МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

Государственное образовательное учреждение
высшего профессионального образования
«Оренбургский государственный университет»

Кафедра вычислительной техники

А.Ю. КРУЧИНИН

ОПЕРАЦИОННЫЕ СИСТЕМЫ

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
К КУРСОВОМУ ПРОЕКТИРОВАНИЮ

Рекомендовано к изданию Редакционно-издательским советом
государственного образовательного учреждения
высшего профессионального образования
«Оренбургский государственный университет»

Оренбург 2009

УДК 004(07)
ББК 32.81я7
К 84

Рецензент
кандидат технических наук А.Л. Коннов

Кручинин А.Ю.
К 84 **Операционные системы: методические указания к курсовому проектированию / А.Ю. Кручинин. – Оренбург, ГОУ ОГУ, 2009. – 59 с.**

Методические указания предназначены для выполнения курсового проектирования по общепрофессиональной дисциплине «Операционные системы» для студентов специальности 230101 «Вычислительные машины, комплексы, системы и сети».

УДК 004(07)
ББК 32.81я7

Содержание

Введение

Выполнение курсового проектирования по дисциплине «Операционные системы» имеет целью закрепить теоретические знания, полученные в процессе изучения лекционного курса и выполнения лабораторного практикума, а также привить навыки самостоятельной работы.

1 Этапы проектирования

В процессе проектирования выполняются следующие этапы:

1 Ознакомление со средой программирования Visual C++ 6.0. Разработка оконного приложения Win32API.

2 Динамическое создание пунктов меню и создание формы окна средствами Win32API.

3 Получение сведений о компьютере в операционной среде Windows программными средствами.

4 Взаимодействие приложения с System Tray.

5 Создание и управление процессами.

6 Обмен информацией между процессами.

7 Управление потоками и работа с файлами средствами Win32API.

8 Синхронизация процессов и потоков.

9 Управление памятью.

10 Дочерние окна и управление «чужим» приложением.

11 Решение задачи производителя и потребителя.

По результатам курсового проектирования оформляется пояснительная записка объемом 25–40 страниц текста с необходимыми для изложения графическими материалами. Записка должна быть оформлена в соответствии с требованиями СТП 101-00 на листах формата А4 на одной стороне листа.

2 Задание на курсовое проектирование

Написать комплекс программ, решающих проблему производителя и потребителя с использованием семафоров. Имеется N производителей и M потребителей. Каждый оформлен в виде отдельного процесса. Данные процессы работают в фоновом режиме и их можно наблюдать только в диспетчере задач. Так же имеется менеджер этих процессов, который:

1) работает свернутым в системный трей;
2) отображает процессы производителей и потребителей;
3) имеет пункт меню запуска работы модели и остановки;
4) показывает состояние буфера в текущий момент времени;
5) запускает все процессы производителей и потребителей при старте менеджера;
6) удаляет все процессы производителей и потребителей при закрытии менеджера;

7) ведет счет и показывает на экране, кто сколько записал и кто сколько прочитал, взаимодействие между процессами осуществляется путем передачи сообщения WM_COPYDATA.

В качестве буфера используется текстовый файл, доступ к которому регулируется семафорами. Скорость записи и чтения данных из буфера для различных производителей и потребителей разная и задается путем передачи параметров создаваемому процессу. В результате курсового проектирования должно получиться 3 exe-файла:
Menedger.exe,
Proizv.exe,
Potreb.exe.

Менеджер запускает столько производителей и потребителей, сколько нужно, устанавливая таймер чтения и записи буфера следующим образом. Для производителей запись в буфер для первого процесса осуществляется через ***K*** секунд, для второго – через 2***K***, для третьего – через 3***K*** и т.д. Для потребителей: ***L*** секунд, 2***L***, 3***L*** и т.д. Варианты заданий для студентов перечислены в таблице 1.

Таблица 1 – Варианты заданий

№ варианта	Условия задания	Буква
1	N=4, M=4, K=1, L=0.7	А
2	N=5, M=3, K=0.8, L=1	Б
3	N=3, M=5, K=1, L=1.2	В
4	N=3, M=2, K=1, L=0.4	Г
5	N=6, M=2, K=2, L=1	Д
6	N=3, M=3, K=1, L=1	Е
7	N=4, M=7, K=1.5, L=2	Ж
8	N=1, M=6, K=0.5, L=1	З
9	N=1, M=5, K=1, L=1.3	И
10	N=6, M=1, K=1.6, L=1.4	К
11	N=7, M=2, K=1, L=0.7	Л
12	N=2, M=5, K=0.3, L=0.9	М
13	N=3, M=3, K=1.2, L=1.7	Н
14	N=7, M=2, K=2, L=0.5	О
15	N=8, M=6, K=0.3, L=0.4	П
16	N=4, M=7, K=0.4, L=0.4	Р
17	N=2, M=5, K=0.7, L=0.9	С
18	N=6, M=3, K=1, L=1.1	Т

Продолжение таблицы 1

19	N=4, M=8, K=1.1, L=1.7	У
20	N=1, M=9, K=0.2, L=0.7	Ф
21	N=9, M=1, K=2, L=0.5	Х
22	N=5, M=4, K=1, L=1	Ц
23	N=6, M=6, K=1.2, L=1.5	Ч
24	N=6, M=3, K=0.8, L=0.9	Ш
25	N=3, M=5, K=1.2, L=1.4	Щ
26	N=2, M=5, K=1, L=1.7	Э
27	N=5, M=3, K=0.8, L=0.4	Ю
28	N=6, M=2, K=1, L=1.2	Я

Перед выполнением курсового проекта студент уточняет номер выполняемого им варианта – ключом к выбору задания является начальная буква фамилии. Например: студент Сидоров для выполнения курсового проекта выбирает 17 номер.

3 Теоретические сведения для выполнения задания

Рассмотрим некоторые примитивы межпроцессного взаимодействия, применяющиеся вместо циклов ожидания, в которых лишь напрасно расходуется процессорное время. Эти примитивы блокируют процессы в случае запрета на вход в критическую область. Одной из простейших является пара примитивов **sleep** и **wakeup**. Примитив **sleep** – системный запрос, в результате которого вызывающий процесс блокируется, пока его не запустит другой процесс. У запроса **wakeup** есть один параметр – процесс, который следует запустить. Также возможно наличие одного параметра у обоих запросов – адреса ячейки памяти, используемой для согласования запросов ожидания и запуска.

Два процесса совместно используют буфер ограниченного размера. Один из них, производитель, помещает данные в буфер, а потребитель считывает их оттуда. Трудности начинаются в тот момент, когда производитель хочет поместить в буфер очередную порцию данных и обнаруживает, что буфер полон. Для производителя решением является ожидание, пока потребитель полностью или частично не очистит буфер. Аналогично, если потребитель хочет забрать данные из буфера, а буфер пуст, потребитель уходит в состояние ожидания и выходит из него, как только производитель положит что-нибудь в буфер и разбудит его.

Это решение кажется достаточно простым, но оно приводит к состояниям состязания. Нам нужна переменная **count** для отслеживания количества элементов в буфере. Если максимальное число элементов, хранящихся в буфере, равно ***N***, программа производителя должна проверить, не равно ли ***N*** значение **count** прежде, чем поместить в буфер следующую порцию данных. Если значение **count** равно ***N***, то производитель уходит в состояние ожидания; в противном случае производитель

помещает данные в буфер и увеличивает значение **count**.

Код программы потребителя прост: сначала проверить, не равно ли значение **count** нулю. Если равно, то уйти в состояние ожидания; иначе забрать порцию данных из буфера и уменьшить значение **count**. Каждый из процессов также должен проверять, не следует ли активизировать другой процесс, и в случае необходимости проделывать это. Программы обоих процессов представлены в листинге 1.

```
#define N 100
int count = 0;

void producer()
{
        int item;
        while (TRUE) {
                item=produce_item();          //сформировать следующий элемент
                if (count==N) sleep();        //буфер полон – состояние ожидания
                insert_item(item);            //поместить элемент в буфер
                count++;
                if (count==1) wakeup(consumer);
        }
}

void consumer()
{
        int item;
        while (TRUE) {
                if (count==0) sleep();        //буфер пуст – состояние ожидания
                item=remove_item(item);       //забрать элемент из буфера
                count--;
                if (count==N-1)  wakeup(producer);
        }
}
```

Листинг 1 – Проблема производителя и потребителя с неустранимым состоянием соревнования

Для описания на языке С системных вызовов **sleep** и **wakeup** мы представили их в виде вызовов библиотечных процедур. В стандартной библиотеке С их нет, но они будут доступны в любой системе, в которой присутствуют такие системные вызовы. Процедуры **insert_item** и **remove_item** помещают элементы в буфер и извлекают их оттуда.

Возникновение состояния состязания возможно, поскольку доступ к переменной **count** не ограничен. Может возникнуть следующая ситуация: буфер пуст, и потребитель только что считал значение переменной **count**, чтобы проверить, не равно ли оно нулю. В этот момент планировщик передал управление производителю, производитель поместил элемент в буфер и увеличил значение **count**, проверив, что теперь оно стало равно 1. Зная, что перед этим оно было равно 0 и потребитель находился в состоянии ожидания, производитель активизирует его с помощью вызова **wakeup**.

Но потребитель не был в состоянии ожидания, так что сигнал активизации пропал впустую. Когда управление перейдет к потребителю, он вернется к считанному когда-то значению **count**, обнаружит, что оно равно 0, и уйдет в состояние ожидания. Рано или поздно производитель наполнит буфер и также уйдет в состояние ожидания. Оба процесса так и останутся в этом состоянии.

Суть проблемы в данном случае состоит в том, что сигнал активизации, пришедший к процессу, не находящемуся в состоянии ожидания, пропадает. Если бы не это, проблемы бы не было. Быстрым решением может быть добавление бита ожидания активизации. Если сигнал активизации послан процессу, не находящемуся в состоянии ожидания, этот бит устанавливается. Позже, когда процесс пытается уйти в состояние ожидания, бит ожидания активизации сбрасывается, но процесс остается активным. Этот бит исполняет роль копилки сигналов активизации.

Несмотря на то что введение бита ожидания запуска спасло положение в этом примере, легко сконструировать ситуацию с несколькими процессами, в которой одного бита будет недостаточно. Мы можем добавить еще один бит, или 8, или 32, но это не решит проблему.

В 1965 году Дейкстра (E. W. Dijkstra) предложил использовать целую переменную для подсчета сигналов запуска, сохраненных на будущее. Им был предложен новый тип переменных, так называемые семафоры, значение которых может быть нулем (в случае отсутствия сохраненных сигналов активизации) или некоторым положительным числом, соответствующим количеству отложенных активизирующих сигналов.

Дейкстра предложил две операции, **down** и **up** (обобщения **sleep** и **wakeup**). Операция **down** сравнивает значение семафора с нулем. Если значение семафора больше нуля, операция **down** уменьшает его (то есть расходует один из сохраненных сигналов активации) и просто возвращает управление. Если значение семафора равно нулю, процедура down не возвращает управление процессу, а процесс переводится в состояние ожидания. Все операции проверки значения семафора, его изменения и перевода процесса в состояние ожидания выполняются как единое и неделимое элементарное действие. Тем самым гарантируется, что после начала операции ни один процесс не получит доступа к семафору до окончания или блокирования операции. Элементарность операции чрезвычайно важна для разрешения проблемы синхронизации и предотвращения состояния состязания.

Операция **up** увеличивает значение семафора. Если с этим семафором связаны один или несколько ожидающих процессов, которые не могут завершить более раннюю операцию **down**, один из них выбирается системой (например, случайным образом) и ему разрешается завершить свою операцию **down**. Таким образом, после операции **up**, примененной к семафору, связанному с несколькими ожидающими процессами, значение семафора так и останется равным 0, но число ожидающих процессов уменьшится на единицу. Операция увеличения значения семафора и активизации процесса тоже неделима. Ни один процесс не может быть блокирован во время выполнения операции **up**, как ни один процесс не мог быть блокирован во время выполнения операции **wakeup** в предыдущей модели.

В оригинале Дейкстра использовал вместо **down** и **up** обозначения P и V соответственно. Мы не будем в дальнейшем использовать оригинальные

обозначения, поскольку тем, кто не знает датского языка, эти обозначения ничего не говорят (да и тем, кто знает язык, говорят немного). Впервые обозначения down и up появились в языке Algol 68.

Как показано в листинге 2, проблему потерянных сигналов запуска можно решить с помощью семафоров. Очень важно, чтобы они были реализованы неделимым образом. Стандартным способом является реализация операций down и up в виде системных запросов, с запретом операционной системой всех прерываний на период проверки семафора, изменения его значения и возможного перевода процесса в состояние ожидания. Поскольку для выполнения всех этих действий требуется всего лишь несколько команд процессора, запрет прерываний не приносит никакого вреда. Если используются несколько процессоров, каждый семафор необходимо защитить переменной блокировки с использованием команды TSL, чтобы гарантировать одновременное обращение к семафору только одного процессора. Необходимо понимать, что использование команды TSL принципиально отличается от активного ожидания, при котором производитель или потребитель ждут наполнения или опустошения буфера. Операция с семафором займет несколько микросекунд, тогда как активное ожидание может затянуться на существенно больший промежуток времени.

```
#define N 100                   /* количество сегментов в буфере */
typedef int semaphore;          /* семафоры - особый вид целочисленных переменных */
semaphore mutex = 1;            /* контроль доступа в критическую область */
semaphore empty = N;            /* число пустых сегментов буфера */
semaphore full = 0;             /* число полных сегментов буфера */
void producer(void)
{
   int item;
   while (TRUE) {               /* TRUE - константа, равная 1*/
      item = produce_item();    /* создать данные, помещаемые в буфер */
      down(&empty);             /* уменьшить счетчик пустых сегментов буфера */
      down(&mutex);             /* вход в критическую область */
      insert_item(item);        /* поместить в буфер новый элемент */
      up(&mutex);               /* выход из критической области */
      up(&full);                /* увеличить счетчик полных сегментов буфера */
   }
}
void consumer(void)
{
   int item;
   while (TRUE) {               /* бесконечный цикл */
      down(&full);              /* уменьшить числа полных сегментов буфера */
      down(&mutex);             /* вход в критическую область */
      item = remove_item();     /* удалить элемент из буфера */
      up(&mutex);               /* выход из критической области */
      up(&empty);               /* увеличить счетчик пустых сегментов буфера */
      consume_item(item);       /* обработка элемента */
   }
}
```

Листинг 2 – Проблема производителя и потребителя с семафорами

В представленном решении используются три семафора: один для подсчета заполненных сегментов буфера (**full**), другой для подсчета пустых сегментов (**empty**), а третий предназначен для исключения одновременного доступа к буферу производителя и потребителя (**mutex**). Значение счетчика **full** исходно равно нулю, счетчик **empty** равен числу сегментов в буфере, а **mutex** равен 1. Семафоры, исходное значение которых равно 1, используемые для исключения одновременного нахождения в критической области двух процессов, называются двоичными семафорами. Взаимное исключение обеспечивается, если каждый процесс выполняет операцию **down** перед входом в критическую область и **up** после выхода из нее.

4 Методические указания к выполнению этапов проекта

4.1 Ознакомление со средой программирования Visual C++ 6.0. Разработка оконного приложения Win32API

Среда программирования Visual C++ 6.0 была разработана в 1998 году. Однако до сих пор ею пользуются многие разработчики консольных и написанных на чистом Win32 API приложениях, так как она облает отличным компилятором, нетребовательна к компьютерам (можно работать даже на Pentium I), занимает мало места и может работать без установки на компьютер – путём простого копирования. Внешний вид среды представлен на рисунке 1.

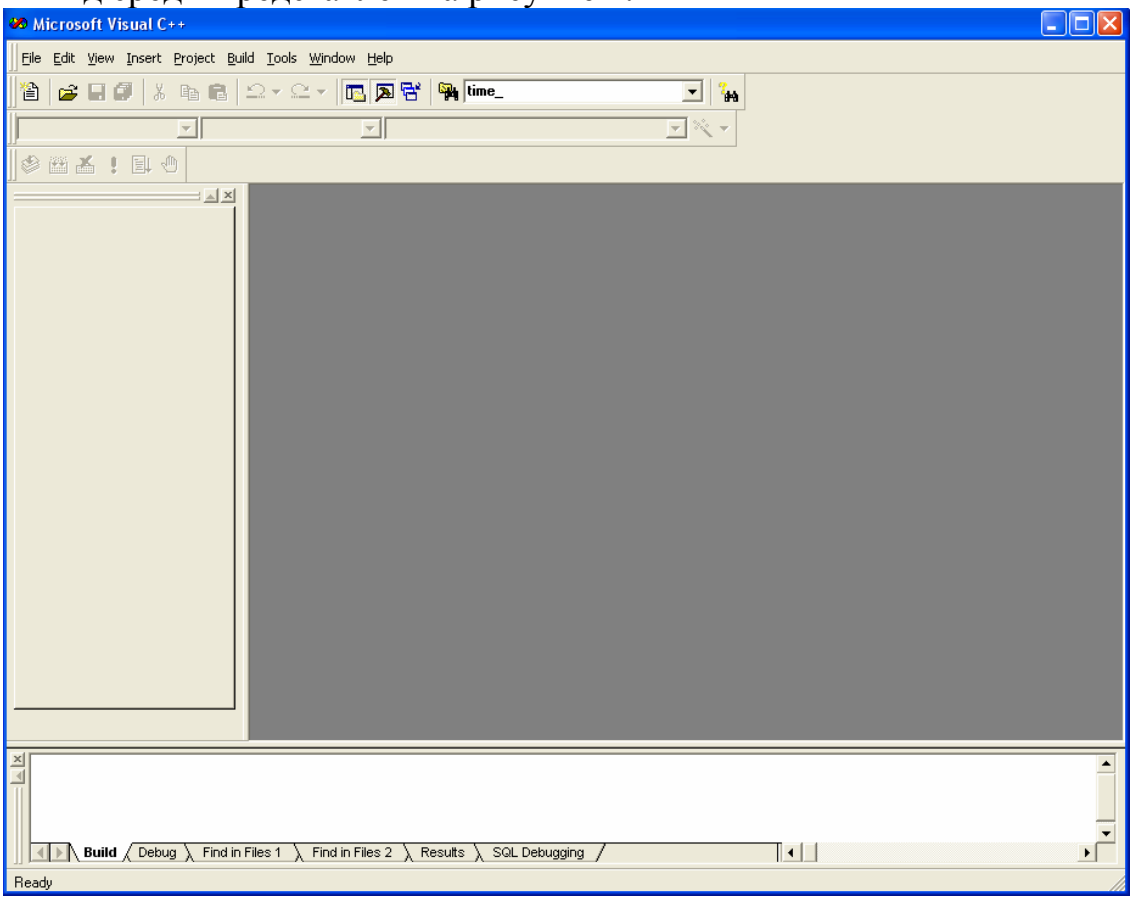

Рисунок 1 – Среда программирования Visual C++ 6.0

Перед тем, как начать программировать, необходимо проверить состояние настроек среды, которые находятся в Tools→Options→Directories (Рисунок 2).

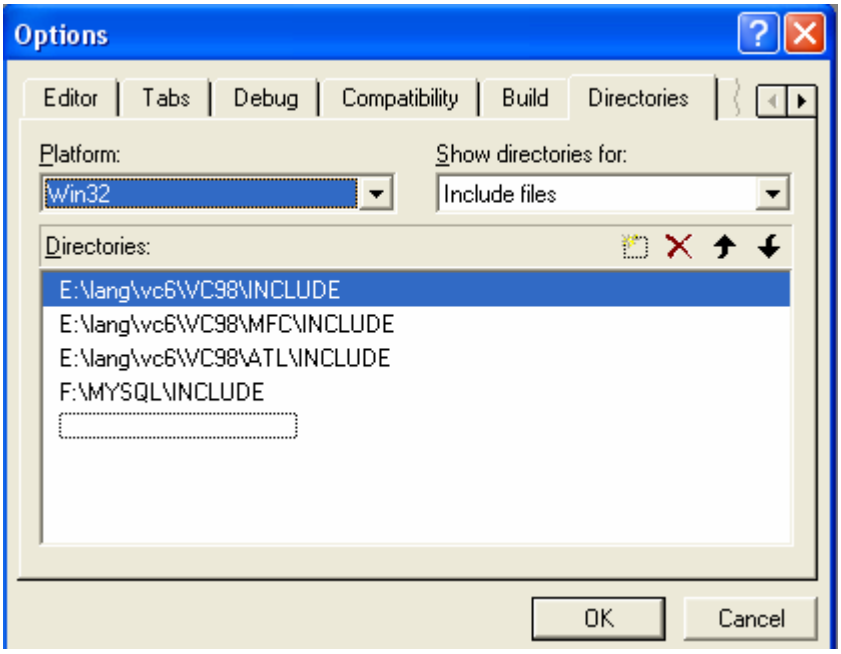

Рисунок 2 – Окно настроек директорий

Пути до include и library файлов должны соответствовать их действительному местонахождению. Сюда же вы можете добавлять и свои пути.

Для создания нового проекта выберете File→New, в результате у вас появится окно (Рисунок 3).

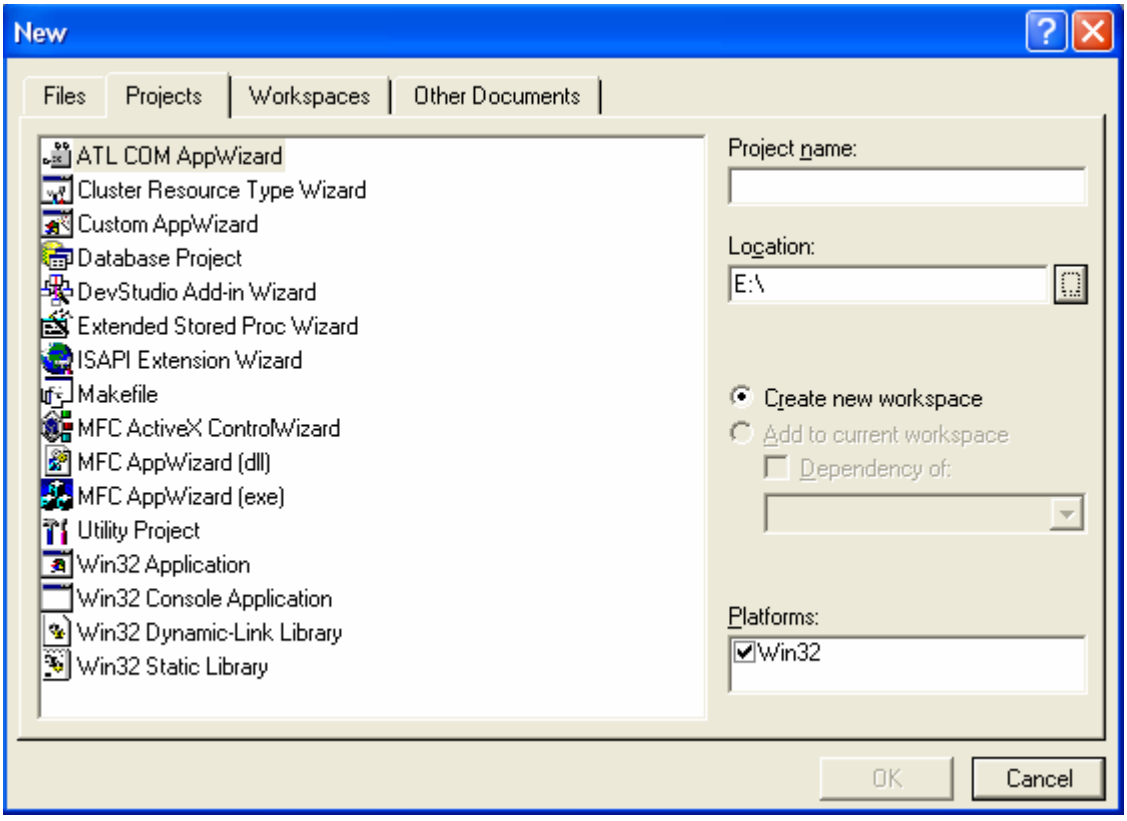

Рисунок 3 – Окно создания нового проекта

Выберете из списка пункт «Win32 Application», в графу «Project Name» внесите название проекта и выберите путь до него в графе «Location», после этого нажмите «OK» и у вас появится следующее окно (Рисунок 4).

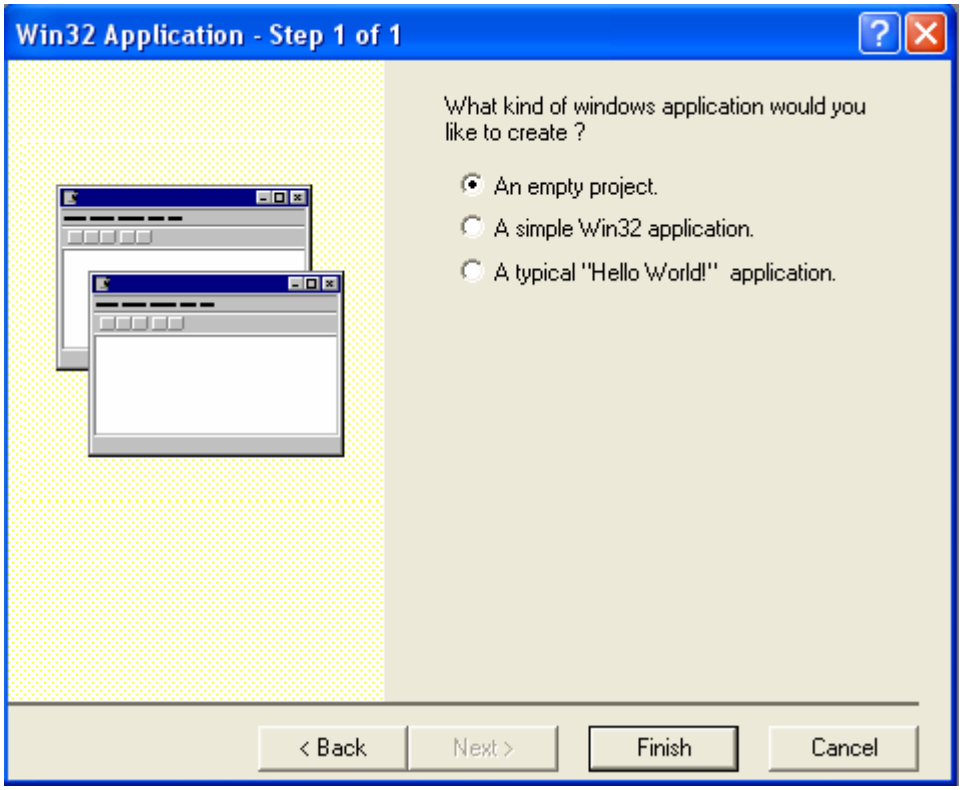

Рисунок 4 – Окно Шага 1 создания проекта

Нажмите «Finish», и у вас создастся пустой проект. Теперь можно добавлять файлы в проект. Выберите Project→Add to Project→New – появится окно (Рис. 5).

Выберите «C++ Source File» и наберите в графе «File name» имя файла: main. Затем нажмите «OK» – у вас добавился файл main.cpp. Аналогичным образом добавьте «C/C++ Header File» – у вас появится файл main.h.

Для создания вашего первого приложения на Win32API заполните эти файлы следующим ниже содержимым.

```
#ifndef _MAIN_
#define _MAIN_ 1

#include <windows.h>

HINSTANCE hInst;

//функция обработки сообщений
LRESULT CALLBACK WIN32PROC(HWND hWnd,UINT Message,UINT wParam,LONG lParam);


#endif
```

Листинг 3 – Файл main.h

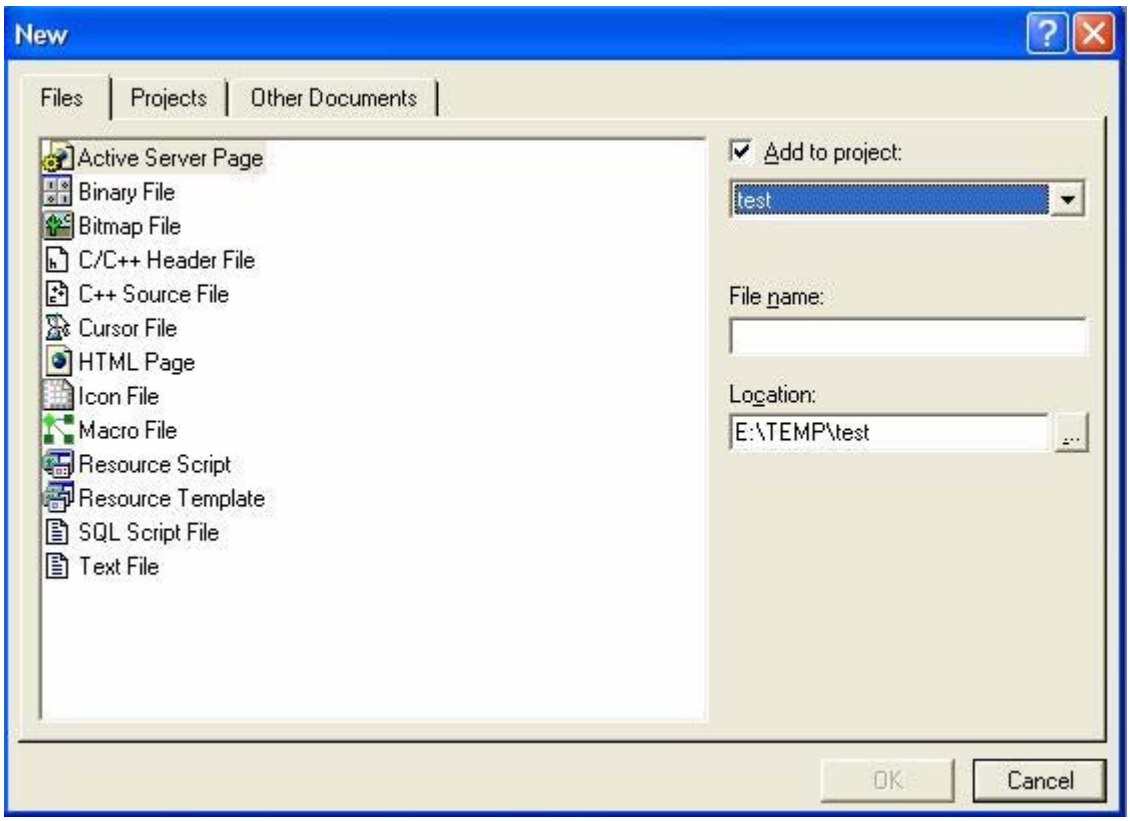

Рисунок 5 – Окно добавления новых файлов

```
#include "main.h"

//Главная функция
int APIENTRY WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,
                LPSTR lpszCmdParam,int nCmdShow)
{
        hInst=hInstance;

        WNDCLASS WndClass;
   MSG Msg;
   char ClassName[]="MYPROJECT";
        HWND hWnd;

        WndClass.style=CS_HREDRAW | CS_VREDRAW;
   WndClass.lpfnWndProc=WIN32PROC;
   WndClass.cbClsExtra=0;
   WndClass.cbWndExtra=0;
   WndClass.hInstance=hInstance;
   WndClass.hIcon=NULL;
   WndClass.hCursor=LoadCursor(NULL,IDC_ARROW);
   WndClass.hbrBackground=(HBRUSH)GetStockObject(BLACK_BRUSH);;
   WndClass.lpszMenuName=NULL;
   WndClass.lpszClassName=ClassName;

   if (!RegisterClass(&WndClass))
        {
                MessageBox(NULL,"Cannot register class","Error",MB_OK | MB_ICONERROR);
                return 0;
```

```
        }


    hWnd=CreateWindowEx(0, ClassName,"Моя первая программа",
                WS_OVERLAPPEDWINDOW,
                CW_USEDEFAULT,CW_USEDEFAULT,
                CW_USEDEFAULT,CW_USEDEFAULT,
                NULL,NULL,hInstance,NULL);

        if (hWnd==NULL)
        {
                MessageBox(NULL,"Cannot create window","Error",MB_OK | MB_ICONERROR);
                return 0;
        }

        ShowWindow(hWnd,nCmdShow);
    UpdateWindow(hWnd);

        MSG msg;

    while (1) {
       while (PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE) == TRUE)
       {
          if (GetMessage(&msg, NULL, 0, 0) )
          {
             TranslateMessage(&msg);
             DispatchMessage(&msg);
          } else {
             return TRUE;
          }
       }

    }

    return Msg.wParam;

}

//функция обработки сообщений
LRESULT CALLBACK WIN32PROC(HWND hWnd,UINT Message,
                    UINT wParam,LONG lParam)
{
        switch(Message)
    {
        case WM_DESTROY:
                PostQuitMessage(0);
                break;
        }

        return DefWindowProc(hWnd,Message,wParam,lParam);
}
```

Листинг 4 – Файл main.cpp

Сохраните всё и откомпилируйте, используя F7 или через пункт меню «Build». После запуска у вас должно появиться чёрное окно. Ваша первая программа на Win32 API готова.

Кратко опишем то, что происходит внутри функции **WinMain**. Первоначально заполняется структура окна **WNDCLASS**, в которой устанавливаются основные свойства окна. После этого с помощью функции **RegisterClass** ваш класс регистрируется в Windows. Как вы заметили, у вашей программы отсутствует иконка, чтобы добавить её необходимо воспользоваться редактором ресурсов. Для этого выберите Project→Add to Project→New, а там выберите «Resource Script» и в графе «File name» – res. У вас должно создаться два файл: res.rc и resource.h. Ниже списка файлов в проекте у вас должна появиться закладка «Resource View». Переключитесь на закладку и, щёлкнув правой кнопкой на «res resources», выберите «Insert» – появится окно (Рисунок 6).

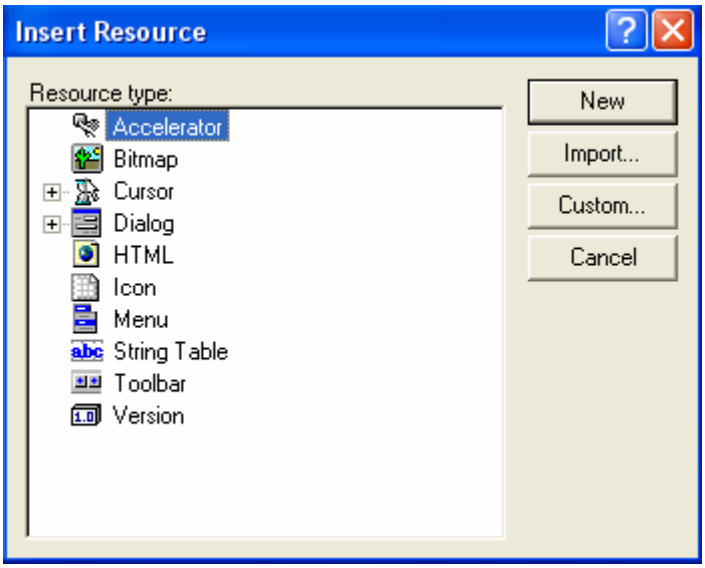

Рисунок 6 – Окно добавления ресурсов

Выберите «Icon» и нажмите «New». Отредактируйте и сохраните иконку. Теперь можно подключать её к программе. Сначала подключите к проекту файл resource.h с помощью Project→Add to Project→Files. Добавьте в основную программу строку: #include "resource.h" (ниже строки #include "main.h"). Замените строку в создании класса

```
WndClass.hIcon=NULL; на
WndClass.hIcon=LoadIcon(hInstance,MAKEINTRESOURCE(IDI_ICON1));
```

Здесь IDI_ICON1 указывает на вашу иконку. Компилируйте проект – иконка присоединена. Описание остальных элементов структуры приведено ниже.

Структура WNDCLASS.

```
typedef struct tagWNDCLASS
{
    UINT style;
    WNDPROC lpfnWndProc;
    int cbClsExtra;
```

```
    int cbWndExtra;
    HINSTANCE hInstance;
    HICON hIcon;
    HCURSOR hCursor;
    HBRUSH hbrBackground;
    LPCSTR lpszMenuName;
    LPCSTR lpszClassName;
} WNDCLASS;
```

style – определяет стиль класса. Стили можно объединять, используя |. Вот какие они бывают:
CS_BYTEALIGNCLIENT – (по горизонтали) выравнивание рабочей области окна по границе байта. Влияет на ширину окна и его горизонтальное положение на экране;
CS_BYTEALIGNWINDOW – (по вертикали) выравниается окна по границе байта;
CS_CLASSDC – контекст устройства, который будет разделяться всеми окнами класса. При нескольких потоках операционная система разрешит доступ только одному потоку;
CS_DBLCLKS – посылать сообщение от мыши при двойном щелчке в пределах класса окна;
CS_GLOBALCLASS – Создавать глобальный класс, который можно поместить в динамическую библиотеку DLL;
CS_HREDRAW – перерисовывать всё окно при изменении ширины;
CS_NOCLOSE – отключить команду «Закрыть»;
CS_OWNDC – у каждого окна уникальный контекст устройства;
CS_PARENTDC – у дочернего окна будет область отсечки от родительского. Повышает производительность;
CS_SAVEBITS – позволяет сохранять область экрана в виде битовой матрицы закрытую в данный момент другим окном, используется для восстановления экрана;
CS_VREDRAW – перерисовывать окно при изменении вертикальных размеров.
WNDPROC – указатель на процедуру окна вызываемую функцией **CallWindowProc**.
cbClsExtra – объем памяти выделяемый за структурой класса.
cbWndExtra – объем дополнительной памяти за экземпляром окна.
hInstance – дескриптор экземпляра.
hIcon – дескриптор иконы окна.
hCursor – дескриптор курсора окна.
hbrBackground – дескриптор для закраски фона.
lpszMenuName – имя меню в ресурсах.
lpszClassName – имя класса.

После регистрации класса окна вызывается функция **CreateWindowEx**, предназначенная для создания окна.

Функция CreateWindowEx

```
HWND CreateWindowEx
```

```
(
    DWORD dwExStyle,          // улучшенный стиль окна
    LPCTSTR lpClassName,      // указатель на зарегистрированное имя класса
    LPCTSTR lpWindowName,     // указатель на имя окна
    DWORD dwStyle,            // стиль окна
    int x,                    // горизонтальная позиция окна
    int y,                    // вертикальная позиция окна
    int nWidth,               // ширина окна
    int nHeight,              // высота окна
    HWND hWndParent,          // дескриптор родительского или окна
                              // собственника
    HMENU hMenu,              // дескриптор меню или идентификатор
                              // дочернего окна
    HINSTANCE hInstance,      // дескриптор экземпляра прикладной
                              // программы
    LPVOID lpParam            // указатель на данные создания окна
);
```

Параметры:

dwExStyle – определяет расширенный стиль окна. Этот параметр может быть одно из следующих значений:

WS_EX_ACCEPTFILES – Определяет, что окно, созданное с этим стилем принимает файлы при помощи информационной технологии «перетащи и вставь».

WS_EX_APPWINDOW – Активизирует окно верхнего уровня на панель задач, когда окно свернуто.

WS_EX_CLIENTEDGE – Определяет, что окно имеет рамку с углубленным краем.

WS_EX_CONTEXTHELP – Включает вопросительный знак в строку заголовка окна. Когда пользователь щелкает мышью по вопросительному знаку, курсор меняется на вопросительный знак с указателем. Если пользователь затем щелкает мышью по дочернему окну, потомок принимает сообщение WM_HELP. Дочернее окно должно передать сообщение родительской оконной процедуре, которая должна вызваться функцией **WinHelp**, использующей команду HELP_WM_HELP. Прикладная программа Справки показывает выскакивающее окно, которое обычно содержит справку для дочернего окна. WS_EX_CONTEXTHELP не может использоваться со стилями WS_MAXIMIZEBOX или WS_MINIMIZEBOX.

WS_EX_CONTROLPARENT – Позволяет пользователю передвигаться среди дочерних окон основного окна, используя клавишу табуляции (TAB).

WS_EX_DLGMODALFRAME – Создает окно, которое имеет двойную рамку; окно может быть создано (необязательно) со строкой заголовка, которую определяет стиль WS_CAPTION в параметре dwStyle.

WS_EX_LEFT – Окно имеет общеупотребительные свойства «выравнивания по левой границе». Это – по умолчанию.

WS_EX_LEFTSCROLLBAR – Если язык оболочки Еврейский, Арабский или другой язык, который придерживается иного порядка чтения, вертикальная линейка прокрутки (если появляется) – слева от рабочей области. Для других языков, этот стиль игнорируется и не обрабатывается как ошибка.

WS_EX_LTRREADING – Текст окна отображается, используя свойство порядка чтения «Слева – Направо». Это – по умолчанию.

WS_EX_MDICHILD – Создает MDI дочернее окно.
WS_EX_NOPARENTNOTIFY – Определяет, что дочернее окно, созданное с этим стилем не посылает сообщение WM_PARENTNOTIFY родительскому окну, когда оно создается или разрушается.
WS_EX_OVERLAPPEDWINDOW – Объединяет стили WS_EX_CLIENTEDGE и WS_EX_WINDOWEDGE.
WS_EX_PALETTEWINDOW – Объединяет стили WS_EX_WINDOWEDGE, WS_EX_TOOLWINDOW и WS_EX_TOPMOST.
WS_EX_RIGHT – Окно имеет общеупотребительные свойства «выравнивание по правому краю». Оно зависит от класса окна. Этот стиль имеет эффект только тогда, если язык оболочек Еврейский, Арабский или другой язык, который поддерживает иной порядок выравнивания для чтения; иначе, стиль игнорируется и не обрабатывается как ошибка.
WS_EX_RIGHTSCROLLBAR – Вертикальная линейка прокрутки (если появляется) – справа от рабочей области. Это - по умолчанию.
WS_EX_RTLREADING – Если язык оболочки Еврейский, Арабский или другой язык, который придерживается иного порядка выравнивания для чтения, текст в окне отображается, используя свойства порядка чтения «Справа – Налево». Для других языков, стиль игнорируется и не обрабатывается как ошибка.
WS_EX_STATICEDGE – Создает окно с трехмерным стилем рамки, предполагается использовать для элементов, которые не принимают вводимую информацию от пользователя.
WS_EX_TOOLWINDOW – Создает окно инструментальных средств; то есть окно предполагается использовать как плавающую инструментальную панель. Окно инструментальных средств имеет строку заголовка, которая является короче, чем нормальная строка заголовка, а заголовок окна выводится, с использованием меньшего шрифта. Окно инструментальных средств не появляется в панели задач или в диалоговом окне, которое появляется, когда пользователь нажимает ALT+TAB.
WS_EX_TOPMOST – Определяет, что окно, созданное с этим стилем должно быть размещено выше всех, не самых верхних окон и должно стоять выше их, даже тогда, когда окно дезактивировано. Чтобы добавить или удалить этот стиль, используйте функцию **SetWindowPos**.
WS_EX_TRANSPARENT – Определяет, что окно, созданное с этим стилем должно быть прозрачным. То есть любые окна, которые появляются из-под окна, не затеняются им. Окно, созданное с этим стилем принимает WM_PAINT сообщения только после того, как все сестринские окна под ним модифицировались.
WS_EX_WINDOWEDGE – Определяет, что окно имеет рамку с выпуклым краем.

Использование стиля WS_EX_RIGHT для статического или редактируемого элементов управления имеет тот же самый эффект как и использование стиля SS_RIGHT или ES_RIGHT, соответственно. Использование этого стиля с командными кнопками имеет тот же самый эффект как и использование стилей BS_RIGHT и BS_RIGHTBUTTON.

lpClassName – указывает на строку с нулевым символом в конце или на целочисленный атом. Если **lpClassName** – атом, он должен быть глобальным

атомом, созданным предыдущим вызовом функции **GlobalAddAtom**. Атом, 16-разрядное значение меньше чем 0xC000, должно быть младшим словом в **lpClassName**; старшее слово должно быть нулевое. Если **lpClassName** – строка, она определяет имя класса окна. Имя класса может быть любое имя, зарегистрированное функцией **RegisterClass** или любым из предопределенных имен класса элементов управления.
lpWindowName – указывает на строку с нулевым символом в конце, которая определяет имя окна.
dwStyle – определяет стиль создаваемого окна. Основные стили:
WS_BORDER – Создание окна с рамкой.
WS_CAPTION – Создание окна с заголовком (невозможно использовать одновременно со стилем WS_DLGFRAME).
WS_CHILD, 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 обрисовки в дочернем окне областей клипа, 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_MINIMIZE
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 popup-окно (невозможно использовать совместно со стилем WS_CHILD.
WS_POPUPWINDOW – Создает popup-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 – Определяет элементы управления, переход к которым может быть выполнен по клавише TAB.
WS_THICKFRAME – Создает окно с рамкой, используемой для изменения размера окна.
WS_VISIBLE – Создает первоначально неотображаемое окно.
WS_VSCROLL – Создает окно с вертикальной полосой прокрутки.

x – определяет начальную горизонтальную позицию окна. Для перекрывающего или выскакивающего окна параметр **x** – начальная x-координата левого верхнего угла окна, в экранных координатах устройства. Для дочернего окна **x** – x-координата левого верхнего угла окна относительно левого верхнего угла рабочей области родительского окна. Если **x** установлен как CW_USEDEFAULT, Windows выбирает заданную по умолчанию позицию для левого верхнего угла окна и игнорирует **y** параметр. Стиль CW_USEDEFAULT допустим только для перекрывающих окон; если он определен для всплывающего или дочернего окна параметры **x** и **y** устанавливаются в нуль.
y – определяет начальную вертикальную позицию окна. Для перекрывающего или выскакивающего окна, параметр **y** – начальная y-координата левого верхнего угла окна, в экранных координатах устройства. Для дочернего окна, **y** – начальная y-координата левого верхнего угла дочернего окна относительно левого верхнего угла рабочей области родительского окна. Для окна со списком, **y** – начальная y-координата левого верхнего угла рабочей области окна со списком относительно левого верхнего угла рабочей области родительского окна. Если перекрывающее окно создано в стиле WS_VISIBLE с набором битов, а параметр **x** установлен как CW_USEDEFAULT, Windows игнорирует параметр **y**.
nWidth – определяет ширину окна в единицах измерения устройства. Для перекрывающих окон **nWidth** – ширина окна в экранной системе координат или CW_USEDEFAULT. Если **nWidth** – CW_USEDEFAULT, Windows выбирает заданную по умолчанию ширину и высоту для окна; заданная по умолчанию ширина простирается от начальных x-координат до правого края экрана; заданная по умолчанию высота простирается от начальной y-координаты до верхней части области значка. Стиль CW_USEDEFAULT допустим только для перекрывающих окон; если CW_USEDEFAULT определен для выскакивающего или дочернего окна, параметры **nWidth** и **nHeight** устанавливаются в нуль.
nHeight – определяет высоту окна в единицах измерения устройства. Для перекрывающих окон, **nHeight** – высота окна в экранной системе координат. Если параметр **nWidth** установлен как CW_USEDEFAULT, Windows игнорирует **nHeight**.
hWndParent – идентифицирует родительское окно или владельца создаваемого окна. Допустимый дескриптор окна должен быть дан, когда дочернее окно или находящееся в собственности окно созданы. Дочернее окно ограничено рабочей областью родительского окна. Находящееся в собственности окно - перекрывающее окно, которое разрушается, когда окно его владельца разрушено или скрыто, когда его владелец свернут; оно всегда отображается на верхней части окна его владельца. Несмотря на то, что этот параметр должен определять правильный дескриптор, если параметр **dwStyle** включает в себя стиль WS_CHILD, это необязательно, если **dwStyle** включает в себя стиль WS_POPUP.
hMenu – идентифицирует меню или, в зависимости от стиля окна, определяет идентификатор дочернего окна. Для перекрывающего или выскакивающего окна, **hMenu** идентифицирует меню, которое будет использоваться окном; этот параметр может быть значением NULL, если меню класса будет использовано. Для дочернего окна, **hMenu** определяет идентификатор дочернего окна, целочисленное значение, используемое элементом управления диалогового окна, что-бы сообщать родителю

о событиях. Прикладная программа определяет идентификатор дочернего окна; он должно быть уникальным для всех дочерних окон того же самого родительского окна.
hInstance – идентифицирует экземпляр модуля, который будет связан с окном.
lpParam – указывает на значение, переданное окну через структуру **CREATESTRUCT**, вызванную параметром **lParam** сообщения WM_CREATE. Если прикладная программа вызвала **CreateWindow**, чтобы создать пользовательское окно многодокументной среды, **lpParam** должен указывать на структуру **CLIENTCREATESTRUCT**.

Возвращаемые значения

Если функция успешно завершила работу, возвращаемое значение – дескриптор для созданного окна. Если функция потерпела неудачу, возвращаемое значение – NULL.

После успешного создания окна вызываются функции включения отображения окна **ShowWindow** и обновления окна **UpdateWindow**. Затем происходит вход в цикл обработки сообщений. Сообщения обрабатываются в функции **WIN32PROC**, которую предварительно указали в классе окна.

Задание:
1. Ознакомиться со средой программирования Visual C++ 6.0.
2. Разработать программу создающую окно с иконкой средствами Win32 API.

Контрольные вопросы:
1. Каково назначение структуры WNDCLASS?
2. Какие операции производит функция CreateWindowEx?
3. Зачем нужен цикл по приёму сообщений?
4. Что происходит в функции обработки сообщений?

Для самостоятельного изучения:
1. Различные способы построения циклов обработки сообщений.

4.2 Динамическое создание пунктов меню и создание формы окна средствами Win32API

Те, кто программировал на Borland C++ Builder или Delphi знают, как легко там создавать меню (изменять его) и внешний вид главного окна, перетаскивая туда кнопки, списки и другие элементы. Однако, средствами Win32 API это делается немного по-другому.

Создать меню статически просто – это можно сделать по аналогии с присоединением иконки к окну. Но как вы будете менять элементы окна, добавлять и удалять пункты меню? Для этого надо знать, как работать с меню динамически. Основные функции для создания меню перечислены ниже.

Функция CreateMenu

```
HMENU CreateMenu (VOID);
```

Функция **CreateMenu** создает меню. Изначально меню пустое, но оно может быть заполнено пунктами меню, используя функции **InsertMenuItem**, **AppendMenu** и **InsertMenu**. В случае успеха возвращается дескриптор созданного меню. В случае неудачи возвращается NULL.

Функция AppendMenu

```
BOOL AppendMenu
(
    HMENU hMenu,          // дескриптор меню, которое будет изменено
    UINT uFlags,          // флажки пункта меню
    UINT uIDNewItem,      // идентификатор пункта меню или дескриптор
                          // «выскакивающего» меню
                          // или подменю
    LPCTSTR lpNewItem     // пункт контекстного меню
);
```

Функция **AppendMenu** добавляет в конец определяемой строки меню, «выскакивающего» меню, подменю или контекстного меню новый пункт. Вы можете использовать эту функцию, чтобы определить содержание, внешний вид и характеристики пункта меню.

Параметры:

hMenu – идентифицирует строку меню, «выскакивающее» меню, подменю или контекстное меню, которое будет изменено.

uFlags – определяет флажки, которые управляют внешним видом и характеристиками нового пункта меню. Этот параметр может быть комбинация значений.

uIDNewItem – определяет или идентификатор нового пункта меню или, если параметр **uFlags** установлен в MF_POPUP, дескриптор «выскакивающего» меню или подменю.

lpNewItem – определяет содержание нового пункта меню. Интерпретация **lpNewItem** зависит от того, включает ли параметр **uFlags** в себя флажок MF_BITMAP, MF_OWNERDRAW или MF_STRING, как указано ниже:

MF_BITMAP – Содержит дескриптор растрового рисунка.

MF_OWNERDRAW – Содержит 32-разрядное значение, предоставленное прикладной программой, которое может быть использовано, чтобы утвердить, что дополнительные данные касаются пункта меню. Значение является членом **itemData** структуры, указываемой параметром **lParam** при помощи передачи сообщения WM_MEASURE или WM_DRAWITEM, когда создается меню, или его внешний вид модифицируется.

MF_STRING – Содержит указатель на строку с символом нуля в конце.

Возвращаемые значения

Если функция завершается успешно, величина возвращаемого значения – не ноль. Если функция не выполняет задачу, величина возвращаемого значения – ноль.

Чтобы получать расширенные данные об ошибках, вызовите **GetLastError**.

Функция CreatePopupMenu

```
HMENU CreatePopupMenu (VOID);
```

Функция **CreatePopupMenu** создает выпадающее меню (drop-down menu), подменю (submenu) или меню быстрого вызова (shortcut menu). Меню изначально пустое. Вы можете вставить или добавить пункты меню, используя функцию **InsertMenuItem**. Вы также можете использовать функцию **InsertMenu** для вставки пунктов меню и функцию **AppendMenu** для добавления пунктов меню. В случае успеха возвращается дескриптор созданного меню. В случае неудачи возвращается NULL.

Функция SetMenuItemInfo

```
BOOL WINAPI SetMenuItemInfo
(
    HMENU hMenu,
    UINT uItem,
    BOOL fByPosition,
    LPMENUITEMINFO lpmii
);
```

Функция **SetMenuItemInfo** изменяет информацию о пункте меню.

Параметры:

hMenu – дескриптор меню, которое содержит пункт меню.

uItem – идентификатор или позиция пункта меню, который измениться. Предназначение этого параметра зависит от значения **fByPosition**.

fByPosition – значение, определяющее предназначение **uItem**. Если этот параметр – FALSE, то **uItem** – идентификатор пункта меню. Иначе, он - позиция пункта меню.

lpmii – указатель на структуру MENUITEMINFO, которая содержит информацию о пункте меню и определяет, какие атрибуты пункта меню изменятся.

Возвращаемые значения

Если функция завершается успешно, величина возвращаемого значения – не ноль. Если функция не выполняет задачу, величина возвращаемого значения – ноль. Чтобы получать расширенные данные об ошибках, используйте функцию **GetLastError**.

Функция SetMenu

```
BOOL SetMenu
(
    HWND hWnd,        // дескриптор окна
    HMENU hMenu       // дескриптор меню
);
```

Функция **SetMenu** связывает новое меню с заданным окном.

Параметры:

hWnd – идентифицирует окно, с которым должно быть связано меню.

hMenu – идентифицирует новое меню. Если этот параметр имеет значение NULL, текущее меню окна удаляется.

Возвращаемые значения

Если функция завершается успешно, величина возвращаемого значения – не ноль. Если функция не выполняет задачу, величина возвращаемого значения – ноль. Чтобы получать расширенные данные об ошибках, вызовите **GetLastError**.

В листинге 5 представлен участок программного кода динамического создания меню.

```
//Создание меню
int i;
HMenu=CreateMenu();
if (HMenu==NULL) return 1;

//Добавить "File"
i=AppendMenu(HMenu,MF_POPUP,M_FILE,"File");
if (i==0) return 1;

//Создать подменю "File"
HMENU h;
h=CreatePopupMenu();
i=AppendMenu(h,MFT_STRING,M_FILE1,"Новый");
if (i==0) return 1;
i=AppendMenu(h,MFT_STRING,M_FILE2,"Выход");
if (i==0) return 1;

//Добавить к меню
MENUITEMINFO mi;
mi.cbSize=sizeof(mi);
mi.fMask=MIIM_SUBMENU;
mi.hSubMenu=h;
i=SetMenuItemInfo(HMenu,M_FILE,false,&mi);
if (i==0) return 1;

//Добавление меню к окну
i=SetMenu(hWnd,HMenu);
if (i==0) return 1;
```

Листинг 5 – Динамическое добавление меню к окну

Создание меню должно происходить во время обработки сообщения WM_CREATE. В программе **hWnd** – это дескриптор окна (HWND). M_FILE, M_FILE1, M_FILE2 и **HMenu** должны быть объявлены ранее, например как показано в листинге 6.

```
#define M_FILE        10001
```

```
#define M_FILE1      10002
#define M_FILE2      10003
HMENU HMenu=NULL;
```

Листинг 6 – Объявление определений и переменных

Меню добавлено, однако надо получать сигналы о выборах пунктов меню. Для этого в функции обработки сообщений вы должны отлавливать сообщение WM_COMMAND и просматривать параметр **wParam** (листинг 7).

```
int wID = LOWORD(wParam);

switch (wID)
{
case M_FILE2:
        DestroyWindow(hWnd);
        break;
case M_FILE1:
        break;
}
```

Листинг 7 – Обработка выбора пункта меню

Теперь перейдём к созданию формы. Для этого надо перейти к редактору ресурсов и добавить диалоговое окно. Нажмите правой кнопкой на диалоговом окне и выберите «Properties» (Рисунок 7).

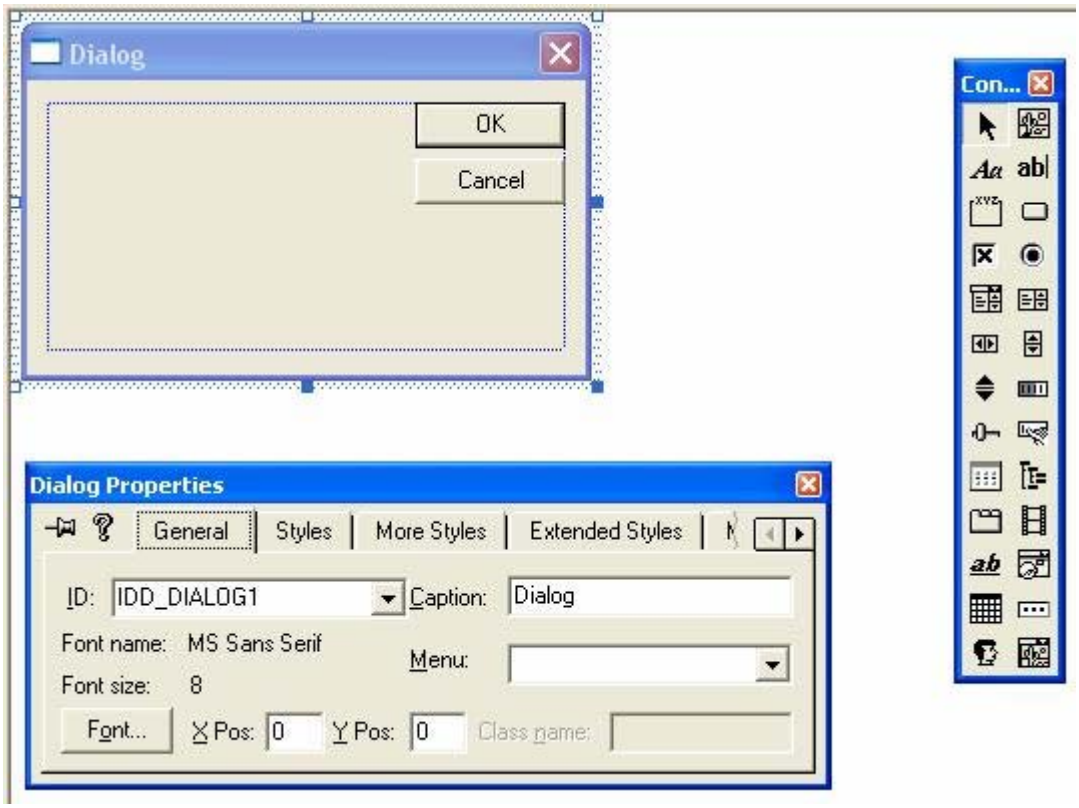

Рисунок 7 – Изменение свойств диалогового окна

В программе нужно, чтобы это диалоговое окно работало, как форма (Builder

или Delphi). Поэтому переходим во вкладку «Styles» и меняем свойства: в пункте «Style» на «Child», в пункте «Border» на «None». Не забываем ставить флажок в More Stules→Visible.

Автоматически созданное содержимое диалогового окна удаляем и переносим туда компонент «Edit Box». Открываем его свойства и помечаем галочкой следующие пункты: Visible, Tab stop, Multiline, Horizontal scroll, Auto HScroll, Vertical scroll, Auto VScroll, No hide selection, Border, Read-only, Static edge. А также растягиваем «Edit Box» до размеров окна.

Заготовка готова, теперь надо её натянуть на главное окно. Первоначально изменяем класс окна:

```
WndClass.hbrBackground=(HBRUSH)NULL;
```

А затем также при обработке сообщения WM_CREATE создаём диалоговое окно (Листинг 8).

```
hEdit=CreateDialog(hInst,MAKEINTRESOURCE(IDD_DIALOG1),hWnd,(DLGPROC) WIN32DIAL);
if (hEdit==NULL)
{
	return 1;
}
```

Листинг 8 – Создание немодального диалогового окна

Перед вызовом **hEdit** должен быть у вас объявлен как **HWND**, а также должна существовать функция обработки сообщения диалогового окна (Листинг 9).

```
LRESULT CALLBACK WIN32DIAL(HWND hWnd,UINT Message,
		UINT wParam,LONG lParam)
{
	return DefWindowProc(hWnd,Message,wParam,lParam);
}
```

Листинг 9 – Функция обработки сообщений диалогового окна

Компилируем – картинка не очень хорошая, т.к. диалоговое окно не растягивается самостоятельно. Для этого нужно написать свою функцию (Листинг 10).

```
int SizeEdit(HWND hWnd)
{
	RECT Rect;
	//получаем размер клиентской области окна
	GetClientRect(hWnd,&Rect);
	//меняем размеры окон
	SetWindowPos(hEdit,hWnd,0,0,Rect.right-Rect.left,Rect.bottom-Rect.top,SWP_NOZORDER);
	HWND h=GetDlgItem(hEdit,IDC_EDIT1);
	GetClientRect(hEdit,&Rect);
	SetWindowPos(h,hEdit,0,0,Rect.right-Rect.left,Rect.bottom-Rect.top,SWP_NOZORDER);
	return 0;
}
```

Листинг 10 – Функция установки размеров

Добавьте вызов этой функции в двух местах – непосредственно после создания диалогового окна и во время обработке сообщения WM_SIZE главного окна. Теперь другое дело. Ваша простейшая форма готова. Простейший пример добавления текста в Edit Box:

```
char new_str[3]={ 0x0d, 0x0a,0 };
char buf[256];
lstrcpy(buf,"Привет!");
lstrcat(buf,new_str);
lstrcat(buf,"Это моя первая программа!");
SetDlgItemText(hEdit,IDC_EDIT1,buf);
```

Листинг 11 – Добавление текста

Задание:
1. Ознакомиться с динамическим способом добавления меню.
2. Ознакомиться с элементами создания формы рабочего окна.
3. Разработать программу, в которой есть несколько пунктов меню и форма с «Edit Box». Названия пунктов меню должны загружаться из файла. По нажатию пунктов должен появляться и удаляться текст с формы.

Контрольные вопросы:
1. Чем отличается статическое и динамическое присоединение меню к окну?
2. Какие операции производит функция AppendMenu?
3. Как обрабатывается выбор пункта меню?
4. Чем отличается модальное окно от немодального?
5. Объясните назначения свойств диалогового окна.
6. Каким образом осуществляется изменение размеров формы?

Для самостоятельного изучения:
1. Функции работы с меню InsertMenuItem и InsertMenu.
2. Статистический способ добавления меню.
3. Удаление пунктов меню.
4. Использование различных элементов диалогового окна на форме.

4.3 Получение сведений о компьютере в операционной среде Windows программными средствами

Сведения об устройствах компьютера нужны в ряде случаев, когда параметры важны для выполнения программы. Сведения об устройствах компьютера в Windows 2000, ME, XP хранятся в реестре. Сведения о BIOS материнской платы и видеокарте хранятся в реестре в ключе (Табл. 2):
HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\System.

Сведения о процессоре содержаться в реестре в ключе:
HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\System\CentralProcessor\0.

Параметры: **Identifier**, **ProcessorNameString** – соответственно сведения об идентификаторе и имени процессора. Тип этих параметров можно посмотреть в

реестре, используя приложение **regedit**. Для доступа к ключам реестра из программы можно воспользоваться функциями **RegOpenKeyEx**, **RegQueryValueEx** и **RegCloseKey**.

Таблица 2 – Сведения реестра о BIOS и видеокарте

Параметр	Тип	Описание
SystemBiosDate	Строка	Дата Bios
SystemBiosVersion	Несколько строк	Версия Bios
VideoBiosDate	Строка	Дата видеокарты

Функция RegOpenKeyEx

```
LONG RegOpenKeyEx
(
        HKEY hKey,
        LPCTSTR lpSubKey,
        DWORD ulOptions,
        REGSAM samDesired,
        PHKEY phkResult
);
```

Функция открывает раздел реестра.

Параметры:

hKey – описатель открываемого раздела, который может быть получен функциями **RegCreateKeyEx** и **RegOpenKey**. Действуют стандартные описатели:

- HKEY_CLASSES_ROOT
- HKEY_CURRENT_CONFIG
- HKEY_CURRENT_USER
- HKEY_LOCAL_MACHINE
- HKEY_USERS
- Для Windows Me/98/95 также: HKEY_DYN_DATA.

lpSubKey – указатель на строку, завершающуюся нулевым байтом, которая содержит имя открываемого раздела. Этот раздел должен быть подразделом, идентифицируемого описателем раздела. Если этот параметр NULL, то функция вернет описатель самого раздела, т. е. раздела, идентифицируемого описателем.

ulOptions – зарезервировано и равно нулю.

samDesired – определяет права доступа (действия, которые будет проделывать с разделом программист). Как уже упоминалось, раздел реестра является системным объектом, а следовательно он имеет дескриптор защиты, именно в нем перечисляются права пользователей на объект. Определены следующие стандартные макросы:

- KEY_ALL_ACCESS – разрешаются любые действия над разделом;
- KEY_ENUMERATE_SUB_KEYS – разрешается перечисление подразделов данного раздела;
- KEY_READ – разрешается чтение раздела;
- KEY_SET_VALUE – разрешается создавать, удалять параметр или

устанавливать его значение;

- KEY_QUERY_VALUE – разрешается запрос параметра раздела.

phkResult – указатель на переменную, получающую описатель открытого раздела.

Если открытие произошло успешно, функция вернет ERROR_SUCCESS, в противном случае вернет ненулевой код ошибки, определенный в Winerror.h

Функция RegQueryValueEx

```
LONG RegQueryValueEx
(
        HKEY hKey,
        LPCTSTR lpValueName,
        LPDWORD lpReserved,
        LPDWORD lpType,
        LPBYTE lpData,
        LPDWORD lpcbData
);
```

Функция возвращает информацию о параметре раздела и значение этого параметра.

Параметры:

hKey – описатель открытого раздела. Раздел должен быть открыт с правами KEY_QUERY_VALUE.

lpValueName – указатель на строку, содержащую название параметра, о котором получается информация. Если параметр – NULL или пустая строка, то возвращается информация о параметре по умолчанию.

lpReserved – зарезервирован и равен NULL.

lpType – указатель на переменную, которая получает тип данных, сохраненных в параметре. Если равен NULL, то соответственно, информация не возвращается.

lpData – указатель на массив, получающий данные параметра. Если параметр – NULL, то данные не возвращаются. Если данные – это строка, то функция проверяет наличие нулевого символа.

lpcbData – указатель на переменную, которая определяет размер буфера, принимающего данные из параметра, в байтах. После того, как функция вернет значение, эта переменная будет содержать размер данных, скопированных в буфер. Если данные носят текстовый характер (REG_xxx_SZ), то также включается и нулевой символ (нулевые символы для REG_MULTI_SZ). Если размер буфера, недостаточен для сохранения данных, то функция вернет ERROR_MORE_DATA и сохранит требуемый размер буфера в переменную, на которую указывает этот параметр. Если **lpData** – NULL, а параметр **lpcbData** не нулевой, функция возвращает ERROR_SUCCESS и сохраняет размер данных в переменной, на которую указывает **lpcbData**.

Если функция выполнена успешно, возвращается ERROR_SUCCESS, в противном случае возвращается ненулевой код ошибки, определенный в Winerror.h

Функция RegCloseKey

```
LONG RegCloseKey(HKEY hKey);
```

Функция закрывает описатель раздела реестра.

Параметры:

hKey – описатель открытого раздела, который подлежит закрытию.

Если описатель успешно освобожден, функция возвращает ERROR_SUCCESS, в противном случае вернет ненулевой код ошибки, определенный в Winerror.h

Пример использования описанных функций для получения информации о дате BIOS представлен в листинге 12.

```
HKEY  hKeyResult = 0;
DWORD dwType;
DWORD dwBytes=256;
char buf[256];

LONG lResult = RegOpenKeyEx( HKEY_LOCAL_MACHINE,
"HARDWARE\\DESCRIPTION\\System" ,0, KEY_ALL_ACCESS, &hKeyResult );

lResult=RegQueryValueEx( hKeyResult, "SystemBiosDate", 0, &dwType,(BYTE*)buf, &dwBytes );

RegCloseKey(hKeyResult);
```

Листинг 12 – Получение информации из ключа реестра

При типе «несколько строк» в буфер **buf** возвращается массив строк, конец которого определяется двумя нулевыми символами.

Информацию о памяти и её текущем состоянии можно получить с помощью функции (winbase.h):

Функция GlobalMemoryStatus

```
void GlobalMemoryStatus(LPMEMORYSTATUS lpBuffer);
```

После вызова функции информацией о состоянии памяти заполняется структура **MEMORYSTATUS.**

Структура MEMORYSTATUS

```
typedef struct _MEMORYSTATUS
{
        DWORD dwLength;         // размер структуры
        DWORD dwMemoryLoad;     // процент занятой памяти (0-100)
        SIZE_T dwTotalPhys;     // объём физической памяти в байтах
        SIZE_T dwAvailPhys;     // свободный объём физической памяти в байтах
        SIZE_T dwTotalPageFile; // объём в байтах файла подкачки
        SIZE_T dwAvailPageFile; // свободный объём файла подкачки
```

```
    SIZE_T dwTotalVirtual;      // объём в байтах текущего адресного пространства
    SIZE_T dwAvailVirtual;      // свободный объём в байтах адресного пространства
} MEMORYSTATUS, *LPMEMORYSTATUS;
```

Получить имя компьютера в сети и имя пользователя можно с помощью описанных ниже функций.

Функция GetComputerName

```
BOOL GetComputerName
(
    LPTSTR lpBuffer,            // указатель на буфер
    LPDWORD lpnSize             // указатель на размер буфера
);
```

Параметр **lpBuffer** указывает буфер, в который будет записано имя компьютера. **lpnSize** – максимальное количество символов. Это значение должно быть не менее MAX_COMPUTERNAME_LENGTH+1. Если функция успешно выполнена, она возвращает true.

Функция GetUserName

```
BOOL GetUserName
(
    LPTSTR lpBuffer,
    LPDWORD nSize
);
```

Параметр **lpBuffer** указывает буфер, в который будет записано имя пользователя. **nSize** – максимальное количество символов.

Задание:

1. Написать функцию, которая будет выводить информацию:
   - дата Bios;
   - версия Bios;
   - дата видеокарты;
   - идентификатор процессора;
   - производитель процессора;
   - объём физической памяти;
   - процент занятой памяти;
   - объём файла подкачки;
   - свободный объём файла подкачки;
   - адресное пространство;
   - имя компьютера;
   - имя пользователя.

Контрольные вопросы:

1. Для чего предназначен реестр?
2. Какие основные разделы реестра?
3. Каким образом можно работать со списком строк, получаемых из реестра?

Для самостоятельного изучения:
1. Функции работы с реестром RegOpenKey, RegCreateKeyEx, RegQueryInfoKey, RegEnumKeyEx, RegEnumValue, RegSetValueEx.

4.4 Взаимодействие приложения с System Tray

Область System Tray представляет собой часть полосы задач, размещается в правом нижнем углу и содержит значки, такие как часы и т.п. Эту область активно используют программы, работающие в процессе всего сеанса Windows. Обычно при этом программы делают себя невидимыми в полосе задач. Для этого при создании необходимо вызвать функцию (Handle – указатель на ваше окно):

Функция SetWindowLong

```
LONG SetWindowLong
(
        HWND hWnd,              // дескриптор окна
        int nIndex,             // значение смещения, которое устанавливается
        LONG dwNewLong          //новое значение
);
```

Функция SetWindowLong изменяет атрибуты определяемого окна. Функция также устанавливает 32-разрядное (длинное) значение при заданном смещении в дополнительном пространстве памяти об окне.

Параметры:
hWnd – идентифицирует окно и, косвенно, класс, которому окно принадлежит.
nIndex – определяет значение смещения, отсчитываемое от нуля, которое будет установлено. Допустимые значения находятся в диапазоне от нуля до числа байтов дополнительного пространства в памяти, минус 4; например, если бы Вы установили 12 или большее количество байтов памяти дополнительного пространства, значение 8 было бы индексом к третьему 32-разрядному целому числу. Чтобы установить любое другое значение, определите одно из следующих значений:
GWL_EXSTYLE – Устанавливает новый расширенный стиль окна.
GWL_STYLE – Устанавливает новый стиль окна.
GWL_WNDPROC – Устанавливает новый адрес для оконной процедуры.
GWL_HINSTANCE – Устанавливает новый дескриптор экземпляра прикладной программы.
GWL_ID – Устанавливает новый идентификатор окна.
GWL_USERDATA – Устанавливает 32-разрядное значение, связанное с окном. Каждое окно имеет соответствующее 32-разрядное значение, предназначенное для использования прикладной программой, которая создала окно.

Следующие значения также доступны, когда параметр hWnd идентифицирует диалоговое окно:
DWL_DLGPROC – Устанавливает новый адрес процедуры диалогового окна.
DWL_MSGRESULT – Устанавливает возвращаемое значение сообщения, обработанного в процедуре диалогового окна.
DWL_USER – Устанавливает новую дополнительную информацию, которая является частной для прикладной программы, типа дескрипторов или указателей.
dwNewLong – устанавливает восстановленное значение.

Возвращаемые значения

Если функция завершается успешно, возвращаемое значение - предыдущее значение заданного 32-разрядного целого числа. Если функция не выполняет задачу, возвращаемое значение нулевое. Чтобы получать расширенные данные об ошибках, вызовите **GetLastError**.

Функция Shell_NotifyIcon

```
WINSHELLAPI BOOL WINAPI Shell_NotifyIcon
(
	DWORD dwMessage,
	PNOTIFYICONDATA pnid
);
```

Используется для добавления и удаления иконки в System Tray.

Параметры:
dwMessage – должен содержать одно из следующих значений:
NIM_ADD - добавить значок в область состояния,
NIM_DEL - удалить значок из области состояния,
NIM_MODIFY - изменить значок в области состояния.
pnid – указывает на структуру типа NOTIFYICONDATA, значения полей которой зависят от параметра dwMessage.

Функция **Shell_NotifyIcon** возвращает ненулевое значение, если операция прошла успешно, и ноль в случае ошибки.

Структура NOTIFYICONDATA

```
typedef struct _NOTIFYICONDATA
{
	DWORD cbSize;
	HWND hWnd;
	UINT uID;
	UINT uFlags;
	UINT uCallbackMessage;
	HICON hIcon;
	WCHAR szTip[64];
} NOTIFYICONDATA, *PNOTIFYICONDATA;
```

И ее поля имеют следующий смысл:

cbSize – размер структуры NOTIFYICONDATA.
hWnd – манипулятор окна, которое будет получать сообщения от значка в области состояния.
uID – идентификатор значка. Это значение передается приложению в качестве первого параметра (WPARAM) сообщения от значка.
uFlags – набор флагов, которые определяют, какие поля структуры заданы корректно. Могут использоваться следующие значения или их комбинации с помощью логического «ИЛИ»: NIF_ICON – поле **hIcon** корректно, NIF_MESSAGE – поле **uCallbackMessage** корректно, NIF_TIP – поле **szTip** корректно.
uCallbackMessage – идентификатор сообщения, посылаемого окну **hWnd** при возникновении события «мыши» над значком в области состояния. Можно использовать значения WM_USER+***N***, где ***N*** – неотрицательное число;
hIcon – манипулятор иконки, которую нужно разместить (изменить, удалить) в System Tray.
szTip – ASCIIZ-строка, которая будет использоваться в качестве «всплывающего» текста, когда указатель «мыши» остановится над значком. Если текст отсутствует, первый байт строки должен быть нулевым.

Перед вызовом функции **Shell_NotifyIcon** нужно подготовить экземпляр структуры NOTIFYICONDATA. Поля **cbSize**, **hWnd** и **uID** нужно заполнять всегда, остальные – по мере необходимости. В соответствии с заполнением полей **uCallbackMessage**, **hIcon** и **szTip** формируется поле флагов **uFlags**.

Чтобы добавить значок в область состояния, нужно вызвать функцию **Shell_NotifyIcon**, передав ей в качестве параметра **dwMessage** значение NIM_ADD, а в качестве **pnid** – указатель на инициализированный экземпляр структуры NOTIFYICONDATA. Если все выполнено правильно, функция вернет ненулевое значение, а в System Tray появится новая иконка. Если планируется, что окно должно принимать сообщения от значка, следует обратить внимание, чтобы поле **hWnd** перед вызовом **Shell_NotifyIcon** было инициализировано значением манипулятора реально существующего окна. В противном случае значок будет исчезать из области состояния, как только над ним остановится указатель «мыши» Если было инициализировано поле **uCallbackMessage**, система будет посылать окну hWnd сообщения о событиях «мыши» над значком. При этом параметр сообщения WPARAM будет содержать идентификатор значка **uID**, а параметр LPARAM – тип сообщения.

Приложение, разместившее значок в System Tray, может в любой момент изменить иконку или всплывающую подсказку. Для этого нужно внести изменения в соответствующие поля структуры NOTIFYICONDATA, поправить значение **uFlags** (значения **cbSize**, **hWnd** и **uId** изменяться не должны) и вызвать функцию **Shell_NotifyIcon** со значением NIM_MODIFY в качестве параметра **dwMessage**.

Для удаления значка из System Tray достаточно правильно заполнить поля **cbSize**, **hWnd**, **uId** и вызвать функцию **Shell_NotifyIcon** со значением параметра **dwMessage** равным NIM_DELETE. Пример добавления иконки к окну представлен в листинге 13.

```
const TrayIcon = WM_USER + 1;

NOTIFYICONDATA NID;
NID.cbSize = sizeof(NOTIFYICONDATA );
NID.hWnd = hWnd;
NID.uID = 1;
NID.uFlags = NIF_ICON | NIF_MESSAGE | NIF_TIP;
NID.uCallbackMessage = TrayIcon; //указатель на создаваемое событие от иконки
NID.hIcon = LoadIcon(hInst,MAKEINTRESOURCE(IDI_ICON1));
strcpy(NID.szTip,"Имя приложения");
Shell_NotifyIcon(NIM_ADD,&NID);
```

Листинг 13 – Добавление иконки программы в System Tray

Теперь необходимо написать обработку сообщения TrayIcon в функции обработки сообщений главного окна. Так в листинге 14 показано как обрабатывается сообщение о нажатии на иконку и разворачивании окна после этого.

```
// Часть функции обработки сообщений
case TrayIcon:
        OnTray(hWnd,wParam,lParam);
        break;

// Функция обработки сообщений от иконки
int OnTray(HWND hWnd,UINT wParam,LONG lParam)
{
        switch(lParam)
        {
        case 514://WM_LBUTTONDBLCLK:
                ShowWindow(hWnd,SW_SHOWNORMAL);
                SetForegroundWindow(hWnd);
                break;
        }
        return 0;
}
```

Листинг 14 – Обработка нажатия на иконку в System Tray

Задание:

1. Модифицировать программу для работы с System Tray:
   – удаление собственной иконки из System Tray;
   – добавление иконки в System Tray;
   – добавление меню к иконке;
   – обработка нажатия левой и правой кнопок на меню;
   – обработка выбора пункта меню.
2. Удалить программу из Панели задач.

Контрольные вопросы:

1. Для чего предназначен System Tray?
2. Для чего нужно сообщение WM_USER?
3. Какие операции по работе с иконками существуют?

4. Как удалить программу из Панели задач?
5. Как обработать нажатие правой кнопки мыши на иконке?

Для самостоятельного изучения:
1. Удаление, добавление и модификация иконок в System Tray.
2. Присоединение меню к System Tray.

4.5 Создание и управление процессами

Процессом обычно называют экземпляр выполняемой программы.

Хотя на первый взгляд кажется что программа и процесс понятия практически одинаковые, они фундаментально отличаются друг от друга. Программа представляет собой статический набор команд, а процесс это набор ресурсов и данных, использующихся при выполнении программы. Процесс в Windows состоит из следующих компонентов:

– структура данных, содержащая всю информацию о процессе, в том числе список открытых дескрипторов различных системных ресурсов, уникальный идентификатор процесса, различную статистическую информацию и т.д.;

– адресное пространство – диапазон адресов виртуальной памяти, которым может пользоваться процесс;

– исполняемая программа и данные, проецируемые на виртуальное адресное пространство процесса.

Поток (thread) –сущность внутри процесса, получающая процессорное время для выполнения. В каждом процессе есть минимум один поток. Этот первичный поток создается системой автоматически при создании процесса. Далее этот поток может породить другие потоки, те в свою очередь новые и т.д. Таким образом, один процесс может владеть несколькими потоками, и тогда они одновременно исполняют код в адресном пространстве процесса. Каждый поток имеет:

– уникальный идентификатор потока;

– содержимое набора регистров процессора, отражающих состояние процессора;

– два стека, один из которых используется потоком при выполнении в режиме ядра, а другой – в пользовательском режиме;

– закрытую область памяти, называемую локальной памятью потока (thread local storage, TLS) и используемую подсистемами, run-time библиотеками и DLL.

Чтобы все потоки работали, операционная система отводит каждому из них определенное процессорное время. Тем самым создается иллюзия одновременного выполнения потоков (разумеется, для многопроцессорных компьютеров возможен истинный параллелизм). В Windows реализована система вытесняющего планирования на основе приоритетов, в которой всегда выполняется поток с наибольшим приоритетом, готовый к выполнению. Выбранный для выполнения поток работает в течение некоторого периода, называемого квантом. Квант определяет, сколько времени будет выполняться поток, пока операционная система не прервет его. По окончании кванта операционная система проверяет, готов ли к

выполнению другой поток с таким же (или большим) уровнем приоритета. Если таких потоков не оказалось, текущему потоку выделяется еще один квант. Однако поток может не полностью использовать свой квант. Как только другой поток с более высоким приоритетом готов к выполнению, текущий поток вытесняется, даже если его квант еще не истек

Для создания процессов используется функция **CreateProcess** – это основная функция запуска процесса, все остальные функции такие как **WinExec** и LoadModule оставлены для совместимости и используют **CreateProcess**.

Функция CreateProcess

```
BOOL CreateProcess
(
LPCTSTR lpApplicationName,                  // имя исполняемого модуля
LPTSTR lpCommandLine,                       // Командная строка
LPSECURITY_ATTRIBUTES lpProcessAttributes,  // Указатель на структуру
                                            // SECURITY_ATTRIBUTES
LPSECURITY_ATTRIBUTES lpThreadAttributes,   // Указатель на структуру
                                            // SECURITY_ATTRIBUTES
BOOL bInheritHandles,                       // Флаг наследования текущего процесса
DWORD dwCreationFlags,                      // Флаги способов создания процесса
LPVOID lpEnvironment,                       // Указатель на блок среды
LPCTSTR lpCurrentDirectory,                 // Текущий диск или каталог
LPSTARTUPINFO lpStartupInfo,                // Указатель нас структуру STARTUPINFO
LPPROCESS_INFORMATION lpProcessInformation  // Указатель нас структуру
                                            // PROCESS_INFORMATION
);
```

Параметры:

pApplicationName – указатель на строку которая заканчивается нулем и содержит имя выполняемого модуля. Этот параметр может быть NULL тогда имя модуля должно быть в **lpCommandLine** самым первым элементом. Если операционная система NT и модуль 16 разрядов этот параметр NULL обязательно. имя модуля может быть абсолютным или относительным. Если относительное то будет использована информация из **lpCurrentDirectory** или текущий каталог.

lpCommandLine – командная строка. Здесь передаются параметры. Она может быть NULL. Здесь можно указать и путь и имя модуля.

lpProcessAttributes – здесь определяются атрибуты защиты для нового приложения. Если указать NULL то система сделает это по умолчанию.

lpThreadAttributes – здесь определяются атрибуты защиты для первого потока созданного приложением. NULL опять приводит к установке по умолчанию.

bInheritHandles – флаг наследования от процесса производящего запуск. Здесь наследуются дескрипторы. Унаследованные дескрипторы имеют те же значения и права доступа, что и оригиналы.

dwCreationFlags. Флаг способа создание процесса и его приоритет:

CREATE_DEFAULT_ERROR_MODE – Новый процесс не наследует режим ошибок (error mode) вызывающего процесса.

CREATE_NEW_CONSOLE – Новый процесс получает новую консоль вместо того, чтобы унаследовать родительскую.
CREATE_NEW_PROCESS_GROUP – Создаваемый процесс - корневой процесс новой группы.
CREATE_SEPARATE_WOW_VDM – Только Windows NT: Если этот флаг установлен, новый процесс запускается в собственной Virtual DOS Machine (VDM).
CREATE_SHARED_WOW_VDM – Только Windows NT: Этот флаг указывает функции CreateProcess запустит новый процесс в разделяемой Virtual DOS Machine.
CREATE_SUSPENDED – Первичная нить процесса создается в спящем (suspended) состоянии и не выполняется до вызова функции **ResumeThread**.
CREATE_UNICODE_ENVIRONMENT – Если этот флаг установлен, блок переменных окружения, указанный в параметре **lpEnvironment**, использует кодировку Unicode. Иначе – кодировку ANSI.
DEBUG_PROCESS – Если этот флаг установлен, вызывающий процесс считается отладчиком, а новый процесс - отлаживаемым.
DEBUG_ONLY_THIS_PROCESS – Если этот флаг не установлен и вызывающий процесс находится под отладкой, новый процесс так же становится отлаживаемым тем же отладчиком.
DETACHED_PROCESS – Создаваемый процесс не имеет доступа к родительской консоли. Этот флаг нельзя использовать с флагом CREATE_NEW_CONSOLE.
HIGH_PRIORITY_CLASS – Указывает на то, что процесс выполняет критичные по времени задачи .
IDLE_PRIORITY_CLASS – Указывает процесс, выполняются только когда система находится в состоянии ожидания.
NORMAL_PRIORITY_CLASS – Указывает на процесс, без каких либо специальных требований к выполнению.
REALTIME_PRIORITY_CLASS – Указывает процесс имеющий наивысший возможный приоритет.
lpEnvironment – указывает на блок среды. Если NULL, то будет использован блок среды родительского процесса. Блок среды это список переменных имя = значение в виде строк с нулевым окончанием.
lpCurrentDirectory – Указывает текущий диск и каталог. Если NULL, то будет использован диск и каталог процесса родителя.
lpStartupInfo – Указатель на структуру STARTUPINFO, которая определяет параметры главного окна порожденного процесса;
lpProcessInformation – Указатель на структуру PROCESSINFO, которая будет заполнена информацией о порожденном процессе после возврата из функции.

В результате выполнение функций вернет FALSE или TRUE. В случае успеха TRUE. Пример использования функции представлен в листинге 15.

```
STARTUPINFO cif;
ZeroMemory(&cif,sizeof(STARTUPINFO));
PROCESS_INFORMATION pi;
if (CreateProcess("c:\\windows\\notepad.exe",NULL,
        NULL,NULL,FALSE,NULL,NULL,NULL,&cif,&pi)==TRUE)
{
```

```
        cout << "process" << endl;
        cout << "handle " << pi.hProcess << endl;
        Sleep(1000);                            // подождать
        TerminateProcess(pi.hProcess,NO_ERROR);         // убрать процесс
}
```

Листинг 15 – Пример использования функции **CreateProcess**

Для удаления процесса используется функция:

Функция TerminateProcess

```
BOOL TerminateProcess
(
        HANDLE hProcess,   // Указатель процесса
        UINT uExitCode       // Код возврата процесса
);
```

Параметры:
hProcess – Дескриптор процесса, который завершает работу.
uExitCode – Код выхода, который использует процесс и потоки, чтобы завершить работу в результате этого вызова. Используйте функцию **GetExitCodeProcess**, чтобы извлечь значение выхода процесса. Используйте функцию **GetExitCodeThread**, чтобы извлечь значение выхода потока.

Возвращаемые значения

Если функция завершается успешно, величина возвращаемого значения – не ноль. Если функция завершается с ошибкой, величина возвращаемого значения – ноль. Чтобы получать расширенные данные об ошибках, вызовите **GetLastError**.

При удалении процесса, открытого не нашим приложением, необходимо получить информацию об окне и открыть процесс. Сначала необходимо найти окно с помощью функции:

Функция FindWindowEx

```
HWND FindWindowEx
(
        HWND hwndParent ,          // дескриптор родительского окна
        HWND hwndChildAfter ,      // дескриптор дочернего окна
        LPCTSTR lpszClass ,        // указатель на имя класса
        LPCTSTR lpszWindow         // указатель на имя окна
);
```

Функция **FindWindowEx** извлекает дескриптор окна, чье имя класса и имя окна совпадают с указанными строками. Функция также просматривает дочерние окна, начиная с того, дескриптор которого передан функции в качестве параметра **hwndChildAfter**.

Параметры:
hwndParent – идентифицирует родительское окно, среди дочерних окон которого

будет проводиться поиск. Если значение параметра **hwndParent** равно NULL, функция использует рабочий стол Windows в качестве родительского окна. Функция проводит поиск среди окон, являющимися дочерними окнами рабочего стола.
hwndChildAfter – идентифицирует дочернее окно. Поиск начинается со следующего окна в Z-последовательности. Окно, указанное параметром **hwndChildAfter**, должно быть прямым дочерним окном указанного параметром **hwndParent** окна, а не порожденным окном. Если значение параметра **hwndChildAfter** равно NULL, поиск начинается с первого дочернего окна.
lpszClass – указывает на завершающуюся нулем строку, определяющую имя класса или атом, идентифицирующий строку – имя класса. Если этот параметр является атомом, он должен быть глобальным атомом, созданным предыдущим вызовом функции **GlobalAddAtom**.
lpszWindow – указывает на завершающуюся нулем строку, определяющую имя окна (заголовок окна). Если значение этого параметра равно NULL, то совпадающими со строкой считаются все имена окон.

Возвращаемые значения

В случае успеха возвращается дескриптор окна, которое имеет заданные имя класса и имя окна. В случае неудачи возвращается NULL. Для получения дополнительной информации об ошибке вызовите функцию **GetLastError**.

Данная функция позволяет искать окно по названию класса окна или по названию титула окна. После получения HWND искомого приложения, необходимо получить номер (ID) процесса, например с помощью функции:

Функция GetWindowThreadProcessId

```
DWORD GetWindowThreadProcessId
(
    HWND hWnd,                // дескриптор окна
    LPDWORD lpdwProcessId   // адрес переменной для идентификатора процесса
);
```

Параметры:
hWnd – идентифицирует окно.
lpdwProcessId – указывает на 32-разрядное значение, которое принимает идентификатор процесса. Если этот параметр – не NULL, **GetWindowThreadProcessId** копирует идентификатор процесса в 32-разрядное значение; иначе, она этого не делает.

Возвращаемое значение – идентификатор потока, который создает окно.

Для того, чтобы завершить процесс, его необходимо открыть с использованием функции:

Функция OpenProcess

```
HANDLE OpenProcess
```

```
(
        DWORD dwDesiredAccess,          // флажок доступа
        BOOL bInheritHandle,            // параметр дескриптора наследования
        DWORD dwProcessId               // идентификатор процесса
);
```

Параметры:

dwDesiredAccess – устанавливает уровень доступа к объекту процесса. Этот параметр может состоять из одного нескольких прав доступа к процессу. Windows NT /200/XP: Это право доступа проверяется у любого дескриптора безопасности для процесса.

bInheritHandle – если этот параметр является TRUE, дескриптор наследуем. Если этот параметр является FALSE, дескриптор не может наследоваться.

dwProcessId – идентификатор процесса, который открыт.

Возвращаемые значения

Если функция завершается успешно, величина возвращаемого значения – открытый дескриптор заданного процесса. Если функция завершается с ошибкой, величина возвращаемого значения NULL. Чтобы получить дополнительные данные об ошибке, вызовите **GetLastError**.

Полезной функцией, необходимой для получения сведений о выполнении процессов, является:

Функция GetProcessTimes

```
BOOL GetProcessTimes
(
        HANDLE hProcess,                // дескриптор процесса
        LPFILETIME lpCreationTime,      // время создания процесса
        LPFILETIME lpExitTime,          // время выхода из работы процесса
        LPFILETIME lpKernelTime,        // время, работы процесса в режиме ядра
        LPFILETIME lpUserTime           // время, работы процесса в режиме пользователя
);
```

Параметры:

hProcess – дескриптор процесса, информация о распределении интервалов времени которого разыскивается. Этот дескриптор должен быть создан с правами доступа PROCESS_QUERY_INFORMATION. Для получения дополнительной информации, см. статью Защита процесса и права доступа.

lpCreationTime – указатель на структуру FILETIME, которая принимает время создания процесса. Так как время возвращается по количеству интервалов по 100нс., отсчитанных с полуночи 1 января 1601 года, то для получения нормального времени необходимо воспользоваться функцией **FileTimeToSystemTime**.

lpExitTime – указатель на структуру FILETIME, которая принимает время выхода из работы процесса. Если процесс не вышел из работы, содержание этой структуры не определенное.

lpKernelTime – указатель на структуру FILETIME, которая принимает величину

времени, в течение которого процесс выполнялся в привилегированном режиме (режиме ядра). Чтобы получить это значение, определяется время, в ходе которого каждый из потоков процесса выполнялся в режиме ядра, а затем все эти периоды суммируются вместе.

lpUserTime – указатель на структуру FILETIME, которая принимает величину времени, в течение которого процесс выполнялся в непривилегированном (пользовательском) режиме. Чтобы получить это значение, определяется время, в ходе которого каждый из потоков процесса выполнялся в режиме ядра, а затем все эти периоды суммируются вместе.

Возвращаемые значения

Если функция завершается успешно, величина возвращаемого значения – не ноль. Если функция завершается с ошибкой, величина возвращаемого значения – ноль. Чтобы получать расширенные данные об ошибках, вызовите **GetLastError**.

Задание:

1. Модифицировать программу:
   – запустить процесс калькулятора из приложения;
   – запустить из приложения процесс команды ping с параметрами запрашиваемого IP-адреса соседнего компьютера;
   – получить информацию о времени выполнения дочерних процессах калькулятора и ping;
   – удалить оба дочерних процесса;
   – запустив через меню «Пуск» программы калькулятора и графического редактора Paint, найти и удалить их с помощью вашего приложения.

Контрольные вопросы:

1. Из чего состоят процесс и поток в среде Windows?
2. Создание и завершение процесса.
3. Каковы параметры функции CreateProcess?
4. Назначение и состав структуры PROCESS_INFORMATION.
5. Назначение и состав структуры STARTUP_INFO.
6. Назначение и состав структуры LPSECURITY_ATTRIBUTES.

Для самостоятельного изучения:

1. Функции ExitProcess, GetCurrentProcess, GetPriorityClass, GetProcessVersion, GetProcessWorkingSetSize, SetPriorityClass, SetProcessWorkingSetSize.

4.6 Обмен информацией между процессами

Параллельно работающие процессы часто должны обмениваться данными. На данном этапе необходимо осуществить взаимодействие между двумя процессами через файл и путём использования API сообщения WM_COPYDATA.

Для того, чтобы два приложения обменивались данными через файл необходимо, чтобы один процесс записывал туда, к примеру, текстовую строку, а другой считывал. При этом, осуществлялся поочередный доступ к файлу – только

один процесс может работать с ним. Для работы с файлами средствами API предназначена функция:

Функция CreateFile

```
HANDLE CreateFile
(
    LPCTSTR  lpFileName,                        // имя файла
    DWORD  dwDesiredAccess,                     // режим доступа
    DWORD  dwShareMode,                         // совместный доступ
    LPSECURITY_ATTRIBUTES  lpSecurityAttributes,  // SD (дескр. защиты)
    DWORD  dwCreationDisposition,               // как действовать
    DWORD  dwFlagsAndAttributes,                // атрибуты файла
    HANDLE  hTemplateFile                       // дескр.шаблона файла
);
```

Функция **CreateFile** создает или открывает каталог, физический диск, том, буфер консоли (CONIN$ или CONOUT$), устройство на магнитной ленте, коммуникационный ресурс, почтовый слот или именованный канал. Функция возвращает дескриптор, который может быть использован для доступа к объекту.

Параметр **lpFileName** – адрес строки, содержащей имя файла, который вы собираетесь создать или открыть. Параметр **dwDesiredAccess** определяет тип доступа, который должен быть предоставлен к открываемому файлу.

С помощью параметра **dwShareMode** задаются режимы совместного использования открываемого или создаваемого файла.

Через параметр **lpSecurityAttributes** необходимо передать указатель на дескриптор защиты или значение NULL, если этот дескриптор не используется. Параметр **dwCreationDistribution** определяет действия, выполняемые функцией **CreateFile**, если приложение пытается создать файл, который уже существует.

Параметр **dwFlagsAndAttributes** задает атрибуты и флаги для файла.

И, наконец, последний параметр **hTemplateFile** предназначен для доступа к файлу шаблона с расширенными атрибутами создаваемого файла.

В случае успешного завершения функция CreateFile возвращает идентификатор созданного или открытого файла (или каталога).

При ошибке возвращается значение INVALID_HANDLE_VALUE (а не NULL, как можно было бы предположить). Код ошибки можно определить при помощи функции **GetLastError**.

Закрытие файла осуществляется с использование функции **CloseHandle**. Запись и чтение из файла осуществляется с помощью функций **ReadFile** и **WriteFile**:

Функция ReadFile

```
BOOL ReadFile
(
    HANDLE hFile,                   // дескриптор файла
    LPVOID lpBuffer,                // буфер данных
```

```
    DWORD nNumberOfBytesToRead,      // число байтов для чтения
    LPDWORD lpNumberOfBytesRead,     // число прочитанных байтов
    LPOVERLAPPED lpOverlapped        // асинхронный буфер
);
```

Функция **ReadFile** читает данные из файла, начиная с позиции, обозначенной указателем файла. После того, как операция чтения была закончена, указатель файла перемещается на число действительно прочитанных байтов, если дескриптор файла не создан с атрибутом асинхронной операции. Если дескриптор файла создается для асинхронного ввода - вывода, приложение должно переместить позицию указателя файла после операции чтения.

Параметры:

hFile – дескриптор файла, который читается. Дескриптор файла должен быть, создан с правом доступа GENERIC_READ.

lpBuffer – указатель на буфер, который принимает прочитанные данные из файла.

nNumberOfBytesToRead – число байтов, которые читаются из файла.

lpNumberOfBytesRead – указатель на переменную, которая получает число прочитанных байтов. Функция **ReadFile** устанавливает это значение в нуль перед началом любой работы или проверкой ошибок.

lpOverlapped – указатель на структуру OVERLAPPED. Эта структура требуется тогда, если параметр **hFile** создавался с флажком FILE_FLAG_OVERLAPPED.

Возвращаемые значения

Функция **ReadFile** возвращает значение тогда, когда выполнено одно из ниже перечисленных условий:

– операция записи завершается на записывающем конце канала,

– затребованное число байтов прочитано,

– или происходит ошибка.

Если функция завершается успешно, величина возвращаемого значения – не ноль. Если функция завершается с ошибкой, величина возвращаемого значения – ноль. Чтобы получить дополнительные сведения об ошибке, вызовите **GetLastError**.

Функция WriteFile

```
BOOL WriteFile
(
    HANDLE hFile,                    // дескриптор файла
    LPCVOID lpBuffer,                // буфер данных
    DWORD nNumberOfBytesToWrite,     // число байтов для записи
    LPDWORD lpNumberOfBytesWritten,  // число записанных байтов
    LPOVERLAPPED lpOverlapped        // асинхронный буфер
);
```

Функция **WriteFile** пишет данные в файл с места, обозначенного указателем позиции в файле. Эта функция предназначена и для синхронной, и для асинхронной операции.

Параметры:

hFile – дескриптор файла. Дескриптор файла, должен быть создан с правом доступа GENERIC_WRITE.
lpBuffer – указатель на буфер, содержащий данные, которые будут записаны в файл.
nNumberOfBytesToWrite – число байтов, которые будут записаны в файл.
lpNumberOfBytesWritten – указатель на переменную, которая получает число записанных байтов. Функция **WriteFile** устанавливает это значение в нуль перед выполнением какой-либо работы или выявлением ошибок.
lpOverlapped – указатель на структуру OVERLAPPED. Эта структура требуется тогда, если параметр **hFile** создавался с флажком FILE_FLAG_OVERLAPPED.

Возвращаемые значения

Если функция завершается успешно, величина возвращаемого значения – не ноль. Если функция завершается с ошибкой, величина возвращаемого значения – ноль. Чтобы получить дополнительные сведения об ошибке, вызовите **GetLastError**.

В листинге 16 приведён пример открытия файла для чтения.

```
void CreateBMPFile(HWND hwnd, LPTSTR pszFile, PBITMAPINFO pbi,
            HBITMAP hBMP, HDC hDC)
 {
    HANDLE hf;                 // file handle
   BITMAPFILEHEADER hdr;       // bitmap file-header
   PBITMAPINFOHEADER pbih;     // bitmap info-header
   LPBYTE lpBits;              // memory pointer
   DWORD dwTotal;              // total count of bytes
   DWORD cb;                   // incremental count of bytes
   BYTE *hp;                   // byte pointer
   DWORD dwTmp;

   pbih = (PBITMAPINFOHEADER) pbi;
   lpBits = (LPBYTE) GlobalAlloc(GMEM_FIXED, pbih->biSizeImage);

   if (!lpBits)
       errhandler("GlobalAlloc", hwnd);

   // Retrieve the color table (RGBQUAD array) and the bits
   // (array of palette indices) from the DIB.
   if (!GetDIBits(hDC, hBMP, 0, (WORD) pbih->biHeight, lpBits, pbi,
      DIB_RGB_COLORS))
   {
      errhandler("GetDIBits", hwnd);
   }

   // Create the .BMP file.
   hf = CreateFile(pszFile,
              GENERIC_READ | GENERIC_WRITE,
              (DWORD) 0,
               NULL,
              CREATE_ALWAYS,
```

```
            FILE_ATTRIBUTE_NORMAL,
            (HANDLE) NULL);
   if (hf == INVALID_HANDLE_VALUE)
      errhandler("CreateFile", hwnd);
   hdr.bfType = 0x4d42;        // 0x42 = "B" 0x4d = "M"
   // Compute the size of the entire file.
   hdr.bfSize = (DWORD) (sizeof(BITMAPFILEHEADER) +
            pbih->biSize + pbih->biClrUsed
            * sizeof(RGBQUAD) + pbih->biSizeImage);
   hdr.bfReserved1 = 0;
   hdr.bfReserved2 = 0;

   // Compute the offset to the array of color indices.
   hdr.bfOffBits = (DWORD) sizeof(BITMAPFILEHEADER) +
            pbih->biSize + pbih->biClrUsed
            * sizeof (RGBQUAD);

   // Copy the BITMAPFILEHEADER into the .BMP file.
   if (!WriteFile(hf, (LPVOID) &hdr, sizeof(BITMAPFILEHEADER),
      (LPDWORD) &dwTmp,  NULL))
   {
     errhandler("WriteFile", hwnd);
   }

   // Copy the BITMAPINFOHEADER and RGBQUAD array into the file.
   if (!WriteFile(hf, (LPVOID) pbih, sizeof(BITMAPINFOHEADER)
            + pbih->biClrUsed * sizeof (RGBQUAD),
            (LPDWORD) &dwTmp, ( NULL))
      errhandler("WriteFile", hwnd);

   // Copy the array of color indices into the .BMP file.
   dwTotal = cb = pbih->biSizeImage;
   hp = lpBits;
   if (!WriteFile(hf, (LPSTR) hp, (int) cb, (LPDWORD) &dwTmp,NULL))
        errhandler("WriteFile", hwnd);

   // Close the .BMP file.
    if (!CloseHandle(hf))
        errhandler("CloseHandle", hwnd);

   // Free memory.
   GlobalFree((HGLOBAL)lpBits);
}
```

Листинг 16 – Функция для открытия BMP файла (MSDN)

Сообщение WM_COPYDATA передается тогда, когда одна программа пересылает данные в другую программу. Синтаксис:

```
wParam = (WPARAM) (HWND) hWnd; // дескриптор передающего окна
lParam = (LPARAM) (PCOPYDATASTRUCT) pcds; // указатель на структуру с данными
```

Принимающий данные процесс должен обрабатывать сообщение WM_COPYDATA в главном окне. Данные получаются в виде структуры:

Структура COPYDATASTRUCT

```
typedef struct tagCOPYDATASTRUCT
{
        DWORD dwData;
        DWORD cbData;
        PVOID lpData;
} COPYDATASTRUCT;
```

dwData – устанавливает до 32 битов данных, которые будут переданы в принимающую прикладную программу, **cbData** – устанавливает размер, в байтах, данных, указанных элементом структуры **lpData**, **lpData** – указывает на данные, которые будут переданы в принимающую прикладную программу. Этот элемент структуры может быть значением NULL.

Чтобы послать сообщение из процесса в процесс нужно воспользоваться API-функцией SendMessage:

Функция SendMessage

```
LRESULT SendMessage(
        HWND hWnd,
        UINT message,
        WPARAM wParam = 0,
        LPARAM lParam = 0
);
```

Параметры:
hWnd – дескриптор окна, которому посылается сообщение.
message – определяет сообщение которое будет послано.
wParam – определяет дополнительную зависимую от сообщения информацию.
pParam – определяет дополнительную зависимую от сообщения информацию.
Возвращаемое значение
Результат обработки сообщения, значение зависит от посланного сообщения.

Для этого необходимо знать HWND окна, в который вы будете посылать сообщение. Его вы можете получить с помощью функции **FindWindowEx**.

Задание:
1. Написать программу «Источник данных» со следующими функциями:
– запись строки в файл средствами API, блокируя доступ к записи и чтению файла другими программами;
– передача текстовой строки с помощью сообщения WM_COPYDATA другому приложению.
2. Написать программу «Приёмник данных» со следующими функциями:
– опрос по таймеру файла, на наличие в нём сообщений;

– приём и обработка сообщения WM_COPYDATA от «Источника данных».

Контрольные вопросы:
1. Каковы параметры функции CreateProcess?
2. Каковы режимы чтения и записи файлов?
3. Какой формат сообщения WM_COPYDATA?
4. Как осуществляется приём и обработка сообщения WM_COPYDATA?

Для самостоятельного изучения:
1. Другие способы сообщения между независимыми процессами.

4.7 Управление потоками и работа с файлами средствами Win32API

Для создания дополнительных потоков в программе используется функция:

Функция CreateThread

```
HANDLE CreateThread
(
    LPSECURITY_ATTRIBUTES lpThreadAttributes,    // дескриптор защиты
    SIZE_T dwStackSize,                          // начальный размер стека
    LPTHREAD_START_ROUTINE lpStartAddress,       // функция потока
    LPVOID lpParameter,                          // параметр потока
    DWORD dwCreationFlags,                       // опции создания
    LPDWORD lpThreadId                           // идентификатор потока
);
```

Функция **CreateThread** создает поток, который выполняется в пределах виртуального адресного пространства вызывающего процесса. Чтобы создавать поток, который запускается в виртуальном адресном пространстве другого процесса, используется функция **CreateRemoteThread**.

Параметры:

lpThreadAttributes – указатель на структуру SECURITY_ATTRIBUTES, которая обуславливает, может ли возвращенный дескриптор быть унаследован дочерними процессами. Если **lpThreadAttributes** является значением NULL, дескриптор не может быть унаследован.

dwStackSize – начальный размер стека, в байтах. Система округляет это значение до самой близкой страницы памяти. Если это значение нулевое, новый поток использует по умолчанию размер стека исполняемой программы.

lpStartAddress – указатель на определяемую программой функцию типа LPTHREAD_START_ROUTINE, код которой исполняется потоком и обозначает начальный адрес потока. Для получения дополнительной информации о функции потока, см. **ThreadProc**.

lpParameter – указатель на переменную, которая передается в поток.

dwCreationFlags – флажки, которые управляют созданием потока. Если установлен флажок CREATE_SUSPENDED, создается поток в состоянии ожидания и не

запускается до тех пор, пока не будет вызвана функция **ResumeThread**. Если это значение нулевое, поток запускается немедленно после создания.
lpThreadId – указатель на переменную, которая принимает идентификатор потока.

Возвращаемые значения

Если функция завершается успешно, величина возвращаемого значения – дескриптор нового потока. Если функция завершается с ошибкой, величина возвращаемого значения – NULL. Чтобы получать дополнительные данные об ошибках, вызовите **GetLastError**.

При успешном выполнении функция создаёт поток, возвращает его дескриптор, а в переменную, на которую указывает параметр **lpThreadId**, заносится идентификатор потока. Выполнение потока начинается с выполнения функции, адрес которой указывает параметр **lpStartAddress**. Это любая функция, в которую передается один аргумент, являющийся указателем на любую величину – **lpParameter**. Если передавать ничего не надо, то надо присваивать значение NULL. Формат функции:

```
DWORD WINAPI Название_функции(LPVOID);
```

Работа потока завершается оператором **return** в конце функции. Пример создания потока представлен в листинге 17. Здесь **Func** – указывает на функцию, с которой выполняется новый поток.

```
DWORD lpT;
HANDLE h=CreateThread(NULL,0,Func,NULL,0,&lpT);
CloseHandle(h);
```

Листинг 17 – Создание нового потока

В качестве примера использования нескольких потоков в одном процессе, можно привести ситуацию, когда приложению нужно записать большой файл на диск. При использовании одного потока – доступ к другим функциям программы будет недоступен до окончания операции.

Одной из распространенных задач, является поиск файлов в заданном каталоге. Для поиска файлов используются две функции **FindFirstFile** и **FindNextFile**. Алгоритм работы следующий – создается рекурсивная функция, например **FindFile**. В неё передается начальный каталог, в котором необходимо найти все входящие файлы и каталоги. Первоначально вызывается функция:

Функция FindFirstFile

```
HANDLE FindFirstFile
(
    LPCTSTR lpFileName,                 // адрес пути для поиска
    LPWIN32_FIND_DATA lpFindFileData);  // адрес структуры
                                        // LPWIN32_FIND_DATA, куда будет записана
                                        // информация о файлах
);
```

Через параметр **lpFileName** вы должны передать функции адрес строки,

содержащей путь к каталогу и шаблон для поиска. В шаблоне можно использовать символы «?» и «*». Через параметр **lpFindFileData** следует передать адрес структуры типа WIN32_FIND_DATA, в которую будет записана информация о найденных файлах. Эта структура определена следующим образом:

Структура WIN32_FIND_DATA

```
typedef struct _WIN32_FIND_DATA
{
        DWORD    dwFileAttributes;          // атрибуты файла
        FILETIME ftCreationTime;            // время создания файла
        FILETIME ftLastAccessTime;          // время доступа
        FILETIME ftLastWriteTime;           // время записи
        DWORD    nFileSizeHigh;             // размер файла (старшее слово)
        DWORD    nFileSizeLow;              // размер файла (младшее слово)
        DWORD    dwReserved0;               // зарезервировано
        DWORD    dwReserved1;               // зарезервировано
        TCHAR    cFileName[MAX_PATH];       // имя файла
        TCHAR    cAlternateFileName[14];    // альтернативное имя файла
} WIN32_FIND_DATA;
```

Если поиск завершился успешно, функция **FindFirstFile** возвращает идентификатор поиска, который будет затем использован в цикле при вызове функции **FindNextFile**. При ошибке возвращается значение INVALID_HANDLE_VALUE. После вызова функции **FindFirstFile** вы должны выполнять в цикле вызов функции **FindNextFile**:

Функция FindNextFile

```
BOOL FindNextFile
(
        HANDLE hFindFile,                   // идентификатор поиска
        LPWIN32_FIND_DATA lpFindFileData);  // адрес структуры
                                            // WIN32_FIND_DATA
);
```

Через параметр **hFindFile** этой функции следует передать идентификатор поиска, полученный от функции **FindFirstFile**. Что же касается параметра **lpFindFileData**, то через него вы должны передать адрес той же самой структуры типа WIN32_FIND_DATA, что была использована при вызове функции **FindFirstFile**.

Если функция **FindNextFile** завершилась успешно, она возвращает значение TRUE. При ошибке возвращается значение FALSE. Код ошибки вы можете получить от функции **GetLastError**. В том случае, когда были просмотрены все файлы в каталоге, эта функция возвращает значение ERROR_NO_MORE_FILES. Вы должны использовать такую ситуацию для завершения цикла просмотра содержимого каталога.

После завершения цикла просмотра необходимо закрыть идентификатор

поиска, вызвав для этого функцию **FindClose**:

Функция FindClose

```
BOOL FindClose(HANDLE hFindFile);
```

Задание:
1. Написать программу:
– создающую отдельный поток, которому передается название каталога, в котором будут искаться файлы;
– имеющую рекурсивную функцию FindFile, строящую «дерево» файлов и каталогов от выбранного каталога.

Контрольные вопросы:
1. Для чего необходимо использование потоков?
2. Приведите пример многопоточного приложения.
3. Как работает функция CreateThread?
4. Объясните принцип рекурсивного поиска файлов.

Для самостоятельного изучения:
1. Функции ExitProcess, ExitThread, CreateRemoteThread.
2. Существующие компоненты для автоматизированного поиска файлов.

4.8 Синхронизация процессов и потоков

Семафор – объект синхронизации, который может регулировать доступ к некоторому ресурсу. Мьютекс – переменная, которая может находиться в одном из двух состояний: блокированном или неблокированном. Мьютекс может охранять неразделенный ресурс, к которому в каждый момент времени допускается только один поток, а семафор может охранять ресурс, с которым может одновременно работать не более **N** потоков.

Мьютексный объект создаётся функцией:

Функция CreateMutex

```
HANDLE CreateMutex
(
    LPSECURITY_ATTRIBUTES lpMutexAttributes,    // атрибут безопастности
    BOOL bInitialOwner,                         // флаг начального владельца
    LPCTSTR lpName                              // имя объекта
);
```

Функция возвращает дескриптор мьютексного объекта с именем, заданным параметром lpName. Имя мьютекса не должно совпадать с именем уже существующего события, семафора и других объектов межпроцессной синхронизации. Функция **GetLastError** при вызове будет выдавать

ERROR_ALREADY_EXISTS.

Для открытия существующего мьютекса используется функция **OpenMutex**, освобождения – **ReleaseMutex**.

Семафор создаётся функцией:

Функция CreateSemaphore

```
HANDLE CreateSemaphore
(
    LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,    // атрибут доступа
    LONG lInitialCount,                             // инициализированное начальное
                                                    // состояние счетчика
    LONG lMaximumCount,                             // максимальное количество
                                                    // обращений
    LPCTSTR lpName                                  // имя объекта
);
```

При успешном выполнении функция вернет идентификатор семафора, в противном случае NULL. После того как необходимость в работе с объектом отпала нужно вызвать функцию **ReleaseSemaphore**, чтобы освободить счетчик. Для открытия существующего семафора используется функция **OpenSemaphore**.

Объекты синхронизации используются совместно с функциями ожидания. Эти функции связываются с одним или несколькими синхронизирующими объектами и ждут, когда эти объекты перейдут в сигнальное состояние. В результате выполнение процесса приостанавливается до тех пор, пока в системе не произойдёт некоторое событие. Ожидание одного события организуется функцией:

Функция WaitForSingleObject

```
DWORD WaitForSingleObject
(
    HANDLE hHandle,
    DWORD dwMilliseconds
);
```

Здесь первый параметр – дескриптор объекта синхронизации, а второй – время ожидания в миллисекундах. Для ожидания нескольких синхронизирующих объектов используется функция **WaitForMultipleObjects**.

В листинге 18 приведен пример программы использования мьютекса. При запуске программы она создает объект с именем «MMM», если захватить его удается **WaitForSingleObject**, то программа выполняется якобы, ждет пока введете число и нажмете Enter. Если захватить его не удается, то выходит с надписью fail. Если Вы попробуете, то Вы сможете запустить только одну копию.

```
#include "windows.h"
#include "iostream.h"
```

```
void main()
{
	HANDLE mut;
	mut = CreateMutex(NULL, FALSE, "MMM");
	DWORD result;
	result = WaitForSingleObject(mut,0);
	if (result == WAIT_OBJECT_0)
	{
		cout << "programm running" << endl;
		int i;
		cin >> i;
		ReleaseMutex(mut);
	}
	else
		cout << "fail programm running" << endl;
	CloseHandle(mut);
}
```

Листинг 18 – Пример использования мьютекса

Задание:

1. Написать программы с требованиями:

– может запуститься только один экземпляр приложения, для определения запущено приложение или нет использовать мьютекс (проверка, не создан ли мьютекс предыдущим экземпляром);

– запускаются два экземпляра программы, при нажатии на кнопку первого экземпляра эмулируется захват ресурса и устанавливается мьютекс, второй экземпляр, при нажатии на кнопку ожидает с использованием функции WaitForMultipleObjects освобождение ресурса в течении 3 секунд и, если ресурс освобождается, то он захватывается 2-ым экземпляром, в противном случае выводится на экран невозможность захвата ресурса;

– может запуститься только 4 экземпляра одного приложения, реализация с использованием семафора.

Контрольные вопросы:

1. Для чего нужны объекты синхронизации?
2. Чем отличается мьютекс от семафора?
3. Как работает функция CreateSemaphore?
4. Для чего нужны функции ожидания?
5. Чем отличаются функции WaitForSingleObject и WaitForMultipleObjects?

Для самостоятельного изучения:

1. Функции OpenMutex, OpenSemaphore, WaitForMultipleObjects.
2. Объекты событий.
3. Таймеры ожидания.

4.9 Управление памятью

Для управления памятью в операционной системе Windows предусмотрен ряд функций.

Функция GlobalAlloc

```
HGLOBAL GlobalAlloc
(
        UINT uFlags,
        SIZE_T dwBytes
);
```

Функция выделяет из глобальной кучи память запрошенного размера.

Параметры:

uFlags – маска флагов.

dwBytes – размер выделяемой памяти

Возвращаемое значение:

Идентификатор выделенного блока глобальной памяти; 0 - если ошибка.

В листинге 19 пример вызова функции Windows для выделения блока памяти для указателя на целые.

```
DWORD dwSize = 1024;
UINT uiFlags = 0;
p = (int *)GlobalAlloc(uiFlags, dwSize);
```

Листинг 19 – Выделение памяти с помощью GlobalAlloc

За исключением одной, для каждой функции, начинающейся со слова Global, существует другая, начинающаяся со слова Local. Эти два набора функций в Windows идентичны. Два различных слова сохранены для совместимости с предыдущими версиями Windows, где функции Global возвращали дальние указатели, а функции Local – ближние. Если первый параметр задать нулевым, то это эквивалентно использованию флага GMEM_FIXED. Такой вызов функции **GlobalAlloc** эквивалентен вызову функции **malloc**.

Следующий пример демонстрирует использование функции изменения размера блока памяти:

```
p = (int *)GlobalReAlloc(p, dwSize, uiFlags);
```

Для определения размера выделенного блока памяти используется функция **GlobalSize**:

```
dwSize = GlobalSize(p);
```

Функция освобождения памяти:

```
GlobalFree(p);
```

Функция **GlobalAlloc** поддерживает флаг GMEM_MOVEABLE и комбинированный флаг для дополнительного обнуления блока памяти (описано в заголовочных файлах Windows):

```
#define GHND (GMEM_MOVEABLE | GMEM_ZEROINIT)
```

Флаг GMEM_MOVEABLE позволяет перемещать блок памяти в виртуальной памяти, при этом функция возвращает не адрес выделенного блока, а 32-разрядный

описатель (дескриптор) блока памяти. Это необязательно означает, что блок памяти будет перемещен в физической памяти, но адрес, которым пользуется программа для чтения и записи, может измениться. Для фиксации блока используется вызов:

```
p =(int *)GlobalLock(hGlobal);
```

Эта функция преобразует описатель памяти в указатель. Пока блок зафиксирован, Windows не изменяет его виртуальный адрес. Когда работа с блоком заканчивается, для снятия фиксации вызывается функция:

```
GlobalUnlock(hGlobal);
```

Этот вызов дает Windows свободу перемещать блок в виртуальной памяти. Для того чтобы правильно осуществлять этот процесс следует фиксировать и снимать фиксацию блока памяти в ходе обработки одного сообщения. Когда нужно освободить перемещаемую память, надо вызывать функцию **GlobalFree** с описателем, но не с указателем на блок памяти.

Если в данный момент нет доступа к описателю, то необходимо использовать функцию:

```
hGlobal = GlobalHandle(p);
```

Для преднамеренного удаления блока памяти можно использовать следующий вызов:

```
GlobalDiscard(hGlobal);
```

Другим доступным для использования в функции **GlobalAlloc** является флаг GMEM_SHARE или GMEM_DDESHARE (идентичны). Как следует из его имени, этот флаг предназначен для динамического обмена данными. Функции **GlobalAlloc** и **GlobalReAlloc** могут также включать флаги GMEM_NODISCARD и GMEM_NOCOMPACT. Эти флаги дают указание Windows не удалять и не перемещать блоки памяти для удовлетворения запросов памяти.

Функция **GlobalFlags** возвращает комбинацию флагов GMEM_DISCARDABLE, GMEM_DISCARDED и GMEM_SHARE. Наконец, вы можете вызвать функцию **GlobalMemoryStatus** (для этой функции нет функции – двойника со словом Local) с указателем на структуру типа MEMORYSTATUS для определения количества физической и виртуальной памяти, доступной приложению.

Windows также поддерживает некоторые функции, реализованные программистом или дублирующие библиотечные функции С. Это функции **FreeMemory** (заполнение конкретным байтом), **ZeroMemory** (обнуление памяти), CopyMemory и **MoveMemory** – обе копируют данные из одной области памяти в другую. Если эти области перекрываются, то функция **CopyMemory** может работать некорректно. Вместо нее используйте функцию **MoveMemory**.

Windows поддерживает ряд функций, начинающихся со слова Virtual. Эти функции предоставляют значительно больше возможностей управления памятью. Однако, только очень необычные приложения требуют использования этих функций.

Последняя группа функций работы с памятью – это функции, имена которых начинаются со слова Heap. Эти функции создают и поддерживают непрерывный блок виртуальной памяти, из которого можно выделять память более мелкими блоками. Следует начинать с вызова функции **HeapCreate**. Затем, использовать

функции **HeapAllocate**, **HeapReAllocate** и **HeapFree** для выделения и освобождения блоков памяти в рамках «кучи».

Задание:
1. Написать программу:

– средствами Win32 API выделяющая память для массива размерностью M на N, размерность вводится с клавиатуры;

– случайным образом заполняется весь массив информацией;

– необходимо в каждой строке найти элемент с наименьшим значением, а затем среди этих чисел найти наибольшее. На экран вывести индексы этого элемента, а также все элементы матрицы.

Контрольные вопросы:
1. Какие средства поддерживает язык C/C++ для управления памятью?
2. Назовите основные флаги функции GlobalAlloc и их назначение?
3. Чем отличаются функции Global и Local?
4. Какими функциями предпочтительнее выделять память для функций Win32API?

Для самостоятельного изучения:
1. Методы C/C++ выделения памяти.

4.10 Дочерние окна и управление «чужим» приложением

Некоторые из окон Windows являются дочерними – т.е. помещаются в клиентской области другого, родительского окна. Если известен дескриптор некоторого окна, то можно получить указатель на его родительское окно с помощью функции:

Функция GetParent

```
HWND GetParent
(
    HWND hWnd          // дескриптор дочернего окна
);
```

Параметры:
hWnd – идентифицирует окно, дескриптор родительского окна которого должен быть найден.

Возвращаемые значения

Если функция завершилась успешно, возвращаемое значение – дескриптор родительского окна. Если у окна нет родительского окна, возвращаемое значение – NULL. Чтобы получить более подробную информацию об ошибке, вызовите **GetLastError**.

Добраться до родительского окна можно с помощью следующей операции:

```
while (H=GetParent(hWnd)) hWnd=H;
```

где сначала в **hWnd** заносится дочернее окно, а после выполнения – остаётся родительское.

Перебрать все дочерние окна некоторого родительского окна можно с помощью функции:

Функция EnumChildWindows

```
BOOL EnumChildWindows
(
    HWND hWndParent,            // дескриптор родительского окна
    WNDENUMPROC lpEnumFunc,     // указатель на функцию обратного вызова
    LPARAM lParam               // значение, определяемое программой
);
```

Параметры:

hWndParent – идентифицирует родительское окно, чьи дочерние окна должны перечисляться.

lpEnumFunc – указывает на определяемую программой функцию повторного вызова. Для получения дополнительной информации относительно функции повторного вызова, см. функцию повторного вызова **EnumChildProc**.

lParam – устанавливает 32-разрядное, определяемое программой значение, которое будет передано в функцию повторного вызова.

Возвращаемые значения

Если функция завершилась успешно, возвращается значение отличное от нуля. Если функция потерпела неудачу, возвращаемое значение – ноль.

Пользовательская функция объявляется следующим образом:

```
BOOL __stdcall EnumChild(HWND hWnd, LPARAM lParam);
```

Эту функцию объявляет и обрабатывает пользователь, при этом первым параметром приходит дескриптор дочернего окна.

Для получения имени дочернего окна можно воспользоваться функцией **SendMessage**, например так:

```
SendMessage(hWnd,WM_GETTEXT,sizeof(buf),(LPARAM) (LPCTSTR) buf);
```

где **buf**, должен быть объявлен ранее, например, так:

```
char buf[256];
```

Для получения класса окна, необходимо воспользоваться функцией **GetClassName**.

Для полного контроля над чужим приложением необходимо получить доступ в меню, это можно сделать с помощью функции:

```
HMENU GetMenu(IN HWND hWnd);
```

Определив дескриптор меню, легко узнать количество его разделов:

```
int GetMenuItemCount(IN HMENU hMenu);
```

А тексты разделов получаются с помощью функции:

```
int GetMenuString(IN HMENU hMenu, IN UINT uIDItem, OUT LPSTR lpString, IN int nMaxCount,
IN UINT uFlag);
```

или с помощью функции:

```
BOOL GetMenuItemInfo(IN HMENU hMenu, IN UINT uItem, IN BOOL fByPosition,
```

IN OUT LPMENUNITEMINFO lpmii);

Следующая функция позволяет получить дескриптор выпадающего окна:

HMENU GetSubMenu(IN MENU hMenu, IN int nPos);

Для имитации щелчка на кнопке достаточно проделать следующую операцию:

```
SendMessage(hWnd, WM_LBUTTONDOWN,0,0);
SendMessage(hWnd, WM_LBUTTONUP,0,0);
```

где, естественно **hWnd** должно содержать в себе указатель на кнопку чужого приложения.

Для имитации щелчка в разделе меню необходимо сделать следующее:

```
SendMessage(hWnd, WM_COMMAND,*(UINT*)Param,0);
```

где **param** – номер пункта меню.

В листинге 20 приведен пример получения доступа к меню Калькулятора.

```
#include "windows.h"
#include "iostream.h"

void main()
{
	HWND hwnd;
	hwnd=FindWindow("SciCalc","Калькулятор");
	if (hwnd!=NULL)
	{
		HMENU hMenu;
		hMenu=GetMenu(hwnd);
		if (hMenu!=NULL)
		{
			int iCount;
			Count=GetMenuItemCount(hMenu);
			cout << "Menu Item - " << iCount << endl;
		}
		else cout << " Error Loading Menu" << endl;
	}
	else cout << " Error Find Windows" << endl;
}
```

Листинг 20 – Получения доступа к меню программы «Калькулятор»

Задание:

1. Написать программу:

– запускающую «Калькулятор» и получающую список его дочерних окон и пунктов меню;

– эмулировать программно нажатия пунктов меню и кнопок.

Контрольные вопросы:

1. Как можно в операционной системе Windows найти запущенную программу?
2. Что такое дочернее окно и как к нему получить доступ из внешней программы?
3. Как, зная информацию о дочернем окне, найти главное окно?
4. Как получить доступ к кнопке?
4. Как получить доступ к пункту меню?

Для самостоятельного изучения:
1. Управление сообщениями основного цикла главного окна из внешнего приложения.

4.11 Решение задачи производителя и потребителя

На данном этапе необходимо воспользоваться накопленными в ходе выполнения предыдущих этапов знаниями для решения задачи производителя и потребителя, подробно описанной в параграфах 2-3.

Список использованных источников

1 Архангельский, А.Я. Приёмы программирования в C++ Builder. Механизмы Windows, сети / А.Я. Архангельский, М.А. Тагин. – М. : ООО «Бином-Пресс», 2004. – 656 с.
2 Баженова, И.Ю Visual C++ 6.0. Уроки программирование / И.Ю. Баженова. – М. : Диалог-МИФИ, 1999. – 416 с.
3 Операционные системы. Основы и принципы. 3-е изд. / Х.М. Дейтел, П.Дж. Дейтел. – М. : Бином, 2006. – 1024 с.
4 Саймон, Р. Windows 2000 API. Энциклопедия программиста : пер. с англ. / Р. Саймон. – СПб. : ООО «ДиаСофтЮП», 2002. – 1088 с.
5 Сетевые операционные системы / В.Г. Олифер, Н.А. Олифер. – СПб. : Питер, 2001. – 544 с.
6 Таненбаум, Э. Современные операционные системы : пер. с англ. / Э. Таненбаум. – СПб. : Питер, 2002. – 2 изд. –1040 с.
7 Щупак, Ю. А. Win32 API. Эффективная разработка приложений / Ю. А. Щупак. – СПб. : Питер, 2007. – 572 с.