Операционные системы. Учебный курс. Теоретическая часть.

Кудряшов И. Г.

2002

Оглавление

Введение.

Настоящее пособие представляет собой курс лекций по предмету «Операционные системы», рассчитанный на 18 академических часов. Курс ориентирован на лиц, имеющих практические навыки работы с персональным компьютером и операционными системами Windows 95 или Windows NT. Для практического применения материала последней главы пособия необходимо иметь навыки программирования на любом языке для операционных систем, поддерживающих Win32 API. В данном пособии предполагается, что учащийся владеет навыками программирования в системе Borland Delphi версии не ниже 2.

Глава 1

Основные понятия и определения.

1.1 Операционные системы и их классификация

Под операционной системой (далее будет часто применяться традиционное сокращение ОС) понимается программный комплекс, который осуществляет управление функционированием компьютера и обеспечивает взаимодействие устройств, входящих в его состав, с пользователями и прикладными программами. Основная функция ОС — управление ресурсами компьютера, распределение их между устройствами, программами и пользователями. Кроме того, ОС обеспечивает взаимодействие прикладных программ с оборудованием и взаимодействие пользователя с компьютером, его устройствами и программами.

По сути, ОС представляет собой некий «промежуточный слой» между пользователями, прикладными программами и оборудованием компьютера. Сама по себе ОС не решает никаких прикладных задач, но обеспечивает пользователю возможность запускать прикладные программы, решающие его задачи, а прикладным программам — возможность удобного и стандартизованного взаимодействия с оборудованием и с пользователем.

С точки зрения пользователя, ОС — это просто рабочая среда, с которой он взаимодействует при работе с компьютером. Собственно, для пользователя ОС — это и есть компьютер, взаимодействие с ОС воспринимается как «работа с компьютером».

Существует множество типов ОС с различными техническими характеристиками, предназначенные для различных областей использования и для различных компьютеров. Вряд ли есть смысл приводить полную, абсолютно исчерпывающую классификацию, которая бы описывала все типы ОС и все их особенности. Поэтому мы рассмотрим только некоторые общепринятые варианты классификации, чтобы получить представление о терминологии и наиболее важных особенностях ОС.

Операционные системы подразделяются на однозадачные и многозадачные, в зависимости от того, может или не может ОС поддерживать одновременное выполнение нескольких запущенных программ. Однозадачная ОС может, в соответствии с

названием, исполнять одновременно только одну программу. Если необходимо выполнить несколько программ, то это можно сделать только последовательно, запуская каждую следующую программу 1после завершения предыдущей. Каждая запущенная программа в однозадачной ОС может иметь полный доступ ко всем системным ресурсам (процессору, памяти, внешним устройствам), возможно, за исключением тех из них, которые использует для своего функционирования сама ОС.

В многозадачной ОС могут одновременно выполняться несколько программ. То есть, если в системе уже запущена одна программа, то нет необходимости дожидаться её завершения, чтобы запустить следующую — можно просто запустить новую программу, и она будет выполняться одновременно с уже работающей. Разумеется, если компьютер физически имеет только один процессор, в каждый конкретный момент времени процессор исполняет только какую-то одну из запущенных программ. В этом случае ОС обеспечивает постоянное переключение процессора между выполняющимися программами, что и создаёт эффект одновременного выполнения. В многозадачной системе несколько одновременно выполняющихся программ вынуждены тем или иным способом разделять между собой системные ресурсы, и ОС должна содержать для такого разделения соответствующие средства.

Как правило, однозадачные системы предоставляют больше возможностей для обеспечения максимальной эффективности программ. Это преимущество обусловлено отсутствием средств разделения системных ресурсов, которые сами по себе эти ресурсы поглощают. Тем не менее, сейчас в большинстве случаев предпочитают использовать многозадачные системы, поскольку они обеспечивают большее удобство работы.

Операционная система может быть однопользовательской или многопользовательской, в зависимости от того, предназначена она для управления работой компьютера, которым пользуется только один человек, или для организации работы (одновременной или последовательной) с одним компьютером нескольких пользователей. Однопользовательские ОС используются исключительно на персональных компьютерах и во встроенных системах.

Системы этих двух видов различаются уровнем развитости средств обеспечения безопасности и защиты данных. Эти средства дают возможность распределять права пользователей на ресурсы компьютера, защищать данные, принадлежащие одному пользователю, от несанкционированного их использования другими пользователями, защищать саму ОС от некорректных действий пользователей и запущенных ими программ. Однопользовательские ОС, как правило, имеют очень примитивные средства обеспечения безопасности. В них единственный пользователь является полновластным хозяином компьютера, поэтому нет смысла защищать систему от его действий. А поскольку нет других пользователей, данные которых требовалось бы защищать, нет и механизмов защиты данных. Единственное, для чего используются средства защиты в однопользовательских ОС — это для защиты компьютера от несанкционированного проникновения извне (например, из компьютерной сети). В многопользовательских

ОС, напротив, средства защиты обычно достаточно развиты. Они необходимы для поддержания в рабочем состоянии системы, с которой работает много людей.

Как было сказано, многопользовательская ОС может обеспечивать либо только последовательную, либо последовательную и одновременную работу нескольких пользователей. Последовательная работа предполагает, что в каждый конкретный момент времени с системой работает только один человек. При параллельной работе системой могут одновременно пользоваться несколько пользователей. Для обеспечения параллельной работы нескольких пользователей ОС обязательно должна быть многозадачной, кроме того, она должна поддерживать работу технических средств, через которые с ней могут одновременно работать несколько человек. Раньше такие системы работали на вычислительных комплексах, содержащих один центральный компьютер и несколько (иногда очень много) специализированных терминалов. При этом терминалы не являлись самостоятельными вычислительными устройствами. Каждый из них содержал только устройство ввода (клавиатуру), устройство вывода (ЭЛТ-дисплей или АЦПУ) и схемы, необходимые для соединения с центральным компьютером.

Сейчас, чаще всего, работа с многопользовательскими системами осуществляется через обычные персональные компьютеры, объединённые сетью с центральным компьютером, на котором и работает ОС. В этом случае пользователи могут одни задачи решать непосредственно на своём персональном компьютере, который может использовать и однопользовательскую ОС, а другие задачи — на общем компьютере (сервере приложений) с многопользовательской ОС. Более того, иногда одна часть программы, решающей задачи пользователя, работает на персональном компьютере, а другая — на сервере (широко используемая сейчас схема «клиент–сервер»). В последнее время интерес к многотерминальным системам снова возрос. Терминалы достаточно просты и гораздо в меньшей степени, чем полномасштабные компьютеры,подвержены моральному старению.

Операционная система может являться либо не являться ОС реального времени. Компьютерная система (не операционная система, а полная компьютерная система, включающая в себя оборудование, ОС и прикладное программное обеспечение) называется системой реального времени, если для такой системы можно гарантировать некоторое наперёд известное максимально возможное время реакции системы на поступление данных. Системы реального времени используются для решения задач, для которых существенным является фактор времени выполнения операций. К таким задачам относятся задачи управления устройствами и агрегатами, например, технологическим оборудованием на предприятии. В них необходимо гарантировать, что система сможет выполнить ту или иную последовательность операций за время, не превышающее некоторого заданного предела. Выход за предельное время приводит к аварии, так что его необходимо полностью исключить. С точки зрения теории автоматического управления, системой реального времени можно называть такую управляющую систему, время отклика которой на поступление данных не превышает времени протекания переходных процессов в управляемом объекте. ОС реального времени (ОСРВ)

используются в компьютерных системах реального времени. Их характеристической особенностью является гарантированное время исполнения любой операции, благодаря чему возможно исполнение программ в заданном временном темпе. При управлении физическими объектами, когда ограничения по времени срабатывания могут быть довольно жёсткими, ОСРВ незаменимы[1].

Необходимо понимать, что ОСРВ не обязательно обеспечивает *малое* время выполнения тех или иных операций. Дело не в конкретном размере промежутка времени, а в том, что максимально возможный размер этого промежутка *гаранитрован*. ОСРВ может быть (и обычно бывает) в среднем медленнее, чем могла бы быть при прочих равных условиях обычная ОС общего назначения, но ОСРВ *гарантирует выполнение операций в строго ограниченное время*.

Операционные системы бывают специализированными и универсальными. Специализированные ОС разрабатываются для использования в конкретных узких областях применения. Такие системы, как правило, исключительно эффективны в своей области, но совершенно неприменимы где бы то ни было ещё. Примерами специализированных ОС могут быть ОС встроенных систем (торговых и кассовых автоматов, систем управления крылатыми ракетами, банковских смарт-карт, станков с ЧПУ и так далее).

Универсальные ОС предназначаются для универсальных вычислительных комплексов, таких, как домашний или офисный компьютер, сервер, обслуживающий информационные нужны предприятия или организации. Универсальные ОС разрабатываются в расчёте на максимально широкий круг наиболее типичных задач, которые может потребоваться решать в управляемой вычислительной системе. Соответственно, возможности, предоставляемые такими ОС программам и пользователям, наиболее разнообразны. Неизбежной платой за универсальность является меньшая эффективность. Универсальная ОС, как правило, объёмнее, требует больше вычислительных ресурсов, медленнее работает и хуже поддерживает выполнение прикладных программ, чем специализированная ОС, разработанная исключительно для решения тех задач, которые в данном случае требуется решать.

[1]На самом деле, сейчас ОС общего назначения зачастую используются в тех областях, где следует использовать только ОСРВ. Скажем, во многих технологических установках программное обеспечение работает под управлением ОС Microsoft Windows NT, которая ОСРВ не является. Для того, чтобы снизить вероятность недопустимо больших задержек в работе, разработчики таких систем вынуждены проектировать аппаратную часть с учётом возможности задержек и устанавливать значительно (иногда в несколько раз) более быстродействующие компьютеры. Но даже такие действия, строго говоря, не гарантируют от сбоев в работе систем. Эта практика, безусловно, порочна, но достаточно распространена.

1.2 Структура ОС.

Несмотря на различные подходы к проектированию, различное назначение и различные компьютеры, для которых разрабатываются операционные системы, в любой ОС можно выделить четыре основных компонента. Это ядро системы и три подсистемы, обеспечивающие взаимодействие ядра с внешними устройствами, прикладными программами и пользователем соответственно.

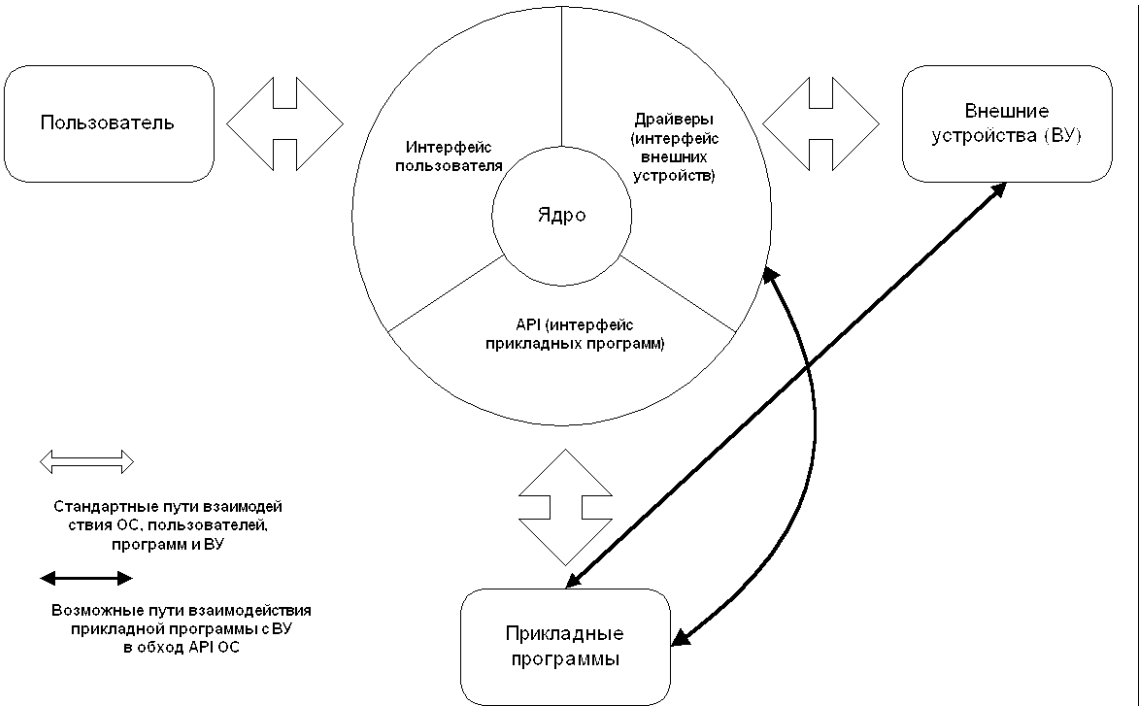

Рис. 1.1: Структура ОС.

На рисунке 1.1 схематически показаны основные компоненты ОС и порядок их взаимодействия друг с другом, пользователями, внешними устройствами и прикладными программами.

Следует понимать, что в реальных ОС описанные компоненты обычно не выделяются явно, скажем, в виде отдельных программных модулей. Просто существуют фрагменты системы, выполняющие функции каждого из компонентов. Так что описанное разделение ОС на подсистемы следует воспринимать как логическую схему, а не как техническое описание.

Рассмотрим более подробно каждый из компонентов.

1.2.1 Ядро.

Центральным элементом ОС является ядро, выполняющее функции главного управляющего модуля. Ядро выполняет функции разделения процессорного времени, управле-

ния памятью, оборудованием (на уровне логики), файловой системой, правами пользователей и приложений, защиты от несанкционированного доступа, а также ряд других. Физически ядро представляет собой один или несколько программных модулей, которые загружаются при старте ОС и работают всё время, пока компьютер не будет выключен.

Архитектура ядра является одним из основных моментов, определяющих особенности функционирования ОС, её надёжность, скорость работы программ, затраты системных ресурсов. Каждая операционная система имеет собственное ядро, со своей архитектурой и своими особенностями. Тем не менее, можно выделить два основных типа архитектуры ядер: монолитное ядро и микроядро.

Монолитное ядро ОС представляет собой единый модуль, который выполняет и управляющие функции, и ввод-вывод, и работу с оборудованием. При использовании микроядерной архитектуры собственно ядро занято только управлением запущенными в системе программами. Все остальные функции ядра выполняются модулями, работающими отдельно и выполняющимися в тех же условиях, что и обычные пользовательские задачи. У каждого из видов архитектуры ядра есть свои положительные и отрицательные стороны. Монолитное ядро обеспечивает несколько большую скорость выполнения операций ядра. С другой стороны, в моменты, когда выполняются операции ядра, все остальные программы, находящиеся в системе, вынуждены простаивать. Это повышает время отклика системы на запросы програм. Микроядро, в силу меньших размеров и меньшего количества выполняемых функций, отнимает меньше процессорного времени и прочих системных ресурсов, что снижает время отклика. В то же время функции ядра, выполняемые отдельными модулями, не могут выполняться так же быстро, как в системе с монолитным ядром, так что средняя производительность системы может оказаться ниже.

Микроядерная архитектура обычно выбирается при разработке операционных систем реального времени и операционных систем, в которых предполагается активная работа пользователей в интерактивном режиме, поскольку в таких ОС время отклика — более важный параметр, чем средняя производительность.

Помимо чисто монолитных и чисто миркроядерных ОС существуют и промежуточные варианты — ОС, ядро которых не является микроядром в строгом смысле этого слова, но часть функций которого вынесена в отдельно выполняющиеся непривилегированные модули.

Для более глубокого понимания различий функционирования ядер различной архитектуры необходимо изучить работу системы разделения времени. В разделе, посвящённом этой теме, мы ещё раз вернёмся к вопросу архитектуры ядра ОС.

1.2.2 Интерфейс пользователя.

Ясно, что в ходе взаимодействия с ОС человек должен иметь возможность отдавать команды системе, а система должна иметь возможность передавать пользователю данные и сигнализировать о происходящих в системе событиях. Именно эту задачу решает интерфейс пользователя.

Внешне (с точки зрения пользователя) интерфейс пользователя может выглядеть как угодно (зачастую даже в пределах одной и той же ОС есть возможность использовать несколько вариантов интерфейса пользователя, разительно отличающихся по внешнему виду), но его содержание всегда одно и то же — пользователь отдаёт команды и получает информацию о реакции системы на них. Можно выделить два основных варианта организации интерфейса пользователя — визуальный интерфейс и интерфейс, основанный на текстовых командах.

Командный интерфейс.

Этот вид интерфейса пользователя исторически появился первым, потому что первыми устройствами ввода и вывода данных ЭВМ были клавиатура и АЦПУ. Это естественным образом привело к организации диалога пользователей с системой посредством обмена фрагментами текста. Сущность командного интерфейса очень проста. ОС имеет некоторый командный язык — набор команд, каждая из которых записывается в виде слова или фразы и обозначает какое-то требуемое от ОС действие. Реакция системы на команду представляет собой некоторое текстовое сообщение. Пользователь вводит команды с клавиатуры и наблюдает реакцию системы по сообщениям на терминале или ином устройстве текстового вывода.

Обычно команда представляет собой слово [2], значение которого в некоторой мере соответствует смыслу команды. Кроме имени команды, пользователь должен ввести параметры, если они необходимы, и указать режимы работы команды. Типичная команда ОС выглядит обычно следующим образом:

```
<команда> -<ключ1> -<ключ2>... <параметр1> <параметр2>...
```

Здесь `<команда>` — слово, обозначающее команду ОС, `<параметр>` — слово, сообщающее команде некоторую переменную информацию, необходимую для выполнения, `<ключ>` — слово, обозначающее один из предварительно заданных режимов работы команды. Символ «-» (минус), предваряющий ключи, используется как указатель на то, что следующее за ним слово является не параметром, а ключом. Помимо минуса для этой цели в некоторых системах используется символ «/» (слэш) или два минуса подряд. Правила написания ключей и параметров команды зависят от принятых в ОС соглашений и от самой команды. Как правило, при создании командного языка стараются придерживаться некоторой общей системы, но, несмотря на это, одни и те же ключи в разных командах могут иметь совершенно различный смысл.

Следующий пример показывает типичную команду командного языка операционной системы MS-DOS:

```
dir *.txt /p /o:n
```

[2] Это может быть правильное слово естественного языка, сокращение, а в некоторых случаях словосочетание.

Эта команда требует от ОС вывести на стандартное устройство отображения (обычно терминал) содержимое текущего каталога (об этом говорит имя команды — `dir`). При этом должны быть выведены имена только тех файлов, имя которых подходит под маску «*.txt» (параметр «`*.txt`»), то есть содержащие любую последовательность символов, которая заканчивается на «.txt». Необходимо делать паузу после вывода каждого полного экрана текста (этот режим задаёт ключ «`/p`»). При выводе необходимо сортировать файлы по именам (ключ «`/o:n`»). Здесь, как очевидно из описания, «`dir`» – название команды, «`*.txt`» – параметр, а «`/p`» и «`/o:n`» — ключи.

Следующий пример показывает ту же по смыслу команду, но на языке командного интерпретатора «shell», используемого в ОС UNIX.

```
ls -l *.txt | sort | more
```

Эта команда требует последовательно выполнить три операции. Здесь фактически вызываются три команды. Сначала выполняется команда `ls`, выводящая содержимое текущего каталога. У неё задан один ключ — `-l`, который требует выводить список файлов в один столбец и со всеми атрибутами: именем, размером, правами и так далее, и один параметр — маска имени файлов, которые нужно выводить. Результаты исполнения команды `ls` предаются команде `sort` (передача выходных данных одной программы в другую обозначается вертикальной чертой). Команда `sort` сортирует строки, подающиеся ей на вход, в алфавитном порядке. Результат сортировки передаётся команде `more`, которая обеспечивает вывод текста поэкранно. После вывода каждого экрана делается задержка до нажатия пользователем любой клавиши на клавиатуре.

Достоинством командного интерфейса является возможность практически неограниченно наращивать командный язык, добавляя новые команды и практически не увеличивая при этом видимую сложность системы. Кроме того, ОС всегда предоставляет возможность написания программ на командном языке системы (обычно их называют скриптами или пакетными файлами). Как правило, в командном языке есть конструкции, позволяющие организовывать ветвления и циклы, поэтому пакетный файл может быть полноценной программой. С помощью таких программ можно сконструировать собственные команды и заставить систему по одной команде пользователя самостоятельно выполнять сложные последовательности программ и системных команд, зависящие от указанных входных данных.

Недостатком интерфейса текстовых команд является то, что командный язык обычно достаточно сложен для пользователя — непрограммиста и его изучение требует определённых затрат времени и усилий. Впрочем, эти трудности не стоит переоценивать. Практика показывает, что большинство непрофессионалов (правда, действительно не все) обучаются работе с командным интерфейсом достаточно быстро.

Визуальный интерфейс.

Визуальный интерфейс основан на представлении состояния системы в виде некоторого набора визуальных образов на устройстве отображения (терминале). Каждый из компонентов системы, с которым пользователь может работать, отображается в виде значка или надписи, расположенных в определённом месте экрана. Особенности отображения этого значка или надписи могут служить индикатором состояния данного компонента системы. Например, принтер может отображаться в виде иконки со стилизованным изображением принтера, которая при печати может изменяться, а при невозможности печати (отсутствии бумаги, неисправности принтера) перекрываться или заменяться каким-либо значком — символом неисправности.

Пользователь отдаёт команды ОС путём манипулирования предоставленным ему набором образов с помощью манипуляторов (мышь, трекбол) и клавиатуры. Например, каталог на диске часто отображается в виде экранного окна, в котором файлы изображены как иконки либо надписи. Предположим, что нужно скопировать файл «xxx.txt» из каталога «/home/user/work» в каталог «/home/user/archive». Для этого пользователь, вместо того, чтобы писать команду вида:

```
cp /home/user/work/xxx.txt /home/user/archive/
```

может просто открыть на экране два окна, одно из которых соответствует каталогу «/home/user/work», другое — каталогу «/home/user/archive», и мышью перетащить надпись «xxx.txt» из первого окна во второе. Для пользователя — непрофессионала такое действие гораздо проще и естественнее, чем написание малопонятной команды.

Визуальный интерфейс чаще всего строится в виде системы экранных меню и диалоговых окон, где пользователь выбирает ту или иную команду путём выбора пункта меню или нажатия на экранную кнопку, а параметры команды задаёт, манипулируя визуальными элементами и вводя параметры в диалоговых окнах. Реакция системы на команду может состоять в изменении наблюдаемой пользователем системы образов или в выводе сообщения. Результаты обработки выводятся в экранных формах разнообразного вида. Такого типа интерфейс сейчас является фактическим стандартом для большинства массовых прикладных программ.

Важно понимать, что тип интерфейса определяется сущностью способа взаимодействия пользователя с системой, а не используемыми изобразительными средствами. Вовсе не обязательно визуальный интерфейс использует устройство отображения, работающее в графическом режиме. Визуальный интерфейс может быть как текстовым, при котором визуальные образы на экране формируются с помощью обычных текстовых символов, возможно, дополненных псевдографикой (типа интерфейса оболочек Midnight Commander, FAR, Norton Commander или DOS Navigator), так и графическим, при котором визуальные образы рисуются на экране в графическом режиме и могут иметь практически неограниченное разнообразие. Но на суть интерфейса это не оказывает решающего влияния, хотя, безусловно, графический интерфейс внешне обычно выглядит намного привлекательнее.

Достоинством визуального интерфейса является его понятность и минимальные затраты времени на изучение. Визуальное представление разрабатывается так, чтобы манипуляции с ним были естественны, интуитивно понятны и не требовали специального запоминания того, как сделать то или иное действие.

К недостаткам его относится то, что такого рода системы обычно менее удобно дополнять новыми командами, они склонны к разрастанию и, как следствие, к утрате простоты и понятности. Так, если систему из ста текстовых команд дополнить ещё десятью командами, то для пользователя ничего не изменится. Если он изучит эти десять команд, то сможет ими пользоваться, а если не изучит, то будет пользоваться остальными и от наличия новых команд не испытает никаких неудобств. Добавление же новых команд в уже существующий визуальный интерфейс означает появление в системе новых визуальных образов, новых пунктов в меню, новых кнопок в диалоговых окнах, появление новых меню и окон. В результате пользователь, который не хочет пользоваться новыми возможностями, всё равно будет видеть средства их использования в меню и окнах, что неизбежно ухудшит ориентировку в системе и может стать причиной случайного выбора не тех команд. Частично этот недостаток может быть устранён организацией достаточно полной системы настройки визуального интерфейса, с помощью которой пользователь сможет самостоятельно организовать этот интерфейс так, чтобы видеть в нём только используемые им управляющие элементы.

Кроме того, с помощью чисто визуального интерфейса обычно невозможно задать системе последовательность команд для выполнения, если только при разработке системы подобная возможность не была предусмотрена специально. То есть, в визуальном интерфейсе нет полного аналога программ на командном языке, о которых упоминалось выше. Частичным аналогом таких программ являются системы записи макрокоманд, которые позволяют записать последовательность воздействий пользователя на визуальные элементы управления (так называемый «макрос») и впоследствии повторить эту последовательность в автоматическом режиме. Но такие макросы намного менее гибки, чем пакетные файлы, поскольку они жёстко заданы и их, как правило, невозможно выполнять в различных режимах, в то время как исполнение пакетных файлов может происходить всякий раз по-разному, в зависимости от того, с какими параметрами выполнен каждый конкретный запуск.

Все современные ОС имеют командный интерфейс пользователя. При этом большинство имеет также либо встроенные, либо дополнительные подсистемы, позволяющие организовать визуальный интерфейс. Во многих системах визуальный интерфейс настолько развит, что пользователь может работать только с ним, даже не догадываясь о наличии системы команд. Однако практически всегда имеется ряд специфических операций, которые можно выполнить только через командный интерфейс.

Иногда в визуальный интерфейс намеренно не включается возможность запуска некоторых команд системы. Это могут быть команды, неправильное использование которых может привести к нежелательным последствиям – потере данных, приведению программ в неработоспособное состояние, даже разрушению системы. Неопытный пользователь, как правило, работает только с визуальным интерфейсом и не может случайно наткнуться на команды, необдуманное выполнение которых может приве-

сти к нежелательным последствиям. Опытный пользователь, желающий выполнить небезопасную операцию, сможет воспользоваться и командным языком системы.

1.2.3 Интерфейс прикладных программ.

Интерфейс прикладных программ (API — Application Programming Interface) – часть ОС, через которую происходит взаимодействие ОС с приложениями, решающими задачи пользователя. API — это набор средств (обычно – процедур и функций), входящих в состав ОС, который может использоваться любой прикладной программой для обеспечения взаимодействия её с ядром ОС, внешними устройствами и пользователем [3]. Обычно в API входят средства, обеспечивающие выделение ресурсов, работу с файловой системой, взаимодействие с пользователем и управление работой самой ОС. Физически API обычно представляет собой одну или несколько общедоступных библиотек процедур, каждая из которых является частью ОС и может быть вызвана любой программой, выполняющейся под управлением этой ОС. Конкретный механизм вызова процедур API зависит от типа ОС.

Использование API избавляет разработчика прикладных программ от необходимости программировать ряд рутинных операций, таких как чтение файлов, ввод с клавиатуры, запуск программ и многое другое. Теоретически, использование программой для работы с системой, пользователями и внешними устройствами исключительно API ОС обеспечивает программе полную независимость от оборудования, которое физически установлено на компьютере и от особенностей реализации ОС. Программа, работающая только через API, будет (после перекомпиляции в код нужного процессора, естественно) безо всяких изменений работать на компьютерах другого типа, нежели тот, на котором она написана, если на этих компьютерах работает ОС, для которой написана программа. Таким образом, использование API обеспечивает лучшую переносимость программы.

Поскольку API, как правило, разрабатывается как универсальная библиотека, которая, по определению, должна подходить для практически всех применений, её процедуры обычно неоптимальны для решения каждой конкретной задачи. Поэтому часть программ в некоторых ОС работает с оборудованием ЭВМ напрямую, минуя ОС и API, что может значительно увеличить производительность, но создаёт проблемы с работой этих программ на некоторых конкретных компьютерах.

1.2.4 Интерфейс внешних устройств. Драйверы.

Внешние устройства (ВУ) компьютера очень разнообразны. Для того чтобы прикладные программы могли работать с ними, программы, теоретически, должны содержать

[3] Вообще, под API понимается программный интерфейс любой программы или устройства, а не только ОС. API может быть (и обычно есть) у любого устройства и у многих программ (API платы видеоускорителя, API программы пересылки почты через Интернет...)

модули, обеспечивающие взаимодействие с каждым типом ВУ, который может встретиться. В какой-то мере эта задача решается путём стандартизации самих ВУ, что делает возможной работу со многими устройствами с помощью одного и того же программного кода. Так, например, для вывода текста на большинство матричных принтеров можно воспользоваться системой команд принтера Epson, поскольку эта система команд поддерживается большинством аналогичных принтеров других производителей.

Однако постоянно появляются новые ВУ, которые реализуют принципиально новые возможности или делают возможным выполнение ранее поддерживаемых операций более простыми и эффективными методами. Ограничение командного языка ВУ только общим подмножеством языков всех ВУ такого типа ограничит использование специфических возможностей каждого конкретного устройства.

В связи с этим наиболее приемлемым решением проблемы взаимодействия программ и ВУ является перекладывание функции непосредственного взаимодействия с ВУ на ОС. ОС должна содержать модули, умеющие работать с каждым конкретным ВУ, а программы должны работать с ВУ через API ОС. Интерфейс внешних устройств реализуется в ОС с помощью специального вида программ, называемых драйверами. Драйвер — это программа, которая обеспечивает взаимодействие между ВУ определённого типа и операционной системой.

ОС имеет жёстко заданный стандарт на перечень функций, которые должно выполнять устройство того или иного типа и на порядок взаимодействия с таким ВУ (то есть обобщённый интерфейс ВУ данного типа). Фактически, этот интерфейс описывает некоторое виртуальное устройство, с которым и работает ядро системы.

Драйвер пишется так, что та его часть, с которой взаимодействует ядро ОС, полностью соответствует заданному интерфейсу виртуального ВУ. Драйвер принимает от ОС команды в стандартизованном виде, перекодирует их в команды конкретного типа ВУ и передаёт в ОС данные от ВУ также в стандартном формате.

ОС может работать с любым ВУ, для которого имеется драйвер, реализующий стандартный протокол взаимодействия. Ядро ОС, а значит и прикладные программы, использующие API, работают с ВУ через драйвер. В результате они могут общаться с ВУ, не зная ничего о его конкретном типе, наборе команд и прочих особенностях.

Более того, некоторые функции, поддерживаемые драйвером, устройство может вообще не реализовывать аппаратно. Если, в соответствии с протоколом, в данной ОС устройство данного типа обязано реализовывать данную функцию, то эта функция может быть реализована не самим устройством, а его драйвером. Впрочем, и такое поведение драйвера вовсе не обязательно. Часто бывает, что ряд функций ВУ считается необязательным. Для таких функций предусматриваются средства проверки, с помощью которых обращающаяся к драйверу программа (или ядро ОС) могут определить, поддерживается ли определённая функция. Если необходимая функция поддерживается, она используется, а если нет, то программа должна самостоятельно решить эту проблему.

Так, например, если видеоадаптер компьютера не поддерживает возможности рисования ломаных линий, но ОС требует, чтобы такая возможность была, драйвер

может перекодировать запрос на рисование ломаной линии в набор из нескольких запросов на рисование отрезков и именно этот набор передать устройству. Естественно, такая реализация даст худшую производительность, чем на устройстве, аппаратно поддерживающем рисование ломаных, однако, функция всё-таки будет реализована, и устройство можно будет использовать. С другой стороны, драйвер может просто отказаться рисовать ломаные. Если функция вывода ломаных не является обязательной, система, прежде чем её использовать, должна проверить, поддерживается ли она используемым устройством, и если нет, то выполнить рисование ломаной отрезками самостоятельно.

1.2.5 Взаимодействие прикладных программ с ВУ.

Прикладная программа может взаимодействовать с ВУ либо через API операционной системы, либо через драйвер ВУ, либо непосредственно (см. схему). Как уже говорилось, работа через API операционной системы — это наиболее предпочтительный способ с точки зрения обеспечения независимости программы от оборудования. Однако, его использование приводит к дополнительным затратам ресурсов за счёт того, что взаимодействие осуществляется по длинной цепочке "программа — API — драйвер — ВУ". При этом программа, обращаясь к API, вызывает функцию, API, обращаясь к драйверу, вызывает его функции, драйвер обращается к устройству, а устройство выполняет операцию. Если устройство по команде должно вернуть какую-либо информацию, то эта информация также будет передаваться по той же цепочке, только уже в обратном направлении. Естественно, на эту цепочку вызовов затрачивается время.

Для увеличения производительности программа может работать с ВУ без использования API операционной системы — либо через драйвер, либо непосредственно с ВУ. В первом случае программа пользуется API драйвера, во втором — API самого ВУ. Работа непосредственно с ВУ наиболее эффективна, но приводит к зависимости программы от типа ВУ. Для сколько-нибудь массовой программы придётся реализовать работу как минимум с несколькими наиболее распространёнными типами ВУ, то есть фактически написать драйвера этих ВУ для конкретной программы и включить их в состав этой программы.

Работа через стандартный драйвер ВУ — способ, который обеспечивает хорошую переносимость программы при достаточно высокой эффективности. Конечно, универсальный драйвер ВУ, входящий в состав ОС, может оказаться несколько хуже специализированного драйвера, созданного для конкретной программы, однако разница (при хорошо разработанном API драйвера и качественной реализации самих драйверов) не так уж велика (если она вообще окажется в пользу специализированного драйвера, что бывает далеко не всегда), а преимущества, которые даёт его использование, достаточно значительны.

Для некоторых устройств ОС в качестве стандартного способа работы программы с устройством устанавливают работу программы через драйвер ВУ. То есть сама ОС в своём API не содержит функций работы с данным ВУ, и любая программа, рабо-

тающая с ним, работает помимо ядра ОС, через драйвер. Таким способом, например, все программы под Windows 95/98, не использующие библиотеку DirectX, работают с джойстиками.

Глава 2

Структура и принципы построения отдельных подсистем операционной системы.

2.1 Разделение процессорного времени.

2.1.1 Критерии эффективности системы разделения времени.

Проблема распределения процессорного времени между различными одновременно выполняющимися процессами возникла с момента, когда потребовалось обеспечить одновременное решение на вычислительной машине с единственным процессором множества задач. На ЭВМ первого поколения этой проблемы не существовало, задачи решались последовательно, очередью задач управляли вручную (операторы просто ставили на выполнение одну поступающую задачу за другой), что почти всегда приводило к неоптимальному расходованию ресурсов ЭВМ. На следующих поколениях компьютеров функция распределения процессорного времени была возложена на операционную систему. Рассмотрим методы, которыми ОС может выполнять эту функцию, и исследуем, насколько оптимальное распределение процессорного времени между процессами они обеспечивают.

Прежде всего, определимся с тем, какие именно критерии мы будем использовать для определения степени оптимальности той или иной схемы распределения процессорного времени. Качество системы разделения времени может быть определено на основании двух соображений.

Во-первых, хорошая система должна максимально эффективно использовать процессорное время. Это означает, что полезная загрузка процессора (естественно, при наличии нерешённых задач) должна быть как можно ближе к 100%. Данное требование выдвигается потому, что процессорное время — самый дорогой ресурс ЭВМ, и его потеря не может быть восполнена никаким образом.

Во-вторых, желательно, чтобы система обеспечивала такой режим работы, при ко-

тором (при условии постоянства количества задач, одновременно решаемых в системе) наблюдалась бы пропорциональная зависимость между временем, которое задача пробудет в системе и чистым временем, необходимым для её решения. Не должно быть зависимости времени пребывания задачи в системе от параметров других задач, находящихся в системе одновременно с нею. Это условие выдвигается для того, чтобы простые (требующие мало процессорного времени) задачи выполнялись быстрее сложных (требующих большого объёма вычислений). Такой режим необходим хотя бы потому, что простых задач обычно больше, чем сложных, и они требуют более оперативного решения.

Для анализа схем разделения времени примем некоторые допущения. Будем исследовать прохождение решения некоторой контрольной задачи в однопроцессорной вычислительной системе. Будем считать, что очередь задач, которые необходимо решить, имеет на момент входа нашей (контрольной) задачи длину n-1. После входа контрольной задачи длина очереди становится равной n, после чего она остаётся постоянной до момента выхода контрольной задачи из системы. Таким образом, в любой момент времени, когда контрольная задача уже поступила в систему, но ещё не решена, количество задач равно n. Естественно, принятие таких допущений делает ситуацию несколько искусственной, однако, выводы, которые можно сделать на этой основе, в целом вполне приложимы к общему случаю.

Для описания задач, выполняющихся в системе, введём следующие обозначения.

n — общее число задач в системе, причём n-я задача — контрольная.

t_i- чистое процессорное время, необходимое для решения i-й задачи.

k_i — количество обращений к ВУ за время выполнения i-й задачи.

l_i — время, необходимое i-й задаче на выполнение обращений к ВУ.

Ясно, что абсолютный минимум времени нахождения i-й задачи в системе составляет $t_i + l_i$. Обозначим это время f_i. Задача будет решена за это время, если в системе не выполняется, кроме неё, ни одной задачи.

Прежде чем рассматривать структуру реальных систем разделения времени, попробуем представить себе наиболее очевидные способы выполнения нескольких задач на одном процессоре и определим их достоинства и недостатки.

Наиболее очевидным способом разделения процессорного времени между задачами является простая очередь (см. рис. 2.1). Работать она может следующим образом. Операционная система создаёт очередь задач, ожидающих решения. Поступающая в систему задача ставится в очередь. Как только процессор освобождается, первая задача выбирается из очереди и ставится на процессор. Задача выполняется полностью, либо до завершения, либо до возникновения фатальной ошибки (поскольку нам неважно, произошло корректное завершение задачи или ошибка, в дальнейшем мы не будем различать эти два случая, и будем всегда говорить о завершении задачи). После её завершения из очереди выбирается следующая по порядку задача и так далее.

Достоинством такой схемы является только то, что она чрезвычайно проста и не требует практически никаких затрат на реализацию. Все остальные соображения говорят против неё.

Во-первых, использование простой очереди привело бы к значительным непро-

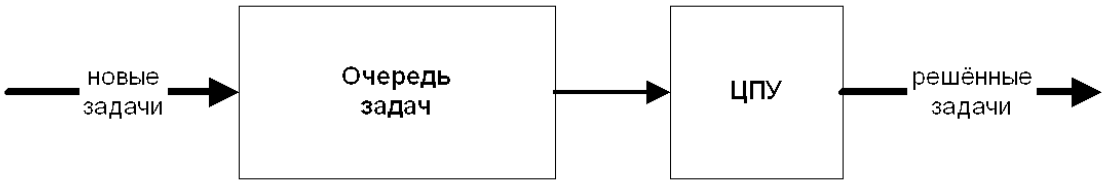

Рис. 2.1: Простая очередь.

изводительным простоям процессора. Простои эти возникают потому, что никакая программа не выполняет только обработку данных. Любой программе за время работы необходимо, по крайней мере, дважды обратиться к устройствам ввода-вывода — для ввода исходных данных и для вывода результатов работы программы. Возможны и промежуточные обращения. Обращения к этим устройствам характерны тем, что они занимают, по компьютерным меркам, очень много времени. Так, чтобы считать данные с магнитного диска, требуется время, равное, по меньшей мере, времени одного оборота диска. Даже если на это понадобится всего 10 мс, то за это время процессор с производительностью 1 млн. операций в секунду успевает произвести 10 тыс. операций, что эквивалентно решению небольшой задачи. Но в нашей схеме в моменты, когда происходит обращение к внешним устройствам, процессор просто простаивает, ожидая завершения операции ввода-вывода. Такое положение приводит к тому, что для задач со значительным объёмом ввода-вывода простои процессора могут составить свыше 50% полезного времени.

С точки зрения второго из выбранных нами критериев, простая очередь также неприемлема. Суммарное время, затраченное на решение задачи в такой системе, будет, очевидно, рассчитываться как общее время решения и операций ввода-вывода для всех задач в очереди, включая контрольную задачу.

Таким образом, время на решение контрольной задачи складывается из процессорного времени, необходимого собственно для решения данной задачи, времени на операции ввода-вывода для этой задачи и полного времени решения всех остальных задач в очереди. Зависимость между полным временем пребывания задачи в системе и параметрами очереди и задач в ней выражается формулой $T = \sum_{i=1}^{n} f_i$, или, что для наших целей более показательно, $T = f_n + \sum_{i=1}^{n-1} f_i$

Из последней формулы видно, что время пребывания задачи в системе складывается из чистого времени её решения (f_n) и суммарного времени решения всех задач, находящихся в очереди перед контрольной задачей. Фактически, никакой связи между сложностью решаемой задачи и временем, которое потребуется для её решения, не существует, зато существует связь между временем пребывания контрольной задачи в системе и сложностью задач, которые будут решаться вместе с ней. В результате на решение сложной задачи может потребоваться меньше времени, чем на решение простой. Одна сложная задача или задача с большим объёмом ввода-вывода, оказавшаяся в очереди первой, может существенно замедлить решение множества простых

задач, оказавшихся в очереди после неё.

Мы видим, что схема простой очереди безнадёжно неэффективна как в смысле загрузки процессора, так и в смысле обеспечения решения задач за время, пропорциональное их сложности. Попытаемся улучшить её.

Поскольку причиной неэффективности простой очереди являются простои процессора при обращении задач к ВУ, напрашивается решение — на время обращения к ВУ снимать задачу с процессора. Схема работы подобной системы приведена на рис. 2.2. Порядок решения задач следующий.

- Задача выбирается из очереди и ставится на процессор.
- Задача выполняется либо до завершения, либо до обращения к ВУ. В любом случае задача снимается с процессора, после чего на процессор ставится следующая задача из очереди.
- Завершившаяся задача покидает систему.
- Если задача обратилась к ВУ, то она ставится в список заблокированных задач и ждёт, пока завершится операция ввода-вывода.
- После завершения обращения к ВУ задача удаляется из списка заблокированных задач и снова ставится в конец очереди задач на выполнение.

При описанном порядке исполнения после каждого обращения к ВУ задача должна снова ожидать в очереди к процессору.

С точки зрения оптимизации использования процессорного времени такая схема практически идеальна. Простои процессора в ней очень малы, процессор простаивает только в моменты переключения с одной задачи на другую, на что тратится очень немного времени.

Рассмотрим, как выглядит такой метод с точки зрения второго критерия. Контрольная задача за время выполнения должна поступить на процессор $(k_n + 1)$ раз. Перед каждой постановкой на процессор задача должна ожидать в очереди, пока $(n - 1)$ задач будут либо завершены, либо заблокированы на ввод/вывод. Предполагая, что обращения к ВУ происходят в среднем через равные промежутки времени, получим, что среднее время непрерывного нахождения задачи на процессоре составляет $\frac{t_i}{k_i+1}$. Время одного ожидания в очереди составит $\sum_{i=1}^{n-1} \frac{t_i}{k_i+1}$, а полное время решения задачи можно, соответственно, оценить как $T = t_n + l_n + (k_n+1) \cdot \sum_{i=1}^{n-1} \frac{t_i}{k_i+1} = f_n + \sum_{i=1}^{n-1} t_i \cdot \frac{k_n+1}{k_i+1}$.

Сравнивая вычисленные затраты времени с затратами в предыдущем случае, можно отметить следующее. Полное время решения задачи зависит от частоты обращений задач (в том числе самой контрольной задачи) к ВУ, поэтому точно указать, приведёт ли блокировка к сокращению времени решения контрольной задачи, невозможно. Однако если предположить, что в очереди присутствуют задачи, примерно одинаковые по структуре, и количества обращений к ВУ у них приблизительно равные, то есть отношение $\frac{k_n}{k_i}$ близко к единице для всех i, то время задержки (то есть разницу между

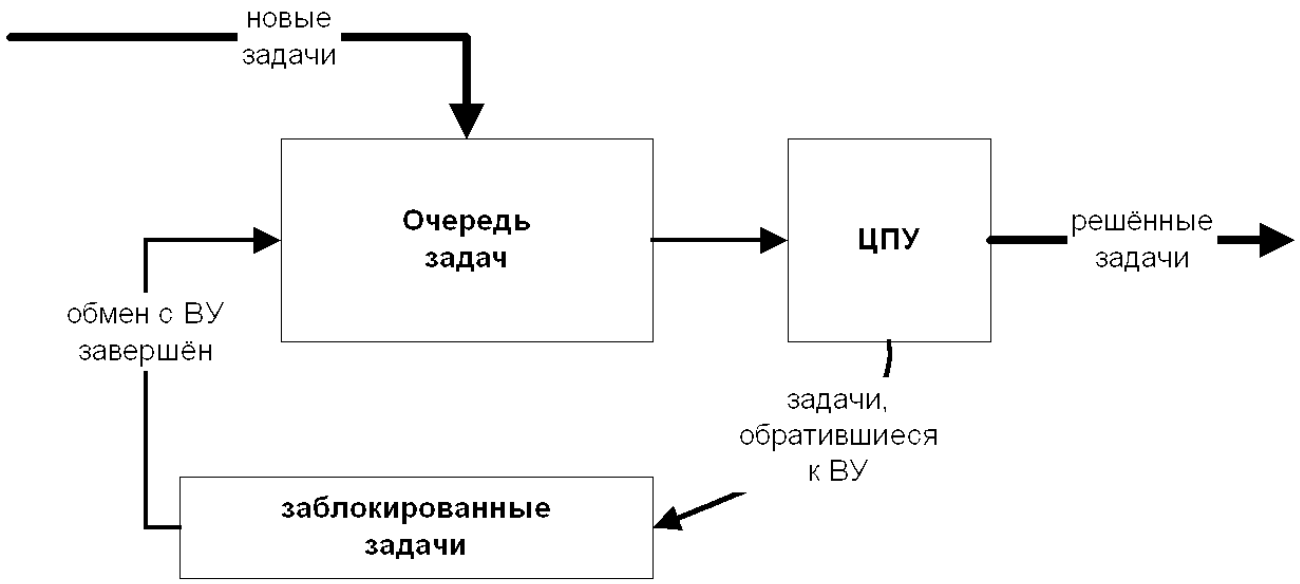

Рис. 2.2: Очередь с блокировкой.

T и f_n) можно оценить как $\sum_{i=1}^{n-1} t_i$, что на $\sum_{i=1}^{n-1} l_i$ меньше, чем при использовании схемы простой очереди.

Таким образом, можно сказать, что очередь с блокировкой обеспечивает меньшее время ожидания, чем простая очередь. Тем не менее, она имеет тот же основной недостаток — потери времени на ожидание задачи в очереди не зависят от сложности самой задачи. То есть, несмотря на некоторое улучшение, остаётся та же проблема — сложная задача, требующая большого объёма вычислений, блокирует решение простых задач, которые стоят в очереди после неё. Правда, у простых задач есть шанс раньше попасть на процессор, за счёт того, что сложная задача при обращении к ВУ будет снята с процессора, но ведь сложная задача может работать, не обращаясь к ВУ, длительное время.

2.1.2 Круговорот.

Системы разделения времени реальных ОС, в большинстве своём, работают по схеме, называемой «Round Robin» (русское название — «Круговорот»), с теми или иными дополнениями и/или изменениями.

Схема круговорота похожа на придуманную нами схему очереди с блокировкой, но в неё добавлена одна принципиальная особенность (см. рис. 2.3). Процессорное время, которое в ранее описанных схемах рассматривается как непрерывное, в данной схеме квантуется, то есть разделяется на элементарные единицы – кванты. Планировщик задач ОС (модуль, который реализует управление процессами) выделяет задачам

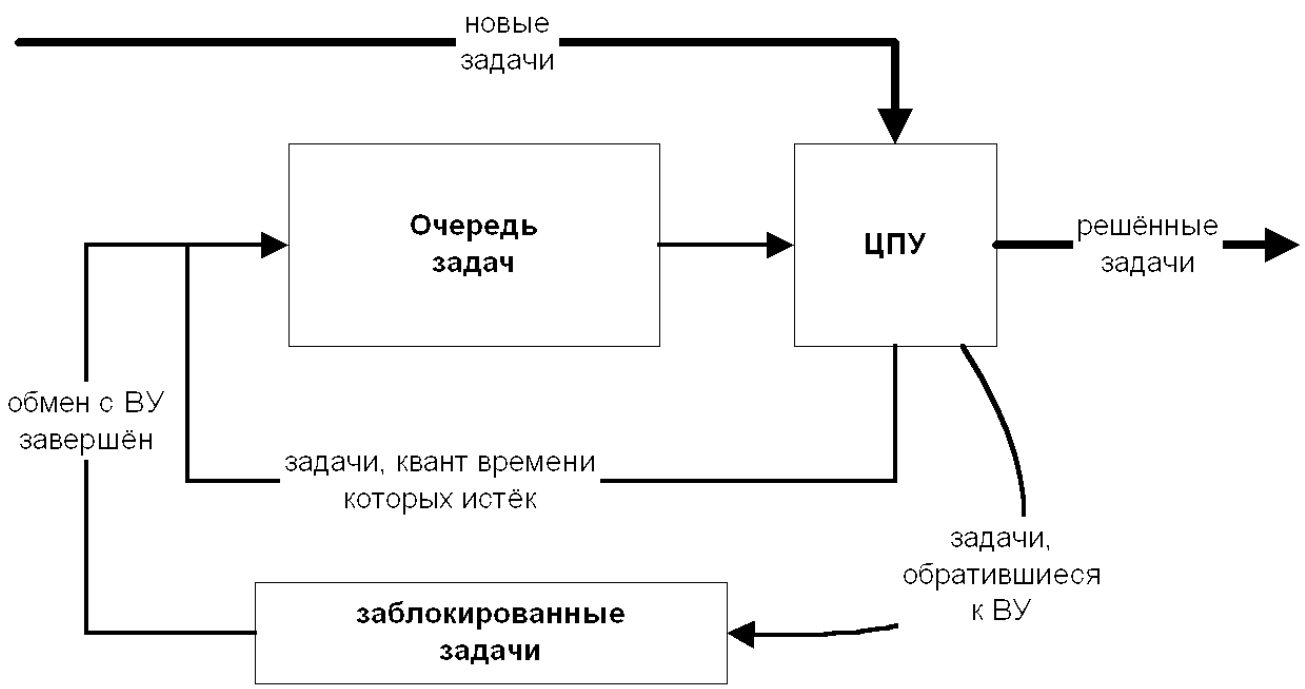

Рис. 2.3: Круговорот.

процессорное время квантами. По завершении кванта времени планировщик задач принимает решение, какому процессу выделить следующий квант. Конкретный размер кванта зависит от ОС, а в некоторых ОС может настраиваться. При выборе размера кванта времени разработчикам ОС приходится находить некую «золотую середину» между двумя крайностями. Выбор слишком маленького кванта времени приводит к увеличению непроизводительного расхода процессорного времени – ведь после каждого кванта планировщик должен определять, кому предоставить следующий, а на это тоже затрачивается процессорное время. При выборе слишком большого размера кванта снижается оптимальность распределения времени.

Схема круговорота функционирует следующим образом. Поступающая в систему задача сразу ставится на процессор. Если на момент поступления новой задачи процессор уже занят какой-то другой задачей, то эта задача снимается с процессора и ставится в очередь. Поступившая задача выполняется в течение одного кванта времени, либо до момента, когда ей потребуется обратиться к ВУ, либо до момента поступления в систему новой задачи, в зависимости от того, что случится раньше. Если истёк квант времени, но задача ещё не решена, то задача ставится в конец очереди и ждёт, а на процессор поступает либо первая задача из очереди. Если пришла новая задача, то она ставится на процессор, не дожидаясь конца кванта, а решаемая до этого задача ставится в очередь. Если задача завершилась, то она покидает систе-

му, а планировщик завершает текущий квант времени и начинает следующий. Если, наконец, задача обратилась к ВУ, то она снимается с процессора и блокируется на время обращения, а после работы с ВУ ставится в конец очереди к процессору.

Полезная загрузка процессора при использовании круговорота несколько меньше, чем при использовании простой очереди с блокировкой, поскольку ОС вынуждена вмешиваться в работу приложений достаточно часто — один раз за один квант времени, а иногда и чаще. Однако при разумно выбранной величине кванта времени реальные потери невелики. Полезный же эффект, как будет показано ниже, достаточно значителен.

Обозначим квант времени Δ. Для решения контрольной задачи она должна попасть на процессор $\frac{t_n}{\Delta} + k_n$ раз. В результате время на решение задачи, без учёта времени ожидания в очереди, составит $(\frac{t_n}{\Delta}+k_n)\cdot\Delta$. Для того, чтобы получить каждый следующий квант времени, задача должна попасть в очередь и ожидать там освобождения процессора. Количество попаданий в конец очереди составит $\frac{t_n}{\Delta}+k_n-1$. После каждой постановки в очередь задача должна ожидать $(n-1)$ квант времени, прежде чем снова попадёт на процессор. Кроме того, время l_n будет затрачено на ожидание при обращении к ВУ. Таким образом, общее время решения задачи составит

$$T = l_n+(\frac{t_n}{\Delta}+k_n)\cdot\Delta+(n-1)\cdot(\frac{t_n}{\Delta}+k_n-1)\cdot\Delta = t_n+l_n+k_n\cdot\Delta+n\cdot\Delta\cdot(k_n-1)+t_n\cdot(n-1)-k_n\cdot\Delta = f_n+g(t_n,k_n,n,\Delta$$

Здесь $g(...)$ — некоторая функция от t_n, k_n, Δ, n, содержащая остальную часть выражения. Она выделена просто для того, чтобы не загромождать результирующее выражение. Для нас в данном случае важен не конкретный вид этой функции, а тот факт, что эта функция зависит только от самой решаемой задачи (параметры t_n и k_n), от размера кванта времени, который в ОС является постоянным, и от количества задач в очереди. При постоянном размере кванта времени и неизменной длине очереди задач функция g зависит (причём линейно) от от t_n и k_n, то есть только от характеристик самой задачи. Практически, время решения будет несколько больше, если в систему во время решения контрольной задачи будут поступать новые задачи. Но поступление новой задачи — достаточно редкое событие, так что значительного влияния оно не окажет. В результате мы видим, что в данной схеме дополнительное время, которое будет потрачено на ожидание освобождения процессора, при постоянном количестве задач в системе зависит только от самой задачи. Это даёт возможность точно рассчитать, какое время будет затрачено на решение задачи при известных чистом времени решения и общей загрузке системы. На время решения контрольной задачи совершенно не влияет степень сложности других задач, находящихся в системе одновременно с ней. Таким образом, схема круговорота вполне удовлетворяет выдвинутым нами требованиям.

Из всего сказанного ясно, что в системе круговорота планировщик задач операционной системы может в любой момент времени снять с процессора любую задачу и поставить любую другую. Реализацию многозадачности, соответствующую этому принципу, называют *вытесняющей многозадачностью*.

2.1.3 Круговорот с приоритетами.

Схема круговорота обычно используется с одним важным дополнением – с учётом приоритетов задач. Каждой задаче перед первой постановкой на процессор пользователем или системой приписывается целое число, называемое приоритетом. Одни значения приоритета считаются более высокими, другие — более низкими [1]. Считается, что высокоприоритетные задачи более важны, чем низкоприоритетные, и поэтому они должны решаться быстрее. Схема разделения времени функционирует при этом точно так же, как было описано, но при постановке задачи в очередь к процессору она ставится не в конец очереди, а после самой последней в очереди задачи с тем же приоритетом или, если таких задач нет, после последней задачи с более высоким приоритетом. Если в системе появляется задача с приоритетом более высоким, чем приоритет задачи, находящейся на процессоре, то процессор немедленно освобождается для более высокоприоритетной задачи.

Таким образом, задача с самым высоким приоритетом всегда будет оказываться в очереди к процессору первой и выполнится без задержек, а задача с самым низким приоритетом сможет попасть на процессор только в те моменты, когда процессор не занят решением других задач, то есть когда высокоприоритетных задач в системе нет, либо все они заблокированы на обращение к ВУ.

2.1.4 Система динамических приоритетов.

В некоторых случаях оказывается желательным предоставление простым задачам дополнительных преимуществ перед сложными. Это бывает нужно, когда, наряду с постоянной загрузкой системы относительно сложными задачами, время от времени возникают задачи простые, но требующие быстрого решения. В таких случаях может применяться система динамических приоритетов.

Эта система аналогична круговороту с приоритетами, но приоритет задачи не задаётся изначально, а изменяется по ходу её выполнения. Первоначально, при входе задачи в систему, ей присваивается нулевой (наивысший) приоритет [2]. После каждого снятия задачи с процессора её приоритет увеличивается на единицу (то есть становится более низким). Таким образом, после первого кванта времени обработки приоритет задачи будет равен 1, после второго — 2 и т.д. В очереди к процессору задачи располагаются в порядке возрастания значения приоритета. Чем дольше задача выполняется, тем больше значение её динамического приоритета и тем реже, соответственно, ей предоставляется очередной квант времени. При такой схеме короткая

[1] В зависимости от выбора разработчиков ОС более высоким может считаться приоритет, числовое значение которого больше, или приоритет, числовое значение которого меньше. В первом случае приоритет 0 (нулевой) будет самым низким, и задача с таким приоритетом будет считаться наименее срочной, во втором - тот же приоритет будет считаться наивысшим, и задача с таким приоритетом будет считаться наиболее срочной. Возможны и другие варианты.

[2] См. предыдущую сноску. Здесь приоритеты считаются по второму варианту: нулевой - наивысший, все остальные — ниже.

задача, требующая всего несколько квантов времени, не успевает набрать большое значение динамического приоритета и попадает на процессор часто. Если же задача в ходе выполнения набирает самое большое значение динамического приоритета среди всех задач в системе, то она сможет попадать на процессор только тогда, когда тот не будет занят никакой другой задачей, то есть превратится в фоновую задачу.

Система динамических приоритетов осуществляет перераспределение процессорного времени в пользу коротких задач. За счёт этого происходит увеличение времени решения задач сложных, требующих большого объёма вычислений. С этим эффектом приходится мириться, правда, он не особенно существенен, так как сложных задач обычно не так много и они не требуют столь оперативного решения, как простые.

2.1.5 Функционирование ядра в ОС с вытесняющей многозадачностью.

Ядро операционной системы является, как и все задачи пользователя, процессом, запущенным на том же компьютере. Однако, в силу своего особого положения, ядро выполняется несколько необычным образом.

В состав ядра входит планировщик задач ОС. Планировщик задач должен периодически получать управление, чтобы обеспечивать переключение процессора с одной задачи на другую. Естественно, модуль ядра, содержащий планировщик, является привилегированным. Он может в любой момент вытеснить с процессора любую задачу, его же не может вытеснить никто. Планировщик запускается периодически, независимо ни от каких событий, происходящих в системе.

Порядок работы остальных компонентов ядра во многом зависит от его архитектуры. В ОС с монолитным ядром всё ядро (в силу своей нераздельности) имеет тот же приоритет, что и планировщик задач. Это означает, что выполнение любой операции в ядре не может быть приостановлено планировщиком задач в интересах какого-то другого процесса. Поэтому когда в ядре системы происходит какая-нибудь задержка, вся система на время этой задержки «подвисает» и никакие программы выполняться не могут. Естественно, в правильно спроектированных ядрах такого рода задержки редки и не особенно велики, но, тем не менее, они имеют место.

В ОС с микроядром ситуация достаточно сильно отличается. Само микроядро содержит, можно считать, только планировщик задач. Все остальные модули ядра выполняются на общих основаниях, хотя и могут иметь больший, чем программы пользователя, приоритет. Следовательно, прикладная программа, выполняющаяся в системе, вполне может «подвинуть» выполняющуюся на процессоре операцию ядра (например, связанную с вводом-выводом). Именно по этой причины микроядерные системы (или системы, в которых реализованы сходные механизмы), иногда называют «системами с вытесняемым ядром», хотя этот термин и не вполне точен.

Возвращаясь к тому, что было выше рассмотрено в разделе 1.2.1, теперь можно объяснить особенности монолитных и микроядер. Большая средняя производительность систем с монолитными ядрами достигается за счёт привилегированного режима

исполнения *всех* операций ядра, но эта же привилегированность является причиной увеличения времени отклика. Низкое время отклика микроядерных систем достигается за счёт того, что меньший объём кода исполняется в привилегированном режиме, но, по той же причине, в некоторых случаях может снизиться средняя производительность системы.

2.1.6 Кооперативная многозадачность.

Вытесняющая многозадачность удобна и надёжна, но для её эффективной реализации аппаратная часть системы должна содержать специальные поддержки, обеспечивающие выполнение необходимых действий (таких как переключение задач). Сказанное не означает, что без таких средств реализовать вытесняющую многозадачность нельзя, просто такая реализация будет, скорее всего, технически неэффективна.

Одним из способов реализации многозадачности в системах, не имеющих для этого соответствующих аппаратных средств, является кооперативная многозадачность. В системе с кооперативной многозадачностью ОС не в состоянии в произвольный момент времени вытеснить выполняющуюся задачу с процессора. Переключение процессов происходит в моменты, когда процессы обращаются к внешним устройствам или ожидают от пользователя ввода данных или команды. Для выполнения ввода-вывода процесс обычно вызывает функции API, а эти функции написаны таким образом, что при их вызове ОС производит переключение процессора с одной задачи на другую.

Для того чтобы программы, не обращающиеся к функциям ввода-вывода, могли выполняться параллельно в такой системе, в ней к написанию программ предъявляются специальные требования, заключающиеся в следующем. Программа, написанная для ОС с кооперативной многозадачностью, если она содержит достаточно длительные операции, не связанные с вводом-выводом, должна во время выполнения этих операций достаточно часто вызывать некоторую стандартную функцию API. При вызове этой функции задача приостанавливается и активизируется ОС. ОС принимает решение, какую задачу поставить на процессор далее, и запускает её. Таким образом, задача в тех случаях, когда её не может снять с процессора ОС, фактически снимает себя с процессора самостоятельно. Естественно, чтобы задачи выглядели выполняющимися параллельно, вызовы ОС из них должны производиться достаточно часто.

Единственным плюсом кооперативной многозадачности является простота её реализации. Платой за эту простоту является принципиальная неустойчивость системы по отношению к неправильно работающим прикладным программам. Если запущенная задача не активизирует ОС, то эта задача единолично захватывает процессор. При этом остальные задачи и ядро самой ОС будут просто стоять. Если одна задача зависнет, то вместе с нею зависнет вся система. Естественно, в реальных системах, использующих кооперативную многозадачность, принимаются специальные меры для недопущения блокировки системы одной задачей, но до тех пор, пока система не реализует вытесняющую многозадачность во всей полноте, эти меры не обеспечат устойчивости системы во всех ситуациях.

2.1.7 Автоматическое распределение процессорного времени в многопроцессорных системах.

Рассмотренные методы разделения процессорного времени были приведены в предположении, что система, в которой работает ОС, имеет физически только один процессор. Однако желательно, чтобы ОС могла своими средствами обеспечивать оптимальную загрузку и нескольких процессоров. Такая возможность необходима для работы в системах, имеющих несколько процессоров (обычно два-четыре), в которых одновременно выполняется множество процессов.

Задача эта решается ОС путём небольшого дополнения к системе распределения процессорного времени с использованием квантования. Если в однопроцессорной системе ОС просто ставит очередную задачу на процессор, как только он освободится, то в многопроцессорной системе первая задача из очереди ставится на тот процессор, который оказался свободен раньше. В результате любой освободившийся процессор немедленно занимается следующей задачей из очереди, и все процессоры системы загружаются равномерно и полно. Обычно нет однозначной привязки одного определённого процесса к одному определённому процессору — процесс ставится на выполнение на тот процессор, который раньше освободится.

Кроме того, в некоторых ОС, поддерживающих многопроцессорные системы, имеется также возможность захватить для одной задачи отдельный процессор. Возможность эта используется обычно для запуска задач, работающих в реальном времени или особенно критичных по времени выполнения. [3]

2.1.8 Процессы и потоки.

Помимо многозадачности, обеспечивающей возможность одновременного выполнения нескольких процессов в системе [4], ОС может предоставлять возможность запускать в пределах одного процесса (одной задачи) несколько фрагментов, выполняющихся одновременно. Такие фрагменты называются потоками (threads).

В качестве примера использования потоков можно привести достаточно распространённую ситуацию, когда программа выполняет некоторую длительную операцию (большой по объёму расчёт), которую пользователь должен иметь возможность прервать, если возникнет такая необходимость.

При традиционном подходе необходимо в ходе расчёта постоянно (причём с достаточно малой периодичностью) проверять, не дал ли пользователь команду на пре-

[3] Естественно, чтобы выполнить такую операцию, необходимо, чтобы, во-первых, у пользователя, запускающего задачу, были соответствующие права, а во-вторых, чтобы в системе имелась возможность выделить под задачу отдельный процессор. Если в системе два процессора и уже запущена одна задача, захватившая процессор, то вторую такую задачу запустить будет невозможно до завершения первой, поскольку ОС требует для работы, по крайней мере, один процессор, работающий в обычном (многозадачном) режиме.

[4] Разумеется, если в системе имеется меньше процессоров, чем выполняемых процессов, то физически одновременно выполнять все процессы система не может. Здесь и далее имеется в виду не физическая, а логическая параллельность.

кращение вычислений. Это значит, что нужно внести в расчёт совершенно инородный для него код проверки завершения. Кроме того, если расчёт зависнет (зациклится), то процесс не сможет отреагировать на команду пользователя, потому что выполнение его никогда не дойдёт до очередной точки проверки наличия команды завершения.

Гораздо удобнее было бы запустить процедуру расчёта так, чтобы она выполнялась параллельно с остальной частью процесса. При этом процесс должен запустить процедуру расчёта и ожидать либо её завершения, либо команды отмены. Сама процедура должна выполняться одновременно с фрагментом, ожидающим команду на прекращение вычислений.

Такая возможность имеется в тех ОС, которые поддерживают возможность создания нескольких потоков исполнения в одном процессе. Процесс в таких системах представляет собой совокупность одного или более одновременно выполняемых потоков. При создании процесса (запуске программы) автоматически создаётся один (первичный) поток. Он, при необходимости, может создать любое количество потоков, которые будут работать одновременно с ним. Используя средства, предоставляемые API ОС, существующий поток создаёт новые потоки и запускает в них требуемые вычисления процедуру. При этом одновременно продолжает выполняться первичный поток [5]. Когда выполнение процедуры завершается, завершается и поток. Процесс считается завершённым тогда, когда завершены все его потоки.

Реализация потоков может быть различной. В одних системах, например, в Windows, поток — это особый системный объект. Он имеет свои, присущие только ему особенности. Поток и процесс для системы — кардинально различные понятия. Поток существует только в каком-то процессе. Процесс в такой ОС — это некое «вместилище потоков», само по себе не выполняющее никаких действий. Вычислениями в процессе занимаются один или несколько потоков. Ресурсы системы выделяются процессу и закрепляются за ним. Все потоки одного процесса используют одну и ту же память, имеют доступ к файлам, открытым друг другом, сосуществуют в одном адресном пространстве. Потоки одного процесса могут взаимодействовать через общую для них память процесса.

В других системах, например, таких, как практически все ОС, базирующиеся на UNIX, поток — это просто дочерний процесс (т.е. полноправный процесс, порождённый первичным процессом путём вызова соответствующей функции API). Дочерний процесс может быть «неполноценным» процессом (не иметь командной строки, иметь ограниченные права и пр.), но и в этом случае он остаётся отдельным процессом. Он может выделять себе системные ресурсы, которые будут недоступны его родительскому процессу и другим дочерним процессам того же родительского процесса. В результате многопоточная программа, фактически, представляет собой группу относительно самостоятельных процессов. При такой организации потоки взаимодействуют между собой через стандартные системные средства, предназначенные для взаимодействия процессов. Если возникает необходимость обеспечить работу потока с памятью,

[5] Здесь одновременность нужно понимать в том же смысле, что и для процессов (см. предыдущую сноску)

выделенной родительским процессом, то это обычно возможно путём задания специальных параметров в команде запуска потока. Например, можно создать дочерний процесс так, что он унаследует адресное пространство своего родителя и, следовательно, будет иметь доступ к памяти, выделенной родительским процессом. Через эту память два дочерних процесса могут, например, передавать данные друг другу.

В ОС, поддерживающих потоки, система разделения времени распределяет процессорное время между отдельными потоками, а в остальном работает так же, как было описано выше. Можно сказать, что для системы разделения времени в таких ОС единицей исполнения является не процесс, а поток.

Если в ОС потоки могут быть реализованы только как дочерние процессы, каждый из таких дочерних процессов имеет собственный приоритет, который в этом случае можно назвать *абсолютным приоритетом*. Абсолютность приоритета заключается в том, что именно его значение используется ОС для определения порядка постановки процессов в очередь к процессору.

Если ОС реализует потоки как специфические системные объекты, принадлежащие процессам, то и процессы, и потоки могут иметь собственные приоритеты (так обстоит дело в ОС Windows). При этом процесс при запуске получает какой-то приоритет, а каждый поток в процессе запускается с собственным приоритетом, который можно назвать *относительным*. Относительный приоритет потока определяет приоритет данного потока относительно приоритета содержащего его процесса. Диапазон числовых значений, в пределах которого может быть установлен относительный приоритет потока, обычно существенно меньше, чем диапазон значения приоритетов процессов. *Абсолютный приоритет потока*, то есть приоритет, в соответствии с которым система разделения времени устанавливает потоки в очереди к процессору, вычисляется системой как сумма (или разность) приоритета процесса и относительного приоритета потока. В результате высокоприоритетный поток низкоприоритетного процесса, возможно, окажется в очереди дальше от процессора, чем низкоприоритетный поток высокоприоритетного процесса. Например, если в процессе, запущенном с приоритетом 1000, выполняется поток с приоритетом 20, то абсолютный приоритет данного потока в системе может составлять 1020. Естественно, что любой поток процесса, запущенного с приоритетом 1500, будет иметь более высокий абсолютный приоритет, даже если относительный приоритет такого потока нулевой. [6]

2.1.9 Управление планировщиком задач.

Многозадачная ОС может предоставлять возможность настройки системы разделения времени и управления её работой. При наличии такой возможности система разделения времени должна настраиваться в соответствии с требованиями к системе в конкретных условиях её применения. Настраиваться могут следующие параметры.

[6] Естественно, здесь предполагается, что в ОС большее числовое значение соответствует более высокому приоритету.

Размер кванта времени (очень редко). Как уже говорилось, увеличение размера кванта времени приводит к снижению качества распределения времени, а уменьшение размера кванта — к снижению полезной загрузки процессора. Обычно значение, установленное по умолчанию, представляет собой разумный компромисс между качеством и эффективностью. Изменение его следует производить с осторожностью и на небольшую величину. Лучше всего подобрать оптимальное значение экспериментальным путём, ориентируясь на работу системы при реальной загрузке.

Автоматическое увеличение приоритета процесса, с которым работает пользователь. Некоторые ОС могут автоматически увеличивать приоритет того процесса, с которым в данный момент работает пользователь. Такая возможность есть, в частности, в ОС семейства Windows NT. Система может автоматически увеличивать приоритет текущего процесса, обеспечивая тем самым максимальный комфорт пользователю при работе с программой, и уменьшать его, если пользователь начинает работать с другой программой. Естественно, это увеличение приоритета приводит к некоторому замедлению работы фоновых приложений. Обычно рекомендуют включать автоматическое увеличение приоритета в ОС, установленных на рабочих станциях. Это делает работу пользователя более удобной, поскольку обычно пользователь активно работает только с одной программой, а остальные, запущенные одновременно в фоновом режиме, могут быть несколько замедлены. Не рекомендуется включать автоувеличение приоритета на серверах, с которыми одновременно работают многие пользователи, и на которых одновременно работает множество программ, поскольку в таких условиях оно приводит лишь к уменьшению средней производительности системы. Если на рабочей станции активно работает большое количество взаимодействующих программ, то автоувеличение приоритета также лучше выключить.

Помимо настройки параметров самого планировщика задач, пользователь может иметь возможность управлять приоритетом запускаемых или уже запущенных программ. Он может задать или изменить приоритет процесса, обеспечив ему тем самым наиболее или наименее благоприятные условия. Используя описанную возможность, необходимо проявлять осторожность, поскольку задание процессу слишком высокого приоритета приведёт к замедлению работы всей системы в целом, а снижение приоритета может значительно увеличить время работы программы. В некоторых случаях процессы реагируют на изменение приоритета парадоксальным образом – замедляются при увеличении приоритета. Такое поведение характерно для процессов, работающих с медленными внешними устройствами в режиме опроса.

Рекомендуемой практикой при управлении приоритетами процессов является следующая: нужно не увеличивать приоритет тех задач, которые должны выполняться быстрее, а уменьшать приоритет тех, которые могут работать с замедлением. Эффект от такого действия будет тем же самым, но оно заведомо не приведёт к замедлению работы самой ОС, вызванному тем, что пользовательская задача получит больший приоритет, чем некоторые системные процессы. Увеличение приоритета задачи выше установленного системой следует рассматривать как исключительную, крайнюю меру,

применять которую нужно с осторожностью.

2.2 Управление памятью.

Вторым (после процессорного времени) критическим ресурсом компьютера является его оперативная память.

Физической оперативной памяти в ЭВМ, как правило, всегда не хватает для удовлетворения потребности в памяти всех выполняющихся в системе задач. Таким образом, ОС должна выполнять функцию распределения оперативной памяти между запущенными процессами.

Кроме того, при одновременной работе многих приложений существует проблема защиты памяти, принадлежащей ОС, от несанкционированного изменения её запущенными приложениями и защиты памяти процессов от изменения её из других процессов.

2.2.1 Физическая память и адресное пространство.

Компьютерная система располагает определённым объёмом физической оперативной памяти. Любая ячейка физической памяти имеет определённый *физический адрес*. Один и тот же физический адрес всегда относится к одной и той же ячейке оперативной памяти. Физический адрес ячейки памяти не может меняться со временем. Другими словами, между множеством ячеек памяти и множеством физических адресов существует взаимно однозначное соответствие. Объём физической памяти и её физическая адресация определяется аппаратной архитектурой ЭВМ.

Логическим адресом называется адрес, который используется программой для доступа к памяти. *В общем случае физический и логический адрес одной и той же ячейки памяти не совпадают*. Более того, одной и той же физической ячейке памяти в одни моменты времени может соответствовать несколько логических адресов одновременно, а в других — не соответствовать ни одного логического адреса (в последнем случае ни одна из программ не может иметь доступа к данной ячейке памяти). Естественно, любая доступная ячейка памяти имеет определённый физический адрес (причём, возможно, тоже не один). Физический адрес (адреса) ячейки памяти, как правило — постоянное значение. Без технической переналадки компьютера физические адреса ячеек его памяти всегда остаются одними и теми же. Логические адреса, напротив, могут изменяться прямо во время работы системы. Как правило, в любом компьютере есть физические адреса, которым не соответствует никакой ячейки памяти. Как правило, в любой ОС есть логические адреса, которым также не соответствует реальная физическая память.

Адресным пространством называется множество *логических* адресов, в пределах которого в данной системе может (принципиально) обратиться в память программа. Размер адресного пространства есть сумма размеров всех ячеек памяти, адресуемых всеми адресами адресного пространства, он зависит от ОС и может не совпадать (и

почти всегда не совпадает) с объёмом физической памяти, фактически установленной в компьютере. Так, например, в операционной системе MS-DOS размер адресного пространства составляет 1 Мбайт, из которых первые 640 Кбайт доступны для прикладных программ. На первых компьютерах, на которых она выполнялась, ставилась физическая память 128, 256 или 512 Кбайт, то есть объём физической памяти был меньше размера адресного пространства. На современных персональных компьютерах объём оперативной памяти может составлять десятки и даже сотни мегабайт, но программы, созданные для работы под управлением MS-DOS, не могут использовать более 1Мбайт из них, потому что эта ОС не позволяет использовать больший объём адресного пространства. [7]

2.2.2 Структура адресного пространства.

ОС может обеспечивать различные режимы размещения данных и кода программ, равно как и собственных данных и кода, в памяти.

Простейшим вариантом является размещение ОС и всех программ и данных в едином адресном пространстве. При таком варианте ОС организует одно общее адресное пространство, в котором работают все программы и сама ОС (см. рис. 2.4). Любые две программы, обратившиеся по одному и тому же адресу, всегда обращаются к одним и тем же данным или коду. Достоинством такого варианта является его простота. Система может однозначно отобразить любой логический адрес на строго определённый физический. Физический адрес ячейки с заданным логическим адресом не зависит от того, из какого процесса происходит обращение по этому логическому адресу.

Кроме того, в таком варианте упрощается взаимодействие между одновременно выполняемыми процессами — достаточно тем или иным способом передать другому процессу адрес в памяти, чтобы тот мог свободно работать с данными, относящимися к другому процессу. Недостатки такой организации достаточно очевидны. Во-первых, процессы могут повредить данные и код друг друга и, что ещё хуже, операционной системы, то есть такая система принципиально неустойчива. Во-вторых, общее адресное пространство создаёт проблемы с загрузкой программ и данных и с использованием абсолютной адресации. Так, для такой системы невозможно непосредственно поместить в программу команду вида «перейти на адрес XXXXX», поскольку в зависимости от того, как будут загружены в память программы, по этому адресу будут находиться разные команды, причём, возможно, даже относящиеся к различным программам.

Второй вариант — предоставление каждому процессу своего собственного адресного пространства (см. рис. 2.5). В этом случае один и тот же логический адрес для двух разных процессов может означать совершенно разные физические адреса. ОС и аппаратура ЭВМ должны обеспечивать возможность перекодировки адреса в

[7] При использовании XMS и EMS и некоторых дополнительных средств, программы DOS могут, на самом деле, работать более чем с одним мегабайтом ОЗУ, но эти средства не относятся собственно к DOS, они предназначены для преодоления её ограничений.

Рис. 2.4: Общее адресное пространство.

Рис. 2.5: Раздельные адресные пространства.

адресном пространстве процесса в физический адрес. Для этого используется механизм разделения адресного пространства на страницы – фрагменты жёстко заданного объёма.

Для каждого процесса в системе создаётся при запуске так называемая дескрипторная таблица, или таблица страниц. Логический адрес содержит две разделяемые части — номер страницы и смещение искомого адреса внутри страницы. Таблица страниц для каждого процесса хранится в памяти ОС. Для каждой страницы там указан начальный физический адрес. При любом обращении в память происходит разделение адреса на номер страницы и смещение, в таблице страниц процесса находится физический адрес начала страницы, к нему добавляется смещение. В результате получается физический адрес, по которому и происходит обращение.

Естественно, ОС должна при выделении памяти процессу предварительно проинициализировать соответствующие системные таблицы. Ясно также, что при такой системе, в принципе, можно так организовать адресное пространство процессов, что некоторые страницы в памяти будут общими, некоторые — относиться только к одно-

му процессу. Таким способом можно построить очень гибкую и многофункциональную систему доступа к памяти. Например, можно легко обеспечить доступ к некоторым страницам только на чтение, к другим — на чтение и запись. Можно открыть некоторые данные ОС для чтения, но запретить их изменение. Можно выделить некоторые страницы, которые будут общими для нескольких процессов или для процесса и ОС, обеспечив тем самым совместный доступ к данным.

Для организации отдельного адресного пространства для каждого процесса компьютер, на котором работает ОС, должен иметь соответствующее аппаратное обеспечение. Если таблицы страниц и перекодировка адресов в соответствии с ними не поддерживаются процессором, то реализовать размещение каждого процесса в отдельном адресном пространстве достаточно эффективно практически невозможно.

2.2.3 Виртуальная память. Организация подкачки страниц.

Как уже говорилось, оперативная память, установленная в компьютере, редко бывает достаточного объёма для того, чтобы разместить в ней все одновременно выполняющиеся программы. Чтобы обеспечить одновременное выполнение в системе нескольких программ, которые требуют больше памяти, чем имеется в ОЗУ ЭВМ, ОС может предоставлять (и обычно предоставляет) программам так называемую виртуальную память.

Идея виртуальной памяти принципиально очень проста: использовать для хранения информации, не помещающейся в ОЗУ, внешнюю память, то есть накопители на магнитных дисках. При этом ОС организует это хранение так, чтобы для программ, выполняющихся под её управлением, работа с памятью не зависела от того, где физически находится информация – в ОЗУ или на диске.

Стандартный способ организации виртуальной памяти базируется на уже упомянутом разделении адресного пространства на страницы. Работает он следующим образом. Когда программа требует некоторый объём памяти, память физически не выделяется. Вместо этого в таблице страниц процесса просто инициализируется соответствующее количество строк, описывающих параметры страниц памяти. Таким образом, пока процесс не начнёт реально пользоваться физической памятью, ОС может использовать её для других процессов. Когда программа делает попытку обратиться в выделенную память по какому-либо адресу, ОС пытается найти в таблице соответствующую страницу, взять оттуда адрес физической памяти и обратиться по нему. Однако память под страницу ещё не выделена. Возникает ситуация, которая именуется «страничным сбоем». ОС пытается выделить новую страницу физической памяти. Если есть возможность выделить пустую страницу, так и делается. Если же все страницы заняты, то ОС выбирает ту, которая, исходя из некоторого критерия, менее всего необходима в данный момент, и выгружает её на диск, а освободившуюся страницу физической памяти отдаёт запросившему её процессу, записывая начальный физический адрес страницы в соответствующую ячейку таблицы страниц. При этом в таблице страниц процесса, чья страница была выгружена, ставится флаг «страница выгружена на диск». Если этот процесс попытается обратиться к выгруженной стра-

нице, то также произойдёт страничный сбой, в результате чего ОС выгрузит на диск одну из страниц, подгрузит на её место нужную и позволит процессу его использовать.

Результатом использования этого механизма является то, что объём памяти, который система может предоставить программам, ограничен только объёмом её дисков и объёмом адресного пространства. При этом одни и те же данные, принадлежащие одному и тому же процессу, в разные моменты времени могут находиться в разных местах физической памяти, хотя их логические адреса всё время остаются одними и теми же.

В принципе, для правильного функционирования ОС безразлично, какую именно страницу выгружать на диск, если свободная память исчерпана. Однако выбор этой страницы влияет на общую производительность системы. Страничный сбой требует достаточно длительной операции — подгрузки страницы с диска. Если на диск попадают страницы, которые используются редко, то страничные сбои происходят также редко, следовательно, система работает эффективно. Если же выгружаются часто используемые страницы, то ОС вынуждена будет часто подгружать память с диска, теряя на этом время. Поэтому алгоритм выбора страницы для вытеснения из ОЗУ стараются выбирать так, чтобы обеспечить наименьшее среднее количество страничных сбоев. Как правило, этот алгоритм основывается на хранении предыстории использования страницы (либо общего числа обращений, либо частоты обращений, либо времени от последнего обращения). На основании этой предыстории выбирается страница, которая, предположительно, не будет затребована дольше всего.

Некоторые программы требуют, чтобы выделяемая им память не выгружалась на диск вообще и всегда находилась по одному и тому же физическому адресу. Примером таких программ являются драйверы устройств. Как правило, в ОС есть средства, которыми можно запретить системе выгружать определённые страницы, но этими средствами лучше не злоупотреблять, поскольку их неправильное использование может ухудшить производительность системы.

2.2.4 Работа программ с памятью.

Причины использования системных средств работы с памятью в программе.

Обычно при создании прикладных программ относительно невысокой сложности программист пользуется средствами работы с памятью, предоставляемыми используемым языком программирования, не задумываясь над тем, каким именно образом эти команды соотносятся с механизмами распределения памяти операционной системы. Но иногда приходится непосредственно использовать средства управления памятью, предоставляемые ОС. Эти средства входят в качестве обязательной составной части в API любой операционной системы.

В каких случаях действительно возникает необходимость в работе с памятью средствами API? Конкретных примеров много, но, в общем, все они характеризуются одной и той же особенностью. Это недостаточность или неэффективность средств управ-

ления памятью, предоставляемых языком программирования. Например, если нужно динамически выделить блок памяти заранее неизвестного размера, а язык программирования не поддерживает такой возможности, то приходится вызывать функции API. Бывают ситуации, когда программист вынужден использовать реализации языков программирования, не вполне эффективно реализующие работу с памятью в том режиме, который требуется в конкретной программе. Наконец, обычно языки программирования в принципе не содержат средств, позволяющих управлять в полном объёме всеми системно-зависимыми особенностями выделения, размещения и освобождения памяти. Например, ни один из языков высокого уровня, реализованных под 32-разрядными версиями Windows, не содержит встроенных в сам язык средств, с помощью которых можно было бы запретить системе выгружать на диск выделенный участок памяти.

Выделение и освобождение памяти.

Любая операционная система содержит, как минимум, следующие функции API, предназначенные для работы с памятью:

- функции выделения памяти;
- функции освобождения ранее выделенной памяти.

Первые обеспечивают выделение области памяти требуемого размера и возвращают указатель на выделенную область. Вторые, как нетрудно догадаться, выполняют обратную операцию — освобождают область памяти, ранее выделенную функцией первой группы. В зависимости от особенностей работы механизма распределения памяти, функций каждого из типов может быть больше или меньше. Типовая схема работы с памятью при использовании этих функций очевидна: блок памяти выделяется одной из функций первой группы, указатель на выделенный блок сохраняется в переменной и используется для доступа к памяти. После завершения использования блока вызывается функция второй группы, в качестве параметра ей передаётся обычно тот адрес, который был получен при выделении памяти. Обычно (хотя, возможно, и не всегда) каждой функции выделения памяти соответствует вполне определённая функция освобождения памяти, и использовать эти функции можно только попарно. Если блок памяти был выделен с помощью одной из функций выделения, то освободить его можно только с помощью соответствующей функции освобождения. Попытка освободить блок с помощью функции освобождения, не соответствующей ранее применённой функции выделения, обычно приводит к ошибке.

Двухэтапное выделение и освобождение памяти.

В некоторых ОС используется двухэтапная схема выделения памяти. В них программа, вызвав функцию выделения памяти, получает не адрес блока, а хэндл — некоторое число, которое идентифицирует выделенный блок. Использовать хэндл для

непосредственного доступа к памяти невозможно. Для того чтобы начать использовать выделенную память, программа должна вызвать функцию блокировки, передав ей в качестве параметра хэндл нужного блока. Функция блокировки возвращает указатель, который уже можно использовать для работы с памятью. После завершения текущей работы с блоком программа может вызвать функцию разблокирования, также передав ей через параметр хэндл блока. Разблокирование не приводит к уничтожению данных, записанных в блок памяти, но делает невозможным доступ к ним до очередной блокировки. Когда программе снова нужно поработать с блоком, она снова его блокирует, работает, и опять разблокирует. Когда необходимость в выделенном блоке исчезает, программа освобождает его, вызвав функцию освобождения памяти. При этом данные, находящиеся в блоке, теряются.

Важнейшим моментом в использовании такой схемы работы с памятью является то, что при каждом блокировании программа может получать различные начальные адреса блока памяти. Выделенный блок памяти имеет фиксированный адрес только в те периоды времени, когда он заблокирован. Как только для блока вызывается функция разблокирования, ОС может переместить его в адресном пространстве, куда пожелает.

Если программа сохранит указатель на блок, который однажды был возвращён функцией блокировки, потом разблокирует память, затем снова вызовет функцию блокировки, то новый адрес вполне может отличаться от старого. Поэтому, если программе, например, нужен доступ к сотому байту блока, то каждый раз, после очередной блокировки, адрес этого байта должен вычисляться по формуле ptr+100, где ptr — адрес начала блока, который вернула функция блокировки в последний раз, когда была вызвана.

Зачем нужна такая усложнённая схема? Главным образом, для повышения эффективности управления памятью. Когда вызывается функция выделения памяти, то система выделяет память. Разумеется, выделенный блок уже имеет вполне определённый адрес, но этот адрес не сообщается программе. Пока блок памяти ещё не заблокирован программой, программа не работает с ним. И если системе для оптимизации памяти в это время потребуется «передвинуть» выделенный блок в адресном пространстве, то она может легко это сделать, не опасаясь за работоспособность программы (потому что программа ещё не получила адрес блока и не может с ним работать). Заблокированный фрагмент уже не может быть перемещён, поскольку изменение его адреса нарушит логику работы программы. После разблокирования фрагмента он не удаляется, но считается, что программа с ним пока не работает, и он снова может быть перемещён в адресном пространстве в случае возникновения такой необходимости. В результате ОС получает возможность передвигать в адресном пространстве выделенные блоки памяти. За этот счёт можно, например, обеспечить частичную дефрагментацию адресного пространства прямо во время работы программы. Можно заметить, что такая схема применяется обычно в ОС, которые не используют механизм страничной организации памяти. В системах, где страничная организация применяется, всё вышеописанное может быть выполнено и без двухэтапной схемы, поскольку там выделенный блок можно как угодно перемещать в физической памяти,

сохраняя у него при этом один и тот же логический адрес.

Управление физическими параметрами выделения памяти.

Как правило, системные функции управления памятью предоставляют возможность задавать некоторые физические параметры выделяемой памяти и режим работы с этой памятью программы и системы.

Прежде всего, в некоторых случаях программа нуждается в выделении памяти по фиксированному физическому адресу. Чаще всего такая необходимость возникает у драйверов устройств, взаимодействующих с компьютером через определённые адреса памяти. Такая возможность обеспечивается обычно специальными функциями (или специальными параметрами функций) API ОС. Программист должен задать физический адрес начала выделяемого блока, а ОС попытается выделить блок требуемого размера, начиная с данного физического адреса. Если это по каким-то причинам невозможно, то функция выделения памяти вернёт сообщение об ошибке. Описанная возможность довольно специфична. Для рядовых прикладных программ её использование не только бесполезно, но и нежелательно, поскольку в таких программах она не даёт никакого практического эффекта, но повышает вероятность отказа в выделении памяти и различных сбоев, связанных с тем, что может оказаться занятым фрагмент ОЗУ, который требуется для нормальной работы какому-либо драйверу устройства.

Если подсистема управления памятью ОС поддерживает работу с виртуальной памятью, то система может, по мере необходимости, целиком или частично, выгружать выделенные программой блоки памяти на диск, восстанавливая их, когда это потребуется. Такой порядок работы подсистемы управления памятью обычен и наиболее оптимален, но в некоторых случаях он бывает неприемлем. В API ОС, использующих виртуальную память, функции выделения памяти, как правило, позволяют программисту задать параметр, определяющий, разрешено ли системе выгружать выделенный этой функцией блок памяти. Таким образом, программист может запретить выгрузку фрагмента, что гарантирует, что при обращении к нему не будет происходить страничных сбоев.

Теоретически, использование невыгружаемых блоков памяти повышает быстродействие программы. На практике дело обстоит гораздо сложнее. С одной стороны, запрет на выгрузку блока уменьшает количество страничных сбоев при обращении к нему до нуля, с другой — он исключает из ведения системы виртуальной памяти некоторый объём памяти, который та могла бы использовать более рационально, останься он под её управлением. Вследствие этого практическое быстродействие программы при использовании невыгружаемой памяти может даже упасть.

Прямая работа с подсистемой выделения виртуальной памяти.

API ОС, использующих виртуальную память, может позволять программе самостоятельно управлять процессом выделения виртуальной памяти. Для этого могут существовать функции, обеспечивающие резервирование региона виртуальной памяти,

выделение региону физической памяти, освобождение региона, блокировки и разблокирования региона. В системах с виртуальной памятью эти функции, собственно, являются базовыми.

Функция резервирования региона виртуальной памяти выполняет первый этап выделения блока памяти — резервирует регион адресного пространства, который будет использоваться программой. Резервирование не выделяет физическую память, а лишь помечает необходимое количество страниц памяти как занятые.

Функция выделения региону физической памяти обеспечивает выделение системой нужного количества страниц физической памяти и запись адресов этих страниц в таблицу страниц процесса. Выделение физической памяти, очевидно, должно произойти до первого обращения к региону. Как правило, если программа обращается к зарезервированному региону виртуальной памяти, не выделив ему физическую память, ничего страшного не случается – происходит обычный страничный сбой, в результате которого ОС совершенно самостоятельно выделит региону физическую память. Зачем, в таком случае, нужна функция выделения памяти? Она позволяет выделить физическую память до начала использования блока, избежав, таким образом, страничного сбоя при первом обращении. В некоторых случаях это бывает важно.

Функция освобождения региона, как ясно из названия, освобождает ранее выделенный регион виртуальной памяти. Если для этого региона были выделены страницы физической памяти, то они также, естественно, освобождаются. Если часть данных, находящихся в этом регионе, была выгружена на диск, то освобождаются и соответствующие элементы файла или области подкачки на диске.

Функция блокировки региона предназначена для того, чтобы запретить выгрузку региона на диск. Блокируют обычно те регионы, которые активно используются на протяжении продолжительного времени, возможно, со значительными перерывами. В таких условиях система виртуальной памяти может счесть, что данный регион (если к нему, например, некоторое время не было обращений) долго не будет использован, в то время как на самом деле следующее обращение к нему произойдёт очень скоро.

Функция разблокирования региона противоположна по содержанию предыдущей – она уведомляет ОС, что для данного региона должны использоваться стандартные правила работы с виртуальной памятью, и ОС может свободно выгружать его, целиком или частично, по мере необходимости.

Необходимо отметить, что многие из функций управления виртуальной памятью являются, строго говоря, не командами, а рекомендациями, в том смысле, что ОС иногда может просто игнорировать их. Например, если подсистема виртуальной памяти пожелает выделить региону физическую память сразу после резервирования самого региона, она сделает это, не дожидаясь, пока программа вызовет функцию выделения памяти.

2.2.5 Настройка подсистемы виртуальной памяти.

Обычно ОС позволяет управлять некоторыми из параметров системы управления виртуальной памятью. Рассмотрим их.

Использование или неиспользование виртуальной памяти.

Некоторые ОС позволяют полностью отключить систему виртуальной памяти [8]. В этом случае объём доступной памяти в системе окажется ограничен объёмом реально установленной оперативной памяти. Виртуальную память имеет смысл отключать, когда совершенно точно известно, что установленной физической памяти наверняка хватит для всех решаемых системой задач. В таких случаях отключение виртуальной памяти приводит обычно к увеличению (причём достаточно заметному) быстродействия системы. При рассмотрении вопроса об отключении виртуальной памяти следует иметь в виду, что системе необходима оперативная память не только для выполняющихся программ, но и для нужд самой ОС, в частности для дискового кэша. Прежде чем отключать виртуальную память, желательно запустить какую-нибудь программу мониторинга использования памяти, способную записывать результаты работы в файл и дать ей поработать, как минимум, несколько суток. Потом нужно проанализировать результаты работы. Если окажется, что на некоторых этапах работы объём используемой пользовательскими и системными программами памяти плюс минимальный размер дискового кэша значительно превышает физический размер ОЗУ, то от выключения системы виртуальной памяти лучше воздержаться.

Расположение выгруженных страниц.

В различных системах применяются различные способы хранения выгруженных страниц виртуальной памяти. В UNIX-подобных системах для этого чаще всего используется специально выделенный раздел на жёстком диске (swap-раздел). Этот раздел имеет собственную структуру и не может использоваться для хранения обычных файлов. Естественно, swap-раздел имеет постоянный размер, изменение которого сопряжено со всеми трудностями, возникающими обычно при изменении разделов на жёстких дисках. Могут использоваться так называемые swap-файлы (файлы подкачки). Файл подкачки хранится как обычный файл в обычной файловой системе. Файл подкачки может быть динамическим (переменного размера) или статическим (постоянного размера). Если файл подкачки имеет постоянный размер, этот размер должен быть задан администратором при настройке системы. Динамический файл подкачки способен самостоятельно увеличиваться при необходимости помещения в него новых страниц и уменьшаться при сокращении объёма выделенной виртуальной памяти. Система может поддерживать использование одновременно нескольких файлов подкачки или даже одновременное использование нескольких разделов подкачки и нескольких файлов подкачки.

Расположение выгруженных страниц в отдельном разделе даёт возможность форматировать этот раздел специальным образом, оптимизируя доступ к выгруженным страницам. А файл подкачки находится в файловой системе, которая оптимизируется для ускорения доступа к обычным файлам. Кроме того, файл подкачки может быть

[8] Важно понимать, что при этом отключается только выгрузка страниц на диск, но сам механизм страничной организации памяти продолжает работать.

фрагментирован (если ОС это допускает), что дополнительно снижает скорость работы с ним. Динамические файлы подкачки системе сложнее поддерживать, чем файлы постоянного размера, поскольку требуется выполнять увеличение и сокращение файла подкачки и анализировать загрузку памяти, на что требуется дополнительное время.

Вследствие сказанного, общим правилом является следующее: при прочих равных условиях размещение выгруженных страниц в отдельном разделе обеспечивает большее быстродействие, чем использование файлов подкачки, а статический файл подкачки даёт большее быстродействие, чем динамический. С другой стороны, при внесении изменений в настройку системы виртуальной памяти swap-раздел труднее изменить, чем файл подкачки. Динамический файл подкачки вообще не потребуется изменять вручную, поскольку он автоматически подстраивается под потребности системы в памяти. Если есть возможность выбора, то следует исходить из того, что в данном конкретном случае более важно. Если важнее быстродействие, то лучше использовать отдельный swap-раздел или статический файл подкачки. Если более важны экономия дискового пространства и удобство настройки, следует выбирать файл подкачки, лучше динамический.

Расположение swap-области или файла (файлов) подкачки на физических дисках.

При активном использовании виртуальной памяти обращения к выгруженным страницам могут происходить достаточно часто. Если, как это обычно бывает, программы активно работают с данными, находящимися на дисках, то ОС приходится попеременно обращаться то к области или файлу подкачки, то к файлам с программами и данными. Если выгруженные страницы находятся на том же физическом диске, что и используемые системой и работающими программами файлы, то попеременное обращение то к одним, то к другим областям диска приводит к постоянному перемещению головки диска, что существенно уменьшает скорость чтения и записи данных. Если дисковая подсистема имеет собственные аппаратные средства кэширования данных, то их эффективность в таком режиме работы также снижается.

Если компьютер имеет только один физический диск, с таким положением приходится мириться. Единственное, что можно сделать в такой ситуации для ускорения работы — расположить swap-область или файл подкачки в начале диска (поскольку время доступа к секторам, расположенным в начале диска, существенно меньше времени доступа к секторам, находящимся в конце). Если используется файл подкачки, крайне желательно сделать его нефрагментированным.

Если же имеется несколько жёстких дисков, то следует расположить swap-раздел или файл подкачки либо вообще на отдельном физическом диске, либо на диске, к которому в процессе работы система и программы обращаются реже всего. При работе с программами, активно использующими диск, (например, с мощными СУБД) это приведёт к заметному увеличению скорости работы. Вообще говоря, эта рекомендация является частным случаем общего правила, согласно которому данные, используемые одновременно, лучше располагать на различных физических носителях.

Особенно ярко эффект от правильного расположения данных и файлов подкачки или swap-областей проявляется при использовании современных дисковых накопителей, имеющих внутреннюю кэш-память большого объёма и способных выполнять операции без постоянного контроля со стороны процессора.

2.3 Файловая система.

2.3.1 Файлы и файловая система.

Данные и программы, с которыми работают пользователи компьютерной системы, только во время работы с ними оказываются в оперативной памяти компьютера. Всё остальное время они находятся на внешних носителях.

В качестве внешних носителей, применяемых для постоянного хранения информации, в разное время использовались различные устройства. Первоначально единственными носителями информации были перфоленты и перфокарты. Очевидные недостатки таких носителей вынудили отказаться от них и перейти к другим типам носителей, главным образом, работающим по принципу сохранения информации в виде неоднородной намагниченности магнитного материала. Исторически первыми в этом ряду были магнитные ленты и магнитные барабаны, позже вошли в употребление магнитные диски. В настоящее время также весьма распространены оптические и магнитооптические носители информации.

С момента начала использования магнитных лент и по настоящее время информация хранится на внешних носителях в виде так называемых файлов (files). *Файлом называется поименованный участок памяти на внешнем носителе, не имеющий фиксированного размера и хранящий некоторый блок информации, представляющий собой в каком-либо смысле единое целое.* Уточним это определение.

Файл — это *поименованный участок памяти* на внешнем носителе. Это означает, что у этого участка памяти имеется некоторое фиксированное имя, конкретный порядок построения которого определяется правилами, существующими в ОС.

Файл — это участок внешней памяти, который *не имеет фиксированного размера*. Это означает, что ОС не задаёт для файлов конкретного размера, а позволяет создавать файлы, размер которых соответствует объёму хранимого блока информации. Размер файла может изменяться на протяжении времени его существования. Из соображений удобства реализации операционная система , как правило, разделяет внешний носитель на участки относительно малого объёма и выделяет память файлу этими участками. Так, у дисковых накопителей такой участок - один сектор дорожки диска. Вследствие этого объём физического пространства, занимаемого файлом, кратен размеру этого наименьшего участка. Несмотря на это, логический размер файла (тот размер, который ОС сообщает пользователю и программам, работающим с фалом) в точности соответствует размеру хранимого в файле блока данных.

Файл *хранит блок информации, представляющий собой в каком-либо смысле единое целое*. Это означает, что файл содержит, как правило, некоторый целостный

набор данных, обрабатываемых совместно, обычно одними и теми же средствами. Кроме того, в виде файлов в системе хранятся программы и элементы самой ОС.

Естественно, что технические принципы поддержки хранения файлов зависят от типа носителя (ясно, что на магнитном диске и на CD-ROM файлы должны храниться по-разному, уже хотя бы потому, что физические принципы чтения и записи данных для этих видов носителей существенно различаются). Даже на носителях одного и того же типа можно организовать хранение файлов различным образом.

Комплекс правил, требований и соглашений, в соответствии с которым хранятся на носителе файлы, называется файловой системой (далее в некоторых случаях будет использоваться сокращение ФС). Кроме того, термин «файловая система» может использоваться в смысле «весь набор данных на носителе (носителях), размещённый там в соответствии с правилами, заданными для ФС такого типа». Помимо дисковых файловых систем, обеспечивающих хранение данных на магнитных дисках различной конструкции и ёмкости, существуют также другие файловые системы, например, сетевые. Сетевая файловая система обеспечивает имитацию носителя, содержащего данные, фактически хранящиеся на физических носителях другого компьютера, соединённого с первым с помощью того или иного коммуникационного оборудования. Здесь обсуждаются только дисковые файловые системы.

Любая файловая система имеет две стороны, или, если угодно, два уровня организации. Первый уровень, который можно назвать «логическим», или «внешним», определяет логику работы файловой системы «извне», с точки зрения пользователей и прикладных программ. Он задаёт способы обращения к файлам, правила именования файлов, набор и способы задания их атрибутов, принципы группировки файлов. Второй уровень, который можно определить как «физический» или «внутренний» задаёт технические особенности хранения файлов на носителях и алгоритмы исполнения операций с файлами, которые могут быть затребованы программами.

Дисковых файловых систем существует довольно много — только самых распространённых насчитывается несколько десятков. Практически каждая операционная система использует для хранения файлов на магнитных дисках свою собственную файловую систему (или собственную разновидность одной из распространённых ФС). Так, MS-DOS поддерживает файловую систему FAT, Windows 95/98 — VFAT (это та же FAT, только дополненная для обеспечения хранения длинных имён файлов и каталогов), Windows NT — NTFS, OS/2 — HPFS, Linux — ext2fs (в последнее время ещё ReiserFS и ext3fs), практически каждая реализация ОС UNIX поддерживает свой собственный тип ФС. При этом большинство систем, помимо своей «родной» файловой системы, поддерживают ещё несколько (а то и несколько десятков) «чужих». Существуют также файловые системы, поддерживаемые всеми (во всяком случае, большинством) ОС. Это, например, ISO9660 — файловая система стандартных CD-ROM дисков. Все ОС, работающие на платформе IBM PC (Intel x86), поддерживают файловую систему FAT16.

Как это ни странно, при таком многообразии файловых систем реальные отличия между ними не особенно велики, зачастую различия заключатся лишь в мелких деталях.

2.3.2 Логическая структура файловой системы.

Атрибуты файлов.

Для любого файла в файловой системе, помимо имени и данных, обязательно хранится набор дополнительных сведений (атрибутов). К ним относятся размер, дата и время последнего изменения, дата и время создания, дата и время последнего обращения (она фиксирует дату и время последнего открытия файла какой-либо программой), права доступа. Могут храниться и другие атрибуты. Конкретный набор атрибутов определяется набором возможностей, поддерживаемых ФС. Строго говоря, имя файла — это тоже лишь один из его атрибутов.

Так, скажем, в файловой системе FAT, помимо имени и времени создания, для каждого файла хранятся атрибуты «скрытый», «системный», «только для чтения» и «архивирован». [9] Предполагается, что файлы с атрибутом «скрытый» не будут видны пользователю (они не выводятся в списке файлов по стандартной команде dir), что исключит возможность их случайного изменения. Атрибут «системный» предназначен для файлов ОС, атрибут «только для чтения» исключает возможность изменения содержимого файла, а атрибут «архивирован», по мысли разработчиков, должен устанавливаться программой архивации содержимого диска, чтобы при последующих архивациях можно было не добавлять в архив ранее уже сохранённые файлы. [10] Никаких атрибутов, связанных с правами пользователей на файл, нет, поскольку ОС DOS, для которой предназначалась данная ФС, является однопользовательской и проблема разграничения прав различных пользователей на одни и те же файлы там просто не существует.

В файловых системах, предназначенных для ОС семейства UNIX и подобных им (например, ФС ext2, ReiserFS) отсутствуют такие понятия как «скрытый» или «системный» файл. Зато для каждого файла хранится набор прав, позволяющий определить, кому данный файл принадлежит и выяснить, для каждого конкретного пользователя, разрешён ли ему доступ к файлу на чтение, изменение файла и его запуск (последнее, конечно, касается только файлов, содержащих программы).

Система каталогов.

Как правило, число файлов, хранящихся на магнитном диске современного компьютера, исчисляется тысячами и десятками тысяч, а то и более. Ясно, что без определённой дисциплины в именовании файлов и создания специальной системы их классификации разобраться в таком количестве файлов практически невозможно. По этой причине большинство файловых систем поддерживают концепцию каталогов (directory). Каждый файл, находящийся на диске, относится к некоторому вполне определённому каталогу, причём файл обязательно относится к какому-либо каталогу и не может

[9] На самом деле, в FAT больше атрибутов файлов, но здесь описываются только указанные четыре, поскольку только они могут быть нормальным образом изменены и проверены пользователем.

[10] Мне неизвестно современное и сколько-нибудь широко используемое программное обеспечение, которое бы в действительности использовало атрибут архивации.

одновременно относиться к нескольким. ФС обеспечивает хранение списка каталогов и связь между файлами и их каталогами. Как правило, сами каталоги хранятся в системе как файлы специального вида. Каталог может содержать в себе другие каталоги (подкаталоги).

В файловой системе обязательно имеется так называемый корневой каталог, в котором находятся некоторые файлы и подкаталоги. В этих подкаталогах могут также находиться файлы и подкаталоги следующего уровня. Таким образом, образуется так называемое дерево каталогов [11]. Корнем этого дерева является корневой каталог, узлами — подкаталоги, листьями — файлы.

Дерево каталогов позволяет распределять файлы по каталогам в соответствии со смыслом хранящейся в них информации и по другим, выбираемым пользователем, критериям. При этом пользователь получает возможность размещать файлы по каталогам так, чтобы в одном каталоге не было слишком много файлов и все файлы, имеющие что-то общее, хранились в одном каталоге или в подкаталогах одного каталога.

В такой системе для указания файла недостаточно только его имени, поскольку в различных каталогах могут находиться файлы с одинаковыми именами. Поэтому файл однозначно указывается с помощью полного пути, представляющего собой совокупность имени диска (если оно используется), на котором находится файл, списка каталогов, который необходимо пройти от корневого каталога до каталога, содержащего файл, включая и сам этот каталог, и собственно имени файла.

Ссылки.

Иногда удобно было бы сделать так, чтобы один и тот же файл или каталог находился одновременно более чем в одном каталоге или имел более чем одно имя. Но такая возможность нарушила бы логику организации структуры каталогов, что недопустимо. Тем не менее, некоторые файловые системы позволяют сделать это, но не «в лоб», а путём создания так называемых *ссылок*.

Ссылка (link) — это элемент файловой системы, выполняющий функции псевдонима для какого-либо файла или каталога.

Ссылка располагается в каком-то из каталогов файловой системы, как обычный файл, имеет имя, сформированное по тем же правилам, что и имена файлов, может использоваться программами как файл. [12] Использование ссылок позволяет создать видимость того, что один и тот же файл одновременно располагается в нескольких различных местах дерева каталогов. Для этого достаточно расположить файл в одном

[11] Название вызвано полным соответствием между структурой каталогов и направленным графом специального вида, который в теории графов именуется деревом.

[12] То есть программа может выполнить по отношению к ссылке операции «открытие файла», «чтение», «запись», «закрытие», «удаление» (разумеется, если у неё есть соответствующие права на файл, на который указывает ссылка), хотя выполнение некоторых из этих операций в применении к ссылкам имеет немного другую семантику.

из каталогов, а во всех остальных каталогах, где требуется его наличие, разместить ссылки на него. Обращение к ссылке на файл приводит к обращению к объекту, на который эта ссылка указывает. Это бывает удобно тогда, когда в системе одновременно используется несколько программ, работающих с одним и тем же файлом, но требующих наличия его в различных каталогах и, возможно, под различными именами. [13]

Ссылки могут быть двух видов — символические и прямые (в англоязычной литературе для первых используется термин «symbolic link», для вторых – «hard link»). Символическая ссылка содежит символьное имя файла или каталога, и при работе программ с этой ссылкой обращение происходит к объекту файловой системы, имя которого хранит ссылка. Символическая ссылка может использоваться для работы с этим файлом или каталогом, но через неё невозможно выполнить некоторые операции, например, удаление объекта, на который она указывает. Удаление самого объекта (файла или каталога), на который указывает символическая ссылка, приводит к тому, что любая символическая ссылка на него становится некорректной (фактически указывает в никуда). Попытки дальнейшей работы с этой ссылкой приведут только к получению сообщений об ошибках. Если же указываемый объект будет снова создан, то символические ссылки автоматически станут корректными и их можно будет использовать дальше. Применение операции удаления к символической ссылке приводит к удалению ссылки, но никак не отражается на объекте, на который она указывает. Физически символическая ссылка обычно реализуется как отдельный файл специального вида.

Прямая ссылка — это ещё одно полноправное имя объекта файловой системы. Для лучшего уяснения сути понятия прямой ссылки необходимо понять, что файл и его имя — это не одно и то же. При создании нового файла в файловой системе размещается сам файл (то есть блок данных на диске) и регистрируется имя файла, которое являетя первой прямой ссылкой на этот файл. При создании новой прямой ссылки в файловой системе регистрируется ещё одно имя для того же самого файла, имеющее все те же особенности, что и первое имя (под которым файл был создан). В результате в файловой системе появляется файл, который имеет несколько имён и, возможно, зарегистрирован в нескольких каталогах. С точки зрения пользователя и программ ситуация выглядит так, как будто в файловой системе есть несколько различных файлов, обладающих тем свойством, что изменения в одном из них тут же отражаются в других. Все прямые ссылки на файл равноправны в том смысле, что ни одна из них не занимает какого-то исключительного положения и не даёт каких-то возможностей для работы с файлом, которые недоступны через другие ссылки.

Через прямую ссылку можно работать с файлом, то есть открывать его, записывать данные, читать, переименовывать (в последнем случае переименованию подвергнется только данная прямая ссылка). Удаление прямой ссылки не уничтожает объект, на ко-

[13] Кстати, во избежание путинацы необходимо заметить: известные пользователям Windows так называемые «ярлыки» (shortcuts), ссылками не являются. Они не дают доступа к файлу, на который указывают. С их помощью можно только запустить то или иное приложение.

торый она ссылается (то есть сам файл или каталог), непосредственно, но как только в файловой системе будут удалены все прямые ссылки на объект, система уничтожит сам этот объект. Вследствие этой особенности прямые ссылки никогда не бывают некорректными (по крайней мере, если нет нарушений в структуре файловой системы). Использование прямых ссылок ограничивается тем, что прямая ссылка может располагаться только в пределах того физического раздела диска, в котором фактически хранится сам файл (или каталог), на который она ссылается. Для символических ссылок таких ограничений нет.

Поддержка нескольких физических устройств.

Как правило, компьютерная система, в которой работает ОС, содержит более одного устройства для хранения файлов. Как минимум, система должна содержать одно устройство хранения информации на сменных носителях (в персональном компьютере это обычно флоппи-диск или zip-накопитель) и одно устройство хранения информации большого объёма с малым временем доступа (обычно жёсткий диск или один из разделов такого диска). Кроме того, обычно имеются другие устройства (CD-ROM, стример). Устройством хранения файлов можно также считать сетевое подключение к другому компьютеру, через которое можно читать и/или записывать файлы. Возможно (и нередко встречается) присутствие в системе нескольких устройств одного типа.

Существует два подхода к организации файловой системы, данные в которой хранятся на различных устройствах. Первый подход, реализованный, например, в ОС DOS, Windows 95/98, Windows NT, заключается в поддержке концепции *логических устройств*. Каждое отдельное физическое устройство или специальным образом организованная его часть может быть логическим устройством. Как правило, каждое отдельное физическое устройство обязательно является отдельным логическим устройством (хотя в некоторых системах несколько различных физических устройств могут быть объединены в одно логическое). Каждое логическое устройство имеет собственное имя (в ОС фирмы Microsoft это буква). Для каждого логического устройства файловая система содержит отдельное дерево каталогов. Дерево каталогов каждого логического устройства является совершенно самостоятельным. Деревья каталогов двух различных логических устройств не могут являться частями одного и того же общего дерева каталогов. Каждый вновь создаваемый файл сразу же приписывается к определённому логическому устройству (естественно, к тому, на котором размещается сам этот файл). В таких системах полный путь к файлу обычно начинается с имени логического устройства, за которым следует список каталогов, разделённых между собой прямым или обратным слэшем; после списка каталогов следует собственно имя файла.

Описанный порядок фактически отображает физическую структуру устройств хранения данных системы на структуру системы логических устройств. Несмотря на внешнюю простоту и логичность, этот порядок имеет серьёзный недостаток. Дело в том, что привязка логической структуры файловой системы к физическому набору

устройств делает менее гибким управление ресурсами системы. Приведём простейший пример. Пусть нам необходимо установить некоторый программный продукт. Наша система имеет два жёстких диска: один — объёмом 2 Гбайт, другой — объёмом 1 Гбайт. На первом диске 800 Мбайт уже занято ОС и другими программами. Пусть устанавливаемому пакету необходимо 1500 Мбайт дискового пространства. Сможем ли мы установить этот пакет? Судя по объёму незанятого дискового пространства в системе, сможем (свободно в общей сложности более 2 Гбайт). Но пакет (как это обычно бывает) устанавливается в один каталог (в котором он создаёт подкаталоги). Поскольку содержимое одного каталога не может находиться на разных логических устройствах, нам придётся выбирать, на какой из дисков устанавливать пакет. На первом диске свободно около 1200 Кбайт, что недостаточно, на втором — 1 Гбайт, что также мало. В результате оказывается, что мы не можем установить пакет в системе, хотя свободного дискового пространства достаточно. Чтобы всё-таки выполнить установку, нам придётся переустановить часть уже установленного программного обеспечения так, чтобы освободить на первом диске хотя бы 1500 Кбайт [14].

Второй вариант организации работы файловой системы с несколькими устройствами реализован, в частности, в ОС семейства UNIX. Файловая система имеет только одно дерево каталогов. Чтобы устройство хранения файлов могло быть использовано, оно должно быть смонтировано. При монтировании устройству приписывается некоторая точка монтирования — путь к каталогу на дереве каталогов системы, который будет корневым каталогом для всех файлов на данном устройстве. Файл в файловой системе физически находится на устройстве, точка монтирования которого расположена ближе всего на пути от этого файла к корневому каталогу. Подобным образом монтируются не только жёсткие диски, но и любые другие устройства.

Например, один из двух дисков системы может быть смонтирован с точкой монтирования «/» (корневой каталог), второй диск — с точкой монтирования «/usr/», флоппи-диск — с точкой монтирования «/floppy», накопитель CD-ROM — с точкой монтирования «/cd». Тогда для доступа к дискете в дисководе нужно работать с каталогом «/floppy», для чтения CD-ROM — с каталогом «/cd». Прочие файлы, путь к которым начинается с «/usr», будут физически помещены на второй диск, а все остальные файлы – на первый.

В приведённом ранее примере с инсталляцией большого пакета в системе с двумя дисками можно просто разбить второй диск на разделы так, чтобы размеры разделов совпадали с объёмами, необходимыми для нескольких подкаталогов, используемых устанавливаемым пакетом, и смонтировать каждый из созданных разделов с точками монтирования, совпадающими с именами соответствующих каталогов. После этого можно устанавливать пакет. Часть файлов пакета окажется на одном физическом диске, часть – на другом, но с точки зрения файловой системы все они будут находиться в подкаталогах одного и того же каталога, поэтому никаких проблем с установкой не

[14] На самом деле, есть ещё один вариант — можно записать часть файлов пакета на другое устройство, а вместо реальных файлов и каталогов разместить на диске, куда устанавливается пакет, символические ссылки.

будет, и переустанавливать прочее программное обеспечение не потребуется. Хотя, разумеется, проблема всё же возникнет, если хотя бы один из подкаталогов должен содержать не разбитый на вложенные подкаталоги набор файлов, общий объём которого превышает свободное пространство на одном из дисков. Но такая ситуация достаточно редка. Кроме того, в последнее время для некоторых ОС появляются средства организации файловой системы, позволяющие хранить на физически разных устройствах даже файлы, расположенные в одном каталоге.

Возможно совмещение обоих подходов к организации файловой системы. В этом случае физическое устройство, в зависимости от необходимости, может быть либо включено в файловую систему либо как отдельное логическое устройство, либо смонтировано в точку монтирования на другом логическом устройстве.

Именование файлов и каталогов.

Порядок именования файлов и каталогов и возможность использования в именах тех или иных символов определяется ОС. Как правило, в именах файлов допускается использование латинских букв, цифр, знака подчёркивания и ряда других символов, которые не используются в ОС как специальные. Обычно ОС накладывают ограничения на длину имён, хотя часто максимальная длина имени настолько велика, что это ограничение не проявляется. Строчные и прописные буквы могут различаться, а могут и не различаться. Использование букв других алфавитов (в частности, русского) возможно далеко не всегда.

Обычно имя состоит как минимум из двух частей — собственно имени и расширения, разделённых точкой. Во многих системах (UNIX, Windows NT) имя может быть набором нескольких групп допустимых символов, разделённых точками. В таких случаях обычно расширением считается последняя из этих групп. Практическим стандартом стал порядок, при котором расширение указывает на тип файла (точнее, на тип и формат содержащихся в нём данных). Обычно файлы, содержащие исполнимые программы, имеют фиксированные расширения. Для прочих файлов пользователь может выбирать расширения сам, но при этом рекомендуется придерживаться использования традиционных для данного типа файлов расширений. Так, текстовые файлы обычно имеют расширение txt, и использование другого расширения просто введёт в заблуждение и самого пользователя, и тех, кому он, возможно, тем или иным образом передаст этот файл. Большинство программ, сохраняющих обрабатываемые данные в файлах, автоматически приписывают к указанному пользователем имени файла расширение, которое позволит программе впоследствии опознать этот файл как «свой». Во многих ОС (в частности, Windows 95, Windows NT, OS/2) программа обработки файла автоматически выбирается ОС по расширению этого файла при попытке пользователя запустить его.

Кэширование.

Для оптимизации работы с диском большинство ОС используют так называемое кэширование. Суть его заключается в том, что часть данных, хранимых на диске, во время работы системы фактически располагается в оперативной памяти компьютера. Область памяти, предназначенная для хранения таких данных, называется кэш-памятью. Объём кэш-памяти определяется ОС и может ограничиваться настройками, заданными пользователем. Система кэширования выполняет (может выполнять) следующие действия.

- Хранение ранее прочитанных данных. Данные, прочитанные из файла, сохраняются в кэш-памяти до тех пор, пока позволяет её объём. При повторном обращении программ к тем же самым данным производится не повторное чтение с диска, а извлечение ранее прочитанных данных из кэш-памяти. В результате при многократных обращениях к одним и тем же данным физически считывание выполняется только один раз. Разумеется, по мере выполнения всё новых и новых операций чтения с диска кэш-память заполняется прочитанными данными, и наиболее старые фрагменты заменяются новыми, так что при активной работе с диском и достаточно больших интервалах между операциями чтения повторное считывание данных с диска всё-таки приходится производить.

- Опережающее чтение. Часто программа выполняет чтение данных с диска последовательно, относительно небольшими порциями. Система кэширования может попытаться спрогнозировать, какие данные потребуется прочитать далее, и выполнить их чтение с диска и загрузку в кэш-память, не дожидаясь поступления команды от программы. Например, при попытке программы прочитать один сектор диска может быть считана и помещена в кэш сразу целая дорожка. Если программа читает файл последовательно и файл не фрагментирован (то есть располагается на диске в следующих друг за другом секторах), то при последующих обращениях к файлу требуемые данные окажутся уже прочитанными и будут передаваться программе из кэша. Это экономит время, поскольку избавляет от необходимости при каждом чтении данных выполнять полный цикл операций обращения к диску и снижает количество перемещений головок диска.

- Отложенная запись. При попытке программы записать данные в файл система может не выполнять команду записи на диск немедленно, а запомнить записываемые данные в кэш-области ОЗУ. Физическая запись на диск производится системой позже, тогда, когда будет исчерпана кэш-память, либо тогда, когда загрузка системы упадёт и ОС сможет выделить время для записи данных, не отнимая его у работающих программ.

- Хранение информации о текущем состоянии файловой системы. В кэш-памяти могут храниться данные о физическом расположении всех открытых в данный момент файлов. Кроме того, там могут храниться наиболее часто используемые

каталоги и данные о свободных на текущий момент областях на диске (эти данные используются в тех случаях, когда система создаёт новый файл или увеличивает в размере уже существующий), а также любая другая служебная информация, относящаяся к работе файловой системы. Всё это позволяет при обращении к файлам не выполнять повторного считывания системных областей диска.

Кэширование ускоряет работу с файлами, но создаёт и некоторые дополнительные проблемы. Во-первых, для кэширования необходим достаточно большой объём оперативной памяти под кэш, иначе оно не будет достаточно эффективным. Кроме того, при внезапном выключении компьютера (например, при обрыве питающей сети) данные из кэша пропадут. Если кэш используется только для хранения данных, уже имеющихся на диске, то ничего страшного, естественно, не произойдёт. Однако, если система использует отложенную запись, то та часть данных, которая была предназначена для отложенной записи и ещё не была сброшена на диск, окажется утерянной.

ОС правило, совершенно самостоятельно управляют процессом кэширования дисков. В зависимости от того, сколько оперативной памяти требуется работающим программам и сколько физической памяти имеется в системе, ОС выделяет определённую часть ОЗУ под дисковый кэш, увеличивая объём кэша при уменьшении загрузки памяти и уменьшая в противном случае. Тем не менее, в некоторых случаях возникает необходимость изменить настройки кэширования, выполненные системой.

Некоторые системы предоставляют возможность задать либо точный размер кэша, либо границы, в которых система может его изменять. Ограничение максимального размера кэша исключает его разрастание при освобождении памяти и, соответственно, его уменьшение при запуске программ, требующих памяти. В некоторых случаях это позволяет сократить время загрузки и запуска программ. Ограничение минимального размера кэша позволяет избежать ситуации, когда при большой загрузке компьютера с малым объёмом ОЗУ кэш уменьшается настолько, что его использование лишь снижает скорость дисковых операций.

Известно, что эффективность кэширования при неизменных условиях применения компьютера можно представить зависимостью, приведённой на рис. 2.6.

Здесь по горизонтальной оси откладывается объём кэш-памяти, а по вертикальной – вызванный кэшированием *прирост* скорости дисковых операций. Ясно, что зависимость является чисто качественной — она отражает некоторые средние условия работы некоторого среднего компьютера, поэтому на осях и нет конкретных значений объёма и производительности.

На графике можно выделить четыре части.

Часть от 0 до 1 — увеличение быстродействия при росте размера кэша от нуля до некоторого минимального значения. Обычно такой участок довольно мал. На нём быстродействие растёт быстрее, чем объём кэша. Связан этот рост с тем, что в практически любой системе имеется некоторое, обычно не слишком большое, количество данных, которые постоянно повторно считываются с диска. К ним, например, отно-

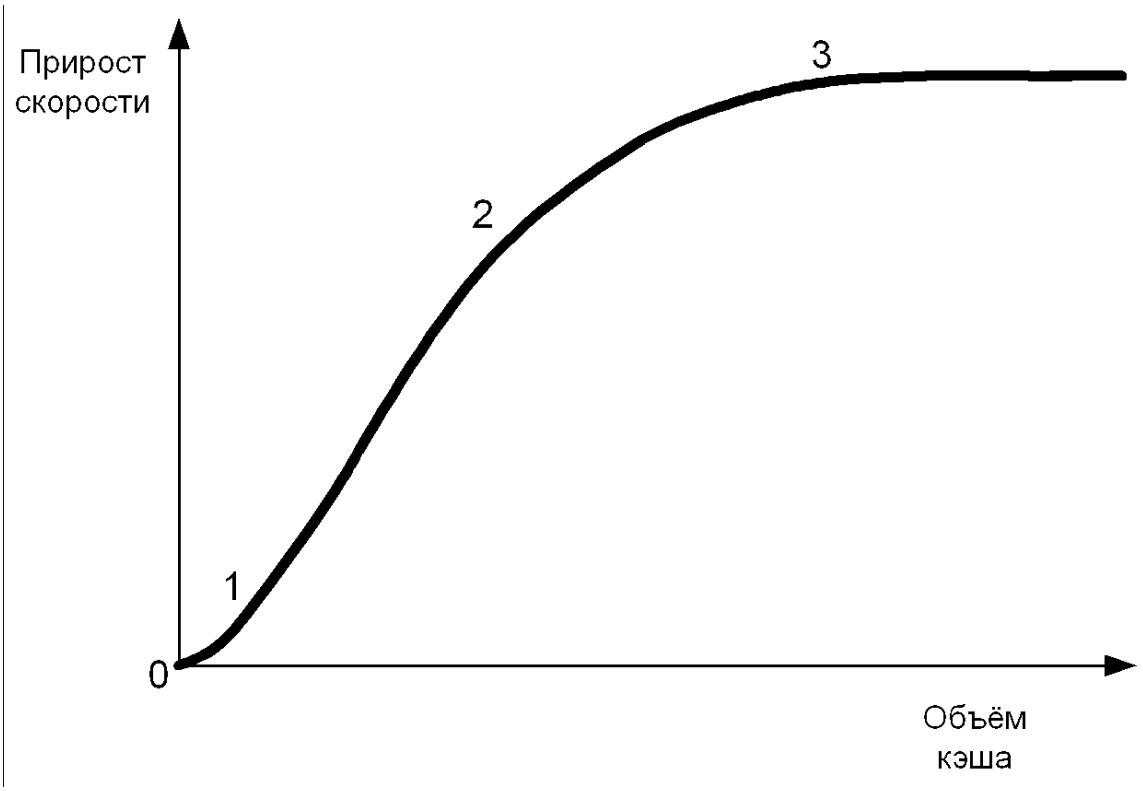

Рис. 2.6: Эффективность кэширования.

сится содержимое каталогов и файловых таблиц дисков. Перемещение таких данных в память очень серьёзно увеличивает скорость всех дисковых операций.

Далее идёт участок (1-2), где прирост скорости дисковых операций зависит от объёма кэша почти линейно. Здесь наиболее активно используемые системные данные уже находятся в кэше, и рост быстродействия определяется тем, какая доля данных, читаемых прикладными программами с дисков, может быть помещена в кэш. Кроме того, увеличение объёма кэша увеличивает эффективность опережающего чтения и отложенной записи, что также вносит свой вклад в общий рост скорости.

Затем, на участке (2-3) увеличение кэша всё ещё увеличивает скорость дисковых операций, но уже медленнее, причём, чем больше возрастает объём кэша, тем меньше дополнительный прирост скорости. На этом участке большинство данных, кэширование которых могло повысить скорость работы, уже находятся в кэше. В кэш-памяти увеличивается доля бесполезных данных, которые никогда не будут повторно использованы. Увеличение скорости происходит практически только за счёт того, что больше полезных данных считываются при опережающем чтении и больше данных можно хранить для отложенной записи.

Наконец, участок, следующий за точкой 3 — практически горизонтальный. На нём увеличение кэша уже не влияет на скорость. Этот участок достигается тогда, когда в кэш-памяти могут поместиться все данные, которые нужны для работы программам и операционной системе. В предельном случае (когда количество физической оперативной памяти не ограничено) при увеличении размера кэша в какой-то момент времени дисковые операции просто прекратятся, так как все необходимые данные будут находиться в кэше. Подобную картину можно наблюдать в нынешних ОС для персональных компьютеров при установке гигабайта или более физической памяти. ОС MS-DOS переходила в такое состояние уже при 16-32 Мбайт ОЗУ.

Нужно заметить, что в реальности, когда объём ОЗУ ограничен, общая производительность системы при увеличении объёма кэша не растёт такими же темпами, поскольку память под кэш выделяется за счёт программ. Увеличение объёма кэш-памяти приводит к тому, что большее количество страниц памяти, выделенных прикладным программам, вытесняется на диск. В результате при обращении программ в память происходит больше страничных сбоев, а каждый из них требует считывания данных с диска, что тормозит исполнение программ. Поэтому выделение под кэш чрезмерно большого объёма памяти лишь уменьшает производительность.

Оптимальный размер кэша находится где-то в пределах участка (2-3), если у системы достаточно оперативной памяти, чтобы такой объём кэша ещё не оказывал заметного влияния на производительность программ. Если же памяти мало, то падение производительности программ из-за её нехватки может начаться раньше. В этом случае как раз и может оказаться уместным ограничение объёма кэша. В любом случае, необходимо поэкспериментировать с различными объёмами кэша и выбрать тот, который при типичной загрузке системы обеспечивает наибольшую производительность.

Виртуальные диски.

При наличии в системе достаточно большого объёма свободной оперативной памяти и необходимости иметь диск с очень высокой скоростью доступа, некоторые ОС обеспечивают возможность создания так называемых виртуальных дисков. Виртуальный диск — это участок ОЗУ системы, который с помощью специального драйвера организован так, что программы и пользователь воспринимают его как ещё один, дополнительный логический диск. Работа с виртуальным диском ничем не отличается от работы с физическим диском, но данные на виртуальном диске могут храниться только до выключения питания компьютера, а скорость доступа к этому диску на два-три порядка выше, чем скорость доступа к информации на физических дисках. В последнее время виртуальные диски практически не применяются, поскольку гораздо более эффективным стало использование ОЗУ большого объёма другими средствами, хотя ещё относительно недавно их использование было весьма распространено.

2.3.3 Физическая структура файловой системы.

Разделы на дисках.

Физические диски, установленные в компьютере, должны быть разделены на разделы. Раздел (partition) — это непрерывный участок диска, который может рассматриваться системой как отдельное физическое устройство. Максимальное количество разделов на одном диске, минимальные и максимальные размеры разделов определяются особенностями конкретной аппаратной платформы и используемой ОС. Диск разбивается на разделы с помощью специальных программ. Каждый раздел должен быть отформатирован. Весь раздел форматируется целиком и содержит после форматирования одну файловую систему.

Разделение дисков на разделы может быть вызвано одной из следующих причин.

- Использование файловой системы, не поддерживающей разделы большого размера. В этом случае диск разбивается на разделы для того, чтобы каждый из разделов мог быть отформатирован под требуемую файловую систему.

- Требования разграничения доступа к дисковому пространству. Если ОС не позволяет выделять пользователям квоты дискового пространства, то единственный способ ограничить объём, занимаемый файлами пользователя – это выделить пользователю отдельный раздел на диске. Тогда он в принципе не сможет превысить размеры этого раздела.

- Соображения эффективности хранения и доступа. Как правило, чем меньше раздел, тем быстрее в пределах этого раздела осуществляется доступ к данным. С другой стороны, если используемые одновременно файлы не помещаются в пределах одного раздела, то вместо ускорения работы будет, вероятнее всего, обеспечено её замедление.

Элементы ФС и их размеры.

Минимальным элементом файловой системы является сектор. Весь диск разделён на сектора, имеющие равные размеры. Файл на диске всегда занимает целое число секторов. Естественно, обычно размер файла несколько меньше, чем суммарный объём секторов, в которых он находится (за счёт того, что последний сектор файла обычно заполнен не до конца). Результатом такого способа организации является то, что чем больше размер сектора, тем больший объём дискового пространства расходуется непроизводительно. Например, если в разделе с размером сектора 512 байт находятся файлы, размер которых равен 1024 байта, то максимальное количество файлов в разделе будет равно количеству секторов, делённому на два (разумеется, в реальности часть пространства будет занята служебной информацией), причём весь объём диска будет занят полезной информацией. Если те же самые файлы будут помещены в раздел, где размер сектора — 8196 байт, то файлов можно будет записать столько же, сколько и секторов (потому что один сектор не может использоваться разными файлами), при этом семь восьмых дискового пространства останутся пустыми, но не будут использованы. Каждый сектор будет содержать файл малого размера, а незанятый "хвост"сектора останется незаполненным.

Вследствие сказанного, с точки зрения максимального использования дискового пространства выгодно создавать размеры с минимально возможным размером сектора. Но, с другой стороны, чем меньше размер сектора, тем больше количество секторов в разделе и, соответственно, тем больше размер служебных областей, хранящих таблицы размещения файлов. Кроме того, увеличение количества секторов увеличивает время поиска всех секторов, относящихся к требуемому файлу. Следовательно, уменьшение размера сектора приводит к падению производительности.

Если при форматировании раздела есть возможность выбора размера сектора, следует выбирать значение, наиболее приемлемое с точки зрения требуемого характера хранимой информации и режима её использования. Если в создаваемом разделе будет храниться большое количество малых по размеру файлов, то такой раздел лучше отформатировать с минимально возможным размером сектора. Если же в разделе будут храниться файлы исключительно большого объёма, то выгоднее выбрать максимальный размер сектора, поскольку потери будут невелики (если файлы большие, их относительно немного, а значит, немного и незанятых «хвостов»), а скорость доступа увеличится.

Необходимо отметить, что в некоторых из появившихся недавно ФС в одном секторе могут содержаться данные нескольких файлов, что делает неактуальной вышеописанную проблему. Такова, например, ФС ReiserFS, которая может использоваться сейчас на компьютерах, работающих под управлением ОС Linux.

Данные и метаданные.

Любая файловая система хранит два вида информации — данные и метаданные. Данные — это собственно информация пользователя, программ и операционной системы.

Метаданные — это «информация об информации» — сведения о расположении данных на диске и о логической структуре файловой системы. В более принятых в настоящее время терминах, данные – это содержимое файлов, а метаданные — это содержимое системных областей диска, то есть атрибуты файлов, структура каталогов, файловые таблицы, индексы, дескрипторы, в общем, вся служебная информация, которая описывает файлы, позволяет выделить цепочки секторов, содержащие нужный файл и организует множество файлов в единую систему — дерево каталогов.

Механизм хранения собственно данных в большинстве дисковых файловых систем совершенно одинаков: данные хранятся в наборе секторов диска. Некоторые файловые системы обеспечивают автоматическую упаковку и/или шифрование данных, но такие различия малосущественны. В большинстве случаев различия между физическими уровнями разных дисковых ФС состоят только в том, какие метаданные и в каком виде они хранят.

Рассмотрим требования, предъявляемые к метаданным дисковой файловой системы. Очевидно, что, прежде всего, метаданные ФС должны содержать достаточный набор сведений о файле. Минимум в настоящее время — это список секторов, в которых находятся данные файла, имя файла, ссылка на каталог, в котором он находится, размер, дата и время последнего изменения. Обычно хранятся также такие атрибуты, как сведения о правах пользователей на каждый объект ФС, дата и время последнего обращения к файлу.

Но мало просто располагать метаданными. С ними необходимо работать. Из этого вытекают следующие требования к метаданным и работе ФС с ними.

1. Как можно меньший размер. Метаданные, по сути, являются для ФС балластом (поскольку в них нет данных пользователя). Естественно, с точки зрения максимально эффективного использования дискового пространства, желательно, чтобы при хранении всех необходимых сведений метаданные ФС занимали как можно меньше дискового пространства.

2. Как можно меньшее время обращения. Для выполнения любой операции с данными необходимо сначала обратиться к метаданным (чтобы определить расположение данных и проверить права обращающегося на доступ к ним). Время, которое затрачивается на извлечение и анализ метаданных, входит в общее время обращения к данным. Поэтому ФС должна обеспечивать максимально быструю работу с метаданными.

3. Третье, по счёту, но не по значению, требование: структура метаданных должна обеспечивать достаточную надёжность хранения данных в ФС. Под этим понимается минимальная вероятность возникновения сбоев в файловой системе, способность ФС к самовосстановлению при таких сбоях и минимальная вероятность потери информации при сбоях, вызванных различными причинами. Ниже мы подробнее рассмотрим проблему обеспечения надёжности файловой системы.

Хранение метаданных.

Естественно, метаданные, как и данные, тоже хранятся на диске, в его секторах. Каталоги обычно представляют собой просто файлы специального вида, которые ФС отличает от обычных файлов (файлов данных) по специальному системному атрибуту. Этот атрибут обычно недоступен пользователю и не может быть изменён командами ОС, так что невозможно превратить, например файл в каталог — можно только уничтожить файл и создать каталог с тем же именем, либо наоборот. Соответственно, каталоги хранятся на диске как обычные файлы, с использованием того же механизма размещения. Исключение может составлять (и то, вообще говоря, не обязательно) только корневой каталог файловой системы. Он может иметь отличия в структуре и на него могут распространятся некоторые дополнительные ограничения. Так, в ФС FAT корневой каталог диска имеет фиксированный максимальный размер (этот размер задаётся при форматировании диска и не может быть изменён) и располагается на диске сразу после системной области, в то время как все остальные каталоги могут располагаться как угодно, а количество файлов в них ограничено только объёмом диска.

Атрибуты файлов обычно хранятся в каталогах, в которых зарегистрированы сами файлы. Каталог представляет собой последовательность элементов, каждый из которых описывает один файл. Элемент каталога и содержит имя, размер, атрибуты файла. В некоторых типах ФС атрибуты файлов хранятся в специальных системных таблицах, а каталоги содержат только ссылки на эти таблицы.

Файловые таблицы (в них находятся списки секторов диска, в которых располагаются данные каждого из файлов) и прочие метаданные (индексы, журналы) размещаются в соответствии с правилами ФС. Файловая таблица – основа ФС. Разрушение файловой таблицы приводит к потере всех данных, хранящися на диске. Разумеется, сами данные при такой аварии остаются там, где и находились, но использовать их становится невозможно, так как теряются сведения о том, к какому файлу относятся данные, находящиеся в каждом конкретном секторе. Естественно, можно попытаться считать диск по секторам и восстановить цепочки секторов для каждого из файлов, но на практике это нереально. Максимум, что удаётся восстановить в таких случаях — это несколько процентов данных, преимущественно текстовых, хранившихся на диске.

Вследствие сказанного, в ФС принимаются меры к недопущению разрушения файловых таблиц. Прежде всего, это дублирование файловых таблиц в нескольких экземплярах. Оно может позволить восстановить файловую таблицу из копии-дубля, если основная файловая таблица оказывается почему-то разрушенной. Ещё одним способом повышения стойкости файловых таблиц является случайное распределение их по диску. Так, если в примитивных ФС (таких, как FAT) файловые таблицы размещаются всегда в строго определённом месте диска (в FAT — в начале диска), то в более развитых системах они могут располагаться где угодно, причём вовсе не обязательно непрерывно. Такое расположение файловых таблиц приводит к тому, что при приведении в негодность какого-то небольшого локального участка диска гарантированно

не происходит полного разрушения файловой системы. Даже если при этом пострадает фрагмент файловой таблицы, то по дублирующим таблицам удаётся восстановить если и не всю информацию, то значительную её часть.

2.3.4 Сбои в работе ФС.

Теоретически, в нормальных условиях, то есть при работе на исправном оборудовании, в отсутствии аварий и использовании только безошибочного программного обеспечения, любая корректно разработанная ФС функционирует без ошибок и без потерь данных. Но в реальной жизни часто встречаются некорректно работающие программы, происходят зависания операционной системы, сбои питания компьютера, наконец, физические нарушения в оборудовании. Всё это может приводить и приводит к потерям данных в ФС даже при условии, что сама ФС и всё связанное с ней программное обеспечение работают совершенно корректно.

Некоторые причины сбоев самоочевидны. Скажем, очевидно, что при зависании ОС или при выключении питания компьютера во время записи программой данных в файл, часть данных не будет записана. Очевидно, что при физической аварии диска часть информации на нём (а то и вообще вся информация) будет потеряна.

Менее очевидно, что использование кэширования при записи данных (а оно используется при записи на несъёмные диски во всех современных ОС) может способствовать потере данных, которые, казалось бы, уже записаны на диск. Например, предположим, что некоторая программа записала данные в файл и благополучно завершилась. Записанные данные считаются находящимися на диске, но физически их там может и не быть. В момент выполнения программой операции записи данные помещаются в кэш, который сбрасывается на диск позже, когда ОС сочтёт текущий момент наиболее благоприятным для этой операции. Момент этот может наступить через несколько секунд, минут, а то и часов после выполнения операции записи. Если сбой питания или зависание ОС произойдёт в промежутке времени, в котором данные ещё не помещены на диск, то после перезагрузки окажется, что данные, которые, казалось бы, были записаны, на диске отсутствуют.

Можно выделить два вида ошибок, возникающих при сбоях в работе ФС. Это разрушение (потеря) данных и нарушение согласованности данных и метаданных.

Разрушение данных возникает в тех случаях, когда в ФС оказываются уничтоженными нужные данные или (что бывает гораздо чаще) данные, предназначенные для записи в файл, оказываются по какой-либо причине не записаны туда. Разрушение имеющихся данных — явление достаточно редкое. Обычно его возникновение свидетельствует о некорректно работающем (или написанном в целях нанесения вреда) программном обеспечении, которое стирает или перезаписывает нужные данные, либо о физическом повреждении носителя. Единственный способ защититься от его последствий (вернее, минимизировать потери) — следовать правилам регулярного резервного копирования важной информации.

Незапись данных, как было уже указано выше, может произойти при любом сбое в системе. Вообще говоря, сбои неизбежны. Не может быть стопроцентной уверенности в физической исправности носителя. Может внезапно пропасть напряжение в питающей сети (даже блок бесперебойного питания спасает не всегда). А уж от ошибок в программах (да и в самой ОС), совершенно точно, не застрахован никто.

Уберечься от уничтожения данных можно (правда, опять же, не всегда) путём правильной установки прав. Снизить вероятность незаписи данных тоже можно, используя программное обеспечение, которое обеспечивает гарантированную запись данных на диск. Так, например, в ОС семейства Windows NT стандартные средства API позволяют принудительно сбросить на диск кэш отложенной записи путём вызова предназначенной для этого системной функции. Программы, работающие с важными данными, могут использовать эту функцию для того, чтобы быть уверенными в фактическом помещении записанных данных на диск. Однако применение этого механизма ограничивается тем, что отказ от отложенной записи снижает (причём весьма значительно снижает) производительность системы.

Так или иначе, вероятность потери данных в ФС имеется в любой ОС и не может быть сведена к нулю, хотя имеющиеся в любой ОС средства позволяют снизить её до практически приемлемых величин.

Рассогласование данных и метаданных. Поскольку данные и метаданные, хранящиеся в файловой системе, логически связаны между собой, практически при каждом изменении данных требуется соответствующее изменение метаданных. При создании файла требуется вычисление цепочки секторов и регистрация её в файловой таблице, создание элемента каталога, запись правильного имени и атрибутов файла. При добавлении данных в файл или при усечении файла требуется изменить размер и дату последнего изменения файла. При редактировании файла без изменения его размера нужно изменить дату последнего изменения. При чтении файла требуется заменить дату последнего обращения к нему. При создании каталога нужно создать специальный файл, который будет содержать каталог, и выполнить в ФС все изменения, связанные с созданием этого файла.

Поскольку данные и метаданные хранятся в разных местах, они в принципе не могут быть изменены одной-единственной командой записи на диск, а значит, возможна ситуация, когда сбой в работе ФС произойдёт как раз в тот момент, когда часть изменений произведена, а часть — ещё нет. Скажем, данные в новый файл уже записаны, а вот содержащий его каталог ещё не изменён. Или, наоборот, файловая таблица и каталог уже изменены, а вот данные, предназначенные для записи в файл, всё ещё находятся в кэше. Сбой в любой из упомянутых моментов времени приведёт к тому, что метаданные, записанные на диске, окажутся несоответствующими данным.

Рассогласование данных и метаданных может внешне выглядеть как потеря данных, то есть выразиться в отсутствии части записанных данных в файле или в том, что в части файла вместо данных окажутся нули или мусор. Кроме того, оно может проявиться в появлении на диске секторов, не зарегистрированных в списке свобод-

ных и, в то же время, не принадлежащих ни одному из файлов. Ещё менее приятным является такое проявление рассогласования, как пересекающиеся файлы. Они появляются тогда, когда по каким-либо причинам один и тот же сектор диска оказывается закреплён в файловой таблице за двумя или более файлами. Ясно, что физически в данном секторе находятся данные только одного из пересекающихся файлов. Помещение данного сектора в состав прочих файлов — ошибка. Проблема состоит в том, что невозможно каким-либо формальным методом определить, к какому из файлов должен принадлежать данный сектор. Возможны и ещё более серьёзные последствия, например, полное разрушение структуры каталогов.

Рассогласование данных и метаданных является гораздо более опасной для системы в целом неисправностью, чем разрушение части данных. Связано это с тем, что разрушение данных приводит, как правило, к потере лишь относительно небольшой части данных (содержимого одного или нескольких файлов). При правильной организации архивирования важной информации и при разумно организованном порядке работы пользователей с ОС разрушение данных лишь в редких случаях может сказаться на работоспособности системы. Даже в тех случаях, когда разрушенные данные относятся к ключевой информации ОС, их восстановление обычно не требует значительных затрат средств и времени. Рассогласование же данных и метаданных приводит ФС в состояние, когда любая операция в ФС может привести к непоправимому разрушению системы. Так, например, если из-за рассогласования произойдёт совмещение по секторам какого-либо пользовательского файла и области диска, хранящей каталоги или файловые таблицы (то есть в файловой таблице ФС файлу пользователя будут ошибочно приписаны те сектора, в которых фактически уже находятся каталоги или файловые таблицы), любая операция записи в такой файл, выполненная запущенной пользователем программой, может привести к полному разрушению *всей* структуры ФС и, как следствие, к полному краху операционной системы.

2.3.5 Восстановление ФС после сбоя.

Поскольку сбои ФС в определённых ситуациях просто неизбежны, разумеется, существуют программы, предназначенные для приведения нарушенной ФС в корректное (то есть внутренне согласованное) состояние. Основные действия, которые в состоянии выполнить эти программы, обычно заключаются в следующем.

- Проверка и коррекция структуры метаданных. Если метаданные содержат явные формальные ошибки [15], такие ошибки обнаруживаются и исправляются. Например, если несколько копий файловых таблиц, хранящихся на диске, отличаются друг от друга, делается попытка согласовать их путём выбора наименее пострадавшего варианта и копирования его в другие копии таблицы. Находятся и устраняются пустые элементы каталогов (элементы каталогов, которым

[15]Обратите особое внимание на слова *явные* и *формальные*! Программа восстановления ФС не является интеллектуальной и не в состоянии оценить смысловую корректность данных и метаданных. Она лишь проверяет их соответствие набору формальных критериев, ничего более!

не соответствуют реальные файлы или каталоги). Выполняются все возможные проверки корректности метаданных.

- Поиск и устранение потерянных секторов на диске. Путём проверки файловых таблиц разыскиваются сектора диска, которые числятся занятыми, но не принадлежат ни одному из файлов. Такие сектора либо преобразуются в файлы, либо просто помечаются как свободные.

- Поиск и устранение пересечений файлов. Проверяется, не зарегистрированы ли одни и те же сектора диска как принадлежащие различным файлам. Если такая ситуация действительно имеет место, то пересекающийся сектор копируется столько раз, сколько существует использующих его файлов, после чего каждая из копий приписывается к своему файлу. Очевидно, что при этом все файлы, кроме одного, окажутся испорченными. Данное действие предназначено для того, чтобы устранить ошибку и, если повезёт, сохранить в порядке хотя бы один из пересёкшихся файлов.

- Приводятся в соответствие зарегистрированные длины файлов и фактические их размеры, вычисленные по файловым таблицам.

- Проверяется и, если необходимо, исправляется структура каталогов.

- Производится проверка самого физического диска на способность к нормальному чтению и записи.

Программа проверки и восстановления приводит файловую систему во внутренне согласованное состояние. Вообще говоря, при этом не гарантируется, что будет сохранён максимум данных, но гарантируется, что после выполнения проверки и коррекции файловая система в дальнейшем будет функционировать правильно и новые изменения в ней не вызовут непредсказуемых последствий.

2.3.6 Журналируемые ФС.

Как было уже сказано, ошибки рассогласованности данных и метаданных в ФС являются наиболее серьёзными и опасными. Но, в отличие от ошибок потери данных, появление таких ошибок можно предотвратить. Для исключения ошибок рассогласования данных и метаданных разработаны специальные методы организации ФС, на основании которых строятся так называемые *журналируемые файловые системы*. Слово «журналируемые» в наименование типа ФС включается потому, что важнейшим системным элементом таких ФС является так называемый *журнал*, который и используется для предотвращения рассогласования данных и метаданных.

Причиной появляения ошибок рассогласования является, как уже говорилось, то, что данные и метаданные, как правило, физически невозможно обновлять одновременно. С помощью специального строения ФС и журнала журналируемая ФС функционирует так, что каждая операция изменения ФС (в которую входит как изменение

данных, так и изменение метаданных) выполняется как *транзакция*, то есть если операция по какой-либо причине не была завершена корректно, то она *целиком* откатывается, то есть отменяются все изменения, сделанные в ходе этой операции, как в данных, так и в метаданных. Если же операция выполняется до конца, то *все* изменения, которые она требует, подтверждаются одновременно, одной-единственной командой.

Наличие журнала как элемента ФС и связанная с этим необходимость работы с этим журналом требует (по сравнению с аналогичной, но не журналируемой ФС) дополнительного дискового пространства и машинного времени. Однако преимущества журналируемых ФС настолько превышают дополнительные затраты, что на эти затраты обычно не обращают внимания.

Что даёт использование журналируемых ФС? В различного рода рекламных материалах, посвящённых современным ОС, их распространители обычно подчёркивают поддержку журналируемых ФС, убеждая потенциальных покупателей, что такая ФС обеспечит гарантированную сохранность данных. Необходимо понимать, что подобные утверждения не имеют ничего общего с истинным положением вещей. Журналируемые ФС не спасают от потери данных. Во-первых, потому, что, как отмечалось выше, во многих случаях избежать потери данных просто невозможно. Во-вторых, потому, что целью создания журналируемых ФС было вовсе не это. Задача журналируемой ФС — не спасение всех данных любой ценой, а предотвращение рассогласование данных и метаданных, которое может привести к серьёзному нарушению работы ФС в целом или даже к её полному логическому разрушению.

Более того, в некоторых случаях в журналируемой ФС может быть потеряно больше данных, чем в обычной. Например, пусть питание компьютера было неожиданно выключено в момент, когда завершалась операция добавления данных в файл. Был указан новый размер файла и произведена запись всех данных, но в момент записи последнего сектора файла компьютер отключился. При использовании обычной ФС после перезагрузки выяснится, что практически все данные записаны в файл, только последний сектор содержит нули либо мусор. Вполне возможно, что сама ФС (точнее, её программа контроля) даже не заметит ошибки. Если же используется журналируемая ФС, то произойдёт следующее. Завершение операции изменения файла должно быть подтверждено соответствующей записью в журнале. До тех пор, пока такая запись не появилась, изменения не учитываются. Естественно, записи о завершении операции в журнале не окажется, ведь операция не успела завершиться. Поэтому после перезагрузки система обнаружит в журнале ФС запись о неподтверждённой операции и выполнит откат, то есть отменит все изменения, относящиеся к этой операции. В результате файл окажется неизменённым.

Таким образом, налицо потеря большего количества данных, чем в первом случае. Но плохо ли это? Нет! Ведь в первом случае неизвестно точно, какая часть секторов не успела записаться на диск, а значит, невозможно сказать наверняка, какая часть данных, записанных в файл, верна, а какая — нет. В результате всё содержимое файла

оказывается «на подозрении». Можно лишь догадываться, с большими или меньшими основаниями, о том, верны или неверны те или иные данные в файле. Во втором же случае можно сказать совершенно определённо, что все данные в файле верны. В результате вместо полных данных, верность которых сомнительна и формально неопределима, мы получаем файл, хотя и без последних данных, но зато с гарантией корректности того, что в нём находится. Тот факт, что новые данные не записались, может быть легко установлен, после чего возможны любые действия, направленные на исправление ситуации.

2.3.7 Выбор типа файловой системы.

Обычно ОС позволяет использовать более одного типа файловой системы. При этом какой-то один из типов является для системы основным («родным», если угодно), а остальные поддерживаются, главным образом, для обеспечения совместимости с другими системами или более ранними версиями той же системы. Так, например, для ОС Linux базовой является файловая система ext2, но поддерживается ещё несколько десятков различных файловых систем — часть полностью, часть только в режиме чтения.

Выбор файловой системы, которая будет использоваться на каждом из дисков — ответственное решение. Смена типа файловой системы на диске обычно требует переформатирования разделов, которое сопряжено с потерей всех находящихся там данных.

Если имеется возможность выбора файловой системы, то в выборе следует исходить из свойств и качеств доступных систем и требуемых возможностей системы. Обычно не каждая файловая система позволяет использовать все возможности ОС. Так, Windows NT поддерживает работу с файловой системой FAT16, но её использование не даёт возможности управлять правами пользователей на доступ к каталогам и файлам. Это ограничение связано с тем, что в структуре FAT просто не предусмотрено возможности хранения прав доступа.

Использование каждой файловой системы требует определённых затрат памяти и обеспечивает более или менее эффективное использование дискового пространства и большую или меньшую скорость работы. Выбирая файловую систему, следует исходить из следующих соображений.

- Файловая система должна обеспечивать реализацию всех необходимых возможностей. Так, если Windows 2000 устанавливается на компьютер, который будет работать как сервер с большим количеством пользователей, то использование на его основных дисках файловой системы FAT недопустимо, поскольку она не обеспечивает необходимых возможностей администрирования.

- Файловая система должна быть надёжной и восстановимой. Желательно, чтобы файловая система обеспечивала надёжную работу даже в случае каких-либо

аппаратных проблем. Должны иметься все необходимые утилиты для восстановления данных в случае системных сбоев. Для ответственных применений лучше использовать журналируемые ФС, так как они более надёжны.

- Файловая система должна обеспечивать приемлемую скорость работы на используемом компьютере. Так, ОС Windows 2000, установленная на компьютер с 32 мегабайтами ОЗУ, при использовании файловой системы NTFS работает недопустимо медленно. Если возникает необходимость использования именно этой ОС именно на этом компьютере, придётся выбрать одно из двух – либо использовать FAT, либо добавить в компьютер памяти.

- Если компьютер (или жёсткий диск, или раздел на диске) используется более чем с одной ОС, следует выбирать файловую систему, поддерживаемую другими используемыми ОС. Если это нежелательно, то для данных, разделяемых между разными системами, нужно создать отдельный раздел, отформатированный под файловую систему, поддерживаемую всеми ОС.

2.3.8 Работа с файлами из программ.

Работа с файлами традиционными средствами.

Программа, работающая с файлом, обязана выполнять ряд операций, обеспечивающих корректное прочтение и изменение файла. Прежде всего, система требует,чтобы файл, прежде чем он начнёт использоваться программой, был этой программой открыт. Для этого программа должна выполнить вызов одной из функций API.

При открытии файла ОС регистрирует файл как открытый и создаёт внутренние структуры данных, необходимые для работы с ним. Как правило, открытому файлу ОС приписывает некоторое целое число — хэндл. Хэндл является идентификационным номером файла, который используется для указания этого файла при вызове функций API для работы с файлом. Хэндл не имеет физического смысла, он присваивается каждый раз при открытии файла и уничтожается при его закрытии. Открыв несколько раз один и тот же файл, программа получит несколько различных хэндлов. Если несколько одновременно выполняемых программ откроют один и тот же файл, то каждая из них также получит свой хэндл этого файла, не совпадающий с хэндлами, полученными другими программами.

При открытии файла обычно указывается режим работы с ним. Файл может быть открыт только на чтение, только на запись, на чтение и запись, на добавление. Кроме того, в многозадачных системах программа должна указать режим разделения файла — параметр, указывающий на то, как будет осуществляться совместный доступ к файлу при попытке его использования одновременно несколькими программами. Обычно можно разрешить другим программам читать файл, записывать в него, удалять его. Несколько параметров могут комбинироваться.

Программа может работать с файлом, вызывая функции API ОС. При написании программы на языках высокого уровня, которые имеют собственные встроенные воз-

можности для работы с файлами, программист использует именно эти встроенные возможности, но фактически всё равно происходят вызовы API. Работа с файлом базируется на понятии *файлового указателя*. Считается, что файловый указатель связан с каждым открытым программой файлом. Файловый указатель всегда указывает на определённое место в файле. При открытии файла на запись или чтение файловый указатель указывает на начало файла (на первый байт файла), при открытии на добавление — на конец файла (на позицию за последним байтом файла).

Как правило, ОС обеспечивает возможность чтения из файла буфера произвольного размера, запись в файл буфера произвольного размера и перемещение файлового указателя на другое место. Операции чтения или записи данных применяются к содержимому файла, начиная от позиции файлового указателя. Операция чтения данных прочитает указанное количество байт с указываемого места файла, а операция записи — перезапишет указанное количество байт там же. В любом случае, после выполнения операции чтения или записи файловый указатель переместится "вперёд" (то есть в направлении конца файла) на то же количество байт, которое было считано или записано. В результате следующая операция чтения или записи будет работать с данными, которые ещё не были прочитаны или перезаписаны. Сейчас в большинстве систем есть возможность произвольно устанавливать позицию файлового указателя в файле и осуществлять, таким образом, произвольный доступ к хранящимся в нём данным.

После окончания работы с файлом программа обязана его закрыть. Закрытие выполняется с помощью вызова соответствующей функции API. Закрытие файла необходимо по нескольким причинам. Во-первых, в момент закрытия на диск сбрасываются те данные, которые ещё не были записаны туда. Завершая программу без закрытия открытых ею файлов можно потерять часть записанных в файлы данных. Во-вторых, при закрытии файла ОС уничтожает связанные с ним внутренние структуры данных и фиксирует, что файл более не занят программой, что даёт возможность другим программам работать с ним без ограничений. Во многих современных системах закрытие файла не является обязательной операцией. Если программа завершается, не закрыв какие-то из открытых ею файлов, то система сама закроет эти файлы. Но злоупотреблять такими возможностями не следует.

Файлы, отображаемые в память.

Многие алгоритмы работы с данными разработаны в предположении, что обрабатываемые данные представляют собой массив в ОЗУ компьютера. Иногда возникает необходимость применения этих алгоритмов к данным, хранящимся в файлах. Если при этом следовать стандартной методике работы с файлами, описанной выше, то придётся либо существенно переработать алгоритм, либо согласиться с серьёзной потерей производительности. Как уже было сказано, программа читает и записывает данные блоками. Чем больше мелких блоков должна переместить на диск или с диска программа, тем медленнее идёт обработка, ведь для каждого блока должен пройти вызов API, а при чтении/записи блока каждый раз выполняется обращение к

диску, которое занимает относительно много времени. Кэширование диска частично оптимизирует эти операции, но полностью исключить лишние обращения к диску не может.

Для того, чтобы программа могла применять к файлам в точности те же алгоритмы, что и к массивам в ОЗУ, ОС может предоставлять программам возможность работы с файлами, отображаемыми в память. Это не какой-то особый вид файлов, это просто другая методика работы с теми же самыми файлами. С помощью одного или нескольких вызовов функций API программа может потребовать от ОС после открытия файла отобразить этот файл на адресное пространство программы. При этом ОС возвращает программе некоторый адрес в памяти. Затем программа может обращаться к блоку памяти, начинающемуся по указанному адресу и имеющего размер, соответствующий размеру файла. При этом все операции чтения и записи памяти в пределах этого блока будут фактически приводить к чтению или записи данных в открытый файл. Операции чтения и записи на диск при этом выполняет сама ОС, причём специальные программные модули обеспечивают оптимизацию этого процесса с целью минимизировать обмен данными с диском.

Используя файлы, отображаемые в память, программа может работать с файлом с помощью тех же алгоритмов, что и с областью памяти. Более того, если в программе уже имеется ранее откомпилированный модуль, реализующий нужный алгоритм для данных в ОЗУ, его можно безо всяких изменений и без перекомпиляции использовать для работы с данными в файле.

Асинхронный доступ к файлам.

Как уже говорилось выше, работа с файлами на внешних устройствах занимает относительно много времени и составляет довольно заметную часть общего времени, затраченного на исполнение программы. При этом в моменты обращения к диску исполнение программы приостанавливается. Если в системе решается несколько задач одновременно, ОС отдаёт процессорное время, не используемое ожидающей завершения операции с диском программой, какой-либо другой программе. В противном случае это время просто теряется.

Потерь времени на ожидание завершения операций ввода/вывода можно было бы избежать, если бы программа могла, запустив такую операцию, продолжать выполняться, обрабатывая данные, не зависящие от данного конкретного ввода или вывода. Например, в таких условиях программа, занимающаяся обработкой последовательности данных, могла бы, начиная обработку только что загруженной с диска порции информации, тут же дать команду на чтение следующей порции. По завершении обработки очередной порции данных она могла бы дать команду на запись результатов в файл и тут же, не дожидаясь, пока ОС выполнит запись, начать обработку новой порции, которая к тому моменту была бы уже загружена.

Нечто подобное обеспечивает система кэширования дисков, но эта система организуется в расчёте на достижение максимальной средней производительности и, как правило, ничего не знает об особенностях каждой конкретной задачи, решаемой в

системе. В лучшем случае программа может задать режимы кэширования файлов, с которыми она работает, но это обычно даёт лишь небольшой прирост производительности.

Если ОС поддерживает возможность создания многопоточных программ, то программист может организовать работу программы по вышеописанной схеме, создав в программе три потока вместо одного: первый поток должен читать данные из входного файла, второй — обрабатывать их, а третий – записывать результаты в выходной файл. Тогда в те моменты, когда потоки чтения и записи простаивают в ожидании завершения очередной операции ввода/вывода, основной поток (второй) может обрабатывать данные, используя невостребованное процессорное время.

Некоторые операционные системы позволяют программам выполнять операции чтения из файлов и записи в них без ожидания завершения операции. Механизм, обеспечивающий эту возможность, называется асинхронным доступом к файлу. Слово "асинхронный"в названии означает, что, в отличие от традиционных методов доступа к файлу, невозможно точно определить, когда (после выполнения какой команды) произойдёт завершение каждой конкретной операции ввода/вывода. Программа может запустить (в данном случае можно даже сказать «заказать») ту или иную асинхронную операцию, но, поскольку реальная скорость её исполнения зависит от множества факторов, которые невозможно учесть, про момент завершения операции можно сказать только, что он наступит позже момента её запуска. Естественно, когда программе требуются данные, которые она пыталась прочитать, или когда должна быть уверенность, что данные, которые требовалось записать, уже записаны, программа должна иметь возможность проверить, завершилась ли уже операция и, если необходимо, дождаться её завершения.

В связи с изложенным, программа обычно использует асинхронный доступ к файлу следующим образом.

- Программа открывает файл в режиме асинхронного доступа. То, что файл будет использоваться асинхронно, должно быть известно ОС при его открытии, так как для открытия таких файлов ОС могут потребоваться некоторые специальные действия.

- Чтобы выполнить операцию асинхронного чтения или записи, программа использует соответствующие функции API. Это могут быть те же самые функции, что используются для обычного (синхронного) доступа. В этом случае они просто работают иначе, если файл, для которого они вызываются, открыт в асинхронном режиме. На режимы работы и параметры команд асинхронного доступа ОС может накладывать дополнительные (по сравнению с синхронным доступом) ограничения.

- После выполнения команды работы с файлом программа продолжает работу, при этом программист должен учитывать, что операция, выполнение которой было запущено, будет выполняться ОС параллельно с работой основной части

программы. В момент, когда программа должна удостовериться в том, что соответствующая операция завершилась, или дождаться её завершения, программа должна вызвать одну из функций API, называемых функциями ожидания, передав ей в параметре хэндл файла. Функция ожидания приостанавливает программу до тех пор, пока не завершится операция асинхронного доступа с заданным файлом. Обычно функции ожидания можно задать размер промежутка времени (таймаута), в течение которого функция должна ожидать. По завершении функции ожидания программа может проверить результат асинхронной операции и определить, выполнилась ли она корректно, или произошла ошибка, или истёк таймаут, и функция ожидания не дождалась завершения операции. Последнее может означать, что операция по каким-то причинам «зависла» или просто её выполнение требует в данном случае больше времени, чем предполагал программист. Обработка подобных ситуаций зависит от особенностей программы.

Таким образом, программа, используя асинхронный доступ, может, имея только один поток, не терять время на ожидание операций ввода-вывода.

Асинхронный доступ позволяет во многих случаях повысить эффективность работы программ, обрабатывающих значительные объёмы данных, хранящихся в файлах. С другой стороны, использование этого механизма требует от программиста повышенного внимания и аккуратности, поскольку при асинхронном доступе намного легче допустить ошибку, связанную с попыткой обработать данные, ещё не считанные из файла или с порчей данных, которые считаются уже записанными в файл, но на самом деле таковыми ещё не являются.

2.3.9 Концепция «Устройство как файл».

Некоторые ОС (UNIX и ряд других, построенных по той же идеологии) поддерживают концепцию «устройство как файл». Она состоит в том, что все (или, во всяком случае, многие) системные устройства отображаются как специального вида файлы в файловой системе. Например, звуковая карта компьютера может отображаться в файловой системе как файл, например, /dev/sound. Запись в этот файл (стандартными командами записи в файл) фактически приводит к передаче данных в звуковую карту, чтение из этого файла — к получению данных от звуковой карты. По тому же принципу на файловую систему отображаются порты ввода-вывода, большинство системных устройств... Более того, сами диски представлены в файловой системе как отдельные файлы. Открыв, например, в Linux на чтение файл /dev/hda1, программа получает доступ к первому физическому разделу на первом жёстком диске, подключенном к IDE-контроллеру. Этот файл содержит соответствующий раздел безо всякого структурирования и даже разделения на файлы и каталоги. Такое представление позволяет работать с диском как с файлом и использовать для работы с физическим диском обычные команды, предназначенные для работы с файлами.

Использование концепции «устройство как файл» позволяет упростить API операционной системы и уменьшить количество системных утилит. Вместо реализации

нескольких десятков для работы с каждым из видов устройств, можно реализовать только набор команд и утилит для работы с файлами, и использовать его для управления любыми устройствами.

2.4 Подсистема администрирования.

В любой ОС, предназначенной для поддержания работы нескольких пользователей, обязательно предусмотрены средства разграничения доступа пользователей к системе, к программам, внешним устройствам и данным. Степень сложности этих средств определяется назначением системы, важностью данных, хранимых в ней, возможными потерями от утраты, порчи или разглашения информации или от несанкционированного доступа к программам.

2.4.1 Идентификация пользователей.

В простейших случаях (в однопользовательских ОС) обычно достаточно обеспечить проверку того, действительно ли в систему пытается войти её хозяин. Поскольку пользователь только один, в системе по определению отсутствуют ресурсы, которые ему недоступны и ОС может не заниматься контролем доступа к ним. В таких системах достаточным является предоставление пользователю возможности указать пароль входа в систему. ОС должна запрашивать пароль при входе и допускать к работе только того, кто ввёл правильный пароль. Этот вариант системы контроля доступа настолько прост, что он реализуется даже на уровне BIOS большинства персональных компьютеров.

ОС, предназначенные для работы (одновременной или последовательной) с ними нескольких пользователей, требуют организации более сложной системы разграничения доступа. Как минимум, система контроля доступа должна предусматривать регистрацию пользователей и проверку их при попытке входа в систему.

Каждый пользователь, прежде чем получить доступ к системе, должен быть зарегистрирован в ней. [16] При регистрации пользователю приписывается некоторое слово или словосочетание, называемое именем входа или идентификатором пользователя. Подлинное имя пользователя также может быть где-то зафиксировано, но в системе пользователь фигурирует именно под именем входа. При входе в систему пользователь должен ввести своё имя входа и пароль. При подтверждении правильности пароля пользователю разрешается вход в систему. Процесс входа пользователя в систему, сопряжённый с проверкой прав доступа, называется ещё *авторизацией*.

Как правило, для хранения паролей в системе используются специальные методы шифрования. Пароли в открытом виде в системе не хранятся вообще. При регистрации пароля он сразу же шифруется и в зашифрованном виде записывается. При этом

[16] Регистрацию проводит пользователь, который имеет в ОС соответствующее право. Обычно такие пользователи называются администраторами.

алгоритм шифрования пароля таков, что непосредственно получить из зашифрованного пароля незашифрованный невозможно. [17] Введённый в момент входа пользователя в систему пароль шифруется тем же методом и сравнивается системой с зашифрованным паролем, который был сохранён в момент регистрации пользователя, и если зашифрованные пароли совпадают, то считается, что введённый пароль правилен. Важно отметить, что даже администратор не знает паролей пользователей, и, следовательно, не может работать под их именами.

2.4.2 Управление доступом пользователей к ресурсам.

В многопользовательской системе ОС должна реализовывать разграничение прав на ресурсы системы. Конкретный список видов ресурсов, доступ к которым может контролировать система, зависит от конкретного типа системы и от её настройки. Так, пользователю может быть предоставлено право на чтение или запись в определённый файл или каталог, право печатать на присоединённом к компьютеру принтере, право доступа к коммуникационным портам, право на запуск определённой программы и прочее.

Поскольку многие пользователи системы имеют один и тот же набор основных прав, обычно ОС поддерживает концепцию профилей пользователей. Профиль – это зарегистрированный в системе поименованный перечень прав. При регистрации пользователя, вместо того, чтобы подробно указывать все права на ресурсы, ему приписывается определённый профиль, в результате чего пользователь автоматически становится обладателем всех перечисленных в профиле прав. Дополнительные по сравнению с профилем права и ограничения указываются отдельно.

В пределах своих прав пользователь может работать с системой и создавать собственные ресурсы (каталоги, файлы данных, программы). Созданный пользователем ресурс находится в его полном распоряжении, и пользователь может предоставить права на его использование другим зарегистрированным пользователям системы. Правом отчуждения пользовательского ресурса (изменения владельца) обладает только администратор системы.

Существует два способа предоставления прав на свои ресурсы пользователям. Они не являются взаимоисключающими и могут дополнять друг друга. Один способ заключается в следующем. Каждый пользователь при регистрации в системе всегда регистрируется как член некоторой группы пользователей. Любой ресурс в системе

[17] Невозможно не в смысле "требует слишком большого объёма расчётов", как в случае с некоторыми шифрами, а вообще. Обычно при запоминании пароля используются алгоритмы, которые в принципе работают только в одну сторону (при шифровании пароль преобразуется в некоторую контрольную сумму, при этом теряется часть информации о пароле, поэтому восстановить пароль по контрольной сумме невозможно). "Взломать"зашифрованный таким образом пароль можно только одним способом – последовательно перебирая все возможные пароли, шифруя их и сравнивая с известным зашифрованным паролем. При достаточно большой длине пароля, сложном алгоритме шифрования и избегании использования в качестве паролей распространённых слов и выражений этот подбор может длиться настолько долго, что подобранный пароль к моменту его получения окажется устаревшим.

принадлежит некоторому пользователю и некоторой группе пользователей. На каждый ресурс хранится три набора прав – для владельца, для группы и для всех остальных пользователей. Права обычно предоставляются на чтение, запись, исполнение, возможность входа в каталог (для каталогов). Пользователь, зарегистрированный как владелец ресурса, имеет права первой группы, все пользователи, зарегистрированные в группе ресурса, имеют права второй группы, а все остальные пользователи имеют права третьей группы. Такая схема предоставления прав реализована, например, в ОС UNIX. Исходные права, устанавливаемые на создаваемый ресурс, определяются настройкой системы. Обычной практикой является предоставление полного доступа владельцу. Доступ для группы и всех остальных пользователей устанавливается в зависимости от условий. Например, пользователь может установить права на файл таким образом, что файл будет доступен на чтение и запись для самого пользователя, на чтение — членам одной с ним группы и полностью закрыт для всех остальных. Такая схема достаточно примитивна, но проста и часто вполне достаточна.

Другой способ — предоставление прав отдельно каждому пользователю или группе пользователей. При этом любой пользователь может находиться в нескольких группах или не быть зарегистрированным ни в одной из них. Права на любой ресурс владелец ресурса (или администратор системы) может предоставлять отдельно любым пользователям (независимо от того, в каких группах они зарегистрированы) или любой группе пользователей. Например, владелец может предоставить полный доступ к файлу членам группы «Команда разработчиков #1», на чтение — членам группы "Бухгалтерия"и отдельно пользователю с именем входа «VASYA», запретив остальным пользователям доступ к файлу вообще. Такой способ более гибок, но сложнее в реализации, потому что для каждой группы ресурсов с общим набором прав доступа необходимо хранить полный список пользователей, групп и прав.

2.4.3 Централизованное администрирование в компьютерных сетях.

Описанный выше механизм регистрации пользователей и распределения прав применяется тогда, когда каждый компьютер является «сам себе администратором», то есть правами доступа управляет подсистема администрирования ОС именно того компьютера, к которому осуществляется доступ. Такая схема логична, проста, естественна, но не единственна. В настоящее время широко применяется также механизм авторизации пользователей с использованием централизованной системы администрирования, работающей на одном из компьютеров локальной сети. Выглядит это следующим образом.

Группа компьютеров, связанных локальной компьтерной сетью, объединяется в так называемый *домен*. Домен отличается от обычной группы объединённых сетью компьютеров тем, что один из компьютеров в нём выступает как главный, а все остальные — как подчинённые. Главный компьютер называется *контроллером домена*. На доменной организации локальных сетей построено достаточно много механизмов, но

нас сейчас интересует только один из них — механим централизованного администрирования. Он реализован, например, в доменах с контроллерами, использующими ОС Microsoft Windows NT/2000 Server. Под UNIX похожий механизм реализуется с помощью системы NIS+. Весь домен представляет собой среду, администрируемую с помощью единого механизма. Все пользователи регистрируются в домене. Все права пользователей задаются администратором домена. Когда компьютер запускается, то он входит в домен и за любыми данными, касающимися распределения доступа, обращается к контроллеру домена. Когда пользователь пытается подключиться к такому компьютеру, то его имя и введённый им пароль сообщаются контроллеру домена, и тот разрешает либо запрещает пользователю вход. Если вход разрешается, то контроллер домена передаёт ОС пользовательского компьютера перечень прав этого пользователя. Администратору домена доступны для администрирования ресурсы всех входящих в домен компьютеров. Он, например, может своим решением разрешить пользователю X осуществлять печать документов на принтере, подключенном к компьютеру пользователя Y, не спрашивая при этом разрешения самого пользователя Y и даже не поставив его в известность. Более того, даже права на администрирование своих собственных компьютеров пользователи могут не иметь. В этом случае любые административные действия на пользовательских компьютерах, включая, например, установку программного обеспечения, могут и должны выполнять администраторы домена.

Сложно дать однозначную оценку механизму централизованного администрирования домена. С одной стороны, его наличие позволяет упростить работу службы технической поддержки в организациях, где компьютеры в массовом порядке используются для выполнения текущей работы персоналом, не имеющим соответствующих технических знаний. Таким работникам выдаётся в пользование компьютер с установленным и настроенным программным обеспечением, в ОС которого пользователь при всём желании не может что-то испортить, поскольку просто не имеет на это прав. Пользователь может не задумываться над техническими вопросами функционирования своего компьютера, при любых нештатных ситуациях просто обращаясь к службе техподдержки. При необходимости внесения как локальных, относящихся к одному или нескольким пользователям, так и глобальных изменений в компьютерную сеть предприятия или организации, работники техподдержки могут сделать их самостоятельно, причём, зачастую, не отходя от своего основного рабочего места. Можно перенастраивать компьютеры пользователей, устанавливать и удалять программное обеспечение, контролировать работу пользователей и их компьютеров, причём всё это — без беготни от пользователя к пользователю, не отрывая людей от их основной работы. Пользователи могут даже не догадываться, что с их рабочими местами производятся какие-то изменения.

С другой стороны, тот же механизм может стать причиной крупных технических и организационных проблем. Обычная компьютерная сеть (одноранговая или иерархическая — это не так уж важно) — это конгломерат более-менее самостоятельных вычислительных единиц, каждая из которых используется и администрируется своим пользователем. Эти вычислительные единицы, конечно, связаны между собой в

той мере, в какой они используют ресурсы друг друга, но каждый из компьютеров зависит только от тех своих «соседей по сети», с которыми он работает. Нарушение работы какой-то части компьютеров сети, конечно, неприятно, но оно лишь косвенно затрагивает функционирование оставшейся части сети. И, самое главное, в обычной сети нет компьютера, от работы которого жёстко зависит функционирование всех остальных. А механизм централизованного администрирования очень сильно связывает компьютеры домена с контроллером домена, по сути, превращая домен в один большой компьютер с большим количеством процессоров и внешних устройств, но с единственным центральным модулем (контроллером домена), который определяет права всех членов домена на любые ресурсы и любые действия.

Какие конкретно проблемы порождает централизованное администрирование? Перечислим хотя бы наиболее существенные.

- Как в любой централизованной системе, главный компьютер — контроллер домена — неизбежно становится критическим элементом. Его отказ приводит к нарушению работы всего домена, причём даже тогда, когда все прочие компьютеры домена в полном порядке и для текущей работы им не требуются ресурсы сервера и друг друга. Ведь при отказе контроллера ни один пользователь просто не сможет авторизоваться, а значит, не сможет и использовать свой компьютер. Перегрузка контроллера домена приводит к замедлению работы всего домена, поскольку происходит замедление авторизации и проверки прав на всех компьютерах. Если же в контроллере домена произойдёт серьёзная авария с потерей данных подсистемы администрирования, то для восстановления работоспособности системы и предоставления пользователям доступа к данным (даже к их собственным данным, хранящимся на их собственных компьютерах!) может потребоваться масса времени и сил.

- Вся работа по администрированию всех компьютеров ложится на администраторов, хотя в значительной мере её вполне могли бы выполнять сами пользователи. Если же администратор разрешит пользователю администрировать собственный компьютер, то совместные действия двух (а то и больше) лиц с административными правами могут привести к самым неприятным последствиям для пользовательского компьютера.

- Администратор домена получает возможность свободно распоряжаться всей информацией, с которой работают пользователи. Но часть данных на компьютерах пользователей в принципе не предназначается для посторонних лиц. Это, например, конфиденциальная информация и личная переписка. Если компьютер пользователя администрируются им самим, то даже главный администратор предприятия или организации не сможет получить доступ к данным, которые пользователь не предназначает для его (или для всеобщего) сведения. В централизованно управляемом домене защититься от недобросовестного администратора просто невозможно.

- Компрометация системы безопасности контроллера домена или учётной записи администратора домена означает компрометацию всего домена. Если злоумышленник подберёт пароль главного администратора или найдёт способ технического взлома, который позволит ему войти в домен с правами админстратора, то в своих дальнейших действиях по получению информации с компьютеров домена он не встретит более никаких препятствий. Если же система администрирования не централизована, то взлом одного из её компьютеров обеспечит доступ только к информации, хранящейся на нём и, возможно, к информации с одного или нескольких компьютеров, владельцы которой разрешили пользователю взломанного компьютера доступ к ней. Остальные компьютеры и учётные записи их пользователей останутся в целости и сохранности, и для проникновения в них злоумышленнику придётся повторить все действия по взлому.

В защиту централизованного администрирования можно сказать следующее: во многих случаях оно действительно упрощает сопровождение системы, а главное, при низком уровне подготовки пользователей оно является практически единственным способом поддержания системы безопасности на пользовательских компьютерах в нормальном состоянии. С другой стороны, того же можно добиться и без централизованного администрирования, просто путём правильного подбора ОС и настройки обычных средств администрирования пользовательских компьютеров.

2.4.4 Разделение функций администратора между двумя работниками.

Обычным сейчас порядком является наделение администратора ОС сразу всеми административными функциями. Один администратор занимается и настройкой системы, и установкой программного обеспечения, и контролем за техническим состоянием, и управлением правами. При этом, естественно, администратор имеет права доступа ко всем данным, в том числе данным пользователей, и имеет право управлять правами пользователей. Такой порядок, несмотря на свою привычность, плох сразу по двум причинам.

- Безопасность всей системы держится на знании одного-единственного секретного слова — пароля администратора. Тот, кто узнает пароль администратора, или сможет взломать его учётную запись, получит полный, ничем не ограниченный доступ к системе.

- Действия администратора не могут быть эффективно проконтролированы, а его доступ к конфиденциальной информации, хранящейся в системе, не может быть ограничен. Ни технически, ни организационно невозможно ограничить доступ администратора к каким бы то ни было данным, хранящимся в компьютерной системе, так как права доступа ко всему есть непременное условие самой возможности его работы.

Для решения обоих этих проблем (точнее, для снижения их остроты) была предложена схема разделения функций администратора. Когда в ОС реализовано раздельное администрирование, в ней вообще нет ни одной учётной записи, владелец которой имел бы права доступа ко всему, что есть в системе. Вместо этого есть два особых, привелигированных пользователя, совместно управляющих системой. Первый — это *администратор системы*, второй — *администратор безопасности*.

Администратор системы имеет обычный для своей должности набор прав, за исключением доступа в каталоги с личными файлами пользователя и управления правами пользователей. Он по-прежнему может управлять системой, устанавливать и удалять программное обеспечение, контролировать правильность функционирования системы, даже устанавливать права пользователей на системные ресурсы.

Администратор безопасности — это пользователь, который управляет работой системы безопасности ОС. Он может предоставлять пользователям права доступа к данным и права на запуск программ, может регистрировать пользователей в группах, но сам не имеет прав доступа к данным пользователей и на общее управление ОС, установку и удаление программного обеспечения, причём в принципе не может предоставить эти права кому бы то ни было, включая себя самого.

Администраторы могут ограничивать возможность доступа друг друга к ресурсам системы. Администратор системы может запретить администратору безопасности доступ в определённые каталоги и к определённым файлам, но и администратор безопасности может ограничить права администратора системы. Главным является администратор системы, и если он закроет доступ к чему-либо, никто, включая администратора безопасности, не сможет воспользоваться соответствующим ресурсом. Но администратор безопасности, в свою очередь, может запретить администратору системы доступ в каталоги пользователей, удаление или замену некоторых системных файлов, изменение некоторых системных настроек.

Естественно, изменения в системе не исчерпываются только разделением прав администраторов. Одновременно с этим ужесточается ряд правил безопасности, чтобы ни один из администраторов не мог, воспользовавшись каким-то «обходным путём», получить права другого. Некоторые из изменений отражаются на ядре ОС.

Результатом разделения функций является увеличение защищённости системы как от проникновения извне, так и от недобросовестных действий администраторов. В чём это выражается?

- Компрометация учётной записи одного из администраторов не даёт полного контроля над системой. Если будет взломана учётная запись администратора системы, то злоумышленник сможет навредить системе, возможно, уничтожить данные, которые в ней хранятся, но не получит к этим данным доступа, поскольку соответствующие права отсутствуют у данной учётной записи. Взлом учётной записи администратора безопасности также не фатален. Конечно, и здесь злоумышленник имеет возможность нанести системе вред (например, нарушив права пользователей, что сделает невозможной их работу с системой), но и в этом случае он не получит доступа к данным пользователей, так как

и эта учётная запись не имеет соответствующих прав доступа. Чтобы всё-таки получить доступ к данным, злоумышленник должен взломать как минимум две учётные записи — учётную запись администратора безопасности и учётную запись одного из пользователей, либо учётные записи обоих администраторов. Любой из вариантов заведомо сложнее, чем взлом только одной учётной записи главного администратора в обычной системе. Кроме того, все действия по взлому займут больше времени и оставят заведомо больше следов.

- Недобросовестный администратор (неважно, который из двух) не может получить доступ к личным данным пользователей, используя для этого только свои технические права. Ни один из администраторов системы сам не имеет таких прав и не может их самому себе предоставить. Чтобы всё-таки добраться до данных, должен быть организован сговор двух администраторов или, как минимум, одного из обычных пользователей и администратора безопасности. Такое положение вещей резко снижает вероятность злоупотреблений, так как необходимость вовлекать в неправомерные действия (наказуемые, кстати, в нашей стране в уголовном порядке) ещё кого-то многократно повышает риск.

Пока раздельное администрирование не особенно широко распространено. Это объясняется относительной молодостью данной технологии и, как следствие, малой распространённостью соответствующих программных средств и недостаточным знакомством с ними как технических специалистов, так и администраторов. Тем не менее, заложенный в раздельном администировании потенциал, безусловно, приведёт к широкому его распространению. В настоящее время известны системы администрирования LIDS и RSBAC, поддерживающие раздельное администрирование. RSBAC входит, например, в дистрибутив Linux Castle, выпускаемый российской фирмой ALT Linux.

2.4.5 Права программ.

Для работы с системными ресурсами программы, запускаемые пользователем, должны иметь права на эти ресурсы. Если бы запускаемые программы имели права на вообще все ресурсы системы, то это скомпрометировало бы всю систему администрирования, поскольку можно было бы получить доступ к любому ресурсу, написав и запустив программу, которая этот доступ осуществит. Невозможно и просто приписать список прав к программе. Это потребовало бы держать в системе столько экземпляров различно настроенных программ, сколько пользователей этими программами пользуется. Поэтому в большинстве систем принято правило, согласно которому любая запущенная программа получает набор прав, который имеет в системе пользователь, запустивший её. Это решение, как показывает практика, наиболее оптимально. С одной стороны, любая запускаемая пользователем программа автоматически получает доступ ко всем данным этого пользователя, не требуя дополнительной настройки. С

другой стороны, пользователь никогда, даже по случайности или небрежности администратора, не получит посредством программы доступа к тем данным, на которые он не имеет прав.

Описанный порядок присвоения прав запущенным программам общепринят, но его использование иногда создаёт некоторые проблемы. Типичный пример – программа изменения пароля пользователя. Пользователь должен иметь возможность менять свой пароль. Для этого в его распоряжение должна быть предоставлена какая-то программа. Пароли пользователей хранятся в каком-то системном файле, прав на запись в который, очевидно, никто, кроме администратора системы, иметь не должен. Но чтобы поменять пароль пользователя, программа замены пароля должна иметь доступ на запись к этому файлу (иначе как она сможет записать в него новый пароль?)! Получается противоречие: с одной стороны, нельзя давать пользователю права на запись в файл паролей, с другой — без этих прав запущенная пользователем программа замены пароля просто не сможет этот пароль поменять.

Данные проблемы решаются одним из двух способов (иногда в пределах одной системы в разных случаях используются разные способы). Первый способ — организация работы программ, требующих повышенных прав, по технологии «клиент-сервер». Программа, выполняющая требуемое действие, разбивается на две различные программы — клиент и сервер. Клиент — это обычная пользовательская программа, получающая права пользователя, который её запускает. Сервер — это задача, запускаемая системой при старте и имеющая все необходимые права. Взаимодействие клиента и сервера происходит по некоторому протоколу, через системные средства межпроцессного взаимодействия. Программа-клиент формирует требование на выполнение необходимого действия и передаёт его программе-серверу. Программа-сервер проверяет, вправе ли данный клиент, запущенный данным пользователем, выполнять данное действие, и, если всё в порядке, выполняет его. В приведённом примере с изменением пароля программа, запускаемая пользователем, должна ввести старый и новый пароль и передать серверу запрос на изменение пароля. Сервер должен проверить, действительно ли переданный старый пароль соответствует паролю пользователя, запустившего программу, и, если это так, изменить файл паролей.

Второй способ решения той же проблемы — это использование программ с фиксированными правами. Как уже говорилось, предоставление прав самой программе в большинстве случаев — порочная практика. Но для особых ситуаций она может быть полезна. Обычный способ реализации такой возможности — это хранение для каждого исполняемого файла специального атрибута — признака использования прав владельца. Если данный флаг установлен, то программа, будучи запущена кем угодно, получает тот набор системных прав, который имеет её владелец (а не пользователь, запустивший её). Если владелец программы — администратор, то программа имеет полный доступ ко всем системным ресурсам, без каких бы то ни было ограничений. Естественно, установить данный флаг может только сам владелец файла программы или администратор. Приведённая выше задача изменения пароля решается, таким образом, очень просто. Программа изменения пароля должна принадлежать администратору и иметь флаг использования прав владельца. Тогда, будучи запущена любым

пользователем, она всё равно сможет поменять пароль.

Первый из способов обхода ограничений прав программы используется, например, в системе WindowsNT, второй — в UNIX-подобных системах. Сложно сказать, какой из подходов лучше. Принято считать, что более перспективным и менее подверженным взлому является первый метод.

Второй подход обычно называют более опасным с точки зрения возможного взлома. В принципе, достаточно тем или иным способом заставить любую программу с правами администратора запустить командный интерпретатор – и любой пользователь может получить возможность работать от имени администратора. Это действительно так, но возможностей сделать это, если только администратор не проявляет явной халатности, практически нет. Три десятилетия использования UNIX показали, что, в принципе, этот подход достаточно надёжен.

Преимущества первого способа также, в общем, неочевидны. Во-первых, системы «клиент-сервер», строго говоря, также не гарантированы от взлома. Принципиально возможен, например, взлом таких систем методом «подмены клиента» — создания программы-клиента, во всём подобной «правильному» клиенту, но инициирующей выполнение вредоносных действий. Но и этот взлом технически достаточно труден. Более существенным недостатком можно считать тот факт, что использование такой технологии требует либо постоянного наличия программы-сервера в памяти (а когда число используемых программ достаточно велико, расход памяти может оказаться значительным), либо наличия в API специальных функций для запуска тех или иных системных серверов. И то, и другое несколько загромождает систему и увеличивает затраты системных ресурсов на её функционирование.

2.4.6 Квотирование дискового пространства.

Многие (хотя и не все) многопользовательские системы обеспечивают возможность ограничения общего объёма дискового пространства, которое пользователь может использовать для хранения своих файлов. Зарегистрированному пользователю выделяется некоторый объём дискового пространства, называемый квотой. Все файлы, принадлежащие пользователю, не могут занимать в сумме больший объём, чем установленная для данного пользователя квота.

Механизм квотирования дискового пространства жизненно необходим для системы, активно используемой в качестве файл-сервера. При наличии механизма квотирования пользователь ограничен выделенной ему квотой. Это даёт возможность распределять между пользователями дисковое пространство, а не только каталоги на общем диске. В его отсутствие пользователи никак не ограничены в объёме хранимых в системе данных. Администратор не в состоянии заранее разделить свободное дисковое пространство между пользователями, в результате чего любой пользователь может занять своими файлами весь диск.

2.4.7 Протоколирование действий пользователей.

Все многопользовательские ОС обязательно предоставляют возможность ведения журнала. Журнал хранит протокол действий пользователей в системе. Администратор имеет возможность настроить систему протоколирования, чтобы она фиксировала только те действия пользователей, которые нужно. Обычно существует возможность протоколировать вход пользователя в систему, выход пользователя из системы, операции с файлами, запуск программ, действия административного характера (изменение прав пользователей на ресурсы, например).

Система протоколирования (или, как её ещё называют, система аудита) необходима для нормальной работы многопользовательской системы, так как она позволяет обнаружить, когда и каким именно пользователем произведены те или иные изменения в системе. Однако её использование отнимает некоторое процессорное время и дисковое пространство. Как правило, включение абсолютно всех возможностей протоколирования приводит к тому, что скорость работы системы существенно снижается. Поэтому в использовании аудита обычно стараются найти разумный компромисс между полнотой контроля и приемлемым снижением скорости работы.

2.4.8 Защита от халатности пользователей.

Как правило, ОС имеют в своём составе средства для поддержания списка пользователей, прав и паролей в состоянии, которое затрудняет несанкционированное проникновение в систему. Пользователи часто пренебрегают правилами безопасности либо из-за недостаточного понимания принципов функционирования системы разграничения доступа, либо (чаще всего) просто пытаясь максимально облегчить себе работу. Они могут не задавать пароль (использовать в качестве пароля пустую строку, чтобы не было необходимости каждый раз вводить слово пароля), задавать простые пароли (из одной-трёх букв), не менять пароли в течение длительного времени, постоянно использовать как пароли два-три «любимых» слова, чередуя их. Все эти действия приводят к тому, что становится возможным несанкционированное проникновение в систему путём простейшего подбора пароля из числа нескольких известных слов и выражений. Поэтому и созданы технические средства, призванные принудить пользователя соблюдать правила безопасности даже тогда, когда он сам не хочет этого делать.

Прежде всего, любая система контроля доступа имеет возможность ограничить максимальное число попыток ввода пароля при входе в систему (обычно двумя-тремя попытками). Если за заданное число попыток пользователь не смог ввести правильный пароль, то его учётная запись блокируется, либо на определённое, заранее заданное время, либо до разблокировки её администратором. Это ограничение позволяет предотвратить проникновение в систему путём подбора пароля «вручную» (когда злоумышленник просто пробует одно слово за другим, надеясь, что попадётся нужное), поскольку после двух-трёх неудачных попыток пользователь, под именем которого пытались войти, блокируется, а угадать хорошо придуманный пароль за несколько

попыток обычно нереально.

Возможно задание системе ограничения на минимальную длину пароля. Если пользователь пытается установить себе более короткий пароль, то этот пароль просто не принимается системой. Считается, что для относительно надёжной защиты достаточно пароля из не менее чем шести символов, причём не являющегося осмысленным словом и содержащего буквы в разных регистрах и цифры.

Можно также установить максимальное время действия пароля, по истечении которого ОС потребует от пользователя поменять пароль, а без изменения пароля откажется работать. Это потребует от пользователя менять пароли с определённой частотой, что ограничит возможность несанкционированного проникновения за счёт случайного узнавания пароля.

Требование на обязательную периодическую смену пароля некоторые пользователи стараются обойти следующим путём. Когда система требует поменять пароль, пользователь его изменяет, после чего сразу же снова меняет новый пароль на старый. Также часто используют два-три «любимых» пароля, чередуя их. Для предотвращения этого в ОС может быть предусмотрена возможность хранения нескольких последних паролей и запрета на повторный ввод пароля, находящегося в списке. Это вынуждает пользователя использовать попеременно, по крайней мере, несколько разных паролей.

Есть и другие, более специфические методы контроля. Так, перед сменой пароля может автоматически проводиться проверка на простоту подбора. Обычно недопустимо простыми считаются пароли, состоящие из последовательности одних и тех же символов или пароли, которые набираются нажатием на клавиатуре последовательно (во всех направлениях: слева направо, справа налево, сверху вниз, снизу вверх) нескольких символов в одном ряду. Кроме того, в системах, где имеются стандартные средства проверки орфографии, программа проверки пароля может признавать недопустимо простыми пароли, которые совпадают с одним из слов системного орфографического словаря. Такой порядок практически исключает использование в качестве паролей коротких осмысленных слов, имён, широко распространённых названий и сокращений.

2.4.9 Защита учётной записи администратора.

При инсталляции ОС в её подсистеме администрирования, как правило, создаётся некоторое количество стандартных учётных записей пользователей. Как минимум, ОС автоматически создаёт учётную запись главного администратора. Главному администратору приписывается стандартное имя входа («Администратор», «Administrator», «ADMIN», «SYSTEM», «SUPERVISOR», «ROOT», «MASTER» или какое-нибудь ещё), а иногда ещё и стандартный пароль. Права главного администратора таковы, что он может регистрировать пользователей, предоставлять им права и создавать профили.

Прежде чем начинать работу с системой, администратор должен создать новую учётную запись, приписав ей все права главного администратора. Имя входа не должно совпадать ни с одним из распространённых стандартных имён входа с административными правами. Лучше всего, если это имя будет вообще случайной комбинацией

символов. Пароль нужно выбирать так, чтобы его невозможно было угадать. Лучшим вариантом является случайная комбинация не менее чем шести-семи буквенно-цифровых символов, не составляющих осмысленного слова, причём в пароле должны присутствовать буквы, набранные как в верхнем, так и в нижнем регистре. Ни в коем случае не следует выбирать в качестве пароля собственное имя и его производные, "перевёрнутое"имя входа, имена близких, знакомых, широко известных людей, названия, любые осмысленные слова, памятные даты, элементы домашнего адреса, номер машины или телефона. Пароль следует периодически менять.

После создания новой учётной записи администратор должен выйти из системы, войти под именем только что созданного пользователя и уничтожить стандартную учётную запись администратора. Эта операция необходима потому, что имя стандартной учётной записи широко известно, а значит, при попытке взлома системы оно обязательно будет использовано злоумышленником. Вход под именем администратора даст взломщику возможность выполнять в системе любые операции. Уничтожив стандартную учётную запись, администратор лишает взломщика такой возможности. При попытке взлома системы придётся не только подбирать пароль, но и угадывать, под каким именем работает администратор, что практически невозможно, если администратор держит имя входа в тайне. Таким же образом должны быть уничтожены и созданы под другими именами все остальные стандартные учётные записи.

Если ОС не позволяет уничтожать стандартные учётные записи, то их следует заблокировать, а если и это невозможно, то им следует приписать пароли, представляющие собой случайные последовательности буквенных и цифровых символов максимально возможной длины.

Если есть техническая возможность, на стандартные имена входа можно установить ловушки — настроить систему так, чтобы при попытке входа под одним из этих имён автоматически фиксировались все сведения о пользователе, попытавшемся проникнуть в систему.

Если ОС поддерживает возможность раздельного администрирования, то её имеет смысл использовать. Некоторое усложнение административных процедур, вызванное наличием двух администраторов, будет скомпенсировано повышением стойкости системы и лучшим контролем за действиями администраторов.

2.4.10 Управление правами пользователей.

В зависимости от характера работы пользователя ему необходимо предоставить права доступа к системным ресурсам. Наиболее удобным является создание для каждого вида работы отдельного профиля, в котором указаны все необходимые права. Пользователю, выполняющему в системе работу определённого вида, просто приписывается соответствующий профиль. Если пользователь выполняет несколько видов работ, то, в зависимости от возможностей ОС и требуемого режима работы пользователя, можно предоставить ему необходимые права одним из следующих способов.

- Можно приписать учётной записи пользователя два или более профиля, если

это допускает система.

- Можно создать отдельный профиль для нужной комбинации прав и приписать этот профиль учётной записи пользователя.
- Можно просто предоставить пользователю все необходимые права, не создавая для этого профиля.
- Наконец, можно создать для пользователя две отдельные учётные записи. Для выполнения работы каждого вида пользователь будет входить в систему под нужным именем.

Если работник выполняет некоторые текущие работы с использованием системы и одновременно является её администратором, то у него обязательно должны быть две различные учётные записи: одна — для выполнения административных функций, другая — для прочей работы. Причём список прав второй учётной записи не должен включать права администратора. Это ограничение снижает вероятность того, что во время выполнения текущей работы администратор повредит систему.

Если необходимо хранить файлы пользователя, для них должен быть создан отдельный персональный каталог. Пользователю нужно выделить достаточную квоту дискового пространства. При первоначальной настройке списка пользователей не следует выделять квоты так, чтобы было распределено всё свободное дисковое пространство, поскольку в дальнейшем при увеличении квоты одного из пользователей или регистрации нового пользователя придётся сокращать квоты других пользователей, что явно нежелательно. Гораздо лучше оставить максимально большой объём нераспределённым, увеличивая затем квоты по мере необходимости.

Для контроля действий пользователей необходимо использовать систему аудита. Все действия пользователей, кроме манипуляций с файлами в личном каталоге, должны протоколироваться.

2.4.11 Организационное обеспечение безопасности системы.

Все вышеописанные методы администрирования имеют одно общее свойство: они являются техническими. Их применение заключается в правильной настройке и эксплуатации имеющихся в системе техинческих средств защиты. Но, к сожалению, для того, чтобы гарантировать безопасность системы, технических средств недостаточно. Необходимы организационные меры, направленные на поддержание определённого режима работы с системой и устанавливающие меру ответственности каждого работника в случае каких-либо нарушений. Технические работники часто недооценивают организационные меры обеспечения безопасности, и в этом их большая ошибка, потому что чисто техническими мерами гарантировать безопасность системы *невозможно*.

Порядок организации работы администратора и пользователей определяется специфическими особенностями организации, использующей вычислительную систему. Тем не менее, существует ряд общих рекомендаций по организационным вопросам

этого использования. Некоторые из них противоречат установившейся практике использования вычислительной техники, однако, при ближайшем рассмотрении оказывается, что эта устоявшаяся практика не лучшим образом отражается на эффективности использовании системы и производственной дисциплине. Рассмотрим некоторые из рекомендаций.

Прежде всего, у системы должен быть один, назначенный соответствующим распоряжением руководства, главный администратор. Главный администратор является лицом с наибольшими техническими правами в системе. Он несёт ответственность за работу системы. У системы не может быть двух или более главных администраторов, поскольку наличие нескольких равно ответственных лиц затрудняет, а иногда и делает невозможным однозначное определение виновного в случае нарушения работы системы. Естественно, если используется система раздельного администрирования, то должен быть один главный администратор системы, и один главный администратор безопасности, причём ни в коем случае нельзя допускать, чтобы обе эти должности занимал один и тот же человек.

Если режим работы требует, возможно наличие ещё одного или нескольких работников с административными правами. Это могут быть администраторы домена, администраторы определённых серверов или приложений, например, администратор СУБД. Права каждого из них должны ограничиваться сферой их ответственности.

Если несколько администраторов работают с системой последовательно (например, при посменном дежурстве), то каждый из них при сдаче смены должен полностью информировать сменщика обо всех произведённых в системе изменениях. Желательно ведение журнала изменений и ознакомление с ним каждого очередного дежурного перед началом дежурства под расписку. При несоблюдении этого правила становится обычной следующая ситуация. Выявляется ошибка в действиях администратора. Выяснение обстоятельств показывает, что действия допустившего ошибку дежурного были сами по себе правильными, но, в силу изменений, произведённых предыдущим дежурным, они оказались неверными. Естественно, при разбирательстве один из дежурных будет утверждать, что он поставил сменщика в известность о сделанных изменениях, а другой — что ему никто ничего не говорил. Найти объективно виновного, таким образом, становится невозможно, так что наказывают либо обоих, либо никого. При наличии журнала изменений проблемы не возникнет. Если изменение записано в журнале, то виновен работник, допустивший ошибку, если нет — то работник, сделавший изменение и не зарегистрировавший его.

Администратор должен нести ответственность за любую попытку злоупотребления своими правами. В большинстве организаций администратора при обнаружении злоупотреблений сначала несколько раз предупреждают и наказывают, и только после нескольких нарушений увольняют с работы. Такая практика, фактически, является разрешением на злоупотребления до того момента, когда нарушитель будет первый раз пойман. Поэтому при выявлении факта злоупотребления администратор должен быть немедленно уволен с работы, безо всяких предварительных предупреждений. Эта практика должна быть официально закреплена соответствующими внутренними документами организации. Администратор должен быть поставлен в известность о

ней перед поступлением на работу.

Работы, которые будут выполняться пользователями в системе, и, соответственно, основные права пользователей, должны определяться руководством. Администратор не должен иметь права по своему личному усмотрению расширять или урезать права пользователей. Пользователь, нуждающийся в расширении прав, должен обращаться не к администратору, а к соответствующему руководителю, а руководитель должен отдавать распоряжение администратору, либо (если администратор ему прямо не подчиняется) обращаться за этим к непосредственному начальнику администратора. Если администратор считает, что у пользователя имеются избыточные для его работы права, то он должен обратиться к руководителю, который может либо подтвердить необходимость наличия имеющихся прав, либо распорядиться об их уменьшении. Это дополнительное усложнение процесса изменения прав пользователей необходимо. Если администратор имеет право по собственному усмотрению расширять и урезать права пользователей, то его затруднительно привлечь к ответственности за предоставление избыточных прав или необоснованное урезание прав кого-либо из пользователей, поскольку это неизбежно приведёт к спорам об обоснованности таких действий. Кроме того, рекомендуемый порядок работы исключает любые споры о необходимости предоставления или урезания прав между пользователем и администратором. Единственным допустимым исключением из этого правила является решение вопроса о расширении квоты дискового пространства пользователя. Право самостоятельного решения этого вопроса вполне можно предоставить администратору, поскольку он (вопрос) имеет обычно чисто технический характер.

Пользователи, работающие с системой, должны нести ответственность за свои некорректные действия. Нужно обратить внимание на то, что, как правило, невозможно техническими средствами проконтролировать, какой именно человек работает с системой. Для системы существует только имя входа и пароль. Технически невозможно отличить действия пользователя, работающего под своим именем, от действий взломщика, использовавшего чужое имя входа и пароль. Вследствие этого необходимо административным решением ввести такой порядок ответственности пользователей, чтобы пользователь нёс ответственность за все действия, которые совершены в системе под его именем. При работе с ответственной информацией желательно перед допуском пользователя к работе получить с него письменный документ, подтверждающий его согласие работать на таких условиях. Описанный порядок чётко определяет ответственного за любые некорректные действия и побуждает пользователей к соблюдению правил безопасности. Кроме того, он устраняет возможность возникновения споров об истинных виновниках того или иного нарушения, поскольку виновником автоматически считается тот, чьи некорректные действия зафиксированы в системном журнале.

Глава 3

Функционирование операционной системы.

3.1 Инсталляция ОС.

3.1.1 Инсталляция ОС на компьютер, не содержащий другой ОС.

Перед началом использования ОС необходимо разместить на устройствах хранения информации модули, составляющие ОС, записать модуль загрузки в нужное место, настроить систему на тип компьютера и внешних устройств, подключенных к нему, скопировать на диски сопутствующие программы. Этот процесс называется инсталляцией (англ. Installation — установка).

Инсталляция ОС отличается от инсталляции любой другой программы тем, что ОС должна устанавливаться на компьютер, на котором нет вообще никакого программного обеспечения. Компьютер может не иметь размеченных разделов на жёстких дисках, разделы могут не соответствовать требованиям ОС, могут быть не отформатированы. Поэтому процедура инсталляции ОС обычно состоит из нескольких этапов.

Загрузка инсталляционного модуля и копирования базовых файлов ОС. В инсталляционный пакет любой ОС обязательно входит хотя бы один съёмный носитель данных для устройства, имеющегося в компьютере, на который будет устанавливаться система. До последнего времени в качестве такого носителя обычно использовалась одна или несколько дискет. Сейчас, когда большинство компьютеров стали снабжаться зашитой в ПЗУ программой загрузки с оптического диска, более распространённым вариантом стал CD-ROM, не имеющий столь жёстких ограничений по объёму. На носителе находится стартовый модуль, который может опознать процедура загрузки ОС. Инсталляция ОС начинается с помещения этого носителя в считывающее устройство и перезагрузки системы. После перезагрузки с носителя считывается стартовый модуль, который загружает в память первую часть программы установки ОС. Как правило, эта программа не содержит ядра ОС, она предназначена только для того, чтобы скопировать файлы ядра с носителя, на котором они поставляются, на жёсткий диск

компьютера. Носителем, с которого копируются файлы, может быть тот же носитель, с которого была запущена программа установки, но может быть и другое устройство. Так, первые версии ОС Windows95 поставлялись на одном CD-ROM и двух дискетах. С дискет загружалась программа установки, а все прочие данные извлекались с CD-ROM. Как правило, программа установки перед копированием файлов на жёсткий диск предоставляет возможность разметить диски системы и отформатировать нужные разделы. Кроме того, программа установки записывает данные в область, проверяемую загрузчиком, чтобы загрузчик мог найти и загрузить ОС при старте системы. Уже на этом этапе программа инсталляции может потребовать у пользователя указать список драйверов и утилит, которые требуется установить.

После выполнения первого этапа инсталляции система обычно перезагружается (либо производится загрузка ядра системы без перезагрузки компьютера). Затем запускается программа инсталляции, работающая уже с использованием средств, содержащихся в ядре ОС. Эта программа устанавливает необходимые драйверы устройств и дополнительные программы.

Затем производится запуск ОС (либо непосредственно, либо после перезагрузки системы). Возможно, на этом этапе будет запущена какая-либо программа, предназначенная для настройки системы. После того, как все операции выполнены и ОС выдала приглашение пользователю на ввод команд, инсталляция считается завершённой.

Все перечисленные операции могут и не выполняться программой инсталляции. Обязателен лишь первый этап, на котором производится запись стартового модуля и перемещение модулей ядра ОС на диск компьютера. Прочие этапы программа инсталляции может и не предусматривать. В этом случае пользователь должен сам скопировать все второстепенные файлы на диск компьютера и настроить их.

3.1.2 Инсталляция поверх другой ОС.

Если на компьютере уже установлена другая ОС, то процедура инсталляции сильно упрощается. Достаточно загрузить систему и запустить из-под неё программу инсталляции. Эта программа выполнит операции, указанные в первом пункте списка, приведённого в предыдущем разделе. После этого она перезагрузит систему либо запустит ядро новой ОС. Затем инсталляция новой ОС может продолжаться далее в том же порядке, что и на компьютер, на котором не была ранее установлена ОС. При инсталляции новой ОС поверх старой в загрузочной области устройства, с которого будет производиться загрузка, старая запись (соответствующая предыдущей ОС) заменяется новой. Соответственно, после перезагрузки будет загружена уже новая ОС.

Иногда фирмы-производители выпускают специальные версии дистрибутивов ОС, рассчитанные исключительно на такой способ инсталляции (так называемые Upgrades — обновления). Делается это с различными целями. Обновления проще распространять, чем полноценные инсталляционные пакеты, поскольку они меньше по объёму. Кроме того, такой вариант системы может быть установлен только пользователем,

который уже имеет в своём распоряжении определённую базовую ОС (обычно той же фирмы).

3.1.3 Инсталляция новой ОС в дополнение к уже установленной. Менеджеры загрузки.

Некоторые ОС можно устанавливать на один и тот же компьютер так, что пользователь сможет попеременно работать то с одной системой, то с другой.

Существует два основных способа использования на одном компьютере нескольких ОС. В обоих случаях ОС последовательно устанавливаются на один компьютер, причём таким образом, что при инсталляции следующей системы предыдущая не разрушается. Если ОС используют один и тот же тип файловой системы, то они могут располагаться в одном и том же разделе одного и того же жёсткого диска. В противном случае обязательным является расположение ОС в разных разделах или на разных устройствах.

Первый способ совместного использования нескольких ОС состоит в установке специальных программ-загрузчиков, которые, будучи запущены из-под ОС одного типа, запускают ОС другого типа. Так, существуют программы, которые могут из-под ОС MS-DOS загрузить ОС Linux. Для использования таких программ, как правило, необходимо, чтобы модули ядра запускаемой ОС находились на устройстве, доступном программе-загрузчику. Способ использования программ-загрузчиков не особенно удобен, так как он может потребовать дублирования ядра в файловой системе другой ОС, и, кроме того, требует для каждой пары ОС, которые должны запускать друг друга, отдельной программы-загрузчика.

Второй способ состоит в использовании специальных программ, называемых менеджерами загрузки (boot managers). Менеджер загрузки записывается на загрузочное устройство, туда, куда, по стандартам данного типа компьютеров, должен устанавливаться модуль загрузки ОС. Модули загрузки ОС после установки менеджера загрузки помещаются либо в файлы на диске, либо каждый в свой раздел. Когда система перезагружается, загрузчик вместо стартового модуля ОС загружает и запускает модуль менеджера загрузки. Менеджер загрузки выдаёт пользователю запрос ОС, которую требуется запустить. После того, как пользователь выберет запускаемую ОС, менеджер загрузки находит стартовый модуль соответствующей ОС и запускает его. Далее ОС загружается обычным образом.

Менеджеры загрузки могут быть специализированными или универсальными. Универсальный менеджер загрузки позволяет настроить себя для запуска любого количества разнотипных ОС. При этом ОС, как правило, должны располагаться в отдельных разделах жёсткого диска системы. Примером такого загрузчика может служить Boot Manager, поставляемый с ОС OS/2 или загрузчик LILO, поставляемый с ОС Linux Red Hat. Специализированный менеджер загрузки предназначен для запуска только ОС определённого типа. Примерами таких менеджеров загрузки являются встроенные

менеджеры систем WindowsNT [1] или Windows 95/98. Эти менеджеры настраиваются при инсталляции системы так, чтобы обеспечить возможность запуска ранее установленной в системе ОС MS-DOS.

3.2 Загрузка и выгрузка операционной системы.

Прежде чем ОС начнёт функционировать на компьютере, она должна быть загружена. Как правило, эта операция выполняется непосредственно после включения питания компьютера (или после перезагрузки). Рассмотрим процесс запуска компьютера и загрузки ОС.

Весь процесс запуска компьютера можно грубо разделить на два этапа – запуск собственно компьютера и запуск ОС.

После включения питания компьютера процессор запускается и начинает выполнять программу, расположенную по некоторому адресу в памяти (обычно, в ПЗУ). Начальный адрес, естественно, определяется конструкцией системы. Обычно работа начинается с выполнения процедур самотестирования системы. Может быть-выполнено также определение конфигурации системы, опознание подключенных ВУ и их проверка. После этого начинается этап загрузки ОС.

Для загрузки ОС программа-загрузчик, запускаемая после процедур самотестирования системы, опрашивает внешние устройства, которые могут быть использованы для загрузки ОС. Обычно это дисководы для гибких дисков, жёсткие диски и накопитель CD-ROM. Кроме того, может опрашиваться сеть (если компьютер подключен к сети и программа-загрузчик может запустить процедуру загрузки из сети) и ZIP-накопитель. Возможна попытка загрузки и с других устройств, способных хранить данные. Конкретный тип устройства, с которого будет загружаться система, не имеет значения. Важно лишь, чтобы в ПЗУ системы имелись процедуры для считывания данных с этого устройства, а само устройство содержало необходимую для загрузки ОС информацию. Конкретный набор опрашиваемых устройств и порядок их опроса определяется конструкцией и настройкой системы. Обычно система пытается загрузить ОС с устройства, которое указано в настройках как первое загрузочное, а если это устройство не содержит ОС, то продолжает попытки для следующих в списке устройств, пока не будет загружена ОС или пока список устройств не закончится.

Обычно загрузчик считает, что опрашиваемое устройство содержит ОС, готовую для загрузки, если определённый блок данных, прочитанный с устройства, содержит некоторую заданную последовательность байтов. Эта последовательность и её местонахождение на устройстве задаётся стандартом на данный тип компьютеров, и любая ОС, предназначенная для компьютеров этого типа, обязана записывать эту последовательность байтов в заданную область устройства загрузки, чтобы система могла запустить эту ОС.

[1] Менеджер загрузки WindowsNT позволяет загружать и любые другие ОС, но он зависит от самой NT и в случае нарушения её загрузчика работать не будет.

Когда устройство загрузки найдено, загрузчик считывает с него стартовый модуль ОС. Этот модуль принято также называть загрузчиком, но мы будем использовать термин «стартовый модуль», чтобы не путать его с программой-загрузчиком, зашитой в ПЗУ. Стартовый модуль имеет определённый, обычно очень небольшой, размер, и располагается в определённом, стандартизованном месте устройства. После загрузки стартовый модуль запускается.

Стартовый модуль производит поиск компонентов запускаемой ОС и их загрузку. Конкретная процедура запуска при этом зависит уже от типа ОС. В ходе этой процедуры обычно загружается ядро системы, драйверы устройств и оболочка, обеспечивающая интерфейс пользователя. Поскольку стартовый модуль обычно слишком мал, чтобы поместить в него весь код загрузки ОС, загрузка, чаще всего, является многоступенчатой операцией. Стартовый модуль находит на устройстве и загружает более мощную и имеющую больший размер программу загрузки ОС. Эта программа загружает ядро и, в свою очередь, передаёт управление ему. Ядро инициирует загрузку драйверов и вспомогательных программ, после чего загружает интерфейсный модуль. Моментом завершения процесса загрузки обычно считают момент, когда пользователю выдаётся приглашение на ввод команд.

По завершении работы с компьютером ОС должна быть выгружена из памяти. Конечно, компьютер можно просто выключить, но абсолютное большинство ныне существующих ОС при этом могут в большей или меньшей степени пострадать. Почему?

Во время работы ОС часть данных, которые должны быть записаны на диск, всегда находится в оперативной памяти — в дисковом кэше. При внезапном выключении компьютера кэш не будет сброшен на диск, в результате часть данных, которые программы и пользователь уже считают сохранёнными, будут потеряны. Не будут закрыты открытые файлы, не будут остановлены корректным образом процессы, работающие с дисками. Всё это может привести к нарушению целостности файловой системы в целом или отдельных файлов. Хуже всего то, что может нарушиться целостность системных файлов, хранящих параметры настройки самой ОС. При некоторых из таких ошибок система просто не сможет больше загрузиться. Например, в ОС фирмы Microsoft настройки системы и многих программ хранятся в системном реестре. Сам реестр — это файл специального недокументированного формата, хранящийся на диске. Если из-за каких-либо сбоев будет разрушен файл реестра, система при загрузке не сможет восстановить настройку.

Конечно, в современных ОС, как уже говорилось выше, предпринимаются достаточно серьёзные меры для защиты системы от сбоев, вызванных внезапным выключением компьютера. Файловые системы проектируются так, чтобы не разрушаться при случайном выключении питания. Наиболее важные данные дублируются в нескольких различных файлах, после некоторых операций принудительно выполняются команды сброса кэша. Но, тем не менее, стопроцентно надёжной защиты от таких нарушений быть не может. В любом случае, потеря данных из кэша при отключении питания неизбежна. Можно, конечно, запретить системе отложенную запись на диск, но такой запрет ударит (причём довольно чувствительно) по производительности системы в целом.

Так что при завершении работы система должна быть выгружена из памяти правильным образом. Как производится выгрузка? Пользователь должен дать команду на остановку компьютера. Это может быть выбор пункта системного меню при использовании визуального интерфейса, или какая-то текстовая команда. После получения команды на остановку (или перезагрузку, которая выполняется почти также) система выполняет ряд последовательных операций.

- Прежде всего, должны быть завершены все программы пользователя. Система может либо завершить их принудительно, либо попросить пользователя выгрузить их, предоставив для этого некоторое, обычно небольшое, время.
- После завершения программ пользователя система выгружает некоторые свои собственные программы, ненужные на этапе завершения работы.
- Производится остановка системных служб.
- Затем производится сброс кэша, в результате чего все данные, которые должны быть записаны на диски, физически переносятся по месту назначения.
- Обычно после загрузки системы куда-то на диск записывается специальная метка, наличие которой в момент загрузки означает, что в прошлый раз система не была выгружена корректным образом. На заключительном этапе выгрузки системы такие метки стираются.
- После всех этих действий либо происходит выдача пользователю приглашения на выключение компьютера, либо перезагрузка (если была дана команда перезагрузки, а не остановки), либо просто выполняется команда выключения питания (если, конечно, аппаратная часть системы поддерживает такую возможность).

3.3 Исполнение прикладных программ.

С точки зрения ОС программа, которая может выполняться в системе, представляет собой просто файл определённого вида. Обычно такой файл состоит из заголовка и объектного кода. Заголовок — это последовательность данных, имеющая вполне определённый формат, задаваемый ОС. Как правило, в заголовке указываются некоторые параметры программы, необходимые для правильной загрузки программы в память и корректного запуска программы (размер программы, её контрольная сумма, начальный адрес загрузки, начальный адрес исполнения и пр.). Следующий за заголовком объектный код и есть собственно программа, представленная в виде машинного кода того процессора, на котором она выполняется.

Важно понять, что далеко не каждый файл, содержимое которого может быть исполнено, является программой с точки зрения ОС. Файл может содержать программу не в объектном коде, а в виде исходного текста (например, программу на языке

BASIC), либо содержать документ, который должен обрабатываться какой-либо другой программой. Сама ОС не в состоянии выполнить такой файл как программу, она может лишь запустить на выполнение какую-нибудь другую программу, которая сможет обработать этот файл. Всё, что будет говориться далее, относится исключительно к программам, которые может непосредственно выполнить ОС, то есть к программам в объектном коде и со стандартным заголовком.

ОС может поддерживать один или несколько форматов исполняемых файлов. Программу в любом из этих форматов ядро ОС в состоянии загрузить в память и запустить. По команде запуска программы ОС находит содержащий её файл, открывает его на чтение и по заголовку пытается идентифицировать формат программы. Если по заголовку невозможно установить формат файла, то предполагается, что файл не содержит программы, и ядро возвращает ошибку. В противном случае начинается процесс загрузки программы в память. Загрузка обычно проводится в несколько этапов.

Первый этап — перемещение объектного кода программного модуля с дискового носителя в оперативную память. На этом этапе содержимое файла программы размещается либо в общем адресном пространстве системы, либо в отдельном адресном пространстве, созданном ядром ОС для запускаемой программы. Загрузка программы вовсе не обязательно предполагает физическое перемещение всего объектного кода в ОЗУ. ОС может загружать код программы в память по мере её исполнения. Так, например, в ОС WindowsNT и Windows95 файл программы открывается как файл, проецируемый в память, то есть его содержимое просто проецируется на адресное пространство, без физического перемещения данных. Когда участок кода требуется для загрузки команд в процессор, подсистема поддержки виртуальной памяти извлекает из файла соответствующий фрагмент и перемещает его в ОЗУ. При такой организации загрузки возможно исполнение в системе программы, физический объём которой превышает объём оперативной памяти.

После загрузки могут следовать другие этапы, на которых может производиться преобразование кода программы и подготовка его к исполнению. В частности, может производиться пересчёт таблицы ссылок. Дело в том, что для большинства ОС программы пишутся так, чтобы их можно было загрузить в память с произвольного начального адреса. Это приводит к невозможности использования в программах команд передачи управления по абсолютному адресу. Для того, чтобы обеспечить возможность использования в программах команд перехода по абсолютному адресу, но при этом сохранить и возможность загрузки программ с произвольного адреса, могут использоваться таблицы ссылок. Эта таблица включается в заголовок файла программы и содержит адреса команд перехода по абсолютным адресам, имеющихся в программе. После загрузки на основании этой таблицы производится перерасчёт всех адресов перехода в программе, в результате чего программа остаётся полностью корректной, независимо от того, с какого адреса она загружается.

После того, как программа загружена и обработана нужным образом, она запускается. Адрес команды, с которой должно начаться исполнение программы, обычно также извлекается из заголовка файла. Программа начинает выполняться и выполня-

ется до завершения либо до возникновения критической ошибки, которую она не в состоянии обработать.

Программа завершается обычно вызовом некоторой системной функции, приводящей к снятию программы с процессора и выгрузке её из памяти. Чаще всего при написании программ на языках высокого уровня нет необходимости специально вызывать эту функцию, поскольку её вызов автоматически вставляется в программу компилятором. При завершении программы ОС может самостоятельно освободить все системные ресурсы, которые были захвачены и не освобождены программой. В большинстве систем программа при завершении может вернуть в систему так называемый код возврата. Обычно это целое число, значение которого может быть задано программой перед завершением и проверено ОС после завершения программы. Обычно код возврата используется для того, чтобы сигнализировать системе или главной программе об условиях завершения программы (одно значение может соответствовать корректному завершению, другое — ошибке и так далее).

3.4 Динамическая компоновка.

В большинстве операционных систем существует возможность использования программ, различные части которых располагаются в различных файлах. Первоначально при запуске загружается главный модуль программы. Из главного модуля могут вызываться фрагменты, расположенные в других модулях. Обычно сама ОС не обеспечивает автоматическую загрузку нужных модулей, а просто требует, чтобы соответствующий модуль перед вызовом был загружен. Такая загрузка выполняется вызовом одной из функций API ОС.

Существует два варианта организации загрузки модулей. В первом варианте на этапе компиляции программы компилятору указывается, какие внешние модули будет использовать программа, и компилятор сам добавляет в стартовый код программы вызовы процедур загрузки для них. При этом программист освобождается от необходимости самостоятельно загружать библиотеки. Кроме того, он может быть совершенно уверен, что для любой вызываемой им библиотечной функции соответствующий модуль на момент вызова гарантированно загружен. Однако все модули в этом случае загружаются при старте программы и выгружаются перед её завершением, что в некоторых случаях бывает не особенно удобно.

Во втором варианте программист должен самостоятельно выполнить загрузку библиотек, явно вызвав из программы функции загрузки модулей. Второй вариант несколько более громоздок, зато и более экономен, поскольку позволяет избежать загрузки модулей, которые в данный момент не требуются, и даёт возможность выгрузить модуль сразу после завершения его использования. Недостатком этого метода является то, что в результате ошибки программиста в некоторых случаях могут возникать ситуации, когда в момент вызова некоторой библиотечной функции модуль, в котором она располагается, не загружен. Естественно, такой вызов приведёт к ошибке.

Изначально динамическая компоновка появилась как средство написания про-

грамм, размер которых превышает объём ОЗУ системы. В настоящее время эта проблема решается системами виртуальной памяти, так что формально размер программы, располагающейся в одном файле, практически не ограничен, но динамическая компоновка используется всё равно. Её преимущество состоит в экономии ресурсов (можно не загружать модули программы, которые не требуются в данный момент, а, загрузив, выгружать из сразу после завершения использования). Кроме того, если многие программы используют один и тот же загружаемый модуль, то этот модуль может храниться в системе в единственном экземпляре. При статической компоновке в каждом программном файле должны были бы находиться копии соответствующих модулей. По этой причине в виде загружаемых модулей часто оформляют различного рода стандартные библиотеки, поставляемые вместе с системами разработки программ. Как правило, несколько программ, разработанных в одной среде, могут использовать одни и те же динамические библиотеки. Ещё одним преимуществом является возможность исправления и доработки динамических библиотек без изменения и перекомпиляции основных модулей.

Недостатки динамической компоновки являются следствием её же преимуществ. Если несколько программ требуют одну и ту же динамическую библиотеку, но различных версий (например, если одна из программ использует какую-либо ошибку в коде библиотеки, впоследствии исправленную) то приходится либо хранить для каждой из программ отдельную копию «своей» версии библиотеки, либо отказаться от использования одной из программ. Кроме того, если использующая библиотеку программа всего одна, то суммарный объём файла программы и файла загружаемого модуля, как правило, больше, чем объём той же программы, написанной без использования динамической компоновки. Ясно также, что на загрузку динамической библиотеки затрачивается дополнительное время.

3.5 Сервисы, демоны, фоновые задачи.

Для решения своих задач пользователь запускает прикладные программы, работает с ними, использует их результаты. Исполнение этих программ наглядно наблюдается, поскольку они осуществляют ввод-вывод, ведут диалог с пользователем. Есть, однако, класс программ, которые пользователю не видны, и о существовании которых пользователь в некоторых случаях может даже не догадываться. Эти программы называются демонами.

Термин «демон» (daemon) — это просто сокращение от фразы Disk And Execution MONitor — монитор дисков и исполнения. Эта терминология используется в отношении ОС UNIX и базирующихся на UNIX систем. Для пользователей DOS более привычным будет термин «резидентная программа» (хотя он и не вполне точно соответствует понятию демона), для пользователей WindowsNT — «сервис». Демоны обычно запускаются самой операционной системой, причём ещё до того, как в системе зарегистрируется пользователь, хотя возможен запуск демонов и по команде администратора. Демоны часто не осуществляют обычного ввода-вывода. Взаимодей-

ствовать с ними можно только специально предназначенными для этого системными средствами.

Основное назначение демонов — реагировать на те или иные события, происходящие в системе. Например, некий демон может быть занят отслеживанием появления в определённом каталоге файлов и, как только файлы появляются, выполнять с ними некоторое действие. Так, в Unix стандартная система вывода текстов на принтер реализована как демон. Программы, осуществляющие печать, просто формируют файлы заданного формата и помещают их в определённый системный каталог (для чего выполняют стандартную системную команду lp). А демон печати, если он запущен, при появлении в заданном каталоге файла определяет его формат и в соответствии с этим форматом отправляет файл на принтер. Иногда приложение, вызвавшее событие, может даже не знать о том, что данное событие приведёт к активизации того или иного демона.

Демоны (сервисы) — это один из видов так называемых *фоновых задач*. Фоновая задача — это задача, которая запускается и работает, выполняя какие-то свои действия, либо ожидая воздействия других задач, без непосредственного взаимодействия с пользователем. Обычно фоновые задачи либо выполняют какие-то несрочные операции, либо являются серверными приложениями, используемыми другими программами. Примером первого типа является программа дефрагментации диска во время работы системы, в те моменты, когда к диску никто не обращается. Примером второго типа — любой сервер СУБД. Выделять демонов в отдельный подкласс из прочих фоновых задач имеет смысл потому, что демоны имеют отличительные особенности — принципиальное отсутствие диалога с пользователем и активизации по событию. А вообще говоря, фоновая задача может взаимодействовать с пользователем и с помощью стандартных интерфейсных средств ("может— не обязательно значит "взаимодействует").

Глава 4

Некоторые конкретные типы ОС и их особенности.

В этой главе мы кратко рассмотрим некоторые типы операционных систем, которые в настоящее время используются на персональных компьютерах. Рассмотрение будет поверхностным, поскольку глубокое изучение каждой из упомянутых ОС потребовало бы отдельного учебного курса довольно большого объёма.

4.1 MS-DOS.

Операционная система MS-DOS фирмы Microsoft появилась в начале 80-х годов, одновременно с началом производства персональных компьютеров IBM PC. Непосредственным предком этой системы является ОС CP/M, распространённая в то время на восьмиразрядных ПЭВМ.

Система MS-DOS изначально была ориентирована на ПЭВМ с процессором Intel 8086(8088) и до настоящего момента работает на компьютерах с процессорами семейства Intel x86, Pentium, Pentium Pro, Pentium II-4, а также совместимыми с ними. Эта ОС — однопользовательская и однозадачная. Объём адресного пространства — 1 Мбайт, из которого младшие 640 Кбайт предназначены для прикладных программ, а оставшаяся часть отводится под размещение ПЗУ, видеопамяти и прочие служебные нужды. MS-DOS позволяет загружать в память одновременно несколько программ, но она не управляет распределением процессорного времени между ними. Когда одна из программ выполняется, все остальные находящиеся в памяти программы простаивают. Программа, находящаяся в памяти, но не выполняющаяся, может быть активизирована по прерыванию (с использованием механизма прерываний процессоров Intel x86). Все программы располагаются в общем адресном пространстве, никаких средств защиты памяти не предусмотрено.

Под управлением MS-DOS программы могут свободно обращаться к оборудованию компьютера и максимально использовать все его возможности. Это связано с тем, что в однозадачной системе нет никаких ограничений, направленных на обес-

печение безопасной одновременной работы многих программ, следовательно, нет и непроизводительных затрат времени на их реализацию.

Основной вид интерфейса пользователя — текстовые команды. Существует довольно много дополнительных программных пакетов, обеспечивающих работу с MS-DOS через визуальный интерфейс, работающих как в текстовом, так и в графическом режиме. Надстройка Windows обеспечивает графический визуальный интерфейс пользователя, позволяет выполнять одновременно несколько программ, имеет собственный формат исполняемых программ, которые работают в режиме кооперативной многозадачности.

Первоначально (в момент появления и несколько лет после этого) система MS-DOS была вполне удовлетворительным средством для работы с ПЭВМ, но на настоящий момент она безнадёжно устарела. Тем не менее, она до настоящего времени всё ещё не полностью вытеснена более развитыми системами.

4.2 Windows NT.

ОС Windows NT появилась в начале 90-х годов. Это ОС для персональных рабочих станций и серверов приложений, построенных на процессорах Intel x86, DECAlpha, Motorola Power PC. Программы WindowsNT используют для работы API Win32. Система способна, кроме «родных» для неё программ (написанных под Win32 API), выполнять многие (но не все) программы, разработанные для MS-DOS, OS/2, POSIX, Presentation Manager 2.x, 16-битной Windows и Windows 95. ОС может работать в многопроцессорной системе, обеспечивая достаточно равномерную загрузку всех процессоров.

Каждому приложению, запущенному под Windows NT, предоставляется собственное адресное пространство объёмом 4 Гбайт. Из них 2 Гбайт отводятся самой программе, оставшиеся два — для служебных нужд (в них проецируются код и данные ОС). Адресные пространства различных процессов не пересекаются, они полностью изолированы друг от друга. Данные и код самой ОС также защищены от изменения их запущенными программами.

Система многопользовательская, она поддерживает регистрацию многих пользователей и управление их правами. Одновременная работа нескольких пользователей возможна через сеть. Пользователи могут получать доступ к ресурсам компьютера и запускать на нём программы. На любой объект могут быть установлены права доступа для любого пользователя и любой группы пользователей. Поддерживаются профили пользователей. Поддерживаются средства обеспечения секретности.

ОС имеет собственную файловую систему — NTFS. В отличие от файловой системы FAT, используемой в ОС DOS и Windows 95/98, NTFS позволяет работать с дисками большого объёма, обеспечивает возможность управления правами пользователей на объекты файловой системы, поддерживает возможность упаковки файлов на диске «на лету», с автоматической упаковкой данных при записи и распаковкой при чтении. Кроме того, она существенно надёжнее FAT.

Система продаётся в двух вариантах — Workstation (рабочая станция) и Server (сервер). Серверный вариант, помимо всех возможностей рабочей станции, обеспечивает работу компьютера в режиме контроллера сетевого домена.

Последняя выпущенная версия системы — 4.0. Последнее выпущенное для неё обновление (Service Pack) имеет номер 5. Дальнейшее развитие системы привело к созданию ОС Windows2000, о которой рассказано ниже.

Интерфейс пользователя Windows NT — графический. Однако ОС имеет и мощную систему команд, многие из которых в визуальной части интерфейса не продублированы.

Перечислим некоторые недостатки Windows NT. К ним можно отнести сравнительно высокие требования к оборудованию, постоянно растущие от версии к версии. Недостатком является также необходимость перезагрузки системы при изменении многих параметров настройки (например, при изменении адреса сетевого шлюза). Перезагрузка может быть нежелательна или даже вовсе неприемлема, если компьютер используется в качестве сервера приложений, с которым постоянно работают многие пользователи. При использовании компьютера с Windows NT в качестве файл-сервера проявляется такой недостаток, как отсутствие возможности задания для пользователей квот дискового пространства. В результате единственным способом ограничить пользователя в объёме дискового пространства, которое он может занять своими файлами, является создание для пользователя отдельного дискового раздела и разрешение пользователю записывать данные только в него.

Надёжность системы Windows NT была и остаётся предметом споров. Сторонники Windows NT представляют её как достаточно надёжную, противники же указывают на многочисленные недостатки, ухудшающие это качество системы. Можно говорить о том, что собственно система (версия 3.51 либо версия 4.0 с установленным Service Pack 3) работает стабильно и практически не может самопроизвольно перейти в неработоспособное состояние. Однако действия некорректно работающих программ могут нарушить работу системы и привести к необходимости перезагрузки компьютера.

4.3 Windows 95/98/Me.

ОС Windows 95 была выпущена как замена для Windows 3.1 и DOS. Windows 95 поддерживает вытесняющую многозадачность и реализует Win32 API (правда, не в полном объёме). По сравнению с Windows NT эта система менее требовательна к компьютеру, лучше поддерживает выполнение некоторых специфических программ MS-DOS и практически не имеет подсистемы администрирования. Windows 95 — однопользовательская система. Хотя в ней и есть возможность регистрации пользователей, разграничение прав этих пользователей практически полностью отсутствует. Штатными средствами возможно лишь управление правами пользователей на доступ в сеть. [1]

[1]Имеются дополнительные программные средства, которые включают в Windows 95 несколько более развитую систему администрирования.

Лучшая производительность на слабых машинах достигнута в Windows 95 за счёт упрощения ядра и отказа от некоторых средств обеспечения устойчивости системы. В Windows 95 каждый процесс, как и в Windows NT, имеет собственное адресное пространство объёмом 4 Гбайт, и младшие 2 Гбайт также изолированы от других процессов. Но старшие 2 Гбайт, в которые проецируется код и данные ОС, не защищены от изменения. В результате процесс, запущенный под этой ОС, может повредить данные и код операционной системы. Поэтому Windows 95 принципиально неустойчива.

Поскольку Windows 95 предназначена главным образом для домашних компьютеров, она была снабжена некоторыми дополнительными возможностями для работы с мультимедиа и игр. Для неё была создана библиотека DirectX, обеспечивающая быстрый вывод графики. Она используется, главным образом, в большом количестве динамических игр.

ОС Windows 98 является дальнейшим развитием Windows 95 и отличается от неё несколько переработанным ядром, некоторыми "косметическими"изменениями в интерфейсе пользователя, ориентированном на максимально простую работу пользователя в сети Internet, и поддержкой ряда дополнительных функций API. Кроме того, эта система несколько больше по объёму и, насколько можно судить, несколько быстрее работает на компьютерах с достаточно большим объёмом ОЗУ.

В 2000 году фирма Microsoft выпустила ОС Windows Millenium (Me), которая по структуре и возможностям мало отличается от Windows 98. Как и предыдущие ОС этого ряда, Windows Me ориентирована на использование в домашних и офисных компьютерах. Изменения коснулись главным образом интерфейса пользователя, кроме того, был ликвидирован явный рудимент MSDOS, сохранявшийся ранее — первоначальная загрузка в DOS-режме.

4.4 Windows 2000, Windows XP.

ОС Windows 2000 была официально выпущена 17 февраля 2000 года. Она разрабатывалась в течение нескольких предыдущих лет, в beta-версиях была известна как Windows NT 5. Эта система — очередной шаг в развитии семейства Windows NT. В ней сделана попытка объединить надёжное и довольно эффективное ядро Windows NT, возможности Multimedia, наиболее полно реализованные в Windows 95/98 и ряд новых для ОС фирмы Microsoft технологий. В отношении использования программного обеспечения Windows 2000 совместима с Windows 95/98 и Windows NT. Она позволяет выполнять большинство программ, написанных для более старых систем, за исключением тех, которые используют какие-либо способы оптимизации работы, жёстко связанные с архитектурой этих систем.

Из наиболее важных новых особенностей можно отметить следующие.

- Служба Active Directory. Она предназначена для обеспечения централизованного управления ресурсами крупной компьютерной сети, (например, сети предприятия или организации) и правами пользователей на работу с этими ресурсами.

Предоставляет следующие возможности.

- Общий механизм регистрации пользователя в сети позволяет пользователю работать с доступными ему сетевыми ресурсами с любого компьютера, подключенного к сети, предварительно зарегистрировавшись в ней.
- Система безопасности Kerberos обеспечивает возможность работы с данными, находящимися на другом компьютере, с использованием идентификации пользователя по паролю и шифрования передаваемых данных.
- Централизованное управление компьютерами в сети, правами пользователей и настройками ресурсов. Администратор имеет возможность предоставлять права доступа для любого ресурса или группы ресурсов в сети любому пользователю или группе пользователей.
- Возможность гибкого управления, как видимой структурой сетевых ресурсов, так и их физическим расположением.
- Объединение со службой DNS позволяет использовать одни и те же имена ресурсов для объектов в локальной сети и в Интернет.
- Возможность масштабирования, то есть объединения нескольких сетей более низкого уровня, также управляемых Active Directory, и централизованного управления сразу всей этой группой.
- DFS (Distributed File System) — распределённая файловая система. Она является одним из инструментов Active Directory и позволяет объединять данные, хранимые на различных компьютерах в сети, в единой виртуальной файловой системе. Для пользователя Active Directory такая файловая система выглядит как обычный сетевой ресурс, и факт нахождения различных её элементов на различных компьютерах никак не влияет на порядок работы с ней.
- В отличие от предыдущих систем, Windows 2000 наряду с механизмом логических устройств поддерживает возможность монтирования физических устройств хранения данных в общее дерево каталогов.
- В этой версии используется новая версия файловой системы NTFS 5, в которой реализован совершенно необходимый для файл-сервера механизм квотирования дискового пространства.
- Помимо упомянутых особенностей, внесён ряд более мелких изменений. Доработан интерфейс пользователя, реализована библиотека DirectX версии 7, обеспечивающая быстрый вывод графики.

Windows 2000 предназначена для профессиональных рабочих станций и серверов. Система поставляется в вариантах Professional, Server, Advanced Server и Data Center.

Первый вариант ориентирован на рабочие станции, остальные три — на сервера различной мощности. Ядро системы поддерживает компьютеры с двумя, четырьмя, восемью и шестидесяти четырьмя процессорами соответственно (все варианты количества процессоров, кроме первого, по сути, бесполезны, поскольку не существует хоть сколько-нибудь массовых реализаций компьютеров с архитектурой x86 с более чем двумя процессорами). Кроме этого, поставки различаются настройкой и количеством автоматически устанавливаемых служб и сервисных программ. Служба Active Directory может использоваться любым из вариантов системы, но версия Professional не содержит средств создания доменов Active Directory и управления ими, а лишь даёт возможность работы компьютера в таком домене.

Следующим шагом явился выпуск ОС Windows XP. Это очередная попытка объединить линии операционных систем 95/98 и NT. Система выпускается в нескольких модификациях: домашней (Home), профессиональной (Professional), и серверных. Серверные версии предназначены для поддержки серверов в сетях различного масштаба: от систем класса предприятия до небольших локальных сетей. Профессиональная версия по возможностям ближе всего к Windows 2000 Professional. Версия Home предназначена для домашних компьютеров (как замена Windows 98 или Windows Me). Во всех версиях используется новое оформление рабочей среды, в остальном же все варианты XP ближе всего к Windows 2000, как по возможностям, так и по ограничениям. В целях обеспечения совместимости с программами для предыдущих версий систем Windows предусмотрен запуск приложений в режиме совместимости (когда приложение запускается в среде, имитирующей одну из старых версий Windows), но это далеко не всегда помогает. Windows XP требует заметно большего количества оперативной памяти, чем её предшественники. Никаких уникальных возможностей и особенностей, которые бы оправдывали замену предыдущих версий на XP, не замечено.

4.5 UNIX и подобные системы.

Операционная система UNIX была разработана в фирме Bell Laboratories, входящей в корпорацию AT&T, в начале 70-х годов. Первоначально эта ОС разрабатывалась как переносимая система для мини-ЭВМ. Свыше 90% кода системы написаны на языке высокого уровня C, который, кстати, был разработан специально для реализации этой ОС.

Перечислить все возможности, предоставляемые UNIX-системами, в рамках относительно небольшого курса довольно затруднительно. Конкретные технические параметры систем могут варьироваться в очень широких пределах. Даже разрядность этих систем не является чем-то жёстко заданным, поскольку сейчас UNIX-подобные системы функционируют на всех распространённых видах компьютеров, от 16-разрядных персональных до 64- (и более) разрядных миникомпьютеров. Все UNIX-системы — это многопользовательские ОС, поддерживающие многозадачность и многопоточность (потоки реализованы как дочерние процессы). Работа с устройствами в UNIX бази-

руется на механизме «устройство как файл». Файловая система использует единое дерево каталогов, в которое монтируются устройства хранения файлов.

Система имеет развитый механизм администрирования, построенный с учётом того, что значительное число пользователей может работать с компьютером удалённо (через терминалы, сеть или телефонную линию). Существуют варианты UNIX для одно- и многопроцессорных компьютеров.

Базовый интерфейс пользователя ОС UNIX — текстовые команды. Командный язык исключительно богат. Все действия, связанных с управлением системой, могут быть выполнены с помощью одних только системных команд и программ, входящих в состав системы, без необходимости обращения к внешним программам и утилитам. Средства командного языка позволяют писать на нём программы большого объёма и значительной сложности.

UNIX имеет несколько командных интерпретаторов, аналогичных по возможностям, но несколько отличающихся по командному языку. Пользователи могут применять любой из них, или даже несколько одновременно, в зависимости от собственных предпочтений.

Графическая система XWindow System обеспечивает поддержку средств организации визуального интерфейса. В отличие от графических оболочек систем Windows, эта система не является неотъемлемой частью ОС. XWindow — это обычное приложение, выполняемое под управлением ОС. Недостатком такого подхода является несколько меньшая производительность графической подсистемы (если сравнивать с графической частью Windows NT, работающей в аналогичных условиях). Однако разница не особенно велика, а замедление вполне компенсируется гораздо более важным преимуществом. UNIX-система в целом существенно устойчивее, чем ОС интегрированной графической частью. Вызвано это тем, что нарушение работы графической подсистемы не влияет на работу ядра и приложений, не связанных с графикой. Кроме того, при использовании системы в качестве сервера графические возможности могут вовсе не использоваться. В таких случаях в UNIX можно просто не загружать XWindow, в то время как, например, в Windows NT графическая подсистема всё равно будет загружена, хотя в ней нет никакой необходимости. Учитывая, что графическая подсистема требует только для загрузки десятки мегабайт ОЗУ, преимущество может быть довольно заметным.

ОС UNIX, в отличие от таких ОС, как DOS, Windows 95/98, Windows NT, OS/2, MacOS, изначально разрабатывалась как максимально открытая система. Исходный код ядра UNIX-систем часто бывает доступен. Более того, некоторые варианты UNIX-систем поставляются, целиком или частично, в исходном коде (в виде текстов программ на языке C). Пользователь в случае необходимости может проанализировать порядок работы системы и даже внести в неё изменения. API UNIX совместимо со стандартом POSIX, что гарантирует корректное исполнение под любой UNIX-подобной ОС большинства программ, разработанных в соответствии с этим стандартом. Естественно, это может потребовать перекомпиляции программы в код процессора, на котором она будет исполняться, но другие изменения вносить не потребуется.

На настоящий момент существует, по меньшей мере, несколько десятков наиболее

распространённых клонов системы. Можно сказать, что UNIX является абсолютным лидером по числу типов компьютеров и процессоров, для которых она реализована. Такое многообразие вызвано изначальной открытостью системы и реализацией значительной её части на языке высокого уровня, которые позволяют легко переносить систему на различные платформы и столь же легко создавать совместимые с ней системы-клоны, позволяющие выполнять программы, написанные для оригинального UNIX.

Системы, совместимые с UNIX, можно разделить на две группы — клоны UNIX и UNIX-подобные ОС. Первые построены на основе оригинального кода UNIX, полученного по лицензии у правообладателя. Названия этих систем обычно содержат слово UNIX. Вторые создаются разработчиками без использования оригинального исходного кода. Они обычно не называются UNIX, а имеют названия, близкие по звучанию или написанию (XENIX, VENIX, LINUX). В целом все эти системы совместимы друг с другом и с оригинальными версиями UNIX, но у UNIX-клонов совместимость с оригинальной UNIX обычно более полная. Для UNIX-подобных ОС полная совместимость с оригиналом обычно не гарантируется, хотя к ней, безусловно, стремятся. Следует заметить, что сказанное вовсе не означает, что любой UNIX-клон обязательно лучше любой UNIX-подобной ОС. Речь идёт только о совместимости с оригинальной версией UNIX.

В настоящее время всё более широкое распространение на персональных компьютерах и сетевых серверах получают свободно распространяемые UNIX-подобные ОС, такие как Linux, FreeBSD, NetBSD, OpenBSD. Такие системы позволяют работать со всеми или с большинством программ, написанных для UNIX, обладают практически всеми свойствами и средствами, предоставляемыми коммерчески распространяемыми UNIX-системами, но выпускаются под свободными лицензиями (такими, как лицензия BSD или GNU GPL, например). В отличие от лицензий, под которыми распространяется коммерческое программное обеспечение, свободные лицензии позволяют любому желающему изучать, модифицировать, использовать и распространять систему и её части, причём как за плату, так и бесплатно. В результате свободно распространяемые UNIX-подобные системы существуют в огромном количестве совместимых друг с другом вариантов, на всех распространённых аппаратных платформах. Цены на них на несколько порядков ниже цен на коммерческие версии UNIX (как и на коммерческие версии других ОС, близких по возможностям), при очень близком качестве и возможностях.

Глава 5

Программирование с использованием Win32 API.

В этой главе будут описаны некоторые особенности программирования для ОС, поддерживающих Win32 API. Поскольку наиболее полно Win32 API реализуется в ОС Windows NT, именно эта ОС и будет описываться. За некоторыми исключениями, которые будут специально оговорены, всё изложенное верно также для ОС Windows 95. Объём курса делает невозможным сколько-нибудь подробное описание программирования с использованием Win32API, поэтому будут рассмотрены лишь некоторые моменты, иллюстрирующие особенности реализации ОС и практическую работу с механизмами ОС, описанными в предыдущих главах.

5.1 Организация адресного пространства процессов.

Как уже говорилось, Win32 предполагает, что каждый процесс имеет своё отдельное адресное пространство размером 4 Гб. Это адресное пространство делится на регионы, причём деление в системах Windows NT и Windows 95 отличается. Рассмотрим структуру адресного пространства этих систем.

5.1.1 Windows NT.

Адресное пространство делится на четыре региона.

Адреса 00000000 — 0000FFFF — регион размером 64 Кб, предназначенный для выявления пустых указателей. Обращение из программы по любому адресу в этом регионе приведёт к возникновению ошибки GPF (General Protection Fault) — общей ошибки защиты. Известно, что одной из наиболее частых и наиболее труднообнаруживаемых ошибок в программах является обращение в память по неправильному адресу. Такая ошибка может произойти, например, если не инициализирована (или неверно инициализирована) переменная-указатель. Попытка обращения в память по этому указателю может привести к непредсказуемым последствиям. Как правило,

пустой (неинициализированный) указатель содержит адрес в диапазоне от 0000 до FFFF или от 7FFF0000 до 7FFFFFFF. Поэтому система просто запрещает обращение по этим адресам, в результате чего на ошибки такого рода следует немедленная реакция.

Адреса 00010000 — 7FFEFFFF — регион размером 2Гб минус 64 Кб, собственная память процесса. В этот регион проецируется сама программа (код программы из exe-файла), все её данные, все блоки памяти, выделяемые программе для работы, все ресурсы. В этот же регион проецируются системные динамические библиотеки и объекты ядра. В Windows NT процесс имеет право обращаться к памяти только в этом регионе. Регион полностью защищён от попыток обращения из других процессов.

Адреса 7FFF0000 -7FFFFFFF — регион для выявления пустых указателей. Регион, аналогичный ранее рассмотренному региону 00000000 — 0000FFFF.

Адреса 80000000 — FFFFFFFF — регион операционной системы. На эту область памяти проецируются код операционной системы и драйверы устройств. Регион недоступен для программы.

5.1.2 Windows 95.

Адреса 00000000 — 00000FFF — регион размером 4 Кб для MS-DOS и 16-битной Windows. Недоступен, используется для выявления пустых указателей. Фактически этот регион содержит таблицу векторов прерываний, которые используют программы DOS при работе под Windows 95. Обращение к региону из программ Win32 приведёт к ошибке защиты.

Адреса 00001000 — 003FFFFF — регион размером 4 Мб для MS-DOS и 16-битной Windows. Доступен на чтение и запись, однако, категорически не рекомендуется что-либо записывать в него, так как это может привести к непредсказуемым последствиям для системы.

Адреса 00400000 — 7FFFFFFF — собственная память процесса. Регион, в котором располагается сама программа и её данные. Регион доступен на чтение и запись и полностью защищён от попыток обращения из других процессов.

Адреса 80000000 — BFFFFFFF — регион размером 1 Гб. В этом регионе находятся код и данные, доступные всем процессам в системе. В него загружаются системные библиотеки и объекты ядра системы. Регион доступен всем процессам на чтение и запись, причём все данные, находящиеся в нём, расположены для всех выполняемых процессов по одним и тем же адресам.

Адреса C0000000 — FFFFFFFF — регион ОС. В нём располагается код и данные ядра ОС и драйвера устройств. Из соображений эффективности данный регион не защищён, то есть любой процесс имеет свободный доступ к коду и данным, находящимся в нём. Попытка записать что-либо в этот регион может привести к краху ОС, поэтому не рекомендуется обращаться к нему.

Из описанного видно, что структура адресного пространства Windows NT и Windows 95 близка, но не идентична. Если в Windows NT процесс полностью изолирован от других процессов, а ОС изолирована выполняющихся в ней процессов, то в Windows

95 процесс может получить доступ (причём доступ на запись) к областям памяти, принадлежащим ОС. Причиной такой организации Windows 95 было желание разработчиков максимально упростить систему и, таким образом, обеспечить максимальную скорость её работы и минимальные затраты памяти. Однако это привело к меньшей устойчивости системы.

5.2 Объекты ядра.

Приложение Win32 может оперировать рядом объектов ОС — файлами, процессами, потоками, иконками, меню, шрифтами, курсорами и многими другими. Среди этих объектов можно выделить специфическую группу, называемую объектами ядра. Объекты ядра, в отличие от всех остальных объектов ОС, не создаются для каждого процесса отдельно. Если два процесса попытаются создать каждый по одному экземпляру одного и того же объекта ядра, то первый действительно создаст его, а второй будет подключён к уже созданному.

К этой группе относятся следующие объекты.

- Каналы (pipe objects).
- Объекты mutex.
- Семафоры (semaphore objects).
- "Почтовые ящики"(MailSlot objects).
- Процессы (process objects).
- Файлы, проецируемые в память (file-mapping objects).
- События (event objects).
- Файлы (file objects).
- Потоки (thread objects).

Для того чтобы создать объект ядра, приложение должно вызвать соответствующую функцию API. Имена функций, создающих объекты ядра, стандартны: имя состоит из слова Create, за которым следует полное либо сокращённое наименование объекта ядра (функции CreateProcess, CreateThread, CreateMutex и прочие). В числе параметров функции создания объекта ядра всегда имеется параметр, позволяющий задать атрибуты защиты. Этот параметр определяет, из каких процессов может быть доступен создаваемый объект ядра. Функция создания объекта ядра возвращает значение хэндла – описателя объекта. Хэндл — это просто 32-разрядное число, по которому система может идентифицировать объект, когда процесс обращается к нему.

Каждый объект ядра имеет счётчик числа пользователей. Если программа создаёт новый объект ядра, то ОС создаёт его, при этом счётчик устанавливается в единицу. Если же объект ядра уже существует, то ОС, вместо того, чтобы создавать новый объект, увеличивает счётчик существующего объекта на единицу. Когда программа перестаёт нуждаться в объекте ядра, она должна вызвать функцию API CloseHandle, передав ей в качестве параметра хэндл объекта. Вызов CloseHandle приводит к тому, что счётчик числа пользователей объекта уменьшается на единицу. Если счётчик становится равен нулю, то ОС автоматически удаляет объект.

5.3 Управление процессами.

Процесс в Win32 — это любая задача, выполняющаяся в системе. С точки зрения ОС процесс является объектом ядра. Для того чтобы запустить какую-либо программу, необходимо создать объект ядра "процесс"с помощью функции `CreateProcess`, входящей в API [1]. Заголовок этой функции выглядит следующим образом (описан с использованием синтаксиса и типов данных, применяемых в системе Delphi).

```
function CreateProcess(lpApplicationName: PChar;
    lpCommandLine: PChar; lpProcessAttributes,
    lpThreadAttributes: PSecurityAttributes;
    bInheritHandles: BOOL; dwCreationFlags: DWORD;
    lpEnvironment: Pointer; lpCurrentDirectory: PChar;
    const lpStartupInfo: TStartupInfo;
    var lpProcessInformation: TProcessInformation): BOOL;
```

Функция возвращает `true`, если процесс создан, и `false`, если произошла ошибка.

Рассмотрим параметры этой функции.

`lpApplicationName` — указатель на строку, завершающуюся нулём, содержащую полное имя исполняемого файла запускаемой программы. Имя должно содержать расширение (.exe по умолчанию не подставляется). Если имя не содержит пути, то файл ищется только в текущем каталоге, и, если его там нет, то функция возвращает значение `false`, что означает, что процесс создать не удалось. Вместо этого параметра допустимо указывать `nil` (пустой указатель), в этом случае вся информация о запускаемой задаче будет взята из второго параметра.

`lpCommandLine` – указатель на строку, завершающуюся нулём, содержащую полную командную строку запуска программы (то есть путь, имя файла

[1] Для запуска программы можно, помимо непосредственного создания процесса, воспользоваться специализированными функциями Win32 API. Но эти функции, в конечном итоге, всё равно создают новый процесс. Они отличаются только тем, что имеют меньше параметров и их проще использовать.

и параметры, если они нужны). Если у запускаемого файла не указано расширение, то считается, что это расширение — exe. Если указан полный путь, то файл ищется по этому пути, и, если его там нет, фиксируется ошибка. Если же путь не указан, то последовательно просматриваются следующие каталоги: каталог, в котором находится exe-файл вызывающего процесса; рабочий каталог вызывающего процесса; системный каталог Windows; основной каталог Windows; каталоги, перечисленные в переменной окружения `PATH`. Если в первом параметре (`lpApplicationName`) находится ненулевой указатель, то запущен будет файл, на который указывает `lpApplicationName`, но при этом ему будет передана командная строка, находящаяся в `lpCommandLine`.

`lpProcessAttributes`, `lpThreadAttributes` – указатели на структуры типа `TSecurityAttributes`, содержащие параметры защиты для процесса и его главного потока. Структура `TSecurityAttributes` содержит три поля — `nLength` типа `Integer`, `bInheritHandle` типа `Boolean` и `lpSecurityDescriptor` — указатель. `nLenght` должен содержать размер структуры в байтах, `bInheritHandle` — логическое значение, которое показывает, будет ли создаваемый объект доступен в процессах, порождённых из текущего, а `lpSecurityDescriptor` указывает на структуру, задающую права доступа и обычно устанавливается равным `nil`.

`bInheritHandles` – логический параметр, указывающий на то, должен ли создаваемый процесс получить доступ к наследуемым объектам, созданным в текущем процессе.

`dwCreationFlags` – содержит набор флагов, воздействующих на способы создания процесса. Несколько флагов можно передать в функцию, объединив их операцией `OR`. Помимо ряда сравнительно редко используемых флагов, можно указать флаг приоритета запускаемого процесса. Это должен быть один из флагов `IDLE_PRIORITY_CLASS`, `NORMAL_PRIORITY_CLASS`, `HIGH_PRIORITY_CLASS`, `REALTIME_PRIORITY_CLASS`. Процесс с приоритетом `IDLE` получает процессорное время только тогда, когда система не занята выполнением процессов с более высоким приоритетом. `NORMAL` — стандартный приоритет, `HIGH` – высокий, `REALTIME` — наивысший приоритет. При отсутствии флага приоритета система создаёт процесс с приоритетом по умолчанию (`NORMAL`).

`lpEnvironment` – указатель на блок памяти, содержащий набор переменных окружения для процесса. Обычно вместо него передаётся `nil`, и набор переменных окружения наследуется создаваемым процессом от текущего.

`lpCurrentDirectory` – рабочий каталог запускаемого процесса. Если вместо него

указан `nil`, то рабочим каталогом будет тот же, что и у запускающего процесса.

`lpStartupInfo` – указатель на структуру типа `TStartupInfo`, которая содержит набор флагов и параметров, используемых при создании процесса. Для запуска программы достаточно обнулить всю структуру и указать три параметра — `dwFlags = STARTF_USESHOWWINDOW` и `wShowWindow = SW_SHOWDEFAULT`, а в поле `cb` установить размер структуры. Это приведёт к тому, что программа будет запущена с набором параметров по умолчанию.

`lpProcessInformation` – указатель на структуру типа `TProcessInformation`, которую необходимо предварительно создать. Эта структура содержит четыре поля, которые после вызова `CreateProcess` будут заполнены. Поле `hProcess` содержит хэндл созданного процесса, поле `hThread` – хэндл его главного потока, `dwProcessId` и `dwThreadId` – идентификаторы, присвоенные системой процессу и потоку, с помощью которых запустивший процесс может работать с запущенными.

Типичный порядок использования функции `CreateProcess` продемонстрирован в примере 5.1.

Здесь создаются и инициализируются все необходимые структуры, после чего вызывается процедура `CreateProcess`, которая запускает редактор notepad с параметром — именем текстового файла для редактирования.

По завершении работы с процессом вызывающая программа должна закрыть его хэндл вызовом функции `CloseHandle(ProcHandle)`. При этом, если процесс ещё выполняется, он не будет завершён. Процесс продолжит выполнение, просто к нему уже не будет доступа из родительского процесса.

Процесс завершается, когда из него вызывается функция API `ExitProcess`. Данная функция имеет один параметр — код завершения процесса, который передаётся в окружение, в котором был запущен процесс. Код может быть любым, он используется для того, чтобы сигнализировать о том, как завершился процесс (корректно, из-за ошибки и т.п.). При завершении процесса автоматически завершаются все потоки, закрываются объекты ядра и уничтожаются все остальные объекты ОС, возданные процессом, освобождается память, занятая процессом. При написании программы обычно нет необходимости в вызове `ExitProcess`, поскольку этот вызов добавляется компилятором автоматически после завершения выполнения всего кода программы.

Процесс может также быть остановлен с помощью вызова предназначенной для этого функции API `TerminateProcess`. Этой функцией можно остановить процесс не из него самого, а из любого процесса в системе, который имеет хэндл останавливаемого процесса. Например, из процесса, запустившего данный. Функция имеет два параметра. Первый параметр — хэндл останавливаемого процесса, второй — код возврата. Процесс останавливается также, как и после вызова `ExitProcess`. Необходимо отметить, что завершение процесса функцией `TerminateProcess` не явля-

```
StInfo : TStartupInfo; // параметры запуска
SeAttr : TSecurityAttributes; // атрибуты защиты
ProcInfo : TProcessInformation; // параметры процесса
ProcHandle : THandle; // Хэндл процесса.

begin
  // Инициализация служебных записей
  // для процедуры CreateProcess

  // Инициализация структуры StInfo.
  FillChar(StInfo,SizeOf(StInfo),0); // Очистка структуры.
  StInfo.cb := SizeOf(StInfo);
  stInfo.dwFlags:=STARTF_USESHOWWINDOW;
  StInfo.wShowWindow:=SW_SHOWDEFAULT;
  // Инициализация структуры SeAttr.
  with SeAttr do begin
    nLength:=SizeOf(SeAttr);
    bInheritHandle:=true;
    lpSecurityDescriptor:=nil;
  end;
  // Запуск процесса
  if CreateProcess(nil,
      PChar('c:\windows\notepad.exe c:\texts\a.txt'),
      @SeAttr,@SeAttr,false,NORMAL_PRIORITY_CLASS,
      nil,
      PChar('c:\windows\'),
      StInfo, ProcInfo) then begin
    ProcHandle := ProcInfo.hProcess; // Получить хэндл процесса
    end
  else begin
    raise Exception.Create('Невозможно запустить процесс');
  end;
end;
```

Пример 5.1: Использование CreateProcess

ется безопасным. Хотя ОС и удаляет объекты, созданные процессом и освобождает память, выделенную ему, но внезапная остановка процесса почти всегда приводит к нарушению логики его функционирования, что нежелательно.

5.4 Многопоточные программы. Управление потоками средствами Win32 API.

Поток представляет собой последовательность исполнения кода внутри процесса. При создании процесса ОС автоматически создаёт первичный поток, в котором и выполняется код программы. Большинство программ обходятся этим потоком и не нуждаются в создании дополнительных. Когда оказывается удобным организовать программу так,

чтобы в ней одновременно выполнялось несколько потоков, Win32 предоставляет такую возможность. Программа может создать, помимо первичного, любое количество потоков. Процесс считается завершённым, если завершились все потоки, созданные в нём. С этой точки зрения первичный поток не имеет никаких привилегий по сравнению с порождёнными потоками. Он может быть завершён до того, как завершатся остальные потоки, при этом процесс будет работать.

Поток представляет собой объект ядра Win32. Для создания нового потока нужно вызвать функцию API `CreateThread`.

```
function CreateThread(lpThreadAttributes: Pointer;
    dwStackSize: DWORD; lpStartAddress: TFNThreadStartRoutine;
    lpParameter: Pointer; dwCreationFlags: DWORD;
    var lpThreadId: DWORD): THandle;
```

Функция возвращает хэндл созданного потока. Её параметры имеют следующий смысл.

`lpThreadAttributes` — указатель на структуру типа `TSecurityAttributes`, обычно `nil`.

`dwStackSize` — начальный размер стека потока (обычно нуль).

`lpStartAddress` — адрес функции, которая будет выполняться в потоке.

`lpParameter` — указатель на структуру в памяти, который будет передан функции потока при запуске.

`dwCreationFlags` — флаги запуска. Можно указать флаг `CREATE_SUSPENDED`, при этом поток будет создан, но не запущен.

`lpThreadId` — переменная, в которую будет записан идентификатор потока.

Для завершения запущенного потока из функции потока можно вызвать функцию API `ExitThread` с одним параметром — кодом возврата. Если требуется завершить поток из другого потока, можно вызвать функцию `TerminateThread` с двумя параметрами – хэндлом останавливаемого потока и кодом возврата. Поток принудительно завершается. Использовать `TerminateThread` рекомендуется только в крайних случаях по той же причине, что и `TerminateProcess` – принудительное завершение потока может привести к нарушению логики его функционирования.

Для того, чтобы приостановить выполняющийся поток, можно вызвать функцию API `SuspendThread` с одним параметром — хэндлом потока. Поток будет приостановлен — ему перестанут выделяться кванты процессорного времени.

Для запуска приостановленного потока используется функция `ResumeThread` с одним параметром — хэндлом потока. Её вызов приводит к продолжению выполнения потока, если он ранее был приостановлен функцией `SuspendThread` или если он был создан с флагом `CreateSuspended`.

5.5 Работа с файлами. Файлы, отображаемые в память.

5.5.1 Работа с файлами стандартными средствами.

Win32 API содержит традиционный набор функций для работы с файлами, который обычно имеют все сколько-нибудь развитые операционные системы. Методика работы с файлами этими средствами также стандартна.

Для того чтобы использовать данные, находящиеся в файле на внешнем носителе, либо сохранить результаты работы программы в файле, программа должна, прежде всего, выполнить операцию открытия файла. Открытие файла выполняется с помощью вызова функции API `CreateFile`. Необходимо понимать, что файл для Win32 — это объект ядра. Поэтому смысл функции `CreateFile` состоит именно в создании объекта ядра "файл". Заголовок функции `CreateFile` выглядит следующим образом.

```
function CreateFile(lpFileName: PChar;
  dwDesiredAccess,
  dwShareMode: DWORD;
  lpSecurityAttributes: PSecurityAttributes;
  dwCreationDisposition,
  dwFlagsAndAttributes: DWORD;
  hTemplateFile: THandle): THandle; stdcall;
```

`lpFileName` – имя файла. Если имя не содержит полного пути, то файл будет открыт только в том случае, если он находится в текущем каталоге.

`dwDesiredAccess` – способ доступа к файлу. Может принимать значения `GENERIC_READ` (доступ на чтение), `GENERIC_WRITE` (доступ на запись) и 0 (доступ, без права чтения или записи, используется только для проверки атрибутов файла), а также быть комбинацией этих значений, объединённых операцией `OR`.

`dwShareMode` – режим совместного доступа к файлу. Может принимать значения `FILE_SHARE_READ` (разрешено открытие файла другим приложением на чтение), `FILE_SHARE_WRITE` (разрешено открытие файла на запись) или `FILE_SHARE_DELETE` (разрешено удаление, доступно только в Windows NT).

`lpSecurityAttributes` – указатель на структуру атрибутов защиты, обычно `nil`, в Windows 95 этот параметр не используется.

`dwCreationDisposition` – параметр, определяющий требуемую операцию (открытие существующего файла или его создание) и действия, которые должны быть произведены, если файл существует и если файл не существует. Возможны значения: `CREATE_NEW` (создаётся новый файл, а если

файл уже существует, то выполнение функции заканчивается ошибкой), CREATE_ALWAYS (создаётся новый файл, либо существующий файл открывается и обрезается до нулевой длины), OPEN_EXISTING (открывается существующий файл, если файла нет, то происходит ошибка), OPEN_ALWAYS (существующий файл открывается, а если его нет, то он создаётся пустым). TRUNCATE_EXISTING (существующий файл открывается и обрезается до нулевой длины, если файл не существует, то происходит ошибка).

dwFlagsAndAttributes – набор флагов, устанавливающих атрибуты файла и влияющих на работу с ним. Используются атрибуты: FILE_ATTRIBUTE_ARCHIVE (устанавливается атрибут "архивный файл"), FILE_ATTRIBUTE_COMPRESSED (устанавливается атрибут "сжатый файл", под Windows NT он влияет на способ хранения данных), FILE_ATTRIBUTE_HIDDEN (атрибут "скрытый файл"), FILE_ATTRIBUTE_NORMAL (сбрасываются все прочие атрибуты файла, этот параметр не может использоваться в комбинации с другими), FILE_ATTRIBUTE_OFFLINE (атрибут указывает на то, что в данный момент данные, находящиеся в файле, физически недоступны), FILE_ATTRIBUTE_READONLY (атрибут "файл только для чтения"), FILE_ATTRIBUTE_SYSTEM (атрибут "системный файл"), FILE_ATTRIBUTE_TEMPORARY (атрибут "временный файл", указывает системе на то, что данные записываются в файл для временного хранения, поэтому необходимо по возможности не сбрасывать их на диск, а держать в оперативной памяти). Кроме того, могут использоваться флаги: FILE_FLAG_WRITE_THROW (указывает на то, что данные, записываемые в файл, должны сбрасываться ОС на диск немедленно, а не оставаться в кэше для отложенной записи), FILE_FLAG_OVERLAPPED (используется для открытия файла в режиме асинхронного доступа, в Windows95 не поддерживается), FILE_FLAG_NO_BUFFERING (требует от ОС не выполнять с данным файлом операции чтения/записи без промежуточной буферизации; в специальных случаях может повысить эффективность, но накладывает дополнительные ограничения на порядок работы с файлом), FILE_FLAG_DELETE_ON_CLOSE (указывает, что ОС должна удалить файл немедленно после его закрытия всеми приложениями, которые его открыли), FILE_FLAG_RANDOM_ACCESS (сообщает ОС, что программа будет работать с файлом в режиме произвольного доступа), FILE_FLAG_SEQUENTIAL_ACCESS (сообщает ОС, что приложение будет читать/записывать файл в последовательном режиме, строго от начала к концу). Последние два флага используются для оптимизации кэширования файла. ОС выбирает режим кэширования в зависимости от предполагаемого режима доступа. Указание одного этих из флагов не накладывает никаких ограничений на режим доступа программы к файлу, но если режим доступа не будет соответствовать установленному фла-

гу, то скорость работы с файлом будет несколько меньше. Кроме того, имеются ещё два редко используемых флага, которые мы не будем рассматривать.

`hTemplateFile` – хэндл файла, атрибуты и флаги которого будут скопированы и использованы при открытии нового файла. Файл, хэндл которого указывается, должен быть открыт на чтение или на чтение и запись. Параметр может использоваться, когда нужно открыть несколько файлов с совпадающими наборами атрибутов. Если параметр не используется, нужно передать вместо него нуль.

Функция `CreateFile` возвращает хэндл открытого файла. Если файл открыт корректно, то функция `GetLastError`, вызванная непосредственно после `CreateFile`, вернёт значение `ERROR_SUCCESS`, в противном случае — вернёт код ошибки.

После завершения работы с файлом приложение должно закрыть файл, вызвав функцию API `CloseHandle` с одним параметром — хэндлом открытого ранее файла. Если приложение не закроет файл, то файл будет автоматически закрыт по завершении этого приложения.

Работа с открытым файлом базируется на стандартной идеологии последовательного доступа. Данные могут читаться из файла и записываться в него только последовательно. Для работы с файлом используется понятие файлового указателя. Файловый указатель понимается как указатель на текущую позицию в файле, то есть на позицию, с которой будет производиться чтение или запись при следующей операции с файлом. После открытия файла файловый указатель всегда указывает на начало файла. При выполнении операции чтения или записи данных файловый указатель перемещается в направлении конца файла на размер считанного или записанного буфера.

К файлу применимы три операции — чтения, записи и изменения положения файлового указателя.

Для работы с файлом традиционными методами используются функции `ReadFile`, `WriteFile` и `SetFilePointer`.

Чтение из файла производится функцией `ReadFile`.

```
function ReadFile(
  hFile: THandle;
  var Buffer;
  nNumberOfBytesToRead: DWORD;
  var lpNumberOfBytesRead: DWORD;
  lpOverlapped: POverlapped): BOOL; stdcall;
```

`hFile` – хэндл открытого файла.

`Buffer` – буфер, в который будет производиться считывание данных из файла. В реализации Delphi может быть любого типа.

`nNumberOfBytesToRead` – количество байт, которое требуется прочитать.

`lpNumberOfBytesRead` – фактически прочитанное количество байт.

`lpOverlapped` – указатель на структуру `TOverlapped`, применяется для асинхронного чтения, обычно `nil`.

Функция возвращает логическое значение. Значение true указывает на корректное выполнение операции. Значение false сигнализирует об ошибке. Расширенную информацию об ошибке можно получить, вызвав функцию `GetLastError`.

Если функция вернула значение `true`, то о результате её работы можно судить по значению, записанному в `lpNumberOfBytesRead`. Это значение всегда равно фактически прочитанному количеству байт. Если оно равно запрошенному числу байт, значит, прочитан требуемый объём данных. Если значение меньше запрошенного, значит, при чтении был достигнут конец файла. В этом случае данными заполнена только часть буфера. Если значение равно нулю, значит, на момент операции чтения файловый указатель уже находился в конце файла.

Для выполнения чтения из файла файл должен быть открыт с атрибутом `FILE_READ`.

Запись данных в файл производится функцией `WriteFile`.

```
function WriteFile(hFile: THandle;
    const Buffer; nNumberOfBytesToWrite: DWORD;
    var lpNumberOfBytesWritten: DWORD;
    lpOverlapped: POverlapped): BOOL; stdcall;
```

`hFile` – хэндл открытого файла.

`Buffer` – буфер, из которого будет производиться запись данных в файл.

`nNumberOfBytesToWrite` – количество байт, которое требуется записать.

`lpNumberOfBytesWritten` – фактически записанное количество байт.

`lpOverlapped` – указатель на структуру `TOverlapped`, применяется для асинхронного чтения, обычно `nil`.

Функция возвращает логическое значение. Значение true указывает на корректное выполнение операции. Значение `false` сигнализирует об ошибке. Расширенную информацию об ошибке можно получить, вызвав функцию `GetLastError`.

Для выполнения записи в файл программа должна открыть его с атрибутом `FILE_WRITE`.

Файл может быть открыт одновременно на чтение и запись. В этом случае записываемые данные помещаются в файл "поверх"данных, ранее находившихся в этом месте файла. При записи данных после конца файла происходит увеличение размера файла.

Переместить файловый указатель в любое место файла можно с помощью функции SetFilePointer.

```
function SetFilePointer(hFile: THandle;
    lDistanceToMove: Longint;
    lpDistanceToMoveHigh: Pointer;
    dwMoveMethod: DWORD): DWORD; stdcall;
```

`hFile` – хэндл открытого файла.

`lDistanceToMove` – младшие 4 байта расстояния перемещения.

`lpDistanceToMoveHigh` – указатель на переменную, содержащую старшие четыре байта расстояния перемещения. Может быть `nil`, если расстояние перемещения целиком помещается в четырёхбайтовом числе.

`dwMoveMethod` – константа, указывающая точку отсчёта.

Обратим внимание на второй и третий параметр. Если третий параметр – `nil`, то второй представляет собой целое число со знаком. В этом случае файловый указатель будет перемещён на указанное в параметре число байт. При положительном значении параметра файловый указатель перемещается вперёд (в направлении конца файла), при отрицательном – назад (в направлении начала файла). Если третий параметр — не `nil`, то второй параметр и четырёхбайтовое значение, на которое указывает `lpDistanceToMoveHigh` составляют в совокупности восьмибайтовое целое число со знаком, определяющее расстояние, на которое переместится файловый указатель.

Такой способ указания расстояния перемещения указателя выбран для того, чтобы можно было перемещать указатель на расстояние, превышающее 2 Гбайт, что может потребоваться при работе с очень большими файлами. Для большинства применений вполне достаточно использовать только параметр `lDistanceToMove`.

Функция возвращает младшие четыре байта нового положения файлового указателя. Старшие четыре байта записываются в переменную по адресу `lpDistanceToMoveHigh`, если при вызове функции указан этот параметр.

Параметр `dwMoveMethod` используется для указания положения, от которого будет отсчитываться новое положение файлового указателя. Он может принимать одно из трёх значений — `FILE_BEGIN`, `FILE_END` и `FILE_CURRENT`. При первом значении параметра новая позиция файлового указателя отсчитывается от начала файла, при втором — от его конца, при третьем — от текущего положения.

Вызов функции с нулевой дистанцией перемещения указателя разрешён. Он может использоваться для получения текущего положения файлового указателя (если указан режим отсчёта `FILE_CURRENT`) или для определения размера файла (если указан режим отсчёта `FILE_END`).

Если в результате выполнения функции файловый указатель должен был бы стать отрицательным, то происходит ошибка. При этом текущее положение файлового указателя не изменяется. Перемещение указателя за конец файла не является ошибкой.

При такой операции указатель перемещается в указанную позицию. При попытке прочитать данные из файла, указатель которого был перемещён за конец файла, функция чтения вернёт нулевое количество прочитанных байтов. При попытке записи данных файл будет увеличен так, чтобы файловый указатель указывал на его конец. При этом содержимое добавленного участка файла не определено. После этого данные будут записаны. Тот же эффект даст вызов функции `SetEndOfFile`, только при этом не будут записываться в файл никакие данные.

5.5.2 Использование файлов, отображаемых в память.

Win32 API предоставляет программисту возможность использовать файлы, отображаемые в память. Эта возможность реализуется через объекты ядра `FileMapping`. Использование отображаемых файлов является наиболее быстрым способом работы с данными на внешних носителях.

Для создания объекта "проецируемый файл"необходимо вызвать функцию API `CreateFileMapping`.

```
function CreateFileMapping(hFile: THandle;
    lpFileMappingAttributes: PSecurityAttributes;
    flProtect,
    dwMaximumSizeHigh,
    dwMaximumSizeLow: DWORD;
    lpName: PChar): THandle; stdcall;
```

`hFile` – хэндл ранее открытого файла, который проецируется в память. Если в качестве хэндла передаётся шестнадцатеричное значение `FFFFFFFF`, то объект связывается не с физическим файлом на диске, а с группой страниц страничного файла ОС, то есть происходит просто резервирование памяти, не связанной с каким-либо конкретным файлом.

`lpFileMappingAttributes` – указатель на структуру параметров защиты, обычно `nil`, в Windows 95 игнорируется.

`flProtect` – набор объединённых операцией `OR` флагов. Поддерживаются флаги: `PAGE_READONLY` (данные в спроецированном файле будут доступны только для чтения), `PAGE_READWRITE` (данные будут доступны на чтение и запись), `PAGE_WRITECOPY` (данные будут доступны на чтение и запись, но при записи изменённые данные будут попадать на специально созданные дополнительные копии изменённой страницы в памяти и не будут видны другим процессам, работающим с тем же проецируемым файлом), `SEC_IMAGE` (указывает ОС, что открываемый файл является исполнимым файлом Win32, при его проецировании необходимо следовать некоторым специальным правилам; Windows 95 этот флаг игнорирует), `SEC_NOCACHE` (запрещает кэшировать страницы, относящиеся к

файлу, используется редко, Windows 95 его игнорирует), SEC_COMMIT и SEC_RESERVE – флаги, используемые только для объектов "проецируемый файл", не связанных с физическим файлом на диске.

dwMaximumSizeHigh и dwMaximumSizeLow – старшие 4 байта и младшие 4 байта размера файла. Если проецируется файл на диске и известно, что его не понадобится дополнять, то можно указать в обоих параметрах нули — размер будет установлен равным размеру файла. Для обеспечения возможности дописывания файла нужно устанавливать размер больше реального (с запасом). Можно указать размер до 18 экзабайт (18 квинтиллионов байт). Фактически, обычно нет необходимости работать с файлами более 4Гб, поэтому в старшие разряды передаётся нуль, а в младшие — необходимый размер.

lpName – указатель на строку, завершающуюся нулём, задающую имя объекта "проецируемый файл". Если имя не нужно, можно передать nil.

Функция создаёт объект и возвращает его хэндл.

Чтобы получить возможность работать со спроецированным файлом, необходимо выполнить ещё одну операцию. Нужно, чтобы ОС передала содержимое файла в выделенные для этого страницы и сообщила приложению адрес, по которому в его адресном пространстве находится выделенный регион. Эта операция выполняется с помощью вызова функции MapViewOfFile.

```
function MapViewOfFile(hFileMappingObject: THandle;
    dwDesiredAccess: DWORD;     dwFileOffsetHigh, dwFileOffsetLow,
    dwNumberOfBytesToMap: DWORD): Pointer; stdcall;
```

hFileMappingObject – хэндл объекта "проецируемый файл", созданного ранее.

dwDesiredAccess – указывает способ доступа к данным. Возможные значения: FILE_MAP_WRITE (разрешены чтение и запись), FILE_MAP_READ (разрешено только считывание), FILE_MAP_ALL_ACCESS (аналогично параметру FILE_MAP_WRITE), FILE_MAP_COPY (разрешены чтение и запись, запись приводит к копированию изменённой страницы).

dwFileOffsetHigh, dwFileOffsetLow – старшие и младшие четыре байта смещения в файле, с которого начнётся проецирование (указание в этих параметрах 0 и 65536 соответственно приведёт к тому, что файл будет спроецирован начиная с 65-го килобайта). Смещение должно быть чётным числом, кратным гранулярности выделения ресурсов в системе, которая во всех существующих реализациях Win32 равна 64Кбайт.

dwNumberOfBytesToMap – размер спроецированного фрагмента в байтах.

Последние три параметра позволяют спроецировать на адресное пространство процесса не весь файл сразу, а его отдельные фрагменты. Указав те или иные параметры, приложение создаёт своего рода "окно", имеющее заданный размер, не превышающий 4Гбайт и начинающееся с произвольного места в файле. Обычно это необходимо, когда файл очень велик и его полное проецирование невозможно либо приводит к слишком большому расходу ресурсов. Если же у системы нет сложностей с выделением памяти при проецировании файла (файл не очень велик), то при прочих равных условиях быстрее будет работать программа, проецирующая сразу весь файл.

Для того чтобы спроецировать весь файл сразу, необходимо указать оба параметра смещения нулевыми, а размер — равным размеру файла.

Функция `MapViewOfFile` возвращает указатель, ссылающийся на область адресного пространства процесса, в которую спроецирован указанный фрагмент файла. Обращаясь по этому адресу, процесс получает доступ к данным, хранящимся в файле. Данные можно изменять, если в вызовах функций открытия файла и проецирования его указаны соответствующие флаги. Чтобы дописать данные в конец файла, необходимо при создании объекта "проецируемый файл"указать больший размер, чем фактический размер файла.

При указании флагов, определяющих режим доступа к данным, необходимо учитывать, что попытка установка флага, требующего больших прав, чем те, которые уже были предоставлены, приведёт к ошибке. Так, если файл открыт функцией `CreateFile` в режиме "только для чтения", попытка указать в вызовах функций `CreateFileMapping` или `MapViewOfFile` флага, разрешающего запись, приведёт к тому, что эти функции завершатся с сообщением об ошибке.

При неоднократном вызове функции `MapViewOfFile` каждый раз резервируется новый регион адресного пространства, при этом ранее выделенные регионы не освобождаются. Для освобождения выделенного региона нужно вызвать функцию `UnmapViewOfFile(lpBaseAddress: Pointer)`, передав ей в качестве параметра ранее полученный адрес региона. Если эта функция не будет вызвана, то все выделенные и не освобождённые регионы автоматически освободятся при завершении процесса, их создавшего.

После того, как завершена работа с объектом "проецируемый файл", объект нужно освободить, вызвав функцию `CloseHandle` с параметром — хэндлом объекта.

Следует иметь в виду, что функция `MapViewOfFile`, будучи вызвана, увеличивает счётчики числа пользователей объектов "файл"и "проецируемый файл", с которыми она работает, на единицу, а функция `UnmapViewOfFile`, соответственно, уменьшает эти счётчики на единицу. Поэтому, если создаётся только один регион, на который проецируется файл, а хэндлы файла и объекта "проецируемый файл"не нужны в программе, то можно вызвать `CloseHandle` для файла сразу после создания объекта "проецируемый файл", а `CloseHandle` для проецируемого файла — сразу после вызова `MapViewOfFile`. При этом файл останется открытым, а объект "проецируемый файл"не будет уничтожен до тех пор, пока не будет вызвана процедура `UnmapViewOfFile` для выделенного региона.