцию-обработчик сигнала, используется системный вызов signal():

```
#include <signal.h>
void signal(int signal_num, void *handler_address);
```

Здесь signal_num задает номер сигнала, при получении которого процесс начинает выполнение функции-обработчика. Ее адрес задается параметром handler_address.

Как уже 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 получении 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 клавиш [Ctrl+C] программа выведет следующее:

```
Signal with number 2 caught
```

так как нажатие сочетания клавиш [Ctrl+C] посылает текущему процессу сигнал SIGINT.

Здесь в функции main() определяется функция-обработчик для сигналов SIGALRM и SIGINT, затем программа встает в бесконечный цикл ожидания прихода сигнала. По получении 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.

10.3.4. Временные характеристики обмена сигналами

При анализе последнего примера следует обратить внимание на то, что если сигналы SIGALRM или SIGINT будут получены процессом до того, как функцией signal() установлены обработчики этих сигналов, то будет вызвана не функция handler(), а обработчик сигнала по умолчанию. Для данной конкретной программы 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 временных 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.

Для того 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 тот момент времени, когда получение сигнала наиболее вероятно. Если процесс получит сигнал в приостановленном состоянии, никакой вычислительный процесс этого гарантированно не прервет.

Приостановить выполнение процесса можно двумя способами. Первый останавливает выполнение процесса, и выход из состояния останова возможен только по получении 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 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, которое равно 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 значения времени, оставшегося до истечения интервала, если процесс получил сигнал во время работы функции usleep().

10.3.5. 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, по получении 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(), а затем входит в бесконечный цикл, в котором приостанавливает свое выполнение до получения сигнала, о чем выводит соответствующее сообщение. По получении 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 необязательно 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 при получении 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
```

Рассматриваемые 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), 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, 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, будет вызван обработчик по умолчанию, который может и вовсе завершить выполнение процесса.

Для того 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 обработке 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 будет выглядеть в этом случае так:

```
void handler(int sig_no)
{
signal( sig_no, SIG_IGN);
...
...
signal( sig_no, &handler);
}
```

3) воспользоваться сигнальной маской.

10.3.6. Сигнальные маски

Сигнальная маска процесса определяет, какие сигналы, получаемые процессом, игнорируются и не обрабатываются. При создании процесс 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, которые указаны в маске, передаваемой в качестве параметра *new_mask*. Добавленные сигналы начинают игнорироваться процессом;
- SIG_UNBLOCK — удаляет из маски процесса сигналы, которые указаны в маске, передаваемой в качестве параметра *new_mask*. Удаленные 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, которая задана 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 0, 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 получении 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 начинает ожидать прихода сигнала. Если сигнал SIGINT придет во время 5-секундной паузы, он будет проигнорирован. 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 следующего 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 мало́. Однако если существует вероятность прихода сигнала во время выполнения этих функций, можно

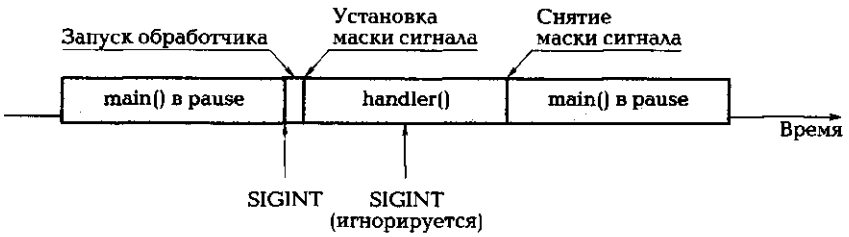

Рис. 10.3. Выполнение процесса при установленной сигнальной маске

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. 10.3 (второй сигнал SIGINT игнорируется).

10.3.7. Таймер

Все рассмотренные выше примеры программ, основанных на использовании сигналов, имеют один общий недостаток — полное время их выполнения целиком зависит от внешней среды. Завершить выполнение этих программ могут либо другие процессы, послав сигнал SIGKILL или SIGINT, либо пользователь, нажавший [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 определен 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 значе-

ние параметра time_interval при этом будет равно нулю, то посылка сигнала будет отменена.

Функция alarm() может быть использована 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
```

10.3.8. Потери сигналов

Рассматриваемые в данном разделе сигналы получили название «ненадежные сигналы». Причиной такого названия послужило то, что для этих сигналов не гарантируются их доставка до процесса-получателя и их обработка процессом. Основная причина этого заключается в том, что после прихода сигнала сбрасывается адрес функции-обработчика в соответствующей строке таблицы обработчиков, а также в том, что при одновременном приходе нескольких одинаковых сигналов процесс обрабатывает только один из них. Чтобы разобраться в причинах этого, рассмотрим более подробно структуру таблицы обработчиков сигналов и доставку 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. 10.4) [5].

Массив сигнальных флагов хранится в соответствующей процессу строке таблицы процессов, массив спецификаций обработки сигналов — в так называемой U-области процесса, которая со-

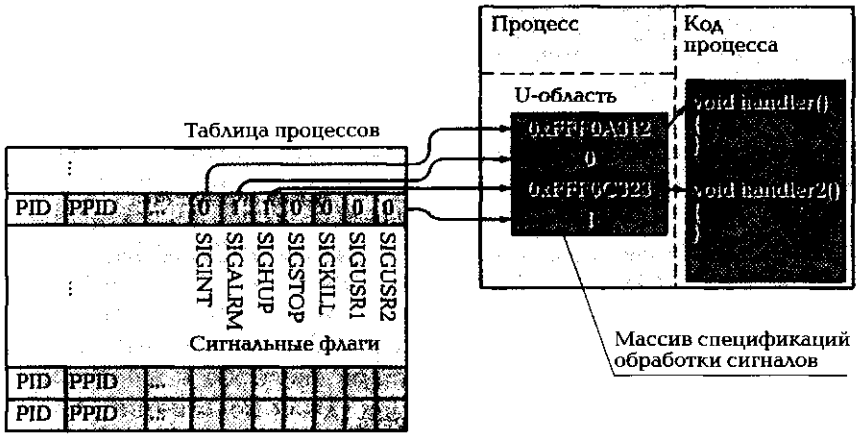

Рис. 10.4. Структура таблицы обработчиков сигналов

держит 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 активен, ядро считывает значение элемента массива спецификаций обработки сигнала, который соответствует установленному сигнальному флагу. Если элемент массива содержит нулевое значение, процесс выполнит обработчик по умолчанию. Если элемент массива содержит единицу, 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, заданная пользователем функция-обработчик будет вызвана только для первого сигнала, для остальных будет вызван обработчик по умолчанию.

Как уже упоминалось ранее, для того чтобы корректно обрабатывать все поступающие последовательно сигналы, необходимо восстанавливать адрес функции-обработчика в соответствующем элементе массива спецификаций обработки.

Рис. 10.5. Реакция на сигнал при его поступлении в различные моменты времени

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 небольшое время), то возникнет ситуация гонок — неизвестно, какой из обработчиков будет вызван: обработчик по умолчанию или восстановленный пользовательский обработчик (рис. 10.5).

Все эти особенности должны учитываться при разработке приложений, принимающих сигналы, посланные через очень малые промежутки времени. Для того чтобы устранить рассмотренные в этом разделе недостатки ненадежных сигналов, в UNIX-системах появился 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 [5] и [16].

10.3.9. Синхронизация процессов

Для того чтобы избежать проблем с потерями сигналов, причины которых были рассмотрены в предыдущем подразделе, 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.

Один из возможных способов реализации такого подхода состоит в том, что процесс-передатчик посылает сигналы через промежутки времени, заведомо больше требуемых процессу-получателю для обработки предыдущего сигнала и восстановления обработчика. Например, следующая программа инициирует два процесса — родительский, порождаемый при ее запуске, и дочерний, порождаемый запуском fork(). Дочерний процесс выступает в роли передатчика и посылает родительскому процессу сигнал SIGINT 10 раз с интервалом между посылками в одну секунду. Получатель (родительский процесс) в ответ на получение сигнала выводит общее число полученных им сигналов SIGINT. После посылки серии из 10 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 количества 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 при помощи либо сигнальной маски, либо константы SIG_IGN.

10.4. СООБЩЕНИЯ

Механизм сигналов, рассмотренный ранее, 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.

Однако сигналы могут применяться и в этом случае, например, каждый символ может кодироваться определенной последовательностью сигналов, начало и конец передачи символа также могут обозначаться сигналами определенного типа. Такой способ неудо-

бен тем, что для передачи одного символа требуется передача достаточно большого числа сигналов (для передачи букв латинского алфавита — около 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, указывая на конверте имя адресата, а адресаты, периодически проверяя содержимое почтового ящика, забирают предназначенные им сообщения, устанавливая 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 — операционная система 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 принципу 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. 10.6).

Таким образом, операционная система 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. Таким образом, возможна ситуация,

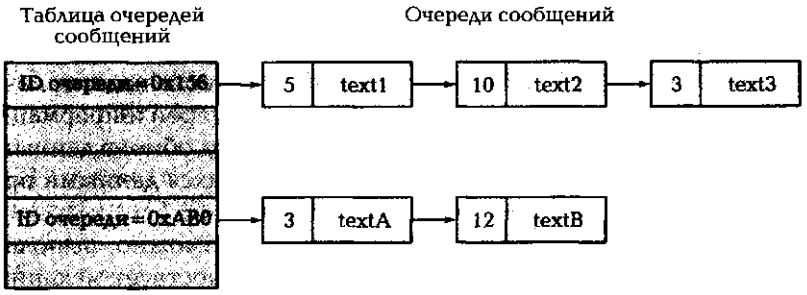

Рис. 10.6. Структура очередей сообщений

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 операционной системы.

Другая ситуация возникает в том 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 идентификаторов очередей сообщений, kernel.msgmax — максимальный размер сообщения в байтах, а kernel.msgmnb задает максимальный размер очере-

ди 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
```

Для того 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
```

10.4.1. Структуры данных для сообщений в 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, работающие с оче-

редью 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 содержится линейный список с сообщениями, хранящимися в очереди. Каждый элемент такого линейного списка содержит данные сообщения и его идентификатор. Кроме того, элемент списка содержит 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 задаются 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.

10.4.2. Системные вызовы для работы с сообщениями

Интерфейс для работы с сообщениями в 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 существует четыре основных 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 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, 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.

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 модуля 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 возвращает количество принятых байтов текстовой части сообщения в случае успешного его получения или –1 в случае неуспешного, например если длина сообщения превышает длину, ука-

занную параметром len. Для того 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.

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; пара-

метр 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;
print ("%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 в очередь или извлекается из нее.

10.5. СЕМАФОРЫ

10.5.1. Основные понятия

Одна из наиболее важных задач при межпроцессном взаимодействии — задача синхронизации выполнения процессов. Под синхронизацией в первую очередь понимается параллельное выполнение 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 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 0 для того, чтобы заблокировать доступ к ресурсу другим процессам. После окончания использования ресурса процесс опять устанавливает значение семафора 0. Такой семафор получил название «*бинарный семафор*».

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 голландского Verhogen, часто обозначается как wait) останавливает выполнение процесса в случае, если его значение меньше либо равно нулю, или уменьшает значение семафора на единицу в противном случае. Операция P (от голландского Prolagen, часто обозначается как post) увеличивает значение семафора на единицу 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 единицу, отражая таким образом уменьшение количества доступных устройств. Если не осталось ни одного доступного устройства, 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 единицу.

Для нормальной работы семафоров-счетчиков в UNIX-системах выполняются 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.

10.5.2. Системные вызовы для работы с семафорами

В UNIX-системах существует несколько интерфейсов для работы с семафорами. В данном разделе рассматривается 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 либо объединить эти операции.

Для создания или открытия существующего набора семафоров в UNIX используется функция semget():

```
#include <sys/sem.h>
/*В 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 — 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 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.

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 его использования необходимо присвоить начальное значение каждому семафору набора. Это можно сделать при помощи функции semctl():

```
#include <sys/sem.h>
/*В 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 следующие:

- 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.

Для того чтобы установить значения всех семафоров, можно воспользоваться командой SETALL или циклическим вызовом команды SETVAL:

```
#include <unistd.h>
#include <sys/sem.h> /* В 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. Для того 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 рассмотрены в [18].

Для просмотра всех созданных на данный момент семафоров в системе используется команда ipcs, которую необходимо запустить из командной строки 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, semmsl — максимальное количетво семафоров в массиве, параметр semmns задает максимальный число семафоров в системе и вычисляется как произведение semmsl и semmni. Параметр semopm задает максимальное число операций над семафорами, которое может быть выполнено за один раз.

Эти 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 seopm и четвертое — параметра semni.

Для того 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 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>
/* В 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 оно равно нулю, 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 значение семафора станет отрицательным, то ядро также приостановит выполнение процесса. Выполнение процесса будет продолжено тогда, когда значение семафора станет неотрицательным.

Если в качестве флага sem_flg указать константу IPC_NOWAIT, приостановки процесса не произойдет. Если указать в качестве флага константу SEM_UNDO, ядро будет отслеживать изменения семафоров. Если процесс, уменьшивший значение семафоров до нуля или отрицательного числа, будет завершен, сделанные им изменения будут отменены, чтобы не вызвать «вечного» ожидания процессами 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. Для того 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 единицу и соответствует операции V, op[1] проверяет значение семафора на нуль и не используется в примере, op[1] увеличивает значение семафора на единицу и соответствует операции P:

```
#include <sys/sem.h>
/* В 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 сек) */
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 сек) */
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

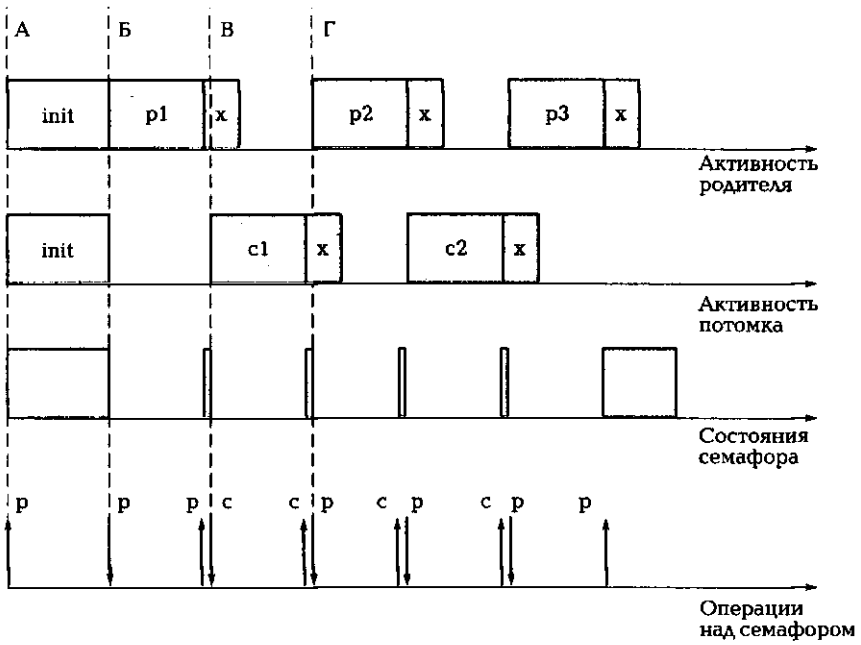

Рис. 10.7. Временная диаграмма использования семафора для синхронизации процессов

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. 10.7. 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 линия времени отражает операции над семафором; стрелкой, направленной вниз отмечено уменьшение значения семафора; стрелкой, направленной вверх, — увеличение значения. Рядом со стрелкой указан процесс, выполнивший операцию. Буквой «p» обозначен процесс-родитель, буквой «c» — процесс-потомок.

Оба процесса стартуют практически одновременно и сначала инициализируют собственные данные, родитель при этом открывает семафор (срез А). После завершения инициализации процесс-родитель закрывает семафор, начиная тем самым критическую секцию (срез Б и p1 на графике).

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 (x на графике). Процесс-потомок в этот момент активизируется, уменьшает значение семафора и начинает свое выполнение в критической секции (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 (срез Г). Затем процесс повторяется.

10.6. ПРОЦЕССЫ И 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. В данном подразделе мы рассмотрим некоторые из существующих методов организации взаимодействия в операционных системах семейства Windows.

10.6.1. Процессы и потоки

Как и в других операционных системах, *процессы* (process) являются основными объектами, позволяющими пользователям решать их задачи. Каждый процесс предоставляет ресурсы, необходимые для выполнения соответствующего приложения. Каждый процесс 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 размеры доступной процессу виртуальной памяти; по крайней мере один поток управления.

Каждый процесс при старте запускает один-единственный *поток управления* (thread), который называют *первичным потоком* (primary thread). Но каждый поток может создавать новый поток. 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 операционные системы семейства Windows) и на ядре Windows 9x (Windows 95, Windows 98 и Windows ME), поддерживают так называемую *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 операционная система, 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. Данный механизм носит название «переключение 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 операционные системы 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 данной функции процесс называется процессом-потомком. Процесс-потомок полностью независим от породившего его процесса. Но родительский процесс получает возможность следить за порожденным процессом и отслеживать некоторые события, связанные с ним.

Параметр lpApplicationName задает имя программы, которая должна быть выполнена. Если данный 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 от прав по умолчанию. Кроме того, один из элементов структуры, на которую указывает данный параметр, используется для указания того, что дескриптор процесса, созданного в результате вызова функции CreateProcess(), может быть унаследован процессами-потомками.

Параметр lpThreadAttributes используется так же, как и параметр lpProcessAttributes. Но 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. Например, если 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 завершается нулем и состоит из строк вида:

```
name=value\0
```

Если данный параметр имеет значение NULL, окружение наследуется от процесса-родителя.

Параметр lpCurrentDirectory используется для задания текущего диска и каталога для создаваемого процесса. Если параметр равен NULL, то текущий диск и каталог наследуются от процесса-родителя.

Параметр lpStartupInfo также предназначен для изменения характеристик создаваемого процесса. Данный параметр 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 либо на устройство. Данный параметр позволяет изменять и другие свойства создаваемого процесса.

Параметр lpProcessInformation является возвращаемым. В данном параметре возвращаются дескрипторы и идентификаторы созданного процесса и его первичного потока. Данная функция возвращает 0, если по какой-либо причине не удалось создать новый процесс, или значение, отличное от 0, если процесс был успешно создан.

Причинами отказа системы в создании нового процесса может быть исчерпание виртуальной памяти, переполнение таблицы процессов или тот факт, что запускаемая программа не является программой либо сценарием. 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 для данного параметра в системах Windows XP и Vista установлено 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 от прав доступа, отличных прав по умолчанию. Также данный параметр используется для указания того, что дескриптор созданного потока может быть унаследован процессами-потомками. Если же параметр имеет значение NULL, поток получаст 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 соответствует размеру стека по умолчанию, определенного 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 данный параметр имеет значение CREATE_SUSPENDED, то созданный поток переводится в приостановленное состояние до тех пор, пока другой головной поток не возобновит его.

Возвращаемый параметр lpThreadId используется для получения идентификатора созданного потока. Если данный параметр равен NULL, то идентификатор созданного потока не возвращается.

Если вызов функции закончился созданием нового потока, 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 причиной неуспешной попытки создания нового потока является исчерпание виртуальной памяти, если в системе уже существует большое количество потоков.

Для получения дескриптора и идентификатора текущего процесса используются функции GetCurrentProcess() и GetCurrentProcessId() соответственно:

```
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: команда распечатки
   // переменных окружения и идентификатор 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
  // Проверяем, что поток был успешно создан
  if (hThread == NULL)
  {
  ErrorReport(TEXT("CreateThread()"));
  return(1);
  }
  // Ждем ожидания завершения потока без тайм-аута
  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 тайм-аута
  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)
      lpszFunction)+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 данной 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 приведенной выше 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 данном 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 имеет 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 тогда, когда 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 данной 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 для параметра bWaitAll установлено значение TRUE, то функция возвратит управление ожидающему процессу только тогда, когда все объекты из массива будут 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 данная функция возвращает значение WAIT_TIMEOUT, если истекло время ожидания перевода объек-

тов в сигнальное состояние. Если при вызове функции параметр bWaitAll равен TRUE и все объекты были переведены в сигнальное состояние до истечения периода ожидания, функция возвращает значение WAIT_OBJECT_0. Если же параметр bWaitAll равен в FALSE и какой-либо объект был переведен в сигнальное состояние, 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, последние 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.

10.6.2. Синхронизация. События, семафоры, мьютексы

Как и большинство операционных систем, поддерживающих одновременную работу нескольких конкурирующих процессов, операционные системы семейства Windows поддерживают целый 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.

Операционные системы семейства Windows поддерживают такие механизмы синхронизации, как события, семафоры, мьютексы, критические области, и еще целый ряд методов организации разделения ресурсов между конкурирующими процессами. В данном подразделе мы 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 сбрасываемые автоматически.

Автоматически сбрасываемые события характеризуются тем, что если такое событие переводится в сигнальное состояние, система автоматически выбирает ожидающий сигнального состояния поток

и передает управление ему. При этом такое событие автоматически переводится в несигнальное состояние. При работе со сбрасываемыми вручную событиями ответственность за восстановление несигнального состояния события берет на себя программист. В этом случае он должен явно 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 данный параметр имеет 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 данный 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 данный 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 лимитируется 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. Данный параметр может иметь значение EVENT_ALL_ACCESS, позволяющее получить полный доступ к событию, либо EVENT_MODIFY_STATE. Данное значение 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 данный параметр равен 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 данную функцию передается 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 рассмотренных ранее 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:

```
#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)
        lpszFunction)+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
```

Видно, что, несмотря на разное время блокировки потоков, их вывод строго чередуется. Если же в приведенной выше 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
```

Видно, 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, уже встречался при ознакомлении со средствами синхронизации в Linux — семафоры. В Windows семафоры представляют собой

объекты синхронизации, имеющие внутренний счетчик, значение которого может находиться в диапазоне от нуля до заданного максимального значения. Значение этого счетчика уменьшается на единицу каждый раз, когда одному из потоков возвращается управление из функции ожидания (WaitForSingleObject() и WaitForMultipleObjects()), и увеличивается , когда поток освобождает семафор. Семафор переходит в сигнальное состояние каждый раз, когда значение счетчика не равно 0, и в несигнальное состояние, когда значение счетчика равно 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 данный параметр имеет значение NULL, 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 данный параметр равен NULL, создается анонимный семафор. Если же передается имя, которое уже было использовано при создании системных объектов Windows, таких как события, семафоры, мьютексы и т.д., семафор не создается и функция завершается 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. Данный параметр может принимать значение SEMAPHORE_ALL_ACCESS, позволяющее получить полный доступ к объекту, либо SEMAPHORE_MODIFY_STATE.

Параметр bInheritHandle определяет, может ли наследоваться дочерними процессами создаваемый дескриптор семафора. Если данный параметр имеет 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 данной функции можно получать доступ только к именованным семафорам, а неименованные семафоры могут использоваться только процессом, создающим данный объект, и его потоками.

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. Данное значение должно быть больше 0. Параметр lpPreviousCount используется для возвращения предыдущего значения счетчика семафора.

Еще одним средством синхронизации процессов и потоков в Windows являются мьютексы. Мьютекс представляет собой объект синхронизации, который переводится 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 CreateMutex():

```
#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 данный параметр имеет значение NULL, дескриптор не наследуется и устанавливаются права по умолчанию.

Если параметр bInitialOwner равен TRUE, то поток, создавший мьютекс, объявляется его владельцем. Если же данный параметр равен FALSE, то создающий мьютекс вычислительный поток не получает его во владение.

Параметр lpName используется для задания имени мьютекса. Если данный параметр равен NULL, создается анонимный объект. Если же передается имя, которое уже было использовано при создании системных объектов Windows, таких как события, семафоры, мьютексы и т.д., семафор не создается и функция завершается ошибкой. Если функция завершается успешно и удается создать семафор, 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. Данный параметр может принимать значение MUTEX_ALL_ACCESS, позволяющее получить полный доступ к объекту, либо MUTEX_MODIFY_STATE.

Параметр bInheritHandle определяет, может ли наследоваться дочерними процессами создаваемый дескриптор мьютекса. Если данный параметр имеет значение TRUE, создаваемый дескриптор наследуется дочерними процессами, в противном случае создается ненаследуемый дескриптор.

Параметр lpName задает имя мьютекса, доступ к которому необходимо получить. С помощью данной функции можно получать доступ только к именованным семафорам, а неименованные семафоры могут использоваться лишь 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 модифицируем ранее рассмотренный пример и используем оба механизма синхронизации: 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)
      lpszFunction)+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 применяться для организации взаимно исключающего доступа пер-

вичного и дочернего вычислительных потоков к консоли. В дальнейшем семафор не используется, а применяется 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
```

Так же, 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, 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, когда при отсутствии синхронизации потоков первичный поток завершает свою работу раньше дочернего потока и вывод данных на консоль двух вычислительных потоков не синхронизируется.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. В чем различие механизмов прерываний и сигналов?
2. Каким образом производится обработка полученного сигнала процессом? Каковы возможности пользователя для обработки сигнала?
3. Каким образом принимаются из очереди сообщения, размер которых больше ожидаемого?
4. В чем основные отличия бинарных семафоров от семафоров-счетчиков?
5. Почему необходимо использовать средства синхронизации процессов при использовании общей памяти?
6. Чем отличаются неименованные каналы от именованных? Как процесс может записать и прочитать данные из канала?

Приложения

ПРИЛОЖЕНИЕ 1

«КОНТРОЛЬ ЗНАНИЙ». СТРУКТУРА КАТАЛОГОВ

Общая постановка задачи разработки системы «Контроль знаний» приведена в подразд. 1.5.

Каталоги и файлы системы «Контроль знаний» 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* размещаются рабочие области студентов. Каждая рабочая область представляет собой каталог с именем, соответствующим имени студента, в котором хранятся работы, выполняемые студентом, и подкаталог *ready*, в который помещаются уже выполненные работы.

В каталоге *teacher* хранится рабочая область преподавателя, которая содержит базу вариантов контрольных работ и каталог 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 структура?
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 *vasya-theme2_var1.txt*.

С системой работают 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* становится возможным предотвратить несанкционированный доступ студентов к каталогу с вариантами контрольных, а разграничение прав доступа сводит к минимуму вероятность списывания (просмотра контрольных у другого студента).

Основная постановка задачи разработки системы «Контроль знаний» приведена в подразд. 1.5.

Задания, обеспечивающие выполнение работ в рамках системы «Контроль знаний», написаны на языке командного интерпретатора BASH.

Задания предлагается разделить на три группы:

- для работы преподавателя, которые размещаются в каталоге check/scripts/teacher;
- для работы студентов, которые размещаются в каталоге check/scripts/students;
- служебные задания, которые размещаются в каталоге check/scripts.

Для обеспечения корректной передачи заданий преподавателя и студентов необходимо присвоить значение переменной окружения BASEDIR, которая задает полное имя каталога системы «Контроль». Для тех пакетных файлов, которыми пользуются студенты, необходимо еще установить значение переменной NAME, указывающее имя каталога, хранящего рабочую область студента. Для этого используется задание *scripts/env.sh*. Ниже приведены примеры текстов некоторых пакетных файлов, входящих в состав системы «Контроль».

Примеры исходных текстов заданий:

scripts/env.sh — пакет установки переменных окружения

```
#!/bin/bash

# Задание предназначено для установки переменных
# окружения пользователя для корректной
# работы с системой

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 — пакет установки прав доступа к каталогам системы

```
#!/bin/bash

# Задание предназначено для установки прав доступа к
# каталогам системы

# 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 — пакет создания каталогов системы

```
#!/bin/bash

# Задание предназначено для первоначального создания
# всех каталогов системы "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 — пакет выдачи 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 недоступен"
   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 недоступен \
        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 недоступен \
        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/give_all.sh — пакет выдачи контрольной задачи 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 недоступен"
   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 недоступен"
   exit 101
fi

WORKSDIR=$BASEDIR/teacher/works
if [ ! -d $WORKSDIR -o ! -r $WORKSDIR -o ! \
       -x $WORKSDIR ]; then
   echo "$WORKSDIR не является каталогом или \
         недоступен"
   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 недоступен \
            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
```

КРАТКИЙ СПРАВОЧНИК ПО КОМАНДАМ UNIX

Краткая справочная информация по основным командам операционной системы UNIX приведена в табл. П.1. Более болные сведения можно получить, вызвав страницу справочной системы *man* для каждой команды. Если команда помечена как внутренняя команда *bash* (под ее названием сделана пометка (BASH)), информацию по ней можно получить, вызвав страницу справочной системы *bash*.

Условные обозначения: в квадратных скобках [ ] указаны необязательные конструкции, например, необязательные параметры, в угловых скобках < > — обязательные параметры.

Таблица П1. Формат основных команд системы 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]` **Параметры:** *n* — количество вложенных циклов, из которых происходит выход.

Продолжение табл. П.1

Команда	Описание

	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>", 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>` **Параметры:** *-R* — рекурсивное изменение владельца во всех подкаталогах каталогов, указанных в списке; *-v* — вывод на экран имени каждого обрабатываемого файла или каталога;

Продолжение табл. П.1

Команда	Описание
	-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.
	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>`

Продолжение табл. П.1

Команда	Описание

	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>`
	Параметры:
	<имя программы> — полное, относительное или краткое путевое имя исполняемого файла.
	Пример использования:
	`exec ls`

Продолжение табл. П.1

Команда	Описание
	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; <список файлов> — список имен файлов, в которых производится поиск. Команда возвращает 0, если подстрока найдена, и 1, если не найдена.

Продолжение табл. П.1

Команда	Описание

	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 байтов> — вывод заданного числа байтов из начала файла.
	При указании отрицательного числа *n* строк или байтов выводится весь текст, кроме последних *n* строк или байтов.
	Пример использования:
	`head -n 15 file.txt`

kill	Посылка сигнала процессу с заданным PID. **Формат вызова:** `kill [параметры] <PID>`

Продолжение табл. П.1

Команда	Описание
	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 прочитано 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 табл. П.1

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 табл. П.1

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);
	s — вывод информации об обрабатываемых сигналах.
	Пример использования:
	`ps aux`

Продолжение табл. П.1

Команда	Описание
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/`

rmdir	Удаление пустых каталогов (не содержащих файлов и подкаталогов). **Формат вызова:** `rmdir [параметры] <имя каталога>`

Команда	Описание
rmdir	**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 1 позицию
	Пример использования:
	задание запущено с параметрами *a b c d e f*
	`echo $1`
	(выведет a)
	`shift 4`
	`echo $1`
	(выведет e)

sleep	Приостановка выполнения задания на заданное число секунд. **Формат вызова:** `sleep <n>`

Продолжение табл. П.1

Команда	Описание
sleep	**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 байтов> — вывод заданного числа байтов из конца файла;
	-f — не завершать выполнение программы по достижении 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*.
	При указании отрицательного числа *n* строк или байтов выводится весь текст, кроме последних *n* строк или байтов.
	Пример использования:
	`tail -F logfile.txt`

tar	Склеивание файлов и каталогов в один файл для подготовки к архивированию.

Продолжение табл. П.1

Команда	Описание

	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`

test	Вычисление значения условного выражения. Более подробно см. подразд. 5.8.6 и 7.2. **Формат вызова:** `test <выражение>`

Продолжение табл. П.1

Команда	Описание
	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)

wait (BASH)	Ожидание завершения процесса с заданным PID и возврат его кода возврата. **Формат вызова:** `wait <PID>`

Окончание табл. П.1

Команда	Описание
	Параметры: <PID> — PID процесса, завершения которого ожидает команда. **Пример использования:** `wait 1078`

wc	Вывод количества байтов, слов или строк в файле; сначала выводится количество, затем через пробел — имя файла.
	Формат вызова:
	`wc [ параметры ] <имя файла>`
	Параметры:
	-c — выводится количество байтов 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. : Изд-во ИМВС РАН, 2000. — 316 с.

3. *Дунаев С.* UNIX-сервер : В 2 т. / С.Дунаев. — М. : Диалог-МИФИ, 1998. — 304 с.

4. *Краковяк С.* Основы организации и функционирования ОС ЭВМ / С.Краковяк. — М. : Мир, 1988. — 480 с.

5. *Орлов В.Н.* Мобильная операционная система МОС ЕС / В.Н.Орлов, В.Ю.Блажнов, О.А.Барвин. — М. : Финансы и статистика, 1990. — 208 с.

6. *Померанц О.* Ядро Linux. Программирование модулей / О.Померанц. — М. : КУДИЦ-Образ, 2000. — 112 с.

7. *Робачевский А.М.* Операционная система UNIX / А.М.Робачевский. — СПб. : BHV-Петербург, 1999. — 528 с.

8. Руководство системного администратора / Э.Немет, Г.Снайдер, Е.Сибасе, Т.Хейн. — СПб. : Питер ; Киев : BHV, 2002. — 928 с.

9. *Стивенс У.Р.* UNIX. Взаимодействие процессов / У.Р.Стивенс. — СПб. : Питер, 2002. — 576 с.

10. *Стивенс У.Р.* UNIX. Разработка сетевых приложений / У.Р.Стивенс. — СПб. : Питер, 2003. — 1088 с.

11. *Строкин Г.* BASH-конспект / Г.Строкин // http://www.linux.org.ru/books/bash-conspect.html.

12. *Таненбаум Э.* Современные операционные системы / Э.Таненбаум. — СПб. : Питер, 2002. — 1040 с.

13. *Фридл Дж.* Регулярные выражения. Библиотека программиста / Дж.Фридл. — СПб. : Питер, 2001. — 352 с.

14. *Хендриксен Д.* Интеграция UNIX и Windows NT / Д.Хендриксен. — М. : Диа-Софт, 1999. — 352 с.

15. *Цилюрик О.* QNX/UNIX. Анатомия параллелизма / О.Цилюрик, Е.Горошко. — М. : Символ-Плюс, 2006. — 288 с.

16. *Чан Т.* Системное программирование на C++ для UNIX / Т.Чан. — Киев. : BHV, 1997. — 592 с.

17. *Шнитман В.З.* Аппаратно-программные платформы корпоративных информационных систем / В.З.Шнитман, С.Д.Кузнецов. // http://www.citforum.ru/hardware/app_kis/contents.shtml.

18. *Dijkstra E.W.* Cooperating Sequential Processes, in Programming Languages / Ed. F. Genuys. — Academic Press, 1968. — P. 43—112.

19. *Maurice J. Bach.* The Design of the UNIX Operating System. — Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1986. — 471 p.

Оглавление

Учебное издание

Батаев Алексей Владимирович,
Налютин Никита Юрьевич,
Синицын Сергей Владимирович

Операционные системы и среды

Учебник

2-е издание, стереотипное

Редактор *Е. А. Тульсанова*
Технический редактор *Е. Ф. Коржуева*
Компьютерная верстка: *Д. В. Федотов*
Корректор *И. А. Ермакова*

Изд. № 102116318. Подписано в печать 09.07.2015. Формат 60 × 90/16.
Гарнитура «Балтика». Бумага офсетная № 1. Печать офсетная. Усл. печ. л. 17,0.
Тираж 1 000 экз. Заказ №4834

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 16679 от 25.05.2015.

Отпечатано с готовых файлов заказчика
в АО «Первая Образцовая типография»,
филиал «УЛЬЯНОВСКИЙ ДОМ ПЕЧАТИ»
432980, г. Ульяновск, ул. Гончарова, 14

ОПЕРАЦИОННЫЕ СИСТЕМЫ И СРЕДЫ

ISBN 978-5-4468-2474-8

Издательский центр «Академия»
www.academia-moscow.ru