Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное
учреждение высшего образования
«Севастопольский государственный университет»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к лабораторному практикуму
по дисциплине
"Операционные системы и среды"
для студентов дневной и заочной формы обучения
09.03.02 «Информационные системы и технологии» и
09.03.03 «Прикладная информатика»

часть 2

Севастополь
2020

УДК 004.8

Методические указания к лабораторному практикуму по дисциплине «**Операционные системы и среды**» для студентов дневного и заочного отделения направлений 09.03.02 «Информационные системы и технологии» и 09.03.03 «Прикладная информатика» **/Сост. А. Л. Овчинников – Севастополь: Изд-во СевГУ, 2020. – 39с.**

Методические указания предназначены для проведения лабораторных занятий по дисциплине **«Операционные системы и среды»**. Целью настоящих методических указаний является обучение студентов практическим навыкам разработки приложений с использованием возможностей Win API, сценариев оболочки и программ для операционных систем семейства Unix.

Методические указания составлены в соответствии с требованиями программы дисциплины «Операционные системы и среды» для студентов специальности 09.03.02, 09.03.03 и утверждены на заседании кафедры информационных систем, протокол № от _____.2020.

Допущено учебно-методическим центром СевГУ в качестве методических указаний.

СОДЕРЖАНИЕ

ЛАБОРАТОРНАЯ РАБОТА №5

Исследование возможностей управления памятью и обмена данными между процессами в ОС Windows.

1. ЦЕЛЬ РАБОТЫ

Изучить возможности программного интерфейса приложений (API) операционных систем Windows по управления памятью и обмена данными между процессами. Приобрести практические навыки использования Win API для управления памятью и обмена данными между процессами.

2. ОСНОВНЫЕ ПОЛОЖЕНИЯ

2.1. Страничная организация памяти

В основе многих современных операционных систем располагается специальный механизм управления памятью, который называют MMU (Memory Management Unit — устройство управления памятью). MMU осуществляет преобразование линейных адресов виртуального адресного пространства каждого из запущенных процессов в адреса физической оперативной памяти, реально установленной на компьютере.

В операционной системе Windows каждый процесс получает в свое распоряжение виртуальное адресное пространство размером 4 Гбайт. Как правило, только часть этого пространства отображается на физическую оперативную память, реально установленную на компьютере.

MMU рассматривает всю оперативную память как набор небольших блоков, которые называют *страницами,* и осуществляет трансляцию адресов в соответствии с границами этих страниц. MMU, встроенный в процессоры типа Pentium, использует страницы размером 4 Кбайт (другие процессоры могут использовать другой размер страниц; узнать этот размер можно при помощи функции GetSystemInfo).

MMU может пометить страницу виртуальной памяти как *отсутствующую* (absent). Такой механизм лежит в основе организации *виртуальной памяти.* MMU сообщает операционной системе, что произошло обращение к отсутствующей в физической оперативной памяти странице. ОС загружает отсутствующую страницу в физическую память и разрешает программе вновь попробовать обратиться к ней. Конечно же, чтобы освободить место для загрузки отсутствующей страницы, ОС может записать на диск какую-либо другую страницу виртуальной памяти и пометить ее как отсутствующую.

2.2. Использование функции VirtualAlloc

Выделить необходимое пространство виртуальной памяти в ОС Windows можно при помощи вызова VirtualAlloc. Прототип этой функции представлен ниже:

LPVOID VirtualAlloc(LPVOID lpAddress, DWORD dwSize, DWORD flAllocationType, DWORD flProtect);

В качестве аргументов функция принимает адрес области памяти(*lpAddress*), необходимый размер(*dwSize*), режим выделения(*flAllocationType*) и уровень защиты(*flProtect*).

Если в качестве первого аргумента функции передать значение NULL, то ОС самостоятельно определит подходящий адрес выделяемой области. При указании конкретного адреса ОС попытается выделить память с заданным начальным адресом (этот адрес должен соответствовать границе страницы). Второй аргумент - количество байт, которое необходимо выделить. Следует отметить, что Windows автоматически округлит это количество в большую сторону таким образом, чтобы оно равнялось числу, кратному размеру страницы. Если необходимо выделить (а не просто зарезервировать) память, в качестве третьего аргумента необходимо указать значение MEM_COMMIT. Последний аргумент — уровень защиты - определяет устанавливаемый уровень доступа выделяемой памяти (см. таблицу 2.1).

Таблица 2.1 - Константы уровня защиты

Флаг	**Значение**
PAGE_READONLY	Выделяется память только для чтения
PAGE_READWRITE	Выделяется память как для чтения, так и для записи
PAGE_WRITECOPY	После осуществления записи страница заменяется ее новой копией (используется совместно с общей памятью или отраженными на память файлами)
PAGE_EXECUTE	Разрешается исполнение программного кода, расположенного в выделяемой памяти
PAGE_EXECUTE_ READ	Разрешается исполнение программного кода и чтение данных, расположенных в выделяемой памяти
PAGE_EXECUTE_ READWRITE	Разрешается исполнение программного кода, а также чтение и запись данных, расположенных в выделяемой памяти
PAGE_NOACCESS	Запрещает любой доступ к выделенной памяти. В случае любого доступа возникает ошибка General Protection Fault (GPF)
PAGE_GUARD	Страницы в выделяемой памяти становятся «сторожевыми». При попытке осуществить чтение или запись данных в сторожевой странице операционная система инициирует исключение STATUS_GUARD_PAGE и сбрасывает статус сторожевой страницы. Таким образом, сторожевые страницы работают в режиме «тревога-при-первом-обращении». Обычно такой режим используется совместно с другими флагами (за исключением флага PAGE_NOACCESS)
PAGE_NOCACHE	Запрещает кэширование страниц (не рекомендуется). Используется совместно с другими флагами (за исключением флага PAGE.NOACCESS)

Если использовать VirtualAlloc описанным образом, эта функция выделит участок памяти указанного объема точно так же, как если бы для этой цели был использован стандартный вызов malloc. Но в отличие от malloc, функция VirtualAlloc выделяет участок памяти, выровненный по границе страницы, который будет обладать указанными атрибутами доступа. Кроме того, в случае необходимости, возможно корректно освободить этот участок памяти и передать его в ведение ОС. Чтобы освободить память, выделенную при помощи VirtualAlloc, используется функция Virtual Free.

Если необходимо изменить уровень защиты для одной или нескольких страниц выделенного участка, используется вызов VirtualProtect:

BOOL VirtualProtect(LPVOID lpAddress, DWORD dwSize, DWORD flNewProtect, PDWORD lpflOldProtect);

При этом участку памяти размером *dwSize*, начинающемуся с начального адреса *lpAddress* можно присвоить(*flNewProtect*) любой из атрибутов, перечисленных в таблице 2.1.

Помимо перечисленных, функция VirtualAlloc обладает некоторыми другими возможностями. Если в качестве третьего аргумента функции VirtualAlloc вместо флага MEM_COMMIT использовать флаг MEM_RESERVE, то функция только резервирует указанный объем линейного адресного пространства, при этом реальная физическая или виртуальная память не выделяется. В этом случае происходит лишь резервирование диапазона адресов – что выполняется гораздо быстрее. Когда же потребуется выделить реальную память, соответствующую этим адресам, необходимо обратиться к вызову VirtualAlloc и передать ему адрес этого участка памяти и флаг MEM_COMMIT. Таким образом, выделять память сразу для всего зарезервированного диапазона адресов (особенно, если он велик) не обязательно. Например, можно зарезервировать 10 Мбайт адресного пространства, а затем выделять память из этого диапазона адресов фрагментами по 100 Кбайт (в этом случае задержка при выделении каждого фрагмента памяти будет небольшой).

2.3. Работа с атрибутами страниц

Одним из основных преимуществ использования VirtualAlloc является возможность управления уровнем доступа к страницам оперативной памяти как при ее выделении, так и позже (при помощи функции VirtualProtect, описанной выше).

Для примера рассмотрим следующую программу:

```
#include <windows.h>
char *getmsg(){
   static char *msg;
   if (msg==NULL){
     char tmp[1024];
     DWORD oldprot;
     FILE *f;
     f=fopen("strings.txt","r");
     if (!f) return NULL;
     fgets(tmp,sizeof(tmp),f);
// Выделяется целая страница(что приемлемо для данного примера)
     msg=(char*)VirtualAlloc(NULL,strlen(tmp)+1,MEM_COMMIT, PAGE_READWRITE);
     strcpy(msg,tmp);
     fclose(f);
//Переводит страницу в режим только для чтения
     VirtualProtect(msg,strlen(msg)+1,PAGE_READONLY,&oldprot);
   }
  return msg;
}
```

```
void main(){
  char *m=getmsg();
  try {
   printf("Entire string: %s\n",m);
   m=strtok(m," \t");
   printf("token1=%s\n",m);
   m=getmsg();
   printf("Entire string: %s\n",m);
  }
  catch (...)  // Обработка исключения
   {
    printf("Произошло исключение!\n");
   }
   getch();
}
```

Основная программа получает статическую строку, а затем обращается к strtok, чтобы разбить ее на части. При этом произойдет исключение, так как функция strtok пытается модифицировать строку, которая ей передается. Код, приведенный выше, следит за возникновением исключений (try .. catch реагирует на любое возникающее исключение), таким образом, в случае обращения по записи к защищенной странице будет выведено сообщение об ошибке. Если убрать из текста программы операторы try…catch, система прервет работу программы и выведет на экран системное сообщение об ошибке.

2.4. Организация памяти общего доступа в Win API

Один из способов реализации памяти общего доступа в Win API заключается в использовании механизма отображения файлов на память(FileMapping). Операционная система позволяет создать объект отображения файла на память, связывая с файлом область виртуального адресного пространства процесса. Процесс в этом случае записывает данные в память, а операционная система, по мере возможности, осуществляет запись содержимого памяти в файл.

Два процесса могут одновременно получить доступ к одному и тому же файлу, отображенному на память, что позволяет им обмениваться данными. Однако при этом существенно снижается производительность, так как любое обращение к памяти общего доступа будет сопровождаться операцией файлового ввода/вывода. Если при создании объекта отображения файла на память в качестве дескриптора файла указать константу INVALID_HANDLE_VALUE (значение (HANDLE)0xFFFFFFFF в Си или $FFFFFFFF в Delphi), операционная система не будет ассоциировать этот объект с каким-либо файлом. В этом случае объект отображения будет использоваться только для обеспечения общего доступа разных процессов к одному участку оперативной памяти, не связанному с файлом.

Для создания объекта отображения на память используется вызов CreateFileMapping. Прототип этой функции представлен ниже:

HANDLE CreateFileMapping(HANDLE hFile, LPSECURITY_ATTRIBUTES lpFileMappingAttributes, DWORD flProtect, DWORD dwMaximumSizeHigh, DWORD dwMaximumSizeLow, LPCTSTR lpName);

Аргументы вызова CreateFileMapping описаны в таблице 2.2:

Таблица 2.2 - Аргументы вызова CreateFileMapping

Аргумент	**Тип**	**Описание**
hFile	HANDLE	Дескриптор файла (значение 0xFFFFFFFF в случае, если требуется общая память, а не файл)
lpFileMappingAttributes	LPSECURITY_ ATTRIBUTES	Структура атрибутов безопасности (управляет наследованием и доступом)
flProtect	DWORD	Устанавливает защиту (только чтение, зарезервировано или выделено и т. п.)
dwMaximumSizeHigh	DWORD	Старшие 32 бита 64-битного размера файла
dwMaximumSizeLow	DWORD	Младшие 32 бита 64-битного размера файла
lpName	LPCTSTR	Имя объекта отображения файла на память

Таблица 2.3 - Аргументы вызова MapViewOfFile

Аргумент	**Тип**	**Описание**
hFileMappingObject	HANDLE	Дескриптор объекта отображения файла на память
dwDesiredAccess	DWORD	Тип доступа: чтение, запись, полный, копирование
dwFlleOffsetHIgh	DWORD	Старшие **32** бита 64-битной стартовой позиции
dwFlleQffsetLow	DWORD	Младшие **32** бита 64-битной стартовой позиции
dwNumberOfBytesToMap	DWORD	Количество байт, которое следует отобразить(0 в случае, если требуется отобразить весь файл)

Чтобы создать область памяти общего доступа с использованием механизма отображения файлов на память, следует выполнить следующие действия:

1. Обратиться к вызову CreateFileMapping. При этом в качестве дескриптора файла указать значение 0xFFFFFFFF. Необходимо присвоить объекту отображения файла на память уникальное имя, используя которое другие процессы смогут обратиться к этому объекту.

2. Обратиться к вызову MapViewOfFile(аргументы этой функции представлены в таблице 2.3) (или MapViewOfFileEx), чтобы получить указатель, который можно использовать для доступа к созданной области памяти.

3. После завершения работы с общей памятью необходимо использовать вызов UnmapViewOfFile, чтобы освободить выделенную память.

4. Чтобы уничтожить дескриптор объекта отображения файла на память, используется вызов CloseHandle.

Чтобы получить доступ к общей памяти из другого процесса, необходимо выполнить следующие шаги:

1. Используя вызов OpenFileMapping, открыть объект отображения файла на память с указанным именем.

2. Используя вызов MapViewOfFile, получить указатель для работы с общей памятью.

3. После завершения работы с общей памятью использовать UnmapViewOfFile, чтобы освободить указатель.

4. Обратиться к CloseHandle и передать этому вызову дескриптор объекта отображения файла на память, полученный на первом шаге. После этого общая память будет освобождена.

3. ВАРИАНТЫ ЗАДАНИЙ И ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

3.1. Написать программу MemShare, выполняющую следующие действия:

3.1.1. Создание(запуск) процесса MemSort(действия, выполняемые этим процессом описаны в п. 3.2), используя вызов CreateProcess;

3.1.2. Выделение с помощью вызова VirtualAlloc заданного по варианту задания количества страниц памяти (страница 4096 байт);

3.1.3. Заполнение выделенной памяти случайными числами в диапазоне от 0 до MAX (значение MAX выбирается исходя из типа данных, заданному по варианту задания);

3.1.4. Перевод режима доступа выделенной памяти в ReadOnly (с помощью вызова VirtualProtect);

3.1.5. Используя механизм отображения файлов на память (функции CreateFileMapping и MapViewOfFile), создание памяти общего доступа, с размером соответствующим размеру выделенной с помощью VirtualAlloc памяти (задать имя создаваемого объекта отображения "memshare");

3.1.6. Копирование данных из памяти, выделенной с помощью VirtualAlloc, в память общего доступа (например, с помощью вызова CopyMemory);

3.1.7. Ожидание процесса MemSort (используя заданный по варианту задания объект синхронизации необходимо ожидать пока процесс MemSort не подготовит данные);

3.1.8. Перевод режима доступа выделенной в п. 3.1.1 памяти в ReadWrite(с помощью вызова VirtualProtect);

3.1.9. Копирование данных из памяти общего доступа в память, выделенную с помощью VirtualAlloc (например, с помощью вызова CopyMemory);

3.1.10. Вывод на экран данных из памяти, выделенной с помощью VirtualAlloc;

3.1.11. Освобождение выделенной памяти (с помощью вызова VirtualFree);

3.1.12. Освободить память общего доступа (используя вызовы UnmapViewOfFile и CloseHandle);

3.2. Написать программу MemSort выполняющую следующие действия:

3.2.1. Ожидание пока процесс MemShare не подготовит данные в памяти общего доступа (использовать заданный по варианту задания тип объекта синхронизации);

3.2.2. Открытие памяти общего доступа "memshare" (используя вызовы OpenFileMapping и MapViewOfFile);

3.2.3. Сортировка данных в памяти общего доступа, методом указанным в варианте задания.

Примечание: среда программирования Dev C++ или Visual Studio.

Таблица 3.1 - Варианты заданий

№ варианта	Количество страниц памяти	Тип данных для заполнения памяти	Тип сортировки	Объект синхронизации
1	1	char	Вставки	Мютекс
2	2	short	Шелла	Бинарный семафор
3	3	int	Быстрая	Мютекс
4	4	char	Пузырька	Считающий семафор
5	1	short	Выбора	Мютекс
6	2	int	Вставки	Бинарный семафор
7	3	char	Шелла	Мютекс
8	4	short	Быстрая	Считающий семафор
9	1	int	Пузырька	Мютекс
10	2	char	Выбора	Бинарный семафор
11	3	short	Вставки	Мютекс
12	4	int	Шелла	Считающий семафор
13	1	char	Быстрая	Мютекс
14	2	short	Пузырька	Бинарный семафор
15	3	int	Выбора	Мютекс
16	4	char	Вставки	Считающий семафор
17	1	short	Шелла	Мютекс
18	2	int	Быстрая	Бинарный семафор
19	3	char	Пузырька	Мютекс
20	4	short	Выбора	Считающий семафор
21	1	int	Вставки	Мютекс
22	2	char	Шелла	Бинарный семафор
23	3	short	Быстрая	Мютекс
24	4	int	Пузырька	Считающий семафор
25	1	char	Выбора	Мютекс
26	2	short	Вставки	Бинарный семафор
27	3	int	Шелла	Мютекс
28	4	char	Быстрая	Считающий семафор
29	1	short	Пузырька	Мютекс
30	2	int	Выбора	Бинарный семафор

4. СОДЕРЖАНИЕ ОТЧЕТА

Отчет должен содержать следующие пункты:

- постановка задачи;
- тексты программ с комментариями;
- результаты выполнения программ;
- выводы.

5. КОНТРОЛЬНЫЕ ВОПРОСЫ

1) Поясните понятие виртуальной памяти?
2) Расскажите о страничном распределении памяти?
3) Каковы преимущества и недостатки сегментного распределения памяти?
4) Какой объем виртуальной памяти получает каждый процесс в ОС Windows?
5) Для чего используется функция VirtualAlloc?
6) Каким образом можно изменить режим доступа к выделенной функцией VirtualAlloc памяти?
7) Каким образом можно создать область памяти общего доступа в ОС Windows?
8) Каким образом можно использовать область памяти общего доступа в ОС Windows?

ЛАБОРАТОРНАЯ РАБОТА №6
Исследование командного интерпретатора ОС семейства UNIX

1. ЦЕЛЬ РАБОТЫ

Ознакомится с ОС семейства UNIX. Изучить основные команды оболочки bash ОС семейства UNIX. Приобрести практические навыки написания сценариев командного интерпретатора bash ОС семейства UNIX.

2. ОСНОВНЫЕ ПОЛОЖЕНИЯ

ОС Unix была создана Кеном Томпсоном и Деннисом Ритчи в Bell Laborotories (AT&T). Первые версии этой системы были написаны в 1969-1970 годах. Широко распространяться Unix/v7 (версия 7) начала в 79 - 80-м годах. Вручение создателям Unix в 1983 году Международной премии А.Тьюринга в области программирования ознаменовало признание этой системы мировой научной компьютерной общественностью.

Беспрецедентным является то, что ОС UNIX может работать практически на всех существующих аппаратных платформах. Основная причина этого - переносимость программного обеспечения системы. UNIX была одной из первых ОС, написанных в основном на языке высокого уровня С. В то время большинство ОС писалось на ассемблере, что крайне затрудняло перенос программного кода на другие архитектуры. Таким образом, UNIX с момента своего появления могла работать на разных платформах (всех платформах, для которых был реализован компилятор языка С).

Простота переносимости ОС UNIX стала основной причиной появления множества версий UNIX. В их названиях редко содержится само слово "UNIX" (что связано с долгосрочными авторскими правами создателя UNIX – компании AT&T). К семейству UNIX относятся такие коммерческие ОС, как SunOS и Solaris (Sun), HP-UX (Hewlett-Packard), Irix (Silicon Graphics), AIX (IBM) и др. Помимо коммерческих продуктов, созданы и бесплатные реализации UNIX, такие как Linux, FreeBSD и NetBSD.

2.1. Файловая система ОС семейства UNIX

2.1.1. Структура файловых систем ОС семейства UNIX

В ОС семейства UNIX, в отличие от ОС семейства MS-DOS/Windows, не предусматривается использование какой-либо одной, стандартной файловой системы. Существует ряд файловых систем, различающихся своей реализацией, но все они построены по одному принципу, общему для всех файловых систем UNIX.

Файловая система ОС UNIX имеет иерархическую (древовидную) структуру. В вершинах дерева находятся каталоги, содержащие списки файлов. Эти файлы в свою очередь могут быть либо снова каталогами, либо обычными файлами, либо специальными файлами, представляющими различные устройства ввода-вывода.

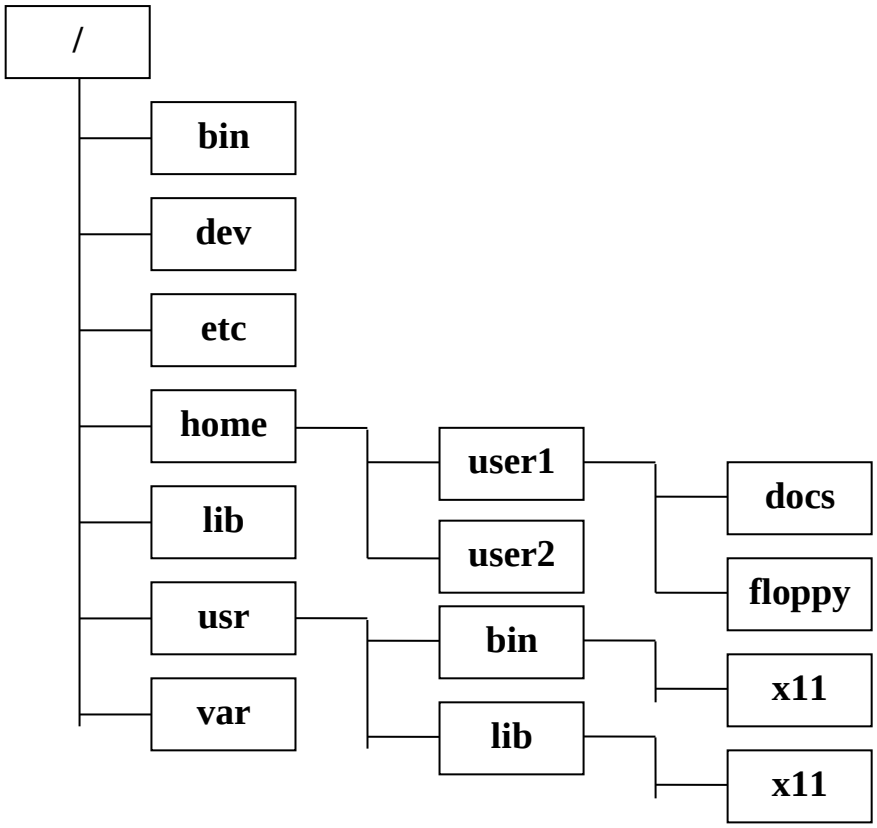

Рисунок 2.1 - Дерево каталогов UNIX

На рисунке 2.1 представлен пример дерева каталогов. Корневой каталог всегда обозначается символом "/". В отличие от MS-DOS и MS Windows, в которых имена присваиваются отдельным дискам (A,B,C и т.д.), в UNIX местоположение любого файла отсчитывается относительно корневого каталога (/). Диск, на котором расположена сама ОС и корневой каталог, принято называть *первичным* диском, остальные диски - *вторичными*. Вторичные диски и их разделы монтируются в файловой системе в *точке монтирования*, или *точке подключения*. Точка монтирования представляет собой не что иное, как имя каталога на первичном диске. На пример, на рисунке 2.1.1.1 в каталог /**home/user1/floppy** может быть смонтирована файловая система флоппи-диска, находящегося в дисководе.

В корневом каталоге обычно размещается ряд стандартных каталогов, выполняющих определенные функции:

- bin – для хранения основных команд (программ) ОС;
- dev – для специальных файлов, представляющих устройства (диски, порты и т.д.);
- etc – для хранения основных системных конфигурационных файлов;
- home - для хранения рабочих каталогов пользователей системы (домашних каталогов);
- lib - для хранения системных библиотек;
- usr – для хранения пользовательских программ и их библиотек
- var – для хранения вспомогательных программных файлов
- mnt – для монтирования вторичных дисков
- root – домашний каталог администратора системы
- и др.

2.1.2. Файлы в ОС семейства UNIX

Файлом называется поименованная область памяти, идентифицируемая именем. В качестве имени файла, как правило, может использоваться любая последовательность из букв, цифр и подчеркиваний. Могут использоваться и другие символы, однако ряд этих символов при использовании в имени требует специального экранирования. (Лучше не пользоваться специальными символами в именах - иногда это может привести к сложностям в обращении к таким именам, поскольку спецсимволы могут иметь в определенных программах некоторый специальный смысл). Не рекомендуется использовать в имени файла пробелы (возникают сложности с ипользованием таких файлов в качестве параметров команд). Имена файлов, начинающихся с точки, имеют особый статус, наподобие скрытых файлов MS-DOS. В ряде систем длина имени ограничивается 14-ю символами (этого ограничения желательно придерживаться для переносимости файлов), однако в других системах допускаются более длинные имена - например, до 256 символов. Имена файлов в ОС семейства UNIX чувствительны к регистру, поэтому имена "FILE", "file" и "File" - это три различных имени (в MS-DOS и MS Windows они указывали бы на один файл). В ОС семейства UNIX не существует понятия *расширения* имени файла, но так как возможно использовать в имени файла практически любые символы, в том числе и точку, то ничто не мешает пользователю отделить часть имени точкой и считать эту часть расширением. Кроме того, многие программы требуют, чтобы их файлы имели имя, заканчивающееся подобным «расширением» (например, компилятор языка С ищет исходные тексты программ в файлах, имена которых заканчиваются на ".c"). Возможно использование «нескольких расширений», например имя файла "documents.tar.gz" обычно означает, что какие-то файлы были собраны в один файл ("documents.tar") архиватором **tar**, а затем сжаты компрессором **gzip**.

Файл однозначно определяется комбинацией имени и полного пути к нему. Полный путь к файлу начинается из корневого каталога ("/"). Подкаталоги в пути разделяются символом "/". Пример указания файла с полным путем: "/etc/passwd". Существует также понятие текущего каталога, позволяющее определять файлы и каталоги указанием не полного пути, а пути относительно текущего каталога (относительного пути). В относительном пути кроме имен существующих каталогов можно использовать также два зарезервированных специальных имени. Это комбинация символов «..», обозначающая каталог верхнего уровня (каталог, содержащий текущий каталог в качестве подкаталога), а также символ «.», обозначающий текущий каталог.

Каждый файл в ОС семейства UNIX имеет идентификатор своего владельца (UID – User ID), а также идентификатор группы, владеющей файлом (GID – Group ID). Кроме того, каждому файлу приписывается набор атрибутов, регулирующих права доступа к файлу. Первый из этих атрибутов – атрибут, указывающий тип файла. Файл может являться обычным файлом, каталогом, ссылкой, блочным или символьным устройством. Остальные атрибуты делятся на три группы (триады), каждая из трех двоичных полей. Первая триада полей описывает права владельца файла, вторая – права группы, третья – права остальных пользователей. Первое поле

каждой триады определяет право на чтение, второе – право на запись, третья – право на выполнение данного файла. Хоть эти поля и являются двоичными, для лучшего понимания их записи обычно используют не символы "1" и "0", а символ "r" для единицы в первом поле (чтение), "w" для единицы во втором поле (запись), "x" для единицы в третьем поле (выполнение) и "-" для нуля в любом из полей. Аналогично атрибут типа файла обозначается как "-" для файла, "d" для каталога, "l" для ссылки, "b" для блочного и "c" для символьного устройства. Примеры такой записи атрибутов:

- **drwxrwxrwx** - каталог с полностью открытым доступом;
- **-rwxrwxrwx** - исполняемый файл с полностью открытым доступом;
- **-rw-rw-r--** - неисполняемый файл, чтение которого разрешено всем, а запись – только владельцу и группе владельцев;
- **-rwxr-xr-x** - исполняемый файл, чтение и запуск которого разрешены всем, а запись в него (т.е. изменение файла) – только владельцу.

Такая запись более удобна для понимания, но иногда (например, в команде установки атрибутов **chmod**) приходится применять числовую запись атрибутов. При этом каждая триада записывается в виде отдельной восьмеричной цифры, например:

rwx = 111_2 = 7_8
r-x = 101_2 = 5_8
rw- = 110_2 = 6_8

Затем данные восьмеричные цифры записываются последовательно, согласно своему порядку, формируя одно восьмеричное число:

rwxr-xrw- = 111101110_2 = 756_8.

Одним из специальных типов файлов ОС семейства UNIX являются ссылки. Ссылки выглядят для пользователя как файлы, но сами по себе файлами не являются, а ссылаются на какой-либо существующий файл.

Другим типом специальных файлов являются файлы устройств. Они делятся на два класса – блочные и символьные устройства. Блочные устройства – это устройства с произвольным доступом (например, диски), т.е. устройства, с которых можно считать(записать) любой произвольный блок информации. Символьные устройства – это устройства с последовательным доступом (например, порты ввода-вывода), которые можно представить как входной или выходной поток данных (символов). Для хранения файлов устройств предназначен каталог "/dev". К примеру, файл "/dev/hda" представляет собой первый физический диск на первом IDE-канале, "/dev/hda1" – первичный дисковый раздел на этом диске, "/dev/fd0" – флоппи-диск в дисководе, "/dev/tty" – терминал пользователя (консоль) и т.д.

Для генерации имен файлов используются метасимволы:

***** произвольная (возможно пустая) последовательность символов;

? один произвольный символ;

[...] любой из символов, указанных в скобках перечислением и/или с указанием диапазона;

2.2. Командная оболочка bash

2.2.1. Командная оболочка bash

Взаимодействие пользователя с ОС семейства UNIX может быть реализовано в текстовом режиме (без использования графического интерфейса) в виде диалога с использованием специальной программы - оболочки. В ОС семейства UNIX оболочка представляет собой внешнюю программу, запускаемую в момент входа пользователя в систему. Благодаря этому в ОС семейства UNIX существует целый ряд командных оболочек, из которых пользователь может выбирать подходящую. Исторически первой командной оболочкой была оболочка Борна – **sh** (Bourne Shell), появившаяся вместе с первыми выпусками UNIX. Она установила фактический стандарт среди оболочек. Кроме нее существуют такие оболочки, как **csh** (C Shell, оболочка, построенная на основе языка C) , **tcsh** (развитие csh), **ksh** (Korn Shell), **zsh**, **ash** и другие. В данной работе будет рассматриваться оболочка **bash** (Bourne Again Shell), развитие традиционной оболочки **sh**.

Оболочка **bash** реализует консольный пользовательский интерфейс, т.е. поддерживает ввод команд с клавиатуры и вывод данных на дисплей. Команды оболочки делятся на внутренние и внешние. Внутренние команды обрабатываются непосредственно командной оболочкой. Внешние команды представляют собой вызовы внешних программ. Команда запуска (вызова) программы не имеет ключевого слова. Для запуска программы необходимо просто ввести имя программы (если она находится вне текущего каталога, то необходимо указать путь – абсолютный или относительный) и после имени (через пробел) необходимые параметры (если они, конечно, требуются). Одной из самых необходимых команд оболочки **bash** является команда **man**, предназначенная для вывода документации по командам оболочки, внешним программам, системным конфигурационным файлам и т.д. Формат этой команды - **man <имя>**. Для вывода справки по внутренним командам оболочки используется команда **help <имя>**. Информация по различным командам и программам также может быть получена с помощью команд **info <имя>** и **apropos <имя>**.

Далее будут перечислены основные команды оболочки **bash**.

Таблица 2.1 - Файловые команды

Команда	Описание
chgrp	Изменение принадлежности файла группе
chown	Изменение принадлежности файла пользователю
chmod	Изменение прав доступа к файлу
cp	Копирование файлов
df	Отображение сведений о свободном дисковом пространстве
du	Отображение сведений об использовании дискового пространства
find	Выполнение поиска файлов
ln	Создание ссылки
ls	Вывод содержимого каталогов
mkdir	Создание каталога
mv	Перемещение файлов

Команда	Описание
rm	Удаление файлов
rmdir	Удаление каталогов
touch	Модифицирование временных отметок файлов

Таблица 2.2 - Команды преобразования данных

Команда	Описание
cat	Выполнение конкатенации файлов
cmp	Сравнение файлов
diff	Вывод различия между файлами
grep	Выполнение поиска текста по шаблону
sort	Сортировка текстовых файлов

Таблица 2.3 - Системные команды

Команда	Описание
basename	Извлечение имени файла из пути
date	Вывод или установка системной даты и времени
dirname	Извлечение имени каталога из пути
echo	Вывод строки текста
env	Установка переменной среды
expr	Вычисление выражения
groups	Вывод групп, в которых состоит пользователь
hostname	Вывод или установка имени системы
id	Вывод информации о пользователе
ice	Модификация приоритета процесса
pwd	Вывод имени текущего каталога
sleep	Ожидание в течение указанного периода времени
su	Запуск оболочки от имени другого пользователя
who	Список пользователей, работающих в системе
read	Чтение из стандартного ввода
kill	Посылка сигнала процессу
ps	Список процессов

ОС семейства UNIX изначально являются многозадачными. Оболочка **bash** позволяет пользоваться этой многозадачностью. Команды в оболочке **bash** могут быть запущены на выполнение не только последовательно, но и параллельно. Управление порядком выполнения команд реализуется с помощью группировки команд. Средства группировки:

"**;**" и "**перевод строки**" определяют последовательное выполнение команд;

"**&**" - асинхронное (фоновое) выполнение предшествующей команды;

"**&&**" - выполнение последующей команды при условии нормального завершения предыдущей, иначе игнорировать;

"||" - выполнение последующей команды при ненормальном завершении предыдущей, иначе игнорировать;

"**{<список команд>}**" – объединение команд;

"**(<список команд>)**" – объединение команд с их исполнением в отдельном экземпляре оболочки.

При выполнении команды в асинхронном режиме (после команды стоит один амперсанд) на экран выводится номер процесса, соответствующий выполняемой команде, и оболочка, запустив этот фоновый процесс, вновь выходит на диалог с пользователем.

Стандартный ввод - "stdin" в ОС UNIX осуществляется с клавиатуры терминала, а стандартный вывод - "stdout" направлен на экран терминала. Существует еще и стандартный файл диагностических сообщений - "stderr". Оболочка **bash** поддерживает перенаправление этих потоков на другие файлы. Символы ">" и ">>" обозначают перенаправление вывода. Их различие в том, что ">" перезаписывает целевой файл, а ">>" – добавляет в него. Стандартные файлы имеют номера: 0 - stdin, 1 - stdout и 2 – stderr. Эти номера могут быть использованы в операциях перенаправления. Например, команда вида

<имя_команды> <параметры> 2>error.log

будет выводить все сообщения об ошибках в файл "error.log".

Символ "<" обозначает перенаправление ввода. Последовательность символов "**<< <строка>**" заставляет оболочку считывать данные из потока ввода до тех пор, пока не будет встречена <**строка**>.

Средство, объединяющее стандартный выход одной команды со стандартным входом другой, называется *конвейером* и обозначается вертикальной чертой "|".

2.2.2. Переменные окружения

Командная оболочка предоставляет возможность использования переменных. Имя переменной оболочки - это начинающаяся с буквы последовательность букв, цифр и подчеркиваний. Значение переменной оболочки - строка символов. Вообще, строки – это единственный тип данных, с которым работает интерпретатор оболочки. Все переменные и все константы являются строками.

Для присваивания значений переменным может использоваться оператор присваивания "=" (обратите внимание на то, что, как переменная, так и ее значение должны быть записаны без пробелов относительно символа "="). При обращении к переменной оболочки необходимо перед ее именем ставить символ "$".

Переменной можно присвоить значение не только какой-либо константы. Можно присвоить переменной значение, являющееся выводом какой-либо команды. Для этого в правой части оператора присваивания записывается имя нужной команды (со всеми параметрами), заключенное в обратные апострофы (символ "`"). Например, для присваивания переменной CURRENT_DIR имени текущего каталога можно воспользоваться командой

CURRENT_DIR=`pwd`.

Также переменной может быть присвоено значение, введенное пользователем, с помощью команды

read <имя_переменной>.

Для конкатенации переменных и констант используется их последовательная запись. В том случае, когда при такой записи имя переменной сливается с

конкатенируемой константой, они должны быть разделены с помощью фигурных скобок. Например, последовательность команд:

My_path=/home/user1/dir1/

cat ${My_path}my_text

выведет на экран содержимое файла /home/user1/dir1/my_text.

Язык оболочки поддерживает средства экранирования - двойные кавычки (" "), одинарные кавычки (' ') и обратную косую черту (бэкслэш) (\). Двойные кавычки экранируют в строке символы, использующиеся для управления оболочкой. Например, не допускается наличие в правой части оператора присваивания пробелов (т.к. они используются оболочкой как разделители команд). Если же необходимо присвоить переменной значение строки, содержащей пробел, то эту строку необходимо заключить в двойные кавычки. Одинарные кавычки способны также экранировать символ обращения к значению переменной, например, из команд:

echo "$My_path is the path to my files"

echo '$My_path is the path to my files'

первая выдаст на экран текст «/home/user1/dir1/ is the path to my files», а вторая - «$My_path is the path to my files». Символ обратной косой черты позволяет экранировать единичный следующий за ним символ, например, пробел или метасимвол в присваиваемой строке. Этот символ может экранировать и сам себя.

2.2.3. Сценарии оболочки

Часто повторяемые последовательности команд можно объединить в сценарий. Сценарий – это разновидность интерпретируемой программы. Сценарии оболочки **bash** представляют собой текстовые файлы, содержащие последовательность команд оболочки, вызовов внешних программ и других сценариев, специальных управления последовательностью выполнения, а также комментариев. Для запуска сценария на выполнение он должен быть передан оболочке в качестве параметра, например:

bash <имя_сценария>.

Для того чтобы сценарий автоматически распознавался оболочкой, необходимо проделать следующие действия. Во-первых, если сценарий пишется под оболочку **bash,** первая строка сценария должна начинаться с последовательности "**#!/bin/bash**". Во-вторых, файлу сценария должен быть присвоен атрибут исполнимости (с помощью команды **chmod**). После этого сценарий может быть вызван как обычная команда оболочки.

Сценарий может принимать и обрабатывать параметры из командной строки, а также переменные среды. Параметры командной строки доступны внутри сценария по именам %0-%9. Переменная %0 содержит имя самого файла сценария, а переменные %1-%9 – первые девять параметров командной строки. Если необходимо использовать большее число параметров, то после обработки первых девяти необходимо произвести сдвиг параметров командой **shift**. После первой команды **shift** в переменную %0 записывается значение переменной %1, в %1 - %2 и т.д. В переменную %9 записывается десятый параметр. Для получения прочих

параметров можно использовать команду **shift** необходимое число раз. Значения параметров могут быть установлены (или изменены) из самого сценария с помощью команды **set**. Команда **set** позволяет также осуществлять контроль выполнения программы. Некоторые параметры команды приведены в таблице 2.4:

Таблица 2.4 – Параметры команды set

set -v	на терминал выводятся строки, читаемые shell.
set +v	отменяет предыдущий режим.
set -x	на терминал выводятся команды перед выполнением.
set +x	отменяет предыдущий режим.

Комментарии в сценариях оформляются с помощью символа "#", за которой следует текст комментария. Концом комментария считается конец строки.

Команда **test** проверяет выполнение некоторого условия. В случае истинности условия возвращается значение «истина» (0), иначе – «ложь» (1). С использованием этой (встроенной) команды формируются операторы выбора и цикла языка оболочки. Два возможных формата команды: **test <условие>** или **[<условие>]**. Второй вариант используется чаще, т.к. является более привычным для программиста. Необходимо отметить, что и обе квадратные скобки, и составляющие условия должны быть отделены пробелами (т.к. на самом деле существует только команда с именем "**[**", а все остальное – и части условия, и символ "]" представляет собой параметры этой команды). В оболочке используются условия различных "типов"(см. таблицы 2.5-2.8).

Таблица 2.5 – Файловые условия

-f file	файл "file" является обычным файлом;
-d file	файл "file" - каталог;
-c file	файл "file" - специальный файл;
-r file	имеется разрешение на чтение файла "file";
-w file	имеется разрешение на запись в файл "file";
-s file	файл "file" не пустой.

Таблица 2.6 – Строчные условия:

str1 = str2	строки "str1" и "str2" совпадают;
str1 != str2	строки "str1" и "str2" не совпадают;
-n str1	строка "str1" существует (непустая);
-z str1	строка "str1" не существует (пустая).

Таблица 2.7 – Условия сравнения целых чисел:

x -eq y	"x" равно "y",
x -ne y	"x" неравно "y",
x -gt y	"x" больше "y",
x -ge y	"x" больше или равно "y",
x -lt y	"x" меньше "y",
x -le y	"x" меньше или равно "y".

Таблица 2.8 – Сложные условия

!	(not) инвертирует значение кода завершения.
-o	(or) соответствует логическому "ИЛИ".
-a	(and) соответствует логическому "И".

Существуют две специальные команды, **true** и **false**, не делающие ничего, но возвращающие значения «истина» и «ложь» соответственно.

Условный оператор **if** имеет структуру

if <условие>
then <список_команд>
[elif <условие>
then <список_команд >]
[else <список_команд >]
fi

Конструкция **elif** представляет собой сокращение от «else if» и предназначена для организации проверки вложенных условий. Данная конструкция, как и конструкция **else**, является необязательной (что и отмечено квадратными скобками – не путать с оператором проверки условия!). Условие здесь представляет собой любое выражение, имеющее значение «0» в качестве истины или «1» в качестве лжи. В качестве условия может использоваться подстановка любой команды, при этом команда выполнится, и на место вызова подставится ее код завершения. Чаще всего используется команда проверки условия **test ([])**.

Оператор выбора **case** имеет структуру:

case <строка> in
<шаблон>) <список_команд>;;
<шаблон>) <список_команд>;;
...
esac

Здесь **case**, **in** и **esac** - служебные слова. **<Строка>** (это может быть и один символ) сравнивается с **<шаблон>**ом. Затем выполняется **<список команд>** выбранной строки. Непривычным будет служебное слово **esac**, но оно необходимо для завершения структуры. Непривычно выглядят в конце строк выбора ";;", но написать здесь ";" было бы ошибкой. Для каждой альтернативы может быть выполнено несколько команд. Если эти команды будут записаны в одну строку, то символ ";" будет использоваться как разделитель команд. Обычно последняя строка выбора имеет шаблон "*", что в структуре "case" означает "любое значение". Эта строка выбирается, если не произошло совпадение значения переменной ни с одним из ранее записанных шаблонов. Значения просматриваются в порядке записи. Возможные значения в шаблоне могут объединяться через символ "|".

Оператор цикла **for** имеет структуру:

for <имя> [in <список_значений>]
do
<список_команд>
done

где **for** - служебное слово, определяющее тип цикла, **do** и **done** - служебные слова, выделяющие тело цикла. Фрагмент **in <список_значений>** может отсутствовать. В цикле переменной **<имя>** поочередно присваиваются значения из **<списка_значений>,** и для каждого значения переменной выполняется последовательность команд, заданная **<списком_команд>**. Часто используются формы "**for i in ***", означающая "для всех файлов текущего каталога" и "**for i in**", означающая "для всех параметров сценария".

Оператор итерационного цикла с предусловием **while** имеет структуру:

while <условие>
do
<список_команд>
done

где **while** - служебное слово, определяющее тип цикла предусловием. Список команд в теле цикла (между **do** и **done**) повторяется до тех пор, пока сохраняется истинность условия или цикл не будет прерван изнутри специальными командами (**break**, **continue** или **exit**). При первом входе в цикл условие должно выполняться.

Команда **break [n]** позволяет выходить из цикла. Если **n** отсутствует, то это эквивалентно **break 1**. **n** указывает число вложенных циклов, из которых надо выйти, например, **break 3** - выход из трех вложенных циклов.

В отличие от команды **break** команда **continue [n]** лишь прекращает выполнение текущего цикла и возвращает на начало цикла. Она также может быть с параметром. Например, **continue 2** означает выход на начало второго (если считать из глубины) вложенного цикла.

Команда **exit [n]** позволяет выйти вообще из функции с кодом возврата "0" или "n" (если параметр **n** указан). Эта команда может использоваться не только в циклах, но и для выхода из функций.

Оператор итерационного цикла с постусловием **until** имеет структуру:

until <условие>
do
<список_команд>
done

где **until** - служебное слово, определяющее тип цикла с постусловием. Список команд в теле цикла (между **do** и **done**) повторяется до тех пор, пока сохраняется ложность условия или цикл не будет прерван изнутри специальными командами ("break", "continue" или "exit"). При первом входе в цикл условие не должно выполняться. Отличие от оператора "while" состоит в том, что условие цикла проверяется на ложность после каждого (в том числе и первого) выполнения команд тела цикла.

Пустой оператор имеет формат

:

Ничего не делает. Возвращает значение "0" («истина»).

Язык сценариев оболочки **bash** предусматривает определение пользовательских функций внутри сценария. Описание функции имеет вид:

имя()
{
список команд
}

после чего обращение к функции происходит по имени. При выполнении функции не создается нового процесса. Она выполняется в среде соответствующего процесса. Аргументы функции становятся ее позиционными параметрами; имя функции - ее нулевой параметр. Прервать выполнение функции можно оператором **return [n]**, где (необязательное) **n** - код возврата.

3. ВАРИАНТЫ ЗАДАНИЙ И ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Загрузите ОС Linux и выполните процедуру входа в систему;
2. Создайте с помощью команд оболочки **bash** свое дерево каталогов и свой набор файлов согласно описанным ниже правилам (создание файла может быть осуществлено как вывод нужных данных с помощью команды **echo** и последующим перенаправлением этого вывода в файл, либо копированием с пользовательского терминала /dev/tty1 в файл). Создайте каталог с именем, образованным вашей фамилией, именем и номером учебной группы, разделенных точками, например "Ivanov.Vasiliy.i21d". В данном каталоге создайте три подкаталога с именами, образованными из ваших инициалов и порядкового номера каталога, например, "I.V.I.1", "I.V.I.2", "I.V.I.3". В первом подкаталоге создать файл, имя которого совпадает с Вашим и заканчивается последовательностью символов ".txt", например "Vasiliy.txt". В файл поместить фамилию, имя, отчество, номер группы;
3. Произведите ряд операций над файлами и каталогами с использованием команд оболочки **bash**:
   - 3.1. Скопировать созданный файл во второй каталог.
   - 3.2. Переименовать файл во втором каталоге, переставив буквы имени в обратном порядке, например "yilisaV.txt".
   - 3.3. Объединить файлы из первых двух подкаталогов и результат поместить в третий с одним из выбранных Вами имен.
   - 3.4. Переместить результирующий файл из третьего каталога в каталог верхнего уровня.
   - 3.5. Вывести содержимое этого файла на экран дисплея.
   - 3.6. Продемонстрировать преподавателю результаты работы.
   - 3.7. Уничтожить созданные файлы и каталоги.
4. Написать сценарий оболочки **bash**, реализующие описанные в предыдущем пункте действия со следующими особенностями. Необходимо предусмотреть ввод имени третьего файла в качестве параметра командной строки, а при отсутствии параметра задать пользователю вопрос о том, какое имя следует использовать для создания третьего файла (имя файла в этом случае должно вводиться пользователем). Продублировать выводимые на дисплей сообщения в файл протокола с именем, соответствующим имени файла сценария с добавлением в конце имени последовательности символов «.log».

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1) Основные характеристики ОС семейства UNIX?
2) Каковы основные особенности структуры файловой системы ОС семейства UNIX?
3) Какие атрибуты файлов поддерживаются ОС семейства UNIX?
4) Что такое абсолютный и относительный путь?
5) Как реализуется взаимодействие ОС семейства UNIX с пользователем?
6)Что такое сценарии оболочки UNIX?
7)Что такое переменные окружения и как их использовать в сценариях оболочки UNIX?
8) Как производится ввод-вывод из сценария?
9) Как производится управление выполнением сценария?
10) Какие способы перенаправления потоков ввода-вывода поддерживаются ОС семейства UNIX?

ПРИЛОЖЕНИЕ А

Соответствие некоторых команд command.com и bash

command.com	bash	Примечание
ATTRIB (+-)attr file	chmod <mode> file	полностью отличаются
CD dirname\\	cd dirname/	почти тот же самый синтаксис
COPY file1 file2	cp file1 file2	то же самое
DEL file	rm file	нет восстановления файлов
DELTREE dirname	rm -R dirname/	то же самое
DIR	ls	не полностью похожий синтаксис
FORMAT	fdformat, mount, umount	достаточно отличный синтаксис
HELP command	man command,info command	
MD dirname	mkdir dirname/	почти тот же самый синтаксис
MORE file	less file	намного лучше
MOVE file1 file2	mv file1 file2	то же самое
NUL	/dev/null	то же самое
PRINT file	lpr file	то же самое
PRN	/dev/lp0,/dev/lp1	то же самое
RD dirname	rmdir dirname/	почти тот же самый синтаксис
REN file1 file2	mv file1 file2	не для множества файлов
TYPE file	less file	намного лучше

ЛАБОРАТОРНАЯ РАБОТА №7
Исследование подсистемы управления процессами и потоками в ОС семейства UNIX

1. ЦЕЛЬ РАБОТЫ

Изучение понятий процесса и потока ОС семейства UNIX. Приобретение практических навыков разработки программ на языке Си в ОС UNIX, а также написания программ с использованием системных вызовов создания и управления процессами и потоками в ОС UNIX.

2. ОСНОВНЫЕ ПОЛОЖЕНИЯ

2.1. Процесс в UNIX

2.1.1. Контекст процесса

Контекст процесса операционной системы UNIX складывается из пользовательского контекста и контекста ядра, как изображено на рисунке 2.1.

Под пользовательским контекстом процесса понимают код и данные, расположенные в адресном пространстве процесса. Все данные подразделяются на:

- инициализируемые неизменяемые данные (например, константы);
- инициализируемые изменяемые данные (все переменные, начальные значения которых присваиваются на этапе компиляции);
- не инициализируемые изменяемые данные (все статические переменные, которым не присвоены начальные значения на этапе компиляции);
- стек пользователя;
- данные, расположенные в динамически выделяемой памяти (например, с помощью стандартных библиотечных C функций malloc(), calloc(), realloc()).

Исполняемый код и инициализируемые данные составляют содержимое файла программы, который исполняется в контексте процесса. Пользовательский стек применяется при работе процесса в пользовательском режиме (user-mode).

Под понятием «контекст ядра» объединяются системный контекст и регистровый контекст. В контексте ядра можно выделить стек ядра, который используется при работе процесса в режиме ядра (kernel mode), и данные ядра, хранящиеся в структурах, являющихся аналогом блока управления процессом — PCB. В данные ядра, кроме прочих, входят: идентификатор пользователя — UID, групповой идентификатор пользователя — GID, идентификатор процесса — PID, идентификатор родительского процесса — PPID.

Рисунок 2.1 - Контекст процесса в UNIX

2.1.2. Идентификация процесса

Каждый процесс в операционной системе получает уникальный идентификационный номер – PID (process identificator). При создании нового процесса операционная система пытается присвоить ему свободный номер больший, чем у процесса, созданного перед ним. Если таких свободных номеров не оказывается (например, мы достигли максимально возможного номера для процесса), то операционная система выбирает минимальный номер из всех свободных номеров. В операционной системе Linux присвоение идентификационных номеров процессов начинается с номера 0, который получает процесс kernel при старте операционной системы. Этот номер впоследствии не можетбыть присвоен никакому другому процессу. Максимально возможное значение для номера процесса в Linux на базе 32-разрядных процессоров Intel составляет 2^{31}-1.

2.1.3. Состояния процесса. Краткая диаграмма состояний

Модель состояний процессов в операционной системе UNIX представляет собой детализацию модели состояний, принятой в лекционном курсе. Краткая диаграмма состояний процессов в операционной системе UNIX изображена на рисунке 2.2:

Рисунок 2.2 - Сокращенная диаграмма состояний процесса в UNIX

Состояние *исполнение* процесса в Unix разделяется на два состояния: исполнение в режиме ядра и исполнение в режиме пользователя. В состоянии исполнение в режиме пользователя процесс выполняет прикладные инструкции пользователя. В состоянии исполнение в режиме ядра выполняются инструкции ядра операционной системы в контексте текущего процесса (например, при обработке системного вызова или прерывания). Из состояния исполнение в режиме пользователя процесс не может непосредственно перейти в состояния *ожидание*, *готовность* и *закончил исполнение*. Такие переходы возможны только через промежуточное состояние «исполняется в режиме ядра». Также запрещен прямой переход из состояния *готовность* в состояние исполнение в режиме пользователя.

Следует отметить, что приведенная выше диаграмма состояний процессов в UNIX не является полной.

2.1.4. Иерархия процессов

В операционной системе UNIX все процессы, кроме одного, создающегося при старте операционной системы, могут быть порождены только какими-либо другими процессами. В качестве прародителя всех остальных процессов в подобных UNIX системах могут выступать процессы с номерами 1 или 0. В операционной системе Linux таким родоначальником, существующим только при загрузке системы, является процесс kernel с идентификатором 0.

Таким образом, все процессы в UNIX связаны отношениями процесс-родитель – процесс-ребенок и образуют генеалогическое дерево процессов. Для сохранения целостности генеалогического дерева в ситуациях, когда процесс-родитель завершает свою работу до завершения выполнения процесса-ребенка, идентификатор родительского процесса в данных ядра процесса-ребенка (PPID – parent process identificator) изменяет свое значение на значение 1, соответствующее идентификатору процесса init, время жизни которого определяет время функционирования операционной системы. Тем самым процесс init как бы усыновляет осиротевшие процессы.

2.1.5. Системные вызовы getppid() и getpid()

Данные ядра, находящиеся в контексте ядра процесса, не могут быть прочитаны процессом непосредственно. Для получения информации о них процесс должен совершить соответствующий системный вызов. Значение идентификатора текущего процесса может быть получено с помощью системного вызова *getpid()*, а значение идентификатора родительского процесса для текущего процесса – с помощью системного вызова *getppid()*. Прототипы этих системных вызовов и соответствующие типы данных описаны в системных файлах <sys/types.h> и <unistd.h>. Системные вызовы не имеют параметров и возвращают идентификатор текущего процесса и идентификатор родительского процесса соответственно. Ниже представлены прототипы системных вызовов данных функций:

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

Тип данных pid_t является синонимом одного из целочисленных типов языка С.

2.1.6. Создание процесса в UNIX. Системный вызов fork()

В операционной системе UNIX новый процесс может быть порожден единственным способом – с помощью системного вызова *fork()*. При этом вновь созданный процесс будет являться практически полной копией родительского процесса. У порожденного процесса по сравнению с родительским процессом изменяются значения следующих параметров:

- идентификатор процесса – PID;
- идентификатор родительского процесса – PPID.

Ниже предствлен прототип системного вызова:

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

Процесс, который инициировал системный вызов *fork*, принято называть родительским процессом (parent process). Вновь порожденный процесс принято называть процессом-ребенком (child process). Процесс-ребенок является почти полной копией родительского процесса. У порожденного процесса по сравнению с родительским изменяются значения следующих параметров:

- идентификатор процесса;
- идентификатор родительского процесса;
- время, оставшееся до получения сигнала SIGALRM;
- сигналы, ожидавшие доставки родительскому процессу, не будут доставляться порожденному процессу.

В процессе выполнения системного вызова *fork()* порождается копия родительского процесса и возвращение из системного вызова будет происходить уже как в родительском, так и в порожденном процессах. Этот системный вызов

является единственным, который вызывается один раз, а при успешной работе возвращается два раза (один раз в процессе-родителе и один раз в процессе-ребенке). После выхода из системного вызова *fork()* оба процесса продолжают выполнение кода, следующего за этим системным вызовом.

Ниже приведена программа пример создания нового процесса с одинаковой работой процессов ребенка и родителя:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(){
   pid_t pid, ppid;
   int a = 0;
   (void)fork();
   /* При успешном создании нового процесса  с этого места
псевдопараллельно  начинают работать два процесса: старый и новый */
   /* Перед выполнением следующего выражения  значение переменной a в
обоих процессах равно 0 */
   a = a+1;
   /* Узнаем идентификаторы текущего и родительского процесса (в каждом
из  процессов!) */
   pid = getpid();
   ppid = getppid();
   /* Печатаем значения PID, PPID и вычисленное значение переменной a (в
каждом из процессов !!!) */
   printf("My pid = %d, my ppid = %d, result = %d\n", (int)pid, (int)ppid, a);
   return 0;
}
```

Для того чтобы после возвращения из системного вызова *fork()* процессы могли определить, кто из них является ребенком, а кто родителем, и, соответственно, по-разному организовать свое поведение, системный вызов возвращает в них разные значения. При успешном создании нового процесса в процесс-родитель возвращается положительное значение, равное идентификатору процесса-ребенка. В процесс-ребенок же возвращается значение 0. Если по какой-либо причине создать новый процесс не удалось, то системный вызов вернет в инициировавший его процесс значение -1. Таким образом, общая схема организации различной работы процесса-ребенка и процесса-родителя выглядит так:

```
pid = fork();
if(pid == -1){
   ...
   /* ошибка */
   ...
} else if (pid == 0){
   ...
   /* ребенок */
```

```
    ...
} else {
    ...
    /* родитель */
    ...
}
```

2.1.7. Завершение процесса. Функция exit()

Существует два способа корректного завершения процесса в программах, написанных на языке С. Первый способ: процесс корректно завершается по достижении конца функции main() или при выполнении оператора return в функции main(), второй способ применяется при необходимости завершить процесс в каком-либо другом месте программы. Для этого используется функция *exit()* из стандартной библиотеки функций для языка С. Прототип этой функции представлен ниже:

```
#include <stdlib.h>
void exit(int status);
```

При выполнении этой функции происходит сброс всех частично заполненных буферов ввода-вывода с закрытием соответствующих потоков, после чего инициируется системный вызов прекращения работы процесса и перевода его в состояние *закончил исполнение*.

Возврата из функции в текущий процесс не происходит. Значение параметра *status* функции *exit()* – кода завершения процесса – передается ядру операционной системы и может быть получено процессом, породившим завершившийся процесс. Следует отметить, что при достижении конца функции main() любой программы также неявно вызывается эта функция со значением параметра 0.

2.1.8. Изменение пользовательского контекста процесса. Семейство функций для системного вызова exec()

Для изменения пользовательского контекста процесса применяется системный вызов *exec()*, который пользователь не может вызвать непосредственно. Вызов *exec()* заменяет пользовательский контекст текущего процесса на содержимое некоторого исполняемого файла и устанавливает начальные значения регистров процессора (в том числе устанавливает программный счетчик на начало загружаемой программы). Этот вызов требует для своей работы задания имени исполняемого файла, аргументов командной строки и параметров окружающей среды. Для осуществления вызова программист может воспользоваться одной из шести функций: *execlp(), execvp(), execl() , execv(), execle()* и *execve()*, отличающихся друг от друга представлением параметров, необходимых для работы системного вызова *exec()*. Взаимосвязь указанных выше функций изображена на рисунке 2.3:

Рисунок 2.3 - Взаимосвязь функций для выполнения системного вызова exec()

Прототипы функций представлены ниже:

```
#include <unistd.h>
int execlp(const char *file,  const char *arg0, const char *argN,(char *)NULL)
int execvp(const char *file, char *argv[])
int execl(const char *path, const char *arg0, const char *argN,(char *)NULL)
int execv(const char *path, char *argv[])
int execle(const char *path, const char *arg0, const char *argN,(char *)NULL,
   char * envp[])
int execve(const char *path, char *argv[],char *envp[])
```

Аргумент *file* является указателем на имя файла, который должен быть загружен. Аргумент *path* – это указатель на полный путь к файлу, который должен быть загружен.

Аргументы *arg0*, ..., *argN* представляют собой указатели на аргументы командной строки. Заметим, что аргумент *arg0* должен указывать на имя загружаемого файла. Аргумент *argv* представляет собой массив из указателей на аргументы командной строки. Начальный элемент массива должен указывать на имя загружаемой программы, а заканчиваться массив должен элементом, содержащим указатель NULL.

Аргумент *envp* является массивом указателей на параметры окружающей среды, заданные в виде строк «переменная=строка». Последний элемент этого массива должен содержать указатель NULL.

Поскольку вызов функции не изменяет системный контекст текущего процесса, загруженная программа унаследует от загрузившего ее процесса следующие атрибуты:

- идентификатор процесса;
- идентификатор родительского процесса;
- групповой идентификатор процесса;
- идентификатор сеанса;
- время, оставшееся до возникновения сигнала SIGALRM;
- текущую рабочую директорию;
- маску создания файлов;
- идентификатор пользователя;
- групповой идентификатор пользователя;
- явное игнорирование сигналов;
- таблицу открытых файлов (если для файлового дескриптора не устанавливался признак «закрыть файл при выполнении *exec()*»).

В случае успешного выполнения возврата из функций в программу, осуществившую вызов, не происходит, а управление передается загруженной программе. В случае неудачного выполнения в программу, инициировавшую вызов, возвращается отрицательное значение

Поскольку системный контекст процесса при вызове *exec()* остается практически неизменным, большинство атрибутов процесса, доступных пользователю через системные вызовы (PID, UID, GID, PPID и другие), после запуска новой программы также не изменяется.

Важно понимать разницу между системными вызовами *fork()* и *exec()*. Системный вызов *fork()* создает новый процесс, у которого пользовательский контекст совпадает с пользовательским контекстом процесса-родителя. Системный вызов *exec()* изменяет пользовательский контекст текущего процесса, не создавая новый процесс.

Для иллюстрации использования системного вызова *exec()* рассмотрим следующую программу:

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(int argc, char *argv[], char *envp[]){
(void) execle("/bin/cat", "/bin/cat", "demo.c", 0, envp);
/* Сюда попадаем только при  возникновении ошибки */
printf("Error on program start\n");
exit(-1);
return 0;     /* Никогда не выполняется, нужен  для того, чтобы компилятор
не выдавал warning */
}
```

В данном примере в качестве программы запускается команда cat с аргументом командной строки "demo.c" без изменения параметров среды, т.е. фактически выполняется команда "cat demo.c", которая должна вывести содержимое файла demo.c на экран. Для функции *execle* в качестве имени программы указывается ее полное имя с путем от корневой директории - /bin/cat. Первое слово в командной строке должно совпадать с именем запускаемой программы. Второе слово в командной строке (параметр команды cat) – это имя файла, содержимое которого необходимо распечатать.

2.2. Нити исполнения (thread) в UNIX.

2.2.1. Идентификатор нити исполнения. Функция pthread_self()

Во многих современных операционных системах существует расширенная реализация понятия процесс, когда процесс представляет собой совокупность выделенных ему ресурсов и набора нитей(потоков) исполнения. Нити процесса разделяют его программный код, глобальные переменные и системные ресурсы, но каждая нить имеет собственный программный счетчик, свое содержимое регистров и свой стек. Поскольку глобальные переменные у нитей исполнения являются

общими, они могут использовать их, как элементы разделяемой памяти, не прибегая к механизму, создание разделяемой памяти.

В различных версиях операционной системы UNIX существуют различные интерфейсы, обеспечивающие работу с нитями исполнения. Мы кратко ознакомимся с некоторыми функциями, позволяющими разделить процесс на потоки и управлять их поведением, в соответствии со стандартом POSIX. Нити исполнения, удовлетворяющие стандарту POSIX, принято называть POSIX thread'ами или, кратко, pthread'ами.

Операционная система Linux не полностью поддерживает нити исполнения на уровне ядра системы. При создании нового потока запускается новый традиционный процесс, разделяющий с родительским традиционным процессом его ресурсы, программный код и данные, расположенные вне стека, т.е. фактически действительно создается новый thread, но ядро не умеет определять, что эти потоки являются составными частями одного целого. Это «знает» только специальный процесс-координатор, работающий на пользовательском уровне и стартующий при первом вызове функций, обеспечивающих POSIX интерфейс для нитей исполнения.

Каждая нить исполнения, как и процесс, имеет в системе уникальный номер – идентификатор потока. Поскольку традиционный процесс в концепции нитей исполнения трактуется как процесс, содержащий единственную нить исполнения, мы можем узнать идентификатор этой нити и для любого обычного процесса. Для этого используется функция *pthread_self()*. Нить исполнения, создаваемую при рождении нового процесса, принято называть *начальной* или *главной* нитью исполнения этого процесса.

Ниже приведен прототип функции *pthread_self()*:

```
#include <pthread.h>
pthread_t pthread_self(void);
```

Тип данных pthread_t является синонимом одного из целочисленных типов языка С.

2.2.2. Создание и завершение потока. Функции pthread_create(), thread_exit(), pthread_join()

Нити исполнения, как и традиционные процессы, могут порождать нити-потомки, правда, только внутри своего процесса. Каждый будущий поток внутри программы должен представлять собой функцию с прототипом

```
void *thread(void *arg);
```

Параметр arg передается этой функции при создании потока и может, до некоторой степени, рассматриваться как аналог параметров функции main(). Возвращаемое функцией значение может интерпретироваться как аналог информации, которую родительский процесс может получить после завершения процесса-ребенка. Для создания новой нити исполнения применяется функция *pthread_create()*.Прототип этой функции представлен ниже:

```
#include <pthread.h>
int pthread_create(pthread_t *thread, pthread_attr_t *attr,
   void * (*start_routine)(void *), void *arg);
```

Прототип функции потока представлен ниже:

*void *start_routine(void *)*

Новый поток будет выполнять функцию *start_routine,* передавая ей в качестве аргумента параметр *arg*. Если требуется передать более одного параметра, они собираются в структуру, и передается адрес этой структуры.

Значение, возвращаемое функцией *start_routine,* не должно указывать на динамический объект данного потока.

Параметр *attr* служит для задания различных атрибутов создаваемого потока. Для использования параметров заданных по умолчанию, необходимо подставить в качестве аргумента значение NULL.

При удачном завершении функция возвращает значение 0 и помещает идентификатор новой нити исполнения по адресу, на который указывает параметр thread. В случае ошибки возвращается *положительное значение* (а не отрицательное, как в большинстве системных вызовов и функций), которое определяет код ошибки, описанный в файле <errno.h>. Значение системной переменной *errno* при этом не устанавливается.

Созданный поток может завершить свою деятельность тремя способами:

- С помощью выполнения функции *pthread_exit()*;
- С помощью возврата из функции, ассоциированной с нитью исполнения;
- Если в процессе выполняется возврат из функции main() или где-либо в процессе (в любой нити исполнения) осуществляется вызов функции exit(), это приводит к завершению всех потоков процесса.

Функция *pthread_exit* служит для завершения нити исполнения(thread) текущего процесса. Прототип этой функции:

#include <pthread.h>

*void pthread_exit(void *status);*

Функция *pthread_exit* никогда не возвращает результат в вызвавший ее поток. Объект, на который указывает параметр *status*, может быть впоследствии проанализирован в другой нити исполнения, например в нити, породившей завершившуюся нить. Параметр *status* не должен указывать на динамический объект завершившегося потока и должен указывать на объект, не являющийся локальным для завершившегося потока.

Одним из вариантов получения адреса, возвращаемого завершившимся потоком, с одновременным ожиданием его завершения является использование функции *pthread_join()*:

#include <pthread.h>

*int pthread_join (pthread_t thread,void **status_addr);*

Функция *pthread_join* блокирует работу вызвавшей ее нити исполнения до завершения потока с идентификатором thread. После разблокирования в указатель, расположенный по адресу status_addr, заносится адрес, который вернул завершившийся поток либо при выходе из ассоциированной с ним функции, либо при выполнении функции *pthread_exit()*. Если нет необходимости анализа, значения возвращаемого нитью исполнения, в качестве этого параметра можно использовать значение NULL.

Для иллюстрации вышесказанного рассмотрим программу, в которой работают две нити исполнения:

```
/* Программа для иллюстрации работы двух нитей исполнения. Каждая нить
исполнения просто увеличивает на 1 разделяемую переменную a. */
#include <pthread.h>
#include <stdio.h>
int a = 0;
/* Переменная a является глобальной статической для всей программы, поэтому
она будет разделяться обеими нитями исполнения.*/
/* Ниже следует текст функции, которая будет ассоциирована со 2-м потоком */
void *mythread(void *dummy)
/* Параметр dummy в нашей функции не используется и присутствует только для
совместимости типов данных. По той же причине функция возвращает значение
void *, хотя это никак не используется в программе.*/
{
   pthread_t mythid; /* Для идентификатора нити исполнения */
   /* Заметим, что переменная mythid является динамической локальной переменной
функции mythread(), т. е. помещается в стеке и, следовательно, не разделяется
нитями исполнения. */
   mythid = pthread_self();    /* Запрашиваем идентификатор потока */
   a = a+1;
   printf("Thread %d, Calculation result = %d\n",
      mythid, a);
   return NULL;
}
/* Функция main() – она же ассоциированная функция главного потока */
int main(){
   pthread_t thid, mythid;
   int result;
   /* Пытаемся создать новую нить исполнения, ассоциированную с функцией
mythread(). Передаем ей в качестве параметра значение NULL. В случае удачи в
переменную thid занесется идентификатор нового потока.
Если возникнет ошибка, то прекратим работу. */
   result = pthread_create( &thid,
      (pthread_attr_t *)NULL, mythread, NULL);
   if(result != 0){
   printf ("Error on thread create, return value = %d\n", result);
   exit(-1);
   }
   printf("Thread created, thid = %d\n", thid);
   /* Запрашиваем идентификатор главного потока */
   mythid = pthread_self();
   a = a+1;
   printf("Thread %d, Calculation result = %d\n",   mythid, a);
   /* Ожидаем завершения порожденного потока, не интересуясь, какое значение
он вернет. Если не  выполнить вызов этой функции, то возможна ситуация, когда
мы завершим функцию main() до того, как выполнится порожденный thread, что
```

```
автоматически повлечет за  собой его завершение, и искажение результатов. */
   pthread_join(thid, (void **)NULL);
   return 0;
}
```

Обратите внимание на отличие результатов этой программы от похожей программы, иллюстрировавшей создание нового процесса. Программа, создававшая новый процесс, печатала дважды одинаковые значения для переменной a, так как адресные пространства различных процессов независимы, и каждый процесс прибавлял 1 к своей собственной переменной a. Рассматриваемая программа печатает два разных значения, так как переменная a является разделяемой, и каждый поток прибавляет 1 к одной и той же переменной.

2.3. Компиляция программ на языке C в UNIX и их запуск

Для компиляции программ, написанных на языке C в Linux можно использовать компилятор *gcc*. В простейшем случае откомпилировать программу можно, запуская компилятор с именем файла, содержащего исходный текст программы на языке C:

gcc имя_исходного_файла

Необходимо, чтобы исходные файлы, содержащие текст программы, имели имена, заканчивающиеся на *".c"*.

Если программа была написана без ошибок, то компилятор создаст исполняемый файл с именем a.out. Изменить имя создаваемого исполняемого файла можно, задав его с помощью опции -o:

gcc имя_исходного_файла -o имя_исполняемого_файла

Для сборки исполняемого файла, использующего функции для работы с pthread'ами при работе редактора связей необходимо явно подключить библиотеку функций pthread, которая не подключается автоматически. Для этого к команде компиляции и редактирования связей необходимо добавить параметр *-lpthread:*

gcc имя_исходного_файла -o имя_исполняемого_файла -lpthread.

Для запуска полученной после компиляции программы необходимо набрать в командной строке имя исполняемого файла, указав что файл находится в текущей директории(для этого перед именем поставить "./"):

./имя_исполняемого_файла

3. ВАРИАНТЫ ЗАДАНИЙ И ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

3.1. Написать на языке C программу **Sort**, реализующую следующие действия:

3.1.1. Получить целое число(номер процесса) из первого аргумента программной строки;

3.1.2. Получить с использованием системных вызовов getppid() и getpid() идентификаторы родительского и текущего процесса;

3.1.3. Вывести на экран номер процесса, идентификаторы родительского и текущего процесса;

3.1.4. Заполнить массив целых чисел случайными значениями из диапазона 0-100;

3.1.5. Отсортировать массив;

3.1.6. Вывести на экран отсортированный массив (перед каждым элементом массива выводить номер процесса, полученный в п 3.1.1).

Метод и направление сортировки, а также количество элементов массива N выбирается в соответствии с вариантом задания, приведенным в таблице 3.1.

3.2. Написать на языке С программу Master, выполняющую следующие действия:

3.2.1. Получить с использованием системных вызовов getppid() и getpid() идентификаторы родительского и текущего процесса и вывести их на экран;

3.2.2. Используя системные вызовы *fork()* и *exec()* создать M процессов (функция для выполнения системного вызова *exec()* и количество процессов M определяются по варианту задания таблица 3.1)

3.3. Написать на языке С программу **Threads,** содержащую процедуру сортировки массива - **sort**, и процедуру вывода массива на экран – **mass_print**. Процедура **sort** должна получать идентификатор собственного потока и выводить его на экран (в формате **sort:pthreadId=***threadId*), после чего сортировать массив. Процедура **mass_print** должна получать идентификатор собственного потока, выводить его на экран (в формате **mass_print:pthreadId=***threadId*) и выводить на экран массив. Программа **Threads** должна выполнять следующие действия:

3.3.1. Заполнить массив случайными числами;

3.3.2. Используя системные вызовы *pthread_create()***,** создать потоки **sort** и **mass_print** и получить идентификаторы созданных потоков.

3.3.3. Вывести на экран идентификаторы созданных потоков (в формате **threads:sort:pthreadId=***threadId***; mass_print:pthreadId=***threadId*).

3.3.4. Ожидать завершения потоков (с использованием вызова *pthread_join*).

3.4. Модифицировать программу **Threads** так, чтобы обеспечить синхронизацию потоков. Поток **mass_print**, после запуска должен ожидать завершения потока **sort** (с использованием вызова *pthread_join)*, и лишь затем выводить массив на экран (для этого необходимо при создании потока **mass_print** в параметре *arg* передать идентификатор потока **sort**).

Таблица 3.1 - Варианты заданий

№ варианта	Метод сортировки	Направление сортировки	N	Функция для выполнения *exec()*	Количество процессов M
1	Шелла	Убыв.	30	*execlp()*	4
2	Быстрая	Убыв.	35	*execvp()*	3
3	Пузырька	Убыв.	40	*execl()*	2
4	Выбора	Убыв.	45	*execv()*	4
5	Вставки	Убыв.	64	*execle()*	3
6	Шелла	Убыв.	31	*execve()*	2
7	Быстрая	Убыв.	36	*execlp()*	4
8	Пузырька	Убыв.	41	*execvp()*	3
9	Выбора	Убыв.	46	*execl()*	2
10	Вставки	Убыв.	32	*execv()*	4
11	Шелла	Убыв.	32	*execle()*	3
12	Быстрая	Убыв.	37	*execve()*	2
13	Пузырька	Убыв.	42	*execlp()*	4

№ варианта	Метод сортировки	Направление сортировки	N	Функция для выполнения *exec()*	Количество процессов M
14	Выбора	Убыв.	47	*execvp()*	3
15	Вставки	Убыв.	128	*execl()*	2
16	Шелла	Возр.	33	*execv()*	4
17	Быстрая	Возр.	38	*execle()*	3
18	Пузырька	Возр.	43	*execve()*	2
19	Выбора	Возр.	48	*execlp()*	4
20	Вставки	Возр.	64	*execvp()*	3
21	Шелла	Возр.	34	*execl()*	2
22	Быстрая	Возр.	39	*execv()*	4
23	Пузырька	Возр.	44	*execle()*	3
24	Выбора	Возр.	49	*execve()*	2
25	Вставки	Возр.	32	*execlp()*	4
26	Шелла	Возр.	35	*execvp()*	3
27	Быстрая	Возр.	40	*execl()*	2
28	Пузырька	Возр.	45	*execv()*	4
29	Выбора	Возр.	50	*execle()*	3
30	Вставки	Возр.	64	*execve()*	2

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1) Поясните отличия между пользовательским контекстом и контекстом ядра процесса в ОС UNIX?
2) Как операционная система UNIX различает процессы?
3) Приведите и поясните сокращенную диаграмму состояний процессов в ОС UNIX?
4) Как образуется генеалогическое дерево процессов UNIX?
5) Какие процессы в ОС UNIX имеют PPID=1?
6) Как узнать идентификатор текущего процесса в ОС UNIX?
7) Как узнать идентификатор процесса-родителя в ОС UNIX?
8) Каким образом в ОС UNIX порождаются новые процессы?
9) Какое значение возвращает системный вызов *fork()*?
10) Каковы способы завершения текущего процесса в ОС UNIX?
11) Поясните назначение и отличия системных вызовов *fork()* и *exec()*?
12) Расскажите о функциях, используемых для выполнения системного вызова *exec()*?
13) Что возвращается в результате системного вызова *exec()*?
14) Расскажите о нитях исполнения в ОС UNIX?
15) Какие функции используются для создания и завершения потоков ОС UNIX?
16) Как можно осуществить синхронизацию потоков в ОС UNIX?
17) Для чего используется функция *pthread_self()*?

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Вильямс А. Системное программирование в Windows2000 С-Пб:Питер, 2001. – 624 с. ил.

2. Соловьев А. Программирование на Shell (UNIX). Учебное пособие.

3. Лепаж И., Яррера, П. UNIX. Библия системного администратора. :Пер. с англ. – М.:Диалектика, 1999.

4. ASP Linux 9.0 Man Pages.

5. Guido Gonzato. DOS-Win-to-Linux-HOWTO.

6. Фигурнов В.Э. IBM PC для пользователя.- М.: Финансы и статистика, 1994.- 367 с.

7. Справочник по персональным ЭВМ/ Н.И. Алишов, Н.В. Нестеренко, Б.В. Новиков и др.; Под ред. чл.-кор. АН УССР Б.Н. Малиновского. – К.: Тэхника, 1990.- 384 с.

Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего профессионального образования «Севастопольский государственный университет»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ

к лабораторному практикуму

по дисциплине

"Операционные системы "

для студентов дневной и заочной формы обучения

специальности 09.03.02 "Информационные системы и технологии"

часть 1

Севастополь

2015

УДК 004.8

Методические указания к лабораторному практикуму по дисциплине по дисциплине «**Операционные системы**» для студентов дневного и заочного отделения 09.03.02 «Информационные системы и технологии» **/Сост. А. Л. Овчинников – Севастополь: Изд-во СевНТУ, 2015. – 63с.**

Методические указания предназначены для проведения лабораторных занятий по дисциплине «**Операционные системы**». Целью настоящих методических указаний является обучение студентов практическим навыкам разработки сценариев для командного интерпретатора операционной системы Windows, создания приложений с использованием возможностей программного интерфейса приложений Win API по управлению и синхронизации процессов и потоков в ОС семейства Windows.

Методические указания составлены в соответствии с требованиями программы дисциплины «Операционные системы» для студентов специальности 09.03.02 и утверждены на заседании кафедры информационных систем, протокол № от 2015 года.

Допущено учебно-методическим центром СевНТУ в качестве методических указаний.

Рецензент Сергеев Г.Г., к.т.н., доцент кафедры КиВТ.

СОДЕРЖАНИЕ

ЛАБОРАТОРНАЯ РАБОТА №1
Исследование возможностей командного интерпретатора ОС Windows.

1. Цель работы

Ознакомится с возможностями командного интерпретатора ОС Windows. Изучить основные команды интерпретатора ОС Windows. Приобрести практические навыки написания сценариев интерпретатора ОС Windows (bat-файлов).

2. Основные положения

2.1. Семейство операционных систем Microsoft Windows

Microsoft Windows - семейство операционных систем корпорации Microsoft, изначально ориентированных на применение графического интерфейса при управлении и использовании.

Первые версии ОС Windows(1.0-3.11) не были полноценными операционными системами, а являлись надстройками к операционной системе MS-DOS(ОС MS DOS была создана корпорацией Microsoft в 1981 году в качестве основной ОС для появившихся в то время компьютеров серии IBM PC), и были по сути многофункциональным расширением, добавляя поддержку новых режимов работы процессора, поддержку многозадачности, обеспечивая стандартизацию интерфейсов аппаратного обеспечения и единообразие для пользовательских интерфейсов программ.

Развитием ОС Windows стало семейство Windows 9x (включает в себя Windows 95, Windows 98 и Windows Me). Отличительными особенностями являлись: новый пользовательский интерфейс, поддержка длинных имён файлов, автоматическое определение и конфигурация периферийных устройств Plug and Play, способность исполнять 32-разрядные приложения, наличие встроенной поддержки локальных сетей, использование вытесняющей многозадачности и выполнение каждого 32-разрядного приложения в своём изолированном адресном пространстве.

Следующим шагом в семействе Windows стала ОС Windows NT, которая дала начало семейству, в которое входят: собственно Windows NT, Windows 2000, Windows XP, Windows Server 2003, Windows Vista, Windows Server 2008, Windows 7, Windows 8.

2.2. Оболочка командной строки Windows

Во всех версиях операционной системы Windows поддерживается интерактивная оболочка командной строки (command shell) и по умолчанию устанавливается определенный набор утилит командной строки (количество и состав этих утилит зависит от версии операционной системы). Вообще, любую операционную систему можно представить в виде совокупности ядра системы, которое имеет доступ к аппаратуре и оперирует файлами и процессами, и оболочки (командного интерпретатора) с утилитами, которые позволяют пользователю получить доступ к функциональности

ядра операционной системы. Механизм работы оболочек в разных системах одинаков: в ответ на приглашение, выдаваемое находящейся в ожидании оболочкой, пользователь вводит некоторую команду (функциональность этой команды может быть реализована либо самой оболочкой, либо определенной внешней утилитой), оболочка выполняет ее, при необходимости выводя на экран какую-либо информацию, после чего снова выводит приглашение и ожидает ввода следующей команды.

Командный интерпретатор или *оболочка командной строки* – это программа, которая, находясь в оперативной памяти, считывает набираемые команды и обрабатывает их. В Windows 9x, как и в MS-DOS, командный интерпретатор по умолчанию был представлен исполняемым файлом command.com. Начиная с версии Windows NT, в операционной системе реализован интерпретатор команд Cmd.exe, обладающий гораздо более мощными возможностями.

Запуск оболочки

В Windows NT/2000/XP/7 файл Cmd.exe, как и другие исполняемые файлы, соответствующие внешним командам операционной системы, находятся в каталоге %SystemRoot%\SYSTEM32 (значением переменной среды %SystemRoot% является системный каталог Windows, обычно C:\Windows или C:\WinNT). 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. В результате откроется новое окно (см. рис. 2.1), в котором можно запускать команды и видеть результат их работы.

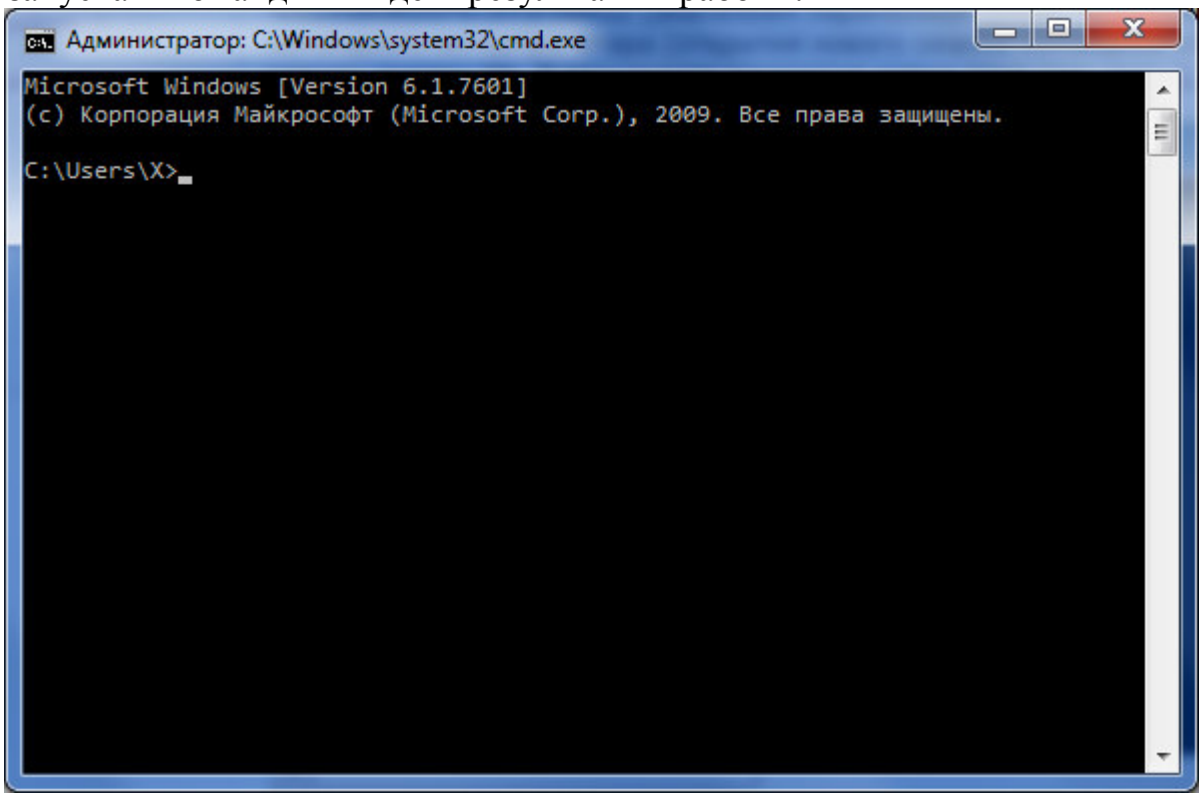

Рисунок 2.1 - Командное окно интерпретатора Cmd.exe в Windows 7

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 D:\ /V

Имя команды здесь – COPY, параметры – C:\myfile.txt и D:\, 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 /?. Например, если выполнить команду ATTRIB /?, то мы увидим следующее:

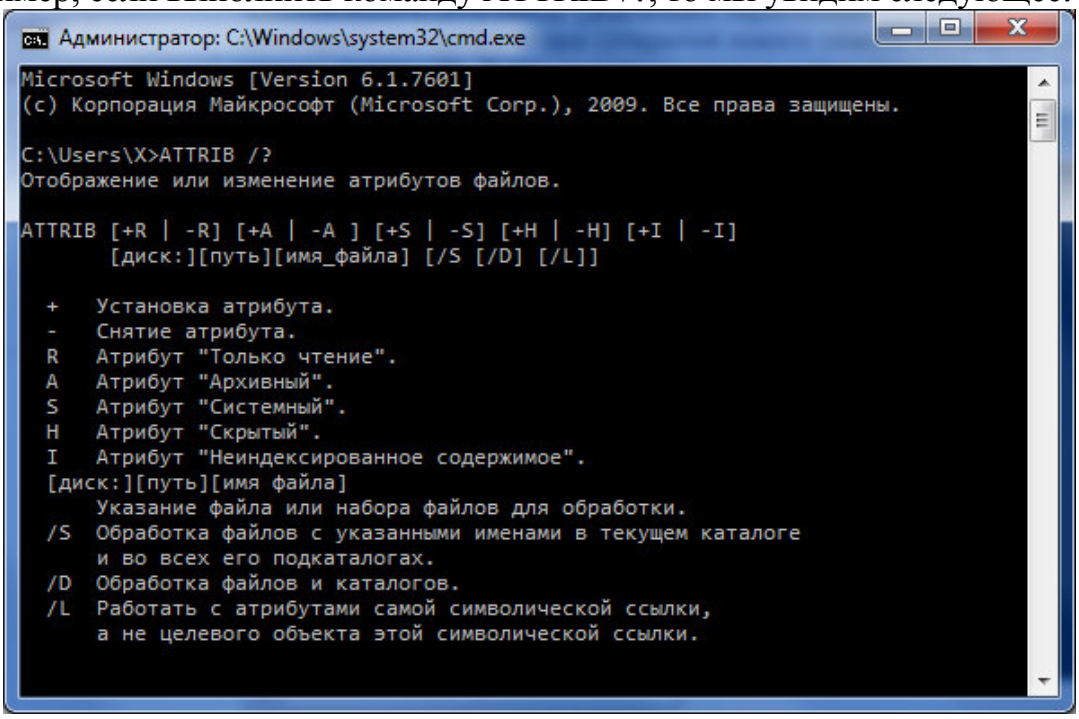

Рисунок 2.2 - Получение справки по команде

Для некоторых команд текст встроенной справки может быть довольно большим и не умещаться на одном экране. В этом случае помощь можно выводить последовательно по одному экрану с помощью команды MORE и символа конвейеризации |, например:

XCOPY /? | MORE

В этом случае после заполнения очередного экрана вывод помощи будет прерываться до нажатия любой клавиши. Кроме того, используя символы перенаправления вывода > и >>, можно текст, выводимый на экран, направить в текстовый файл для дальнейшего просмотра. Например, для вывода текста справки к команде XCOPY в текстовый файл xcopy.txt, используется следующая команда:

XCOPY /? > XCOPY.TXT

Вместо имени файла можно указывать обозначения устройств компьютера. В Windows поддерживаются следующие имена устройств: PRN (принтер), LPT1–LPT3 (соответствующие параллельные порты), AUX (устройство, присоединяемое к последовательному порту 1), COM1–COM3 (соответствующие последовательные порты), CON (терминал: при вводе это клавиатура, при выводе - монитор), NUL (пустое устройство, все операции ввода/вывода для него игнорируются).

Перенаправление ввода/вывода и конвейеризация (композиция) команд

Рассмотрим более подробно поддерживаемые в Windows UNIX-подобные концепции переназначения устройств стандартного ввода/вывода и конвейерного выполнения команд.

С помощью переназначения устройств ввода/вывода одна программа может направить свой вывод на вход другой или перехватить вывод другой программы, используя его в качестве своих входных данных. Таким образом, имеется возможность передавать информацию от процесса к процессу при минимальных программных издержках. Практически это означает, что для программ, которые используют стандартные входные и выходные устройства, операционная система позволяет:

- выводить сообщения программ не на экран (стандартный выходной поток), а в файл или на принтер (перенаправление вывода);
- читать входные данные не с клавиатуры (стандартный входной поток), а из заранее подготовленного файла (перенаправление ввода);
- передавать сообщения, выводимые одной программой, в качестве входных данных для другой программы (конвейеризация или композиция команд).

Из командной строки эти возможности реализуются следующим образом. Для того, чтобы перенаправить текстовые сообщения, выводимые какой-либо командой, в текстовый файл, нужно использовать конструкцию:

команда > имя_файла

Если при этом заданный для вывода файл уже существовал, то он перезаписывается (старое содержимое теряется), если не существовал – создается. Можно также не создавать файл заново, а дописывать информацию, выводимую командой, в конец существующего файла. Для этого команда перенаправления вывода должна быть задана так:

команда >> имя_файла

С помощью символа < можно прочитать входные данные для заданной команды не с клавиатуры, а из определенного (заранее подготовленного) файла:

команда < имя_файла

Приведем несколько примеров перенаправления ввода/вывода.

Вывод встроенной справки для команды COPY в файл copy.txt:

COPY /? > copy.txt

Добавление текста справки для команды XCOPY в файл copy.txt:

XCOPY /? >> copy.txt

Ввод новой даты из файла date.txt (DATE – это команда для просмотра и изменения системной даты):

DATE < date.txt

Если при выполнении определенной команды возникает ошибка, то сообщение об этом по умолчанию выводится на экран. В случае необходимости сообщения об ошибках (стандартный поток ошибок) можно перенаправить в текстовый файл с помощью конструкции:

команда 2> имя_файла

В этом случае стандартный вывод будет производиться на экран. Также имеется возможность информационные сообщения и сообщения об ошибках выводить в один и тот же файл. Делается это следующим образом:

команда > имя_файла 2>&1

Например, в приведенной ниже команде стандартный выходной поток и стандартный поток ошибок перенаправляются в файл copy.txt:

XCOPY A:\1.txt C: > copy.txt 2>&1

Наконец, с помощью конструкции

команда1 | команда2

можно использовать сообщения, выводимые первой командой, в качестве входных данных для второй команды (конвейер команд).

Используя механизмы перенаправления ввода/вывода и конвейеризации, можно из командной строки посылать информацию на различные устройства и автоматизировать ответы на запросы, выдаваемые командами или программами, использующими стандартный ввод. Для решения таких задач подходит команда:

ECHO [сообщение]

которая выводит сообщение на экран. Рассмотрим примеры использования этой команды.

Посылка символа прогона на принтер:

ECHO ^L > PRN

Удаление всех файлов в текущем каталоге без предупреждения (автоматический положительный ответ на запрос об удалении):

*ECHO y | DEL *.**

Команды MORE и SORT

Одной из наиболее часто использующихся команд, для работы с которой применяется перенаправление ввода/вывода и конвейеризация, является MORE. Эта

команда считывает стандартный ввод из конвейера или перенаправленного файла и выводит информацию частями, размер каждой из которых не больше размера экрана. Используется MORE обычно для просмотра длинных файлов. Возможны три варианта синтаксиса этой команды:

MORE [диск:][путь]имя_файла
MORE < [диск:][путь]имя_файла
имя_команды | MORE

Параметр [диск:][путь]имя_файла определяет расположение и имя файла с просматриваемыми на экране данными. Параметр имя_команды задает команду, вывод которой отображается на экране (например, DIR или команда TYPE, использующаяся для вывода содержимого текстового файла на экран). Приведем два примера.

Для поэкранного просмотра помощи команды DIR используется команда:

DIR /? | MORE

Для поэкранного просмотра текстового файла news.txt возможны следующие варианты команд:

MORE news.txt
MORE < news.txt
TYPE news.txt | MORE

Другой распространенной командой, использующей перенаправление ввода/вывода и конвейеризацию, является SORT. Эта команда работает как фильтр: она считывает символы в заданном столбце, упорядочивает их в возрастающем или убывающем порядке и выводит отсортированную информацию в файл, на экран или другое устройство. Возможны два варианта синтаксиса этой команды:

SORT [/R] [/+n] [[диск1:][путь1]файл1] [> [диск2:][путь2]файл2]
или
[команда] | SORT [/R] [/+n] [> [диск2:][путь2]файл2]

В первом случае параметр [диск1:][путь1]файл1 определяет имя файла, который нужно отсортировать. Во втором случае будут отсортированы выходные данные указанной команды. Если параметры файл1 или команда не заданы, то SORT будет считывать данные с устройства стандартного ввода.

Параметр [диск2:][путь2]файл2 задает файл, в который будет направляться сортированный вывод; если этот параметр не задан, то вывод будет направлен на устройство стандартного вывода.

По умолчанию сортировка выполняется в порядке возрастания. Ключ /R позволяет изменить порядок сортировки на обратный (от Z к A и затем от 9 до 0). Например, для поэкранного просмотра отсортированного в обратном порядке файла price.txt, нужно задать следующую команду:

SORT /R < price.txt |MORE

Ключ /+n задает сортировку в файле по символам n-го столбца. Например, /+10 означает, что сортировка должна осуществляться, начиная с 10-й позиции в каждой строке. По умолчанию файл сортируется по первому столбцу.

Условное выполнение и группировка команд

В командной строке Windows NT/2000/XP/7 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 амперсанда &, 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 амперсанд && запускает команду, стоящую за ним в командной строке, только в том случае, если команда, стоящая перед амперсандами 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.

Пути к объектам файловой системы

Напомним, что файловая система логически имеет древовидную структуру и имена файлов задаются в формате [диск:][путь\]имя_файла, то есть обязательным параметром является только имя файла. При этом, если путь начинается с символа "\", то маршрут вычисляется от корневого каталога, иначе – от текущего каталога. Например, имя C:123.txt задает файл 123.txt в текущем каталоге на диске C:, имя C:\123.txt – файл 123.txt в корневом каталоге на диске C:, имя ABC\123.txt – файл 123.txt в подкаталоге ABC текущего каталога.

Существуют особые обозначения для текущего каталога и родительского каталогов. Текущий каталог обозначается символом . (точка), его родительский каталог – символами .. (две точки). Например, если текущим каталогом является C:\WINDOWS, то 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 text911.txt, под шаблон text*.txt – имена text.txt, textab12.txt, а под шаблон text.* – все файлы с именем text и произвольным расширением.

Для того, чтобы использовать длинные имена файлов при работе с командной строкой, их нужно заключать в двойные кавычки. Например, чтобы запустить файл с именем 'Мое приложение.exe' из каталога 'Мои документы', нужно в командной строке набрать "C:\Мои документы\Мое приложение.exe" и нажать клавишу <Enter>.

Перейдем теперь непосредственно к командам для работы с файловой системой.

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. 2.1.

Таблица 2.1. 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) несколько 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 файлы 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.

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.

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.

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.

2.3. 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. Например, если записать в файл deltmp.bat следующие команды:

```
C:\
CD %TEMP%
DEL /F *.tmp
```

и запустить его на выполнение (аналогично исполняемым файлам с расширением com или exe), то мы удалим все файлы во временной директории Windows. Таким образом, исполнение командного файла приводит к тому же результату, что и последовательный ввод записанных в нем команд. При этом не проводится никакой предварительной компиляции или проверки синтаксиса кода; если встречается строка с ошибочной командой, то она игнорируется. Очевидно, что если вам приходится часто выполнять одни и те же действия, то использование командных файлов может сэкономить много времени.

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. Например,

@ECHO OFF
ECHO Привет!
ECHO.
ECHO Пока!

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

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. 2.2

Таблица 2.2. 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. 3.1, 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
```

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 опе-

рационной 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
```

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 файла 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

Во-вторых, из переменной среды можно выделять подстроки с помощью конструкции %имя_переменной:~n1,n2%, где число n1 определяет смещение (количество пропускаемых символов) от начала (если n1 положительно) или от конца (если n1 отрицательно) соответствующей переменной среды, а число n2 – количество выделяемых символов (если n2 положительно) или количество последних символов в переменной, которые не войдут в выделяемую подстроку (если n2 отрицательно). Если указан только один отрицательный параметр -n, то будут извлечены последние n символов. Например, если в переменной хранится строка "21.09.2012" (символьное представление текущая дата при определенных региональных настройках), то после выполнения следующих команд

SET dd1=%DATE:~0,2%
SET dd2=%DATE:~0,-8%
SET mm=%DATE:~-7,2%
SET yyyy=%DATE:~-4%

новые переменные будут иметь такие значения: %dd1%=21, %dd2%=21, %mm%=09, %yyyy%=2012.

В-третьих, можно выполнять процедуру замены подстрок с помощью конструкции %имя_переменной:s1=s2% (в результате будет возвращена строка, в которой каждое вхождение подстроки s1 в соответствующей переменной среды заменено на s2). Например, после выполнения команд

SET a=123456
SET b=%a:23=99%

в переменной b будет храниться строка "199456". Если параметр s2 не указан, то подстрока s1 будет удалена из выводимой строки, т.е. после выполнения команды

SET a=123456
SET b=%a:23=%

в переменной b будет храниться строка "1456".

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=
```

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.

Связывание времени выполнения для переменных

При работе с составными выражениями (группы команд, заключенных в круглые скобки) нужно учитывать, что переменные среды в командных файлах используются в режиме раннего связывания. С точки зрения логики выполнения командного файла это может привести к ошибкам. Например, рассмотрим командный файл 1.bat со следующим содержимым:

```
SET a=1
ECHO a=%a%
SET a=2
ECHO a=%a%
```

и командный файл 2.bat:

```
SET a=1
ECHO a=%a%
(SET a=2
ECHO a=%a% )
```

Казалось бы, результат выполнения этих двух файлов должен быть одинаковым: на экран выведутся две строки: "a=1" и "a=2". На самом же деле таким образом сработает только файл 1.bat, а файл 2.bat два раза выведет строку "a=1"!

Данную ошибку можно обойти, если для получения значения переменной вместо знаков процента (%) использовать восклицательный знак (!) и предвари-

тельно включить режим связывания времени выполнения командой SETLOCAL ENABLEDELAYEDEXPANSION. Таким образом, для корректной работы файл 2.bat должен иметь следующий вид:

```
SETLOCAL ENABLEDELAYEDEXPANSION
SET a=1
ECHO a=%a%
(SET a=2
ECHO a=!a! )
```

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
```

Команду PAUSE обязательно нужно использовать при выполнении потенциально опасных действий (удаление файлов, форматирование дисков и т.п.). Например,

```
ECHO Сейчас будут удалены все файлы в C:\Мои документы!
ECHO Для отмены нажмите Ctrl-C
PAUSE
DEL "C:\Мои документы\*.*"
```

Вызов внешних командных файлов

Из одного командного файла можно вызвать другой, просто указав его имя. Например:

```
@ECHO OFF
CLS
REM Вывод списка log-файлов
DIR C:\*.log
REM Передача выполнения файлу f.bat
f.bat
COPY A:\*.* C:\
PAUSE
```

Однако в этом случае после выполнения вызванного файла управление в вызывающий файл не передается, то есть в приведенном примере команда

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

(и все следующие за ней команды) никогда не будет выполнена.

Для того, чтобы вызвать внешний командный файл с последующим возвратом в первоначальный файл, нужно использовать специальную команду CALL файл

Например:

@ECHO OFF
CLS
REM Вывод списка log-файлов
DIR C:.log*
REM Передача выполнения файлу f.bat
CALL f.bat
COPY A:.* C:*
PAUSE

В этом случае после завершения работы файла f.bat управление вернется в первоначальный файл на строку, следующую за командой CALL (в нашем примере это команда COPY A:*.* C:\ ).

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 аргумента

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.

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. 2.3

Таблица 2.3. Операторы сравнения в IF

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

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. 2.3 Например:

IF ERRORLEVEL LEQ 1 GOTO Case1

Иногда более удобным для работы с кодами завершения программ может оказаться использование переменной %ERRORLEVEL%. (строковое представление текущего значения кода ошибки ERRORLEVEL).

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]

Перед названием переменной должны стоять именно два знака процента (%%), а не один, как это было при использовании команды FOR непосредственно из командной строки. 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.

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.

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. 2.4 ключевые слова:

Таблица 2.4. 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 БББББ ВВВВ,ГГГГГ ДДДД
ЕЕЕЕЕ,ЖЖЖЖ 3333
;ККККК ЛЛЛЛЛ ММММММ

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

БББББ ВВВВ ГГГГГ ДДДД
ЖЖЖЖ 3333

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 БББББ ВВВВ,ГГГГГ ДДДД") DO @ECHO %%i %%j %%k
```

при своем выполнении напечатает

БББББ ВВВВ ГГГГГ ДДДД

Вместо явного задания строки для разбора можно пользоваться переменными среды, например:

```
@ECHO OFF
SET M=ААА БББББ 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. 2.5).

Таблица 2.5. 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

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

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".

3. Варианты заданий и порядок выполнения работы

1. Запустить командный интерпретатор Microsoft Windows - cmd.exe(см п. 2.2)
2. Изучить внутренние и внешние команды ОС Windows(см п. 2.2). Для просмотра подробного описание команд, набрать в командной строке «<команда> /?»;
3. Создать с помощью команд ОС Windows дерево каталогов и набор файлов согласно описанным ниже правилам (создание файла следует производить с помощью команды «*COPY CON <имя_файла>*», при этом ввод с консоли (клавиатуры) будет направляться в файл до прихода символа конца файла ^Z, который может быть введен нажатием комбинации Ctrl+Z, либо нажатием F6).
   3.1. Создайте каталог с именем, образованным вашей фамилией (с усечением до восьми символов) и расширением, соответствующим номеру учебной группы, например: SIDORENK.I32.
   3.2. В данном каталоге создайте три подкаталога с именами, образованными из Ваших инициалов и порядкового номера каталога, например: SVYu1,SVYu2,SVYu3.
   3.3. В первом подкаталоге создать файл, имя которого совпадает с Вашим и расширением «.txt», например VLAD.TXT. В файл поместите Ваши фамилию, имя, отчество и номер группы;
4. Произведите ряд операций над файлами и каталогами с использованием команд ОС Windows:
   4.1. Скопировать созданный файл во второй каталог.
   4.2. Переименовать файл во втором каталоге, переставив буквы имени в обратном порядке (для примера, указанного выше, - DALV.TXT).

4.3. Объединить файлы из первых двух подкаталогов и результат поместить в третий файл с расширением.doc и с одним из выбранных имен.

4.4. Переместить результирующий файл из третьего каталога в каталог верхнего уровня.

4.5. Вывести содержимое этого файла на экран дисплея.

4.6. Продемонстрировать преподавателю результаты работы.

4.7. Уничтожить файлы и каталоги.

5. Написать сценарии оболочки ОС Windows, реализующие описанные в предыдущем пункте действия со следующими особенностями.

5.1. Перед копированием файлов следует производить проверку их существования и в случае отсутствия файлов выводить на дисплей предупреждение и аварийно завершать сценарий.

5.2. Необходимо предусмотреть ввод имени третьего файла в качестве параметра командной строки, а при отсутствии параметра задать пользователю вопрос о том, какое имя (первого или второго файла) следует использовать для создания третьего файла. Расширение этого файла должно быть задано (до запуска сценария) переменной окружения EXT.

5.3. Продублировать все выводимые на дисплей сообщения в файл протокола с именем, соответствующим имени файла сценария и расширением «.log».

4 Содержание отчета

Отчет должен содержать следующие пункты:

- постановка задачи (вариант задания);
- текст сценария с комментариями;
- выводы.

5 Контрольные вопросы

1) Расскажите о семействе ОС Windows?
2) Что такое абсолютный и относительный путь?
3) Что такое переменные окружения?
4) Что такое сценарии оболочки?
5) Как производится ввод-вывод из сценария оболочки Windows?
6) Как организуются ветвление и циклы в сценариях оболочки Windows?
7) Какие способы перенаправления потоков информации поддерживаются ОС Windows?

ЛАБОРАТОРНАЯ РАБОТА №2
Использование программного интерфейса Win API. Процессы и потоки в ОС Windows.

1. ЦЕЛЬ РАБОТЫ

Изучение возможности использования программного интерфейса приложений(API) операционных систем Windows 95-2000, NT, XP, 7. Приобретение практических навыков создания и управления процессами и потоками, используя Win API в средах программирования Borland Delphi, C++ Builder или Visual Studio.

2. ОСНОВНЫЕ ПОЛОЖЕНИЯ

2.1. Программный интерфейс приложений Win API

Программный интерфейс приложений Win API позволяет пользовательским приложениям использовать всю мощь операционных систем семейства Windows. Функции, структуры, сообщения, макросы, и интерфейсы, предоставляемые Win API, формируют непротиворечивый и однородный API для всех платформ Microsoft: Windows 95, 98, 2000, NT, XP, 7. Используя различные функции, предоставляемые Win API, возможно разрабатывать приложения, которые будут успешно функционировать на всех перечисленных выше платформах, а так же будут иметь возможность использовать уникальные особенности каждой из этих платформ.

Функции Win API могут быть разделены на следующие категории:

- Управление окнами (Window Management);
- Оконные объекты управления (Window Controls);
- Возможности оболочки (Shell Features);
- Интерфейс Графических Устройств (Graphics Device Interface);
- Системные Услуги (System Services);
- Сетевые Услуги (Network Services).

В данной лабораторной работе будут использоваться функции, относящиеся к категории System Services. Функции данной группы дают приложениям возможность, обращаться к ресурсам компьютера и использовать возможности операционной системы по работе с памятью, файловой системой, внешними устройствами, а также по управлению процессами и потоками. Например, приложение может использовать функции управления памятью, чтобы распределить свободную память и функции управление процессами и синхронизации, чтобы запустить и координировать выполнение других приложений или параллельных потоков в пределах единственного приложения.

2.2. Понятие процесса и потока в Windows

При запуске программы в Windows, операционная система создает новый *процесс.* В ОС Windows процесс владеет оперативной памятью, открытыми файла-

ми и другими ресурсами, Однако в Windows исполняется не процесс, а программный *поток*. В момент создания процесса ОС так же создает и единственный программный поток процесса. Поток, который является последовательностью машинных команд, обладает так же указателем на точку исполнения (команда, выполняемая в текущий момент), указателем на стек (где хранятся данные, принадлежащие потоку). Кроме того, важным для возможности приостановки и последующего возобновления потока (например, при переключении между потоками) является состояние регистров микропроцессора.

Какая разница между процессом и потоком? С точки зрения пользователя компьютера, особенно в том случае, если каждый процесс обладает только одним потоком, то разницы фактически никакой. С точки зрения ОС, как было отмечено выше, процесс – это единица владения ресурсами, а поток – единица диспетчеризации и распределения процессорного времени. Кроме того, в ходе выполнения потока могут быть созданы другие потоки, которые в свою очередь, могут создавать другие потоки и т.д. Два процесса не могут обладать общими ресурсами, если при этом они не используют специальных механизмов операционной системы. Напротив, все потоки, принадлежащие одному процессу, имеют доступ ко всем ресурсам этого процесса, включая память, открытые файлы и другие ресурсы.

При программировании в Windows, чтобы запустить какую-либо существующую программу, используется системный вызов CreateProcess.

В ОС Windows (в отличии от UNIX) новый процесс имеет мало общего с породившим его процессом. В частности, новый процесс не сможет получить доступа к данным родительского процесса, не воспользовавшись для этого специальными механизмами операционной системы. Однако, в рамках текущего процесса Windows можно создать несколько программных потоков, выполняющихся одновременно (параллельно) друг с другом, и все они будут иметь доступ к единственному экземпляру данных процесса. Таким образом, если один из потоков модифицирует эти данные, то они изменятся и для всех остальных потоков.

2.2.1. Процессы в Windows

Как было отмечено выше, если нужно запустить какую-либо существующую программу в ОС Windows используется стандартный системный вызов CreateProcess. Краткое описание аргументов функции CreateProcess приведено в таблицах 2.1 и 2.2 (более полное описание может быть получено в справочном руководстве Win Programmer's Reference):

Таблица 2.1 - Аргументы вызова CreateProcess

Аргумент	Описание
lpApplicationName	Имя программы (или NULL, если имя программы указано в командной строке)
lpCommandLine	Командная строка
lpProcessAttributes	Атрибуты безопасности для дескриптора процесса, возвращаемого функцией

lpThreadAttributes	Атрибуты безопасности для дескриптора потока, возвращаемого функцией
bInheritHandlers	Указывает, наследует ли новый процесс дескрипторы, принадлежащие текущему процессу
dwCreationFlags	Параметры создания процесса (см. табл. 2.2)
lpEnvironment	Значения переменных окружения (или NULL, *в* случае, если наследуется текущее окружение)
lpCurrentDirectory	Текущий каталог (или NULL, если используется текущий каталог текущего процесса)
lpStartupInfo	Структура STARTUPINFO, содержащая информацию о запуске процесса
lpProcessInformation	Возвращаемые функцией дескрипторы и идентификаторы ID процесса и потока

Таблица 2.2 – Параметры dwCreationFlags, используемые при создании процесса

Флаг	Значение
CREATE_DEFAULT_ERROR_MODE	Не наследовать текущий режим сообщений об ошибках
CREATE_NEW_CONSOLE	Создать новую консоль
CREATE_NEW_PROCESS_GROUP	Создать новую группу процессов
CREATE_SEPARATE_WOW-VDM	Запустить 16-битное приложение в его собственном адресном пространстве
CREATE_SHARED_WOW_VDM	Запустить 16-битное приложение в адресном пространстве общего доступа
CREATE_SUSPENDED	Создать процесс в приостановленном состоянии
CREATE_UNICODE_ENVIRONMENT	Блок переменных окружения записан в формате UNICODE
DEBUG_PROCESS	Запустить процесс в отладочном режиме
DEBUG_ONLY_THIS_PROCESS	Предотвратить отладку процесса текущим отладчиком
DETACHED PROCESS	Новый консольный процесс не имеет доступа к консоли родительского консольного процесса

Аргумент lpStartupInfo – это указатель на структуру STARTUPINFO (см. Win Programmer's Reference). Поля этой структуры содержат заголовок консоли, начальный размер и позицию нового окна и перенаправление стандартных потоков ввода/вывода. Поле dwFlags этой структуры содержит флаги, установленные в соответствии с тем, какие из остальных полей структуры вы хотели бы использовать при запуске новой программы. Например, если вы сбросите флаг STARTF_USEPOSITION, поля dwX и dwY структуры STARTUPINFO, содержащие координаты основного окна запускаемой программы, будут проигнорированы. Часто нет надобности использовать какие-либо из этих полей. В этом случае необходимо передать функции CreateProcess корректный указатель на пустую структуру STARTUPINFO.

Параметр dwCreationFlags(см. таблицу 2.2) используется для установки свойств нового процесса (например, для создания потока с новой консолью используется CREATE_NEW_CONSOLE), а также управляет классом приоритета нового процесса. В ОС Windows определены следующие классы приоритетов процессов:

- HIGH_PRIORITY_CLASS - указывает процесс, который исполняет критические по времени задачи, которые должны быть выполнены практически немед-

ленно. Потоки процесса такого приоритета выгружают потоки нормально-приоритетных или низко-приоритетных процессов. Пример - Windows Task List (Список задач), который должен появиться быстро, независимо от загрузки операционной системы. Необходимо проявлять осторожность при использовании приоритета HIGH_PRIORITY_CLASS, потому что такие процессы ограниченные возможностями процессора могут использовать почти все доступное процессорное время, что значительно замедлит выполнение остальных задач.

- IDLE_PRIORITY_CLASS - указывает процесс, потоки которого работают только, когда система простаивает, поэтому выгружается потоками любого процесса, выполняющегося в более высоком приоритете. Пример - экранная заставка.
- NORMAL_PRIORITY_CLASS - указывает нормальный процесс без специальных потребностей планирования.
- REALTIME_PRIORITY_CLASS - указывает процесс, который имеет самый высокий возможный приоритет. Потоки процесса такого класса приоритета (потоки реального времени) выгружают потоки всех других процессов, включая процессы операционной системы, выполняющие важные задачи. Такой класс приоритета используется ТОЛЬКО для создания приложений работающих с аппаратурой, в случае если время реакции является критичным.

Если ни один из вышеперечисленных флагов приоритета не определен, значение по умолчанию - NORMAL_PRIORITY_CLASS (исключая тот случай, когда приоритет процесса-создателя - IDLE_PRIORITY_CLASS, в этом случае заданный по умолчанию класс приоритета дочернего процесса тоже IDLE_PRIORITY_CLASS).

Функция CreateProcess возвращает значение типа BOOL, но по завершении работы чрезвычайно полезная для программиста информация размещается этой функцией в структуре типа PROCESS_INFORMATION, указатель на которую возвращается при помощи параметра lpProcessInformation. В структуре PROCESS_INFORMATION содержится идентификатор и дескриптор нового процесса, а также идентификатор и дескриптор самого первого потока, принадлежащего новому процессу. Эти сведения могут использоваться для того, чтобы сообщить о новом процессе другим программам, а также для того, чтобы контролировать новый процесс.

Обладая дескриптором процесса, возможно управлять процессом при помощи системных вызовов, перечисленных в таблице 2.3. Чтобы определить момент завершения процесса, можно воспользоваться одним из нескольких методов. Во-первых, можно использовать вызов GetExitCodeProcess. Этот вызов возвращает либо значение STILL_ACTIVE (если процесс все еще продолжает работу), либо код завершения процесса (если процесс завершен). В качестве одного из аргументов необходимо передать этой функции указатель на переменную, в которую помещается возвращаемое значение. Узнать дескриптор текущего процесса можно при помощи функции GetCurrentProcess.

Чтобы управлять процессом из другого процесса, вы должны обратиться к функции OpenProcess, которой необходимо передать идентификатор ID процесса. Процесс, создающий новый процесс при помощи функции CreateProcess, уже обладает дескриптором нового процесса (этот дескриптор помещается в структуру PROCESS_INFORMATION).

Таблица 2.3. - Системные вызовы управления процессом

Вызов	Назначение	Требуемый уровень доступа
CreateRemoteThread	Создает поток в другом процесса	PROCESS_CREATE_ THREAD
GetExitCodeProcess	Возвращает код завершения процесса	PROCESS_QUERY_ INFORMATION
GetGuiRecources	Определяет, сколько объектов USER или GDI (Graphical Device Interface) используется процессом	PROCESS_QUERY_ INFORMATION
SetPriorityClass	Устанавливает базовый приоритет процесса	PROCESS_SET_ INFORMATION
GetPriorityClass	Возвращает базовый приоритет процесса	PROCESS_QUERY_ INFORMATION
SetProcessAffinityMask	Определяет, какие из процессоров используются процессом в качестве основных	PROCESS_SET_ INFORMATION
GetProcessAffinityMask	Устанавливает, какие из процессоров используются процессом в качестве основных	PROCESS_QUERY_ INFORMATION
SetProcessPriority Boost	Позволяет или запрещает Windows динамически изменять приоритет процесса	PROCESS_SET_ INFORMATION
GetProcessPriorityBoost	Возвращает статус изменения приоритета процесса	PROCESS_GET_ INFORMATION
SetProcessShutDownParameters	Определяет, в каком порядке система закрывает процессы при завершении работы всей системы	N/A (необходимо вызвать изнутри процесса)
GetProcessShutDownParameters	Возвращает статус механизма завершения работы системы	PROCESS_QUERY_ INFORMATION
SetProcessWorkingSetSize	Устанавливает минимальный и максимальный допустимый объем физической оперативной памяти, используемый процессом	PROCESS_SET_ INFORMATION
GetProcessWorkingSetSize	Возвращает информацию об использовании физической памяти процессом	PROCESS_GET_ INFORMATION
TerminateProcess	Корректное завершение работы процесса	PROCESS_TERMINATE
ExitProcess	Немедленное завершение процесса	N/A (необходимо вызвать изнутри процесса)
GetProcessVersion	Возвращает версию Windows, в среде которой хотел бы работать процесс	N/A (использует ID процесса)
GetProcessTimes	Возвращает степень использования CPU процессом	PROCESS_GET_ INFORMATION
GetStartUpInfo	Возвращает структуру STARTUPINFO, переданную процессу при обращении к CreateProcess	N/A (необходимо вызвать изнутри процесса)

Еще один способ определения текущего состояния процесса подразумевает использование системной функции WaitForSingleObject. Этот вызов можно использовать для ожидания различных системных объектов. Основное назначение WaitForSingleObject – определить, находится ли дескриптор определенного системного объекта в сигнальном состоянии. Дескриптор процесса переходит в сигнальное состояние тогда, когда процесс завершает свою работу. При обращении к функции WaitForSingleObject указывается дескриптор процесса и интервал времени в миллисекундах. Если заданный интервал времени равен 0, функция завершает работу немедленно, возвращая текущее состояние процесса. Если интервал времени равен константе INFINITE, функция будет ждать до тех пор, пока интересующий вас процесс не завершит работу. Если указано конкретное цифровое значение интервала времени, функция будет ожидать завершения процесса в течение этого интервала, а затем вернет управление вызвавшей ее программе. Если в течение указанного времени процесс завершит работу, функция WaitForSingleObject вернет управление вызвавшей программе с результатом WAIT_OBJECT_0 (дескриптор целевого процесса перешел в сигнальное состояние). В противном случае эта функция вернет WAIT_TIMEOUT.

В случае ошибки функция WaitForSingleObject вернет значение WAIT_FAILED.

Следует отметить, что идентификатор ID процесса – это не то же самое, что дескриптор процесса. Уникальный идентификатор ID процесса служит для идентификации процесса в списке других процессов ОС. Но для управления процессом из другого процесса, прежде всего следует, каким-либо образом получить дескриптор управляемого процесса. Преобразовать идентификатор ID процесса в его дескриптор можно при помощи функции OpenProcess, однако для этого необходимо обладать требуемыми привилегиями. Узнать идентификатор текущего процесса можно при помощи функции GetCurrentProcessId. Используя этот вызов, возможно узнать идентификатор собственного процесса и, например, передать этот идентификатор другому процессу. Получив ID процесса, другой процесс сможет открыть его дескриптор для управления.

При обращении к функции OpenProcess необходимо указать требуемый уровень доступа к открываемому процессу. Например, чтобы узнать код завершения процесса, достаточно владеть уровнем доступа PROCESS_QUERY_INFORMATION. Чтобы иметь возможность завершить работу процесса, необходимо обладать уровнем доступа PROCESS_TERMINATE. Уровни доступа к процессу перечислены в таблице 2.3. Для предоставления полного набора прав доступа к процессу предназначен уровень доступа PROCESS_ALL_ACCESS.

Ниже приведены два листинга простых консольных приложений на языке Си. Первая программа (MASTER) запускает вторую (SLAVE) и переходит в режим ожидания. Программа SLAVE читает идентификатор процесса (PID, Process Identifier) запустившей ее программы из командной строки и ожидает завершения работы программы MASTER. В командной строке программы MASTER нужно указать полный путь к исполняемому файлу программы SLAVE:

//Программа master

```
#include <windows.h>
#include <iostream.h>
#include <stdio.h>
#include <string.h>
void main(int argc, char *argv[]){
   char cmd[128];
      if (argc!=1) strcpy(cmd,argv[1]);
      else strcpy(cmd,"slave.exe");
      int pid=GetCurrentProcessId();
      sprintf(cmd+strlen(cmd)," %d",pid);
      cout<<"Master: Starting:" << cmd << "\n";
      cout.flush();
      STARTUPINFO info;
      memset(&info,0,sizeof(info));
      info.cb=sizeof(info);
      PROCESS_INFORMATION pinfo;
      if (!CreateProcess(NULL,cmd,NULL,NULL,FALSE,NORMAL_PRIORITY_CLASS,
            NULL,NULL,&info,&pinfo))  {
            cout<<"Master: Slave процесс не запущен\n";
            cout<<"Master: проверьте правильность указания имени процесса в коммандной
строке"; }
      cout<<"Master: Sleeping\n";
      Sleep(15000);
      cout<<"Master: Exiting\n";
      exit(0);
}
```

//Программа Slave

```
#include <windows.h>
#include <iostream.h>
#include <stdio.h> // temp
void main(int argc,char *argv[]){
      if (argc!=2){
            cerr<<"Slave: Необходимо сначала запустить MASTER.EXE.\n";
            exit(1);
      }
      int pid=atoi(argv[1]);
      HANDLE process=OpenProcess(PROCESS_QUERY_INFORMATION | SYNCHRONIZE,
FALSE, pid);
      if (!process) cout<<"Slave:Ошибка открытия процесса\n";
      cout<<"Slave:Ожидание завершения процесса Master\n";
      cout.flush();
      if (WaitForSingleObject(process,INFINITE)==STATUS_WAIT_0)
            cout<<"Slave: Master завершил работу\n";
      else
            cout<<"Slave: Не известная ошибка\n";
      exit(0);
}
```

Обе программы иллюстрируют использование функций CreateProcess, OpenProcess и WaitForSingleObject.

2.2.2. Потоки в Windows

Как отмечалось выше, при создании процесса операционная система Windows создает и новый программный поток, принадлежащий этому процессу. Вначале любой только что созданный процесс обладает лишь одним потоком. Этот поток может создавать новые потоки, а эти новые потоки, в свою очередь, могут создавать другие новые потоки. Процесс продолжает свое существование до тех пор, пока в его владении находится, по крайней мере, один программный поток (или до тех пор, пока не произойдет событие, в результате которого весь процесс прекратит работу, например, обращение к функции TerminateProcess).

Зачем процессу несколько потоков? Потоки могут выполнять какие-либо действия в фоновом режиме относительно вашей основной программы. Например, можно создать новый программный поток, который будет в фоновом режиме осуществлять вывод информации на принтер. Потоки удобно использовать также в случае, если блокирование или зависание какой-либо процедуры не должно стать причиной нарушений функционирования основной программы. Например, в то время как основная программа выполняет сложные математические вычисления, отдельный программный поток может осуществлять обмен данными через асинхронный последовательный канал связи (например, через модем). В случае замедления передачи данных через канал или в случае зависания модема функционирование основной программы не будет нарушено.

Базовый системный вызов, предназначенный для создания потока, это CreateThread, принимающий аргументы указанные в табл. 2.4:

Таблица 2.4 - Аргументы вызова CreateThread

Аргумент	Описание
lpThreadAttributes	Указатель на структуру SECURITY_ATTRIBUTES. Используется только в Windows NT.
dwStackSize	Определяет размер стека выделяемого новому потоку. Если значение равно 0, размер стека равен размеру стека создающего потока.
lpStartAddress	Стартовый 32-разрядный адрес нового потока(адрес функции потока)
lpParameter	Определяет 32 разрядный параметр передаваемый потоку
dwCreationFlags	Параметры создания потока. Если установлен флаг CREATE_SUSPENDED, то создаваемый поток после запуска будет находиться в приостановленном состоянии до вызова функции ResumeThread. Если dwCreationFlags равен нулю, то поток запускается сразу после создания.
lpThreadId	Указатель на 32-разрядную переменную, которая будет содержать идентификатор потока.

Вновь создаваемый поток имеет приоритет THREAD_PRIORITY_NORMAL. Используя функции GetThreadPriority и SetThreadPriority можно получить текущий и установить требуемый приоритет созданного потока. Вместе с классом приоритета процесса приоритет потока определяет базовый уровень приоритета потока (он изменяется от 0 до 31).

Для потоков определены следующие приоритеты:

- THREAD_PRIORITY_ABOVE_NORMAL – устанавливает приоритет на один уровень выше нормального для соответствующего класса приоритета процесса.
- THREAD_PRIORITY_BELOW_NORMAL – устанавливает приоритет на один уровень ниже нормального для соответствующего класса приоритета процесса.
- THREAD_PRIORITY_HIGHEST – устанавливает приоритет на два уровня выше нормального для соответствующего класса приоритета процесса.
- THREAD_PRIORITY_IDLE – устанавливает базовый уровень приоритета потока равный 1 для классов приоритета процесса: IDLE_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, или HIGH_PRIORITY_CLASS, и базовый уровень приоритета потока 16 для класса приоритета процесса REALTIME_PRIORITY_CLASS.
- THREAD_PRIORITY_LOWEST - устанавливает приоритет на два уровня ниже нормального для соответствующего класса приоритета процесса.
- THREAD_PRIORITY_NORMAL устанавливает нормальный приоритет для соответствующего класса приоритета процесса.
- THREAD_PRIORITY_TIME_CRITICAL – устанавливает базовый уровень приоритета потока 15 для классов приоритета процесса: IDLE_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, или HIGH_PRIORITY_CLASS, и базовый уровень приоритета потока 31 для класса приоритета процесса REALTIME_PRIORITY_CLASS.

ОС Windows распределяет процессорное время не между процессами, а между потоками. Каждый поток в системе работает независимо. Количество процессорного времени, выделяемое конкретному потоку, определяется многими факторами. Во-первых, каждый процесс обладает собственным базовым уровнем приоритета, который присваивается всем принадлежащим ему потокам. Во-вторых, каждый из потоков обладает собственным приоритетом, который добавляется к базовому значению. Наконец, сама операционная система имеет право динамически изменить приоритет некоторого потока. Например, если система обнаружит, что устройство ввода готово передать одному из потоков какие-либо данные, и ожидает, пока управление будет передано этому потоку, система может временно повысить приоритет этого потока.

Для управления потоками служат системные вызовы, перечисленные в таблице 2.5.

Таблица 2.5 - Системные вызовы для управления потоками

Функция	**Требуемый уровень доступа**	**Описание**
AttachThreadInput	Нет	Связывает обработку ввода потока с другим потоком. Позволяет передавать фокус ввода окнам другого потока, а также использовать общее состояние ввода
CreateRemoteThread	PROCESS_ CREATE_THREAD	Создает поток в другом процессе
CreateThread	Нет	Создает поток в текущем процессе
ExitThread	Нет	Завершает работу текущего потока
TerminateThread	THREAD_	Останавливает работу потока без

Функция	Требуемый уровень доступа	Описание
	TERMINATE	выполнения необходимых подготовительных процедур
GetCurrentThread	Нет	Возвращает дескриптор текущего потока
GetCurrentThreadld	Нет	Возвращает идентификатор IDтекущего потока
GetExitCodeThread	THREAD_QUERY_ INFORMATION	Возвращает код завершения потока;
SetThreadPriority	THREAD_SET_ INFORMATION	Устанавливает уровень приоритета потока
GetThreadPriority	THREAD_QUERY_ INFORMATION	Возвращает уровень приоритета потока
SetThreadPriorityBoost	THREAD_SET_ INFORMATION	Разрешает или запрещает динамическое изменение уровня приоритета данного потока
GetThreadPriorityBoost	THREAD_QUERY_ INFORMATION	Возвращает статус динамического изменения приоритета для данного процесса
GetThreadTimes	THREAD_QUERY_ INFORMATION	Возвращает время, когда был создан или уничтожен поток и какое количество процессорного времени было им использовано
SuspendThread	THREAD_ SUSPEND_RESUME	Временно приостанавливает выполнение потока
ResumeThread	THREAD_ SUSPEND_RESUME	Продолжает работу потока, приостановленного в результате обращения к SuspendThread
SetThreadAfflnityMask	THREAD_SET_ INFORMATION	Устанавливает, какие процессоры могут использоваться для выполнения данного потока
SetThreadIdealProcessor	THREAD_SET_ INFORMATION	Устанавливает, какой процессор предпочтительней использовать для выполнения потока(в многопроцессорных системах)
SwitchToThread	Нет	Переключает процессор на выполнение другого потока

Пример: программа на языке Pascal, представленная ниже, выводит на экран простое диалоговое окно с текстовым сообщением (используя системный вызов MessageBox) и при этом каждую секунду издает короткий звуковой сигнал (для этого используется системная функция MessageBeep). Для воспроизведения этого сигнала используется отдельный программный поток.

```
program ALERT;
{$APPTYPE CONSOLE}
uses  SysUtils, Windows;
var mainthread:UINT;//переменная для хранения дескриптора нового потока
id:DWORD;//переменная для хранения идентификатора потока
procedure beepthread;//функция потока
var   exitcode:DWORD;
```

```
begin
 while((GetExitCodeThread(mainthread,exitcode))AND (exitcode=STILL_ACTIVE)) do
     begin
      MessageBeep($0FFFFFFFF);//звуковой сигнал
      Sleep(1000);//задержка
     end;
end;
Begin
  mainthread:=GetCurrentThread;//определяем дескриптор текущего потока
  CreateThread(nil,0,addr(beepthread),nil,0,id);//создаем новый поток
  MessageBox(0,'Red Alert','Alert',MB_OK);
end.
```

Основная программа сохраняет дескриптор потока в глобальной переменной *mainthread*. Это позволяет новому потоку, воспроизводящему звук, определить момент, когда основная программа завершит свою работу. В этот момент поток, воспроизводящий звук, также завершит свою работу.

Конечно, для того чтобы установить факт завершения работы основного потока, можно использовать вызов WaitForSingleObject, однако в данном случае удобнее обратиться к GetExitCodeThread. Если эта функция возвращает значение STILL_ACTIVE, значит, основной поток продолжает функционирование, а, следовательно, поток beepthread может продолжать звучать.

Что произойдет, если дочерний поток не будет следить за завершением работы родительского потока? В данном случае ничего. Программа будет вести себя абсолютно точно также. Дело в том, что при завершении работы главной функции процесса происходит вызов функции ExitProcess, которая автоматически уничтожает все потоки процесса.

2.3. Создание консольных приложений в Delphi, C++ Builder и Visual Studio

Для создания консольного приложения в Delphi(C++ Builder) необходимо выбрать пункт меню File->New, после чего в появившемся окне выбрать пиктограмму Console Application и нажать кнопку OK внизу окна (см. рисунок 1).

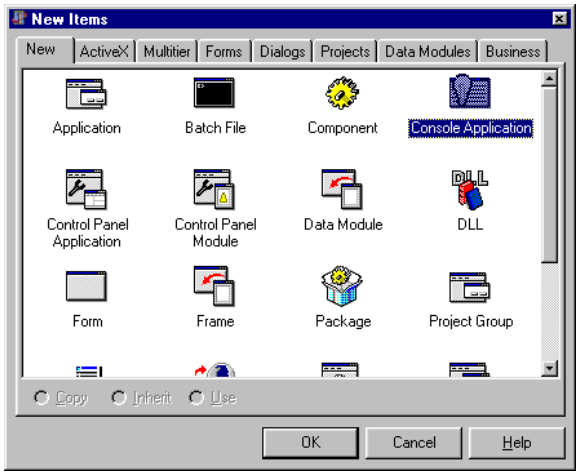

Рисунок 1 – Окно выбора типа нового проекта Delphi

После этого появится окно редактора программы, представленное на рисунке 2. Директива {$APPTYPE CONSOLE} определяет тип консольного приложения.

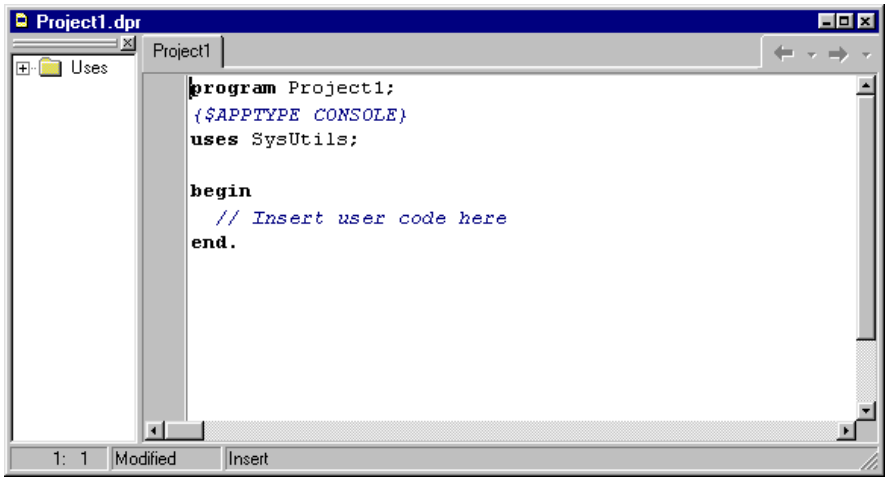

Рисунок 2 – Окно редактора программы вDelphi

Для использования системных функций создания и управления процессами и потоками в разделе **uses** Pascal-программы необходимо указать модуль **windows**. Аналогично в директиве #**include** Си-программы указывается заголовочный файл **windows.h**.

Для начала создания консольного приложения **в Visual Studio** необходимо создать новый проект (последовательность действий через пункты меню и закладки должна быть следующей: File -> New -> Projects -> Win32 Console Application, в строке ввода Project Name необходимо указать имя проекта, а в строке ввода Location – путь к файлам проекта). Для упрощения создания файлов проекта можно выбрать в мастере создания проектов один из вариантов: пустой проект, простое консольное приложение (содержит главный файл приложения с кодом функции main), простое консольное приложение «Hello, World!» (приложение, содержащее код простейшей программы, выводящей на экран указанное сообщение), а также приложение с использование MFC.

Главный файл проекта с исходным текстом программы будет называться также как и сам проект и иметь расширение cpp. Открыть этот файл для редактирования можно через закладку FileView ->WorkSpace -> Files -> Source Files.

Выполнение компиляции производится путем нажатия комбинации клавиш Ctrl+F7, создание исполняемого файла – F7, запуск программы на выполнение – Ctrl+F5. Настройки создаваемого проекта могут быть изменены с помощью пункта меню Projects -> Settings или по нажатию комбинации клавиш Alt+F7.

3. ВАРИАНТЫ ЗАДАНИЙ И ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

3.1.Написать программу **Sort**, реализующую следующий алгоритм:

3.1.1. Зафиксировать время начала **Tstrt** выполнения программы (пример получения текущего системного времени и вычисления разности между двумя значениями времени приведен в программе TimeDifference, представленной в Приложении).

3.1.2. Вывести на экран время **Tstrt** в формате: минуты:секунды:миллисекунды;

3.1.3. Зафиксировать текущий момент времени t1;

3.1.4. Для **i** от 1 до 100(количество повторений может быть изменено в зависимости от быстродействия процессора) повторять:

3.1.4.1. Заполнить массив целых чисел случайными значениями из диапазона 0-10000;

3.1.4.2. Отсортировать массив;

3.1.5. Зафиксировать текущий момент времени t2;

3.1.6. Определить среднее время одной сортировки: (t2-t1)/100;

3.1.7. Вывести на экран среднее время одной сортировки(в миллисекундах);

3.1.8. Зафиксировать время окончания **Tend** выполнения программы;

3.1.9. Вывести на экран время **Tend** в формате: минуты:секунды: миллисекунды;

Язык программирования, метод и направление сортировки, а также количество элементов массива N выбирается в соответствии с вариантом задания, приведенным в таблице 3.1.

3.2.Написать программу Master, выполняющую следующие действия:

3.2.1. Для **i** от 1 до 3 повторять:

3.2.2.1. Используя системные вызовы **CreateProcess,** создать два процесса **Sort** с классами приоритетов, в соответствии с вариантом задания (таблица 3.1). Необходимо, чтобы каждый процесс имел собственную консоль и окно консоли имело заголовок: "Process: *NP*; Prioritet: *PP*", где *NP* – номер процесса (1 или 2), а *PP* – приоритет соответствующего процесса. (Для изменения свойств окна консоли использовать структуру STARTUPINFO).

3.2.2.2. Ожидать окончания процессов, созданных в п. 3.2.2.1 (использовать функцию WaitForSingleObject).

3.3. Зафиксировать для отчета значения времени, получаемые при выполнении процессов в п 3.2.1.

3.4. Написать программу **Threads,** содержащую процедуру сортировки массива (разработанную при выполнении пункта 3.1), содержащего N/50 элементов и процедуру вывода массива на экран – **mass_print**. Программа должна выполнять следующие действия:

3.4.1. Генерировать случайный массив, содержащий N/50 элементов.

3.4.2. Используя системные вызовы **CreateThread,** создать программные потоки **sort** и **mass_print** в приостановленном состоянии.

3.4.3. Установить приоритеты потоков в THREAD_PRIORITY_NORMAL, используя системный вызов **SetThreadPriority.**

3.4.4. Активизировать потоки, используя системные вызовы **ResumeThread**;

3.4.5. Изменяя приоритеты потоков в разработанной программе в различных сочетаниях фиксировать получаемые результаты.

Таблица 3.1 - Варианты заданий

№ варианта	Среда программирования	Метод сортировки	Направление сортировки	N	Приоритеты*
1	Delphi	Пузырька	Возр.	3000	1-2, 1-3, 2-2
2	C++ Builder	Выбора	Возр.	3500	1-3, 1-1, 2-1
3	Visual Studio	Вставки	Возр.	4000	3-2, 3-3, 1-3
4	Delphi	Шелла	Возр.	4500	1-1, 2-3, 3-1
5	C++ Builder	Быстрая	Возр.	5000	1-2, 1-3, 2-2
6	Visual Studio	Пузырька	Возр.	3100	1-3, 1-1, 2-1
7	Delphi	Выбора	Возр.	3600	3-2, 3-3, 1-3
8	C++ Builder	Вставки	Возр.	4100	1-1, 2-3, 3-1
9	Visual Studio	Шелла	Возр.	4600	1-2, 1-3, 2-2
10	Delphi	Быстрая	Возр.	5100	1-3, 1-1, 2-1
11	C++ Builder	Пузырька	Возр.	3200	3-2, 3-3, 1-3
12	Visual Studio	Выбора	Возр.	3700	1-1, 2-3, 3-1
13	Delphi	Вставки	Возр.	4200	1-2, 1-3, 2-2
14	C++ Builder	Шелла	Возр.	4700	1-3, 1-1, 2-1
15	Visual Studio	Быстрая	Возр.	5200	3-2, 3-3, 1-3
16	Delphi	Пузырька	Убыв.	3300	1-1, 2-3, 3-1
17	C++ Builder	Выбора	Убыв.	3800	1-2, 1-3, 2-2
18	Visual Studio	Вставки	Убыв.	4300	1-3, 1-1, 2-1
19	Delphi	Шелла	Убыв.	4800	3-2, 3-3, 1-3
20	C++ Builder	Быстрая	Убыв.	5300	1-1, 2-3, 3-1
21	Visual Studio	Пузырька	Убыв.	3400	1-2, 1-3, 2-2
22	Delphi	Выбора	Убыв.	3900	1-3, 1-1, 2-1
23	C++ Builder	Вставки	Убыв.	4400	3-2, 3-3, 1-3
24	Visual Studio	Шелла	Убыв.	4900	1-1, 2-3, 3-1
25	Delphi	Быстрая	Убыв.	5400	1-2, 1-3, 2-2
26	C++ Builder	Пузырька	Убыв.	3500	1-3, 1-1, 2-1
27	Visual Studio	Выбора	Убыв.	4000	3-2, 3-3, 1-3
28	Delphi	Вставки	Убыв.	4500	1-1, 2-3, 3-1
29	C++ Builder	Шелла	Убыв.	5000	2-2, 1-2, 3-3
30	Visual Studio	Быстрая	Убыв.	5500	3-1, 1-1, 3-3

*) 1- IDLE_PRIORITY_CLASS; 2 - NORMAL_PRIORITY_CLASS; 3 - HIGH_PRIORITY_CLASS.

4. Содержание отчета

Отчет должен содержать следующие пункты:

- постановка задачи;
- тексты программ с комментариями;
- результаты выполнения программ;
- выводы.

5. Контрольные вопросы

1. Что такое Win API?
2. Что понимается под процессом в ОС Windows?
3. В чем разница между процессом и потоком в ОС Windows?
4. С помощью какого системного вызова можно создать новый процесс в ОС Windows? Приведите пример вызова и расскажите о его аргументах.
5. Какие классы приоритетов процессов определены в Windows?
6. Каким образом в ОС Windows возможно управлять запущенным процессом?
7. В чем отличие между дескриптором и идентификатором процесса?
8. С какой целью могут использоваться потоки в ОС Windows?
9. С помощью какого системного вызова можно создать новый поток в ОС Windows? Приведите пример вызова и расскажите о его аргументах.
10. Какие приоритеты потоков определены в Windows?
11. Каким образом можно изменить приоритет потока в Windows? Приведите пример.
12. Что такое базовый уровень приоритета потока?

ПРИЛОЖЕНИЕ

```
Program TimeDifference;
uses
  SysUtils,
  Windows;
var
   st:SYSTEMTIME;
    ft,ft1:FILETIME;
    li,li1,delta:LARGE_INTEGER;
begin

 getsystemtime(st);//фиксируем время начала выполнения
//...
// операторы программы
// ...
 getsystemtime(st1);//фиксируем время окончания выполнения
  SystemTimeToFileTime(st,ft);//преобразуем из формата SYSTEMTIME в //формат
                                  FILETIME
  SystemTimeToFileTime(st1,ft1);

  li.LowPart:=ft.dwLowDateTime;//преобразуем в формат LARGE_INTEGER(чтобы вычис-
                                  лить разницу)
  li.HighPart:=ft.dwHighDateTime;
  li1.LowPart:=ft1.dwLowDateTime;
  li1.HighPart:=ft1.dwHighDateTime;
  delta.QuadPart:=li1.QuadPart-li.QuadPart;// вычисляем разницу
  writeln('Start              Stop');
  writeln('____               ____');
  writeln(st.wMinute,'        ',st1.wMinute,' m');
  writeln(st.wSecond,'        ',st1.wSecond,' s');
  writeln(st.wMilliseconds,'          ',st1.wMilliseconds,' ms');
  write('delta:',delta.QuadPart/10000:10:5,' ms');
```

ЛАБОРАТОРНАЯ РАБОТА №3
Использование программного интерфейса Win API. Синхронизация процессов и потоков в ОС Windows.

1. Цель работы

Изучение программного интерфейса приложений(API) операционных систем Windows 9x, NT, ME, XP. Приобретение практических навыков синхронизации процессов и потоков, используя Win API в средах программирования Borland Delphi, C++ Builder или Visual Studio.

2. Основные положения

2.1. Синхронизация процессов и потоков

Операционные системы семейства Win (Windows 9x, NT, ME, XP, 7) позволяют организовывать параллельную работу множества процессов и множества потоков внутри процесса. При этом возникает задача правильной организации совместной работы процессов и потоков во времени – задача синхронизации их работы. Для решения этой задачи в Win API вводится ряд *синхронизационных объектов* и функций для работы с ними. Логика работы с синхронизационными объектами проста – одним из процессов (или потоков) создается синхронизационный объект и производится управление его состоянием. Синхронизационный объект имеет собственное уникальное *имя*, используя которое другие процессы могут *открывать* данный объект (получать его дескриптор) и проверять его состояние. Проверка состояния синхронизационного объекта другими процессами (или потоками) производится с помощью вызова функции WaitForSingleObject (WaitForMultipleObject для проверки состояния сразу нескольких объектов), проверяющей вошел ли синхронизационный объект в свое *сигнальное состояние*. Обращаясь к этой функции, вы можете указать, какой именно период времени следует ожидать перехода объекта в сигнальное состояние. Время указывается в миллисекундах. Если время равно нулю, функция просто проверит текущее состояние объекта и немедленно вернет управление вызвавшей программе. Если указать интервал времени, равный константе INFINITE, функция будет ожидать перехода объекта в сигнальное состояние бесконечно долго. Если поток ожидает перехода объекта в сигнальное состояние, он фактически не требует для своего функционирования дополнительного процессорного времени. Таким образом, перевод потока в режим ожидания объекта синхронизации весьма эффективен с точки зрения производительности всей системы. Вызов WaitForSingleObject возвращает несколько различных значений в зависимости от результатов проверки состояния объекта синхронизации. Если объект перешел в сигнальное состояние, функция возвращает значение WAIT_OBJECT_0. Если указанный при вызове период времени истек, а объект все еще не перешел в сигнальное состояние, функция возвращает WAIT_TIMEOUT. Далее будут рассмотрены некоторые специфические случаи использования функции WaitForSingleObject.

Рассмотрим несколько примеров задач, для решения которых необходима синхронизация процессов и синхронизационные объекты, наиболее подходящие для решения каждой из задач.

2.2. Мьютексы и критические секции

Предположим, что необходимо разработать пакет программ, которые одновременно работают с общим файлом данных – вносят в него какие-либо изменения. Программы данного пакета могут работать одновременно – например, одна из них записывает в файл данные, приходящие от каких-либо датчиков, подключенных к компьютеру, а другая позволяет пользователю редактировать данные в файле. Что произойдет, если одна из программ откроет файл для записи, начнет запись данных, а затем ее выполнение будет прервано операционной системой, и управление будет передано другой программе, которая попытается открыть тот же файл? Если файл был открыт для эксклюзивного доступа, то вторая программа не сможет его открыть - возникнет ошибка. Если файл был открыт для общего доступа, то данные, записываемые обеими программами, перемешаются – нарушится структура файла данных. Для того, чтобы избежать подобной ситуации, необходимо вводить какой-либо флаг, который должен сигнализировать занятость ресурса. Программа, желающая получить доступ к данному ресурсу, должна считать состояние флага, проверить, не установлен ли флаг (свободен ли ресурс), и если он еще не установлен, то установить флаг и занять ресурс. В качестве флага может использоваться какой-либо признак, разделяемый между различными процессами, например, наличие или отсутствие определенного файла и т.д. Данный алгоритм, хотя и применяется в некоторых системах, имеет большой недостаток. Предположим, что программа считала состояние флага и была прервана ОС, управление было передано другой программе, которая установила флаг и получила доступ к ресурсу, затем управление было передано вновь первой программе. Она приступает к следующей стадии – анализу состояния флага, но его состояние было считано до того, как его изменила вторая программа, и возникает ошибка. Таким образом, необходимо обеспечить блокирование доступа к флагу других процессов, пока какой-либо процесс считывает и анализирует его состояние. Для этого в Win API введен объект под названием *мьютекс* (сокращение от «mutual exception» - «взаимное исключение»). Поток может попытаться *занять* мьютекс. Если мьютексом не *владеет* никакой другой поток, то данный поток может стать *занять* его. После этого никакие другие потоки не смогут занять данный мьютекс до тех пор, пока первый поток не *освободит* его. Таким образом, мьютекс, как и разделяемый ресурс, может одновременно принадлежать только одному потоку, что исключает конфликты между потоками. В то же время, поток - владелец мьютекса может стать владельцем мьютекса повторно. Если поток присваивал себе мьютекс несколько раз, то он должен освободить его такое же количество раз.

Сигнальным состоянием мьютекса является его *свободное* состояние, то есть при проверке состояния свободного мьютекса функция WaitForSingleObject вернет значение WAIT_OBJECT_0. Если мьютекс был занят и не освободился до окончания периода ожидания, то будет возвращено значение WAIT_TIMEOUT, а если процесс, владеющий мьютексом, завершился, не освободив его, то система освобождает

мьютекс и функция WaitForSingleObject возвращает значение WAIT_ABANDONED. Набор функций Win API для работы с мьютексами представлен в таблице 2.1.

Таблица 2.1. Вызовы функций Win API для работы с мьютексами.

Вызов	Назначение
CreateMutex	Создает новый мьютекс или открывает уже существующий
OpenMutex	Открывает существующий мьютекс
ReleaseMutex	Освобождает мьютекс и делает его доступным для других потоков

Необходимо помнить, что мьютекс сам по себе не обеспечивает управления ресурсом. Он только указывает программисту (при правильном использовании), что ресурс был занят, а управление ресурсом необходимо реализовывать самостоятельно.

Ниже приведен пример консольного Win приложения на языке C++, реализующего синхронизацию процессов с помощью мьютексов. В качестве синхронизируемых процессов выступают запущенные пользователем экземпляры данного приложения. При запуске нескольких экземпляров данного приложения, первый запущенный процесс выведет на экран окно с сообщением о захвате мьютекса, и до тех пор, пока данное окно не будет закрыто, другие приложения не смогут вывести свои окна. После закрытия окна мьютекс освобождается и захватывается другим процессом.

```
#include <windows.h>
#include <iostream.h>
void main(){
        HANDLE mtx=CreateMutex (NULL, FALSE, "MyMutexName");
        cout <<"Проверяем мьютекс\n";
        WaitForSingleObject(mtx, INFINITE);
        cout <<"Мьютекс получен! \n";
        MessageBox (NULL, "Программа обладает мьютексом", "MUTEX", MB_OK);
        ReleaseMutex(mtx);
        CloseHandle(mtx);
}
```

Механизм мьютексов, предназначен, в основном, для синхронизации *процессов*. Конечно, мьютексы можно использовать и для синхронизации потоков внутри одного процесса, но для этого есть более подходящее решение – механизм *критических секций*. Критическая секция – это участок кода, одновременное выполнение которого несколькими потоками недопустимо.

Чтобы защитить участок кода от совместного использования несколькими потоками, определите переменную типа CRITICAL_SECTION и передайте ее адрес функции InitializeCriticalSection. К этой функции следует обращаться из главного потока вашей программы (обычно из потока main). В начале критической секции кода следует расположить обращение к функции EnterCriticalSection. Если в этот момент критическая секция исполняется каким-либо другим потоком, исполнение потока, обратившегося к EnterCriticalSection, будет блокировано до тех пор, пока критический участок не освободится. Если вы не хотите блокировать работу потока

и желаете, чтобы во время ожидания освобождения критической секции поток выполнял какую-либо другую полезную работу, вы можете использовать вызов TryEnterCriticalSection. Этот вызов возвращает значение TRUE в случае, если критический участок кода никем не занят. Если вызов вернул значение FALSE, поток может выполнить какую-либо другую работу, а затем вновь обратиться к TryEnterCritlcalSection.

Вызов TryEnterCriticalSecflon будет продолжать возвращать FALSE, а вызов EnterCriticalSection будет продолжать блокировать выполнение потоков до тех пор, пока поток, выполняющий критическую секцию, не обратится к LeaveCriticalSection. Таким образом, чаще всего функция, содержащая критический участок кода, выглядит следующим образом:

```
void one_at_a_time_please(){
  EnterCriticalSection(&section); // инициализация выполняется в другом месте
       // здесь расположен критический участок кода
  LeaveCriticalSection(&section);
 }
```

Другой вариант аналогичной функции:

```
BOOL only_one_at_a_time(){
        if (!TryEnterCriticalSection (&section))
        return FALSE;   // функция занята другим потоком
        // здесь расположен критический участок кода
        LeaveCriticalSection(Ssection); return TRUE; // мы сделали это!
}
```

Если вы больше не нуждаетесь в критической секции, вы обратитесь к DeleteCriticalSection, чтобы уничтожить ее. При этом система освободит ресурсы, связанные с критической секцией.

3. Варианты заданий и порядок выполнения работы

3.1. Написать программу **Sort3**, реализующую следующий алгоритм:

3.1.1. Полученить в качестве параметра командной строки номера процесса N_PRC;

3.1.2. Заполненить массив N целых чисел случайными значениями из диапазона 0-100;

3.1.3. Отсортировать массив;

3.1.4. Вывести на экран отсортированный массив в формате N_PRC:A[i](то есть перед выводом каждого элемента массива необходимо выводить значение параметра N_PRC);

Язык программирования, метод и направление сортировки, а также количество элементов массива N выбирается в соответствии с вариантом задания, приведенным в таблице 3.1.

3.2. Написать программу **Master3**, создающую процессы **Sort3**, используя системные вызовы **CreateProcess,** передавая в качестве параметра командной строки номер процесса – **N_PRC**. Необходимо, чтобы все процессы использовали одну консоль и имели класс приоритета NORMAL_PRIORITY_CLASS. Количество процессов выбирается исходя из варианта задания.

Так как процессы имеют одинаковый приоритет, то при выводе элементов массива на экран, используя одну консоль, процессы будут прерывать друг друга. Таким образом, выводимые массивы различных процессов будут «перемешаны». Убедитесь в этом запуская программу **Master3**.

В данном случае консоль выступает в качестве разделяемого ресурса и очевидна необходимость использования механизмов взаимоисключения, для предотвращения “смешивания” массивов при выводе.

3.3. Переписать программы **Sort3** и **Master3**, с использованием механизма взаимоисключений (необходимо использовать объект Mutex). Полученная программа **Master3_mtx** должна создавать мьютекс (с помощью вызова **CreateMutex)** перед созданием процессов, а программа **Sort3_mtx** должна получать идентификатор (handle) мьютекса (с помощью вызова **OpenMutex)**, занимать мьютекс перед выводом отсортированного массива на экран (с помощью вызова **WaitForSingleObject)**, а после вывода массива на экран освобождать мьютекс (используя вызов **ReleaseMutex)**.

3.4 Разработать программу **Master3_Crit** для исследования механизма критических секций.

Программа должна содержать функцию **Sort**, запускаемую несколько раз в виде отдельных потоков, с помощью вызовов CreateThread. Функция **Sort** должна: генерировать случайный массив из N элементов, сортировать его и выводить его на экран.

Убедиться в том, что при выполнении программы без использования механизма критических секций при выводе на экран из различных потоков отсортированные массивы будут «перемешиваться».

Добавить вызов InitializeCriticalSection перед созданием потоков и EnterCriticalSection, LeaveCriticalSection, соответственно, перед и после вывода массива на экран в функции **Sort**. Убедиться в том, что отсортированные массивы выводятся на экран не “перемешиваясь”.

Таблица 3.1 - Варианты заданий

№ варианта	Среда программирования	Метод	Направление сортировки	Количество процессов	N
1	Visual Studio	Выбора	Убыв.	3	100
2	Delphi	Вставки	Убыв.	4	90
3	C++ Builder	Шелла	Убыв.	5	70
4	Visual Studio	Быстрая	Убыв.	6	50
5	Delphi	Пузырька	Убыв.	3	80

№ варианта	Среда програм-мирования	Метод	Направление сортировки	Количество процессов	N
6	C++ Builder	Выбора	Убыв.	4	70
7	Visual Studio	Вставки	Убыв.	5	60
8	Delphi	Шелла	Убыв.	6	40
9	C++ Builder	Быстрая	Убыв.	3	90
10	Visual Studio	Пузырька	Убыв.	4	85
11	Delphi	Выбора	Убыв.	5	75
12	C++ Builder	Вставки	Убыв.	6	45
13	Visual Studio	Шелла	Убыв.	3	120
14	Delphi	Быстрая	Убыв.	4	100
15	C++ Builder	Пузырька	Убыв.	5	90
16	Visual Studio	Выбора	Возр.	6	30
17	Delphi	Вставки	Возр.	3	95
18	C++ Builder	Шелла	Возр.	4	80
19	Visual Studio	Быстрая	Возр.	5	65
20	Delphi	Пузырька	Возр.	6	35
21	C++ Builder	Выбора	Возр.	3	130
22	Visual Studio	Вставки	Возр.	4	100
23	Delphi	Шелла	Возр.	5	60
24	C++ Builder	Быстрая	Возр.	6	40
25	Visual Studio	Пузырька	Возр.	3	160
26	Delphi	Выбора	Возр.	4	125
27	C++ Builder	Вставки	Возр.	5	75
28	Visual Studio	Шелла	Возр.	6	85
29	Delphi	Быстрая	Возр.	2	200
30	C++ Builder	Пузырька	Возр.	3	155

4 Содержание отчета

Отчет должен содержать следующие пункты:

- постановка задачи;
- тексты программ с комментариями;
- результаты выполнения программ;
- выводы.

5 Контрольные вопросы

1. Что такое синхронизация потоков?
2. Какие синхронизационные объекты предоставляет Win API?
3. Что называется сигнальным состоянием синхронизационного объекта?
4. Каким образом проверяется состояние синхронизационного объекта?
5. Для чего предназначены мьютексы?
6. Как осуществляется синхронизация с помощью мьютексов?
7. Как осуществляется синхронизация с помощью критических секций?
8. Какой объект синхронизации предпочтительней применять при работе с потоками?
9. В каких случаях предпочтительно применять синхронизацию с помощью мьютексов, а в каких случаях – с помощью критических секций?
10. Приведите примеры необходимости синхронизации процессов и потоков?

ЛАБОРАТОРНАЯ РАБОТА №4
Синхронизация процессов и потоков в ОС Windows.
Семафоры.

1. Цель работы

Изучение программного интерфейса приложений(API) операционных систем Windows 9x, NT, ME, XP, 7. Приобретение практических навыков синхронизации потоков, с использованием семафоров в средах программирования Borland Delphi, C++ Builder или Visual Studio.

2. Основные положения

2.1. Семафоры

Все ключевые понятия, относящиеся к взаимоисключению, Дейкстра суммировал в концепции семафоров. *Семафор* – это защищенная переменная, значение которой можно опрашивать и менять только при помощи специальных операций P и V и операции инициализации. Семафоры, которые принимают только значения 0 и 1 называются *двоичными*(*бинарными*) семафорами. Семафоры, которые принимают любые неотрицательные целые значения называют *обобщенными* семафорами или *считающими* семафорами (семафорами со счетчиками).

Операция P над семафором записывается как P(S) и выполняется следующим образом:

если S>0, то S:=S-1
иначе (ожидать на S).

Операция V над семафором S записывается как V(S) и выполняется следующим образом:

если (один или более процессов ожидают на S)
то (разрешить одному из этих процессов продолжить работу)
иначе S:=S+1.

Операции P и V являются неделимыми. Участки взаимоисключения по семафору S в процессах обрамляются операциями P(S) и V(S). Если одновременно несколько процессов попытаются выполнить операцию P(S), это будет разрешено только одному из них, а остальным придется ждать.

Семафоры и операции над ними могут быть реализованы как аппаратно, так и программно. Как правило, они реализуются в ядре операционной системы, где осуществляется управление сменой состояния процессов.

С помощью двоичного семафора процессы могут организовать взаимное исключение, просто заключив свои критические участки в «скобки», роль которых играют операции P(S) и V(S).

С помощью обобщенных семафоров легко решается одна из традиционных задач параллельных вычислений, задача поставщика-потребителя, заключающаяся в следующем: имеются один или несколько производителей, генерирующих данные

некоторого типа и помещающих их в буфер, а также единственный потребитель, который извлекает данные из буфера. Требуется защитить систему от перекрытия операций с буфером, т.е. чтобы производитель не имел возможности записи в заполненный буфе, производитель не имел возможности чтения из пустого буфера, и одновременно доступ к буферу мог получить только один процесс (производитель или потребитель).

Рещение задачи производитель-потребитель с использованием обобщенных семафоров представлено в листинге ниже. Переменная СВОБ – это семафор, значение, которого равно числу свободных элементов буфера. Переменная ЗАПОЛН – семафор, значение которого равно числу заполненных элементов буфера. Семафор ВЗАИСК (ВЗАимное ИСКлючение) – двоичный; он гарантирует, что в каждый момент только один процесс сможет работать с буфером.

```
Begin
 integer СВОБ, ЗАПОЛН, ВЗАИСК;
  СВОБ:= количество свободных буферов;
  ЗАПОЛН:= 0;
  ВЗАИСК:=1;
     ПОСТАВЩИК:  begin
                 do while (true);
                   приготовить данные;
                   P(СВОБ);
                   P(ВЗАИСК);
                   поместить данные в буфер;
                   V(ВЗАИСК);
                   V(ЗАПОЛН);
                 end
     end ;
     ПОТРЕБИТЕЛЬ: begin
                 do while (true);
                    P(ЗАПОЛН);
                    P(ВЗАИСК);
                    получить данные из буфера;
                    V(ВЗАИСК);
                    V(СВОБ);
                    обработать данные;
                 end
     end
  end
```

С помощью обобщенных семафоров можно естественным образом организовать управление ресурсами. В решении задачи о поставщике и потребителе общие семафоры применены для учета заполненных и свободных элементов буфера. Их можно точно так же применить и для распределения лентопротяжных устройств,

дисков, и т.п., а также для любых *делимых* ресурсов (делимыми называются ресурсы, которые могут быть в состоянии "частично занят", как, например, рассмотренный выше буфер).

2.2. Семафоры в Win API.

Как было описано выше, *семафор* – это синхронизационный объект, содержащий счетчик, который изначально устанавливается в какое-либо значение (обычно равное количеству разделяемых ресурсов). Каждый раз, когда поток хочет захватить ресурс, значение семафора уменьшается на 1. Когда поток освобождает ресурс, значение семафора увеличивается на 1. Когда значение семафора равно нулю, семафор становится недоступным для каких-либо потоков.

Сигнальным состоянием семафора является ненулевое его состояние. Логика работы с семафорами очень проста. При инициализации семафора указывается его начальное значение. Каждый раз, когда программа обращается к семафору (с помощью вызова WaitForSingleObject или WaitForMultipleObjects), его значение уменьшается на 1. Когда значение семафора становится равно 0, семафор становится недоступен. Когда поток освобождает семафор (с помощью вызова ReleaseSemaphore), значение семафора увеличивается на 1. Набор функций Win API для работы с семафорами представлен в таблице 2.1:

Таблица 2.1 – Системные вызовы Win API для работы с семафорами

Вызов	**Назначение**
CreateSemaphore	Создает новый семафор или открывает уже существующий
OpenSemaphore	Открывает существующий семафор
ReleaseSemaphore	Добавляет некоторое значение (обычно 1) к значению семафора, делая его доступным для большего количества потоков

Ниже приведен пример консольного Win приложения на языке C++, реализующий с помощью семафоров ограничение количество запущенных экземпляров приложения. В качестве синхронизируемых процессов выступают запущенные пользователем экземпляры данного приложения. При запуске первого экземпляра приложения создастся семафор с максимальным значением 3. Первые три запущенных процесса смогут обратиться к семафору и вывести на экран окно с сообщением. Последующие запущенные экземпляры приложения не смогут вывести свои окна, пока не будет закрыто одно из открытых окон. После закрытия окна значение семафора увеличивается, и он становится доступен для других ожидающих процессов. Таким образом, сколько бы экземпляров данного приложения не было запущено, будет выведено не более трех окон *MessageBox* одновременно.

```
#include <windows.h>
#include <iostream.h>
void main(){
```

```
    HANDLE sem==CreateSemaphore(NULL, 3, 3, "MySemaphoreName");
    cout<<"Проверяем состояние семафора \n";
    WaitForSingleQbject (sem, INFINITE);
    cout<<" Получен доступ к семафору \n";
    MessageBox (NULL, "Получен доступ к семафору", "Semaphore", MB_OK);
    ReleaseSemaphore (sem, l, NULL);
    CloseHandle (sem);
}
```

3. Варианты заданий и порядок выполнения работы

3.1 Написать программу, содержащую два потока. Первый поток генерирует последовательность чисел и помещает их в кольцевой* буфер из N_{buf} элементов (с проверкой на свободное место в буфере с использованием механизма семафоров). Второй считывает данные из буфера и выводит их на экран.

*) при заполнении кольцевого буфера добавление элементов продолжается сначала, т.е. для вычисления индекса очередного элемента используется операция деления по модулю N_{buf}(Pascal: i mod N_{buf}; Си: I % N_{buf}).

Количество элементов, тип последовательности и длина буфера выбираются в соответствии с вариантом задания (таблица 3.1).

3.2 Определить максимальную длину буфера при различных приоритетах оттоков.

Таблица 3.1 – Варианты заданий

№ варианта	Среда программирования	Вид последовательности	Nbuf	N
1	C++ Builder	Натуральные числа	5	500
2	Visual Studio	Арифм. Прогресс. С разностью 2	6	300
3	Delphi	Арифм. Прогресс. С разностью 3	7	200
4	C++ Builder	Арифм. Прогресс. С разностью 4	8	100
5	Visual Studio	Натуральные числа	9	450
6	Delphi	Арифм. Прогресс. С разностью 2	10	250
7	C++ Builder	Арифм. Прогресс. С разностью 3	5	150
8	Visual Studio	Арифм. Прогресс. С разностью 4	6	70
9	Delphi	Натуральные числа	7	520
10	C++ Builder	Арифм. Прогресс. С разностью 2	8	330
11	Visual Studio	Арифм. Прогресс. С разностью 3	9	170
12	Delphi	Арифм. Прогресс. С разностью 4	10	90
13	C++ Builder	Натуральные числа	5	600
14	Visual Studio	Арифм. Прогресс. С разностью 2	6	350
15	Delphi	Арифм. Прогресс. С разностью 3	7	250
16	C++ Builder	Арифм. Прогресс. С разностью 4	8	120
17	Visual Studio	Натуральные числа	9	600
18	Delphi	Арифм. Прогресс. С разностью 2	10	310
19	C++ Builder	Арифм. Прогресс. С разностью 3	5	240
20	Visual Studio	Арифм. прогресс. с разностью 4	6	80
21	Delphi	Натуральные числа	7	490
22	C++ Builder	Арифм. прогресс. с разностью 2	8	320

23	Visual Studio	Арифм. прогресс. с разностью 3	9	230
24	Delphi	Арифм. прогресс. с разностью 4	10	140
25	C++ Builder	Натуральные числа	5	460
26	Visual Studio	Арифм. прогресс. с разностью 2	6	280
27	Delphi	Арифм. прогресс. с разностью 3	7	340
28	C++ Builder	Арифм. прогресс. с разностью 4	8	230
29	Visual Studio	Натуральные числа	9	320
30	Delphi	Арифм. прогресс. с разностью 2	10	370

4. Содержание отчета

Отчет должен содержать следующие пункты:

- постановка задачи;
- тексты программ с комментариями;
- результаты выполнения программ;
- выводы.

5. Контрольные вопросы

1. Что такое синхронизация потоков?
2. Какие синхронизационные объекты предоставляет Win API?
3. Как осуществляется синхронизация с помощью семафоров?
4. В каком случае целесообразно использовать семафоры?
5. Как выполняется операция P над семафором?
6. Как выполняется операция V над семафором?
7. Какие системные вызовы используются в Windows для работы с семафорами?

Библиографический список

1. Попов А.В. - Командная строка и сценарии Windows [Електронний ресурс]: – Режим доступа: http://www.intuit.ru/department/os/compromtwin/ – Название с экрана.
2. Олифер В.Г. Сетевые операционные системы : Учебник/ В.Г. Олифер, Н.А. Олифер. - СПб.;М.;Харьков: Питер, 2001.-538,[6] с. :ил.
3. Столлингс В. Операционные системы 4-е издание./ В.Столлингс. - М. : Издательский дом "Вильямс", 2002 - 848 с.: ил.
4. Вильямс А. Системное программирование в Windows2000 С-Пб:Питер, 2001. – 624 с. ил.
5. Рихтер Дж. Windows для профессионалов: создание эффективных Win приложений с учетом специфики 64-разрядной версии Windows/Пер, англ - 4-е изд. - СПб; Питер; М.: Издательско-торговый дом "Русская Редакция", 2001. - 752 с.; ил.