Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Факультет компьютерного проектирования

Кафедра инженерной психологии и эргономики

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Рекомендовано УМО по образованию в области информатики и радиоэлектроники в качестве учебно-методического пособия для направления специальности 1-40 05 01-09 «Информационные системы и технологии (в обеспечении промышленной безопасности)»

Минск БГУИР 2017

УДК 004.45(076.5)
ББК 32.973.26-018.2я73
С40

А в т о р ы:
И. Ф. Киринович, К. Д. Яшин, А. А. Быков, И. А. Рубанова

Р е ц е н з е н т ы:
кафедра информационных систем управления
Белорусского государственного университета
(протокол №4 от 24.11.2016);

кафедра информационных систем и технологий учреждения образования
«Белорусский государственный технологический университет»
(протокол №4 от 21.11.2016);

кафедра интеллектуальных систем Белорусского национального
технического университета (протокол №2 от 27.09.2016);

доцент кафедры программного обеспечения вычислительной техники
и автоматизированных систем Белорусского национального технического
университета, кандидат технических наук, доцент Н. А. Разоренов;

профессор кафедры вычислительных методов и программирования
учреждения образования «Белорусский государственный университет
информатики и радиоэлектроники», доктор физико-математических наук,
профессор С. В. Колосов

Системное программное обеспечение : учеб.-метод. пособие /
С40 И. Ф. Киринович [и др.]. – Минск : БГУИР, 2017. – 132 с. : ил.
ISBN 978-985-543-344-7.

Содержит материалы к лабораторным занятиям, предназначенным для освоения основ системного программирования в операционных системах семейства Unix/Linux и Windows и включающим теоретическую часть, варианты заданий и контрольные вопросы.

УДК 004.45(076.5)
ББК 32.973.26-018.2я73

ISBN 978-985-543-344-7

Содержание

Введение

В учебно-методическом пособии дается описание 12 лабораторных работ по дисциплине «Системное программное обеспечение», предназначенной для изучения принципов организации, проектирования и анализа современных операционных систем, освоения основ системного программирования в операционных системах семейства Unix/Linux и Windows. Выполнение работ рассчитано на два семестра и позволит получить практические умения в области выбора, конфигурирования, администрирования и программирования системного программного обеспечения.

Теоретический и практический материал, представленный в учебно-методическом пособии, поможет студентам в эффективном освоении курса.

Авторский коллектив благодарит заведующего кафедрой информационных технологий автоматизированных систем БГУИР А. А. Навроцкого и ассистента кафедры программного обеспечения информационных технологий БГУИР В. А. Леванцевича за помощь, оказанную при подготовке пособия.

Лабораторная работа №1

Работа с файлами и каталогами в ОС Linux

Цель работы – изучение основных системных вызовов и функций в ОС Linux для работы с файлами и каталогами.

Теоретическая часть

Файловая система – это структура, с помощью которой ядро операционной системы предоставляет пользователям (и процессам) ресурсы долговременной памяти системы, т. е. памяти на различного вида долговременных носителях информации – жестких дисках, флэш-памяти, CD-ROM и т. д.

С точки зрения пользователя, файловая система – это логическая структура каталогов и файлов. В отличие от Windows, где каждый логический диск хранит отдельное дерево каталогов, во всех UNIX-подобных системах эта древовидная структура растет из одного корня: она начинается с корневого каталога, родительского по отношению ко всем остальным, а физические файловые системы разного типа, находящиеся на разных разделах и даже на удаленных машинах, представляются как ветви этого дерева.

Имена файлов и каталогов могут иметь длину до 255 символов. Символы «/» (слэш) и символ с кодом 0 запрещены. Кроме этого, ряд символов имеет специальное значение для командного интерпретатора и их использование не рекомендуется. Это символы: ~ ! @ # $ & % * ( ) [ ] { } ' " \ : ; > < пробел.

Следует заметить, что символа «.» среди специальных символов нет, и имена вроде this.is.a.text.file.containing.the.famous.string.hello.world допустимы и широко распространены.

Если имя файла начинается с точки, то этот файл считается скрытым: некоторые команды его «не видят». Например, введя в своем домашнем каталоге команду просмотра содержимого каталога ls с ключом -a, означающим «показывать скрытые файлы», вы увидите больше файлов, чем введя ту же команду без ключей.

Linux различает регистр символов в именах файлов: так, в одном каталоге могут находиться два разных файла README и Readme.

В Linux существуют следующие типы файлов:

1. Обычный файл.
2. Каталог.
3. Специальный файл устройства.
4. Файл FIFO, или именованный канал.
5. Связь (link).
6. Сокет (socket).

Обычный файл – наиболее общий тип файлов, содержащий данные в некотором формате. Для операционной системы такие файлы представляют собой просто последовательность байтов. Интерпретация содержимого определяется прикладной программой, обрабатывающей файлы.

Каталог – файл, содержащий имена находящихся в нем файлов, а также указатели на дополнительную информацию (индексные дескрипторы), позволяющую операционной системе производить операции над этими файлами. Каталоги определяют положение файла в дереве файловой системы, т. к. сам файл не содержит информации о своем местонахождении. Каталог представляет собой таблицу, где каждая запись соответствует одному файлу (таблица 1).

Таблица 1

Индексные дескрипторы	Имена файлов
1753	.
2036	..
751	P1.txt
854	P2.c

Специальный файл устройства обеспечивает доступ к физическому устройству. В Linux различаются символьные и блочные файлы устройств. Символьные файлы применяются для небуферизированного обмена данными с устройствами (клавиатура, экран терминала), а блочные – для обмена данными в виде пакетов фиксированной длины (дисковые накопители). К некоторым устройствам доступ может быть как блочный, так и символьный. Доступ к устройствам, как и к файлам, осуществляется путем открытия, чтения и записи в специальные файлы устройств.

Файл FIFO (или именованный канал) – специальный файл для связи между процессами на запись и чтение; позволяет связать по данным несколько процессов. Поддержка файла FIFO появилась в Linux, начиная с Release-5.

Связь (link) – файлы для создания символических ссылок.

Сокет (socket) – специальный файл, предназначенный для взаимодействия между процессами. Интерфейс сокетов часто используется для доступа к сети TCP/IP, в некоторых реализациях с его помощью осуществляется межпроцессорное взаимодействие.

Имена каталогов строятся по точно тем же правилам, что и имена файлов. Полным именем файла (или путем к файлу) называется список вложенных друг в друга каталогов, заканчивающийся собственно именем файла. Начинаться он

может с любого каталога, т. к. в древовидной структуре между любыми двумя узлами существует путь. Если этот список начинается с корневого каталога, то путь называется абсолютным, если с любого другого – то относительным (по отношению к этому каталогу).

Корневой каталог обозначается символом «/» (слэш), и этим же символом разделяются имена каталогов в списке. Таким образом, абсолютным именем файла README в домашнем каталоге пользователя petrov будет /home/petrov/README.

На рисунке 1 изображена структура файловой системы Linux.

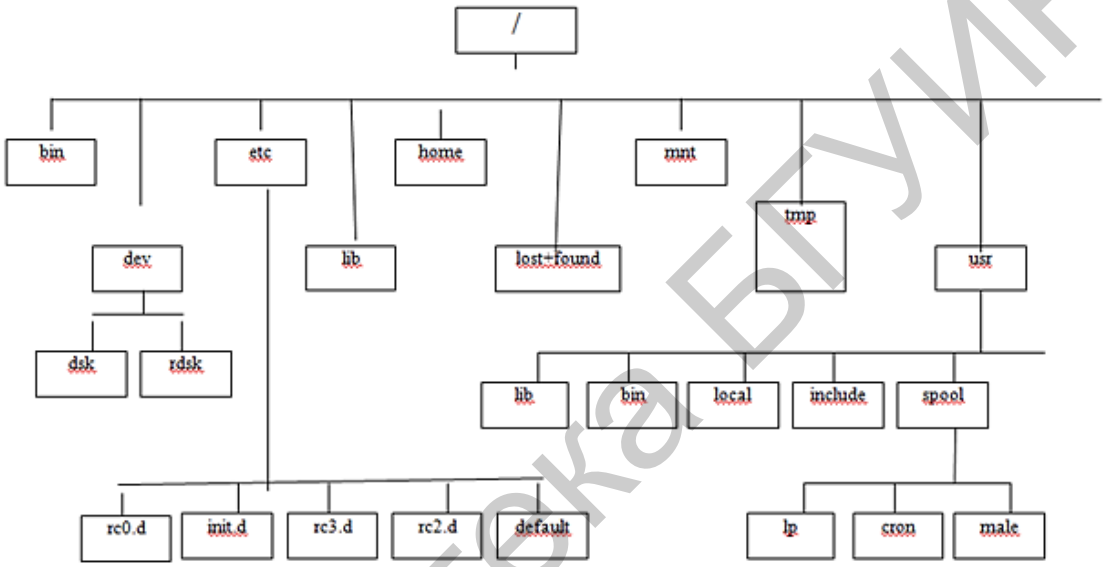

Рисунок 1 – Структура файловой системы Linux

В каждом каталоге существуют два особых «подкаталога» с именами «две точки» и «точка». Первый из них служит указанием на однозначно определенный родительский каталог, а второй – на данный каталог. Для корневого каталога, у которого нет родителя, оба эти «подкаталога» указывают на корневой каталог. С помощью этих имен образуются относительные имена файлов.

Для выполнения операций записи и чтения данных в существующем файле его следует открыть при помощи системного вызова ***open()***.

int open (const char *pathname, int flags, [mode_t mode]);

intfopen (const char *pathname, int flags, [mode_t mode]);

Второй аргумент системного вызова ***open-flags*** имеет целочисленный тип и определяет метод доступа. Параметр ***flags*** принимает одно из значений, заданных постоянными в заголовочном файле ***fcnt1.h***. В файле определены три постоянные:

O_RDONLY – открыть файл только для чтения;

O_WRONLY – открыть файл только для записи;

O_RDWR – открыть файл для чтения и записи,

или «***r***», «***w***», «***rw***» для ***fopen().***

Третий параметр ***mode*** устанавливает права доступа к файлу и является необязательным, он используется только вместе с флагом ***O_CREAT***.

Пример создания нового файла:

```
# include <sys / types.h>
# include <sys / stat.h>
# include <fcnt1.h>
int Fd1;
FILE *F1;
F1=fopen (“Myfile2.txt”, “w”, 644);
Fd1=open (“Myfile1.txt”, O_CREAT, 644);
```

Системные вызовы ***stat*** и ***fstat*** позволяют процессу определить значения свойств в существующем файле.

```
#include <sys/types.h>
#include <sys/stat.h>
int stat (const char *pathname, struct stat *buf);
intfstat (int filedes, struct stat *buf);
```

Пример: ***stat(“1.exe”, &st1);*** где: ***pathname*** – полное имя файла, ***buf*** – структура типа ***stat***. Эта структура после успешного вызова будет содержать связанную с файлом информацию.

Поля структуры ***stat*** включают следующие элементы:

```
structstat {
dev_tst_dev;            /* логическое устройство, где находится файл */
ino_tst_ino;            /*номер индексного дескриптора */
mode_tst_mode;          /* права доступа к файлу*/
nlink_tst_nlink;        /* количество жестких ссылок на файл */
uid_tst_uid;            /* ID пользователя-владельца */
gid_tst_gid;            /* ID группы-владельца */
dev_tst_rdev;           /* тип устройства */
off_tst_size;           /* общий размер в байтах */
unsignedlongst_blksize;     /* размер блока ввода-вывода */
unsignedlongst_blocks;      /* число блоков, занимаемых файлом */
time_tst_atime;         /* время последнего доступа */
time_tst_mtime;         /* время последней модификации */
time_tst_ctime;         /* время последнего изменения */
        };
```

Права доступа в Linux. Права доступа к файлам представлены в виде последовательности бит, где каждый бит означает разрешение на запись (***w***), чтение (***r***) или выполнение (***x***). Права доступа записываются для владельца-создателя файла (***owner***), группы, к которой принадлежит владелец-создатель файла (***group***), и всех остальных (***other***).

Например, при выводе команды ***dir*** запись типа

-rwxr-xr-w1.exe

означает, что владелец файла ***1.exe*** имеет права на чтение, запись и выполнение, группа имеет права только на чтение и выполнение, все остальные имеют права только на чтение. В восьмеричном виде получится значение ***0754***. В действительности манипулирует файлами не сам пользователь, а запущенный им процесс. Для просмотра прав доступа можно использовать функцию ***stat***.

Для записи прав доступа служит функция ***chmod***:

```
#include <sys/types.h>
#include <sys/stat.h>
intchmod(const char *pathname, mode_t mode);
```

Пример: ***chmod("1.exe", 0777);***

Каталоги в ОС Linux – это особые файлы. Для открытия или закрытия каталогов существуют вызовы:

```
#include <dirent.h>
DIR *opendir (const char *dirname);
intclosedir( DIR *dirptr);
```

Для работы с каталогами существуют системные вызовы:

intmkdir (const char *pathname, mode_t mode) – создание нового каталога;
intrmdir(const char *pathname) – удаление каталога.

Первый параметр – имя создаваемого каталога, второй – права доступа:

```
retval=mkdir("/home/s1/t12/alex",0777);
retval=rmdir("/home/s1/t12/alex");
```

Заметим, что вызов ***rmdir("/home/s1/t12/alex")*** будет успешен, только если удаляемый каталог пуст, т. е. содержит записи «точка» (.) и «двойная точка» (..).

Для чтения записей каталога существует вызов:

```
structdirent *readdir(DIR *dirptr);
```

Структура ***dirent*** такова:

```
structdirent {
        longd_ino;
```

```
off_td_off;
unsigned short d_reclen;
chard_name [1];
        };
```

Поле ***d_ino*** - это число, которое уникально для каждого файла в файловой системе. Значением поля ***d_off*** служит смещение данного элемента в реальном каталоге. Поле ***d_name*** является началом массива символов, задающего имя элемента каталога. Данное имя ограничено нулевым байтом и может содержать не более ***MAXNAMLEN*** символов. Таким образом, описываемая структура имеет переменную длину, хранящуюся в поле ***d_reclen***.

Пример вызова:

```
DIR *dp;
structdirent *d;
d=readdir(dp);
```

При первом вызове функции ***readdir*** в структуру ***dirent*** будет считана первая запись каталога. После прочтения всего каталога в результате последующих вызовов ***readdir*** будет возвращено значение ***NULL***. Для возврата указателя в начало каталога на первую запись существует вызов:

```
voidrewindir(DIR *dirptr).
```

Чтобы получить имя текущего рабочего каталога, существует функция:

```
char  *getcwd(char *name, size_t size).
```

Время в Linux отсчитывается в секундах, прошедших с начала этой эпохи (***00:00:00 UTC, 1 Января 1970 года***). Для получения системного времени можно использовать следующие функции:

```
#include<sys/time.h>
time_t time (time_t *tt);
intgettimeofday(structtimeval *tv, structtimezone *tz);
structtimeval {
                    longtv_sec;       /* секунды */
                    longtv_usec;      /* микросекунды */
                  };
```

Для выполнения программ на языке С в ОС Linux необходимо использовать встроенный компилятор ***gcc***. Процесс отладки состоит из следующих шагов:

1. Набрать текст программы. Для этого удобно использовать текстовый редактор ***gedit*** или встроенный редактор ***VI***.

Например, набрать текст:

```
#include <stdio.h>
int main(void)
{
printf("Hello world!\n");
return(0);
}
```

2. Сохранить текст программы в текущей папке домашнего каталога. При этом сохраняемому файлу присваивается имя (например ***hello***) и расширение С (программа на языке С).

3. Открыть терминал. Если необходимо, перейти в каталог, где сохранен файл с расширением С. Набрать команду:

gcc hello.c

В каталоге появился новый файл ***a.out***. Это и есть исполняемый файл. Чтобы его выполнить, надо в консоли набрать команду: ***./a.out***

Программа должна запуститься, т. е. должен появиться текст:

Hello world!

Компилятор ***gcc*** по умолчанию присваивает всем созданным исполняемым файлам имя ***a.out***. Если хотите назвать его по-другому, нужно к команде на компиляцию добавить флаг -o и имя, которым вы хотите его назвать.

При выполнении команды

gcc hello.c -o hello.exe

в каталоге появился исполняемый файл с названием ***hello.exe***. После его запуска ***./hello.exe*** получится такой же исполняемый файл, только с более удобным названием.

Флаг **-o** является одним из многочисленных флагов компилятора ***gcc***. Чтобы просмотреть все возможные флаги, можно воспользоваться справочной системой ***man***.

Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы.
2. Написать программу вывода сообщения на экран.
3. Написать программу ввода символов с клавиатуры и записи их в файл (в качестве аргумента при запуске программы вводится имя файла). Для чтения или записи файла использовать только функции посимвольного ввода-вывода ***getc(), putc()***, ***fgetc()***, ***fputc()***. Предусмотреть выход после ввода определенного символа (например ***ctrl-F***). Предусмотреть контроль ошибок открытия/закрытия/чтения файла.

4. Написать программу вывода содержимого текстового файла на экран (в качестве аргумента при запуске программы передается имя файла, второй аргумент (***N***) устанавливает вывод по группам строк (по ***N*** строк) или сплошным текстом (***N***=0)). Для вывода очередной группы строк необходимо ожидать нажатия пользователем любой клавиши. Для чтения или записи файла использовать ***только*** функции посимвольного ввода-вывода ***getc(), putc()***, ***fgetc()***, ***fputc()***. Предусмотреть контроль ошибок открытия/закрытия/чтения/записи файла.

5. Написать программу копирования одного файла в другой. В качестве параметров при вызове программы передаются имена первого и второго файлов. Для чтения или записи файла использовать ***только*** функции посимвольного ввода-вывода ***getc(), putc()***, ***fgetc()***, ***fputc()***. Предусмотреть копирование прав доступа к файлу и контроль ошибок открытия/закрытия/чтения/записи файла.

6. Написать программу вывода на экран содержимого текущего и корневого каталогов. Предусмотреть контроль ошибок открытия/закрытия/чтения каталога.

7. Написать отчет.

Варианты индивидуальных заданий

1. Выполнить сортировку файлов в заданном каталоге (аргумент 1 командной строки) и во всех его подкаталогах по следующим критериям (аргумент 2 командной строки задается в виде целого числа): 1 – по размеру файла, 2 – по имени файла. Записать отсортированные файлы в новый каталог (аргумент 3 командной строки).

2. Найти в заданном каталоге (аргумент 1 командной строки) и всех его подкаталогах заданный файл (аргумент 2 командной строки). Вывести на консоль полный путь к файлу, имя файла, его размер, дату создания, права доступа, номер индексного дескриптора. Вывести общее количество просмотренных каталогов и файлов.

3. Для заданного каталога (аргумент 1 командной строки) и всех его подкаталогов вывести в заданный файл (аргумент 2 командной строки) и на консоль имена файлов, их размер и дату создания, удовлетворяющие заданным условиям: 1 – размер файла находится в заданных пределах от $N1$ до $N2$ ($N1$, $N2$ задаются в аргументах командной строки), 2 – дата создания находится в заданных пределах от $M1$ до $M2$ ($M1$, $M2$ задаются в аргументах командной строки).

4. Найти совпадающие по содержимому файлы в двух заданных каталогах (аргументы 1 и 2 командной строки) и всех их подкаталогах. Вывести на консоль и в файл (аргумент 3 командной строки) его имя, размер, дату создания, права доступа, номер индексного дескриптора.

5. Подсчитать суммарный размер файлов в заданном каталоге (аргумент 1 командной строки) и для каждого его подкаталога отдельно. Вывести на консоль и в файл (аргумент 2 командной строки) название подкаталога, количество файлов в нем, суммарный размер файлов, имя файла с наибольшим размером.

6. Написать программу, находящую в заданном каталоге и всех его подкаталогах все файлы заданного размера. Имя каталога задается пользователем в качестве первого аргумента командной строки. Диапазон (min – max) размеров файлов задается пользователем в качестве второго и третьего аргументов командной строки. Программа выводит результаты поиска в файл (четвертый аргумент командной строки) в следующем виде: полный путь, имя файла, его размер. На консоль выводится общее число просмотренных файлов.

Содержание отчета

Отчет по лабораторной работе должен быть оформлен в соответствии с требованиями, предъявляемыми к текстовым документам, и содержать:

- цель работы;
- краткие теоретические сведения, необходимые для выполнения работы;
- описание порядка выполнения работы с приведением листинга кода, скриншотов рабочих окон;
- выводы по работе.

Контрольные вопросы

1. Что такое файл?
2. Какие задачи решает файловая система?
3. Что такое права доступа?
4. Какие виды прав доступа вы знаете?

Лабораторная работа №2

Процессы в ОС Linux

Цель работы – изучение вопросов порождения процессов и управления ими в ОС Linux.

Теоретические сведения

Термины «программа» и «задание» предназначены для описания статических, неактивных объектов. Программа в процессе исполнения является динамическим, активным объектом. Не существует взаимно однозначного соответствия между процессами и программами. В некоторых ОС для работы программ может организовываться более одного процесса. Один и тот же процесс могут исполнять последовательно несколько различных программ.

Понятие «процесс» характеризует некоторую совокупность набора исполняющихся команд, ассоциированных с ним ресурсов (выделенная для исполнения память или адресное пространство, стеки, используемые файлы, устройства ввода-вывода и т. д.) и текущего момента его выполнения (значения регистров, программного счетчика, состояние стека и значения переменных). Изменением состояния процессов занимается ОС, совершая операции над ними. Основные операции над процессами удобно объединить в три пары:

- создание процесса – завершение процесса (одноразовые);
- приостановка процесса (перевод из состояния «исполнение» в состояние «готовность») – запуск процесса (перевод из состояния «готовность» в состояние «исполнение»);
- блокирование процесса (перевод из состояния «исполнение» в состояние «ожидание») – разблокирование процесса (перевод из состояния «ожидание» в состояние «готовность»).

Существует еще одна (непарная) операция – изменение приоритета процесса.

У процесса выделяют следующие контексты:

- регистровый (содержимое всех регистров процессора);
- системный (запись в таблице процессов, управляющая информация о процессе и пр.);
- пользовательский (код и данные).

Совокупность всех вышеуказанных контекстов называют контекстом процесса, в любой момент полностью характеризующим процесс.

В ОС Linux для создания процессов используется системный вызов ***fork():***

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

В результате успешного вызова ***fork()*** ядро создает новый процесс, который является почти точной копией вызывающего процесса. Другими словами, новый процесс выполняет копию той же программы, что и создавший его процесс. При этом все его объекты данных имеют те же самые значения, что и в вызывающем процессе.

Созданный процесс называется ***дочерним***, а процесс, осуществивший вызов ***fork()***, – ***родительским***.

После вызова родительский процесс и его вновь созданный потомок выполняются одновременно, при этом оба процесса продолжают выполнение с оператора, который следует сразу же за вызовом ***fork()***. Процессы выполняются в разных адресных пространствах, поэтому прямой доступ к переменным одного процесса из другого невозможен.

Например, следующая программа более наглядно показывает работу вызова ***fork()*** и использование процесса:

```
#include <stdio.h>
#include <unistd.h>
intmain ()
{
pid_tpid;                              /* идентификатор процесса */
printf ("Пока всего один процесс\n");
pid = fork ();                         /*Создание нового процесса */
printf ("Ужедва процесса\n");
if (pid = = 0){
            printf ("Это дочерний процесс, его pid=%d\n", getpid());
            printf ("А pid его родительского процесса=%d\n", getppid());
      }
elseif (pid> 0)
            printf ("Это родительский процесс pid=%d\n", getpid());
else
printf ("Ошибка вызова fork, потомок не создан\n");
}
```

Для корректного завершения дочернего процесса в родительском процессе необходимо использовать функцию ***wait()*** или ***waitpid()***:

```
pid_twait(int*status);
pid_twaitpid(pid_tpid, int *status, int options);
```

Функция ***wait*** приостанавливает выполнение родительского процесса до тех пор, пока дочерний процесс не прекратит выполнение, или до появления сигнала, который либо завершает текущий процесс, либо требует вызвать функцию-обработчик. Если дочерний процесс к моменту вызова функции уже завершился,

то функция немедленно возвращается. Системные ресурсы, связанные с дочерним процессом, освобождаются.

Функция ***waitpid ()*** приостанавливает выполнение родительского процесса до тех пор, пока дочерний процесс, указанный в параметре ***pid***, не завершит выполнение, или пока не появится сигнал, который либо завершает родительский процесс, либо требует вызвать функцию-обработчик. Если указанный дочерний процесс к моменту вызова функции уже завершился (так называемый ***«зомби»***), то функция немедленно возвращается. Системные ресурсы, связанные с дочерним процессом, освобождаются. Параметр ***pid*** может принимать несколько значений:

pid* < – *1 означает ожидание любого дочернего процесса, чей идентификатор группы процессов равен абсолютному значению ***pid***;

pid* = – *1 означает ожидание любого дочернего процесса (функция ***wait*** ведет себя аналогично);

pid* = *0 означает ожидание любого дочернего процесса, чей идентификатор группы процессов равен идентификатору текущего процесса;

pid* > *0 означает ожидание дочернего процесса, чей идентификатор равен ***pid***.

Значение ***options*** создается путем битовой операции ***ИЛИ*** над следующими константами:

WNOHANG немедленно возвращает управление, если ни один из дочерних процессов не завершил выполнение;

WUNTRACED возвращает управление для остановленных дочерних процессов, о чьем статусе еще не было сообщено.

Каждый дочерний процесс при завершении работы посылает своему процессу-родителю специальный сигнал ***SIGCHLD***, на который у всех процессов по умолчанию установлена реакция «игнорировать сигнал». Наличие такого сигнала совместно с системным вызовом ***waitpid()*** позволяет организовать асинхронный сбор информации о статусе завершившихся порожденных процессов процессом-родителем.

Для перегрузки исполняемой программы можно использовать функции семейства ***exec***:

intexecl(char *pathname, char *arg0, arg1, ...,argn, NULL);
intexecle(char *pathname, char *arg0, arg1, ..., argn, NULL, char **envp);
intexeclp(char *pathname, char *arg0, arg1, ..., argn, NULL);
intexeclpe(char *pathname, char *arg0, arg1, ..., argn, NULL, char **envp);
intexecv(char *pathname, char *argv[]);
intexecve(char *pathname, char *argv[],char **envp);
intexecvp(char *pathname, char *argv[]);

intexecvpe(char *pathname, char *argv[],char **envp);

Основное отличие между разными функциями в семействе состоит в способе передачи параметров. Как видно из рисунка 2, все эти функции выполняют один системный вызов ***execve.***

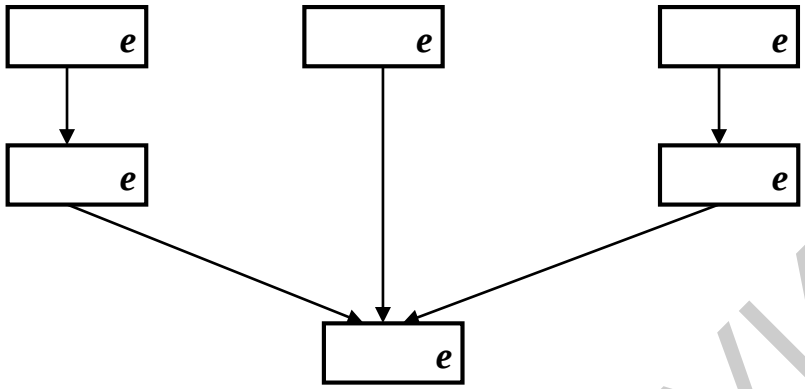

Рисунок 2 – Дерево семейства вызовов ***exec***

Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы.

2. Написать программу, создающую два дочерних процесса с использованием двух вызовов ***fork()***. Родительский и два дочерних процесса должны выводить на экран свой ***pid*** и ***pid*** родительского процесса, а также текущее время в формате: ***часы: минуты: секунды: миллисекунды***. Используя вызов ***system(),*** выполнить команду ***ps -x*** в родительском процессе. Найти свои процессы в списке запущенных процессов.

Варианты индивидуальных заданий

1. Написать программу нахождения массива ***K*** последовательных значений функции ***y[i]=sin(2*PI*i/N),*** где ***i=0, 1, 2…K – 1,*** с использованием ряда Тейлора. Пользователь задает значения ***K, N*** и количество ***n*** членов ряда Тейлора. Для расчета каждого члена ряда Тейлора запускается отдельный поток. Каждый поток выводит на экран свой ***id*** и рассчитанное значение ряда. Главный процесс суммирует все члены ряда Тейлора и полученное значение ***y[i]*** записывает в файл.

2. Написать программу синхронизации двух каталогов, например, ***Dir1*** и ***Dir2***. Пользователь задает имена ***Dir1*** и ***Dir2***. В результате работы программы файлы, имеющиеся в ***Dir1***, но отсутствующие в ***Dir2***, должны скопироваться в ***Dir2*** вместе с правами доступа. Процедуры копирования должны запускаться в отдельном процессе для каждого копируемого файла. Каждый процесс выводит на экран свой ***pid,*** имя копируемого файла и число скопированных байтов. Число

одновременно работающих процессов не должно превышать ***N*** (вводится пользователем).

3. Написать программу поиска одинаковых по их содержимому файлов в двух каталогах, например ***Dir1*** и ***Dir2***. Пользователь задает имена ***Dir1*** и ***Dir2***. В результате работы программы файлы, имеющиеся в ***Dir1***, сравниваются с файлами в ***Dir2*** по их содержимому. Процедуры сравнения должны запускаться в отдельном процессе для каждой пары сравниваемых файлов. Каждый процесс выводит на экран свой ***pid***, имя файла, общее число просмотренных байтов и результаты сравнения. Число одновременно работающих процессов не должно превышать ***N*** (вводится пользователем).

4. Написать программу поиска заданной пользователем комбинации из ***m*** байт (***m< 255***) во всех файлах текущего каталога. Пользователь задает имя каталога. Главный процесс открывает каталог и запускает для каждого файла каталога отдельный процесс поиска заданной комбинации из ***m*** байт. Каждый процесс выводит на экран свой ***pid***, имя файла, общее число просмотренных байтов и результаты поиска. Число одновременно работающих процессов не должно превышать ***N*** (вводится пользователем).

5. Разработать программу «***интерпретатор команд***», которая воспринимает команды, вводимые с клавиатуры (например ***ls -l /bin/bash***), и осуществляет их корректное выполнение. Для этого каждая вводимая команда должна выполняться в отдельном процессе с использованием вызова ***exec()***. Предусмотреть контроль ошибок.

6. Создать дерево процессов по индивидуальному заданию. Каждый процесс постоянно, через время ***t***, выводит на экран следующую информацию: ***номер процесса/потока pidppid, текущее время*** (в миллисекундах). Время ***t =(номер процесса/потока по дереву)*200*** (мс)**.**

7. Написать отчет.

Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы.

2. Организовать функционирование процессов следующей структуры: отец – сын – сын.

3. Обеспечить работу системы так, чтобы процессы определяли свою работу выводом на экран сообщений вида:

- N pidppid текущее время (мс) (***N*** – текущий номер сообщения). «Отец» одновременно посылает сигнал ***SIGUSR1*** «сыновьям». «Сыновья», получив данный сигнал, посылают в ответ «отцу» сигнал ***SIGUSR2***. «Отец», получив сигнал

SIGUSR2, через время ***t=100*** (мс) одновременно посылает сигнал ***SIGUSR1*** «сыновьям». И так далее… Написать функции-обработчики сигналов, которые при получении сигнала выводят соответствующие сообщение о его получении на экран:

N pidppid текущее время (мс) сын такой-то ***get/putsigusrm***.

4. Предусмотреть механизм для определения «отцом», от кого из «сыновей» получен сигнал.

5. Написать программу, создающую дочерний процесс. Родительский процесс создает неименованный канал. Дочерний процесс записывает в канал 100 строк вида: ***номер_строки pid_процесса текущее_время*** (мс). Родительский процесс читает из канала строки и выводит их на экран в следующем виде: pid строка, прочитанная из файла.

Содержание отчета

Отчет по лабораторной работе должен быть оформлен в соответствии с требованиями, предъявляемыми к текстовым документам, и содержать:

- цель работы;
- краткие теоретические сведения, необходимые для выполнения работы;
- описание порядка выполнения работы с приведением листинга кода, скриншотов рабочих окон;
- выводы по работе.

Контрольные вопросы

1. Объясните смысл понятия «процесс».
2. Сколько процессов могут одновременно исполняться системой?
3. Каковы основные операции над процессами?
4. Назовите виды процессов.
5. Назовите виды взаимодействия процессов.
6. Как создаются процессы в ОС Linux?
7. Объясните смысл понятия «дочерний процесс».
8. Объясните смысл понятия «родительский процесс».

Лабораторная работа №3

Взаимодействие процессов в ОС Linux

Цель работы – изучение механизма взаимодействия процессов с использованием сигналов.

Теоретическая часть

Процессы выполняются в раздельных адресных пространствах. Для организации межпроцессного взаимодействия существуют специальные методы:

- общие файлы;
- общая или разделяемая память;
- очереди сообщений (queue);
- каналы (pipe);
- семафоры;
- сигналы (signal).

При использовании общих файлов оба процесса открывают один и тот же файл, с помощью которого и обмениваются информацией.

Использование разделяемой памяти заключается в создании специальной области памяти, позволяющей иметь к ней доступ нескольким процессам.

Очереди сообщений (queue) являются более сложным методом связи взаимодействующих процессов по сравнению с программными каналами. С помощью очередей можно из одного или нескольких процессов независимым образом посылать сообщения некоторому процессу-приемнику. При этом только процесс-приемник может читать и удалять сообщения из очереди, а процессы-клиенты имеют право помещать в очередь свои сообщения. Очередь работает только в одном направлении, если необходима двухсторонняя связь, следует создать две очереди.

Программный канал – это файл особого типа (***FIFO***: «первым вошел – первым вышел»). Процессы могут записывать и считывать данные из канала как из обычного файла. Если канал заполнен, то процесс записи в канал останавливается до тех пор, пока не появится свободное место, чтобы снова заполнить его данными. С другой стороны, если канал пуст, то читающий процесс останавливается до тех пор, пока пишущий процесс не запишет данные в этот канал. В отличие от обычного файла здесь нет возможности позиционирования по файлу с использованием указателя.

Семафор – переменная определенного типа, которая доступна параллельным процессам. В простейшем случае эта переменная может устанавливаться процессом при занятии им определенного ресурса в состояние **0**, при освобождении ресурса переменная устанавливается в состояние **1**. Другой процесс, отслеживая значение переменной, узнает о состоянии ресурса (реально механизм семафоров работает сложнее).

Сигналы не могут непосредственно переносить информацию, что ограничивает их применимость в качестве общего механизма межпроцессного взаимодействия. Тем не менее каждому типу сигналов присвоено мнемоническое имя (например ***SIGINT***), которое указывает на то, для чего используется сигнал этого типа. Имена сигналов определены в стандартном заголовочном файле <***signal.h***> при помощи директивы препроцессора ***#define***. Эти имена соответствуют небольшим положительным целым числам. С точки зрения пользователя получение процессом сигнала выглядит как возникновение прерывания. Процесс прерывает исполнение, и управление передается функции-обработчику сигнала. По окончании обработки сигнала процесс может возобновить регулярное исполнение. Типы сигналов принято задавать специальными символьными константами. Системный вызов ***kill()*** предназначен для передачи сигнала одному или нескольким специфицированным процессам в рамках полномочий пользователя.

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_tpid, int signal);
```

Послать сигнал (не имея полномочий суперпользователя) можно только процессу, у которого эффективный идентификатор пользователя совпадает с эффективным идентификатором пользователя для процесса, посылающего сигнал. В зависимости от значений аргумент ***pid*** указывает процесс, которому посылается сигнал, а аргумент ***sig*** – какой сигнал посылается:

pid > 0 – сигнал посылается процессу с идентификатором ***pid***;

pid = 0 – сигнал посылается всем процессам в группе, которой принадлежит посылающий процесс;

pid = 1 – посылающий процесс не является процессом суперпользователя, сигнал посылается всем процессам в системе, для которых идентификатор пользователя совпадает с эффективным идентификатором пользователя процесса, посылающего сигнал;

pid = – 1 – посылающий процесс является процессом суперпользователя, и тогда сигнал посылается всем процессам в системе, за исключением системных процессов (обычно всем, кроме процессов с ***pid = 0*** и ***pid = 1***);

pid*< *0, но не ***–1***, и тогда сигнал посылается всем процессам из группы, идентификатор которой равен абсолютному значению аргумента ***pid*** (если позволяют привилегии);

sig = 0, и тогда производится проверка на ошибку, а сигнал не посылается. Данное значение можно использовать для проверки правильности аргумента ***pid*** (есть ли в системе процесс или группа процессов с соответствующим идентификатором).

Системные вызовы для установки собственного обработчика сигналов:

#include<signal.h>
void (*signal (intsig, void (*handler) (int)))(int);
intsigaction(int sig, conststructsigaction *act, structsigaction *oldact);

Структура ***sigaction*** имеет следующий формат:

structsigaction {
void (*sa_handler)(int);
void (*sa_sigaction)(int, siginfo_t *, void *);
sigset_tsa_mask;
intsa_flags;
void (*sa_restorer)(void);

Системный вызов ***signal*** служит для изменения реакции процесса на какой-либо сигнал. Параметр ***sig*** – это номер сигнала, обработку которого предстоит изменить. Параметр ***handler*** описывает новый способ обработки сигнала. Это может быть указатель на пользовательскую функцию-обработчик сигнала, специальное значение ***SIG_DFL*** (восстановить реакцию процесса на сигнал ***sig*** по умолчанию) или специальное значение ***SIG_IGN*** (игнорировать поступивший сигнал ***sig***). Системный вызов возвращает указатель на старый способ обработки сигнала, значение которого можно использовать для восстановления старого способа в случае необходимости.

Пример пользовательской обработки сигнала ***SIGUSR1***:

void *my_handler(intnsig) { код функции-обработчика сигнала ***}***
int main() {
(void)signal(SIGUSR1, my_handler); }

Системный вызов ***sigaction*** используется для изменения действий процесса при получении соответствующего сигнала. Параметр ***sig*** задает номер сигнала и может быть равен любому номеру. Если параметр ***act*** не равен нулю, то новое действие, связянное с сигналом ***sig***, устанавливается соответственно ***act***. Если ***oldact*** не равен нулю, то предыдущее действие записывается в ***oldact***.

Большинство типов сигналов ***UNIX*** предназначены для использования ядром, хотя есть несколько сигналов, которые посылаются от процесса к процессу:

SIGALRM – сигнал таймера (***alarmclock***). Посылается процессу ядром при срабатывании таймера. Каждый процесс может устанавливать не менее трех таймеров. Первый из них измеряет прошедшее реальное время. Этот таймер устанавливается самим процессом при помощи системного вызова ***alarm().***

SIGCHLD – сигнал останова или завершения дочернего процесса (***childprocessterminatedorstopped***). Если дочерний процесс останавливается или завершается, то ядро сообщит об этом родительскому процессу, послав ему данный сигнал. По умолчанию родительский процесс игнорирует этот сигнал, поэтому, если в родительском процессе необходимо получать сведения о завершении дочерних процессов, то нужно перехватывать этот сигнал.

SIGHUP – сигнал освобождения линии (***hangupsignal***). Посылается ядром всем процессам, подключенным к управляющему терминалу (***controlterminal***) при отключении терминала. Он также посылается всем членам сеанса, если завершает работу лидер сеанса (обычно процесс командного интерпретатора), связанного с управляющим терминалом.

SIGINT – сигнал прерывания программы (***interrupt***). Посылается ядром всем процессам сеанса, связанного с терминалом, когда пользователь нажимает клавишу прерывания. Это также обычный способ остановки выполняющейся программы.

SIGKILL – сигнал удаления процесса (***kill***). Это довольно специфический сигнал, который посылается от одного процесса к другому и приводит к немедленному прекращению работы получающего сигнал процесса.

SIGPIPE – сигнал о попытке записи в канал или сокет, для которых принимающий процесс уже завершился.

SIGPOLL – сигнал о возникновении одного из опрашиваемых событий (***pollableevent***). Этот сигнал генерируется ядром, когда некоторый открытый дескриптор файла становится готовым для ввода или вывода.

SIGPROF – сигнал профилирующего таймера (***profilingtimeexpired***). Как было упомянуто для сигнала ***SIGALRM***, любой процесс может установить не менее трех таймеров. Второй из этих таймеров может использоваться для измерения времени выполнения процесса в пользовательском и системном режимах. Сигнал SIGPROF генерируется, когда истекает время, установленное в этом таймере, и поэтому может быть использован средством профилирования программы.

SIGQUIT – сигнал о выходе (***quit***). Как и сигнал SIGINT, этот сигнал посылается ядром, когда пользователь нажимает клавишу выхода используемого терминала. В отличие от SIGINT этот сигнал приводит к аварийному завершению и сбросу образа памяти.

SIGSTOP – сигнал останова (***stop executing***). Это сигнал управления заданиями, который останавливает процесс. Его, как и сигнал SIGKILL, нельзя проигнорировать или перехватить.

SIGTERM – программный сигнал завершения (***softwareterminationsignal***). Программист может использовать этот сигнал для того, чтобы дать процессу время для «наведения порядка», прежде чем посылать ему сигнал SIGKILL.

SIGTRAP – сигнал трассировочного прерывания (***tracetrap***). Это особый сигнал, который в сочетании с системным вызовом ***ptrace*** используется отладчиками, такими как ***sdb, adb, gdb***.

SIGTSTP – терминальный сигнал остановки (***terminalstopsignal***). Он формируется при нажатии специальной клавиши останова.

SIGTTIN – сигнал о попытке ввода с терминала фоновым процессом (***backgroundprocessattemptingread***). Если процесс выполняется в фоновом режиме и пытается выполнить чтение с управляющего терминала, то ему посылается этот сигнал. Действие сигнала по умолчанию – остановка процесса.

SIGTTOU – сигнал о попытке вывода на терминал фоновым процессом (***backgroundprocessattemptingwrite***). Аналогичен сигналу SIGTTIN, но генерируется, если фоновый процесс пытается выполнить запись в управляющий терминал. Действие сигнала по умолчанию – остановка процесса.

SIGURG – сигнал о поступлении в буфер сокета срочных данных (***highbandwidthdataisavailableatasocket***). Он сообщает процессу, что по сетевому соединению получены срочные внеочередные данные.

SIGUSR1 и ***SIGUSR2*** – пользовательские сигналы (***user defined signals 1 and 2***). Так же как и сигнал SIGTERM, эти сигналы никогда не посылаются ядром и могут использоваться для любых целей по выбору пользователя.

SIGVTALRM – сигнал виртуального таймера (***virtualtimerexpired***). Третий таймер можно установить так, чтобы он измерял время, которое процесс выполняет в пользовательском режиме.

Наборы сигналов формуются при помощи типа ***sigset_t***, который определен в заголовочном файле <***signal.h***>. Выбрать определенные сигналы можно, начав либо с полного набора сигналов и удалив ненужные сигналы, либо с пустого набора, включив в него нужные. Инициализация пустого и полного набора сигналов выполняется при помощи процедур ***sigemptyset*** и ***sigfillset*** соответственно.

После инициализации с наборами сигналов можно оперировать при помощи процедур ***sigaddset*** и ***sigdelset***, соответственно добавляющих и удаляющих указанные вами сигналы.

Описание данных процедур:

#include<signal.h>

/* Инициализация*/

intsigemptyset (sigset_t *set);
intsigfillset (sigset_t *set);

/*Добавление и удаление сигналов*/

intsigaddset (sigset_t *set, int signo);
intsigdelset (sigset_t *set, int signo);

Процедуры ***sigemptyset*** и ***sigfillset*** имеют единственный параметр – указатель на переменную типа ***sigset_t***. Вызов ***sigemptyset*** инициализирует набор ***set***, исключив из него все сигналы. И наоборот, вызов ***sigfillset*** инициализирует набор, на который указывает ***set***, включив в него все сигналы. Приложения должны вызывать ***sigemptyset*** или ***sigfillset*** хотя бы один раз для каждой переменной типа ***sigset_t***.

Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы.
2. Организовать функционирование процессов в соответствии со структурой, изображенной на рисунке 3.

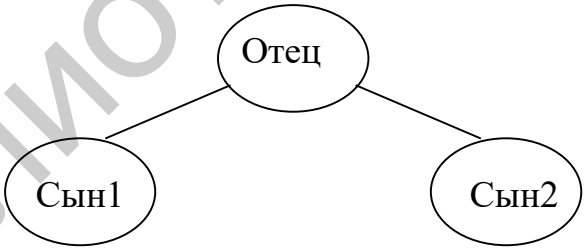

Рисунок 3 – Структура процесса

Процессы определяют свою работу выводом на экран сообщений следующего вида:

N pid ppid текущее время (мс) (***N*** – текущий номер сообщения). «Отец» одновременно посылает сигнал ***SIGUSR1*** «сыновьям». «Сыновья», получив данный сигнал, посылают в ответ «отцу» сигнал ***SIGUSR2***. «Отец», получив сигнал ***SIGUSR2***, через время ***t=100*** (мс) одновременно посылает сигнал ***SIGUSR1*** «сыновьям». И так далее… Написать функции-обработчики сигналов, которые при получении сигнала выводят сообщение о получении сигнала на экран. При получении/посылке сигнала они выводят соответствующее сообщение:

N pid ppid текущее время (мс) ***сын такой-то get/put SIGUSRm***.

Предусмотреть механизм для определения «Отцом», от кого из «Сыновей» получен сигнал.

3. Написать отчет.

Варианты индивидуальных заданий

1. Организовать функционирование процессов следующей структуры:

1-2-3-4 (процесс 1 создает процесс 2, процесс 2 создает процесс 3, процесс 3 создает процесс 4). Далее организовать передачу/прием сигналов в следующей последовательности: ***1-2 (SIGUSR1), 2-3 (SIGUSR1), 3-4 (SIGUSR1), 4-1 (SIGUSR2), 1-2 (SIGUSR2), 2-3 (SIGUSR2), 3-4 (SIGUSR2), 4-1 (SIGUSR1)*** и т. д. Каждый процесс выдерживает паузу ***t=100*** (мс) между приемом и посылкой сигнала и выводит на консоль следующую информацию:

N pid ppid текущее время (мс) ***процесс_такой-то get/put SIGUSRm***.

2. Аналогично пункту 1, но для процессов написать функции-обработчики сигналов от клавиатуры, которые запрашивали бы подтверждение на завершение работы при получении такого сигнала.

3. Организовать функционирование процессов следующей структуры:

1-(2,3), 3-4 (процесс 1 создает процессы 2 и 3, процесс 3 создает процесс 4). Далее организовать передачу/прием сигналов в следующей последовательности: ***1-(2,3) (SIGUSR1), 3-4 (SIGUSR1), 4-1 (SIGUSR2), 1-(2,3) (SIGUSR2), 3-4 (SIGUSR2), 4-1 (SIGUSR1)*** и т. д. Каждый процесс выдерживает паузу ***t=100*** (мс) между приемом и посылкой сигнала и выводит на консоль следующую информацию:

N pid ppid текущее время (мс) ***процесс_такой-то get/put SIGUSRm***.

Содержание отчета

Отчет по лабораторной работе должен быть оформлен в соответствии с требованиями, предъявляемыми к текстовым документам, и содержать:

- цель работы;
- краткие теоретические сведения, необходимые для выполнения работы;
- описание порядка выполнения работы с приведением листинга кода, скриншотов рабочих окон;
- выводы по работе.

Контрольные вопросы

1. Какую функцию выполняют сигналы?
2. Что такое мнемоническое имя?
3. Какие типы сигналов вы знаете?

Лабораторная работа №4

Семафоры в ОС Linux

Цель работы – изучение механизма взаимодействия процессов с использованием семафоров.

Теоретическая часть

Семафор – переменная определенного типа, которая доступна параллельным процессам для проведения над ней только трех операций:

1) A(S, n) – увеличить значение семафора ***S*** на величину ***n***;

2) D(S, n) – если значение семафора ***S < n***, процесс блокируется. Далее ***S = = S – n***;

3) Z(S) – процесс блокируется до тех пор, пока значение семафора ***S*** не станет равным 0.

Семафор играет роль вспомогательного критического ресурса, т. к. операции ***A*** и ***D*** неделимы при своем выполнении и взаимно исключают друг друга. Семафорный механизм работает по схеме, в которой сначала исследуется состояние критического ресурса, а затем уже осуществляется допуск к критическому ресурсу или отказ от него на некоторое время. Основным достоинством семафорных операций является отсутствие состояния «активного ожидания», что может существенно повысить эффективность работы мультипрограммной вычислительной системы.

Для работы с семафорами имеются системные вызовы для создния набора семафоров и получение доступа к нему, для изменения значений семафоров и для выполнения управляющих операций над набором семафоров.

Создание набора семафоров и получение доступа к нему:

intsemget(key_t key, int nsems, int semflg);

Параметр ***key*** является ключом для массива семафоров, т. е. фактически его именем. В качестве значения этого параметра может использоваться значение ключа, полученное с помощью функции ***ftok()***, или специальное значение ***IPC_PRIVATE***. Использование значения ***IPC_PRIVATE*** всегда приводит к попытке создания нового массива семафоров с ключом, который не совпадает со значением ключа ни одного из уже существующих массивов и не может быть получен с помощью функции ***ftok()*** ни при одной комбинации ее параметров.

Параметр ***nsems*** определяет количество семафоров в создаваемом или уже существующем массиве. В случае если массив с указанным ключом уже имеется, но его размер не совпадает с указанным в параметре ***nsems***, констатируется возникновение ошибки.

Параметр ***semflg*** – флаги – играет роль только при создании нового массива семафоров и определяет права различных пользователей при доступе к массиву, а также необходимость создания нового массива и поведение системного вызова при попытке создания. Он является некоторой комбинацией (с помощью операции: побитовое или – «|») следующих предопределенных значений и восьмеричных прав доступа:

IPC_CREAT – если массива для указанного ключа не существует, он должен быть создан;

IPC_EXCL – применяется совместно с флагом ***IPC_CREAT***; при совместном их использовании и существовании массива с указанным ключом доступ к массиву не производится и констатируется ошибка, при этом переменная ***errno***, описанная в файле <***errno.h***>, примет значение ***EEXIST***;

0400 – разрешено чтение для пользователя, создавшего массив;

0200 – разрешена запись для пользователя, создавшего массив;

0040 – разрешено чтение для группы пользователя, создавшего массив;

0020 – разрешена запись для группы пользователя, создавшего массив;

0004 – разрешено чтение для всех остальных пользователей;

0002 – разрешена запись для всех остальных пользователей.

Пример: ***semflg= IPC_CREAT|0022***

Изменение значений семафоров:

intsemop(int semid, structsembuf *sops, int nsops);

Параметр ***semid*** является дескриптором System V IPC для набора семафоров, т. е. значением, которое вернул системный вызов ***semget()*** при создании набора семафоров или при его поиске по ключу. Каждый из ***nsops*** элементов массива, на который указывает параметр ***sops***, определяет операцию, которая должна быть совершена над каким-либо семафором из массива IPC семафоров, и имеет тип структуры:

structsembuf{
shortsem_num; //номер семафора в массиве IPC семафоров (начиная с 0);
shortsem_op;//выполняемая операция;
shortsem_flg; // флаги для выполнения операции.
}

Значение элемента структуры ***sem_op*** определяется следующим образом:

- для выполнения операции ***A(S,n)*** значение должно быть равно ***n***;
- для выполнения операции ***D(S,n)*** значение должно быть равно ***–n***;
- для выполнения операции ***Z(S)*** значение должно быть равно **0**.

Семантика системного вызова подразумевает, что все операции будут в реальности выполнены над семафорами только перед успешным возвращением из системного вызова. Если при выполнении операции ***D*** или ***Z*** процесс перешел в состояние ожидания, то он может быть выведен из этого состояния при возникновении следующих форс-мажорных ситуаций: массив семафоров был удален из системы; процесс получил сигнал, который должен быть обработан.

Выполнение разнообразных управляющих операций (включая удаление) над набором семафоров:

intsemctl(intsemid, intsemnum, intcmd, unionsemunarg);

Изначально все семафоры инициируются нулевым значением.

Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы.
2. Написать программу, создающую дочерний процесс. Родительский процесс создает семафор (***сем1***) и общий файл. Дочерний процесс записывает в файл по одной строке всего ***100*** строк вида

номер_строкиpid_процессатекущее_время (мс).

Родительский процесс читает из файла строки и выводит их на экран в следующем виде: ***pidстрока_прочитанная_из_файла***. Семафор ***сем1*** используется процессами для разрешения одному из процессов получить доступ к файлу.

3. Написать отчет.

Варианты индивидуальных заданий

1. Организовать функционирование процессов следующей структуры:

1-2-3-4 (процесс 1 создает процесс 2, процесс 2 создает процесс 3, процесс 3 создает процесс 4). Далее организовать с использованием общего файла передачу/прием следующей информации в последовательности: ***1-2, 2-3, 3-4, 4-1, 1-2, 2-3, 3-4, 4-1*** и т. д. Каждый процесс выдерживает паузу ***t=100*** (мс) между приемом и посылкой информации. Каждый процесс читает из файла переданные ему строки, выводит их на консоль, затем добавляет свои ***M*** (***M*** – номер процесса) строк вида ***M pid ppid текущее время*** мс ***процесс_такой-то*** и передает их далее. Для синхронизации работы использовать семафоры.

2. Организовать функционирование процессов следующей структуры:

1-(2,3), 3-4 (процесс 1 создает процессы 2 и 3, процесс 3 создает процесс 4). Далее организовать с использованием общего файла передачу/прием следующей информации в последовательности: ***1-2, 2-3, 3-4, 4-1, 1-2, 2-3, 3-4, 4-1*** и т. д. Каждый процесс читает из файла переданные ему строки, выводит их на консоль,

затем добавляет свои ***M*** (***M*** – номер процесса) строк вида ***M pid ppid текущее время*** (мс) ***процесс_такой-то*** и передает их далее. Для синхронизации работы использовать семафоры.

Содержание отчета

Отчет по лабораторной работе должен быть оформлен в соответствии с требованиями, предъявляемыми к текстовым документам, и содержать:

- цель работы;
- краткие теоретические сведения, необходимые для выполнения работы;
- описание порядка выполнения работы с приведением листинга кода, скриншотов рабочих окон;
- выводы по работе.

Контрольные вопросы

1. Что такое семафор?
2. Какие функции выполняет семафор?
3. Какие параметры семафоров вы знаете?

Лабораторная работа №5

Управление потоками в ОС Linux

Цель работы – изучение потоков в ОС Linux.

Теоретическая часть

Существует расширенная реализация понятия ***процесс***, когда ***процесс*** представляет собой совокупность выделенных ему ресурсов и набора ***нитей исполнения***. ***Нити (threads)***, или потоки процесса, разделяют его программный код, глобальные переменные и системные ресурсы, но каждая ***нить*** имеет собственный программный счетчик, свое содержимое регистров и свой стек. Все глобальные переменные доступны в любой из дочерних нитей. Каждая нить исполнения имеет в системе уникальный номер – идентификатор ***нити***. 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. Для создания нитей используется функция ***pthread_create***:

#include<pthread.h>

intpthread_create(pthread_t*thread, constpthread_attr_t *attr,
void *(*start_routine)(void*),void *arg);

Функция создает новую нить, в которой выполняется функция пользователя ***start_routine***, передавая ей в качестве аргумента параметр ***arg***. Если требуется передать более одного параметра, они собираются в структуру, и передается адрес этой структуры. При удачном вызове функция ***pthread_create*** возвращает значение ***0*** и помещает идентификатор новой нити исполнения по адресу, на который указывает параметр ***thread***. В случае ошибки возвращается положительное значение, которое определяет код ошибки, описанный в файле ***<errno.h>***. Значение системной переменной ***errno*** при этом не устанавливается. Параметр ***attr*** служит для задания различных атрибутов создаваемой нити. Функция нити должна иметь заголовок вида

void * start_routine (void *)

Завершение функции потока происходит, если:

- функция нити вызвала функцию ***pthread_exit()***;
- функция нити достигла точки выхода;
- нить была досрочно завершена другой нитью.

Функция ***pthread_join()*** используется для перевода нити в состояние ожидания:

#include <pthread.h>

intpthread_join (pthread_t thread, void **status_addr);

Функция ***pthread_join()*** блокирует работу вызвавшей ее нити исполнения до завершения нити с идентификатором ***thread***. После разблокирования в указатель, расположенный по адресу ***status_addr***, заносится адрес, который вернул завершившийся ***thread*** либо при выходе из ассоциированной с ним функции, либо при выполнении функции ***pthread_exit()***. Если нас не интересует, что вернула нам нить исполнения, в качестве этого параметра можно использовать значение ***NULL***.

Для компиляции программы с нитями необходимо подключить библиотеку ***pthread.lib*** следующим способом:

gcc1.c–o 1.exe -lpthread

Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы.

2. Написать программу, создающую два дочерних потока. Родительский процесс и два дочерних потока должны выводить на экран свой ***id*** и ***pid*** родительского процесса и текущее время в формате: ***часы: минуты: секунды: миллисекунды***.

3. Написать отчет.

Варианты индивидуальных заданий

1. Написать программу нахождения массива ***K*** последовательных значений функции ***y[i]=sin(2*PI*i/N), где i=0, 1, 2…K–1*** с использованием ряда Тейлора. Пользователь задает значения ***K***, ***N*** и количество ***n*** членов ряда Тейлора. Для расчета каждого члена ряда Тейлора запускается отдельный поток. Каждый поток выводит на экран свой ***id*** и рассчитанное значение ряда. Головной процесс ожидает завершения работы всех потоков и суммирует все члены ряда Тейлора. Полученное значение ***y[i]*** записывается в файл результата.

2. Написать программу синхронизации двух каталогов, например ***Dir1*** и ***Dir2***. Пользователь задает имена ***Dir1*** и ***Dir2***. В результате работы программы файлы, имеющиеся в ***Dir1***, но отсутствующие в ***Dir2***, должны скопироваться в ***Dir2*** вместе с правами доступа. Процедуры копирования должны запускаться в отдельном потоке для каждого копируемого файла. Каждый поток выводит на экран ***id,*** имя копируемого файла и число скопированных байтов. Число одновременно работающих потоков не должно превышать ***N*** (вводится пользователем).

3. Написать программу поиска одинаковых по их содержимому файлов в двух каталогах, например ***Dir1*** и ***Dir2***. Пользователь задает имена ***Dir1*** и ***Dir2***. В

результате работы программы файлы, имеющиеся в ***Dir1***, сравниваются с файлами в ***Dir2*** по их содержимому. Процедуры сравнения должны запускаться в отдельном потоке для каждой пары сравниваемых файлов. Каждый поток выводит на экран ***id***, имя файла, общее число просмотренных байтов и результаты сравнения. Число одновременно работающих потоков не должно превышать ***N*** (вводится пользователем).

4. Написать программу поиска заданной пользователем комбинации из ***m*** байт (***m< 255***) во всех файлах текущего каталога. Пользователь задает имя каталога. Главный процесс открывает каталог и запускает для каждого файла каталога отдельный поток поиска заданной комбинации из ***m*** байт. Каждый поток выводит на экран свой ***id***, имя файла, общее число просмотренных байтов и результаты поиска. Число одновременно работающих потоков не должно превышать ***N*** (вводится пользователем).

5. Создать дерево потоков по индивидуальному заданию. Каждый поток постоянно, через время ***t***, выводит на экран следующую информацию: ***номер процесса/потока idppid текущее время*** (мс). Время ***t =(номер процесса/потока по дереву)·200*** (мс).

Содержание отчета

Отчет по лабораторной работе должен быть оформлен в соответствии с требованиями, предъявляемыми к текстовым документам, и содержать:

- цель работы;
- краткие теоретические сведения, необходимые для выполнения работы;
- описание порядка выполнения работы с приведением листинга кода, скриншотов рабочих окон;
- выводы по работе.

Контрольные вопросы

1. Дайте определение понятию «нити исполнения».
2. Что собой представляет поток?
3. Назовите функции для работы с потоками.

Лабораторная работа №6

Разработка многопоточных приложений в ОС Windows

Цель работы – программная реализация многопоточных приложений в операционной системе Windows.

Теоретическая часть

В современных ОС пользователям предлагается несколько типов параллельной работы, основными из которых являются процессы и потоки. Процессы – это программы на этапе выполнения. Потоки – это минимальная единица работы. Однако с точки зрения распределения ресурсов именно потоки являются главными, поскольку им, а не процессам предоставляется на определенное время центральный процессор для выполнения какой-либо работы [7]. Рассмотрим несколько функций ***WinAPI***, выполняющих некоторые операции над потоками.

Функция ***CreateThread*** создает поток для выполнения внутри адресного пространства вызывающего процесса.

```
HANDLE CreateThread (
  PSECURITY_ATTRIBUTES lpThreadAttributes,
    // атрибуты защиты потока
  DWORD dwStackSize,        // начальный размер стека потока, в байтах
  PTHREAD_START_ROUTINE lpStartAddress,
    // указатель на функцию потока
  PVOID lpParameter,        // параметр для нового потока
  DWORD dwCreationFlags,    // флаги создания потока
  PDWORD lpThreadId
    //указатель на возвращаемый идентификатор потока
);
```

При успешном завершении функции возвращается дескриптор нового потока. При неудачном завершении функции возвращается *NULL*.

Параметры функции:

LpThreadAttributes – указатель на структуру *SECURITY_ATTRIBUTES*, которая определяет, может ли возвращенный дескриптор быть унаследован дочерними процессами. Если ***lpThreadAttributes NULL***, то дескриптор не может быть унаследован.

DwStackSize – определяет размер (в байтах) стека для нового потока. Если определен 0, то размер стека по умолчанию равен размеру стека порождающего потока. Стек распределяется автоматически в пространстве памяти процесса и освобождается при завершении потока.

LpStartAddress – начальный адрес нового потока. Это обычно адрес функции, объявленной с соглашением о вызовах *WINAPI*, которое принимает одиночный 32-разрядный указатель как параметр и возвращает 32-разрядный код завершения. Прототип: `DWORD WINAPI ThreadFunc (LPVOID);`

LpParameter – определяет единственное 32-разрядное значение параметра, передаваемое потоку.

DwCreationFlags – определяет дополнительные флажки, которые управляют созданием потока. Если флажок определен *CREATE_SUSPENDED*, то поток создается в состоянии ожидания и не будет выполняться, пока не будет вызвана функция ***ResumeThread.*** Если это значение нуль, то поток выполняется немедленно после создания.

LpThreadId – указатель на 32-разрядную переменную, которая получает значение идентификатора потока [7].

Поток можно завершить принудительно с помощью вызова следующих функций *WinAPI*:

```
VOID ExitThread (
DWORD ExitCode);       // код завершения потока
```

и

```
VOID TerminateThread (
  HANDLE hThread,      // поток, который требуется завершить
  DWORD ExitCode       // код завершения потока
);
```

Пример фрагмента программы, которая вычисляет произведение всех чисел от 1 до 100.

```
. . .
// Функция потока
DWORD WINAPI ThreadFuction (PVOID Parametr)
{ int proizv = 1;                    // результат произведения
  int ii, *kk;
  kk = (int *) Parametr;
for (ii = *k; ii < (*kk) + 50; ii ++) proizv *= ii;
return proizv;
}
...
// код вызовов функций для создания потоков
DWORD idThread;
int k1 = 1; k2 = 51;
HANDLE h1, h2;
// Создается два потока в приостановленном состоянии
```

h1= CreateThread (NULL, 0, ThreadFunction, &k1, CREATE_SUSPENDED, &idThread);

h2= CreateThread (NULL, 0, ThreadFunction, &k2, CREATE_SUSPENDED, &idThread);

// Выполнение потоков

ResumeThread (h1);

ResumeThread (h2);

...

В данном коде встретилась весьма полезная *WinAPI* функция ***ResumeThread,*** которая возобновляет выполнение приостановленного потока и описывется как

```
DWORD ResumeThread (
  HANDLE hThread        // поток, который требуется возобновить
);
```

Если вызов этой функции успешен, то возвращается предыдущее значение счетчика простоев данного потока, в противном случае – 0*xFFFFFFFF*.

Выполнение отдельного потока можно приостанавливать несколько раз (точно такое же число раз он должен возобновляться), а производится это вызовом функции ***SuspendThread***, описанной как

```
DWORD SuspendThread (
  HANDLE hThread        // поток, который требуется приостановить
);
```

Поток может сообщить ОС, чтобы она не выделяла ему процессор определенное время, указанное в миллисекундах.

```
VOID Sleep (DWORD MilliSeconds);
```

В ОС Windows есть также функция, которая находит и открывает поток по идентификатору:

```
HANDLE OpenThread (
DWORD DesiredAccess,
BOOL InheritHandle,
DWORD dwThreadId
);
```

Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы.
2. Получить у преподавателя собственный вариант задания, который предусматривает распараллеливание работы на несколько потоков.
3. Используя изученные механизмы, разработать программу, реализующую полученное задание.
4. Написать отчет.

Варианты заданий

Вариант №1

Разработать программу, которая вычисляет сумму и произведение чисел от L до U, где L – это нижняя граница диапазона, U – верхняя граница диапазона. Вычисление суммы и произведения оформить как две функции потока. Значения границ диапазон вводятся пользователем, затем запускаются два требуемых потока, а потом на экран выводятся полученные значения.

Вариант №2

Разработать программу, которая вычисляет число Фибоначчи по номеру, введенному пользователем, и формуле $F_i = F_{i-1} + F_{i-2}, F_0 = F_1 = 1$. Вычисление числа Фибоначчи оформить как функцию потока. По завершении функции потока программа выводит число на экран.

Вариант №3

Разработать программу для перевода целого числа со знаком в его строковый эквивалент прописью. Перевод числа оформить как функцию потока. Ввод числа происходит до запуска потока, а вывод строки – по его завершении. Например, ввод «–1211» должен приводить к выводу «минус тысяча двести одиннадцать».

Вариант №4

Разработать программу для перевода знакового числа с плавающей точкой в его строковый эквивалент прописью. Перевод числа оформить как функцию потока. Ввод числа происходит до запуска потока, а вывод строки – по его завершении. Например, ввод «–12.11» должен приводить к выводу «минус двенадцать целых одиннадцать сотых».

Вариант №5

Разработать программу, осуществляющую ввод двух строк, введенных пользователем. Далее, если обе строки хранят целые числа со знаком, то на экран выводится сумма чисел, в противном случае – конкатенация двух введенных строк. Проверку на соответствие строки целому числу, вычисление суммы чисел и конкатенацию строк оформить как три разные функции потока. Ввод строк осуществляется до запуска всех потоков, а вывод результатов – после их завершения.

Вариант №6

Разработать программу, вычисляющую сумму и произведение двух матриц. Выполнение этих операций оформить как две функции потока. Сначала программа осуществляет ввод элементов матриц, далее запускает оба потока, а затем выводит результаты на экран.

Вариант №7

Разработать программу для упорядочивания одномерного целочисленного массива. Сортировка массива по возрастанию должна осуществляться любым из так называемых «улучшенных алгоритмов» сортировки и оформляется как функция потока. Сначала выполняется ввод элементов матрицы, затем запускается поток и далее – вывод упорядоченного массива.

Вариант №8

Разработать программу для упорядочивания одномерного массива чисел с плавающей точкой. Сортировка массива по возрастанию должна осуществляться любым из так называемых «улучшенных алгоритмов» сортировки и оформляется как функция потока. Сначала выполняется ввод элементов матрицы, затем запускается поток и далее – вывод упорядоченного массива.

Вариант №9

Разработать программу для упорядочивания одномерного массива строк. Сортировка массива по возрастанию должна осуществляться любым из так называемых «улучшенных алгоритмов» сортировки и оформляется как функция потока. Сначала выполняется ввод элементов матрицы, затем запускается поток и далее – вывод упорядоченного массива.

Вариант №10

Разработать программу для вычисления суммы элементов, лежащих на главной и побочной диагоналях квадратной матрицы. Выполнение этой операции оформляется как функция потока. Ввод элементов матрицы осуществляется до запуска потока, а вывод полученного значения – по его завершении.

Вариант №11

Разработать программу для вычисления суммы элементов, не лежащих на главной и побочной диагоналях квадратной матрицы. Выполнение этой операции оформляется как функция потока. Ввод элементов матрицы осуществляется до запуска потока, а вывод полученного значения – по его завершении.

Вариант №12

Разработать программу для вычисления полного количества дней, прошедших между двумя датами. Даты – это строки вида ЦЦ.ЦЦ.ЦЦЦЦ, где Ц – это любая цифра из диапазона 0–9. Вычисление разницы между датами оформляется как функция потока. Сначала осуществляется ввод дат, затем запускается поток, и далее результат выводится на экран.

Вариант №13

Разработать программу для вычисления полного количества секунд, прошедших между двумя значениями времени. Значение времени – это строки вида ЦЦ.ЦЦ.ЦЦ, где Ц – это любая цифра из диапазона 0–9. Вычисление разницы между временами оформляется как функция потока. Сначала осуществляется ввод значений времени, затем запускается поток, и далее результат выводится на экран.

Вариант №14

Разработать программу для поиска вхождения подстроки в строку. Эта операция оформляется как функция потока и реализует любой из известных методов поиска подстроки, кроме прямого. Сначала осуществляется ввод двух строк, затем запускается поток, и далее выводятся результаты: значение индекса элемента первой строки, с которого началось совпадение, или «–1» в противном случае.

Вариант №15

Разработать программу для подсчета количества вхождений подстроки в строку. Эта операция оформляется как функция потока и реализует любой из известных методов поиска подстроки, кроме прямого. Сначала осуществляется ввод двух строк, затем запускается поток, и далее выводится результат – целое число.

Вариант №16

Разработать программу для получения строкового эквивалента даты прописью. Дата – это строка вида ЦЦ.ЦЦ.ЦЦЦЦ, где Ц – это любая цифра из диапазона 0–9. Получение строкового эквивалента оформляется как функция потока. Сначала осуществляется ввод даты, затем запускается поток, и далее результат выводится на экран: число и месяц прописью, а за последними четырьмя – слово «года» (например, ввод «29.02.2008» приводит к выводу «Двадцать девятое февраля 2008 года»).

Вариант №17

Разработать программу для получения строкового эквивалента значения времени прописью. Время – это строка вида ЦЦ.ЦЦ.ЦЦ, где Ц – это любая цифра из диапазона 0–9. Получение строкового эквивалента оформляется как функция потока. Сначала осуществляется ввод значения времени, затем запускается поток, и далее результат выводится на экран: значение часов, минут и секунд прописью (например, ввод «12.01.20» приводит к выводу «двенадцать часов одна минута двадцать секунд»).

Вариант №18

Разработать программу, осуществляющую инвертирование битовой строки, а также ее перевод в десятичное число. Битовая строка – это строка, состоящая из нулей и единиц. Инвертирование битовой строки и перевод строки в десятичное число оформляются как две функции потока. Сначала осуществляется ввод битовой строки, затем запускаются два потока, и далее выводятся результаты.

Вариант №19

Разработать программу, осуществляющую реверс битовой строки, а также ее перевод в десятичное число. Битовая строка – это строка, состоящая из нулей и единиц. Реверс битовой строки (все нули заменяются на единицы, а единицы на нули) и перевод строки в десятичное число оформляются как две функции потока. Сначала осуществляется ввод битовой строки, затем запускаются два потока, и далее выводятся результаты.

Вариант №20

Разработать программу, осуществляющую вычисление факториала числа. Выполнение данной операции (по формуле $N! = N \cdot (N-1)!$, где $0!=1$) оформляется как функция потока. Сначала осуществляется ввод числа, затем запускается поток, и далее результат выводится на экран.

Вариант №21

Разработать программу, вычисляющую сумму крайних элементов квадратной матрицы. Выполнение данной операции оформляется как функция потока. Сначала осуществляется ввод элементов матрицы, затем запускается поток, и далее результат выводится на экран.

Вариант №22

Разработать программу, осуществляющую поиск элемента по ключу в целочисленном векторе любым известным методом, кроме прямого. Выполнение данной операции оформляется как функция потока. Сначала осуществляется ввод элементов вектора и значение ключа поиска, затем запускается поток, и далее результат выводится на экран: значение индекса найденного элемента или «–1» в противном случае.

Вариант №23

Разработать программу для замены всех латинских букв в строке на их аналоги из кириллицы. Выполнение данной операции оформляется как функция потока. Сначала осуществляется ввод строки, затем запускается поток, и далее измененная строка выводится на экран.

Вариант №24

Разработать программу для смены регистра всех символов в строке. Выполнение данной операции оформляется как функция потока. Сначала осуществляется ввод строки, затем запускается поток, и далее измененная строка выводится на экран.

Вариант №25

Разработать программу для проверки того факта, что беззнаковое целое число является степенью двойки. Выполнение данной операции оформляется как функция потока. Сначала осуществляется ввод числа, затем запускается поток, и далее на экран выводится результат: если число является степенью двойки, то выводится показатель степени, в противном случае выводится сообщение «не является степенью двойки».

Содержание отчета

Отчет по лабораторной работе должен быть оформлен в соответствии с требованиями, предъявляемыми к текстовым документам, и содержать:

- цель работы;
- краткие теоретические сведения, необходимые для выполнения работы;
- описание порядка выполнения работы с приведением листинга кода, скриншотов рабочих окон;
- выводы по работе.

Лабораторная работа №7

Управление приоритетами потоков в ОС Windows

Цель – определение и изменение приоритетов потоков в операционной системе Windows.

Теоретическая часть

В основе многих алгоритмов планирования процессов лежит концепция приоритетного обслуживания. У потоков изначально известна характеристика – приоритет, на основании которого определяется порядок выполнения потоков. Приоритет – это число, характеризующее степень привилегированности потока при использовании ресурсов вычислительной машины, в том числе (и в первую очередь) времени процессора: чем выше приоритет, тем выше привилегии, а значит, поток меньше времени будет проводить в очередях. Приоритет может выражаться числом. В большинстве ОС, поддерживающих потоки, их приоритет непосредственно связан с приоритетом процесса, в рамках которого выполняется данный поток. Значение приоритета из описателя процесса используется при назначении приоритета потокам этого процесса.

Во многих ОС предусматривается возможность изменения приоритетов в течение жизни потока. Изменения приоритета могут происходить по инициативе самого потока, когда он обращается с соответствующим вызовом к ОС, или по инициативе пользователя, когда он выполняет соответствующую команду. Кроме того, ОС сама может изменять приоритеты потоков в зависимости от ситуации, складывающейся в системе. В последнем случае приоритеты называются динамическими, в отличие от неизменяемых, фиксированных приоритетов [7].

В ОС Windows процессу можно задать класс приоритета. Поддерживается шесть классов приоритетов: простаивающий (***IDLE_PRIORITY_CLASS***), когда потоки выполняются, если система не занята другой работой; ниже нормального (***BELOW_NORMAL_PRIORITY_CLASS***); нормальный (***NORMAL_PRIORITY_CLASS***), когда у потоков нет особых требований; выше нормального (***ABOVE_NORMAL_PRIORITY_CLASS***); высокий (***HIGH_PRIORITY_CLASS***), где потоки немедленно реагируют на события; реального времени (***REALTIME_PRIORITY_CLASS***), где потоки тоже немедленно реагируют на события и могут вытеснять даже потоки ОС.

Класс приоритета устанавливается с помощью функции

```
BOOL SetPriorityClass (
  HANDLE Process,
  DWORD Priority);
```

Например, процесс может поменять свой собственный класс приоритета
SetProrityClass (GetCurrentProcess (),HIGH_PRIORITY_CLASS);

Чтобы узнать класс приоритета, можно воспользоваться функцией

DWORD GetPriorityClass (
HANDLE Process);

Выбрав класс приоритета для процесса, не нужно забывать о потоках. В ОС Windows поддерживается семь относительных приоритетов потоков: ***THREAD_PRIORITY_TIME_CRITICAL*** (уровень приоритета 31 в классе ***REALTIME*** и 15 в других); ***THREAD_PRIORITY_HIGHEST*** (на два уровня выше для текущего класса, для ***NORMAL*** уровень приоритета равен 10); ***THREAD_PRIORITY_ABOVE_NORMAL*** (на один уровень выше, для ***NORMAL*** уровень приоритета равен 9); ***THREAD_PRIORITY_NORMAL*** (обычный приоритет для класса, для ***NORMAL*** уровень равен 8); ***THREAD_PRIORITY_BELOW_NORMAL*** (на один уровень ниже, для ***NORMAL*** уровень приоритета равен 7); ***THREAD_PRIORITY_LOWEST*** (на два уровня ниже для текущего класса, для ***NORMAL*** уровень приоритета равен 6); ***THREAD_PRIORITY_IDLE*** (16 – для ***REALTIME*** и 1 в других классах).

Задать относительный приоритет потока можно с помощью функции

BOOL SetThreadPriority (
HANDLE Thread,
int Priority);

Узнать относительный приоритет потока позволяет функция

int GetThreadPriority (
HANDLE Thread);

Следующий фрагмент кода показывает, как создать поток с относительным приоритетом ***HIGH,*** а по умолчанию поток создается с приоритетом ***NORMAL***:

DWORD ThreadId;
HANDLE Thread = CreateThread (NULL, 0, ThreadFunction, NULL, CREATE_SUSPENDED, &ThreadId);
SetThreadPriority (Thread, THREAD_PRIORITY_HIGH);
ResumeThread (Thread);
CloseHandle (Thread);

Иногда бывает полезным знать, сколько времени поток затратил на выполнение каких-либо операций:

BOOL GetThreadTimes (

HANDLE Thread,
PFILETIME Created,
// Время с 01.01.1601 в сотнях наносекунд до создания потока
PFILETIME Exited,
// Время с 01.01.1601 в сотнях наносекунд до завершения потока
PFILETIME Kernel,
// Время в сотнях наносекунд, затраченное потоком на код ОС
PFILETIME User);
// Время в сотнях наносекунд, затраченное потоком на код программы

Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы.
2. Используя изученный материал, разработать программу, реализующую полученное задание с измерением времени работы каждого потока.
3. Написать отчет.

Варианты заданий

Вариант №1

Разработать программу, которая вычисляет сумму и произведение чисел от *L* до *U*, где *L* – это нижняя граница диапазона, *U* – верхняя граница диапазона. Вычисление суммы и произведения оформить как две функции потока. Значения границ диапазон вводятся пользователем, затем запускаются два требуемых потока (первый с приоритетом *THREAD_PRIORITY_HIGHEST*, другой – *THREAD_PRIORITY_IDLE*), а потом на экран выводятся полученные значения, а также значения времени работы обоих потоков.

Вариант №2

Разработать программу, которая вычисляет число Фибоначчи по номеру, введенному пользователем, и формуле $F_i = F_{i-1} + F_{i-2}$, $F_0 = F_1 = 1$. Вычисление числа Фибоначчи оформить как функцию потока. По завершении функции потока программа выводит число на экран. Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_TIME_CRITICAL*, второй – *THREAD_PRIORITY_NORMAL*, и вывести на экран значения времени работы потока.

Вариант №3

Разработать программу для перевода целого числа со знаком в его строковый эквивалент прописью. Перевод числа оформить как функцию потока. Ввод

числа происходит до запуска потока, а вывод строки – по его завершении. Например, ввод «–1211» должен приводить к выводу «минус тысяча двести одиннадцать». Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_ABOVE_NORMAL*, второй – *THREAD_PRIORITY_LOWEST*, и вывести на экран значения времени работы потока.

Вариант №4

Разработать программу для перевода знакового числа с плавающей точкой в его строковый эквивалент прописью. Перевод числа оформить как функцию потока. Ввод числа происходит до запуска потока, а вывод строки – по его завершении. Например, ввод «–12.11» должен приводить к выводу «минус двенадцать целых одиннадцать сотых». Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_ABOVE_NORMAL*, второй – *THREAD_PRIORITY_HIGHEST*, и вывести на экран значения времени работы потока.

Вариант №5

Разработать программу, осуществляющую ввод двух строк, введенных пользователем. Далее, если обе строки хранят целые числа со знаком, то на экран выводится сумма чисел, в противном случае – конкатенация двух введенных строк. Проверку на соответствие строки целому числу, вычисление суммы чисел и конкатенацию строк оформить как три разные функции потока с приоритетами соответственно *THREAD_PRIORITY_ABOVE_NORMAL, HREAD_PRIORITY_LOWEST* и *THREAD_PRIORITY_IDLE*. Ввод строк осуществляется до запуска всех потоков, а вывод результатов – после их завершения. Также выводятся значения времени работы каждого потока.

Вариант №6

Разработать программу, вычисляющую сумму и произведение двух матриц. Выполнение этих операций оформить как две функции потока. Сначала программа осуществляет ввод элементов матриц, далее запускает оба потока с приоритетами *THREAD_PRIORITY_IDLE* и *THREAD_PRIORITY_TIME_CRITICAL*, а затем выводит на экран результаты, а также значения времени работы каждого потока.

Вариант №7

Разработать программу для упорядочивания одномерного целочисленного массива. Сортировка массива по возрастанию должна осуществляться любым из так называемых «улучшенных алгоритмов» сортировки и оформляется как функция потока. Сначала выполняется ввод элементов матрицы, затем запускается

поток и далее – вывод упорядоченного массива. Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_ABOVE_NORMAL*, второй – *THREAD_PRIORITY_LOWEST*, и вывести на экран значения времени работы потока.

Вариант №8

Разработать программу для упорядочивания одномерного массива чисел с плавающей точкой. Сортировка массива по возрастанию должна осуществляться любым из так называемых «улучшенных алгоритмов» сортировки и оформляется как функция потока. Сначала выполняется ввод элементов матрицы, затем запускается поток и далее – вывод упорядоченного массива. Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_ABOVE_NORMAL*, второй – *THREAD_PRIORITY_HIGHEST*, и вывести на экран значения времени работы потока.

Вариант №9

Разработать программу для упорядочивания одномерного массива строк. Сортировка массива по возрастанию должна осуществляться любым из так называемых «улучшенных алгоритмов» сортировки и оформляется как функция потока. Сначала выполняется ввод элементов матрицы, затем запускается поток и далее – вывод упорядоченного массива. Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_BELOW_NORMAL*, второй – *THREAD_PRIORITY_ABOVE_NORMAL*, и вывести на экран значения времени работы потока.

Вариант №10

Разработать программу для вычисления суммы элементов, лежащих на главной и побочной диагоналях квадратной матрицы. Выполнение этой операции оформляется как функция потока. Ввод элементов матрицы осуществляется до запуска потока, а вывод полученного значения – по его завершении. Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_NORMAL*, второй – *THREAD_PRIORITY_LOWEST*, и вывести на экран значения времени работы потока.

Вариант №11

Разработать программу для вычисления суммы элементов, не лежащих на главной и побочной диагоналях квадратной матрицы. Выполнение этой операции оформляется как функция потока. Ввод элементов матрицы осуществляется до запуска потока, а вывод полученного значения – по его завершении. Запустить

программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_NORMAL*, второй – *THREAD_PRIORITY_LOWEST*, и вывести на экран значения времени работы потока.

Вариант №12

Разработать программу для вычисления полного количества дней, прошедших между двумя датами. Даты – это строки вида ЦЦ.ЦЦ.ЦЦЦЦ, где Ц – это любая цифра из диапазона 0–9. Вычисление разницы между датами оформляется как функция потока. Сначала осуществляется ввод дат, затем запускается поток, и далее результат выводится на экран. Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_BELOW_NORMAL*, второй – *THREAD_PRIORITY_ABOVE_NORMAL*, и вывести на экран значения времени работы потока.

Вариант №13

Разработать программу для вычисления полного количества секунд, прошедших между двумя значениями времени. Значение времени – это строки вида ЦЦ.ЦЦ.ЦЦ, где Ц – это любая цифра из диапазона 0–9. Вычисление разницы между временами оформляется как функция потока. Сначала осуществляется ввод значений времени, затем запускается поток, и далее результат выводится на экран. Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_NORMAL*, второй – *THREAD_PRIORITY_LOWEST*, и вывести на экран значения времени работы потока.

Вариант №14

Разработать программу для поиска вхождения подстроки в строку. Эта операция оформляется как функция потока и реализует любой из известных методов поиска подстроки, кроме прямого. Сначала осуществляется ввод двух строк, затем запускается поток, и далее выводятся результаты: значение индекса элемента первой строки, с которого началось совпадение, или «–1» в противном случае. Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_TIME_CRITICAL*, второй – *THREAD_PRIORITY_NORMAL*, и вывести на экран значения времени работы потока.

Вариант №15

Разработать программу для подсчета количества вхождений подстроки в строку. Эта операция оформляется как функция потока и реализует любой из известных методов поиска подстроки, кроме прямого. Сначала осуществляется ввод двух строк, затем запускается поток, и далее выводится

результат – целое число. Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_ABOVE_NORMAL*, второй – *THREAD_PRIORITY_HIGHEST*, и вывести на экран значения времени работы потока.

Вариант №16

Разработать программу для получения строкового эквивалента даты прописью. Дата – это строка вида ЦЦ.ЦЦ.ЦЦЦЦ, где Ц – это любая цифра из диапазона 0–9. Получение строкового эквивалента оформляется как функция потока. Сначала осуществляется ввод даты, затем запускается поток, и далее результат выводится на экран: число и месяц прописью, а за последними четырьмя – слово «года» (например, ввод «29.02.2008» приводит к выводу «Двадцать девятое февраля 2008 года»). Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_TIME_CRITICAL*, второй – *THREAD_PRIORITY_NORMAL*, и вывести на экран значения времени работы потока.

Вариант №17

Разработать программу для получения строкового эквивалента значения времени прописью. Время – это строка вида ЦЦ.ЦЦ.ЦЦ, где Ц – это любая цифра из диапазона 0–9. Получение строкового эквивалента оформляется как функция потока. Сначала осуществляется ввод значения времени, затем запускается поток, и далее результат выводится на экран: значение часов, минут и секунд прописью (например, ввод «12.01.20» приводит к выводу «двенадцать часов одна минута двадцать секунд»). Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_BELOW_NORMAL*, второй – *THREAD_PRIORITY_ABOVE_NORMAL*, и вывести на экран значения времени работы потока.

Вариант №18

Разработать программу, осуществляющую инвертирование битовой строки, а также ее перевод в десятичное число. Битовая строка – это строка, состоящая из нулей и единиц. Инвертирование битовой строки и перевод строки в десятичное число оформляется как две функции потока. Сначала осуществляется ввод битовой строки, затем запускаются два потока: первый с приоритетом *THREAD_PRIORITY_HIGHEST*, второй – *THREAD_PRIORITY_IDLE*, и далее выводятся результаты, а также значения времени работы каждого потока.

Вариант №19

Разработать программу, осуществляющую реверс битовой строки, а также ее перевод в десятичное число. Битовая строка – это строка, состоящая из нулей и единиц. Реверс битовой строки (все нули заменяются на единицы, а единицы на нули) и перевод строки в десятичное число оформляется как две функции потока. Сначала осуществляется ввод битовой строки, затем запускаются два потока: первый с приоритетом *THREAD_PRIORITY_LOWEST,* второй – *THREAD_PRIORITY_ABOVE_NORMAL*, и далее выводятся результаты, а также значения времени работы каждого потока.

Вариант №20

Разработать программу, осуществляющую вычисление факториала числа. Выполнение данной операции (по формуле N! = N · (N – 1)!, где 0 != 1) оформляется как функция потока. Сначала осуществляется ввод числа, затем запускается поток, и далее результат выводится на экран. Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_BELOW_NORMAL*, второй – *THREAD_PRIORITY_ABOVE_NORMAL*, и вывести на экран значения времени работы потока.

Вариант №21

Разработать программу, вычисляющую сумму крайних элементов квадратной матрицы. Выполнение данной операции оформляется как функция потока. Сначала осуществляется ввод элементов матрицы, затем запускается поток, и далее результат выводится на экран. Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_ABOVE_NORMAL*, второй – *THREAD_PRIORITY_HIGHEST*, и вывести на экран значения времени работы потока.

Вариант №22

Разработать программу, осуществляющую поиск элемента по ключу в целочисленном векторе любым известным методом, кроме прямого. Выполнение данной операции оформляется как функция потока. Сначала осуществляется ввод элементов вектора и значения ключа поиска, затем запускается поток, и далее результат выводится на экран: значение индекса найденного элемента или «–1» в противном случае. Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_NORMAL*, второй – *THREAD_PRIORITY_LOWEST,* и вывести на экран значения времени работы потока.

Вариант №23

Разработать программу для замены всех латинских букв в строке на их аналоги из кириллицы. Выполнение данной операции оформляется как функция потока. Сначала осуществляется ввод строки, затем запускается поток, и далее измененная строка выводится на экран. Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_TIME_CRITICAL*, второй – *THREAD_PRIORITY_NORMAL*, и вывести на экран значения времени работы потока.

Вариант №24

Разработать программу для смены регистра всех символов в строке. Выполнение данной операции оформляется как функция потока. Сначала осуществляется ввод строки, затем запускается поток, и далее измененная строка выводится на экран. Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_HIGHEST*, второй – *THREAD_PRIORITY_IDLE*, и вывести на экран значения времени работы потока.

Вариант №25

Разработать программу для проверки того факта, что беззнаковое целое число является степенью двойки. Выполнение данной операции оформляется как функция потока. Сначала осуществляется ввод числа, затем запускается поток, и далее на экран выводится результат: если число является степенью двойки, то выводится показатель степени, в противном случае выводится сообщение «не является степенью двойки». Запустить программу два раза: первый раз с приоритетом потока *THREAD_PRIORITY_LOWEST*, второй – *THREAD_PRIORITY_TIME_CRITICAL*, и вывести на экран значения времени работы потока.

Содержание отчета

Отчет по лабораторной работе должен быть оформлен в соответствии с требованиями, предъявляемыми к текстовым документам, и содержать:

- цель работы;
- краткие теоретические сведения, необходимые для выполнения работы;
- описание порядка выполнения работы с приведением листинга кода, скриншотов рабочих окон;
- выводы по работе.

Лабораторная работа №8

Синхронизация потоков в среде ОС Windows

Цель – изучение объектов синхронизации потоков в операционной системе Windows.

Теоретическая часть

Существует обширный класс средств ОС, с помощью которых обеспечивается взаимная синхронизация процессов и потоков. Потребность в синхронизации потоков связана с совместным использованием ресурсов вычислительной системы. Синхронизация необходима для исключения гонок и тупиков при обмене данными между потоками, разделении данных, доступе к центральному процессору и УВВ.

Во многих ОС эти средства называются средствами межпроцессного взаимодействия (*IPC*), поскольку обычно к ним относятся не только средства межпроцессной синхронизации, но и средства межпроцессного обмена данными.

Синхронизация заключается в согласовании скоростей процессов и потоков путем приостановки потока до наступления некоторого события и последующей его активизации при наступлении этого события. ОС может предоставлять обширный набор средств синхронизации.

Очень важным понятием синхронизации является понятие «критической секции» программы. Критическая секция – это часть программы, результат выполнения которой может непредсказуемо меняться, если переменные, относящиеся к этой части программ, изменяются другими потоками в то время, когда выполнение этого кода еще не закончено. Критическая секция всегда определяется по отношению к конкретным критическим данным, при несогласованном изменении которых могут возникнуть нежелательные эффекты. Во всех потоках, работающих с критическими данными, должна быть определена критическая секция, в общем случае состоящая из разных последовательностей команд. Взаимное исключение позволяет обеспечить нахождение только одного потока в критической секции.

Для синхронизации потоков одного процесса программист может использовать глобальные блокирующие переменные. С этими переменными, к которым все потоки имеют прямой доступ, программист работает, не обращаясь к системным вызовам ОС. Каждому набору критических данных ставится в соответствие двоичная переменная, которой поток присваивает значение 0, когда он входит в критическую секцию, и значение 1, когда он ее покидает. Недостаток: во время нахождения одного потока в критической секции другой поток, требующий тот

же ресурс, получив доступ к процессору, будет с завидной регулярностью опрашивать блокирующую переменную, бесполезно затрачивая процессорное время. Для устранения этого недостатка во многих ОС предусмотрены специальные системные вызовы для работы с критическими секциями [7].

В Windows перед изменением критических данных поток выполняет системный вызов ***EnterCriticalSection,*** в рамках которого сначала выполняется проверка блокирующей переменной, отражающей состояние ресурса. Если он занят (значение блокирующей переменной равно 0), он блокирует поток и делает отметку о том, что поток должен быть активизирован, когда соответствующий ресурс освободится. Поток, который в это время использует данный ресурс, после выхода из критической секции должен выполнить вызов ***LeaveCriticalSection,*** в результате блокирующая переменная получает значение 1 (ресурс свободен), а ОС просматривает очередь ожидающих этот ресурс потоков и переводит первый поток в состояние готовности. Эти, а также некоторые другие функции *Win API* для работы с критическими секциями приведены ниже:

```
VOID EnterCriticalSection (PCRITICAL_SECTION Section);
VOID LeaveCriticalSection (PCRITICAL_SECTION Section);
```

Когда ни один поток не использует критическую секцию, ее можно удалить:

```
VOID DeleteCriticalSection (PCRITICAL_SECTION Section);
```

Когда критическая секция является локальной, ее нужно инициализировать;

```
VOID InitializeCriticalSection (PCRITICAL_SECTION Section);
```

Попытаться войти в критическую секцию без блокирования потока можно с помощью функции

```
BOOL TryEnterCriticalSection (PCRITICAL_SECTION Section);
```

Она возвращает *FALSE*, если ресурс занят другим потоком, и *TRUE*, если поток захватил нужный ресурс.

Для того чтобы организовать доступ к двум ресурсам, нужно создать две критические секции, что и демонстрирует следующий фрагмент кода:

```
int Numbers[500];                    // первый разделяемый ресурс
CRITICAL_SECTION Nums;
double Doubles[500];
CRITICAL_SECTION DoubleNums;         // вт орой разделяемый ресурс

DWORD ThFunction (PVOID Parametr)    // функция пот ока
{
// Вход в обе крит ические секции
EnterCriticalSection (&Nums);
EnterCriticalSection (&DoubleNums);
```

```
  // В эт ом коде т ребует ся одновременный дост уп
  // к обоим разделяемым ресурсам
for(int j = 0; j < 500; j ++) Doubles[j] = Numbers[j] = 500 – j;
  // Покидаем крит ические секции в обрат ном порядке
LeaveCriticalSection (&DoubleNums);
LeaveCriticalSection (&Nums);
return 0;
}
```

Эффект взаимной блокировки может возникнуть, если попытаться добавить еще одну функцию потока, где производится занятие тех же критических секций, но в обратном порядке:

```
DWORD ThFunction1 (PVOID Parametr)      // функция пот ока
{ // Вход в обе крит ические секции
EnterCriticalSection (&DoubleNums);
EnterCriticalSection (&Nums);
  // В эт ом коде т ребует ся одновременный дост уп
  // к обоим разделяемым ресурсам
for(int j = 0; j < 500; j ++) Doubles[j] = Numbers[j] = 500 – j;
  // Покидаем крит ические секции в обрат ном порядке
LeaveCriticalSection (&Nums);
LeaveCriticalSection (&DoubleNums);
return 0;
}
```

Здесь существует вероятность того, что *ThFunction* занимает критическую секцию *Nums*, а поток с функцией *ThFunction1* захватывает *DoubleNums*. И теперь, какая бы функция не выполнялась, она не сумеет войти в другую, так необходимую ей критическую секцию.

Замечательное свойство критической секции заключается в том, что она не использует переход из режима пользователя в режим ядра. Следовательно, скорость ее работы достаточно высока, и этот объект может быть использован для большинства программ, требующих синхронизации. К недостаткам можно отнести то, что их можно использовать для синхронизации потоков, принадлежащих только одному и тому же процессу.

Обобщением блокирующих переменных являются так называемые семафоры Дийкстры. Вместо двоичных переменных Эдсгер Дийкстра предложил использовать переменные, которые могут принимать целые неотрицательные значения. Такие переменные, используемые для синхронизации, получили название семафоров.

Для работы с семафорами вводятся два примитива (действия) *P* и *V*. Пусть

переменная S представляет собой семафор, тогда действия $V(S)$ и $P(S)$ определяются так:

- $V(S)$: переменная S увеличивается на 1. Выборка, инкремент и сохранение не могут быть прерваны. К переменной S нет доступа другим потокам во время выполнения этой операции;

- $P(S)$: уменьшение S, если это возможно. Если S равно 0, то поток вызывает операцию P, пока декремент станет возможным. Проверка и уменьшение являются неделимой операцией.

В частном случае, когда семафор может принимать только значения 0 и 1, он превращается в блокирующую переменную, которую часто называют двоичным семафором.

Блокирующие переменные и семафоры Дийкстры не подходят для синхронизации потоков разных процессов. ОС должна предоставлять потокам системные объекты синхронизации, которые были бы видны для всех потоков, даже если они принадлежат разным процессам и работают в разных адресных пространствах. Набор таких объектов зависит от конкретной ОС, которая создает их по запросам пользователей. Примерами синхронизирующих объектов являются системные семафоры, мьютексы, события, таймеры и др. Работа с синхронизирующими объектами подобна работе с файлами: их можно создавать, открывать, закрывать, уничтожать. Кроме того, для синхронизации могут использоваться файлы, процессы и потоки. Все синхронизирующие объекты могут находиться в двух состояниях: сигнальном (свободном) и несигнальном (занятом). Для каждого объекта смысл сигнального состояния зависит от типа объекта [7].

Потоки с помощью специального системного вызова сообщают ОС, что они хотят синхронизировать свое выполнение с состоянием некоторого объекта (*WaitForSingleObject* в Windows). Другой системный вызов может переводить объект в сигнальное состояние (например, *SetEvent* в Windows).

Поток может ожидать установку сигнального состояния не одного объекта, а нескольких (*WaitForMultipleObjects)*. При этом он может попросить ОС активизировать его при установке либо одного указанного объекта, либо всех. Поток может в качестве аргумента системного вызова ожидания указать также максимальное время, в течение которого он будет ожидать переход объекта в сигнальное состояние, после чего ОС должна активизировать его в любом случае. Сразу несколько потоков могут ожидать установку некоторого объекта в сигнальное состояние. В зависимости от объекта в состояние готовности могут переводиться либо все ожидающие это событие потоки, либо один из них.

В ОС Windows есть довольно богатый набор функций, которые ожидают переход в сигнальное состояние одного или нескольких объектов.

```
DWORD WaitForSingleObject (
  HANDLE Object,
  DWORD Milliseconds);          // определяет  ож идания,
                    // INFINITE – бесконечное ож идание
```

В следующем коде поток блокирован, пока не выполнится другой поток *Th*1:

```
WaitForSingleObject (Th1, INFINITE);
```

Второй пример демонстрирует значение тайм-аута, не равное *INFINITE*:

```
DWORD dw = WaitForSingleObject(Th1, 10000);
switch (dw) {
  case WAIT_OBJECT_0:       // пот ок завершил работ у
    break;
  case WAIT_TIMEOUT:        // пот ок не завершился через 10 с
    break;
  case WAIT_FAILED:         // произошла какая-т о ошибка
    break;
}
```

Сразу несколько объектов или один из списка можно вызвать с помощью функции:

```
DWORD WaitForMultipleObjects (
  DWORD Counter,            // количест во объект ов ядра (от  1 до 64)
  HANDLE *Objects,          // массив описат елей объект ов
  BOOL WaitForAll,          // ож идат ь все (TRUE) или
            // один из списка (FALSE)
  DWORD Milliseconds);      // определяет  ож идания,
                    // INFINITE – бесконечное ож идание
```

Следующий код демонстрирует использование этой функции:

```
HANDLE hh[2];
hh[0] = Th1; hh[1] = Th2;
DWORD = WaitForMultipleObjects(2, hh, FALSE, 10000);
switch (dw) {
  case WAIT_FAILED: // произошла какая-т о ошибка
    break;
  case WAIT_TIMEOUT: // пот ок не завершился через 10 с
    break;
  case WAIT_OBJECT_0: // пот ок Th1 завершил работ у
    break;
  case WAIT_OBJECT_0 + 1: // пот ок Th2 завершил работ у
    break;
}
```

В числе других в ОС можно встретить такие объекты, как событие, мьютекс, системный семафор, таймер.

Объект-событие используется для того, чтобы оповестить другие потоки о том, что некоторые действия завершены. События обычно используют в том случае, когда какой-то поток выполняет инициализацию, а затем сигнализирует другому потоку, что он может продолжить работу. Инициализирующий поток переводит «событие» в несигнальное состояние и приступает к своим операциям. Закончив, он сбрасывает «событие» в сигнальное состояние. Тогда другой поток, ждавший перехода события в сигнальное состояние, переводится в состояние готовности. В ОС Windows события содержат счетчик числа пользователей и две логические переменные: тип события и состояние. Эти объекты могут быть двух типов: с автосбросом и с ручным сбросом. Событие создается функцией

```
HANDLE CreateEvent (
  PSECURITY_ATTRIBUTES Attributes,        // ат рибут ы защит ы
  BOOL ManualOrAuto,                      // ручной (TRUE) или
                                          // авт омат ический (FALSE) сброс
  BOOL Initial,                  // Начальное сост ояние: свободен (TRUE) или
                                 // занят  (FALSE)
  PCTSTR Name);                  // Символьное имя объект а
```

Созданное событие может открываться с помощью функции

```
HANDLE OpenEvent (
  DWORD Access,                  // реж им дост упа
  BOOL Inherit,                  // наследование
  PCTSTR Name);                  // Символьное имя объект а
```

После создания события можно управлять его состоянием. Для этого существуют две функции. Перевод в сигнальное состояние осуществляется вызовом функции

```
BOOL SetEvent (HANDLE Event);
```

Сменить его на занятое можно при помощи функции

```
BOOL ResetEvent (HANDLE Event);
```

Когда поток успешно дождался события с автосбросом, он автоматически сбрасывается в занятое состояние, и для них, как правило, не нужно вызывать функцию *ResetEvent.*

Как используются события, показано в следующем фрагменте кода:

```
HANDLE Event1;
int WinMain(...)
{
// Создает ся событ ие с ручным сбросом в сигнальном сост оянии
Event1 = CreateEvent (NULL, FALSE, FALSE, NULL);
// Создают ся два пот ока, причем пропущены все парамет ры,
```

```
// кроме функции пот ока
HANDLE Th1 = CreateThread (..., Function1,...);
HANDLE Th2 = CreateThread (..., Function2,...);
...
// далее мож но выполнят ь любые дейст вия
...
CloseHandle (Event1);
}
DWORD WINAPI Function1 (PVOID Parametr)
{
WaitForSingleObject (Event1, INFINITE);
...
SetEvent (Event1);
return 0;
}
DWORD WINAPI Function2 (PVOID Parametr)
{
WaitForSingleObject (Event1, INFINITE);
...
SetEvent (Event1);
return 0;
}
```

Ожидаемый таймер – это объект, который самостоятельно переходит в сигнальное состояние в определенное время или через регулярные промежутки времени. Объект создается функцией

```
HANDLE CreateWaitableTimer (
  PSECURITY_ATTRIBUTES Attributes,       // ат рибут ы защит ы
  BOOL ManualOrAuto,
    // ручной (TRUE) или авт омат ический (FALSE) сброс сост ояния
  PCTSTR Name);                          // Символьное имя объект а
```

Ожидаемый таймер всегда создается в несигнальном состоянии. Чтобы перевести таймер в сигнальное состояние, достаточно вызвать функцию

```
HANDLE SetWaitableTimer (
  HANDLE Timer,                          // нуж ный т аймер
  const LARGE_INTEGER *DueTime,
    // Когда т аймер долж ен сработ ат ь впервые
  LONG Period,                    // сколько будет  срабат ыват ь т аймер
                  // 0 – для одного срабат ывания
  PTIMERAPCROUTINE ComplRoutine,
    // процедура для асинхронного вызова
```

```
    PVOID ArgumentsToRoutine,
      // аргумент  для процедуры асинхронного вызова
    BOOL Resume); // необходим для компьют еров с поддерж кой спящего реж има
```

Чтобы перевести ожидаемый таймер в несигнальное состояние, нужно вызвать функцию

```
HANDLE CancelWaitableTimer (
  HANDLE Timer);              // нуж ный т аймер
```

В следующем фрагменте кода таймер настраивается так, чтобы сработать в первый раз 29 февраля 2004 года в 17.30, и после этого – каждые три часа, т. е. 10 800 000 мс:

```
HANDLE Timer1;
SYSTEMTIME Time1;
FILETIME LocalTime, UTC_Time;
LARGE_INTEGER IntUTC;

// создает ся т аймер с авт осбросом
Timer1 = CreateWaitableTimer (NULL, FALSE, NULL);
// задают ся парамет ры для т аймера
Time1.wYear = 2004; Time1.wMonth = 2; Time1.wDay = 29;
Time1.wHour = 17; Time1.wMinute = 30; Time1.wSecond = 0;
Time1.wMilliseconds = 0;

SystemTimeToFileTime (&Time1, &LocalTime);
LocalFileTimeToFileTime (&LocalTime, &UTC_Time);
IntUTC.LowPart = UTC_Time.dwLowDateTime;
IntUTC.HighPart = UTC_Time.dwHighDateTime;

// наконец уст анавливает ся т аймер
SetWaitableTimer (Timer1, &IntUTC, 10800000, NULL, NULL, FALSE);
...
CloseHandle (Timer1);
```

Можно также устанавливать время срабатывания не в абсолютных единицах, а в относительных, которые рассчитываются в блоках по 100 нс (т. е. 0,1 с равна миллиону таких блоков), число при этом должно быть отрицательным. В следующем коде показано, как установить таймер на срабатывание через 20 с после вызова соответствующей функции:

```
HANDLE Timer1;
```

```
LARGE_INTEGER LargeInt;
// создает ся т аймер с авт осбросом
Timer1 = CreateWaitableTimer (NULL, FALSE, NULL);
// задают ся парамет ры для т аймера,
// кот орый долж ен сработ ат ь через 20 с
// Время берет ся в блоках по 100 нс
int UnitsBySeconds = 10000000;
LargeInt = -20 * UnitsBySeconds;
// наконец уст анавливает ся т аймер,
// кот орый сработ ает  вначале через 20 с,
// а пот ом каж дые т ри часа
SetWaitableTimer (Timer1, &LargeInt, 10800000, NULL, NULL, FALSE);
...
CloseHandle (Timer1);
```

Примечание – Когда освобождается таймер с ручным сбросом, то все потоки, ожидавшие данный объект, возобновляют свою работу, а когда в сигнальное состояние переходит таймер с автосбросом, возобновляется выполнение только одного потока.

Семафор в целом был описан ранее. Что касается ОС Windows, то семафор в них является объектом ядра, и чаще всего такие объекты используются для учета ресурсов. В них помимо остальных параметров, характерных для многих объектов ядра, есть еще два специфичных: один используется для установки максимально возможного числа ресурсов, а второй – это счетчик настоящего количества ресурсов. Для семафоров определены следующие правила работы:

1. Семафор переходит в сигнальное состояние, если значение счетчика ресурсов больше 0.
2. Семафор занят, если значение счетчика равно 0.
3. Не допускается установка отрицательного значения счетчика.
4. Счетчик не может иметь значение, большее максимального числа ресурсов.

Семафор создается вызовом функции

```
HANDLE CreateSemaphore (
  PSECURITY_ATTRIBUTES Attributes,      // ат рибут ы защит ы
  LONG Initial,          // количест во ресурсов, дост упных изначально
  LONG Maximum,          // максимальное количест во ресурсов
  PCTSTR Name);          // Символьное имя объект а
```

Получить описатель существующего семафора можно с помощью функции

```
HANDLE OpenSemaphore (
```

```
  DWORD Access,                 // реж им дост упа
  BOOL Inherit,                 // наследование
  PCTSTR Name);                 // Символьное имя объект а
```

Поток может увеличить счетчик настоящего количества ресурсов, вызвав функцию

```
BOOL ReleaseSemaphore (
  HANDLE Semaphore,             // описат ель объект а-семафора
  LONG ReleaseCount,
    // насколько увеличит ь счет чик ресурсов (обычно 1)
  PLONG PreviousCount);
```

Процесс может использовать семафор, чтобы ограничить число окон, которые он создаст. Сначала он использует функцию *CreateSemaphore*, чтобы создать семафор и определить начальное и максимальное значения счетчика:

```
HANDLE Semaphore;   LONG Max = 12, PreviousCount;
// создание семафора с одинаковыми значениями счет чиков, равными 12
Semaphore = CreateSemaphore(NULL, cMax, cMax, NULL);
// безымянный семафор
if (Semaphore == NULL)
{               // проверка ошибок
}
```

Прежде чем любой поток создаст новое окно, он вызывает функцию *WaitForSingleObject*, чтобы определить, разрешает ли текущий счетчик семафора создание дополнительных окон. Параметр блокировки времени функции ожидания установлен на 0.

```
DWORD WaitResult;
WaitResult = WaitForSingleObject(Semaphore, 0);
switch (WaitResult) {
   // Семафор свободен
   case WAIT_OBJECT_0:
      // мож но создат ь следующее окно
      break;
   // Семафор занят , время прошло
   case WAIT_TIMEOUT:
      // Не создават ь следующее окно
      break;
}
```

Когда поток закрывает окно, он вызывает функцию *ReleaseSemaphore*,

чтобы увеличить счетчик семафора:

```
if (!ReleaseSemaphore(Semaphore, 1, NULL))
{     // Возникла ошибка
}
```

Объект ядра «мьютексы» гарантирует потокам взаимоисключающий доступ к единственному ресурсу. Отсюда и пошло название объекта. Поток, пытаясь получить доступ к критическим данным, выполнил соответствующий системный вызов. Мьютекс находится в сигнальном состоянии, в этом случае поток тут же становится его владельцем, устанавливая его в несигнальное состояние, и входит в критическую секцию. После того как поток выполнил работы с критическими данными, он отдает мьютекс, устанавливая его в сигнальное состояние. В этот момент мьютекс свободен и не принадлежит ни одному потоку. Если какой-либо поток ожидает его освобождения, то он становится следующим владельцем этого мьютекса, одновременно мьютекс переходит в несигнальное состояние. Таким образом, эти объекты должны содержать счетчик числа пользователей, счетчик рекурсии и идентификатор потока, а для мьютекса определены следующие правила:

1. Если идентификатор потока равен 0, мьютекс находится в сигнальном состоянии и не захвачен ни одним потоком.
2. Если идентификатор потока не равен 0, мьютекс захвачен одним потоком и находится в несигнальном состоянии.
3. Мьютексы могут нарушать правила, действующие в ОС.

Процесс может создать мьютекс вызовом функции

```
HANDLE CreateMutex (
  PSECURITY_ATTRIBUTES Attributes,     // атрибуты защиты
  BOOL InitialOwner,                   // начальное состояние мьютекса
  PCTSTR Name);                        // Символьное имя объекта
```

Получить описатель существующего мьютекса можно с помощью функции

```
HANDLE OpenMutex (
  DWORD Access,                        // режим доступа
  BOOL Inherit,                        // наследование
  PCTSTR Name);                        // Символьное имя объекта
```

Параметр *InitialOwner* определяет начальное состояние мьютекса. Если передается *FALSE*, то мьютекс находится в сигнальном состоянии и не принадлежит ни одному потоку, причем идентификатор потока и счетчик рекурсии равны 0. Если передается *TRUE*, то счетчик рекурсии становится равным 1, а

идентификатор потока в мьютексе становится равным идентификатору вызвавшего потока, и теперь мьютекс находится в несигнальном состоянии. Поток получает доступ к ресурсу, вызывая какую-либо ожидающую функцию с передачей ей описателя мьютекса. Если она определяет, что мьютекс занят, то вызывающий поток переходит в состояние ожидания. Это запоминается, и когда идентификатор потока обнуляется, в него записывается идентификатор ждущего потока, счетчику рекурсии присваивается при этом значение 1. После этого ожидающий поток может быть активизирован [7].

Надо отметить, что система обязательно проверит, не совпадает ли идентификатор потока у мьютекса с идентификатором потока, ожидающего мьютекс. В случае совпадения система выделит потоку процессорное время, хотя мьютекс еще занят. Счетчик рекурсии увеличивается на 1, когда поток захватывает мьютекс, и значение этого счетчика больше 1 только в том случае, если поток захватывает один и тот же объект несколько раз. Когда ожидание мьютекса завершается, поток получает монопольный доступ к ресурсу. Все остальные потоки переходят в состояние ожидания. По окончании работы с ресурсом поток обязан освободить мьютекс вызовом функции

BOOL ReleaseMutex (HANDLE Mutex);

Данная функция уменьшает счетчик рекурсии на 1, и если мьютекс передавался во владение потока несколько раз, он должен вызвать *ReleaseMutex* такое же число раз, чтобы, в конце концов, обнулить счетчик рекурсии. Когда это произойдет, идентификатор потока также станет равным 0, и мьютекс станет свободным. Система проверит, нет ли ожидающих потоков, и выберет один из них, чтобы передать тому мьютекс.

Как видно, этот объект обладает одним замечательным свойством, которого нет у остальных синхронизирующих объектов: мьютекс запоминает поток, которому он принадлежит. Если посторонний поток попытается вызвать ***ReleaseMutex,*** то эта функция просто вернет *FALSE*. Если же какой-то поток завершится, не успев освободить мьютекс, то считается, что произошел отказ от мьютекса, и система переведет его в сигнальное состояние:

```
HANDLE Thread1, Thread2;
HANDLE Mutex1;
int WinMain(...)
{
Mutex1 = CreateMutex (NULL, FALSE, “Mutex1”);
// Создает ся мьют екс
// Создают ся два пот ока, причем пропущены все парамет ры,
```

```
// кроме функции пот ока
HANDLE Th1 = CreateThread (..., Function1,...);
HANDLE Th2 = CreateThread (..., Function2,...);
// далее мож но выполнят ь любые дейст вия
...
CloseHandle (Mutex1);
}
DWORD WINAPI Function1 (PVOID Parametr)
{
WaitForSingleObject (Mutex1, INFINITE);
...
ReleaseMutex (Mutex1);
return 0;
}
DWORD WINAPI Function2 (PVOID Parametr)
{
WaitForSingleObject (Mutex1, INFINITE);
...
ReleaseMutex (Mutex1);
return 0;
}
```

Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы.
2. Получить индивидуальный вариант задания у преподавателя, предусматривающий использование какого-либо объекта синхронизации.
3. Разработать программу в соответствии с полученным заданием и примерами использования соответствующих объектов.
4. Написать отчет.

Варианты заданий

Вариант №1

В пансионе отдыхают и предаются размышлениям 5 философов, пронумерованных от 1 до 5. В столовой расположен круглый стол, вокруг которого расставлены 5 стульев, также пронумерованных от 1 до 5. На столе находится одна большая тарелка со спагетти, которая пополняется бесконечно, также там расставлены 5 тарелок, в которые накладывается спагетти, и 5 вилок, назначение которых очевидно.

Для того чтобы пообедать, философ входит в столовую и садится на стул

со своим номером. При этом есть философ сможет только в том случае, если свободны две вилки – справа и слева от его тарелки. При выполнении этого условия философ поднимает одновременно обе вилки и может поглощать пищу в течение какого-то заданного времени. В противном случае философу приходится ждать освобождения обеих вилок. Пообедав, философ кладет обе вилки на стол одновременно и уходит. Описанный процесс происходит бесконечно.

Воспользоваться объектами синхронизации типа «ожидаемый таймер».

Вариант №2

Условие описано в варианте №1.

Воспользоваться объектами синхронизации типа «событие».

Вариант №3

Условие описано в варианте №1.

Воспользоваться объектами синхронизации типа «семафор».

Вариант №4

Условие описано в варианте №1.

Воспользоваться объектами синхронизации типа «мьютекс».

Вариант №5

Условие описано в варианте №1.

Воспользоваться объектами синхронизации типа «критическая секция».

Вариант №6

В парикмахерской расположено единственное кресло, на котором спит парикмахер, и несколько стульев для клиентов.

Когда клиент приходит в парикмахерскую, он будит парикмахера, садится в кресло. Стрижка производится в течение заданного времени. Если же кресло занято другим клиентом, то вновь прибывший клиент занимает любой свободный стул и ожидает своей очереди (клиенты обслуживаются в порядке очередности, например, времени прибытия). Если все стулья заняты, то клиент поворачивается и уходит.

Когда обслужены все клиенты, парикмахер садится в кресло и снова засыпает. Описанный процесс происходит бесконечно.

Воспользоваться объектами синхронизации типа «ожидаемый таймер».

Вариант №7

Условие описано в варианте №6.
Воспользоваться объектами синхронизации типа «событие».

Вариант №8

Условие описано в варианте №6.
Воспользоваться объектами синхронизации типа «семафор».

Вариант №9

Условие описано в варианте №6.
Воспользоваться объектами синхронизации типа «мьютекс».

Вариант №10

Условие описано в варианте №6.
Воспользоваться объектами синхронизации типа «критическая секция».

Вариант №11

Рассмотрим взаимодействие двух потоков, один из которых пишет данные в буферный пул, а другой считывает их из пула. Буферный пул состоит из N буферов, каждый содержит одну запись. В общем случае поток-писатель и поток-читатель имеют разные скорости и обращаются к пулу с переменной интенсивностью. Для правильной работы поток-писатель приостанавливается, когда все буферы заняты, и переходит в активное состояние при наличии хотя бы одного свободного буфера. Поток-читатель приостанавливается, когда все буферы пусты, и активизируется, когда появляется по крайней мере одна запись. Описанный процесс происходит бесконечно.

Воспользоваться объектами синхронизации типа «ожидаемый таймер».

Вариант №12

Условие описано в варианте №11.
Воспользоваться объектами синхронизации типа «событие».

Вариант №13

Условие описано в варианте №11.
Воспользоваться объектами синхронизации типа «семафор».

Вариант №14

Условие описано в варианте №11.
Воспользоваться объектами синхронизации типа «мьютекс».

Вариант №15

Условие описано в варианте №11.

Воспользоваться объектами синхронизации типа «критическая секция».

Вариант №16

В пансионе отдыхают и предаются размышлениям 5 философов, пронумерованных от 1 до 5. В столовой расположен круглый стол, вокруг которого расставлены 5 стульев, также пронумерованных от 1 до 5. На столе находится одна большая тарелка со спагетти, которая пополняется бесконечно, также там расставлены 5 тарелок, в которые накладывается спагетти, и 5 вилок, назначение которых очевидно.

Для того чтобы пообедать, философ входит в столовую и садится на любой стул. При этом есть философ сможет только в том случае, если свободны две вилки – справа и слева от его тарелки. При выполнении этого условия философ поднимает одновременно обе вилки и может поглощать пищу в течение какого-то заданного времени. В противном случае философу приходится ждать освобождения обеих вилок. Пообедав, философ кладет обе вилки на стол одновременно и уходит. Описанный процесс происходит бесконечно.

Воспользоваться объектами синхронизации типа «ожидаемый таймер».

Вариант №17

Условие описано в варианте №16.

Воспользоваться объектами синхронизации типа «событие».

Вариант №18

Условие описано в варианте №16.

Воспользоваться объектами синхронизации типа «семафор».

Вариант №19

Условие описано в варианте №16.

Воспользоваться объектами синхронизации типа «мьютекс».

Вариант №20

Условие описано в варианте №16.

Воспользоваться объектами синхронизации типа «критическая секция».

Вариант №21

В парикмахерской расположено единственное кресло, на котором спит парикмахер, и несколько стульев для клиентов, которые делятся на два класса – обычные и «блатные». Сначала всегда обслуживаются «блатные» клиенты, и

только после этого парикмахер может работать с обычными клиентами.

Когда клиент приходит в парикмахерскую, он будит парикмахера, садится в кресло. Стрижка производится в течение заданного времени. Если же кресло занято другим клиентом, то вновь прибывший клиент занимает любой свободный стул и ожидает своей очереди. Далее клиенты обслуживаются в порядке приоритета и очередности (времени прибытия). Если все стулья заняты, то клиент поворачивается и уходит. Когда обслужены все клиенты, парикмахер садится в кресло и снова засыпает. Описанный процесс происходит бесконечно.

Воспользоваться объектами синхронизации типа «ожидаемый таймер».

Вариант №22

Условие описано в варианте №21.

Воспользоваться объектами синхронизации типа «событие».

Вариант №23

Условие описано в варианте №21.

Воспользоваться объектами синхронизации типа «семафор».

Вариант №24

Условие описано в варианте №21.

Воспользоваться объектами синхронизации типа «мьютекс».

Вариант №25

Условие описано в варианте №21.

Воспользоваться объектами синхронизации типа «критическая секция».

Содержание отчета

Отчет по лабораторной работе должен быть оформлен в соответствии с требованиями, предъявляемыми к текстовым документам, и содержать:

- цель работы;
- краткие теоретические сведения, необходимые для выполнения работы;
- описание порядка выполнения работы с приведением листинга кода, скриншотов рабочих окон;
- выводы по работе.

Лабораторная работа №9

Использование механизма виртуальной памяти в ОС Windows

Цель – изучение виртуальной памяти в операционной системе Windows.

Теоретическая часть

Виртуальное адресное пространство каждого процесса в ОС Windows организовано следующим образом. В момент своего создания оно почти полностью пусто. Для использования какой-то его части необходимо выделить в нем определенные регионы с помощью функции ***VirtualAlloc***, эта операция называется резервированием. При этом ОС должна выравнивать начало региона в соответствии с так называемой гранулярностью выделения памяти, которая составляет на текущий момент времени 64 КБ. Также система должна учитывать, что размер региона должен быть кратен размеру страницы. Для процессоров *Pentuim* размер страницы составляет 4 КБ. Иными словами, если процесс попытается зарезервировать 10 КБ, то будет выделен регион размером 12 КБ. Когда регион становится не нужен, его необходимо освободить вызовом функции ***VirtualFree***.

Чтобы использовать выделенный регион виртуального адресного пространства (ВАП), для него необходимо также выделить физическую память, спроецировав ее на регион. Эта операция называется передачей физической памяти и осуществляется с помощью функции ***VirtualAlloc***. Для физической памяти определяют ее возврат, что выполняется с помощью функции ***VitualFree*** [7]. Обе упомянутые функции будут описаны далее.

Отдельным страницам физической памяти можно устанавливать атрибуты защиты:

PAGE_NOACCESS	Любая операция вызовет нарушение доступа.
PAGE_READONLY	Попытки записи или исполнения могут вызвать нарушение доступа.
PAGE_READWRITE	Попытки исполнить содержимое страницы вызывают нарушение доступа.
PAGE_EXECUTE	Чтение и запись могут вызвать нарушение доступа.
PAGE_EXECUTE_READ	Нарушение доступа при попытке записи.
PAGE_EXECUTE_READWRITE	Возможны любые операции.
PAGE_WRITECOPY	При исполнении этой страницы происходит нарушение доступа; при записи процессу дается личная копия страницы.

PAGE_EXECUTE_WRITECOPY	Любые операции. При записи процессу дается личная копия страницы.
PAGE_NOCACHE	Отключает кэширование страницы (флаг).
PAGE_WRITECOMBINE	Объединение нескольких операций записи (флаг).
PAGE_GUARD	Используется для получения информации о записи на какую-либо страницу (флаг).

Узнать размеры страницы, гранулярность выделения памяти и другие параметры ОС можно с помощью функции ***WinAPI***:

VOID GetSystemInfo (LPSYSTEM_INFO SysInfo);

В эту функцию в качестве параметра передается адрес структуры данных, описанной следующим образом:

```
typedef struct {
  union {
    DWORD Oem;              // не используется
    struct {
      WORD ProcArchitecture;        // тип архитектуры процессора
      WORD Reserved;                // не используется
    };
  };
  DWORD PageSize;           // размер страницы в байтах
  LPVOID MinApplnAddress; // минимальный адрес доступного ВАП
  LPVOID MaxApplnAddress; // максимальный адрес доступного ВАП
  DWORD_PTR ActiveProcessors; // процессоры, выполняющие потоки
  DWORD NumberOfProc; // количество установленных процессоров
  DWORD ProcType; // тип процессора для Windows
  DWORD Granularity; // гранулярность
  WORD ProcLevel, ProcRevizion;
```

Если есть необходимость узнать параметры, которые имеют отношение к памяти (а их всего четыре), достаточно выполнить всего два оператора:

```
SYSTEM_INFO SysInfo;
GetSystemInfo (&SysInfo);
```

После этого можно спокойно просматривать содержимое полей структуры ***SysInfo***, где и будут находиться искомые системные параметры.

Следующая функция позволяет отслеживать состояние памяти на текущий момент времени, однако она имеет нестандартное название:

VOID GlobalMemoryStatus (LPMEMORY_STATUS MemStat);

Как видно, ей необходимо передать адрес некоторой структуры, которая описана следующим образом:

```
typedef struct {
  DWORD Length;                 // размер структуры в байтах
  DWORD MemLoad;
    // Занятость подсистемы управления памятью (0–100)
  SIZE_T TotalPhysMem;          // объем физической памяти
  SIZE_T AvailablePhysMem;      // объем свободной физической памяти
  SIZE_T TotalPageFile;         // максимальный размер файла подкачки
  SIZE_T AvailablePageFile;
    // размер свободного места в файле подкачки
  SIZE_T TotalVirtual;          // максимальный размер ВАП процесса
  SIZE_T AvailableVirtual;      // размер доступного ВАП процесса
} MEMORY_STATUS, *LPMEMORY_STATUS;
```

Если нужно получить какие-то параметры памяти, потребуется еще несколько операторов, но предварительно потребуется установить длину всей структуры:

```
MEMORY_STATUS MemStat;
MemStat.Length = {sizeof (MemStat)};
GlobalMemoryStatus (&SysInfo);
```

В дальнейшем можно смело пользоваться значениями остальных полей структуры ***MemStat.***

В ОС Windows есть функция, которая позволяет получать информацию о конкретном участке памяти в пределах ВАП процесса:

```
DWORD VirtualQuery (
  LPCVOID Address,              // адрес участка памяти
  PMEMORY_BASIC_INFORMATION MemBase,
    // адрес структуры с информацией о памяти
  DWORD Length);                // размер структуры с информацией о памяти
```

Этой функции требуется адрес структуры типа *MEMORY_BASIC_INFORMATION*, описанной как:

```
typedef struct {
  PVOID Base;
   // Address, округленный до адреса, кратного размеру страницы
  PVOID AllocBase;
    // Базовый адрес региона, в который входит адрес Address
```

```
  DWORD AllocProtection; // атрибут защиты для региона
  SIZE_T RegionSize;
    // размер страниц, имеющих одинаковые атрибуты
  DWORD State; // состояние всех смежных страниц
  DWORD Protection; // атрибуты защиты всех смежных страниц
} MEMORY_BASIC_INFORMATION,
*PMEMORY_BASIC_INFORMATION;
```

В случае, если понадобится вывести информацию о регионах ВАП текущего процесса, можно это сделать следующим образом:

```
// сначала необходимо получить размер страницы для данной системы
SYSTEM_INFO SysInfo;
GetSystemInfo (&SysInfo);

PVOID BaseAddr = NULL;
MEMORY_BASIC_INFORMATION MemBase;
for (;;) {
  if(VirtalQuery(BaseAddr,&MemBase,sizeof(MemBase)) !=
    sizeof(MemBase)) break;
  printf ("%Lp\t\", MemBase.AllocBase); // Базовый адрес
  printf ("%d\t», MemBase.RegionSize / SysInfo.PageSize);
    // Страниц на регион
  switch (MemBase.State) {          // Состояние региона
    case MEM_FREE: printf ("Свободный\t"); break;
    case MEM_RESERVE: printf ("Зарезервирован,
         память не передана\t"); break;
    case MEM_COMMIT: printf ("Зарезервирован,
         память передана\t"); break;
    }
  switch (MemBase.Protection) {        // Атрибуты защиты региона
    case PAGE_NOACCESS: printf ("----\t"); break;
    case PAGE_READONLY: printf ("R---\t"); break;
    case PAGE_READWRITE: printf ("RW--\t"); break;
    case PAGE_EXECUTE: printf ("--E-\t"); break;
    case PAGE_EXECUTE_READ: printf ("R-E-\t"); break;
    case PAGE_EXECUTE_ READWRITE: printf ("RWE-\t"); break;
    case PAGE_EXECUTE_ WRITECOPY: printf ("RWEC\t"); break;
    }
  switch (MemBase.Type) { // Тип региона
```

```
    case MEM_FREE: printf (“Свободный”); break;
    case MEM_PRIVATE: printf (“Закрытый”); break;
    case MEM_IMAGE: printf (“Образ файла”); break;
    case MEM_MAPPED: printf (“Отображаемый файл”); break;
    default: printf (“Неизвестно”);
    }
  printf (“\n”);
  BaseAddr = MemBase.BaseAddress+ MemBase.RegionSize;
  }
```

Виртуальная память очень удобна для работы с большими массивами данных. Для малых по размеру объектов больше подходят так называемые кучи. Если есть надобность в обмене данными между процессами, то ОС Windows предлагает еще один механизм – отображаемые на память файлы. О них более подробно будет рассказано там, где речь пойдет о подсистеме управления файлами. Далее будут описаны функции для управления кучами. Функции, имеющие дело с виртуальной памятью, позволяют резервировать регион, отдавать ему физическую память и устанавливать параметры защиты [7].

Для резервирования предназначена функция

```
PVOID VirtualAlloc (
  PVOID Address,
    // адрес, где система должна резервировать память
  SIZE_T Size, // размер региона, который надо зарезервировать
  DWORD AllocType,
   //тип резервирования (резервировать или передать физ. память)
  DWORD Protect);   // атрибут защиты (PAGE_*)
```

Функция возвращает *NULL*, если не удалось выделить память для региона. Для резервирования достаточно при вызове указать тип ***MEM_RESERVE***. Если теперь надо передать ему физическую память, нужно еще раз вызвать ***VirtualAlloc*** с уже знакомым флагом доступа ***MEM_COMMIT***. Можно выполнить обе операции одновременно:

```
PVOID Region = VirtualAlloc (NULL, 25 * 1024,
  MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);
```

Здесь содержится запрос на выделение региона размером 25 КБ и передачу ему физической памяти. Поскольку первый параметр функции равен *NULL*, ОС попытается найти подходящее место, просматривая все ВАП. Регион и переданная ему память получат одинаковый атрибут защиты ***PAGE_EXECUTE_READWRITE.***

Для освобождения зарезервированного региона или возврата физической памяти можно пользоваться функцией

BOOL VirtualFree (
PVOID Address, *// адрес, где система зарезервировала память*
SIZE_T Size, *// размер региона, который был зарезервирован*
DWORD FreeType); *//тип освобождения*

Для освобождения региона нужно вызвать ***VirtualFree*** с его адресом, в ***Size*** указать 0, поскольку система знает размер региона, а в ***FreeType*** – ***MEM_RELEASE***. Если же возникла необходимость просто вернуть часть физической памяти, то ***Address*** должен адресовать первую возвращаемую страницу, ***Size*** – количество освобождаемых байтов, а ***FreeType*** – идентификатор ***MEM_COMMIT.***

PVOID Region;
VirtualFree (Region, 0, MEM_RELEASE);

Атрибуты защиты страницы памяти можно вызвать функцией

PVOID VirtualProtect (
PVOID Address, *// адрес, где система зарезервировала память*
SIZE_T Size, *// число байтов, для которых меняется защита*
DWORD NewProtection, *// новые атрибуты защиты*
PDWORD OldProtecttion); *//старые атрибуты защиты*

Для изменения атрибутов защиты у тех страниц, которые принадлежат разным регионам, функцию ***VirtualProtect*** использовать нельзя.

PVOID Region = VirtualAlloc (NULL, 25 * 1024,
MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);
DWORD OldProt;
VirtualProtect (Region, 3 * 1024, PAGE_READONLY, &OldProt);

Еще один механизм управления памятью – куча. Это также регион зарезервированного адресного пространства. Этому региону большая часть физической памяти не передается. В зависимости от того, что делает программа со своими данными, специализированный «менеджер куч» передает региону физическую память или возвращает страницы.

В ОС Windows при инициализации процесса в его ВАП создается стандартная куча, размер которой 1 МБ. Описатель этой кучи можно вызвать при помощи функции

HANDLE GetProcessHeap ();

Можно создать и дополнительные кучи, для этого нужна функция

HANDLE HeapCreate (

DWORD Options, *// способ выполнения операций над кучей*
SIZE_T StartSize, *// начальное количество байтов в куче*
SIZE_T MaxSize); *// максимальное количество байтов в куче*

Если в ***Options*** указан 0, то к куче могут одновременно обращаться несколько потоков. Атрибут ***HEAP_NO_SERIALIZE*** позволяет потоку осуществлять доступ к куче монопольно, однако пользоваться таким способом не рекомендуется. Другой флаг ***HEAP_GENERATE_EXCE****PTIONS* при ошибке обращения к куче дает системе возможность уведомлять программы об этом. Если в третьем параметре ***MaxSize*** указать значение больше 0, то будет создана нерасширяемая куча именно такого размера, в противном случае система резервирует регион и может расширять его до максимального размера. Данная функция в случае успеха возвращает описатель вновь созданной кучи.

Для выделения блока памяти из кучи необходимо вызвать функцию

PVOID HeapAlloc (
HANDLE Heap, *// описатель кучи*
DWORD Flags, *// флаги выделения памяти*
SIZE_T Bytes); *// количество выделяемых байтов*

Если в качестве флага указан ***HEAP_ZERO_MEMORY***, функция возвратит блок памяти, заполненный нулями.

Назначение ***HEAP_GENERATE_EXCEPTIONS*** и ***HEAP_NO_SERIALIZE*** очевидно.

Иногда требуется изменить размер выделенного блока памяти: уменьшить или увеличить. Для этого вызывается функция

PVOID HeapReAlloc (
HANDLE Heap, *// описатель кучи*
DWORD Flags, *// флаги изменения памяти*
PVOID Memory, *// текущий адрес блока*
SIZE_T Bytes); *// новый размер в байтах*

Возможны четыре значения флага: ***HEAP_NO_SERIALIZE***, ***HEAP_GENERATE_EXCEPTIONS***, ***HEAP_ZERO_MEMORY*** и ***HEAP_REALLOC_IN_PLACE_ONLY***.

Два первых из них знакомы по ***HeapAlloc.***

При использовании ***HEAP_ZERO_MEMORY*** нулями заполняются только дополнительный байты. ***HEAP_REALLOC_IN_PLACE_ONLY*** говорит о том, что блок перемещать внутри кучи нельзя. Возвращает эта функция адрес нового блока либо *NULL*, если не удалось изменить размер.

После того как выделен блок памяти, можно узнать его размер:

```
SIZE_T HeapSize (
  HANDLE Heap,      // описатель кучи
  DWORD Flags,
    // флаги изменения памяти (0 или HEAP_NO_SERIALIZE)
  PVOID Memory);    // текущий адрес блока
```

После того как блок перестал быть нужным, его освобождают функцией

```
BOOL HeapFree (
  HANDLE Heap,      // описатель кучи
  DWORD Flags,
    // флаги изменения памяти (0 или HEAP_NO_SERIALIZE)
  PVOID Memory);    // текущий адрес блока
```

В случае успеха эта функция возвращает *TRUE*. Аналогично работает функция ***HeapDestroy***, которая освобождает все блоки памяти внутри кучи и возвратит системе регион, занятый кучей.

```
BOOL HeapDestroy (HANDLE Heap);      // описатель кучи
```

Небольшой фрагмент кода демонстрирует использование некоторых из этих функций:

```
// получение описателя стандартной кучи активного процесса
HANDLE SysHeap = GetProcessHeap ();

UINT MAX_ALLOCATIONS = 15;
  // максимальное количество выделений памяти
UINT NumOfAllocations = 0;
  // текущее количество выделений памяти

for (;;) {
  // выделение из кучи 2 КБ памяти
  if (HeapAlloc (SysHeap, 0, 2 * 1024) == NULL) break;
  else ++ NumOfAllocations;
  // условие прерывания цикла
  if (NumOfAllocation == MAX_ALLOCATIONS) break;
  }

// вывод соответствующих сообщений в зависимости от ситуации
if (NumOfAllocations == 0)
    printf ("Память из кучи не выделялась.");
else printf ("Память из кучи выделялась %d раз.",
```

```
    NumOfAllocations);
```

В ОС Windows есть пара функций ***WinAPI***, которые позволяют блокировать (или зафиксировать) и разблокировать страницу в оперативной памяти. Функция ***VirtualLock*** позволяет предотвратить запись памяти на диск.

```
BOOL VirtualLock (
  LPVOID Address,   // адрес начала памяти
  SIZE_T Size);    // количество байтов
```

Если фиксация больше не нужна, то ее можно убрать функцией

```
BOOL VirtualUnlock (
  LPVOID Address,   // адрес начала памяти
  SIZE_T Size);     // количество байтов
```

Обе функции возвращают ненулевое значение в случае успеха. Следующий фрагмент демонстрирует их использование:

```
int MEMSIZE = 4096;
PVOID Mem = NULL;

int num;

Mem = VirtualAlloc(NULL,4 * 1024, MEM_RESERVE,
            PAGE_EXECUTE_READWRITE);
if (Mem != NULL) {
  if (VirtualLock (Mem, MEMSIZE)) printf ("Привязка\n");
  else printf ("Ошибка привязки");
  scanf ("%d", &num);

  if (VirtualUnlock (Mem, MEMSIZE))
     printf ("Привязка снята\n");
  else printf ("Ошибка снятия привязки\n");

  if (VirtualFree (Mem, 0, MEM_RELEASE))
     printf ("Память освобождена\n");
  else printf ("Память не освобождена\n");
  }
else printf ("Память не выделена\n");
}
```

Порядок выполнения работы

1. Ознакомиться со спецификациями функций *WinAPI* по работе с разделами виртуального адресного пространства процессов и кучами.

2. Разработать программу в соответствии с полученным заданием, в которой должны использоваться кучи или механизм захвата и освобождения разделов виртуальной памяти.

3. Написать отчет.

Варианты заданий

Вариант №1

Разработать программу, которая демонстрирует управление структурами данных типа «стек», элементы которых занимают 10 КБ. Операции, выполняемые над стеком:

- проверить, стек пуст/не пуст;
- втолкнуть элемент;
- вытолкнуть элемент;
- просмотреть вершину стека;
- обменять значения двух верхних элементов стека.

Воспользоваться механизмом управления разделами виртуальной памяти.

Вариант №2

Разработать программу, которая демонстрирует управление структурами данных типа «стек», элементы которых занимают 15 КБ. Операции, выполняемые над стеком:

- проверить, стек пуст/не пуст;
- втолкнуть элемент;
- вытолкнуть элемент;
- просмотреть вершину стека;
- продублировать вершину стека.

Воспользоваться механизмом управления разделами виртуальной памяти.

Вариант №3

Разработать программу, которая демонстрирует управление структурами данных типа «стек», элементы которых занимают 12 КБ. Операции, выполняемые над стеком:

- проверить, стек пуст/не пуст;
- втолкнуть элемент;

● вытолкнуть элемент;
● просмотреть вершину стека;
● обменять значения второго и третьего сверху элементов стека.

Воспользоваться механизмом управления разделами виртуальной памяти.

Вариант №4

Разработать программу, которая демонстрирует управление структурами данных типа «очередь», элементы которых занимают 10 КБ. Операции, выполняемые над очередью:

● проверить, очередь пуста/не пуста;
● добавить элемент в хвост очереди;
● удалить элемент из головы очереди;
● просмотреть голову очереди;
● обменять значения из головы и хвоста очереди.

Воспользоваться механизмом управления разделами виртуальной памяти.

Вариант №5

Разработать программу, которая демонстрирует управление структурами данных типа «очередь», элементы которых занимают 15 КБ. Операции, выполняемые над очередью:

● проверить, очередь пуста/не пуста;
● добавить элемент в хвост очереди;
● удалить элемент из головы очереди;
● просмотреть голову очереди;
● продублировать хвост очереди.

Воспользоваться механизмом управления разделами виртуальной памяти.

Вариант №6

Разработать программу, которая демонстрирует управление структурами данных типа «очередь», элементы которых занимают 12 КБ. Операции, выполняемые над очередью:

● проверить, очередь пуста/не пуста;
● добавить элемент в хвост очереди;
● удалить элемент из головы очереди;
● просмотреть голову очереди;
● продублировать голову очереди.

Воспользоваться механизмом управления разделами виртуальной памяти.

Вариант №7

Разработать программу, которая демонстрирует управление структурами данных типа «дек» (очередь с двумя концами), элементы которых занимают 10 КБ. Операции, выполняемые над деком:

- проверить, дек пуст/не пуст;
- добавить элемент в левый конец дека;
- добавить элемент в правый конец дека;
- удалить элемент слева;
- удалить элемент справа;
- просмотреть элемент слева;
- просмотреть элемент справа.

Воспользоваться механизмом управления разделами виртуальной памяти.

Вариант №8

Разработать программу, которая демонстрирует управление структурами данных типа «ограниченный слева дек» (очередь с двумя концами), элементы которых занимают 15 КБ. Операции, выполняемые над деком:

- проверить, дек пуст/не пуст;
- добавить элемент в левый конец дека;
- добавить элемент в правый конец дека;
- удалить элемент справа;
- просмотреть элемент справа.

Воспользоваться механизмом управления разделами виртуальной памяти.

Вариант №9

Разработать программу, которая демонстрирует управление структурами данных типа «ограниченный справа дек» (очередь с двумя концами), элементы которых занимают 12 КБ. Операции, выполняемые над деком:

- проверить, дек пуст/не пуст;
- добавить элемент в левый конец дека;
- добавить элемент в правый конец дека;
- удалить элемент слева;
- просмотреть элемент слева.

Воспользоваться механизмом управления разделами виртуальной памяти.

Вариант №10

Разработать программу, которая демонстрирует управление структурами данных типа «линейный однонаправленный список» (*L*1-*list*), элементы которых

занимают 10 КБ. Операции, выполняемые над списком (при этом определяется указатель списка и элемент списка за указателем):

- проверить, список пуст/не пуст;
- установить указатель в начало списка;
- добавить элемент за указателем;
- удалить элемент за указателем;
- просмотреть элемент за указателем;
- переместить указатель вправо.

Воспользоваться механизмом управления разделами виртуальной памяти.

Вариант №11

Разработать программу, которая демонстрирует управление структурами данных типа «линейный двунаправленный список» (*L2-list*), элементы которых занимают 15 КБ. Операции, выполняемые над списком (при этом определяется указатель списка, элемент списка за указателем и элемент до указателя):

- проверить, список пуст/не пуст;
- установить указатель в начало списка;
- установить указатель в конец списка;
- добавить элемент за указателем;
- добавить элемент до указателя;
- удалить элемент за указателем;
- удалить элемент до указателя;
- просмотреть элемент за указателем;
- просмотреть элемент до указателя;
- переместить указатель вправо;
- переместить указатель влево.

Воспользоваться механизмом управления разделами виртуальной памяти.

Вариант №12

Разработать программу, которая демонстрирует управление структурами данных типа «динамический вектор» (одномерный массив), элементы которых занимают 12 КБ. Операции, выполняемые над вектором (при этом определяются начало и конец вектора, индекс элемента вектора):

- проверить, вектор пуст/не пуст;
- прочитать элемент с указанным индексом;
- изменить значение элемента с указанным индексом;
- добавить элемент в конец вектора;
- опустошить вектор.

Воспользоваться механизмом управления разделами виртуальной памяти.

Вариант №13

Разработать программу, которая демонстрирует управление структурами данных типа «последовательность» (файл в оперативной памяти), элементы которых занимают 10 КБ. Операции, выполняемые над последовательностью (при этом определяются указатель на текущий элемент, начало и конец последовательности):

- проверить, последовательность пуста/не пуста;
- установить указатель в начало последовательности;
- прочитать элемент последовательности;
- добавить элемент в конец последовательности;
- опустошить последовательность.

Воспользоваться механизмом управления разделами виртуальной памяти.

Вариант №14

Разработать программу, которая демонстрирует управление структурами данных типа «кольцевой однонаправленный список», элементы которых занимают 15 КБ. Операции, выполняемые над списком (при этом определяется указатель списка, который может автоматически перемещаться на начало списка, если достигнут его конец, и элемент списка за указателем):

- проверить, список пуст/не пуст;
- установить указатель в начало списка;
- добавить элемент за указателем;
- удалить элемент за указателем;
- просмотреть элемент за указателем;
- переместить указатель вправо.

Воспользоваться механизмом управления разделами виртуальной памяти.

Вариант №15

Разработать программу, которая демонстрирует управление структурами данных типа «кольцевой двунаправленный список», элементы которых занимают 12 КБ. Операции, выполняемые над списком (при этом определяется указатель списка, который может автоматически перемещаться на начало списка, если достигнут его конец, и в конец списка – в случае достижения его начала, а также элемент списка за указателем и элемент до указателя):

- проверить, список пуст/не пуст;
- установить указатель в начало списка;

● установить указатель в конец списка;
● добавить элемент за указателем;
● добавить элемент до указателя;
● удалить элемент за указателем;
● удалить элемент до указателя;
● просмотреть элемент за указателем;
● просмотреть элемент до указателя;
● переместить указатель вправо;
● переместить указатель влево.

Воспользоваться механизмом управления разделами виртуальной памяти.

Вариант №16

Разработать программу, которая демонстрирует управление структурами данных типа «двоичное упорядоченное дерево», элементы которых занимают 10 КБ. Операции, выполняемые над деревом (при этом определяется один узел дерева, являющийся его корнем, все значения в узлах дерева разные):

● проверить, дерево пусто/не пусто;
● добавить элемент в дерево;
● удалить элемент из дерева;
● найти элемент с заданным значением;
● опустошить дерево.

Воспользоваться механизмом управления разделами виртуальной памяти.

Вариант №17

Разработать программу, которая демонстрирует управление структурами данных типа «сбалансированное двоичное упорядоченное дерево», элементы которых занимают 15 КБ. Операции, выполняемые над деревом (при этом определяется один узел дерева, являющийся его корнем, все значения в узлах дерева разные, длины путей от корня до всех узлов-листьев отличаются не более чем на единицу):

● проверить, дерево пусто/не пусто;
● добавить элемент в дерево;
● удалить элемент из дерева;
● найти элемент с заданным значением;
● опустошить дерево.

Воспользоваться механизмом управления разделами виртуальной памяти.

Вариант №18

Разработать программу, которая демонстрирует управление структурами данных типа «неупорядоченное 2-3-дерево», элементы которых занимают 12 КБ.

Операции, выполняемые над деревом (при этом определяется один узел дерева, являющийся его корнем, все значения в узлах дерева разные, количество потомков каждого узла – не более трех):

- проверить, дерево пусто/не пусто;
- добавить элемент в дерево;
- удалить элемент из дерева;
- найти элемент с заданным значением;
- опустошить дерево.

Воспользоваться механизмом управления разделами виртуальной памяти.

Вариант №19

Разработать программу, которая демонстрирует управление структурами данных типа «линейный двунаправленный список» (*L2-list*), элементы которых занимают 10 КБ. Операции, выполняемые над списком (при этом определяется указатель списка, элемент списка за указателем и элемент до указателя):

- проверить, список пуст/не пуст;
- установить указатель в начало списка;
- установить указатель в конец списка;
- обменять значения элементов за указателем и до указателя, если это возможно;
- добавить элемент за указателем;
- добавить элемент до указателя;
- удалить элемент за указателем;
- удалить элемент до указателя;
- просмотреть элемент за указателем;
- просмотреть элемент до указателя;
- переместить указатель вправо;
- переместить указатель влево.

Воспользоваться механизмом управления разделами виртуальной памяти.

Вариант №20

Разработать программу, которая демонстрирует управление структурами данных типа «линейный однонаправленный список» (*L1-list*), элементы которых занимают 15 КБ. Операции, выполняемые над списком (при этом определяется указатель списка и элемент списка за указателем):

- проверить, список пуст/не пуст;
- установить указатель в начало списка;
- добавить элемент за указателем;
- удалить элемент за указателем;

● просмотреть элемент за указателем;
● переместить указатель вправо;
● обменять значения начала списка и элемента за указателем.

Воспользоваться механизмом управления разделами виртуальной памяти.

Вариант №21

Разработать программу, которая демонстрирует управление структурами данных типа «линейный однонаправленный список» (*L*1-*list*), элементы которых занимают 12 КБ. Операции, выполняемые над списком (при этом определяется указатель списка и элемент списка за указателем):

● проверить, список пуст/не пуст;
● установить указатель в начало списка;
● добавить элемент за указателем;
● удалить элемент за указателем;
● просмотреть элемент за указателем;
● переместить указатель вправо;
● обменять значения конца списка и элемента за указателем.

Воспользоваться механизмом управления разделами виртуальной памяти.

Вариант №22

Разработать программу, которая демонстрирует управление структурами данных типа «стек», элементы которых занимают 10 КБ. Операции, выполняемые над стеком:

● проверить, стек пуст/не пуст;
● втолкнуть элемент;
● вытолкнуть элемент;
● просмотреть вершину стека;
● обменять значения первого и третьего сверху элементов стека.

Воспользоваться механизмом управления разделами виртуальной памяти.

Вариант №23

Разработать программу, которая демонстрирует управление структурами данных типа «дек» (очередь с двумя концами), элементы которых занимают 15 КБ. Операции, выполняемые над деком:

● проверить, дек пуст/не пуст;
● добавить элемент в левый конец дека;
● добавить элемент в правый конец дека;
● удалить элемент слева;

● удалить элемент справа;
● просмотреть элемент слева;
● просмотреть элемент справа;
● обменять значениями элементы на концах дека.

Воспользоваться механизмом управления разделами виртуальной памяти.

Вариант №24

Разработать программу, которая демонстрирует управление структурами данных типа «динамический вектор» (одномерный массив), элементы которых занимают 12 КБ. Операции, выполняемые над вектором (при этом определяются начало и конец вектора, индекс элемента вектора):

● проверить, вектор пуст/не пуст;
● прочитать элемент с указанным индексом;
● изменить значение элемента с указанным индексом;
● добавить элемент в конец вектора;
● опустошить вектор;
● обменять значениями текущий элемент и конец вектора.

Воспользоваться механизмом управления разделами виртуальной памяти.

Вариант №25

Разработать программу, которая демонстрирует управление структурами данных типа «последовательность» (файл в оперативной памяти), элементы которых занимают 10 КБ. Операции, выполняемые над последовательностью (при этом определяются указатель на текущий элемент, начало и конец последовательности):

● проверить, последовательность пуста/не пуста;
● установить указатель в начало последовательности;
● прочитать элемент последовательности;
● добавить элемент в конец последовательности;
● опустошить последовательность;
● обменять значениями текущий элемент и конец последовательности.

Воспользоваться механизмом управления разделами виртуальной памяти.

Содержание отчета

Отчет по лабораторной работе должен быть оформлен в соответствии с требованиями, предъявляемыми к текстовым документам, и содержать:

- цель работы;

- краткие теоретические сведения, необходимые для выполнения работы;

- описание порядка выполнения работы с приведением листинга кода, скриншотов рабочих окон;

- выводы по работе.

Лабораторная работа №10

Использование механизма обмена сообщениями для управления окнами в ОС Windows

Цель – реализация механизма обмена сообщениями в операционной системе Windows.

Теоретическая часть

Программы в ОС Windows управляются сообщениями. Все действия пользователей перехватываются системой и преобразуются в сообщения, направляемые программе, которая владеет окнами, к которым обращены действия пользователя. У каждой программы, состоящей хотя бы из одного потока, есть несколько собственных очередей сообщений: очередь асинхронных сообщений, очередь синхронных сообщений, очередь ответных сообщений и очередь виртуального ввода, куда и направляются все сообщения, касающиеся действий с окнами. В каждой программе есть главный цикл, который состоит из получения следующего сообщения и его обработки с помощью внутренней процедуры, соответствующей данному типу сообщений. Иногда некоторые сообщения вызывают эти процедуры, минуя очереди сообщений [7].

Практически каждая программа в ОС Windows будет содержать код, похожий на следующий:

```
#include <windows.h>

long CALLBACK WndProcedure (HWND Window, UINT Mess, UINT Parametr, long Prm)
{
// обработка сообщений
switch (Mess) {
  case WM_PAINT: ; return 0;          // перерисовка содержимого окна
  case WM_CREATE: ; return 0;         // создание окна
  case WM_CLOSE: DestroyWindow (Window); return 0;
                    // закрытие окна
  case WM_DESTROY: PostQuitMessage (0); return 0;
                    // удаление окна
  }
return (DefWindowProc (Window, Mess, Parametr, Prm));
                    // действия по умолчанию
}

int WINAPI WinMain (HINSTANCE h, HINSTANCE hprev, char *CmdLine, int CmdShow)
```

```
{
WNDCLASS WinClass;                    // объект окна
MSG Mess;                             // входящие сообщения
HWND Window;                          // дескриптор окна
WinClass.lpfnWndProc = WndProcedure;
WinClass.lpszClassName = "Okoshko";
WinClass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
WinClass.hCursor = LoadCursor (NULL, IDC_ARROW);
  // курсор мыши – стрелка
  // перерисовка всего окна при изменении высоты и ширины окна
WinClass.Style = CS_HREDRAW | CS_VREDRAW;
  // зарегистрировать в системе объект окна
  // здесь нет проверки возвращаемого значения
RegisterClass (&WinClass);
  // запрос ресурсов для окна
Window = CreateWindow (WinClass.lpszClassName, «Zagolovok okna",
  WS_OVERLAPPEDWINDOW, 0, 0, 600, 400, HWND_DESKTOP, NULL,
h,
  NULL);
ShowWindow (Window, CmdShow);         // отображение окна
UpdateWindow (Window);                // перерисовка окна
while (GetMessage (&Mess, NULL, 0, 0)) {
  // получение сообщения из очереди
  TranslateMessage (&Mess);           // трансляция сообщения
  DispatchMessage (&Mess);
    // сообщение соответствующей процедуре
  }
return (Mess.wParam);
}
```

В ОС Windows прежде чем создавать окно, его нужно зарегистрировать. Регистрацию класса окна и производит функция

```
ATOM RegisterClass (CONST WNDCLASS *WndClass);
  // указатель на структуру с данными класса
```

- *WS_BORDER* – создание окна с рамкой.
- *WS_CAPTION*– создание окна с заголовком (невозможно использовать одновременно со стилем *WS_DLGFRAME*).
- *WS_CHILDWINDOW*– создание дочернего окна (невозможно использовать одновременно со стилем *WS_POPUP*).
- *WS_CLIPCHILDREN* – исключает область, занятую дочерним окном, при выводе в родительское окно.
- *WS_CLIPSIBLINGS* – используется совместно со стилем *WS_CHILD* для отрисовки в дочернем окне областей, перекрываемых другими окнами.
- *WS_DISABLED* – создает окно, которое недоступно.

- *WS_DLGFRAME* – создает окно с двойной рамкой без заголовка.
- *WS_GROUP* – позволяет объединять элементы управления в группы.
- *WS_HSCROLL* – создает окно с горизонтальной полосой прокрутки.
- *WS_MAXIMIZE* – создает окно максимального размера.
- *WS_MAXIMIZEBOX* – создает окно с кнопкой развертывания окна.
- *WS_ICONIC* – создает первоначально свернутое окно (используется только со стилем *WS_OWERLAPPED*).
- *WS_MINIMIZEBOX* – создает окно с кнопкой свертывания.
- *WS_OVERLAPPED* – создает перекрывающееся окно, которое, как правило, имеет заголовок и *WS_TILED* рамку.
- *WS_OVERLAPPEDWINDOW* – создает перекрывающееся окно, имеющее стили *WS_OVERLAPPED*, *WS_CAPTION*, *WS_SYSMENU*, WS_THICKFRAME, *WS_MINIMIZEBOX*, *WS_MAXIMIZEBOX*.
- *WS_POPUP* – создает всплывающее окно (невозможно использовать совместно со стилем *WS_CHILD*).
- *WS_POPUPWINDOW* – создает всплывающее окно, имеющее стили *WS_BORDER*, *WS_POPUP*, *WS_SYSMENU*.
- *WS_SYSMENU* – создает окно с кнопкой системного меню (можно использовать только с окнами, имеющими строку заголовка).
- *WS_TABSTOP* – спределяет элементы управления, переход к которым может быть выполнен при помощи клавиши *TAB*.
- *WS_THICKFRAME* – создает окно с рамкой, используемой для изменения.
- *WS_SIZEBOX* – задает размера окна.
- *WS_VISIBLE* – создает первоначально неотображаемое окно.
- *WS_VSCROLL* – создает окно с вертикальной полосой прокрутки.

Функция *CreateWindow* используется программой для создания окна (пример ее использования приводился ранее).

```
HWND CreateWindow (
  LPCTSTR ClassName,
    // указатель на зарегистрированное имя класса
  LPCTSTR WindowName,        // указатель на имя окна
  DWORD dwStyle,             // стиль окна
  int x,                     // горизонтальная позиция окна
  int y,                     // вертикальная позиция окна
  int Width,                 // ширина окна
  int Height,                // высота окна
  HWND WndParent,        // дескриптор родительского окна или окна владельца
  HMENU Menu,
    // дескриптор меню или идентификатор дочернего окна
  HANDLE Instance,   // дескриптор экземпляра программы
```

LPVOID Param); *// указатель на данные создания окна*

Следующие предопределенные классы элементов управления могут быть определены в параметре ***ClassName***:

• *BUTTON* (КНОПКА). Обозначает маленькое прямоугольное дочернее окно, которое представляет собой кнопку и пользователь может щелкать по ней кнопкой мыши, чтобы включить или отключить ее.

• *COMBOBOX* (КОМБИНИРОВАННОЕ ОКНО). Обозначает элемент управления, состоящий из окна со списком и поля выбора, похожего на элемент редактирования текста. При использовании этого стиля прикладная программа должна отображать все время или окно со списком или включать раскрывающийся список.

• *EDIT* (ОКНО РЕДАКТИРОВАНИЯ). Обозначает прямоугольное дочернее окно, внутри которого пользователь может напечатать с клавиатуры текст. Пользователь выбирает элемент управления и дает ему фокус клавиатуры, щелкая кнопкой мыши по нему или перемещаясь в него путем нажатия клавиши ТАБУЛЯЦИИ (*TAB*). Пользователь может напечатать текст, когда элемент управления редактируемого окна отображает мигающую каретку (*caret*); используйте мышь, чтобы перемещать курсор и выбирать символы, которые будут заменены, или установите курсор для вставки символов, или используйте *BACKSPACE*, чтобы удалять символы. Элементы управления редактируемого окна используют шрифт системы с переменным шагом и показывают на экране символы из символьного набора *ANSI*.

• *LISTBOX* (ОКНО СО СПИСКОМ). Обозначает список строк символов. Этот элемент управления определяется всякий раз, когда прикладная программа должна представить список наименований, типа имен файлов, из которых пользователь может выбирать. Пользователь может выбрать строку, щелкая по ней кнопкой мыши. Выбранная строка выделяется, а уведомительное сообщение передается в родительское окно. Чтобы листать списки, которые являются слишком длинными для элемента управления окна, используется вертикальная или горизонтальная линейка прокрутки окна со списком.

• *MDICLIENT*. Обозначает рабочее окно МНОГОДОКУМЕНТНОГО ИНТЕРФЕЙСА (*MDI*). Это окно принимает сообщения, которые управляют дочерними окнами прикладной программы МНОГОДОКУМЕНТНОГО ИНТЕРФЕЙСА.

• *SCROLLBAR* (ЛИНЕЙКА ПРОКРУТКИ). Обозначает прямоугольник, который содержит бегунок и имеет стрелки, направленные в оба конца. Линейка прокрутки посылает уведомительное сообщение своему родительскому

окну всякий раз, когда пользователь щелкает кнопкой мыши по элементу управления. В случае необходимости родительское окно ответственно за модификацию позиции бегунка.

• *STATIC* (СТАТИЧЕСКИЙ ЭЛЕМЕНТ). Обозначает простое текстовое поле, окно или прямоугольник, используемый для надписей, окно или другие отдельные элементы управления. Статические элементы управления не берут никакой вводимой информации и не обеспечивают никакой выводимой информации.

Перечислим стили кнопок (в классе *BUTTON*), которые могут быть определены в параметре ***dwStyle***:

• *BS_3STATE* – создает кнопку, которая является такой же, как окошко для флажка, за исключением того, что поле окна может стать недоступным, так же как это делается при установке флажка («галочки») проверки (*checked*) или при отмене его. Используйте недоступное состояние, чтобы показать, что состояние окошка для флажка не определено.

• *BS_AUTO3STATE* – создает кнопку, которая является таким же переключателем с тремя состояниями, за исключением того, что поле окна изменяет свое состояние, когда пользователь выбирает его. Состояние циклически проходит фазы установки флажка проверки, недоступности и отмены установки.

• *BS_AUTOCHECKBOX* – создает кнопку, которая также является окошком для флажка, за исключением того, что состояние установки флажка проверки автоматически переключается между установленным и неустановленным параметром каждый раз, когда пользователь выбирает эту кнопку.

• *BS_AUTORADIOBUTTON* – создает кнопку, которая то же, что и «радиокнопка», за исключением того, что, когда пользователь выбирает ее, Windows автоматически устанавливает состояние кнопки в режим контроля флажка, отметив ее «галочкой», и автоматически устанавливает проверку состояния для всех других кнопок в той же самой группе без проверки флажка.

• *BS_CHECKBOX* – создает маленькое, пустое окошко для флажка с текстом. По умолчанию текст отображается справа от окошка. Чтобы отображать текст слева от окошка, объедините этот флажок со стилем *BS_LEFTTEXT* (или с эквивалентным стилем *BS_RIGHTBUTTON*).

• *BS_DEFPUSHBUTTON* – создает командную кнопку, которая ведет себя подобно кнопке стиля *BS_PUSHBUTTON* и к тому же имеет черную рамку, выделенную полужирным шрифтом. Если кнопка находится в диалоговом окне, пользователь может выбрать кнопку, нажав клавишу *ENTER*, даже тогда, когда кнопка не имеет фокуса ввода.

- *BS_GROUPBOX* – создает прямоугольник, в котором могут быть сгруппированы другие элементы управления. Любой текст, связанный с этим стилем, отображается в верхнем левом углу прямоугольника.
- *BS_LEFTTEXT* – помещает текст слева от «радиокнопки» или окошечка-переключателя, когда объединен со стилем переключателя или «радиокнопкой». То же самое, что и стиль *BS_RIGHTBUTTON*.
- *BS_OWNERDRAW* – создает кнопку, представляемую владельцем. Окно владельца принимает сообщение *WM_MEASUREITEM*, когда кнопка создана, и сообщение *WM_DRAWITEM*, когда внешний вид кнопки изменился. Не объединяйте стиль *BS_OWNERDRAW* с любыми другими стилями кнопки.
- *BS_PUSHBUTTON* – создает командную кнопку, которая отправляет сообщение *WM_COMMAND* окну владельца, когда пользователь выбирает эту кнопку.
- *BS_RADIOBUTTON* – создает маленький кружок с текстом. По умолчанию текст отображается справа от кружка. Чтобы отображать текст слева от кружка, объедините этот флажок со стилем *BS_LEFTTEXT* (или его эквивалентом – стилем *BS_RIGHTBUTTON*). Используйте «радиокнопки» для групп связанного, но взаимоисключающего выбора.
- *BS_BITMAP* – определяет, что кнопка отображает точечный рисунок.
- *BS_BOTTOM* – помещает текст внизу прямоугольника кнопки.
- *BS_CENTER* – выравнивает текст горизонтально по центру в прямоугольнике кнопки.
- *BS_ICON* – определяет, что кнопка отображается как значок.
- *BS_LEFT* – выравнивает текст слева в прямоугольнике кнопки. Однако, если кнопка – это окошечко-переключатель или «радиокнопка», которые не имеют стиля *BS_RIGHTBUTTON*, текст остается выровненным справа от переключателя или «радиокнопки».
- *BS_MULTILINE* – переносит по словам текст кнопки в дополнительные строки, если текстовая строка слишком длинна, чтобы поместиться в одной строке в прямоугольнике кнопки.
- *BS_NOTIFY* – дает возможность кнопке послать уведомительные сообщения *BN_DBLCLK, BN_KILLFOCUS* и *BN_SETFOCUS* в ее родительское окно. Обратите внимание, что кнопки посылают уведомительное сообщение *BN_CLICKED* независимо от того, имеет ли она этот стиль.
- *BS_PUSHLIKE* – создает кнопку (типа переключателя, переключателя с тремя состояниями или «радиокнопки»), имеющую вид и действующую подобно командной кнопке. У кнопки выпуклый вид, когда она не нажата или не выбрана, и притопленный, когда она нажата или выбрана.

- *BS_RIGHT* – выравнивает справа текст в прямоугольнике кнопки. Однако, если кнопка – это окошко для флажка или «радиокнопка», которые не имеют стиля *BS_RIGHTBUTTON*, текст выровнен по правому краю справа от окошка для флажка или «радиокнопки».
- *BS_TEXT* – определяет, что кнопка отображает текст.
- *BS_TOP* – размещает текст вверху прямоугольника кнопки.
- *BS_VCENTER* – размещает текст в середине (вертикально) прямоугольника кнопки.

Перечислим стили комбинированного окна (в классе *COMBOBOX*), которые могут быть определены в параметре ***dwStyle***:

- *CBS_AUTOHSCROLL* – автоматически прокручивает текст в поле редактирования текста вправо, когда пользователь вводит с клавиатуры символ в конце строки. Если этот стиль не установлен, принимается только текст, который помещается внутри прямоугольной границы поля.
- *CBS_DISABLENOSCROLL* – в окне со списком показывает вертикальную линейку прокрутки заблокированной, когда поле окна содержит недостаточно элементов для прокрутки. Без этого стиля линейка прокрутки скрыта, если окно со списком содержит недостаточно элементов.
- *CBS_HASSTRINGS* – определяет, что представляемое владельцем комбинированное окно содержит элементы, состоящие из строк. Комбинированное окно поддерживает память и адрес для строк так, что прикладная программа может использовать сообщение *CB_GETLBTEXT*, чтобы восстановить текст для отдельного элемента.
- *CBS_LOWERCASE* – преобразовывает в нижний регистр любые символы верхнего регистра, введенные в поле редактирования текста комбинированного окна.
- *CBS_NOINTEGRALHEIGHT* – определяет, что размер комбинированного окна – это точный размер, определенный прикладной программой, когда она создала комбинированное окно. Обычно Windows устанавливает размеры комбинированного окна так, чтобы оно не отображало элементы частично.
- *CBS_OEMCONVERT* – преобразует текст, введенный в поле редактирования текста комбинированного окна. Текст преобразуется из набора символов Windows в набор символов *OEM*, а затем обратно в набор Windows.
- *CBS_SIMPLE* – всегда отображает окно со списком. Текущий выбор в окне со списком отображается в поле редактирования текста.
- *CBS_SORT* – автоматически сортирует строки, введенные в окно со списком.
- *CBS_UPPERCASE* – преобразовывает любые символы нижнего регистра

в символы верхнего регистра, введенные в поле редактирования текста комбинированного окна.

Перечислим стили поля редактирования текста (в классе *EDIT*), которые могут быть определены в параметре ***dwStyle***:

- *ES_AUTOHSCROLL* – автоматически прокручивает текст вправо на 10 символов, когда пользователь напечатает символ в конце строки. Когда пользователь нажимает клавишу *ENTER*, управление прокручивает весь текст обратно, чтобы установить нуль.
- *ES_AUTOVSCROLL* – автоматически перемещает текст вверх на одну страницу, когда пользователь нажимает клавишу *ENTER* на последней строке.
- *ES_CENTER* – выравнивает по центру текст в многостроковом поле редактирования текста.
- *ES_LEFT* – выравнивает текст слева.
- *ES_LOWERCASE* – преобразовывает все символы в нижний регистр, поскольку они печатаются внутри поля редактирования текста.
- *ES_MULTILINE* – обозначает многостроковое окно редактирования текста. Значение по умолчанию – одностроковое окно редактирования текста.
- *ES_NOHIDESEL* – отрицает заданное по умолчанию поведение для поля редактирования текста.
- *ES_NUMBER* – позволяет ввести в поле редактирования только цифры.
- *ES_OEMCONVERT* – преобразует текст, введенный в окно редактирования. Текст преобразуется из набора символов Windows – в набор символов *OEM*, а затем обратно – в набор Windows. Это гарантирует соответствующее символьное преобразование, когда из прикладной программы вызывается функция *CharToOem*, чтобы преобразовать строку Windows в окне редактирования в символы *OEM*. Этот стиль наиболее полезен для окон редактирования текста, которые содержат имена файлов.
- *ES_PASSWORD* – отображает звездочку (*) вместо каждого символа, введенного с клавиатуры в окно редактирования; можно использовать сообщение *EM_SETPASSWORDCHAR*, чтобы заменить ею символ, который отображается.
- *ES_READONLY* – не допускает пользователя к вводу или редактированию текста в окне редактирования.
- *ES_RIGHT* – выравнивает по правому краю текст в многострочном окне редактирования.
- *ES_UPPERCASE* – преобразует все символы в символы верхнего регистра, когда они вводятся в окно редактирования.
- *ES_WANTRETURN* – определяет, чтобы служебный код возврата каретки

был вставлен тогда, когда пользователь нажимает клавишу *ENTER* при вводе текста в многострочное поле редактирования текста в диалоговом окне. Если не определить этот стиль, то при нажатии клавиши *ENTER* получится тот же самый эффект, как при нажатии заданной по умолчанию командной кнопки диалогового окна. Этот стиль не имеет никакого влияния в однострочном окне редактирования.

Следующие стили элемента управления окна со списком (в классе *LISTBOX*) могут быть определены в параметре ***dwStyle***:

- *LBS_DISABLENOSCROLL* – показывает заблокированную вертикальную линейку прокрутки в окне со списком, когда поле окна не содержит достаточно элементов для прокрутки. Если не определить этот стиль, линейка прокрутки будет скрыта, когда окно со списком не содержит достаточно элементов.
- *LBS_EXTENDEDSEL* – позволяет многочисленным элементам быть выбранным при помощи использования клавиши *SHIFT* и мыши или специальной комбинации клавиш.
- *LBS_HASSTRINGS* – определяет, что окно со списком содержит элементы, состоящие из строк. Окно со списком сохраняет память и адреса строк так, что прикладная программа может использовать сообщение *LB_GETTEXT*, чтобы восстановить текст для отдельного элемента. По умолчанию все окна со списком за исключением окон со списком, предоставленных владельцем, имеют этот стиль.
- *LBS_MULTICOLUMN* – определяет многостолбцовое окно со списком, которое прокручивается горизонтально. Сообщение *LB_SETCOLUMNWIDTH* устанавливает ширину столбцов.
- *LBS_MULTIPLESEL* – включает или выключает выбор последовательности символов каждый раз, когда пользователь одним или двойным щелчком мыши активизирует строку символов в окне со списком. Пользователь может выбрать любое число строк.
- *LBS_NODATA* – определяет «отсутствие данных» в окне со списком. Этот стиль определяется тогда, когда число элементов в окне со списком может превысить одну тысячу.
- *LBS_NOINTEGRALHEIGHT* – определяет, что размер окна со списком соответствует размеру, определенному прикладной программой во время создания окна со списком.
- *LBS_NOREDRAW* – определяет, что вид окна со списком не модифицируется, когда производятся изменения. Можно в любое время изменить этот стиль, посылая сообщение *WM_SETREDRAW*.

- *LBS_NOSEL* – определяет, что окно со списком содержит элементы, которые могут просматриваться, но не выбираться.
- *LBS_NOTIFY* – сообщает родительскому окну о входящем сообщении всякий раз, когда пользователь щелкает кнопкой мыши или дважды щелкает по строке в окне списка.
- *LBS_OWNERDRAWFIXED* – определяет, что владелец окна со списком ответственен за прорисовку его содержания и что элементы в окне со списком появляются одинаковой высоты. Окно владельца принимает сообщение *WM_MEASUREITEM*, когда окно со списком создано, а сообщение *WM_DRAWITEM*, когда внешний вид окна изменился.
- *LBS_OWNERDRAWVARIABLE* – определяет, что владелец окна со списком ответственен за прорисовку его содержания и что элементы в окне со списком появляются переменными по высоте. Окно владельца принимает сообщение *WM_MEASUREITEM* для каждого элемента в окне со списком, когда оно создано, а сообщение *WM_DRAWITEM*, когда внешний вид окна изменился.
- *LBS_SORT* – сортирует строки в окне со списком по алфавиту.
- *LBS_STANDARD* – сортирует строки в окне со списком в алфавитном порядке. Родительское окно принимает входящее сообщение всякий раз, когда пользователь щелкает кнопкой мыши или дважды щелкает по строке. Окно со списком имеет рамку со всех сторон.
- *LBS_WANTKEYBOARDINPUT* – определяет, что владелец окна списка принимает сообщения *WM_VKEYTOITEM* всякий раз, когда пользователь нажимает клавишу, а окно со списком имеет фокус ввода. Это дает возможность прикладной программе выполнить специальную обработку при вводе с клавиатуры.

Следующие стили линейки прокрутки (в классе *SCROLLBAR*) могут быть определены в параметре ***dwStyle***:

- *SBS_BOTTOMALIGN* – выравнивает нижнюю кромку линейки прокрутки с нижней кромкой прямоугольника, определенного параметрами *x*, *y*, *Width* и *Height*. Линейка прокрутки имеет заданную по умолчанию высоту для системных линеек прокрутки. Используйте этот стиль со стилем *SBS_HORZ*.
- *SBS_HORZ* – обозначает горизонтальную линейку прокрутки. Если не определен ни стиль *SBS_BOTTOMALIGN*, ни стиль *SBS_TOPALIGN*, линейка прокрутки имеет высоту, ширину и позицию, определенные *x, y, Width* и *Height*.
- *SBS_LEFTALIGN* – выравнивает левый край линейки прокрутки с левым краем прямоугольника, определенного параметрами *x, y, Width* и *Height*. Линейка прокрутки имеет заданную по умолчанию ширину для системных линеек прокрутки. Используйте этот стиль со стилем *SBS_VERT*.

- *SBS_RIGHTALIGN* – выравнивает правый край линейки прокрутки с правым краем прямоугольника, определенного параметрами *x, y, Width* и *Height*. Линейка прокрутки имеет заданную по умолчанию ширину для системных линеек прокрутки. Используйте этот стиль со стилем *SBS_VERT*.
- *SBS_SIZEBOX* – обозначает размер окна. Если не определен ни стиль *SBS_SIZEBOXBOTTOMRIGHTALIGN*, ни стиль *SBS_SIZEBOXTOPLEFTALIGN*, размер окна имеет высоту, ширину и позицию, определенные параметрами *x, y, Width* и *Height*.
- *SBS_SIZEBOXBOTTOMRIGHTALIGN* – выравнивает размер нижнего правого угла окна с нижним правым углом прямоугольника, определенного параметрами *x, y, Width* и *Height*. Размер окна имеет заданный по умолчанию размер для системы размера окон. Используйте этот стиль со стилем *SBS_SIZEBOX*.
- *SBS_SIZEBOXTOPLEFTALIGN* – выравнивает размер верхнего левого угла окна с левым верхним углом прямоугольника, определенного параметрами *x, y, Width* и *Height*. Размер окна имеет заданный по умолчанию размер для системы размера окон. Используйте этот стиль со стилем *SBS_SIZEBOX*.
- *SBS_SIZEGRIP* – подобен стилю *SBS_SIZEBOX*, но с выпуклой рамкой.
- *SBS_TOPALIGN* – выравнивает верхний край линейки прокрутки с верхним краем прямоугольника, определенного параметрами *x, y, Width* и *Height*. Линейка прокрутки имеет заданную по умолчанию высоту для системы линеек прокрутки. Используйте этот стиль со стилем *SBS_HORZ*.
- *SBS_VERT* – обозначает вертикальную линейку прокрутки. Если не определен ни стиль *SBS_RIGHTALIGN*, ни стиль *SBS_LEFTALIGN*, линейка прокрутки имеет высоту, ширину и позицию, определенные параметрами *x, y, Width* и *Height*.

Следующие стили статического элемента управления (в классе *STATIC*) могут быть определены в параметре ***dwStyle***. Статический элемент управления может иметь только один из этих стилей:

- *SS_BITMAP* – определяет, что в статическом элементе управления должен отобразиться точечный рисунок. Текст кода ошибки – имя точечного рисунка (не имя файла) – определен в другом месте файла ресурса. Стиль игнорирует параметры *Width* и *Height*; элемент управления автоматически устанавливает собственные размеры, чтобы поместить точечный рисунок.
- *SS_BLACKFRAME* – определяет окно с рамкой, использующей тот же самый цвет, как и у рамки основного окна. Этот цвет черный по умолчанию в системе цветов Windows.
- *SS_BLACKRECT* – определяет прямоугольник, заполненный текущим цветом рамки окна. По умолчанию этот цвет черный в системе цветов Windows.

- *SS_CENTER* – определяет простой прямоугольник и выравнивает по центру текст кода ошибки в прямоугольнике. Текст форматируется перед отображением его на экране. Слова, которые выходят за пределы конца строки, автоматически переносятся в начало следующей центрированной строки.
- *SS_GRAYFRAME* – определяет поле окна с рамкой, выведенной тем же самым цветом, что и экранный фон (рабочий стол). По умолчанию в системе цветов Windows этот цвет серый.
- *SS_GRAYRECT* – определяет прямоугольник, заполненный текущим экранным цветом фона. По умолчанию в системе цветов Windows этот цвет серый.
- *SS_ICON* – определяет пиктограмму, отображаемую в диалоговом окне. Данный текст – имя пиктограммы (не имя файла) – определен в другом месте файла ресурса. Стиль игнорирует параметры *nWidth* и *nHeight*; пиктограмма автоматически устанавливает свою величину.
- *SS_LEFT* – определяет простой прямоугольник и выравнивание по левому краю текста, помещенного в прямоугольнике. Текст форматируется перед его отображением. Слова, которые выходят за пределы конца строки, автоматически переносятся в начало следующей выровненной по левой границе строки.
- *SS_LEFTNOWORDWRAP* – определяет простой прямоугольник и выравнивание по левому краю текста, помещенного в прямоугольнике. Планшеты расширяются, но слова не переносятся. Текст, который выходит за пределы конца строки, отсекается.
- *SS_METAPICT* – определяет, что изображение метафайла должно отобразиться в статическом элементе управления. Данный текст – имя изображения метафайла (не имя файла) – определен в другом месте в файле ресурса. Статический элемент управления метафайла имеет фиксированный размер; изображение метафайла масштабируется, чтобы приспособить рабочую область статического элемента управления.
- *SS_NOPREFIX* – предотвращает интерпретацию любого символа амперсанта (&) в тексте элемента управления как символа префикса акселератора. Они отображаются с удаленным амперсантом и следующим за ним подчеркнутым символом в строке. Этот стиль статического элемента управления может быть включен с любым из определенных статических элементов управления.
- *SS_NOTIFY* – посылает родительскому окну уведомительные сообщения *STN_CLICKED* и *STN_DBLCLK*, когда пользователь щелкает кнопкой мыши или дважды щелкает по элементу управления.

- *SS_RIGHT* – определяет простой прямоугольник и выравнивание по правому краю текста, помещенного в прямоугольнике. Текст форматируется перед его отображением на экране. Слова, которые выходят за пределы конца строки, автоматически переносятся в начало следующей выровненной по правой границе строки.
- *SS_RIGHTIMAGE* – определяет, что угол правой нижней части статического элемента управления со стилем *SS_BITMAP* или *SS_ICON* должен остаться фиксированным, когда элемент управления изменяется. Только верхняя и левая стороны корректируются, чтобы поместить новый точечный рисунок или пиктограмму.
- *SS_SIMPLE* – определяет простой прямоугольник и отображает одиночную строку выровненного по левой границе текста в прямоугольнике. Текстовая строка не может быть сокращена или изменена в любом случае. Родительское окно панели управления или диалоговое окно не должно обрабатывать сообщение *WM_CTLCOLORSTATIC*.
- *SS_WHITEFRAME* – определяет поле окна с рамкой, выведенной тем же самым цветом, как фон окна. По умолчанию в системе цветов Windows этот цвет белый.
- *SS_WHITERECT* – определяет прямоугольник, заполненный текущим цветом фона окна. По умолчанию в системе цветов *Windows* – этот цвет белый.

Перечислим стили диалогового окна, которые могут быть определены в параметре ***dwStyle***:

- *DS_3DLOOK* – обеспечивает диалоговое окно обычным (не полужирным) шрифтом и выводит трехмерные рамки вокруг элементов управления окна в блоке диалога.
- *DS_ABSALIGN* – указывает, что координаты диалогового окна – экранные координаты; иначе Windows принимает их за координаты пользователя.
- *DS_CENTER* – выравнивает по центру диалоговое окно в рабочей области, т. е. в области, не загораживаемой панелью.
- *DS_CENTERMOUSE* – выравнивает по центру курсор мыши в диалоговом окне.
- *DS_CONTEXTHELP* – включает вопросительный знак в строке заголовка диалогового окна. Когда пользователь щелкает кнопкой мыши по вопросительному знаку, курсор изменятся на вопросительный знак со стрелкой-указателем. Если пользователь затем щелкает кнопкой мыши по элементу управления в диалоговом окне, элемент управления принимает сообщение *WM_HELP*. Элемент управления должен передать сообщение для диалоговой процедуры, которая

должна вызвать функцию *WinHelp*, использующую команду *HELP_WM_HELP*.

- *DS_CONTROL* – создает диалоговое окно, которое работает так же, как дочернее окно другого диалогового окна, и очень похоже на страницу в окне свойств. Этот стиль позволяет пользователю перемещаться среди элементов управления дочернего диалогового окна, использовать его клавиши-ускорители и т. д.
- *DS_FIXEDSYS* – использует *SYSTEM_FIXED_FONT* вместо *SYSTEM_FONT*.
- *DS_MODALFRAME* – создает диалоговое окно с модальной рамкой диалогового окна, которая может быть объединена со строкой заголовка и меню окна путем определения стилей *WS_CAPTION* и *WS_SYSMENU*.
- *DS_NOFAILCREATE* – создает диалоговое окно даже если происходят ошибки, например, если дочернее окно не может быть создано или система не может создать специальный сегмент данных для элементов редактирования.
- *DS_NOIDLEMSG* – подавляет сообщение *WM_ENTERIDLE*, которое Windows иначе послал бы владельцу диалогового окна, в то время как диалоговое окно отображается на экране.
- *DS_SETFONT* – указывает, что шаблон диалогового окна (структура *DLGTEMPLATE*) содержит два дополнительных элемента, определяющих имя шрифта и размер в пунктах. Соответствующий шрифт используется, чтобы отображать текст внутри рабочей области диалогового окна и внутри элементов управления диалогового окна. Windows передает дескриптор шрифта диалоговому окну и каждому элементу управления, посылая им сообщение *WM_SETFONT*.

Функция ***ShowWindow*** устанавливает режим отображения окна:

```
BOOL ShowWindow (
  HWND hWnd,              // указатель на окно
  int CmdShow);           // режим
```

Функция ***UpdateWindow*** обновляет указанное окно, посылая ему сообщение ***WM_PAINT***. Это сообщение посылается непосредственно процедуре указанного окна, обходя очередь других сообщений.

```
BOOL UpdateWindow (HWND Wnd);  // указатель на окно
```

Функция ***DestroyWindow*** удаляет определенное окно, она посылает сообщения ***WM_DESTROY*** и ***WM_NCDESTROY*** окну, чтобы дезактивировать его и удалить фокус клавиатуры. Если определенное окно – родитель или владелец окон, функция ***DestroyWindow*** автоматически удаляет связанные дочерние или

находящиеся в собственности окна во время удаления окна владельца или родителя. Функция сначала удаляет дочерние окна, а затем окно родителя [7].

```
BOOL DestroyWindow (HWND hWnd);     // дескрипт ор для удаления окна
```

С набором сообщений работает несколько функций ***WinAPI***. Сообщения ставятся в очередь асинхронных сообщений при помощи функции

```
BOOL PostMessage (
  HWND Window,              // дескрипт ор окна
  UINT Message,             // передаваемое сообщение
  WPARAM Parametr,
  LPARAM Par);
```

При вызове этой функции определяется поток, создавший окно с дескриптором Windows. Далее выделяется память для параметров сообщения и производится добавление в очередь асинхронных сообщений. Возврат из этой функции происходит немедленно и, вероятно, что окно даже не получит данное сообщение. Упомянутая ранее функция ***PostQuitMessage*** тоже добавляет сообщение в очередь асинхронных сообщений потока.

Оконное сообщение можно отправить оконной процедуре вызовом функции ***SendMessage***, которая производит добавление в очередь синхронных сообщений.

```
BOOL SendMessage (
  HWND Window,              // дескрипт ор окна
  UINT Message,             // передаваемое сообщение
  WPARAM Parametr,
  LPARAM Par);
```

Оконная процедура обработает сообщение ***Message*** и только после этого вернет управление. Пока сообщение не обработано, поток находится в состоянии ожидания. Поток, обрабатывающий синхронное сообщение, может содержать бесконечный цикл, тогда поток-отправитель может «зависнуть». Избежать таких ситуаций можно при помощи нескольких функций ***WinAPI***. Одна из них ожидает в течение некоторого времени ответа от другого потока:

```
BOOL SendMessageTimeout (
  HWND Window,              // дескрипт ор окна
  UINT Message,             // передаваемое сообщение
  WPARAM Parametr,
  LPARAM Par,
  UINT Flags,               // флаги
  UINT Timeout,             // время ож идания от вет а, в миллисекундах
  PDWORD_PTR Result);       // возвращаемое значение оконной процедуры
```

В параметре ***Flags*** указываются следующие флаги: ***SMTO_NORMAL***,

SMTO_ABORTIFHUNG (если поток «завис», вернуть управление), *SMTO_NOTIMEOUTIFNOTHUNG* (если поток «не завис», игнорировать ограничение по времени), *SMTO_SMTO_BLOCK* (не обрабатывать другие синхронные сообщения, пока функция не вернет управление). Последний флаг, однако, может «помочь» потокам перейти в состояние взаимной блокировки.

Вторая функция также предназначена для отправки оконных сообщений:

```
BOOL SendMessageCallback (
  HWND Window,                    // дескрипт ор окна
  UINT Message,          // передаваемое сообщение
  WPARAM Parametr,
  LPARAM Par,
  SENDASYNCPROC PrcResCallBack,          // асинхронная процедура
  ULONG_PTR Data);          // данные для передачи оконной процедуре
```

При вызове потоком этой функции сообщение добавляется в очередь синхронных сообщений потоком-приемником, а управление сразу же возвращается потоку-отправителю. По окончании обработки сообщения приемник асинхронно отправляет свое сообщение в очередь ответных сообщений. Чтобы поток-отправитель смог получить этот ответ, должна быть предусмотрена функция, имеющая следующий прототип:

```
VOID CALLBACK ResCallBack (
  HWND Window,                    // дескрипт ор окна
  UINT Message,                   // передаваемое сообщение
  ULONG_PTR Data,          // данные для передачи оконной процедуре
  LRESULT Res);//результат обработки сообщения от оконной процедуры
```

Адрес этой функции передается в параметре *PrcResCallBack* функции *SendMessageCallback*. При вызове *ResCallBack* ей передается одноименный параметр *Data* из *SendMessageCallback*. Параметр *Res* содержит результат обработки сообщения от оконной процедуры.

Еще одна функция *WinAPI* предназначена для обмена сообщениями между потоками:

```
BOOL SendNotifyMessage (
  HWND Window,                    // дескрипт ор окна
  UINT Message,                   // передаваемое сообщение
  WPARAM Parametr,
  LPARAM Par);
```

Она также добавляет элемент в очередь синхронных сообщений и немедленно возвращает управление вызывающему потоку, т. е. ее поведение подобно функции *PostMessage*. Однако есть и отличия. Если *SendNotifyMessage* посылает

сообщения окну другого потока, то они извлекаются из очереди раньше сообщений, отправленных ***PostMessage***, т. е. имеют перед ними приоритет. Если же посылается сообщение тому окну, которое создано потоком-отправителем, то управление не возвращается до окончания обработки сообщения, что очень похоже на поведение функции ***SendMessage***.

Четвертая функция ***WinAPI*** связана с обработкой сообщений:

BOOL ReplyMessage (LRESULT Res);

Эта функция вызывается потоком, принимающим оконное сообщение. Ответ (результат обработки) асинхронно помещается в очередь ответных сообщений потока-отправителя, а тот может теперь получить результат с помощью параметра ***Res*** и продолжить свою работу. Функция возвращает *TRUE*, если обрабатывается межпоточное сообщение, а *FALSE* используется для внутрипоточных сообщений. Для того чтобы узнать, является ли сообщение меж- или внутрипоточным, можно вызвать функцию

BOOL InSendMessage (LRESULT Res);

Возвращаемое значение: *TRUE*, если обрабатывается межпоточное синхронное сообщение, и *FALSE* при обработке синхронного или асинхронного внутрипоточного сообщения.

Перечислим некоторые сообщения, которые могут посылаться окну, в дополнение к тем, что давались ранее:

- *WM_MOUSEMOVE* – посылается окну, когда курсор меняет свою позицию. При этом в процедуре обработки сообщений надо обязательно учитывать, что ее первый (если быть более точным, то третий) параметр *Parametr* будет содержать флаги виртуальных клавиш, а второй (четвертый, *Prm*) – координаты курсора. Для их получения надо воспользоваться макросом *MAKEPOINTS*.
- *WM_NCMOUSEMOVE* – посылается окну, когда курсор перемещается над неклиентской областью окна. Параметры обозначают координаты курсора и так называемые *hit-test* значения, они будут описаны далее.
- *WM_NCHITTEST* – посылается окну, когда перемещается курсор мыши или нажимается/отпускается кнопка мыши. Параметры обозначают координаты курсора. Возвращает *hit-test* значения.
- *WM_NCLBUTTONUP* – посылается окну, когда пользователь отпускает левую кнопку мыши, ее курсор находится над неклиентской областью окна. Параметры: *hit-test* значение и позиция курсора.
- *WM_NCLBUTTONDOWN* – посылается окну, когда пользователь нажимает левую кнопку мыши, ее курсор находится над неклиентской областью окна. Параметры: *hit-test* значение и позиция курсора.
- *WM_NCRBUTTONUP* и *WM_NCRBUTTONDOWN* аналогичны двум предыдущим, только «работают» с правой кнопкой мыши.
- *WM_NCHITTEST* посылается окну, когда перемещается курсор мыши

или нажимается/отпускается одна из ее кнопок. Параметры обозначают координаты курсора.

При запуске система создает особый поток необработанного ввода и системную очередь аппаратного ввода. Поток ввода бездействует, пока в очереди нет ни одного элемента. Когда пользователь нажимает кнопку мыши, перемещает курсор либо давит на клавишу, драйвер устройства добавляет аппаратное событие в системную очередь. Поток необработанного ввода активизируется, извлекает из очереди элемент и преобразует его в сообщение, а затем ставит в хвост очереди виртуального ввода конкретного потока. После этого поток ввода ждет появления следующего элемента в системной очереди. Именно системный поток необработанного ввода отвечает за обработку в ОС Windows особых комбинаций клавиш: *Ctrl+Alt+Del, Alt+Esc, Alt+Tab*. Ни один из других потоков не в состоянии «перехватить» эти клавиши.

Помимо виртуального ввода потока в ОС Windows поддерживается также концепция локального состояния ввода потока, под которой понимается следующее: для мыши – окно, захватившее мышь, форма курсора мыши, видимость курсора; для клавиатуры – нажатые клавиши, активное окно, окно в фокусе клавиатуры, состояние курсора [7].

Для того чтобы вывести окно на передний план и подключить его ввод к системному потоку ввода, достаточно вызвать функцию

```
BOOL SetForegroundWindow (HWND Window);
```

Для получения дескриптора окна, находящегося на переднем плане, используется функция

```
HWND GetForegroundWindow ();
```

Иногда нужно, чтобы два (или более) потока совместно использовали одну и ту же виртуальную очередь ввода и локальное состояние ввода. Сделать это можно при помощи функции

```
BOOL AttachThreadInput (
  DWORD AttachId,
    // идент ификат ор пот ока, чья очередь не нуж на
  DWORD AttachToId,
    // идент ификат ор пот ока, чья очередь будет  использоват ься
  BOOL Attach);
    // начат ь совмест ное использование или прекрат ит ь
```

При этом локальные очереди сообщений остаются индивидуальными. Совместно, повторимся, используется только виртуальная очередь ввода. Это может быть опасно тем, что один из потоков, обрабатывающих нажатие какой-то клавиши, может «зависнуть», тогда другие потоки вообще не получают никакого

ввода. Следующий фрагмент кода иллюстрирует использование некоторых из описанных функций:

```
HWND win1;
DWORD CurrThread, WinThread;
CurrThread = GetCurrentThreadId ();
WinThread = GetWindowThreadProcesId (GetForegroundWindow (),
                    NULL);
AttachThreadInput (CurrThread, TRUE);
...
SetForegroundWindow (win1);
```

Следующая функция позволяет получить экранные координаты указанного окна:

```
BOOL GetWindowRect (
  HWND Wnd,                  // идент ификат ор окна
  LPRECT Rect);              // адрес ст рукт уры с координат ами окна
```

Окно в общем случае состоит из нескольких частей. Имеется возможность получить координаты клиентской части окна, при этом координаты левой верхней точки всегда равны 0.

```
BOOL GetClientRect (
  HWND Wnd,                  // идент ификат ор окна
  LPRECT Rect);              // адрес ст рукт уры с координат ами окна
```

Структура *RECT* определяет коодинаты левой верхней и правой нижней точек прямоугольника.

```
typedef struct _RECT {
   LONG left;
   LONG top;
   LONG right;
   LONG bottom;
} RECT;
```

Следующая функция определяет, принадлежит ли данная точка определенному прямоугольнику. Точка считается принадлежащей, если лежит внутри четырех сторон прямоугольника, а также на левой или верхней сторонах. Если же точка лежит на правой или нижней сторонах, то она считается не принадлежащей прямоугольнику.

```
BOOL PtInRect(
   CONST RECT *Rect,         // адрес ст рукт уры-прямоугольника
   POINT aPoint);            // ст рукт ура с т очкой
```

Структура с точкой определена как

```
typedef struct tagPOINT {
```

```
    LONG x;
    LONG y;
} POINT;
```

Экранные координаты преобразуются в клиентские функцией

```
BOOL ScreenToClient
    HWND Wnd,                    // окно с исходными координат ами
    LPPOINT Point);          // адрес ст рукт уры, содерж ащей координат ы
```

Для того чтобы получить экранные координаты курсора мыши, необходимо вызывать функцию

```
BOOL GetCursorPos (LPPOINT aPoint);
      // адрес ст рукт уры с координат ами курсора
```

Существует и парная ей функция, которая устанавливает курсор в заданную позицию:

```
BOOL SetCursorPos(
    int XPos,                    // горизонт альная позиция
    int YPos);                   // верт икальная позиция
```

Изменение некоторых параметров окна производится функцией

```
BOOL MoveWindow(
    HWND Wnd,            // Описат ель окна
    int Hor,             // горизонт альная координат а новой позиции окна
    int Vert,            // верт икальная координат а новой позиции окна
    int Width,           // новая ширина
    int Height,          // новая высот а
    BOOL Repaint);
        // Определяет , будет  ли окно перерисовыват ься (TRUE)
```

Для получения и установки параметров полосы прокрутки можно воспользоваться следующей парой функций:

```
BOOL GetScrollInfo(
    HWND Wnd,                    // описат ель окна с полосой прокрут ки
    int Bar,             // т ип полосы прокрут ки (SB_HORZ – горизонт альная,
                         // SB_VERT – верт икальная)
    LPSCROLLINFO ScrollInfo);
                    // указат ель на ст рукт уру с парамет рами скроллинга
int SetScrollInfo(
    HWND Wnd,                    // описат ель окна
    int fnBar,           //т ип полосы прокрут ки (SB_HORZ – горизонт альная,
                         // SB_VERT – верт икальная)
    LPSCROLLINFO ScrollInfo,
          // указат ель на ст рукт уру с парамет рами скроллинга
    BOOL fRedraw);       // флаг перерисовки (TRUE – перерисисовыват ь)
```

Следующая функция позволяет прокручивать содержимое клиентской области окна, однако ее единицы зависят от устройства:

```
int ScrollWindowEx(
  HWND Wnd,                 // описат ель окна
  int dx,                   // горизонт альная величина скроллинга
                            //От рицат ельная – «крут ит ь» влево
  int dy,                   // верт икальная величина скроллинга
                            // От рицат ельная – «крут ит ь» вверх
  CONST RECT *Scroll,       //ст рукт уры с прямоугольником прокрут ки,
                            // TRUE – прокручиват ь все
  CONST RECT *Clip,
        // адрес ст рукт уры с прямоугольником-клипом
  HRGN RegionUpdate,        // обычно NULL
  LPRECT Update,            // обычно NULL
  UINT flags);              // парамет ры прокрут ки, обычно SW_ERASE
```

Еще одна группа функций позволяет направлять и сбрасывать сообщения мыши, связанные с конкретными окнами:

```
// Уст ановка захват а сообщений
HWND SetCapture(HWND hWnd);       // описат ель окна
HWND GetCapture(VOID)
  // получение идент ификат ора окна, перехват ывающего сообщения
BOOL ReleaseCapture(VOID)         // сброс перехват а сообщений
```

Порядок выполнения работы

1. Ознакомиться со спецификациями функций WinAPI по работе с разделами виртуального адресного пространства процессов и кучами.

2. Разработать программу в соответствии с полученным заданием, в которой должны использоваться главные и дочерние окна, различные элементы управления и стили, а также обмен сообщениями.

3. Написать отчет.

Варианты заданий

Вариант №1

Разработать программу, которая демонстрирует эффект «убегания окна от курсора» при его попадании на клиентскую область окна. Завершение «убегания» достигается с помощью двойного щелчка кнопкой мыши.

Вариант №2

Разработать программу, которая демонстрирует эффект «прилипания окна к курсору» при его попадании на клиентскую область окна. «Отлипание окна от

курсора» производится последовательным перемещением курсора вверх, а потом вниз.

Вариант №3

Разработать программу, которая демонстрирует эффект «убегания окна от курсора» при его попадании на неклиентскую область окна. Завершение «убегания» достигается с помощью двойного щелчка кнопкой мыши.

Вариант №4

Разработать программу, которая демонстрирует эффект «прилипания окна к курсору» при его попадании на заголовок окна. Дальнейшее перемещение влево-вправо блокируется, окно может перемещаться только вверх и вниз. «Отлипание окна от курсора» производится двойным щелчком кнопки мыши.

Вариант №5

Разработать программу, которая демонстрирует эффект «прилипания окна к курсору» при его попадании на заголовок окна. Дальнейшее перемещение вверх-вниз блокируется, окно может перемещаться только влево и вправо. «Отлипание окна от курсора» производится двойным щелчком кнопки мыши.

Вариант №6

Разработать программу, которая создает окно с горизонтальной и вертикальной полосами прокрутки. При щелчке кнопкой мыши по заголовку окна блокируется вертикальная прокрутка. Блокировка снимается при щелчке кнопкой мыши по вертикальной полосе.

Вариант №7

Разработать программу, которая создает окно с горизонтальной и вертикальной полосами прокрутки. При щелчке кнопкой мыши по заголовку окна блокируется горизонтальная прокрутка. Блокировка снимается при щелчке кнопкой мыши по горизонтальной полосе.

Вариант №8

Разработать программу, которая создает окно. Закрытие окна должно выполняться щелчком кнопки по значку сворачивания, при этом должно создаваться новое окно с таким же стилем. Однократный щелчок кнопкой мыши по значку закрытия не приводит к выполнению этого действия, работа программы завершается двойным щелчком.

Вариант №9

Разработать программу, которая создает два окна. Действия по сворачиванию, разворачиванию и закрытию одного окна должны выполняться над обоими окнами.

Вариант №10

Разработать программу, которая демонстрирует эффект «прилипания окна к курсору» при его попадании и последующем щелчке кнопкой мыши по какой-либо из полос прокрутки. «Отлипание окна от курсора» производится трехкратным щелчком кнопки мыши по заголовку окна.

Вариант №11

Разработать программу, которая демонстрирует эффект «прилипания окна к курсору» при его попадании и последующем щелчке кнопкой мыши по заголовку окна. «Отлипание окна от курсора» производится трехкратным щелчком кнопки мыши по заголовку окна.

Вариант №12

Разработать программу, которая создает окно с горизонтальной и вертикальной полосами прокрутки. При щелчке кнопкой мыши по вертикальной полосе прокрутки окна блокируется горизонтальная прокрутка. Блокировка снимается при щелчке кнопкой мыши по вертикальной полосе.

Вариант №13

Разработать программу, которая создает окно с горизонтальной и вертикальной полосами прокрутки. При щелчке кнопкой мыши по горизонтальной полосе прокрутки окна блокируется вертикальная прокрутка. Блокировка снимается при щелчке кнопкой мыши по горизонтальной полосе.

Вариант №14

Разработать программу, которая создает окно. Закрытие окна должно выполняться щелчком кнопки мыши по значку разворачивания окна, при этом должно создаваться новое окно с таким же стилем. Однократный щелчок кнопкой мыши по значку закрытия не приводит к выполнению этого действия, работа программы завершается двойным щелчком.

Вариант №15

Разработать программу, которая создает три окна. Действия по сворачиванию и разворачиванию одного окна должны выполняться над обоими окнами.

Вариант №16

Разработать программу, которая демонстрирует эффект «прилипания окна к курсору» при его попадании и последующем двойном щелчке кнопкой мыши по какой-либо из полос прокрутки. «Отлипание окна от курсора» производится повторным двойным щелчком кнопки мыши по полосе прокрутки.

Вариант №17

Разработать программу, которая демонстрирует эффект «прилипания окна к курсору» при его попадании и последующем двойном щелчке кнопкой мыши

по заголовку окна. «Отлипание окна от курсора» производится повторным двойным щелчком кнопки мыши по заголовку окна.

Вариант №18

Разработать программу, которая создает окно. Закрытие окна должно выполняться двойным щелчком кнопки мыши по значку закрытия окна, при этом должно создаваться новое окно с таким же стилем.

Вариант №19

Разработать программу, которая создает три окна. Действия по сворачиванию и закрытию одного окна должны выполняться над обоими окнами.

Вариант №20

Разработать программу, которая демонстрирует эффект «прилипания окна к курсору» при его попадании и последующем щелчке кнопкой мыши по вертикальной полосе прокрутки. «Отлипание окна от курсора» производится повторным щелчком кнопки мыши по этой же полосе прокрутки.

Вариант №21

Разработать программу, которая демонстрирует эффект «прилипания окна к курсору» при его попадании и последующем щелчке кнопкой мыши по горизонтальной полосе прокрутки. «Отлипание окна от курсора» производится последовательным перемещением курсора влево, вправо, вверх.

Вариант №22

Разработать программу, которая создает три окна с заголовками «Окно1», «Окно2» и «Окно3». Щелчок кнопкой мыши по значку закрытия «Окна1» закрывает только «Окно2» и «Окно3», а щелчок по такому же значку на «Окне2» и «Окне3» приводит к завершению работы программы.

Вариант №23

Разработать программу, которая создает три окна. Действия по разворачиванию и закрытию одного окна должны выполняться над обоими окнами.

Вариант №24

Разработать программу, которая демонстрирует эффект «прилипания окна к курсору» при его попадании и последующем щелчке управляющей кнопкой мыши по системному меню. «Отлипание окна от курсора» производится повторным щелчком управляющей кнопкой мыши по любой полосе прокрутки.

Вариант №25

Разработать программу, которая демонстрирует эффект «прилипания окна к курсору» при его попадании и последующем щелчке вспомогательной кнопкой мыши по системному меню. «Отлипание окна от курсора» производится последовательным перемещением курсора вправо, влево, вниз.

Содержание отчета

Отчет по лабораторной работе должен быть оформлен в соответствии с требованиями, предъявляемыми к текстовым документам, и содержать:

- цель работы;
- краткие теоретические сведения, необходимые для выполнения работы;
- описание порядка выполнения работы с приведением листинга кода, скриншотов рабочих окон;
- выводы по работе.

Лабораторная работа №11

Использование механизма сокетов в ОС Windows

Цель – изучение механизма сокетов в операционной системе Windows для протокола UDP.

Теоретическая часть

Взамодействие сетевых процессов по протоколу UDP предусматривает обмен датаграммами, характеризуется минимальным уровнем сервиса со стороны системы и минимальными служебными затратами. Целостность передаваемых данных системой не гарантируется, при необходимости о ней должны заботиться сами взаимодействующие программы. Используемый тип сокета – SOCK_DGRAM, которому в IP-сетях соответствует протокол UDP. Порядок системных вызовов при взаимодействии схематично показан на рисунке 4.

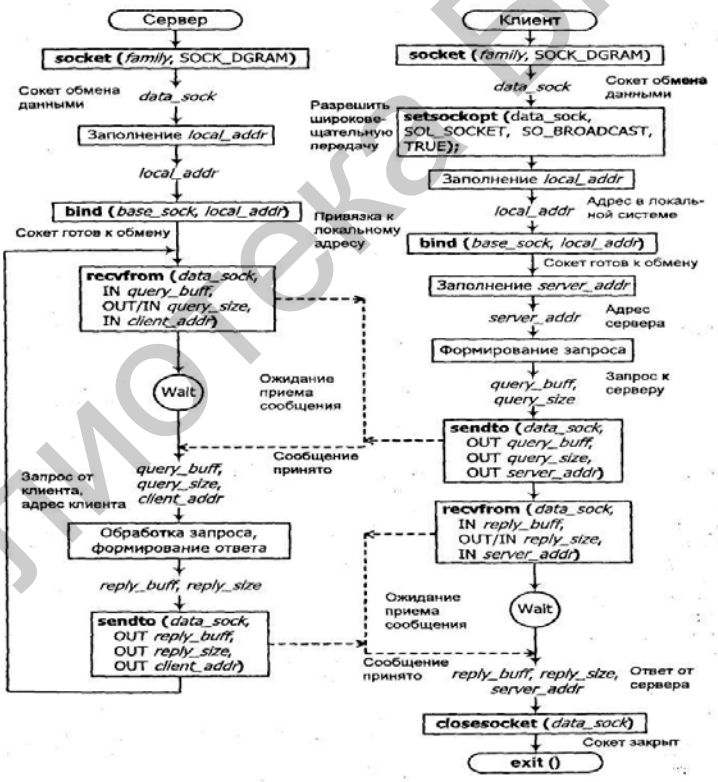

Рисунок 4 – Взаимодействие без установления соединения

Примечание – На рисунке 4 имена системных функций сохранены, но формат их вызова упрощен для повышения наглядности. Аргументы, описывающие передаваемые и принимаемые сообщения, снабжены пометками, являются ли соответствующие объекты параметрами вызова или создаются (заполняются) в результате его выполнения.

В показанном примере действия клиента и сервера в общем схожи, и с точки зрения программирования сокетов отличия между ними сугубо номинальные. Каждый из них использует всего один сокет, служащий и для приема, и для передачи сообщений. Ради упрощения считаем, что клиент отсылает серверу единственной запрос *{query)* и ожидает ответ на него *{reply)*, а сервер в цикле принимает запросы, обрабатывает их и отсылает ответы, используя «обратные адреса» запросов. Цикл сервера показан бесконечным; на практике для управления серверами служат команды, предусмотренные в протоколе обмена, или какие-либо дополнительные средства.

Перед использованием сокетов необходимо загрузить корректную версию библиотеки сокетов ***WinSock*** с помощью функции инициализации

```
int WSAStartup (
  WORD Version,              // номер версии
  LPWSADATA Data);           // информация о версии
```

Связь без установления соединения выполняется при помощи пользовательских дейтаграмм протокола *UDP*. Он не гарантирует надежности, однако может осуществлять передачу данных нескольким адресатам и принимать их от нескольких источников. В частности, данные, отправляемые клиентом, передаются на сервер немедленно, независимо от готовности (или неготовности) сервера. При получении данных сервер не подтверждает их прием. Данные передаются порциями, называемыми дейтаграммами [7].

При использовании протокола *IP* устройствам назначается *IP*-адрес. Для взаимодействия с сервером по *TCP* клиент должен указать *IP*-адрес сервера и номер порта службы. Чтобы прослушивать входящие запросы клиента, сервер тоже должен указать *IP*-адрес и номер порта. В ***WinSock*** *IP*-адрес и порт службы указываются в структуре *SOCKADDR_IN*:

```
struct sockaddr_in {
  short sin_family;          // для IP должен использоваться AF_INET
  u_short sin_port;
    // любой свободный порт из диапазона (1024, 65535)
  struct  in_addr sin_addr;          // IP-адрес в 4-байтовом виде
  char sin_zero[8];                  // заполнитель нулями
};
```

Вспомогательная функция ***inet_addr*** преобразует *IP*-адрес из точечной нотации в беззнаковое длинное целое число, в котором байты следуют в соответствии с сетевым порядком следования:

```
unsigned long inet_addr (const char FAR * cp);
  // строка с адресом в десятично-точечной нотации
```

Существуют два специальных адреса: ***INADDR_ANY***, который позволяет серверу слушать клиента через любой сетевой интерфейс на несущем компьютере, и ***INADDR_BROADCAST***, позволяющий широковещательно рассылать дейтаграммы по сети.

Две следующие функции позволяют преобразовывать соответственно четырех- и двухбайтовые числа из системного порядка в сетевой.

u_long htonl(u_long hostlong);
u_short htons (u_short hostshort);

Следующие две функции решают обратную задачу, т. е. переставляют байты из сетевого порядка в системный:

u_long ntohl(u_long netlong);
u_short ntohs (u_short netshort);

Следующий фрагмент демонстрирует, как создается структура с адресом и номером порта при помощи описанных функций:

```
SOCKADDR_IN     InetAddr;

INT PortNum = 5101;

InetAddr.sin_family = AF_INET;
InetAddr.sin_addr.s_addr = inet_addr ("192.168.123.45");
InetAddr.sin_port.htons (PortNum);
```

Процесс получения данных на сокете, не требующем соединения, прост. Сокет создается функцией

```
SOCKET socket (
  int af,                  // семейство адресов протокола
                           // (для IP надо указать AF_INET)
  int type,          // тип сокета (для UDP надо указать SOCK_DGRAM)
  int namelen);    // транспорт (для UDP – IPPROTO_UDP)
```

Открыть *IP*-сокет при помощи протокола *UDP* можно следующим образом:

```
sock_udp = socket (AF_INET, SOCK_DGRAM, IPPROTO_UDP);
```

Затем осуществляется привязка сокета к интерфейсу, на котором будут приниматься данные, с помощью функции ***bind*** (аналогично протоколам, ориентированным на соединения). После этого нужно просто ожидать

прием входящих данных. Соединения нет, а значит, сокет-приемник получает дейтаграммы от любой станции в сети. Итак, после создания сокета конкретного протокола его надо связать со стандартным адресом при помощи функции

```
int bind (
  SOCKET s,     // сокет, на котором ожидается прием данных
  const struct sockaddr FAR* name, // универсальный буфер
  int namelen);  // размер буфера
```

При возникновении ошибки функция ***bind*** возвращает *SOCKET_ERROR.*

```
if (bind (sock1, (struct sockaddr *) addr, sizeof (addr)) ==
   SOCKET_ERROR) {
    printf ("bind () с ошибкой\n");
   }
```

В сетевом программировании главное – отправлять и принимать данные. Для пересылки используют функцию ***sendto***, а для приема – ***recvfrom***. Первая из них определена так:

```
int sendto (
  SOCKET s,                  // сокет для отправки данных
  const char FAR* buf,       // буфер для отправляемых данных
  int len,                   // число отправляемых байтов или размер буфера
  int flags,                 // флаги
  struct sockaddr FAR* to,
  int tolen);                // длина адреса приемника
```

При успешном выполнении ***sendto*** вернет количество переданных байтов, иначе – ошибку ***SOCKET_ERROR***. Возвращаемое количество байтов может отличаться от значения параметра ***len***.

Функция приема аналогична отправке, она возвращает количество полученных байтов. Если сокет к этому времени был уже закрыт, то возвращаемым значением является 0. Если же возникла ошибка – ***SOCKET_ERROR.***

```
int recvfrom (
  SOCKET s,                  // сокет для приема данных
  char FAR* buf,             // буфер с данными для получаемых данных
  int len,                   // число принимаемых байтов или размер буфера
  int flags,                 // флаги
  struct sockaddr FAR* from,      // адрес станции-отправителя
  int FAR* fromlen);              // длина адреса отправителя
```

По окончании работы с сокетом необходимо закрыть соединение и освободить все ресурсы, связанные с дескриптором сокета, с помощью функции ***closesocket***.

int closesocket (SOCKET s); *// закрытие сокета*

Порядок выполнения работы

1. Ознакомиться с описанием функций ***WinSock API***, которые служат для работы с протоколами, которые не устанавливают соединение.
2. Разработать программу взаимодействия клиента и сервера по протоколу *UDP* согласно заданию.
3. Написать отчет.

Варианты заданий

Вариант №1

Разработать две программы – сервер и клиент. Клиент отсылает серверу два числа *L* и *U*, введенные пользователем, где *L* – это нижняя граница диапазона, *U* – верхняя граница диапазона. Сервер принимает значения границ диапазона, вычисляет сумму и произведение чисел от *L* до *U* и выводит полученные значения на экран. Для взаимодействия воспользоваться механизмом сокетов и протоколом *UDP*.

Вариант №2

Разработать две программы – сервер и клиент. Клиент отсылает серверу введенный пользователем номер числа Фибоначчи. Сервер принимает номер, вычисляет число Фибоначчи с этим номером по формуле $F_i = F_{i-1} + F_{i-2}$, $F_0 = F_1 = 1$ и выводит его на экран. Для взаимодействия воспользоваться механизмом сокетов и протоколом *UDP*.

Вариант №3

Разработать две программы – сервер и клиент. Клиент отсылает серверу введенную пользователем строку, хранящую знаковое целое число. Сервер принимает строку, хранящую знаковое целое число, и выводит на экран строковый эквивалент этого числа прописью (например, ввод «–1211» должен приводить к выводу «минус тысяча двести одиннадцать»). Для взаимодействия воспользоваться механизмом сокетов и протоколом *UDP*.

Вариант №4

Разработать две программы – сервер и клиент. Клиент отсылает серверу

введенную пользователем строку, хранящую число со знаком и плавающей точкой. Сервер принимает строку, хранящую число со знаком и плавающей точкой, и выводит на экран строковый эквивалент этого числа прописью (например, ввод «–12.11» должен приводить к выводу «минус двенадцать целых одиннадцать сотых»). Для взаимодействия воспользоваться механизмом сокетов и протоколом *UDP*.

Вариант №5

Разработать две программы – сервер и клиент. Клиент отсылает серверу две строки, введенные пользователем. Сервер принимает две строки. Далее, если обе строки хранят целые числа со знаком, то на экран выводится сумма чисел, в противном случае – конкатенация двух введенных строк. Для взаимодействия воспользоваться механизмом сокетов и протоколом *UDP*.

Вариант №6

Разработать две программы – сервер и клиент. Клиент отсылает серверу элементы двух прямоугольных матриц, введенные пользователем. Сервер принимает две прямоугольные матрицы, а затем выводит на экран их сумму и произведение. Для взаимодействия воспользоваться механизмом сокетов и протоколом *UDP*.

Вариант №7

Разработать две программы – сервер и клиент. Клиент отсылает серверу элементы вектора (одномерного целочисленного массива), введенные пользователем. Сервер принимает вектор , упорядочивает его по возрастанию любым из так называемых «улучшенных алгоритмов» сортировки массивов и выводит на экран. Для взаимодействия воспользоваться механизмом сокетов и протоколом *UDP*.

Вариант №8

Разработать две программы – сервер и клиент. Клиент отсылает серверу элементы вектора (одномерного массива чисел с плавающей точкой), введенные пользователем. Сервер принимает вектор, упорядочивает его по возрастанию любым из так называемых «улучшенных алгоритмов» сортировки массивов и выводит на экран. Для взаимодействия воспользоваться механизмом сокетов и протоколом *UDP*.

Вариант №9

Разработать две программы – сервер и клиент. Клиент отсылает серверу элементы вектора (одномерного массива строк), введенные пользователем. Сер-

вер принимает вектор, упорядочивает его по возрастанию любым из так называемых «улучшенных алгоритмов» сортировки массивов и выводит на экран. Для взаимодействия воспользоваться механизмом сокетов и протоколом *UDP*.

Вариант №10

Разработать две программы – сервер и клиент. Клиент отсылает серверу элементы введенной пользователем квадратной матрицы. Сервер принимает матрицу, затем вычисляет сумму элементов, лежащих на главной и побочной диагоналях, и выводит на экран. Для взаимодействия воспользоваться механизмом сокетов и протоколом *UDP*.

Вариант №11

Разработать две программы – сервер и клиент. Клиент отсылает серверу элементы введенной пользователем квадратной матрицы. Сервер принимает матрицу, затем вычисляет сумму элементов, не лежащих на главной и побочной диагоналях, и выводит на экран. Для взаимодействия воспользоваться механизмом сокетов и протоколом *UDP*.

Вариант №12

Разработать две программы – сервер и клиент. Клиент принимает от пользователя две даты – строки вида ЦЦ.ЦЦ.ЦЦЦЦ, где Ц – это любая цифра из диапазона 0–9, и отсылает серверу. Сервер принимает даты , вычисляет полное количество дней, прошедших между двумя полученными датами, и выводит его на экран. Для взаимодействия воспользоваться механизмом сокетов и протоколом *UDP*.

Вариант №13

Разработать две программы – сервер и клиент. Клиент принимает от пользователя два значения времени – строки вида ЦЦ.ЦЦ.ЦЦ, где Ц – это любая цифра из диапазона 0–9, и отсылает серверу. Сервер принимает обе строки, вычисляет полное количество секунд, прошедших между двумя значениями времени, и выводит его на экран. Для взаимодействия воспользоваться механизмом сокетов и протоколом *UDP*.

Вариант №14

Разработать две программы – сервер и клиент. Клиент отсылает серверу две строки, введенные пользователем. Сервер принимает, осуществляет поиск вхождения второй строки в первую любым известным методом, кроме прямого (алгоритм Кнута – Мориса – Пратта, алгоритм Боуэра – Мура), и выводит на экран значение индекса элемента первой строки, с которого началось совпаде-

ние, или «–1» в противном случае. Для взаимодействия воспользоваться механизмом сокетов и протоколом *UDP*.

Вариант №15

Разработать две программы – сервер и клиент. Клиент отсылает серверу две строки, введенные пользователем. Сервер принимает две строки, осуществляет поиск количества вхождений второй строки в первую любым известным методом, кроме прямого (алгоритм Кнута – Мориса – Пратта, алгоритм Боуэра – Мура), и выводит на экран полученное значение. Для взаимодействия воспользоваться механизмом сокетов и протоколом *UDP*.

Вариант №16

Разработать две программы – сервер и клиент. Клиент принимает от пользователя дату – строку вида ЦЦ.ЦЦ.ЦЦЦЦ, где Ц – это любая цифра из диапазона 0–9, и отсылает серверу. Сервер принимает дату и выводит на экран число и месяц прописью, а за последними четырьмя – слово «года» (например, ввод «29.02.2008» приводит к выводу «Двадцать девятое февраля 2008 года»). Для взаимодействия воспользоваться механизмом сокетов и протоколом *UDP*.

Вариант №17

Разработать две программы – сервер и клиент. Клиент принимает от пользователя значение времени – строку вида ЦЦ.ЦЦ.ЦЦ, где Ц – это любая цифра из диапазона 0–9, и отсылает серверу. Сервер принимает значение времени и выводит на экран значение часов, минут и секунд прописью (например, ввод «12.01.20» приводит к выводу «двенадцать часов одна минута двадцать секунд»). Для взаимодействия воспользоваться механизмом сокетов и протоколом *UDP*.

Вариант №18

Разработать две программы – сервер и клиент. Клиент принимает от пользователя строку из нулей и единиц («битовую строку») и отсылает серверу. Сервер принимает битовую строку, инвертирует ее, выводит на экран значение инвертированной строки, переводит ее в число в десятичный формат и выводит полученное число на экран. Для взаимодействия воспользоваться механизмом сокетов и протоколом *UDP*.

Вариант №19

Разработать две программы – сервер и клиент. Клиент принимает от пользователя строку из нулей и единиц («битовую строку») и отсылает серверу. Сервер принимает битовую строку, осуществляет ее реверс (нули заменяются на единицы, а единицы на нули). Полученная строка выводится на экран, затем программа переводит ее в число в десятичном формате и выводит полученное число

на экран. Для взаимодействия воспользоваться механизмом сокетов и протоколом *UDP*.

Вариант №20

Разработать две программы – сервер и клиент. Клиент отсылает число, введенное пользователем, серверу. Сервер принимает число, вычисляет его факториал по формуле $N! = N * (N – 1)!$, где 0 != 1, и выводит его на экран. Для взаимодействия воспользоваться механизмом сокетов и протоколом *UDP*.

Вариант №21

Разработать две программы – сервер и клиент. Клиент принимает от пользователя квадратную матрицу и отсылает на сервер. Сервер принимает матрицу, осуществляет обход только крайних ее элементов, вычисляет их сумму и выводит на экран. Для взаимодействия воспользоваться механизмом сокетов и протоколом *UDP*.

Вариант №22

Разработать две программы – сервер и клиент. Клиент принимает от пользователя элементы целочисленного вектора (одномерного массива), а также значение ключа для поиска, и отсылает серверу. Сервер принимает вектор и ключ, затем осуществляет поиск элемента по ключу любым известным методом, кроме прямого (двоичный, случайный, золотого сечения), и выводит результат на экран. Для взаимодействия воспользоваться механизмом сокетов и протоколом *UDP*.

Вариант №23

Разработать две программы – сервер и клиент. Клиент принимает от пользователя строку символов и отсылает серверу. Сервер принимает строку, осуществляет замену всех латинских букв на их аналоги из кириллицы и выводит результат на экран. Для взаимодействия воспользоваться механизмом сокетов и протоколом *UDP*.

Вариант №24

Разработать две программы – сервер и клиент. Клиент принимает от пользователя строку символов и отсылает серверу. Сервер принимает строку, осуществляет смену регистра всех букв и выводит результат на экран. Для взаимодействия воспользоваться механизмом сокетов и протоколом *UDP*.

Вариант №25

Разработать две программы – сервер и клиент. Клиент принимает от пользователя беззнаковое целое число и отсылает серверу. Сервер принимает число. Если оно является степенью двойки, то на экран выводится показатель степени

или в противном случае – сообщение «не является степенью двойки». Для взаимодействия воспользоваться механизмом сокетов и протоколом *UDP*.

Содержание отчета

Отчет по лабораторной работе должен быть оформлен в соответствии с требованиями, предъявляемыми к текстовым документам, и содержать:

- цель работы;
- краткие теоретические сведения, необходимые для выполнения работы;
- описание порядка выполнения работы с приведением листинга кода, скриншотов рабочих окон;
- выводы по работе.

Лабораторная работа №12

Использование программных средств для шифрования файлов

Цель работы – изучение криптографического преобразования данных.

Теоретические сведения

Шифрование является краеугольным камнем всех служб информационной безопасности. Любая процедура шифрования, превращающая информацию из обычного «понятного» вида в «нечитабельный» зашифрованный, должна быть дополнена процедурой дешифрования, применение которой к зашифрованному тексту снова приводит его в понятный вид. **Пара процедур** – шифрование и дешифрирование – называется **криптосистемой**.

Информацию, над которой выполняются функции шифрования и дешифрирования, будем условно называть «текст» и учитывать, что это может быть также числовой массив или графические данные.

В современных алгоритмах шифрования предусматривается наличие параметра – секретного ключа.

Алгоритм шифрования считается раскрытым, если найдена процедура, позволяющая подобрать ключ за реальное время. Сложность алгоритма раскрытия является одной из важных характеристик криптосистемы и называется криптостойкостью.

Существуют два класса криптосистем – симметричные и асимметричные.

В симметричных схемах шифрования (классическая криптография) секретный ключ зашифровки совпадает с секретным ключом расшифровки.

В асимметричных схемах шифрования (криптография с открытым ключом) открытый ключ зашифровки не совпадает с секретным ключом расшифровки.

На рисунке 5 представлена модель симметричного алгоритма шифрования.

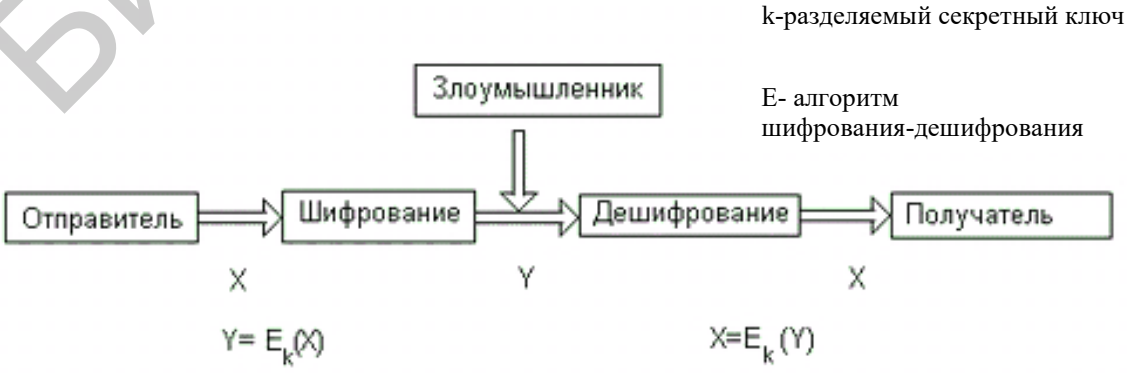

Рисунок 5 – Модель симметричного шифрования

На рисунке 5 приведена классическая модель симметричной криптосистемы, теоретические основы которой впервые были изложены в 1949 году в работе Клода Шеннона.

В данной модели три участника: отправитель, получатель, злоумышленник.

Задача отправителя заключается в том, чтобы по открытому каналу передать некоторое сообщение в защищенном виде. Для этого он на ключе *k* зашифровывает открытый текст *X* и передает шифрованный текст *Y*.

Задача получателя заключается в том, чтобы расшифровать *Y* и прочитать сообщение *X*.

Предполагается, что отправитель имеет свой источник ключа. Сгенерированный ключ заранее по надежному каналу передается получателю.

Задача злоумышленника заключается в перехвате и чтении передаваемых сообщений, а также в имитации ложных сообщений.

Наиболее популярным стандартным симметричным алгоритмом шифрования данных является *DES (Data Encryption Standard)*. Алгоритм разработан фирмой *IBM* и в 1976 году был рекомендован Национальным бюро стандартов к использованию в открытых секторах экономики. Схема шифрования по алгоритму *DES* представлена на рисунке 6.

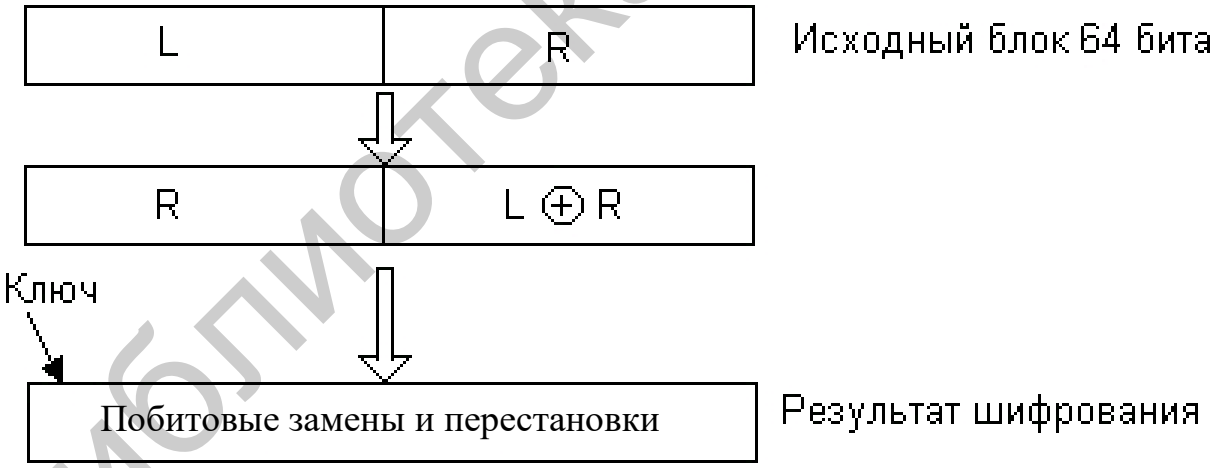

Рисунок 6 – Схема шифрования по алгоритму DES

В алгоритме *DES* данные шифруются поблочно. Перед шифрованием любая форма представления данных преобразуется в числовую. Эти числа получают путем любой открытой процедуры преобразования блока текста в число. Например, ими могли бы быть значения двоичных чисел, полученные слиянием *ASCII*-кодов последовательных символов соответствующего блока текста [7].

На вход шифрующей функции поступает блок данных размером 64 бита, он делится пополам на левую (*L*) и правую (*R*) части.

На первом этапе на место левой части результирующего блока помещается правая часть исходного блока. Правая часть результирующего блока вычисляется как сумма по модулю 2 (операция *XOR*) левой и правой частей исходного блока.

Затем на основе случайной двоичной последовательности по определенной схеме в полученном результате выполняются побитовые замены и перестановки. Используемая двоичная последовательность, представляющая собой ключ данного алгоритма, имеет длину 64 бита, из которых 56 действительно случайны, а 8 предназначены для контроля ключа.

Для повышения криптостойкости алгоритма *DES* иногда применяют его усиленный вариант, называемый «тройным *DES*», включающий троекратное шифрование с использованием двух разных ключей. При этом можно считать, что длина ключа увеличивается с 56 бит до 112 бит, а значит, криптостойкость алгоритма существенно повышается. Но за это приходится платить производительностью – «тройной *DES*» требует в три раза больше времени, чем обычный.

В симметричных алгоритмах главную проблему представляют ключи. Проблема с ключами возникает даже в системе с двумя абонентами, а в системе с несколькими абонентами, желающими обмениваться секретными данными по принципу «каждый с каждым», потребуется количество ключей, пропорциональное квадрату количества абонентов, что при большом числе абонентов делает задачу чрезвычайно сложной.

Несимметричные алгоритмы, основанные на использовании открытых ключей, снимают эту проблему .

В середине 70-х годов XX века двое ученых – Винфилд Диффи и Мартин Хеллман – описали принципы шифрования с открытыми ключами.

Особенность шифрования на основе открытых ключей состоит в том, что одновременно генерируется уникальная пара ключей, таких, что текст, зашифрованный одним ключом, может быть расшифрован только с использованием второго ключа и наоборот.

На рисунке 7 представлена модель несимметричного шифрования.

Рисунок 7 – Модель несимметричного шифрования

В модели криптосхемы с открытым ключом также три участника: отправитель, получатель, злоумышленник.

Задача отправителя заключается в том, чтобы по открытому каналу связи передать некоторое сообщение в защищенном виде.

Получатель генерирует на своей стороне два ключа: открытый *E* и закрытый *D*. Закрытый ключ *D* (часто называемый также личным ключом) абонент должен сохранять в защищенном месте, а открытый ключ *E* он может передать всем, с кем он хочет поддерживать защищенные отношения. Открытый ключ используется для шифрования текста, но расшифровать текст можно только с помощью закрытого ключа. Поэтому открытый ключ передается отправителю в незащищенном виде.

Отправитель, используя открытый ключ получателя, шифрует сообщение *X* и передает его получателю.

Получатель расшифровывает сообщение своим закрытым ключом *D*.

Поясним принципиальную связь между закрытым и открытым ключами следующей аналогией.

Пусть абонент 1 (рисунок 8, а) решает вести секретную переписку со своими сотрудниками на малоизвестном языке, например санскрите. Для этого он приобретает санскрито-русский словарь, а всем своим абонентам посылает русско-санскритские словари. Каждый из них, пользуясь словарем, пишет сообщения на санскрите и посылает их абоненту 1, который переводит их на русский язык, пользуясь доступным только ему санскрито-русским словарем. Очевидно, что здесь роль открытого ключа *E* играет русско-санскритский словарь, а роль закрытого ключа *D* – санскрито-русский словарь. Могут ли абоненты 2, 3 и 4 прочитать чужие сообщения S_2, S_3, S_4, которые посылает каждый из них абоненту 1? Нет, т. к. для этого им нужен санскрито-русский словарь, обладателем которого является только абонент 1. Но теоретическая возможность этого имеется, т. к.

можно прямым перебором составить санскрито-русский словарь по русско-санскритскому словарю. Такая процедура, требующая больших временных затрат, является отдаленной аналогией восстановления закрытого ключа по открытому [7].

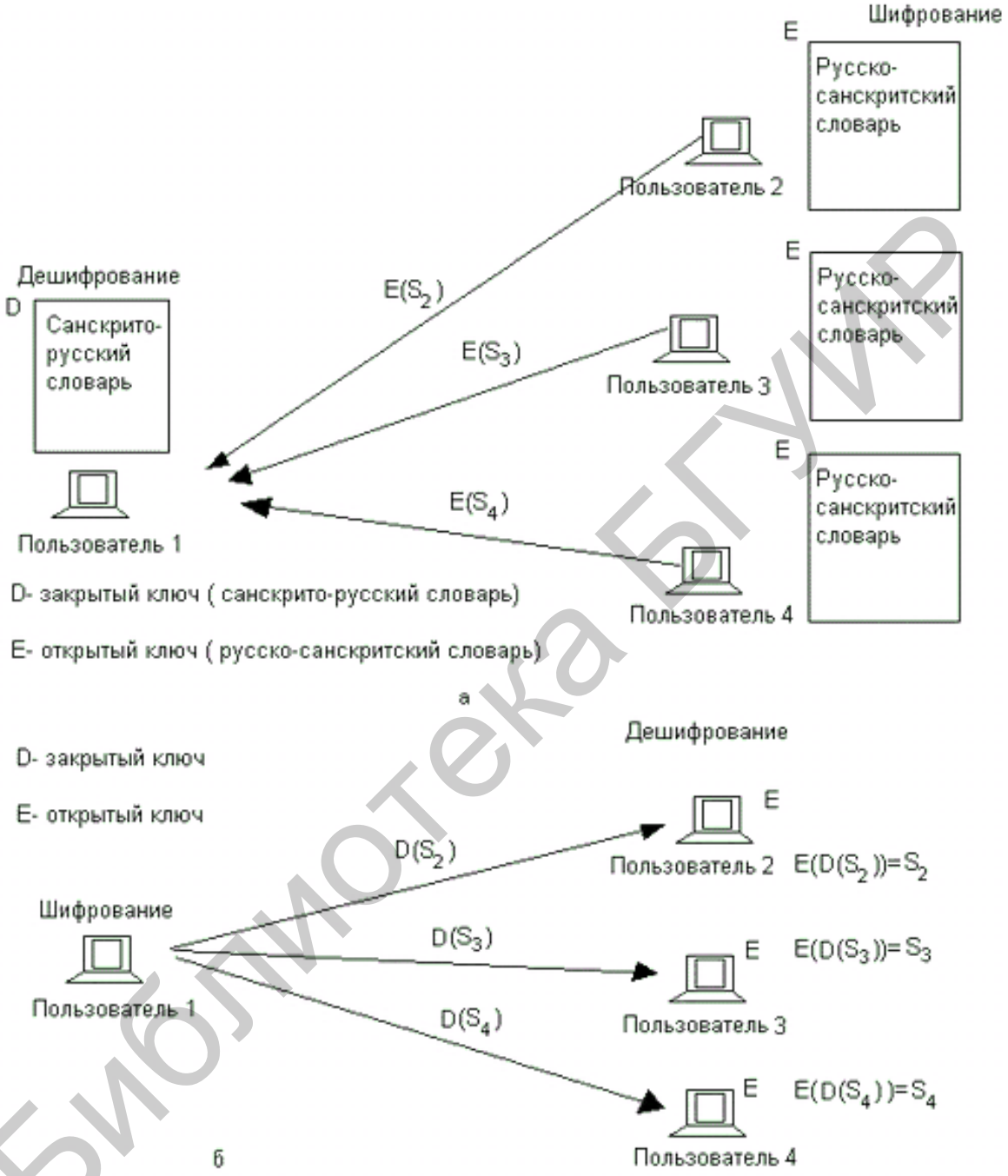

Рисунок 8 – Две схемы использования открытого и закрытого ключей

Другая схема использования открытого и закрытого ключей, целью которой является подтверждение авторства (аутентификация или электронная подпись) посылаемого сообщения, показана на рисунке 8, б. В этом случае поток сообщений имеет обратное направление – от абонента 1, обладателя закрытого ключа *D*, к его корреспондентам, обладателям открытого ключа *E*. Если абонент 1 хочет аутентифицировать себя (поставить электронную подпись), то он шифрует

известный текст своим закрытым ключом *D* и передает шифровку своим корреспондентам. Если им удается расшифровать текст открытым ключом абонента 1, то это доказывает, что текст был зашифрован его же закрытым ключом, а значит, именно он является автором этого сообщения. В этом случае сообщения S_2, S_3, S_4, адресованные разным абонентам, не являются секретными, т. к. все они – обладатели одного и того же открытого ключа, с помощью которого они могут расшифровывать все сообщения, поступающие от абонента 1.

Для того чтобы в сети все *n* абонентов имели возможность не только принимать зашифрованные сообщения, но и сами посылать таковые, каждый абонент должен обладать своей собственной парой ключей *E* и *D*. Всего в сети будет 2*n* ключей: *n* открытых ключей для шифрования и *n* секретных ключей для дешифрования. Таким образом, решается проблема масштабируемости – квадратичная зависимость количества ключей от числа абонентов в симметричных алгоритмах заменяется линейной зависимостью в несимметричных алгоритмах. Исчезает и задача секретной доставки ключа.

Хотя информация об открытом ключе не является секретной, ее нужно защищать от подлогов, чтобы злоумышленник под именем легального пользователя не навязал свой открытый ключ, после чего с помощью своего закрытого ключа он может расшифровывать все сообщения, посылаемые легальному пользователю, и отправлять свои сообщения от его имени. Проще всего было бы распространять списки, связывающие имена пользователей с их открытыми ключами широковещательно, путем публикаций в средствах массовой информации (бюллетени, специализированные журналы и т. п.). Однако при таком подходе мы снова, как и в случае с паролями, сталкиваемся с плохой масштабируемостью. Решением этой проблемы является технология цифровых сертификатов. Сертификат – это электронный документ, который связывает конкретного пользователя с конкретным ключом.

В настоящее время одним из наиболее популярных криптоалгоритмов с открытым ключом является криптоалгоритм *RSA* [7].

В 1978 году трое ученых (Ривест, Шамир и Адлеман) разработали систему шифрования с открытыми ключами *RSA* (*Rivest, Shamir, Adleman*), полностью отвечающую всем принципам Диффи – Хеллмана. Этот метод состоит в следующем:

1. Случайно выбираются два очень больших простых числа *p* и *q*.
2. Вычисляются два произведения *n*=*p***q* и *nr*=(*p*–l)*(*q*–l).
3. Выбирается случайное целое число *E*, не имеющее общих сомножителей с *m*.
4. Находится *D*, такое, что *DE*=1 по модулю *m*.
5. Исходный текст *X* разбивается на блоки таким образом, чтобы 0<*X*<*n*.

6. Для шифрования сообщения необходимо вычислить $C=XE$ по модулю n.

7. Для дешифрования вычисляется $X=CD$ по модулю n.

Таким образом, чтобы зашифровать сообщение, необходимо знать пару чисел (E, n), а чтобы дешифровать – пару чисел (D, n). Первая пара – это открытый ключ, а вторая – закрытый.

Зная открытый ключ (E, n), можно вычислить значение закрытого ключа D. Необходимым промежуточным действием в этом преобразовании является нахождение чисел p и q, для чего нужно разложить на простые множители очень большое число n, а на это требуется очень много времени. Именно с огромной вычислительной сложностью разложения большого числа на простые множители связана высокая криптостойкость алгоритма *RSA*.

Для решения задачи аутентификации информации используется концепция цифровой (или электронной) подписи.

Цифровая подпись – методы, позволяющие устанавливать подлинность автора сообщения (документа) при возникновении спора относительно авторства этого сообщения.

Основная область применения цифровой подписи – это финансовые документы, сопровождающие электронные сделки, документы, фиксирующие международные договоренности и т. п.

Схема формирования цифровой подписи показана на рисунке 9.

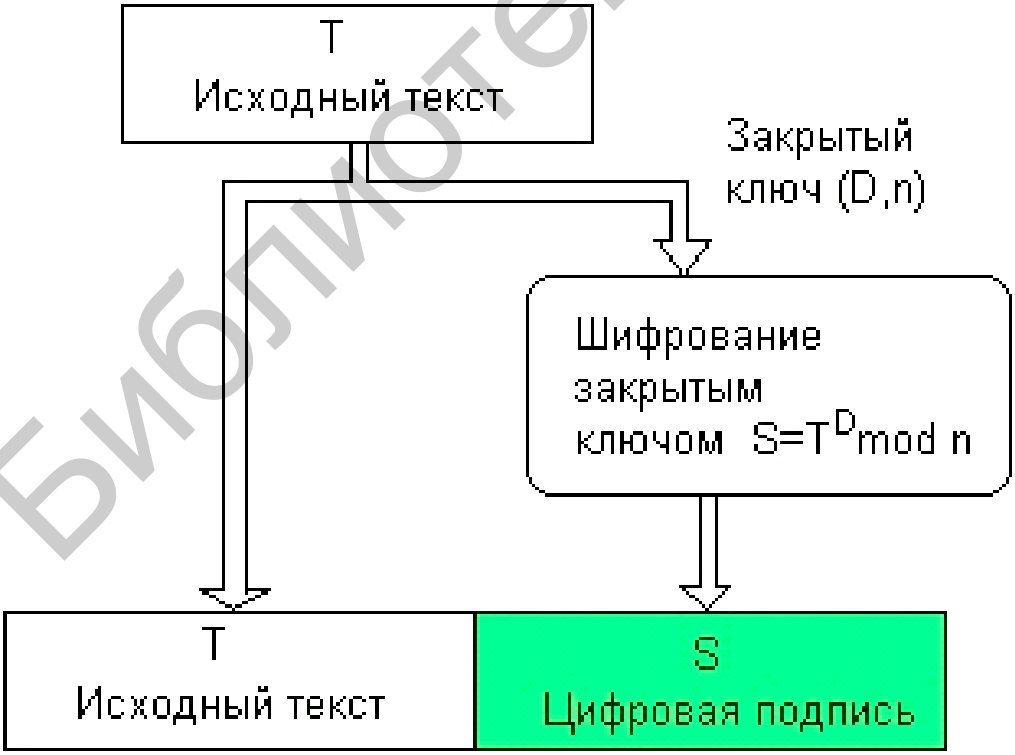

Рисунок 9 – Схема формирования цифровой подписи по алгоритму *RSA*

Если результат расшифровки цифровой подписи совпадает с открытой частью сообщения, то считается, что документ подлинный, не претерпел никаких изменений в процессе передачи, а автором его является именно тот человек, который передал свой открытый ключ получателю. Если сообщение снабжено цифровой подписью, то получатель может быть уверен, что оно не было изменено или подделано по пути.

Такие схемы аутентификации называются асимметричными.

К недостаткам данного алгоритма можно отнести то, что длина подписи в этом случае равна длине сообщения, что не всегда удобно.

Цифровые подписи применяются к тексту до того, как он шифруется.

Если помимо снабжения текста электронного документа цифровой подписью надо обеспечить его конфиденциальность, то вначале к тексту применяют цифровую подпись, а затем шифруют все вместе: и текст, и цифровую подпись.

Другие методы цифровой подписи основаны на формировании соответствующей сообщению контрольной комбинации с помощью классических алгоритмов типа *DES*. Такие алгоритмы имеют более высокую производительность по сравнению с алгоритмом *RSA* и являются более эффективными для подтверждения аутентичности больших объемов информации.

А для коротких сообщений, например, платежных поручений или квитанций подтверждения приема лучше подходит алгоритм *RSA*.

Порядок выполнения работы

1. Изучить теоретическую часть лабораторной работы.
2. Разработать программное средство, предназначенное для шифрования/расшифрования данных.
3. Реализовать электронную цифровую подпись на базе алгоритма *RSA*.
4. Предусмотреть при формировании цифровой подписи схему с использованием хэш-функций.
5. Разработать эргономичный интерфейс программы формирования и проверки электронной цифровой подписи.
6. Написать отчет.

Содержание отчета

Отчет по лабораторной работе должен быть оформлен в соответствии с требованиями, предъявляемыми к текстовым документам, и содержать:

- цель работы;
- краткие теоретические сведения, необходимые для выполнения работы;

- описание порядка выполнения работы с приведением листинга кода, скриншотов рабочих окон;

- выводы по работе.

Контрольные вопросы

1. Охарактеризуйте понятие криптографии.
2. Назовите цель и задачи криптографии.
3. На какие два класса делится криптография?
4. Назовите преимущества и недостатки криптографического шифрования.

Список использованной литературы

1. Таннебаум, Э. Современные операционные системы / Э. Таннебаум, Ч. Бос. – СПб. : Питер, 2015.
2. Русинович, М. Внутреннее устройство Windows / М. Русинович, Д. Соломон. – СПб. : Питер, 2013.
3. Буч, Г. UML / Г. Буч, А. Якобсон, Дж. Рамбо. – СПб. : Питер, 2006.
4. Зыков, С. В. Основы современного программирования. Разработка гетерогенных систем в Интернет-ориентированной среде / С. В. Зыков. – М. : Горячая линия – Телеком, 2006.
5. Голиков, В. Ф. Криптографическая защита информации в телекоммуникационных системах / В. Ф. Голиков, А. В. Курилович. – Минск : БГУИР, 2006.
6. Мейер, Б. Объектно-ориентированное конструирование программных систем / Б. Мейер. – М. : ИНТУРИТ : Русская редакция, 2005.
7. Ковалев, И. В. Учебное пособие по выполнению лабораторных работ по курсам «Операционные системы» и «Системное программное обеспечение» / И. В. Ковалев, А. С. Кузнецов, Р. Ю. Царев. – Красноярск, 2008.
8. Столлингс, В. Операционные системы / В. Столлингс. – М. : Изд. дом «Вильямс», 2004.
9. Гордеев, А. В. Системное программное обеспечение / А. В. Гордеев, А. Ю. Молчанов. – СПб. : Питер, 2002.
10. Олифер, В. Г. Сетевые операционные системы / В. Г. Олифер, Н. А. Олифер. – СПб. : Питер, 2002.

Св. план 2016, поз. 19

Учебное издание

Киринович Ирина Федоровна
Яшин Константин Дмитриевич
Быков Антон Алексеевич
Рубанова Ирина Александровна

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ

Редактор *М. А. Зайцева*
Корректор *Е. Н. Батурчик*
Компьютерная правка, оригинал-макет *М. В. Касабуцкий*

Подписано в печать 02.11.2017. Формат 60×84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 7,79. Уч.-изд. л. 8,0. Тираж 50 экз. Заказ 138.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/238 от 24.03.2014,
№2/113 от 07.04.2014, №3/615 от 07.04.2014.
ЛП №02330/264 от 14.04.2014.
220013, Минск, П. Бровки, 6