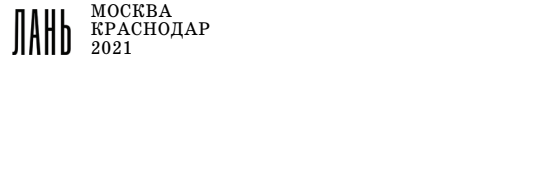
В. Г. КОБЫЛЯНСКИЙ

ОПЕРАЦИОННЫЕ СИСТЕМЫ, СРЕДЫ И ОБОЛОЧКИ

Учебное пособие
Издание второе, стереотипное



САНКТ-ПЕТЕРБУРГ
МОСКВА
КРАСНОДАР
2021

УДК 004.03
ББК 32.973-018.2я73

К 55 **Кобылянский В. Г.** Операционные системы, среды и оболочки : учебное пособие для вузов / В. Г. Кобылянский. — 2-е изд., стер. — Санкт-Петербург : Лань, 2021. — 120 с. : ил. — Текст : непосредственный.

ISBN 978-5-8114-8187-3

Учебное пособие предназначено для студентов направлений «Прикладная математика и информатика» и «Математическое обеспечение и администрирование информационных систем» и содержит описание языка shell, файловых систем и инструментальных средств, предназначенных для подготовки к исполнению программ пользователей. Отдельный раздел посвящен операционным системам реального времени. Все темы изучаются на примере операционных систем Linux, Windows и QNX.

Пособие содержит большое количество примеров и может быть полезным для студентов всех направлений, изучающих принципы функционирования операционных систем и технологии разработки программного обеспечения, и для преподавателей.

УДК 004.03
ББК 32.973-018.2я73

Обложка
П. И. ПОЛЯКОВА

ВВЕДЕНИЕ

Современное поколение молодых программистов часто достаточно слабо знакомы с алгоритмами, которые реализует операционная система, управляя системными ресурсами и доступом к ним со стороны исполняемых программ. Высокая производительность и большие объемы памяти современных компьютеров дают возможность не задумываться об экономии этих ресурсов, а интегрированные среды разработки программного обеспечения (ПО) скрывают от своих пользователей все детали, связанные с подготовкой программ к выполнению, и не позволяют в полной мере понять такие важные вопросы, как структура адресного пространства процесса, работа с картой памяти, технология доступа к внешней памяти, методы сборки и отладки исполняемого модуля и т. д.

Операционные системы являются очень сложным программным комплексом. Достаточно сказать, что дистрибутив Linux Fedora 9 содержит более 200 млн строк исходного кода, а затраты на его разработку составляют более 60 тыс. человеко-лет, что в денежном выражении выражается суммой около 11 млрд долларов. На сегодняшний день издано много классических учебников, посвященных операционным системам, начиная с монографии У. Дэвиса [1] и заканчивая выдержавшей несколько изданий книгой Э. Таненбаума из серии «Классика Computer Science» объемом более тысячи страниц [2]. Учебники содержат очень большой объем хорошо изложенного и структурированного материала по курсу «Операционные системы», изучаемого в ведущих университетах мира в течение нескольких семестров.

В большинстве университетов России дисциплины «Операционные системы» и «Операционные системы, среды и оболочки» изучаются в течение всего одного семестра. На это ограничение накладываются сложившаяся в последнее время тенденция к сокращению лекционных занятий, а также нежелание части студентов внимательно работать с литературой и их переход на получение знаний из различных интернет-ресурсов. Совокупность этих факторов приводит к определенным трудностям в освоении материала.

Целью данного пособия является рассмотрение некоторых вопросов, обычно вызывающих затруднения при изучении операционных систем. В первом разделе рассмотрены основы командного интерфейса, второй раздел посвящен изучению основных объектов управления операционной системы — процессов и потоков, третий раздел содержит описание различных файловых систем, а в четвертом разделе рассматриваются основные этапы прохождения программ в среде операционной системы и инструментальные средства разработки ПО. В отдельный раздел вынесены вопросы, связанные с функционированием операционных систем реального времени, которые используются в системах управления сложными техническими объектами и технологическими процессами.

В конце пособия приведен краткий словарь используемых терминов. В приложения вынесены методика создания на основе легковесного дистрибутива Tiny Core Linux виртуальной Linux-машины, работающей в среде опера-

ционной системы Windows, а также таблица соответствия некоторых команд Linux и Windows.

Каждый раздел содержит теоретический материал, практические примеры и задания, а также набор вопросов для самоконтроля. Практическое задание представляет собой набор действий, выполнив которые вы сможете более глубоко усвоить теоретический материал. Для выполнения задания в дополнение к разделам пособия желательно изучить рекомендованные источники информации, содержащие методические рекомендации и примеры программ.

Все темы изучаются на примере операционной системы Linux, по семейству Windows рассмотрены только файловые системы FAT и NTFS. Операционные системы реального времени рассматриваются на примере операционной системы QNX.

Пособие рассчитано на читателя, владеющего основными понятиями информатики и вычислительной техники, содержит большое число примеров и может быть полезно преподавателям для проведения практических занятий и лабораторных работ. Изучение желательно начинать с первого раздела, остальные разделы могут изучаться в произвольной последовательности.

1. ОСНОВЫ КОМАНДНОГО ИНТЕРФЕЙСА ОПЕРАЦИОННОЙ СИСТЕМЫ

Прочитав эту главу, вы узнаете:
– формат командной строки;
– классификацию команд;
– основы программирования на языке shell;
– способы передачи данных в сценарий;
– определение следующих терминов: командный файл, сценарий, командный интерпретатор, стандартные переменные.

1.1. Общие сведения

Командный интерфейс исторически был первым типом интерфейса, предоставляемого операционной системой (ОС) пользователю, и представлял собой набор команд, каждая из которых выполняла определенное действие. Этот интерфейс требует от пользователя более глубоких знаний устройства компьютера и ОС, поэтому он прежде всего предназначен для применения IT-специалистами.

Основным достоинством командного интерфейса является доступность для пользователя абсолютно всех возможностей, заложенных в систему разработчиками ОС, а недостатком — его сложность, так как число команд может быть достаточно большим и каждая команда может иметь свой набор ключей (опций), задающих определенный режим выполнения этой команды.

Командная строка в общем случае имеет следующий формат:

команда [ключи] [аргументы]

В качестве аргументов команды выступают имена устройств, каталогов или файлов, а ключи уточняют действие команды. Ключи и аргументы не являются обязательными элементами командной строки.

Обработка командной строки выполняется интерпретатором команд, входящим с состав ОС. Для семейства Linux разработано большое количество интерпретаторов, наиболее часто используются программы **bash** или **sh**. Интерпретаторы Linux поддерживают ввод-вывод переменных, ветвления (if-then-else, case), циклы (for, while, until), работу с функциями, пошаговый режим отладки и другие возможности, характерные для языков высокого уровня. Список интерпретаторов, поддерживаемых вашей версией Linux, можно посмотреть в файле /etc/shells.

В семействе ОС Windows в течение долгого времени основным интерпретатором команд является программа **cmd,** которая существенно уступает по функциональности интерпретаторам Linux. Однако современные версии Windows имеют в своем составе новый интерпретатор **PowerShell**, предоставляющий пользователям мощный интерфейс командной строки и встроенный язык сценариев.

Часть команд интерпретатор выполняет самостоятельно, такие команды называются *внутренними*. Программный код внутренних команд содержится

внутри интерпретатора. Остальные команды реализуются программами, поставляемыми вместе с ОС в виде отдельных исполняемых файлов, и называются *внешними*. Пути поиска файлов внешних команд хранятся в системной глобальной переменной PATH.

Алгоритм исполнения команд следующий. Сначала интерпретатор пытается выполнить заданную команду самостоятельно, считая ее внутренней. Если он не находит имя команды в своей внутренней таблице, то начинает поиск исполняемого файла, имя которого совпадает с именем команды, во всех каталогах, указанных в переменной PATH. При нахождении такого файла он запускает соответствующую программу, иначе выводит сообщение об ошибке «*Файл не является внутренней или внешней командой, исполняемой программой или пакетным файлом*».

В командной строке можно запускать любые исполняемые файлы, они рассматриваются интерпретатором как обычные внешние команды.

После ввода команды интерпретатор ждет ее завершения и после этого выводит на экран системную подсказку, сообщая пользователю о готовности к приему новой команды. Возможен запуск команд в фоновом режиме, когда интерпретатор возвращает управление пользователю без ожидания их завершения, например при выполнении длительных по времени действий. В ОС Linux для выполнения команды в фоновом режиме в конце командной строки необходимо указать символ «&», а в Windows надо запускать нужную команду с помощью вспомогательной команды **start** следующим образом:

start /b команда

В одной командной строке можно записывать несколько команд, в этом случае строка называется составной, а порядок выполнения команд зависит от используемых разделителей (табл. 1.1).

Таблица 1.1

Способы формирования составной командной строки

Порядок выполнения команд	Linux	Windows
Команды выполняются последовательно	команда1; команда2	команда1 & команда2
Вторая команда выполняется только в случае успешного завершения первой команды	команда1 && команда2	команда1 && команда2
Вторая команда выполняется только тогда, когда первая команда не была выполнена	команда1 || команда2	команда1 || команда2
Команды выполняются последовательно, результаты выполнения первой команды передаются на вход второй команды	команда1 | команда2	команда1 | команда2

Примеры:

а) cd/mydir; ls/mydir — выполняется переход в каталог cd /mydir и затем выводится содержимое этого каталога;

б) cd/mydir&&ls/mydir содержимое каталога выводится только при успешном входе в него (в случае отсутствия прав доступа команда ls не выполняется);

в) cd/mydir || echoerror в случае отсутствия прав доступа к каталогу на экран выводится сообщение об ошибке;

г) cat file1 | wc — l содержимое файла file1 подается на вход команды wc, которая выводит количество строк в этом файле.

1.2. Общие сведения о файловой системе Linux

Основным отличием ОС Linux от Windows, с точки зрения пользователя, является организация файловой системы. В Linux файловая система представляет собой единую иерархическую структуру, отображаемую в виде дерева, корню которого соответствует основной (корневой) каталог, а листьям — файлы. Корневой каталог имеет имя «/» и содержит системные файлы и подкаталоги.

Под файлом в Linux понимается не только поименованная совокупность информации на ВЗУ, но и любое устройство, которое может хранить, поставлять или потреблять информацию. При этом устройство подключается (монтируется) к существующему дереву файловой системы в указанной пользователем точке с помощью команды **mount,** после чего пользователь может обращаться к любым доступным файлам, при этом в имени никак не отражается имя устройства, на котором файл находится или создается.

Linux поддерживает следующие типы файлов: регулярные, каталоги, каналы, блочные специальные, символьные специальные, ссылки. Регулярные файлы — это обычные файлы, которые содержат данные (тексты, рисунки, программный код и др.).

Канал — это временный файл ограниченного размера, информация из которого после чтения исчезает. Каналы используются для организации обмена данными между различными процессами.

Специальные файлы соответствуют устройствам (блочные — дискам, символьные — всем прочим). Введение этого понятия позволяет единообразно организовать обмен информацией с любым источником или приемником информации. Например, для чтения флеш-накопителя достаточно обычным образом открыть файл /dev/usb, позиционироваться в нем и прочесть нужное число блоков. То есть прикладные программы одинаково обмениваются информацией с обычным файлом, каналом или устройством.

В Linux вся информация о файле (размер, дата и время создания, номера выделенных блоков диска и т. д.) хранится в специальной структуре данных, которая создается для каждого файла и называется дескриптором этого файла. Все дескрипторы имеют уникальные номера (индексы), а имя файла является всего лишь ссылкой на данный дескриптор. Каталоги в Linux устанавливают соответствие между именем и номером дескриптора файла.

Возможна ситуация, когда несколько элементов каталога ссылаются на один номер дескриптора, в этом случае мы будем иметь несколько ссылок на один и тот же файл, доступ к которому может проводиться по разным именам. Такое отклонение от древовидной структуры весьма полезно, так как позволяет любому файлу дать новое имя, не дублируя информацию на диске. Такие ссыл-

ки называются жесткими, количество ссылок на файл хранится в отдельном поле его дескриптора — счетчике ссылок.

Жесткая ссылка может создаваться только для файлов и ссылается на номер дескриптора оригинального файла. Такая ссылка используется в пределах одной файловой системы, так как уникальность номеров дескрипторов обеспечивается только в пределах отдельной файловой системы. При удалении жесткой ссылки или оригинального файла счетчик ссылок в дескрипторе уменьшается на единицу, а реальное удаление файла с диска проводится только тогда, когда удаляется последняя ссылка на него и счетчик ссылок становится равным нулю. Восстановление удаленного файла в Linux невозможно.

Существуют также мягкие (символьные) ссылки, которые ссылаются на имя файла, а не на номер его дескриптора. Мягкая ссылка представляет собой обычный текстовый файл, в котором хранится имя оригинального файла. Мягкие ссылки могут указывать на объекты разных файловых систем (файлы и каталоги), так как ссылаются не на дескрипторы, а на имена этих объектов. Удаление мягкой ссылки не приводит к удалению объекта, на которую указывает эта ссылка. Удаление оригинального объекта не приводит к автоматическому удалению мягких ссылок, но делает такую ссылку неработоспособной.

Полное имя файла показывает его местоположение в файловой системе. Существуют два типа имени: *абсолютное* и *относительное*. Абсолютное всегда начинается с символа «/», обозначающего корневой каталог. Кроме того, этот символ используется и в абсолютном, и в относительном имени в качестве разделителя имен каталогов (например, /u/home/bpi). Абсолютное имя показывает положение файла по отношению к корневому каталогу файловой системы. Относительное имя показывает положение файла по отношению к текущему или домашнему каталогу.

Имена каталогов и файлов в Linux являются регистрозависимыми, т. е. имена newfile и NEWFILE обозначают разные файлы. Если имя файла начинается с символа «.» (точка), то такой файл называется скрытым и его характеристики не будут выводиться при просмотре содержимого каталогов.

Каждый файл имеет права доступа, которые определяют, кто и что может делать с содержимым файла. Возможны три категории пользователей файла — владелец (u), группа пользователей, в которую входит владелец (g), и остальные пользователи (o), а также две категории прав доступа — стандартные и специальные. Стандартными правами доступа являются чтение (r), запись (w) и выполнение (x), а к специальным правам относятся выполнение от имени владельца (s) и запрет на удаление файла (t).

Права доступа к файлу хранятся в его дескрипторе и отображаются при просмотре каталога, в котором зарегистрирован этот файл, в виде строки из 10 символов. Первый символ строки обозначает тип файла (дефис — это регулярный файл, d — каталог, l — мягкая ссылка, b — блочный специальный файл, c — символьный специальный файл). Далее в строке идут три группы, состоящие из трех символов каждая: первая группа определяет права владельца файла, вторая — права группы пользователей, третья — права остальных пользователей. В каждой из этих групп первый символ определяет право доступа

для чтения, второй — для записи, третий — для выполнения файла. Если в какой-либо позиции в этих группах выводится символ «-» (дефис), то соответствующее право доступа отключено. В таблице 1.2 представлена система определения прав доступа.

Таблица 1.2

Права доступа в Linux

Право	Обозначение	Файл	Каталог
Чтение	r	Файл можно посмотреть и скопировать	Можно посмотреть список входящих файлов
Запись	w	Файл можно изменить и переименовать	Можно создавать и удалять файлы
Выполнение	x	Файл можно запустить на выполнение (скрипты и программы)	Можно входить, делать текущим
Выполнение от имени владельца	s	Файл можно запустить на выполнение с правами его владельца	Все объекты, создаваемые внутри каталога, наследуют имя его группы
Запрет удаления чужих файлов	t	Не применяется	Можно удалять только собственные файлы

Право на выполнение от имени владельца предполагает запуск исполняемого файла пользователем, который не является владельцем этого файла и не входит в группу, включающую владельца. Наиболее часто такая передача права проводится от имени суперпользователя (root), которым обычно является администратор вычислительной системы, работающей под управлением Linux. Администратор имеет полный набор прав на любой объект файловой системы и может давать разрешение некоторым пользователям на запуск привилегированных программ без ввода администраторского пароля (например, для запуска команд **passwd**, **chown**, **chgrp** и др.). Для передачи прав используется команда **sudo**, настройки которой хранятся в файле **/etc/sudoers** (имена пользователей и групп, список выполняемых программ, необходимость введения паролей и т. д.).

Запрет на удаление файлов используется для каталогов с общим доступом, где пользователи могут создавать, читать или выполнять файлы, но не могут удалять файлы, принадлежащие другим пользователям. Примером такого каталога является каталог /tmp.

Примеры:

а) ***-rw-r--r--*** *— владелец имеет право читать и изменять файл, члены группы и остальные пользователи могут только читать файл;*

б) ***-rwx- - - - - -*** *— только владелец файла имеет право читать, изменять и выполнять файл;*

в) ***drwxr-x--x*** *— владелец может просматривать, изменять содержимое и входить в каталог, члены группы могут входить и просматривать его, все остальные — только входить;*

г) ***drwxrwxrwt*** *— владелец, члены группы и остальные пользователи могут просматривать, изменять содержимое и входить в каталог, но не могут удалять чужие файлы.*

1.3. Общие сведения о командах Linux

В данном разделе приведены наиболее часто используемые команды. Более полное описание команд можно найти в [3–6, 11].

Обратите внимание:

– в командах при указании имен файлов можно использовать метасимволы (* — заменяет любое количество символов, в том числе ни одного, ? — заменяет любой одиночный символ, [список символов] — заменяет одиночный символ из указанного списка или диапазона символов);

– в одной командной строке можно записывать несколько команд, разделяя их разделителями в соответствии с таблицей 1;

– все ключи и имена файлов в командной строке Linux являются регистрозависимыми;

– для выполнения некоторых команд необходимо иметь соответствующие права доступа;

– некоторые команды могут не поддерживаться в вашей версии командного процессора.

1.3.1. Команды для работы с каталогами

Команда **mkdir.**

Назначение: создать каталог.

Формат: mkdir имя_каталога1 [имя_каталога2...]

Пример: mkdir abcmykat

Команда **ls.**

Назначение: просмотр каталога.

Формат: ls [-опции] [путь]

Комментарий: чтобы увидеть имена скрытых файлов, используйте опцию **a**. Для получения информации о типах файлов (каталог, исполняемый файл, ссылка) используйте опцию **F**. При использовании этой опции в поле имени выводится символ, который определяет тип файла (табл. 1.3).

Таблица 1.3

Обозначения типов файлов

Тип файла	Каталог	Исполняемый файл	Мягкая ссылка	Обычный файл
Символ	/	*	@	отсутствует

Для получения подробной информации о файлах и каталогах используйте опцию **l.** При этом о каждом файле и каталоге вы получите следующую информацию: тип файла, права доступа, число ссылок, имя владельца, имя группы, размер, дата последнего изменения, имя файла или каталога.

Пример: ls –l abc

Команда **cd.**

Назначение: переход в указанный каталог.

Формат: cd [имя_каталога]

Комментарий: для перехода в домашний каталог используйте команду cd без параметров. Для перемещения по файловой системе можно использовать сокращенные имена каталогов, приведенные в таблице 1.4.

Пример: cd/u/home/bpi

Таблица 1.4

Сокращенные имена каталогов

Символ	Значение
~	Домашний каталог
.	Текущий каталог
..	Родительский каталог

Команда **pwd**.

Назначение: вывод абсолютного имени текущего каталога.

Пример: pwd

Команда **rmdir**.

Назначение: удаление каталогов.

Формат: rmdir [-опции] имя_каталога

Комментарий: для удаления каталога, содержащего файлы, используйте опцию **r**, без указания этой опции команда не будет выполняться. Для этой цели можно также применить команду

rm –rf имя_каталога

Пример: rmdir kat1 kat2

1.3.2. Команды для работы с файлами

Команда **cat**.

Назначение: просмотр текстовых файлов.

Формат: cat имя_файла

Комментарий: для просмотра больших файлов используйте команду **more**, так как она позволяет проводить постраничный просмотр файлов (страница соответствует размеру экрана).

Пример: cat file1

Команда **more**.

Назначение: постраничный просмотр текстовых файлов.

Формат: more имя_файла

Пример: more/etc/passwd

Команда **head**.

Назначение: просмотр указанного числа начальных строк файла.

Формат: head [-n] имя_файла.

Параметр: n — количество выводимых строк (по умолчанию выводится 10 строк).

Пример: head/etc/passwd

Команда **tail**.

Назначение: просмотр указанного числа конечных строк файла.

Формат: tail [-n] имя_файла

Параметр: n — количество выводимых строк (по умолчанию выводится 10 строк).

Пример: tail/etc/passwd

Команда **cp**.

Назначение: копирование файлов и каталогов.

Формат: cp [-опции] исходный_файл целевой_файл

Комментарий: команда **cp** с опцией **r** позволяет копировать каталоги вместе с входящими в них файлами и каталогами.

Примеры:

а) cp abc1 ./tmp/November копирование файла;

б) cp kat_1 kat_2 копирование каталога kat_1 в каталог kat_2

Команда **mv**.

Назначение: перемещение и переименование файлов и каталогов.

Формат: mv [-опции] старое_имя новое_имя

Пример: mv file1 file2

Команда **rm**.

Назначение: удаление файлов.

Формат: rm [-опции] имя_файла

Комментарий: если вы хотите, чтобы команда запрашивала подтверждение на удаление файла, то используйте опцию **i**.

Пример: rmfile1 file2

Команда **touch**.

Назначение: создание пустого файла.

Формат: touch [-опции] имя_файла

Пример: touch myfile

Команда **chmod**.

Назначение: изменение прав доступа к файлу или каталогу. Права доступа к файлу может изменить только владелец или суперпользователь (администратор).

Формат: chmod режим имя_файла

Здесь «режим» имеет структуру и способ записи, показанные на рисунке 1.1.

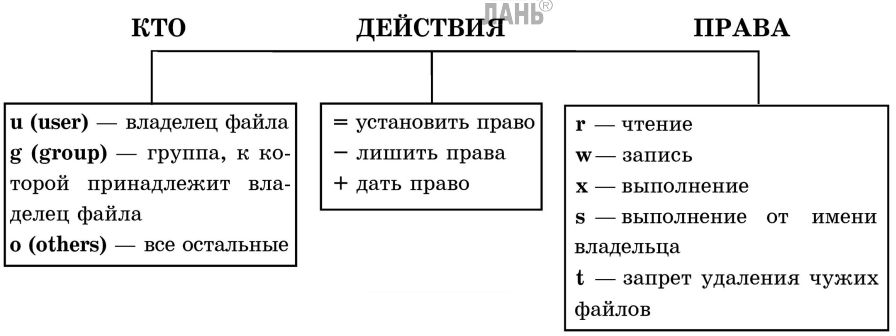


Рис. 1.1
Модель прав доступа

Права доступа могут быть заданы в команде не только в символьном виде, но и в цифровой форме (восьмеричное значение). Связь между цифровой и символьной формами приведена в таблице 1.5.

Таблица 1.5

Цифровая форма установки прав доступа

Цифровая форма	0	1	2	3	4	5	6	7
Символьная форма	---	--x	-w-	-wx	r--	r-x	rw-	rwx

Установка специальных прав доступа в цифровой форме реализуется путем добавления дополнительных битов следующим образом:

chmod 1nnn имя каталога — запрет удаления чужих файлов;

chmod 2nnn имя файла — разрешение на запуск от имени группы, с которой связан файл;

chmod 4nnn имя файла — разрешение на запуск от имени владельца файла.

Отключение специальных прав выполняется следующим образом:

chmod 0nnn имя файла или каталога — для запрета на удаление;

chmod g-s имя файла — для разрешения на запуск от имени группы;

chmod u-s имя файла — для разрешения на запуск от имени владельца.

Здесь *nnn* — стандартный набор прав доступа к файлу или каталогу.

Примеры:

а) chmod g-x june лишить членов группы права на выполнение файла ./june;

б) chmod 774 october дать все права доступа к файлу ./october владельцу и членам группы и право чтения для остальных пользователей системы;

в) chmod g+rw,o+r april или chmod 764 april добавить права чтения и записи в файл ./april членам группы пользователей и права чтения для всех остальных пользователей;

г) chmod +t mykatalog добавить запрет на удаление чужих файлов в каталоге ./mykatalog;

д) chmod 4rwx myfile разрешить запуск файла от имени владельца.

Команда **find**.

Назначение: используется для поиска и вывода имен файлов, соответствующих заданной строке символов.

Формат: find начальная_точка_поиска [-опции],

где «начальная_точка_поиска» определяет каталог, начиная с которого по всем подкаталогам будет вестись поиск.

Комментарий: команда **find** при выполнении поиска пытается войти во все каталоги, начиная с указанной начальной точки поиска, поэтому при отсутствии соответствующих прав доступа на экран часто будет выводиться сообщение об ошибке доступа. Для того чтобы отфильтровать эти сообщения, рекомендуется отправить их на фиктивное null-устройство следующим образом:

find / -name «memo» 2>/dev/null

Здесь проводится поиск во всей файловой системе файла с именем **memo**, а поток данных с сообщениями об ошибках перенаправляется на фиктивное

устройство. Напомним, что по системным соглашениям любой процесс имеет следующие стандартные потоки данных: 0 — ввод, 1 — вывод, 2 — диагностические сообщения.

Примеры:

а) Вывести на экран имена файлов из домашнего каталога и его подкаталогов, начинающихся на f:

find ~ -name «f» -print, где*

-name — после этой опции указывается имя файла, который нужно найти,

«f» — строка символов, определяющая имя файла,*

-print — опция, задающая вывод результатов поиска на экран.

б) Вывести на экран имена файлов в каталоге /etc, начинающихся с символа «p»:

find /etc -name «p» -print*

Команда **wc**.

Назначение: вывод числа символов или строк в файле.

Формат: wc [-опции] имя_файла

Примеры:

а) wc –c file1 выводит число символов в файле file1;

б) wc –l file2 выводит число строк в файле file1

Команда **ln**.

Назначение: создание ссылки на файл.

Формат: ln [-опции] имя_файла имя_ссылки

*Пример: ln folder1/file1 intro — создает в текущем каталоге жесткую ссылку с именем intro на файл folder1/file1. Для создания мягкой ссылки необходимо использовать опцию **-s***

Команда **mount**.

Назначение: подключение (монтирование) нового устройства к файловой системе.

Формат: mount [-опции] имя_файла имя_ссылки

Пример: mount –t vfat /dev/fd0 /tc — монтирует файловую систему vfat раздела fd0 в каталог /tc

1.3.3. Команды для управления сеансом работы пользователя

Команда **who**.

Назначение: вывод списка активных пользователей.

Формат: who

Комментарий: для вывода имени текущего пользователя существует команда **whoami.**

Команда **uname**.

Назначение: вывод информации о версии операционной системы.

Формат: uname

Команда **type**.

Назначение: определение типа команды (внутренняя или внешняя).

Формат: type команда

Комментарий: для внутренних команд на экран выводится сообщение о том, что команда является встроенной в командный процессор, для внешних — имя каталога, в котором находится соответствующая программа.

Пример: type find

Команда **passwd**.

Назначение: изменение пароля.

Формат: passwd

Комментарий: после ввода команды необходимо ввести старый пароль (old pasword) и дважды ввести новый пароль (new password).

Команда **exit**.

Назначение: завершение сеанса работы с командным интерпретатором.

Формат: exit

1.4. Командный сценарий

1.4.1. Общие сведения о сценариях

Современные командные интерпретаторы могут исполнять не только отдельные команды, но и программы, написанные на языке программирования **shell**. Такие программы называются *сценарием*, *скриптом* или *командным файлом* и часто используются для администрирования и управления ОС. Сценарий — это исполняемый текстовый файл, каждая строка которого является командой ОС или вызовом некоторой программы. Интерпретатор последовательно читает и исполняет все строки такого файла.

Запуск сценария Linux может проводиться несколькими способами:

– выполнить сценарий в текущем экземпляре **shell**, например:

. myscript.sh или ***source myscript.sh***

– запустить второй экземпляр интерпретатора, указав ему в качестве аргумента имя файла сценария, например:

sh myscript.sh

– выполнить сценарий как исполняемый файл, предварительно установив для него право на выполнение:

./myscript.sh

При написании сценария необходимо соблюдать следующие требования:

– в первой строке обычно указывается имя интерпретатора, который будет исполнять сценарий, например #!/bin/bash;

– в конце сценария обязательно вводится символ перевода строки (нажатие ENTER).

Каждый запущенный экземпляр командного интерпретатора имеет определенный набор стандартных глобальных переменных, значения которых можно просмотреть командами **env** или **printenv**. В таблице 1.6 приведены назначения некоторых переменных.

Пользователь может использовать в сценариях глобальные переменные интерпретатора и собственные локальные переменные. Имена переменных представляют собой регистрозависимую последовательность букв, цифр и сим-

волов подчеркивания, имя должно начинаться с буквы или символа подчеркивания. Переменные всегда имеют строковый тип и задаются сразу со значением (возможно с пустым).

Таблица 1.6

Глобальные переменные интерпретатора команд

Переменная	Значение
?	Код завершения последней команды (0 — нормальное завершение)
$	PID текущего процесса shell
!	PID последнего фонового процесса
#	Число параметров, переданных в shell
*	Список параметров shell в виде одной строки
@	Список параметров shell в виде набора слов
-	Флаги, передаваемые в shell
HOME	Имя домашнего каталога
PATH	Пути поиска исполняемых файлов
PS1	Вид системной подсказки
SHELL	Имя файла, содержащего текущий командный интерпретатор
USER	Имя учетной записи текущего пользователя

При объявлении переменной ее имя и значение должны быть записаны без пробелов относительно символа равенства. Например, создадим две переменные, присвоив переменной VAR_1 значение «245», а переменной VAR_2 пустое значение:

```
VAR_1=245
VAR_2=
```

Если для переменной задается значение, содержащее пробелы, то нужно заключать его в кавычки:

```
VAR_3=«program files»
```

В качестве значения переменной можно присвоить результат выполнения команды Linux, указав имя команды в обратных кавычках (обычно этот символ находится на клавише для ввода символов «~» и «ё»):

```
VAR_2=`pwd`
```

Значение переменным можно задать в диалоговом режиме, например команда **read ABC** присваивает значения переменным A, B и C после их ввода с клавиатуры.

Вывод значений переменных на экран проводится командой ***echo*** с указанием символа «$» перед именем переменной:

```
echo $VAR_3
```

При необходимости выполнения конкатенации значения переменной и произвольной символьной строки используются фигурные скобки:

```
echo ${VAR_3}binary
```

Командный интерпретатор имеет возможность обработки переменных как целых чисел, выполняя операции сложения (+), вычитания (–), умножения (*), целочисленного деления (/) и получения остатка от деления (%):

A=`expr $x + $y`; B=`expr $x – $y`

C=`expr $x / $y`; D=`expr $x % $y`; E=`expr $x «*» $y`

Обратите внимание: *символ умножения должен быть взят в кавычки, так как это служебный символ интерпретатора, а имена переменных и знаки операций всегда разделяются пробелами.*

Удаление переменных выполняется командой **unset**, например unset VAR_3.

Все локальные переменные доступны только в том процессе интерпретатора, в котором они были объявлены. Для того чтобы переменная стала глобальной и к ней можно было обращаться из дочерних процессов, ее следует экспортировать командой **export**. Возможны два способа экспорта:

– при создании переменной (например, export VAR_4=myfolder);

– после создания переменной (например, export VAR_4).

Команда **export** без аргументов выводит список всех экспортированных переменных интерпретатора.

Для выполнения командного сценария Linux создает отдельный экземпляр командного интерпретатора, поэтому переменные ‘*’, ‘#’ и ‘@’ имеют собственные значения для каждого запущенного сценария. Все переменные, известные сценарию во время выполнения, образуют его окружение, включающее переменные, которые сценарий получает путем наследования от родительского процесса (от интерпретатора shell), и собственные локальные переменные.

Файл сценария может быть создан любым текстовым редактором, работающим с кодировкой ASCII, а также с помощью команд **echo** или **cat** следующим образом:

echo -e «ls -l \n man echo» > echo_primer.sh

или

cat>cat_primer.sh

<команды сценария>

нажатие ENTER

нажатие Ctrl+D

Здесь файлы **echo_primer.sh** и **cat_primer.sh** создаются соответствующими командами в режиме переназначения вывода.

1.4.2. Параметры сценария

Сценарий может принимать аргументы, используя механизм формальных и фактических параметров. Для описания формальных параметров в программе используются специальные переменные, имена которых состоят из одной цифры в диапазоне от 0 до 9, перед которой стоит символ ‘$’.

При запуске сценария в командной строке указываются значения фактических параметров, которые подставляются на место соответствующих фор-

мальных параметров. Формальный параметр $0 содержит имя сценария, параметр $1 — значение первого фактического параметра, параметр $2 — второго фактического параметра и т. д. Общее число параметров содержится в переменной #.

Если число фактических параметров превышает число формальных параметров, то в сценарии можно использовать команду **shift n**, которая сдвигает влево список фактических параметров относительно списка формальных параметров на **n** позиций (по умолчанию n=1).

Обратите внимание: *при использовании команды* ***shift*** *значение параметра $0 в ОС Linux, в отличие от Windows, не изменяется.*

При загрузке системы командный интерпретатор выполняет команды, содержащиеся в файле **/etc/profile**, затем команды, содержащиеся в файле **$HOME /.profile**. Для того чтобы при входе в систему автоматически устанавливались необходимые переменные, следует откорректировать файл **.profile**, находящийся в вашем домашнем каталоге.

1.4.3. Операторы языка Shell

Оператор **test**

Оператор test проверяет выполнение заданного условия и возвращает логическое значение «истина» (0) или «ложь» (1).

Общий формат оператора:

test *условие* или [*условие*]

Вторая форма записи оператора предусматривает наличие обязательных пробелов между квадратными скобками и выражением условия. Если строка условия в команде test пустая, то возвращается значение 1.

Возможны условия нескольких видов:

1) проверка файлов или переменных

формат команды: test ключ имя_файла | имя_переменной

ключ команды задает режим проверки и может принимать следующие значения:

-f — указанный файл является обычным файлом;

-d — указанный файл является обычным каталогом;

-c — указанный файл является символьным устройством;

-r — указанный файл доступен для чтения;

-w — указанный файл доступен для записи;

-s — указанный файл не является пустым;

-n — указанная переменная имеет значение (не пустая);

-z — указанная переменная не имеет значения.

Примеры:

test –f /usr/user1/file_1 возвращает 0, если файл /usr/user1/file_1 является обычным файлом;

test –w /usr/user1/file_1 возвращает 0, если файл /usr/user1/file_1 доступен для записи;

test –n $ALFA возвращает 0, если переменная ALFA имеет значение.

2) сравнение строк

формат команды: test строка1 отношение строка2

операции отношения строк: = (равно) или != (не равно).

Примеры:

test $A = $B возвращает 0, если значение переменной A равно значению переменной B;

test $A != $B возвращает 0, если значение переменной A не равно значению переменной B.

Комментарий: операция отношения обязательно с двух сторон окружается пробелами.

3) сравнение целых чисел

формат команды: test число1 отношение число2

операции отношения чисел:

-eq — равно; -ne — не равно;

-gt — больше; -ge — больше или равно;

-lt — меньше; -le — меньше или равно.

Оператор **if**

Условный оператор if имеет следующий формат:

if *условие* then

список_операторов

[elif *условие* then

список_операторов]

[else *список_операторов*]

fi

Здесь *условие* — любой оператор, возвращающий значение true или false (обычно используется оператор *test*); *список_операторов* — операторы, разделенные точкой с запятой (;).

Пример:

```
if [ -f /tmp/myfile ] then
  cat /tmp/myfile
else
  ls /tmp > /tmp/myfile; cksum /tmp/myfile
fi
```

Оператор **case**

Оператор множественного выбора case имеет следующий формат:

case *строка* in

шаблон_1) *список_операторов;;*

шаблон_2) *список_операторов;;*

шаблон_n) *список_операторов;;*

esac

Пример:

```
echo –n ‘please enter symbol from A to C: ‘
read ENTRY
case $ENTRY in
  A|a) echo ‘Apple’;;
```

```
  B|b) echo 'Baby';;
  C|c) echo 'Coke';;
  *) echo 'Please type A, B or C ! ';;
esac
```

Оператор **for**

Оператор предопределенного цикла for имеет следующий формат:

for *имя* [in *список*] do
список_операторов
done

Если список значений для переменной не задан, то в качестве списка используется значение внутренней переменной @, содержащей список аргументов сценария (см. табл. 1.6).

Пример:

```
list=«word 1 word2 word3 word4»
for VAL in $LIST do
  echo $VAL
done
```

Операторы **while** и **until**

Операторы while и until предназначены для создания циклов с неопределенным числом исполнения списка операторов. *Список_операторов* оператора while исполняется только тогда, когда *условие* истинно, а для оператора until — когда условие ложно. Выход из циклов проводится путем контроля значения *условия* или принудительно по команде **break**.

Форматы операторов:

while условие	until условие
do	do
список_операторов	список_операторов
done	done

Использование функций в сценариях

Функции языка Shell описываются следующим образом:

```
имя_функции ()
(
  список_операторов
)
```

Функции поддерживают механизм формальных и фактических параметров. Формальные параметры являются локальными и имеют имена от 0 до 9. Если функция должна вернуть код завершения, то используется оператор *return*

1.5. Практическое задание

1.5.1. Общие сведения

В настоящее время имеются десятки различных дистрибутивов, основанных на ядре Linux и отличающихся друг от друга функциональными возможностями, используемым графическим интерфейсом и предпочтениями пользователей. Практическое задание можно выполнять с любым дистрибутивом Linux.

При выборе дистрибутива необходимо ориентироваться на цель использования (начальное изучение, домашнее применение, офисный или игровой компьютер, сервер и т. д.), требования к аппаратной части компьютера, опыт работы с Linux и предпочтительный для вас графический интерфейс. В Интернете вы можете найти множество обзоров дистрибутивов и выбрать для себя наиболее подходящий.

Если вы предполагаете работать с Linux постоянно, то можно использовать дистрибутивы Arch Linux, Debian, Manjaro. Для учебных целей достаточно установить Ubuntu или любой легковесный дистрибутив, например Tiny Core Linux или Puppy Linux, но следует иметь в виду, что командные интерпретаторы таких дистрибутивов могут выполнять ограниченное множество команд.

Дистрибутив можно установить в двух вариантах: в отдельный раздел диска или в виртуальную машину, созданную в Windows с помощью среды виртуализации Oracle VirtualBox или VMWare Player. Существуют портируемые дистрибутивы (например, AntiX или Slax), которые можно не устанавливать на жесткий диск, а просто запускать с внешнего носителя, подключенного через порт USB. В приложении 1 приведен пример установки дистрибутива Tiny Core Linux в среду виртуализации VirtualBox.

1.5.2. Задание

1. Посмотрите справку по команде **echo**, изучите ее ключи.

2. Освойте способы создания сценария, описанные в разделе 1.4.

3. Разработайте сценарий для создания файла данных с информацией по студенческой группе. Каждая запись файла должна иметь следующие поля: фамилия (0), год рождения (20), место рождения (25), средний балл (40). В скобках указано смещение поля относительно начала записи, количество записей в файле должно быть не менее 10. Данные в сценарий передавать через параметры, форматирование полей записи проводить путем циклического добавления необходимого числа пробелов, например:

```
len_fam=` expr length $fam`;
[ $len_fam –lt 20 ]
do
  #записываем пробел без перевода строки
  echo –n « « >>$file_dat;
  len_fam=`expr $len_fam + 1`;
done
```

Здесь len_fam — переменная, в которой хранится текущая длина введенного пользователем значения поля.

Ниже приведен пример содержимого файла данных

Иванов И. И.	1999	Новосибирск	4,58
Петров П. С.	2000	Черепаново	4,86
Сидоров С. К.	1999	Барабинск	3,41
Жернова А. И.	2000	Омск	4,12
Павлова Е. Ф.	1998	Томск	4,96
Ильин А. В.	2000	Владивосток	3,98

4. Разработайте сценарий для поиска в файле данных заданной пользователем строки символов.

5. Разработайте сценарий для сортировки файла данных по заданному пользователем полю записи.

6. Разработайте сценарий, формирующий меню для опроса пользователя (пункты меню — добавление новой записи, поиск данных, сортировка данных, просмотр данных, выход), и выполнение этого меню с помощью вызова соответствующих сценариев, разработанных в п. 3–5 задания.

Контрольные вопросы

1. Классификация команд операционной системы.
2. Какие команды называются внутренними и внешними?
3. Алгоритм работы командного интерпретатора.
4. Как узнать, какие интерпретаторы команд поддерживаются вашей версией Linux?
5. Командный сценарий: определение, назначение.
6. Способы запуска командного сценария.
7. Способы передачи данных в командный сценарий.
8. Переменные в командном сценарии: типы, область видимости, допустимые операции.
9. Какие виды циклов могут использоваться в сценарии?
10. Какие виды ветвлений могут использоваться в сценарии?
11. Поясните алгоритм форматирования записей при выполнении практического задания.

2. УПРАВЛЕНИЕ ПРОЦЕССАМИ

Прочитав эту главу, вы узнаете:
– классификацию процессов;
– состав процесса;
– возможные состояния задач;
– способы получения информации о процессах;
– определение следующих терминов: задание, процесс, поток, задача, примитив, зомби.

2.1. Объекты управления операционной системы

Объектами управления операционной системы являются задания, процессы и потоки.

Задание — это единица работы, выполняемой для одного пользователя. В многопользовательских ОС на обработку одновременно может поступать множество заданий, поэтому основной функцией ОС для работы с заданиями является организация очереди и ее обработка.

Если вы в среде разработки создали исходный текст программы на алгоритмическом языке и нажали на кнопку «Выполнить», то вы отправляете в систему задание, которое предусматривает выполнение нескольких этапов:

– перевод вашей программы с алгоритмического языка высокого уровня на язык двоичных машинных команд; этот этап реализуется специальной программой-компилятором, входными данными которого являются строки исходного текста, а результатом — откомпилированный (объектный) модуль;

– сборка исполняемого файла, когда ваш объектный модуль объединяется с другими ранее компилированными модулями; этот этап реализуется программой-компоновщиком, на вход которого подаются все объектные модули, а результатом является сформированный исполняемый модуль вашей программы;

– запуск вашего исполняемого модуля, на вход которого подаются необходимые для обработки данные и получается требуемый результат.

В указанном примере для выполнения одного задания требуется выполнить три программы (компилятор, компоновщик и программа пользователя), каждая из которых работает с собственными входными и выходными наборами данных.

Для каждой выполняющейся программы ОС создает *процесс*, который является для нее основным объектом управления и представляет собой объединение программы и данных. Именно процессу она должна выделить адресное пространство в оперативной памяти, процессорное время, содержимое файлов из внешней памяти и другие системные ресурсы. Для управления процессами ОС использует функции диспетчеризации, синхронизации и обработки прерываний, а также службу времени, подробное описание которых можно найти в [2].

Процесс создается при запуске любой программы независимо от того, кто запускает эту программу — пользователь или какая-то другая, уже работающая

в системе программа. Во втором случае запускающая программа называется родительской, а запускаемая — дочерней, соответствующие им процессы также называются родительским и дочерним. Процессы создаются как для пользовательских, так и для системных программ, входящих в состав ОС.

Например, любой сценарий запускается следующим образом:

– из первой строки сценария определяется необходимый интерпретатор для его выполнения;

– запускается командный интерпретатор, создавая новый процесс, в рамках которого выполняются команды сценария; при этом новый процесс наследует от родительского процесса все его окружение, включая глобальные переменные;

– родительский процесс, из которого был запущен второй экземпляр интерпретатора, приостанавливает свою работу;

– после выполнения последней команды дочерний процесс интерпретатора завершается и родительский процесс продолжает свою работу.

В компьютере одновременно выполняются множество процессов, которые могут исполняться центральным процессором (ЦП), а также процессорами видеокарты, клавиатуры, сетевой карты и т. д. Выделением процессорного времени ЦП управляет системная программа, называемая диспетчером. Она организует очередь, в которой регистрируются готовые к выполнению процессы, и обслуживает эту очередь в соответствии с заданными алгоритмами.

С точки зрения диспетчера все процессы делятся на две группы — дискретные и непрерывные. Дискретные процессы в любой момент времени могут быть прерваны и возобновлены, такие процессы называются *задачами*. Непрерывные процессы не могут прерываться и, если они были запущены, то всегда должны завершиться самостоятельно. Эти процессы называются *примитивами*. Пользовательские процессы всегда выполняются в режиме задач, а большинство программ ОС выполняются в режиме примитивов.

Для каждого процесса можно выделить образ и метаданные (рис. 2.1). Образ процесса называется потоком и представляет собой совокупность программного кода и данных, необходимых на этапе выполнения программы. Он включает следующие объекты: команды программы, данные (статические и динамические переменные программы), стек, содержимое регистровой памяти процессора, счетчик команд и приоритет.

Метаданные — это информация о процессе, которая хранится в специальных структурах данных ОС. Информация включает идентификаторы текущего и родительского процессов (PID, PPID), идентификаторы владельца и группы (UID, GID), адресное пространство, глобальные переменные, идентификаторы открытых файлов и дочерних процессов. Понятие процесса используется операционной системой для выделения ресурсов, а понятие потока — для указания команд, поочередно исполняемых в ЦП.

Каждый процесс всегда имеет три стандартных набора данных: 0 — стандартный ввод, 1 — стандартный вывод, 2 — сообщения об ошибках. Эти имена можно использовать в командах ОС. Например, команда **find / -name «unix» 2>/dev/null** при поиске файлов с именем «unix» будет пытаться найти его во

всей файловой системе, начиная с корневого каталога, и будет выводить множество сообщений об ошибках из-за отсутствия прав доступа к большинству каталогов. Для того чтобы не выводить на экран все эти сообщения, поток ошибок перенаправляется на фиктивное устройство **/dev/null**.

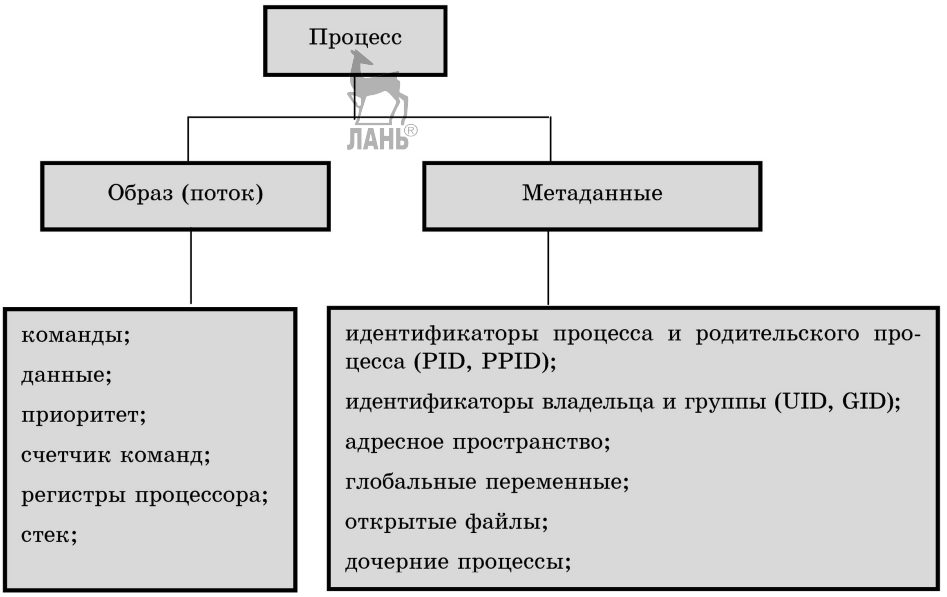


Рис. 2.1
Структура процесса

Процесс может содержать несколько *потоков* команд, достаточно независимых друг от друга. Такой процесс называется многопоточным и может рассматриваться как контейнер потоков. Все потоки при этом могут пользоваться любыми ресурсами процесса и работают в пределах одного адресного пространства. Любой поток имеет доступ к любому адресу в адресном пространстве процесса и даже может удалить информацию из стека другого потока.

Многопоточные процессы используются в следующих случаях:

– разработка приложений с архитектурой клиент-сервер, в которой возможно одновременное обращение к серверу нескольких клиентов;

– распараллеливание задач для выполнения на многоядерных процессорах и многопроцессорных ЭВМ;

– совместное использование данных: любой поток имеет свободный доступ к данным всех потоков этого процесса;

– более эффективное использование процессора: пока один поток ожидает пользовательский ввод, другой может выполнять расчеты.

Для работы с потоками в языках программирования существует специальный класс, например в C# — это класс Thread, в C++ — std::thread, в Delphi — TThread. Методы этих классов позволяют выполнять создание, временную остановку, возобновление исполнения, ожидание завершения другого потока и уничтожение потока.

На рисунке 2.2 приведены возможные состояния задач и диаграмма переходов из одного состояния в другое. В состоянии порождения для задачи создаются системные структуры данных и выделяются все требуемые ресурсы за исключением процессора, после чего она регистрируется в очереди диспетчера и переходит в состояние готовности, ожидая выделения процессорного времени.

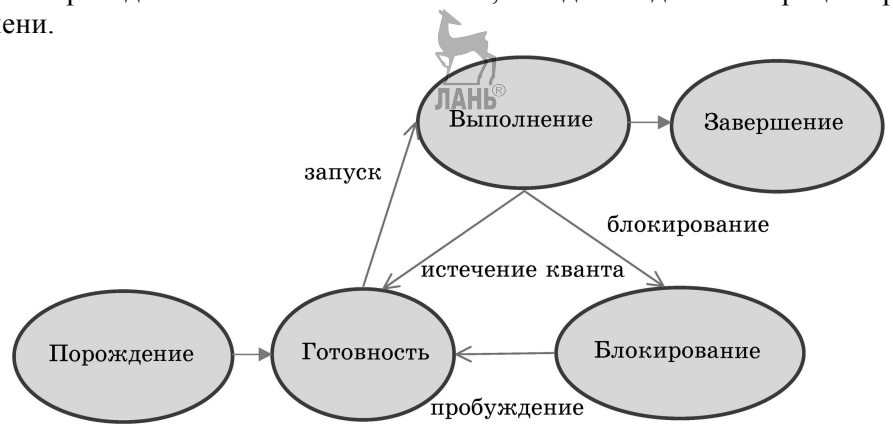


Рис. 2.2
Возможные состояния задач

В ОС общего назначения диспетчеры реализуют различные вытесняющие алгоритмы планирования задач в режиме разделения времени, когда процессор выделяется для каждой задачи на определенный интервал времени, называемый квантом. Значение кванта обычно составляет от 10 до 20 мс. При обслуживании очереди часто учитывается приоритет задачи. Если задача не успевает завершиться в течение кванта, то она переводится в конец очереди.

Когда диспетчер выделяет задаче процессорное время, она переходит в состояние выполнения, такая задача называется активной, а переход из состояния готовности в состояние выполнения называется запуском задачи.

Из состояния выполнения возможны три перехода:

– в состояние завершения, если задача успела завершиться в течение кванта, при этом ОС освобождает все выделенные ей ресурсы;

– в состояние готовности, если задача не успела завершиться в течение кванта;

– в состояние блокирования, если задаче во время выполнения требуется подождать наступления какого-либо события, например завершения чтения очередного блока данных из внешней памяти, освобождения какого-либо ресурса или завершения чтения нужной страницы памяти из файла подкачки виртуальной памяти. При наступлении ожидаемого события задача из состояния блокирования переходит в состояние готовности, этот переход называется пробуждением задачи.

Возможны ситуации, когда какая-то программа зависает, т. е. во время работы перестает реагировать на действия пользователя или на сообщения от других программ. Например, некоторое приложение запустило дочерний про-

цесс, а затем было уничтожено или неожиданно завершено, оставив после себя зависший процесс. Такое состояние процесса называется *зомби*.

Более полную информацию о процессах и потоках можно найти в [2, 6, 12].

2.2. Получение информации о процессах

Изучение характеристик процессов удобно проводить с помощью соответствующих команд ОС.

Команда **ps**

Назначение: вывод информации об активных процессах.

Формат: ps [опции]

Основные опции программы:

-e — вывод краткой информации обо всех процессах;

-a — вывод краткой информации обо всех процессах, запущенных пользователями;

-u — вывод подробной информации о процессах, запущенных пользователем;

-f — вывод расширенной информации, используется с ключами -e, -a-u;

-l — вывод подробной информации, используется с ключами -e, -a;

-x — вывод информации о процессах, отсоединенных от терминала.

При вызове команды без параметров выводится краткая информация о процессах, запущенных пользователем. Вывод команды содержит следующую информацию:

– USER — идентификатор пользователя, запустившего процесс;
– PID — идентификатор процесса;
– %CPU — интенсивность использования процессора;
– %MEM — интенсивность использования памяти;
– VSZ — размер занимаемой виртуальной памяти;
– RSS — размер занимаемой физической памяти (число страниц по 0,5 К);
– TTY — терминал, с которого был запущен процесс;
– STAT — состояние процесса;
– STIME — время старта процесса;
– TIME — использованное время процессора;
– COMAND — выполняемая команда.

Состояние процесса обозначается следующим образом:

– R — процесс в режиме выполнения;
– D — процесс находится в ожидании дисковой операции;
– I — процесс в режиме ожидания более 20 с;
– S — процесс в режиме ожидания менее 20 с;
– T — процесс остановлен;
– Z — процесс зомби.

На рисунке 2.3 приведен пример вывода команды **ps**, из которого видно, что пользователь kvg открыл два сеанса работы с сервером с терминалов pts/0 и pts/1. В первом сеансе открыты интерпретатор **bash** и два экземпляра команды **top**, во втором сеансе работают второй экземпляр интерпретатора **bash** и команда **ps**.

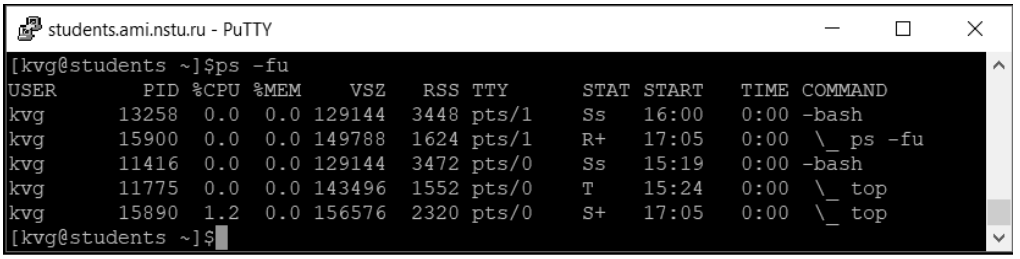


Рис. 2.3
Результат работы команды **ps**

Команда **top**

Назначение: периодический мониторинг состояния активных процессов.

Формат: top [опции]

Команда выводит на экран информацию по общей загрузке системы и по всем запущенным процессам, обновляя данные с определенным периодом времени (по умолчанию 3 с). Она часто используется системными администраторами для диагностики серверов в случае возникновения каких-либо проблем.

Основные опции программы:

-d число — задает период обновления данных о процессах, с;

-n число — максимальное число периодов обновления данных, выводимых программой;

-u имя_пользователя — отображать процессы указанного пользователя.

Завершение команды выполняется нажатием Ctrl+C, временная приостановка мониторинга — нажатием Ctrl+Z. Для продолжения работы приостановленной программы необходимо ввести команду **fg**;

Пример:

top –u user_1 –n 5 — включить мониторинг процессов, запущенных пользователем user_1 с периодом 3 с, вывести 5 итераций.

Вывод команды состоит из двух частей — заголовка, в который выводится общая информация о загрузке системы, и списка процессов (рис. 2.4).

В первой строке заголовка выводится текущее системное время, общее время работы системы, число подключенных пользователей и средняя загрузка в течение последних 1, 5 и 15 мин. Вторая строка содержит данные об общем числе процессов в системе (total), а также о числе процессов, находящихся в состоянии выполнения (running), блокирования или готовности (sleeping), завершения (stopped) и зомби (zombie). Остальные строки заголовка содержат информацию о загрузке ЦП, оперативной памяти и виртуальной памяти. Более подробную информацию по выводимым в заголовке данным можно найти в [13, 14].

Список процессов по умолчанию сортируется по степени загрузки ЦП и содержит следующие поля:

– PID — идентификатор процесса;

– USER — имя пользователя, который является владельцем процесса;

– PR — приоритет процесса;

– NI — значение «NICE», влияющее на приоритет процесса;

– VIRT — объем виртуальной памяти, используемый процессом;

– RES — объем физической памяти, используемый процессом;

– SHR — объем разделяемой памяти процесса;

– S — указывает на статус процесса: S (блокирован), R (работает), Z (зомби);

– %CPU — процент использования центрального процессора данным процессом;

– %MEM — процент использования оперативной памяти данным процессом;

– TIME+ — общее время использования процессора;

– COMMAND — имя процесса.

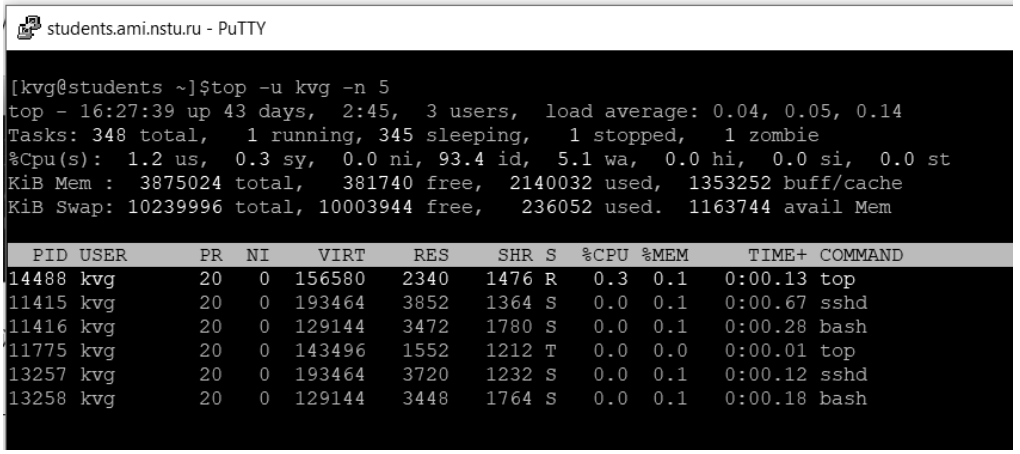


Рис. 2.4
Результат работы команды **top**

Команда **kill**

Назначение: формирование сигнала для уничтожения процесса.

Формат: kill [опции] PID

Команда используется в случае некорректной работы или зависания процесса. Механизм уничтожения основан на посылке процессу определенного сигнала, что обычно делается ядром ОС. Количество сигналов достаточно большое, их список можно посмотреть, указав опцию –l. Наиболее часто используются сигналы (в скобках указан номер сигнала):

– **SIGHUP (1) —** сообщает процессу, что соединение с управляющим терминалом разорвано, обычно отправляется системой при разрыве соединения с Интернетом;

– **SIGINT (2)** — отправляется процессу, запущенному из терминала, с помощью сочетания клавиш Ctrl+C. Процесс правильно завершает все свои действия и возвращает управление;

– **SIGQUIT (3)** — сигнал, который отправляется с помощью сочетания клавиш Ctrl+/ программе, запущенной в терминале. В этом случае программа может выполнить корректное завершение или проигнорировать сигнал, в случае завершения формируется дамп памяти;

– **SIGTERM (15)** — сигнал немедленно завершает процесс, но обрабатывается программой и поэтому позволяет ей завершить дочерние процессы, а также освободить все ресурсы;

– **SIGKILL (9) —** сигнал немедленно завершает процесс, но, в отличие от **SIGTERM**, он не передается самому процессу, а обрабатывается ядром. Поэтому занятые процессом ресурсы не освобождаются и дочерние процессы остаются запущенными.

Все сигналы могут быть направлены процессу с помощью команды **kill** путем указания в качестве опции имени сигнала или его номера. По умолчанию генерируется сигнал **SIGTERM**, использование сигнала **SIGKILL** рекомендуется только в случае неудачи с сигналом **SIGTERM**.

Обратите внимание: *для освобождения ресурсов и завершения дочерних процессов необходимо определенное время, поэтому с передачей сигнала* ***SIGKILL*** *не следует торопиться.*

Примеры:

а) kill –l — вывод списка всех доступных сигналов;

б) kill –9 1192 — принудительно завершить процесс с идентификатором 1192.

2.3. Практическое задание

1. Создайте в файле **loop.sh** сценарий, реализующий бесконечный цикл:

```
while true
do
  true
done
```

2. Из файла **/etc/shells** определите имена командных интерпретаторов, которые установлены в вашей системе Linux.

3. Определите имя текущего интерпретатора команд. Это можно сделать с помощью команд **echo $0** или **echo $SHELL**.

4. Запустите мониторинг процессов командой **top** с периодом вывода 4 с и определите общее количество процессов в системе, а также распределение процессов по их состояниям (активные, блокированные и т. д.).

5. Определите загрузку процессора в течение последних 5 мин.

6. Запустите сценарий **loop.sh** в фоновом режиме с помощью второго экземпляра текущего интерпретатора. Например, если текущим интерпретатором является **sh,** то это действие реализуется командой **sh loop.sh&**

7. С помощью команды **ps** в режиме расширенного вывода посмотрите список ваших активных процессов и общий список процессов, поясните результаты. Сравните информацию по вашим процессам с результатами, полученными из команды **top**.

8. С помощью команды **source** повторно запустите в фоновом режиме сценарий **loop.sh** и посмотрите новый список ваших процессов, сравните результаты с полученными в п. 6.

9. Запустите сценарий **loop.sh** с помощью интерпретатора, не являющегося текущим (например, с помощью **bash**).

10. Прервите выполнение сценариев, запущенных ранее, с помощью сигналов **SIGINT, SIGTERM** и **SIGKILL.**

Контрольные вопросы

1. Дайте определение и поясните область применения понятий процесса и потока.
2. Назовите признаки, по которым можно классифицировать процессы.
3. Основные этапы взаимодействия родительского и дочернего процессов.
4. Назовите стандартные потоки данных процесса, приведите примеры.
5. Укажите действия, которые можно выполнять с потоком.
6. Какие характеристики процесса относятся к метаданным?
7. Возможные состояния задач и примитивов.
8. Как просмотреть информацию о процессах?

3. ФАЙЛОВЫЕ СИСТЕМЫ СОВРЕМЕННЫХ ОПЕРАЦИОННЫХ СИСТЕМ

Прочитав эту главу, вы узнаете:

– физическую и логическую модели диска;

– структуру системной области диска для различных файловых систем;

– принципы хранения и доступа к файлам в операционных системах Windows и Linux;

– инструментальные средства анализа файловых систем;

– определение следующих терминов: файловая система, раздел, таблица разделов, блок, кластер, менеджер логических томов, индексный дескриптор.

Термин «*файловая система*» имеет два значения: а) способ хранения информации на внешнем запоминающем устройстве (ВЗУ) и организация доступа к ней; б) компонент ОС, реализующий весь комплекс действий по работе с информацией, хранящейся на ВЗУ (распределение внешней памяти, контроль прав доступа, запись, удаление и т. д.). В данном пособии мы будем рассматривать файловую систему как способ хранения и организации доступа.

Устройства внешней памяти современных компьютеров очень разнообразны. К ним относятся различные типы дисков (магнитные, оптические, магнитооптические), USB флеш-накопители, карты памяти и т. д. Логическая модель всех устройств основана на модели магнитных дисков как наиболее часто используемых ВЗУ, поэтому остановимся именно на этом классе устройств. Рассмотрим сначала общие принципы хранения данных на дисках, а затем особенности хранения для различных операционных систем.

3.1. Модель внешней памяти

3.1.1. Физическая модель диска

Информация на диске размещается вдоль концентрических окружностей, называемых *дорожками*. Дорожки с одинаковыми номерами на различных *поверхностях* диска образуют *цилиндр*, для записи и чтения каждой дорожки диска выделяется отдельная магнитная головка. Дорожка содержит определенное количество *секторов*, под которым понимается участок дорожки, хранящий минимальный объем информации, который может быть считан или записан за одно обращение к диску (обычно 512 байт). Дорожки нумеруются в пределах поверхности, а сектора — в пределах дорожки.

Возможны два способа разбивки дорожек на сектора — с постоянным количеством секторов на дорожке и с переменным количеством секторов на дорожке. В первом способе каждая дорожка диска имеет одинаковое число секторов, что обеспечивается изменением плотности записи при переходе с одной дорожки на другую. Максимальная плотность записи используется на дорожках с минимальным радиусом. Для получения доступа к произвольному сектору диска необходимо указать его *физический адрес*, который включает номер цилиндра (cylinder), номер головки (head) и номер сектора (sector). Такой способ

адресации называется *CHS* и может использоваться только для дисков небольшого объема.

Современные диски большого объема используют второй способ разбиения, при котором дорожки на каждой поверхности диска группируются в зоны. Все дорожки в одной зоне имеют одинаковое количество секторов, дорожки в зонах с меньшим радиусом имеют меньше секторов, чем зоны с большим радиусом, а плотность записи остается постоянной. Таким образом, без изменения технологии производства увеличивается общее число секторов на диске и, следовательно, его объем. При этом используется линейная адресация всех секторов диска (LBA). В настоящее время для задания LBA-номера сектора используется 6 байт, что дает возможность адресовать на диске 2^{48} секторов.

Физическая модель диска включает понятия дорожки, цилиндра, поверхности и сектора и используется его *контроллером*. Диски могут отличаться друг от друга по набору этих параметров и поэтому могут иметь разные физические модели. Синонимом термина «*физический диск*» является термин «*физический том*».

Обратите внимание: *программы, предназначенные для обслуживания дисков, использующих LBA-адресацию, получают от контроллеров дисков не реальную, а фиктивную геометрию, определяемую ограничениями интерфейсов, но не соответствующую действительности. Обычно они сообщают, что на каждой поверхности диска имеется 255 дорожек, каждая из которых содержит 63 сектора.*

3.1.2. Логическая модель диска

С логической точки зрения все адресное пространство диска представляет собой набор последовательно пронумерованных секторов. Небольшая часть этих секторов выделяется для хранения служебной информации и называется системной областью диска, остальные сектора предназначены для хранения файлов и каталогов и образуют область данных. С логической моделью диска работает операционная система.

Единица размещения дисковой памяти называется блоком, который состоит из одного или нескольких секторов. Файлы на диске могут храниться в несмежных блоках. Размер блока в Linux обычно по умолчанию равен 1 Кбайт (2 сектора), в Windows — 4 Кбайт (8 секторов). В Windows вместо термина «*блок*» часто применяется термин «*кластер*».

Все линейное дисковое пространство обычно делится на несколько частей — разделов (partitions). В один раздел объединяется группа смежных цилиндров, для каждого раздела необходимо хранить информацию о его начале и конце, т. е. номера первого и последнего из задействованных в нем цилиндров или секторов. Разделение всего дискового пространства на разделы полезно по нескольким причинам:

– уменьшаются «дальние перемещения» головок чтения/записи и увеличивается скорость выполнения операций чтения и записи;

– появляется возможность структурирования данных (например, можно отделить файлы пользователя от файлов ОС);

– на одном диске можно установить несколько ОС;

– на одном диске можно хранить информацию в разных файловых системах или в одинаковых файловых системах, но с разным размером кластера.

Структура данных, хранящая информацию о логической организации диска вместе с небольшой программой, которая находит и загружает в оперативное запоминающее устройство (ОЗУ) программу загрузки ОС, называется главной загрузочной записью (Master Boot Record, MBR) и располагается в самом первом секторе диска, т. е. в секторе с физическим адресом **0-0-1.**

В MBR расположена также таблица разбиения диска на разделы (*partition table*), которая содержит четыре записи по 16 байт. Каждая запись может хранить информацию по одному разделу, который называется первичным. Таким образом, число первичных разделов на диске ограничено, их не может быть более четырех. По мере увеличения объема дисковой памяти стало ясно, что четырех разделов мало, поэтому разработчиками был предложен механизм расширенных и логических разделов, основанный на том, что один из первичных разделов объявляется расширенным и в нем создаются логические разделы.

Расширенные разделы не используются для хранения данных, они могут лишь хранить информацию о логических разделах. Первый сектор расширенного раздела хранит расширенную таблицу разделов (EBR) с четырьмя записями: первая используется для описания логического раздела, вторая описывает следующий расширенный раздел, а две другие не используются и заполняются нулями. Следующий расширенный раздел имеет свою таблицу EBR, в которой также используются только две записи, описывающие один логический и один расширенный разделы. Последний расширенный раздел использует только одну из четырех записей таблицы разбиения. Каждая таблица EBR занимает 1 сектор, между EBR и логическим разделом оставлены свободными 19 секторов, которые не входят в адресное пространство логических разделов.

Пример. На рисунке 3.1 приведена структура таблиц размещения разделов на примере диска, имеющего три первичных и два логических раздела, и реализация этой структуры в ОС Windows для диска размером 1000 блоков.

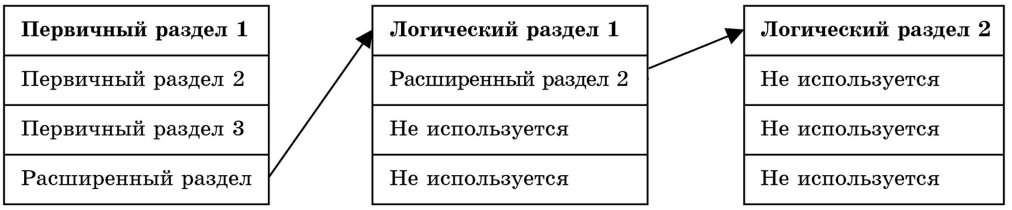
В семействе ОС UNIX диски и их разделы представляются пользователю в виде специальных файлов, зарегистрированных в каталоге /dev. Имена этих файлов состоят из трех элементов — префикса, собственного имени диска и номера раздела:

– для дисков с интерфейсом Parallel ATA (PATA) префикс имеет значение **hd,** для современных дисков с интерфейсом Serial ATA (SATA) — **sd**;

– собственные имена дисков обозначаются буквами латинского алфавита **a, b, c** и т. д. (аналогично дискам Windows);

– разделы идентифицируются порядковыми номерами; цифры с 1-й по 4-ю отведены под первичные разделы, а логические разделы нумеруются, начиная с цифры 5.

Поэтому файлы для дисков PATA именуются как **hda**, **hdb** и т. д., а для дисков SATA — **sda**, **sdb** и т. д. Имена разделов соответственно могут быть **sda1, sda2** и т. д.



Главная загрузочная запись

Раздел	Начало	Размер	Тип
C	0	200	первичный
D	200	200	первичный
E	400	200	первичный
	600	400	расширенный

Расширенная загрузочная запись раздела F:

Раздел	Начало	Размер	Тип
F	620	180	логический
	800	200	расширенный

Расширенная загрузочная запись раздела G:

Раздел	Начало	Размер	Тип
G	820	180	логический

Общая структура физического диска

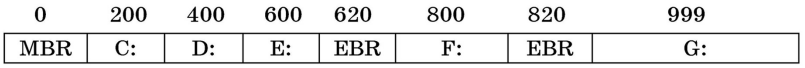


Рис. 3.1
Пример разбиения диска на разделы

Пример. Если на SATA-диске имеются один первичный раздел и три логических раздела, то соответствующие им устройства будут именоваться так:

/dev/sda1 — первичный раздел;

/dev/sda2 — расширенный раздел;

/dev/sda5, /dev/sda6, /dev/sda7 — логические разделы.

Дисковые разделы могут создаваться не только для разделения дисковой памяти, но и для объединения, в результате которого физические диски или их отдельные части предстают перед системой в качестве непрерывного дискового пространства, на котором может быть создана единая файловая система. Существуют два основных способа объединения — организация дисковых массивов и применение менеджера логических томов.

Технология дисковых массивов (RAID) используется для объединения нескольких дисков в один логический объект для избыточности и повышения производительности и поддерживается большинством современных ОС. Существует более 10 различных спецификаций RAID. Например, RAID уровня 1 предназначен для увеличения надежности хранения данных и реализуется путем зеркалирования (дублирования) дисков.

Менеджер логических томов (LVM) — компонент Linux, реализованный с помощью подсистемы devicemapper (dm), позволяющий использовать разные

области одного жесткого диска или области различных жестких дисков как один логический том. LVM позволяет:

– иметь файловую систему, которая превышает размер наибольшего диска;

– добавлять диски или разделы в дисковую группу и расширять существующие файловые системы на лету;

– уменьшить размеры файловых систем и удалить диски из дисковой группы, когда их емкость больше не требуется;

– создавать резервные копии на основе мгновенных копий файловой системы.

В LVM используются понятия *физического тома* (physical volume, PV), *группы томов* (volume groups, VG) и *логического тома* (logical volume, LV). Физические тома являются физическими дисками или разделами физических дисков, группы томов объединяют физические тома. Группа томов может быть логически разделена на логические тома, которые для пользователей представляются физическими дисковыми разделами.

Пример. На рисунке 3.2 показан пример отображения физической структуры томов в логическую. Здесь четыре раздела физического диска **hda** и два физических диска (**hdb** и **hdc**) отображаются в группу томов VG0, в которой создан логический том LV0 и оставлено свободное место для других логических томов или для последующего роста LV0.

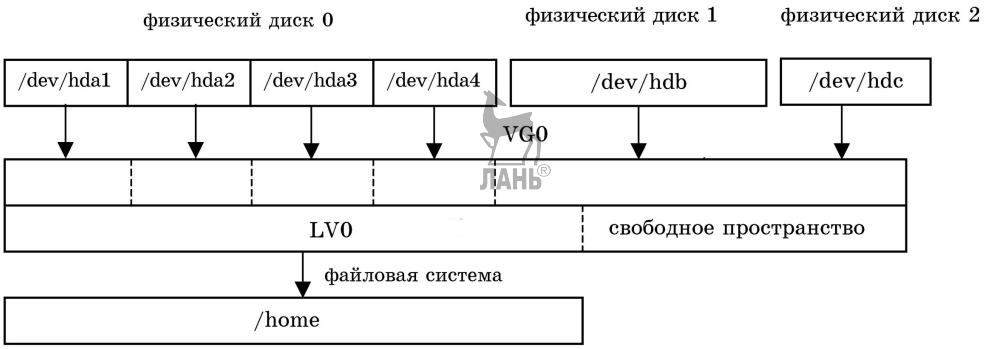


Рис. 3.2
Пример использования LVM

Файлы блочных логических томов LVM хранятся в каталоге /dev/mapper. Каждый раздел дисковой памяти, включая логические тома, имеет собственную файловую систему и характеризуется следующими основными параметрами:

– имя, используемое ядром Linux, например dm-1;

– тип (первичный, расширенный, логический);

– тип установленной файловой системы;

– точка монтирования — имя каталога, к которому подключается раздел;

– имя устройства — имя, под которым раздел представлен пользователю, например centos-tmp;

– номер устройства, состоящий из двух частей: старший номер (major) идентифицирует драйвер, ассоциированный с устройством, а младший номер (minor) указывает номер устройства. Например, диск sda может идентифициро-

ваться парой номеров 8:0, а его разделы sda1 и sda2 — номерами 8:1 и 8:2 соответственно;

– тип устройства, например VirtualCD/ROM.

На рисунке 3.3 приведен пример разбиения на разделы физического диска sda, где NAME — имя раздела, KNAME — внутреннее имя раздела, используемое ядром Linux, FSTYPE — тип файловой системы, TYPE — тип раздела, SIZE — размер, MOUNTPOINT — точка монтирования раздела. Из примера видно, что:

– физический диск sda разбит на два первичных раздела sda1 и sda2 (тип *part*);

– sda1 является загрузочным разделом (точка монтирования */boot*);

– sda2 отдан под управление LVM и разбит на четыре логических тома, имеющих внутренние имена dm-0 – dm-3.

```
[kvg@students ~]$lsblk -o NAME,KNAME,FSTYPE,TYPE,SIZE,MOUNTPOINT
NAME            KNAME FSTYPE      TYPE  SIZE MOUNTPOINT
fd0             fd0               disk    4K
sda             sda               disk  127G
├─sda1          sda1  xfs         part  500M /boot
└─sda2          sda2  LVM2_member part 97.7G
  ├─centos-swap dm-0  swap        lvm   9.8G [SWAP]
  ├─centos-root dm-1  ext4        lvm  39.1G /
  ├─centos-tmp  dm-2  ext4        lvm   9.8G /tmp
  └─centos-home dm-3  ext4        lvm  39.1G /home
sr0             sr0               rom  1024M
```

Рис. 3.3
Пример разбиения диска

В последнее время вместо разбиения на основе таблицы MBR начинает использоваться новый формат размещения информации о разделах в виде таблицы GPT (GUID Partition Table, таблица с глобальными идентификаторами), которая может хранить данные по размещению 128 первичных разделов.

3.2. Файловые системы операционной системы Windows

3.2.1. Файловая система FAT

Файловая система FAT была основной в ранних версиях ОС Windows, а в настоящее время она используется для различных внешних устройств, подключаемых к компьютеру (флеш-накопители, карты памяти и т. д.).

На рисунке 3.4 приведена логическая модель диска с файловой системой FAT. В системной области находятся MBR, таблица размещения файлов и корневой каталог. MBR имеет размер 512 байт, всегда хранится в нулевом секторе и используется в процессе загрузки операционной системы.

Загрузочная запись	Таблица размещения файлов (2 копии)	Корневой каталог	Кластеры данных (файлы и каталоги)

Рис. 3.4
Логическая модель диска FAT

Таблица размещения файлов, в оригинальной литературе называемая FAT (File Allocation Table), содержит информацию о размещении файлов в области данных. Она всегда занимает сектора, начиная с первого. На любом диске для обеспечения надежного доступа к данным всегда хранятся два экземпляра FAT, которые обновляются одновременно.

Таблица размещения файлов содержит информацию о номерах кластеров, выделенных для хранения каждого файла. Она представляет собой карту (образ) области данных, в которой описывается состояние каждого кластера диска. Размер таблицы зависит от объема диска. Номер начального кластера, выделенного файлу, записывается в элемент каталога, в котором зарегистрирован этот файл.

Каждый элемент таблицы соответствует одному кластеру в области данных. Дефектные кластеры помечаются как «bad». Если кластер свободен, то соответствующий ему элемент FAT имеет значение «0». Если кластер выделен для какого-либо файла, то возможны два варианта:

а) элемент содержит признак конца файла «EOF», если этот кластер является последним кластером, выделенным файлу;

б) элемент содержит значение номера следующего кластера, выделенного файлу.

Таким образом элементы FAT, выделенные одному файлу, связываются в цепочки, позволяющие получить доступ к информации даже в том случае, если файл при записи разбивается на несколько фрагментов, хранящихся в несмежных кластерах диска.

Элементы FAT могут быть 16- или 32-разрядными, в зависимости от этого файловые системы имеют названия FAT-16 или FAT-32 и работают с 16- и 32-разрядными дисковыми адресами соответственно. При этом признаки конца файла будут равны FF FF или FF FF FF FF.

Логическое разбиение области данных на кластеры как совокупность секторов взамен использования одиночных секторов имеет следующий смысл:

– уменьшается размер FAT;

– уменьшается возможная фрагментация файлов;

– ускоряется доступ к файлу, так как в несколько раз сокращается длина цепочек фрагментов дискового пространства, выделенных для него.

Следует иметь в виду, что при увеличении размера кластера ухудшается коэффициент использования дисковой памяти за счет увеличения внутренней фрагментации. Минимальный размер кластера на диске с файловой системой FAT зависит от объема диска ($V_{диска}$) и разрядности элемента FAT (r):

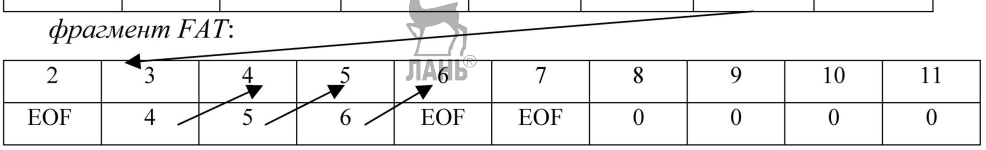
$$V_{кл\_min} = V_{диска}/2^r.$$

Пример. Файл Mst.dat размером 14 150 байт занимает на диске кластеры 3, 4, 5 и 6. Приведем элемент каталога, в котором зарегистрирован этот файл, и фрагмент FAT, а также связь между ними:

элемент каталога:

Имя	Тип	Атрибут	Резерв	Время создания	Дата создания	Номер нач. кластера	Размер файла
Mst	dat	r		12-50	01.10.18	3	14150

фрагмент FAT:



2	3	4	5	6	7	8	9	10	11
EOF	4	5	6	EOF	EOF	0	0	0	0

Корневой каталог — главный каталог диска, который занимают сектора, следующие за FAT. Основным отличием корневого каталога от всех других каталогов является фиксированное число элементов, что связано с его расположением в системной области, которая создается на этапе форматирования диска. Записи корневого каталога имеют длину 32 байта, структура записей представлена в таблице 3.1. Если файл не имеет расширения, то в соответствующем поле хранятся пробелы. Дата и время используются в виде четырехбайтового значения в операциях сравнения. Номер начального кластера определяет дисковый адрес файла и точку входа в FAT для этого файла.

Таблица 3.1

Структура элемента каталога

Номер поля	Длина (байт)	Назначение поля
1	8	имя файла
2	3	расширение имени
3	1	атрибуты
4	10	резерв
5	2	время создания/модификации
6	2	дата создания/модификации
7	2	номер начального кластера
8	4	размер файла

Байт атрибутов файла задает его статус в соответствии с таблицей 3.2. Если байт атрибута равен 8, то метка тома хранится в полях имени и расширения файла элемента корневого каталога. Если элемент каталога указывает на подкаталог, то используются все поля элемента каталога, при этом последнее поле (размер файла) имеет нулевое значение.

Таблица 3.2

Значения атрибутов файла

Значение байта атрибутов	Характеристика файла
1	только чтение
2	скрытый файл
4	системный файл
8	элемент каталога хранит метку тома (11 байт)
16	элемент каталога указывает на подкаталог
32	измененный файл (архивный)

Подкаталоги имеют структуру, аналогичную корневому каталогу, только, в отличие от него, они не имеют фиксированного размера и фиксированного дискового адреса, т. е. хранятся на диске в области данных как обычные файлы. Для того чтобы в процессе работы существовала возможность перемещения по дереву каталогов, в подкаталогах предусмотрено существование двух элементов с именами «.» (одна точка) и «..» (две точки). Первый элемент каталога является указателем на текущий каталог, а второй — на родительский каталог. Соответственно поле «Номер начального кластера» этих элементов содержит дисковые адреса этих каталогов. Если это поле содержит «0», то это означает, что каталог-родитель является корневым каталогом диска.

Файловая система FAT для каждого файла и подкаталога хранит два имени — длинное и короткое. Хранение длинных имен файлов организуется в специально отформатированных записях каталога, у которых байт атрибутов равен **0F**. Такие записи каталога не видны для старых 16-разрядных программ, работающих только с короткими именами. Структура записи каталога для хранения длинных имен приведена в таблице 3.3, откуда видно, что одна такая запись может хранить до 13 символов в кодировке Unicode.

Для регистрации файла с длинным именем в каталоге выделяется необходимое количество специальных записей, а также одна стандартная запись для хранения короткого имени. Блок специальных записей всегда располагается в каталоге перед стандартной записью, поэтому если к каталогу обращается старая 16-разрядная программа, то она будет видеть только короткое имя файла, а 32-разрядные Windows-приложения могут работать с длинными именами.

Таблица 3.3

Структура элемента каталога FAT для хранения длинного имени

Номер поля	Длина (байт)	Назначение поля
1	1	порядок следования
2	10	первые 5 символов имени
3	1	атрибуты (0F)
4	1	указатель типа
5	1	контрольная сумма
6	12	следующие 6 символов имени
7	2	номер начального кластера (всегда 0)
8	4	последние 2 символа в имени

Короткое имя образуется из длинного следующим образом: оставляется 6 символов длинного имени, к которому дописываются знак «~» (тильда) и порядковый номер в пределах каталога. Например, для регистрации файла с именем «Курсовой проект.doc» в каталоге будут выделены две специальные записи и одна стандартная, которая будет хранить имя «Курсов~1». Применение порядкового номера обеспечивает доступ к файлам, у которых первые 6 символов имени совпадают.

При удалении файла файловая система выполняет следующие действия:

– в таблице размещения файлов обнуляются все элементы, выделенные для описания расположения этого файла;

– в соответствующем элементе каталога изменяется имя файла — вместо первого символа в поле имени записывается символ «x».

Остальные характеристики файла в элементе каталога, а также содержимое файла в кластерах диска не изменяются, поэтому всегда есть возможность полностью или частично восстановить удаленный файл. Полное восстановление возможно, если:

– не перезаписан соответствующий элемент каталога;

– имеется доступ к каталогу;

– кластеры, ранее занимаемые файлом, не выделены другим файлам или каталогам;

– удаленный файл был нефрагментированным.

При несоблюдении последнего условия полное восстановление не гарантируется, так как не всегда возможно извлечь данные о том, какие кластеры были выделены файлу.

ОС Windows обычно проводит удаление файлов в специальный скрытый каталог, который называется корзиной. Размер корзины может устанавливаться пользователем. Корзина обслуживается специальной программой, что делает восстановление ошибочно удаленных файлов удобным и быстрым.

3.2.2. Файловая система NTFS

В настоящее время основной файловой системой ОС Windows является файловая система NTFS. Каждый раздел NTFS организован в виде последовательности кластеров размером до 64 Кбайт (по умолчанию размер кластера равен 4 Кбайт) и содержит каталоги, файлы, битовые массивы и другие структуры данных [15, 16]. Основным отличием NTFS от файловой системы FAT является хранение системных структур данных в виде обычных файлов. Например, таким образом хранятся корневой каталог, битовый массив использованных блоков, определения атрибутов файлов и т. д. Имена системных файлов начинаются с символа «$».

На рисунке 3.5 показана организация раздела NTFS. В первом блоке раздела находится загрузочная запись ($Boot), в которой содержится программа загрузки и информация о разделе (тип файловой системы и адреса основных системных файлов). Загрузочная запись занимает обычно 8 Кбайт (16 первых секторов).

Основной структурой данных в NTFS является главная таблица файлов (Master File Table, MFT), в которой регистрируются все файлы раздела, включая системные файлы. Таблица хранится в файле $MFT и представляет собой единый каталог, для которого резервируется 12% от общего объема раздела в виде непрерывной последовательности блоков, называемой MFT-зоной. Запись файлов и каталогов в эту зону не проводится, а ее адрес хранится в загрузочной записи.

MFT состоит из записей размером 1 Кбайт о файлах, расположенных в разделе. Каждая запись имеет уникальный номер — индекс, общее количество записей — до 2^{48}. В записи хранится вся информация о файле (имя, дата и время создания, размер, положение на диске отдельных фрагментов и т. д.). Если

не хватает одной записи MFT, то используется несколько, причем не обязательно подряд. При этом первая запись называется базовой.

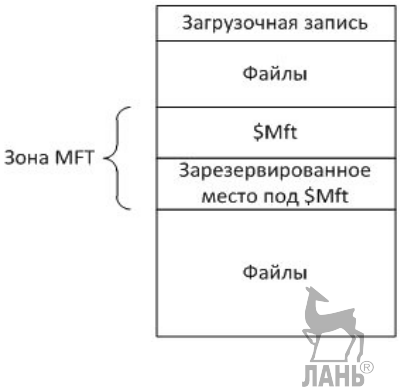


Рис. 3.5
Организация диска NTFS

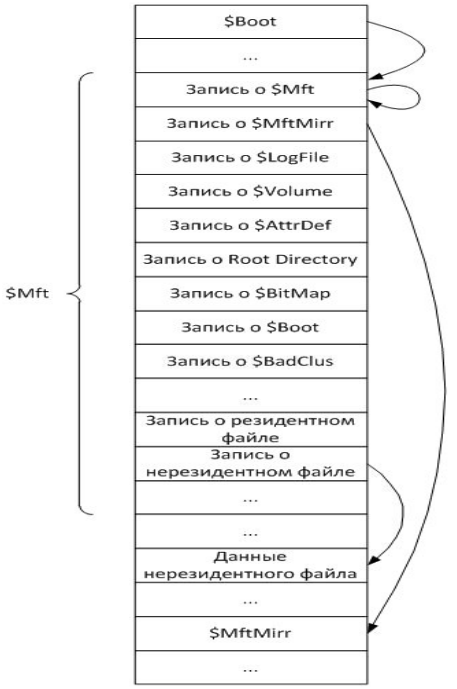


Рис. 3.6
Структура файла $MFT

Структура файла $MFT показана на рисунке 3.6. Первые 16 записей выделены для хранения информации о системных файлах. Самая первая запись в MFT — это запись о самом файле $MFT. Во второй записи содержится информация о файле $MFTMirr, в котором дублируются первые 4 записи таблицы MFT. В случае возникновения сбоя, если MFT окажется недоступным, информация о системных файлах будет считываться из файла $MFTMirr, адрес которого также имеется в загрузочной записи.

Ниже приведено назначение некоторых системных файлов NTFS:

– $LogFile — файл журнала, в котором записывается информация обо всех операциях, изменяющих структуру раздела NTFS, например создание файлов и каталогов. Файл журнала используется при восстановлении тома NTFS после сбоев;

– $Volume — файл информации о томе, в котором содержатся имя тома (Volume label), версия NTFS и набор флагов состояния тома, например флаг, установка которого означает, что том был поврежден и требует восстановления при помощи системной утилиты Chkdsk;

– $AttrDef — таблица определения атрибутов, содержащая возможные на данном томе типы атрибутов файлов;

– Root Directory — файл с информацией о корневом каталоге тома. В нем хранятся ссылки на файлы и каталоги, содержащиеся в корневом каталоге;

– $BitMap — файл битовой карты, каждый бит в которой соответствует одному кластеру: единичное значение бита соответствует занятому кластеру, нулевое — свободному;

– $Boot — файл загрузочной записи тома;

– $BadClus — файл плохих кластеров, содержащий информацию обо всех кластерах, имеющих сбойные секторы.

Структура записи MFT

Файловая запись MFT, структура которой приведена на рисунке 3.7, всегда располагается в начале сектора; ее первые байты кодируют слово «FILE» (ASCII-коды 46 49 4C 45), а конец записи определяется 4-байтовой последовательностью FF FF FF FF. Запись состоит из заголовка и набора атрибутов. В заголовке содержится служебная информация о записи (например, ее тип и размер), а все данные, относящиеся непосредственно к файлу, хранятся в виде атрибутов.

Рис. 3.7
Структура записи MFT

Каждый атрибут имеет заголовок, определяющий тип атрибута и его свойства, и тело, содержащее основную информацию атрибута. Названия атрибутов, как и системных файлов, начинаются с «$», например имя файла ($FILE_NAME), информация о его свойствах ($STANDARD_INFORMATION), данные файла ($DATA). Физически атрибут файла хранится в виде простой последовательности байтов.

По расположению относительно MFT атрибуты бывают резидентные и нерезидентные. Резидентные атрибуты полностью помещаются в файловую запись MFT, нерезидентные атрибуты хранятся вне MFT. Область, в которой расположен нерезидентный атрибут, называется группой. Поскольку нерезидентных атрибутов в файле может быть несколько, то и групп бывает тоже несколько. Множество групп файла называется списком групп (RunList). Файловая запись при наличии нерезидентных атрибутов содержит ссылку на расположение группы на диске.

Некоторые поля заголовка файловой записи, а также резидентных и нерезидентных атрибутов представлены на рисунке 3.8. На том же рисунке справа показан пример файловой записи с конкретными значениями рассматриваемых полей. Числа слева от полей записи обозначают шестнадцатеричное смещение поля от начала записи.

В начале файловой записи находится признак ее начала — слово «FILE» (46 49 4C 45). По смещению 0x14 расположено двухбайтовое поле, в котором записано смещение первого атрибута относительно начала файловой записи. В примере в этом поле записано 38, т. е. первый атрибут расположен по смещению 38.

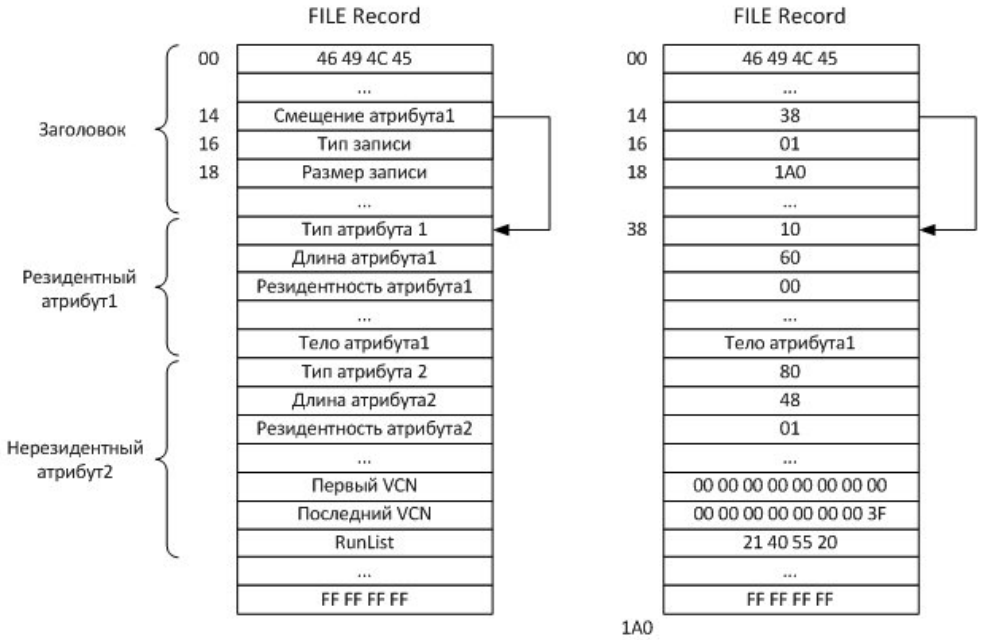


Рис. 3.8
Пример записи MFT

В следующем поле хранится тип файловой записи: значение 01 обозначает файл, 02 — каталог (directory). В примере файловая запись соответствует файлу (значение 01 по смещению 16). Еще одно поле в заголовке содержит размер всей записи. В примере на рисунке 3.7 в этом поле записано 1A0, т. е. размер записи составляет 416 байт.

Каждый атрибут имеет поля, указывающие тип, длину и резидентность атрибута. Все типы атрибутов имеют свои численные значения, например атрибуту $FILE_NAME соответствует значение 0x30, атрибуту $STANDARD_INFORMATION — 0x10, атрибуту $DATA — 0x80.

Если атрибут резидентный, то в поле резидентности записывается 0x00, иначе — 0x01. В случае нерезидентного атрибута предусмотрены поля для хранения номеров кластеров, в которых располагается группа или несколько групп, выделенных для размещения файла.

В примере на рисунке 3.8 показаны два атрибута. Первый атрибут имеет тип $STANDARD_INFORMATION (значение 10), длина атрибута 96 байт ($60_{16} = 96$), атрибут является резидентным (00). У второго атрибута тип $DATA (80), длина — 72 байта ($48_{16} = 72$), атрибут является нерезидентным (01).

Для нумерации кластеров используются два типа номеров — логический номер кластера (LCN) и виртуальный номер кластера (VCN). Логический номер является номером кластера в пределах всего раздела и используется для поиска начального кластера группы. Виртуальный номер обозначает порядковый номер кластера внутри группы.

В случае нерезидентных атрибутов в заголовке атрибута содержатся следующие поля: номер VCN первого кластера группы (обычно равен 0x00), номер VCN последнего кластера группы и список групп (RunList), описывающий расположение групп на диске. В примере, приведенном на рисунке 3.8, значения этих полей следующие:

– первый VCN = 0x00;

– последний VCN = 0x3F;

– список групп (RunList) = 0x21 40 55 20 00.

Расположение кластеров для данного примера приведено на рисунке 3.9.

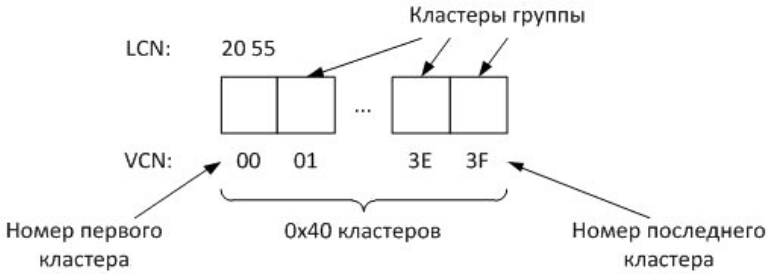


Рис. 3.9
Пример расположения файла NTFS

Здесь значение для списка групп 0x21 40 55 20 00 обозначает следующее:

- 0x21 — первый байт кодирует размер двух полей, которые за ним следуют:

– младший полубайт обозначает размер поля (в байтах), в котором хранится длина группы в кластерах; в данном случае значение «1» указывает, что на длину группы отводится один байт;

– старший полубайт обозначает размер поля (в байтах), в котором расположен логический номер первого кластера группы; в данном случае значение «2» указывает на двухбайтовое поле;

- 0x40 — длина группы. В первом байте размер поля «длина группы» определен в один байт, поэтому в качестве длины группы рассматриваем однобайтовое поле, равное 0x40 (64 кластера);
- 0x2055 — LCN номер первого кластера. В первом байте размер поля «номер первого кластера» определен в два байта, поэтому в качестве LCN-номера первого кластера рассматриваем двухбайтовое поле, которое в примере

равно 0x2055 (обратите внимание, что байты на диске записываются в обратном порядке: сначала младшие — 55, затем старшие — 20);

- 0x00 — признак окончания описания списка групп.

Указанные обозначения приведены на рисунке 3.10.

Рис. 3.10
Определение расположения файла из записи MFT

В рассмотренном примере нерезидентный атрибут содержится всего в одной группе, но в общем случае групп может быть несколько.

Альтернативные потоки NTFS

Каждый файл в NTFS представляет собой набор потоков, в которых хранятся данные. По умолчанию содержимое файла записывается в основной поток, но при необходимости к файлу можно добавлять дополнительные, альтернативные потоки данных (ADS), в которых можно хранить, например, иконки и другую информацию о файле.

Потоки данных файла описываются атрибутом $DATA, одним из свойств которого является имя потока. Основной поток является неименованным, а каждый альтернативный поток должен иметь собственное имя. Альтернативные потоки скрыты от пользователя и не отображаются большинством стандартных программ. Они могут содержать любой тип информации — текстовый, графический, видео и т. д. В альтернативный поток можно даже записать программу, что иногда используется для распространения вредоносного ПО.

Ниже приведен пример хранения в одном файле четырех потоков данных. В качестве основного потока используется строка размером 15 байт, а в трех дополнительных потоках хранятся вторая строка размером 21 байт, графический файл и программа стандартного калькулятора Windows. При этом размер файла, выводимый командой **dir** или Проводником, всегда будет равен 15 байт.

Пример. Пусть имеется файл ***primer.txt*** размером 15 байт, содержащий строку*: «Hello, students»*

1. Создаем в файле именованный поток с именем **potok1.txt**:

echo This is second stream > primer.txt:potok1.txt

Размер файла, выводимый командой **dir** или Проводником, при этом не изменится.

2. Просмотрим содержимое основного и альтернативного потоков:

type primer.txt (выводится строка «Hello, students»)

type primer.txt:potok1.txt (ошибка, команда не видит второй поток!)

more < primer.txt:potok1.txt (выводится строка «This is second stream»)

notepad primer.txt:potok1.txt (Блокнот также видит альтернативный поток)

3. Создадим в файле второй альтернативный поток с именем **potok2.jpg**, содержащий графический файл:

type foto.jpg > primer.txt:potok2.jpg

Размер файла, выводимый командой **dir** или Проводником, при этом также не изменится.

4. Извлечем графические данные из потока с помощью графического редактора:

mspaint primer.txt:potok2.jpg

5. Создадим в файле третий альтернативный поток с именем **calcul.exe**, содержащий программу Калькулятор:

type c:\windows\system32\calc.exe > primer.txt:calcul.exe

Размер файла, выводимый командой **dir** или Проводником, при этом опять не изменится.

6. Запустим калькулятор из текстового файла:

start .\primer.txt:calcul.exe

3.2.3. Инструментальные средства анализа файловой системы

Основным инструментом исследования файловой системы магнитных дисков является специальная программа — дисковый редактор. Современные дисковые редакторы обладают большим набором возможностей: просмотр и редактирование системной области диска; просмотр и редактирование директорий и файлов; восстановление удаленных директорий и файлов; доступ к любому участку диска по номеру сектора или кластера; работа с образом диска; создание загрузочных дисков и т. д.

После загрузки редактора необходимо выбрать тип диска — физический или логический. При выборе физического диска открывается таблица разделов, в которой хранится список логических дисков с указанием типа файловой системы, объема и границ каждого логического диска (рис. 3.11). Для отображения имени разделов диска можно нажать кнопку «Меню» и выбрать пункт «Показать буквы томов».

Индикаторы показывают наличие соответствующих структур:

– T — таблица разделов;

– E — элемент таблицы разделов;

– B — загрузочный сектор тома;

– C — копия загрузочного сектора;

– F — основные структуры ФС (например, начальная запись MFT для NTFS).

С помощью контекстного меню для каждого логического диска можно выполнить следующие действия: открыть, удалить или создать образ. Образ представляет собой файл, содержащий снимок диска, т. е. его точную физическую копию, которую можно использовать для восстановления диска в случае повреждения.

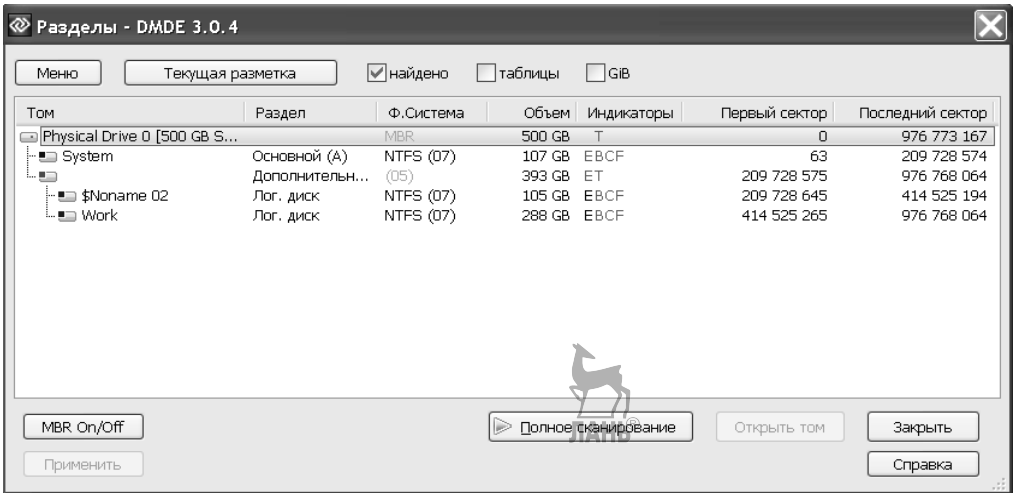


Рис. 3.11
Выбор таблицы разделов физического диска

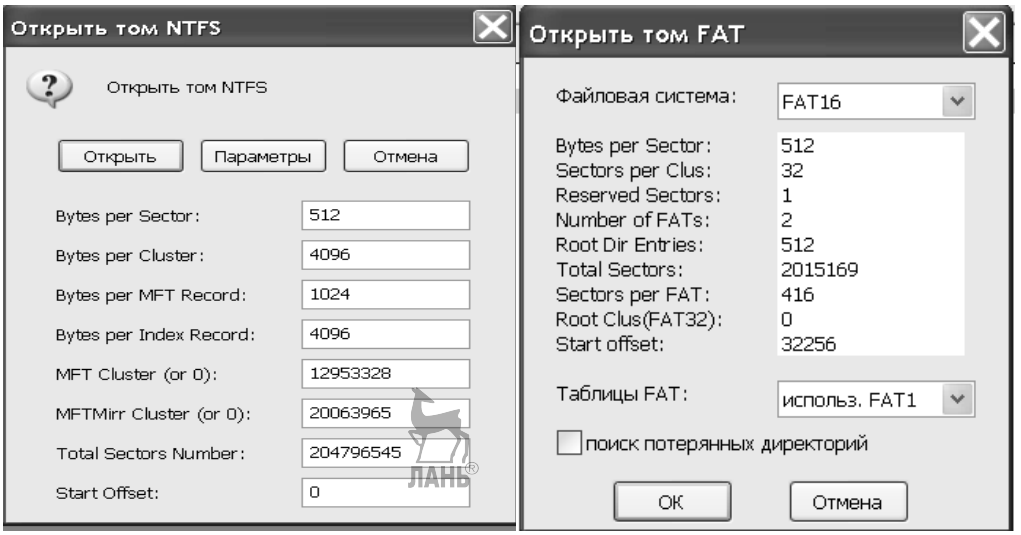


Рис. 3.12
Вывод параметров логических дисков NTFS и FAT

После открытия логического диска редактор выводит его параметры, набор которых зависит от типа установленной файловой системы: размеры сектора и кластера, число элементов корневого каталога или расположение файла MFT и т. д. (рис. 3.12).

Нажатие кнопки «Открыть» переводит редактор в режим просмотра, в котором имеются три панели (просмотр папок, просмотр файлов и панель редактора), показанные на рисунке 3.13. В панель редактора можно выводить содержимое системной области и области данных диска. Управление панелью редактора проводится через меню Редактор.

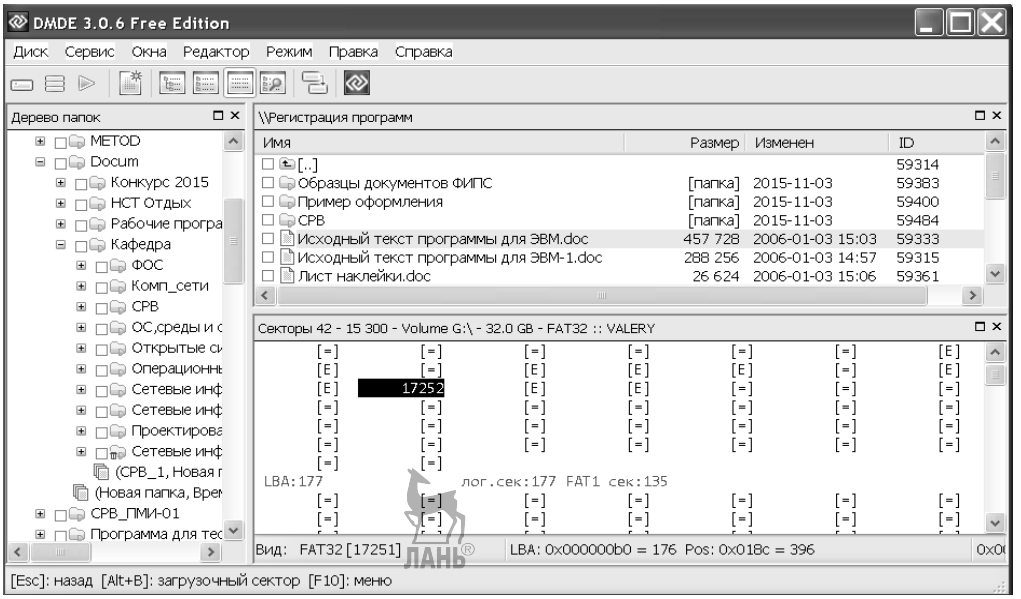


Рис. 3.13
Основное окно редактора DMDE

Работа с файловой системой FAT32

Для FAT32 в панель редактора из системной области можно выводить загрузочную запись, таблицу FAT и корневой каталог, а из области данных — каталоги и файлы. При просмотре таблицы FAT элементы, соответствующие свободным кластерам, выводятся символом «0», занятым кластерам — символом «=«, а занятым последним кластерам — символом «E». Реальные значения элементов FAT выводятся при установке курсора на элемент, при этом в строке статуса отображается название файловой системы и номер кластера, который соответствует текущему элементу FAT (например, FAT32 [17251]). По каждому каталогу и файлу выводится имя, расширение, размер, номер начального кластера, атрибуты и даты создания и изменения.

Просмотр содержимого файла, которое выводится в шестнадцатеричном и символьном виде, проводится двойным щелчком мыши по имени файла; изменение кодировки символов проводится в меню Режим/Кодировка (рис. 3.14). В этом режиме можно также посмотреть цепочку кластеров, выделенных данному файлу (меню Редактор/Карта кластеров).

Элементы каталога, имена которых начинаются с символа «x», соответствуют удаленным файлам. Если в поле имени стоят цифры или символы «e0», то этот элемент предназначен для хранения длинного имени файла (рис. 3.15).

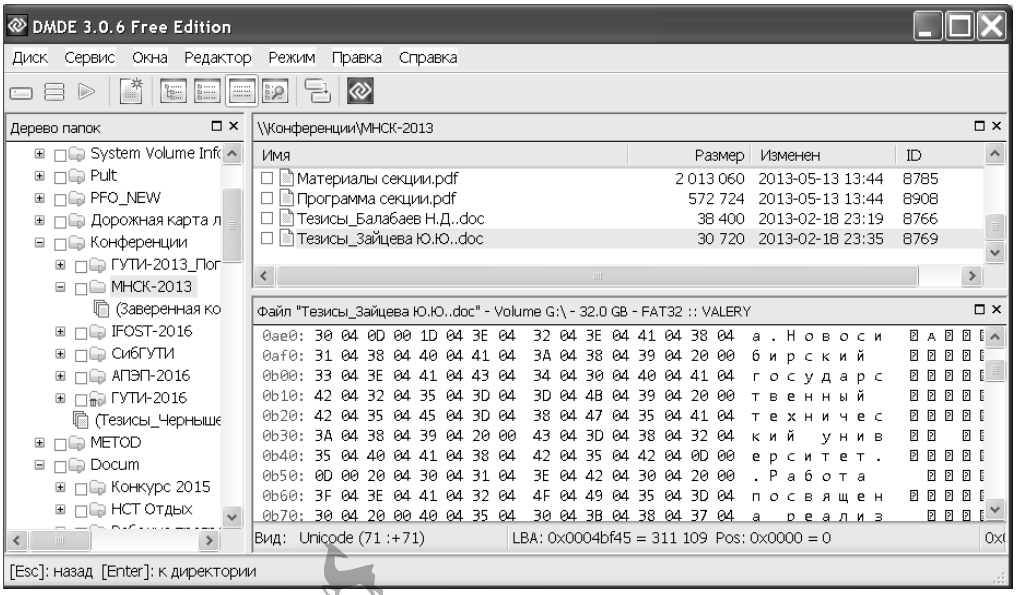


Рис. 3.14
Просмотр содержимого файла

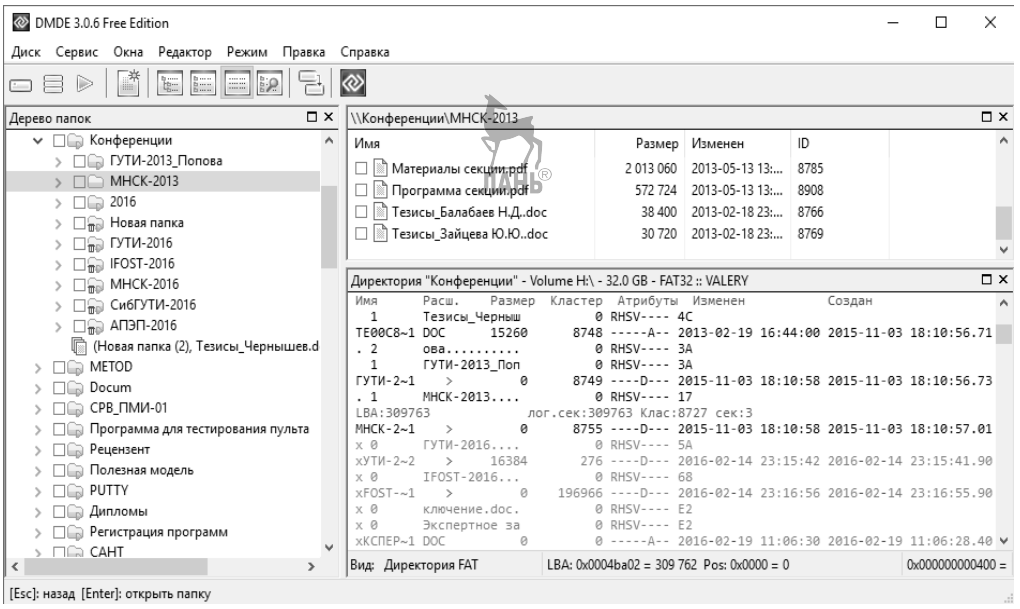


Рис. 3.15
Просмотр содержимого каталога

Работа с файловой системой NTFS

После выбора логического диска в окне редактора будет выведено содержимое файла $MFT (рис. 3.16).

Первая запись описывает сам файл $MFT, а вторая — копию его первых четырех записей ($MFTMirr). Для каждой записи выводится ее адрес на диске (номера кластера и сектора), граничные метки, внутренний номер (индекс) и набор атрибутов. Минимальный набор включает атрибуты $STANDART_INFORMATION, $FILE_NAME и $DATA. Для просмотра содержимого каждого атрибута необходимо в его строке сделать щелчок мыши на символе ‘+’.

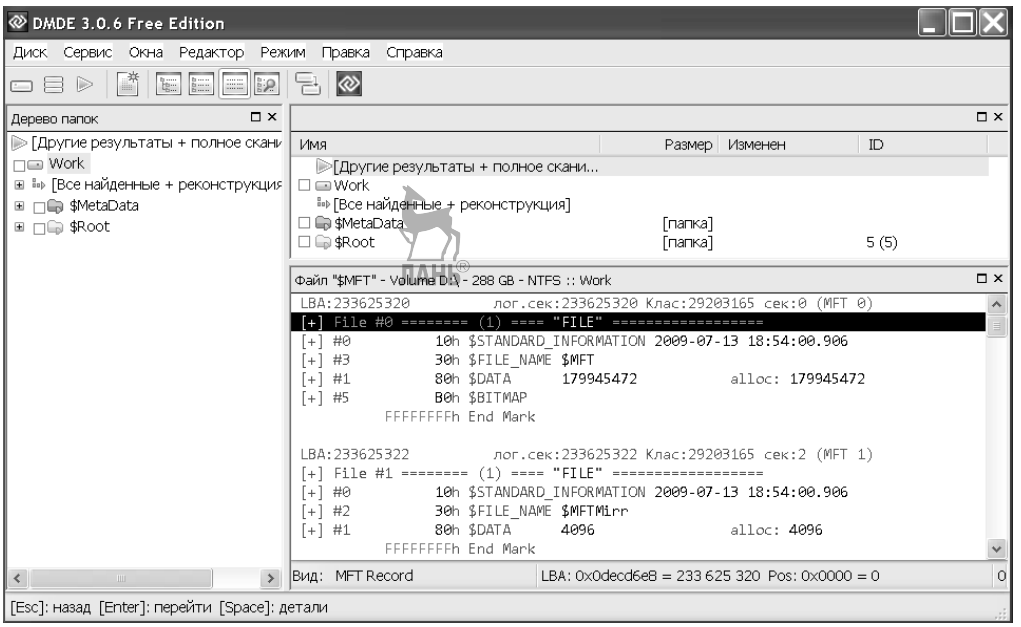


Рис. 3.16
Просмотр содержимого файла $MFT

На рисунке 3.17 показано содержимое атрибута $DATA, указывающего на расположение данных одного из файлов.

Анализ рисунка позволяет сделать следующие выводы:

– индекс файла в MFT — 172808;

– данные файла находятся на диске, так как атрибут является нерезидентным;

– данные занимают 8 кластеров (start vcn=0, end vcn=7) или 32 768 байта, файл не фрагментирован;

– номер начального кластера файла — 18986720;

– длина атрибута — 72 байта.

На рисунке 3.18 показан этот файл в режиме просмотра данных.

Более полную информацию по файловым системам Windows можно найти в [2, 15, 16].

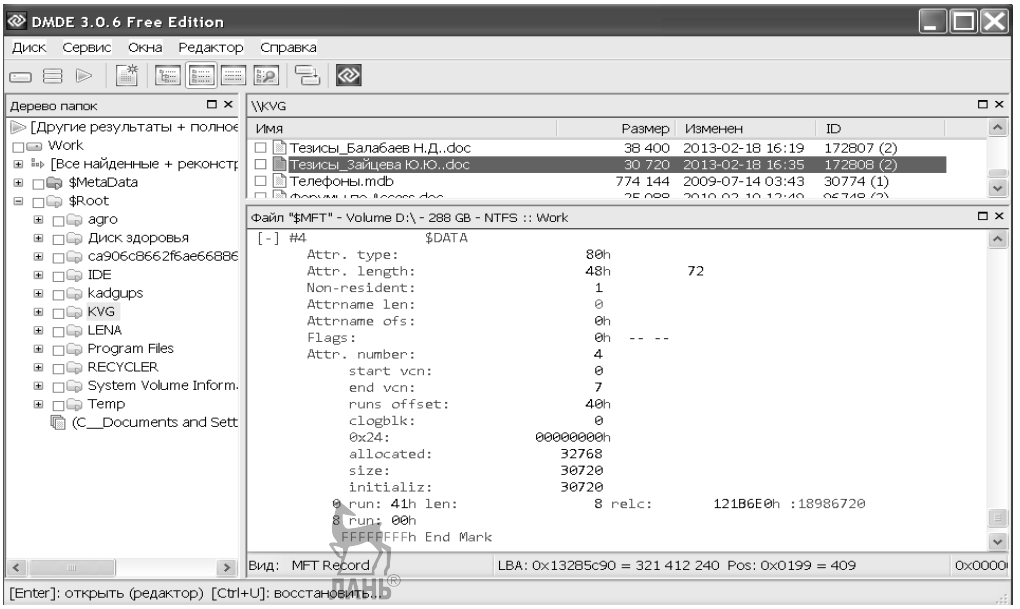


Рис. 3.17
Просмотр содержимого атрибута $DATA

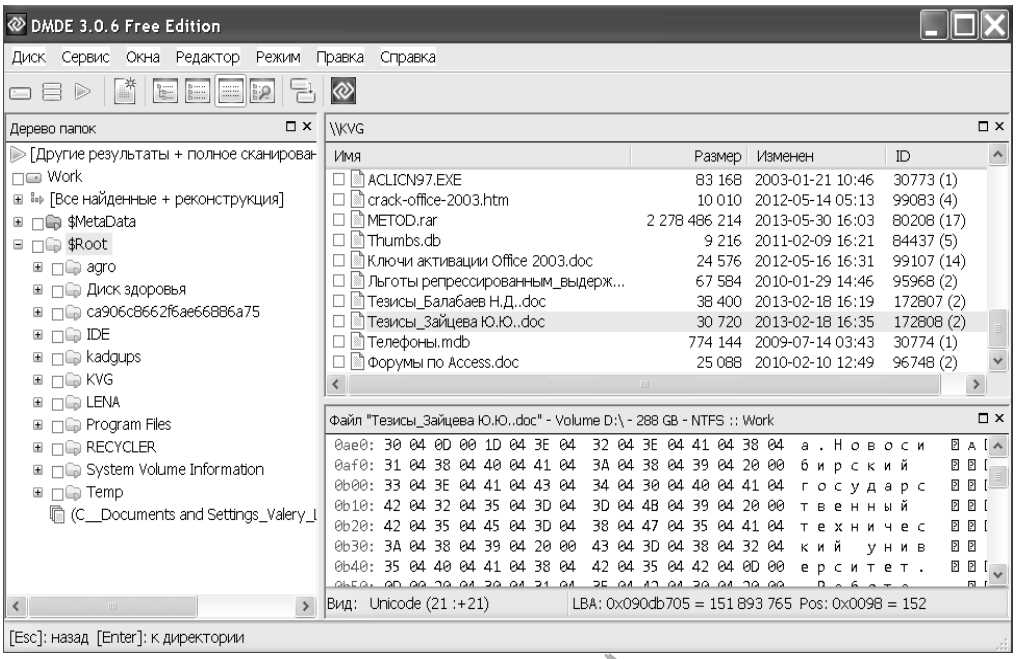


Рис. 3.18
Просмотр содержимого файла

3.3. Файловые системы операционной системы Linux

3.3.1. Общие сведения

Основные особенности файловых систем Linux были рассмотрены в разделе 1. Они делятся на два типа — локальные и распределенные (сетевые). Локальные файловые системы могут располагаться во внешней памяти (ext2, ext3, ext4 и др.) или в оперативной памяти (псевдофайловые системы). К последней группе относятся:

- /proc — используется в качестве интерфейса к структурам данных процессов; большинство расположенных в ней файлов доступны только для чтения, но в некоторые файлы можно записывать данные, что позволяет изменить переменные ядра;
- /tmpfs — используется для хранения временных файлов, которые формируются в оперативной памяти, а затем удаляются; поддерживает работу с виртуальной памятью;
- /devfs — предназначена для управления устройствами;
- /sysfs — используется для получения информации обо всех устройствах и драйверах.

Распределенные файловые системы предназначены для объединения на логическом уровне файловых систем отдельных компьютеров в единое целое. В Linux такой системой является nfs (Network File System). Ниже будут рассмотрены только локальные системы, размещенные во внешней памяти.

3.3.2. Файловая система s5

Одной из первых файловых систем, используемых в семействе ОС UNIX, была файловая система SYSTEM V (s5). Система управляет всеми блоками раздела, разделяя его на две области — область метаданных и область данных. В области метаданных расположены три объекта — загрузочный блок, суперблок и индексные дескрипторы (рис. 3.19).

Рис. 3.19
Логическая модель раздела s5

Загрузочный блок используется для загрузки ОС и занимает первый блок файловой системы в каждом разделе. Однако активным является только один загрузочный блок, находящийся в корневой файловой системе.

Суперблок располагается непосредственно за загрузочным блоком и содержит самую общую информацию о файловой системе (общий размер, размер области индексных дескрипторов, их число, список свободных блоков и индексных дескрипторов и т. д.). При монтировании файловой системы в оперативной памяти создается копия ее суперблока. Все последующие операции по созданию и удалению файлов влекут изменения копии суперблока в оператив-

ной памяти, эта копия периодически записывается на магнитный диск. Часто причиной повреждения файловой системы является отключение электропитания или зависание ОС в тот момент, когда система производит копирование суперблока из оперативной памяти на магнитный диск.

Файловая система s5 использует три главные структуры данных: каталоги, индексные дескрипторы (*inode, i-узлы*) и блоки данных (*data blocks*). Доступ к данным файлов проводится не напрямую через каталог, как в ОС Windows, а через промежуточную структуру, называемую дескриптором файла (рис. 3.20). Каталоги содержат записи с именами файлов и соответствующими им номерами дескрипторов.

Область индексных дескрипторов содержит дескрипторы всех файлов раздела. С каждым файлом связан один i-узел, но одному i-узлу могут соответствовать несколько файлов (см. рис. 3.20). В i-узле размером 64 байта хранится вся информация о файле, кроме его имени. Область дескрипторов имеет фиксированный формат и располагается за суперблоком. Общее число дескрипторов и, следовательно, максимальное число файлов раздела задаются в момент создания файловой системы.

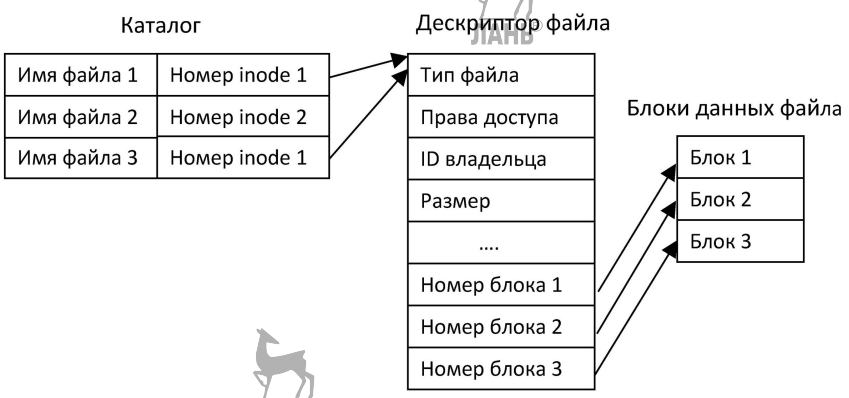


Рис. 3.20
Доступ к данным файла s5

Дескрипторы нумеруются натуральными числами. Первый дескриптор используется ОС для описания специального файла, содержащего информацию о сбойных блоках раздела, которые рассматриваются ОС как принадлежащие специальному файлу и поэтому считаются занятыми. Второй дескриптор содержит описание корневого каталога файловой системы.

В области данных расположены регулярные файлы, ссылки и каталоги (в том числе корневой каталог). Специальные файлы представлены в файловой системе только записями в соответствующих каталогах и индексными дескрипторами специального формата, т. е. места в области данных не занимают.

3.3.3. Файловая система ext2

Для повышения быстродействия файловых операций было предложено проводить дробление файловых систем на относительно независимые части, получившие название групп цилиндров (файловая система ufs в UNIX BSD)

или групп блоков (файловая система ext2 в Linux). В результате этого появилась возможность размещения логически связанной информации в физически смежных областях диска, что минимизировало перемещение дисковых головок, увеличивая скорость доступа к данным.

Расчленение файловых систем также способствовало росту их надежности, так как давало возможность в разных ее частях дублировать критически важную информацию — суперблок, утрата которого (например, из-за физического повреждения дисковой поверхности) ранее влекла за собой невозможность доступа к данным.

Основной файловой системой ОС Linux на начальном этапе была система ext2, структура которой показана на рисунке 3.21. Она позволяла хранить длинные имена файлов (до 255 символов) и обеспечить высокую производительность. В первом блоке файловой системы ext2 располагается загрузчик, а все остальное пространство делится на блоки равного размера (обычно 1 Кбайт). Блоки объединяются в группы, каждая из которых имеет суперблок, описатель группы, два битовых массива (для блоков и для i-узлов) и блоки для хранения данных.

Суперблок хранит информацию о количестве i-узлов и блоков данных в группе, о размере группы и т. д. Описатель группы содержит информацию о расположении битовых массивов, количестве свободных блоков и i-узлов в группе, а также о количестве каталогов в группе. Битовые массивы предназначены для учета свободных блоков и i-узлов, для хранения каждого массива выделяется один блок. При размере блока 1 Кбайт размер группы равен 8192 блока и количество i-узлов равно 8192.

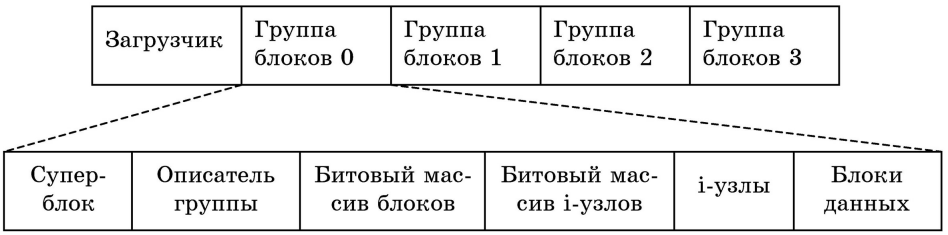


Рис. 3.21
Логическая модель раздела ext2

Далее находятся i-узлы, структура которых показана на рисунке 3.22. Размер каждого узла составляет 128 байт.

Режим	Счетчик связей	UID	GID	Размер	Метки времени	Адреса блоков	Однократный косвенный блок	Двукратный косвенный блок	Трехкратный косвенный блок

Рис. 3.22
Структура дескриптора файла ext2

Здесь обозначено:
– режим — тип файла, биты защиты;
– счетчик связей — число записей каталогов, указывающих на этот i-узел;
– UID — идентификатор владельца файла;

– GID — идентификатор группы владельца;

– размер — размер файла в байтах;

– метки времени — время последнего доступа и последнего изменения файла, а также время последнего изменения i-узла;

– адреса блоков — адреса первых 12 блоков файла, размер адреса — 4 байта;

– однократный косвенный блок — адрес одинарного косвенного блока, который содержит адреса 256 дополнительных блоков файла;

– двукратный косвенный блок — содержит адреса 256 одинарных косвенных блоков, каждый из которых содержит адреса 256 блоков данных;

– трехкратный косвенный блок — содержит адреса 256 двукратных косвенных блоков.

Часть информации из i-узла можно посмотреть командой **stat *имя_файла***.

Достоинством файловой системы ext2 является ее быстродействие, а недостатком — отсутствие средств журналирования, что ухудшает надежность хранения данных. Этот недостаток исправлен в современных файловых системах ext3 и ext4. Кроме того, в ext4 адресация блоков увеличена до 6 байт, а размер i-узлов — до 256 байт, что позволило увеличить максимальный размер одного файла до 16 Тбайт.

Файловая система ext2 до сих пор активно используется на твердотельных накопителях, например на SSD-дисках, где ячейки памяти необратимо изнашиваются после определенного количества операций записи. Отсутствие журналирования дает возможность значительно уменьшить количество перезаписей одного и того же сектора диска и продлить срок службы устройства.

3.3.4. Команды анализа файловой системы

Команда **df**

Назначение: показывает список файловых систем на смонтированных устройствах, их размер, занятое и свободное пространство, а также точки монтирования.

Формат: df [-опции][имя_файла]

Примеры:

а) df -T — вывод списка файловых систем с указанием их типа;

б) df ~ — вывод информации по файловой системе, в которой находится домашний каталог.

Комментарий: для того чтобы вместо информации об использовании блоков выводить информацию об использовании индексных дескрипторов, надо применить опцию –i.

Команда **du**

Назначение: показывает использование дисковой памяти файлами.

Формат: du [-опции] [имя_файла]

Примеры:

а) du — вывод размера всех подкаталогов текущего каталога;

б) du -ah — вывод размера (в байтах) всех файлов и подкаталогов текущего каталога

Комментарий: размер файлов по умолчанию выводится в блоках, для вывода в байтах надо применить опцию -h.

Команда **lsblk**

Назначение: показывает список блочных устройств.

Формат: lsblk [-опции] [имя_устройства]

Примеры:

а) lsblk — вывод списка активных устройств;

б) lsblk -f — вывод списка активных устройств с указанием типа файловой системы;

в) lsblk -a — вывод списка всех устройств;

г) lsblk -p — вывод списка активных устройств с указанием путей доступа;

д) lsblk -o KNAME,FSTYPE,SIZE,MODEL,MOUNTPOINT.

Комментарий: опция –o дает возможность самостоятельного формирования списка вывода, например при задании параметров KNAME, FSTYPE, SIZE, MODEL, MOUNTPOINT будут выводиться имя устройства, тип его файловой системы, размер, модель и точка монтирования.

Команда **cat /proc/partitions**

Назначение: вывод файла, который содержит общую информацию о разделах файловой системы.

Формат: cat /proc/partitions

Комментарий: по каждому разделу выводится имя, размер и номер устройства, на котором расположен раздел.

Команда **cat /etc/fstab**

Назначение: вывод файла, который содержит общую информацию об используемых файловых системах.

Формат: cat /etc/fstab

Комментарий: выводится список смонтированных файловых систем.

Команда **ls /dev/mapper**

Назначение: отображение каталога, в котором содержатся файлы блочных логических томов LVM.

Формат: ls /dev/mapper

Комментарий: логические тома используются при установке в Linux системы LVM, предназначенной для создания абстрактной логической внешней памяти вместо физического управления дисками.

3.4. Практическое задание

Для выполнения задания рекомендуется использовать свободно распространяемый дисковый редактор DMDE (http://dmde.ru), который можно скачать с сайта разработчика. Существуют версии редактора для Windows и для Linux. Задание требует прямого обращения к дискам, что несет потенциальную опасность для вычислительной системы, поэтому желательно работать **только в режиме чтения**.

Если вы выполняете работу на домашнем компьютере, то можно исследовать реальные диски или флеш-накопители. В компьютерном классе в целях безопасности администратор системы может отключить возможность работы с дисками напрямую, в этом случае задание можно выполнять с заранее подготовленным образом диска или использовать виртуальную машину. Удаленное подключение к виртуальной машине из среды Windows проводится следующим образом:

mstsc -vимя_машины

3.4.1. Работа с файловыми системами Windows

1. Откройте дисковый редактор DMDE и определите параметры жесткого диска: общий объем, число и типы разделов, тип установленной файловой системы. Для FAT-раздела определите размеры сектора и кластера; число секторов, выделенных для таблицы FAT, и размер корневого каталога. Для NTFS-раздела определите размеры сектора и кластера, размер файла $MFT и его адрес, размеры записи MFT и индексной записи.

2. Откройте логический диск с файловой системой FAT32 и выполните следующие действия.

2.1. В личном каталоге создайте три файла, в которые запишите один русскоязычный текстовый документ размером 40–60 Кбайт. Файлы должны иметь форматы *.txt*, *.doc* и *.docx*, имена файлов должны содержать не менее 15 символов, например З*адание по файловой системе.* Определите размер каждого файла, поясните отличия.

2.2. Для файла З*адание по файловой системе.txt* выполните следующие действия:

– определите число элементов каталога, выделенных для хранения информации по файлу;

– занесите в таблицу 3.4 содержимое элемента, предназначенного для хранения короткого имени:

Таблица 3.4

Содержимое элемента каталога для короткого имени

Наименование поля	Значение поля
имя файла	
расширение имени	
атрибуты	
время создания	
дата создания	
номер начального кластера	
размер файла	

– просмотрите содержимое и коды первых 16 байт файла, занесите их в отчет;

– определите используемую кодировку символов путем сравнения с кодировочными таблицами DMDE;

– определите список кластеров этого файла, результаты занесите в таблицу 3.5;

Таблица 3.5

Список кластеров, занимаемых файлом

Логический номер кластера в файле	1	2	3	…	n
Номер кластера на диске					
Значение элемента FAT					

2.3. Удалите файл **Задание по файловой системе.txt** из личного каталога, проведите анализ изменений в FAT и в каталоге, посмотрите содержимое начального кластера удаленного файла.

2.4. Восстановите удаленный файл **Задание по файловой системе.txt**.

2.5. Определите используемую кодировку символов для файлов **Задание по файловой системе.doc** и **Задание по файловой системе.docx** с учетом того, что редактор Word сохраняет содержимое документа, начиная с пятого сектора начального кластера файла.

3. Откройте логический диск с файловой системой NTFS и выполните следующие действия.

3.1. В личном каталоге создайте три файла в соответствии с требованиями п. 2.1.

3.2. Определите характеристики файла $MFT (начальный адрес, число записей, размер в байтах и кластерах).

3.3. Определите число записей в файле $MFTmirr.

3.4. Проведите анализ записи MFT, соответствующей файлу **Задание по файловой системе.txt**, и занесите в отчет описания всех атрибутов, включая расположение файла на диске.

3.5. Удалите файл **Задание по файловой системе.txt**, проведите анализ изменений в MFT и в области данных.

3.6. Восстановите удаленный файл.

3.7. С помощью программы Блокнот создайте текстовый файл **primer.txt**, записав в него фразу «Very good weather today!». Проведите анализ соответствующей записи MFT, определите размер файла и его адрес на диске.

3.8. Запишите в файл **primer.txt** второй поток данных, используя для этого, например, любой текстовый файл размером не менее 50 Кбайт. Проведите анализ соответствующей записи MFT и определите расположение данных этого потока на диске. Определите размер файла, сравните с предыдущим пунктом.

3.9. Запишите в файл **primer.txt** третий поток данных, используя для этого любой графический файл (например, фотографию). Проведите анализ соответствующей записи MFT и определите расположение данных этого потока на диске. Определите размер файла, сравните с предыдущим пунктом.

3.4.2. Работа с файловыми системами Linux

1. Посмотрите с помощью команды **stat** информацию из индексного дескриптора файла **~/.bash_history**.

2. С помощью команд **lsblk** и **df** определите основные характеристики разделов внешней памяти сервера (имя и номер устройства, имя и тип раздела, размер, тип файловой системы, коэффициент использования памяти). Результаты представьте в виде таблицы 3.6:

Таблица 3.6

Список разделов внешней памяти Linux

№ п/п	Имя устройства	Имя раздела	Тип раздела	Размер раздела (Гб)	Тип ФС	Номер устройства	Коэф-т использования
1	centos-tmp	dm-2	lvm	9,8	ext4	253:2	2%
2							

3. С помощью команд **df** и **du** определите типы файловых систем, а также в каком из имеющихся разделов расположен ваш домашний каталог и размер домашнего каталога. Поясните назначение каждой из файловых систем.

4. Посмотрите содержимое файлов **/proc/partitions** и **/etc/fstab**, сопоставьте их с результатами, полученными в п. 2 и 3.

Контрольные вопросы

1. Физическая и логическая модели диска.
2. Способы разбивки дорожек на сектора.
3. Разбиение диска на разделы, именование разделов в Linux и Windows.
4. Каким образом поддерживается древовидная многоуровневая система каталогов?
5. Какие действия выполняются файловой системой при удалении файла в файловых системах FAT и NTFS?
6. Механизмы выделения дисковой памяти в FAT и NTFS при записи нового файла на диск.
7. Чем определяется число элементов каталога, выделяемых для хранения метаданных файла в файловой системе FAT?
8. Почему в файловой системе FAT не рекомендуется хранить файлы с длинными именами в корневом каталоге?
9. Основные различия файловых систем FAT и NTFS.
10. Почему в NTFS увеличивается скорость доступа к файлам по сравнению с FAT?
11. Структура файла MFT.
12. Структура файловой записи MFT.
13. Общая структура файловой системы ext2.
14. Опишите действия файловых систем FAT, NTFS и ext2 при обращении к некоторому файлу с запросом на чтение.
15. Принцип работы LVM.

4. ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА РАЗРАБОТКИ ПРОГРАММ

Прочитав эту главу, вы узнаете:

– этапы прохождения программы в среде ОС;

– структуру объектного модуля;

– способы управления компилятором, компоновщиком и отладчиком в командном режиме;

– особенности прохождения программ в платформе .NET;

– определение следующих терминов: компилятор, интерпретатор, препроцессор, компоновщик, отладчик, карта памяти.

4.1. Прохождение программ в среде операционной системы

Для выполнения любой программы необходимо выполнить несколько обязательных требований:

– программа должна быть записана в виде набора машинных двоичных команд;

– программа должна быть размещена в любом свободном участке оперативной памяти компьютера;

– программе необходимо выделить устройства ввода-вывода данных;

– процессору необходимо сообщить адрес первой команды программы.

Первое требование (преобразование программы, написанной на языке высокого уровня, в набор машинных команд) реализуется средой разработки, остальные требования — операционной системой. На рисунке 4.1 приведена общая схема прохождения программы через среду разработки и операционную систему, где ЦП — центральный процессор, ОЗУ — оперативная память.

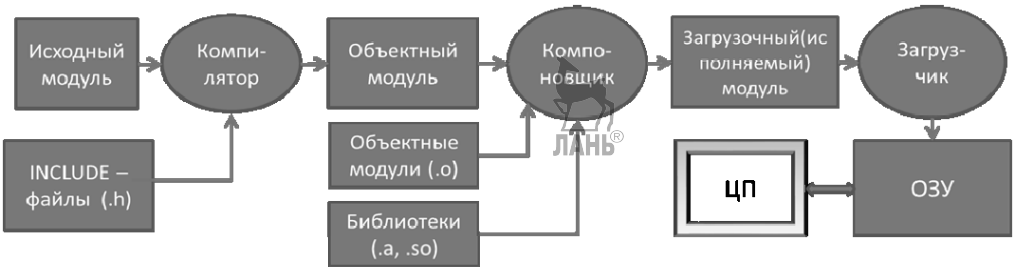


Рис. 4.1
Схема прохождения программы пользователя

Компилятор преобразует исходную программу (исходный модуль) в набор машинных двоичных команд (объектный модуль) с относительной адресацией. Для каждой переменной, используемой в программе, также выделяется участок памяти, имеющий относительной адресацию. Относительная адресация используется из-за того, что на этапе компиляции неизвестен физический адрес

участка оперативной памяти, который будет выделен для загрузки программы в ОЗУ.

При необходимости компилятор может подключить к исходной программе дополнительные модули, написанные на языке высокого уровня (файлы типа .h).

Компоновщик подключает к объектному модулю, сформированному компилятором, другие объектные модули, требуемые для его выполнения, в результате чего получается загрузочный (исполняемый) модуль, также имеющий относительную адресацию. Дополнительные объектные модули могут находиться как в виде последовательных файлов с расширением .o, так и в библиотечных файлах, имеющих расширение .a или .so.

Загрузчик получает от операционной системы физический адрес выделенного для загрузки программы участка и преобразует относительную адресацию исполняемого модуля в абсолютную (физическую) с учетом используемого алгоритма распределения памяти, загружает этот модуль в выделенные участки памяти и передает управление центральному процессору на первую команду программы.

Отметим, что выше была описана схема прохождения, основанная на том, что исходная программа полностью переводится в набор машинных команд, который сохраняется в виде загрузочного модуля, запускаемого на выполнение. Существует альтернативный способ, когда каждая строка исходной программы отдельно переводится в набор машинных команд, который сразу выполняется. Для реализации этого способа вместо компилятора используется специальная программа — интерпретатор. Примерами интерпретирующих языков являются HTML, Python, Basic.

Достоинствами интерпретирующих систем программирования являются меньшие затраты памяти из-за отсутствия необходимости хранения всей программы во время обработки, а также удобство отладки, так как при обнаружении ошибки в любой строке исходной программы ее интерпретация сразу прекращается и указывается место ошибки. Основными недостатками интерпретаторов являются:

– снижение быстродействия программ из-за того, что даже после устранения всех ошибок при запуске программы все равно проводится построчный перевод в машинные команды с их последующим выполнением;

– невозможность получения отделяемого продукта, так как интерпретируемая программа не может исполняться без интерпретатора.

Для увеличения производительности некоторые среды разработки (например, Java, PHP) используют комбинированную схему обработки программ, когда вся исходная программа переводится компилятором в некоторый промежуточный код, который далее исполняется в режиме интерпретации. Такая схема называется интерпретацией компилирующего типа.

Далее будет рассматриваться классическая схема обработки программ с использованием компилятора на примере ОС Linux, которая имеет полный набор инструментов для разработки приложений, поддерживающий все основные этапы разработки:

– создание исходного кода (текста) программы;
– сохранение различных вариантов исходного текста;
– компиляция исходного кода;
– компоновка программы (сборка);
– отладка программы;
– сохранение всех изменений, выполняемых при тестировании и отладке.

В примерах будут использованы компилятор **gcc,** компоновщик **ld**, отладчик **gdb** и система контроля версий **cvs**.

Обратите внимание: *в Linux имена всех файлов, команд и ключей являются регистрозависимыми.*

4.2. Компиляция исходного модуля

Компиляция — процесс перевода программы с алгоритмического языка на язык машинных команд, при этом исходный модуль программы преобразуется в объектный модуль. В общем случае обработка исходной программы компилятором выполняется в несколько этапов:

– препроцессорная обработка, результатом которой является включение в исходную программу содержимого всех заголовочных файлов, указанных в директивах **#include**. В заголовочных файлах обычно находятся объявления функций, используемых в исходной программе, но определенных во внешних файлах с исходным кодом, имеющих расширение .h (например, stdio.h). Результатом препроцессорной обработки является объединенный исходный код программы.

– лексический анализ (выделение ключевых слов и конструкций языка программирования);

– синтаксический анализ (выявление ошибок в синтаксисе операторов);

– генерация команд Ассемблера (необязательный этап);

– генерация машинного кода.

В ходе выполнения анализа исходной программы компилятор может выводить два типа диагностических сообщений — предупреждение (warning) и ошибка (error). *Предупреждение* выводится в случае обнаружения ситуации, которая не влияет на корректность работы программы (например, в программе описана переменная, но она нигде не используется). *Ошибка* является следствием нарушения формальной грамматики языка программирования и делает невозможным формирование объектного модуля.

Ошибки могут быть лексическими и синтаксическими. Лексические ошибки связаны с распознаванием лексем языка (например, *wile* вместо *while*, *dbl* вместо *double* и т. д.), синтаксические — вызваны нарушением синтаксиса конструкций языка (пропущена точка с запятой, неравное число открывающихся и закрывающихся скобок или кавычек, использование неописанных переменных и т. д.). В сообщениях компилятор указывает место обнаружения ошибки в исходном коде.

Часто возникает ситуация, когда число сообщений об ошибках превышает число реальных ошибок в программе. Это объясняется тем, что компилятор

просматривает всю программу целиком и реальная ошибка в одном операторе может приводить к неправильным синтаксическим конструкциям в нескольких последующих операторах программы, что приводит к генерации «наведенных» сообщений об ошибках. Некоторые компиляторы дают возможность пользователю ограничить число таких сообщений, после чего процесс компиляции прекращается.

Результатом работы компилятора является объектный модуль (рис. 4.2), состоящий из машинного кода исходной программы, словаря внешних символов ESD (External Symbol Dictionary) и словаря перемещений RLD (Relocation Dictionary).

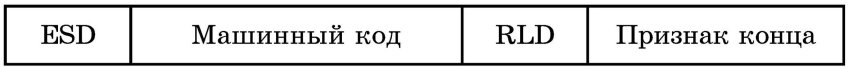


Рис. 4.2
Структура объектного модуля

Словарь ESD используется для организации связи с другими объектными модулями и между функциями внутри объектного модуля. Внешние символы — это имена модулей и вызываемых функций; каждый символ хранится в отдельной строке словаря, куда во время компоновки программы заносятся адреса соответствующих функций. *Словарь RLD* содержит информацию обо всех переменных и других адресных константах, используемых в объектном модуле (имена и адреса). Словари используются компоновщиком на этапе формирования исполняемого модуля.

Все компиляторы, работающие в одной ОС, создают объектные модули с одинаковой структурой для того, чтобы можно было объединять в одной программе модули, написанные на разных языках программирования. В объектном модуле используется *относительная* адресация всех команд и переменных, началом отсчета (нулевым адресом) является адрес первой команды.

Компилятор **gcc** способен по расширению имени файлов определять их типы и действие, которое необходимо с ними выполнить (табл. 4.1). Он имеет множество различных опций, используемых для управления процессом компиляции, некоторые из них приведены в таблице 4.2.

Таблица 4.1

Зависимость действий gcc от расширения имени файла

Имя файла	Содержимое и действие
file.c	исходный код на C, который нуждается в препроцессорной обработке
file.cpp	исходный код на C++, который нуждается в препроцессорной обработке
file.i	исходный код на C, который не нуждается в препроцессорной обработке
file.ii	исходный код на C++, который не нуждается в препроцессорной обработке
file.h	заголовочный файл C
file.S	ассемблерный код, который нуждается в препроцессорной обработке
file.s	ассемблерный код
file.o	объектный код

Таблица 4.2

Некоторые опции gcc

Ключ	Действие
-g	включить в исполняемый файл дополнительную отладочную информацию, используемую отладчиком gdb
-c	отключить автоматический вызов компоновщика, при этом на выходе компилятора формируется объектный файл для каждого исходного входного файла
-S	отключить ассемблирование, на выходе формируется файл с ассемблерным кодом для каждого не ассемблерного входного файла
-E	остановиться после этапа препроцессорной обработки, на выходе формируется исходный файл с подключенными заголовочными файлами
-o file	поместить результат в файл с указанным именем (по умолчанию результат записывается в файл a.out)
-n file	указать имя каталога для записи объектных модулей
-I*каталог*	добавить указанный каталог в список каталогов для поиска заголовочных файлов
-L*каталог*	добавить указанный каталог в список каталогов для поиска библиотек

После успешного формирования объектного модуля компилятор **gcc** без дополнительных команд со стороны пользователя вызывает компоновщик, который формирует исполняемый (загрузочный) модуль. Если при вызове компилятора была указана опция -c, то компоновщик не вызывается.

Для упрощения процесса отладки программы компилятор может включить в состав объектного модуля исходный текст программы и связать каждую строку исходного текста с набором соответствующих ей машинных команд. Это позволяет отладчику проводить пошаговое выполнение программы. Отладочная информация значительно увеличивает размер исполняемого файла, поэтому после отладки рекомендуется ее удалить с помощью программы **strip**.

Примеры:

а) gcc -o abcd -g abcd.c — компиляция программы **abcd.c** и формирование исполняемого модуля **abcd** с включением в него отладочной информации;

б) gcc -c abcd.c — компиляция программы **abcd.c** и формирование объектного модуля **abcd.o**;

в) gcc -E abcd.c — препроцессорная обработка программы **abcd.c** и формирование результата на стандартный вывод;

г) strip a.out — полностью очистить файл **a.out** от отладочной информации.

4.3. Отладка и тестирование

Отладка — этап разработки, связанный с обнаружением, локализацией и устранением ошибок, целью отладки является обеспечение полного выполнения всех требований, предъявляемых к программе. Тестирование — это процесс исследования и испытания программы, имеющий две цели: продемонстрировать разработчикам и заказчикам, что программа соответствует требованиям,

и выявить ситуации, в которых поведение программы является неправильным, нежелательным или не соответствующим требованиям.

Ошибки, возникающие при разработке программ, могут быть лексическими, синтаксическими, сборочными и семантическими. Лексические и синтаксические ошибки связаны с нарушением формальной грамматики языка программирования и обнаруживаются компилятором. Ошибки сборки могут возникать на этапе компоновки из-за отсутствия необходимых файлов или библиотек, они обнаруживаются компоновщиком. При наличии этих типов ошибок формирование исполняемого модуля невозможно.

Семантические ошибки не обнаруживаются компилятором и компоновщиком, но приводят к неправильной работе программы, когда результат ее работы не соответствует требованиям. Обнаружить такие ошибки можно только с помощью специальных программ — отладчиков. Для использования отладчика в программу необходимо включить отладочную информацию на этапе компиляции, используя ключ -g (см. пример из раздела 4.2). Отладчик имеет следующие возможности:

– пошаговое выполнение программы с заходом в вызываемые подпрограммы или с их обходом;

– расстановка точек остановки выполнения;

– вывод на каждом шаге выполнения значений переменных и выражений;

– вывод списка имен функций от главной функции main() к текущей точке остановки.

Запуск отладчика **gdb** проводится командой **gdb имя_программы**, при этом на экран будет выведена информация об отладчике и появится приглашение для ввода команд отладчика, основными из которых являются:

backtrace — выводит весь путь к текущей точке останова, то есть названия всех функций, начиная с main();

break *параметр* — устанавливает точку останова, параметром может быть номер строки или название функции;

continue — продолжает выполнение программы от текущей точки до конца;

display — добавляет выражение в список выражений, значения которых отображаются каждый раз при остановке программы;

finish — выполняет программу до выхода из текущей функции; отображает возвращаемое значение, если таковое имеется;

help *команда* **—** информация о команде или общая информация об использовании отладчика gdb;

list — пролистывает 10 строк вниз, начиная с текущей;

next — пошаговое выполнение программы, но, в отличие от команды **step**, не выполняет пошагово вызываемые функции;

print *выражение* — выводит значение указанного выражения;

run — запускает программу на выполнение;

step — пошаговое выполнение программы;

quit — выход из отладчика.

4.4. Система управления версиями

При коллективной разработке сложных программ желательно отслеживать все изменения исходного кода — кто вносил изменения, когда и какова причина изменений. Система управления версиями позволяет:

– хранить несколько версий каждого программного модуля;

– сопровождать каждую версию комментариями;

– возвращаться при необходимости к предыдущим версиям;

– отслеживать и запрещать одновременное изменение одного участка кода несколькими разработчиками.

В настоящее время часто используемыми системами управления версиями являются CVS, Mercurial, Subversion, Git. В данном разделе будем рассматривать систему CVS (Concurrent Versions System), которая является одной из первых программ этого типа и до сих пор входит в дистрибутивы различных ОС семейства UNIX. Она хранит историю изменений набора текстовых файлов (например, исходного кода программ) и облегчает совместную работу группы программистов над одним проектом. Все файлы располагаются в специальном хранилище — репозитории, который обычно размещается на отдельном выделенном сервере, но может находиться и на локальном компьютере пользователя, так как учет версий может быть удобен и в случае однопользовательской работы. CVS реализована на основе архитектуры клиент-сервер, причем серверная часть управляет репозиторием, а клиентские части устанавливаются на локальные компьютеры пользователей. При выполнении локальных проектов возможна работа сервера и клиента на одном компьютере.

Объектом управления CVS является модуль, под которым понимается отдельный блок информации, который целиком может быть запрошен пользователем. Обычно для каждого проекта или группы проектов заводится свой CVS-модуль, который хранится в одном или нескольких каталогах репозитория.

В репозитории хранится только последняя версия модуля и история всех внесенных в него изменений, начиная с начальной версии. Поэтому в случае необходимости всегда есть возможность вернуться к любой предыдущей версии проекта. Регистрация изменений от лица конкретного пользователя хранится с точностью до строки. Имена версий файлов хранятся в виде **имя_файла номер_версии**, например **myfile.cpp 1.1**, **myfile.cpp 1.2** и т. д.

Для работы с текущей версией исходной программы на локальном компьютере создается рабочий каталог, который не должен находиться в одном каталоге с репозиторием. Рабочий каталог создается отдельно для каждого программного проекта. Локальная копия модуля, полученная из репозитория в рабочий каталог с помощью CVS-клиента, называется *рабочей копией*.

Упрощенная схема системы работы CVS включает следующие основные шаги:

– создание хранилища (репозитория);

– создание рабочего каталога и размещение в нем текущей версии отлаживаемой программы;

– сохранение текущей версии исходного файла в репозитории.

Отладка и изменение исходного файла должны выполняться в рабочем каталоге. После внесения изменений текущая версия передается в хранилище с соответствующим комментарием. После передачи в хранилище соответствующий проект может быть удален из рабочего каталога, чтобы исключить возможность внесения в него изменений без поддержки CVS. Имя каталога, в котором располагается хранилище, можно записать в глобальную переменную $CVSROOT для того, чтобы не указывать его в каждой команде CVS.

Общий синтаксис команд CVS выглядит следующим образом:

cvs [опции_программы] команда [опции _команды] [аргументы]

Здесь *опции_программы* используются для управления режимами CVS, *опции_команды* уточняют действие конкретной команды, *аргументы* задают имена объектов, над которыми выполняется команда. В таблице 4.3 приведены основные команды CVS.

Таблица 4.3

Основные команды CVS

Команда	Назначение	Синтаксис
init	создать репозиторий	cvs init имя_каталога
add	добавить в репозиторий новый файл или каталог	cvs add имя_файла cvs commit -m [«комментарий»] имя_файла
checkout	связать репозиторий с рабочим каталогом извлечь из репозитория копию исходного текста в рабочий каталог проекта	cvs -d имя_репозитория checkout -l имя_каталога *здесь ключ -l задает локальный режим работы команды checkout (без подкаталогов репозитория).* cvs checkout имя_проекта
commit	сохранить в репозитории изменения, внесенные в локальную копию	cvs commit -m [«комментарий»]
release	удалить файлы проекта из локального каталога	cvs release -d имя_проекта
remove	удалить файл из репозитория	cvs remove -f имя_файла cvs commit -m [«комментарий»] имя_файла
log	получить историю работы с файлом	cvs log [имя_файла]
diff, rdiff	просмотреть изменения файла от версии к версии	cvs diff -r версия_1 [-r версия_2] имя_файла
update	1) получить указанную версию файла 2) синхронизировать файлы в локальном каталоге и репозитории	cvs update -r номер_версии имя_файла cvs update [имя_файла]

Пример работы с CVS. Исходные данные — имя домашнего каталога **/home/group_1/user_1**, имя файла **file.c**.

1. Создание хранилища:

а) в домашнем каталоге создаем каталог **cvsroot**;

б) в каталоге **cvsroot** создаем хранилище:

cvs -d /home/group_1/user_1/cvsroot init

2. Создание рабочего каталога:

а) в домашнем каталоге создаем каталог **workdir**;

б) в каталоге **workdir** создаем каталог **project**, в котором будут находиться файлы проекта;

в) в каталог **project** запишем файл **file.c**

3. Связывание рабочего каталога с хранилищем:

а) сделать рабочий каталог текущим;

б) выполнить команду

cvs -d /home/group_1/user_1/cvsroot checkout -l

4. Передача проекта (каталога **project**) и файла **file.c** в хранилище следующими командами:

cvs -d /home/group_1/user_1/cvsroot add project

cvs -d /home/group_1/user_1/cvsroot add project/file.c

cvs -d /home/group_1/user_1/cvsroot commit

При выполнении команды **commit** будет вызван редактор **vi** для ввода комментария, например: «Пользователь user_1 передал файл file.c под управление CVS», после чего выполняются стандартные действия: ESC,»:», «wq». Далее для записи новых версий файла необходимо использовать только команду **commit**, так как каталог **project** и файл **file.c** уже добавлены в репозиторий.

5. Работа в рабочем каталоге (отладка файла **file.c**):

а) формируем исполняемый модуль для отладчика (**gcc** с ключом **-g**);

б) запускаем программу на выполнение и, если результат ее работы удовлетворяет всем требованиям, переходим к пункту ж);

в) запускаем отладчик, находим семантическую ошибку и выходим из отладчика;

г) запускаем редактор **vi** и исправляем ошибку;

д) переходим в пункт а);

е) исправленную версию записываем в хранилище.

6. Вывести обзор исправлений в программе **file.c**

cvs -d ~/cvsroot rdiff -r 1.1

7. Вывести журнал изменений версий файла **file.c**

cvs -d ~/cvsroot log file.c

Упрощенная схема работы CVS представлена на рисунке 4.3, где цифрами обозначены номера этапов из примера. Полное описание команд и ключей CVS можно найти в [19, 20].

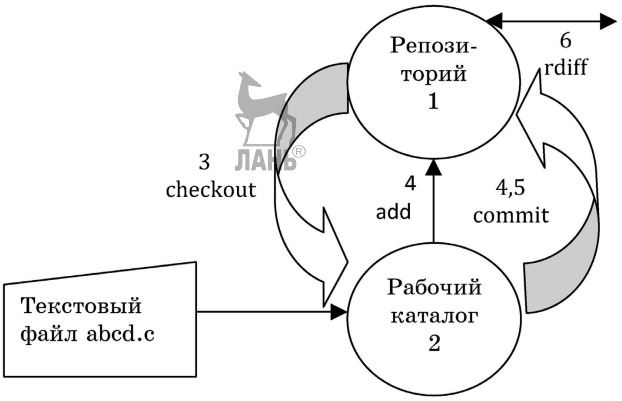


Рис. 4.3
Схема работы CVS

4.5. Сборка программы сложной структуры

Выше была описана технология обработки программы простой структуры, состоящей из одного программного модуля. Такая программа может содержать несколько функций, но физически представляет собой один файл, который обрабатывается компилятором. Простая структура характерна для небольших программ, разрабатываемых одним программистом.

Современные сложные программы разрабатываются коллективом программистов и состоят из большого числа программных модулей, представленных в виде набора файлов, каждый из которых компилируется и отлаживается независимо от других модулей. После отладки всех модулей проводится сборка программы в единый исполняемый файл.

В общем случае сборка программы сложной структуры должна выполняться в следующей последовательности:

– компиляция всех исходных модулей программы, представленных файлами типа **.c**; для исходных модулей, представленных файлами типа **.h**, отдельная компиляция не требуется;

– компоновка объектных модулей, сформированных в результате компиляции, с ранее откомпилированными модулями программы (файлы типа **.o**), служебными объектными модулями (файлы типа **.o**) и библиотеками (файлы типа **.a** и **.so**).

При разработке большой программы, состоящей из нескольких исходных файлов, приходится постоянно следить за файлами, которые требуют перекомпиляции после внесения изменений. Программа **make** освобождает пользователя от такой рутинной работы и служит для документирования взаимосвязей между файлами. Описание взаимосвязей и соответствующих действий хранится в текстовом файле, который должен иметь имя **makefile** или **Makefile**.

В общем случае **makefile** является списком правил. Каждое правило начинается с указателя, называемого целью, после которого стоит двоеточие, а далее через пробел указываются зависимости — имена файлов, от которых

зависит достижение цели. После зависимостей идут команды Linux, каждая из которых должна находиться на отдельной строке и начинаться с символа табуляции. Структура правила может быть сложной и включать ветвления, циклы и другие конструкции языка shell. Строки, которые начинаются с символа «#», являются комментариями.

Например, make-файл для программы, хранящейся в файле **abcd.c,** может иметь следующий вид:

```
# Makefile for abcd.c
# Compile abcd.c normaly
abcd: abcd.c
  gcc -o abcd abcd.c
# Compile abcd.c for debugging
testabcd: abcd.c
  gcc -o testabcd -g abcd.c
# End Makefile
```


Этот файл включает два правила компиляции и построения исполняемого модуля: первое предусматривает обычную компиляцию с построением исполняемого модуля с именем **abcd**, второе позволяет включить в исполняемый модуль **testabcd** информацию для отладки на уровне исходного текста.

В качестве имени правила может быть указана любая символьная строка, но рекомендуется использовать имена файлов или действие. Например, для сборки программы, состоящей из трех объектных модулей, фрагмент make-файла может выглядеть следующим образом:

```
Result: main.o func1.o func2.o
gcc -o Result main.o func1.o func2.o
```

Здесь целью является файл Result (исполняемый файл программы), а правило описывает, как можно получить новую версию этого файла (скомпоновать из перечисленных объектных файлов). В данном примере функции компилятора ограничиваются только вызовом компоновщика для сборки исполняемого модуля.

Make-файл должен находиться в одном каталоге со всеми файлами программы, запуск утилиты **make** проводится следующим образом: **make имя_правила,** например **make abcd.** Если не указывать какой-либо цели в командной строке, то **make** по умолчанию выбирает первое правило make-файла.

Сборка (компоновка) программы — процесс организации межпрограммных связей, позволяющий объединить независимо компилированные объектные модули в единый исполняемый (загрузочный) модуль. Компоновщик формирует из набора объектных модулей, каждый из которых имеет *собственную* относительную адресацию, загрузочный модуль, имеющий *единую* относительную адресацию. Такой модуль является перемещаемым, что позволяет операционной системе загружать его в любой свободный участок оперативной памяти.

Компоновщик последовательно разрешает все внешние ссылки, получая информацию о них из словарей ESD каждого объектного модуля, включаемого в формируемый исполняемый модуль. Процесс разрешения внешней ссылки заключается в поиске модуля с именем, указанным в ESD, и включении его в

состав исполняемого модуля, причем поиск может проводиться только в файлах, указанных пользователем, или в известных компоновщику библиотеках.

Компоновщик **ld** позволяет обеспечить максимально возможный контроль за процессом сборки исполняемого модуля, так как имеет большое число управляющих опций. Опции могут находиться в командной строке в любом порядке, ниже приведены некоторые из них:

-l*имя* — добавляет статическую библиотеку с именем **libимя.a** в список файлов для компоновки; например, опция -lm добавляет в список библиотеку математических функций libm.a;

-L*каталог* — добавляет указанный каталог в список каталогов для поиска библиотек;

-Map=*имя* — записывает в файл с указанным именем карту памяти исполняемого модуля, содержащую информацию об относительных адресах, именах и размерах всех объектных модулей, включенных в состав исполняемого модуля, а также диагностические сообщения, возникшие на этапе компоновки.

Компоновщик поддерживает два метода компоновки — статический и динамический. Статическая компоновка проводится путем включения в состав исполняемого модуля всех необходимых функций из библиотек, которые хранятся в файлах с расширением **.a**. Динамическая компоновка основана на том, что в состав исполняемого модуля включаются только ссылки на библиотечные модули, а непосредственное их извлечение из библиотек и загрузка в оперативную память проводятся на этапе выполнения программы. Для этого метода можно использовать только специальные библиотеки динамической компоновки (Dinamic Link Library, DLL), находящиеся в файлах типа **.so**.

Компоновщик может вызываться неявным или явным способами. Неявный вызов проводится с помощью компилятора и является наиболее простым способом компоновки. Применение явного вызова позволяет максимально полно использовать все возможности компоновщика, например сформировать карту памяти, но в этом случае необходимо указать имена всех дополнительных модулей, создающих среду выполнения программы.

Примеры:

а) неявный вызов

gcc func1.o main.o -o myprogram

б) явный вызов

ld -dynamic-linker /lib64/ld-linux-x86-64.so.2 -o myprogram_2 -Map=abcdmap_2 /usr/lib64/crt1.o /usr/lib64/crti.o /usr/lib/gcc/x86_64-redhat-linux/4.8.5/crtbegin.o main.o func1.o -lc /usr/lib/gcc/x86_64-redhat-linux/4.8.5/crtend.o /usr/lib64/crtn.o

Здесь в первом случае вызывается компилятор **gcc** и на его вход подаются два объектных файла — **func1.o** и **main.o**, результат записывается в файл **myprogram** и для компоновки используется стандартная библиотека **libc.a**, а во втором — вызывается компоновщик **ld**, которому на вход подаются не только файлы пользователя **func1.o** и **main.o**, но и служебные файлы (**crt1.o**, **crti.o**, **crtbegin.o**, **crtend.o**, **crtn.o**).

Назначение служебных модулей:

crt1.o — содержит код, инициализирующий среду исполнения языка C и вызывающий определенную пользователем функцию main;

crti.o — содержит пролог функции _init, помещаемый в начало секции .init;

crtn.o — содержит эпилог функции _init, помещаемый в конец секции .init;

crtbegin.o, crtend.o — обрабатывают глобальные конструкторы и деструкторы языка C++.

Обратите внимание: *в имени файла crt1.o используется символ «единица».*

Для того чтобы не указывать абсолютный путь к служебным файлам, при выполнении практического задания рекомендуется скопировать все эти файлы в ваш рабочий каталог. В этом случае в командной строке можно указывать только имена файлов.

На рисунке 4.4 приведен пример фрагмента карты памяти, содержащий план размещения машинного кода исполняемого модуля программы, исходный текст которой приведен в разделе 4.7. Секция кода имеет стандартное имя *.text*, каждая запись содержит имя секции, ее относительный адрес, размер и имя файла с программной функцией, расположенной по этому адресу. Все адреса и размеры выводятся в шестнадцатеричной системе счисления.

```
 .text           0x00000000004004f0       0x2c /home/NSTU/staff/kvg/obj/crt1.o
                 0x00000000004004f0                _start
 .text           0x000000000040051c        0x0 /home/NSTU/staff/kvg/obj/crti.o
 *fill*          0x000000000040051c        0x4
 .text           0x0000000000400520       0xbd /home/NSTU/staff/kvg/obj/crtbegin.o
 *fill*          0x00000000004005dd        0x3
 .text           0x00000000004005e0      0x16b abcd2.o
                 0x00000000004005e0                GetWords
                 0x00000000004006b7                main
 *fill*          0x000000000040074b        0x1
 .text           0x000000000040074c       0x82 printwords.o
                 0x000000000040074c                PrintWords
 *fill*          0x00000000004007ce        0x2
 .text           0x00000000004007d0       0x72 /usr/lib64/libc_nonshared.a(elf-init.oS)
                 0x00000000004007d0                __libc_csu_init
                 0x0000000000400840                __libc_csu_fini
 *fill*          0x0000000000400842        0x2
 .text           0x0000000000400844        0x0 /home/NSTU/staff/kvg/obj/crtend.o
 .text           0x0000000000400844        0x0 /home/NSTU/staff/kvg/obj/crtn.o
 *(.gnu.warning)
```

Рис. 4.4
Фрагмент карты памяти

Например, по адресу 0x4005E0 находится код объектного модуля **abcd2.o**, имеющий размер 363 байта, а по адресу 0x40074C — код модуля **printwords.o,** занимающий в памяти 130 байт.

4.6. Особенности разработки программ в платформе .NET

В связи с современным широким использованием сетевых технологий, когда необходимо организовать обмен программами между разнородными компьютерами, работающими под управлением различных ОС, возникает про-

блема переносимости ПО. Под переносимостью понимается способность корректной работы программы на различных аппаратных платформах и в среде различных ОС.

В предыдущих разделах описана технология сборки программ, способных выполняться только на одной аппаратной платформе в среде одной ОС (например, на компьютерах архитектуры x86, работающих под управлением Linux). Причиной является то, что разные семейства процессоров могут иметь разные наборы машинных команд, а ОС могут отличаться внутренней структурой исполняемых файлов. Поэтому при переносе сформированного компоновщиком исполняемого кода на другую аппаратную платформу или в среду другой ОС совместимость на уровне исполняемого кода не обеспечивается и необходимо выполнить полную повторную обработку исходной программы (компиляцию и сборку) для новой платформы.

Основным способом решения проблемы переносимости ПО стало создание некоторой универсальной среды исполнения, работающей на основе собственного платформенно-независимого промежуточного языка программирования. Одной из первых удачных реализаций такого способа явилось создание в 1995 г. языка программирования Java. Программы, написанные на Java, транслируются в промежуточный байт-код, который исполняется специальной средой исполнения, представленной виртуальной Java-машиной. Такие виртуальные машины разработаны для многих аппаратных и программных платформ, поэтому Java-программы являются кросс-платформенными.

Виртуальные машины Java обычно содержат интерпретатор байт-кода, однако для повышения производительности во многих машинах также применяется JIT-компиляция (Just in time) часто исполняемых фрагментов байт-кода в машинный код. При этом в текущий момент времени будет компилироваться лишь та часть приложения, к которой непосредственно идет обращение. Если мы обратимся к другой части кода, то она будет скомпилирована в машинный код, но ранее скомпилированная часть приложения сохраняется до завершения работы программы.

В 2002 г. фирмой Microsoft было предложено собственное комплексное решение для обеспечения переносимости программ — платформа .NET Framework [7, 22] для семейства ОС Windows, которая поддерживает языки программирования C#, VB.NET, JScript .NET, C++ и F#. Исходный код программ, написанных на любом из этих языков, компилируется в PE-файл (Portable Executable file), содержащий команды промежуточного языка CIL (Common Intermediate Language). PE-файл имеет расширение **.exe** и может выполняться только на компьютерах, где установлен .NET Framework.

В настоящее время существует также версия .NET Core, которая является кросс-платформенным аналогом .NET Framework с открытым исходным кодом и предназначена для работы в Windows, MacOS и Linux.

Подсистема .NET Framework содержит среду исполнения CLR (Common Language Runtime) и библиотеку классов FCL (Framework Class Library), состоящую из набора классов для работы со строками и числовыми данными, для параллельного вычисления, а также для создания Windows-приложений с графи-

ческим интерфейсом и веб-приложений с доступом через Интернет. Исполнение PE-файла проводится в среде CLR, при этом JIT-компиляция в машинный код осуществляется во время выполнения. Единицей компиляции является метод. Методы компилируются по мере их вызова, а скомпилированный код кэшируется и может использоваться повторно.

Программы в технологии .NET Framework создаются в форме сборок (модулей), представляющих собой самодостаточный компонент для развертывания, тиражирования и повторного использования. Сборка характеризуется следующими свойствами:

– формируется компилятором языка программирования;

– является единицей повторного использования кода;

– может существовать в виде выполняемого файла (с расширением EXE) или файла динамической библиотеки (с расширением DLL);

– в качестве идентификатора использует номер версии.

Сборка может иметь в своем составе один или несколько файлов с программным кодом, а также служебную информацию, которая называется манифестом. Манифест содержит следующие метаданные о сборке: идентификатор автора и версию сборки, список файлов с указанием по каждому из них контрольной суммы, права на запуск и использование, а также зависимости от других сборок, т. е. имена и версии сборок, которые используются данной сборкой.

В технологии .NET Framework легко решается проблема объединения программ, написанных на разных языках программирования. Для этого каждая из программ с помощью собственного компилятора переводится на промежуточный язык CIL и все откомпилированные файлы включаются в одну сборку.

Рассмотрим пример с простейшей программой, написанной на C#:

```
using System;
class MainApp
{
  public static void Main()
  {
    Console.WriteLine(«C# Hello, World!»);
  }
}
```

В результате компиляции программы будет сформирован PE-файл **HelloWorld. exe**, в котором находится промежуточный код программы на языке CIL и манифест, содержащий метаданные о сборке. Для просмотра манифеста можно использовать дизассемблер языка CIL, входящий в Microsoft .NET SDK, запустив его следующей командой:

```
ildasm.exe HelloWorld.exe
```

Окно программы дизассемблера показано на рисунке 4.5. После двойного щелчка мыши на строке М А Н И Ф Е С Т на экран будет выведено содержимое манифеста:

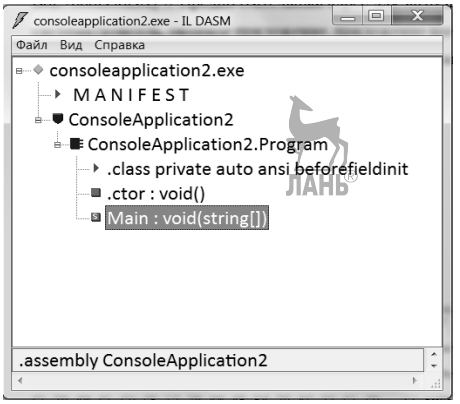


Рис. 4.5
Окно дизассемблера

```
//Описание нашей сборки
.assembly HelloWorld
{
// --- The following custom attribute is added
// automatically, do not uncomment -------
// .custom instance void [mscorlib]
// System.Diagnostics.DebuggableAttribute::.ctor(bool,
// bool) = ( 01 00 00 01 00 00 )
// Алгоритм по которому считается хеш
.hash algorithm 0x00008004
// Версия нашей сборки
.ver 0:0:0:0
}
// Название запускаемого файла
.module HelloWorld.exe
// MVID: {D48BD0D0-AE3E-4E46-AF65-B6E7886D36DA
}
// Предпочтительный адрес для загрузки сборки
.imagebase 0x00400000
// Подсистема (консоль, оконное приложение, приложение времени загрузки)
.subsystem 0x00000003
// Выравнивание секций
.file alignment 512
// Зарезервированный флаг
.corflags 0x00000001
// Image base: 0x02ca0000
```

Секция *.assembly extern mscorlib* предназначена для описания зависимостей от внешних сборок, используемых в данной программе, для каждой сборки указываются версия и контрольная сумма. Эти данные берутся из сборок при компиляции программы, что гарантирует во время работы приложения исполь-

зование именно тех сборок, которые использовались при компиляции и тестировании.

Секция *.assembly* описывает текущую сборку и содержит номер версии (*.ver)*; функцию, по которой будет вычисляться хеш-код (*.hash algorithm*); имя модуля (*.module*), описание подсистемы исполнения (*.subsystem*), информацию о выравнивании секций (*.file alignment*) и другие служебные данные.

В разных версиях .NET Framework набор данных, хранящихся в манифесте, может отличаться. Полное описание структуры манифеста и его наборов данных можно найти в MSDN [23, 27].

4.7. Практическое задание

4.7.1. Описание задания

Практическое задание выполняется на примере программы лексического анализатора, приведенного в [21] и написанного на языке С. Программа читает входной текст из стандартного ввода (клавиатуры) и результаты выводит на стандартный вывод (монитор). Для каждой строки, принятой из стандартного ввода, программа выводит слова по одному в строку. Словом считается последовательность алфавитно-цифровых символов, заключенная между пробелами. Если в качестве аргумента при запуске задан некоторый символ (например, -t), то на стандартный вывод из вводимой строки выводятся только слова, которые включают данный символ.

Ниже приведены два варианта исходного текста программы. Первый вариант представлен в виде программы простой структуры, в которой все функции находятся в одном файле **abcd.c**. Второй вариант реализован в виде программы сложной структуры, в которой одна из функций хранится в отдельном файле.

Текст программы содержит несколько синтаксических и семантических ошибок. При выполнении задания вы должны найти и исправить эти ошибки с целью приобретения практических навыков по использованию инструментальных средств **gcc**, **make**, **gdb, cvs**.

Тестовый входной набор данных будет представлен строкой:

this is test the abcd program

При запуске отлаженной программы **abcd** без параметров после ввода этой строки на экране должно быть:

this
is
test
the
abcd
program

При запуске отлаженной программы **abcd** с параметром **-t** после ввода этой строки на экране должно быть:

this
test
the

Во втором варианте программы функция **printwords()** вынесена в отдельный файл, который в основной функции **main()** описан как внешний. В общем случае возможны два способа описания внешних функций — с помощью предложений **extern** или **include**. В первом случае внешняя функция хранится в виде файла типа **.c**, компилируется отдельно от основной программы и включается в исполняемый файл компоновщиком при сборке программы. Во втором случае функция хранится в файле типа **.h** и подключается компилятором к основной программе на этапе препроцессорной обработки.

Аналогично можно вынести из главной программы функцию **getwords()**.

Вариант 1. Исходный текст программы простой структуры в виде одного файла.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
/* Manifests for state machine to parse input line. */
#define WORD 0
#define IGNORE 1
/* Globals, used by both subroutines. */
char *Words[BUFSIZ/2]; /* Worst case, single letters. */
int WordCount;
/* Walk through the array of works, find those with the matching character, printing them on * stdout. Note that the NULL character will match all words. */
void PrintWords(wc, match)
int wc; /* Number of words in Words[] */
char match; /* Attempt to match this charakter. */
{ register int ix; /* Index in Words[]. */
  register char *cp; /* Pointer for searching. */
  for (ix=0; ix < wc; ix++) {
    cp = Words[ix];
    /* Try to match the given character. Scan the word, attempting to match,
    * or until the end of the word is found. */
    while ((*cp) && (*cp++ != match));
    if (*cp == match) /* Found a match? Write the word on stdout. */
      (void) printf(«%s0, Words[ix]);} return; }
/* Find words in the gives buffer. The Words[] array is set
* to point at words in the buffer, and the buffer modifeid
* with NULL characters to delimit the words. */
int GetWords (buf)
char buf[]; /* The input buffer. */
{ register char *cp; /* Pointer for scanning. */
  int end = strlen(buf); /* length of the buffer. */
  register int wc = 0; /* Number of words found. */
  int state = IGNORE; /* Current state. */
  /* For each character in the buffer. */
  for (cp = &buf[0]; cp <&buf[end]; cp++) {
```

```
    /* A simple state machine to process
    * the current character in the buffer. */
    switch(state) {
    case IGNORE:
      if (!isspace(*cp)) {
        Words[wc++] = cp; /* Just started a word? Save it. */
        state = WORD; /* Reset the state. */ } break;
    case WORD:
      if (isspace(*cp)) {
        *cp = '\0'; /* Just completed aword? terminate it. */
        state = IGNORE; /* Reset the state. */ } break; }}
  return wc; /* Return the word count. */ }
int main(argc, argv) int argc; char *argv[]; { char buf[BUFSIZ], match;
/* Check command line arguments. */
if (argc < 2) match = ' ';
/* No command line argument, match all words. */
else match = *++argv[1]; /* match the char after the first - */
  /* Until no more input on stdin. */
  while(gets(buf) != (char *)NULL) {
    WordCount = GetWords(buf); /* Paste the input buffer. */
    PrintWords(WordCount, match); /* Print the matching words. */ }
  return(0); /* Return success to the shell. *
```

Вариант 2. Исходный текст программы сложной структуры в виде двух файлов.

Файл **abcd2.c**:

```
#include <stdio.h>
  #include <ctype.h>
  #include <string.h>
  /* Manifests for state machine to parse input line */
  #define WORD 0
  #define IGNORE 1
  /* Globals, used by both subroutines. */
  char *Words[BUFSIZ/2]; /* Worst case, single letters. */
  int WordCount;
  extern void PrintWords(int wc, char match, char *Words[]);
  int GetWords (buf)
  char buf[]; /* The input buffer. */
  { register char *cp; /* Pointer for scanning. */
    int end = strlen(buf); /* length of the buffer. */
    register int wc = 0; /* Number of words found. */
    int state = IGNORE; /* Current state. */
    /* For each character in the buffer. */
    for (cp = &buf[0]; cp <&buf[end]; cp++) {
      /* A simple state machine to process
      * the current character in the buffer. */
```

```
        switch(state) {
        case IGNORE:
          if (!isspace(*cp)) {
            Words[wc++] = cp; /* Just started a word? Save it. */
            state = WORD; /* Reset the state. */ } break;
      case WORD:
        if (isspace(*cp)) {
          *cp = '\0'; /* Just completed aword? terminate it. */
          state = IGNORE; /* Reset the state. */ } break; }}
    return wc; /* Return the word count. */ }
    int main(argc, argv) int argc; char *argv[]; { char buf[BUFSIZ], match;
    /* Check command line arguments. */
    if (argc < 2) match = '\0';
    /* No command line argument, match all words. */
    else match = *++argv[1]; /* match the char after the first - */
      /* Until no more input on stdin. */
      while(gets(buf) != (char *)NULL) {
        WordCount = GetWords(buf); /* Paste the input buffer. */
        PrintWords(WordCount, match, Words); /* Print the matching words. */ }
      return(0); /* Return success to the shell. */
    }
```

Файл **printwords.c**:

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>
  void PrintWords(wc, match, Words)
  int wc; /* Number of words in Words[] */
  char match;
  char *Words[BUFSIZ/2]; /* Attempt to match this character. */
  { register int ix; /* Index in Words[]. */
    register char *cp; /* Pointer for searching. */
    for (ix=0; ix < wc; ix++) {
      cp = Words[ix];
      /* Try to match the given character.
      * Scan the word, attempting to match,
      * or until the end of the word is found. */
      while ((*cp) && (*cp != match))
        ++cp;
      if (*cp == match) /* Found a match? Write the word on stdout. */
(void) printf(«%s\n», Words[ix]);
  }
  return;
}
```

4.7.2. Выполнение задания

1. Создайте в домашнем каталоге подкаталоги **examples, cvsroot** и **workdir.** В подкаталоге **examples** разместите файл **abcd.c,** содержащий первый вариант программы **abcd** (см. описание задания). В подкаталоге **workdir** создайте каталог **project**.

2. С помощью редактора **vi** создайте в подкаталоге **examples** make-файл, приведенный в разделе 4.5. Далее различные варианты построения исполняемого модуля должны быть получены с помощью программы **make**.

3. Выполните компиляцию файла **abcd.c** без добавления отладочной информации. Найдите и исправьте все синтаксические ошибки с помощью текстового редактора **vi**.

4. Запустите исполняемый модуль с устраненными синтаксическими ошибками, проверьте работу программы на соответствие заданным требованиям. Если требования выполняются, то перейдите к п. 10, иначе выполните следующий пункт задания.

5. Создайте репозиторий CVS в подкаталоге **cvsroot** (см. раздел 4.4).

6. В каталог **project** скопируйте файл **abcd.c** с устраненными синтаксическими ошибками.

7. Передайте каталог **project** и файл **abcd.c** в репозиторий. При выполнении команды **commit** с помощью редактора **vi** введите комментарий, например: «Пользователь user_1 передал файл abcd.c под управление CVS».

8. Выполните компиляцию файла **abcd.c** с добавлением отладочной информации и с помощью отладчика **gdb** выполните поиск и устранение семантических ошибок в программе. Каждое исправление в программе должно сопровождаться записью в репозиторий новой версии с комментарием, поясняющим сущность исправлений (например, номер строки программы **abcd.c** и причина исправления). После получения корректных результатов с помощью редактора **vi** в начало отлаженной программы введите комментарий: «Программа abcd отлажена пользователем user_1 с помощью отладчика gdb».

9. Просмотрите с помощью CVS историю модификации содержимого файла **abcd.c**.

10. В подкаталоге **examples** разместите файлы **abcd2.c** и **printwords.c**, содержащие второй вариант программы (см. описание задания).

11. Выполните компиляцию и сборку программы, используя неявный вызов компоновщика и задав имя исполняемого файла **adcd2_1**, проверьте корректность работы программы. При необходимости исправьте синтаксические и семантические ошибки, найденные при выполнении п. 3 и 8.

12. Выполните сборку программы, используя явный вызов компоновщика (см. пример в разделе 4.5), проверьте корректность работы программы. Результатом сборки должны быть исполняемый файл **abcd2_2** и карта памяти **abcd2_map.**

13. Из карты памяти **abcd2_map** определите размеры машинного кода модулей **abcd2.o** и **printwords.o**, сравните их с размерами исходного и объектного кода этих модулей (файлы типа **.c** и **.o)**.

14. Добавьте в make-файл, разработанный при выполнении п. 2, два новых правила, реализующие п. 11 и 12 задания. Проверьте корректность его работы.

Контрольные вопросы

1. Назовите и дайте характеристику основным этапам разработки приложений.
2. Компилятор **gcc**: назначение, основные этапы компиляции и опции.
3. Компоновщик **ld**: назначение, алгоритм сборки, явный и неявный вызов.
4. Отладчик **gdb**: назначение, основные команды.
5. Интерпретатор: назначение, принципы работы.
6. Типы ошибок в программах, поясните различия в принципе поиска синтаксических и семантических ошибок.
7. Утилита **make:** назначение, структура и принципы работы.
8. Назначение и архитектура CVS.
9. Основные этапы в схеме функционирования CVS.
10. Карта памяти: назначение и структура.
11. Поясните различия между программами простой и сложной структуры.
12. Понятие внешней функции, способы их описания в языках C и C++.
13. Карта памяти: назначение и структура. На основании рисунка 4.3 нарисуйте план расположения в памяти основных функций программы **adcd2.c.**
14. Отличия платформы .NET от стандартного подхода к разработке программ.
15. Что составляет основу платформы .NET?
16. Компилятор JIT, его функциональное назначение.
17. Понятие сборки и метаданных

5. ОПЕРАЦИОННЫЕ СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ

Прочитав эту главу, вы узнаете:

– понятие системы реального времени;

– отличия операционной системы реального времени от операционной системы общего назначения;

– параметры операционных систем реального времени;

– теоремы о верхней границе коэффициента использования процессора и о времени завершения;

– определение следующих терминов: жесткое и мягкое реальное время; время реакции на прерывание, время переключения контекста, точка планирования.

5.1. Общие сведения о системах реального времени

В настоящее время во многих областях техники необходимо решать различные задачи управления сложными техническими объектами и технологическими процессами в режиме реального времени, который требует разработки программ, обеспечивающих не только выполнение заданного алгоритма, но и получение результата в течение определенного интервала времени. Такие задачи называются задачами реального времени. Области применения систем реального времени — авиация, космонавтика, энергетика, транспорт, робототехника, машиностроение и др.

Системой реального времени (СРВ) называется аппаратно-программный комплекс, реагирующий в предсказуемое время на непредсказуемый поток внешних событий. Можно сказать, что СРВ — это система, время отклика которой на внешние события настолько мало, что она может изменять ход этих событий. Это означает, что:

– система должна успеть отреагировать на событие, произошедшее на объекте управления в течение времени, критичного для этого события. Время реакции системы должно быть предсказано при создании системы, а отсутствие реакции в предсказанное время считается ошибкой;

– система должна успевать реагировать на одновременно происходящие события. Даже если несколько внешних событий происходят одновременно, то система должна успеть отреагировать на каждое из них в течение заданного интервала времени.

По точности соблюдения критических интервалов различают системы *жесткого реального времени* (hard realtime, HRT) и системы *мягкого реального времени* (soft realtime, SRT). Системы HRT не допускают никаких задержек реакции системы ни при каких условиях, так как в лучшем случае результаты могут оказаться бесполезными, в худшем — стоимость опоздания может оказаться бесконечно велика или может произойти катастрофа.

Система HRT должна гарантировать время выполнения всех задач, т. е. при выполнении любого набора задач все они должны остаться внутри своих

временных границ. Такая система должна быть предсказуемой и никогда не опаздывать с реакцией на событие. К системам HRT относятся бортовые системы управления, системы аварийной защиты, регистраторы аварийных событий.

Системы SRT характеризуются тем, что задержка реакции не критична, хотя и может привести к увеличению стоимости формирования результатов или снижению производительности системы в целом. Примерами SRT-систем могут служить различные видеокодеки, которые могут иногда не успевать обрабатывать видеокадры, или сетевые системы передачи данных на основе стека протоколов TCP/IP, предусматривающие повторную передачу в случае потери некоторых пакетов данных.

Для реакции на одновременно возникающие события СРВ должна быть многозадачной. При этом задачи вынуждены делить между собой процессорное время, а для управления вычислительным процессом в этом случае необходимо использовать операционную систему (ОС), выбор которой имеет принципиальное значение для правильного функционирования СРВ.

Классические ОС общего назначения семейств Linux и Windows не могут использоваться для управления задачами реального времени, так как они разрабатывались с целью эффективного управления ресурсами вычислительных систем. Время реакции на события для таких ОС не является основным параметром, так как даже на уровне алгоритмов время реакции и эффективность использования ресурсов являются противоречивыми требованиями. В операционных системах реального времени (ОСРВ) основной целью является обеспечение режима жесткого реального времени для функционирования приложений реального времени.

Задачи реального времени могут классифицироваться как периодические и апериодические. Периодические задачи — это задачи, которые входят в состояние выполнения с постоянным периодом времени и обычно характеризуются жестким крайним сроком. Апериодические задачи — задачи, выполнение которых связано с возникновением некоторых внутренних или внешних событий (например, внезапное появление пешехода перед беспилотным автомобилем). Такие задачи могут характеризоваться жестким или мягким крайним сроком.

Не следует путать термины «система реального времени» и «операционная система реального времени». ОСРВ является, с одной стороны, инструментом разработки СРВ, а с другой — системой управления вычислительными ресурсами СРВ, работающей на объекте управления. При конфигурировании программного обеспечения СРВ разработчик включает в загрузочный образ только те компоненты ОСРВ, которые необходимы для функционирования конкретной СРВ.

Более полную информацию по СРВ можно найти в [8, 9].

5.2. Особенности операционных систем реального времени

Основные отличия ОСРВ от ОС общего назначения:

1) в ОСРВ основной критерий эффективности — обеспечение заданных временных характеристик при возникновении заранее определенных событий на объекте управления;

2) применение ОСРВ всегда связано с аппаратурой, с объектом управления и с событиями, происходящими на объекте. ОСРВ ориентирована на обработку внешних событий, что приводит к существенным отличиям в архитектуре, в функциях ядра и в построении системы ввода-вывода. Пользовательский интерфейс ОСРВ может быть похож на интерфейс ОС общего назначения;

3) ОСРВ служит инструментом для создания конкретного аппаратно-программного комплекса реального времени. Поэтому пользователями ОСРВ являются разработчики систем реального времени, систем управления и сбора данных, которые всегда знают перечень событий, которые должны контролироваться системой, и критические сроки обслуживания каждого из этих событий;

4) в ОСРВ обычно имеется разграничение системы исполнения и системы разработки. *Система исполнения* ОСРВ — минимальный набор инструментов (ядро, драйверы, исполняемые модули), обеспечивающих функционирование приложения реального времени. Система исполнения и компьютер, на котором она исполняется, называют «целевой» (target) системой. Большинство современных ОСРВ поддерживают множество аппаратных архитектур, на которых работают системы исполнения (Intel, Motorola, MIPS, PowerPC и др.).

Система разработки — набор инструментов, обеспечивающих создание и отладку приложений реального времени (компиляторы, отладчики, системы управления версиями и т. д.). Системы разработки могут быть встроенными в ОСРВ (резидентными) или внешними (кроссовыми), работающими в ОС общего назначения, например в Windows или в Linux. Средства разработки ОСРВ отличаются от обычных систем разработки наличием специфических инструментов для удаленной отладки, профилирования (измерение времени выполнения отдельных участков кода), эмуляции целевого процессора, отладки взаимодействующих процессов, моделирования;

5) планировщик ОСРВ имеет следующие особенности по сравнению с планировщиками ОС общего назначения:

– используется вытесняющая многозадачность с применением приоритетов, для потоков с равными значениями приоритетов используются алгоритмы FIFO или круговой циклический (RR);

– используется большой диапазон значений приоритетов для обеспечения возможности разбиения приложения реального времени на множество задач;

– имеется обязательный контроль инверсии приоритетов, когда поток с высоким приоритетом ожидает освобождения ресурса, захваченного потоком с низким значением приоритета;

6) система управления памятью в ОСРВ должна удовлетворять следующим требованиям:

– время доступа к памяти должно быть предсказуемым, поэтому для процессов реального времени необходимо использовать статическое распределение памяти и нельзя использовать подкачку страниц из виртуальной памяти;

– нельзя использовать процедуру «сборки мусора» (дефрагментацию памяти), во время проведения которой все процессы не могут исполняться, что приводит к непредсказуемости временных характеристик.

5.3. Основные параметры операционных систем реального времени

Основными параметрами ОСРВ являются время реакции на прерывание, время переключения контекста, размер системы и возможность исполнения из ПЗУ.

Время реакции на прерывание — это интервал времени между возникновением запроса на прерывание и до выполнения первой инструкции обработчика этого прерывания. Обычно между моментом возникновения события на объекте и началом обработки прерывания проходит определенная последовательность действий (рис. 5.1):

– событие регистрируется датчиками (момент времени t_1);

– данные с датчиков передаются в модули ввода-вывода системы (момент времени t_2);

– модули ввода-вывода преобразуют информацию от датчиков в цифровую форму и генерируют запрос на прерывание в управляющем компьютере, подавая ему сигнал о том, что на объекте произошло событие (момент времени t_3);

– система запускает программу обработки этого события (момент времени t_4).

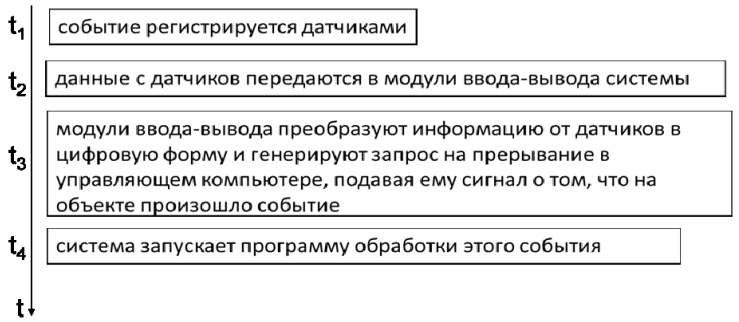


Рис. 5.1
Последовательность обработки события на объекте

Время реакции на событие зависит от нескольких временных интервалов:

$$T_c = T_д + T_в + T_{пр},$$

где T_c — время реакции на событие; $T_д$ — время реакции датчиков и канала связи ($t_2 - t_1$); $T_в$ — время реакции модулей ввода-вывода ($t_3 - t_2$); $T_{пр}$ — время реакции на прерывание ($t_4 - t_3$).

Первые два слагаемых полностью определяются аппаратурой сбора и преобразования данных и не зависят от ОСРВ, последнее слагаемое определяется свойствами ОСРВ и архитектурой компьютера. Время реакции на прерывание нужно уметь оценивать в худшей для системы ситуации, то есть в предположении, что процессор загружен, что в это время могут происходить другие прерывания и т. д. Время реакции на прерывание для ОСРВ обычно составляет единицы микросекунд.

Время переключения контекста — интервал времени, требуемый для передачи управления от одного потока другому. Все современные ОСРВ являются многозадачными, т. е. имеют возможность одновременной обработки нескольких событий. Переключение процессора между потоками, обрабатывающими разные события, требует определенных затрат времени, связанных с необходимостью сохранения текущего состояния прерываемого потока и восстановлением состояния потока, который будет исполняться. Разработчик СРВ должен достоверно знать время переключения контекста для обеспечения режима жесткого реального времени. Для современных ОСРВ это время составляет десятки микросекунд.

Для систем реального времени важным параметром является *размер системы исполнения*, т. е. минимально необходимого для работы приложения системного набора программ. Это позволяет создавать компактные встроенные СРВ повышенной надежности с ограниченным энергопотреблением, без внешних накопителей и с возможностью исполнения системы из постоянного запоминающего устройства (ПЗУ). Например, размер ядра ОСРВ OS9 составляет 22 Кбайт, VxWorks — 16 Кбайт, QNX — 8 Кбайт, CTOS — 4 Кбайт, что позволяет целиком разместить их во внутреннем кэше современных процессоров.

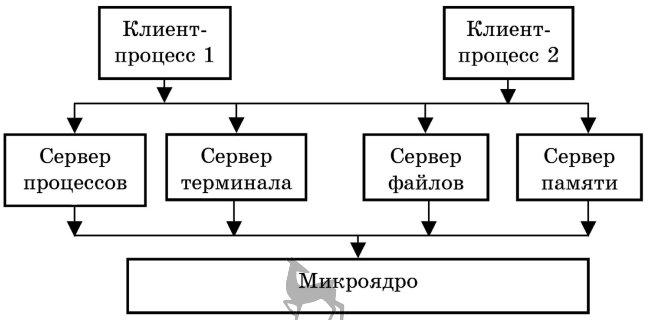


Рис. 5.2
Архитектура микроядерной ОСРВ

Большинство современных ОСРВ имеют клиент-серверную микроядерную архитектуру (рис. 5.2). В таких системах с аппаратной частью работает только микроядро, реализующее минимальный набор функций. Все остальные функции ОС выполнены в виде набора отдельных модулей, реализующих процессы-серверы (например, сервер памяти, сервер файлов, сервер процессов и т. д.), которые выполняются в пользовательском режиме, а не в режиме ядра. Пользовательские процессы выступают в роли клиентов, которые обращаются с запросами к серверным процессам.

В настоящее время на рынке ОСРВ присутствует достаточно много коммерческих и свободно распространяемых систем. Однако специфика применения ОСРВ требует юридических гарантий надежности, поэтому наиболее распространенными являются коммерческие системы VxWorks, QNX, OS-9, PSOS, LynxOS, VRTX. В таблице 5.1 приведены максимальные значения времени реакции на прерывание и переключение контекста для некоторых ОСРВ.

Таблица 5.1

Параметры некоторых ОСРВ

ОСРВ	Время реакции на прерывание (мкС)	Время переключения контекста (мкС)
pSOS	4	120
VRTX	5	145
LynxOS	7	165
VxWorks	4,5	47
PDOS	4,5	90
QNX	4	30

Основную долю рынка ОСРВ занимают системы VxWorks и QNX, временные характеристики которых приведены в таблице 5.2. Далее особенности ОСРВ будем рассматривать на примере операционной системы QNX, подробные сведения о которой можно найти в [10, 24, 25].

Таблица 5.2

Сравнительные параметры ОСРВ QNX и VxWorks

Параметр	QNX Neutrino 6.1		VxWorks AE 1.1	
	Среднее	Максим.	Среднее	Максим.
Время реакции на прерывание, мкс	1,7	4,1	1,7	6,8
Время переключения между двумя потоками в одном процессе, мкс	2,0	8,1	2,9	15,5
Время переключения между 10 потоками в одном процессе, мкс	2,4	7,4	3,4	16,5
Время переключения между 128 потоками в одном процессе/разных процессах, мкс	3,3/7,2	11,6/15,9	6,5/6,8	29,6/46,8

5.4. Операционная система реального времени QNX

С точки зрения улучшения временных характеристик наиболее важными для ОСРВ являются следующие факторы: общая архитектура системы; способы межпоточного взаимодействия; алгоритмы планирования потоков и организация службы времени.

5.4.1. Архитектура системы QNX

Архитектура системы представлена на рисунке 5.3. Основным компонентом QNX является модуль **procnto,** который содержит компактное микроядро, выполняющее все функции по управлению потоками, и администратор процессов, предназначенный для управления процессами. Администратор процессов и администраторы всех других системных ресурсов работают в режиме серверов, выполняя запросы со стороны пользовательских приложений — клиентов. Все программы связаны микроядром и взаимодействуют между собой с помощью механизма сообщений, показанного на рисунке в виде программной шины.

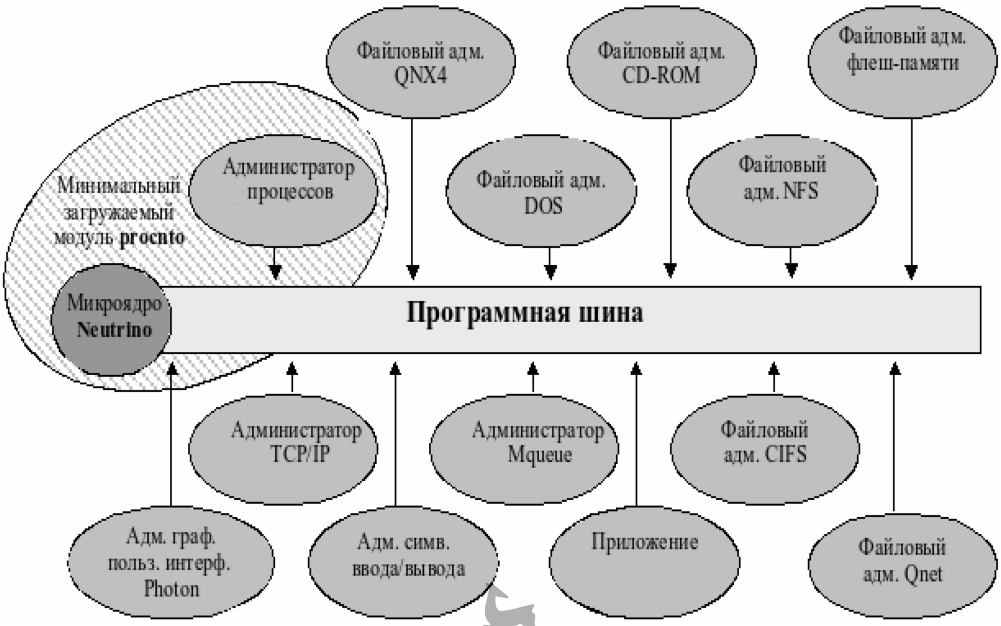


Рис. 5.3
Архитектура QNX

Достоинствами такой архитектуры являются высокая надежность, простота конфигурирования системы и отладки системных сервисов. Сбой в работе любого из системных процессов, кроме **procnto**, не влияет на работу остальных процессов. Недостатки архитектуры — частое переключение контекста с высокими накладными расходами на выполнение ряда операций.

Микроядро QNX — это библиотека, содержащая набор функций, доступ к которым прикладная программа получает с помощью специальных функций вызовов ядра. Вызовы ядра в QNX вытесняемы, т. е. если вызов ядра выполняется в интересах одного потока и более и приоритетный поток тоже вызвал ядро, то низкоприоритетный вызов уступит процессор высокоприоритетному. При этом время реакции на прерывание уменьшается, но увеличиваются затраты времени на переключение контекста вызовов ядра.

Микроядро QNX имеет размер 8 Кбайт, поддерживает только управление потоками (планирование, синхронизация), базовые механизмы межпоточного взаимодействия, обработку прерываний, сетевые службы нижнего уровня и содержит всего 14 системных вызовов. Все другие системные службы взаимодействуют с ядром и друг с другом через строго определенные интерфейсы сообщений.

Объектами управления модуля **procnto** являются процессы и потоки, основные сведения о которых приведены в разделе 2. Напомним, что модель процесса базируется на двух концепциях — группирование ресурсов и выполнение программы, т. е., с одной стороны, процесс — это набор ресурсов (учетная информация, адресное пространство, открытые файлы и т. д.), а с другой стороны, это поток исполняемых команд. Процессы, которые в любой момент времени

могут быть прерваны и возобновлены планировщиком, называются задачами. Поскольку микроядро QNX управляет только потоками, в дальнейшем мы будем в основном использовать термины «поток» и «задача».

Серверные потоки в QNX, как и вызовы ядра, являются вытесняемыми, их приоритеты являются клиент-управляемыми и устанавливаются равными максимальному из приоритетов потоков-клиентов, заблокированных в ожидании освобождения этого потока-сервера, что позволяет существенно уменьшить время реакции.

5.4.2. Механизмы межпоточного взаимодействия

Система QNX поддерживает основные POSIX-совместимые способы межпроцессного взаимодействия: очередь сообщений (реализована в администраторе очередей **mqueue**); разделяемую память (реализована в администраторе процессов); именованные и неименованные каналы (реализованы в администраторе файловой системы QNX4 и в администраторе каналов **pipe** соответственно) и т. д. Однако наиболее эффективными и быстрыми являются собственные способы взаимодействия потоков, реализованные в микроядре QNX — синхронные сообщения и импульсы. Именно эти способы используются для организации программной шины, через которую взаимодействуют между собой все компоненты QNX.

Синхронные сообщения являются основным механизмом обмена данными. Они могут иметь произвольный размер и позволяют организовать синхронный обмен данными без использования специальных методов синхронизации. На рисунке 5.4 приведена типовая схема взаимодействия двух потоков (сервера и клиента) с помощью механизма синхронных сообщений.

Здесь поток-сервер после запуска выполняет функцию приема сообщений MsgReceive(), переходит в режим приема и становится Receive-блокированным до тех пор, пока поток-клиент не отправит ему сообщение с помощью функции MsgSend(). После получения сообщения сервер разблокируется и проводит обработку сообщения, а клиент в это время ожидает ответа, находясь в состоянии Send-блокировки. После нормального завершения обработки сервер отправляет ответ клиенту с помощью функции MsgReply(), при аварийном — с помощью функции MsgError() и продолжает свое выполнение без блокирования, а ОС в это время будет передавать сообщение потоку-клиенту.

Операции отправки и приема сообщения MsgSend() и MsgReceive() являются блокирующими и синхронизирующими, а операции отправки ответа и отправки кода ошибки MsgReply() и MsgError() — не блокирующими. В результате потоку-отправителю не нужно делать непосредственный запрос на блокирование, чтобы дожидаться ответа, как это потребовалось бы при другой форме межпоточного взаимодействия.

Для увеличения производительности системы администраторы ресурсов организованы как многопоточные серверы, т. е. они могут одновременно обслуживать запросы нескольких пользовательских потоков. Для этого сервер должен создать с помощью функции *ChannelCreate()* специальную точку входа — канал, к которому через функцию *ConnectAttach()* подключаются клиен-

ты. Сообщения передаются не напрямую от потока к потоку, а в направлении каналов и соединений. Клиенту для передачи сообщения нужно знать узел сети, на котором выполняется поток-сервер (задается дескриптором узла); PID-процесса, содержащего поток-сервер и номер канала (ChannelID, CHID). Можно считать, что канал в QNX является аналогом сокета в сетевом протоколе TCP/IP.

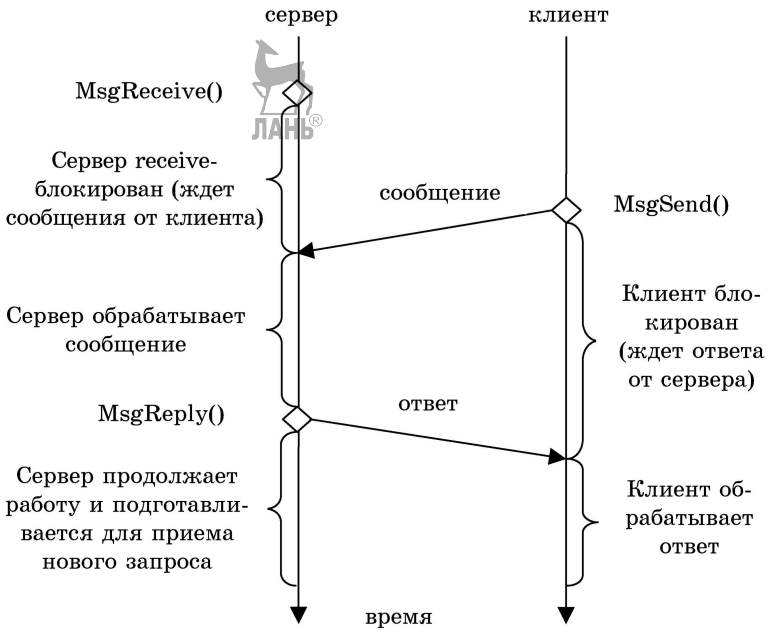


Рис. 5.4
Схема взаимодействия двух потоков

Импульсы — это неблокирующие сообщения малого фиксированного размера. Основные свойства импульса:

– имеет размер 5 байт (8-битный код и 32 бита данных);

– является не блокирующим для отправителя;

– может быть получен как синхронное сообщение;

– ставится в очередь, если получатель не заблокирован в ожидании сообщения.

Импульсы, как и синхронные сообщения, передаются по соединению клиента к каналу сервера. В отличие от синхронных сообщений, после приема импульса на него нельзя ответить, т. е. вызвать функцию *MsgReply()*. Отправка импульса проводится с помощью функции *MsgSendPulse(),* а прием — с помощью общей функции приема сообщений *MsgReceive()* или специальной функции *MsgReceivePulse()*. Функция *MsgReceive()* возвращает ноль для импульса или идентификатор сообщения для обычного сообщения.

Ниже приведен пример программы, в которой потоки взаимодействуют по схеме, изображенной на рисунке 5.4. Поток-сервер основан на функции *server()*, он создает канал с помощью функции *ChannelCreate()* и обрабатывает

сообщения в бесконечном цикле. Поток-клиент основан на функции *client()*, причем запускаются два экземпляра клиента: один посылает обычное сообщение, другой — импульс.

При приеме сообщения сервер анализирует идентификатор сообщения, возвращаемый функцией *MsgReceive()*. Если идентификатор сообщения равен нулю, то это импульс и на него не нужно отвечать.

Пример:

```
#include<pthread.h>
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<sys/neutrino.h>
#define BUFFERSIZE 50
int chid; // идентификатор канала
void *server() // поток-сервер
{
  int rcvid;
  int i=0;
  _int8 code;
  int value;
  char receive_buf[BUFFERSIZE], reply_buf[BUFFERSIZE];
  printf(«# Server thread: Channel creating ...»);
  // создание канала с опциями по умолчанию и запись в chid номера канала
  chid=ChannelCreate(0);
  if(chid<0)
  {
    perror(«Server error»);
    exit(EXIT_FAILURE);
  }
  printf(«# CHID = %d\n», chid);
  printf(«# Server thread: Listen to channel %d\n», chid);
  while(1)// сервер работает в цикле
  {
  // принимаем сообщение из канала с номером chid в буфер receive_buf
  // в rcvid записывается идентификатор полученного сообщения
    rcvid=MsgReceive(chid, &receive_buf, sizeof(receive_buf), NULL);
    if(rcvid>0) // получили обычное сообщение
    {
    printf(«# Server thread: message <%s> has received.\n», receive_buf);
    strcpy(reply_buf, «Answer from server»);
   printf(«# Server thread: answering with <%s> (Status=%d).\n», reply_buf, i);
  // отправляем ответ (буфер reply_buf) по номеру полученного сообщения (rcvid)
  // второй параметр (в данном случае переменная i)
```

```
            // статус ответа, обрабатывается клиентом.
              MsgReply(rcvid, i, &reply_buf, sizeof(reply_buf));
              i++;
              }
              if(rcvid==0)// получили импульс
              {
              code=receive_buf[4]; // 4-й байт — это код импульса (по структуре pulse)
              // байты с 8-го по 11-й — это данные, соберем их в переменную value
              value=receive_buf[11];
              value<<=8;value+=receive_buf[10];
              value<<=8; value+=receive_buf[9];
              value<<=8; value+=receive_buf[8];
              printf(«# Server thread: received pulse - code=%d, value=%d.\n», code, value);
              }
            }
          }
          void *client(void *parametr) // поток-клиент
          {
            int coid, status;
            _int8 code;
            int value;
            pid_t PID;
            pthread_t client;
            char send_buf[BUFFERSIZE], reply_buf[BUFFERSIZE];
            PID=getpid();
            client=pthread_self(); // получаем идентификатор потока-клиента
            printf(«> Client thread %d: connecting to channel ... «, client);
            // создаем соединение с каналом на текущем узле (0)
            // канал принадлежит процессу с идентификатором PID
            // номер канала — chid
            // наименьшее значение для COID — 0
            // флаги соединения не заданы — 0
            coid=ConnectAttach(0, PID, chid, 0, 0);
            // в coid записан идентификатор соединения или ошибочное значение меньше нуля
            if(coid<0)
            {
            perror(«Client error»);
            exit(EXIT_FAILURE);
            }
            printf(«COID = %d\n», coid);
            if(client%2==0) // четные потоки будут отправлять сообщения
            {
```

```
        strcpy(send_buf, «It's very simple example»);
        printf(«> Client thread %d: sending message <%s>.\n», client, send_buf);
        // отправляем сообщение из буфера send_buf в соединение coid
        // ответ принимаем в буфер reply_buf и статус записывается в перемен-
ную status
        status=MsgSend(coid, &send_buf, sizeof(send_buf), &reply_buf,
sizeof(reply_buf));
        printf(«> Client thread %d: I have replied with massage <%s>
(status=%d).\n», client, reply_buf, status);
        }
        else
        {// нечетные потоки будут отправлять импульсы
        code=20;
        value=12345;
        printf(«> Client thread %d: sending pulse - code=%d, value=%d.\n», code,
value);
        // посылаем импульс в соединение coid
        // приоритет импульса 20
        // код — code, данные — value
        MsgSendPulse(coid, 20, code, value);
        }
        // разрываем соединение coid
        ConnectDetach(coid);
        printf(«> Client thread %d: Good bye.\n», client);
        pthread_exit(NULL);
    }
    int main()
    {
        pthread_t client_tid1, client_tid2;
        printf(«Main thread: starting Server & Clients ...\n»);
        // создаем потоки сервера и двух клиентов
        pthread_create(NULL, NULL, server, NULL);
        sleep(1);
        pthread_create(&client_tid1, NULL, client, NULL);
        pthread_create(&client_tid2, NULL, client, NULL);
        printf(«Main thread: waiting for child threads exiting ...\n»);
        // ждем их завершения
        pthread_join(client_tid1, NULL); pthread_join(client_tid2, NULL);
        printf(«Main thread: the end.\n»);
        return EXIT_SUCCESS;
    }
    Результат:
    Main thread: starting Server & Clients ...
    # Server thread: Channel creating ... CHID = 1
    # Server thread: Listen to channel 1
```

> Client thread 3: connecting to channel ... COID = 3
> Client thread 3: sending pulse - code=20, value=12345.
Server thread: received pulse - code=20, value=12345.
> Client thread 3: Good bye.
Main thread: waiting for child threads exiting ...
> Client thread 4: connecting to channel ... COID = 3
> Client thread 4: sending message <It's very simple example>.
Server thread: message <It's very simple example> has received.
Server thread: answering with <Answer from server> (Status=0).
> Client thread 4: I have replied with massage <Answer from server> (status=0).
> Client thread 4: Good bye.
Main thread: the end.

Здесь сервер создал канал с идентификатором 1 (CHID), к которому клиенты-потоки подключаются через разные соединения с номерами 3 и 4 (идентификаторы COID). Поток с идентификатором 3 послал импульс, а поток с идентификатором 4 — сообщение. Поток 4 получил ответ от сервера, так как послал сообщение, а поток 3, отправивший импульс, ответа не ожидает.

5.4.3. Алгоритмы планирования задач

Основной функцией планировщика является распределение процессорного времени между готовыми к выполнению задачами. В ОСРВ используются в основном вытесняющие алгоритмы планирования, основанные на возможности прерывания активной задачи в случае поступления запроса от задачи с более высоким приоритетом. Такие алгоритмы могут быть статическими и динамическими.

Статические алгоритмы определяют план выполнения задач по их априорным характеристикам, назначая каждой задаче фиксированное значение приоритета на все время ее нахождения в системе. Они характеризуются низкими затратами ресурсов, но требуют полной предсказуемости. Динамические алгоритмы определяют текущий план во время исполнения задач и имеют возможность изменения их приоритета. Динамическое планирование связано со значительными затратами, но способно адаптироваться к меняющемуся окружению.

Примером статического вытесняющего алгоритма является алгоритм монотонных частот RMS (Rate Monotonic Scheduling), динамического — алгоритм EDF (Earliest Deadline First), отдающий предпочтение задачам с наиболее ранним предельным временем начала или завершения выполнения.

Алгоритм монотонных частот

Алгоритм монотонных частот (RMS) используется для периодических и апериодических задач. Основными характеристиками периодических задач являются максимальное время выполнения C (процессорное время, необходимое для завершения одного запуска), период запуска T и коэффициент использования процессора, под которым понимается отношение U = C/T.

Алгоритм RMS назначает приоритет задачи обратно пропорционально ее периоду, т. е. чем меньше период задачи, тем выше ее приоритет. Задачи должны удовлетворять следующим требованиям:

– задача должна быть завершена за время ее периода;

– задаче требуется одинаковое процессорное время на каждом периоде;

– задачи являются независимыми;

– прерывание задачи происходит мгновенно;

– приоритеты у всех задач должны быть фиксированными и иметь различные значения;

– апериодические задачи не имеют жестких сроков.

В теории планирования доказана теорема о верхней границе коэффициента использования процессора, устанавливающая необходимое условие для того, чтобы группа, состоящая из n задач, планируемых согласно алгоритму RMS, всегда удовлетворяла временным ограничениям:

$$C_1/T_1 + \ldots + C_n/T_n <= n\,(2^{1/n} - 1) = U(n),$$

где C_n и T_n — время выполнения и период n-й задачи соответственно.

В таблице 5.3 приведены результаты расчета согласно данному условию.

Таблица 5.3

Зависимость верхней границы от числа задач

Число задач	1	2	3	4	5	6	7	8	9	∞
Верхняя граница $U(n)$	1,000	0,828	0,779	0,756	0,743	0,734	0,728	0,724	0,72	0,69

Достоинство алгоритма монотонных частот заключается в том, что он сохраняет устойчивость в условиях краткосрочной перегрузки. Однако в этом случае возможен эпизодический пропуск выполнения задач с низким приоритетом.

Применим теорему о верхней границе коэффициента использования к трем задачам с допущением, что накладные расходы на контекстное переключение содержатся в оценке времени процессора. Здесь и далее время выражено в миллисекундах.

Задача z_1: $C_1 = 20$; $T_1 = 100$; $U_1 = 0{,}2$.

Задача z_2: $C_2 = 30$; $T_2 = 150$; $U_2 = 0{,}2$.

Задача z_3: $C_3 = 60$; $T_3 = 200$; $U_3 = 0{,}3$.

Полный коэффициент использования процессора для всех трех задач равен $0{,}7 < 0{,}779$, поэтому эти задачи будут всегда удовлетворять временным ограничениям. При увеличении времени выполнения третьей задачи до 90 мс ($U_3 = 0{,}45$) полный коэффициент использования процессора равен $0{,}85 > 0{,}779$, следовательно, теорема о верхней границе не дает гарантии выполнения всех задач. Поэтому необходимо получить более точную оценку с помощью теоремы о времени завершения.

Теорема о времени завершения дает более достоверный критерий для оценки производительности СРВ и используется, если для некоторого множества задач полный коэффициент использования процессора больше, чем требует теорема о верхней границе. Теорема утверждает, что множество незави-

симых периодических задач будет планируемым, если при их одновременном запуске каждая задача успевает завершиться в течение своего первого периода.

Теорема о времени завершения графически представляется с помощью временно́й диаграммы, на которой показана упорядоченная последовательность выполнения группы задач. Рассмотрим выполнение группы из трех задач со следующими характеристиками в однопроцессорной системе:

задача z_1: $C_1 = 20$; $T_1 = 100$; $U_1 = 0{,}2$;

задача z_2: $C_2 = 30$; $T_2 = 150$; $U_2 = 0{,}2$;

задача z_3: $C_3 = 90$; $T_3 = 200$; $U_3 = 0{,}45$;

суммарный коэффициент использования процессора — 0,85.

Их выполнение показано на временно́й диаграмме (рис. 5.5), темные участки обозначают интервалы выполнения задач. В худшем случае, когда все три задачи готовы к работе одновременно, первой запускается z_1, так как у нее самый короткий период и самый высокий приоритет. Она завершается через 20 мс, после чего в течение 30 мс исполняется задача z_2, а затем z_3. В конце первой точки планирования $T_1 = 100$ мс задача z_1 уже завершена и, следовательно, удовлетворяет временны́м ограничениям. Задача z_2 также завершила работу задолго до срока, а задача z_3 потратила 50 мс из необходимых 90 мс.

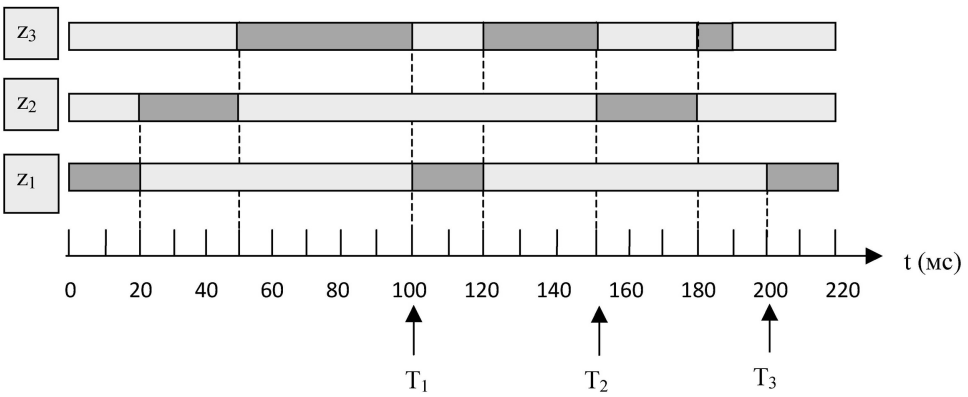


Рис. 5.5
Временна́я диаграмма RMS

В начале второго периода z_1 задача z_3 вытесняется задачей z_1, которая через 20 мс завершается и уступает процессор задаче z_3. Задача z_3 выполняется до конца периода T_2 (150 мс), который соответствует второй точке планирования. В этот момент z_3 использовала 80 мс из необходимых 90.

В момент времени T_2 задача z_2 вытесняет задачу z_3 и через 30 мс возвращает процессор задаче z_3, которая исполняется еще 10 мс, укладываясь в отведенное для нее время. Третья точка планирования T_3, которая расположена в конце второго периода задачи z_1 ($2T_1 = 200$) и одновременно в конце первого периода задачи z_3 ($T_3 = 200$). Из рисунка видно, что все задачи завершают ис-

полнение до конца своего первого периода, поэтому все вместе они удовлетворяют временны́м ограничениям.

На диаграмме видно, что процессор простаивает 10 мк перед началом третьего периода z_1 (этот момент совпадает с началом второго периода z_3). На протяжении интервала длительностью 200 мс процессор работал 190 мс, т. е. задействовался на 95%. По истечении времени, равного наименьшему общему кратному трех периодов (600 мс), средний коэффициент использования окажется равным 0,85.

Алгоритм EDF

Динамический алгоритм EDF устанавливает максимальный приоритет задачам, у которых раньше крайний срок исполнения, т. е. раньше всех наступает момент запуска. Планировщик EDF ведет отсортированную по крайним срокам очередь готовых к работе задач и всегда выбирает из очереди первую задачу. Каждая задача, перешедшая в состояние готовности, регистрируется в этой очереди и объявляет о своем крайнем сроке.

Запуск алгоритма EDF (точка планирования) возникает при завершении очередной задачи или переходе какой-либо задачи в состояние готовности. В последнем случае планировщик проверяет, не наступает ли крайний срок новой задачи раньше, чем у активной задачи, и при положительном ответе активная задача вытесняется новой задачей.

Основные свойства и допущения алгоритма EDF:

– не требует постоянства времени использования процессора задачами;

– может работать с периодическими и апериодическими задачами;

– каждая задача должна быть завершена за время ее периода;

– задачи являются независимыми;

– прерывание задач происходит мгновенно.

Рассмотрим работу алгоритма EDF на примере выполнения следующей группы задач:

задача z_1: $C_1 = 15$; $T_1 = 30$; $U_1 = 0{,}5$;

задача z_2: $C_2 = 15$; $T_2 = 40$; $U_2 = 0{,}375$;

задача z_3: $C_3 = 5$; $T_3 = 50$; $U_3 = 0{,}1$;

суммарный коэффициент использования процессора — 0,975.

В таблице 5.4 приведена схема работы алгоритма EDF в каждой точке планирования на интервале до 120 мс.

Таблица 5.4

Схема работы алгоритма EDF

Точка планирования, мс	Крайний срок завершения задачи, мс			Выбор для исполнения
	z1	z2	z3	
0	30	40	50	z1-1
15	Не готова	40	50	z2-1
30	60	Не готова	50	z3-1
35	60	Не готова	Не готова	z1-2
50	Не готова	80	100	z2-2
65	90	Не готова	100	z1-3

Продолжение табл. 5.4

Точка планирования, мс	Крайний срок завершения задачи, мс			Выбор для исполнения
	z1	z2	z3	
80	Не готова	120	100	z3-2
85	Не готова	120	Не готова	z2-3
100	120	Не готова	150	z1-4
115	Не готова	Не готова	150	z3-3

На рисунке 5.6 приведена временна́я диаграмма выполнения этой группы задач, из которой видно, что даже в высоконагруженной системе алгоритм справляется с обработкой всех задач. Для сравнения там же показана работа статического алгоритма RMS, который в момент времени t = 60 мс дает сбой из-за того, что задача z_3 не может быть выполнена в течение своего первого периода.

Рис. 5.6
Временна́я диаграмма EDF

5.4.4. Служба времени

Фиксация временны́х интервалов и хронометраж выполнения участков кода для ОСРВ более критичны, чем для ОС общего назначения. Основные свойства службы времени QNX можно сформулировать следующим образом:

– микроядро QNX работает в дискретной сетке времени;

– каждый единичный момент времени для микроядра — это такт или «тик» системного времени, который определяет разрешающую способность QNX на шкале времени;

– любые изменения состояний объектов фиксируются микроядром только в узлах этой дискретной шкалы времени;

– микроядро не различает два временных события, если они происходят между соседними тиками системного времени с интервалом меньшим, чем интервал системного тика.

После загрузки системы начальная длительность системного тика составляет 1 мс. Требования к системам реального времени формулируются стандар-

том POSIX 4, который устанавливает, что все события, определенные на временной шкале (завершения ожидания потоков, тайм-ауты и т. д.), наступали *не раньше истечения* установленного временного интервала. Поэтому реальное значение тика, установленного по умолчанию, составляет чуть меньше 1 мс (обычно 999 847 нс).

Значение тика может быть изменено с помощью функции *ClockPeriod(),* но не может быть меньше 10 мкс. Если какой-либо поток изменяет системный тик, то при этом изменяются все временные масштабы в системе, поэтому перед завершением поток должен обязательно восстановить начальное значение системного тика.

Получить значение системного тика можно с помощью функции *clock_getres():*

struct timespec res;
clock_getres(CLOCK_REALTIME, &res);
printf(«Resolution is %ld nanoseconds.\n», res.tv_nsec);

Результат выполнения:

Resolution is 999847 nanoseconds.

Очевидно, что погрешность отсчета временных интервалов возрастает при уменьшении значения этих интервалов и при интервалах, соизмеримых с величиной тика, она может быть достаточно высокой.

Основными объектами службы времени являются таймеры и тайм-ауты ядра. Таймеры используются для отсчета заданных интервалов времени при выполнении потоков, а тайм-ауты ядра — для ограничения времени нахождения потоков в блокированном состоянии.

Таймер может задаваться относительным или абсолютным временем. Относительный таймер срабатывает через заданный интервал времени (отсчет проводится от текущего времени), а абсолютный таймер срабатывает в заданный момент времени. Режим работы таймера может быть однократным (срабатывает только один раз) или периодическим. Периодический таймер срабатывает периодически, уведомляя поток о том, что истек определенный интервал времени.

Основные этапы работы с таймером: создание функцией *timer_create();* включение функцией timer*_settime(),* в которой можно задавать время срабатывания и режим работы таймера; удаление функцией *timer_delete().*

Создание тайм-аута ядра проводится с помощью функции *timer_timeout(),* в которой задаются тип блокировок, на которые будет установлен тайм-аут, и тип уведомления о событии. Блокировки, на которые может быть установлен тайм-аут, могут вызываться механизмами сообщений (прием или отправление сообщений) или синхронизации (ожидание освобождения мутекса, семафора и т. д.). Уведомления о наступлении тайм-аута передаются тремя способами: послать импульс, послать сигнал или создать поток (при наступлении события запустится поток, который будет выполнять заданную функцию).

Тайм-ауты запускаются при входе потока в заданное состояние блокировки, а сброс — при возврате из любого системного вызова. Это означает, что

необходимо заново запускать тайм-аут перед каждым системным вызовом, к которому он применяется.

Следующее приложение демонстрирует работу с таймером. Главный поток создает канал и сам же к нему подключается. Далее создается многократный таймер, уведомляющий импульсом. Главный поток принимает импульсы, которые сообщают, что прошел очередной интервал таймера, и выводит сообщение об этом на экран.

Пример:

```
#include<stdio.h>
#include<time.h>
#include<sys/netmgr.h>
#include<sys/neutrino.h>
// Задаем код импульса
// он должен быть между _PULSE_CODE_MINAVAIL и _PULSE_CODE_
MAXAVAIL
#define MY_PULSE_CODE _PULSE_CODE_MINAVAIL
int main()
{
  int i;
  struct sigevent event;
  struct itimerspec itime;
  timer_t timer_id;
  int chid, coid;
  struct _pulse impulse;
  // создаем канал
  chid=ChannelCreate(0);
  // и сами к нему соединяемся
  coid=ConnectAttach(ND_LOCAL_NODE, 0, chid, 0, 0);
  // устанавливаем структуру event на уведомление
  // импульсом с кодом MY_PULSE_CODE в канал coid.
  // импульс передается с приоритетом текущего потока,
  // который получен с помощью функции getprio(0).
  // последний аргумент равен нулю — данные импульса
  SIGEV_PULSE_INIT(&event, coid, getprio(0), MY_PULSE_CODE, 0);
  // создаем таймер
  timer_create(CLOCK_REALTIME, &event, &timer_id);
  // задаем время срабатывания таймера
  // таймер сработает через 1,8 с
  // (1 с + 800000000 нс = 1,8 с)
  itime.it_value.tv_sec=1;
  itime.it_value.tv_nsec=800000000;
  // таймер повторно сработает через 2,5 с
  // (2 с + 500000000 нс = 2,5 с)
  itime.it_interval.tv_sec=2;
  itime.it_interval.tv_nsec=500000000;
```

```
    // создаем относительный таймер (второй параметр равен нулю)
    timer_settime(timer_id, 0, &itime, NULL);
    // теперь мы получим импульс через 1,8 с и будем
    // получать повторные через 2,5 с
    for(i=10; i>0; i--)
    {
      MsgReceivePulse(chid, &impulse, sizeof(impulse), NULL);
      printf(«We got a pulse from our timer\n»);
    }
    // получили 10 импульсов и выходим
    return 0;
}
```

Результат выполнения:

```
We got a pulse from our timer
We got a pulse from our timer
We got a pulse from our timer
…
```

Сообщения выводятся каждые 2,5 с.

5.5. Практическое задание

Для выполнения практических упражнений в дополнение к разделу 5 желательно изучить литературу [25, 26].

1. Скачайте с сайта разработчика (*http://blackberry.qnx.com/en/products/tools/qnx-momentics*) бесплатную пробную версию интегрированной среды разработки QNX Momentics, которая может устанавливаться в среду Windows или Linux. При необходимости по адресу *blackberry.qnx.com/en/sdp7/sdp70_download* можно скачать пробную версию платформы разработки QNX (SDP) 7.0, которая включает 64-разрядную ОСРВ QNX Neutrino и QNX Momentics.
2. Выполните пример, приведенный в п. 5.4.2, проанализируйте результаты.
3. На основе примера разработайте программу, в которой клиент посылает серверу строки «TIME», «DATE», «RANDOM», а сервер соответственно возвращает время, дату или случайное число.
4. Выполните пример, приведенный в п. 5.4.4, проанализируйте результаты.
5. На основе примера разработайте программу, которая через каждые 5 с с помощью таймера выводит текущее время.
6. Разработайте программу для определения значения системного тика в QNX.
7. Постройте временны́е диаграммы алгоритмов RMS и EDF для следующей группы задач: $C_1 = 1$, $T_1 = 8$; $C_2 = 2$, $T_2 = 5$; $C_3 = 4$, $T_3 = 10$.

Контрольные вопросы

1. Назовите отличия между системами жесткого и мягкого реального времени.
2. Почему архитектура QNX называется клиент-серверной? Назовите достоинства и недостатки этой архитектуры.
3. Особенности QNX, направленные на уменьшение времени реакции.
4. Алгоритмы диспетчеризации потоков в QNX.
5. Синхронные сообщения QNX, достоинства и недостатки.
6. Импульсы QNX, достоинства и недостатки.
7. Назовите основные свойства службы времени QNX.
8. Таймеры и тайм-ауты QNX: назначение, типы, режимы работы.
9. Назовите допущения, положенные в основу алгоритма RMS, и поясните их.
10. Статические и динамические алгоритмы планирования, достоинства и недостатки.

КРАТКИЙ СЛОВАРЬ ТЕРМИНОВ

Абсолютное имя файла — имя, указывающее положение файла по отношению к корневому каталогу файловой системы.

Альтернативный поток данных — дополнительный набор данных, который может храниться совместно с основным потоком данных в одном файле. Альтернативные потоки имеют собственное имя и не отображаются большинством стандартных программ.

Внутренняя команда — команда, программный код которой содержится в командном интерпретаторе.

Внешняя команда — команда, программный код которой находится в отдельном исполняемом файле, запускаемом командным интерпретатором.

Внешнее запоминающее устройство (ВЗУ, внешняя память) — энергонезависимое устройство памяти большого объема с невысоким быстродействием.

Время реакции на прерывание — интервал времени между возникновением запроса на прерывание и до выполнения первой инструкции обработчика этого прерывания.

Время переключения контекста — интервал времени, требуемый для передачи управления от одного потока другому.

Главная загрузочная запись (MBR) — структура данных, хранящая таблицу разделов диска и программу для поиска и загрузки в ОЗУ загрузчика операционной системы.

Динамическая компоновка — способ сборки программы путем включения в состав исполняемого модуля ссылок на имена необходимых функций из динамических библиотек. Непосредственное подключение откомпилированного кода таких функций проводится на этапе выполнения программы.

Дорожка — концентрическая окружность на магнитной поверхности диска, на которую проводится запись данных, для записи и чтения каждой дорожки диска выделяется отдельная магнитная головка.

Индексный дескриптор (inode) — служебная структура данных файловых систем семейства ОС Linux, в которой хранятся все метаданные о файле, кроме его имени. Данные дескриптора используются для организации доступа к содержимому файла.

Исполняемый (загрузочный) модуль — полностью готовая к выполнению программа, сформированная в процессе сборки из независимо компилированных объектных модулей.

Исходный модуль — программа, написанная на языке программирования.

Карта памяти — текстовый файл, содержащий информацию об относительных адресах, именах и размерах всех объектных модулей, включенных в состав исполняемого модуля компоновщиком.

Кластер (блок) — минимальная единица дисковой памяти, выделяемая для хранения файлов.

Компилятор — программа, предназначенная для перевода программы, написанной на алгоритмическом языке, в набор двоичных машинных команд, который называется объектным модулем. В процессе перевода компилятор проверяет исходную программу на наличие лексических и синтаксических ошибок.

Компоновка (сборка) программы — процесс организации межпрограммных связей, позволяющий объединить независимо компилированные объектные модули в единый исполняемый (загрузочный) модуль. Выполняется специальной программой — компоновщиком.

Контроллер — устройство, предназначенное для управления периферийным оборудованием компьютера.

Логическая ошибка — ошибка в исходном коде программы, связанная с реализацией алгоритма решения задачи.

Логический адрес — адрес в адресном пространстве объектного или исполняемого модуля.

Логический раздел — раздел диска, информация о котором хранится в расширенном разделе и в котором установлена отдельная файловая система.

Магнитная головка — устройство для записи, стирания и считывания информации с магнитного носителя.

Менеджер логических томов (LVM) — компонент ОС, позволяющий использовать разные области одного жесткого диска и области нескольких жестких дисков как один логический раздел.

Объектный модуль — программа, представленная в виде набора машинных двоичных команд и служебных структур данных, используемых для сборки.

Оперативное запоминающее устройство (ОЗУ, оперативная память) — энергозависимое быстродействующее устройство памяти.

Операционная система — комплекс программ, предназначенный для управления всеми ресурсами компьютера и обеспечения интерфейса с пользователем.

Отладка — этап разработки, связанный с обнаружением, локализацией и устранением ошибок в программном обеспечении. Выполняется с помощью специальной программы — отладчика.

Относительный (логический) адрес — адрес в логическом адресном пространстве объектного или исполняемого модуля, созданный средой разработки программного обеспечения.

Относительное имя файла — имя, указывающее положение файла по отношению к текущему каталогу файловой системы.

Первичный раздел — раздел диска, информация о котором хранится в таблице разделов MBR.

Препроцессорная обработка — начальный этап работы компилятора, целью которого является подключение к исходной программе дополнительных исходных модулей, хранящихся в файлах с расширением .h. Имена дополнительных модулей указываются в основной программе директивами **#include**.

Раздел диска — часть внешней памяти, в которой установлена собственная файловая система.

Расширенный раздел — раздел диска, предназначенный для хранения информации об одном логическом разделе и ссылки на следующий расширенный раздел, если таковой существует.

Сектор — участок дорожки, хранящий минимальный объем информации, который может быть считан или записан за одно обращение к диску (обычно 512 байт).

Синтаксическая ошибка — ошибка в исходном коде программы, связанная с нарушением формального синтаксиса языка программирования.

Синхронные сообщения — способ обмена данными между потоками, основанный на одновременном использовании механизма обмена сообщениями и механизма синхронизации, является самым быстрым способом обмена данными в ОС QNX.

Система исполнения — минимальный набор прикладного и системного ПО, обеспечивающий функционирование приложения реального времени.

Система разработки — набор инструментов, обеспечивающих создание и отладку приложений реального времени (компиляторы, отладчики, системы управления версиями и т. д.).

Словарь внешних символов (ESD) — таблица в объектном модуле, содержащая имена всех функций, необходимых для выполнения исходной программы. Используется компоновщиком для сборки исполняемого модуля.

Словарь перемещений (RLD) — таблица в объектном модуле, содержащая информацию обо всех переменных и других адресных константах, используемых в объектном модуле. Используется компоновщиком для сборки исполняемого модуля.

Статическая компоновка — способ сборки программы путем включения в состав исполняемого модуля откомпилированного кода всех необходимых функций из статических библиотек. Подключение откомпилированного кода проводится перед загрузкой программы в ОЗУ.

Сценарий (скрипт, командный файл) — исполняемый текстовый файл, каждая строка которого является командой ОС или вызовом некоторой программы.

Таблица разделов диска — часть главной загрузочной записи, хранящая информацию о первичных разделах диска (имя, тип, адрес, размер и т. д.).

Таблица размещения файлов — таблица, используемая в файловых системах семейства FAT для хранения списка кластеров, выделенных для хранения файлов. Каждый элемент таблицы описывает состояние одного кластера диска.

Тестирование — процесс исследования и испытания программы с целью выявления ситуаций, в которых поведение программы является неправильным, нежелательным или не соответствующим требованиям.

Файловая система — 1) способ хранения информации на внешнем запоминающем устройстве и организация доступа к ней; 2) компонент операционной системы, реализующий весь комплекс действий по работе с информацией,

хранящейся во внешней памяти (распределение внешней памяти, контроль прав доступа, запись, удаление и т. д.).

Физический адрес — адрес в адресном пространстве оперативной памяти.

Цилиндр — совокупность дорожек с одинаковыми номерами на всех поверхностях магнитного диска.

ext — семейство файловых систем, используемое в ОС Linux, включает файловые системы ext2, ext3 и ext4.

FAT — 1) семейство файловых систем, используемых в ранних версиях ОС Windows, а также в устройствах внешней памяти мобильных устройств (телефонов, фотоаппаратов, видеорегистраторов и т. д.), включает файловые системы FAT12, FAT16, FAT32, VFAT; 2) таблица размещения файлов, используемая в файловых системах семейства FAT для хранения списка кластеров, выделенных для хранения файлов.

NTFS — журналируемая файловая система, используемая в современных версиях ОС Windows.

MFT — главная таблица файлов, в которой регистрируются все файлы раздела диска, включая системные файлы; используется в файловой системе NTFS в виде файла $MFT.

make — утилита, предназначенная для автоматизации сборки программ, источником входных данных для которой является специальный текстовый файл с именем makefile или Makefile.

ПРИЛОЖЕНИЕ 1

Порядок создания виртуальной машины Tiny Core Linux.

Процесс установки разберем на примере среды виртуализации VirtualBox-5.0.4-102546-Win и дистрибутива CorePlus-5.3.

Этап 1. Установка виртуальной машины без файловой системы.

1. Скачайте с сайтов производителей дистрибутивы среды виртуализации (http://www.oracle.com/technetwork/server-storage/virtualbox/downloads/index.html) и операционной системы (http://tinycorelinux.net/downloads.html).

2. Запустите файл *VirtualBox-5.0.4-102546-Win.exe* и установите VirtualBox, следуя указаниям установщика.

3. Откройте программу VirtualBox, в появившемся окне нажмите кнопку «Создать».

4. Введите имя виртуальной машины (ВМ), например TinyCore; укажите тип операционной системы Linux, версию Other Linux и нажмите кнопку «Вперед» (рис. П.1.1).

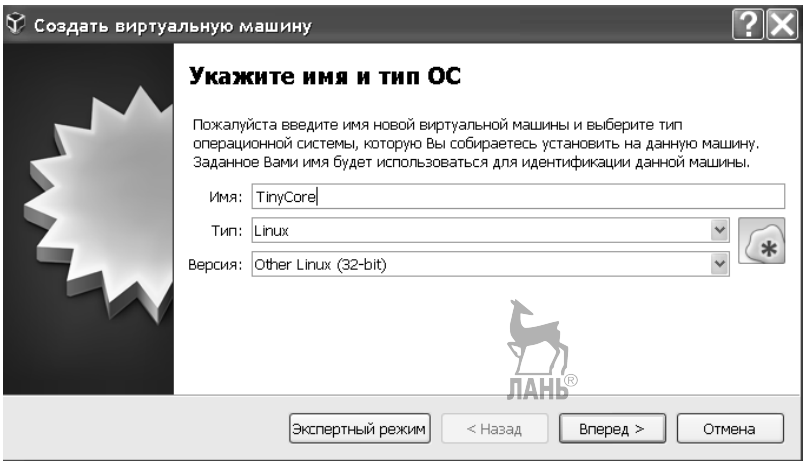


Рис. П.1.1

5. Укажите размер оперативной памяти ВМ и нажмите кнопку *Вперед* (рис. П.1.2).

6. Создайте динамический виртуальный жесткий диск в виде файла типа .VDI размером не менее 512 Мбайт, имя файла может быть любым (например, test.vdi). Нажмите кнопки *Вперед* и *Создать* (рис. П.1.3, П.1.4).

7. В окне VirtualBox нажмите кнопку *Настроить* и в появившемся окне настроек (рис. П.1.5) выберите раздел *Носители*, в списке носителей — пункт *Пусто*, в атрибутах оптического привода установите значение *Первичный слэйв IDE* и, сделав щелчок на значке диска, укажите имя файла с образом CD-диска, в котором находится дистрибутив CorePlus-5.3. После этого в списке носителей появится имя файла образа, например CorePlus-5.3.iso, нажмите ОК и на этом первый этап создания ВМ завершается.

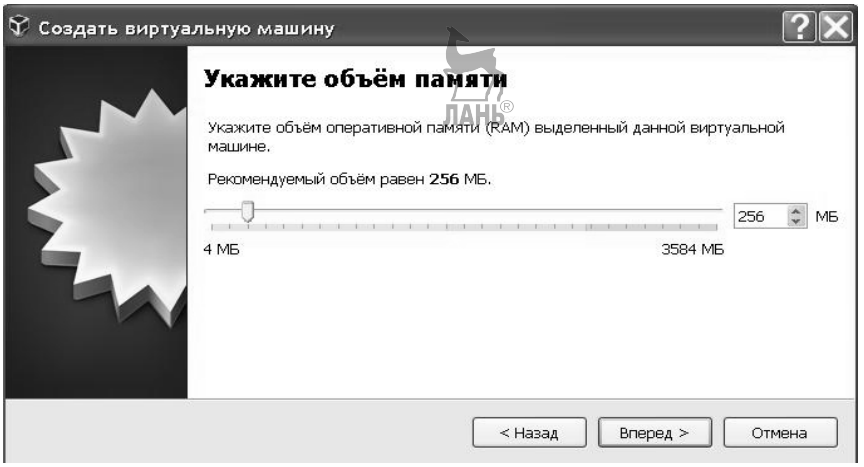


Рис. П.1.2

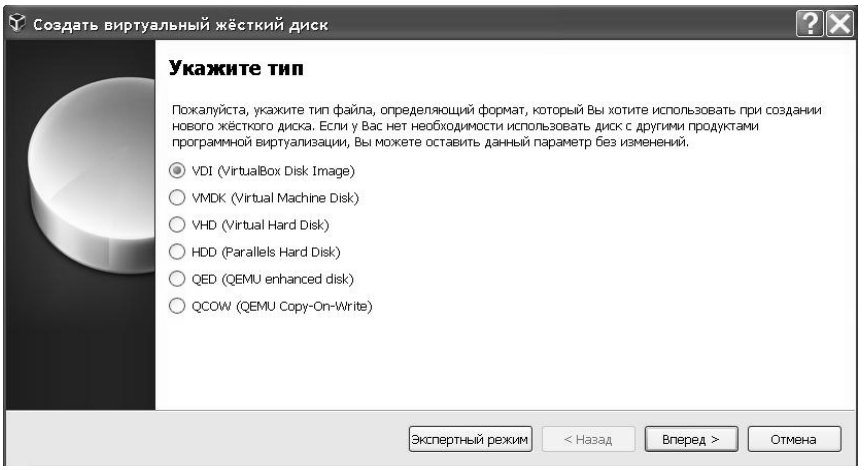


Рис. П.1.3

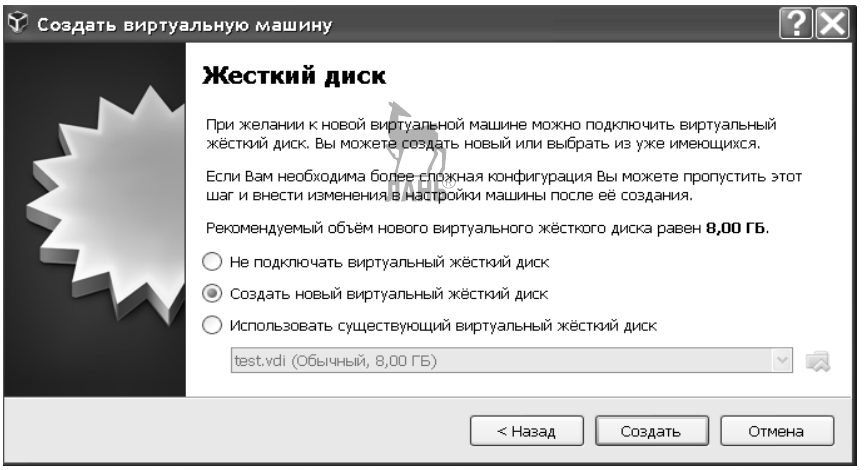


Рис. П.1.4

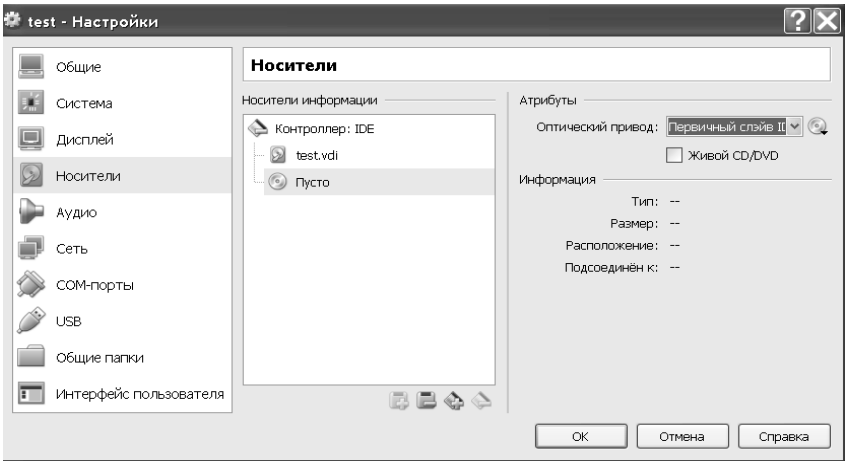


Рис. П.1.5

8. Запустите созданную виртуальную машину, выбрав для загрузки вариант с установками по умолчанию (рис. П.1.6).

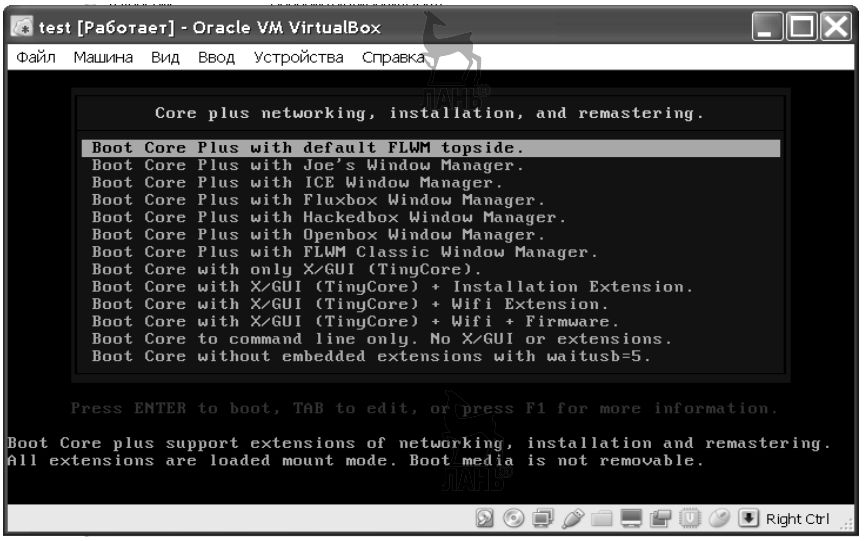


Рис. П.1.6

Этап 2. Установка полного варианта виртуальной машины.

9. Запустите созданную виртуальную машину, выбрав для загрузки вариант «Boot Core with X/GUI (TinyCore) + Installation Extension» (рис. П.1.6).

10. Через иконку TinyCore или через контекстное меню Application/TC_Install запустите установщик *TC_Install*, в окне которого задайте следующие параметры (рис. П.1.7).

11. В следующем окне (рис. П.1.8) выберите тип файловой системы (*ext4*).

12. Выберите тип установки и запустите процесс установки нажатием на кнопку «Proceed» (рис. П.1.9, П.1.10).

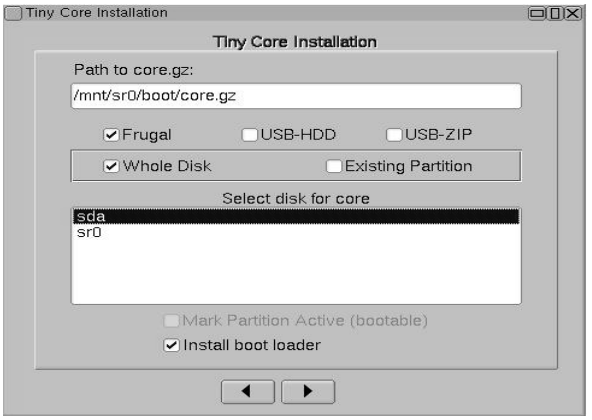


Рис. П.1.7

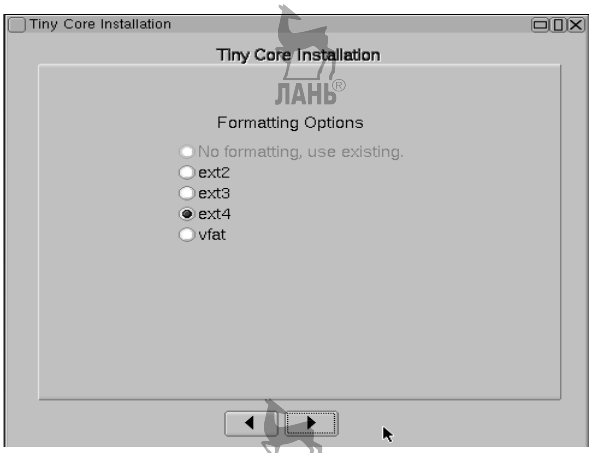


Рис. П.1.8

13. По окончании установки в окне *Настройки* (см. рис. П.1.5) отключите оптический привод, с которого осуществлялась загрузка, и перезагрузите виртуальную машину.

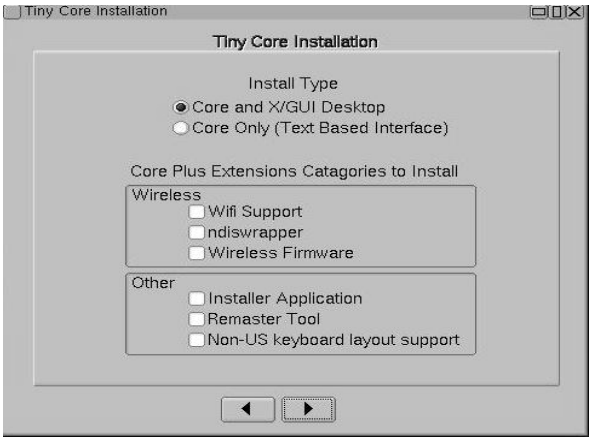


Рис. П.1.9

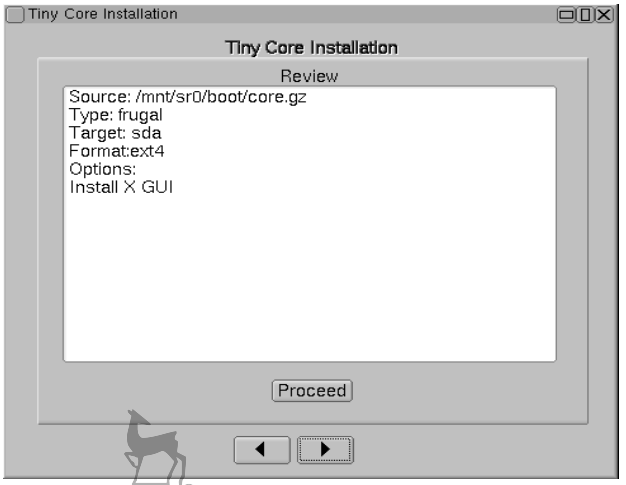

Tiny Core Installation
Tiny Core Installation
Review
Source: /mnt/sr0/boot/core.gz
Type: frugal
Target: sda
Format:ext4
Options:
Install X GUI
Proceed

Рис. П.1.10

ПРИЛОЖЕНИЕ 2

Таблица соответствия некоторых команд Linux и Windows

Таблица П.2.1

Действие	Windows	Linux	Пример команды Linux
Создание каталога	md	mkdir	mkdir abc mykat
Удаление каталога	rd	rmdir	rmdir abc
Вывод имени текущего каталога	cd (без параметров)	pwd	pwd
Изменение текущего каталога	cd	cd	cd /u/home/bpi
Просмотр содержимого каталога	dir	ls	ls mykat
Копирование файла	copy, xcopy	cp	cp myfile1 myfile2
Перемещение файла	move, ren	mv	mv mykat mykat_new
Удаление файла	del	rm	rm my.txt
Получение справки по команде	команда /?	man	man ls
Просмотр содержимого файла	type	cat	cat file1
Постраничный просмотр содержимого файла	more	more	more myfile
Вывод версии ОС	ver	uname	uname -a
Очистка экрана	cls	clear	clear
Поиск заданной строки в файле	find	grep	grep «winter» my.txt
Поиск файла	where	find	find ~ -name «out»
Сортировка данных в файле	sort	sort	sort -nk 2 debts.txt
Вывод списка процессов	tasklist	ps	ps -fu

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

Основной

1. *Дэвис, У.* Операционные системы. Функциональный подход. — М. : Мир, 1980. — 442 с.
2. *Таненбаум, Э.* Современные операционные системы / Э. Таненбаум, Х. Бос. — СПб. : Питер, 2015. — 1120 с.
3. *Баррет, Д. Д.* Linux. Основные команды : карманный справочник. — М. : КУДИЦ-ОБРАЗ, 2007. — 288 с.
4. *Кобылянский, В. Г.* Операционные системы, среды и оболочки : учеб. пособие. — Новосибирск : Изд-во НГТУ, 2018. — 80 с.
5. *Кобылянский, В. Г.* Операционные системы : учеб.-метод. пособие. — Новосибирск : Изд-во СГУПС, 2017. — 70 с.
6. *Колисниченко, Д. Н.* Linux. Полное руководство / Д. Н. Колисниченко, П. В. Аллен. — СПб. : Наука и техника, 2007. — 777 с.
7. *Тай, Т.* Платформа .NET. Основы / Т. Тай, Хонг К. Лэм. — 2-е изд. — СПб. : Символ-Плюс, 2003.
8. *Сулейманова, А. М.* Системы реального времени : учеб. пособие. — Уфа : Изд-во УГАТУ, 2004. — 292 с.
9. *Кобылянский, В. Г.* Системы реального времени : учеб. пособие. — Новосибирск : Изд-во НГТУ, 2015. — 84 с.
10. *Зыль, С. Н.* QNX Momentics: основы применения. — СПб. : БХВ-Петербург, 2005. — 256 с.

Дополнительный

11. Справочник по командам ОС Linux. ИДСТУ СО РАН [Электронный ресурс]. — Режим доступа: http://hpc.icc.ru/documentation/cmnds.pdf.
12. *Колисниченко, Д. Н.* Процессы в Linux [Электронный ресурс]. — Режим доступа: http://www.opennet.ru/docs/RUS/lnx_process/.
13. Linux по-русски. Виртуальная энциклопедия. — [Электронный ресурс]. — Режим доступа: http://rus-linux.net/MyLDP/consol/hdrguide/rusman/ps.htm.
14. Linux по-русски. Виртуальная энциклопедия [Электронный ресурс]. — Режим доступа: http://rus-linux.net/MyLDP/consol/komanda-top-v-linux.html.
15. *Котельников, Е. В.* Введение во внутреннее устройство Windows [Электронный ресурс]. — Режим доступа: https://www.intuit.ru/studies/courses/10471/1078/lecture/16586.
16. *Баранов, А.* Записки исследователя NTFS [Электронный ресурс]. — Режим доступа: http://citforum.ru/operating_systems/windows/ntfs/.
17. *Чирков, М.* Описание редактора связей GNU ld [Электронный ресурс]. — Режим доступа: https://www.opennet.ru/docs/RUS/gnu_ld/gnuld.html#toc1.

18. *Игнатов, В.* Эффективное использование GNU Make [Электронный ресурс]. — Режим доступа: http://citforum.ru/operating_systems/gnumake/gnumake_04.shtml.

19. CVS — система поддержки версий текстов [Электронный ресурс]. — Режим доступа: http://dbserv.pnpi.spb.ru/~shevel/Book/node110.html.

20. Русская документация по CVS [Электронный ресурс]. — Режим доступа: http://www.opennet.ru/docs/RUS/cvsrdp/.

21. *Долозов, Н. Л.* Основы операционных систем и сетевых технологий : учеб.-метод. пособие. — Новосибирск : Изд-во НГТУ, 2008. — 144 с.

22. *Лаврищева, Е. М.* Программная инженерия. Тема 2. Технология программирования : учеб.-метод. пособие. — М. : МФТИ, 2016. — 52 с.

23. Microsoft Developer Network Library [Электронный ресурс]. — Режим доступа: https://msdn.microsoft.com/ru-ru/library.

24. *Кёртен, Р.* Введение в QNX Neutrino 2. — СПб. : Петрополис, 2001. — 480 с.

25. *Зыль, С. Н.* Операционная система реального времени QNX: от теории к практике. — СПб. : БХВ-Петербург, 2004. — 192 с.

26. *Кобылянский, В. Г.* Системы реального времени. Операционная система QNX : метод. указания к выполнению лабораторных работ / Новосиб. гос. техн. ун-т. — 2-е изд. — Новосибирск : НГТУ, 2017. — 66 с.

27. Настройка атрибутов сборки [Электронный ресурс]. — Режим доступа: https://docs.microsoft.com/ru-ru/dotnet/framework/app-domains/set-assembly-attributes.

28. *Кобылянский, В. Г.* Операционные системы : учеб.-метод. пособие. — Новосибирск : Изд-во СГУПС, 2017. — 71 с.

ОГЛАВЛЕНИЕ

Валерий Григорьевич КОБЫЛЯНСКИЙ

ОПЕРАЦИОННЫЕ СИСТЕМЫ, СРЕДЫ И ОБОЛОЧКИ

Учебное пособие

Издание второе, стереотипное

Зав. редакцией
литературы по информационным технологиям
и системам связи *О. Е. Гайнутдинова*

ЛР № 065466 от 21.10.97
Гигиенический сертификат 78.01.10.953.П.1028
от 14.04.2016 г., выдан ЦГСЭН в СПб

Издательство «ЛАНЬ»
lan@lanbook.ru; www.lanbook.com
196105, Санкт-Петербург, пр. Юрия Гагарина, д. 1, лит. А
Тел./факс: (812) 336-25-09, 412-92-72
Бесплатный звонок по России: 8-800-700-40-71

Подписано в печать 30.04.21.
Бумага офсетная. Гарнитура Школьная. Формат 70×100 $^{1}/_{16}$.
Печать офсетная. Усл. п. л. 9,75. Тираж 30 экз.

Заказ № 548-21.

Отпечатано в полном соответствии с качеством
предоставленного оригинал-макета в АО «Т8 Издательские Технологии».
109316, г. Москва, Волгоградский пр., д. 42, к. 5.