Министерство образования и науки Российской Федерации

Кубанский государственный технологический университет

Кафедра вычислительной техники и АСУ

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ
Часть 3. Разработка ПО системного назначения

Лабораторный практикум для студентов всех форм обучения специальности 230101 Вычислительные машины, комплексы, системы и сети

Краснодар
2011

Составители: канд. техн. наук, доц. А. Г. Мурлин;
ст. преп. А. Г. Волик

УДК 681.31(031)

Системное программное обеспечение. Ч. 3: лабораторный практикум для студентов всех форм обучения специальности 230101 Вычислительные машины, комплексы, системы и сети / Сост.: А. Г. Мурлин, А. Г. Волик; Кубан. гос. технол. ун-т. Каф. вычислительной техники и АСУ. – Краснодар.: Изд. КубГТУ, 2011 – 81 с.

Содержат описания лабораторных работ, указания к их выполнению, задания и требования к оформлению отчета. Изложены основы разработки программ системного назначения. Рассмотрены принципы работы микропроцессора в защищенном режиме. Приведены исходные тексты примеров программ и рекомендуемая литература.

Ил. 14. Табл. 12. Библиогр.: 8 назв.

Печатается по решению методического совета Кубанского государственного технологического университета

Рецензенты:
зав. кафедрой ВТиАСУ КубГТУ, д-р техн. наук, проф. В. И. Ключко;
руководитель отдела телекоммуникаций ООО Информационный центр «Консультант», канд. техн. наук Н. Ф. Григорьев

Содержание

Введение

Методические указания содержат лабораторный практикум по дисциплине «Системное программное обеспечение» для специальности 230101 Вычислительные машины, комплексы, системы и сети.

Целью лабораторных работы является закрепление основ и углубление знаний в области устройства современных операционных систем семейства Windows с точки зрения программиста и получения практического опыта написания программ системного назначения, а так же ознакомление студентов со средствами компиляции и отладки программ, которые предназначены для низкоуровневого взаимодействия с ОС.

При выполнении лабораторных работ должен соблюдаться следующий порядок выполнения работы:

- ознакомиться с описанием лабораторной работы;
- получить номер варианта задания у преподавателя;
- изучить необходимый теоретический материал, пользуясь настоящими указаниями и рекомендованной литературой;
- написать программу и отладить ее;
- подготовиться к ответам на теоретические вопросы по теме лабораторной работы;
- оформить отчет.

Все студенты должны предъявить индивидуальный отчет о результатах выполнения лабораторной работы. Допускается предъявление отчета в виде электронного документа.

Отчет должен содержать следующие пункты:

1) Титульный лист.
2) Краткое теоретическое описание.
3) Задание на лабораторную работу, включающее четкую формулировку задачи.
4) Листинг программы.
5) Результаты выполнения работы.

При сдаче отчета студент должен показать знание теоретического материала в объеме, определяемом тематикой лабораторной работы, а также пониманием сущности выполняемой работы.

Лабораторная работа № 12. Многозадачность и многопоточность

1 Цель работы

Изучить поддержку многозадачности и многопоточности в ОС Windows.

2 Краткая теория

Многозадачность (multitasking) – это способность операционной системы выполнять несколько программ одновременно. В основе этого принципа лежит использование операционной системой аппаратного таймера для выделения отрезков времени (time slices) для каждого из одновременно выполняемых процессов. Если эти отрезки времени достаточно малы, и машина не перегружена слишком большим числом программ, то пользователю кажется, что все эти программы выполняются параллельно.

Многопоточность – это возможность программы самой быть многозадачной. Программа может быть разделена на отдельные потоки выполнения (threads), которые, как кажется, выполняются параллельно. На первый взгляд эта концепция может показаться едва ли полезной, но оказывается, что программы могут использовать многопоточность для выполнения протяженных во времени операций в фоновом режиме, не вынуждая пользователя надолго отрываться от машины.

2.1 Многопоточность

В многопоточной среде программы могут быть разделены на части, называемые потоками выполнения (threads), которые выполняются одновременно. Поддержка многопоточности оказывается лучшим решением проблемы последовательной очереди сообщений.

В терминах программы "поток" – это просто функция, которая может также вызывать другие функции программы. Программа начинает выполняться со своего главного (первичного) потока, который в традиционных программах на языке С является функцией `main`, а в Windows-программах – `WinMain`. Будучи выполняемой, функция может создавать новые потоки обработки, выполняя системный вызов с указанием функции инициализации потока (initial threading function). Операционная система переключает управление между потоками подобно тому, как она это делает с процессами.

Првичный или главный (primary) поток программы создает все окна и соответствующие им оконные процедуры, необходимые в программе и обрабатывает все сообщения для этих окон. Все остальные потоки – это просто фоновые задачи. Они не имеют интерактивной связи с пользователем, кроме как через первичный поток.

Один из способов добиться этого состоит в том, чтобы первичный поток обрабатывают пользовательский ввод и другие сообщения, возможно создавая при этом вторичные (secondary) потоки в процессе. Эти вторичные потоки выполняют не связанные с пользователем задачи.

Потоки внутри отдельной программы являются частями одного процесса, поэтому они разделяют все ресурсы процесса, такие как память и открытые файлы. Поскольку потоки разделяют память, отведенную программе, то они разделяют и статические переменные. Однако у каждого потока есть свой собственный стек, и значит, автоматические переменные являются уникальными для каждого потока. Каждый поток, также, имеет свое состояние процессора, которое сохраняется и восстанавливается при переключении между потоками.

2.2 Коллизии, возникающие при использовании потоков

Собственно разработка, программирование и отладка сложного многопоточного приложения являются, естественно, самыми сложными задачами, с которыми приходится сталкиваться программисту для Windows. Поскольку в системе с вытесняющей многозадачностью поток может быть прерван в любой момент для переключения на другой поток, то может случайно произойти любое нежелательное взаимодействие между двумя потоками.

Одной из основных ошибок в многопоточных программах является так называемое состояние гонки (race condition). Это случается, если программист считает, что один поток закончит выполнение своих действий, например, подготовку каких-либо данных, до того, как эти данные потребуются другому потоку. Для координации действий потоков операционным системам необходимы различные формы синхронизации. Одной из таких форм является семафор (semaphore), который позволяет программисту приостановить выполнение потока в конкретной точке программы до тех пор, пока он не получит от другого потока сигнал о том, что он может возобновить работу. Похожи на семафоры критические разделы (critical sections), которые представляют собой разделы кода, во время выполнения которого, поток не может быть прерван.

Но использование семафоров может привести к другой распространенной ошибке, связанной с потоками, которая называется тупиком (deadlock). Это случается, когда два потока блокируют выполнение друг друга, а для того, чтобы их разблокировать необходимо продолжить работу.

2.3 Возможности Windows по работе с потоками

В Windows не существует различия между потоками, имеющими очередь сообщений, и потоками без очереди сообщений. При создании

каждый поток получает свою собственную очередь сообщений. Такая схема организации приложения, почти всегда является наиболее разумной.

В Windows есть функция, которая позволяет одному потоку уничтожить другой поток, принадлежащий тому же процессу.

Помимо этого Windows поддерживают так называемую локальную память потока (thread local storage, TLS). Для того чтобы понять, что это такое вспомним о том, что статические переменные, как глобальные так и локальные по отношению к функциям, разделяются между потоками, поскольку они расположены в зоне памяти данных процесса. Автоматические переменные (которые являются всегда локальными по отношению к функции) – уникальны для каждого потока, т. к. они располагаются в стеке, а каждый поток имеет свой стек.

Иногда бывает удобно использовать для двух и более потоков одну и ту же функцию, а статические данные использовать уникальные для каждого потока. Это и есть пример использования локальной памяти потока. Существует несколько вызовов Функций Windows для работы с локальной памятью потока.

Нет смысла использовать множество потоков в программе, которая в этом не нуждается. В противном случае, вы только усложните себе работу и, возможно, внесете в программу новые ошибки.

Рассмотрим несколько программ, использующих многопоточность.

Программа RNDRCTMT является многопоточной и выводит последовательности случайных прямоугольников используя цикл обработки сообщений, содержащий вызов функции *PeekMessage*.

```
/*----------------------------------------
RNDRCTMT.C -- Displays Random Rectangles
----------------------------------------*/
#include <windows.h>
#include <process.h>
#include <stdlib.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

HWND hwnd;
int  cxClient, cyClient;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
    PSTR szCmdLine, int iCmdShow)
{
    static char szAppName[] = "RndRctMT" ;
    MSG         msg ;
    WNDCLASS  wndclass ;

    wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
    wndclass.lpfnWndProc   = WndProc ;
    wndclass.cbClsExtra    = 0 ;
```

```
    wndclass.cbWndExtra    = 0 ;
    wndclass.hInstance     = hInstance ;
    wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
    wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
    wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
    wndclass.lpszMenuName  = NULL ;
    wndclass.lpszClassName = szAppName ;

    RegisterClass (&wndclass) ;

    hwnd = CreateWindow (szAppName, "Random Rectangles",
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, CW_USEDEFAULT,
        CW_USEDEFAULT, CW_USEDEFAULT,
        NULL, NULL, hInstance, NULL) ;

    ShowWindow (hwnd, iCmdShow) ;
    UpdateWindow (hwnd) ;

    while (GetMessage (&msg, NULL, 0, 0))
    {
        TranslateMessage (&msg);
        DispatchMessage (&msg);
    }
    return msg.wParam;
}

VOID Thread (PVOID pvoid)
{
    HBRUSH hBrush ;
    HDC    hdc ;
    int    xLeft, xRight, yTop, yBottom, iRed, iGreen, iBlue;

    while (TRUE)
    {
        if (cxClient != 0 || cyClient != 0)
        {
            xLeft   = rand () % cxClient ;
            xRight  = rand () % cxClient ;
            yTop    = rand () % cyClient ;
            yBottom = rand () % cyClient ;
            iRed    = rand () & 255 ;
            iGreen  = rand () & 255 ;
            iBlue   = rand () & 255 ;
            hdc = GetDC (hwnd);
            hBrush = CreateSolidBrush (RGB (iRed, iGreen, iBlue));
            SelectObject (hdc, hBrush) ;

            Rectangle (hdc, min (xLeft, xRight), min (yTop, yBottom),
                max (xLeft, xRight), max (yTop, yBottom));

            ReleaseDC (hwnd, hdc);
            DeleteObject (hBrush);
        }
```

```
    }
}

static int IDTh = 0;

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM
lParam)
{
    switch (iMsg)
    {
    case WM_CREATE:

#ifndef __WATCOMC__
        if (! _beginthread( Thread, 0, NULL ) )
#else
        if (! _beginthread( Thread, 0, 0, NULL ) )
#endif
/*      if (!CreateThread (NULL, 0, ( LPTHREAD_START_ROUTINE)Thread, N
ULL, 0, (LPDWORD)& IDTh) ) /**/
            MessageBox(NULL, "CreateThread failed!","Error:", MB_OK);
        return 0 ;

    case WM_SIZE:
        cxClient = LOWORD (lParam) ;
        cyClient = HIWORD (lParam) ;
        return 0 ;

    case WM_DESTROY:
        PostQuitMessage (0);
        return 0 ;
    }
    return DefWindowProc (hwnd, iMsg, wParam, lParam);
}
```

На этапе компиляции необходимо установть переменную CFLAGSMT. Это переменная CFLAGS, но с добавлением флага _MT, который необходимо задать компилятору для компиляции многопоточного приложения. В частности, компилятор вставляет имя файла LIBCMT.LIB вместо LIBC.LIB в файл с расширением .OBJ. Эта информация используется компоновщиком.

Файлы LIBCMT.LIB и LIBC.LIB содержат библиотечные функции языка C. Некоторые библиотечные функции C используют статические переменные. Например, функция *strtok* разработана таким образом, чтобы ее можно было вызывать более чем один раз последовательно, и хранит указатель на строку в статической памяти. В многопоточной программе каждый поток должен иметь свой статический указатель в функции *strtok*. Т.е. многопоточная версия этой функции несколько отличается от однопоточной.

В программу RNDRCTMT.C включен заголовочный файл PROCESS.H. В этом файле описывается функция С с именем *beglnihread.* которая запускает новый поток. Файл не определен, если не определен флаг _MT, и это еще один результат включения его при компиляции.

В функции *WinMain* значение *hwnd,* возвращаемое функцией *CreateWindow,* сохраняется в глобальной переменной. Аналогичное происходит со значениями переменных *cxClient* и *cyClient,* полученными из сообщения WM_SIZE в оконной процедуре.

Оконная процедура вызывает функцию *_beginthread* самым простым способом используя только адрес функции потока, имеющей имя *Thread,* в качестве первого параметра, и остальные нулевые параметры, функция потока возвращает значение типа VOID и имеет один параметр типа указатель на VOID. Функция *Thread* в программе RNDRCTMT не использует этот параметр.

После вызова функции *_beginihread* код функции потока, также как и код любой другой функции, которая может быть вызвана из функции потока, выполняется одновременно с оставшимся кодом программы. Два и более потока могут использовать одну и ту же функцию процесса. В этом случае автоматические локальные переменные (хранящиеся в стеке) уникальны для каждого потока; все статические переменные являются общими для всех потоков процесса. Поэтому, оконная процедура устанавливает значения переменных *cxClient* и *cyClient.* а функция *Thread* может их использовать.

Рассмотрим пример, демонстрирующий эмитацию многопоточности с использоватнием таймера, который выводит в четырех частях окна различные данные: последовательность увеличивающихсяцелых чисел, простых чисел, чисел Фибоначчи и рисует круги различного радиуса.

```
/*--------------------------------------
MULTI1.C -- Multitasking Demo
--------------------------------------*/
#include <windows.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int cyChar ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
   PSTR szCmdLine, int iCmdShow)
{
   static char szAppName[] = "Multi1" ;
   HWND        hwnd ;
   MSG         msg ;
```

```
   WNDCLASS  wndclass ;

   wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
   wndclass.lpfnWndProc   = WndProc ;
   wndclass.cbClsExtra    = 0 ;
   wndclass.cbWndExtra    = 0 ;
   wndclass.hInstance     = hInstance ;
   wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
   wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
   wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
   wndclass.lpszMenuName  = NULL ;
   wndclass.lpszClassName = szAppName ;

   RegisterClass (&wndclass) ;

   hwnd = CreateWindow (szAppName, "Multitasking Demo",
      WS_OVERLAPPEDWINDOW,
      CW_USEDEFAULT, CW_USEDEFAULT,
      CW_USEDEFAULT, CW_USEDEFAULT,
      NULL, NULL, hInstance, NULL) ;

   ShowWindow (hwnd, iCmdShow) ;
   UpdateWindow (hwnd) ;

   while (GetMessage (&msg, NULL, 0, 0))
   {
      TranslateMessage (&msg) ;
      DispatchMessage (&msg) ;
   }
   return msg.wParam ;
}

int CheckBottom (HWND hwnd, int cyClient, int iLine)
{
   if (iLine * cyChar + cyChar > cyClient)
   {
      InvalidateRect (hwnd, NULL, TRUE) ;
      UpdateWindow (hwnd) ;
      iLine = 0 ;
   }
   return iLine ;
}

/* Window 1: Display increasing sequence of numbers */
LRESULT APIENTRY WndProc1 (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM
 lParam)
{
   static int   iNum, iLine ;
   static short cyClient ;
   char         szBuffer[16] ;
   HDC          hdc ;

   switch (iMsg)
   {
```

```
   case WM_SIZE :
      cyClient = HIWORD (lParam) ;
      return 0 ;

   case WM_TIMER :
      if (iNum < 0)
         iNum = 0 ;
      iLine = CheckBottom (hwnd, cyClient, iLine) ;
      wsprintf (szBuffer, "%d", iNum++) ;
      hdc = GetDC (hwnd) ;
      TextOut (hdc, 0, iLine * cyChar, szBuffer, strlen (szBuffer)) ;
      ReleaseDC (hwnd, hdc) ;
      iLine++ ;
      return 0 ;
   }
   return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}

/* Window 2: Display increasing sequence of prime numbers */
LRESULT APIENTRY WndProc2 (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM
 lParam)
{
   static int   iNum = 1, iLine ;
   static short cyClient ;
   char         szBuffer[16] ;
   int          i, iSqrt ;
   HDC          hdc ;

   switch (iMsg)
   {
   case WM_SIZE :
      cyClient = HIWORD (lParam) ;
      return 0 ;

   case WM_TIMER :
      do {
         if (++iNum < 0)
            iNum = 0 ;
         iSqrt = (int) sqrt( (float)iNum );

         for (i = 2 ; i <= iSqrt ; i++)
            if (iNum % i == 0) break ;
      } while (i <= iSqrt) ;

      iLine = CheckBottom (hwnd, cyClient, iLine) ;
      wsprintf (szBuffer, "%d", iNum) ;
      hdc = GetDC (hwnd) ;
      TextOut (hdc, 0, iLine * cyChar, szBuffer, strlen (szBuffer)) ;
      ReleaseDC (hwnd, hdc) ;
      iLine++ ;
      return 0 ;
   }
   return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}
```

```
/* Window 3: Display increasing sequence of Fibonacci numbers */
LRESULT APIENTRY WndProc3 (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM
 lParam)
{
   static int   iNum = 0, iNext = 1, iLine ;
   static short cyClient ;
   char         szBuffer[16] ;
   int          iTemp ;
   HDC          hdc ;

   switch (iMsg)
   {
   case WM_SIZE :
      cyClient = HIWORD (lParam) ;
      return 0 ;

   case WM_TIMER :
      if (iNum < 0)
      {
         iNum  = 0 ;
         iNext = 1 ;
      }
      iLine = CheckBottom (hwnd, cyClient, iLine);
      wsprintf (szBuffer, "%d", iNum);
      hdc = GetDC (hwnd);
      TextOut (hdc, 0, iLine * cyChar, szBuffer, strlen (szBuffer)) ;
      ReleaseDC (hwnd, hdc) ;
      iTemp  = iNum ;
      iNum   = iNext ;
      iNext += iTemp ;
      iLine++ ;
      return 0 ;
   }
   return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}

/* Window 4: Display circles of random radii */
LRESULT APIENTRY WndProc4 (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM
 lParam)
{
   static short cxClient, cyClient ;
   HDC          hdc ;
   int          iDiameter ;

   switch (iMsg)
   {
   case WM_SIZE :
      cxClient = LOWORD (lParam) ;
      cyClient = HIWORD (lParam) ;
      return 0 ;

   case WM_TIMER :
      InvalidateRect (hwnd, NULL, TRUE) ;
```

```
      UpdateWindow (hwnd) ;
      iDiameter = rand() % (max (1, min (cxClient, cyClient))) ;
      hdc = GetDC (hwnd) ;
      Ellipse (hdc, (cxClient - iDiameter) / 2,
         (cyClient - iDiameter) / 2,
         (cxClient + iDiameter) / 2,
         (cyClient + iDiameter) / 2) ;
      ReleaseDC (hwnd, hdc) ;
      return 0 ;
   }
   return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}

/* Main window to create child windows */
LRESULT APIENTRY WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM
lParam)
{
   static char   *szChildClass[] = { "Child1", "Child2",
      "Child3", "Child4" } ;
   static HWND    hwndChild[4] ;
   static WNDPROC ChildProc[] = { WndProc1, WndProc2,
      WndProc3, WndProc4 } ;
   HINSTANCE      hInstance ;
   int            i, cxClient, cyClient ;
   WNDCLASS     wndclass ;

   switch (iMsg)
   {
   case WM_CREATE :
      hInstance = (HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE) ;

      wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
      wndclass.cbClsExtra    = 0 ;
      wndclass.cbWndExtra    = 0 ;
      wndclass.hInstance     = hInstance ;
      wndclass.hIcon         = NULL ;
      wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
      wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
      wndclass.lpszMenuName  = NULL ;

      for (i = 0 ; i < 4 ; i++)
      {
         wndclass.lpfnWndProc   = ChildProc[i] ;
         wndclass.lpszClassName = szChildClass[i] ;
         RegisterClass (&wndclass) ;
         hwndChild[i] = CreateWindow (szChildClass[i], NULL,
            WS_CHILDWINDOW | WS_BORDER | WS_VISIBLE,
            0, 0, 0, 0, hwnd, (HMENU) i, hInstance, NULL) ;
      }
      cyChar = HIWORD (GetDialogBaseUnits ()) ;
      SetTimer (hwnd, 1, 10, NULL) ;
      return 0 ;

   case WM_SIZE :
```

```
            cxClient = LOWORD (lParam) ;
            cyClient = HIWORD (lParam) ;
            for (i = 0 ; i < 4 ; i++)
               MoveWindow (hwndChild[i], (i % 2) * cxClient / 2,
               (i > 1) * cyClient / 2,  cxClient / 2, cyClient / 2, TRUE) ;
            return 0 ;

         case WM_TIMER :
            for (i = 0 ; i < 4 ; i++)
               SendMessage (hwndChild[i], WM_TIMER, wParam, lParam);
            return 0 ;

         case WM_CHAR :
            if (wParam == '\x1B')
               DestroyWindow (hwnd);

            return 0 ;

         case WM_DESTROY :
            KillTimer (hwnd, 1) ;
            PostQuitMessage (0) ;
            return 0 ;
         }
         return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
      }
```

Главное окно создает четыре дочерних окна, каждое из которых занимает четверть рабочей области родительского окна. Главное окно также устанавливает таймер Windows и посылает сообщения WM_TIMER каждому из четырех дочерних окон.

Обычно, Windows-программа могла бы сохранять достаточно информации для того, чтобы обновлять содержимое окон в процессе обработки сообщения WM_PAINT. Программа MULTI1 не делает этого, а окна рисуются и обновляются настолько быстро, что, в этом и нет необходимости.

Использование таймера Windows – это отличный путь имитации многозадачности в ранних версиях Windows. Однако использование таймера иногда замедляет программу. Если программа может обновить все свои окна во время обработки одного сообщения WM_TIMER, имея запас времени, то при этом не используются все возможности компьютера. Одно из возможных решений этой проблемы – выполнение двух и более обновлений во время обработки одного сообщения WM_TIMER.

2.4 Пример использования многопоточности

Рассмотрим решение указанной задачи с применением многопоточности.

```
#include "stdafx.h"
```

```
/*----------------------------------------
   MULTI2.C -- Multitasking Demo
  ----------------------------------------*/

#include <windows.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <process.h>

typedef struct
{
  HWND hwnd ;
  int  cxClient ;
  int  cyClient ;
  int  cyChar ;
  BOOL bKill ;
} PARAMS, *PPARAMS ;

LRESULT APIENTRY WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
  static char szAppName[] = "Multi2" ;
  HWND        hwnd ;
  MSG         msg ;
  WNDCLASS  wndclass ;

  wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
  wndclass.lpfnWndProc   = WndProc ;
  wndclass.cbClsExtra    = 0 ;
  wndclass.cbWndExtra    = 0 ;
  wndclass.hInstance     = hInstance ;
  wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
  wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
  wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
  wndclass.lpszMenuName  = NULL ;
  wndclass.lpszClassName = szAppName ;

  RegisterClass (&wndclass) ;

  hwnd = CreateWindow (szAppName, "Multitasking Demo",
                       WS_OVERLAPPEDWINDOW,
                       CW_USEDEFAULT, CW_USEDEFAULT,
                       CW_USEDEFAULT, CW_USEDEFAULT,
                       NULL, NULL, hInstance, NULL) ;

  ShowWindow (hwnd, iCmdShow) ;
  UpdateWindow (hwnd) ;

  while (GetMessage (&msg, NULL, 0, 0))
  {
```

```
    TranslateMessage (&msg) ;
    DispatchMessage (&msg) ;
  }
  return msg.wParam ;
}

int CheckBottom (HWND hwnd, int cyClient, int cyChar, int iLine)
{
  if (iLine * cyChar + cyChar > cyClient)
  {
    InvalidateRect (hwnd, NULL, TRUE) ;
    UpdateWindow (hwnd) ;
    iLine = 0 ;
  }
  return iLine ;
}

/* Window 1: Display increasing sequence of numbers */
void Thread1 (PVOID pvoid)
{
  int     iNum = 0, iLine = 0 ;
  char    szBuffer[16] ;
  HDC     hdc ;
  PPARAMS pparams ;

  pparams = (PPARAMS) pvoid ;
  while (!pparams->bKill)
  {
    if (iNum < 0)   iNum = 0 ;
    iLine = CheckBottom (pparams->hwnd,   pparams->cyClient,
                                 pparams->cyChar, iLine) ;
    wsprintf (szBuffer, "%d", iNum++) ;
    hdc = GetDC (pparams->hwnd) ;
    TextOut (hdc, 0, iLine * pparams->cyChar,
                  szBuffer, strlen (szBuffer)) ;

    ReleaseDC (pparams->hwnd, hdc) ;

    iLine++ ;
  }
  _endthread ();
}

LRESULT APIENTRY WndProc1 (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM
 lParam)
{
  static PARAMS params ;

  switch (iMsg)
  {
    case WM_CREATE :
      params.hwnd = hwnd ;
      params.cyChar = HIWORD (GetDialogBaseUnits ()) ;
```

```
#ifndef __WATCOMC__
     if (! _beginthread( Thread1, 0, &params ) )
#else
     if (! _beginthread( Thread1, 0, 0, &params ) )
#endif

      return 0 ;

    case WM_SIZE :
      params.cyClient = HIWORD (lParam) ;
      return 0 ;

    case WM_DESTROY :
      params.bKill = TRUE ;
      return 0 ;
  }
  return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}

/* Window 2: Display increasing sequence of prime numbers */
void Thread2 (PVOID pvoid)
{
  char    szBuffer[16] ;
  int     iNum = 1, iLine = 0, i, iSqrt ;
  HDC     hdc ;
  PPARAMS pparams ;

  pparams = (PPARAMS) pvoid ;
  while (!pparams->bKill)
  {
    do {
      if (++iNum < 0)   iNum = 0 ;
      iSqrt = (int) sqrt( (float)iNum );
      for (i = 2 ; i <= iSqrt ; i++)
        if (iNum % i == 0) break ;
    } while (i <= iSqrt) ;
    iLine = CheckBottom (pparams->hwnd,   pparams->cyClient,
                                pparams->cyChar, iLine) ;
    wsprintf (szBuffer, "%d", iNum) ;
    hdc = GetDC (pparams->hwnd) ;
    TextOut (hdc, 0, iLine * pparams->cyChar,
                    szBuffer, strlen (szBuffer)) ;
    ReleaseDC (pparams->hwnd, hdc) ;
    iLine++ ;
  }
  _endthread () ;
}

LRESULT APIENTRY WndProc2 (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM
 lParam)
{
  static PARAMS params ;

  switch (iMsg)
```

```
  {
    case WM_CREATE :
      params.hwnd = hwnd ;
      params.cyChar = HIWORD (GetDialogBaseUnits ()) ;

#ifndef __WATCOMC__
     if (! _beginthread( Thread2, 0, &params) )
#else
     if (! _beginthread( Thread2, 0, 0, &params ) )
#endif

      return 0 ;

    case WM_SIZE :
      params.cyClient = HIWORD (lParam) ;
      return 0 ;

    case WM_DESTROY :
      params.bKill = TRUE ;
      return 0 ;
  }
  return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}

/* Window 3: Display increasing sequence of Fibonacci numbers */
void Thread3 (PVOID pvoid)
{
  char    szBuffer[16] ;
  int     iNum = 0, iNext = 1, iLine = 0, iTemp ;
  HDC     hdc ;
  PPARAMS pparams ;

  pparams = (PPARAMS) pvoid ;
  while (!pparams->bKill)
  {
    if (iNum < 0)
    {
      iNum  = 0 ;
      iNext = 1 ;
    }
    iLine = CheckBottom (pparams->hwnd,   pparams->cyClient,
                                 pparams->cyChar, iLine) ;
    wsprintf (szBuffer, "%d", iNum) ;
    hdc = GetDC (pparams->hwnd) ;
    TextOut (hdc, 0, iLine * pparams->cyChar,
                  szBuffer, strlen (szBuffer)) ;
    ReleaseDC (pparams->hwnd, hdc) ;
    iTemp  = iNum ;
    iNum   = iNext ;
    iNext += iTemp ;
    iLine++ ;
  }
  _endthread () ;
}
```

```
LRESULT APIENTRY WndProc3 (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM
 lParam)
{
  static PARAMS params ;

  switch (iMsg)
  {
    case WM_CREATE :
      params.hwnd = hwnd ;
      params.cyChar = HIWORD (GetDialogBaseUnits ()) ;

#ifndef __WATCOMC__
     if (! _beginthread( Thread3, 0, &params) )
#else
     if (! _beginthread( Thread3, 0, 0, &params ) )
#endif

      return 0 ;

    case WM_SIZE :
      params.cyClient = HIWORD (lParam) ;
      return 0 ;

    case WM_DESTROY :
      params.bKill = TRUE ;
      return 0 ;
  }
  return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}

/* Window 4: Display circles of random radii */
void Thread4 (PVOID pvoid)
{
  HDC     hdc ;
  int     iDiameter ;
  PPARAMS pparams ;

  pparams = (PPARAMS) pvoid ;
  while (!pparams->bKill)
  {
    InvalidateRect (pparams->hwnd, NULL, TRUE) ;
    UpdateWindow (pparams->hwnd) ;
    iDiameter = rand() % (max (1, min (pparams->cxClient, pparams-
>cyClient)));
    hdc = GetDC (pparams->hwnd) ;
    Ellipse (hdc, (pparams->cxClient - iDiameter) / 2,
                  (pparams->cyClient - iDiameter) / 2,
                  (pparams->cxClient + iDiameter) / 2,
                  (pparams->cyClient + iDiameter) / 2) ;
    ReleaseDC (pparams->hwnd, hdc) ;
  }
  _endthread () ;
}
```

```
LRESULT APIENTRY WndProc4 (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM
 lParam)
{
  static PARAMS params ;

  switch (iMsg)
  {
    case WM_CREATE :
      params.hwnd = hwnd ;
      params.cyChar = HIWORD (GetDialogBaseUnits ()) ;

#ifndef __WATCOMC__
     if (! _beginthread( Thread4, 0, &params) )
#else
     if (! _beginthread( Thread4, 0, 0, &params) )
#endif

      return 0 ;

    case WM_SIZE :
      params.cxClient = LOWORD (lParam) ;
      params.cyClient = HIWORD (lParam) ;
      return 0 ;

    case WM_DESTROY :
      params.bKill = TRUE ;
      return 0 ;
  }
  return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
     }

/* Main window to create child windows */

LRESULT APIENTRY WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM
lParam)
{
  static char   *szChildClass[] = { "Child1", "Child2",
                                    "Child3", "Child4" } ;
  static HWND    hwndChild[4] ;
  static WNDPROC ChildProc[] = { WndProc1, WndProc2,
                                 WndProc3, WndProc4 } ;
  HINSTANCE      hInstance ;
  int            i, cxClient, cyClient ;
  WNDCLASS     wndclass ;

  switch (iMsg)
  {
    case WM_CREATE :
      hInstance = (HINSTANCE) GetWindowLong (hwnd, GWL_HINSTANCE) ;

      wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
      wndclass.cbClsExtra    = 0 ;
      wndclass.cbWndExtra    = 0 ;
```

```
          wndclass.hInstance     = hInstance ;
          wndclass.hIcon         = NULL ;
          wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
          wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
          wndclass.lpszMenuName  = NULL ;

          for (i = 0 ; i < 4 ; i++)
          {
            wndclass.lpfnWndProc   = ChildProc[i] ;
            wndclass.lpszClassName = szChildClass[i] ;
            RegisterClass (&wndclass) ;
            hwndChild[i] = CreateWindow (szChildClass[i], NULL,
                           WS_CHILDWINDOW | WS_BORDER | WS_VISIBLE,
                           0, 0, 0, 0, hwnd, (HMENU) i, hInstance, NULL
) ;
          }
          return 0 ;

        case WM_SIZE :
          cxClient = LOWORD (lParam) ;
          cyClient = HIWORD (lParam) ;

          for (i = 0 ; i < 4 ; i++)
            MoveWindow (hwndChild[i], (i % 2) * cxClient / 2,
                                      (i > 1) * cyClient / 2,
                                cxClient / 2, cyClient / 2, TRUE) ;
          return 0 ;

        case WM_CHAR :
          if (wParam == '\x1B')   DestroyWindow (hwnd) ;
          return 0 ;

        case WM_DESTROY :
          PostQuitMessage (0) ;
          return 0 ;
      }
      return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
    }
```

Функции *WinMain* и *WndProc* в программе MULTI2.C очень похожи на соответствующие функции в программе MULTI1.C. Функция *WndProc* регистрирует четыре класса окна для четырех окон, создает эти окна и устанавливает их размер в процессе обработки сообщения WM_SIZE. Единственное отличие в данной функции *WndProc* состоит в том, что она не устанавливает таймер и не посылает сообщений WM_TIMER.

Основное отличие программы MULT12 состоит в том, что каждая дочерняя оконная процедура создает свой поток обработки, вызывая функцию *_beginthtead* во время обработки сообщения WM_CREATE. Всего в программе MULT12 имеется пять потоков обработки, которые выполняются совместно. Первичный поток содержит оконную процедуру главного

окна и четыре оконные процедуры дочерних окон. Остальные четыре потока использует функции с именами *Threadi, Thread2* и т. д. Эти четыре потока отвечают за вывод в четырех окнах на экране.

В коде многопоточной программы RNDRCTMT не использовался третий параметр функции *_beginthread.* Этот параметр позволяет потоку, который создает другой поток, передавать информацию этому потоку в виде 32-разрядней переменной. Обычно такая переменная является указателем на структуру данных. Это дает возможность создающему и создаваемому потокам совместно владеть информацией без использования глобальных переменных. Как вы можете видеть, в программе MULTI2 нет глобальных переменных.

Параметр функции *_beginthread.* Этот параметр позволяет потоку, который создает другой поток, передавать информацию этому потоку в виде 32-разрядной переменной. Обычно такая переменная является указателем на структуру данных. Это дает возможность создающему и создаваемому потокам совместно владеть информацией без использования глобальных переменных. Как вы можете видеть, в программе MULTI2 нет глобальных переменных.

В программе MULTI2 определена структура с именем PARAMS (в начале программы) и указатель на эту структуру — PPARAMS. В этой структуре пять полей: описатель окна, ширина и высота окна, высота символа и булева переменная с именем *bKill.* Последнее поле позволяет создающему потоку передавать создаваемому потоку информацию о том, в какой момент времени он должен закончить работу.

Рассмотрим функцию *WndProc1* — оконную процедуру дочернего окна, которое выводит последовательность увеличивающихся на единицу чисел. Оконная процедура стала очень простой. Единственная локальная переменная — это структура PARAMS. Во время обработки сообщения WM_CREATE устанавливаются значения полей *hwnd* и *cyChar* этой структуры, и вызывается функция *_beginthread* для создания нового потока, использующего функцию *Thread1,* и передавая в качестве параметра указатель на эту структуру. Во время обработки сообщения WM_SIZE функция *WndProcI* устанавливает значение поля *cyClient* этой структуры, а во время обработки сообщения WM_DESTROY устанавливает значение поля *bKill* в TRUE. Выполнение функции *Thread1* завершается при вызове функции *_endthread.*

Это не является строго необходимым, поскольку поток уничтожается после выхода из функции потока. Однако, функция *_endthread* является полезной при выходе из потока, сидящего глубоко в иерархии потоков обработки.

Функция *Thread1* осуществляет рисование в окне, и она выполняется совместно с другими четырьмя потоками в процессе. Функция получает

указатель на структуру PARAMS и выполняется циклически, используя цикл *while,* проверяя каждый раз значение поля *bKill.* Если в поле *bKill* содержится значение FALSE, то функция выполняет те же действия, что и во время обработки сообщения WM_TIMER в программе MULTI1 — формирование числа, получение описателя контекста устройства и отображение числа на экране с использованием функции *TextOut.*

Как вы увидите, когда запустите программу MULTI2 на выполнение под Windows 95. обновление окон происходит значительно быстрее, чем в программе MULTI1. иллюстрируя тем самым, что программа использует мощность процессора более эффективно. Существует еще одно отличие между программами MULTI1 и MULTI2: обычно при перемещении окна или изменении его размеров оконная процедура по умолчанию входит в модальный цикл, и весь вывод в окно приостанавливается. В программе MULT12 вывод в этом случае продолжается.

Может показаться, что программа MULTI2 не настолько убедительна, как могла бы быть. Чтобы увидеть, чего нам действительно удалось достичь, давайте рассмотрим некоторые "недостатки" многопоточности в программе MULTI2.C, взяв для примера функции *WndProcI* и *Thread 1.*

Функция *WndProc1* выполняется в главном потоке программы MULTI2, а функция *Thread1* выполняется параллельно с ней. Моменты времени, когда Windows переключается между этими двумя потоками, являются переменными и заранее непредсказуемыми. Предположим, что функция *Thread1* активна и только что выполнила код, проверяющий, имеет ли поле *bKill* структуры PARAMS значение TRUE. Оно не равно TRUE, а Windows 95 переключает управление на главный поток, в котором пользователь завершает приложение. Функция *WndProc1* получает сообщение WM_DESTROY и устанавливает значение поля *bKill* равным TRUE. Но это уже слишком поздно. Внезапно операционная система переключает управление на функцию *Thread1,* и эта функция пытается получить описатель контекста устройства уже несуществующего окна.

Оказывается, что здесь нет никакой проблемы. Windows достаточно устойчива, и графические функции просто не выполняются, не вызывая при этом никаких проблем.

Правильная техника программирования многопоточных приложений включает использование синхронизации потоков (thread synchronization) (и в частности, критических разделов, critical sections). Синхронизацию потоков мы рассмотрим более детально позже. В нескольких словах, критические разделы ограничиваются вызовами функций *EnterCriticalSection* и *LeawCrMcalSection.* Если один поток входит в критический раздел, то другой поток уже не может войти в него. Для второго потока вызов функции *EnterCriticalSection* приводят к приостановке выполнения внутри этой

функции до тех пор, пока первый поток не вызовет функцию *LeaveCriticalSection.*

Другой возможной проблемой в программе MULTI2 является то, что главный поток может получить сообщение WM_ERASEBKGND или WM_PAINT в то время, как вторичный поток осуществляет рисование в окне. И снова использование критических разделов могло бы помочь предотвратить какие-либо проблемы, которые могли бы возникнуть при попытке двух потоков осуществить одновременное рисование в окне. Однако, эксперименты показывают, что Windows правильно организует последовательность доступа к графическим функциям рисования. Таким образом, один поток не может рисовать в окне, пока другой поток выполняет такое же действие.

Документация Windows предупреждает об одной области, в которой графические функции не упорядочиваются. Это использование объектов GDI, таких как перья, кисти, шрифты, битовые образы и палитры. Существует возможность для одного потока разрушить объект, который в это время используется другим потоком. Решение этой проблемы состоит в использовании критических разделов. Однако, еще лучше делать так, чтобы объекты GDI не разделялись между различными потоками.

2.5 Использование функции Sleep

Выше было показано, как лучше организовать архитектуру программы, использующей многопоточность, а именно, чтобы первичный поток создавал все окна в программе, содержал все оконные процедуры этих окон и обрабатывал все сообщения. Вторичные потоки выполняют фоновые задачи или задачи, протяженные во времени.

Однако, предположим, что вы хотите реализовать анимацию во вторичном потоке. Обычно анимация в Windows осуществляется с использованием сообщения WM_TIMER. Но если вторичный поток не создает окно, то он не может получить это сообщение. А без задания определенного темпа анимация могла бы осуществляться слишком быстро.

Решение состоит в использовании функции *Sleep.* Поток вызывает функцию *Sleep* для того, чтобы добровольно отложить свое выполнение. Единственный параметр этой функции — время, задаваемое в миллисекундах. Функция *Sleep* не осуществляет возврата до тех пор, пока не истечет указанное время. В течение него выполнение потока приостанавливается и выделения для него процессорного времени не происходит (хотя очевидно, что для потока все-таки требуется какое-то незначительное время, за которое система должна определить, пора возобновлять выполнение потока или нет).

Если параметр функции *Sleep* задан равным нулю, то поток будет лишен остатка выделенного ему кванта процессорного времени.

Когда поток вызывает функцию *Sleep,* задержка на заданное время относится только к этому потоку. Система продолжает выполнять другие потоки этого и других процессов. Обычно не требуется использовать функцию *Sleep* в первичном потоке, т.к. она замедляет обработку сообщений.

2.6 Синхронизация потоков

2.6.1 Критическая секция (раздел)

В однозадачной операционной системе обычные программы не нуждаются в координации их действий. Они выполняются так, как будто они являются едиственными программами и не существует ничего, что могло бы вмешаться в то, что они делают.

Даже в многозадачной операционной системе большинство программ выполняются независимо друг от друга. Но некоторые проблемы все же могут возникнуть. Например, двум программам может понадобиться читать и писать в один файл в одно и то же время. Для таких случаев операционная система поддерживает механизм разделения файлов (shared files) и блокирования отдельных фрагментов файла (record locking).

Однако, в операционной системе, поддерживающей многопоточность, такое решение может внести путаницу и создать потенциальную опасность. Разделение данных между двумя и более потоками является общим случаем. Например, один поток может обновлять одну или более переменных, а другой может использовать эти переменные. Иногда в этой ситуации может возникнуть проблема, а иногда — нет. (Помните, что операционная система может переключать управление потоками только между инструкциями машинного кода. Если простое целое число разделяется между двумя потоками, то изменение этой переменной обычно осуществляется одной инструкцией машинного кода, и потенциальные проблемы сводятся к минимуму.)

Однако, предположим, что потоки разделяют несколько переменных или структуру данных. Часто эти сложные переменные или поля структур данных должны быть согласованными между собой. Операционная система может прерывать поток в середине процесса обновления этих переменных. В этом случае поток, который затем использует эти переменные, будет иметь дето с несогласованными данными.

В результате бы возникла коллизия, и нетрудно представить себе, как такого рода ошибка может привести к краху программы. В этой ситуации нам необходимо нечто похожее на светофор, который мог бы синхронизировать и координировать работу потоков. Таким средством и является критический раздел. Критический раздел — это блок кода, при выполнении которого поток не может быть прерван.

Имеется четыре функции для работы с критическими разделами. Чтобы их использовать, вам необходимо определить объект типа критический раздел, который является глобальной переменной типа CRITICAL_SECTION. Например,

```
CRITICAL_SECTION cs ;
```

Тип данных CRITICAL_SECTION является структурой, но ее поля используются только внутри Windows. Объект типа критический раздел сначала должен быть инициализирован одним из потоков программы с помощью функции:

```
InitializeCriticalSection (&cs);
```

Эта функция создает объект критический раздел с именем *cs*. В документации содержится следующее предупреждение: "Объект критический раздел не может быть перемещен или скопирован. Процесс также не должен модифицировать объект, а должен обращаться с ним, как с "черным ящиком"."

После инициализации объекта критический раздел поток входит в критический раздел, вызывая функцию:

```
EnterCriticaiSection (&cs) ;
```

В этот момент поток становится владельцем объекта. Два различных потока не могут быть владельцами одного объекта критический раздел одновременно. Следовательно, если один поток вошел в критический раздел, то следующий поток, вызывая функцию *EnterCriticalSection* с тем же самым объектом критический раздел, будет задержан внутри функции. Возврат из функции произойдет только тогда, когда первый поток покинет критический раздел, вызвав функцию:

```
LeaveCrifcicaiSectior. (&cs) ;
```

В этот момент второй поток, задержанный в функции *EnterCriticalSection.* станет владельцем критического раздела, и его выполнение будет возобновлено.

Когда объект критический раздел больше не нужен вашей программе, его можно удалить с помощью функции:

```
DeieteCriticalSection ;(&cs) ;
```

Это приведет к освобождению всех ресурсов системы; задействованных для поддержки объекта критический раздел.

Механизм критических разделов основан на принципе взаимного исключения (mutual exclusion). Этот термин нам еще встретится при дальнейшем рассмотрении синхронизации потоков. Только один поток может быть владельцем критического раздела в каждый конкретный момент времени. Следовательно, один поток может войти в критический раздел; установить значения полей структуры и выйти из критического раздела. Другой поток, использующий эту структуру, также мог бы войти в критический раздел перед осуществлением доступа, к полям структуры, затеют выйти из критического раздела.

Обратите внимание, что возможно определение нескольких объектов типа критический раздел, например, *cs1* и *cs2.* Если в программе имеется четыре потока, и два первых из них разделяют некоторые данные, то они могут использовать объект критический раздел, а два других потока, также разделяющих другие данные, могут использовать второй объект критический раздел.

Обратите внимание, что надо быть весьма осторожным при использовании критического раздела в главном потоке. Если вторичный поток проводит слишком, много времени в его собственном критическом разделе, то это может привести к "зависанию" главного потока на слишком большой период времени.

2.6.1 Объект Mutex

Существует одно ограничение в использовании критических разделов. Оно заключается в том, что их можно применять для синхронизации потоков только в рамках одного процесса. Но бывают случаи, когда необходимо синхронизировать действия потоков различных процессов, которые разделяют какие-либо ресурсы (например, память). Использовать критические разделы и такой ситуации нельзя. Вместо них подключаются объекты типа mutex (mutex object).

Составное слово "mutex" происходит из словосочетания "mutual exclusion", что означает взаимное исключение, и очень точно отражает назначение объектов. Мы хотим предотвратить возможность прерывания потока в программе до тех пор, пока не будет выполнено обновление или использование разделяемых данных.

Часто бывает, что вторичному потоку надо проинформировать первичный поток о том, что он завершился, или первичному потоку надо прервать работу, выполняемую вторичным потоком. Этот случай мы и рассмотрим.

В качестве гипотетической большой работы рассмотрим последовательность вычислений с плавающей точкой, известную иногда как "дикий" тест производительности. В этих вычислениях значение целого числа увеличивается на единицу в так называемой карусельной манере: сначала

число возводится в квадрат, затем из результата извлекается квадратный корень, выполняются функции *log* и *exp,* отменяющие действия друг друга, затем выполняются функции *atan* и *tan,* и наконец, просто прибавляется 1 для получения результата.

Пример 6.1 – Программа BIGJOB1

```
/*----------------------------------------
   BIGJOB1.C -- Multithreading Demo
  ----------------------------------------*/

#include <windows.h>
#include <math.h>
#include <process.h>

#define REP              1000000

#define STATUS_READY     0
#define STATUS_WORKING   1
#define STATUS_DONE      2

#define WM_CALC_DONE     (WM_USER + 0)
#define WM_CALC_ABORTED  (WM_USER + 1)

typedef struct
{
  HWND hwnd ;
  BOOL bContinue ;
} PARAMS, *PPARAMS ;

LRESULT APIENTRY WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
  static char szAppName[] = "BigJob1" ;
  HWND        hwnd ;
  MSG         msg ;
  WNDCLASS  wndclass ;


  wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
  wndclass.lpfnWndProc   = WndProc ;
  wndclass.cbClsExtra    = 0 ;
  wndclass.cbWndExtra    = 0 ;
  wndclass.hInstance     = hInstance ;
  wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
  wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
  wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
  wndclass.lpszMenuName  = NULL ;
  wndclass.lpszClassName = szAppName ;

  RegisterClass (&wndclass) ;
```

```
  hwnd = CreateWindow (szAppName, "Multithreading Demo",
                       WS_OVERLAPPEDWINDOW,
                       CW_USEDEFAULT, CW_USEDEFAULT,
                       CW_USEDEFAULT, CW_USEDEFAULT,
                       NULL, NULL, hInstance, NULL) ;
  ShowWindow (hwnd, iCmdShow) ;
  UpdateWindow (hwnd) ;
  while (GetMessage (&msg, NULL, 0, 0))
  {
    TranslateMessage (&msg) ;
    DispatchMessage (&msg) ;
  }
  return msg.wParam ;
}

void Thread (PVOID pvoid)
{
  double  A = 1.0 ;
  INT     i ;
  LONG    lTime ;
  PPARAMS pparams ;

  pparams = (PPARAMS) pvoid ;
  lTime = GetCurrentTime () ;
  for (i = 0 ; i < REP && pparams->bContinue ; i++)
          A = tan (atan (exp (log (sqrt (A * A))))) + 1.0 ;

  if (i == REP)
  {
    lTime = GetCurrentTime () - lTime ;
    SendMessage (pparams->hwnd, WM_CALC_DONE, 0, lTime) ;
  }
  else   SendMessage (pparams->hwnd, WM_CALC_ABORTED, 0, 0) ;
  _endthread () ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM
lParam)
{
  static char  *szMessage[] = { "Ready (left mouse button begins)",
                                "Working (right mouse button aborts)",
                                "%d repetitions in %ld msec" } ;
  static INT    iStatus ;
  static LONG   lTime ;
  static PARAMS params ;
  char          szBuffer[64] ;
  HDC           hdc ;
  PAINTSTRUCT   ps ;
  RECT          rect ;

  switch (iMsg)
  {
    case WM_LBUTTONDOWN :
      if (iStatus == STATUS_WORKING)
```

```
        {
          MessageBeep (0) ;
          return 0 ;
        }
        iStatus = STATUS_WORKING ;
        params.hwnd = hwnd ;
        params.bContinue = TRUE ;

#ifndef __WATCOMC__
        _beginthread (Thread, 0, &params) ;
#else
        _beginthread (Thread, NULL, 0, &params) ;
#endif

        InvalidateRect (hwnd, NULL, TRUE) ;
        return 0 ;

      case WM_RBUTTONDOWN :
        params.bContinue = FALSE ;
        return 0 ;

      case WM_CALC_DONE :
        lTime = lParam ;
        iStatus = STATUS_DONE ;
        InvalidateRect (hwnd, NULL, TRUE) ;
        return 0 ;

      case WM_CALC_ABORTED :
        iStatus = STATUS_READY ;
        InvalidateRect (hwnd, NULL, TRUE) ;
        return 0 ;

      case WM_PAINT :
        hdc = BeginPaint (hwnd, &ps) ;
        GetClientRect (hwnd, &rect) ;
        wsprintf (szBuffer, szMessage[iStatus], REP, lTime) ;
        DrawText (hdc, szBuffer, -1, &rect,
                      DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;
        EndPaint (hwnd, &ps) ;
        return 0 ;

      case WM_DESTROY :
        PostQuitMessage (0) ;
        return 0 ;
    }
    return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}
```

Это достаточно простая программа, но на ее примере можно увидеть, как она реализует подход к выполнению больших работ в многопоточной программе. Чтобы использовать программу BIGJOBI, щелкните левой кнопкой мыши в рабочей области окна. Это приведет к запуску 100 000

повторов "диких" вычислений. Когда вычисления будут закончены, затраченное время будет отображено в окне. Во время расчетов вы можете щелкнуть правой кнопкой мыши в рабочей области окна для прерывания вычислений.

Рассмотрим, как работает программа.

Оконная процедура содержит статическую переменную с именем *iStaias* (которая может быть установлена в одно из трех значений, определенных в начале программы, начинающихся со слова STATUS). Эта переменная показывает, находится ли программа в состоянии готовности к вычислениям, в процессе вычислений или вычисления завершены. Программа использует переменную *iStafus* в процессе обработки сообщения WM_PAINT для вывода на экран соответствующей строки символов в центре рабочей области.

Кроме того, оконная процедура содержит статическую структуру типа PARAMS, определенную также в начале программы, для разделения данных между оконной процедурой и вторичным потоком. В этой структуре только два поля — поле *hwnd* (описатель окна программы) и поле *bContinue,* которое является булевой переменной для указания вторичному потоку — продолжать ему вычисления или нет.

Когда вы щелкаете левой кнопкой мыши в рабочей области, оконная процедуры присваивает переменной *iStatus* значение STATUS_WORKING, а также заполняет оба поля структуры PARAMS. В поле *hwnd* заносится, очевидно, значение описателя окна, а в поле *bContinue* заносится значение TRUE.

Затем оконная процедура создает вторичный поток, вызывая функцию *_begwfiread.* Функция вторичного потока с именем *Thread* начинается с вызова функции *GetCurrentTime* для получения значения времени в миллисекундах, прошедшего с момента запуска Windows. Затем она входит в цикл *for* для выполнения "диких" вычислений. Обратите внимание, что поток может выйти из цикла, если в какой-либо момент в поле *bContinue* окажется значение FALSE.

После выполнения цикла *for* функция потока проверяет, действительно ли выполнено 100 000 повторов вычислений. Если да, то она снова вызывает функцию G*ietCJurrenilime,* чтобы определить затраченное на вычисления время, а затем использует функцию *SendMessage* для отправки оконной процедуре определенного в программе сообщения WM_CALC_DONE с указанием затраченного времени в параметре *lParam.* Если вычисления были прерваны (в том случае, когда поле *bContinue* структуры PARAMS было установлено в FALSE во время выполнения цикла), то поток посылает оконной процедуре сообщение WM_CALC_ABORTED. Затем поток красиво завершается вызовом функции *_endthread.*

Внутри оконной процедуры поле *bContinue* структуры PARAMS принимает значение FALSE, когда вы щелкаете правой кнопкой мыши в рабочей области окна. Таким образом происходит прерывание вычислений до их завершения.

Оконная процедура обрабатывает сообщение WM_CALC_DONE, сначала сохраняя затраченное на вычисления время. Обработка сообщений WM_CALC_DONE или WM_CALC_ABORTED продолжается вызовом функции `InvalidateRect` для того, чтобы сгенерировать сообщение WM_PAINT и отобразить на экране новую строку текста в рабочей области.

Неплохо включить в программу заготовку (такую как поле *bContinue* структуры PARAMS), позволяющую потоку завершиться "чисто". Функция *KillThread* могла бы быть использована в том случае, когда "чистое" завершение невозможно. Причина может состоять в том, что потоки имеют возможность захватывать ресурсы, например, память. Если эта память не

освобождается при завершении потока, то она продолжает оставаться захваченной. Потоки – не процессы: захваченные ресурсы разделяются между всеми потоками процесса, и поэтому они не освобождаются автоматически при завершении потока. Хороший стиль программирования предписывает, чтобы поток освобождал все захваченные им ресурсы.

Обратите внимание также на то, что третий поток может быть создан, пока второй еще выполняется. Это может произойти, если Windows переключит управление со второго потока на первый между вызовами функций *SendMessage* и *_endthread.* В этом случае оконная процедура создаст новый поток в ответ на щелчок мыши. Здесь это не является проблемой, но она может возникнуть в одном из ваших собственных приложений, и тогда вам следует использовать критические разделы для того, чтобы избежать коллизий между потоками.

2.6.3 Объект Событие

Программа BIGJOBI создает поток каждый раз, когда необходимо произвести "дикие" вычисления; поток завершается после выполнения вычислений.

Альтернативным вариантом является сохранение потока готовым к выполнению на протяжении всего времени работы программы, и приведение его в действие только при необходимости. Это идеальный случай для применения объекта Событие (event object).

Объект Событие может быть либо свободным (signaled) или установленным (set), либо занятым (non-signaled) или сброшенным (reset). Вы можете создать объект Событие с помощью функции:

```
hEvent = CreateEvent (&sa, fManual, finitial, pszName) ;
```

Первый параметр (указатель на структуру типа SECURITY_ATTRIBUTES) и последний параметр (имя объекта событие) имеют смысл только в том случае, когда объект Событие разделяется между процессами. В случае с одним процессом эти параметры обычно имеют значение NULL. Установите значение параметра *fInitial* равным TRUE, если вы хотите, чтобы объект Событие был изначально свободным, или равным FALSE, чтобы он был занятая. Параметр *fManual* будет описан несколько позже.

Для того чтобы сделать свободным существующий объект Событие, вызовите функцию:

```
SetEvent (hEvent);
```

Чтобы сделать объект Событие занятым, вызовите функцию:

```
ResetEvent (hEvent);
```

Обычно программа вызывает функцию:

```
WaitForSingleObject (hEvent, dwTimeOut);
```

где нарой параметр имеет значение INFINITE. Возврат из функции происходит немедленно, если объект Событие в настоящее время свободен. В противном случае поток будет приостановлен в функции до тех пор, пока событие не станет свободным. Вы можете установить значение тайм-аута во втором параметре, задав его величину в миллисекундах. Тогда возврат из функции произойдет, когда объект Событие станет свободным или истечет тайм-аут.

Если параметр *fManual* при вызове функции *CreateEvent* имеет значение FALSE, то объект Событие автоматически становится занятым, когда осуществляется возврат из функции *WaitForSingleObject.* Эта особенность обычно позволяет избежать использования функции *ResetEvent.*

Рассмотрим программу BIGJOB2.

```
/*----------------------------------------
   BIGJOB2.C -- Multithreading Demo
  ----------------------------------------*/

#include <windows.h>
#include <math.h>
#include <process.h>

HANDLE hTh = 0;
int    IDTh = 0;

#define REP             1000000
```

```
#define STATUS_READY     0
#define STATUS_WORKING   1
#define STATUS_DONE      2

#define WM_CALC_DONE     (WM_USER + 0)
#define WM_CALC_ABORTED  (WM_USER + 1)

typedef struct
{
  HWND   hwnd ;
  HANDLE hEvent ;
  BOOL   bContinue ;
} PARAMS, *PPARAMS ;

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;


int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
  static char szAppName[] = "BigJob2";
  HWND        hwnd ;
  MSG         msg ;
  WNDCLASS  wndclass ;


  wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
  wndclass.lpfnWndProc   = WndProc ;
  wndclass.cbClsExtra    = 0 ;
  wndclass.cbWndExtra    = 0 ;
  wndclass.hInstance     = hInstance ;
  wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
  wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
  wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
  wndclass.lpszMenuName  = NULL ;
  wndclass.lpszClassName = szAppName ;

  RegisterClass (&wndclass) ;
  hwnd = CreateWindow (szAppName, "Multithreading Demo",
                       WS_OVERLAPPEDWINDOW,
                       CW_USEDEFAULT, CW_USEDEFAULT,
                       CW_USEDEFAULT, CW_USEDEFAULT,
                       NULL, NULL, hInstance, NULL) ;
  ShowWindow (hwnd, iCmdShow) ;
  UpdateWindow (hwnd) ;
  while (GetMessage (&msg, NULL, 0, 0))
  {
    TranslateMessage (&msg) ;
    DispatchMessage (&msg) ;
  }
  return msg.wParam ;
}

void Thread (PVOID pvoid)
```

```
{
  double  A = 1.0 ;
  INT     i ;
  LONG    lTime ;
  PPARAMS pparams ;

  pparams = (PPARAMS) pvoid ;
  while (TRUE)
  {
    WaitForSingleObject (pparams->hEvent, INFINITE) ;
    lTime = GetCurrentTime () ;
    for (i = 0 ; i < REP && pparams->bContinue ; i++)
       A = tan (atan (exp (log (sqrt (A * A))))) + 1.0 ;

    if (i == REP)
    {
      lTime = GetCurrentTime () - lTime ;
      SendMessage (pparams->hwnd, WM_CALC_DONE, 0, lTime) ;
    }
    else   SendMessage (pparams->hwnd, WM_CALC_ABORTED, 0, 0) ;
  }
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM
lParam)
{
  static char  *szMessage[] = { "Ready (left mouse button begins)",
                                "Working (right mouse button aborts)",
                                "%d repetitions in %d msec" } ;
  static HANDLE hEvent ;
  static INT    iStatus ;
  static LONG   lTime ;
  static PARAMS params ;
  char          szBuffer[64] ;
  HDC           hdc ;
  PAINTSTRUCT   ps ;
  RECT          rect ;

  switch (iMsg)
  {
    case WM_CREATE :
      hEvent = CreateEvent (NULL, FALSE, FALSE, NULL) ;
      params.hwnd = hwnd ;
      params.hEvent = hEvent ;
      params.bContinue = FALSE ;
/*
#ifndef __WATCOMC__
     if (! _beginthread( Thread, 0, &params ) )
#else
     if (! _beginthread( Thread, NULL, 0, &params ) )
#endif
*/
      hTh = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)Thread, &par
ams,0, (LPDWORD)&IDTh );
```

```
      if (! hTh)
        MessageBox(NULL,"CreateThread failed !","",MB_OK);
      return 0 ;

    case WM_LBUTTONDOWN :
      if (iStatus == STATUS_WORKING)
      {
        MessageBeep (0) ;
        return 0 ;
      }
      iStatus = STATUS_WORKING ;
      params.bContinue = TRUE ;
      SetEvent (hEvent) ;
      InvalidateRect (hwnd, NULL, TRUE) ;
      return 0 ;

    case WM_RBUTTONDOWN :
      params.bContinue = FALSE ;
      return 0 ;

    case WM_CALC_DONE :
      lTime = lParam ;
      iStatus = STATUS_DONE ;
      InvalidateRect (hwnd, NULL, TRUE) ;
      return 0 ;

    case WM_CALC_ABORTED :
      iStatus = STATUS_READY ;
      InvalidateRect (hwnd, NULL, TRUE) ;
      return 0 ;

    case WM_PAINT :
      hdc = BeginPaint (hwnd, &ps) ;
      GetClientRect (hwnd, &rect) ;
      wsprintf (szBuffer, szMessage[iStatus], REP, lTime) ;
      DrawText (hdc, szBuffer, -1, &rect,
                          DT_SINGLELINE | DT_CENTER | DT_VCENTER) ;
      EndPaint (hwnd, &ps) ;
      return 0 ;

    case WM_DESTROY :
/*      _endthread(); */
      if ( hTh)
      {
        TerminateThread(hTh,0);
        CloseHandle(hTh);
      }
      PostQuitMessage (0) ;
      return 0 ;
  }
  return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}
```

Оконная процедура обрабатывает сообщение WM_CREATE, создавая событие с автоматическим сбросом и инициализирует его как занятое, а затем создает поток.

Функция *Thread* входит в бесконечный цикл *while,* но при этом вызывает функцию *WaitForSingleObject* в начале цикла (обратите внимание, что структура PARAMS имеет третье поле, содержащее описатель объекта событие). Поскольку событие изначально занято, поток задерживается в функции. Щелчок левой клавишей мыши заставляет оконную процедуру вызвать функцию *SetEvent.* Это освобождает второй поток, находящийся в ожидании внутри функции *WaitForSingleObject,* и он начинает выполнять "дикие" вычисления. После окончания очередного цикла поток снова вызывает функцию *WaitForSingleObject,* но при этом объект Событие уже оказывается в занятым состоянии с момента предыдущего вызова этой функции. Следовательно, поток задерживается до тех пор, пока не будет произведен следующий щелчок левой клавишей мыши.

В остальном программа BIGJOB2 является такой же, как и программа BIGJOB1.

2.7 Локальная память потока

Глобальные переменные в многопоточных программах (так же как и любая выделенная память) разделяются между всеми потоками в программе. Локальные статические переменные функций также разделяются между всеми потоками, использующими эту функцию. Локальные автоматические переменные в функции являются уникальными для каждого потока, потому что они хранятся в стеке, а каждый поток имеет свой собственный стек.

Может возникнуть необходимость иметь постоянную область памяти, уникальную для каждого потока. Например, функция *strtok* языка С, которая уже упоминалась в этой главе, требует такого типа память. Нет сомнений, что С его не поддерживает. В Windows имеется четыре функции, поддерживающие эту память, которая называется локальной памятью потока (thread local storage, TLS).

Теперь рассмотрим, как она работает. Во-первых, определим структуру, содержащую все данные, которые должны быть уникальны для потоков, например:

```
typedef struct
{int a ; int b;}
DATA, *PDATA ;
```

Первичный поток вызывает функцию *TlsAlloc для* получения значения индекса:

```
dwTlsIndex = TlsAiloc ();
```

Он может храниться в глобальной переменной или может быть передан функции потока в параметре-структуре.

Функция потока начинается с выделения памяти для структуры данных и с вызова функции *TlsSetValue,* используя индекс, полученный ранее.

```
TlsSetValue (dwTlsIndex, GlobalAlloc (GPTR, sizeof (DATA;)) ;
```

Это действие устанавливает соответствие указателя с конкретным потоком и конкретным индексом в потоке. Теперь, любая функция, которой нужно использовать этот указатель (включая саму базовую функцию потока), может использовать код, подобный такому:

```
PDATA pdata ;
[другие строки программы]
pdata = (PDATA) TlsGetValue (dwTIsIndex) ;
```

Теперь она может изменять значения `pdata->a` и `pdata->b`. Перед завершением функции потока необходимо освободить захваченную память:

```
GlobalFree (TlsGetValue (dwTlsIndex) );
```

Когда все потоки, использующие эти данные, будут завершены, первичный поток освобождает индекс:

```
TlsFree (dwTIsIndex);
```

Этот процесс может поначалу вас смутить, поэтому, может быть, было бы полезно посмотреть как организована локальная память потока. Во-первых, функция *TlsAlloc* могла бы просто выделить блок памяти (длиной 0 байт) и вернуть значение индекса, который является указателем на этот блок. Каждый раз при вызове функции *TlsSetValue* с этим индексом блок памяти увеличивается на 8 байт с помощью функции *GlobalAlloc.* В этих 8 байтах хранятся идентификатор потока, вызывающего функцию, полученный с помощью функции *GetCurrentThreadID,* и указатель, переданный функции *TlsSetValue.* Функция *TlsGetValue* просто использует идентификатор потока для поиска в таблице, а затем возвращает указатель. Функция *TlsFree* освобождает блок памяти.

3 Контрольные вопросы

1) Опишите модель мнгозадачности в ОС Windows.
2) Как создать новый поток?
3) Что такое mutex?
4) Как работает критическая секция?
5) Как создать событие?

6) Какие проблемы возникают при написании сногопоточных приложений.

4 Задание

1) Изучить описание лабораторной работы.
2) Составить приложение Windows, использующее многозадачность в свойей работе.
3) Отладить и протестировать полученную программу.
4) Оформить отчёт.

Лабораторная работа № 13. Динамически подключаемые библиотеки

1 Цель работы

Изучить процесс создания динамических библиотек и получить навыки их использования.

2 Кртакая теория

Динамически подключаемые библиотеки (DLL, или динамические библиотеки или библиотеки динамической компоновки, или модули библиотек) являются одним из наиболее важных структурных элементов Windows. Большинство файлов, из которых состоит Windows, представляют собой либо программные модули, либо модули динамически подключаемых библиотек. Большая часть принципов, относящихся к написанию программ, вполне подходит и для написания библиотек, но есть несколько важных отличий.

2.1 Основы библиотек

Как известно, Windows-программа представляет собой исполняемый файл, который обычно создает одно или более окон, а для получения данных от пользователя использует цикл обработки сообщений. Динамически подключаемые библиотеки, как правило, непосредственно не выполняются и обычно не получают сообщений. Они представляют собой отдельные файлы с функциями, которые вызываются программами и другими динамическими библиотеками для выполнения определенных задач. Динамически подключаемая библиотека активизируется только тогда, когда другой модуль вызывает одну из функций, находящихся в библиотеке.

Термин *динамическое связывание* (dinamic linking) относится к процессам, которые Windows использует для того, чтобы связать вызов функции в одном из модулей с реальной функцией из модуля библиотеки. *Статическое связывание* (static linking) имеет место в процессе создания программы, когда для создания исполняемого (.EXE) файла связываются воедино разные объектные (.OBJ) модули, файлы библиотек (.LIB) и, как

правило, скомпилированные файлы описания ресурсов (.RES). В отличие от этого, динамическое связывание имеет место во время выполнения программы.

Некоторые динамически подключаемые библиотеки (например, файл шрифтов) содержат только ресурсы (resource only). В них содержатся только данные (обычно в виде ресурсов), и нет текстов программ. Таким образом, одной из целей существования динамически подключаемых библиотек должно быть обеспечение функциями и ресурсами, которые можно использовать во многих, совершенно разных программах. В традиционной операционной системе, только сама операционная система содержит программы, которые для решения каких-то задач могут вызывать другие программы. В Windows принцип вызова одним модулем функции из другого модуля, распространен на всю операционную систему. Это приводит к тому, что когда вы пишете динамически подключаемую библиотеку — вы пишете расширение Windows. Можете считать динамически подключаемые библиотеки (включая те, которые составляют Windows) дополнением вашей программ

Хотя модуль динамически подключаемой библиотеки может иметь любое расширение (например, .EXE или .FON), стандартным расширением, принятым в Windows, является .DLL. Только те динамически подключает библиотеки, которые имеют расширение .DLL, Windows загрузит автоматически. Если файл имеет другое расширение, то программа должна загрузить модуль библиотеки явно. Для этого используется функция *LoadLibrary* к *LoadLibraryEx.*

Как правило, наибольший смысл динамически подключаемые библиотеки приобретают в контексте большого приложения. Например, предположим мы написали большой пакет приложений для расчетов под Windows, содержащий несколько разных программ. Обычно в таких программах используются некоторые общие подпрограммы. Эти подпрограммы можно поместить в обычную объектную библиотеку (с расширением .LIB) и добавить ее к каждому программному модулю при компоновке программы с помощью компоновщика LINK. Но такой подход избыточен, поскольку в этом случае в каждой программе пакета будет находиться одинаковый код подпрограмм, выполняющих одни и те же задачи. Более того, при изменении одной из подпрограмм в библиотеке, необходимо перекомпоновать все программы, в состав которых входит измененная подпрограмма. Однако, если поместить эти общеиспользуемые подпрограммы в динамически подключаемую библиотеку, то обе проблемы будут решены. Только в одном библиотечном модуле нужно содержать необходимые всем программам подпрограммы (таким образом, для файлов требуется меньше дискового пространства, а, при одновременном запуске двух или более

приложений, меньше оперативной памяти), и можно изменить библиотечный модуль без перекомпоновки программ.

2.2 Виды библиотек

Путаница, связанная с использованием динамически подключаемых библиотек, является результатом появления слова "библиотека" в нескольких разных контекстах. Помимо динамически подключаемых библиотек, оно упоминается в сочетаниях "объектная библиотека" и "библиотека импорта".

Объектная библиотека – это файл с расширением .lib. в котором содержится код, добавляемый к коду программы, находящемуся в файле с расширением .exe, когда идет процесс компоновки программы. Например, в Microsoft Visual C++ обычная объектная библиотека времени выполнения, которая при компоновке присоединяется к вашей программе, называется libc.lib.

Библиотека импорта представляет собой особую форму файла объектной библиотеки. Также как объектные библиотеки, библиотеки импорта имеют расширение .lib и используются компоновщиком для разрешения ссылок на функции в исходном коде программы. Однако в библиотеках импорта программного кода нет. Вместо них в библиотеках импорта находится информация, необходимая компоновщику для установки таблицы ссылок внутри файла .exe, предназначенной для динамического связывания. Файы kernel32.lib, user32.lib и gdi32.lib, поставляемые с компилятором Microsoft, являются библиотеками импорта функций Windows. Если в программе вызывается функция *Rectangle,* gdi32.lib сообщает компоновщику, что данная функция находится в динамически подключаемой библиотеке gdi32.dll. Эта информация попадает в файл .EXE, следовательно Windows может выполнить динамическое связывание с библиотекой gdi32.dll во время выполнения программы.

Объектные библиотеки и библиотеки импорта используются только при разработке программы. Динамически подключаемые библиотеки используются во время выполнения программы. Динамически подключаемая библиотека должна находиться на диске, когда выполняется программа, использующая эту библиотеку. Если операционной системе Windows требуется загрузить модуль динамически подключаемой библиотеки перед запуском программы, для которой этот модуль необходим, то файл библиотеки должен храниться в том же каталоге, что и файл .exe программы, или в текущем каталоге, или в системном каталоге Windows, или в каталоге. доступном из переменной PATH окружения. (Именно в таком порядке в каталогах происходит поиск.)

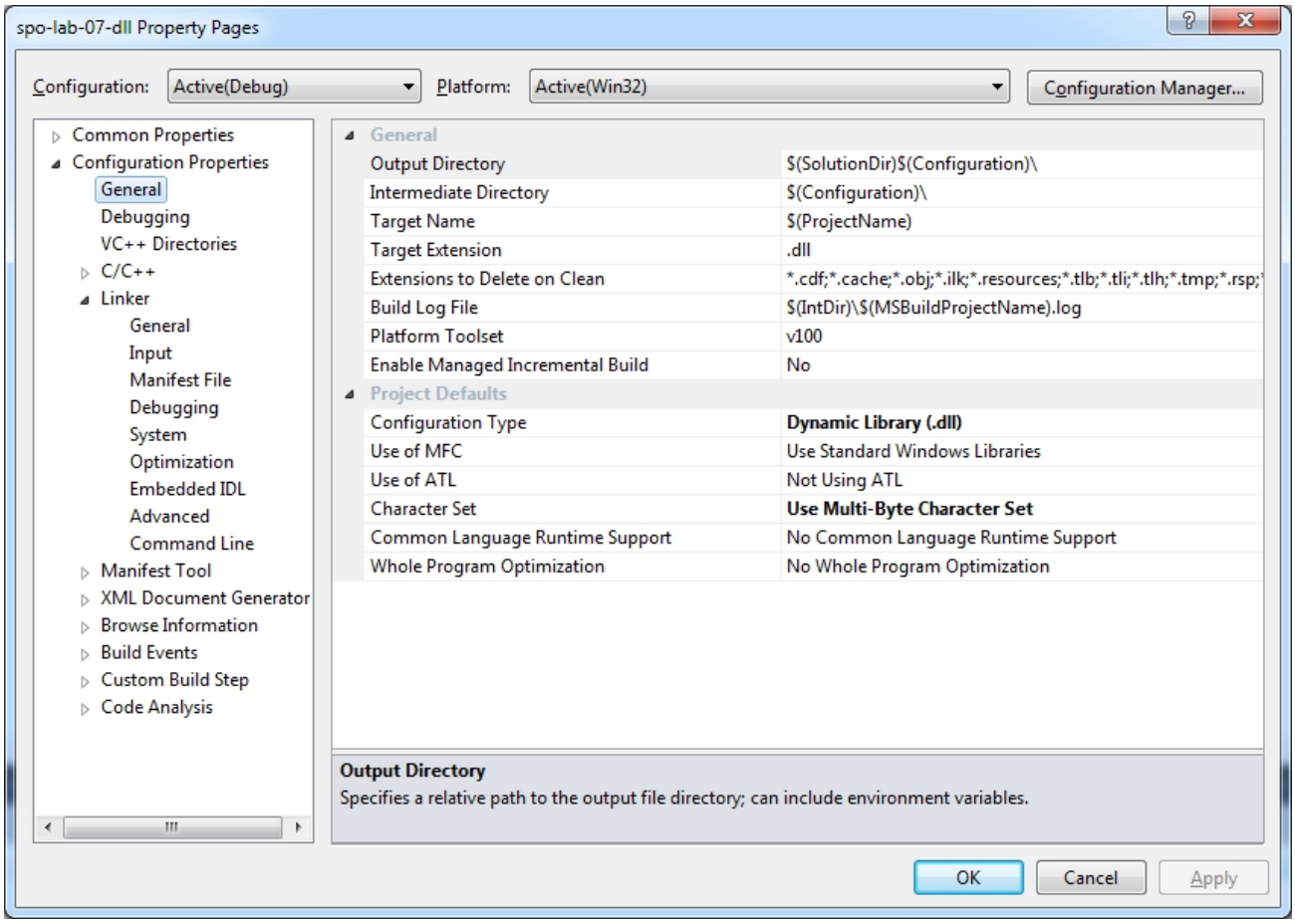
2.3 Пример простой DLL

Как обычно, начнем с простого примера. Ниже представлен исходный текст программы динамически подключаемой библиотеки, которая называется EDRLIB, и содержит одну функцию. Символы "EDR" в этом названии означают "простая подпрограмма рисования" (easy drawing routine). Вы можете легко добавлять в эту библиотеку другие функции, упрощающие процесс рисования в приложениях. Единственной функцией библиотеки EDRLIB является функция *EdrCenterText,* которая просто помещает в центр прямоугольника оканчивающуюся нулем строку текста, так же, как функция *DrawText,* но без такого количества параметров.

```
/* EDRLIB.H */
#ifdef __cplusplus
extern "C" {
#endif

#ifdef __WATCOMC__
#define EXPORT __export __stdcall
#else
#define EXPORT __declspec(dllexport) __stdcall
#endif

BOOL EXPORT EdrCenterText (HDC hdc, PRECT prc, PSTR pString);

#ifdef __cplusplus
}
#endif


/*-------------------------------------------------
   EDRLIB.C -- Easy Drawing Routine Library module
  -------------------------------------------------*/
#include <windows.h>
#include <string.h>
#include "edrlib.h"

int WINAPI DllMain (HINSTANCE hInstance, DWORD fdwReason, PVOID pvRese
rved)
{
  return TRUE ;
}

BOOL EXPORT EdrCenterText (HDC hdc, PRECT prc, PSTR pString)
{
  int  iLength ;
  SIZE size ;

  iLength = strlen (pString) ;
  GetTextExtentPoint32 (hdc, pString, iLength, &size) ;
  return TextOut (hdc, (prc->right - prc->left - size.cx) / 2,
                       (prc->bottom - prc->top - size.cy) / 2,
```

```
                        pString, iLength) ;
}
```

Как можно заметить, в настройке пректа имеется два отличия от тех, которые использовались ранее. Во-первых, в парамантах компоновщика установлен тип DLL (рисунок 7.1). Этот параметр сообщает компоновщику, что результирующий файл должен быть динамически подключаемой библиотекой. Кроме того, вместо создания файла с расширением .EXE, создается файл edrlib.dll.



Вначале файла edrlib.h определяется макроконстанта EXPORT:

```
#define EXPORT extern "C" __declspec (dllexport)
```

Использование ключевого слова EXPORT при определении функции в вашей динамически подключаемой библиотеке сообщит компоновщику, что функции доступны для использования другими программами. Функция *EdrCenterText* определяется в заголовочном файле с помощью ключевого слова EXPORT. Кроме этого, точно так же, как и оконная процедура, эта функция определяется с помощью константы CALLBACK.

Файл edrlib.c также несколько отличается от обычных файлов на языке C для Windows. В нем вместо функции *WinMain* имеется функция *DllMain*. Эта функция используется для выполнения инициализации и деинициализации, о чем будет рассказано позже в этой главе. В настоящий момент все, что нам необходимо — это чтобы ее возвращаемым значением было TRUE. И наконец, в файле edrlib.c имеется функция *EdrCenterText*.

После компиляции создается динамически подключаемая библиотека edrlib.dll.

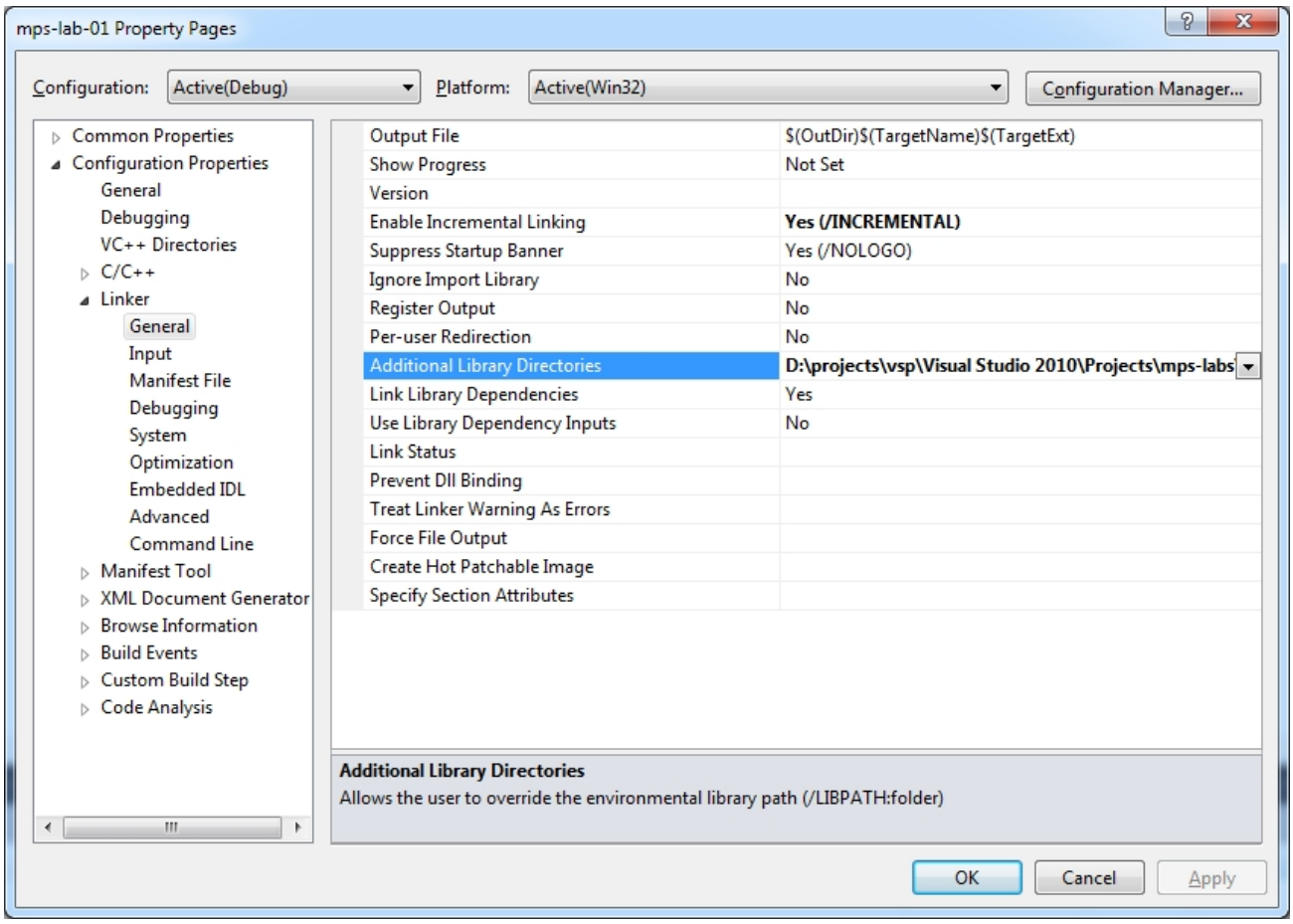
Кроме этого файла появляются еще два новых файла. Библиотека импорта edrlib.lib и библиотека экспорта edrlib.exp — это побочный эффект процесса компоновки. Ее можно удалить.

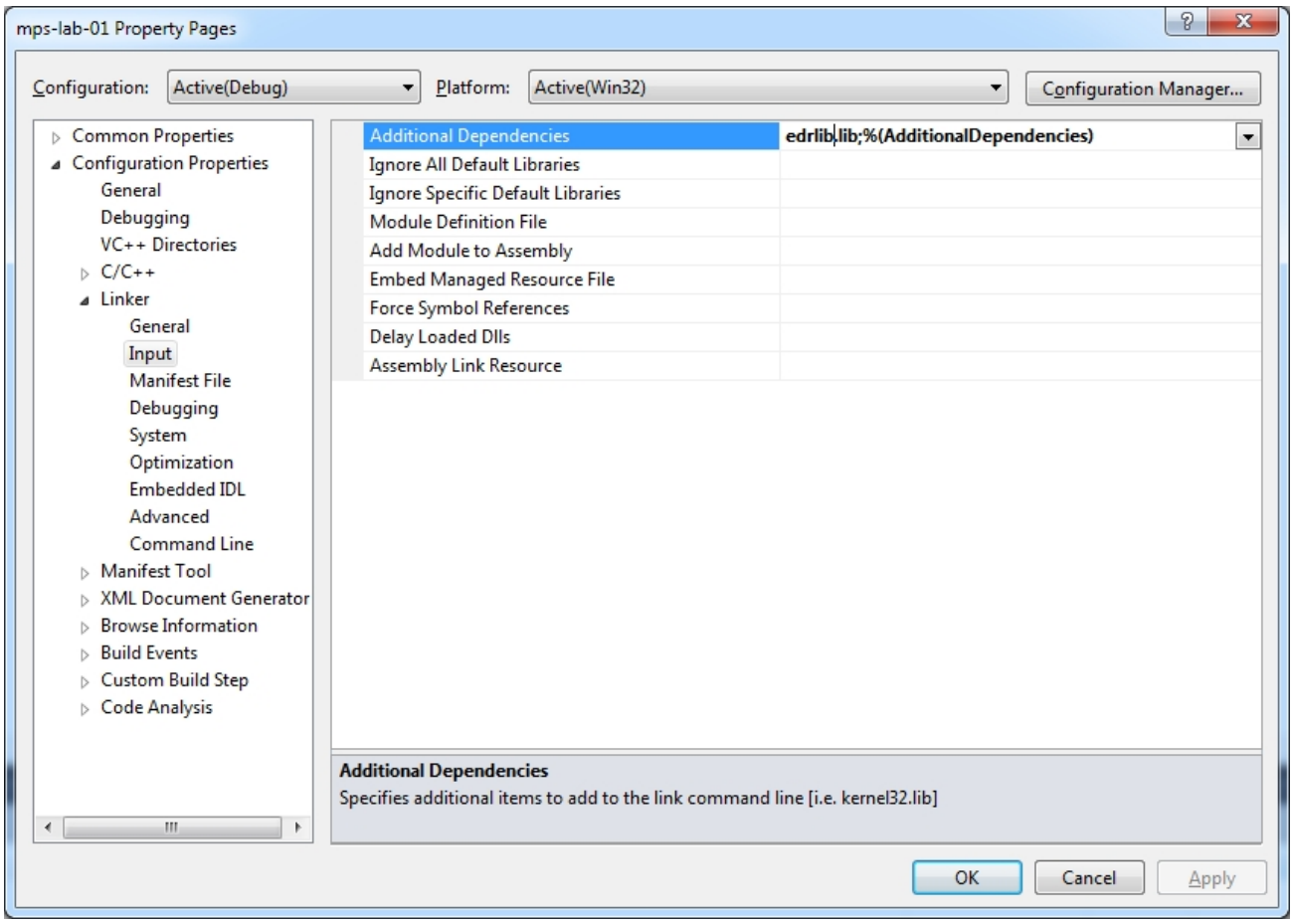
Попытаемся проверить, работает ли библиотека. Программа edrtest, представленная ниже, использует библиотеку edrlib.dll для вывода в центр своей рабочей области строки текста.

```
/*------------------------------------------------------
   EDRTEST.C -- Program using EDRLIB dynamic link library
  ------------------------------------------------------*/

#include <windows.h>
#include "edrlib.h"

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
  static char  szAppName[] = "StrProg" ;
  HWND         hwnd ;
  MSG          msg ;
  WNDCLASS   wndclass ;

/*  wndclass.cbSize        = sizeof (wndclass) ; */
  wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
  wndclass.lpfnWndProc   = WndProc ;
  wndclass.cbClsExtra    = 0 ;
  wndclass.cbWndExtra    = 0 ;
  wndclass.hInstance     = hInstance ;
  wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
  wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
  wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
  wndclass.lpszMenuName  = NULL ;
  wndclass.lpszClassName = szAppName ;
/*  wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ; */
  RegisterClass (&wndclass) ;
  hwnd = CreateWindow (szAppName, "DLL Demonstration Program",
                       WS_OVERLAPPEDWINDOW,
                       CW_USEDEFAULT, CW_USEDEFAULT,
                       CW_USEDEFAULT, CW_USEDEFAULT,
                       NULL, NULL, hInstance, NULL) ;
```

```
  ShowWindow (hwnd, iCmdShow) ;
  UpdateWindow (hwnd) ;
  while (GetMessage (&msg, NULL, 0, 0))
  {
    TranslateMessage (&msg) ;
    DispatchMessage (&msg) ;
  }
  return msg.wParam ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM
lParam)
{
     HDC         hdc ;
     PAINTSTRUCT ps ;
     RECT        rect ;

  switch (iMsg)
  {
    case WM_PAINT :
      hdc = BeginPaint (hwnd, &ps) ;
      GetClientRect (hwnd, &rect) ;
      EdrCenterText (hdc, &rect, "This string was displayed by a DLL")
 ;
      EndPaint (hwnd, &ps) ;
      return 0 ;

    case WM_DESTROY :
      PostQuitMessage (0) ;
      return 0 ;
  }
  return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}
```

Эта программа выглядит как обычная программа для Windows, чем она, в сущности, и является. Тем не менее, обратите внимание, что в файл edrtest.c с помощью инструкции *include* включен заголовочный файл edrlib.h, в котором определяется функция *EdrCenterText,* вызываемая программой при обработке сообщения WM_PAINT. Также необходимо, чтобы в настройках для компоновки был включен файл edrlib.lib и указагн путь к ней (рисунок 7.2 и 7.3). Эта библиотека импорта обеспечивает компоновщик информацией, необходимой для разрешения ссылки на файл edrlib.dll в файле edrtest.exe.


mps-lab-01 Property Pages
Configuration: Active(Debug)
Platform: Active(Win32)
Configuration Manager...
Common Properties
Configuration Properties
General
Debugging
VC++ Directories
C/C++
Linker
General
Input
Manifest File
Debugging
System
Optimization
Embedded IDL
Advanced
Command Line
Manifest Tool
XML Document Generator
Browse Information
Build Events
Custom Build Step
Code Analysis
Output File $(OutDir)$(TargetName)$(TargetExt)
Show Progress Not Set
Version
Enable Incremental Linking Yes (/INCREMENTAL)
Suppress Startup Banner Yes (/NOLOGO)
Ignore Import Library No
Register Output No
Per-user Redirection No
Additional Library Directories D:\projects\vsp\Visual Studio 2010\Projects\mps-labs
Link Library Dependencies Yes
Use Library Dependency Inputs No
Link Status
Prevent Dll Binding
Treat Linker Warning As Errors
Force File Output
Create Hot Patchable Image
Specify Section Attributes
Additional Library Directories
Allows the user to override the environmental library path (/LIBPATH:folder)
OK
Cancel
Apply



Чрезвычайно важно понимать, что сам текст функции *EdrCenterText* не включается в файл edrtest.exe. Вместо этого там просто имеется ссылка на файл edrlib.dll и функцию *EdrCenterText,* которая находится в этом файле, файл edrtest.exe требует запуска файла edrlib.dll.

Заголовочный файл edrlib.h включен в файл с исходным текстом программы edrtest.c так же, как туда включен файл windows.h. Включение библиотеки импорта edrlib.lib для компоновки аналогично включению туда всех библиотек импорта Windows (например, user32.lib). Когда программа работает, она подключается к библиотеке edrlib.dll точно также, как к библиотеке user32.dll.

Необходимо сказать несколько слов о динамически подключаемых библиотеках. Все, что делает динамически подключаемая библиотека, делается от имени приложения. Например, владельцем всей выделяемой библиотекой памяти является приложение. Владельцем всех создаваемых библиотекой окон является приложение. Владельцем всех открываемых библиотекой файлов является приложение, владельцем всех создаваемых библиотекой окон является приложение. Владельцем всех открываемых библиотекой файлов является приложение. Несколько приложений одновременно могут использовать одну и ту же динамически подключаемую

библиотеку, но под Windows они не мешают друг другу. Однако, если написать динамически подключаемую библиотеку с функциями, которые могут вызываться из нескольких потоков одной программы, то компилировать библиотеку следует не с переменной среды GFLAGS, а с переменной среды GFLAGSMT.

Поскольку код защищен от записи, один и тот же код динамически подключаемой библиотеки может использоваться разными процессами. Однако данные, с которыми работает dll – для каждого процесса свои. Каждый процесс имеет собственное адресное пространство для всех данных, которые использует dll. Разделение памяти между процессами требует (как дальше будет видно) дополнительных усилий.

2.4 Разделяемая память в DLL

Windows изолирует друг от друга приложения, которые в одно и то же время используют одни и те же динамически подключаемые библиотеки. Однако, иногда это нежелательно. Может понадобиться написать DLL, содержащую некоторую область памяти, которая могла бы быть разделена между различными приложениями, или, может быть, различными экземплярами одного приложения. Это подразумевает использование разделяемой памяти (фактически, файла, проецируемого в память).

Давайте рассмотрим, как это работает на примере программы STRPROG ("string program") и динамически подключаемой библиотеки STRLIB ("string library"). В STRLIB имеется три экспортируемые функции, которые вызываются из STRPROG, одна из функций в STRLIB использует функцию обратного вызова, определенную в STRPROG.

STRLIB является модулем динамически подключаемой библиотеки, в котором хранятся и сортируются до 256 символьных строк. Строки переводятся в верхний регистр и запоминаются в разделяемой памяти STRLIB. Программа STRPROG может использовать функции из библиотеки STRLIB для добавления строк, удаления строк и получения из STRLIB всех имеющихся там строк. В программе имеется два пункта меню (Enter и Delete), которые вызывают окна диалога для добавления и удаления таких строк. Список всех находящихся в данный момент в STRLIB строк программа STRPROG выводит в своей рабочей области.

Приведенная ниже функция, определенная в STRLIB, добавляет строку в разделяемую память этой библиотеки:

```
EXPORT BOOL CALLBACK AddString (PSTR pStringIn)
```

Параметром *pStringIn* этой функции является указатель на строку, строка преобразуется к верхнему регистру внутри функции *AddString*. Если такая строка в списке строк библиотеки STRLIB уже существует, функция добавляет копию строки, функция *AddString* возвращает TRUE (ненулевое

значение), если ее вызов проходит удачно, и FALSE (0) – в противном случае. Значение FALSE может оказаться результатом того, что мина строки равна 0, или результатом того, что для хранения строки невозможно выделить память, или того, что 256 строк уже хранятся.

Следующая функция библиотеки STRLIB удаляет строку из разделяемой памяти этой библиотеки:

```
EXPORT BOOL CALLBACK DeleteString (PSTR pStringIn)
```

И снова, параметром *pStringIn* этой функции является указатель на строку. Если более чем одна строка совпадает, удаляется только первая. Функция *DeleteString* возвращает TRUE (ненулевое значение), если ее вызов проходит удачно, и FALSE (0) — в противном случае. Значение FALSE означает, что длина строки равна 0, или что совпадающей строки найти не удалось.

Следующая функция библиотеки STRLIB использует для упорядочения строк, которые в данный момент хранятся в разделяемой памяти этой библиотеки, функцию обратного вызова, находящуюся в вызывающей программе:

```
EXPORT int CALLBACK GetStrings (PSTRCB pfnGetStrCallBack, PVOID
pParam)
```

функция обратного вызова должна быть определена следующим образом:

```
export BOOL CALLBACK GetStrCallBack (PSTR pString, PVOID pParam)
```

Параметр *pfnGetStrCallBack* функции *GetStrings* является указателем на функцию обратного вызова. Функция *GetStrings* вызывает функцию *GetStrCallBack* по разу для каждой строки или до тех пор, пока возвращаемым значением функции обратного вызова не станет FALSE (0). Возвращаемым значением функции *GetStrings* является число строк, переданных функции обратного вызова. Параметр *pParam* — это указатель на определенные программистом данные.

Ниже представлено три файла, необходимых для создания модуля динамически подключаемой библиотеки STRLIB.DLL.

```
/*----------------------
   STRLIB.H header file
  ----------------------*/
#ifdef __cplusplus
extern "C" {
#endif

typedef BOOL (CALLBACK *PSTRCB) (PSTR, PVOID) ;
```

```
#define MAX_STRINGS 256

#ifdef __WATCOMC__
#define EXPORT __export __stdcall
#else
#define EXPORT __declspec (dllexport) __stdcall
#endif


BOOL EXPORT AddString    (PSTR) ;
BOOL EXPORT DeleteString (PSTR) ;
int  EXPORT GetStrings   (PSTRCB, PVOID) ;


#ifdef __cplusplus
}
#endif


/* STRLIB.C */
/*-----------------------------------------------
   STRLIB.C -- Library module for STRPROG program
  -----------------------------------------------*/

#include <windows.h>
#include "strlib.h"

#pragma data_seg ("shared")

PSTR pszStrings[MAX_STRINGS] = { NULL } ;
int  iTotal = 0 ;

#pragma data_seg ()

#pragma argsused
int WINAPI DllMain (HINSTANCE hInstance, DWORD fdwReason, PVOID pvRese
rved)
{
  int i ;

  switch (fdwReason)
  {
    /* Nothing to do when process (or thread) begins */
    case DLL_PROCESS_ATTACH :
    case DLL_THREAD_ATTACH :
    case DLL_THREAD_DETACH :
      break ;

    /* When process terminates, free any remaining blocks */
    case DLL_PROCESS_DETACH :
      for (i = 0 ; i < iTotal ; i++)
         UnmapViewOfFile (pszStrings[i]) ;
      break ;
  }
```

```
  return TRUE ;
}

BOOL EXPORT AddString (PSTR pStringIn)
{
     HANDLE hString ;
     PSTR   pString ;
     int    i, iLength, iCompare ;

  if (iTotal == MAX_STRINGS - 1)  return FALSE ;
  iLength = strlen (pStringIn) ;
  if (iLength == 0)         return FALSE ;
  hString = CreateFileMapping ((HANDLE) -1, NULL, PAGE_READWRITE,
                               0, 1 + iLength, NULL) ;
  if (hString == NULL)     return FALSE ;
  pString = (PSTR) MapViewOfFile (hString, FILE_MAP_WRITE, 0, 0, 0) ;
  strcpy (pString, pStringIn) ;
  AnsiUpper (pString) ;
  for (i = iTotal ; i > 0 ; i--)
  {
    iCompare = strcmpi (pStringIn, pszStrings[i - 1]) ;
    if (iCompare >= 0)      break ;
    pszStrings[i] = pszStrings[i - 1] ;
  }
  pszStrings[i] = pString ;
  iTotal++ ;
  return TRUE ;
}

BOOL EXPORT DeleteString (PSTR pStringIn)
{
  int i, j, iCompare ;

  if (0 == strlen (pStringIn))    return FALSE ;
  for (i = 0 ; i < iTotal ; i++)
  {
    iCompare = lstrcmpi (pszStrings[i], pStringIn) ;
    if (iCompare == 0)          break ;
  }
  /* If given string not in list, return without taking action */
  if (i == iTotal)        return FALSE ;
  /* Else free memory occupied by the string and adjust list downward
*/
  UnmapViewOfFile (pszStrings[i]) ;
  for (j = i ; j < iTotal ; j++)
    pszStrings[j] = pszStrings[j + 1] ;
  pszStrings[iTotal--] = NULL ;    /* Destroy unused pointer */
  return TRUE ;
}

int EXPORT GetStrings (PSTRCB pfnGetStrCallBack, PVOID pParam)
{
  BOOL bReturn ;
  int  i ;
```

```
  for (i = 0 ; i < iTotal ; i++)
  {
    bReturn = pfnGetStrCallBack (pszStrings[i], pParam) ;
    if (bReturn == FALSE)      return i + 1 ;
  }
  return iTotal ;
}
```

2.5 Точка входа/выхода библиотеки

Как можно заметить, в файле STRLIB.C мы некоторым образом задействовали функцию *DllMain.* Эта функция вызывается при первом запуске библиотеки и при завершении ее работы. Хотя функция *DllMain* включена еще и в EDRLIB.C, в действительности это не обязательно; функция, выполняющая такие действия, была бы включена туда компоновщиком по умолчанию.

Первым параметром функции *DllMain* является описатель экземпляра библиотеки. Если в библиотеке используются ресурсы, для которых требуется описатель экземпляра (например, *DialogBox),* необходимо сохранить *hInstance* в глобальной переменной. Последний параметр функции *DllMain* резервируется системой.

Параметр *fdwReason* может принимать одно из четырех значений, которое идентифицирует причину вызова функции *DllMain* системой Windows 95. Перед тем, как продолжить, запомните, что одна и та же программа может быть загружена несколько раз, и ее экземпляры могут вместе работать под Windows. Каждая загруженная программа рассматривается как отдельный процесс.

Значение DLL_PROCESS_ATTACH параметра *fdwReason* означает, что динамически подключаемая библиотека отображена в адресном пространстве процесса. Это сигнал библиотеке выполнить какие-то задачи по инициализации, которые требуются для обслуживания последующих запросов от процесса. Такая инициализация может включать в себя, например, выделение памяти. Во время выполнения программы функция *DllMain* вызывается с параметром DLL_PROCESS_ATTACH только однажды. Любой другой процесс, использующий ту же динамически подключаемую библиотеку, приводит к новому вызову функции *DllMain* с параметром DLL_PROCESS_ATTACH, но это происходит уже от имени нового процесса.

Если инициализация проходит удачно, возвращаемым значением функции *DllMain* должно быть ненулевое значение. Нулевое возвращаемое значение приведет к тому, что Windows не запустит программу.

Если параметр *fdwReason* имеет значение DLL_PROCESS_DETACH, то это означает, что динамически подключаемая библиотека больше процессу не нужна. Это дает возможность библиотеке освободить занимаемые ресурсы системы. В Windows 95 это не является абсолютно необходимым, но это хороший стиль программирования.

Аналогично, если функция *DllMain* вызывается с параметром DLL_THREAD_ATTACH, это означает, что связанный процесс создал новый поток. Когда поток завершается, Windows вызывает функцию *DllMain* с параметром DLL_THREA£)_DETACH. Запомните, что существует вероятность того, что вызов функции *DllMain* с параметром DLL_THREAD_DEIACH произойдет 5ез предварительного вызова ее с параметром DLL_THREAD_ATTACH. Это возможно в том случае, если библиотека была загружена после создания потока.

Когда функция *DllMain* вызывается с параметром DLL_THREAD_DETACH, поток еще существует. Динамически подключаемая библиотека может даже посылать сообщения потоку в этот момент. Но при этом нельзя использовать функцию *PostMessage,* поскольку поток, вероятно, закончится еще до того, как такое сообщение будет обработано.

Кроме функции *DllMain,* в STRLIB имеется только три функции, которые оудут экспортированы для использования другими программами. Все эти Функции определяются как EXPORT. Это приводит к тому, что компоновщик заносит их в библиотеку импорта STRLIB.LIB.

2.5.1 Программа STRPROG

Программа STRPROG, представленная на рис. 19.4, совершенно проста и понятна. Две опции меню (Enter и Delete) вызывают появление окон диалога для ввода строки. Затем в программе STRPROG вызываются функции *AddString u DeleteString.* Если программе нужно обновить свою рабочую область, вызывается функция *GetStrings,* а для получения списка перенумерованных строк вызывается функция *GetStrCallBack.*

```
/* STRPROG.H */
/*----------------------
   STRPROG.H header file
  ----------------------*/

#define IDM_ENTER      1
#define IDM_DELETE     2
#define IDD_STRING  0x10


/* STRPROG.C */
/*-------------------------------------------------------
   STRPROG.C -- Program using STRLIB dynamic link library
  -------------------------------------------------------*/
```

```
#include <windows.h>
#include <string.h>
#include "strprog.h"
#include "strlib.h"

#define MAXLEN 32
#define WM_DATACHANGE WM_USER

typedef struct
{
     HDC  hdc ;
     int  xText ;
     int  yText ;
     int  xStart ;
     int  yStart ;
     int  xIncr ;
     int  yIncr ;
     int  xMax ;
     int  yMax ;
}     CBPARAM ;

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

char szAppName[] = "StrProg" ;
char szString[MAXLEN] ;

#pragma argsused
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     HWND        hwnd ;
     MSG         msg ;
     WNDCLASS  wndclass ;

/*     wndclass.cbSize        = sizeof (wndclass) ; */
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = szAppName ;
     wndclass.lpszClassName = szAppName ;
/*  wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ; */
  RegisterClass (&wndclass) ;
  hwnd = CreateWindow (szAppName, "DLL Demonstration Program",
                           WS_OVERLAPPEDWINDOW,
                           CW_USEDEFAULT, CW_USEDEFAULT,
                           CW_USEDEFAULT, CW_USEDEFAULT,
                           NULL, NULL, hInstance, NULL) ;
  ShowWindow (hwnd, iCmdShow) ;
```

```
  UpdateWindow (hwnd) ;
  while (GetMessage (&msg, NULL, 0, 0))
  {
    TranslateMessage (&msg) ;
    DispatchMessage (&msg) ;
  }
  return msg.wParam ;
}

#pragma argsused
BOOL CALLBACK DlgProc (HWND hDlg, UINT iMsg, WPARAM wParam, LPARAM lPa
ram)
{
  switch (iMsg)
  {
    case WM_INITDIALOG :
      SendDlgItemMessage (hDlg, IDD_STRING, EM_LIMITTEXT, MAXLEN -
 1, 0) ;
      return TRUE ;

    case WM_COMMAND :
      switch (wParam)
      {
        case IDOK :
          GetDlgItemText (hDlg, IDD_STRING, szString, MAXLEN) ;
          EndDialog (hDlg, TRUE) ;
          return TRUE ;

        case IDCANCEL :
          EndDialog (hDlg, FALSE) ;
          return TRUE ;
      }
  }
  return FALSE ;
}

BOOL CALLBACK EnumCallBack (HWND hwnd, LPARAM lParam)
{
  char szClassName[16] ;

  GetClassName (hwnd, szClassName, sizeof (szClassName)) ;
  if (0 == strcmp (szClassName, szAppName))
    SendMessage (hwnd, WM_DATACHANGE, 0, 0) ;
  return TRUE ;
}

BOOL CALLBACK GetStrCallBack (PSTR pString, CBPARAM *pcbp)
{
  TextOut (pcbp->hdc, pcbp->xText, pcbp-
>yText,  pString, strlen (pString)) ;
  if ((pcbp->yText += pcbp->yIncr) > pcbp->yMax)
  {
    pcbp->yText = pcbp->yStart ;
    if ((pcbp->xText += pcbp->xIncr) > pcbp->xMax)    return FALSE ;
```

```
  }
  return TRUE ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM
lParam)
{
     static HINSTANCE  hInst ;
     static int        cxChar, cyChar, cxClient, cyClient ;
     CBPARAM           cbparam ;
     HDC               hdc ;
     PAINTSTRUCT       ps ;
     TEXTMETRIC        tm ;

  switch (iMsg)
  {
    case WM_CREATE :
      hInst = ((LPCREATESTRUCT) lParam)->hInstance ;
      hdc   = GetDC (hwnd) ;
      GetTextMetrics (hdc, &tm) ;
      cxChar = (int) tm.tmAveCharWidth ;
      cyChar = (int) (tm.tmHeight + tm.tmExternalLeading) ;
      ReleaseDC (hwnd, hdc) ;
      return 0 ;

    case WM_COMMAND :
      switch (wParam)
      {
        case IDM_ENTER :
          if (DialogBox (hInst, "EnterDlg", hwnd, &DlgProc))
          {
            if (AddString (szString)) EnumWindows (&EnumCallBack, 0) ;
            else                      MessageBeep (0) ;
          }
          break ;

        case IDM_DELETE :
          if (DialogBox (hInst, "DeleteDlg", hwnd, &DlgProc))
          {
          if (DeleteString (szString)) EnumWindows (&EnumCallBack, 0);
            else                        MessageBeep (0) ;
          }
          break ;
      }
      return 0 ;

    case WM_SIZE :
      cxClient = (int) LOWORD (lParam) ;
      cyClient = (int) HIWORD (lParam) ;
      return 0 ;

    case WM_DATACHANGE :
      InvalidateRect (hwnd, NULL, TRUE) ;
      return 0 ;
```

```
    case WM_PAINT :
      hdc = BeginPaint (hwnd, &ps) ;
      cbparam.hdc   = hdc ;
      cbparam.xText = cbparam.xStart = cxChar ;
      cbparam.yText = cbparam.yStart = cyChar ;
      cbparam.xIncr = cxChar * MAXLEN ;
      cbparam.yIncr = cyChar ;
      cbparam.xMax  = cbparam.xIncr * (1 + cxClient / cbparam.xIncr) ;
      cbparam.yMax  = cyChar * (cyClient / cyChar - 1) ;
      GetStrings ((PSTRCB) GetStrCallBack, (PVOID) &cbparam) ;
      EndPaint (hwnd, &ps) ;
      return 0 ;

    case WM_DESTROY :
      PostQuitMessage (0) ;
      return 0 ;
  }
  return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}

/* STRPROG.RC */
/*----------------------------
   STRPROG.RC resource script
  ----------------------------*/

#include <windows.h>
#include "strprog.h"

StrProg MENU
{
     MENUITEM  "&Enter!",  IDM_ENTER
     MENUITEM  "&Delete!", IDM_DELETE
}

EnterDlg DIALOG 24, 24, 190, 44
     STYLE WS_POPUP | WS_DLGFRAME
{
     LTEXT          "&Enter:",  0,            4,  8,  24,  8
     EDITTEXT                   IDD_STRING,  32,  6, 154, 12
     DEFPUSHBUTTON  "Ok",       IDOK,        44, 24,  32, 14
     PUSHBUTTON     "Cancel",   IDCANCEL,   114, 24,  32, 14
}

DeleteDlg DIALOG 24, 24, 190, 44
     STYLE WS_POPUP | WS_DLGFRAME
{
     LTEXT          "&Delete:", 0,            4,  8,  28,  8
     EDITTEXT                   IDD_STRING,  36,  6, 150, 12
     DEFPUSHBUTTON  "Ok",       IDOK,        44, 24,  32, 14
     PUSHBUTTON     "Cancel",   IDCANCEL,   114, 24,  32, 14
}
```

В файл strprog.c включен заголовочный файл strprog.h, в котором просто определены константы, используемые в файле описания ресурсов strprog.rc. Туда также включен заголовочный файл strlib.h, в котором определены три функции из библиотеки strlib, которые будут использоваться в программе strprog.

После того, как созданы файлы strlib.dll и strprog.exe, можно запускать программу STRPROG. Перед тем, как это сделать, удостоверьтесь, что файл strlib.dll находится в текущем каталоге или каталоге, который доступен Windows (об этом ранее говорилось). Windows должна иметь возможность загрузить файл strlib.dll при выполнении программы strprog. Если Windows не сможет найти файл strlib.dll, на экран будет выведено окно сообщения, информирующее об этой ошибке.

При выполнении файла strprog.exe, Windows реализует связывание с функциями во внешних модулях библиотек. Многие из этих функций находятся в обычных динамически подключаемых библиотеках Windows. Но Windows также видит, что программа вызывает три функции из STRLIB, и поэтому. Windows загружает файл strlib.dll в память и вызывает функцию инициализации strlib. Внутри strprog вызовы этих трех функций динамически связываются с функциями библиотеки strlib. После этого программу strprog можно использовать для добавления или удаления строк из внутренней таблицы библиотеки strlib. В рабочей области программы strprog выводятся строки, находящиеся в данный момент в таблице.

Вызов из программы strprog функций *AddString, DeleteString* и *GetStrings,* находящихся в библиотеке strlib, очень эффективен и не требует затрат больших, чем вызов любого другого внешнего модуля. Фактически, связь между strprog и strlib также эффективна, как если бы три функции библиотеки strlib были обычными функциями программы strprog. Вы спросите, почему? Зачем нужно делать эту динамически подключаемую библиотеку? Нельзя ли включить коды этих трех функций в файл strprog.exe?

Можно конечно. В определенном смысле библиотека strlib — это не больше, чем дополнение программы strprog. Однако, вам может быть интересно узнать, что случится, если запустить второй экземпляр программы strprog. Библиотека strlib запоминает строки символов и их указатели в разделяемой памяти, что позволяет всем экземплярам программы strprog совместно использовать эти данные. Рассмотрим, как это делается.

2.5.3 Разделение данных между экземплярами программы

Windows воздвигает как бы стену вокруг адресного пространства процесса Win32. Обычно, адресное пространство является частным, недоступным для других процессов. Но работа нескольких экземпляров программы strprog показывает, что библиотека strlib не имеет проблем при

разделении своих данных между всеми экземплярами программы. При добавлении или удалении строки в окне программы strprog, изменение немедленно отражается в других окнах.

Библиотека strlib предлагает для совместного использования два типа данных: строки и указатели на строки. Для иллюстрации, в библиотеке для каждого типа данных используются разные методы разделения данных. Библиотека strlib запоминает каждую строку в файле, проецируемом в память, делая строку видимой для всех процессов.

Библиотека strlib хранит указатели на строки в специальной области памяти, которая обозначается как разделяемая (shared):

```
#pragma data_seg ("shared")
PSTR pszStrings[MAX_STRINGS] == ( NULL } ;
int iTotal = 0 ;
#pragma data_seg ( )
```

Первая директива *#pragma* создает область данных, которая называется *shared.* Эту область можно назвать как угодно, хотя компоновщик распознает только первые восемь символов. Все инициализированные переменные, расположенные после директивы *#pragma* попадают в область памяти *shared.* Второй директивой *#pragma* отмечен конец области данных. Важно специально инициализировать эти переменные, в противном случае компилятор помещает их вместо области памяти *shared,* в обычную неинициализируемую область.

Компоновщику необходимо сообщить об области памяти *shared.* В командной строке компоновщика задайте параметр -SECTION следующим образом:

```
-SECTION: shared, rws
```

Буквы "rws" обозначают, что область памяти имеет атрибуты для чтения (read), записи (write) и разделения (shared) данных.

Теперь все экземпляры программы strprog видят один экземпляр строк и указателей. Функция *EnumCallBack* программы strprog служит для уведомления всех экземпляров программы strprog об изменении содержимого области данных библиотеки strlib. Вызов функции *EnumWindows* приводит к тому, что Windows вызывает функцию *EnumCallBack* с описателями всех родительских окон. Затем функция *EnumCallBack* проверяет соответствует ли имя класса каждого окна "StrProg"; если да, то функция посылает окну сообщение WM_DATACHANGE, определяемое в программе. Теперь можно легко представить себе модернизированную версию библиотеки strlib, управляющей базой данных, которая совместно используется несколькими экземплярами одной и той же программы или экземплярами нескольких программ.

2.6 Некоторые ограничения библиотек

Как уже говорилось, сам модуль динамически подключаемой библиотеки сообщений не получает. Однако, модуль библиотеки может вызывать функции *GetMessage* и *PeekMessage.* Сообщения, которые с помощью этих функций библиотека извлекает из очереди, фактически предназначены программе вызывающей функции из библиотеки. В общем, библиотека работает от имени вызвавшей ее программы, это правило остается верным для большинства функций Windows, вызываемых из библиотеки.

Динамически подключаемая библиотека может загружать ресурсы (такие, как значки, строки и битовые образы) либо из файла библиотеки, либо из файла программы, которая вызывает библиотеку. Функциям, которые загружают ресурсы, требуется описатель экземпляра. Если библиотека использует собственный описатель экземпляра (который передается библиотеке при инициализации), то библиотека может получить ресурсы из собственного файла. Для загрузки ресурсов из вызывающего библиотеку файла программы с расширением .exe, функции библиотеки требуется описатель экземпляра программы, вызвавшей функцию.

Регистрация классов окна и создание окон в библиотеке может оказаться несколько сложнее. И для структуры класса окна и для вызова функции *CreateWindow* требуется описатель экземпляра. Хотя при создании класса окна и самого окна можно использовать описатель экземпляра модуля библиотеки, сообщения окна по-прежнему будут проходить через очередь сообщений программы, вызвавшей библиотеку. Если необходимо создавать классы окна и сами окна внутри библиотеки, вероятно, лучше пользоваться описателем экземпляра вызывающей программы.

Поскольку сообщения для модальных окон диалога минуют очередь сообщений программы, с помощью вызова функции *DialogBox* можно создать в библиотеке модальное окно диалога. Описатель экземпляра можно взять из библиотеки, а параметр *hwndParent* функции *DialogBox* может быть задан равным NULL.

2.7 Динамическое связывание без импорта

Вместо того, чтобы Windows выполняла динамическое связывание при первой загрузке вашей программы в оперативную память, можно связать программу с модулем библиотеки во время выполнении программы. Например, можно было бы просто вызвать функцию *Rectangle.*

```
Rectangle (hdc, xLeft, yTop, xRight, yBottom) ;
```

Это работает, поскольку программа была скомпонована с библиотекой импорта gdi32.lib, в которой имеется адрес функции *Rectangle.*

Можно также вызвать функцию *Rectangle* и совершенно необычным образом. Сначала воспользуемся функцией *typedef* для определения типа функции *Rectangle:*

```
typedef BOOL (WINAPI *PFNRECT) (HDC, int, int, int, int) ;
```

Затем определяем две переменные:

```
HANDLE hLibrary ;
PFNRECT pfnRectangle ;
```

Теперь устанавливаем значение переменной *hLibrary* равным описателю библиотеки, а значение переменной *pfnRectangle* — равным адресу функции *Rectangle-*.

```
hLibrary = LoadLibrary ("GD132. DLL") ; pfnRectangle = (PFNRECT)
GetProcAddress (hLibrary, "Rectangle"") ;
```

Функции *LoadLibrary* возвращает NULL, если не удается найти файл библиотеки или случается какая-то другая ошибка. Теперь можно вызывать функцию и затем освободить библиотеку:

```
pfnRectangle (hdc, xLeft, yTop, xRight, yBottom) ; FreeLibrary
(hLibrary) ;
```

Если этот прием динамического связывания во время выполнения не имеет особого смысла *для* функции *Rectangle,* то смысл определенно появляется, если до начала выполнения программы неизвестно имя модуля библиотеки.

В приведенном выше коде используются функции *LoadLibrary* и *FreeLibrary.* Windows поддерживает для всех модулей библиотек счетчик ссылок. Вызов функции *LoadLibrary* приводит к увеличению на 1 значения счетчика ссылок. Счетчик ссылок также увеличивается на 1, когда Windows загружает любую программу, в которой используется библиотека. Вызов функции *FreeLibrary* приводит к уменьшению на 1 значения счетчика ссылок; то же происходит при завершении экземпляра программы, в которой используется библиотека. Если значение счетчика ссылок становится равным 0, Windows может удалить библиотеку из памяти, поскольку библиотека больше не нужна.

2.8 Библиотеки, содержащие только ресурсы

Любая функция в динамически подключаемой библиотеке, которую программы Windows или другие библиотеки могут использовать, должны экспортироваться. Однако, динамически подключаемая библиотека может

не содержать никаких экспортируемых функций. Тогда библиотека содержит ресурсы, которые используются в основном приложении.

Давайте поговорим о том, как работает приложение Windows, для которого требуются битовые образы. Обычно, они перечисляются в файле описания ресурсов программы и загружаются в оперативную память с помощью функции *LoadBitmap*. Но возможно необходимо создать несколько наборов битовых образов, причем каждый задается для одного из основных видеоадаптеров, используемых Windows. Имело бы прямой смысл хранить эти разные наборы битовых образов в разных файлах, поскольку пользователю мог бы понадобиться только один набор битовых образов на жестком диске. Такие файлы являются библиотеками, содержащими только ресурсы.

Ниже преддставлен код создания библиотеки, названной bitlib.dll содержащий только ресурсы – девять битовых образов. В файле bitlib.rc перечислены все файлы отдельных битовых образов и каждому присвоен номер. Для создания файла bitlib.dll, необходимо девять битовых образов bitlib1.bmp, bitlib2.bmp и т. д. Можно использовать готовые битовые образы или создать их в программе PAINT, поставляемой с Windows.

```
/* BITLIB.C */
/*-------------------------------------------------------------
   BITLIB.C -- Code entry point for BITLIB dynamic link library
  -------------------------------------------------------------*/

#include <windows.h>

#pragma argsused
int WINAPI DllMain (HINSTANCE hInstance, DWORD fdwReason, PVOID pvRese
rved)
{
  return TRUE ;
}

/* BITLIB.RC */
/*---------------------------
   BITLIB.RC resource script
  ---------------------------*/

1 BITMAP bitmap1.bmp
2 BITMAP bitmap2.bmp
3 BITMAP bitmap3.bmp
4 BITMAP bitmap4.bmp
5 BITMAP bitmap5.bmp
```

Программа showbit, приведенная ниже, считывает ресурсы битовых образов из библиотеки bitlib и отображает их в левом верхнем углу рабочей области. Нажимая клавишу на клавиатуре, можно вывести последовательно все битовые образы.

```
/*------------------------------------------------------------
   SHOWBIT.C -- Shows bitmaps in BITLIB dynamic link library
  ------------------------------------------------------------*/

#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM) ;

#pragma argsused
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    PSTR szCmdLine, int iCmdShow)
{
     static char  szAppName[] = "ShowBit" ;
     HWND         hwnd ;
     MSG          msg ;
     WNDCLASS   wndclass ;

/*     wndclass.cbSize        = sizeof (wndclass) ; */
     wndclass.style         = CS_HREDRAW | CS_VREDRAW ;
     wndclass.lpfnWndProc   = WndProc ;
     wndclass.cbClsExtra    = 0 ;
     wndclass.cbWndExtra    = 0 ;
     wndclass.hInstance     = hInstance ;
     wndclass.hIcon         = LoadIcon (NULL, IDI_APPLICATION) ;
     wndclass.hCursor       = LoadCursor (NULL, IDC_ARROW) ;
     wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH) ;
     wndclass.lpszMenuName  = NULL ;
     wndclass.lpszClassName = szAppName ;
/*     wndclass.hIconSm       = LoadIcon (NULL, IDI_APPLICATION) ; */

  RegisterClass (&wndclass) ;
  hwnd = CreateWindow (szAppName, "Show Bitmaps from BITLIB (Press Key
)",
                       WS_OVERLAPPEDWINDOW,
                       CW_USEDEFAULT, CW_USEDEFAULT,
                       CW_USEDEFAULT, CW_USEDEFAULT,
                       NULL, NULL, hInstance, NULL) ;

  ShowWindow (hwnd, iCmdShow) ;
  UpdateWindow (hwnd) ;
  while (GetMessage (&msg, NULL, 0, 0))
  {
    TranslateMessage (&msg) ;
    DispatchMessage (&msg) ;
  }
  return msg.wParam ;
}
```

```
void DrawBitmap (HDC hdc, int xStart, int yStart, HBITMAP hBitmap)
{
  BITMAP bm ;
  HDC    hMemDC ;
  POINT  pt ;

  hMemDC = CreateCompatibleDC (hdc) ;
  SelectObject (hMemDC, hBitmap) ;
  GetObject (hBitmap, sizeof (BITMAP), (PSTR) &bm) ;
  pt.x = bm.bmWidth ;
  pt.y = bm.bmHeight ;
  BitBlt (hdc, xStart, yStart, pt.x, pt.y, hMemDC, 0, 0, SRCCOPY) ;
  DeleteDC (hMemDC) ;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg, WPARAM wParam, LPARAM
lParam)
{
     static HINSTANCE  hLibrary ;
     static int        iCurrent = 1 ;
     HBITMAP           hBitmap ;
     HDC               hdc ;
     PAINTSTRUCT       ps ;

  switch (iMsg)
  {
    case WM_CREATE :
      if ((hLibrary = LoadLibrary ("BITLIB.DLL")) == NULL)
                    DestroyWindow (hwnd) ;
      return 0 ;

    case WM_CHAR :
      if (hLibrary)
      {
        iCurrent ++ ;
        InvalidateRect (hwnd, NULL, TRUE) ;
      }
      return 0 ;

    case WM_PAINT :
      hdc = BeginPaint (hwnd, &ps) ;
      if (hLibrary)
      {
        if (NULL == (hBitmap = LoadBitmap (hLibrary,
              MAKEINTRESOURCE (iCurrent))))
        {
          iCurrent = 1 ;
          hBitmap = LoadBitmap (hLibrary,   MAKEINTRESOURCE (iCurrent)
) ;
        }
        if (hBitmap)
        {
          DrawBitmap (hdc, 0, 0, hBitmap) ;
          DeleteObject (hBitmap) ;
```

```
          }
        }
        EndPaint (hwnd, &ps) ;
        return 0 ;

      case WM_DESTROY :
        if (hLibrary)    FreeLibrary (hLibrary) ;
        PostQuitMessage (0) ;
        return 0 ;
    }
    return DefWindowProc (hwnd, iMsg, wParam, lParam) ;
}
```

При обработке сообщения WM_CREATE, программа SHOWBIT получает описатель библиотеки BITLIB.DLL:

```
if ( (hLibrary =LoadLibrary ("BITLIB.DLL")) == null)
DestroyWindow (hwnd) ;
```

Если библиотеки BITLIB.DLL нет в текущем каталоге, про1рамм SHOWBIT будет искать ее так, как уже говорилось ранее в этой главе. Если найти библиотеку не удается, Windows выводит на экран окно сообщения, извещающее об этой ошибке. Когда пользователь нажимает клавишу <OK>, функция *LoadLibrary* возвращает значение NULL, и программа SHOWBIT завершается.

Программа SHOWBIT, с помощью вызова функции *LoadBitmap* с описателем библиотеки и номером битового образа, может получить его описатель:

```
hBitmap = LoadBitmap (hLibrary, MAKEINTRESOURCE (iCurrent)) ;
```

Эта функция вернет ошибку, если битовый образ, соответствующий числу *iCurrent,* содержит ошибку, или, если недостаточно памяти для загрузки битового образа.

При обработке сообщения WM_DESTROY, программа SHOWBIT освобождает библиотеку:

FreeLibrary (hLibrary) ;

Когда последний экземпляр программы SHOWBIT завершается, счетчик ссылок библиотеки BITLIB.DLL становится равным 0, и занимаемая библиотекой память освобождается.

3 Контрольные вопросы

1) Опишите структуру динамической библиотеки.
2) Как добавть использование библиотеки в своей программе?
3) Как динамически подгрузить библиотеку?
4) Как работает критическая секция?

5) Как создать событие?
6) Какие проблемы возникают при написании сногопоточных приложений.

4 Задание

1) Изучить описание лабораторной работы.
2) Создать динмическую библиотеку Windows, реализующую некоторую полезную функциональность
3) Составить приложение Windows, использующее эту библиотеку.
4) Отладить и протестировать полученную программу.
5) Оформить отчёт.

Лабораторная работа № 14. Использование каналов (pipes) для организации межпроцессного взаимодействия

1 Цель работы

Изучить процесс создания и использования каналов (pipes) для организации межпроцессного взаимодействия.

2 Кртакая теория

Каналы представляют собой средство межпроцессного взаимодействия. Существует два вида каналов: Анонимные каналы и Именованные каналы.

2.1 Анонимные каналы

Анонимные каналы обеспечивают межпроцессное взаимодействие на локальном компьютере. Анонимные каналы используют меньше системных ресурсов, чем именованные каналы, но их возможности ограничены. Анонимные каналы являются односторонними и не могут использоваться для взаимодействия по сети. Они позволяют использовать только один экземпляр сервера. Анонимные каналы удобно использовать для организации взаимодействия между потоками или между родительскими и дочерними процессами – в этом случае дескриптор канала можно просто передать дочернему процессу при его создании.

В .NET Framework анонимные каналы реализуются с помощью классов AnonymousPipeServerStream и AnonymousPipeClientStream.

Класс `AnonymousPipeServerStream` предоставляет поток для анонимного канала, поддерживающий синхронные и асинхронные операции чтения и записи. Дает возможность родительскому процессу отправлять информацию в дочерний процесс и получать из него информацию.

Класс `AnonymousPipeClientStream` предоставляет сторону клиента в потоке анонимного канала, поддерживающем синхронные и асинхронные

операции чтения и записи. Дает возможность подключить дочерний процесс к родительскому и организовать обмен информацией между ними.

2.2 Именованные каналы

Именованные каналы обеспечивают межпроцессное взаимодействие между сервером и одним или несколькими клиентами. Именованные каналы могут быть односторонними или двусторонними. Они поддерживают связь с помощью сообщений и позволяют нескольким клиентам подключаться к серверному процессу одновременно, используя одно и то же имя канала. Именованные каналы также поддерживают олицетворение, позволяющее подключающимся процессам использовать собственные разрешения на удаленных серверах.

В .NET Framework именованные каналы реализуются с помощью классов NamedPipeServerStream и NamedPipeClientStream.

Рассмотрим листинг программы, осуществляющей передачу информации (10 случайным образом сгенерированных чисел) от приложения-сервера к приложению-клиенту при помощи именованных каналов.

Пример 14.1 Приложение «pipeServer»

```
using System;
using System.IO;
using System.IO.Pipes;
/*
 Осуществляется передача 10-ти случайных чисел приложению-
клиенту при помощи именованного канала.
 */
class PipeServer
{
   static void Main()
   {
      // Бесконечный цикл
      while (true)
      {
         /* Инициализируется новый экземпляр класса NamedPipeClientStr
eam с заданными именами канала и указанным направлением канала.PipeDir
ection.Out указывает направление канала - исходящий канал /**/
         using (NamedPipeServerStream pipeServer = new NamedPipeServer
Stream("testpipe", PipeDirection.Out))
         {
            /*Ждем когда клиент подключится*/
            // Сообщение в консоль
            Console.WriteLine("Объект NamedPipeServerStream создан.");
            // Сообщение в консоль
            Console.Write("Ждем подключения клиента...");
            // Ожидает подключения клиента к данному объекту NamedPipe
ServerStream.
```

```
                pipeServer.WaitForConnection();
                // Переменная целого типа
                int a;
                // Генератор псевдослучайных чисел
                Random rnd = new Random();
                // Сообщение в консоль
                Console.WriteLine("Клиент подключен");
                try
                {
                   /*Генерируем 10 случайных чисел, и отправляем клиенту*/
                   //Создается sw как экземпляр класса StreamReader
                   using (StreamWriter sw = new StreamWriter(pipeServer))
                   {
                      // Цикл от 1 до 10
                      for (int i = 0; i < 10; i++)
                      {
                         // Генерируется следующее случайное число
                         a = rnd.Next(10);
                         // Свойство AutoFlush определяет, будет ли сброше
н буфер после записи. true = да; false = нет
                         sw.AutoFlush = true;
                         // Записывается число в поток.
                         sw.WriteLine(a);
                      }
                   }
                }
                catch (IOException e)
                { Console.WriteLine("ERROR: {0}", e.Message); }
             }
          }
       }
    }
```

Пример 14.2 Приложение «pipeClient»

```
using System;
using System.IO;
using System.IO.Pipes;

/*
 В бесконечном цикле осуществляется получение 10ти случайных чисел от
приложения-сервера при помощи именованного канала.
 */
class PipeClient
{
   static void Main(string[] args)
   {
      //Бесконечный цикл
      while (true)
      {
```

```
            /* Инициализирует новый экземпляр класса NamedPipeClientStrea
m с заданными именами канала и сервера и указанным направлением канала
.PipeDirection.In указывает направление канала
            и указывает что канал входящий /**/
            using (NamedPipeClientStream pipeClient =
            new NamedPipeClientStream(".", "testpipe", PipeDirection.In))
            {
                /*Подключаемся к каналу, или ждем его готовности*/
                //Сообщение в консоль
                Console.Write("Подключаемся к каналу...");
                // Ожидает подключения к серверу. Метод Connect() ожидает,
 пока экземпляр канала не станет доступным
                pipeClient.Connect();
                // Сообщение в консоль
                Console.WriteLine("Подключено");
                // Свойство NumberOfServerInstances получает число экземпл
яров сервера с одинаковым именем канала
                Console.WriteLine("Открыт {0} канал(а) сервера", pipeClien
t.NumberOfServerInstances);
                //Создается sr как экземпляр класса StreamReader
                using (StreamReader sr = new StreamReader(pipeClient))
                {
                    //Переменная строкового типа
                    string temp;
                    //Считываем символы из потока
                    while ((temp = sr.ReadLine()) != null)
                    //Полученное от сервера сообщение выводится в консоль
                    {Console.WriteLine("Получено от сервера: {0}", temp); }
                }
            }
            //Сообщение в консоль
            Console.WriteLine("Enter - продолжить; Escape - завершить");
            //Получаем название нажатой клавиши при помощи ConsoleKeyInfo
            ConsoleKeyInfo info = Console.ReadKey();
            //Если нажат Escape -
 прерывается цикл, и завершается работа приложения
            if (info.Key == ConsoleKey.Escape)
            { break; }
        }
    }
}
```

3 Контрольные вопросы

1) Что такое канал?
2) Какие виды каналов вы знаете?
3) Как создать именованныйц канал?
4) Как подключиться к именованному каналу?

4 Задание

1) Изучить описание лабораторной работы.

2) Создать приложение-клиент и приложение-сервер, реализующую некоторую полезную функциональность в соответствии с вариантом.
3) Отладить и протестировать полученную программу.
4) Оформить отчёт.

5 Варианты заданий

Вариант 1. Отправить 10 случайных цифр по именованному каналу из приложения-клиента в приложение-сервер. Получить сумму цифр, и отправить их обратно.

Вариант 2. Отправить по именованному каналу из приложения-клиента в приложение-сервер число. Найти факториал этого числа. Вывести результат в консоль.

Вариант 3. Отправить по именованному каналу из приложения-сервер в приложение-клиент число. Найти делители этого числа. Вывести результат в консоль.

Вариант 4. Отправить по именованному каналу из приложения-сервер в приложение-клиент два числа. Найти НОД этихчисел. Вывести результат в консоль.

Вариант 5. Отправить по именованному каналу из приложения-сервер в приложение-клиент два числа. Найти НОК этихчисел. Вывести результат в консоль.

Лабораторная работа № 15. Организация сетевого взаимодействия с помощью сокетов (sockets)

1 Цель работы

Изучить процесс работы с сокетами и взаимодействия приложений по сети.

2 Кртакая теория

Со́кеты (англ. socket — углубление, гнездо, разъём) – название программного интерфейса для обеспечения обмена данными между процессами. Процессы при таком обмене могут исполняться как на одной ЭВМ, так и на различных ЭВМ, связанных между собой сетью. Сокет – абстрактный объект, представляющий конечную точку соединения.

Следует различать клиентские и серверные сокеты. Клиентские сокеты грубо можно сравнить с оконечными аппаратами телефонной сети, а серверные — с коммутаторами. Клиентское приложение (например, браузер) использует только клиентские сокеты, а серверное (например, веб-сервер, которому браузер посылает запросы) — как клиентские, так и серверные сокеты.

Интерфейс сокетов впервые появился в BSD Unix. Программный интерфейс сокетов описан в стандарте POSIX.1 и в той или иной мере поддерживается всеми современными операционными системами.

2.1 Принципы работы сокетов

Каждый процесс может создать слушающий сокет (серверный сокет) и привязать его к какому-нибудь порту операционной системы (в UNIX непривилегированные процессы не могут использовать порты меньше 1024). Слушающий процесс обычно находится в цикле ожидания, то есть просыпается при появлении нового соединения. При этом сохраняется возможность проверить наличие соединений на данный момент, установить тайм-аут для операции и т.д.

Каждый сокет имеет свой адрес. ОС семейства UNIX могут поддерживать много типов адресов, но обязательными являются INET-адрес и UNIX-адрес. Если привязать сокет к UNIX-адресу, то будет создан специальный файл (файл сокета) по заданному пути, через который смогут сообщаться любые локальные процессы путём чтения/записи из него (см. Доменный сокет Unix). Сокеты типа INET доступны из сети и требуют выделения номера порта.

Обычно клиент явно подсоединяется к слушателю, после чего любое чтение или запись через его файловый дескриптор будут передавать данные между ним и сервером.

2.2 Пример на C++

В качестве примера для реализации выбран протокол UDP, как наиболее простой вариант.

UDP (англ. User Datagram Protocol — протокол пользовательских дейтаграмм) — это транспортный протокол для передачи данных в сетях IP без установления соединения. Он является одним из самых простых протоколов транспортного уровня модели OSI. Его IP-идентификатор — 0x11.

В отличие от TCP, UDP не гарантирует доставку пакета, поэтому аббревиатуру иногда расшифровывают как Unreliable Datagram Protocol (протокол ненадёжных датаграмм). Это позволяет ему гораздо быстрее и эффективнее доставлять данные для приложений, которым требуется большая пропускная способность линий связи, либо требуется малое время доставки данных.

Пример 15.1 Пример реализации UDP-сервера

```
// Пример простого UDP-эхо сервера
#include <stdio.h>
#include <winsock2.h>
```

```
#define PORT 666 // порт сервера
#define sHELLO "Hello, %s [%s] Sailor\n"

int main(int argc, char* argv[])
 {
     char buff[1024];
     printf("UDP DEMO echo-Server\n");
     // Шаг 1 - подключение библиотеки
     if (WSAStartup(0x202, (WSADATA *)&buff[0]))
     {
         printf("WSAStartup error: %d\n", WSAGetLastError());
         return -1;
     }

     // Шаг 2 - создание сокета
     SOCKET my_sock;
     my_sock = socket(AF_INET,SOCK_DGRAM,0);
     if (my_sock == INVALID_SOCKET)
     {
         printf("Socket() error: %d\n", WSAGetLastError());
         WSACleanup();
         return -1;
     }

     // Шаг 3 - связывание сокета с локальным адресом
     sockaddr_in local_addr;
     local_addr.sin_family = AF_INET;
     local_addr.sin_addr.s_addr = INADDR_ANY;
     local_addr.sin_port = htons(PORT);

     if (bind(my_sock, (sockaddr *)&local_addr, sizeof(local_addr)))
     {
         printf("bind error: %d\n", WSAGetLastError());
         closesocket(my_sock);
         WSACleanup();
         return -1;
     }

     // Шаг 4 обработка пакетов, присланных клиентами
     while (1)
     {
         sockaddr_in client_addr;
         int client_addr_size = sizeof(client_addr);
         int bsize = recvfrom(my_sock,&buff[0],sizeof(buff)-1,0,
         (sockaddr *)&client_addr, &client_addr_size);
         if (bsize == SOCKET_ERROR)
             printf("recvfrom() error: %d\n", WSAGetLastError());

         // Определяем IP-адрес клиента и прочие атрибуты
         HOSTENT *hst;
         hst = gethostbyaddr((char *)&client_addr.sin_addr, 4, AF_INET
);
         printf("+%s [%s:%d] new DATAGRAM!\n",
         (hst) ? hst->h_name : "Unknown host",
```

```
            inet_ntoa(client_addr.sin_addr),
            ntohs(client_addr.sin_port));

            // добавление завершающего нуля
            buff[bsize] = 0;

            // Вывод на экран
            printf("C=>S:%s\n", &buff[0]);

            // посылка датаграммы клиенту
            sendto(my_sock, &buff[0], bsize, 0,
                (sockaddr *)&client_addr, sizeof(client_addr));
        }
        return 0;
    }
```

Пример 15.1 Пример реализации UDP- клиента

```
// пример простого UDP-клиента
#include <stdio.h>
#include <string.h>
#include <winsock2.h>
#include <windows.h>

#define PORT 666
#define SERVERADDR "127.0.0.1"

int main(int argc, char* argv[])
{
   char buff[10 * 1014];
   printf("UDP DEMO Client\nType quit to quit\n");

   // Шаг 1 - инициализация библиотеки Winsocks
   if (WSAStartup(0x202, (WSADATA *)&buff[0]))
   {
      printf("WSAStartup error: %d\n", WSAGetLastError());
      return -1;
   }

   // Шаг 2 - открытие сокета
   SOCKET my_sock = socket(AF_INET, SOCK_DGRAM, 0);
   if (my_sock == INVALID_SOCKET)
   {
      printf("socket() error: %d\n", WSAGetLastError());
      WSACleanup();
      return -1;
   }

   // Шаг 3 - обмен сообщений с сервером
   HOSTENT *hst;
   sockaddr_in dest_addr;

   dest_addr.sin_family = AF_INET;
   dest_addr.sin_port = htons(PORT);
```

```
   // определение IP-адреса узла
   if (inet_addr(SERVERADDR) != INADDR_NONE)
      dest_addr.sin_addr.s_addr = inet_addr(SERVERADDR);
   else
   {
      if (hst = gethostbyname(SERVERADDR))
         dest_addr.sin_addr.s_addr = ((unsigned long **) hst-
>h_addr_list)[0][0];
      else
      {
         printf("Unknown host: %d\n", WSAGetLastError());
         closesocket(my_sock);
         WSACleanup();
         return -1;
      }
   }
   while (1)
   {
      // чтение сообщения с клавиатуры
      printf("S<=C:"); fgets(&buff[0], sizeof(buff) - 1, stdin);
      if (!strcmp(&buff[0], "quit\n")) break;

      // Передача сообщений на сервер
      sendto(my_sock, &buff[0], strlen(&buff[0]), 0, \
         (sockaddr *)&dest_addr, sizeof(dest_addr));

      // Прием сообщения с сервера
      sockaddr_in server_addr;
      int server_addr_size = sizeof(server_addr);

      int n = recvfrom(my_sock, &buff[0], sizeof(buff) - 1, 0, \
         (sockaddr *)&server_addr, &server_addr_size);

      if (n == SOCKET_ERROR)
      {
         printf("recvfrom() error: %d\n", WSAGetLastError());
         closesocket(my_sock);
         WSACleanup();
         return -1;
      }

      buff[n] = 0;

      // Вывод принятого с сервера сообщения на экран
      printf("S=>C:%s", &buff[0]);
   }

   // Шаг последний - выход
   closesocket(my_sock);
   WSACleanup();
   return 0;
}
```

2.3 Пример реализации на C#

Для наглядности напишем два приложения, которые будут общаться друг с другом. Одно будет сервером, другое клиентом. Общаться они будут по UDP протоколу.

2.3.1 Клиентское приложение

Создаем консольный проект. В проекте пишем две функции, одна из которых будет «слушать» и «отвечать» на запросы. Назовем ее Listen.

```
//Для клиента
public void Listen()
{
    int recv; //храним размер полученных данных
    byte data = new byte[1024]; //данные, которые будут передаваться и
ли приниматься

    //создаем новый сокет
    //параметр AddressFamily задает схему адресации. В нашем случае эт
о адреса IPv4
    //параметр SocketType указывает, какой тип сокета применяется.
    //В данном случае SocketType.Dgram это ненадежные сообщения, но дл
я примера хватит.
    //Тем более, что Dgram поддерживает протокол UDP
    //последний параметр, ProtocolType, задает тип протокола
    Socket mysocket = new Socket(AddressFamily.InterNetwork, SocketTyp
e.Dgram, ProtocolType.Udp);

    //создаем конечную точку по адресу сокета. Т.е. будем "слушать" по
рт 9051 и контролировать все сетевые интерфейсы
    IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 9051);
    mysocket.Bind(ipep); //привязываем точку к нашему сокету

    //создаем еще одну точку
    IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);

    //определяем сетевой адрес
    EndPoint Remote = (EndPoint)(sender);

    //отправляем первое сообщение нашему серверу
    string text = "Hello"; //текст сообщения
    data = Encoding.ASCII.GetBytes(text); //переводим строку в байты

    //отправляем на указанны адрес
    //первый параметр это сами данные в виде массива байт
    //второй параметр, какая длиная сообщения должна быть передана.
```

```
    //Если указать меньше, то передаст только то число символов, сколь
ко укажете
    //третий параметр указывает поведение сокета при приеме и получени
и данных. В нашем случае ничего...
    //четвертый парамерт задает адрес и порт сервера, которому нужно о
тправить сообщение
    //что делает функция _getHost, смотрите чуть ниже...
    //в данном примере используется бродкастовый адрес подсети 192.168
.15.255,
    //если мы не знаем по какому адресу находится сервер. Порт подстав
ляем любой (1111), он все равно будет перезаписан в функции _getHost
    //Использование функции SendTo позволяет заранее не соединяться с
сервером
    mysocket.SendTo(data, data.Length, SocketFlags.None, _getHost("192
.168.15.255:1111"));

    //запускаем бесконечный цикл, который будет принимать и отправлять
 данные
    while (true)
    {
        data = new byte[1024];
        //принимаем данные от сервера. recv содержир размер, т.е. коли
чество принятых символов
        recv = mysocket.ReceiveFrom(data, ref Remote);
        //в данном примере сервер шлет сообщение, которое разделено ":
"
        //разбираем его в массив
        //функция Encoding.ASCII.GetString переводит массив байт в стр
оку
        string[] args = Encoding.ASCII.GetString(data, 0, recv).ToLowe
r().Split(':');
        foreach(string item in args)
        {
            Console.WriteLine("Сервер отправил " + item);
        }
        //отправим ответ что мы получили. Например, первый элемент мас
сива
        data = Encoding.ASCII.GetBytes(args[0]);

        //Remote содержит адрес, с которого пришло сообщение. Ему его
назад и отправляем
        mysocket.SendTo(data, data.Length, SocketFlags.None, _getHost(
Remote.ToString()));
    }
}

//функция _getHost
private EndPoint _getHost(string text)
{
    //вырезаем из строки только IP адрес формата IPv4
    string host = text.Remove(text.IndexOf(":"), text.Length -
 text.IndexOf(":"));
```

```
    //создаем объект адреса. Переменная host уже имеет вид [0-255].[0-
255].[0-255].[0-255]
    IPAddress hostIPAddress = IPAddress.Parse(host);

    //создаем конечную точку. В нашем случае это адрес сервера, которы
й слушает порт 9050
    IPEndPoint hostIPEndPoint = new IPEndPoint(hostIPAddress, 9050);
    EndPoint To = (EndPoint)(hostIPEndPoint);
    return To;
}
```

2.3.2 Сервер

Приложение сервер также будет состоять из двух функций, но не будет отправлять запрос с приветствием. Просто будет слушать и в зависимости от принятых данных отправлять сообщения.

Внимание! Порт, который слушает сервер – 9050. Отправляет он на 9051.

Создаем новое консольное приложение для сервера.

```
//Для сервера
public void Listen()
{
    int recv; //храним размер полученных данных
    byte data = new byte[1024]; //данные, которые будут передаваться и
ли приниматься

    Socket mysocket = new Socket(AddressFamily.InterNetwork, SocketTyp
e.Dgram, ProtocolType.Udp);

    //В отличие от клиента "слушаем" порт 9050
    IPEndPoint ipep = new IPEndPoint(IPAddress.Any, 9050);
    mysocket.Bind(ipep); //привязываем точку к нашему сокету
    IPEndPoint sender = new IPEndPoint(IPAddress.Any, 0);
    EndPoint Remote = (EndPoint)(sender);

    //запускаем бесконечный цикл, который будет принимать и отправлять
 данные
    while (true)
    {
        data = new byte[1024];
        //принимаем данные от клиента. recv содержир размер, т.е. коли
чество принятых символов
        recv = mysocket.ReceiveFrom(data, ref Remote);
        string message = Encoding.ASCII.GetString(data, 0, recv);
        switch(message)
        {
            case "Hello" : //на ответ от клиента Hello, шлем ответ OK
                data = Encoding.ASCII.GetBytes("OK:Сервер найден!");
                break;
        }
        //Remote содержит адрес, с которого пришло сообщение. Ему его
назад и отправляем
```

```
        mysocket.SendTo(data, data.Length, SocketFlags.None, _getHost(
Remote.ToString()));
    }
}

//функция _getHost для сервера отличается только портом. Вместо 9050,
отправлять всегда будем на 9051
private EndPoint _getHost(string text)
{
    //вырезаем из строки только IP адрес формата IPv4
    string host = text.Remove(text.IndexOf(":"), text.Length -
 text.IndexOf(":"));

    //создаем объект адреса. Переменная host уже имеет вид [0-255].[0-
255].[0-255].[0-255]
    IPAddress hostIPAddress = IPAddress.Parse(host);

    //создаем конечную точку. В нашем случае это адрес сервера, которы
й слушает порт 9051
    IPEndPoint hostIPEndPoint = new IPEndPoint(hostIPAddress, 9051);
    EndPoint To = (EndPoint)(hostIPEndPoint);
    return To;
}
```

3 Контрольные вопросы

1) Что такое сокет?
2) Опишите принципы работы сокетов.
3) В чем основное отличие протоколов UDP и TCP?

4 Задание

1) Изучить описание лабораторной работы.
2) Создать приложени Windows, использующие возможности взаимодействия по сети.
3) Отладить и протестировать полученную программу.
4) Оформить отчёт.

Список литературы

1. Гордеев А. В., Молчанов А. Ю. Системное программное обеспечение. – СПб.: Питер, 2001. – 736 с.: ил.
2. Харт, Д. М. Системное программирование в среде Win32, 2-е изд.: Пер. с англ.: – М.: Издательский дом «Вильямс», 2001.– 464 с.: ил. – Парал. тит. англ.
3. Петзольд Ч. Программирование для Windows 95; в двух томах. Том 1/ Пер. С англ. — СПб.: BHV— Санкт-Петербург, 1997.— 752 с.: ил.
4. Петзольд Ч. Программирование для Windows 95; в двух томах. Том 2/ Пер. С англ. — СПб.: BHV— Санкт-Петербург, 1997.— 368 с.: ил.
5. Румянцев П.В. Азбука программирования в Win 32 API. – М.: Горячая Линия - Телеком, 2004. – 312 с.
6. Ганеев Р. М. Проектирование интерфейса пользователя средствами Win32 API. – М: Горячая Линия – Телеком, 2001. – 336 с.
7. Харт, Д. М. Системное программирование в среде Windows Windows System Programming. – М.: Вильямс, 2005.– 336 с.: ил.
8. Неббет, Гэри. Справочник по базовым функциям API Windows NT/2000 Windows NT/2000 – М: Вильямс, 2002. – 528 с.

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Часть 3. Разработка ПО системного назначения

Методические указания

Составители: Мурлин Алексей Георгиевич;
Волик Александр Георгиевич

Авторская правка

Компьютерная верстка	А. Г. Волик
Подписано в печать	Изд. № 000
Электронное издание	Изд. каф.
0,9 Мб	

Кубанский государственный технологический университет
350072, г. Краснодар, ул. Московская, 2, кор. А
Типография КубГТУ: 350058, г. Краснодар, ул. Старокубанская, 88/4