Оглавление

Лабораторная работа №1

Командный интерфейс ОС Windows.

Теоретические сведения

1. Оболочка командной строки Windows. Интерпретатор Cmd.exe

В операционной системе Windows, как и в других операционных системах, интерактивные (набираемые с клавиатуры и сразу же выполняемые) команды выполняются с помощью так называемого командного интерпретатора, иначе называемого командным процессором или оболочкой командной строки (command shell). Командный интерпретатор или оболочка командной строки — это программа, которая, находясь в оперативной памяти, считывает набираемые вами команды и обрабатывает их.
Для запуска командного интерпретатора (открытия нового сеанса командной строки) можно выбрать пункт Выполнить… (Run) в меню Пуск (Start), ввести имя файла Cmd.exe и нажать кнопку OK

1.1 Синтаксис командной строки, перенаправление ввода – вывода

Файловая система имеет древовидную структуру и имена файлов задаются в формате *[диск:] [путь\]имя_файла*. Если путь начинается с символа «\», то маршрут вычисляется от корневого каталога – иначе от текущего.
Например, c:123.txt задает файл 123.txt в текущем каталоге, c:\123.txt – в корневом, а DOC\123.txt – в подкаталоге DOC текущего каталога.
Существуют особые обозначения для текущего каталога (точка «.») и трех его верхних уровней (две точки «..» - родительский, три «…» - второго уровня и, наконец, четыре «….» - третьего уровня).
Например, для текущего каталога C:\Windows\Media\Office97 путь к файлу autoexec.bat в корневом каталоге диска C: может быть записан в виде ….\autoexec.bat.
В именах файлов (но не дисков или каталогов) можно применять так называемые групповые символы или шаблоны: ? (вопросительный знак) и * (звездочка). Символ * в имени файла означает произвольное количество любых допустимых символов, символ ? — один произвольный символ или его отсутствие. Скажем, под шаблон `text??1.txt` подходят, например, имена text121.txt и text11.txt, под шаблон `text*.txt` — имена text.txt, textab12.txt, а под шаблон `text.*` — все файлы с именем text и произвольным расширением.
Например, `DIR /? > helpdir.txt` выведет справку по команде DIR в файл. Символ «>>» позволяет не создавать файл заново, а дописать в него. По аналогии символ «<» позволяет читать данные не с клавиатуры, а с файла. Например, `DATE < date.txt` ввод новой даты из файла.

1.2 Переменные окружения

При загрузке ОС Windows в оперативной памяти постоянно хранится набор т.н. *переменных окружения* (environment variables). Хотя в Windows есть более совершенный способ для хранения системных значений – реестр, многие программы по-прежнему используют переменные окружения. Наиболее важные переменные хранят системный путь для поиска (PATH), каталог запуска Windows (WINDIR), место хранения временных файлов (TEMP) и многое другое.
Переменные устанавливаются с помощью команды
`SET [переменная=[строка]]`

Запуск SET без параметров приводит к выводу списка переменных среды. Для получения их значений (всегда строки) нужно имя соответствующей переменной заключить в символы «%», **например**: `%TEMP%`.

1.3 Внутренние и внешние команды. Структура команд

Некоторые команды распознаются и выполняются непосредственно самим командным интерпретатором — такие команды называются внутренними (например, `COPY` или `DIR`) Другие команды операционной системы представляют собой отдельные программы, расположенные по умолчанию в том же каталоге, что и Cmd.exe, которые Windows загружает и выполняет аналогично другим программам. Такие команды называются внешними (например, `MORE` или `XCOPY`).

Рассмотрим структуру самой командной строки и принцип работы с ней. Для того, чтобы выполнить команду, вы после приглашения командной строки (например, `C:\>`) вводите имя этой команды (регистр не важен), ее параметры и ключи (если они необходимы) и нажимаете клавишу <Enter>. Например:

```
C:\>COPY C:\myfile.txt A:\ /V
```

Имя команды здесь — `COPY`, параметры — `C:\myfile.txt` и `A:\`, а ключом является `/V`. Отметим, что в некоторых командах ключи могут начинаться не с символа /, а с символа – (минус), например, `-V`.

Многие команды Windows имеют большое количество дополнительных параметров и ключей, запомнить которые зачастую бывает трудно. Большинство команд снабжено встроенной справкой, в которой кратко описываются назначение и синтаксис данной команды. Получить доступ к такой справке можно путем ввода команды с ключом `/?`.

1.4 Условное выполнение и группировка команд

В командной строке Windows NT/2000/XP можно использовать специальные символы, которые позволяют вводить несколько команд одновременно и управлять работой команд в зависимости от результатов их выполнения. С помощью таких символов условной обработки можно содержание небольшого пакетного файла записать в одной строке и выполнить полученную составную команду.

Используя символ амперсанта `&`, можно разделить несколько утилит в одной командной строке, при этом они будут выполняться друг за другом. Например, если набрать команду `DIR & PAUSE & COPY /?` и нажать клавишу <Enter>, то вначале на экран будет выведено содержимое текущего каталога, а после нажатия любой клавиши — встроенная справка команды `COPY`.

Символ `^` позволяет использовать командные символы как текст, то есть при этом происходит игнорирование значения специальных символов. Например, если ввести в командной строке

```
ECHO Абв & COPY /?
```

и нажать клавишу <Enter>, то произойдет выполнение подряд двух команд: `ECHO Абв` и `COPY /?` (команда `ECHO` выводит на экран символы, указанные в командной строке после нее). Если же выполнить команду

```
ECHO Абв ^& COPY /?
```

то на экран будет выведено

```
Абв & COPY /?
```

В этом случае просто выполняется одна команда `ECHO` с соответствующими параметрами.

Условная обработка команд в Windows осуществляется с помощью символов `&&` и `||` следующим образом. Двойной амперсант `&&` запускает команду, стоящую за ним в командной строке, только в том случае, если команда, стоящая перед амперсантами была выполнена успешно. Например, если в корневом каталоге диска C: есть файл plan.txt, то

выполнение строки TYPE C:\plan.txt && DIR приведет к выводу на экран этого файла и содержимого текущего каталога. Если же файл C:\plan.txt не существует, то команда DIR выполняться не будет.

Два символа || осуществляют в командной строке обратное действие, т.е. запускают команду, стоящую за этими символами, только в том случае, если команда, идущая перед ними, не была успешно выполнена. Таким образом, если в предыдущем примере файл C:\plan.txt будет отсутствовать, то в результате выполнения строки TYPE C:\plan.txt || DIR на экран выведется содержимое текущего каталога.

Отметим, что условная обработка действует только на ближайшую команду, то есть в строке

TYPE C:\plan.txt && DIR & COPY /?

команда COPY /? запустится в любом случае, независимо от результата выполнения команды TYPE C:\plan.txt.

Несколько утилит можно сгруппировать в командной строке с помощью скобок. Рассмотрим, например, две строки:

TYPE C:\plan.txt && DIR & COPY /?

TYPE C:\plan.txt && (DIR & COPY /?)

В первой из них символ условной обработки && действует только на команду DIR, во второй — одновременно на две команды: DIR и COPY.

Примеры команд для работы с файловой системой

Рассмотрим некоторые наиболее часто используемые команды для работы с файловой системой. Отметим сначала несколько особенностей определения путей к файлам в Windows.

1.5 Основные команды

Полный список команд можно вывести набрав HELP в командной строке.

Команда CD

Текущий каталог можно изменить с помощью команды

CD [диск:][путь\]

Путь к требуемому каталогу указывается с учетом приведенных выше замечаний. Например, команда CD \ выполняет переход в корневой каталог текущего диска. Если запустить команду CD без параметров, то на экран будут выведены имена текущего диска и каталога.

Команда COPY

Одной из наиболее часто повторяющихся задач при работе на компьютере является копирование и перемещение файлов из одного места в другое. Для копирования одного или нескольких файлов используется команда COPY.

Синтаксис этой команды:

COPY [/A|/B] источник [/A|/B] [+ источник [/A|/B] [+ ...]]
[результат [/A|/B]] [/V][/Y|/–Y]

Краткое описание параметров и ключей команды COPY приведено в табл. 1.1.

Таблица 1.1. Параметры и ключи команды COPY

Параметр	Описание
источник	Имя копируемого файла или файлов
/A	Файл является текстовым файлом ASCII, то есть конец файла обозначается символом с кодом ASCII 26 (<Ctrl>+<Z>)
/B	Файл является двоичным. Этот ключ указывает на то, что интерпретатор команд должен при копировании считывать из источника число байт, заданное размером в каталоге копируемого файла
результат	Каталог для размещения результата копирования и/или имя создаваемого

	файла
/V	Проверка правильности копирования путем сравнения файлов после копирования
/Y	Отключение режима запроса подтверждения на замену файлов
/-Y	Включение режима запроса подтверждения на замену файлов

Приведем примеры использования команды COPY.

Копирование файла abc.txt из текущего каталога в каталог D:\PROGRAM под тем же именем:

```
COPY abc.txt D:\PROGRAM
```

Копирование файла abc.txt из текущего каталога в каталог D:\PROGRAM под новым именем def.txt:

```
COPY abc.txt D:\PROGRAM\def.txt
```

Копирование всех файлов с расширением txt с диска A: в каталог 'Мои документы' на диске C:

```
COPY A:\*.txt "C:\Мои документы"
```

Если не задать в команде целевой файл, то команда COPY создаст копию файла-источника с тем же именем, датой и временем создания, что и исходный файл, и поместит новую копию в текущий каталог на текущем диске. Например, для того, чтобы скопировать все файлы из корневого каталога диска A: в текущий каталог, достаточно выполнить такую краткую команду:

```
COPY A:\*.*
```

В качестве источника или результата при копировании можно указывать имена не только файлов, но и устройств компьютера. Например, для того, чтобы распечатать файл abc.txt на принтере, можно воспользоваться командой копирования этого файла на устройство PRN:

```
COPY abc.txt PRN
```

Другой интересный пример: создадим новый текстовый файл и запишем в него информацию, без использования текстового редактора. Для этого достаточно ввести команду `COPY CON my.txt`, которая будет копировать то, что вы набираете на клавиатуре, в файл my.txt (если этот файл существовал, то он перезапишется, иначе — создастся). Для завершения ввода необходимо ввести символ конца файла, то есть нажать клавиши <Ctrl>+<Z>.

Команда COPY может также объединять (склеивать) нескольких файлов в один. Для этого необходимо указать единственный результирующий файл и несколько исходных. Это достигается путем использования групповых знаков (? и *) или формата файл1 + файл2 + файл3. Например, для объединения файлов 1.txt и 2.txt в файл 3.txt можно задать следующую команду:

```
COPY 1.txt+2.txt 3.txt
```

Объединение всех файлов с расширением dat из текущего каталога в один файл all.dat может быть произведено так:

```
COPY /B *.dat all.dat
```

Ключ `/B` здесь используется для предотвращения усечения соединяемых файлов, так как при комбинировании файлов команда COPY по умолчанию считает файлами текстовыми. Если имя целевого файла совпадает с именем одного из копируемых файлов (кроме первого), то исходное содержимое целевого файла теряется. Если имя целевого файла опущено, то в его качестве используется первый файл из списка. Например, команда `COPY 1.txt+2.txt` добавит к содержимому файла 1.txt содержимое файла 2.txt. Командой COPY можно воспользоваться и для присвоения какому-либо файлу текущей даты и времени без модификации его содержимого. Для этого нужно ввести команду типа

```
COPY /B 1.txt +,,
```

Здесь запятые указывают на пропуск параметра приемника, что и приводит к требуемому результату.

Команда COPY имеет и свои недостатки. Например, с ее помощью нельзя копировать скрытые и системные файлы, файлы нулевой длины, файлы из подкаталогов. Кроме того, если при копировании группы файлов COPY встретит файл, который в данный момент нельзя скопировать (например, он занят другим приложением), то процесс копирования полностью прервется, и остальные файлы не будут скопированы.

Команда XCOPY

Указанные при описании команды COPY проблемы можно решить с помощью команды XCOPY, которая предоставляет намного больше возможностей при копировании. Необходимо отметить, правда, что XCOPY может работать только с файлами и каталогами, но не с устройствами.

Синтаксис этой команды:

XCOPY источник [результат] [ключи]

Команда XCOPY имеет множество ключей, мы коснемся лишь некоторых из них. Ключ /D[:[дата]] позволяет копировать только файлы, измененные не ранее указанной даты. Если параметр дата не указан, то копирование будет производиться только если источник новее результата. Например, команда

XCOPY "C:\Мои документы*.*" "D:\BACKUP\Мои документы" /D

скопирует в каталог 'D:\BACKUP\Мои документы' только те файлы из каталога 'C:\Мои документы', которые были изменены со времени последнего подобного копирования или которых вообще не было в 'D:\BACKUP\Мои документы'.

Ключ /S позволяет копировать все непустые подкаталоги в каталоге-источнике. С помощью же ключа /E можно копировать вообще все подкаталоги, включая и пустые. Если указан ключ /C, то копирование будет продолжаться даже в случае возникновения ошибок. Это бывает очень полезным при операциях копирования, производимых над группами файлов, например, при резервном копировании данных.

Ключ /I важен для случая, когда копируются несколько файлов, а файл назначения отсутствует. При задании этого ключа команда XCOPY считает, что файл назначения должен быть каталогом. Например, если задать ключ /I в команде копирования всех файлов с расширением txt из текущего каталога в несуществующий еще подкаталог TEXT,

XCOPY *.txt TEXT /I

то подкаталог TEXT будет создан без дополнительных запросов.

Ключи /Q, /F и /L отвечают за режим отображения при копировании. При задании ключа /Q имена файлов при копировании не отображаются, ключа /F — отображаются полные пути источника и результата. Ключ /L обозначает, что отображаются только файлы, которые должны быть скопированы (при этом само копирование не производится).

С помощью ключа /H можно копировать скрытые и системные файлы, а с помощью ключа /R — заменять файлы с атрибутом "Только для чтения". Например, для копирования всех файлов из корневого каталога диска C: (включая системные и скрытые) в каталог SYS на диске D:, нужно ввести следующую команду:

XCOPY C:*.* D:\SYS /H

Ключ /T позволяет применять XCOPY для копирования только структуры каталогов источника, без дублирования находящихся в этих каталогах файлов, причем пустые каталоги и подкаталоги не включаются. Для того, чтобы все же включить пустые каталоги и подкаталоги, нужно использовать комбинацию ключей /T /E.

Используя XCOPY можно при копировании обновлять только уже существующие файлы (новые файлы при этом не записываются). Для этого применяется ключ /U. Например, если в каталоге C:\2 находились файлы a.txt и b.txt, а в каталоге C:\1 — файлы a.txt, b.txt, c.txt и d.txt, то после выполнения команды

XCOPY C:\1 C:\2 /U

в каталоге C:\2 по-прежнему останутся лишь два файла a.txt и b.txt, содержимое которых будет заменено содержимым соответствующих файлов из каталога C:\1.Если с помощью

XCOPY копировался файл с атрибутом "Только для чтения", то по умолчанию у файла-копии этот атрибут снимется. Для того, чтобы копировать не только данные, но и полностью атрибуты файла, необходимо использовать ключ /K.
Ключи /Y и /-Y определяют, нужно ли запрашивать подтверждение перед заменой файлов при копировании. /Y означает, что такой запрос нужен, /-Y — не нужен.

Команда DIR

Еще одной очень полезной командой является DIR [диск:][путь][имя_файла] [ключи], которая используется для вывода информации о содержимом дисков и каталогов. Параметр [диск:][путь] задает диск и каталог, содержимое которого нужно вывести на экран. Параметр [имя_файла] задает файл или группу файлов, которые нужно включить в список. Например, команда

DIR C:*.bat

выведет на экран все файлы с расширением bat в корневом каталоге диска C:. Если задать эту команду без параметров, то выводится метка диска и его серийный номер, имена (в коротком и длинном вариантах) файлов и подкаталогов, находящихся в текущем каталоге, а также дата и время их последней модификации. После этого выводится число файлов в каталоге, общий объем (в байтах), занимаемый файлами, и объем свободного пространства на диске. Например:

```
Том в устройстве C имеет метку PHYS1_PART2
 Серийный номер тома: 366D-6107
 Содержимое папки C:\aditor
.           <ПАПКА>     25.01.00  17:15 .
..          <ПАПКА>     25.01.00  17:15 ..
TEMPLT02 DAT         227  07.08.98   1:00 templt02.dat
UNINST1  000       1 093  02.03.99   8:36 UNINST1.000
HILITE   DAT       1 082  18.09.98  18:55 hilite.dat
TEMPLT01 DAT          48  07.08.98   1:00 templt01.dat
UNINST0  000      40 960  15.04.98   2:08 UNINST0.000
TTABLE   DAT         357  07.08.98   1:00 ttable.dat
ADITOR   EXE      461 312  01.12.99  23:13 aditor.exe
README   TXT        3 974  25.01.00  17:26 readme.txt
ADITOR   HLP       24 594  08.10.98  23:12 aditor.hlp
ТЕКСТО~1 TXT           0  11.03.01   9:02 Текстовый файл.txt
       11 файлов       533 647 байт
        2 папок     143 261 696 байт свободно
```

С помощью ключей команды DIR можно задать различные режимы расположения, фильтрации и сортировки. Например, при использовании ключа /W перечень файлов выводится в широком формате с максимально возможным числом имен файлов или каталогов на каждой строке. Например:

```
Том в устройстве C имеет метку PHYS1_PART2
 Серийный номер тома: 366D-6107
 Содержимое папки C:\aditor
[.]             [..]            TEMPLT02.DAT    UNINST1.000     HILITE.DAT
TEMPLT01.DAT    UNINST0.000     TTABLE.DAT      ADITOR.EXE
        README.TXT
ADITOR.HLP      ТЕКСТО~1.TXT
       11 файлов       533 647 байт
        2 папок     143 257 600 байт свободно
```

С помощью ключа /A[[:]атрибуты] можно вывести имена только тех каталогов и файлов, которые имеют заданные атрибуты (R — "Только чтение", A — "Архивный", S — "Системный", H — "Скрытый", префикс "–" имеет значение НЕ). Если ключ /A

используется более чем с одним значением атрибута, будут выведены имена только тех файлов, у которых все атрибуты совпадают с заданными. Например, для вывода имен всех файлов в корневом каталоге диска C:, которые одновременно являются скрытыми и системными, можно задать команду

DIR C:\ /A:HS

а для вывода всех файлов, кроме скрытых — команду

DIR C:\ /A:-H

Отметим здесь, что атрибуту каталога соответствует буква D, то есть для того, чтобы, например, вывести список всех каталогов диска C:, нужно задать команду

DIR C: /A:D

Ключ `/O[[:]сортировка]` задает порядок сортировки содержимого каталога при выводе его командой `DIR`. Если этот ключ опущен, `DIR` печатает имена файлов и каталогов в том порядке, в котором они содержатся в каталоге. Если ключ `/O` задан, а параметр сортировка не указан, то `DIR` выводит имена в алфавитном порядке. В параметре сортировка можно использовать следующие значения: `N` — по имени (алфавитная), `S` — по размеру (начиная с меньших), `E` — по расширению (алфавитная), `D` — по дате (начиная с более старых), `A` — по дате загрузки (начиная с более старых), `G` — начать список с каталогов. Префикс "–" означает обратный порядок. Если задается более одного значения порядка сортировки, файлы сортируются по первому критерию, затем по второму и т.д. Ключ `/S` означает вывод списка файлов из заданного каталога и его подкаталогов. Ключ `/B` перечисляет только названия каталогов и имена файлов (в длинном формате) по одному на строку, включая расширение. При этом выводится только основная информация, без итоговой. Например:

templt02.dat
UNINST1.000
hilite.dat
templt01.dat
UNINST0.000
ttable.dat
aditor.exe
readme.txt
aditor.hlp
Текстовый файл.txt

Команды MKDIR и RMDIR

Для создания нового каталога и удаления уже существующего пустого каталога используются команды `MKDIR [диск:]путь` и `RMDIR [диск:]путь [ключи]` соответственно (или их короткие аналоги `MD` и `RD`). Например:

```
MKDIR "C:\Примеры"
RMDIR "C:\Примеры"
```

Команда `MKDIR` не может быть выполнена, если каталог или файл с заданным именем уже существует. Команда `RMDIR` не будет выполнена, если удаляемый каталог не пустой.

Команда DEL

Удалить один или несколько файлов можно с помощью команды

DEL [диск:][путь]имя_файла [ключи]

Для удаления сразу нескольких файлов используются групповые знаки `?` и `*`. Ключ `/S` позволяет удалить указанные файлы из всех подкаталогов, ключ `/F` – принудительно удалить файлы, доступные только для чтения, ключ `/A[[:]атрибуты]` – отбирать файлы для удаления по атрибутам (аналогично ключу `/A[[:]атрибуты]` в команде `DIR`).

Команда REN

Переименовать файлы и каталоги можно с помощью команды `RENAME` (`REN`). Синтаксис этой команды имеет следующий вид:

REN [диск:][путь][каталог1|файл1] [каталог2|файл2]

Здесь параметр `каталог1|файл1` определяет название каталога/файла, которое нужно изменить, а `каталог2|файл2` задает новое название каталога/файла. В любом параметре команды `REN` можно использовать групповые символы ? и *. При этом представленные шаблонами символы в параметре `файл2` будут идентичны соответствующим символам в параметре `файл1`. Например, чтобы изменить у всех файлов с расширением txt в текущей директории расширение на doc, нужно ввести такую команду:

REN *.txt *.doc

Если файл с именем файл2 уже существует, то команда `REN` прекратит выполнение, и произойдет вывод сообщения, что файл уже существует или занят. Кроме того, в команде `REN` нельзя указать другой диск или каталог для создания результирующих каталога и файла. Для этой цели нужно использовать команду `MOVE`, предназначенную для переименования и перемещения файлов и каталогов.

Команда MOVE

Синтаксис команды для перемещения одного или более файлов имеет вид:

MOVE [/Y|/–Y] [диск:][путь]имя_файла1[,...] результирующий_файл

Синтаксис команды для переименования папки имеет вид:

MOVE [/Y|/–Y] [диск:][путь]каталог1 каталог2

Здесь параметр `результирующий_файл` задает новое размещение файла и может включать имя диска, двоеточие, имя каталога, либо их сочетание. Если перемещается только один файл, допускается указать его новое имя. Это позволяет сразу переместить и переименовать файл. Например,

MOVE "C:\Мои документы\список.txt" D:\list.txt

Если указан ключ `/-Y`, то при создании каталогов и замене файлов будет выдаваться запрос на подтверждение. Ключ `/Y` отменяет выдачу такого запроса.

1.6 Учебное задание

1. Запустить командный процессор: `cmd.exe`
2. Создать на диске E: папку: `mkdir test или md test`
3. Войти в созданную папку: `cd test`
4. В паке создать текстовый файл с информацией о студенте (ФИО, дата рождения, группа): `Copy con my.txt`
   Примечание: Для окончания ввода нажать CTRL+Z и затем ENTER.
5. Вывести содержимое паки на экран: `dir`
6. Вывести содержимое созданного файла на экран: `type my.txt`
7. В созданной папке создать две папки с именами A1 и A2.
   Примечание: Для повтора предыдущей команды удобно использовать кнопки – «вверх», «вниз», «вправо».
8. Скопировать *my.txt* в папку A1: `copy my.txt A1`.
9. В папке A2 создать файл с деревом каталога *test*: `tree . >A2\tree.txt`
10. Переименовать файл в папке A1 в *old.txt*: `ren a1\ my.txt old.txt`
11. С помощью команды *move* поменять содержимое папок A1 и A2.
12. Удалить все папки и файлы в исходной папке *test*: `rd /s .`

1.7 Контрольное задание

Задание выполняется в командном процессоре. Все команды привести в отчете по пунктам.

1. Создать папку ***Zadanie1***, а в ней три папки A1, A2, A3.
2. В папку A1 скопировать все файлы с расширением ***ini*** из каталога ***c:\windows***.
3. В папке A2 создать файл с именами файлов в паке ***c:\windows***.

4. скопировать содержимое папок A1 и A2 в A3 и вывести ее содержимое на экран. При выполнении задания использовать группировку команд (**&**).
5. Заменить расширение у всех файлов из папки A1 с ***ini*** на ***bak*** и перед именем каждого файла поставить символ **~**.
6. Сохранить значения системных переменных окружения в файле с именем ***set.txt*** в папке A3.
7. Создать свою системную переменную myname с фамилией студента. Сохранить значения системных переменных окружения в файле с именем ***new_set.txt*** в папке A3. Файл привести в отчете.
8. Показать созданную структуру преподавателю .Удалить папки A1, A2, A3.

2. Язык интерпретатора Cmd.exe. Командные файлы

Язык оболочки командной строки (shell language) в Windows реализован в виде командных (или пакетных) файлов. Командный файл в Windows — это обычный текстовый файл с расширением bat или cmd, в котором записаны допустимые команды операционной системы (как внешние, так и внутренние), а также некоторые дополнительные инструкции и ключевые слова, придающие командным файлам некоторое сходство с алгоритмическими языками программирования.

2.1 Вывод сообщений и дублирование команд

По умолчанию команды пакетного файла перед исполнением выводятся на экран, что выглядит не очень эстетично. С помощью команды `ECHO OFF` можно отключить дублирование команд, идущих после нее (сама команда `ECHO OFF` при этом все же дублируется). Например,

```
REM Следующие две команды будут дублироваться на экране …
DIR C:\
ECHO OFF
REM А остальные уже не будут
DIR D:\
```

Для восстановления режима дублирования используется команда `ECHO ON`. Кроме этого, можно отключить дублирование любой отдельной строки в командном файле, написав в начале этой строки символ `@`, например:

```
ECHO ON
REM Команда DIR C:\ дублируется на экране
DIR C:\
REM А команда DIR D:\ — нет
@DIR D:\
```

Таким образом, если поставить в самое начало файла команду

```
@ECHO OFF
```

то это решит все проблемы с дублированием команд.

В пакетном файле можно выводить на экран строки с сообщениями. Делается это с помощью команды

```
ECHO сообщение
```

Например,

```
@ECHO OFF
ECHO Привет!
Команда ECHO. (точка должна следовать непосредственно за словом "ECHO")
выводит на экран пустую строку.
```

Часто бывает удобно для просмотра сообщений, выводимых из пакетного файла, предварительно полностью очистить экран командой `CLS`.

Используя механизм перенаправления ввода/вывода (символы > и >>), можно направить сообщения, выводимые командой `ECHO`, в определенный текстовый файл. Например:

```
@ECHO OFF
ECHO Привет! > hi.txt
ECHO Пока! >> hi.txt
С помощью такого метода можно, скажем,
 заполнять файлы-протоколы с отчетом о
  произведенных действиях. Например:
@ECHO OFF
REM Попытка копирования
XCOPY C:\PROGRAMS D:\PROGRAMS /s
```

```
REM Добавление сообщения в файл report.txt в случае
REM удачного завершения копирования
IF NOT ERRORLEVEL 1 ECHO Успешное копирование >> report.txt
```

2.2 Использование параметров командной строки

При запуске пакетных файлов в командной строке можно указывать произвольное число параметров, значения которых можно использовать внутри файла. Это позволяет, например, применять один и тот же командный файл для выполнения команд с различными параметрами.

Для доступа из командного файла к параметрам командной строки применяются символы `%0, %1, …, %9` или `%*`. При этом вместо `%0` подставляется имя выполняемого пакетного файла, вместо `%1, %2, …, %9` — значения первых девяти параметров командной строки соответственно, а вместо `%*` — все аргументы. Если в командной строке при вызове пакетного файла задано меньше девяти параметров, то "лишние" переменные из `%1 – %9` замещаются пустыми строками. Рассмотрим следующий пример. Пусть имеется командный файл copier.bat следующего содержания:

```
@ECHO OFF
CLS
ECHO Файл %0 копирует каталог %1 в %2
XCOPY %1 %2 /S
```

Если запустить его из командной строки с двумя параметрами, например

```
copier.bat C:\Programs D:\Backup
```

то на экран выведется сообщение

```
Файл copier.bat копирует каталог C:\Programs в D:\Backup
```

и произойдет копирование каталога C:\Programs со всеми его подкаталогами в D:\Backup.

При необходимости можно использовать более девяти параметров командной строки. Это достигается с помощью команды `SHIFT`, которая изменяет значения замещаемых параметров с `%0` по `%9`, копируя каждый параметр в предыдущий, то есть значение `%1` копируется в `%0`, значение `%2` – в `%1` и т.д. Замещаемому параметру `%9` присваивается значение параметра, следующего в командной строке за старым значением `%9`. Если же такой параметр не задан, то новое значение `%9` — пустая строка.

Рассмотрим пример. Пусть командный файл my.bat вызван из командной строки следующим образом:

```
my.bat p1 p2 p3
```

Тогда `%0=my.bat, %1=p1, %2=p2, %3=p3`, параметры `%4 – %9` являются пустыми строками. После выполнения команды `SHIFT` значения замещаемых параметров изменятся следующим образом: `%0=p1, %1=p2, %2=p3`, параметры `%3 – %9` – пустые строки.

При включении расширенной обработки команд `SHIFT` поддерживает ключ `/n`, задающий начало сдвига параметров с номера `n`, где `n` может быть числом от 0 до 9.

Например, в следующей команде:

```
SHIFT /2
```

параметр `%2` заменяется на `%3`, `%3` на `%4` и т.д., а параметры `%0` и `%1` остаются без изменений.

Команда, обратная `SHIFT` (обратный сдвиг), отсутствует. После выполнения `SHIFT` уже нельзя восстановить параметр `(%0)`, который был первым перед сдвигом. Если в командной строке задано больше десяти параметров, то команду `SHIFT` можно использовать несколько раз.

В командных файлах имеются некоторые возможности синтаксического анализа заменяемых параметров. Для параметра с номером `n (%n)` допустимы синтаксические конструкции (операторы), представленные в табл. 2.1.

Таблица 2.1. Операторы для заменяемых параметров	
Операторы	**Описание**
%~Fn	Переменная %n расширяется до полного имени файла
%~Dn	Из переменной %n выделяется только имя диска
%~Pn	Из переменной %n выделяется только путь к файлу
%~Nn	Из переменной %n выделяется только имя файла
%~Xn	Из переменной %n выделяется расширение имени файла
%~Sn	Значение операторов N и X для переменной %n изменяется так, что они работают с кратким именем файла
%~$PATH:n	Проводится поиск по каталогам, заданным в переменной среды PATH, и переменная %n заменяется на полное имя первого найденного файла. Если переменная PATH не определена или в результате поиска не найден ни один файл, эта конструкция заменяется на пустую строку. Естественно, здесь переменную PATH можно заменить на любое другое допустимое значение

Данные синтаксические конструкции можно объединять друг с другом, например:
%~DPn — из переменной %n выделяется имя диска и путь,
%~NXn — из переменной %n выделяется имя файла и расширение.
Рассмотрим следующий пример. Пусть мы находимся в каталоге C:\TEXT и запускаем пакетный файл с параметром Рассказ.doc (%1=Рассказ.doc). Тогда применение операторов, описанных в табл. 2.1, к параметру %1 даст следующие результаты:

```
%~F1=C:\TEXT\Рассказ.doc
%~D1=C:
%~P1=\TEXT\
%~N1=Рассказ
%~X1=.doc
%DP1=C:\TEXT\
%NX1=Рассказ.doc
```

2.3 Работа с переменными среды

Внутри командных файлов можно работать с так называемыми переменными среды (или переменными окружения), каждая из которых хранится в оперативной памяти, имеет свое уникальное имя, а ее значением является строка. Стандартные переменные среды автоматически инициализируются в процессе загрузки операционной системы. Такими переменными являются, например, WINDIR, которая определяет расположение каталога Windows, TEMP, которая определяет путь к каталогу для хранения временных файлов Windows или PATH, в которой хранится системный путь (путь поиска), то есть список каталогов, в которых система должна искать выполняемые файлы или файлы совместного доступа (например, динамические библиотеки). Кроме того, в командных файлах с помощью команды SET можно объявлять собственные переменные среды.

Получение значения переменной

Для получения значения определенной переменной среды нужно имя этой переменной заключить в символы %. Например:

```
@ECHO OFF
CLS
REM Создание переменной MyVar
SET MyVar=Привет
REM Изменение переменной
SET MyVar=%MyVar%!
```

```
ECHO Значение переменной MyVar: %MyVar%
REM Удаление переменной MyVar
SET MyVar=
ECHO Значение переменной WinDir: %WinDir%
```

При запуске такого командного файла на экран выведется строка

```
Значение переменной MyVar: Привет!
Значение переменной WinDir: C:\WINDOWS
```

2.4 Преобразования переменных как строк

С переменными среды в командных файлах можно производить некоторые манипуляции. Во-первых, над ними можно производить операцию конкатенации (склеивания). Для этого нужно в команде `SET` просто написать рядом значения соединяемых переменных. Например,

```
  SET A=Раз
SET B=Два
SET C=%A%%B%
```

После выполнения в файле этих команд значением переменной `C` будет являться строка `'РазДва'`. Не следует для конкатенации использовать знак `+`, так как он будет воспринят просто в качестве символа. Например, после запуска файл следующего содержания

```
  SET A=Раз
SET B=Два
SET C=A+B
ECHO Переменная C=%C%
SET D=%A%+%B%
ECHO Переменная D=%D%
```

на экран выведутся две строки:

```
Переменная C=A+B
Переменная D=Раз+Два
```

2.5 Операции с переменными как с числами

При включенной расширенной обработке команд (этот режим в Windows XP используется по умолчанию) имеется возможность рассматривать значения переменных среды как числа и производить с ними арифметические вычисления. Для этого используется команда `SET` с ключом `/A`. Приведем пример пакетного файла add.bat, складывающего два числа, заданных в качестве параметров командной строки, и выводящего полученную сумму на экран:

```
@ECHO OFF
REM В переменной M будет храниться сумма
SET /A M=%1+%2
ECHO Сумма %1 и %2 равна %M%
REM Удалим переменную M
SET M=
```

2.6 Локальные изменения переменных

Все изменения, производимые с помощью команды `SET` над переменными среды в командном файле, сохраняются и после завершения работы этого файла, но действуют только внутри текущего командного окна. Также имеется возможность локализовать изменения переменных среды внутри пакетного файла, то есть автоматически восстанавливать значения всех переменных в том виде, в каком они были до начала запуска этого файла. Для этого используются две команды: `SETLOCAL` и `ENDLOCAL`.

Команда SETLOCAL определяет начало области локальных установок переменных среды. Другими словами, изменения среды, внесенные после выполнения SETLOCAL, будут являться локальными относительно текущего пакетного файла. Каждая команда SETLOCAL должна иметь соответствующую команду ENDLOCAL для восстановления прежних значений переменных среды. Изменения среды, внесенные после выполнения команды ENDLOCAL, уже не являются локальными относительно текущего пакетного файла; их прежние значения не будут восстановлены по завершении выполнения этого файла.

2.7 Приостановка выполнения командных файлов

Для того, чтобы вручную прервать выполнение запущенного bat-файла, нужно нажать клавиши <Ctrl>+<C> или <Ctrl>+<Break>. Однако часто бывает необходимо программно приостановить выполнение командного файла в определенной строке с выдачей запроса на нажатие любой клавиши. Это делается с помощью команды PAUSE. Перед запуском этой команды полезно с помощью команды ECHO информировать пользователя о действиях, которые он должен произвести. Например:

```
ECHO Вставьте дискету в дисковод A: и нажмите любую клавишу
PAUSE
```

2.8 Операторы перехода

Командный файл может содержать метки и команды GOTO перехода к этим меткам. Любая строка, начинающаяся с двоеточия :, воспринимается при обработке командного файла как метка. Имя метки задается набором символов, следующих за двоеточием до первого пробела или конца строки. Приведем пример.

Пусть имеется командный файл следующего содержания:

```
@ECHO OFF
COPY %1 %2
GOTO Label1
ECHO Эта строка никогда не выполнится
:Label1
REM Продолжение выполнения
DIR %2
После того, как в этом файле мы доходим до команды
GOTO Label1
его выполнение продолжается со строки
REM Продолжение выполнения
```

В команде перехода внутри файла GOTO можно задавать в качестве метки перехода строку :EOF, которая передает управление в конец текущего пакетного файла (это позволяет легко выйти из пакетного файла без определения каких-либо меток в самом его конце). Также для перехода к метке внутри текущего командного файла кроме команды GOTO можно использовать и рассмотренную выше команду CALL:

```
CALL :метка аргументы
```

При вызове такой команды создается новый контекст текущего пакетного файла с заданными аргументами, и управление передается на инструкцию, расположенную сразу после метки. Для выхода из такого пакетного файла необходимо два раза достичь его конца. Первый выход возвращает управление на инструкцию, расположенную сразу после строки CALL, а второй выход завершает выполнение пакетного файла. Например, если запустить с параметром "Копия-1" командный файл следующего содержания:

```
@ECHO OFF
ECHO %1
```

```
CALL :2 Копия-2
:2
ECHO %1
```
то на экран выведутся три строки:
```
Копия-1
Копия-2
Копия-1
```
Таким образом, подобное использование команды `CALL` очень похоже на обычный вызов подпрограмм (процедур) в алгоритмических языках программирования.

2.9 Операторы условия

С помощью команды `IF ... ELSE` (ключевое слово `ELSE` может отсутствовать) в пакетных файлах можно выполнять обработку условий нескольких типов. При этом если заданное после `IF` условие принимает истинное значение, система выполняет следующую за условием команду (или несколько команд, заключенных в круглые скобки), в противном случае выполняется команда (или несколько команд в скобках), следующие за ключевым словом `ELSE`.

Проверка значения переменной

Первый тип условия используется обычно для проверки значения переменной. Для этого применяются два варианта синтаксиса команды `IF`:
```
IF [NOT] строка1==строка2 команда1 [ELSE команда2]
```
(квадратные скобки указывают на необязательность заключенных в них параметров) или
```
IF [/I] [NOT] строка1 оператор_сравнения строка2 команда
```
Рассмотрим сначала первый вариант. Условие `строка1==строка2` (здесь необходимо писать именно два знака равенства) считается истинным при точном совпадении обеих строк. Параметр `NOT` указывает на то, что заданная команда выполняется лишь в том случае, когда сравниваемые строки не совпадают.

Строки могут быть литеральными или представлять собой значения переменных (например, `%1` или `%TEMP%`). Кавычки для литеральных строк не требуются. Например,
```
IF %1==%2 ECHO Параметры совпадают!
IF %1==Петя ECHO Привет, Петя!
```
Отметим, что при сравнении строк, заданных переменными, следует проявлять определенную осторожность. Дело в том, что значение переменной может оказаться пустой строкой, и тогда может возникнуть ситуация, при которой выполнение командного файла аварийно завершится. Например, если вы не определили с помощью команды `SET` переменную `MyVar`, а в файле имеется условный оператор типа
```
IF %MyVar%==C:\ ECHO Ура!!!
```
то в процессе выполнения вместо `%MyVar%` подставится пустая строка и возникнет синтаксическая ошибка. Такая же ситуация может возникнуть, если одна из сравниваемых строк является значением параметра командной строки, так как этот параметр может быть не указан при запуске командного файла. Поэтому при сравнении строк нужно приписывать к ним в начале какой-нибудь символ, например:
```
IF -%MyVar%==-C:\ ECHO Ура!!!
```
С помощью команд `IF` и `SHIFT` можно в цикле обрабатывать все параметры командной строки файла, даже не зная заранее их количества. Например, следующий командный файл (назовем его primer.bat) выводит на экран имя запускаемого файла и все параметры командной строки:
```
@ECHO OFF
ECHO Выполняется файл: %0
ECHO.
ECHO Файл запущен со следующими параметрами...
```

```
REM Начало цикла
:BegLoop
IF -%1==- GOTO ExitLoop
ECHO %1
REM Сдвиг параметров
SHIFT
REM Переход на начало цикла
GOTO BegLoop
:ExitLoop
REM Выход из цикла
ECHO.
ECHO Все.
```

Если запустить primer.bat с четырьмя параметрами:

```
primer.bat А Б В Г
```

то в результате выполнения на экран выведется следующая информация:

```
Выполняется файл: primer.bat

Файл запущен со следующими параметрами:
А
Б
В
Г

Все.
```

Рассмотрим теперь оператор `IF` в следующем виде:

```
IF [/I] строка1 оператор_сравнения строка2 команда
```

Синтаксис и значение `операторов_сравнения` представлены в табл. 2.2.

Таблица 2.2. Операторы сравнения в IF

Оператор	Значение
`EQL`	Равно
`NEQ`	Не равно
`LSS`	Меньше
`LEQ`	Меньше или равно
`GTR`	Больше
`GEQ`	Больше или равно

Приведем пример использования операторов сравнения:

```
@ECHO OFF
CLS
IF -%1 EQL –Вася ECHO Привет, Вася!
IF -%1 NEQ –Вася ECHO Привет, но Вы не Вася!
```

Ключ `/I`, если он указан, задает сравнение текстовых строк без учета регистра. Ключ `/I` можно также использовать и в форме `строка1==строка2` команды `IF`. Например, условие

```
IF /I DOS==dos …
```

будет истинным.

Проверка существования заданного файла

Второй способ использования команды `IF` — это проверка существования заданного файла. Синтаксис для этого случая имеет вид:

```
IF [NOT] EXIST файл команда1 [ELSE команда2]
```

Условие считается истинным, если указанный файл существует. Кавычки для имени файла не требуются. Приведем пример командного файла, в котором с помощью такого варианта команды IF проверяется наличие файла, указанного в качестве параметра командной строки.

```
@ECHO OFF
IF -%1==- GOTO NoFileSpecified
IF NOT EXIST %1 GOTO FileNotExist

REM Вывод сообщения о найденном файле
ECHO Файл '%1' успешно найден.
GOTO :EOF

:NoFileSpecified
REM Файл запущен без параметров
ECHO В командной строке не указано имя файла.
GOTO :EOF

:FileNotExist
REM Параметр командной строки задан, но файл не найден
ECHO Файл '%1' не найден.
```

Проверка наличия переменной среды

Аналогично файлам команда `IF` позволяет проверить наличие в системе определенной переменной среды:

```
IF DEFINED переменная команда1 [ELSE команда2]
```

Здесь условие `DEFINED` применяется подобно условию `EXISTS` наличия заданного файла, но принимает в качестве аргумента имя переменной среды и возвращает истинное значение, если эта переменная определена. Например:

```
@ECHO OFF
CLS
IF DEFINED MyVar GOTO :VarExists
ECHO Переменная MyVar не определена
GOTO :EOF
:VarExists
ECHO Переменная MyVar определена,
ECHO ее значение равно %MyVar%
```

Проверка кода завершения предыдущей команды

Еще один способ использования команды `IF` — это проверка кода завершения (кода выхода) предыдущей команды. Синтаксис для `IF` в этом случае имеет следующий вид:

```
IF [NOT] ERRORLEVEL число команда1 [ELSE команда2]
```

Здесь условие считается истинным, если последняя запущенная команда или программа завершилась с кодом возврата, равным либо превышающим указанное число.

Составим, например, командный файл, который бы копировал файл my.txt на диск C: без вывода на экран сообщений о копировании, а в случае возникновения какой-либо ошибки выдавал предупреждение:

```
@ECHO OFF
XCOPY my.txt C:\ > NUL
REM Проверка кода завершения копирования
IF ERRORLEVEL 1 GOTO ErrOccurred
ECHO Копирование выполнено без ошибок.
GOTO :EOF

:ErrOccurred
```

ECHO При выполнении команды XCOPY возникла ошибка!
В операторе `IF ERRORLEVEL ...` можно также применять операторы сравнения чисел, приведенные в табл. 2.2. Например:
IF ERRORLEVEL LEQ 1 GOTO Case1

2.10 Организация циклов

В командных файлах для организации циклов используются несколько разновидностей оператора `FOR`, которые обеспечивают следующие функции:
выполнение заданной команды для всех элементов указанного множества;
выполнение заданной команды для всех подходящих имен файлов;
выполнение заданной команды для всех подходящих имен каталогов;
выполнение заданной команды для определенного каталога, а также всех его подкаталогов;
получение последовательности чисел с заданными началом, концом и шагом приращения;
чтение и обработка строк из текстового файла;
обработка строк вывода определенной команды.

Цикл FOR … IN … DO …

Самый простой вариант синтаксиса команды `FOR` для командных файлов имеет следующий вид:
FOR %%переменная IN (множество)
DO команда [параметры]
Внимание
Перед названием переменной должны стоять именно два знака процента `(%%)`, а не один, как это было при использовании команды `FOR` непосредственно из командной строки.
Сразу приведем пример. Если в командном файле заданы строки
@ECHO OFF
FOR %%i IN (Раз,Два,Три) DO ECHO %%i
то в результате его выполнения на экране будет напечатано следующее:
Раз
Два
Три
Параметр `множество` в команде `FOR` задает одну или более текстовых строк, разделенных запятыми, которые вы хотите обработать с помощью заданной команды. Скобки здесь обязательны. Параметр `команда [параметры]` задает команду, выполняемую для каждого элемента множества, при этом вложенность команд `FOR` на одной строке не допускается. Если в строке, входящей во множество, используется запятая, то значение этой строки нужно заключить в кавычки. Например, в результате выполнения файла с командами
@ECHO OFF
FOR %%i IN ("Раз,Два",Три) DO ECHO %%i
на экран будет выведено
Раз,Два
Три
Параметр `%%переменная` представляет подставляемую переменную (счетчик цикла), причем здесь могут использоваться только имена переменных, состоящие из одной буквы. При выполнении команда `FOR` заменяет подставляемую переменную текстом каждой строки в заданном множестве, пока команда, стоящая после ключевого слова `DO`, не обработает все такие строки.
Замечание.
Чтобы избежать путаницы с параметрами командного файла `%0 – %9`, для переменных следует использовать любые символы кроме `0 – 9`.

Параметр множество в команде FOR может также представлять одну или несколько групп файлов. Например, чтобы вывести в файл список всех файлов с расширениями txt и prn, находящихся в каталоге C:\TEXT, без использования команды DIR, можно использовать командный файл следующего содержания:

```
@ECHO OFF
FOR %%f IN (C:\TEXT\*.txt C:\TEXT\*.prn) DO ECHO %%f >> list.txt
```

При таком использовании команды FOR процесс обработки продолжается, пока не обработаются все файлы (или группы файлов), указанные во множестве.

Цикл FOR /D … IN … DO …

Следующий вариант команды FOR реализуется с помощью ключа /D:

```
FOR /D %переменная IN (набор) DO команда [параметры]
```

В случае, если набор содержит подстановочные знаки, то команда выполняется для всех подходящих имен каталогов, а не имен файлов. Скажем, выполнив следующий командный файл:

```
@ECHO OFF
CLS
FOR /D %%f IN (C:\*.*) DO ECHO %%f
```

мы получим список всех каталогов на диске C:, например:

```
C:\Arc
C:\CYR
C:\MSCAN
C:\NC
C:\Program Files
C:\TEMP
C:\TeX
C:\WINNT
```

Цикл FOR /R … IN … DO …

С помощью ключа /R можно задать рекурсию в команде: FOR:

```
FOR /R [[диск:]путь] %переменная IN (набор)
  DO команда [параметры]
```

В этом случае заданная команда выполняется для каталога [диск:]путь, а также для всех подкаталогов этого пути. Если после ключа R не указано имя каталога, то выполнение команды начинается с текущего каталога. Например, для распечатки всех файлов с расширением txt в текущем каталоге и всех его подкаталогах можно использовать следующий пакетный файл:

```
@ECHO OFF
CLS
FOR /R %%f IN (*.txt) DO PRINT %%f
```

Если вместо набора указана только точка (.), то команда проверяет все подкаталоги текущего каталога. Например, если мы находимся в каталоге C:\TEXT с двумя подкаталогами BOOKS и ARTICLES, то в результате выполнения файла:

```
@ECHO OFF
CLS
FOR /R %%f IN (.) DO ECHO %%f
```

на экран выведутся три строки:

```
C:\TEXT\.
C:\TEXT\BOOKS\.
C:\TEXT\ARTICLES\.
```

Цикл FOR /L … IN … DO …

Ключ /L позволяет реализовать с помощью команды FOR арифметический цикл, в этом случае синтаксис имеет следующий: вид:

```
FOR /L %переменная IN (начало,шаг,конец) DO команда [параметры]
```

Здесь заданная после ключевого слова IN тройка (начало,шаг,конец) раскрывается в последовательность чисел с заданными началом, концом и шагом приращения. Так, набор (1,1,5) раскрывается в (1 2 3 4 5), а набор (5,-1,1) заменяется на (5 4 3 2 1). Например, в результате выполнения следующего командного файла:

```
@ECHO OFF
CLS
FOR /L %%f IN (1,1,5) DO ECHO %%f
```

переменная цикла %%f пробежит значения от 1 до 5, и на экране напечатаются пять чисел:

```
1
2
3
4
5
```

Числа, получаемые в результате выполнения цикла FOR /L, можно использовать в арифметических вычислениях. Рассмотрим командный файл my.bat следующего содержания:

```
@ECHO OFF
CLS
FOR /L %%f IN (1,1,5) DO CALL :2 %%f
GOTO :EOF
:2
SET /A M=10*%1
ECHO 10*%1=%M%
```

В третьей строке в цикле происходит вызов нового контекста файла my.bat с текущим значением переменной цикла %%f в качестве параметра командной строки, причем управление передается на метку :2 (см. описание CALL в разделе "Изменения в командах перехода"). В шестой строке переменная цикла умножается на десять, и результат записывается в переменную M. Таким образом, в результате выполнения этого файла выведется следующая информация:

```
10*1=10
10*2=20
10*3=30
10*4=40
10*5=50
```

Цикл FOR /F … IN … DO …

Самые мощные возможности (и одновременно самый запутанный синтаксис) имеет команда: FOR с ключом /F:

```
FOR /F ["ключи"] %переменная IN (набор)
 DO команда [параметры]
```

Здесь параметр набор содержит имена одного или нескольких файлов, которые по очереди открываются, читаются и обрабатываются. Обработка состоит в чтении файла, разбиении его на отдельные строки текста и выделении из каждой строки заданного числа подстрок. Затем найденная подстрока используется в качестве значения переменной при выполнении основного тела цикла (заданной команды).

По умолчанию ключ /F выделяет из каждой строки файла первое слово, очищенное от окружающих его пробелов. Пустые строки в файле пропускаются. Необязательный параметр "ключи" служит для переопределения заданных по умолчанию правил обработки строк. Ключи представляют собой заключенную в кавычки строку, содержащую приведенные в табл. 2.3 ключевые слова:

Таблица 2.3. Ключи в команде FOR /F

Ключ	Описание

EOL=C	Определение символа комментариев в начале строки (допускается задание только одного символа)
SKIP=N	Число пропускаемых при обработке строк в начале файла
DELIMS=XXX	Определение набора разделителей для замены заданных по умолчанию пробела и знака табуляции
TOKENS=X,Y,M-N	Определение номеров подстрок, выделяемых из каждой строки файла и передаваемых для выполнения в тело цикла

При использовании ключа TOKENS=X,Y,M-N создаются дополнительные переменные. Формат M-N представляет собой диапазон подстрок с номерами от M до N. Если последний символ в строке TOKENS= является звездочкой, то создается дополнительная переменная, значением которой будет весь текст, оставшийся в строке после обработки последней подстроки.

Разберем применение этой команды на примере пакетного файла parser.bat, который производит разбор файла myfile.txt:

```
@ECHO OFF
IF NOT EXIST myfile.txt GOTO :NoFile
FOR /F "EOL=; TOKENS=2,3* DELIMS=, " %%i IN
  (myfile.txt) DO @ECHO %%i %%j %%k
GOTO :EOF
:NoFile
ECHO Не найден файл myfile.txt!
```

Здесь во второй строке производится проверка наличия файла myfile.txt; в случае отсутствия этого файла выводится предупреждающее сообщение. Команда FOR в третьей строке обрабатывает файл myfile.txt следующим образом:

Пропускаются все строки, которые начинаются с символа точки с запятой (EOL=;).

Вторая и третья подстроки из каждой строки передаются в тело цикла, причем подстроки разделяются пробелами (по умолчанию) и/или запятыми (DELIMS=,).

В теле цикла переменная %%i используется для второй подстроки, %%j — для третьей, а %%k получает все оставшиеся подстроки после третьей.

В нашем примере переменная %%i явно описана в инструкции FOR, а переменные %%j и %%k описываются неявно с помощью ключа TOKENS=. Например, если в файле myfile.txt были записаны следующие три строки:

```
ААА БББ ВВВВ,ГГГГГ ДДДД
ЕЕЕЕЕ,ЖЖЖЖ ЗЗЗЗ
;КККК ЛЛЛЛЛ МММММ
```

то в результате выполнения пакетного файла parser.bat на экран выведется следующее:

```
БББ ВВВВ ГГГГГ ДДДД
ЖЖЖЖ ЗЗЗЗ
```

Замечание

Ключ TOKENS= позволяет извлечь из одной строки файла до 26 подстрок, поэтому запрещено использовать имена переменных, начинающиеся не с букв английского алфавита (a–z). Следует помнить, что имена переменных FOR являются глобальными, поэтому одновременно не может быть активно более 26 переменных.

Команда FOR /F также позволяет обработать отдельную строку. Для этого следует ввести нужную строку в кавычках вместо набора имен файлов в скобках. Строка будет обработана так, как будто она взята из файла. Например, файл следующего содержания:

```
@ECHO OFF
FOR /F "EOL=; TOKENS=2,3* DELIMS=, " %%i IN
  ("ААА БББ ВВВВ,ГГГГГ ДДДД") DO @ECHO %%i %%j %%k
```

при своем выполнении напечатает

```
ББББ ВВВВ ГГГГГ ДДДД
```
Вместо явного задания строки для разбора можно пользоваться переменными среды, например:
```
@ECHO OFF
SET M=AAA ББББ ВВВВ,ГГГГГ ДДДД
FOR /F "EOL=; TOKENS=2,3* DELIMS=,
  " %%i IN ("%M%") DO @ECHO %%i %%j %%k
```
Наконец, команда `FOR /F` позволяет обработать строку вывода другой команды. Для этого следует вместо набора имен файлов в скобках ввести строку вызова команды в апострофах (не в кавычках!). Строка передается для выполнения интерпретатору команд cmd.exe, а вывод этой команды записывается в память и обрабатывается так, как будто строка вывода взята из файла. Например, следующий командный файл:
```
@ECHO OFF
CLS
ECHO Имена переменных среды:
ECHO.
FOR /F "DELIMS==" %%i IN ('SET') DO ECHO %%i
```
выведет перечень имен всех переменных среды, определенных в настоящее время в системе.
В цикле `FOR` допускается применение тех же синтаксических конструкций (операторов), что и для заменяемых параметров (табл. 2.4).

Таблица 2.4. Операторы для переменных команды FOR

Операторы	Описание
`%~Fi`	Переменная `%i` расширяется до полного имени файла
`%~Di`	Из переменной `%i` выделяется только имя диска
`%~Pi`	Из переменной `%i` выделяется только путь к файлу
`%~Ni`	Из переменной `%i` выделяется только имя файла
`%~Xi`	Из переменной `%i` выделяется расширение имени файла
`%~Si`	Значение операторов `N` и `X` для переменной `%i` изменяется так, что они работают с кратким именем файла

Замечание
Если планируется использовать расширения подстановки значений в команде `FOR`, то следует внимательно подбирать имена переменных, чтобы они не пересекались с обозначениями формата.
Например, если мы находимся в каталоге C:\Program Files\Far и запустим командный файл следующего содержания:
```
@ECHO OFF
CLS
FOR %%i IN (*.txt) DO ECHO %%~Fi
```
то на экран выведутся полные имена всех файлов
с расширением txt:
```
C:\Program Files\Far\Contacts.txt
C:\Program Files\Far\FarFAQ.txt
C:\Program Files\Far\Far_Site.txt
C:\Program Files\Far\License.txt
C:\Program Files\Far\License.xUSSR.txt
C:\Program Files\Far\ReadMe.txt
C:\Program Files\Far\register.txt
C:\Program Files\Far\WhatsNew.txt
```

2.11 Циклы и связывание времени выполнения для переменных

Как и в рассмотренном выше примере с составными выражениями, при обработке переменных среды внутри цикла могут возникать труднообъяснимые ошибки, связанные с ранними связыванием переменных. Рассмотрим пример. Пусть имеется командный файл следующего содержания:

```
SET a=
FOR %%i IN (Раз,Два,Три) DO SET a=%a%%%i
ECHO a=%a%
```

В результате его выполнения на экран будет выведена строка `"a=Три"`, то есть фактически команда

```
FOR %%i IN (Раз,Два,Три) DO SET a=%a%%%i
```

равносильна команде

```
FOR %%i IN (Раз,Два,Три) DO SET a=%%i
```

Для исправления ситуации нужно, как и в случае с составными выражениями, вместо знаков процента (`%`) использовать восклицательные знаки и предварительно включить режим связывания времени выполнения командой `SETLOCAL ENABLEDELAYEDEXPANSION`. Таким образом, наш пример следует переписать следующим образом:

```
SETLOCAL ENABLEDELAYEDEXPANSION
SET a=
FOR %%i IN (Раз,Два,Три) DO SET a=!a!%%i
ECHO a=%a%
```

В этом случае на экран будет выведена строка `"a=РазДваТри"`.

2.12 Учебные задания

В каждом задании разработать пакетный файл. **Все параметры передаются через командную строку. Выполнить обязательную проверку на наличие всех требуемых параметров.** Командный файл и результаты его работы привести в отчете. *Обязательна демонстрация работы пакетного файла преподавателю.*

Пример

Выполнить резервное копирование содержимого заданной папки (с вложенными) в указанное место. Папки задать как параметры. Выполнить проверку на наличие заданных параметров.

```
@ECHO OFF
CLS
ECHO Файл %0 копирует каталог %1 в %2
IF -%1==- GOTO NoParam
IF -%2==- GOTO NoParam
XCOPY %1\ %2 /S
GOTO :eof
:NoParam
ECHO Не заданы необходимые параметры командной строки!
PAUSE
```

Задания

1. В заданном месте создать, если ее нет, папку с именем зарегистрированного пользователя. Создать и (или) дописать в файл с именем «Log In. log» время и дату входа в систему.ту входа в систему.

2. Выполнить резервное копирование файлов заданного расширения из указанной папки (с вложенными) в указанное место. Папки и расширение задать как параметры. Выполнить проверку на наличие заданных параметров, отключить вывод на экран списка копируемых файлов.

3. Удалить на заданном диске все файлы с заданным расширением (bak, tmp) вне зависимости от его атрибутов. Диск и расширение задать как параметры. Выполнить проверку на наличие заданных параметров.

4. Скопировать все файлы с заданным расширением в заданную папку без сохранения структуры каталогов (**четные номера компьютеров**), с сохранением структуры (**нечетные номера компьютеров**),. Папки и расширение задать как параметры. Продемонстрировать работу преподавателю.

5. Сохранить местоположение всех файлов с заданным расширением в файле с указанным именем. Расширение и имя файла задать как параметры. Заархивировать полученный список файлов архиватором rar (rar a имя_архива @файл_список)

6. Составить пакетный файл для копирования заданного файла в соответствии со списком папок хранящихся в другом файле. Имена файлов задать как параметры.

7. Создать на заданном диске папку с именем пользователя вошедшего в систему и сделать ее активной. Диск и имя файла задать как параметры.

8. Создать в указанном месте структуру папок, хранящуюся в заданном файле.

9. Выполнить резервное копирование с последующей архивацией файлов заданного расширения. Архиватор rar (rar a имя_архива @файл_список).

10. Разработать пакетный файл для построения системы студенческих каталогов с запросом на создание каталога группы и числа пользователей в группе.

11. Разработать пакетный файл для перезаписи файлов заданного расширения из одного каталога в другой с обновлением.

12. Разработать пакетный файл для копирования только новых и обновленных файлов заданного расширения из одного каталога в другой.

13. Разработать пакетный файл для вывода заданного сообщения заданному зарегистрированному пользователю. Сообщение и имя передаются как параметры.

14. Создать папку с именем зарегистрированного пользователя на заданном диске и выполнить копирование в ней всех файлов с заданным расширением и сохранением структуры папок.

15. Разработать пакетный файл для перехода в каталог с именем зарегистрированного пользователя, если он существует и архивирования его содержимого.

Лабораторная работа №2

Сервер сценариев Windows Script Host

Теоретические сведения

С помощью Windows Script Host (WSH), предназначен для выполнения сценариев, написанные, в принципе, на любом языке (при условии, что для этого языка установлен соответствующий модуль (scripting engine), поддерживающим технологию ActiveX Scripting. В качестве стандартных языков поддерживаются Visual Basic Script Edition (VBScript) и JScript.
WSH предъявляет минимальные требования к объему оперативной памяти, и является очень удобным инструментом для автоматизации повседневных задач пользователей и администраторов операционной системы Windows. Используя сценарии WSH, можно непосредственно работать с файловой системой компьютера, а также управлять работой других приложений (серверов автоматизации). При этом возможности сценариев ограничены только средствами, которые предоставляют доступные серверы автоматизации.

1.1 Создание и запуск простейших сценариев WSH

Простейший WSH-сценарий, написанный на языке JScript или VBScript — это обычный текстовый файл с расширением js или vbs соответственно, создать его можно в любом текстовом редакторе, способном сохранять документы в формате "Только текст".
Размер сценария может изменяться от одной до тысяч строк, предельный размер ограничивается лишь максимальным размером файла в соответствующей файловой системе.
В качестве первого примера создадим JScript-сценарий, выводящий на экран диалоговое окно с надписью "Привет!". Для этого достаточно с помощью, например, стандартного Блокнота Windows (notepad.exe) создать файл First.js, содержащий всего одну строку:

```
WScript.Echo("Привет!");
```

Тот же самый сценарий на языке VBScript, естественно, отличается синтаксисом и выглядит следующим образом:

```
WScript.Echo "Привет!"
```

1.2 Запуск сценария из командной строки в консольном режиме

Можно выполнить сценарий из командной строки с помощью консольной версии WSH cscript.exe. Например, чтобы запустить сценарий, записанный в файле C:\Script\First.js, нужно загрузить командное окно и выполнить в нем команду

```
cscript C:\Script\First.js
```

Сценарий можно выполнить из командной строки с помощью (оконной) графической версии WSH wscript.exe. Для нашего примера в этом случае нужно выполнить команду

```
wscript C:\Script\First.js
```

1.3 Установка и изменение свойств сценариев

В случае необходимости для сценариев можно задавать различные параметры, влияющие на ход их выполнения. Для консольной (cscript.exe) и графической (wscript.exe) версий сервера сценариев эти параметры задаются по-разному.
Если сценарий запускается в консольном режиме, то его исполнение контролируется с помощью параметров командной строки для cscript.exe (см. табл. 1.1), которые включают или отключают различные опции WSH (все эти параметры начинаются с символов "//").

Таблица 1.1. Параметры командной строки для cscript.exe	
Параметр	Описание
`//I`	Выключает пакетный режим (по умолчанию). При этом на экран будут выводиться все сообщения об ошибках в сценарии
`//B`	Включает пакетный режим. При этом на экран не будут выводиться никакие сообщения
`//T:nn`	Задает тайм-аут в секундах, т. е. сценарий будет выполняться nn секунд, после чего процесс прервется. По умолчанию время выполнения не ограничено
`//Logo`	Выводит (по умолчанию) перед выполнением сценария информацию о версии и разработчике WSH
`//Nologo`	Подавляет вывод информации о версии и разработчике WSH
`//H:CScipt` или `//H:Wscript`	Делает cscript.exe или wscript.exe приложением для запуска сценариев по умолчанию. Если эти параметры не указаны, то по умолчанию подразумевается wscript.exe
`//S`	Сохраняет установки командной строки для текущего пользователя
`//?`	Выводит встроенную подсказку для параметров командной строки
`//E:engine`	Выполняет сценарий с помощью модуля, заданного параметром engine
`//D`	Включает отладчик
`//X`	Выполняет программу в отладчике
`//Job:<JobID>`	Запускает задание с индексом `JobID` из многозадачного WS-файла (структура WS-файлов будет описана ниже)
`//U`	Позволяет использовать при перенаправлении ввода-вывода с консоли кодировку Unicode

Например, команда

```
cscript //Nologo C:\Script\First.js
```

запустит сценарий First.js без информации о версии WSH.

Если сценарий запускается в графическом режиме (с помощью wscript.exe), то свойства сценария можно устанавливать с помощью вкладки **Сценарий** (**Script**) диалогового окна, задающего свойства файла в Windows.

1.4 Собственная объектная модель WSH

`WScript`. Это главный объект WSH, который служит для создания других объектов или связи с ними, содержит сведения о сервере сценариев, а также позволяет вводить данные с клавиатуры и выводить информацию на экран или в окно Windows.

`WshArguments`. Обеспечивает доступ ко всем параметрам командной строки запущенного сценария или ярлыка Windows.

`WshNamed`. Обеспечивает доступ к именным параметрам командной строки запущенного сценария.

`WshUnnamed`. Обеспечивает доступ к безымянным параметрам командной строки запущенного сценария.

`WshShell`. Позволяет запускать независимые процессы, создавать ярлыки, работать с переменными среды, системным реестром и специальными папками Windows.

`WshSpecialFolders`. Обеспечивает доступ к специальным папкам Windows.

`WshShortcut`. Позволяет работать с ярлыками Windows.

`WshUrlShortcut`. Предназначен для работы с ярлыками сетевых ресурсов.

WshEnvironment. Предназначен для просмотра, изменения и удаления переменных среды.
WshNetwork. Используется при работе с локальной сетью: содержит сетевую информацию для локального компьютера, позволяет подключать сетевые диски и принтеры.
WshScriptExec. Позволяет запускать консольные приложения в качестве дочерних процессов, обеспечивает контроль состояния этих приложений и доступ к их стандартным входным и выходным потокам.
WshController. Позволяет запускать сценарии на удаленных машинах.
WshRemote. Позволяет управлять сценарием, запущенным на удаленной машине.
WshRemoteError. Используется для получения информации об ошибке, возникшей в результате выполнения сценария, запущенного на удаленной машине.
FileSystemObject обеспечивающий доступ к файловой системе компьютера.

1.5 Объект WScript

Свойства объекта WScript позволяют получить полный путь к использующемуся серверу сценариев (wscript.exe или cscript.exe), параметры командной строки, с которыми запущен сценарий, режим его работы (интерактивный или пакетный). Кроме этого, с помощью свойств объекта WScript можно выводить информацию в стандартный выходной поток и читать данные из стандартного входного потока. Также WScript предоставляет методы для работы внутри сценария с объектами автоматизации и вывода информации на экран (в текстовом режиме) или в окно Windows.
Отметим, что в сценарии WSH объект WScript можно использовать сразу, без какого-либо предварительного описания или создания, так как его экземпляр создается сервером сценариев автоматически. Для использования же всех остальных объектов нужно использовать либо метод CreateObject.
Свойства объекта WScript представлены в табл. 1.2.

Таблица 1.2. Свойства объекта WScript

Свойство	Описание
Application	Предоставляет интерфейс Idispatch для объекта WScript
Arguments	Содержит указатель на коллекцию WshArguments, содержащую параметры командной строки для исполняемого сценария
FullName	Содержит полный путь к исполняемому файлу сервера сценариев (в Windows XP обычно это C:\WINDOWS\SYSTEM32\CSCRIPT.EXE или C:\WINDOWS\SYSTEM32\WSCRIPT.EXE)
Name	Содержит название объекта Wscript (Windows Script Host)
Path	Содержит путь к каталогу, в котором находится cscript.exe или wscript.exe (в Windows XP обычно это C:\WINDOWS\SYSTEM32)
ScriptFullName	Содержит полный путь к запущенному сценарию
ScriptName	Содержит имя запущенного сценария
StdErr	Позволяет запущенному сценарию записывать сообщения в стандартный поток для ошибок
StdIn	Позволяет запущенному сценарию читать информацию из стандартного входного потока
StdOut	Позволяет запущенному сценарию записывать информацию в стандартный выходной поток
Version	Содержит версию WSH

Опишем более подробно некоторые свойства объекта WScript.

Свойство Arguments

В следующем примере с помощью цикла `For Each … Next` на экран выводятся все параметры командной строки, с которыми был запущен сценарий.

```
'**************************************************************
' Имя: Args.vbs
' Язык: VBScript
' Описание: Работа с аргументами запущенного сценария
'**************************************************************
Option Explicit

Dim i,objArgs,Arg
Set objArgs = WScript.Arguments  ' Создаем объект WshArguments
For Each Arg In objArgs
  WScript.Echo Arg  ' Формируем строки со значениями аргументов
Next

WScript.Echo s    ' Выводим сформированные строки
```

Свойства StdErr, StdIn, StdOut

Доступ к стандартным входным и выходным потокам с помощью свойств `StdIn`, `StdOut` и `StdErr` можно получить только в том случае, если сценарий запускался в консольном режиме с помощью cscript.exe. Если сценарий был запущен с помощью wscript.exe, то при попытке обратиться к этим свойствам возникнет ошибка `"Invalid Handle"`.

Таблица 1.3. Методы для работы с потоками

Метод	Описание
`Read(n)`	Считывает из потока `StdIn` заданное параметром n число символов и возвращает полученную строку
`ReadAll()`	Читает символы из потока `StdIn` до тех пор, пока не встретится символ конца файла ASCII 26 (<Ctrl>+<Z>), и возвращает полученную строку
`ReadLine()`	Возвращает строку, считанную из потока `StdIn`
`Skip(n)`	Пропускает при чтении из потока `StdIn` заданное параметром `n` число символов
`SkipLine()`	Пропускает целую строку при чтении из потока `StdIn`
`Write(string)`	Записывает в поток `StdOut` или `StdErr` строку `string` (без символа конца строки)
`WriteBlankLines(n)`	Записывает в поток `StdOut` или `StdErr` заданное параметром `n` число пустых строк
`WriteLine(string)`	Записывает в поток `StdOut` или `StdErr` строку `string` (вместе с символом конца строки)

Кроме этого, с помощью методов, работающих с входным потоком StdIn, можно организовывать диалог с пользователем, то есть создавать интерактивные сценарии.

```
'**************************************************************
'* Имя: Interact.vbs
'* Язык: VBScript
'* Описание: Ввод/вывод строк в консольном режиме
'**************************************************************
Dim  s
' Выводим строку на экран
WScript.StdOut.Write "Введите число: "
' Считываем строку
```

```
s = WScript.StdIn.ReadLine
' Выводим строку на экран
WScript.StdOut.WriteLine "Вы ввели число " & s
'*************  Конец ******************************************
```

1.6 Методы объекта WScript

Объект `WScript` имеет несколько методов:

Таблица 1.4. Методы объекта WScript

Метод	Описание
`CreateObject(strProgID [, strPrefix])`	Создает объект, заданный параметром `strProgID`
`ConnectObject(strObject, strPrefix)`	Устанавливает соединение с объектом `strObject`, позволяющее писать функции-обработчики его событий (имена этих функций должны начинаться с префикса `strPrefix`)
`DisconnectObject(obj)`	Отсоединяет объект `obj`, связь с которым была предварительно установлена в сценарии
`Echo([Arg1] [, Arg2] [,…])`	Выводит текстовую информацию на консоль или в диалоговое окно
`GetObject(strPathname [, strProgID], [strPrefix])`	Активизирует объект автоматизации, определяемый заданным файлом (параметр `strPathName`) или объект, заданный параметром `strProgID`
`Quit([intErrorCode])`	Прерывает выполнение сценария с заданным параметром `intErrorCode` кодом выхода. Если параметр `intErrorCode` не задан, то объект WScript установит код выхода равным нулю
`Sleep(intTime)`	Приостанавливает выполнения сценария (переводит его в неактивное состояние) на заданное параметром `intTime` число миллисекунд

Метод CreateObject

Строковый параметр `strProgID`, указываемый в методе `CreateObject`, называется программным идентификатором объекта (Programmic Identifier, ProgID).

Если указан необязательный параметр `strPrefix`, то после создания объекта в сценарии можно обрабатывать события, возникающие в этом объекте (естественно, если объект предоставляет интерфейсы для связи с этими событиями). Когда объект сообщает о возникновении определенного события, сервер сценариев вызывает функцию, имя которой состоит из префикса `strPrefix` и имени этого события. Например, если в качестве `strPrefix` указано `"MYOBJ_"`, а объект сообщает о возникновении события `"OnBegin,"` то будет запущена функция `"MYOBJ_OnBegin"`, которая должна быть описана в сценарии.

В следующем примере метод `CreateObject` используется для создания объекта `WshNetwork`:

```
set WshNetwork = WScript.CreateObject("WScript.Network")
```

Метод ConnectObject

Объект, соединение с которым осуществляется с помощью метода `ConnectObject`, должен предоставлять интерфейс к своим событиям.

В следующем примере в переменной `MyObject` создается абстрактный объект `"SomeObject"`, затем из сценария вызывается метод `SomeMetod` этого объекта. После этого устанавливается связь с переменной `MyObject` и задается префикс `"MyEvent"` для процедур обработки события этого объекта. Если в объекте возникнет событие с именем

"Event", то будет вызвана функция MyEvent_Event. Метод DisconnectObject объекта WScript производит отсоединение объекта MyObject.

```
var MyObject = WScript.CreateObject("SomeObject");
MyObject.SomeMethod();
WScript.ConnectObject(MyObject,"MyEvent");

function MyEvent_Event(strName){
  WScript.Echo(strName);
  }
WScript.DisconnectObject(MyObject);
```

Метод Echo

Параметры Arg1, Arg2, … метода Echo задают аргументы для вывода. Если сценарий был запущен с помощью wscript.exe, то метод Echo направляет вывод в диалоговое окно, если же для выполнения сценария применяется cscript.exe, то вывод будет направлен на экран (консоль).

```
'*****************************************************************
'* Имя: EchoExample.vbs
'* Язык: VBScript
'* Описание: Использование метода WScript.Echo
'*****************************************************************
WScript.Echo            'Выводим пустую строку
WScript.Echo 1,2,3      'Выводим числа
WScript.Echo "Привет!"  'Выводим строку
'*************  Конец ***************************************
```

Метод Sleep

В следующем примере сценарий переводится в неактивное состояние на 5 секунд:

```
'*****************************************************************
'* Имя: SleepExample.vbs
'* Язык: VBScript
'* Описание: Использование метода WScript.Sleep
'*****************************************************************
WScript.Echo "Сценарий запущен, отдыхаем…"
WScript.Sleep 5000
WScript.Echo "Выполнение сценария завершено"
'*************  Конец ***************************************
```

1.7 Объект WshShell

С помощью объекта WshShell можно запускать новый процесс, создавать ярлыки, работать с системным реестром, получать доступ к переменным среды и специальным папкам Windows. Создается этот объект следующим образом:

```
var WshShell=WScript.CreateObject("WScript.Shell");
```

Таблица 1.5. Свойства объекта WshShell

Свойство	Описание
CurrentDirectory	Здесь хранится полный путь к текущему каталогу (к каталогу, из которого был запущен сценарий)
Environment	Содержит объект WshEnvironment, который обеспечивает доступ к переменным среды операционной системы для Windows NT/2000/XP или к переменным среды текущего командного окна для Windows 9x
SpecialFolders	Содержит объект WshSpecialFolders для доступа к специальным папкам Windows (рабочий стол, меню Пуск (Start) и т. д.)

Опишем теперь методы, имеющиеся у объекта WshShell.

Таблица 1.6. Методы объекта WshShell

Метод	Описание
`AppActivate(title)`	Активизирует заданное параметром `title` окно приложения. Строка `title` задает название окна (например, `"calc"` или `"notepad"`) или идентификатор процесса (`Process ID, PID`)
`CreateShortcut(strPathname)`	Создает объект `WshShortcut` для связи с ярлыком Windows (расширение lnk) или объект `WshUrlShortcut` для связи с сетевым ярлыком (расширение url). Параметр `strPathname` задает полный путь к создаваемому или изменяемому ярлыку
`Environment(strType)`	Возвращает объект `WshEnvironment`, содержащий переменные среды заданного вида
`Exec(strCommand)`	Создает новый дочерний процесс, который запускает консольное приложение, заданное параметром `strCommand`. В результате возвращается объект `WshScriptExec`, позволяющий контролировать ход выполнения запущенного приложения и обеспечивающий доступ к потокам `StdIn`, `StdOut` и `StdErr` этого приложения
`ExpandEnvironmentStrings(strString)`	Возвращает значение переменной среды текущего командного окна, заданной строкой `strString` (имя переменной должно быть окружено знаками `"%"`)
`LogEvent(intType, strMessage[,strTarget])`	Протоколирует события в журнале Windows NT/2000/XP или в файле WSH.log. Целочисленный параметр `intType` определяет тип сообщения, строка `strMessage` — текст сообщения. Параметр `strTarget` может задаваться только в Windows NT/2000/XP, он определяет название системы, в которой протоколируются события (по умолчанию это локальная система). Метод `LogEvent` возвращает `true`, если событие записано успешно и `false` в противном случае
`Popup(strText,[nSecToWait], [strTitle], [nType])`	Выводит на экран информационное окно с сообщением, заданным параметром `strText`. Параметр `nSecToWait` задает количество секунд, по истечении которых окно будет автоматически закрыто, параметр `strTitle` определяет заголовок окна, параметр `nType` указывает тип кнопок и значка для окна
`RegDelete(strName)`	Удаляет из системного реестра заданный параметр или раздел целиком
`RegRead(strName)`	Возвращает значение параметра реестра или значение по умолчанию для раздела реестра
`RegWrite(strName, anyValue[,strType])`	Записывает в реестр значение заданного параметра или значение по умолчанию для раздела
`Run(strCommand, [intWindowStyle], [bWaitOnReturn])`	Создает новый независимый процесс, который запускает приложение, заданное параметром `strCommand`
`SendKeys(string)`	Посылает одно или несколько нажатий клавиш в активное окно (эффект тот же, как если бы вы нажимали эти клавиши на клавиатуре)

`SpecialFolders(strSpecFolder)`	Возвращает строку, содержащую путь к специальной папке Windows, заданной параметром `strSpecFolder`

Метод CreateShortcut

Этот метод позволяет создать новый или открыть уже существующий ярлык для изменения его свойств. Приведем пример сценария, в котором создаются два ярлыка — на сам выполняемый сценарий (объект `oShellLink`) и на сетевой ресурс (`oUrlLink`).

```
'*************************************************************
'* Имя: MakeShortcuts.vbs
'* Язык: VBScript
'* Описание: Создание ярлыков из сценария
'*************************************************************
Dim WshShell,oShellLink,oUrlLink
' Создаем объект WshShell
Set WshShell=WScript.CreateObject("WScript.Shell")
' Создаем ярлык на файл
Set oShellLink=WshShell.CreateShortcut("Current Script.lnk")
' Устанавливаем путь к файлу
oShellLink.TargetPath=WScript.ScriptFullName
' Сохраняем ярлык
oShellLink.Save

' Создаем ярлык на сетевой ресурс
Set oUrlLink = WshShell.CreateShortcut("Microsoft Web Site.URL")
' Устанавливаем URL
oUrlLink.TargetPath = "http://www.microsoft.com"
' Сохраняем ярлык
oUrlLink.Save
'*************  Конец ***************************************
```

Метод Environment

Параметр `strType` задает вид переменных среды, которые будут записаны в коллекции `WshEnvironment`; возможными значениями этого параметра являются `"System"` (переменные среды операционной системы), `"User"` (переменные среды пользователя), `"Volatile"` (временные переменные) или `"Process"` (переменные среды текущего командного окна).

```
'*************************************************************
'* Имя: ShowEnvir.vbs
'* Язык: VBScript
'* Описание: Получение значений некоторых переменных среды
'*************************************************************
Dim WshShell,WshSysEnv
' Создаем объект WshShell
Set WshShell = WScript.CreateObject("WScript.Shell")
' Создание коллекции WshEnvironment
Set WshSysEnv = WshShell.Environment("SYSTEM")
WScript.Echo WshSysEnv("OS")
WScript.Echo WshShell.Environment.Item("NUMBER_OF_PROCESSORS")
'*************  Конец ***************************************
```

Метод Run

Параметр `intWindowStyle` устанавливает вид окна для запускаемого приложения.

Таблица 1.7. Типы окна (intWindowStyle)

Параметр	Константа Visual Basic	Описание
0	`vbHide`	Прячет текущее окно и активизирует другое окно (показывает его и передает ему фокус)
1	`vbNormalFocus`	Активизирует и отображает окно. Если окно было минимизировано или максимизировано, система восстановит его первоначальное положение и размер. Этот

		флаг должен указываться сценарием во время первого отображения окна
2	vbMinimizedFocus	Активизирует окно и отображает его в минимизированном (свернутом) виде
3	vbMaximizedFocus	Активизирует окно и отображает его в максимизированном (развернутом) виде
4	vbNormalNoFocus	Отображает окно в том виде, в котором оно находилось последний раз. Активное окно при этом остается активным
5		Активизирует окно и отображает его в текущем состоянии
6	vbMinimizedNoFocus	Минимизирует заданное окно и активизирует следующее (в Z-порядке) окно
7		Отображает окно в свернутом виде. Активное окно при этом остается активным
8		Отображает окно в его текущем состоянии. Активное окно при этом остается активным
9		Активизирует и отображает окно. Если окно было минимизировано или максимизировано, система восстановит его первоначальное положение и размер. Этот флаг должен указываться, если производится восстановление свернутого окна (его нельзя использовать в методе Run)
10		Устанавливает режим отображения, опирающийся на режим программы, которая запускает приложение

Необязательный параметр bWaitOnReturn является логической переменной, дающей указание ожидать завершения запущенного процесса. Если этот параметр не указан или установлен в false, то после запуска из сценария нового процесса управление сразу же возвращается обратно в сценарий (не дожидаясь завершения запущенного процесса). Если же bWaitOnReturn установлен в true, то сценарий возобновит работу только после завершения вызванного процесса. При этом если параметр bWaitOnReturn равен true, то метод Run возвращает код выхода вызванного приложения. Если же bWaitOnReturn равен false или не задан, то метод Run всегда возвращает ноль.

```
'*************************************************************
'* Имя: RetCode.vbs
'* Язык: VBScript
'* Описание: Вывод кода выхода запущенного приложения
'*************************************************************
' Создаем объект WshShell
Set WshShell = WScript.CreateObject("WScript.Shell")
' Запускаем Блокнот и ожидаем завершения его работы
Return = WshShell.Run("notepad " + WScript.ScriptFullName, 1, true)
' Печатаем код возврата
WScript.Echo "Код возврата:", Return
'************  Конец ********************************************
```

Метод SendKeys

Каждая клавиша задается одним или несколькими символами. Например, Для того чтобы задать нажатие друг за другом букв А, Б и В нужно указать в качестве параметра для SendKeys строку "АБВ": string="АБВ".

Несколько символов имеют в методе SendKeys специальное значение: +, ^, %, ~, (,). Для того чтобы задать один из этих символов, их нужно заключить в фигурные скобки ({}). Например, для задания знака плюс используется {+}. Квадратные скобки ([]) хотя и не имеют в методе SendKeys специального смысла, их также нужно заключать в фигурные

скобки. Кроме этого, для задания самих фигурных скобок следует использовать следующие конструкции: `{{} (левая скобка) и {}} (правая скобка)`.

Таблица 1.8. Коды специальных клавишей для SendKeys

Названия клавиши	Код	Названия клавиши	Код
<Backspace>	{BACKSPACE}, {BS} или {BKSP}	<→>	{RIGHT}
<Break>	{BREAK}	<F1>	{F1}
<Caps Lock>	{CAPSLOCK}	<F2>	{F2}
<Del> или <Delete>	{DELETE} или {DEL}	<F3>	{F3}
<End>	{END}	<F4>	{F4}
<Enter>	{ENTER} или ~	<F5>	{F5}
<Esc>	{ESC}	<F6>	{F6}
<Home>	{HELP}	<F7>	{F7}
<Ins> или <Insert>	{INSERT} или {INS}	<F8>	{F8}
<Num Lock>	{NUMLOCK}	<F9>	{F9}
<Page Down>	{PGDN}	<F10>	{F10}
<Page Up>	{PGUP}	<F11>	{F11}
<Print Screen>	{PRTSC}	<F12>	{F12}
<Scroll Lock>	{SCROLLLOCK}	<F13>	{F13}
<Tab>	{TAB}	<F14>	{F14}
<↑>	{UP}	<F15>	{F15}
<←>	{LEFT}	<F16>	{F16}
<↓>	{DOWN}		

Таблица 5.8. Коды клавиш <Shift>, <Ctrl> и <Alt>

Клавиша	Код
<Shift>	+
<Ctrl>	^
<Alt>	%

Для того чтобы задать комбинацию клавиш, которую нужно набирать, удерживая нажатыми клавиши <Shift>, <Ctrl> или <Alt>, нужно заключить коды этих клавиш в скобки. Например, если требуется сымитировать нажатие клавиш G и S при нажатой клавише <Shift>, следует использовать последовательность `"+(GS)"`. Для того же, чтобы задать одновременное нажатие клавиш <Shift>+<G>, а затем <S> (уже без <Shift>), используется `"+GS"`.

В методе `SendKeys` можно задать несколько нажатий подряд одной и той же клавиши. Для этого необходимо в фигурных скобках указать код нужной клавиши, а через пробел — число нажатий. Например, `{LEFT 42}` означает нажатие клавиши <?> 42 раза подряд; `{h 10}` означает нажатие клавиши h 10 раз подряд.

```
'*************************************************************
'* Имя: RunCalc.vbs
'* Язык: VBScript
'* Описание: Активизация приложения с помощью имени окна
'*************************************************************
Dim WshShell
' Создаем объект WshShell
Set WshShell=WScript.CreateObject("WScript.Shell")
```

```
' Запускаем Калькулятор
WshShell.Run "calc"
' Приостанавливаем сценарий на 0,1 секунды
WScript.Sleep 100
' Активизируем Калькулятор
WshShell.AppActivate "Calculator"
' Приостановка сценария на 0,1 секунды
WScript.Sleep(100)
' Посылаем нажатия клавиш в Калькулятор
WshShell.SendKeys "1{+}"
WScript.Sleep 500
WshShell.SendKeys "2"
WScript.Sleep 500
WshShell.SendKeys "~"
WScript.Sleep 2500
```

1.9 Объект WshArguments

Объект `WshArguments` содержит коллекцию всех параметров командной строки запущенного сценария или ярлыка Windows. Этот объект можно создать только с помощью свойства `Arguments` объектов `WScript` и `WshShortcut`.

С помощью объекта `WshArguments` можно также выделять и отдельно обрабатывать аргументы сценария, у которых имеются имена (например, `/Name:Andrey`) и безымянные аргументы. Ясно, что использование именных параметров более удобно, так как в этом случае нет необходимости запоминать, в каком порядке должны быть записаны параметры при запуске того или иного сценария.

Для доступа к именным и безымянным аргументам используются соответственно два специальных свойства объекта `WshArguments`: `Named` и `Unnamed`. Свойство `Named` содержит ссылку на коллекцию `WshNamed`, свойство `Unnamed` — на коллекцию `WshUnnamed`.

Таким образом, обрабатывать параметры командной строки запущенного сценария можно тремя способами:

- просматривать полный набор всех параметров (как именных, так и безымянных) с помощью коллекции `WshArguments`;
- выделить только те параметры, у которых есть имена (именные параметры) с помощью коллекции `WshNamed`;
- выделить только те параметры, у которых нет имен (безымянные параметры) с помощью коллекции `WshUnnamed`.

```
'**********************************************************************
' Имя: Args.vbs
' Язык: VBScript
' Описание: Работа с аргументами запущенного сценария
'**********************************************************************
Option Explicit

Dim i,Arg,objArgs,s,objNamedArgs,objUnnamedArgs  ' Объявляем переменные

Set objArgs = WScript.Arguments  ' Создаем объект WshArguments
' Определяем общее количество аргументов
s="Всего аргументов: " & objArgs.Count() & vbCrLf
For Each Arg In objArgs
  s=s & Arg & vbCrLf  ' Формируем строки со значениями аргументов
Next

Set objUnnamedArgs=objArgs.Unnamed  ' Создаем объект WshUnnamed
' Определяем количество безымянных аргументов
s=s & vbCrLf & "Безымянных аргументов: " & objUnnamedArgs.length & vbCrLf
For Each Arg In objUnnamedArgs
```

```
  ' Формируем строки со значениями безымянных аргументов
  s=s & Arg & vbCrLf
Next

Set objNamedArgs=objArgs.Named  ' Создаем объект WshNamed
' Определяем количество именных аргументов
s=s & vbCrLf & "Именных аргументов: " & objNamedArgs.Length & vbCrLf
' Проверяем, существует ли аргумент /Имя:
If objNamedArgs.Exists("Имя") Then
  s=s & objNamedArgs("Имя") & vbCrLf
End If
' Проверяем, существует ли аргумент /Comp:
If objNamedArgs.Exists("Comp") Then
  s=s & objNamedArgs("Comp") & vbCrLf
End If

WScript.Echo s    ' Выводим сформированные строки
'*************  Конец ********************************************
```

1.10 Объект WshEnvironment

Объект `WshEnvironment` позволяет получить доступ к коллекции, содержащей переменные среды заданного типа (переменные среды операционной системы, переменные среды пользователя или переменные среды текущего командного окна). Этот объект можно создать с помощью свойства `Environment` объекта `WshShell` или одноименного его метода:

```
Set WshShell=WScript.CreateObject("WScript.Shell")
Set WshSysEnv=WshShell.Environment
Set WshUserEnv=WshShell.Environment("User")
```

Объект `WshEnvironment` имеет свойство `Length`, в котором хранится число элементов в коллекции (количество переменных среды), и методы `Count` и `Item`. Для того чтобы получить значение определенной переменной среды, в качестве аргумента метода `Item` указывается имя этой переменной в двойных кавычках.

```
'*****************************************************************
' Имя: Environment.vbs
' Язык: VBScript
' Описание: Работа с переменными среды
'*****************************************************************
Dim WshShell, WshSysEnv
Set WshShell=WScript.CreateObject("WScript.Shell")
Set WshSysEnv=WshShell.Environment
WScript.Echo "Системный путь:",WshSysEnv.Item("PATH")
```

Можно также просто указать имя переменной в круглых скобках после имени объекта:

```
WScript.Echo "Системный путь:",WshSysEnv("PATH")
```

Кроме этого у объекта `WshEnvironment` имеется метод `Remove(strName)`, который удаляет заданную переменную среды.

1.11 Объект WshSpecialFolders

При установке Windows всегда автоматически создаются несколько специальных папок (например, папка для рабочего стола (Desktop) или папка для меню Пуск (Start)), путь к которым впоследствии может быть тем или иным способом изменен. Объект `WshSpecialFolders` обеспечивает доступ к коллекции, содержащей пути к специальным папкам Windows; задание путей к таким папкам может понадобиться, например, для создания непосредственно из сценария ярлыков на рабочем столе. В Windows XP поддерживаются следующие имена специальных папок:

1. Desktop;

2. Favorites;
3. Fonts;
4. MyDocuments;
5. NetHood;
6. PrintHood;
7. Programs;
8. Recent;
9. SendTo;
10. StartMenu;
11. Startup;
12. Templates;
13. AllUsersDesktop;
14. AllUsersStartMenu;
15. AllUsersPrograms;
16. AllUsersStartup.

Объект `WshSpecialFolders` создается с помощью свойства `SpecialFolders` объекта `WshShell`:

```
var WshShell=WScript.CreateObject("WScript.Shell"),
    WshSpecFold=WshShell.SpecialFolders;
```

Сценарий, формирующий список всех имеющихся в системе специальных папок.

```
'*****************************************************************
' Имя: SpecFold1.vbs
' Язык: VBScript
' Описание: Вывод названий всех специальных папок Windows
'*****************************************************************
Option Explicit

Dim WshShell, WshFldrs, SpecFldr, s  ' Объявляем переменные
' Создаем объект WshShell
Set WshShell = WScript.CreateObject("Wscript.Shell")
' Создаем объект WshSpecialFolders
Set WshFldrs = WshShell.SpecialFolders
s="Список всех специальных папок:" & vbCrLf & vbCrLf
' Перебираем все элементы коллекции WshFldrs
For Each SpecFldr In WshFldrs
  ' Формируем строки с путями к специальным папкам
  s=s & SpecFldr & vbCrLf
Next
WScript.Echo s
```

Объект `WshSpecialFolders` также позволяет получить путь к конкретно заданной специальной папке.

```
'*****************************************************************
' Имя: SpecFold2.vbs
' Язык: VBScript
' Описание: Вывод названий заданных специальных папок Windows
'*****************************************************************
Option Explicit

Dim WshShell, WshFldrs, s   ' Объявляем переменные
' Создаем объект WshShell
Set WshShell = WScript.CreateObject("Wscript.Shell")
' Создаем объект WshSpecialFolders
Set WshFldrs = WshShell.SpecialFolders
' Формируем строки с путями к конкретным специальным папкам
s="Некоторые специальные папки:" & vbCrLf & vbCrLf
s=s+"Desktop:"+WshFldrs("Desktop") & vbCrLf
s=s+"Favorites:"+WshFldrs("Favorites") & vbCrLf
s=s+"Programs:"+WshFldrs("Programs")
WScript.Echo s   ' Выводим сформированные строки на экран
```

```
'*************  Конец ******************************************/
```

1.12 Сценарии WSH для доступа к файловой системе. Объектная модель FileSystemObject

Для работы с файловой системой из сценариев WSH предназначены восемь объектов, главным из которых является `FileSystemObject`. С помощью методов объекта `FileSystemObject` можно выполнять следующие основные действия:

1. копировать или перемещать файлы и каталоги;
2. удалять файлы и каталоги;
3. создавать каталоги;
4. создавать или открывать текстовые файлы;
5. создавать объекты `Drive`, `Folder` и `File` для доступа к конкретному диску, каталогу или файлу соответственно.

С помощью свойств объектов `Drive`, `Folder` и `File` можно получить детальную информацию о тех элементах файловой системы, с которыми они ассоциированы. Объекты `Folder` и `File` также предоставляют методы для манипулирования файлами и каталогами (создание, удаление, копирование, перемещение); эти методы в основном копируют соответствующие методы объекта `FileSystemObject`.

Кроме этого имеются три объекта-коллекции: `Drives`, `Folders` и `Files`. Коллекция `Drives` содержит объекты `Drive` для всех имеющихся в системе дисков, `Folders` — объекты `Folder` для всех подкаталогов заданного каталога, `Files` — объекты `File` для всех файлов, находящихся внутри определенного каталога.

Наконец, из сценария можно читать информацию из текстовых файлов и записывать в них данные. Методы для этого предоставляет объект `TextStream`.

Таблица 1.9. Выполнение основных файловых операций

Операция	Используемые объекты, свойства и методы
Получение сведений об определенном диске (тип файловой системы, метка тома, общий объем и количество свободного места и т.д.)	Свойства объекта `Drive`. Сам объект `Drive` создается с помощью метода `GetDrive` объекта `FileSystemObject`
Получение сведений о заданном каталоге или файле (дата создания или последнего доступа, размер, атрибуты и т.д.)	Свойства объектов `Folder` и `File`. Сами эти объекты создаются с помощью методов `GetFolder` и `GetFile` объекта `FileSystemObject`
Проверка существования определенного диска, каталога или файла	Методы `DriveExists`, `FolderExists` и `FileExists` объекта `FileSystemObject`
Копирование файлов и каталогов	Методы `CopyFile` и `CopyFolder` объекта `FileSystemObject`, а также методы `File.Copy` и `Folder.Copy`
Перемещение файлов и каталогов	Методы `MoveFile` и `MoveFolder` объекта `FileSystemObject`, или методы `File.Move` и `Folder.Move`
Удаление файлов и каталогов	Методы `DeleteFile` и `DeleteFolder` объекта `FileSystemObject`, или методы `File.Delete` и `Folder.Delete`
Создание каталога	Методы `FileSystemObject.CreateFolder` или `Folders.Add`
Создание текстового файла	Методы `FileSystemObject.CreateTextFile` или

	`Folder.CreateTextFile`
Получение списка всех доступных дисков	Коллекция `Drives`, содержащаяся в свойстве `FileSystemObject.Drives`
Получение списка всех подкаталогов заданного каталога	Коллекция `Folders`, содержащаяся в свойстве `Folder.SubFolders`
Получение списка всех файлов заданного каталога	Коллекция `Files`, содержащаяся в свойстве `Folder.Files`
Открытие текстового файла для чтения, записи или добавления	Методы `FileSystemObject.CreateTextFile` или `File.OpenAsTextStream`
Чтение информации из заданного текстового файла или запись ее в него	Методы объекта `TextStream`

Получение сведений о диске

В сценарий DriveInfo.vbs, который выводит на экран некоторые свойства диска C.

```
'*****************************************************************
' Имя: DriveInfo.vbs
' Язык: VBScript
' Описание: Вывод на экран свойств диска C
'*****************************************************************
'Объявляем переменные
Dim FSO,D,TotalSize,FreeSpace,s
'Создаем объект FileSystemObject
Set FSO = WScript.CreateObject("Scripting.FileSystemObject")
'Создаем объект Drive для диска C
Set D = FSO.GetDrive("C:")
s = "Информация о диске C:" & VbCrLf
'Получаем серийный номер диска
s = s &"Серийный номер: " & D.SerialNumber & VbCrLf
'Получаем метку тома диска
s = s & "Метка тома: " & D.VolumeName & VbCrLf
'Вычисляем общий объем диска в килобайтах
TotalSize = D.TotalSize/1024
s = s & "Объем: " & TotalSize & " Kb" & VbCrLf
'Вычисляем объем свободного пространства диска в килобайтах
FreeSpace = D.FreeSpace/1024
s = s & "Свободно: " & FreeSpace & " Kb" & VbCrLf
'Выводим свойства диска на экран
WScript.Echo s
'*************  Конец ***********************************************
```

Получение сведений о каталоге

В сценарии FolderInfo.vbs на экран выводятся свойства каталога, из которого был запущен сценарий.

```
'*****************************************************************
' Имя: FolderInfo.vbs
' Язык: VBScript
' Описание: Вывод на экран даты создания текущего каталога
'*****************************************************************
Dim FSO,WshShell,FoldSize,s  'Объявляем переменные

'Создаем объект FileSystemObject
Set FSO = WScript.CreateObject("Scripting.FileSystemObject")
'Создаем объект WshShell
Set WshShell = WScript.CreateObject("WScript.Shell")

'Определяем каталог, из которого был запущен сценарий
'(текущий каталог)
Set Folder = FSO.GetFolder(WshShell.CurrentDirectory)
'Получаем имя текущего каталога
```

```
s = "Текущий каталог: " & Folder.Name & VbCrLf
'Получаем дату создания текущего каталога
s = s & "Дата создания: " & Folder.DateCreated & VbCrLf
'Вычисляем размер текущего каталога в килобайтах
FoldSize=Folder.Size/1024
s = s & "Объем: " & FoldSize & " Kb" & VbCrLf
'Выводим информацию на экран
WScript.Echo s
'*************  Конец ******************************************
```

Получение сведений о файле

В сценарий FileInfo.vbs, в котором на экран выводятся некоторые свойства файла C:\boot.ini.

```
'*********************************************************************
' Имя: FileInfo.vbs
' Язык: VBScript
' Описание: Вывод на экран некоторых свойств файла
'*********************************************************************
Dim FSO,F,s   'Объявляем переменные

'Создаем объект FileSystemObject
Set FSO = WScript.CreateObject("Scripting.FileSystemObject")
'Создаем объект File
Set F = FSO.GetFile("C:\boot.ini")

'Получаем имя файла
s = "Файл: " & F.Name & VbCrLf
'Получаем дату создания файла
s = s & "Дата создания: " & F.DateCreated & VbCrLf
'Получаем тип файла
s = s & "Тип: " & F.Type & VbCrLf
'Выводим информацию на экран
WScript.Echo s
'*************  Конец ******************************************
```

Проверка существования диска, каталога или файла

В сценарий IsExistsFile.vbs, в котором проверяется наличие на диске C файла boot.ini.

```
'*********************************************************************
' Имя: IsExistsFile.vbs
' Язык: VBScript
' Описание: Проверка существования файла
'*********************************************************************
Dim FSO,FileName 'Объявляем переменные

'Создаем объект FileSystemObject
Set FSO = WScript.CreateObject("Scripting.FileSystemObject")

FileName = "C:\boot.ini"
if  FSO.FileExists(FileName)  Then
  'Выводим информацию на экран
  WScript.Echo "Файл " & FileName & " существует"
else
  'Выводим информацию на экран
  WScript.Echo "Файл " & FileName & " не существует"
end if
'*************  Конец ******************************************
```

1.13 Получение списка всех имеющихся дисков

Каждому из дисков компьютера (включая подключенные сетевые диски и дисководы со сменными носителями) соответствует элемент коллекции `Drives` (объект `Drive`). Таким образом, для построения списка дисков компьютера нужно в цикле перебрать все элементы коллекции `Drives`.

В сценарий ListDrives.vbs, в котором на экран выводятся сведения обо всех доступных дисках.

```
'*********************************************************************
' Имя: ListDrives.vbs
' Язык: VBScript
' Описание: Получение списка всех имеющихся дисков
'*********************************************************************
'Объявляем переменные
Dim FSO,s,ss,Drives,D

'Создаем объект FileSystemObject
Set FSO = WScript.CreateObject("Scripting.FileSystemObject")
'Создаем коллекцию дисков, имеющихся в системе
Set Drives = FSO.Drives
s = ""

'Перебираем все диски в коллекции
For Each D In Drives
  'Получаем букву диска
  s = s & D.DriveLetter
  s = s & " - "
  if (D.DriveType = 3) then  'Проверяем, не является ли диск сетевым
    'Получаем имя сетевого ресурса
    ss = D.ShareName
  else
    'Диск является локальным
    if (D.IsReady) then  'Проверяем готовность диска
      'Если диск готов, то получаем метку тома для диска
      ss = D.VolumeName
    else
      ss = "Устройство не готово"
    end if
  s = s & ss & VbCrLf
  end if
Next

'Выводим полученные строки на экран
WScript.Echo s
'*************  Конец *********************************************
```

Получение списка всех подкаталогов заданного каталога

Для построения списка всех подкаталогов определенного каталога можно воспользоваться коллекцией `Folders`, которая хранится в свойстве `SubFolders` соответствующего объекта `Folder` и содержит объекты `Folder` для всех подкаталогов.

В сценарий ListSubFold.vbs, в котором на экран выводятся названия всех подкаталогов каталога C:\Program Files.

```
'*********************************************************************
' Имя: ListSubFold.vbs
' Язык: VBScript
' Описание: Получение списка всех подкаталогов заданного каталога
'*********************************************************************
'Объявляем переменные
Dim FSO,F,SFold,SubFolders,Folder,s

'Создаем объект FileSystemObject
Set FSO = WScript.CreateObject("Scripting.FileSystemObject")
'Путь к каталогу
SFold = "C:\Program Files"
s = "Каталог " & SFold & VbCrLf
s = s & "Подкаталоги:" & VbCrLf
'Создаем объект Folder для каталога C:\Program Files
Set F=FSO.GetFolder(SFold)

'Создаем коллекцию подкаталогов каталога C:\Program Files
Set SubFolders = F.SubFolders
```

```
'Цикл по всем подкаталогам
For Each Folder In SubFolders
  'Добавляем строку с именем подкаталога
   s = s & Folder & VbCrLf
Next

'Выводим полученные строки на экран
WScript.Echo s
'*************  Конец ******************************************/
```

Получение списка всех файлов заданного каталога

В свойстве `Files` объекта `Folder`, соответствующего определенному каталогу, хранится коллекция находящихся в этом каталоге файлов (объектов `File`). В сценарии ListFiles.vbs, выводящий на экран названия всех файлов, которые содержатся в специальной папке Мои документы.

```
'*********************************************************************
' Имя: ListFiles.vbs
' Язык: VBScript
' Описание: Получение списка всех файлов заданного каталога
'*********************************************************************
'Объявляем переменные
Dim FSO,F,File,Files,WshShell,PathList,s

'Создаем объект FileSystemObject
Set FSO = WScript.CreateObject("Scripting.FileSystemObject")
'Создаем объект WshShell
Set WshShell = WScript.CreateObject("Wscript.Shell")
'Создаем объект WshSpecialFolders
Set WshFldrs = WshShell.SpecialFolders
'Определяем путь к папке Мои документы
PathList = WshFldrs.item("MyDocuments") & "\"
'Создаем объект Folder для папки Мои документы
Set F = FSO.GetFolder(PathList)
'Создаем коллекцию файлов каталога Мои документы
Set Files = F.Files
s = "Файлы из каталога " & PathList & VbCrLf
'Цикл по всем файлам
For Each File In Files
  'Добавляем строку с именем файла
  s = s & File.Name & VbCrLf
Next

'Выводим полученные строки на экран
WScript.Echo s
'*************  Конец ******************************************
```

Создание каталога

Создать новый каталог на диске можно либо с помощью метода `CreateFolder` объекта `FileSystemObject`, либо с помощью метода `Add` коллекции `Folders`. Оба эти метода используются в сценарии MakeFolder.vbs для создания в каталоге C:\Мои документы подкаталогов Новая папка и Еще одна новая папка.

```
'*********************************************************************
' Имя: MakeFolder.vbs
' Язык: VBScript
' Описание: Создание нового каталога
'*********************************************************************
'Объявляем переменные
Dim FSO, F, SubFolders

'Создаем объект FileSystemObject
Set FSO = WScript.CreateObject("Scripting.FileSystemObject")
'Создаем каталог C:\Program Files\Новая папка
```

```
FSO.CreateFolder("C:\Program Files\Новая папка")
0'Создаем объект Folder для каталога C:\Program Files
Set F = FSO.GetFolder("C:\Program Files")
'Создаем коллекцию подкаталогов каталога C:\Program Files
Set SubFolders = F.SubFolders
'Создаем каталог C:\Program Files\Еще одна новая папка
SubFolders.Add "Еще одна новая папка"
'*************  Конец ******************************************
```

Создание текстового файла

Для создания текстового файла используется метод `CreateTextFile` объекта `FileSystemObject`, который имеет один обязательный текстовый параметр (путь к создаваемому файлу) и два необязательных логических параметра (`Overwrite` и `Unicode`).

Параметр `Overwrite` имеет значение в том случае, когда создаваемый файл уже существует. Если `Overwrite` равно `True`, то такой файл перепишется (старое содержимое будет утеряно), если же в качестве `Overwrite` указано `False`, то файл переписываться не будет. Если этот параметр вообще не указан, то существующий файл также не будет переписан.

```
'*********************************************************************
' Имя: CreateTempFile.vbs
' Язык: VBScript
' Описание: Создание временного файла со случайным именем
'*********************************************************************
Dim FSO,FileName,F,s  'Объявляем переменные
'Создаем объект FileSystemObject
Set FSO = WScript.CreateObject("Scripting.FileSystemObject")
'Генерируем случайное имя файла
FileName = FSO.GetTempName
'Создаем файл с именем FileName
Set F = FSO.CreateTextFile(FileName, true)
'Закрываем файл
F.Close
'Сообщаем о создании файла
WScript.Echo "Был создан файл " & FileName
'*************  Конец ******************************************
```

Операции чтения и записи строк для текстового файла

Последовательный (строка за строкой) доступ к текстовому файлу обеспечивает объект `TextStream`. Методы этого объекта позволяют читать информацию из файла и записывать ее в него. Создается объект `TextStream` при открытии или создании текстового файла с помощью следующих методов:

1. CreateTextFile объектов FileSystemObject и Folder;
2. OpenTextFile объекта FileSystemObject;
3. OpenAsTextStream объекта File.

Таблица 1.10. Параметр Iomode

Константа	Значение	Описание
`ForReading`	1	Файл открывается только для чтения, записывать информацию в него нельзя
`ForWriting`	2	Файл открывается для записи. Если файл с таким именем уже существовал, то при новой записи его содержимое теряется
`ForAppending`	8	Файл открывается для добавления. Если файл уже существовал, то информация будет дописываться в конец этого файла

Таблица 1.11. Параметр Format

Константа	Значение	Описание
TristateUseDefault	-2	Файл открывается в формате, используемом системой по умолчанию
TristateTrue	-1	Файл открывается в формате Unicode
TristateFalse	0	Файл открывается в формате ASCII

```
'*****************************************************************
' Имя: TextFile.vbs
' Язык: VBScript
' Описание: Запись строк в текстовый файл и чтение из него
'*****************************************************************
Dim FSO,F,TextStream,s  'Объявляем переменные
' Инициализируем константы
Const ForReading = 1, ForWriting = 2, TristateUseDefault = -2

' Создаем объект FileSystemObject
Set FSO=WScript.CreateObject("Scripting.FileSystemObject")
' Создаем в текущем каталоге файл test1.txt
FSO.CreateTextFile "test1.txt"
' Создаем объект File для файла test1.txt
set F=FSO.GetFile("test1.txt")
' Создаем объект TextStream (файл открывается для записи)
Set TextStream=F.OpenAsTextStream(ForWriting, TristateUseDefault)
' Записываем в файл строку
TextStream.WriteLine "Это первая строка"
' Закрываем файл
TextStream.Close
' Открываем файл для чтения
Set TextStream=F.OpenAsTextStream(ForReading, TristateUseDefault)
' Считываем строку из файла
s=TextStream.ReadLine
' Закрываем файл
TextStream.Close
' Отображаем строку на экране
WScript.Echo "Первая строка из файла test1.txt:" & vbCrLf & vbCrLf & s
'*************  Конец **************************************
```

Копирование и перемещение файлов и каталогов

Для копирования файлов/каталогов можно применять метод `CopyFile/CopyFolder` объекта `FileSystemObject` или метод Copy соответствующего этому файлу/каталогу объекта `File/Folder`. Перемещаются файлы/каталоги с помощью методов `MoveFile/MoveFolder` объекта `FileSystemObject` или метода `Move` соответствующего этому файлу/каталогу объекта `File/Folder`.

В сценарий CopyFile.vbs, иллюстрирующий использование метода Copy. В этом сценариях на диске C создается файл TestFile.txt, который затем копируется на рабочий стол.

```
'*****************************************************************
' Имя: CopyFile.vbs
' Язык: VBScript
' Описание: Создание и копирование файла
'*****************************************************************
'Объявляем переменные
Dim FSO,F,WshShell,WshFldrs,PathCopy

'Создаем объект FileSystemObject
Set FSO = WScript.CreateObject("Scripting.FileSystemObject")
'Создаем файл
Set F = FSO.CreateTextFile("C:\TestFile.txt", true)
'Записываем в файл строку
F.WriteLine "Тестовый файл"
'Закрываем файл
```

```
F.Close

'Создаем объект WshShell
Set WshShell = WScript.CreateObject("Wscript.Shell")
'Создаем объект WshSpecialFolders
Set WshFldrs = WshShell.SpecialFolders
'Определяем путь к рабочему столу
PathCopy = WshFldrs.item("Desktop")+"\"
'Создаем объект File для файла C:\TestFile.txt
Set F = FSO.GetFile("C:\TestFile.txt")
'Копируем файл на рабочий стол
F.Copy PathCopy
'*************  Конец ******************************************
```

Удаление файлов и каталогов

Для удаления файлов/каталогов можно применять метод `DeleteFile/DeleteFolder` объекта `FileSystemObject` или метод `Delete` соответствующего этому файлу/каталогу объекта `File/Folder`. Отметим, что при удалении каталога неважно, является ли он пустым или нет — удаление будет произведено в любом случае. Если же заданный для удаления файл/каталог не будет найден, то возникнет ошибка.

В сценарий DeleteFile.vbs, в котором производится удаление предварительно созданного файла C:\TestFile.txt.

```
'*********************************************************************
' Имя: DeleteFile.vbs
' Язык: VBScript
' Описание: Создание и удаление файла
'*********************************************************************
'Объявляем переменные
Dim FSO,F,FileName

'Создаем объект FileSystemObject
Set FSO = WScript.CreateObject("Scripting.FileSystemObject")
'Задаем имя файла
FileName="C:\TestFile.txt"
'Создаем файл
Set F = FSO.CreateTextFile(FileName, true)
'Записываем в файл строку
F.WriteLine "Тестовый файл"
'Закрываем файл
F.Close
WScript.Echo "Файл создан"
FSO.DeleteFile FileName
WScript.Echo "Файл удален"
'*************  Конец ******************************************
```

2. Практическое задание

Создать сценарий реализующий в консольном режиме диалог с пользователем в виде меню. Сценарий должен выполняться циклически пока не выбран пункт «Выход». Первый пункт меню должен выводить информацию о создателе (ФИО, группа) и краткое описание выполняемых действий, остальные пункты реализуют действия указанные в таблице в соответствии с вариантом. Все параметры задаются в результате диалога с пользователем. При выполнении задания А, допускается использование командных файлов рассмотренных в первой лабораторной работе. Сценарий запускается в **консольной** версии WSH.
Отчет должен содержать краткие теоретические сведения о **использованных** объектах, методах и свойствах.

Вариант	Задание
1	А)Создание паки в указанном месте. Б)Открытие блокнота и сохранение в нем заданного сообщения.
2	А)Копирование файлов с указанного места в заданное. Б)Создание ярлыка для вызова заданной программы и помещение его на рабочий стол.
3	А)Удаление файлов заданного расширения в заданной папке. Б)Создание ярлыка для просмотра содержимого заданной папки и помещение его на рабочий стол.
4	А)Удаление содержимого заданной папки. Б)Создание ссылки на заданный сетевой ресурс и помещение его на рабочий стол.
5	А)Создание файла со списком папок в указанном месте. Б)Открытие блокнота и сохранение в нем имени пользователя.
6	А)Копирование файлов заданного расширения с указанного места в папку «*BackUp*», на указанном диске. Б)Вывод на экран пути к заданной специальной папке.
7	А)Перенос файлов заданного расширения с указанного места в папку «*BackUp*», на заданном диске. Б)Сохранение в блокноте параметра реестра: *HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\BuildLab*
8	А)Создание файла со списком системных файлов в указанном месте. Б)Сохранение в блокноте списка дисков с их размером.
9	А)Создание файла со списком скрытых файлов в указанном месте. Б)Сохранение в блокноте списка дисков с значением свободного места .
10	А)Перенос файлов с указанного места в заданное. Б)Сохранение в блокноте даты создания и размера заданной папки.
11	А)Переименование заданной папки. Б)Сохранение в блокноте даты создания и размера заданного файла.
12	А)Архивирование заданной папки. Б)Сохранение в текстовом файле списка папок в заданном каталоге.
13	А)Архивирование файлов заданного расширения в заданной папке. Б)Сохранение в текстовом файле списка файлов в заданной папке.
14	А)Создание файла со списком файлов заданного расширения на заданном диске. Б)Сохранение в текстовом файле списка специальных папок.
15	А)Копирование всех фалов с заданным расширением с заданного диска в указанную папку. Б)Сохранение в текстовом файле списка текстовых файлов в папке «*мои документы*» с датой их создания.

Пример

Создать сценарий реализующий в консольном режиме диалог с пользователем в виде меню реализующий: создание файла со списком файлов с расширением .ini", создание ярлыка на заданный сетевой ресурс.

```
'* Имя: Interact.vbs
'* Язык: VBScript
'* Описание: пример лаболаторной работы
'*********************************************************************
Dim  s
' Выводим строку на экран
do
 WScript.StdOut.WriteLine "МЕНЮ:"
 WScript.StdOut.WriteLine "--------------------------"
 WScript.StdOut.WriteLine "1. Информация о авторе"
 WScript.StdOut.WriteLine "2. Создание файла со списком файлов с расширением
.ini"
 WScript.StdOut.WriteLine "3. Создание ярлыка на заданный сетевой ресурс"
 WScript.StdOut.WriteLine "4. Выход"
 WScript.StdOut.Write "Выберите пункт меню:"
 ' Считываем строку
 s = WScript.StdIn.ReadLine
 ' Создаем объект WshShell
 Set WshShell = WScript.CreateObject("WScript.Shell")

 if (s="1") Then
      WScript.StdOut.WriteLine "ФИО, группа"

 elseif(s="2") Then
      WScript.StdOut.Write "Введите имя файла:"
  f = WScript.StdIn.ReadLine

      ' Запускаем командный файл и ожидаем окончания ее работы
  Code=WshShell.Run("%COMSPEC% /c 02.cmd >"+f ,0,true)

      elseif(s="3") Then
      WScript.StdOut.Write "Введите имя ярлыка:"
  f = WScript.StdIn.ReadLine
  ' Создаем ярлык на сетевой ресурс
  Set oUrlLink = WshShell.CreateShortcut(f+".URL")
      WScript.StdOut.Write "Введите имя ресурса:"
  f = WScript.StdIn.ReadLine
  ' Устанавливаем URL
  oUrlLink.TargetPath = f
  ' Сохраняем ярлык
  oUrlLink.Save
End if

loop until (s="4")

'*************  Конец ***********************************************
```

Командный файл 02.cmd

```
@ECHO OFF
CLS
FOR /R c:\ %%f IN (*.ini) DO echo %%f
```

Лабораторная работа №3

Файловая система и командный интерфейс ОС Linux

Теоретические сведения

Цель работы

Ознакомиться с командным интерфейсом, структурой файловой системы Линукс, типами файлов и командами управления файловой системой.

1.1 Организация файловой системы

Файловая система ОС Линукс (как и прочих unix-подобных систем) устроена так, что все ресурсы представлены единообразно, в виде файлов. Такой подход позволяет обеспечить универсальный интерфейс доступа к любым ресурсам: от физических устройств, до процессов, выполняющихся в системе. С точки зрения пользователя файловая система представляет логическую структуру каталогов и файлов. С другой стороны, невидимой пользователю, внутреннее устройство файловой системы реализует физические операции чтения/записи файлов на различные носители, алгоритмы доступа и многое другое.

Типы файлов

Для обеспечения единообразного доступа к файлам их прежде всего необходимо классифицировать. В Линукс это сделано следующим образом:

- обычные (regular) файлы - текстовые, исполняемые, графические и пр. файлы, создаваемые пользователями и прикладными программами;
- каталоги (directories) - именованные группы файлов и вложенных каталогов (т.е. содержимое каталога - суть файлы и другие каталоги);
- файлы устройств (devices) - соответствуют присутствующим в системе реальным (жесткие диски, принтеры, мыши, ЦП и т.д.) устройствам и т.н. псевдоустройствам (например, /dev/null). Файлы устройств представляют символьные (последовательного доступа) и блочные (произвольного доступа) устройства. К первыми относятся, например, параллельные и последовательные порты, ко вторым - жесткие диски;
- специальные файлы - сокеты (sockets) и именованные каналы (named pipes), которые предназначены для обмена информацией между процессами;
- символьные ссылки (symlinks) - именованные указатели на физические файлы (аналог ярлыков ОС Windows), содержащие только путь к некоторому файлу. Символьные ссылки могут указывать на файлы, хранящиеся как локальных, так и в сетевых каталогах.

Символьные ссылки (или "мягкие") не нужно путать с "жесткими", которые указывают на inode файла. Inode (идентификатор узла) - это уникальный числовой идентификатор узла (файла или каталога) файловой системы, по которому и осуществляется доступ к нему. Имя же файла (включая полный путь) ориентировано на пользовательское восприятие. Для оператора проще оперировать с осмысленными именами файлов (например: report.txt, myfoto.jpg и т.п.), чем с числовыми значениями. Прочие отличия "жестких" и "мягких" ссылок вам предстоит выяснить в ходе выполнения этой лабораторной работы.

Каталоги Линукс

Все файлы упорядочены по каталогам. Структура и назначение каждого из каталогов, созданных на этапе установке предопределены, хотя и могут быть (что крайне не рекомендуется) изменены суперпользователем.

Файловая система имеет иерархическую структуру и начинается от корневого каталога (/). Его подкаталогами являются:

- /bin - исполняемые файлы общего назначения;
- /boot - содержит образ загружаемого ядра;
- /dev - файлы устройств;
- /etc - конфигурационные файлы общего пользования;
- /home - домашние каталоги пользователей, включая программы и файлы личных предпочтений;
- /lib - общесистемные библиотеки;
- /mnt - каталог монтирования внешних файловых систем;
- /proc - виртуальная файловая система для чтения информации о процессах;
- /root - домашний каталог суперпользователя;
- /sbin - программы системного администрирования;
- /tmp - каталог для хранения временной информации;
- /usr - каталог пользовательских прикладных программ со всеми их исполнимыми и конфигурационными файлами. Например, в подкаталог /usr/local инсталлируются программы, не входящие в дистрибутив Линукс, или собираемые из исходных текстов.
- /var - каталог для хранения часто изменяющихся файлов. Например, спулера печати, различных лог-файлов, почтовых сообщений и т.п.
- /lost+found - каталог для нарушенных фрагментов файлов, обнаруженных в результате проверки файловой системы после сбоя.

Такая стуктура типична для большинства дистрибутивов Линукс, но могут иметься и дополнительные каталоги.

Именование файлов и каталогов

Файловая система Линукс поддерживает "длинные" имена, содержащие символы латиницы, национальных алфавитов, знаки пунктуации и спецсимволы. Абсолютно запрещенными к использованию в имени являются прямой и обратный слэши (/ и \). Максимальное количество символов в имени - 255. Понятие "расширения файла" в unix-системах отсутсвует как таковое, поэтому в имени может быть несколько частей, разделенных точками. Все имена - регистрозависимые.

Приведенные выше правила справедливы и для каталогов.

Файлы и каталоги, названия которых начинаются с точки (т.н. dot-файлы), являются аналогами "скрытых" файлов MS-DOS. Т.е. в общем случае они не отображаются при просмотре содержимого файловой системы.

Для быстрого доступа к файлам в оболочке имеются несколько переменных окружения, хранящих соответвующие пути. Это, например, переменная $HOME, в которой содержится пути к домашнему каталогу текущего пользователя. Т.е. действия команд

```
[usr1@localhost var]$ cd /home/usr1
```

и

```
[usr1@localhost var]$ cd $HOME
```

приведут к одному результату - переходу в домашний каталог пользователя usr1. Более того, в оболочке определен псевдоним для домашнего каталога - символ ~ (тильда) можно использовать аналогично $HOME. Например:

```
[usr1@localhost var]$ cd ~
[usr1@localhost ~]$ pwd
/home/usr1
```

[usr1@localhost var]$

1.2 Регистрация пользователя в системе

Для входа пользователя с терминала в многопользовательскую операционную систему LINUX необходимо зарегистрироваться в качестве пользователя. Для этого нужно после сообщения Login: ввести системное имя пользователя, например, "student". Если имя задано верно, выводится запрос на ввод пароля:
Password:
Наберите пароль "student" и нажмите клавишу *Enter*.
Если имя или пароль указаны неверно, сообщение *login* повторяется. Значение пароля проверяется в системном файле *password*, где приводятся и другие сведения о пользователях. После правильного ответа появляется приветствие LINUX и приглашение:
student@linux:>
Вы получили доступ к ресурсам ОС LINUX.
Выход из системы:
exit - окончание сеанса пользователя.

После успешного ввода имени пользователя и пароля система выводит приглашение к вводу команды.

- для суперпользователя root
$ - для всех остальных пользователей

Часто при первом входе в систему пользователя требуется поменять пароль, назначенный пользователю администратором - используйте команду passwd.
$ passwd

2. Командный интерфейс

Формат команд в ОС LINUX следующий:
имя команды [аргументы] [параметры] [метасимволы]
Имя команды может содержать любое допустимое имя файла; аргументы - одна или несколько букв со знаком минус (-); параметры - передаваемые значения для обработки; метасимволы интерпретируются как специальные операции. В квадратных скобках указываются необязательные части команд.

2.1. Группирование команд

Группы команд или сложные команды могут формироваться с помощью специальных символов (метасимволов):

- & - процесс выполняется в фоновом режиме, не дожидаясь окончания предыдущих процессов;
- ? - шаблон, распространяется только на один символ;
- - шаблон, распространяется на все оставшиеся символы;
- | - программный канал - стандартный вывод одного процесса является стандартным вводом другого;
- - переадресация вывода в файл;
- < - переадресация ввода из файла;
- ; - если в списке команд команды отделяются друг от друга точкой с запятой, то они выполняются друг за другом;

- && - эта конструкция между командами означает, что последующая команда выполняется только при нормальном завершении предыдущей команды (код возврата 0);
- || - последующая команда выполняется только, если не выполнилась предыдущая команда (код возврата 1);
- () - группирование команд в скобки;
- { } - группирование команд с объединенным выводом;
- [] - указание диапазона или явное перечисление (без запятых);
- >> - добавление содержимого файла в конец другого файла.

Примеры.

who | wc - подсчет количества работающих пользователей командой **wc** (word count - счет слов);

cat text.1 > text.2 - содержимое файла text.1 пересылается в файл text.2;

mail student < file.txt - электронная почта передает файл file.txt всем пользователям, перечисленным в командной строке;

cat text.1,text.2 - просматриваются файлы text.1 и text.2;

cat text.1 >> text.2 - добавление файла text.1 в конец файла text.2;

cc primer.c & - трансляция СИ - программы в фоновом режиме.

cc -o primer.o primer.c - трансляция СИ-программы с образованием файла выполняемой программы с именем primer.o;

rm text.* - удаление всех файлов с именем text;

{cat text.1; cat text.2} | lpr - просмотр файлов text.1 и text.2 и вывод их на печать;

2.2 Основные команды ОС

man <название команды> - вызов электронного справочника об указанной команде.
Выход из справочника - нажатие клавиши Q.
Команда **man man** сообщает информацию о том, как пользоваться справочником.

echo выдача на стандартный вывод строки символов, которая задана ей в качестве аргумента. Синтаксис команды **echo**:
echo [-n] [arg1] [arg2] [arg3]...
Команда помещает в стандартный вывод свои аргументы, разделенные пробелами и завершаемые символом перевода строки. При наличии флага *-n* символ перевода строки исключается. Передаваемая строка может быть перенаправлена в файл с использованием оператора перенаправления вывода > . Например:
$echo "Hello, world!" > myfile

who [am i] - получение информации о работающих пользователях.
В квадратных скобках указываются аргументы команды, которые можно опустить. Ответ представляется в виде таблицы, которая содержит следующую информацию:

- идентификатор пользователя;
- идентификатор терминала;

- дата подключения;
- время подключения.
-

date - вывод на экран текущей даты и текущего времени.

cal [[месяц]год] - календарь; если календарь не помещается на одном экране, то используется команда **cal год | more** и клавишей пробела производится постраничный вывод информации.

top – показывает список работающих в данный момент процессов и информацию о них, включая использование ими памяти и процессора. Список интерактивно формируется в реальном времени. Чтобы выйти из программы top, нажмите клавишу [q].

ps [Опции**] [number]** - команда для вывода информации о процессах:
Опции

- -a все терминальные процессы
- -e все процессы.
- -g*список* выбирать процессы по *списку* лидеров групп.
- -p*список* выбирать процессы по *списку* идентификаторов процессов.
- -t*список* выбирать процессы по *списку* терминалов.
- -u*список* выбирать процессы по *списку* идентификаторов пользователей.
- f генерировать полный листинг.
- -l генерировать листинг в длинном формате.
- number - номер процесса.

Команда ps без параметров выводит информацию только об активных процессах, запущенных с данного терминала, в том числе и фоновых. На экран выводится подробная информация обо всех активных процессах в следующей форме:

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
1	S	200	210	7	0	2	20	80	30	703a	03	0:07	cc
1	R	12	419	7	11	5	20	56	20	03	0:12	ps	

- F - флаг процесса (1 - в оперативной памяти, 2 - системный процесс, 4 - заблокирован в ОЗУ, 20 - находится под управлением другого процесса, 10 - подвергнут свопингу);
- S - состояние процесса (O - выполняется процессором , S - задержан, R - готов к выполнению, I - создается);
- UID - идентификатор пользователя;
- PID - идентификатор процесса;
- PPID - номер родительского процесса;
- C - степень загруженности процессора;
- PRI - приоритет процесса, вычисляется по значению переменной
- NICE и чем больше число, тем меньше его приоритет;
- NI - значение переменной NICE для вычисления динамического
- приоритета, принимает величины от 0 до 39;
- ADDR - адрес процесса в памяти;
- SZ - объем ОЗУ, занимаемый процессом;
- WCHAN - имя события, до которого процесс задержан, для ак-
- тивного процесса - пробел;
- TTY - номер управляющего терминала для процесса;
- TIME - время выполнения процесса;
- CMD - команда, которая породила процесс.

nice [-приращение приоритета] команда[аргументы] - команда изменения приоритета. Каждое запущенное задание (процесс) имеет номер приоритета в диапазоне от 0 до 39, на основе которого ядро вычисляет фактический приоритет, используемый для планирования процесса. Значение 0 представляет наивысший приоритет, а 39 - самый низший. Увеличение номера приоритета приводит к понижению приоритета, присвоенного процессу. Команда
nice -10 ls -l
увеличивает номер приоритета, присвоенный процессу ls -l на 10.
renice 5 1836 - команда устанавливает значение номера приоритета процесса с идентификатором 1836 равным 5. Увеличить приоритет процесса может только администратор системы.

kill [-sig] <идентификатор процесса> - прекращение процесса до его программного завершения. sig - номер сигнала. Sig = -15 означает программное (нормальное) завершение процесса, номер сигнала = -9 - уничтожение процесса. По умолчанию sig= -9. Вывести себя из системы можно командой kill -9 0. Пользователь с низким приоритетом может прервать процессы, связанные только с его терминалом.

free – Показывает общее количество свободной и используемой физической памяти и памяти отведенной для свопирования в системе, так же и совместно используемую память и буфера используемые ядром. Синтаксис :
free [-b | -k | -m] [-o] [-s *delay*] [-t] [-V]
Опции :

- -b показывает количество памяти в байтах; опция -k (по умолчанию) показывает количество памяти в килобайтах;
- Опция -m показывает количество памяти в мегабайтах.
- -t показывает строки содержащие полное количество памяти.
- -o запрещает показывать строки относящиеся к "массиву буфера" . Если не определено отнять/добавить память буферов из/в используемую/свободную память (соответственно!).
- -s разрешает безостановочно выводить информацию с промежутком в *delay* секунд.
- -V показывает информацию о версии программы.

tty – информация о терминалах пользователя.

2.3 Команды работы с файловой системой

Для управления файловой системой имеются различные команды, реализующие операции по созданию, чтению, копированию, переименованию/перемещению, изменению и удалению файлов и каталогов. Как правило, это специализированные команды, хорошо выполняющие свою задачу, однако некоторые функции могут частично дублироваться другими командами, что только добавляет гибкости управлению файлами.
Основными командами для выполнения файловых операций являются: pwd, ls, cp, mv, dir, rm, cd, rmdir, mvdir, mkdir, ln. Информацию о их назначении и параметрах доступна в справке.

pwd – Выдача имени текущего каталога

cd – Смена текущего каталога Синтаксис команды:
cd [каталог]

Команда cd применяется для того, чтобы сделать заданный каталог текущим. Если каталог не указан, используется значение переменной окружения $HOME (обычно это каталог, в который Вы попадаете сразу после входа в систему). Если каталог задан полным маршрутным именем, он становится текущим. По отношению к новому каталогу нужно иметь право на выполнение, которое в данном случае трактуется как разрешение на поиск. Текущий каталог - это каталог, в котором в данный момент находится пользователь. При наличии прав доступа, пользователь может перейти после входа в систему в другой каталог. Текущий каталог обозначается точкой (**.**); родительский каталог, которому принадлежит текущий, обозначается двумя точками (**..**).

cat <имя файла> - вывод содержимого файла на экран.
Команда **cat > text.1 –** создает новый файл с именем text.1, который можно заполнить символьными строками, вводя их с клавиатуры. Нажатие клавиши *Enter* создает новую строку. Завершение ввода – нажатие *Ctrl - d*.
Команда **cat text.1 > text.2** пересылает содержимое файла text.1 в файл text.2.
Слияние файлов осуществляется командой **cat text.1 text.2 > text.3**.

ls [-опции] [имя] - вывод содержимого каталога на экран. Если аргумент не указан, выдается содержимое текущего каталога. Аргументы команды:

- l –- список включает всю информацию о файлах;
- t – сортировка по времени модификации файлов;
- a – в список включаются все файлы, в том числе и те, которые начинаются с точки;
- s – размеры файлов указываются в блоках;
- d – вывести имя самого каталога, но не содержимое;
- r – сортировка строк вывода;
- i – указать идентификационный номер каждого файла;
- v – сортировка файлов по времени последнего доступа;
- q – непечатаемые символы заменить на знак ?;
- c – использовать время создания файла при сортировке;
- g – то же что -l, но с указанием имени группы пользователей;
- f – вывод содержимого всех указанных каталогов, отменяет флаги -l, -t, -s, -r и активизирует флаг -a;
- C – вывод элементов каталога в несколько столбцов;
- F – добавление к имени каталога символа / и символа * к имени файла, для которых разрешено выполнение;
- R – рекурсивный вывод содержимого подкаталогов заданного каталога.

Пример
ls -l file.1 - чтение атрибутов файла;

mkdir – Создание каталога. Синтаксис:
mkdir [-m режим_доступа] [-p] каталог ...
По команде mkdir создается дин или несколько каталогов с режимом доступа 0777 [возможно измененном с учетом umask и опции -m]. Стандартные файлы (. - для самого каталога и .. - для вышележащего) создаются автоматически; их нельзя создать по имени. Для создания каталога необходимо располагать правом записи в вышележащий каталог. Идентификаторы владельца и группы новых каталогов устанавливаются соответственно равными реальным идентификаторам владельца и группы процесса. Командой mkdir обрабатываются две опции:

- -m режим_доступа - (явное задание режима_доступа для создаваемых каталогов [см. chmod]).

- -p (при указании этой опции перед созданием нового каталога предварительно создаются все несуществующие вышележащие каталоги).

cp – Копирование файлов. Синтаксис :
cp файл1 [файл2 ...] целевой_файл
Команда cp копирует файл1 в целевой_файл. Файл1 не должен совпадать с целевым_файлом (будьте внимательны при использовании метасимволов shell'a). Если целевой_файл является каталогом, то файл1, файл2, ..., копируются в него под своими именами. Только в этом случае можно указывать несколько исходных файлов.
Если целевой_файл существует и не является каталогом, его старое содержимое теряется. Режим, владелец и группа целевого_файла при этом не меняются.
Если целевой_файл не существует или является каталогом, новые файлы создаются с теми же режимами, что и исходные (кроме бита навязчивости, если Вы не суперпользователь). Время последней модификации целевого_файла (и последнего доступа, если он не существовал), а также время последнего доступа к исходным файлам устанавливается равным времени, когда выполняется копирование.
Если целевой_файл был ссылкой на другой файл, все ссылки сохраняются, а содержимое файла изменяется.
Пример:
cp file.1 file.2 - копирование файла с переименованием;

mv – Перемещение (переименование) файлов. Синтаксис команды:
mv [-f] файл1 [файл2 ...] целевой_файл
Команда mv перемещает (переименовывает) файл1 в целевой_файл. Файл1 не должен совпадать с целевым_файлом (будьте внимательны при использовании метасимволов shell'a).
Если целевой_файл является каталогом, то файл1, файл2, ..., перемещаются в него под своими именами. Только в этом случае можно указывать несколько исходных файлов.
Если целевой_файл существует и не является каталогом, его старое содержимое теряется. Если при этом обнаруживается, что в целевой_файл не разрешена запись, то выводится режим этого файла [см. chmod] и запрашивается строка со стандартного ввода. Если эта строка начинается с символа y, то требуемые действия все же выполняются, при условии, что у пользователя достаточно прав для удаления целевого_файла. Если была указана опция –f или стандартный ввод назначен не на терминал, то требуемые действия выполняются без всяких запросов. Вместе с содержимым целевой_файл наследует режим файла1.
Если файл1 является каталогом, то он переименовывается в целевой_файл, только если у этих двух каталогов общий надкаталог; при этом все файлы, находившиеся в файле1, перемещаются под своими именами в целевой_файл. Если файл1 является файлом, а целевой_файл - ссылкой, причем не единственной, на другой файл, то все остальные ссылки сохраняются, а целевой_файл становится новым независимым файлом.
Пример:
mv file.1 file.2 - переименование файла file.1 в file.2;
mv file.1 file.2 file.3 directory - перемещение файлов file.1, file.2, file.3 в указанную директорию;

rm – удаление файлов**.** Синтаксис команды:
rm [-f] [-i] файл ...
rm -r [-f] [-i] каталог ... [файл ...]

Команда rm служит для удаления указанных имен файлов из каталога. Если заданное имя было последней ссылкой на файл, то файл уничтожается. Для удаления пользователь должен обладать правом записи в каталог; иметь право на чтение или запись файла не обязательно. Следует заметить, что при удалении файла в Linux, он удаляется навсегда. Здесь нет возможностей вроде "мусорной корзины" в windows 95/98/NT или команды undelete в DOS. Так что, если файл удален, то он удален!
Если нет права на запись в файл и стандартный ввод назначен на терминал, то выдается (в восьмеричном виде) режим доступа к файлу и запрашивается подтверждение; если оно начинается с буквы y, то файл удаляется, иначе - нет. Если стандартный ввод назначен не на терминал, команда rm ведет себя так же, как при наличии опции -f. Допускаются следующие три опции:

- -f Команда не выдает сообщений, когда удаляемый файл не существует, не запрашивает подтверждения при удалении файлов, на запись в которые нет прав. Если нет права и на запись в каталог, файлы не удаляются. Сообщение об ошибке выдается лишь при попытке удалить каталог, на запись в который нет прав (см. опцию -r).
- -r Происходит рекурсивное удаление всех каталогов и подкаталогов, перечисленных в списке аргументов. Сначала каталоги опустошаются, затем удаляются. Подтверждение при удалении файлов, на запись в которые нет прав, не запрашивается, если задана опция -f или стандартный ввод не назначен на терминал и не задана опция -i. При удалении непустых каталогов команда rm -r предпочтительнее команды rmdir, так как последняя способна удалить только пустой каталог. Но команда rm -r может доставить немало острых впечатлений при ошибочном указании каталога!
- -i Перед удалением каждого файла запрашивается подтверждение. Опция -i устраняет действие опции -f; она действует даже тогда, когда стандартный ввод не назначен на терминал.

ПРИМЕРЫ
rm file.1 file.2 file.3 - удаление файлов file.1, file.2, file.3;.

Опция -i часто используется совместно с -r. По команде:
rm -ir dirname
запрашивается подтверждение: **directory dirname: ?**
При положительном ответе запрашиваются подтверждения на удаление всех содержащихся в каталоге файлов (для подкаталогов выполняются те же действия), а затем подтверждение на удаление самого каталога.

rmdir – Удаление каталогов. Синтаксис команды:
rmdir [-p] [-s] каталог ...
Команда rmdir удаляет указанные каталоги, которые должны быть пустыми. Для удаления каталога вместе с содержимым следует воспользоваться командой rm с опцией -r. Текущий каталог [см. pwd] не должен принадлежать поддереву иерархии файлов с корнем – удаляемым каталогом. Для удаления каталогов нужно иметь те же права доступа, что и в случае удаления обычных файлов [см. rm]. Командой rmdir обрабатываются следующие опции:

- -p Позволяет удалить каталог и вышележащие каталоги, оказавшиеся пустыми. На стандартный вывод выдается сообщение об удалении всех указанных в маршруте каталогов или о сохранении части из них по каким-либо причинам.
- -s Подавление сообщения, выдаваемого при действии опции -p.

grep – поиск файлов с указанием или без указания контекста (шаблона поиска). Синтаксис:
grep [-vcilns] [шаблон поиска] <имя файла>

Значение ключей:

- v – выводятся строки, не содержащие шаблон поиска;
- c – выводится только число строк, содержащих или не содер-
- жащих шаблон;
- i – при поиске не различаются прописные и строчные буквы;
- l – выводятся только имена файлов, содержащие указанный
- шаблон;
- n – перенумеровать выводимые строки;
- s – формируется только код завершения.

ln (link) – создание ссылок**.** Один файл можно сделать принадлежащим нескольким каталогам. Для этого используется команда:

ln <имя файла 1> <имя файла 2>

Имя 1-го файла - это полное составное имя файла, с которым устанавливается связь; имя 2-го файла - это полное имя файла в новом каталоге, где будет использоваться эта связь. Новое имя может не отличаться от старого. Каждый файл может иметь несколько связей, т.е. он может использоваться в разных каталогах под разными именами.

Команда **ln** с аргументом -s создает символическую связь:

ln -s <имя файла 1> <имя файла 2>

Здесь имя 2-го файла является именем символической связи. Символическая связь является особым видом файла, в котором хранится имя файла, на который символическая связь ссылается. LINUX работает с символической связью не так, как с обычным файлом - например, при выводе на экран содержимого символической связи появятся данные файла, на который эта символическая связь ссылается.

2.4 Режимы доступа к файлам

В LINUX различаются 3 уровня доступа к файлам и каталогам:

1) доступ владельца файла;
2) доступ группы пользователей, к которой принадлежит владелец файла;
3) остальные пользователи.

Для каждого уровня существуют свои байты атрибутов, значение которых расшифровывается следующим образом:

r – разрешение на чтение;
w – разрешение на запись;
x – разрешение на выполнение;
- – отсутствие разрешения.

Первый символ байта атрибутов определяет тип файла и может интерпретироваться со следующими значениями:

- – обычный файл;
d – каталог;
l – символическая связь;
в – блок-ориентированный специальный файл, который соответствует таким периферийным устройствам, как накопители на магнитных дисках;
c – байт-ориентированный специальный файл, который может соответствовать таким периферийным устройствам как принтер, терминал.

В домашнем каталоге пользователь имеет полный доступ к файлам (READ, WRITE, EXECUTE; r, w, x). Атрибуты файла можно просмотреть командой **ls -l** и они представляются в следующем формате:

```
d rwx rwx rwx
```

```
|   |    |   |Доступ для остальных пользователей
|   |    |Доступ к файлу для членов группы
|   |Доступ к файлу владельца
|Тип файла (директория)
```
Пример. Командой **ls -l** получим листинг содержимого текущей директории student:
```
- rwx --- --- 2 student 100 Mar 10 10:30 file_1
- rwx --- r-- 1 adm 200 May 20 11:15 file_2
- rwx --- r-- 1 student 100 May 20 12:50 file_3
```
После байтов атрибутов на экран выводится следующая информация о файле:
- число связей файла;
- имя владельца файла;
- размер файла в байтах;
- дата создания файла (или модификации);
- время;
- имя файла.

Атрибуты файла и доступ к нему, можно изменить командой:
chmod <коды защиты> <имя файла>
Коды защиты могут быть заданы в числовом или символьном виде. Для символьного кода используются:

- знак плюс (+) - добавить права доступа;
- знак минус (-) - отменить права доступа;
- r,w,x - доступ на чтение, запись, выполнение;
- u,g,o - владельца, группы, остальных.

Коды защиты в числовом виде могут быть заданы в восьмеричной форме. Для контроля установленного доступа к своему файлу после каждого изменения кода защиты нужно проверять свои действия с помощью команды ls -l.
Примеры:
chmod g+rw,o+r file.1 - установка атрибутов чтения и записи для группы и чтения для всех остальных пользователей;
ls -l file.1 - чтение атрибутов файла;
chmod o-w file.1 - отмена атрибута записи у остальных пользователей;

2.5 Создание командных файлов

Командный файл в Unix представляет собой обычный текстовый файл, содержащий набор команд Unix и команд Shell. Для того чтобы командный интерпретатор воспринимал этот текстовый файл, как командный необходимо установить атрибут на исполнение.
Установку атрибута на исполнение можно осуществить командой chmod или через mc по клавише F9 выйти в меню и выбрать вкладку File, далее выбрать изменение атрибутов файла.
Например.
$ echo " ps -af " > commandfile
$ chmod +x commandfile
$ ./commandfile
В представленном примере команда echo " ps -af " > commandfile создаст файл с одной строкой " ps -af ", команда chmod +x commandfile установит атрибут на исполнение для этого файла, команда ./commandfile осуществит запуск этого файла.

Задание на лабораторную работу.

1.Ознакомиться с командами Linux. Выполнить команды top, free, ps с различными опциями.

2. Войти в свой домашний каталог. Для этого нужно сделать команду cd ~ . Вы находитесь в своем рабочем каталоге. Здесь хранятся ваши пользовательские файлы и настройки программ, которые вы используете.

3. Создать следующую структуру каталогов и файлов:

1) в домашнем каталоге создать каталог inform

2) Перейти в каталог и inform создать в нем каталог lab1

3) Внутри каталога lab1 создать каталог catalog1, файл file1(например, используя команду echo), каталог catalog2. Перейти в каталог catalog2.

4) Внутри каталога catalog2 создать файлы file3 и file4 , каталог catalog3

5) Внутри каталога catalog3 создать файл file5, жесткую ссылку на файл file1, жесткую
ссылку на каталог catalog2.

6) Создать в каталоге lab1 символичесткую ссылку s_link на файл file5

4. Запустить программу MC (Midnight Commander): mc. Посмотреть структуру созданных вами каталогов и просмотреть содержимое файлов.

Лабораторная работа №4

Shell – ПРОГРАММИРОВАНИЕ

Теоретические сведения

1. ОПЕРАТОРЫ – КОМАНДЫ

Язык shell по своим возможностям приближается к высокоуровневым алгоритмическим языкам программирования. Операторы языка **shell** позволяют создавать собственные программы, не требует компиляции, построения объектного файла и последующей компоновки, так как shell, обрабатывающий их, является транслятором интерпретирующего, а не компилирующего типа.
Текст процедуры набирается как обычный текстовый файл. Проверенный и отлаженный shell-файл может быть вызван на исполнение, например, следующим способом:
$ chmod u+x shfil
$ shfil
$

Такая форма предполагает, что файл процедуры новый и его надо сначала сделать выполняемым. Можно использовать также и следующий способ:
$ sh -c "shfil" или $ sh shfil

В этих случаях по команде sh вызывается вторичный интерпретатор shell, и в качестве аргумента ему передается командная строка, содержащая имя файла процедуры shfil, находящегося в текущем каталоге. Однако, этот способ накладывает ограничения на исполнение некоторых команд ОС управления процессами.
Процедуре при ее запуске могут быть переданы аргументы. В общем случае командная строка вызова процедуры имеет следующий вид:
$ имя_процедуры $1 $2 ...$9

Каждому из девяти первых аргументов командной строки в тексте процедуры соответствует один из позиционных параметров: $1, $2, ..., $9 соответственно. Параметр $0 соответствует имени самой процедуры, т.е. первому полю командной строки. К каждому из 10 первых аргументов можно обратиться из процедуры, указав номер его позиции. Количество аргументов присваивается другой переменной: **$#**(диез). Наконец, имя процедуры - это $0; переменная $0 не учитывается при подсчете $#.
Сам интерпретатор shell автоматически присваивает значения следующим переменным (параметрам):

? — *значение, возвращенное последней командой;*
$ — *номер процесса;*
! номер фонового процесса;
— *число позиционных параметров, передаваемых в shell;*
*** — *перечень параметров, как одна строка;*
@ — *перечень параметров, как совокупность слов;*
- флаги, передаваемые в shell.

При обращении к этим переменным (т.е при использовании их в командном файле - shell-программе) следует впереди ставить "$".

Пример.
Вызов фала – *specific par1 par2 par3* имеющего вид

```
echo $0 - имя расчета
echo $? - код завершения
echo $$ - идентификатор последнего процесса
echo $! - идентификатор последнего фонового процесса
echo
echo $* - значения параметров, как строки
echo $@ - значения параметров, как слов
echo
set -au
echo $- - режимы работы интерпретатора
```

Выдаст на экран

```
specific - имя расчета
0 - код завершения
499 - идентификатор последнего процесса
98 - идентификатор последнего фонового процесса
par1 par2 par3 - значения параметров, как строки
par1 par2 par3 - значения параметров, как слов
au - режимы работы интерпретатора
```

Некоторые вспомогательные операторы:
echo - вывод сообщений из текста процедуры на экран.

```
$ echo "начало строки
> продолжение строки"
```

 # - для обозначения строки комментария в процедуре. (Строка не будет обрабатываться shell-ом)**.**

banner - вывод сообщения на экран заглавными буквами (например для идентификации следующих за ним сообщений).

```
$banner 'hello ira'
HELLO IRA
$
```

Простейший пример. Здесь оператор ***echo*** выполняется в командном режиме.

```
$shfil p1 pp2 petr

$echo $3
petr
$
```

Пример
Передача большого числа параметров

```
echo "$0: Много параметров"
echo " Общее число параметров = $#
Исходное состояние: $1 $5 $9 "
shift
echo "1 сдвиг: первый=$1 пятый=$5 девятый=$9"
shift 2
```

echo "1 + 2 = 3 сдвига: первый=$1 пятый=$5 девятый=$9"
perem=`expr $1 + $2 + $3`
echo $perem

Значения параметрам, передаваемым процедуре, можно присваивать и в процессе работы процедуры с помощью оператора
set - присвоить значения позиционным параметрам;
$set a1 ab2. abc
$echo $1 $2
a1 ab2 - в этом примере параметры указываются в явном виде.
$

Запуск ***set*** без параметров выводит список установленных системных переменных:
HOME=/home/sae
PATH=/usr/local/bin:/usr/bin:/bin:.:/usr/bin/X11:
IFS=
LOGNAME=sae
MAIL=/var/spool/mail/sae
PWD=/home/sae/STUDY/SHELL
PS1=${PWD}:" "
PS2=>
SHELL=/bin/bash
TERM=linux
TERMCAP=console|con80x25|dumb|linux:li#25:co#80::
UID=501

Количество позиционных параметров может быть увеличено до необходимого значения путем "сдвига" их в командной строке влево на одну позицию с помощью команды shift без аргументов:
shift - сдвинуть позиционные параметры влево на одну позицию

После выполнения shift прежнее значение параметра $1 теряется, значение $1 приобретает значение $2, значение $2 - значение $3 и т.д..Продолжение предыдущего примера:
$shift
$echo $1 $2
ab2 abc
$

В UNIX при написании операторов важное значение отводится кавычкам (апострофам):

'...' - для блокирования специальных символов, которые могут быть интерпретированы как управляющие;
"..." - блокирование наиболее полного набора управляющих символов или указания того, что здесь будет обрабатываться не сам аргумент, а его значение;
`...` - (обратные кавычки или знак ударения) для указания того, что они обрамляют команду и здесь будет обрабатываться результат работы этой команды (подстановка результатов работы указанной команды).

Пример 1.
$ date

Apr 3 14:27:07 2005
$ set `date`
$ echo $3
14:30:25
$

Пример 2.
$echo `ls`
fil.1
fil.2
...
$echo "`ls`"
 # одинарные кавычки блокируют действие обратных кавычек
 # т.е. они распечатываются как обычные символы
`ls`
$

Для ввода строки текста со стандартного устройства ввода используется оператор:
read имя1 [имя2 имя3 .] - чтение строки слов со стандартного ввода

Команда вводит строку, состоящую из нескольких полей (слов), со стандартного ввода, заводит переменную для каждого поля и присваивает первой переменной имя1, второй переменной - имя2, и т.д. Если имен больше, чем полей в строке, то оставшиеся переменные будут инициализированы пустым значением. Если полей больше, чем имен переменных, то последней переменной будет присвоена подстрока введенной строки, содержащая все оставшиеся поля, включая разделители между ними. В частности, если имя указано только одно, то соответствующей ему переменной присваивается значение всей строки целиком.
Пример:
#Текст процедуры:
echo "Введите значения текущих: гг мм ччвв"
read 1v 2v 3v
echo "год 1v"
echo "месяц 2v"
echo "сегодня 3v"
 # здесь кавычки используются для блокирования пробелов
#Результат выполнения процедуры:
Введите значения текущих: гг мм ччвв
2005 Март 21 9:30 <Enter>
год 2005
месяц Март
сегодня 21 9:30

2 УПРАВЛЕНИЕ ЛОКАЛЬНЫМИ ПЕРЕМЕННЫМИ

В отличии от рассмотренных в начале курса системных переменных среды, переменные языка shell называются **локальными переменными** и используются в теле процедур для решения обычных задач. Локальные переменные связаны только с породившим их процессом. Локальные переменные могут иметь имя, состоящее из одного или нескольких символов. Присваивание значений переменным осуществляется с помощью известного оператора:

"=" - присвоить (установить) значение переменной.

При этом если переменная существовала, то новое значение замещает старое. Если переменная не существовала, то она строится автоматически shell. Переменные хранятся в области ОП - области локальных данных.

$count=3
$color=red belt
$fildir=lev/d1/d12
$
Еще пример:
 # текст процедуры
b=”1 + 2”
echo c=$b
#в результате выполнения процедуры выводится текст,
 # включающий текст переменной b
c=1+2

3. ПОДСТАНОВКА ЗНАЧЕНИЙ ПЕРЕМЕННЫХ

Процедура подстановки выполняется shell-ом каждый раз когда надо обработать значение переменной если используется следующая конструкция: **первый вид подстановки $имя_переменной** -. на место этой конструкции будет подставлено значение переменной.

Примеры подстановки длинных маршрутных имен:
$echo $HOME
/home/lev
$filname=$HOME/f1
$more $filname
<текст файла f1 из каталога lev>
$
Использование полного маршрутного имени в качестве значения переменной позволяет пользователю независимо от местонахождения в файловой системе получать доступ к требуемому файлу или каталогу.
Если в строке несколько присваиваний, то последовательность их выполнения - справа налево.

Пример 1.	Пример 2.	Пример 3.

$ z=$y y=123
$ echo $z $y
123 123
$ y=abc z=$y
$ echo “z”
 	
123
$ echo “y”
abc
$
	$ var=/user/lab/ivanov
	$ cd $var
	$ pwd
	/udd/lab/ivanov
	$
		$ namdir='ls'
		$ $namdir
		fil1
		fil2
		fil3
		...
		$

В последнем примере переменной ***namdir*** присвоено значение, которое затем используется в качестве командной строки запускаемой команды. Это команда ls.

Второй вид подстановки – подстановка результатов работы команды вместо самой команды.

Пример 4.
В данном случае команда ls непосредственно выполняется уже в первой строке, и переменной ***filnam*** присваивается результат ее работы.

```
$ filnam=`ls`
$ echo $filnam
fil1
fil2
fil3
...
$
```

Пример 5.

```
$ A=1 B=2
$ dat="$A + $B"
$ echo $dat
1 + 2
$
```

С переменными можно выполнять арифметические действия как и с обычными числами с использованием специального оператора:
expr - вычисление выражений.

Для **арифметических операций**, выполнимых командой ***expr***, используются операторы:
+ сложение; - вычитание; * умножение (обратная прямая скобка \ используется для отмены действия управляющих символов, здесь *); / деление нацело; % остаток от деления.
Для **логических операций** арифметического сравнения чисел командой ***expr*** используются следующие обозначения: = равно; != не равно; \< меньше; \<= меньше или равно;
\> больше; \>= больше или равно.
Все операнды и операторы являются самостоятельными аргументами команды ***expr*** и поэтому должны отделяться друг от друга и от имени команды ***expr*** пробелами.
Пример 6.

```
# Текст процедуры:
a=2
a=`expr $a + 7`
b=`expr $a / 3`
c=`expr $a - 1 + $b`
d=`expr $c % 5`
e=`expr $d - $b`
echo $a $b $c $d $e

#Результат работы процедуры:
9 3 11 1 -2
```

Команда ***expr*** выводит результат вычисления на экран. Поэтому, если он не присвоен никакой переменной, то не может быть использован в программе.
При решении логических задач, связанных с **обработкой символьных строк (текстов)** команда ***expr*** может быть использована, например, как средство для подсчета символов в

строках или для вычленения из строки цепочки символов. Операция обработки строк символов задается **кодом операции ":"** и шаблонами. В частности:
'.*' - шаблон для подсчета числа символов в строке,
'...\(.*\)....' - шаблон для выделения подстроки удалением символов строки, соответствующих точкам в шаблоне .

Пример 7. *$ m=aaaaaa* *$ expr $m : '.*'* *6* *$*	*Пример 8.* *$ n=abcdefgh* *$ expr $n : '...\(.*\)..'* *def* *$*

Выводимая информация - количество символов или подстрока - может быть присвоена некоторой переменной и использована в дальнейших вычислениях.

Третий вид подстановки - применяется для **подстановки команд или целых shell-процедур.** Используется для замены кода команды или текста процедуры на результат их выполнения в той же командной строке:
$(командная_строка) - подстановка осуществляется также перед запуском на исполнение командной строки.
В данном случае в качестве подставляемой команды может быть использована также любая имеющая смысл sh-процедура. Shell просматривает командную строку и выполняет все команды между открывающей и закрывающей скобками. Рассмотрим примеры присвоения значений и подстановки значений локальных переменных.

Пример 9.
$ A='string n'
$ count=$(expr $A : '.')*
$ echo $count
8
$
#Продолжение примера:
$ B=$(expr $A : '..\(.\))*
$ echo $B
ring
$
Рассмотрим пример на разработку простейшей линейной процедуры обработки переменных средствами языка shell.
ЗАДАНИЕ:
Создать файл, содержащий процедуру сложения двух чисел. Числа передаются в виде параметров при обращении к процедуре. Выполнить процедуру.
$ cat>comf
SUM=$(expr $1 + $2)
echo "$1 + $2 = $SUM"
*<Ctrl*D>*

$ sh comf 3 5
3 + 5 = 8
$

На экране можно **просмотреть все заведенные локальные переменные** с помощью известной команды:
$ set
$

Удаление переменных:
$unset перем1 [перем2 .....]
$

4. ЭКСПОРТИРОВАНИЕ ЛОКАЛЬНЫХ ПЕРЕМЕННЫХ В СРЕДУ shell

При выполнении процедуры ей можно передавать как позиционные параметры так и ключевые - локальные переменные порожденного процесса. **Локальные переменные** помещаются в область локальных переменных, связанную с конкретным текущим процессом, породившим переменную. Они доступны только этому процессу и недоступны порожденным процессам-потомкам (например – sh-процедурам). Переменные другим процессам можно передавать неявно через среду. Для этого локальная переменная должна быть экспортирована (включена) в среду, в которой исполняется процедура, использующая эту переменную. Среда пользователя, т.е. **глобальные переменные**, доступна всем процессам.
Три формата команды экспортирования:

$export список имен локальных переменных
$export имя_лок_переменной=значение
$export (без параметров) **- выводит перечень всех экспортированных локальных и переменных среды** (аналог команды env)**.**

Рассмотрим некоторый фрагмент протокола работы с системой.
$color = red переменная определена, но не экспортирована
$export count = 1 переменная определена и экспортирована, т.е. потенциально доступна всем порождаемым процессам

$export
PATH = ...
HOME = ...
color = red
count = 1

$cat proc1 создание порожденного процесса процедуры
echo $color
echo $count
exit завершение процесса
<ctrl.D>

$proc1 выполнение процедуры на экран выводится значение только одной, экспортированной переменной; вторая переменная – не определена
1

$cat proc2 еще одна процедура выводятся значения обеих переменных, т.к. они определены в самой процедуре
color = black
count = 2
echo $color
echo $count
exit

$proc2
black
1
.
$echo $color
red
$echo $count
1
$

На экран выводятся первоначальные значения переменных родительского процесса – shell. Новые (измененные) значения локальных переменных существуют только на время существования породившего их порожденного процесса. Чтобы изменить значене переменной родительского процесса ее надо экспортировать. Но после завершения порожденного среда родительского восстанавливается.

5. ПРОВЕРКА УСЛОВИЙ И ВЕТВЛЕНИЕ ВЫЧИСЛИТЕЛЬНЫХ ПРОЦЕССОВ

Все команды UNIX вырабатывают код завершения (возврата), обычно для того, чтобы в дальнейшем можно было выполнить диагностику посредством проверки значения кода завершения и определить: нормально завершилось выполнение команды (=0 или true) или не нормально (# 0 или false). Например, если (=1), то ошибка синтаксическая.
Код завершения после выполнения каждой команды помещается автоматически в некоторую специальную.
Системную переменную и ее значение можно вывести на экран: **echo $?**

Пример:
$true
$echo $?
0
$ls
$echo $?
0
$false
$echo $?
1
$cp
<сообщение о некорректности заданной команды – нет параметров>
$echo $?
1
$echo $?
0
$

Код завершения используется для программирования условных переходов в sh-процедурах. Проверка истинности условий для последующего ветвления вычислительного процесса процедур может быть выполнена с помощью команды:
test <проверяемое отношение/условие>
Вместо мнемоники команды может использоваться конструкция с квадратными скобками:
[проверяемое отношение/условие] синоним команды test.

Аргументами этой команды могут быть имена файлов, числовые или нечисловые строки (цепочки символов). Командой вырабатывается код завершения (код возврата), соответствующий закодированному в команде test условию. Код завершения проверяется следующей командой. Если закодированное параметрами условие выполняется, то вырабатывается логический результат (значение некоторой системной переменной) - true, если нет - false.
Код возврата может обрабатываться как следующей за test командой, так и специальной конструкцией языка: **if-then-else-fi**.
1. **Проверка файлов:**
test -ключ имя_файла
Ключи:
-r *файл существует и доступен для чтения;*
-w *файл существует и доступен для записи;*
-x файл существует и доступен для исполнения;
-f файл существует и имеет тип "-", т.е. обычный файл;
-s файл существует, имеет тип "-" и не пуст;
-d файл существует и имеет тип "d", т.е. файл - каталог.

2. Сравнение числовых значений:
test число1 –к число2
Числа могут быть как просто числовыми строками, так и переменными, которым эти строки присвоены в качестве значений. Ключи для анализа числовых значений:
-eq равно; **-ne** не равно;
-lt меньше; **-le** меньше или равно;
-gt больше; **-ge** больше или равно.
Пример:
```
$x=5
$[$x -lt 7]
$echo $?
0
$[$x -gt 7]
$echo $?
1
$
```

3. Сравнение строк:
test [-n] 'строка' - строка не пуста (n – число проверяемых строк)
test -z 'строка' - строка пуста
test 'строка1' = 'строка2' - строки равны
test 'строка1' != 'строка2' - строки не равны
Необходимо заметить, что все аргументы команды test - строки, имена, числа, ключи и знаки операций являются самостоятельными аргументами и должны разделяться пробелами.

Пример.
```
$x = abc
$["$x" = "abc"]
$echo $?
0
$["$x" != "abc"]
$echo $?
```

1
$
ЗАМЕЧАНИЕ: выражение вида "$переменная" лучше заключать в двойные кавычки, что предотвращает в некоторых ситуациях возможную неподходящую замену переменных shell-ом.
Особенности сравнения чисел и строк. Shell трактует все аргументы как числа в случае, если осуществляется сравнение чисел, и все аргументы как строки, если осуществляется сравнение строк. Пример:
$X = 03
$Y =3
$[$X -eq $Y] - сравниваются значения чисел
$echo $?
0
$["$X" = "$Y"] - числа сравниваются как строки символов
$echo $?
1
$
Ветвление вычислительного процесса в shell-процедурах осуществляется семантической конструкцией:
if список_команд1
then список_команд2
[else список_команд3
fi

Список_команд - это или одна команда или несколько команд, или фрагмент shell-процедуры. Если команды записаны на одной строке, то они разделяются точкой с запятой. Для задания пустого списка команд следует использовать специальный оператор:
: (двоеточие) -пустой оператор.
Список_команд1 передает оператору if код завершения последней выполненной в нем команды. Если он равен 0, то выполняется список_команд2.Таким образом, код возврата 0 эквивалентен логическому значению "истина". В противном случае он эквивалентен логическому значению "ложь" и выполняется либо список_команд3 после конструкции else, либо - завершение конструкции if словом fi.
В качестве списка_команд1 могут использоваться списки любых команд. Однако, чаще других используется команда test. В операторе if так же допускается две формы записи этой команды:
if test аргументы
if [аргументы]

Каждый оператор if произвольного уровня вложенности обязательно должен завершаться словом fi.
ЗАДАНИЕ:
Создать и выполнить файл с процедурой, сравнивающей передаваемый ей параметр с некоторым набором символов (паролем).
$ cat>com
if test 'param' = "$1" - сравниваются строки символов
then echo Y
else echo N
fi
*<Ctrl*D>*

$ chmod u+x com

```
$ com param
Y
$ com parm
N
$
```

ЗАДАНИЕ.

Организовать ветвление вычислительного процесса в процедуре в зависимости от значения переменной X (<10, >10, = 10).

```
if
 [$X -lt 10]
then
 echo X is less 10
else
 if
 [$X -gt 10]
 then
 echo X is greatr 10
 else
 echo X is equal to 10
 fi
fi
```

Для улучшения восприятия программ и облегчения отладки целесообразно придерживаться структурированного стиля написания программы. Каждому if должен соответствовать свой fi.

6. ПОСТРОЕНИЕ ЦИКЛОВ

Циклы обеспечивают многократное выполнение отдельных участков процедуры до достижения заданных условий.

Цикл типа while (пока true)**:**

while список_команд1
do список_команд2
done

Список_команд1 возвращает код возврата последней выполненной команды. Если условие истинно, выполняется список_- команд2, затем снова список_команд1 с целью проверки условия, а если ложно, выполнение цикла завершается. Таким образом, циклический список_команд2 выполняется до тех пор, пока условие истинно.

Пример 1.

Проверка наличия параметров при обращении к данной процедуре. Вывод на экран сообщений о наличии параметров и тексты параметров.

Текст процедуры, которой присвоено имя P1:

```
if $# –eq 0
 then echo “No param”
 else echo “Param:’;
 while test ‘$1’
 do
 echo “$1”
```

```
 shift
 done
fi
```
Результат работы процедуры:
```
$P1
No param
$P1 abc df egh
Param:
abc
df
egh
$
```

Пример 2.
Ввод строки из нескольких слов. Подсчет и вывод числа символов в каждом слове. Текст процедуры, которой присвоено имя P2:
```
echo “Input string:”
read A
set $A
while [ “$1” ]
do
 echo “$1 = `expr ”$1” : ‘.*’`”
 shift
done
```
Результат работы процедуры:
```
$P2
Input string:
df ghghhhh aqw
df = 2
ghghhhh = 7
aqw = 3
$
```

Пример 3.
Вывести на экран слово строки (поле), номер которого (переменная N) указан в параметре при обращении к процедуре, которой присвоено имя P3. Процедура запрашивает ввод строки с клавиатуры. Номер слова вводится как аргумент процедуры.

Текст процедуры P3:
```
i=1 - счетчик номеров слов в строке, формируется при каждом  выполнении цикла
N=$1 -значение первого параметра
echo "Введи строку: "
read a
set $a
while test $i -lt $N
 do
 i=`expr $i + 1` - формирование номера следующего слова
 shift
 done
echo "$N поле строки: \"$1\""
```
Пример работы процедуры P3:
```
$P3 2
```

Введи строку: aa bb cc dd
2 поле строки: "bb"
$

Цикл типа until (пока false)**:**
until список_команд1
do список_команд2
done
Логическое условие, определяемое по коду возврата списка_команд1, инвертируется, т.е. цикл выполняется до техпор, пока условие ложно.
Пример процедуры с именем P4, выполняющей заданное число циклов.
$cat>P4
X = 1 - счетчик числа циклов
until test $X -gt 10 - задано число циклов = 10
do
echo X is $X
X = `expr $X + 1`
done
*<Ctrl*D>*
$sh P4
X is 1
X is 2
.....

X is 10
$

Цикл типа for:
for имя_ переменной [in список_значений]
do список_команд
done
Переменная с указанным в команде именем заводится автоматически. Переменной присваивается значение очередного слова из списка_значений и для этого значения выполняется список_команд. Количество итераций равно количеству значений в списке, разделенных пробелами (т.е. циклы выполняются пока список не будет исчерпан).

Пример текста процедуры, печатающей в столбец список имен файлов текущего каталога.
list =`ls`
for val in $list
do
echo "$val"
done
echo end

Пример
Процедура, должна скопировать все обычные файлы из текущего каталога в каталог, который задается в качестве аргумента при обращении к данной процедуре по имени comfil. Процедура проверяет так же наличие каталога-адресата и сообщает количество скопированных файлов.

m=0 - переменная для счетчика скопированных файлов
if [-d $HOME/$1]

```
 then echo "Каталог $1 существует"
 else
 mkdir $HOME/$1 .
 echo "Каталог $1 создан"
fi
 for file in *
 do
 if [ -f "$file" ]
 then cp "$file" $HOME/$1;
 m=`expr $m + 1`
 fi
 done
echo "Число скопированных файлов: $m"
```

Выполнение процедуры:

```
$sh comfil dir1
Число скопированных файлов:....
$
```

Здесь символ * - имеет смысл <список_имен_файлов_текущего_ каталога>

Пример

Отобразить сведения о файлах текущего каталога

```
for nam in `ls`
do
 echo $ nam
done
```

Пример

Процедура PROC, выводит на экран имена файлов из текущего каталога, число символов в имени которых не превышает заданного параметром числа.

```
if [ “$1” = “” ]
then
 exit
fi
for nam in *
do
 size = `expr $nam : ‘.*’`
 if [ “$size” –le “$1” ]
 then echo “Длина имени $nam $size символа”
 fi
done
```

Вывод содержимого текущего каталога для проверки работы процедуры:

```
$ ls –l
total 4
drwxrwxrwx 2 lev lev 1024 Feb 7 18:18 dir1
-rw-rw-r-- 1 lev lev 755 Feb 7 18:24 out
-rwxr-xr-x 1 lev lev 115 Feb 7 17:55 f1
-rwxr-xr-x 1 lev lev 96 Feb 7 18:00 f2
$
```

Результаты работы процедуры:

```
$ PROC 2
 Длина имени f1 2 символа
```

Длина имени f2 2 символа
$PROC 3
Длина имени out 3 символа
Длина имени f1 2 символа
Длина имени f2 2 символа
$

Пример.
Процедура с именем PR выводит на экран из указанного параметром подкаталога имена файлов с указанием их типа.

```
cd $1
for fil in *
do
 If [ -d $fil]
 then echo "$fil – catalog"
 else echo "$fil – file"
 fi
done
```

Вывод содержимого подкаталога для проверки работы процедуры:

```
$ ls –l pdir
total 4
drwxrwxrwx 2 lev lev 1024 Feb 7 18:18 dir1
-rw-rw-r-- 1 lev lev 755 Feb 7 18:24 out
-rwxr-xr-x 1 lev lev 115 Feb 7 17:55 f1
-rwxr-xr-x 1 lev lev 96 Feb 7 18:00 f2
```

Результаты работы процедуры:

```
$ PR pdir
dir1 – catalog
out – file
f1 – file
f2 – file
$
```

Ветвление по многим направлениям **case**. Команда **case** обеспечивает ветвление по многим направлениям в зависимости от значений аргументов команды. Формат:

case <string> in
s1) <list1>;;
s2) <list2>;;
.
.
sn) <listn>;;
***) <list>**
esac

3десь list1, list2 ... listn - список команд. Производится сравнение шаблона string с шаблонами s1, s2 ... sk ... sn. При совпадении выполняется список команд, стоящий между текущим шаблоном sk и соответствующими знаками ;;.

Пример:

```
echo -n 'Please, write down your age'
read age
case $age in
```

```
 test $age -le 20) echo 'you are so young' ;;
 test $age -le 40) echo 'you are still young' ;;
 test $age -le 70) echo 'you are too young' ;;
 *)echo 'Please, write down once more'
esac
```

В конце текста помещена звездочка * на случай неправильного ввода числа.

Некоторые дополнительные команды, которые могут быть использованы в процедурах:
sleep t - приостанавливает выполнение процесса на t секунд

Пример.
Бесконечная (циклическая) процедура выводит каждые пять секунд сообщение в указанный файл fil.

```
while true
do
 echo "текст_сообщения">fil
 sleep 5
done
```

Примечание: вместо файла (или экрана) может быть использовано фиктивное устройство /dev/null (например для отладки процедуры).
Примечание: в процедуре реализуется бесконечный цикл. Для ограничения числа циклов надо предусмотреть счетчик циклов (см. выше) или пркратить выполнение процесса процедуры с помощью команды управления процессами - $kill (см. ниже).

exit [n] - прекращение выполнения процедуры с кодом завершения [n] или с кодом завершения последней выполненной команды.
В качестве "n" можно использовать любое число, например, для идентификации выхода из сложной процедуры, имеющей несколько причин завершения выполнения.

7. Практическое задание

Создать сценарий реализующий в консольном режиме диалог с пользователем в виде меню. Сценарий должен выполняться циклически пока не выбран пункт «Выход». Первый пункт меню должен выводить информацию о создателе (ФИО, группа) и краткое описание выполняемых действий, второй пункт меню должен вычислять математическое выражение 2.1, а остальные пункты реализуют действия указанные в таблице в соответствии с вариантом. Все параметры задаются в результате диалога с пользователем.

$$x = (№Компьютера + №По\_журналу) \cdot Возраст \quad (2.1)$$

Отчет должен содержать **краткие** теоретические сведения о **использованных** командах и операторах.

Вариант	Задание
1	А)Проверить существует ли папка в указанном месте и если нет создать её. Б) архивации файлов в заданном каталоге, с созданием отдельного архива для каждого файла и удалением заархивированных.
2	А)Копирование файлов с указанного места в заданное. Б). очистки подкаталога с подтверждением.
3	А)Удаление файлов заданного расширения в заданной папке. Б) Разработать пакетный файл для установки даты и времени (параметры – в командной строке)
4	А)Удаление содержимого заданной папки, с помещением всех удаленных объектов в папку tmp. Б) Проверить существование указанного в параметре файла и выдать сообщение о результате

	поиска
5	А)Создание файла со списком папок в указанном месте. Б)Проверить существует ли заданный файл и доступен ли он для записи
6	А)Копирование файлов заданного расширения с указанного места в папку «*BackUp*»в каталоге tmp. Б) Запрос и ввод имени пользователя, сравнение с текущим логическим именем пользователя и вывод сообщения: верно/неверно.
7	А)Перенос файлов заданного расширения с указанного места в папку «*BackUp*»в каталоге tmp. Б) Проверить существует ли заданный файл и доступен ли он для исполнения, если нет установить на него атрибуты исполняемого
8	А)Создание файла со списком папок в указанном месте. Б)определить является ли заданный каталог пустым , если нет определить количество файлов в нем.
9	А)Создание файла со списком файлов с заданным расширением в указанном месте. Б) Разработать пакетный файл для перехода в заданный, если он существует и его архивирования
10	А)Перенос файлов с указанного места в заданное. Б) Разработать пакетный файл для перехода студента в личный каталог. В специальный файл (logon.data) в домашней папке записывается дата и время входа в систему.
11	А) Проверить существует ли папка в указанном месте и если да переименовать её. Б) скопировать файлы из заданного каталога в папку tmp/Имя пользователя, с добавив расширение .bak
12	А)Архивирование заданной папки с копированием архива в папку *«BackUp»* в каталоге tmp Б) В заданной папке вывести все файлы с длиной больше заданного значения
13	А)Архивирование файлов заданного расширения в заданной папке с копированием архива в папку *«BackUp»* в каталоге tmp Б) В заданной папке определить файл с самым длинным именем
14	А). Разработать пакетный файл для построения системы студенческих каталогов с запросом на создание каталогов требуемых курсов, групп и запросом максимального числа пользователей в группе Б). Переход в другой каталог, формирование файла с листингом каталога и возвращение в исходный каталог.
15	А)Копирование всех фалов с заданным расширением с указанного места в указанную папку. Б)Сохранение сведений о файлах домашнего каталога в файле «текущая дата.txt», в домашней папке History.

Лабораторная работа №5

Планирование процессов

1. Теоретические сведения

1. Алгоритмы планирования

Существует достаточно большой набор разнообразных алгоритмов планирования, которые предназначены для достижения различных целей и эффективны для разных классов задач. Многие из них могут использоваться на нескольких уровнях планирования. В этом разделе мы рассмотрим некоторые наиболее употребительные алгоритмы применительно к процессу кратковременного планирования.

1.1 First-Come, First-Served (FCFS)

Простейшим алгоритмом планирования является алгоритм, который принято обозначать аббревиатурой FCFS по первым буквам его английского названия – First-Come, First-Served (первым пришел, первым обслужен). Представим себе, что процессы, находящиеся в состоянии готовность, выстроены в очередь. Когда процесс переходит в состояние готовность, он, а точнее, ссылка на его PCB помещается в конец этой очереди. Выбор нового процесса для исполнения осуществляется из начала очереди с удалением оттуда ссылки на его PCB. Очередь подобного типа имеет в программировании специальное наименование – FIFO1), сокращение от First In, First Out (первым вошел, первым вышел).

Такой алгоритм выбора процесса осуществляет невытесняющее планирование. Процесс, получивший в свое распоряжение процессор, занимает его до истечения текущего CPU burst. После этого для выполнения выбирается новый процесс из начала очереди.

Таблица 1.1.

Процесс	p0	p1	p2
Продолжительность очередного CPU burst	13	4	1

Преимуществом алгоритма FCFS является легкость его реализации, но в то же время он имеет и много недостатков. Рассмотрим следующий пример. Пусть в состоянии готовность находятся три процесса p0, p1 и p2, для которых известны времена их очередных CPU burst. Эти времена приведены в таблице 1.1. в некоторых условных единицах. Для простоты будем полагать, что вся деятельность процессов ограничивается использованием только одного промежутка CPU burst, что процессы не совершают операций ввода-вывода и что время переключения контекста так мало, что им можно пренебречь.

Если процессы расположены в очереди процессов, готовых к исполнению, в порядке p0, p1, p2, то картина их выполнения выглядит так, как показано на рисунке 1.1. Первым для выполнения выбирается процесс p0, который получает процессор на все время своего CPU burst, т. е. на 13 единиц времени. После его окончания в состояние исполнение переводится процесс p1, он занимает процессор на 4 единицы времени. И, наконец, возможность работать получает процесс p2. Время ожидания для процесса p0 составляет 0 единиц времени, для процесса p1 – 13 единиц, для процесса p2 – 13 + 4 = 17 единиц. Таким образом, среднее время ожидания в этом случае – (0 + 13 + 17)/3 = 10 единиц времени. Полное время выполнения для процесса p0 составляет 13 единиц времени, для процесса p1 – 13 + 4 = 17 единиц, для процесса p2 – 13 + 4 + 1 = 18 единиц. Среднее полное время выполнения оказывается равным (13 + 17 + 18)/3 = 16 единицам времени.

Рис. 1.1. Выполнение процессов при порядке p0,p1,p2

Если те же самые процессы расположены в порядке p2, p1, p0, то картина их выполнения будет соответствовать рисунку 1.2. Время ожидания для процесса p0 равняется 5 единицам времени, для процесса p1 – 1 единице, для процесса p2 – 0 единиц. Среднее время ожидания составит (5 + 1 + 0)/3 = 2 единицы времени. Это в 5 (!) раз меньше, чем в предыдущем случае. Полное время выполнения для процесса p0 получается равным 18 единицам времени, для процесса p1 – 5 единицам, для процесса p2 – 1 единице. Среднее полное время выполнения составляет (18 + 5 + 1)/3 = 8 единиц времени, что почти в 2 раза меньше, чем при первой расстановке процессов.

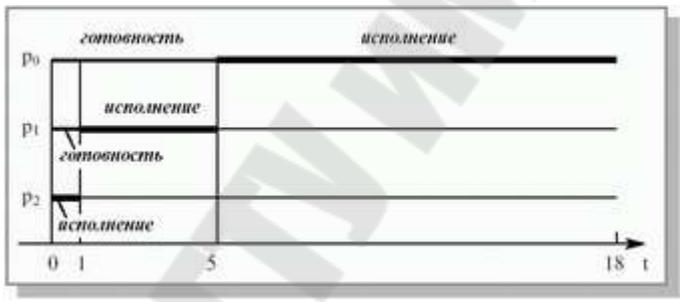

Рис. 1.2. Выполнение процессов при порядке p2, p1, p0

Как мы видим, среднее время ожидания и среднее полное время выполнения для этого алгоритма существенно зависят от порядка расположения процессов в очереди. Если у нас есть процесс с длительным CPU burst, то короткие процессы, перешедшие в состояние готовность после длительного процесса, будут очень долго ждать начала выполнения. Поэтому алгоритм FCFS практически неприменим для систем разделения времени – слишком большим получается среднее время отклика в интерактивных процессах.

1.2 Round Robin (RR)

Модификацией алгоритма FCFS является алгоритм, получивший название Round Robin (Round Robin – это вид детской карусели в США) или сокращенно RR. По сути дела, это тот же самый алгоритм, только реализованный в режиме вытесняющего планирования. Можно представить себе все множество готовых процессов организованным циклически – процессы сидят на карусели. Карусель вращается так, что каждый процесс находится около процессора небольшой фиксированный квант времени, обычно 10 – 100 миллисекунд (см. рис. 1.3.). Пока процесс находится рядом с процессором, он получает процессор в свое распоряжение и может исполняться.

Рис. 1.3. Процессы на карусели

Реализуется такой алгоритм так же, как и предыдущий, с помощью организации процессов, находящихся в состоянии готовность, в очередь FIFO. Планировщик выбирает для очередного исполнения процесс, расположенный в начале очереди, и устанавливает таймер для генерации прерывания по истечении определенного кванта времени. При выполнении процесса возможны два варианта.

- Время непрерывного использования процессора, необходимое процессу (остаток текущего CPU burst), меньше или равно продолжительности кванта времени. Тогда процесс по своей воле освобождает процессор до истечения кванта времени, на исполнение поступает новый процесс из начала очереди, и таймер начинает отсчет кванта заново.
- Продолжительность остатка текущего CPU burst процесса больше, чем квант времени. Тогда по истечении этого кванта процесс прерывается таймером и помещается в конец очереди процессов, готовых к исполнению, а процессор выделяется для использования процессу, находящемуся в ее начале.

Рассмотрим предыдущий пример с порядком процессов p0, p1, p2 и величиной кванта времени равной 4. Выполнение этих процессов иллюстрируется таблицей 1.2. Обозначение "И" используется в ней для процесса, находящегося в состоянии исполнение, обозначение "Г" – для процессов в состоянии готовность, пустые ячейки соответствуют завершившимся процессам. Состояния процессов показаны на протяжении соответствующей единицы времени, т. е. колонка с номером 1 соответствует промежутку времени от 0 до 1.

Таблица 1.2.

Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
p0	И	И	И	И	Г	Г	Г	Г	Г	И	И	И	И	И	И	И	И	И
p1	Г	Г	Г	Г	И	И	И	И										
p2	Г	Г	Г	Г	Г	Г	Г	Г	И									

Первым для исполнения выбирается процесс p0. Продолжительность его CPU burst больше, чем величина кванта времени, и поэтому процесс исполняется до истечения кванта, т. е. в течение 4 единиц времени. После этого он помещается в конец очереди готовых к исполнению процессов, которая принимает вид p1, p2, p0. Следующим начинает выполняться процесс p1. Время его исполнения совпадает с величиной выделенного кванта, поэтому процесс работает до своего завершения. Теперь очередь процессов в состоянии готовность состоит из двух процессов, p2 и p0. Процессор выделяется процессу p2. Он завершается до истечения отпущенного ему процессорного времени, и очередные кванты отмеряются процессу p0 – единственному не закончившему к этому моменту свою работу. Время ожидания для процесса p0 (количество символов "Г" в соответствующей строке) составляет 5 единиц времени, для процесса p1 – 4 единицы времени, для процесса p2 – 8 единиц времени. Таким образом, среднее время ожидания для этого алгоритма

получается равным (5 + 4 + 8)/3 = 5,6(6) единицы времени. Полное время выполнения для процесса p0 (количество непустых столбцов в соответствующей строке) составляет 18 единиц времени, для процесса p1 – 8 единиц, для процесса p2 – 9 единиц. Среднее полное время выполнения оказывается равным (18 + 8 + 9)/3 = 11,6(6) единицы времени.

Легко увидеть, что среднее время ожидания и среднее полное время выполнения для обратного порядка процессов не отличаются от соответствующих времен для алгоритма FCFS и составляют 2 и 6 единиц времени соответственно.

На производительность алгоритма RR сильно влияет величина кванта времени. Рассмотрим тот же самый пример с порядком процессов p0, p1, p2 для величины кванта времени, равной 1 (см. табл. 1.3.). Время ожидания для процесса p0 составит 5 единиц времени, для процесса p1 – тоже 5 единиц, для процесса p2 – 2 единицы. В этом случае среднее время ожидания получается равным (5 + 5 + 2)/3 = 4 единицам времени. Среднее полное время исполнения составит (18 + 9 + 3)/3 = 10 единиц времени.

Таблица 1.3.

Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
p0	И	Г	Г	И	Г	И	Г	И	Г	И	И	И	И	И	И	И	И	И
p1	Г	И	Г	Г	И	Г	И	Г	И									
p2	Г	Г	И															

При очень больших величинах кванта времени, когда каждый процесс успевает завершить свой CPU burst до возникновения прерывания по времени, алгоритм RR вырождается в алгоритм FCFS. При очень малых величинах создается иллюзия того, что каждый из n процессов работает на собственном виртуальном процессоре с производительностью ~ 1/n от производительности реального процессора. Правда, это справедливо лишь при теоретическом анализе при условии пренебрежения временами переключения контекста процессов. В реальных условиях при слишком малой величине кванта времени и, соответственно, слишком частом переключении контекста накладные расходы на переключение резко снижают производительность системы.

1.3 Shortest-Job-First (SJF)

При рассмотрении алгоритмов FCFS и RR мы видели, насколько существенным для них является порядок расположения процессов в очереди процессов, готовых к исполнению. Если короткие задачи расположены в очереди ближе к ее началу, то общая производительность этих алгоритмов значительно возрастает. Если бы мы знали время следующих CPU burst для процессов, находящихся в состоянии готовность, то могли бы выбрать для исполнения не процесс из начала очереди, а процесс с минимальной длительностью CPU burst. Если же таких процессов два или больше, то для выбора одного из них можно использовать уже известный нам алгоритм FCFS. Квантование времени при этом не применяется. Описанный алгоритм получил название "кратчайшая работа первой" или Shortest Job First (SJF).

SJF-алгоритм краткосрочного планирования может быть как вытесняющим, так и невытесняющим. При невытесняющем SJF-планировании процессор предоставляется избранному процессу на все необходимое ему время, независимо от событий, происходящих в вычислительной системе. При вытесняющем SJF-планировании учитывается появление новых процессов в очереди готовых к исполнению (из числа вновь родившихся или разблокированных) во время работы выбранного процесса. Если CPU burst нового процесса меньше, чем остаток CPU burst у исполняющегося, то исполняющийся процесс вытесняется новым.

Рассмотрим пример работы невытесняющего алгоритма SJF. Пусть в состоянии готовность находятся четыре процесса, p0, p1, p2 и p3, для которых известны времена их очередных CPU burst. Эти времена приведены в таблице 1.4. Как и прежде, будем полагать, что вся деятельность процессов ограничивается использованием только одного промежутка CPU burst, что процессы не совершают операций ввода-вывода и что временем переключения контекста можно пренебречь.

Таблица 1.4.

Процесс	p0	p1	p2	p3
Продолжительность очередного CPU burst	5	3	7	1

При использовании невытесняющего алгоритма SJF первым для исполнения будет выбран процесс p3, имеющий наименьшее значение продолжительности очередного CPU burst. После его завершения для исполнения выбирается процесс p1, затем p0 и, наконец, p2. Эта картина отражена в таблице 1.5.

Таблица 1.5.

Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
p0	Г	Г	Г	Г	И	И	И	И	И							
p1	Г	И	И	И												
p2	Г	Г	Г	Г	Г	Г	Г	Г	Г	И	И	И	И	И	И	И
p3	И															

Как мы видим, среднее время ожидания для алгоритма SJF составляет (4 + 1 + 9 + 0)/4 = 3,5 единицы времени. Легко посчитать, что для алгоритма FCFS при порядке процессов p0, p1, p2, p3 эта величина будет равняться (0 + 5 + 8 + 15)/4 = 7 единицам времени, т. е. будет в два раза больше, чем для алгоритма SJF. Можно показать, что для заданного набора процессов (если в очереди не появляются новые процессы) алгоритм SJF является оптимальным с точки зрения минимизации среднего времени ожидания среди класса невытесняющих алгоритмов.

Для рассмотрения примера вытесняющего SJF планирования мы возьмем ряд процессов p0, p1, p2 и p3 с различными временами CPU burst и различными моментами их появления в очереди процессов, готовых к исполнению (см. табл. 1.6.).

Таблица 1.6.

Процесс	Время появления в очереди очередного CPU burst	Продолжительность
p0	0	6
p1	2	2
p2	6	7
p3	0	5

В начальный момент времени в состоянии готовность находятся только два процесса, p0 и p3. Меньшее время очередного CPU burst оказывается у процесса p3, поэтому он и выбирается для исполнения (см. таблицу 1.7.). По прошествии 2 единиц времени в систему поступает процесс p1. Время его CPU burst меньше, чем остаток CPU burst у процесса p3, который вытесняется из состояния исполнение и переводится в состояние готовность. По прошествии еще 2 единиц времени процесс p1 завершается, и для исполнения вновь выбирается процесс p3. В момент времени t = 6 в очереди процессов, готовых к исполнению, появляется процесс p2, но поскольку ему для работы нужно 7 единиц времени, а процессу p3 осталось трудиться всего 1 единицу времени, то процесс p3 остается в состоянии исполнение. После его завершения в момент времени t = 7 в очереди

находятся процессы p0 и p2, из которых выбирается процесс p0. Наконец, последним получит возможность выполняться процесс p2.

Таблица 1.7.

Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
p0	Г	Г	Г	Г	Г	Г	Г	И	И	И	И	И	И							
p1			И	И																
p2							Г	Г	Г	Г	Г	Г	Г	И	И	И	И	И	И	И
p3	И	И	Г	Г	И	И	И													

Основную сложность при реализации алгоритма SJF представляет невозможность точного знания продолжительности очередного CPU burst для исполняющихся процессов. В пакетных системах количество процессорного времени, необходимое заданию для выполнения, указывает пользователь при формировании задания.

1.4 Приоритетное планирование

Алгоритмы SJF и гарантированного планирования представляют собой частные случаи приоритетного планирования. При приоритетном планировании каждому процессу присваивается определенное числовое значение – приоритет, в соответствии с которым ему выделяется процессор. Процессы с одинаковыми приоритетами планируются в порядке FCFS. Для алгоритма SJF в качестве такого приоритета выступает оценка продолжительности следующего CPU burst. Чем меньше значение этой оценки, тем более высокий приоритет имеет процесс. Для алгоритма гарантированного планирования приоритетом служит вычисленный коэффициент справедливости. Чем он меньше, тем больше у процесса приоритет.

Алгоритмы назначения приоритетов процессов могут опираться как на внутренние параметры, связанные с происходящим внутри вычислительной системы, так и на внешние по отношению к ней. К внутренним параметрам относятся различные количественные и качественные характеристики процесса такие как: ограничения по времени использования процессора, требования к размеру памяти, число открытых файлов и используемых устройств ввода-вывода, отношение средних продолжительностей I/O burst к CPU burst и т. д. Алгоритмы SJF и гарантированного планирования используют внутренние параметры. В качестве внешних параметров могут выступать важность процесса для достижения каких-либо целей, стоимость оплаченного процессорного времени и другие политические факторы. Высокий внешний приоритет может быть присвоен задаче лектора или того, кто заплатил $100 за работу в течение одного часа.

Планирование с использованием приоритетов может быть как вытесняющим, так и невытесняющим. При вытесняющем планировании процесс с более высоким приоритетом, появившийся в очереди готовых процессов, вытесняет исполняющийся процесс с более низким приоритетом. В случае невытесняющего планирования он просто становится в начало очереди готовых процессов. Давайте рассмотрим примеры использования различных режимов приоритетного планирования.

Пусть в очередь процессов, находящихся в состоянии готовность, поступают те же процессы, что и в примере для вытесняющего алгоритма SJF, только им дополнительно еще присвоены приоритеты (см. табл. 1.8.). В вычислительных системах не существует определенного соглашения, какое значение приоритета – 1 или 4 считать более приоритетным. Во избежание путаницы, во всех наших примерах мы будем предполагать, что большее значение соответствует меньшему приоритету, т. е. наиболее приоритетным в нашем примере является процесс p3, а наименее приоритетным – процесс p0.

Таблица 1.8.

Процесс	Время появления в очереди	Продолжительность очередного CPU burst	Приоритет
p0	0	6	4
p1	2	2	3
p2	6	7	2
p3	0	5	1

Как будут вести себя процессы при использовании невытесняющего приоритетного планирования? Первым для выполнения в момент времени t = 0 выбирается процесс p3, как обладающий наивысшим приоритетом. После его завершения в момент времени t = 5 в очереди процессов, готовых к исполнению, окажутся два процесса p0 и p1. Больший приоритет из них у процесса p1, он и начнет выполняться (см. табл. 1.9.). Затем в момент времени t = 8 для исполнения будет избран процесс p2, и лишь потом – процесс p0.

Таблица 1.9.

Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
p0	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	И	И	И	И	И	И
p1			Г	Г	Г	И	И													
p2							Г	И	И	И	И	И	И	И						
p3	И	И	И	И	И															

Иным будет предоставление процессора процессам в случае вытесняющего приоритетного планирования (см. табл. 1.10.). Первым, как и в предыдущем случае, начнет исполняться процесс p3, а по его окончании – процесс p1. Однако в момент времени t = 6 он будет вытеснен процессом p2 и продолжит свое выполнение только в момент времени t = 13. Последним, как и раньше, будет исполняться процесс p0.

Таблица 1.10.

Время	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
p0	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	Г	И	И	И	И	И	И
p1			Г	Г	Г	И	Г	Г	Г	Г	Г	Г	Г	И						
p2							И	И	И	И	И	И	И							
p3	И	И	И	И	И															

В рассмотренном выше примере приоритеты процессов с течением времени не изменялись. Такие приоритеты принято называть статическими. Механизмы статической приоритетности легко реализовать, и они сопряжены с относительно небольшими издержками на выбор наиболее приоритетного процесса. Однако статические приоритеты не реагируют на изменения ситуации в вычислительной системе, которые могут сделать желательной корректировку порядка исполнения процессов. Более гибкими являются динамические приоритеты процессов, изменяющие свои значения по ходу исполнения процессов. Начальное значение динамического приоритета, присвоенное процессу, действует в течение лишь короткого периода времени, после чего ему назначается новое, более подходящее значение. Изменение динамического приоритета процесса является единственной операцией над процессами, которую мы до сих пор не рассмотрели. Как правило, изменение приоритета процессов проводится согласованно с совершением каких-либо других операций: при рождении нового процесса, при разблокировке или блокировании процесса, по истечении определенного кванта времени или по завершении процесса. Примерами алгоритмов с динамическими приоритетами являются алгоритм SJF и алгоритм гарантированного планирования. Схемы с динамической приоритетностью гораздо сложнее в реализации и связаны с большими издержками по сравнению со

статическими схемами. Однако их использование предполагает, что эти издержки оправдываются улучшением работы системы.

2. Индивидуальные задания

2.1 Не вытесняющие алгоритмы планирования процессов

Выполнить различные алгоритмы планирований – ***First-Come, First-Served (FCFS)*** (прямой и обратный), ***Round Robin (RR)***, ***Shortest-Job-First (SJF)*** (не вытесняющий), ***Shortest-Job-First (SJF)*** (не вытесняющий приоритетный) для данных приведенных в таблице 2.1 в соответствии со своим вариантом (**номер по журналу**). Вычислить полное время выполнения все процессов и каждого в отдельности, время ожидание для каждого процесса. Рассчитать среднее время выполнения процесса и среднее время ожидания. Результаты оформить в виде таблиц иллюстрирующих работу процессов.

2.2 Вытесняющие алгоритмы планирования процессов

Выполнить различные алгоритмы планирований – ***Shortest-Job-First (SJF)*** (вытесняющий) и ***Shortest-Job-First (SJF)*** (приоритетный) для данных приведенных в таблице 2.1 в соответствии со своим вариантом. Вычислить полное время выполнения все процессов и каждого в отдельности, время ожидание для каждого процесса. Рассчитать среднее время выполнения процесса и среднее время ожидания. Результаты оформить в виде таблиц иллюстрирующих работу процессов.

Таблица 2.1 Варианты заданий

Вариант	Продолжительности процессов	Время появления в очереди	Приоритеты процессов
1	P0 – 5; P1 – 8; P2 – 2; P3 – 4;	P0 – 0; P1 – 3; P2 – 1; P3 – 4;	P0 – 2; P1 – 1; P2 – 3; P3 – 3;
2	P0 – 10; P1 – 3; P2 – 3; P3 – 7;	P0 – 2; P1 –0 ; P2 – 4; P3 –5;	P0 – 1; P1 – 3; P2 – 3; P3 – 2;
3	P0 – 5; P1 – 8; P2 – 2; P3 – 4;	P0 – 3; P1 – 1; P2 – 3; P3 –0;	P0 – 1; P1 – 2; P2 – 3; P3 – 1;
4	P0 – 2; P1 – 5; P2 – 9; P3 – 3;	P0 – 0; P1 – 4; P2 – 6; P3 – 8;	P0 – 1; P1 – 3; P2 – 3; P3 – 2;
5	P0 – 7; P1 – 8; P2 – 3; P3 – 4;	P0 – 4; P1 – 0; P2 – 3; P3 – 5;	P0 – 1; P1 – 2; P2 – 3; P3 – 4;
6	P0 – 9; P1 – 2; P2 – 2; P3 – 3;	P0 – 3; P1 – 0; P2 – 0; P3 – 4;	P0 – 3; P1 – 1; P2 – 3; P3 – 3;
7	P0 – 10; P1 – 2; P2 – 3; P3 – 1;	P0 – 0; P1 – 4; P2 – 3; P3 – 0;	P0 – 1; P1 – 3; P2 – 2; P3 – 3;
8	P0 – 2; P1 – 3; P2 – 1; P3 – 8;	P0 – 4; P1 – 4; P2 – 0; P3 – 0;	P0 – 3; P1 – 1; P2 – 3; P3 – 4;
9	P0 – 3; P1 – 5; P2 – 2; P3 – 7;	P0 – 3; P1 – 0; P2 – 1; P3 – 7;	P0 – 1; P1 – 3; P2 – 4; P3 – 1;
10	P0 – 7; P1 – 1; P2 – 4; P3 – 6;	P0 – 1; P1 – 2; P2 – 4; P3 – 0;	P0 – 2; P1 – 1; P2 – 3; P3 – 4;
11	P0 – 3; P1 – 1; P2 – 5; P3 – 6;	P0 – 3; P1 – 4; P2 – 4; P3 – 0;	P0 – 1; P1 – 1; P2 – 3; P3 – 3;
12	P0 – 8; P1 – 3; P2 – 10; P3 – 2;	P0 – 8; P1 – 4; P2 – 0; P3 – 3;	P0 – 2; P1 – 2; P2 – 1; P3 – 4;
13	P0 – 3; P1 – 7; P2 – 1; P3 – 3;	P0 – 4; P1 – 0; P2 – 0; P3 – 0;	P0 – 4; P1 – 3; P2 – 3; P3 – 2;
14	P0 – 4; P1 – 7; P2 – 6; P3 – 3;	P0 – 4; P1 – 4; P2 – 0; P3 – 7;	P0 – 1; P1 – 4; P2 – 3; P3 – 3;
15	P0 – 1; P1 – 3; P2 – 2; P3 – 7;	P0 – 0; P1 – 4; P2 – 6; P3 – 4;	P0 – 2; P1 – 1; P2 – 4; P3 – 1;
16	P0 – 5; P1 – 3; P2 – 4; P3 – 2;	P0 – 0; P1 – 4; P2 – 8; P3 – 4;	P0 – 3; P1 – 1; P2 – 3; P3 – 1;
17	P0 – 2; P1 – 4; P2 – 1; P3 – 6;	P0 – 3; P1 – 0; P2 – 6; P3 – 7;	P0 – 3; P1 – 4; P2 – 1; P3 – 3;
18	P0 – 3; P1 – 6; P2 – 3; P3 – 1;	P0 – 4; P1 – 1; P2 – 1; P3 – 0;	P0 – 1; P1 – 3; P2 – 3; P3 – 1;
19	P0 – 10; P1 – 2; P2 – 5; P3 – 3;	P0 – 11; P1 – 7; P2 – 3; P3 – 0;	P0 – 3; P1 – 3; P2 – 1; P3 – 2;
20	P0 – 8; P1 – 3; P2 – 2; P3 –3 ;	P0 – 0; P1 – 4; P2 – 7; P3 – 3;	P0 – 2; P1 – 2; P2 – 3; P3 – 1;
21	P0 – 3; P1 – 6; P2 – 4; P3 – 2;	P0 – 7; P1 – 0; P2 – 5; P3 – 3;	P0 – 4; P1 – 2; P2 – 4; P3 – 1;
22	P0 – 7; P1 – 6; P2 – 5; P3 – 3;	P0 – 0; P1 – 4; P2 – 0; P3 – 4;	P0 – 1; P1 – 3; P2 – 4; P3 – 2;
23	P0 – 3; P1 – 7; P2 – 4; P3 – 2;	P0 – 6; P1 – 0; P2 – 0; P3 – 7;	P0 – 3; P1 – 1; P2 – 2; P3 – 1;
24	P0 – 2; P1 – 5; P2 – 1; P3 – 7;	P0 – 1; P1 – 8; P2 – 0; P3 – ;	P0 – 2; P1 – 1; P2 – 2; P3 – 3;
25	P0 – 3; P1 – 4; P2 – 7; P3 – 1;	P0 – 3; P1 – 4; P2 – 4; P3 – 0;	P0 – 1; P1 – 1; P2 – 2; P3 – 2;

Примечания:

*Для **Round Robin (RR)** величина кванта времени 3 для всех вариантов.*

Для приоритетных алгоритмов меньшее значение соответствует более высокому приоритету.

Сравнить полученные результаты и сделать выводы.

Лабораторная работа №6

Синхронизация процессов

1. Теоретические сведения

1.1 Синхронизация процессов

Синхронизация процессов имеет важное значение, при использовании различными процессами одних и тех же ресурсов. Введем некоторые понятия. Под **активностями** мы будем понимать последовательное выполнение ряда операций, направленных на достижение определенной цели. Неделимые операции могут иметь внутренние невидимые действия. Мы же называем их неделимыми потому, что считаем выполняемыми за раз, без прерывания деятельности.
Пусть имеется две активности

```
P:   a b c
Q:   d e f
```

где a, b, c, d, e, f – атомарные операции. При последовательном выполнении активностей мы получаем такую последовательность атомарных действий:

```
PQ: a b c d e f
```

Что произойдет при исполнении этих активностей псевдопараллельно, в режиме разделения времени? Активности могут расслоиться на неделимые операции с различным чередованием, то есть может произойти то, что на английском языке принято называть словом interleaving. Возможные варианты чередования:

```
a b c d e f
a b d c e f
a b d e c f
a b d e f c
a d b c e f
......
d e f a b c
```

Атомарные операции активностей могут чередоваться всевозможными различными способами с сохранением порядка расположения внутри активностей. Так как псевдопараллельное выполнение двух активностей приводит к чередованию их неделимых операций, результат псевдопараллельного выполнения может отличаться от результата последовательного выполнения. Рассмотрим пример. Пусть у нас имеется две активности P и Q, состоящие из двух атомарных операций каждая:

```
P:  x=2          Q:  x=3
    y=x-1            y=x+1
```

Что мы получим в результате их псевдопараллельного выполнения, если переменные x и y являются для активностей общими? Очевидно, что возможны четыре разных набора значений для пары (x, y): (3, 4), (2, 1), (2, 3) и (3, 2). . Мы будем говорить, что набор активностей (например, программ) **детерминирован**, если всякий раз при псевдопараллельном исполнении для одного и того же набора входных данных он дает **одинаковые** выходные данные. В противном случае он **недетерминирован**. Выше приведен пример недетерминированного набора программ.
Можно ли до получения результатов определить, является ли набор активностей детерминированным или нет? Для этого существуют достаточные условия Бернстайна.

Введем наборы входных и выходных переменных программы. Для каждой атомарной операции наборы входных и выходных переменных – это наборы переменных, которые атомарная операция считывает и записывает. Набор входных переменных программы R(P) (R от слова read) суть объединение наборов входных переменных для всех ее неделимых действий. Аналогично, набор выходных переменных программы W(P) (W от слова write) суть объединение наборов выходных переменных для всех ее неделимых действий. Например, для программы

```
P:   x=u+v
     y=x*w
```

получаем R(P) = {u, v, x, w}, W(P) = {x, y}. Заметим, что переменная x присутствует как в R(P), так и в W(P).

Теперь сформулируем условия Бернстайна.

Если для двух данных активностей P и Q:

1. пересечение W(P) и W(Q) пусто;
2. пересечение W(P) с R(Q) пусто;
3. пересечение R(P) и W(Q) пусто,

тогда выполнение P и Q детерминировано.

Если эти условия не соблюдены, возможно, параллельное выполнение P и Q детерминировано, а может быть, и нет.

Условия Бернстайна информативны, но слишком жестки. По сути дела, они требуют практически невзаимодействующих процессов. А нам хотелось бы, чтобы детерминированный набор образовывали активности, совместно использующие информацию и обменивающиеся ею. Для этого нам необходимо ограничить число возможных чередований атомарных операций, исключив некоторые чередования с помощью механизмов синхронизации выполнения программ, обеспечив тем самым упорядоченный доступ программ к некоторым данным.

Задачу упорядоченного доступа к разделяемым данным (устранение race condition) в том случае, когда нам не важна его очередность, можно решить, если обеспечить каждому процессу эксклюзивное право доступа к этим данным. Каждый процесс, обращающийся к разделяемым ресурсам, исключает для всех других процессов возможность одновременного общения с этими ресурсами, если это может привести к недетерминированному поведению набора процессов. Такой прием называется взаимоисключением (mutual exclusion). Если очередность доступа к разделяемым ресурсам важна для получения правильных результатов, то одними взаимоисключениями уже не обойтись, нужна взаимосинхронизация поведения программ.

1.2 Критическая секция

Важным понятием при изучении способов синхронизации процессов является понятие критической секции (critical section) программы. Критическая секция – это часть программы, исполнение которой может привести к возникновению race condition для определенного набора программ. Чтобы исключить эффект гонок по отношению к некоторому ресурсу, необходимо организовать работу так, чтобы в каждый момент времени только один процесс мог находиться в своей критической секции, связанной с этим ресурсом. Иными словами, необходимо обеспечить реализацию взаимоисключения для критических секций программ. Реализация взаимоисключения для критических секций программ с практической точки зрения означает, что по отношению к другим процессам,

участвующим во взаимодействии, критическая секция начинает выполняться как атомарная операция.
В общем случае структура процесса, участвующего во взаимодействии, может быть представлена следующим образом:

```
while (some condition) {
   entry section
      critical section
   exit section
      remainder section
}
```

Здесь под remainder section понимаются все атомарные операции, не входящие в критическую секцию.

Требования, предъявляемые к алгоритмам:

1. задача должна быть решена чисто программным способом на обычной машине, не имеющей специальных команд взаимоисключения. При этом предполагается, что основные инструкции языка программирования (такие примитивные инструкции, как load, store, test) являются атомарными операциями.
2. Не должно существовать никаких предположений об относительных скоростях выполняющихся процессов или числе процессоров, на которых они исполняются.
3. Если процесс Pi исполняется в своем критическом участке, то не существует никаких других процессов, которые исполняются в соответствующих критических секциях. Это условие получило название условия взаимоисключения (mutual exclusion).
4. Процессы, которые находятся вне своих критических участков и не собираются входить в них, не могут препятствовать другим процессам входить в их собственные критические участки. Если нет процессов в критических секциях и имеются процессы, желающие войти в них, то только те процессы, которые не исполняются в remainder section, должны принимать решение о том, какой процесс войдет в свою критическую секцию. Такое решение не должно приниматься бесконечно долго. Это условие получило название условия прогресса (progress).
5. Не должно возникать неограниченно долгого ожидания для входа одного из процессов в свой критический участок. От того момента, когда процесс запросил разрешение на вход в критическую секцию, и до того момента, когда он это разрешение получил, другие процессы могут пройти через свои критические участки лишь ограниченное число раз. Это условие получило название условия ограниченного ожидания (bound waiting).

2. Программные алгоритмы организации взаимодействия процессов.

2.1 Переменная-замок

В качестве следующей попытки решения задачи для пользовательских процессов рассмотрим другое предложение. Возьмем некоторую переменную, доступную всем процессам, с начальным значением равным 0. Процесс может войти в критическую секцию только тогда, когда значение этой переменной-замка равно 0, одновременно изменяя ее значение на 1 – закрывая замок. При выходе из критической секции процесс сбрасывает ее значение в 0 – замок открывается (как в случае с покупкой хлеба студентами в разделе "Критическая секция").

```
shared int lock = 0;
/* shared означает, что */
/* переменная является разделяемой */

while (some condition) {
   while(lock); lock = 1;
      critical section
   lock = 0;
      remainder section
}
```

К сожалению, при внимательном рассмотрении мы видим, что такое решение не удовлетворяет условию взаимоисключения, так как действие while(lock); lock = 1; не является атомарным. Допустим, процесс P0 протестировал значение переменной lock и принял решение двигаться дальше. В этот момент, еще до присвоения переменной lock значения 1, планировщик передал управление процессу P1. Он тоже изучает содержимое переменной lock и тоже принимает решение войти в критический участок. Мы получаем два процесса, одновременно выполняющих свои критические секции.

2.2 Строгое чередование

Попробуем решить задачу сначала для двух процессов. Очередной подход будет также использовать общую для них обоих переменную с начальным значением 0. Только теперь она будет играть не роль замка для критического участка, а явно указывать, кто может следующим войти в него. Для i-го процесса это выглядит так:

```
shared int turn = 0;

while (some condition) {
   while(turn != i);
      critical section
   turn = 1-i;
      remainder section
}
```

Очевидно, что взаимоисключение гарантируется, процессы входят в критическую секцию строго по очереди: P0, P1, P0, P1, P0, ... Но наш алгоритм не удовлетворяет условию прогресса. Например, если значение turn равно 1, и процесс P0 готов войти в критический участок, он не может сделать этого, даже если процесс P1 находится в remainder section.

2.3 Флаги готовности

Недостаток предыдущего алгоритма заключается в том, что процессы ничего не знают о состоянии друг друга в текущий момент времени. Давайте попробуем исправить эту ситуацию. Пусть два наших процесса имеют разделяемый массив флагов готовности входа процессов в критический участок

```
shared int ready[2] = {0, 0};
```

Когда i-й процесс готов войти в критическую секцию, он присваивает элементу массива ready[i] значение равное 1. После выхода из критической секции он, естественно, сбрасывает это значение в 0. Процесс не входит в критическую секцию, если другой процесс уже готов к входу в критическую секцию или находится в ней.

```
while (some condition) {
   ready[i] = 1;
   while(ready[1-i]);
      critical section
   ready[i] = 0;
      remainder section
}
```

Полученный алгоритм обеспечивает взаимоисключение, позволяет процессу, готовому к входу в критический участок, войти в него сразу после завершения эпилога в другом процессе, но все равно нарушает условие прогресса. Пусть процессы практически одновременно подошли к выполнению пролога. После выполнения присваивания ready[0]=1 планировщик передал процессор от процесса 0 процессу 1, который также выполнил присваивание ready[1]=1. После этого оба процесса бесконечно долго ждут друг друга на входе в критическую секцию. Возникает ситуация, которую принято называть тупиковой (deadlock).

2.4 Алгоритм Петерсона

Первое решение проблемы, удовлетворяющее всем требованиям и использующее идеи ранее рассмотренных алгоритмов, было предложено датским математиком Деккером (Dekker). В 1981 году Петерсон (Peterson) предложил более изящное решение. Пусть оба процесса имеют доступ к массиву флагов готовности и к переменной очередности.

```
shared int ready[2] = {0, 0};
shared int turn;
while (some condition) {
   ready[i] = 1;
   turn =1-i;
   while(ready[1-i] && turn == 1-i);
      critical section
   ready[i] = 0;
      remainder section
}
```

При исполнении пролога критической секции процесс Pi заявляет о своей готовности выполнить критический участок и одновременно предлагает другому процессу приступить к его выполнению. Если оба процесса подошли к прологу практически одновременно, то они оба объявят о своей готовности и предложат выполняться друг другу. При этом одно

из предложений всегда следует после другого. Тем самым работу в критическом участке продолжит процесс, которому было сделано последнее предложение.

2.5 Алгоритм булочной (Bakery algorithm)

Алгоритм Петерсона дает нам решение задачи корректной организации взаимодействия двух процессов. Давайте рассмотрим теперь соответствующий алгоритм для n взаимодействующих процессов, который получил название алгоритм булочной, хотя применительно к нашим условиям его следовало бы скорее назвать алгоритм регистратуры в поликлинике. Основная его идея выглядит так. Каждый вновь прибывающий клиент (он же процесс) получает талончик на обслуживание с номером. Клиент с наименьшим номером на талончике обслуживается следующим. К сожалению, из-за неатомарности операции вычисления следующего номера алгоритм булочной не гарантирует, что у всех процессов будут талончики с разными номерами. В случае равенства номеров на талончиках у двух или более клиентов первым обслуживается клиент с меньшим значением имени (имена можно сравнивать в лексикографическом порядке). Разделяемые структуры данных для алгоритма – это два массива

```
shared enum {false, true} choosing[n];
shared int number[n];
```

Изначально элементы этих массивов инициируются значениями false и 0 соответственно. Введем следующие обозначения

```
(a,b) < (c,d), если a < c
или если a == c и b < d
max(a0, a1, ...., an) – это число k такое, что
k >= ai для всех i = 0, ...,n
```

Структура процесса Pi для алгоритма булочной приведена ниже

```
while (some condition) {
   choosing[i] = true;
   number[i] = max(number[0], ...,
                   number[n-1]) + 1;
   choosing[i] = false;
   for(j = 0; j < n; j++){
      while(choosing[j]);
      while(number[j] != 0 && (number[j],j) <
         (number[i],i));
   }
      critical section
   number[i] = 0;
      remainder section
}
```

Доказательство того, что этот алгоритм удовлетворяет условиям 1 – 5, выполните самостоятельно в качестве упражнения.

2. Индивидуальные задания

2.1 Алгоритм взаимодействия двух процессов «**Переменная – замок**»

Выполнить алгоритм синхронизации двух процессов (P0, P1) «*переменная – замок*», использующих общие ресурсы, для данных приведенных в таблице 2.1. Алгоритм планирования процессов ***Round Robin (RR)***, величина кванта времени **3**. Результаты оформить в виде таблицы иллюстрирующей работу процессов.

Таблица 2.1 Варианты заданий

№	Время возникновения входа в критическую секцию для **P0**	Время возникновения входа в критическую секцию для **P1**	Время выполнения критической секции **P0**	Время выполнения критической секции **P1**
1	1-4-11-15-19-28	8-13-18-22-27-30	2-1-1-1-2-1	1-1-1-1-1-4
2	2-5-12-19-22-27	8-15-19-22-28-30	1-1-1-1-1-2	1-1-2-1-1-3
3	1-4-10-16-21-29	8-12-15-18-22-26	2-1-1-1-1-1	1-1-1-1-2-1
4	2-5-13-18-20-23	8-13-15-19-22-29	1-1-1-1-1-1	1-1-2-1-1-2
5	1-5-10-14-19-25	7-10-15-20-26-30	2-1-1-2-2-1	1-2-1-1-2-1
6	2-4-16-22-29-36	8-13-19-23-27-32	1-1-1-2-1-3	1-2-2-1-1-1
7	1-5-19-23-26-34	9-13-17-23-27-33	2-1-2-1-1-2	1-1-2-1-1-1
8	2-5-16-22-29-35	9-15-22-25-29-35	2-1-1-1-1-1	1-1-1-2-2-1
9	1-4-14-18-25-32	8-14-18-22-27-35	1-1-1-1-1-2	1-2-1-1-2-1
10	1-4-16-21-26-32	9-14-18-25-30-35	2-1-1-1-1-2	1-1-1-1-1-3
11	2-5-12-16-20-24	10-13-20-23-27-32	1-1-2-1-1-2	1-1-1-2-1-1
12	1-5-12-18-25-32	10-16-19-22-26-30	1-1-1-1-2-2	1-1-1-2-1-4
13	2-5-12-17-23-32	10-14-18-25-30-33	1-1-2-1-2-3	1-1-1-1-2-1
14	1-5-9-15-22-27-33	10-16-19-25-28-30	1-1-1-1-1-2	1-1-1-1-2-1
15	1-5-11-18-27-34	10-13-19-24-27-34	1-1-1-1-2-1	1-1-1-1-1-1

Пример

Выполнить алгоритм синхронизации двух процессов (P0, P1) «*переменная – замок*»:

Время возникновения входа в критическую секцию для **P0**	Время возникновения входа в критическую секцию для **P1**	Время выполнения критической секции **P0**	Время выполнения критической секции **P1**
1-6-15	3-9-15	4-3-1	2-4-2

КС – выполнение критической секции, ГК – готовность критической секции к выполнению, И – выполнение не критической секции, Г – готовность некритической секции к выполнению.

Т	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
P0	КС	КС	КС	КС	Г	ГК	ГК	КС	КС	КС	Г	Г	Г	Г	КС	И	И	Г	Г
P1	Г	Г	Г	ГК	КС	КС	И	Г	ГК	ГК	КС	КС	КС	КС	ГК	ГК	ГК	КС	КС

2.2 Алгоритм взаимодействия двух процессов «**Строгое – чередование**»

Выполнить алгоритм синхронизации двух процессов (P0, P1) «*строгое – чередование*», использующих общие ресурсы, для данных приведенных в таблице 2.1. Алгоритм планирования процессов ***Round Robin (RR)***, величина кванта времени **3**. Результаты оформить в виде таблицы иллюстрирующей работу процессов.

2.3 Алгоритм взаимодействия трех процессов

Выполнить алгоритмы синхронизации процессов (P0, P1) «*переменная – замок*» и «*строгое – чередование*», использующих общие ресурсы, при наличии третьего процесса (P2), не использующего ресурсы процессов P0, P1. Данные процессов (P0, P1) «приведенных в таблице 2.1, процесс P2 появляется каждый 6 квант времени, длительность процесса равна **3** квантам. Алгоритм планирования процессов ***Round Robin (RR)***, величина кванта времени **3**. Если процесс P2 выполниться не успел, новый его экземпляр в очередь не ставится. Процесс P2 не может прервать выполнение критической секции. Результаты оформить в виде таблиц иллюстрирующих работу процессов.

2.4 Алгоритм взаимодействия нескольких процессов

Выполнить алгоритм синхронизации четырех процессов (P0, P1, P2, P3) «*алгоритм булочной*», использующих общие ресурсы. Процессы выбираются из таблицы 2.1, согласно таблице 2.2. При каждой постановке в очередь критической секции, вычисляется номер присваиваемый процессу.
Алгоритм планирования процессов ***Round Robin (RR)***, величина кванта времени **3**.
Результаты оформить в виде таблицы иллюстрирующей работу процессов, в таблице указывать номер

Таблица 2.2 Варианты заданий

№	Процессы P0, P1	Процессы P2, P3
1	1	2
2	2	3
3	4	5
4	6	7
5	8	9
6	10	11
7	12	13
8	14	15
9	1	8
10	2	9
11	3	10
12	4	11
13	5	12
14	6	13
15	7	14

Вопросы

1. Охарактеризовать проблему синхронизации потоков.
2. Понятия активностей, детерминированных и недетерминирован активностей.
3. Условия детерминированности активностей Бернстайна.
4. Понятие критической секции.
5. Требования, предъявляемые к алгоритмам взаимодействия процессов.
6. Алгоритм взаимодействия процессов – «Запрет прерываний».
7. Алгоритм взаимодействия процессов – «Пременная замок».
8. Алгоритм взаимодействия процессов – «Строгое чередование».
9. Алгоритм взаимодействия процессов – «Флаги готовности».
10. Алгоритм взаимодействия процессов – Петерсона.
11. Алгоритм «Булочной» взаимодействия процессов.

Лабораторная работа №7

Тупиковые ситуации и подходы к их разрешению

Цель работы: Изучить причины возникновения тупиковых ситуаций и подходов к их разрешению.

1. Теоретические сведения

1.1 Тупики

Предположим, что несколько процессов конкурируют за обладание конечным числом ресурсов. Если запрашиваемый процессом ресурс недоступен, ОС переводит данный процесс в состояние ожидания. В случае когда требуемый ресурс удерживается другим ожидающим процессом, первый процесс не сможет сменить свое состояние. Такая ситуация называется тупиком (deadlock). Говорят, что в мультипрограммной системе процесс находится в состоянии тупика, если он ожидает события, которое никогда не произойдет. Системная тупиковая ситуация, или "зависание системы", является следствием того, что один или более процессов находятся в состоянии тупика. Иногда подобные ситуации называют взаимоблокировками. В общем случае проблема тупиков эффективного решения не имеет.

Рассмотрим пример. Предположим, что два процесса осуществляют вывод с ленты на принтер. Один из них успел монополизировать ленту и претендует на принтер, а другой наоборот. После этого оба процесса оказываются заблокированными в ожидании второго ресурса (см. рис. 1.1).

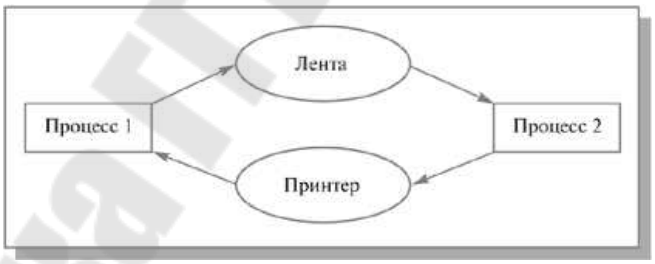

Рис. 1.1. Пример тупиковой ситуации

Определение. Множество процессов находится в тупиковой ситуации, если каждый процесс из множества ожидает события, которое может вызвать только другой процесс данного множества. Так как все процессы чего-то ожидают, то ни один из них не сможет инициировать событие, которое разбудило бы другого члена множества и, следовательно, все процессы будут спать вместе.

Выше приведен пример взаимоблокировки, возникающей при работе с так называемыми выделенными устройствами. Тупики, однако, могут иметь место и в других ситуациях. В этом случае может получиться так, что один из процессов заблокировал записи, необходимые другому процессу, и наоборот. Таким образом, тупики могут иметь место как на аппаратных, так и на программных ресурсах.

Тупики также могут быть вызваны ошибками программирования. Например, процесс может напрасно ждать открытия семафора, потому что в некорректно написанном приложении эту операцию забыли предусмотреть. Другой причиной бесконечного ожидания может быть дискриминационная политика по отношению к некоторым процессам. Однако чаще всего событие, которого ждет процесс в тупиковой ситуации, –

освобождение ресурса, поэтому в дальнейшем будут рассмотрены методы борьбы с тупиками ресурсного типа.
Ресурсами могут быть как устройства, так и данные. Некоторые ресурсы допускают разделение между процессами, то есть являются разделяемыми ресурсами. Например, память, процессор, диски коллективно используются процессами. Другие не допускают разделения, то есть являются выделенными, например лентопротяжное устройство. К взаимоблокировке может привести использование как выделенных, так и разделяемых ресурсов. Например, чтение с разделяемого диска может одновременно осуществляться несколькими процессами, тогда как запись предполагает исключительный доступ к данным на диске. Можно считать, что часть диска, куда происходит запись, выделена конкретному процессу. Поэтому в дальнейшем мы будем исходить из предположения, что тупики связаны с выделенными ресурсами , то есть тупики возникают, когда процессу предоставляется эксклюзивный доступ к устройствам, файлам и другим ресурсам.
Традиционная последовательность событий при работе с ресурсом состоит из запроса, использования и освобождения ресурса. Тип запроса зависит от природы ресурса и от ОС. Запрос может быть явным, например специальный вызов request, или неявным – open для открытия файла. Обычно, если ресурс занят и запрос отклонен, запрашивающий процесс переходит в состояние ожидания.
Далее в данной лекции будут рассматриваться вопросы обнаружения, предотвращения, обхода тупиков и восстановления после тупиков. Как правило, борьба с тупиками – очень дорогостоящее мероприятие. Тем не менее для ряда систем, например для систем реального времени, иного выхода нет.

1.2 Основные направления исследований по проблеме тупиков

В связи с проблемой тупиков были проведены одни из наиболее интересных исследований в области информатики и операционных систем. Результаты этих исследований оказались весьма эффективными в том смысле, что позволили разработать четкие методы решения многих распространенных проблем. В исследованиях по проблеме тупиков можно выделить следующие четыре основные направления:

1. предотвращение тупиков;
2. обход тупиков;
3. обнаружение тупиков;
4. восстановление после тупиков.

При *предотвращении тупиков* целью является обеспечение условий, исключающих возможность возникновения тупиковых ситуаций. Такой подход является вполне корректным решением в том, что касается самого тупика, однако он часто приводит к нерациональному использованию ресурсов. Тем не менее различные методы предотвращения тупиков широко применяются в практике разработчиков.

Цель средств обхода тупиков заключается в том, чтобы можно было предусматривать менее жесткие ограничения, чем в случае предотвращения тупиков, и тем самым обеспечить лучшее использование ресурсов. При наличии средств обхода тупиков не требуется такой реализации системы, при которой опасность тупиковых ситуаций даже не возникает. Напротив, методы обхода тупиков учитывают подобную возможность, однако в случае увеличения вероятности конкретной тупиковой ситуации здесь принимаются меры по аккуратному обходу тупика. (См. обсуждение алгоритма банкира, который предложил Дейкстра, ниже в настоящей главе.)

Методы *обнаружения* тупиков применяются в системах, которые допускают возможность возникновения тупиковых ситуаций как следствие либо умышленных, либо неумышленных действий программистов. Цель средств обнаружения тупиков — установить сам факт возникновения тупиковой ситуации, причем точно определить те процессы и ресурсы, которые оказались вовлеченными в данную тупиковую ситуацию.

После того как все это сделано, данную тупиковую ситуацию в системе можно будет устранить.

Методы *восстановления* после тупиков применяются для устранения тупиковых ситуаций, с тем чтобы система могла продолжать работать, а процессы, попавшие в тупиковую ситуацию, могли завершиться с освобождением занимаемых ими ресурсов. Восстановление — это, мягко говоря, весьма серьезная проблема, так что в большинстве систем оно осуществляется путем полного выведения из решения одного или более процессов, попавших в тупиковую ситуацию. Затем выведенные процессы обычно перезапускаются с самого начала, так что почти вся или даже вся работа, проделанная этими процессами, теряется.

1.3 Предотвращение тупиков

До сих пор разработчики систем при решении проблемы тупиков чаще всего шли по пути исключения самих возможностей тупиков. В настоящем разделе рассматриваются различные методы предотвращения тупиков, а также оцениваются различные последствия их реализации как для пользователя, так и для систем, особенно с точки зрения эксплуатационных характеристик и эффективности работы.

Хавендер показал, что возникновение тупика невозможно, если нарушено хотя бы одно из указанных выше четырех необходимых условий. Он предложил следующую стратегию.

1. Каждый процесс должен запрашивать все требуемые ему ресурсы сразу, причем не может начать выполняться до тех пор, пока все они не будут ему предоставлены.
2. Если процесс, удерживающий определенные ресурсы, получает отказ в удовлетворении запроса на дополнительные ресурсы, этот процесс должен освободить свои первоначальные ресурсы и при необходимости запросить их снова вместе с дополнительными.
3. Введение линейной упорядоченности по типам ресурсов для всех процессов; другими словами, если процессу выделены ресурсы данного типа, в дальнейшем он может запросить только ресурсы более далеких по порядку типов.

Отметим, что Хавендер предлагает три стратегических принципа, а не четыре. Каждый из указанных принципов, как мы увидим в следующих разделах, имеет целью нарушить какое-нибудь одно из необходимых условий существования тупика. Первое необходимое условие, а именно условие взаимоисключения, согласно которому процессы получают право на монопольное управление выделяемыми им ресурсами, мы не хотим нарушать, поскольку нам, в частности, нужно предусмотреть возможность работы с *закрепленными* ресурсами.

1.4 Нарушение условия «ожидания дополнительных ресурсов»

Первый стратегический принцип Хавендера требует, чтобы процесс сразу же запрашивал все ресурсы, которые ему понадобятся. Система должна предоставлять эти ресурсы по принципу «все или ничего». Если набор ресурсов, необходимый процессу, имеется, то система может предоставить все эти ресурсы данному процессу сразу же, так что он получит возможность продолжить работу. Если в текущий момент полного набора ресурсов, необходимых процессу, нет, этому процессу придется ждать, пока они не освободятся. Однако, когда процесс находится в состоянии ожидания, он не должен удерживать какие-либо ресурсы. Благодаря этому предотвращается возникновение условия «ожидания дополнительных ресурсов» и тупиковые ситуации просто невозможны.

На первый взгляд это звучит неплохо, однако подобный подход может привести к серьезному снижению эффективности использования ресурсов. Например, программа, которой в какой-то момент работы требуются 10 лентопротяжных устройств, должна запросить — и получить — все 10 устройств, прежде чем начать выполнение. Если все 10

устройств необходимы программе в течение всего времени выполнения, то это не приведет к серьезным потерям. Рассмотрим, однако, случай, когда программе для начала работы требуется только одно устройство (или, что еще хуже, ни одного), а оставшиеся устройства потребуются лишь через несколько часов. Таким образом, требование о том, что программа должна запросить и получить все эти устройства, чтобы начать выполнение, на практике означает, что значительная часть ресурсов компьютера будет использоваться вхолостую в течение нескольких часов.

Один из подходов, к которому часто прибегают разработчики систем с целью улучшения использования ресурсов в подобных обстоятельствах, заключается в том, чтобы разделять программу на несколько программных шагов, выполняемых относительно независимо друг от друга. Благодаря этому можно осуществлять выделение ресурсов для каждого шага программы, а не для целого процесса. Это позволяет уменьшить непроизводительное использование ресурсов, однако связано с увеличением накладных расходов на этапе проектирования прикладных программ, а также на этапе их выполнения.

Такая стратегия существенно повышает вероятность бесконечного откладывания, поскольку не все требуемые ресурсы могут оказаться свободными в нужный момент. Вычислительная система должна позволить завершиться достаточно большому числу заданий и освободить их ресурсы, чтобы ожидающее задание смогло продолжить выполнение. В течение периода, когда требуемые ресурсы накапливаются, их нельзя выделять другим заданиям, так что эти ресурсы простаивают. В настоящее время существуют противоречивые мнения о том, кто должен платить за эти неиспользуемые ресурсы. Некоторые разработчики считают, что, поскольку ресурсы накапливаются для конкретного пользователя, этот пользователь должен платить за них, даже в течение периода простоя ресурсов. Однако другие разработчики считают, что это привело бы к нарушению принципа *предсказуемости платы за ресурсы,* если пользователь попытается выполнить свою работу в период пиковой загрузки машины, ему придется платить гораздо больше, чем в случае, когда машина относительно свободна.

1.5 Нарушение условия неперераспределяемости

Второй стратегический принцип Хавендера, рассматриваемый здесь независимо от других, предотвращает возникновение условия неперераспределяемости. Предположим, что система позволяет процессам, запрашивающим дополнительные ресурсы, удерживать за собой ранее выделенные ресурсы. Если у системы достаточно свободных ресурсов, чтобы удовлетворять все запросы, тупиковая ситуация не возникнет. Посмотрим, однако, что произойдет, когда какой-то запрос удовлетворить не удастся. В этом случае один процесс удерживает ресурсы, которые могут потребоваться второму процессу для продолжения работы, а этот второй процесс может удерживать ресурсы, необходимые первому. Это и есть тупик.

Второй стратегический принцип Хавендера требует, чтобы процесс, имеющий в своем распоряжении некоторые ресурсы, если он получает отказ на запрос о выделении дополнительных ресурсов, должен освобождать свои ресурсы и при необходимости запрашивать их снова вместе с дополнительными. Такая стратегия действенно ведет к нарушению условия неперераспределяемости, подобным образом можно забирать ресурсы у удерживающих их процессов до завершения этих процессов.

Но этот способ предотвращения тупиковых ситуаций также не свободен от недостатков. Если процесс в течение некоторого времени использует определенные ресурсы, а затем освобождает эти ресурсы, он может потерять всю работу, проделанную до данного момента. На первый взгляд такая цена может показаться слишком высокой, однако необходим реальный ответ на вопрос «насколько часто приходится платить такую цену?» Если такая ситуация встречается редко, то можно считать, что в нашем распоряжении имеется относительно недорогой способ предотвращения тупиков. Если, однако, такая ситуация встречается часто, то подобный способ обходится дорого, причем приводит к

печальным результатам, особенно когда не удается вовремя завершить высокоприоритетные или срочные процессы.

Одним из серьезных недостатков такой стратегии является возможность бесконечного откладывания процессов. Выполнение процесса, который многократно запрашивает и освобождает одни и те же ресурсы, может откладываться на неопределенно долгий срок. В этом случае у системы может возникнуть необходимость вывести данный процесс из решения, с тем чтобы другие процессы получили возможность продолжения работы. Нельзя игнорировать и вероятный случай, когда процесс, откладываемый бесконечно, может оказаться не очень «заметным» для системы. В такой ситуации возможно поглощение процессом значительных вычислительных ресурсов и деградация производительности системы.

1.6 Нарушение условия «кругового ожидания»

Третий стратегический принцип Хавендера исключает круговое ожидание. Поскольку всем ресурсам присваиваются уникальные номера и поскольку процессы должны запрашивать ресурсы в порядке возрастания номеров, круговое ожидание возникнуть не может.

Такой стратегический принцип реализован в ряде операционных систем, однако это оказалось сопряжено с определенными трудностями.

Поскольку запрос ресурсов осуществляется в порядке возрастания их номеров и поскольку номера ресурсов назначаются при установке машины и должны «жить» в течение длительных периодов времени (месяцев или даже лет), то в случае введения в машину новых типов ресурсов может возникнуть необходимость переработки существующих программ и систем.

Очевидно, что назначаемые номера ресурсов должны отражать нормальный порядок, в котором большинство заданий фактически используют ресурсы. Для заданий, выполнение которых соответствует этому порядку, можно ожидать высокой эффективности. Если, однако, заданию требуются ресурсы не в том порядке, который предполагает вычислительная система, то оно должно будет захватывать и удерживать ресурсы, быть может, достаточно долго еще до того, как они будут фактически использоваться. А это означает потерю эффективности.

Одной из самых важных задач современных операционных систем является предоставление пользователю максимальных удобств для работы («дружественной» обстановки). Пользователи должны иметь возможность разрабатывать свои прикладные программы при минимальном «загрязнении своей окружающей среды» из-за ограничений, накладываемых недостаточно удачными аппаратными и программными средствами. Последовательный порядок заказа ресурсов, предложенный Хавендером, действительно исключает возможность возникновения кругового ожидания, однако безусловно отрицательно сказывается при этом на возможности пользователя свободно и легко писать прикладные программы.

1.7 Предотвращение тупиков и алгоритм банкира

Если даже необходимые условия возникновения тупиков не нарушены, то все же можно избежать тупиковой ситуации, если рационально распределять ресурсы, придерживаясь определенных правил. По-видимому, наиболее известным алгоритмом обхода тупиковых ситуаций является алгоритм банкира, который предложил Дейкстра; алгоритм получил такое любопытное наименование потому, что он как бы имитирует действия банкира, который, располагая определенным источником капитала, выдает ссуды и принимает платежи. Ниже мы изложим этот алгоритм применительно к проблеме распределения ресурсов в операционных системах.

1.7.1 Алгоритм банкира, предложенный Дейкстрой

Когда при описании алгоритма банкира мы будем говорить о ресурсах, мы будем подразумевать ресурсы одного и того же типа, однако этот алгоритм можно легко распространить на пулы ресурсов нескольких различных типов. Рассмотрим, например, проблему распределения некоторого количества *t* идентичных лентопротяжных устройств.
Операционная система должна обеспечить распределение некоторого фиксированного числа *t* одинаковых лентопротяжных устройств между некоторым фиксированным числом пользователей *u.* Каждый пользователь заранее указывает максимальное число устройств, которые ему потребуются во время выполнения своей программы на машине. Операционная система примет запрос пользователя в случае, если максимальная потребность этого пользователя в лентопротяжных устройствах не превышает *t.*
Пользователь может занимать или освобождать устройства по одному. Возможно, что иногда пользователю придется ждать выделения дополнительного устройства, однако операционная система гарантирует, что ожидание будет конечным. Текущее число устройств, выделенных пользователю, никогда не превысит указанную максимальную потребность этого пользователя. Если операционная система в состоянии удовлетворить максимальную потребность пользователя в устройствах, то пользователь гарантирует, что эти устройства будут использованы и возвращены операционной системе в течение конечного периода времени.
Текущее состояние вычислительной машины называется *надежным,* если операционная система может обеспечить всем текущим пользователям завершение их заданий в течение конечного времени. (Здесь опять-таки предполагается, что лентопротяжные устройства являются единственными ресурсами, которые запрашивают пользователи.) В противном случае текущее состояние системы называется ненадежным.
Предположим теперь, что работают *n* пользователей.
Пусть *l(i)* представляет текущее количество лентопротяжных устройств, выделенных *i* пользователю. Если, например, пользователю 5 выделены четыре устройства, то *l*(5)=4. Пусть *m(i)* — максимальная потребность пользователя *i*, так что если пользователь 3 имеет максимальную потребность в двух устройствах, то *m*(3)=2. Пусть *c(t)* — текущая потребность пользователя, равная его максимальной потребности минус текущее число выделенных ему ресурсов. Если, например, пользователь 7 имеет максимальную потребность в шести лентопротяжных устройствах, а текущее количество выделенных ему устройств составляет четыре, то мы получаем

$$c\,(7)=m\,(7)\,-\,l(7)=6-4=2.$$

В распоряжении операционной системы имеются *t* лентопротяжных устройств. Число устройств, остающихся для распределения, обозначим через *a.* Тогда *a* равно *t* минус суммарное число устройств, выделенных различным пользователям.

Алгоритм банкира, который предложил Дейкстра, говорит о том, что выделять лентопротяжные устройства пользователям можно только в случае, когда после очередного выделения устройств состояние системы остается надежным. Надежное состояние — это состояние, при котором общая ситуация с ресурсами такова, что все пользователи имеют возможность со временем завершить свою работу. Ненадежное состояние — это состояние, которое может со временем привести к тупику.

1.7.2 Пример надежного состояния

Предположим, что в системе имеются двенадцать одинаковых лентопротяжных устройств, причем эти накопители распределяются между тремя пользователями, как показывает состояние I.

Состояние I

	Текущее количество выделенных устройств		Максимальная

			потребность
Пользователь (1)	1		4
Пользователь (2)	4		6
Пользователь (3)	5		8
Резерв		2	

Это состояние «надежное», поскольку оно дает возможность всем трем пользователям закончить работу. Отметим, что в текущий момент пользователь (2) имеет четыре выделенных ему устройства и со временем потребует максимум шесть, т. е. два дополнительных устройства. В распоряжении системы имеются двенадцать устройств, из которых десять в настоящий момент в работе, а два — в резерве. Если эти два резервных устройства, имеющихся в наличии, выделить пользователю (2), удовлетворяя тем самым максимальную потребность этого пользователя, то он сможет продолжать работу до завершения. После завершения пользователь (2) освободит все шесть своих устройств так что система сможет выделить их пользователю (1) и пользователю (3). Пользователь (1) имеет одно устройство и со временем ему потребуется еще три. У пользователя (3) — пять устройств и со временем ему потребуется еще три. Если пользователь (2) возвращает шесть накопителей, то три из них можно выделить пользователю (1), который получит таким образом возможность закончить работу и затем возвратить четыре накопителя системе. После этого система может выделить три накопителя пользователю (3), который тем самым также получает возможность закончить работу. Таким образом, основной критерий надежного состояния — это существование последовательности действий, позволяющей всем пользователям закончить работу.

1.7.3 Пример ненадежного состояния

Предположим, что двенадцать лентопротяжных устройств, имеющихся в системе, распределены согласно состоянию II.

Состояние II

	Текущее количество выделенных устройств		Максимальная потребность
Пользователь (1)	8		10
Пользователь (2)	2		5
Пользователь (3)	1		3
Резерв		1	

Здесь из двенадцати устройств системы одиннадцать в настоящий момент находятся в работе и только одно остается в резерве. В подобной ситуации независимо от того, какой пользователь запросит это резервное устройство, мы не можем гарантировать, что все три пользователя закончат работу. И действительно, предположим, что пользователь (1) запрашивает и получает последнее оставшееся устройство. При этом тупиковая ситуация может возникать в трех случаях, когда каждому из трех процессов потребуется запросить по крайней мере одно дополнительное устройство, не возвратив некоторое количество устройств в общий пул ресурсов.

Здесь важно отметить, что термин *«ненадежное состояние'!) не. предполагает, что в данный момент существует или в какое-то время обязательно возникнет тупиковая ситуация. Он просто говорит о том, что в случае некоторой неблагоприятной последовательности событий система может зайти в тупик.*

1.7.4 Пример перехода из надежного состояния в ненадежное

Если известно, что данное состояние надежно, это вовсе не означает, что все последующие состояния также будут надежными. Наш механизм распределения ресурсов должен тщательно анализировать все запросы на выделение ресурсов, прежде чем удовлетворять их. Рассмотрим, например, случай, когда система находится в текущем состоянии, показанном как состояние III.

Предположим теперь, что пользователь (3) запрашивает дополнительный ресурс. Если бы система удовлетворила этот запрос, она перешла бы в новое состояние, состояние IV.

Конечно, состояние IV не обязательно приведет к тупиковой ситуации. Если, однако, состояние III было надежным, то состояние IV стало ненадежным. Состояние IV характеризует систему,

Состояние III

	Текущее количество выделенных устройств		Максимальная потребность
Пользователь (1)	1		4
Пользователь (2)	4		6
Пользователь (3)	5		8
Резерв		2	

Состояние IV

	Текущее количество выделенных устройств		Максимальная потребность
Пользователь (1)	1		4
Пользователь (2)	4		6
Пользователь (3)	6		8
Резерв		1	

в которой нельзя гарантировать успешное завершение всех процессов пользователей. В резерве здесь остается только один ресурс, в то время как в наличии должно быть минимум два ресурса, чтобы либо пользователь (2), либо пользователь (3) могли завершить свою работу, возвратить занимаемые ими ресурсы системе и тем самым позволить остальным пользователям закончить выполнение.

1.7.5 Распределение ресурсов согласно алгоритму банкира

Сейчас уже должно быть ясно, каким образом осуществляется распределение ресурсов согласно алгоритму банкира, который предложил Дейкстра. Условия «взаимоисключения», «ожидания дополнительных ресурсов» и «неперераспределяемости» выполняются. Процессы могут претендовать на монопольное использование ресурсов, которые им требуются. Процессам реально разрешается удерживать за собой ресурсы, запрашивая и ожидая выделения дополнительных ресурсов, причем ресурсы нельзя отбирать у процесса, которому они выделены. Пользователи не ставят перед системой особенно сложных задач, запрашивая в каждый момент времени только один ресурс. Система может либо удовлетворить, либо отклонить каждый запрос. Если запрос отклоняется, пользователь удерживает за собой уже выделенные ему ресурсы и ждет определенный конечный период времени, пока этот запрос в конце концов не будет удовлетворен. *Система удовлетворяет только те запросы, при которых ее состояние остается надежным.* Запрос пользователя, приводящий к переходу системы в ненадежное состояние, откладывается до момента, когда его все же можно будет выполнить. Таким образом, поскольку система всегда поддерживается в надежном состоянии, рано или поздно (т. е. в течение конечного

времени) все запросы можно будет удовлетворить и все пользователи смогут завершить свою работу.

1.7.6 Алгоритм банкира для одного вида ресурса

Алгоритм банкира (рисунок 1.2):

1. Банкиру поступает запрос от клиента на получение кредита
2. Банкир проверяет, приводит ли этот запрос к небезопасному состоянию.
3. Банкир в зависимости от этого дает или отказывает в кредите.

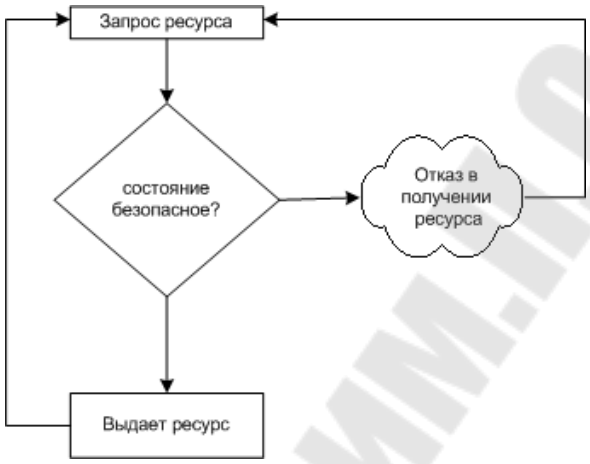

Рисунок 1.2 Алгоритм банкира для одного вида ресурса

1.7.7 Алгоритм банкира для нескольких видов ресурсов. Алгоритм поиска безопасного или небезопасного состояния.

Рассмотрим систему.

m - число классов ресурсов (например: принтеры это один класс);

n - количество процессов;

P(n) – процессы;

E - вектор существующих ресурсов;

E(i) - количество ресурсов класса I;

A - вектор доступных (свободных) ресурсов;

A(i) - количество доступных ресурсов класса I;

C - матрица текущего распределения (какому процессу, какие ресурсы принадлежат);

R - матрица запросов (какой процесс, какой ресурс запросил);

C(i, j) - количество экземпляров ресурса j, которое занимает процесс P(i);

R(i, j) - количество экземпляров ресурса j, которое максимум хочет получить процесс P(i).

Существующие ресурсы

$$(E_1 \quad E_2 \quad E_3 \quad \dots \quad E_m)$$

Доступные ресурсы

$$(A_1 \quad A_2 \quad A_3 \quad \dots \quad A_m)$$

Матрица текущего распределения

$$\begin{bmatrix} C_{11} & C_{12} & \dots & C_{1m} \\ C_{21} & C_{22} & \dots & C_{2m} \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ C_{n1} & C_{n2} & \dots & C_{nm} \end{bmatrix}$$

Строка **n** предоставлена процессу **n**

Матрица запроса

$$\begin{bmatrix} R_{11} & R_{12} & \dots & R_{1m} \\ R_{21} & R_{22} & \dots & R_{2m} \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ R_{n1} & R_{n2} & \dots & R_{nm} \end{bmatrix}$$

Строка **n** нужна процессу **n**

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

Алгоритм обнаружения тупиков состоит из следующих шагов:

1. Ищем немаркированный процесс P_i для которого i-я строка матрицы R меньше вектора A или равна ему.
2. Если такой процесс найден, прибавляем f-ю строку матрицы C к вектору A, маркируем процесс и возвращаемся к шагу 1.
3. Если таких процессов не существует, работа алгоритма заканчивается.

Завершение алгоритма означает, что все немаркированные процессы, если такие есть, попали в тупик.

Рисунок 1.3 Алгоритм банкира для несколько видов ресурсов

На первом шаге алгоритм ищет процесс, который может доработать до конца. Такой процесс характеризуется тем, что все требуемые для него ресурсы должны находиться среди доступных в данный момент ресурсов. Тогда выбранный процесс проработает до своего завершения и после этого вернет ресурсы, которые он занимал, в общий фонд доступных ресурсов. Затем процесс маркируется как законченный. Если окажется, что все процессы могут работать, тогда ни один из них в данный момент не заблокирован. Если

некоторые из них никогда не смогут запуститься, значит, они попали в тупик. Несмотря на то что алгоритм не является детерминированным (поскольку он может просматривать процессы в любом допустимом порядке), результат всегда одинаков.
Для иллюстрации работы алгоритма обнаружения тупиков рассмотрим рис. 1.4. Здесь у нас есть три процесса и четыре класса ресурсов, которые мы произвольно назвали так: накопители на магнитной ленте, плоттеры, сканеры и устройство для чтения компакт-дисков. Процесс 1 использует один сканер. Процесс 2 занял два накопителя на магнитной ленте и устройство для чтения компакт-дисков. Процесс 3 занимает плоттер и два сканера. Каждый процесс нуждается в дополнительном устройстве, как показывает матрица *R*.
Работая с алгоритмом обнаружения взаимоблокировок, мы ищем процесс, чей запрос ресурсов может быть удовлетворен в данной системе. Требования первого процесса нельзя выполнить, потому что в системе нет доступного устройства для чтения компакт-дисков. Второй запрос также нельзя удовлетворить, так как нет свободных сканеров. К счастью, третий процесс может получить все желаемое, следовательно, он работает, завершается и возвращает все свои ресурсы, давая:

$$A = (2\ 2\ 2\ 0).$$

С этого момента может выполняться процесс 2, по окончании возвращая свои ресурсы в систему. Мы получим:

$$A = (4\ 2\ 2\ 1).$$

Теперь может работать оставшийся процесс. В этой системе не возникает взаимоблокировки.

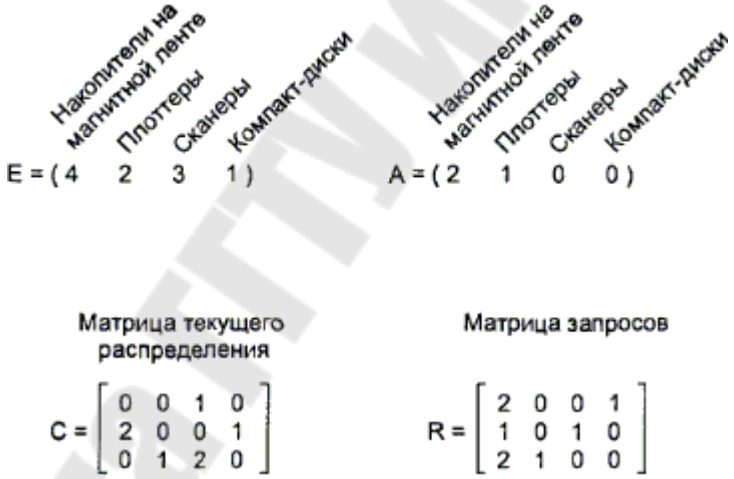

Рисунок 1.4 Пример использования алгоритма обнаружения тупиков

1.7.8 Недостатки алгоритма банкира

Алгоритм банкира представляет для нас интерес потому, что он дает возможность распределять ресурсы таким образом, чтобы обходить тупиковые ситуации. Он позволяет продолжать выполнение таких процессов, которым в случае системы с предотвращением тупиков пришлось бы ждать. Однако у этого алгоритма имеется ряд серьезных недостатков, из-за которых разработчик системы может оказаться вынужденным выбрать другой подход к решению проблемы тупиков.

1. Алгоритм банкира исходит из фиксированного количества распределяемых ресурсов. Поскольку устройства, представляющие ресурсы, зачастую требуют технического обслуживания либо из-за возникновения неисправностей, либо в целях профилактики, мы не можем считать, что количество ресурсов всегда остается фиксированным.
2. Алгоритм требует, чтобы число работающих пользователей оставалось постоянным. Подобное требование также является нереалистичным. В современных мультипрограммных системах количество работающих пользователей все время меняется. Например, большая система с разделением времени вполне может обслуживать 100 или более пользователей одновременно. Однако текущее число

обслуживаемых пользователей непрерывно меняется, быть может, очень часто, каждые несколько секунд.

3. Алгоритм требует, чтобы банкир—распределитель ресурсов гарантированно удовлетворял все запросы за некоторый конечный период времени. Очевидно, что для реальных систем необходимы гораздо более конкретные гарантии.
4. Аналогично, алгоритм требует, чтобы клиенты (т. е. задания или процессы) гарантированно «платили долги» (т. е. возвращали выделенные им ресурсы) в течение некоторого конечного времени. И опять-таки для реальных систем требуются гораздо более конкретные гарантии.
5. Алгоритм требует, чтобы пользователи заранее указывали свои максимальные потребности в ресурсах. По мере того как распределение ресурсов становится все более динамичным, все труднее оценивать максимальные потребности пользователя. Вообще говоря, поскольку компьютеры становятся более «дружественными» по отношению к пользователю, все чаще встречаются пользователи, которые не имеют ни малейшего представления о том, какие ресурсы им потребуются.

2. Индивидуальные задания

В соответствии с вариантом, выполнить построение последовательности **надежных** состояний системы при удовлетворении запросов на ресурсы в соответствии с алгоритмом «банкира». Исходные данные представлены в таблицах 2.1 и 2.2.

2.1 Задание 1 – Один ресурс

Таблица 2.1 Варианты заданий для одного ресурса

№	Ресурсы	Процесс 1	Процесс 2	Процесс 3	Макс. ресурсов
1	Выдано	1	3	2	9
	Потребность	4	8	6	
2	Выдано	1	2	2	7
	Потребность	3	5	7	
3	Выдано	3	0	5	12
	Потребность	7	4	10	
4	Выдано	1	2	1	7
	Потребность	3	7	5	
5	Выдано	1	2	0	6
	Потребность	5	5	5	
6	Выдано	2	2	1	8
	Потребность	8	3	4	
7	Выдано	3	2	1	9
	Потребность	7	9	3	
8	Выдано	0	3	2	8
	Потребность	1	8	4	
9	Выдано	5	1	2	12
	Потребность	10	5	8	
10	Выдано	3	1	2	10
	Потребность	9	3	6	
11	Выдано	2	1	1	8
	Потребность	7	7	3	
12	Выдано	0	2	1	8
	Потребность	3	8	4	
13	Выдано	5	0	3	12
	Потребность	11	2	7	
14	Выдано	1	3	3	10
	Потребность	5	6	7	
15	Выдано	4	3	2	13
	Потребность	10	10	5	

Пример 1.

№	Ресурсы	Процесс 1	Процесс 2	Процесс 3	Макс. ресурсов
1	Выдано	6	2	1	11
	Потребность	9	14	3	

Решение

Шаг 0

Процессы	Текущее количество выделенного ресурса	Резерв	Максимальная потребность
Процесс1	6	<	9
Процесс2	2	<	7
Процесс3	1	<	3
Итого	9	2	

Шаг 1

Процессы	Текущее количество выделенного ресурса	Резерв	Максимальная потребность
Процесс1	6	<	9
Процесс2	2	<	7
Процесс3	1+2	=	3
Итого	11	0	

Шаг 2

Процессы	Текущее количество выделенного ресурса	Резерв	Максимальная потребность
Процесс1	6	<	9
Процесс2	2	<	7
Процесс3	–		–
Итого	8	3	

Шаг 3

Процессы	Текущее количество выделенного ресурса	Резерв	Максимальная потребность
Процесс1	6+3	=	9
Процесс2	2	<	7
Процесс3	–		–
Итого	11	0	

Шаг 4

Процессы	Текущее количество выделенного ресурса	Резерв	Максимальная потребность
Процесс1	–		–
Процесс2	2	<	7
Процесс3	–		–
Итого	2	9	

Шаг 5

Процессы	Текущее количество выделенного ресурса	Резерв	Максимальная потребность
Процесс1	–		–
Процесс2	2+5	=	7

Процесс3	–		–
Итого	7	4	

Шаг 6

Процессы	Текущее количество выделенного ресурса	Резерв	Максимальная потребность
Процесс1	–		–
Процесс2	–		–
Процесс3	–		–
Итого	0	11	

2.2 Задание 2 – Несколько ресурсов

Максимальное количество ресурсов P1 – 7, P2 – 6. Ресурсы выделяются последовательно (в соответствии со значениями приведенными в скобках).

№	Максимальная потребность (и последовательность запрашиваемых ресурсов)											
	Процесс 1		Процесс 2		Процесс 3		Процесс 4		Процесс 5		Процесс 6	
	P1	P2	P1	P2	P1	P2	P1	P2	P1	P2	P1	P2
1	4(4+0+0)	3(1+1+1)	6(5+0+1)	3(0+0+3)	2(1+0+1)	2(0+0+2)	5(4+0+1)	4(1+1+2)	4(4+0+0)	2(0+1+1)	2(1+0+1)	2(2+0+0)
2	2(2+0+0)	2(0+2+0)	4(4+0+0)	6(5+1+0)	3(0+3+0)	3(0+2+1)	4(0+3+1)	2(1+0+1)	5(1+4+0)	4(3+1+0)	2(0+0+2)	4(0+0+4)
3	3(1+1+1)	2(2+0+0)	6(6+0+0)	3(3+0+0)	4(1+1+2)	4(0+4+0)	2(2+0+0)	3(1+2+0)	2(2+0+0)	2(0+2+0)	5(4+1+0)	3(0+2+1)
4	6(1+5+0)	6(0+5+1)	2(2+0+0)	4(3+1+0)	5(4+1+0)	3(0+3+0)	3(1+0+2)	2(0+2+0)	5(4+1+0)	4(1+3+0)	3(0+3+0)	3(0+0+3)
5	3(1+1+1)	3(1+0+2)	2(0+0+2)	2(0+2+0)	3(3+0+0)	6(4+0+2)	4(1+3+0)	2(0+2+0)	4(4+0+0)	5(5+0+0)	2(0+0+2)	3(2+0+1)
6	2(1+0+1)	3(3+0+0)	4(0+0+4)	5(1+1+3)	5(3+1+1)	2(0+0+2)	2(2+0+0)	2(0+0+2)	3(2+0+1)	2(0+0+2)	3(3+0+0)	4(4+0+0)
7	3(2+1+0)	3(0+1+2)	4(4+0+0)	5(0+4+1)	6(5+0+1)	2(0+2+0)	3(2+0+1)	4(3+0+1)	5(4+0+1)	5(1+0+4)	3(3+0+0)	2(1+0+1)
8	3(2+1+0)	2(1+0+1)	2(0+2+0)	2(2+0+0)	6(3+0+3)	5(2+0+3)	2(1+0+1)	3(3+0+0)	4(1+3+0)	4(3+1+0)	3(2+0+1)	3(1+2+0)
9	4(3+1+0)	2(1+0+1)	4(4+0+0)	4(0+4+0)	5(4+0+1)	6(3+0+3)	5(4+1+0)	3(2+1+0)	2(1+1+0)	5(1+1+3)	2(0+2+0)	2(1+1+0)
10	6(5+1+0)	2(1+0+1)	4(4+0+0)	3(3+0+0)	3(1+1+1)	2(1+0+1)	4(3+1+0)	5(4+1+0)	5(5+0+0)	2(0+1+1)	2(0+0+2)	3(1+1+1)
11	5(4+1+0)	2(2+0+0)	3(1+1+1)	3(0+0+3)	2(1+0+1)	2(0+0+2)	6(3+2+1)	4(1+1+2)	3(3+0+0)	2(0+2+0)	5(3+0+2)	4(4+0+0)
12	6(4+1+1)	3(3+0+0)	2(1+1+0)	2(0+1+1)	5(4+0+1)	6(4+0+2)	4(2+1+1)	2(0+1+1)	2(1+1+0)	2(1+0+1)	4(3+1+0)	5(4+0+1)
13	5(3+2+0)	3(1+1+1)	4(3+1+0)	4(3+0+1)	4(4+0+0)	3(3+0+0)	5(4+0+1)	4(4+0+0)	2(0+2+0)	4(1+1+2)	3(1+1+1)	3(2+1+0)
14	2(1+1+0)	4(1+3+0)	4(1+3+0)	2(1+0+1)	6(3+2+1)	4(3+1+0)	2(1+1+0)	3(1+1+1)	3(1+1+1)	4(1+1+2)	5(4+1+0)	2(2+0+0)
15	5(4+0+1)	2(0+2+0)	2(1+1+0)	2(2+0+0)	5(5+0+0)	2(1+1+0)	4(3+1+0)	2(0+1+1)	6(5+0+1)	2(1+0+1)	5(3+1+1)	4(0+4+0)

Вопросы.

1. Дайте определение понятия тупика. Приведите пример.
2. Сформулируйте условия возникновения тупика.
3. Перечислите подходы к разрешению проблемы тупиков.
4. Алгоритм банкира.
5. Понятие надежных и не надежных состояний.
6. Алгоритм банкира для одного и нескольких видов ресурсов.
7. Недостатки алгоритма банкира

Лабораторная работа №8

Простейшие схемы управления памятью

Цель работы: *изучение алгоритмов управления памятью, разработка программы менеджера памяти.*

1. Теоретические сведения

1.1 Физическая организация памяти компьютера

Часть ОС, которая отвечает за управление памятью, называется менеджером памяти. Запоминающие устройства компьютера разделяют, как минимум, на два уровня: основную (главную, оперативную, физическую) и вторичную (внешнюю) память.

Основная память представляет собой упорядоченный массив однобайтовых ячеек, каждая из которых имеет свой уникальный адрес (номер). Вторичную память (это главным образом диски) также можно рассматривать как одномерное линейное адресное пространство, состоящее из последовательности байтов. В отличие от оперативной памяти, она является энергонезависимой, имеет существенно большую емкость и используется в качестве расширения основной памяти.
Разновидности памяти могут быть объединены в иерархию по убыванию времени доступа, возрастанию цены и увеличению емкости (рис. 1.).

Рис. 1. Иерархия памяти

Многоуровневую схему используют следующим образом. Информация, которая находится в памяти верхнего уровня, обычно хранится также на уровнях с большими номерами. Если процессор не обнаруживает нужную информацию на i-м уровне, он начинает искать ее на следующих уровнях. Когда нужная информация найдена, она переносится в более быстрые уровни.

1.2 Функции системы управления памятью

Чтобы обеспечить эффективный контроль использования памяти, ОС должна выполнять следующие функции:

- отображение адресного пространства процесса на конкретные области физической памяти;

- распределение памяти между конкурирующими процессами;
- контроль доступа к адресным пространствам процессов;
- выгрузка процессов (целиком или частично) во внешнюю память, когда в оперативной памяти недостаточно места;
- учет свободной и занятой памяти.

1.3 Простейшие схемы управления памятью

Первые ОС применяли очень простые методы управления памятью. Вначале каждый процесс пользователя должен был полностью поместиться в основной памяти, занимать непрерывную область памяти, а система принимала к обслуживанию дополнительные пользовательские процессы до тех пор, пока все они одновременно помещались в основной памяти. Затем появился "простой свопинг" (система по-прежнему размещает каждый процесс в основной памяти целиком, но иногда на основании некоторого критерия целиком сбрасывает образ некоторого процесса из основной памяти во внешнюю и заменяет его в основной памяти образом другого процесса). В настоящее время они применяются в учебных и научно-исследовательских модельных ОС, а также в ОС для встроенных (embedded) компьютеров.

1.4 Схема с фиксированными разделами

Самым простым способом управления оперативной памятью является ее предварительное (обычно на этапе генерации или в момент загрузки системы) разбиение на несколько разделов фиксированной величины. Поступающие процессы помещаются в тот или иной раздел. При этом происходит условное разбиение физического адресного пространства. Связывание логических и физических адресов процесса происходит на этапе его загрузки в конкретный раздел, иногда – на этапе компиляции.

Каждый раздел может иметь свою очередь процессов, а может существовать и глобальная очередь для всех разделов (см. рис. 2).

Подсистема управления памятью оценивает размер поступившего процесса, выбирает подходящий для него раздел, осуществляет загрузку процесса в этот раздел и настройку адресов.

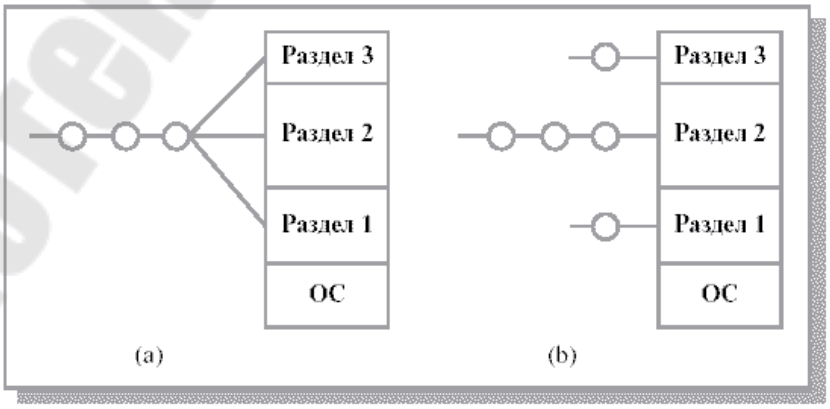

Рис. 2. Схема с фиксированными разделами: (a) – с общей очередью процессов, (b) – с отдельными очередями процессов

Очевидный недостаток этой схемы – число одновременно выполняемых процессов ограничено числом разделов.

Другим существенным недостатком является то, что предлагаемая схема сильно страдает от внутренней фрагментации – потери части памяти, выделенной процессу, но не используемой им. Фрагментация возникает потому, что процесс не полностью занимает выделенный ему раздел или потому, что некоторые разделы слишком малы для выполняемых пользовательских программ.

1.5 Один процесс в памяти

Частный случай схемы с фиксированными разделами – работа менеджера памяти однозадачной ОС. В памяти размещается один пользовательский процесс. Остается определить, где располагается пользовательская программа по отношению к ОС – в верхней части памяти, в нижней или в средней. Причем часть ОС может быть в ROM (например, BIOS, драйверы устройств). Главный фактор, влияющий на это решение, – расположение вектора прерываний, который обычно локализован в нижней части памяти, поэтому ОС также размещают в нижней. Примером такой организации может служить ОС MS-DOS.

Защита адресного пространства ОС от пользовательской программы может быть организована при помощи одного граничного регистра, содержащего адрес границы ОС.

1.6 Оверлейная структура

Так как размер логического адресного пространства процесса может быть больше, чем размер выделенного ему раздела (или больше, чем размер самого большого раздела), иногда используется техника, называемая оверлей (overlay) или организация структуры с перекрытием. Основная идея – держать в памяти только те инструкции программы, которые нужны в данный момент.

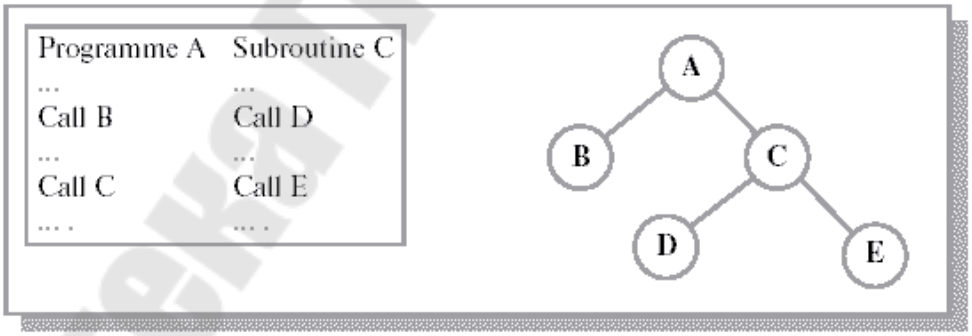

Рис. 3. Организация структуры с перекрытием. Можно поочередно загружать в память ветви A-B, A-C-D и A-C-E программы

Коды ветвей оверлейной структуры программы находятся на диске как абсолютные образы памяти и считываются драйвером оверлеев при необходимости. Для описания оверлейной структуры обычно используется специальный несложный язык (overlay description language). Совокупность файлов исполняемой программы дополняется файлом (обычно с расширением .odl), описывающим дерево вызовов внутри программы. Для примера, приведенного на рис. 3, текст этого файла может выглядеть так:
A-(B,C)
C-(D,E)

Синтаксис подобного файла может распознаваться загрузчиком. Привязка к физической памяти происходит в момент очередной загрузки одной из ветвей программы.

Оверлеи могут быть полностью реализованы на пользовательском уровне в системах с простой файловой структурой. ОС при этом лишь делает несколько больше операций ввода-вывода. Типовое решение – порождение линкером специальных команд, которые включают загрузчик каждый раз, когда требуется обращение к одной из перекрывающихся ветвей программы.

Тщательное проектирование оверлейной структуры отнимает много времени и требует знания устройства программы, ее кода, данных и языка описания оверлейной структуры. По этой причине применение оверлеев ограничено компьютерами с небольшим логическим адресным пространством. Как мы увидим в дальнейшем, проблема оверлейных сегментов, контролируемых программистом, отпадает благодаря появлению систем виртуальной памяти.

Заметим, что возможность организации структур с перекрытиями во многом обусловлена свойством локальности, которое позволяет хранить в памяти только ту информацию, которая необходима в конкретный момент вычислений.

1.7 Динамическое распределение. Свопинг

Имея дело с пакетными системами, можно обходиться фиксированными разделами и не использовать ничего более сложного. В системах с разделением времени возможна ситуация, когда память не в состоянии содержать все пользовательские процессы. Приходится прибегать к свопингу (swapping) – перемещению процессов из главной памяти на диск и обратно целиком. Частичная выгрузка процессов на диск осуществляется в системах со страничной организацией (paging) и будет рассмотрена ниже.

Выгруженный процесс может быть возвращен в то же самое адресное пространство или в другое. Это ограничение диктуется методом связывания. Для схемы связывания на этапе выполнения можно загрузить процесс в другое место памяти.

Свопинг не имеет непосредственного отношения к управлению памятью, скорее он связан с подсистемой планирования процессов. Очевидно, что свопинг увеличивает время переключения контекста. Время выгрузки может быть сокращено за счет организации специально отведенного пространства на диске (раздел для свопинга). Обмен с диском при этом осуществляется блоками большего размера, то есть быстрее, чем через стандартную файловую систему. Во многих версиях Unix свопинг начинает работать только тогда, когда возникает необходимость в снижении загрузки системы.

1.8 Схема с переменными разделами

В принципе, система свопинга может базироваться на фиксированных разделах. Более эффективной, однако, представляется схема динамического распределения или схема с переменными разделами, которая может использоваться и в тех случаях, когда все процессы целиком помещаются в памяти, то есть в отсутствие свопинга. В этом случае вначале вся память свободна и не разделена заранее на разделы. Вновь поступающей задаче выделяется строго необходимое количество памяти, не более. После выгрузки процесса память временно освобождается. По истечении некоторого времени память представляет собой переменное число разделов разного размера (рис. 4). Смежные свободные участки могут быть объединены.

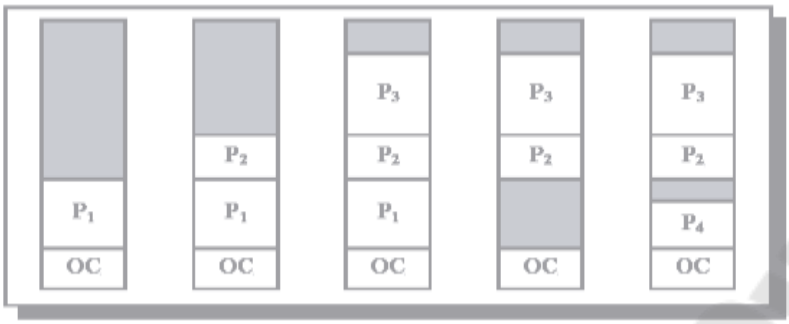

Рис. 4. Динамика распределения памяти между процессами (серым цветом показана неиспользуемая память)

В какой раздел помещать процесс? Наиболее распространены три стратегии:

- стратегия первого подходящего (First fit). Процесс помещается в первый подходящий по размеру раздел;
- стратегия наиболее подходящего (Best fit). Процесс помещается в тот раздел, где после его загрузки останется меньше всего свободного места;
- стратегия наименее подходящего (Worst fit). При помещении в самый большой раздел в нем остается достаточно места для возможного размещения еще одного процесса.

Моделирование показало, что доля полезно используемой памяти в первых двух случаях больше, при этом первый способ несколько быстрее. Попутно заметим, что перечисленные стратегии широко применяются и другими компонентами ОС, например для размещения файлов на диске.

Типовой цикл работы менеджера памяти состоит в анализе запроса на выделение свободного участка (раздела), выборе его среди имеющихся в соответствии с одной из стратегий (первого подходящего, наиболее подходящего и наименее подходящего), загрузке процесса в выбранный раздел и последующих изменениях таблиц свободных и занятых областей. Аналогичная корректировка необходима и после завершения процесса. Связывание адресов может осуществляться на этапах загрузки и выполнения.

Этот метод более гибок по сравнению с методом фиксированных разделов, однако ему присуща внешняя фрагментация – наличие большого числа участков неиспользуемой памяти, не выделенной ни одному процессу. Выбор стратегии размещения процесса между первым подходящим и наиболее подходящим слабо влияет на величину фрагментации. Любопытно, что метод наиболее подходящего может оказаться наихудшим, так как он оставляет множество мелких незанятых блоков.

Статистический анализ показывает, что пропадает в среднем 1/3 памяти! Это известное правило 50% (два соседних свободных участка в отличие от двух соседних процессов могут быть объединены).

Одно из решений проблемы внешней фрагментации – организовать сжатие, то есть перемещение всех занятых (свободных) участков в сторону возрастания (убывания) адресов, так, чтобы вся свободная память образовала непрерывную область. Этот метод иногда называют схемой с перемещаемыми разделами. В идеале фрагментация после сжатия должна отсутствовать. Сжатие, однако, является дорогостоящей процедурой, алгоритм выбора оптимальной стратегии сжатия очень труден и, как правило, сжатие осуществляется в комбинации с выгрузкой и загрузкой по другим адресам.

1.9 Страничная память

Описанные выше схемы недостаточно эффективно используют память, поэтому в современных схемах управления памятью не принято размещать процесс в оперативной памяти одним непрерывным блоком.

В самом простом и наиболее распространенном случае страничной организации памяти (или paging) как логическое адресное пространство, так и физическое представляются состоящими из наборов блоков или страниц одинакового размера. При этом образуются логические страницы (page), а соответствующие единицы в физической памяти называют физическими страницами или страничными кадрами (page frames). Страницы (и страничные кадры) имеют фиксированную длину, обычно являющуюся степенью числа 2, и не могут перекрываться. Каждый кадр содержит одну страницу данных. При такой организации внешняя фрагментация отсутствует, а потери из-за внутренней фрагментации, поскольку процесс занимает целое число страниц, ограничены частью последней страницы процесса.

Логический адрес в страничной системе – упорядоченная пара (p,d), где p – номер страницы в виртуальной памяти, а d – смещение в рамках страницы p, на которой размещается адресуемый элемент. Заметим, что разбиение адресного пространства на страницы осуществляется вычислительной системой незаметно для программиста. Поэтому адрес является двумерным лишь с точки зрения операционной системы, а с точки зрения программиста адресное пространство процесса остается линейным.

Описываемая схема позволяет загрузить процесс, даже если нет непрерывной области кадров, достаточной для размещения процесса целиком. Но одного базового регистра для осуществления трансляции адреса в данной схеме недостаточно. Система отображения логических адресов в физические сводится к системе отображения логических страниц в физические и представляет собой таблицу страниц, которая хранится в оперативной памяти. Иногда говорят, что таблица страниц – это кусочно-линейная функция отображения, заданная в табличном виде.

Интерпретация логического адреса показана на рис. 5. Если выполняемый процесс обращается к логическому адресу v = (p,d), механизм отображения ищет номер страницы p в таблице страниц и определяет, что эта страница находится в страничном кадре p', формируя реальный адрес из p' и d.

Таблица страниц (page table) адресуется при помощи специального регистра процессора и позволяет определить номер кадра по логическому адресу. Помимо этой основной задачи, при помощи атрибутов, записанных в строке таблицы страниц, можно организовать контроль доступа к конкретной странице и ее защиту.

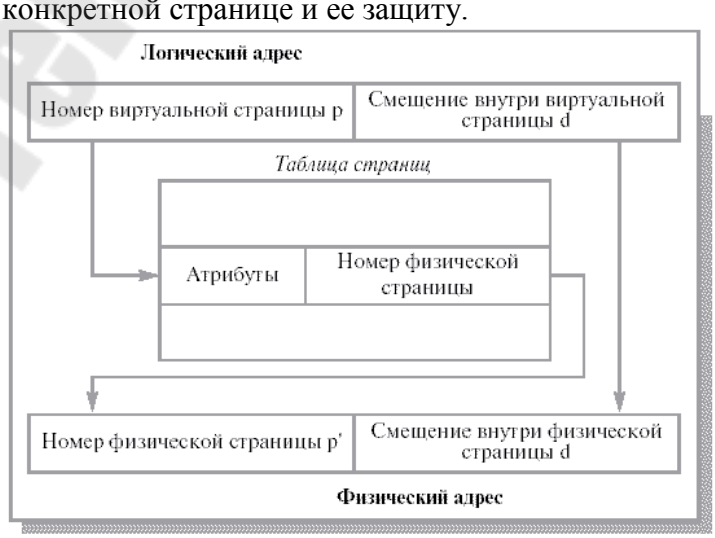

Рис. 5. Связь логического и физического адресов при страничной организации памяти

Отметим еще раз различие точек зрения пользователя и системы на используемую память. С точки зрения пользователя, его память – единое непрерывное пространство, содержащее только одну программу. Реальное отображение скрыто от пользователя и контролируется ОС. Заметим, что процессу пользователя чужая память недоступна. Он не имеет возможности адресовать память за пределами своей таблицы страниц, которая включает только его собственные страницы.

Для управления физической памятью ОС поддерживает структуру таблицы кадров. Она имеет одну запись на каждый физический кадр, показывающий его состояние.

Отображение адресов должно быть осуществлено корректно даже в сложных случаях и обычно реализуется аппаратно. Для ссылки на таблицу процессов используется специальный регистр. При переключении процессов необходимо найти таблицу страниц нового процесса, указатель на которую входит в контекст процесса.

1.10 Сегментная и сегментно-страничная организация памяти

Существуют две другие схемы организации управления памятью: сегментная и сегментно-страничная. Сегменты, в отличие от страниц, могут иметь переменный размер. Идея сегментации изложена во введении. При сегментной организации виртуальный адрес является двумерным как для программиста, так и для операционной системы, и состоит из двух полей – номера сегмента и смещения внутри сегмента. Подчеркнем, что в отличие от страничной организации, где линейный адрес преобразован в двумерный операционной системой для удобства отображения, здесь двумерность адреса является следствием представления пользователя о процессе не в виде линейного массива байтов, а как набор сегментов переменного размера (данные, код, стек...).

Программисты, пишущие на языках низкого уровня, должны иметь представление о сегментной организации, явным образом меняя значения сегментных регистров (это хорошо видно по текстам программ, написанных на Ассемблере). Логическое адресное пространство – набор сегментов. Каждый сегмент имеет имя, размер и другие параметры (уровень привилегий, разрешенные виды обращений, флаги присутствия). В отличие от страничной схемы, где пользователь задает только один адрес, который разбивается на номер страницы и смещение прозрачным для программиста образом, в сегментной схеме пользователь специфицирует каждый адрес двумя величинами: именем сегмента и смещением.

Каждый сегмент – линейная последовательность адресов, начинающаяся с 0. Максимальный размер сегмента определяется разрядностью процессора (при 32-разрядной адресации это 232 байт или 4 Гбайт). Размер сегмента может меняться динамически (например, сегмент стека). В элементе таблицы сегментов помимо физического адреса начала сегмента обычно содержится и длина сегмента. Если размер смещения в виртуальном адресе выходит за пределы размера сегмента, возникает исключительная ситуация.

Логический адрес – упорядоченная пара v=(s,d), номер сегмента и смещение внутри сегмента.

В системах, где сегменты поддерживаются аппаратно, эти параметры обычно хранятся в таблице дескрипторов сегментов, а программа обращается к этим дескрипторам по номерам-селекторам. При этом в контекст каждого процесса входит набор сегментных регистров, содержащих селекторы текущих сегментов кода, стека, данных и т. д. и

определяющих, какие сегменты будут использоваться при разных видах обращений к памяти. Это позволяет процессору уже на аппаратном уровне определять допустимость обращений к памяти, упрощая реализацию защиты информации от повреждения и несанкционированного доступа.

Аппаратная поддержка сегментов распространена мало (главным образом на процессорах Intel). В большинстве ОС сегментация реализуется на уровне, не зависящем от аппаратуры.

Хранить в памяти сегменты большого размера целиком так же неудобно, как и хранить процесс непрерывным блоком. Напрашивается идея разбиения сегментов на страницы. При сегментно-страничной организации памяти происходит двухуровневая трансляция виртуального адреса в физический. В этом случае логический адрес состоит из трех полей: номера сегмента логической памяти, номера страницы внутри сегмента и смещения внутри страницы. Соответственно, используются две таблицы отображения – таблица сегментов, связывающая номер сегмента с таблицей страниц, и отдельная таблица страниц для каждого сегмента.

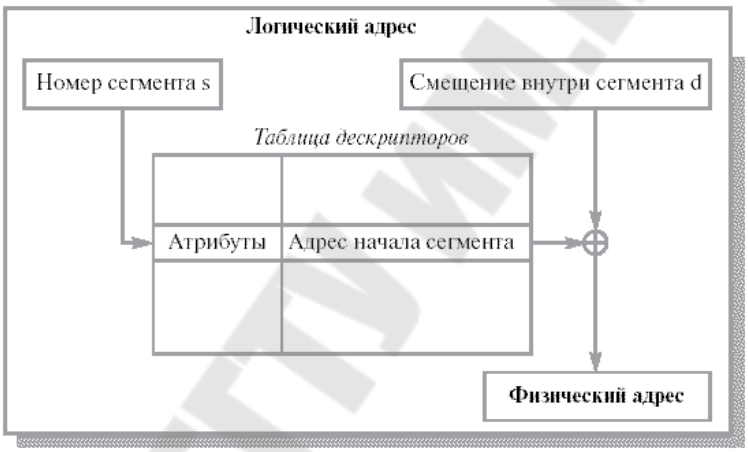

Рис. 6. Преобразование логического адреса при сегментной организации памяти

Сегментно-страничная и страничная организация памяти позволяет легко организовать совместное использование одних и тех же данных и программного кода разными задачами. Для этого различные логические блоки памяти разных процессов отображают в один и тот же блок физической памяти, где размещается разделяемый фрагмент кода или данных.

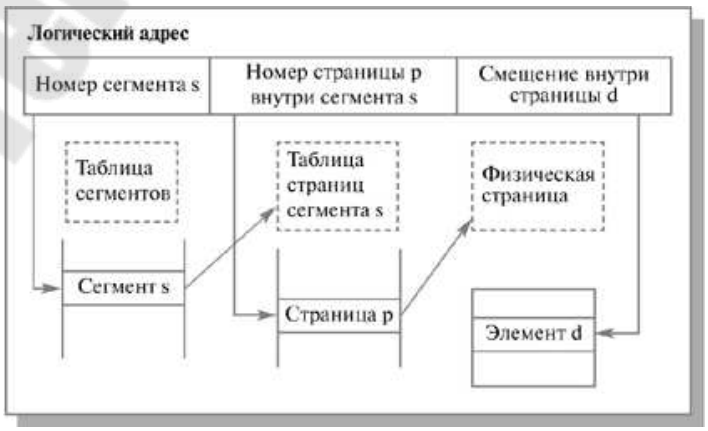

Рис. 7. Упрощенная схема формирования физического адреса при сегментно-страничной организации памяти

2. Задание

Разработать программу, реализующую заданный алгоритм выделения памяти.
Менеджер памяти должен:

1. По запросу процесса выделять память, согласно заданного алгоритма (таблица). На экран должна выводиться следующая информация о состоянии памяти: объем памяти, объём свободной памяти, размер наибольшего свободного блока, количество запросов на выделение памяти, количество удовлетворённых запросов (%).
2. Для выделения памяти указывается имя процесса и размер блока. После нажатия на кнопку «ДОБАВИТЬ» память выделяется или выдаётся сообщение о невозможности выделения.
3. Удалять из памяти заданный блок или все блоки заданного процесса (по нажатию кнопки «УДАЛИТЬ»). Указывается номер удаляемого блока и имя процесса.
4. Реализовать возможность последовательной записи/чтения информации в/из выделенную память по логическому адресу. Вывести физического адреса ячейки памяти, в которую была осуществлена запись.
5. Организовывать циклическое выделение и освобождение памяти. При этом случайным образом задается количество выделяемых блоков и их размер.

Таблица вариантов зданий

Вариант	Задание
01	**Схема с фиксированными разделами с общей очередью процессов.** Количество разделов и их размер задать с клавиатуры.
02	**Схема с фиксированными разделами с отдельными очередями процессов.** Количество разделов и их размер задать с клавиатуры.
03	**Свопинг.** Выгружается процесс, занимающий наименьший объём памяти.
04	**Свопинг.** Выгружается процесс, занимающий наибольший объём памяти.
05	**Свопинг.** Выгружается процесс, с наименьшим приоритетом.
06	**Сегментная схема организации памяти.** Для каждого процесса создавать 3 сегмента: сегмент стека (1 кб), сегмент кода (1 кб) и сегмент данных.
07	**Страничная организация памяти.** Размер страницы 2^n
08	**Свопинг.** Выгружается процесс, бывший в состоянии исполнение наиболее давно.
09	**Схема с переменными разделами.** Стратегия первого подходящего.
10	**Схема с переменными разделами.** Стратегия наиболее подходящего.
11	**Схема с переменными разделами.** Стратегия наименее подходящего.
12	**Схема с переменными разделами.** Случайный выбор раздела.
13	**Страничная организация памяти.** Размер страницы 3^k
14	**Сегментная схема организации памяти.** Для каждого процесса создавать 2 сегмента: сегмент кода (3 кб) и сегмент данных.
15	**Сегментно-страничная организации памяти.** Для каждого

	процесса создавать 2 сегмента: сегмент кода (1 кб) и сегмент данных. Размер страниц выбрать равным 256 байтам.

Вопросы к защите

1. Физическая организация памяти компьютера.
2. Функции системы управления памятью.
3. Схемы управления памятью. Схема с фиксированными разделами.
4. Схемы управления памятью. Схема с фиксированными разделами. Один процесс в памяти.
5. Схемы управления памятью. Схемы управления памятью. Оверлейная структура.
6. Схемы управления памятью. Динамическое распределение. Свопинг.
7. Схемы управления памятью. Схема с переменными разделами.
8. Схемы управления памятью. Страничная память.
9. Схемы управления памятью. Сегментная организация памяти.
10. Схемы управления памятью. Сегментно – страничная организация памяти.

Лабораторная работа №9

Управление виртуальной памятью.

Алгоритмы замещения страниц.

1. Теоретические сведения

1.1 Понятие виртуальной памяти

Разработчикам программного обеспечения часто приходится решать проблему размещения в памяти больших программ, размер которых превышает объем доступной оперативной памяти. Один из вариантов решения данной проблемы – организация структур с перекрытием – рассмотрен в предыдущей лекции. При этом предполагалось активное участие программиста в процессе формирования перекрывающихся частей программы. Развитие архитектуры компьютеров и расширение возможностей операционной системы по управлению памятью позволило переложить решение этой задачи на компьютер. Одним из главных достижений стало появление виртуальной памяти (virtual memory). Впервые она была реализована в 1959 г. на компьютере «Атлас», разработанном в Манчестерском университете.

Суть концепции виртуальной памяти заключается в следующем. Информация, с которой работает активный процесс, должна располагаться в оперативной памяти. В схемах виртуальной памяти у процесса создается иллюзия того, что вся необходимая ему информация имеется в основной памяти. Для этого, во-первых, занимаемая процессом память разбивается на несколько частей, например страниц. Во-вторых, логический адрес (логическая страница), к которому обращается процесс, динамически транслируется в физический адрес (физическую страницу). И наконец, в тех случаях, когда страница, к которой обращается процесс, не находится в физической памяти, нужно организовать ее подкачку с диска. Для контроля наличия страницы в памяти вводится специальный бит присутствия, входящий в состав атрибутов страницы в таблице страниц.

Таким образом, в наличии всех компонентов процесса в основной памяти необходимости нет. Важным следствием такой организации является то, что размер памяти, занимаемой процессом, может быть больше, чем размер оперативной памяти. Принцип локальности обеспечивает этой схеме нужную эффективность.

Возможность выполнения программы, находящейся в памяти лишь частично, имеет ряд вполне очевидных преимуществ.
Программа не ограничена объемом физической памяти. Упрощается разработка программ, поскольку можно задействовать большие виртуальные пространства, не заботясь о размере используемой памяти.
Поскольку появляется возможность частичного помещения программы (процесса) в память и гибкого перераспределения памяти между программами, можно разместить в памяти больше программ, что увеличивает загрузку процессора и пропускную способность системы.
Объем ввода-вывода для выгрузки части программы на диск может быть меньше, чем в варианте классического свопинга, в итоге каждая программа будет работать быстрее.
Напомним, что в системах с виртуальной памятью те адреса, которые генерирует программа (логические адреса), называются виртуальными, и они формируют виртуальное адресное пространство. Термин «виртуальная память» означает, что программист имеет

дело с памятью, отличной от реальной, размер которой потенциально больше, чем размер оперативной памяти.

Хотя известны и чисто программные реализации виртуальной памяти, это направление получило наиболее широкое развитие после соответствующей аппаратной поддержки.

Следует отметить, что оборудование компьютера принимает участие в трансляции адреса практически во всех схемах управления памятью. Но в случае виртуальной памяти это становится более сложным вследствие разрывности отображения и многомерности логического адресного пространства. Может быть, наиболее существенным вкладом аппаратуры в реализацию описываемой схемы является автоматическая генерация исключительных ситуаций при отсутствии в памяти нужных страниц (page fault).

Любая из трех ранее рассмотренных схем управления памятью – страничной, сегментной и сегментно-страничной – пригодна для организации виртуальной памяти.

1.2 Страничная виртуальная память

Как и в случае простой страничной организации, страничная виртуальная память и физическая память представляются состоящими из наборов блоков или страниц одинакового размера. Виртуальные адреса делятся на страницы (page), соответствующие единицы в физической памяти образуют страничные кадры (page frames), а в целом система поддержки страничной виртуальной памяти называется пейджингом (paging). Передача информации между памятью и диском всегда осуществляется целыми страницами.

После разбиения менеджером памяти виртуального адресного пространства на страницы виртуальный адрес преобразуется в упорядоченную пару (p,d), где p – номер страницы в виртуальной памяти, а d – смещение в рамках страницы p, внутри которой размещается адресуемый элемент. Процесс может выполняться, если его текущая страница находится в оперативной памяти. Если текущей страницы в главной памяти нет, она должна быть переписана (подкачана) из внешней памяти. Поступившую страницу можно поместить в любой свободный страничный кадр.

Поскольку число виртуальных страниц велико, таблица страниц принимает специфический вид (см. раздел «Структура таблицы страниц»), структура записей становится более сложной, среди атрибутов страницы появляются биты присутствия, модификации и другие управляющие биты.

При отсутствии страницы в памяти в процессе выполнения команды возникает исключительная ситуация, называемая страничное нарушение (page fault) или страничный отказ. Обработка страничного нарушения заключается в том, что выполнение команды прерывается, затребованная страница подкачивается из конкретного места вторичной памяти в свободный страничный кадр физической памяти и попытка выполнения команды повторяется. При отсутствии свободных страничных кадров на диск выгружается редко используемая страница.

1.3 Исключительные ситуации при работе с памятью

Отображение виртуального адреса в физический осуществляется при помощи таблицы страниц. Для каждой виртуальной страницы запись в таблице страниц содержит номер соответствующего страничного кадра в оперативной памяти, а также атрибуты страницы для контроля обращений к памяти.

Что же происходит, когда нужной страницы в памяти нет или операция обращения к памяти недопустима? Естественно, что операционная система должна быть как-то оповещена о происшедшем. Обычно для этого используется механизм исключительных ситуаций (exceptions). При попытке выполнить подобное обращение к виртуальной странице возникает исключительная ситуация "страничное нарушение" (page fault), приводящая к вызову специальной последовательности команд для обработки конкретного вида страничного нарушения.

Страничное нарушение может происходить в самых разных случаях: при отсутствии страницы в оперативной памяти, при попытке записи в страницу с атрибутом "только чтение" или при попытке чтения или записи страницы с атрибутом "только выполнение". В любом из этих случаев вызывается обработчик страничного нарушения, являющийся частью операционной системы. Ему обычно передается причина возникновения исключительной ситуации и виртуальный адрес, обращение к которому вызвало нарушение.

1.4 Стратегии управления страничной памятью

Программное обеспечение подсистемы управления памятью связано с реализацией следующих стратегий:

Стратегия выборки (fetch policy) - в какой момент следует переписать страницу из вторичной памяти в первичную. Существует два основных варианта выборки - по запросу и с упреждением. Алгоритм выборки по запросу вступает в действие в тот момент, когда процесс обращается к отсутствующей странице, содержимое которой находится на диске. Его реализация заключается в загрузке страницы с диска в свободную физическую страницу и коррекции соответствующей записи таблицы страниц.

Алгоритм выборки с упреждением осуществляет опережающее чтение, то есть кроме страницы, вызвавшей исключительную ситуацию, в память также загружается несколько страниц, окружающих ее (обычно соседние страницы располагаются во внешней памяти последовательно и могут быть считаны за одно обращение к диску). Такой алгоритм призван уменьшить накладные расходы, связанные с большим количеством исключительных ситуаций, возникающих при работе со значительными объемами данных или кода; кроме того, оптимизируется работа с диском.

Стратегия размещения (placement policy) - в какой участок первичной памяти поместить поступающую страницу. В системах со страничной организацией все просто - в любой свободный страничный кадр. В случае систем с сегментной организацией необходима стратегия, аналогичная стратегии с динамическим распределением.

Стратегия замещения (replacement policy) - какую страницу нужно вытолкнуть во внешнюю память, чтобы освободить место в оперативной памяти. Разумная стратегия замещения, реализованная в соответствующем алгоритме замещения страниц, позволяет хранить в памяти самую необходимую информацию и тем самым снизить частоту

страничных нарушений. Замещение должно происходить с учетом выделенного каждому процессу количества кадров. Кроме того, нужно решить, должна ли замещаемая страница принадлежать процессу, который инициировал замещение, или она должна быть выбрана среди всех кадров основной памяти.

1.5 Алгоритмы замещения страниц

Итак, наиболее ответственным действием менеджера памяти является выделение кадра оперативной памяти для размещения в ней виртуальной страницы, находящейся во внешней памяти. Напомним, что мы рассматриваем ситуацию, когда размер виртуальной памяти для каждого процесса может существенно превосходить размер основной памяти. Это означает, что при выделении страницы основной памяти с большой вероятностью не удастся найти свободный страничный кадр. В этом случае операционная система в соответствии с заложенными в нее критериями должна:

- найти некоторую занятую страницу основной памяти;
- переместить в случае
- надобности ее содержимое во внешнюю память;
- переписать в этот страничный кадр содержимое нужной виртуальной страницы из внешней памяти;
- должным образом модифицировать необходимый элемент соответствующей таблицы страниц;
- продолжить выполнение процесса, которому эта виртуальная страница понадобилась.

Заметим, что при замещении приходится дважды передавать страницу между основной и вторичной памятью. Процесс замещения может быть оптимизирован за счет использования бита модификации (один из атрибутов страницы в таблице страниц). Бит модификации устанавливается компьютером, если хотя бы один байт был записан на страницу. При выборе кандидата на замещение проверяется бит модификации. Если бит не установлен, нет необходимости переписывать данную страницу на диск, ее копия на диске уже имеется. Подобный метод также применяется к read-only-страницам, они никогда не модифицируются. Эта схема уменьшает время обработки page fault.

Существует большое количество разнообразных алгоритмов замещения страниц. Все они делятся на локальные и глобальные. Локальные алгоритмы, в отличие от глобальных, распределяют фиксированное или динамически настраиваемое число страниц для каждого процесса. Когда процесс израсходует все предназначенные ему страницы, система будет удалять из физической памяти одну из его страниц, а не из страниц других процессов. Глобальный же алгоритм замещения в случае возникновения исключительной ситуации удовлетворится освобождением любой физической страницы, независимо от того, какому процессу она принадлежала.

Глобальные алгоритмы имеют ряд недостатков. Во-первых, они делают одни процессы чувствительными к поведению других процессов. Например, если один процесс в системе одновременно использует большое количество страниц памяти, то все остальные приложения будут в результате ощущать сильное замедление из-за недостатка кадров памяти для своей работы. Во-вторых, некорректно работающее приложение может подорвать работу всей системы (если, конечно, в системе не предусмотрено ограничение на размер памяти, выделяемой процессу), пытаясь захватить больше памяти. Поэтому в многозадачной системе иногда приходится использовать более сложные локальные

алгоритмы. Применение локальных алгоритмов требует хранения в операционной системе списка физических кадров, выделенных каждому процессу. Этот список страниц иногда называют резидентным множеством процесса. В одном из следующих разделов рассмотрен вариант алгоритма подкачки, основанный на приведении резидентного множества в соответствие так называемому рабочему набору процесса.

Эффективность алгоритма обычно оценивается на конкретной последовательности ссылок к памяти, для которой подсчитывается число возникающих page faults. Эта последовательность называется строкой обращений (reference string). Мы можем генерировать строку обращений искусственным образом при помощи датчика случайных чисел или трассируя конкретную систему. Последний метод дает слишком много ссылок, для уменьшения числа которых можно сделать две вещи:
для конкретного размера страниц можно запоминать только их номера, а не адреса, на которые идет ссылка; несколько подряд идущих ссылок на одну страницу можно фиксировать один раз.

Как уже говорилось, большинство процессоров имеют простейшие аппаратные средства, позволяющие собирать некоторую статистику обращений к памяти. Эти средства обычно включают два специальных флага на каждый элемент таблицы страниц. Флаг ссылки (reference бит) автоматически устанавливается, когда происходит любое обращение к этой странице, а уже рассмотренный выше флаг изменения (modify бит) устанавливается, если производится запись в эту страницу. Операционная система периодически проверяет установку таких флагов, для того чтобы выделить активно используемые страницы, после чего значения этих флагов сбрасываются.

1.6 Алгоритм FIFO. Выталкивание первой пришедшей страницы

Простейший алгоритм. Каждой странице присваивается временная метка. Реализуется это просто созданием очереди страниц, в конец которой страницы попадают, когда загружаются в физическую память, а из начала берутся, когда требуется освободить память. Для замещения выбирается старейшая страница. К сожалению, эта стратегия с достаточной вероятностью будет приводить к замещению активно используемых страниц, например страниц кода текстового процессора при редактировании файла. Заметим, что при замещении активных страниц все работает корректно, но page fault происходит немедленно.

На первый взгляд кажется очевидным, что чем больше в памяти страничных кадров, тем реже будут иметь место page faults. Удивительно, но это не всегда так. Как установил Билэди с коллегами, определенные последовательности обращений к страницам в действительности приводят к увеличению числа страничных нарушений при увеличении кадров, выделенных процессу. Это явление носит название "аномалии Билэди" или "аномалии FIFO". Система с тремя кадрами (9 faults) оказывается более производительной, чем с четырьмя кадрами (10 faults), для строки обращений к памяти 012301401234 при выборе стратегии FIFO.

Рис. 10.1. Аномалия Билэди: (a) - FIFO с тремя страничными кадрами; (b) - FIFO с четырьмя страничными кадрами

Аномалию Билэди следует считать скорее курьезом, чем фактором, требующим серьезного отношения, который иллюстрирует сложность ОС, где интуитивный подход не всегда приемлем.

1.7 Оптимальный алгоритм (OPT)

Одним из последствий открытия аномалии Билэди стал поиск оптимального алгоритма, который при заданной строке обращений имел бы минимальную частоту page faults среди всех других алгоритмов. Такой алгоритм был найден. Он прост: замещай страницу, которая не будет использоваться в течение самого длительного периода времени.

Каждая страница должна быть помечена числом инструкций, которые будут выполнены, прежде чем на эту страницу будет сделана первая ссылка. Выталкиваться должна страница, для которой это число наибольшее.

Этот алгоритм легко описать, но реализовать невозможно. ОС не знает, к какой странице будет следующее обращение. (Ранее такие проблемы возникали при планировании процессов - алгоритм SJF).

Зато мы можем сделать вывод, что для того, чтобы алгоритм замещения был максимально близок к идеальному алгоритму, система должна как можно точнее предсказывать обращения процессов к памяти. Данный алгоритм применяется для оценки качества реализуемых алгоритмов.

1.8 Выталкивание дольше всего не использовавшейся страницы. Алгоритм LRU

Одним из приближений к алгоритму OPT является алгоритм, исходящий из эвристического правила, что недавнее прошлое - хороший ориентир для прогнозирования ближайшего будущего.

Ключевое отличие между FIFO и оптимальным алгоритмом заключается в том, что один смотрит назад, а другой вперед. Если использовать прошлое для аппроксимации будущего,

имеет смысл замещать страницу, которая не использовалась в течение самого долгого времени. Такой подход называется least recently used алгоритм (LRU). Работа алгоритма проиллюстрирована на рис. рис. 10.2. Сравнивая рис. 10.1 b и 10.2, можно увидеть, что использование LRU алгоритма позволяет сократить количество страничных нарушений.

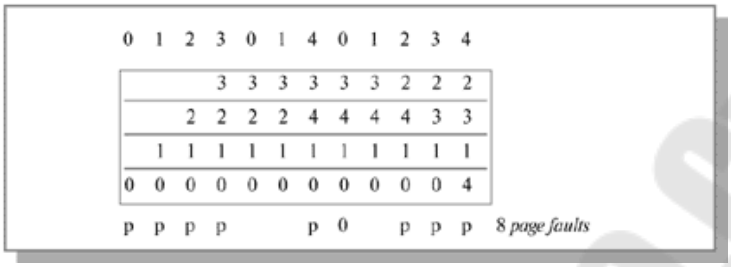

Рис. 10.2. Пример работы алгоритма LRU

LRU - хороший, но труднореализуемый алгоритм. Необходимо иметь связанный список всех страниц в памяти, в начале которого будут хранится недавно использованные страницы. Причем этот список должен обновляться при каждом обращении к памяти. Много времени нужно и на поиск страниц в таком списке.

В [Таненбаум, 2002] рассмотрен вариант реализации алгоритма LRU со специальным 64-битным указателем, который автоматически увеличивается на единицу после выполнения каждой инструкции, а в таблице страниц имеется соответствующее поле, в которое заносится значение указателя при каждой ссылке на страницу. При возникновении page fault выгружается страница с наименьшим значением этого поля.

Как оптимальный алгоритм, так и LRU не страдают от аномалии Билэди. Существует класс алгоритмов, для которых при одной и той же строке обращений множество страниц в памяти для n кадров всегда является подмножеством страниц для n+1 кадра. Эти алгоритмы не проявляют аномалии Билэди и называются стековыми (stack) алгоритмами.

1.9 Выталкивание редко используемой страницы. Алгоритм NFU

Поскольку большинство современных процессоров не предоставляют соответствующей аппаратной поддержки для реализации алгоритма LRU, хотелось бы иметь алгоритм, достаточно близкий к LRU, но не требующий специальной поддержки.

Программная реализация алгоритма, близкого к LRU, - алгоритм NFU(Not Frequently Used).

Для него требуются программные счетчики, по одному на каждую страницу, которые сначала равны нулю. При каждом прерывании по времени (а не после каждой инструкции) операционная система сканирует все страницы в памяти и у каждой страницы с установленным флагом обращения увеличивает на единицу значение счетчика, а флаг обращения сбрасывает.

Таким образом, кандидатом на освобождение оказывается страница с наименьшим значением счетчика, как страница, к которой реже всего обращались. Главный недостаток алгоритма NFU состоит в том, что он ничего не забывает. Например, страница, к которой очень часто обращались в течение некоторого времени, а потом обращаться перестали, все равно не будет удалена из памяти, потому что ее счетчик содержит большую величину. Например, в многопроходных компиляторах страницы, которые активно использовались во

время первого прохода, могут надолго сохранить большие значения счетчика, мешая загрузке полезных в дальнейшем страниц.

К счастью, возможна небольшая модификация алгоритма, которая позволяет ему "забывать". Достаточно, чтобы при каждом прерывании по времени содержимое счетчика сдвигалось вправо на 1 бит, а уже затем производилось бы его увеличение для страниц с установленным флагом обращения.

Другим, уже более устойчивым недостатком алгоритма является длительность процесса сканирования таблиц страниц.

1.10 Алгоритм Second-Chance

Например, алгоритм Second-Chance - модификация алгоритма FIFO, которая позволяет избежать потери часто используемых страниц с помощью анализа флага обращений (бита ссылки) для самой старой страницы. Если флаг установлен, то страница, в отличие от алгоритма FIFO, не выталкивается, а ее флаг сбрасывается, и страница переносится в конец очереди. Если первоначально флаги обращений были установлены для всех страниц (на все страницы ссылались), алгоритм Second-Chance превращается в алгоритм FIFO. Данный алгоритм использовался в Multics и BSD Unix.

2. Задание

Разработать программу, реализующую заданный алгоритм замещения страниц в памяти. Менеджер памяти должен:

6. Разбивать память заданного размера на указанное количество страниц. На экран должна выводиться следующая информация о состоянии памяти: объем памяти, число страниц, число свободных страниц (%), размер страницы;
7. Размещать в памяти страницу заданного процесса, с замещением занятой по заданному алгоритму (по нажатию кнопки «ДОБАВИТЬ»). Для размещения страницы в памяти, указывается имя процесса и ее номер (вводятся отдельно). Например: Pro 3. После нажатия на кнопку «ДОБАВИТЬ» страница размещается в свободной странице памяти. Если задано **глобальное размещение** (см. вариант задания), то выбирается любая не занятая страница. При **локальном размещении** страница размещается только среди виртуальных страниц выделенных этому процессу. Выделение страниц в памяти выполняется при первом ее занесении процесса в память. Алгоритм замещения выполняется **только при отсутствии свободных страниц** под процесс.;
8. Удалять из памяти заданную страницу или все страницы заданного процесса (по нажатию кнопки «УДАЛИТЬ»). Указывается номер удаляемой страницы в **памяти**;
9. Организовывать циклическое обращение к страницам размещенным в памяти по нажатию на кнопку. При этом случайным образом задается количество обращений к страницам (диапазон 1..10). Для каждого обращения генерируется, случайным образом, номер страницы из диапазона [0; количество страниц памяти]. При обращении к странице в зависимости, от варианта, увеличивается ее внутренний счетчик обращений или устанавливается флаг обращения

Таблица вариантов зданий

Вариант	Задание
01	**Глобальное размещение.** Алгоритм замещения – Random.

	Замещается случайная страница.
02	**Глобальное размещение.** Алгоритм замещения – FIFO. Реализуется очередь страниц в конец которой попадают страницы размещенные в памяти, а из начала берутся для замещения.
03	**Глобальное размещение.** Алгоритм замещения – LRU. Замещается страница с наименьшим количеством обращений.
04	**Глобальное размещение.** Алгоритм замещения – LRU CLOCK. Существует глобальный счетчик обращений к страницам. При каждом обращении к станице в ее внутренний регистр заносится значение глобального счетчика. Выгружается страница с наименьшим значением счетчика.
05	**Глобальное размещение.** Алгоритм замещения – NFU. По таймеру 2 (с большей задержкой чем T1) происходит опрос всех страниц, для каждой страницы с установленным флагом обращений увеличивается соответствующий счетчик, флаг обращений сбрасывается. Выгружается страница с наименьшим значением счетчика.
06	**Локальное размещение.** Алгоритм замещения – Random. Замещается случайная страница.
07	**Локальное размещение.** Алгоритм замещения – FIFO. Реализуется очередь страниц в конец которой попадают страницы размещенные в памяти, а из начала берутся для замещения.
08	**Локальное размещение.** Алгоритм замещения – LRU. Замещается страница с наименьшим количеством обращений.
09	**Локальное размещение.** Алгоритм замещения – LRU CLOCK. Существует глобальный счетчик обращений к страницам. При каждом обращении к станице в ее внутренний регистр заносится значение глобального счетчика. Выгружается страница с наименьшим значением счетчика.
10	**Локальное размещение.** Алгоритм замещения – NFU. По таймеру 2 (с большей задержкой чем T1) происходит опрос всех страниц, для каждой страницы с установленным флагом обращений увеличивается соответствующий счетчик, флаг обращений сбрасывается. Выгружается страница с наименьшим значением счетчика.
11	**Локальное размещение с динамическим увеличением количества выделенных страниц при количестве запросов к страницам больше установленного порога.** Алгоритм замещения – Random. Замещается случайная страница.
12	**Локальное размещение с динамическим увеличением количества выделенных страниц при количестве запросов к страницам больше установленного порога.** Алгоритм замещения – FIFO. Реализуется очередь страниц в конец которой попадают страницы размещенные в памяти, а из начала берутся для замещения.
13	**Локальное размещение с динамическим увеличением количества выделенных страниц при количестве запросов к страницам больше установленного порога.** Алгоритм замещения – LRU. Замещается страница с наименьшим количеством обращений.

14	**Локальное размещение с динамическим увеличением количества выделенных страниц при количестве запросов к страницам больше установленного порога.** Алгоритм замещения – LRU CLOCK. Существует глобальный счетчик обращений к страницам. При каждом обращении к станице в ее внутренний регистр заносится значение глобального счетчика. Выгружается страница с наименьшим значением счетчика.
15	**Локальное размещение с динамическим увеличением количества выделенных страниц при количестве запросов к страницам больше установленного порога.** Алгоритм замещения – NFU. По таймеру 2 (с большей задержкой чем T1) происходит опрос всех страниц, для каждой страницы с установленным флагом обращений увеличивается соответствующий счетчик, флаг обращений сбрасывается. Выгружается страница с наименьшим значением счетчика.

В процессе работы менеджера на экран должна выводиться следующая статистическая информация:
1. общее число обращений к страницам;
2. количество выталкиваний из памяти (дополнительно в %).
Провести исследования для разного числа страниц при постоянном объеме памяти.

Лабораторная работа №10

Реализация файловой системы

Цель работы: **разработать модель файловой системы**

1. Теоретические сведения

Файловая система - это часть операционной системы, назначение которой состоит в том, чтобы организовать эффективную работу с данными, хранящимися во внешней памяти, и обеспечить пользователю удобный интерфейс при работе с такими данными.

1.1 Алгоритмы выделения дискового пространства

Главный вопрос, какой тип структур используется для учета отдельных блоков файла, то есть способ связывания файлов с блоками диска. В ОС используется несколько методов выделения файлу дискового пространства. Для каждого из методов запись в директории, соответствующая символьному имени файла, содержит указатель, следуя которому можно найти все блоки данного файла.

Выделение непрерывной последовательностью блоков

Простейший способ - хранить каждый файл как непрерывную последовательность блоков диска. При непрерывном расположении файл характеризуется адресом и длиной (в блоках). Файл, стартующий с блока b, занимает затем блоки b+1, b+2, ... b+n-1.

Эта схема имеет два преимущества. Во-первых, ее легко реализовать, так как выяснение местонахождения файла сводится к вопросу, где находится первый блок. Во-вторых, она обеспечивает хорошую производительность, так как целый файл может быть считан за одну дисковую операцию.

Этот способ распространен мало, и вот почему. В процессе эксплуатации диск представляет собой некоторую совокупность свободных и занятых фрагментов. Не всегда имеется подходящий по размеру свободный фрагмент для нового файла.

Кроме того, непрерывное распределение внешней памяти неприменимо до тех пор, пока неизвестен максимальный размер файла.

Единственным приемлемым решением перечисленных проблем является периодическое уплотнение содержимого внешней памяти, или "сборка мусора", цель которой состоит в объединении свободных участков в один большой блок. Но это дорогостоящая операция, которую невозможно осуществлять слишком часто.

Таким образом, когда содержимое диска постоянно изменяется, данный метод нерационален. Однако для стационарных файловых систем, например для файловых систем компакт-дисков, он вполне пригоден.

Связный список

Внешняя фрагментация - основная проблема рассмотренного выше метода - может быть устранена за счет представления файла в виде связного списка блоков диска. Запись в директории содержит указатель на первый и последний блоки файла (иногда в качестве варианта используется специальный знак конца файла - EOF). Каждый блок содержит указатель на следующий блок (см. рис. 1).

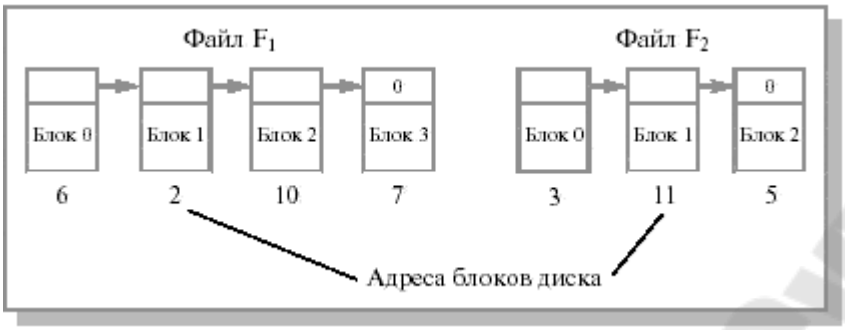

Рис. 1. Хранение файла в виде связного списка дисковых блоков

Внешняя фрагментация для данного метода отсутствует. Любой свободный блок может быть использован для удовлетворения запроса. Заметим, что нет необходимости декларировать размер файла в момент создания. Файл может расти неограниченно.
Связное выделение имеет, однако, несколько существенных недостатков:

- при прямом доступе к файлу для поиска i-го блока нужно осуществить несколько обращений к диску, последовательно считывая блоки от 1 до i-1, то есть выборка логически смежных записей, которые занимают физически несмежные секторы, может требовать много времени. Здесь мы теряем все преимущества прямого доступа к файлу.
- данный способ не очень надежен. Наличие дефектного блока в списке приводит к потере информации в оставшейся части файла и потенциально к потере дискового пространства, отведенного под этот файл.
- для указателя на следующий блок внутри блока нужно выделить место, что не всегда удобно. Емкость блока, традиционно являющаяся степенью двойки (многие программы читают и пишут блоками по степеням двойки), таким образом, перестает быть степенью двойки, так как указатель отбирает несколько байтов.

Поэтому метод связного списка обычно в чистом виде не используется.

Таблица отображения файлов

Одним из вариантов предыдущего способа является хранение указателей не в дисковых блоках, а в индексной таблице в памяти, которая называется таблицей отображения файлов (FAT - file allocation table) (см. рис. 2). Этой схемы придерживаются многие ОС (MS-DOS, OS/2, MS Windows и др.)
По-прежнему существенно, что запись в директории содержит только ссылку на первый блок. Далее при помощи таблицы FAT можно локализовать блоки файла независимо от его размера. В тех строках таблицы, которые соответствуют последним блокам файлов, обычно записывается некоторое граничное значение, например EOF.
Главное достоинство данного подхода состоит в том, что по таблице отображения можно судить о физическом соседстве блоков, располагающихся на диске, и при выделении нового блока можно легко найти свободный блок диска, находящийся поблизости от других блоков данного файла. Минусом данной схемы может быть необходимость хранения в памяти этой довольно большой таблицы.

Номера блоков диска		
1		
2	10	
3	11	Начало файла F_2
4		
5	EOF	
6	2	Начало файла F_1
7	EOF	
8		
9		
10	7	
11	5	

Рис. 2. Метод связного списка с использованием таблицы в оперативной памяти

Индексные узлы

Наиболее распространенный метод выделения файлу блоков диска - связать с каждым файлом небольшую таблицу, называемую индексным узлом (i-node), которая перечисляет атрибуты и дисковые адреса блоков файла (см. рис 3). Запись в директории, относящаяся к файлу, содержит адрес индексного блока. По мере заполнения файла указатели на блоки диска в индексном узле принимают осмысленные значения.

Индексирование поддерживает прямой доступ к файлу, без ущерба от внешней фрагментации. Индексированное размещение широко распространено и поддерживает как последовательный, так и прямой доступ к файлу.

Обычно применяется комбинация одноуровневого и многоуровневых индексов. Первые несколько адресов блоков файла хранятся непосредственно в индексном узле, таким образом, для маленьких файлов индексный узел хранит всю необходимую информацию об адресах блоков диска. Для больших файлов один из адресов индексного узла указывает на блок косвенной адресации. Данный блок содержит адреса дополнительных блоков диска. Если этого недостаточно, используется блок двойной косвенной адресации, который содержит адреса блоков косвенной адресации. Если и этого не хватает, используется блок тройной косвенной адресации.

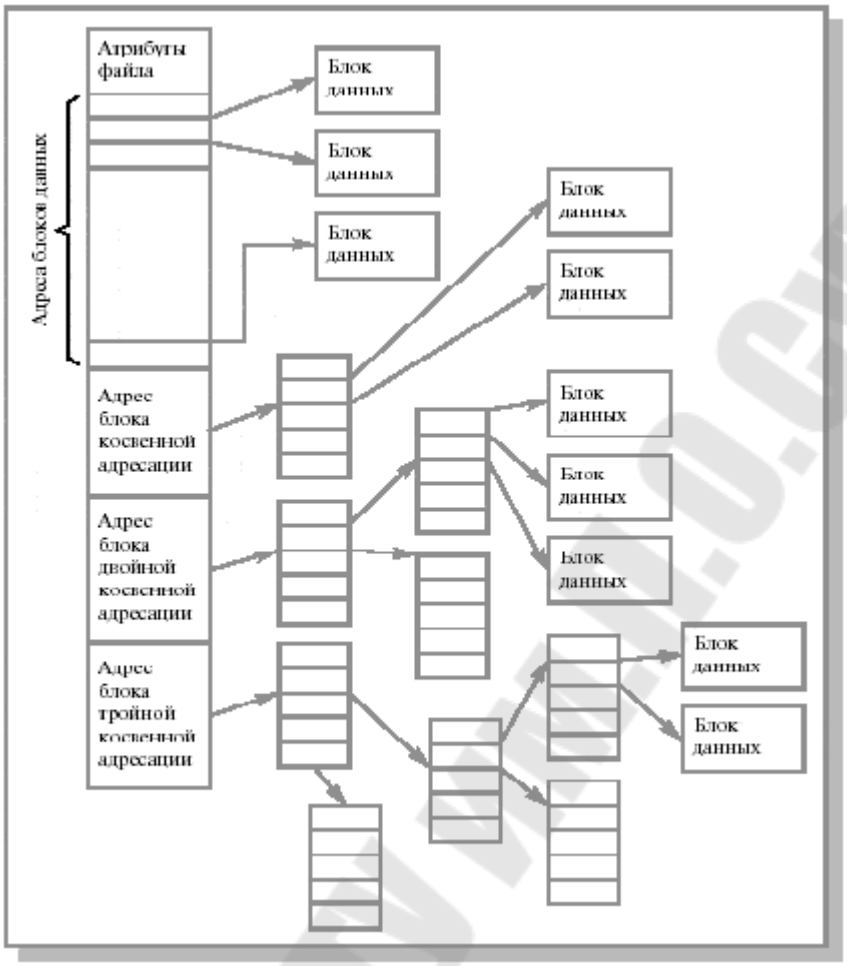

Рис. 3. Структура индексного узла

Данную схему используют файловые системы Unix (а также файловые системы HPFS, NTFS и др.). Такой подход позволяет при фиксированном, относительно небольшом размере индексного узла поддерживать работу с файлами, размер которых может меняться от нескольких байтов до нескольких гигабайтов. Существенно, что для маленьких файлов используется только прямая адресация, обеспечивающая максимальную производительность.

1.2 Управление свободным и занятым дисковым пространством

Дисковое пространство, не выделенное ни одному файлу, также должно быть управляемым. В современных ОС используется несколько способов учета используемого места на диске. Рассмотрим наиболее распространенные.

Учет при помощи организации битового вектора

Часто список свободных блоков диска реализован в виде битового вектора (bit map или bit vector). Каждый блок представлен одним битом, принимающим значение 0 или 1, в зависимости от того, занят он или свободен.

Например, 00111100111100011000001 ... .

Главное преимущество этого подхода состоит в том, что он относительно прост и эффективен при нахождении первого свободного блока или n последовательных блоков на диске. Многие компьютеры имеют инструкции манипулирования битами, которые могут использоваться для этой цели.

Несмотря на то что размер описанного битового вектора наименьший из всех возможных структур, даже такой вектор может оказаться большого размера. Поэтому данный метод эффективен, только если битовый вектор помещается в памяти целиком, что возможно лишь для относительно небольших дисков. Например, диск размером 4 Гбайт с блоками по 4 Кбайт нуждается в таблице размером 128 Кбайт для управления свободными блоками. Иногда, если битовый вектор становится слишком большим, для ускорения поиска в нем его разбивают на регионы и организуют резюмирующие структуры данных, содержащие сведения о количестве свободных блоков для каждого региона.

Учет при помощи организации связного списка

Другой подход - связать в список все свободные блоки, размещая указатель на первый свободный блок в специально отведенном месте диска, попутно кэшируя в памяти эту информацию. Подобная схема не всегда эффективна. Для трассирования списка нужно выполнить много обращений к диску. Однако, к счастью, нам необходим, как правило, только первый свободный блок.

Иногда прибегают к модификации подхода связного списка, организуя хранение адресов n свободных блоков в первом свободном блоке. Первые n-1 этих блоков действительно используются. Последний блок содержит адреса других n блоков и т. д.

Существуют и другие методы, например, свободное пространство можно рассматривать как файл и вести для него соответствующий индексный узел.

Размер блока

Размер логического блока играет важную роль. В некоторых системах (Unix) он может быть задан при форматировании диска. Небольшой размер блока будет приводить к тому, что каждый файл будет содержать много блоков. Чтение блока осуществляется с задержками на поиск и вращение, таким образом, файл из многих блоков будет читаться медленно. Большие блоки обеспечивают более высокую скорость обмена с диском, но из-за внутренней фрагментации (каждый файл занимает целое число блоков, и в среднем половина последнего блока пропадает) снижается процент полезного дискового пространства.

Для систем со страничной организацией памяти характерна сходная проблема с размером страницы.

Проведенные исследования показали, что большинство файлов имеют небольшой размер. Например, в Unix приблизительно 85% файлов имеют размер менее 8 Кбайт и 48% - менее 1 Кбайта.

Можно также учесть, что в системах с виртуальной памятью желательно, чтобы единицей пересылки диск-память была страница (наиболее распространенный размер страниц памяти - 4 Кбайта). Отсюда обычный компромиссный выбор блока размером 512 байт, 1 Кбайт, 2 Кбайт, 4 Кбайт.

1.3 Структура файловой системы на диске

Рассмотрение методов работы с дисковым пространством дает общее представление о совокупности служебных данных, необходимых для описания файловой системы. Структура служебных данных типовой файловой системы, например Unix, на одном из разделов диска, таким образом, может состоять из четырех основных частей (см. рис. 4).

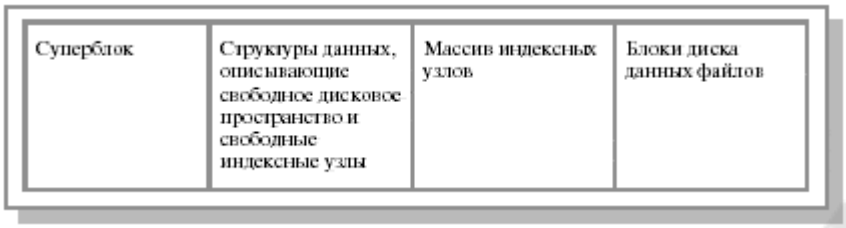

Рис. 4. Примерная структура файловой системы на диске

В начале раздела находится суперблок, содержащий общее описание файловой системы, например:

- тип файловой системы;
- размер файловой системы в блоках;
- размер массива индексных узлов;
- размер логического блока.

Описанные структуры данных создаются на диске в результате его форматирования (например, утилитами format, makefs и др.). Их наличие позволяет обращаться к данным на диске как к файловой системе, а не как к обычной последовательности блоков.
В файловых системах современных ОС для повышения устойчивости поддерживается несколько копий суперблока. В некоторых версиях Unix суперблок включал также и структуры данных, управляющие распределением дискового пространства, в результате чего суперблок непрерывно подвергался модификации, что снижало надежность файловой системы в целом. Выделение структур данных, описывающих дисковое пространство, в отдельную часть является более правильным решением.
Массив индексных узлов (ilist) содержит список индексов, соответствующих файлам данной файловой системы. Размер массива индексных узлов определяется администратором при установке системы. Максимальное число файлов, которые могут быть созданы в файловой системе, определяется числом доступных индексных узлов.
В блоках данных хранятся реальные данные файлов. Размер логического блока данных может задаваться при форматировании файловой системы. Заполнение диска содержательной информацией предполагает использование блоков хранения данных для файлов директорий и обычных файлов и имеет следствием модификацию массива индексных узлов и данных, описывающих пространство диска. Отдельно взятый блок данных может принадлежать одному и только одному файлу в файловой системе.

Реализация директорий

Как уже говорилось, директория или каталог - это файл, имеющий вид таблицы и хранящий список входящих в него файлов или каталогов. Основная задача файлов-директорий - поддержка иерархической древовидной структуры файловой системы. Запись в директории имеет определенный для данной ОС формат, зачастую неизвестный пользователю, поэтому блоки данных файла-директории заполняются не через операции записи, а при помощи специальных системных вызовов (например, создание файла).
Для доступа к файлу ОС использует путь (pathname), сообщенный пользователем. Запись в директории связывает имя файла или имя поддиректории с блоками данных на диске (см. рис. 5). В зависимости от способа выделения файлу блоков диска (см. раздел "Методы выделения дискового пространства") эта ссылка может быть номером первого блока или номером индексного узла. В любом случае обеспечивается связь символьного имени файла с данными на диске.

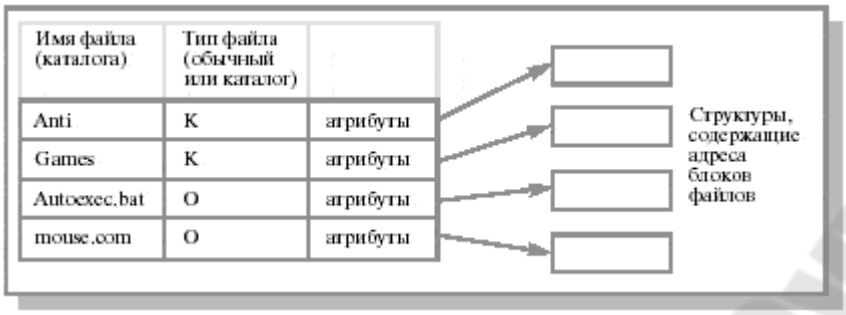

Рис. 5. Реализация директорий

Когда система открывает файл, она ищет его имя в директории. Затем из записи в директории или из структуры, на которую запись в директории указывает, извлекаются атрибуты и адреса блоков файла на диске. Эта информация помещается в системную таблицу в главной памяти. Все последующие ссылки на данный файл используют эту информацию. Атрибуты файла можно хранить непосредственно в записи в директории, как показано на рис. 5. Однако для организации совместного доступа к файлам удобнее хранить атрибуты в индексном узле, как это делается в Unix.

Примеры реализации директорий в некоторых ОС

Директории в ОС MS-DOS

В ОС MS-DOS типовая запись в директории имеет вид, показанный на рис. 6.

Рис. 6. Вариант записи в директории MS-DOS

В ОС MS-DOS, как и в большинстве современных ОС, директории могут содержать поддиректории (специфицируемые битом атрибута), что позволяет конструировать произвольное дерево директорий файловой системы.
Номер первого блока используется в качестве индекса в таблице FAT. Далее по цепочке в этой таблице могут быть найдены остальные блоки.

Директории в ОС Unix
Структура директории проста. Каждая запись содержит имя файла и номер его индексного узла (см. рис. 7). Вся остальная информация о файле (тип, размер, время модификации, владелец и т. д. и номера дисковых блоков) находится в индексном узле.

Рис. 7. Вариант записи в директории Unix

В более поздних версиях Unix форма записи претерпела ряд изменений, например имя файла описывается структурой. Однако суть осталась прежней.

1.4 Поиск в директории

Список файлов в директории обычно не является упорядоченным по именам файлов. Поэтому правильный выбор алгоритма поиска имени файла в директории имеет большое влияние на эффективность и надежность файловых систем.

Линейный поиск

Существует несколько стратегий просмотра списка символьных имен. Простейшей из них является линейный поиск. Директория просматривается с самого начала, пока не встретится нужное имя файла. Хотя это наименее эффективный способ поиска, оказывается, что в большинстве случаев он работает с приемлемой производительностью. Например, авторы Unix утверждали, что линейного поиска вполне достаточно. По-видимому, это связано с тем, что на фоне относительно медленного доступа к диску некоторые задержки, возникающие в процессе сканирования списка, несущественны. Метод прост, но требует временных затрат. Для создания нового файла вначале нужно проверить директорию на наличие такого же имени. Затем имя нового файла вставляется в конец директории (если, разумеется, файл с таким же именем в директории не существует, в противном случае нужно информировать пользователя). Для удаления файла нужно также выполнить поиск его имени в списке и пометить запись как неиспользуемую. Реальный недостаток данного метода - последовательный поиск файла. Информация о структуре директории используется часто, и неэффективный способ поиска будет заметен пользователями. Можно свести поиск к бинарному, если отсортировать список файлов. Однако это усложнит создание и удаление файлов, так как требуется перемещение большого объема информации.

Хеш-таблица

Хеширование (см. например, [Ахо, 2001]) - другой способ, который может использоваться для размещения и последующего поиска имени файла в директории. В данном методе имена файлов также хранятся в каталоге в виде линейного списка, но дополнительно используется хеш-таблица. Хеш-таблица, точнее построенная на ее основе хеш-функция, позволяет по имени файла получить указатель на имя файла в списке. Таким образом, можно существенно уменьшить время поиска.

В результате хеширования могут возникать коллизии, то есть ситуации, когда функция хеширования, примененная к разным именам файлов, дает один и тот же результат. Обычно имена таких файлов объединяют в связные списки, предполагая в дальнейшем осуществление в них последовательного поиска нужного имени файла. Выбор подходящего алгоритма хеширования позволяет свести к минимуму число коллизий. Однако всегда есть вероятность неблагоприятного исхода, когда непропорционально большому числу имен файлов функция хеширования ставит в соответствие один и тот же результат. В таком случае преимущество использования этой схемы по сравнению с последовательным поиском практически утрачивается.

2. Индивидуальные задания

Разработать приложение, создающее виртуальный файл и позволяющее

- форматировать виртуальный файл с возможностью задания размера кластера;
- создавать каталоги в виртуальном файле;
- производить учёт свободного пространства;
- реализовывать поиск файлов и директорий;
- сохранять в виртуальный файл файлы с жёсткого диска;
- удалять файлы из виртуального файла;
- записывать на жёсткий диск файлы из виртуального файла;
- создавать в виртуальном файле текстовые файлы;
- предоставлять возможность редактировать текстовые файлы внутри виртуального файла.

Файловую систему внутри виртуального файла выбрать согласно варианта.

Таблица 13. Варианты заданий

Вариант	Условие задачи	Учёт свободных блоков	Поиск файлов и папок
1	Последовательная файловая система. Выделение непрерывной последовательности блоков. (Best Fit).	Связанный список	Хэш-таблицы
2	Индексно-последовательная файловая система. Связанный список.	Битовый вектор	Бинарный поиск
3	Одноуровневые индексные узлы.	Файловый	B-дерево
4	Индексно-последовательная файловая система. Таблица отображения файлов.	Связанный список	Линейный поиск
5	Многоуровневые индексные узлы.	Файловый	Линейный поиск
6	Индексно-последовательная файловая система. Связанный список.	Связанный список	Хэш-таблицы
7	Последовательная файловая система. Выделение непрерывной последовательности блоков. (Worst Fit).	Связанный список	Бинарный поиск
8	Индексно-последовательная файловая система. Таблица отображения файлов.	Файловый	Бинарный поиск
9	Одноуровневые индексные узлы.	Битовый вектор	Бинарный поиск
10	Индексно-последовательная файловая система. Связанный список.	Битовый вектор	Хэш-таблицы
11	Многоуровневые индексные узлы.	Битовый вектор	Линейный поиск
12	Последовательная файловая система. Выделение непрерывной последовательности блоков. (First Fit).	Связанный список	Хэш-таблицы
13	Индексно-последовательная файловая система. Связанный список.	Битовый вектор	B-дерево
14	Одноуровневые индексные узлы.	Связанный список	Хэш-таблицы
15	Индексно-последовательная файловая система. Таблица отображения файлов.	Битовый вектор	Хэш-таблицы

Лабораторная работа №11

Управление файловой системой

Цель работы:

1. Теоретические сведения

См. условие задания лабораторной работы №7

2. Индивидуальные задания

Дополнить программу, разработанную при выполнении лабораторной работы №13 вспомогательной утилитой, согласно варианта (см. табл. 14).

Таблица 14. Варианты заданий

Вариант	Дополнительная утилита
1	Дефрагментация
2	Скандиск
3	Управление доступом
4	Журнализация
5	Журнализация
6	Кэширование
7	«Сборка мусора»
8	Рекласторизатор
9	Дефрагментация
10	Дефрагментация
11	Управление доступом
12	Рекласторизатор
13	Управление доступом
14	Дефрагментация
15	Рекласторизатор