ВОРОНКИН Р. А.

ПРАКТИКУМ ПО СИСТЕМНОМУ ПРОГРАММИРОВАНИЮ

УЧЕБНОЕ ПОСОБИЕ (Л практикум)

СТАВРОПОЛЬ

2014

Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего профессионального образования

«Северо-Кавказский федеральный университет»

Воронкин Р.А.

ПРАКТИКУМ ПО СИСТЕМНОМУ ПРОГРАММИРОВАНИЮ

Учебное пособие

(Практикум)

Часть 1

Ставрополь, 2014

УДК

ББК

Рецензенты:

Директор Института информационных технологий и телекоммуникаций

Заведующий кафедрой информационной безопасности автоматизированных систем

Кандидат технических наук, профессор,

Чипига А.Ф.

(Северо-Кавказский федеральный университет)

Воронкин Р.А. Практикум по системному программированию. Учебное пособие (лабораторный практикум). – Ставрополь: Изд-во СКФУ, 2014. – 54 с.

Пособие содержит теоретический и практический материал для выполнения самостоятельных работ по дисциплине «Практикум по системному программированию». Предназначено для студентов специальности «Информационная безопасность автоматизированных систем», изучающих дисциплину «Практикум по системному программированию».

УДК

ББК

Кавказский

федеральный

университет», 2014

Содержание

Самостоятельная работа 1

Программа на Си для Windows, как и для любой другой платформы, должна обязательно содержать некоторую "стартовую" функцию, которой передается управление при запуске программы. Вообще говоря, имя такой "стартовой" функции может различаться в различных компиляторах, но исторически сложилось так (а, кроме того, имеются еще и стандарты ANSI и ISO, к которым, правда, производители коммерческих компиляторов типа Microsoft и Borland/Inprise относятся без особого трепета), что такой функцией является:

```
int main()
```

У этой функции может быть до трех параметров:

```
int main(int argc, char *argv[], char *env[])
```

- `argc` - количество параметров в командной строке (включая имя программы),
- `argv` - массив строк-параметров (argv[0] - имя программы),
- `env` - массив строк-переменных окружения.

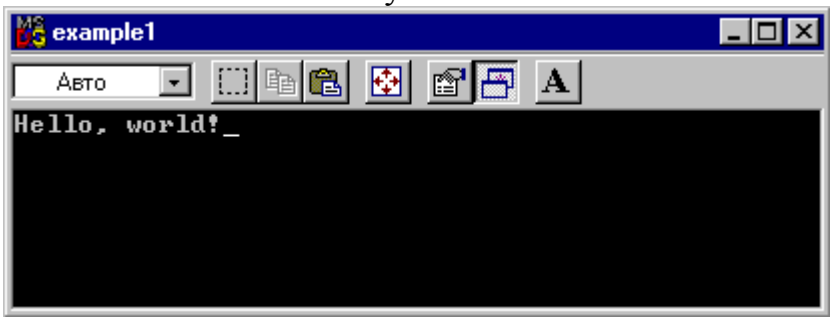
Многие компиляторы для Windows "понимают" такую стартовую функцию. Однако при этом они создают хотя и 32-битное, но консольное приложение. Пример 1 (example1.cpp):

```
#include <stdio.h>
int main() {
 printf("Hello, world!");
 getc(stdin);
 return 0;
}
```

Компилируем:

```
bcc32 example1.cpp
```

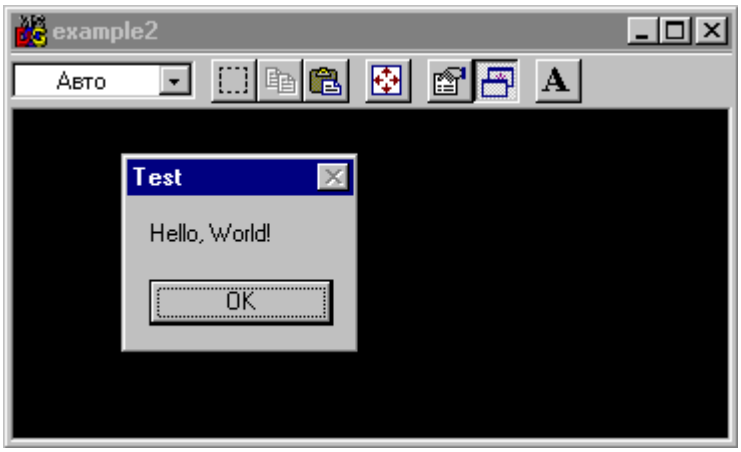
Запускаем:



При использовании стандартных библиотек (stdio, stdlib и т. п.) вам не потребуется никаких лишних телодвижений по сравнению с обычными методами написания программ на Си. Если же ваша цель - 32-битное приложение с графическим интерфейсом, то черное консольное окошко будет вас раздражать. Пример (example2.cpp):

```
#include <windows.h>
```

```
int main() {
 MessageBox(NULL,"Hello, World!","Test",MB_OK);
 return 0;
}
```



В общем, чтобы получить нормальное приложение без каких-либо "довесков" типа консольного окошка, используется другая стартовая функция:

```
int WINAPI WinMain (HINSTANCE hInst, HINSTANCE hpi, LPSTR cmdline, int ss)
```

- `hInst` - дескриптор для данного экземпляра программы,
- `hpi` - в Win32 не используется (всегда NULL),
- `cmdline` - командная строка,
- `ss` - код состояния главного окна.

Самостоятельная работа 2

Прямоугольники, регионы и пути

Графическая библиотека интерфейса GDI (Graphic Device Interface) имеет большой набор функций, работающих с регионами (regions) и путями (paths). Эти функции были введены в Windows NT и существенно улучшают возможности графических построений. Также были добавлены функции для работы с прямоугольниками. Прямоугольники (rectangles) широко используются в графике, поэтому представля-

ет интерес набор функций для манипуляций с ними: BOOL WINAPI CopyRect(LPRECT lprcDst, CONST RECT *lprcSrc);

Функция копирует один прямоугольник в другой.

 BOOL EqualRect(CONST RECT *lprc1,CONST RECT *lprc2);

Функция определяет, равны ли два прямоугольника, сравнивая их координаты. BOOL InflateRect(LPRECT lprc, int dx, int dy);

Функция увеличивает или уменьшает ширину и высоту указанного прямоугольника. int WINAPI FillRect(HDC hDC, CONST RECT *lprc, HBRUSH hbr); Функция заполняет прямоугольник кистью hbr. int WINAPI FrameRect(HDC hDC, CONST RECT *lprc, HBRUSH hbr); Функция рисует контур прямоугольника кистью hbr.

 BOOL WINAPI IntersectRect(LPRECT lprcDst, CONST RECT *lprcSrc1, CONST

RECT *lprcSrc2);

Функция выполняет пересечение двух прямоугольников.

 BOOL WINAPI InvertRect(HDC hDC, CONST RECT *lprc);

Выполняется побитовая инверсия цвета прямоугольника.

 BOOL WINAPI IsRectEmpty(CONST RECT *lprc);

Осуществляется определение, является ли прямоугольник пустым?

 BOOL WINAPI OffsetRect(LPRECT lprc, int dx, int dy);

Выполняется перемещение координат прямоугольника на dx и dy.

 BOOL WINAPI PtInRect(CONST RECT *lprc, POINT pt);

Определение, содержится ли точка pt внутри прямоугольника *lprc.Интерфейс Windows-приложения 47

 BOOL WINAPI SetRect(LPRECT lprc, int xLeft, int yTop, int xRight,int

yBottom);

Задание полей структуры прямоугольника.

 BOOL WINAPI SetRectEmpty(LPRECT lprc);

Установка полей структуры прямоугольника в ноль.

 BOOL SubtractRect(LPRECT lprcDst, CONST RECT *lprcSrc1, CONST RECT
*lprcSrc2);
Функция определяет координаты прямоугольника, вычитая один прямоугольник
из другого.
 BOOL WINAPI UnionRect(LPRECT lprcDst, CONST RECT *lprcSrc1, CONST RECT
*lprcSrc2);
Осуществление объединения двух прямоугольников.
Прямоугольники удобны тем, что они рисуются кистью, которую нет необходимости устанавливать в качестве текущей. Регион — это область экрана, представляющая собой комбинацию прямоугольников, полигонов и эллипсов. Регионы можно использовать для заливки сложных фигур, а также для установки области отсечения, т. е. области вывода. Простейшие регионы — прямоугольный и эллиптический — создаются функциями:
HRGN WINAPI CreateRectRgn(int x1, int y1, int x2, int y2);
HRGN WINAPI CreateRectRgnIndirect(CONST RECT *lprect);
HRGN WINAPI CreateEllipticRgn(int x1, int y1, int x2, int y2);
HRGN WINAPI CreateEllipticRgnIndirect(CONST RECT *lprect);
Функции возвращают дескриптор созданного региона.
По завершении работы регион должен быть удален функцией DeleteObject(). Из множества функций, работающих с регионами, рассмотрим лишь несколько:
 int WINAPI CombineRgn(HRGN hrgnDst,HRGN hrgnSrc1,HRGN hrgnSrc2, int iMode); Функция объединяет два региона hrgnSrc1 и hrgnSrc2, результат помещает в hrgnDst. Все три региона должны быть действительны. Параметр iMode определяет, как объединяются 2 региона:

iMode Новый регион

RGN_AND Область пересечения двух исходных регионов

RGN_OR Объединение двух исходных регионов

RGN_XOR Объединение двух исходных регионов за исключением области пе ресечения

RGN_DIFF Часть региона hrgnSrc1, не входящая в регион hrgnSrc2 RGN_COPY Копия региона hrgnSrc148 Глава 1

 BOOL WINAPI FillRgn(HDC hdc, HRGN hrgn, HBRUSH hbr);

Функция закрашивает область hrgn кистью hbr.

BOOL WINAPI PaintRgn(HDC hdc, HRGN hrgn);

Функция закрашивает область hrgn текущей кистью.

int WINAPI OffsetRgn(HRGN hrgn, int dx, int dy);

Функция смещает регион hrgn на dx и dy.

BOOL WINAPI PtInRegion(HRGN hrgn, int x, int y);

Функция определяет, входит ли точка (x, y) в регион hrgn?

BOOL RectInRegion(HRGN hrgn, CONST RECT *lprc);

Функция определяет, лежит ли какая-либо часть указанного прямоугольника в пределах границ региона.

int WINAPI SetPolyFillMode(HDC hdc, int mode);

Функция устанавливает режим закрашивания перекрывающихся областей. При значении параметра mode — ALTERNATE перекрывающиеся области не закраши ваются, если же значение параметра WINDING — будет закрашена вся фигура.

Рассмотрим пример графического построения с использованием регионов (лис-тинг 1.11). Фигуру, изображенную на рис. 1.17, без использования регионов построить было бы сложно.

Листинг 1.11. Использование регионов для графических построений

```
RECT pRect = {-100, -100, 100, 100};
RECT pEllips = {-120, -80, 120, 80};
RECT pSm = {-60, -40, 60, 40};
```

```
const int WIDTH = 400;
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
PAINTSTRUCT ps;Интерфейс Windows-приложения 49
HDC hdc;
static int sx, sy;
HRGN hRgnEllipse;
HRGN hRgn;
static HBRUSH hBrush;
switch (message)
{
case WM_SIZE:
sx = LOWORD(lParam);
sy = HIWORD(lParam);
break;
case WM_CREATE:
hBrush = CreateSolidBrush(RGB(0, 0, 255));
break;
case WM_PAINT:
hdc = BeginPaint(hWnd, &ps);
SetMapMode(hdc, MM_ANISOTROPIC);
SetWindowExtEx(hdc, WIDTH, - WIDTH, NULL);
SetViewportExtEx(hdc, sx, sy, NULL);
SetViewportOrgEx(hdc, sx/2, sy/2, NULL);
hRgn = CreateRectRgnIndirect(&pRect);
hRgnEllipse = CreateEllipticRgnIndirect(&pEllips);
CombineRgn(hRgn, hRgn, hRgnEllipse, RGN_DIFF);
DeleteObject(hRgnEllipse);
hRgnEllipse = CreateEllipticRgnIndirect(&pSm);
CombineRgn(hRgn, hRgn, hRgnEllipse, RGN_OR);
DeleteObject(hRgnEllipse);
FillRgn(hdc, hRgn, hBrush);
DeleteObject(hRgn);
EndPaint(hWnd, &ps);
break;
case WM_DESTROY: PostQuitMessage(0); break;
default: return DefWindowProc(hWnd, message, wParam, lParam);
}
return 0;
}
```

Опишем в программе переменные типа RECT для задания координат прямоуголь-

ника и двух эллипсов. В сообщении WM_PAINT установим логическую систему ко-
ординат размером 400 400 с началом в центре окна. Строим прямоугольный реги-
он hRgn и эллиптический hRgnEllipse, после чего объединяем их:
CombineRgn(hRgn, hRgn, hRgnEllipse, RGN_DIFF);50 Глава 1
Режим RGN_DIFF обеспечит "вычитание" области эллиптического региона из пря-
моугольного. Теперь строим еще один эллиптический регион и "складываем" его с предыдущим регионом. Отображаем полученную фигуру, закрашивая ее синей кистью:
FillRgn(hdc, hRgn, hBrush); Не забываем, для экономии памяти, удалить регионы, как только надобность в них отпадает. Путь — это набор прямых линий и кривых. Его можно применять для графических построений, а также конвертировать в регион и использовать для отсечения. Рассмотрим пример использования путей для графики (листинг 1.12), вот некото рые из необходимых функций: HDC WINAPI BeginPath(HDC hdc);
Открывает путь, теперь графические функции в окно ничего не выводят, а стро ят путь. HDC WINAPI CloseFigure(HDC hdc);
Закрывает открытую фигуру в пути. Замыкает первую и последнюю точки.
HDC WINAPI EndPath(HDC hdc);
Закрывает путь и помещает его в контекст устройства.
HDC WINAPI FillPath(HDC hdc);
Закрашивает текущей кистью область, ограниченную путем.
HRGN WINAPI PathToRegion(HDC hdc);
Преобразует путь в область, возвращает ее дескриптор.
HDC WINAPI StrokePath(HDC hdc);
Обводит путь текущим пером.

Самостоятельная работа 3

Рассмотрим листинг, представленный ниже:

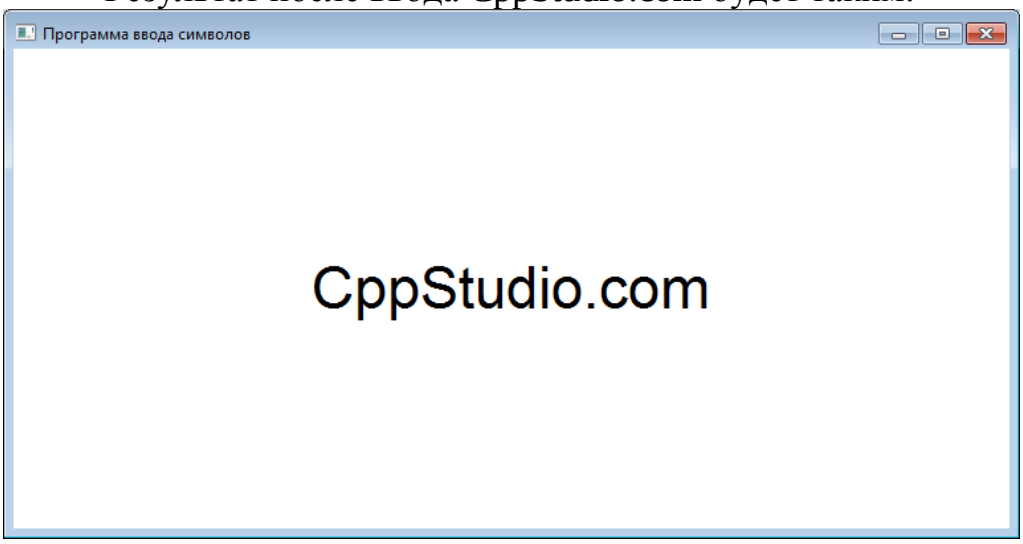
```
LRESULT CALLBACK WindowProcess(HWND hWindow, UINT uMessage,
            WPARAM wParameter, LPARAM lParameter)
{
   HDC hDeviceContext;
   PAINTSTRUCT paintStruct;
   RECT rectPlace;
   HFONT hFont;

   static PTCHAR text;
   static int size = 0;

   switch (uMessage)
   {
   case WM_CREATE:
      text=(PTCHAR)GlobalAlloc(GPTR,50000*sizeof(TCHAR));
      break;
   case WM_PAINT:
      hDeviceContext = BeginPaint(hWindow, &paintStruct);
      GetClientRect(hWindow, &rectPlace);
      SetTextColor(hDeviceContext, NULL);
      hFont=CreateFont(50,0,0,0,0,0,0,0,
            DEFAULT_CHARSET,
            0,0,0,VARIABLE_PITCH,
            "Arial Bold");
      SelectObject(hDeviceContext,hFont);
      if(wParameter != VK_RETURN)
         DrawText(hDeviceContext,
            (LPCSTR)text,
            size, &rectPlace,
            DT_SINGLELINE|DT_CENTER|DT_VCENTER);
      EndPaint(hWindow, &paintStruct);
      break;
   case WM_CHAR:
      switch(wParameter)
      {
      case VK_RETURN:
         size=0;
         break;
      default:
         text[size]=(char)wParameter;
         size++;
```

```
            break;
        }
        InvalidateRect(hWindow, NULL, TRUE);
              break;
    case WM_DESTROY:
        PostQuitMessage(NULL);
        GlobalFree((HGLOBAL)text);
        break;
    default:
        return DefWindowProc(hWindow, uMessage, wParameter, lParameter);
    }
    return NULL;
}
```

Результат после ввода CppStudio.com будет таким:



Код для функции WinMain не приводится, так как он не менялся с предыдущих уроков. Данная программа позволяет вводить текст по мере возможности и стирать его клавишей ENTER.

Самостоятельная работа 4

Многие прикладные программы также используют диалоговые окна, чтобы отобразить информацию или параметры, в то время как пользователь работает в другом окне. Например, прикладные программы обработки текстов часто используют диалоговое окно с командой поиска фрагмента текста. До тех пор, пока прикладная программа ищет текст, диалоговое окно остается на экране. Пользователь может затем возвратиться в блок диалога и искать то же самое слово снова; или может изменить введенное в диалоговом окне и искать новое слово. Прикладные программы, которые используют блоки диалога таким образом, обычно создают его тогда, когда пользователь

выбирает команду и продолжают показывать его до тех пор, пока прикладная программа выполняется или пока пользователь явно не закроет диалоговое окно. Чтобы поддерживать использование диалоговых окон различными прикладными программами, Windows предоставляет два типа блока диалога: модальное и немодальное. Модальное диалоговое окно (modal dialog box) требует, чтобы пользователь предоставил информацию или отменил диалоговое окно перед разрешением продолжения работы прикладной программе. Приложения используют модальные блоки диалога вместе с командами, которые требуют дополнительной информации прежде, чем они могут продолжать действовать. Немодальное диалоговое окно (modeless dialog box) позволяет пользователю предоставлять информацию и возвращаться к предыдущей задаче без закрытия блока диалога. Модальные диалоговые окна более простые для управления, чем немодальные блоки диалога, потому что они создаются, исполняют свою задачу и разрушаются вызовом единственной функции. Чтобы создать или модальное или немодальное диалоговое окно, прикладная программа должна снабдить блок диалога шаблоном, чтобы описать стиль и содержание диалогового окна; приложение должно также снабдить блок диалога процедурой, чтобы выполнять задачи. Шаблон диалогового окна (**dialog box template**) - бинарное описание блока диалога и элементов управления, которое оно содержит. Разработчик может создать этот шаблон как ресурс, который будет загружен из исполняемого файла прикладной программы, или создать его в памяти, пока выполняется прикладная программа. Процедура диалогового окна (**dialog box procedure**) - определяемая программой функция повторного вызова, которую Windows вызывает, когда операционная система получает ввод данных для диалогового окна или задачу для выполнения в блоке диалога. Хотя процедура диалогового окна подобна оконной процедуре, у неё не те же самые обязанности.

Самостоятельная работа 5

Панель инструментов – это дочернее окно, расположенное под меню приложения, содержащее одну или несколько кнопок. Сами по себе кнопки не являются окнами. Они имеют одинаковые размеры и реализованы как графические объекты на поверхности окна панели инструментов. Традиционно кнопки панели инструментов соответствуют некоторым пунктам меню приложения, т.е. индентификаторы кнопок совпадают

с идентификаторами дублируемых пунктов меню. Кнопка, выбранная на панели инструментов, посылает сообщение WM_COMMAND родительскому окну. Кнопка может содержать, кроме изображения, текстовую метку, которая может находиться правее или ниже картинки. Как правило, кнопки на панели инструментов содержат только растровые изображения, а их назначение поясняется с помощью всплывающих окон подсказок. Кроме кнопок, панель инструментов может содержать и другие дочерние окна элементов управления, например комбинированный список (combo box). Встроенные элементы управления создаются с помощью функции CreateWindowEx().

Для добавления к приложению панели инструментов необходимо выполнить следующие действия:

1) определить ресурс растрового образа панели инструментов;

2) объявить и инициализировать массив структур типа TBBUTTON, содержащий информацию о кнопках панели инструментов;

3) вызвать функцию CreateToolBarEx() для создания и инициализации панели инструментов.

Для определения ресурса растрового образа панели инструментов необходимо в главном меню Visual Studio выполнить команду Project −> Add Resource −> Toolbar и нажать кнопку New. В результате будет открыто окно редактора панели инструментов. Для создаваемой кнопки следует нарисовать картинку и определить идентификатор кнопки. Если кнопка дублирует некоторый пункт меню, то идентификатор кнопки должен быть таким же, как у этого пункта.

Самостоятельная работа 6

Немодальные диалоговые окна должны быть выскакивающими окнами, у которых есть Системное меню, строка заголовка и тонкая рамка; то есть, шаблон блока диалога должен установить стилиWS_POPUP, WS_CAPTION, WS_BORDER и WS_SYSMENU. Windows автоматически не показывает диалоговое окно, если в шаблоне не установлен стиль WS_VISIBLE.

Прикладная программа создает немодальное диалоговое окно, используя функции или CreateDialog, или CreateDialogIndirect. CreateDialog требует названия или идентификатора ресурса, содержащего шаблон блока диалога; CreateDialogIndirect требует дескриптора объекта памяти, содержащего шаблон диалогового окна. Две другие функции, CreateDialogParam и CreateDialogIndirectParam, тоже создают немодальное диалоговое окно; функции, когда окно создается, посылают заданный параметр в процедуру блока диалога.

CreateDialog и другие создающие функции возвращают значение дескриптора родительского окна диалоговому окну. Прикладная программа и процедура блока диалога могут использовать этот дескриптор для управления диалоговым окном. Например, если в шаблоне блока диалога не определен стиль WS_VISIBLE, то приложение может показать диалоговое окно путем передачи дескриптора родительского окна в функцию ShowWindow.

Немодальное диалоговое окно не блокирует окно владельца, не передает какие-либо сообщения для него. Когда создается блок диалога, Windows делает его активным окном, однако пользователь или прикладная программа могут в любое время заменить активное окно. Если диалоговое окно становиться неактивным, оно остается в Z-последовательности выше окна владельца, даже если окно владелец активное.

Прикладная программа ответственна за извлечение и распределение входящих сообщений для диалогового окна. Большинство приложений используют для этого главный цикл сообщений. Чтобы дать возможность пользователю перемещаться и выбирать элементы управления, используя клавиатуру, как угодно, но прикладная программа должна вызвать функцию IsDialogMessage. Более подробную информацию об этой функции, смотри статье Интерфейс клавиатуры диалогового окна.

Немодальное диалоговое окно не может возвратить значение в прикладную программу, как это делает модальное диалоговое окно, однако процедура блока диалога может передать информацию в окно владельца, используя функцию SendMessage. Прикладная программа перед завершением работы должна разрушить все немодальные диалоговые окна. Она может разрушить немодальный блок диалога, используя функцию DestroyWindow. В большинстве случаев, процедура диалогового окна вызывает DestroyWindow в ответ на ввод пользователем данных, например таких как выбор кнопки Отменить (Cancel). Если пользователь не закрывает диалоговое окно таким способом, тогда прикладная программа должна вызвать DestroyWindow. DestroyWindow аннулирует дескриптор родительского окна для диалогового окна, так что любой более поздний вызов функции, которая использует этот дескриптор, возвратит значение ошибки. Чтобы не допустить ошибки, процедура диалогового окна должна оповестить владельца, что блок диалога был разрушен. Многие прикладные программы сохраняют глобальную переменную, которая содержит дескриптор для диалогового окна. Когда процедура блока диалога разрушает диалоговое окно, она также и глобальную переменную устанавливает в значение ПУСТО (NULL), показывая, что диалоговое окно больше не действует. Процедура блока диалога не должна вызывать функцию EndDialog, чтобы разрушить не модальное диалоговое окно.

Самостоятельная работа 7

Метафайл — это протокол обращений к функциям GDI, сохраненный в двоичном формате. Метафайлы имеют такое же значение для векторной графики, как и битовые образы для растровой графики. Метафайл состоит из набора записей, соответствующих вызовам графических функций, таких как создание и выбор в контекст устройства пера или кисти, рисование линий, фигур, вывод текста, и иных операций.

При воспроизведении метафайлов достигается такой же результат, как и при непосредственном использовании функций GDI. Разница между их воспроизведением и непосредственным вызовом функций состоит в том, что метафайлы могут храниться в памяти или в файлах на диске, загружаться и воспроизводиться приложением столько раз, сколько это нужно.

Метафайлы используются для передачи изображений между программами через буфер обмена, а также для сохранения изображений в виде файлов на диске. Для их хранения требуется значительно меньше места, чем для хранения растровых изображений. В то же время для отображения

метафайлов требуется обычно больше времени, чем для вывода растровых изображений.

Поскольку метафайл описывает изображение в терминах команд графического вывода, то изображение может быть масштабировано при воспроизведении без потери разрешения. Для битовых образов масштабирование всегда связано с потерей качества изображения.

Первоначальный 16-битный формат метафайлов WMF1 появился в Windows 2.O. Однако этот формат был аппаратно-зависимым и не содержал информацию о размерах изображения, поэтому его использование было связано со многими проблемами. В Win32 появился новый 32-битный формат EMF (Enhanced Metafile Format). Расширенные метафайлы содержат дополнительную информацию о размерах изображения и цветовой палитре, что обеспечивает их аппаратную независимость. Кроме того, они поддерживают все 32-битные функции рисования. Win32 API позволяет использовать файлы форматов WMF и EMF.

Поскольку в новых приложениях рекомендуется использовать формат EMF, то в дальнейшем будут рассматриваться только расширенные метафайлы.

Самостоятельная работа 8

Процессом (process) называется экземпляр программы, загруженной в память. Этот экземпляр может создавать нити (thread), которые представляют собой последовательность инструкций на выполнение. Важно понимать, что выполняются не процессы, а именно нити.

Причем любой процесс имеет хотя бы одну нить. Эта нить называется главной (основной) нитью приложения.

Так как практически всегда нитей гораздо больше, чем физических процессоров для их выполнения, то нити на самом деле выполняются не одновременно, а по очереди (распределение процессорного времени происходит именно между нитями). Но переключение между ними происходит так часто, что кажется, будто они выполняются параллельно.

В зависимости от ситуации нити могут находиться в трех состояниях. Во-первых, нить может выполняться, когда ей выделено процессорное время, т.е. она может находиться в состоянии активности. Во-вторых, она может быть неактивной и ожидать выделения процессора, т.е. быть в состоянии

готовности. И есть еще третье, тоже очень важное состояние - состояние блокировки. Когда нить заблокирована, ей вообще не выделяется время. Обычно блокировка ставится на время ожидания какого-либо события. При возникновении этого события нить автоматически переводится из состояния блокировки в состояние готовности. Например, если одна нить выполняет вычисления, а другая должна ждать результатов, чтобы сохранить их на диск. Вторая могла бы использовать цикл типа "while(!isCalcFinished) continue;", но легко убедиться на практике, что во время выполнения этого цикла процессор занят на 100 % (это называется активным ожиданием). Таких вот циклов следует по возможности избегать, в чем оказывает неоценимую помощь механизм блокировки. Вторая нить может заблокировать себя до тех пор, пока первая не установит событие, сигнализирующее о том, что чтение окончено.

Синхронизация нитей в ОС Windows

В Windows реализована вытесняющая многозадачность - это значит, что в любой момент система может прервать выполнение одной нити и передать управление другой. Ранее, в Windows 3.1, использовался способ организации, называемый кооперативной многозадачностью: система ждала, пока нить сама не передаст ей управление и именно поэтому в случае зависания одного приложения приходилось перезагружать компьютер.

Все нити, принадлежащие одному процессу, разделяют некоторые общие ресурсы - такие, как адресное пространство оперативной памяти или открытые файлы. Эти ресурсы принадлежат всему процессу, а значит, и каждой его нити. Следовательно, каждая нить может работать с этими ресурсами без каких-либо ограничений. Но... Если одна нить еще не закончила работать с каким-либо общим ресурсом, а система переключилась на другую нить, использующую этот же ресурс, то результат работы этих нитей может чрезвычайно сильно отличаться от задуманного. Такие конфликты могут возникнуть и между нитями, принадлежащими различным процессам. Всегда, когда две или более нитей используют какой-либо общий ресурс, возникает эта проблема.

Пример. Несинхронизированная работа нитей: если временно приостановить выполнение нити вывода на экран (пауза), фоновая нить заполнения массива будет продолжать работать.

```
#include <windows.h>
#include <stdio.h>
int a[5];
HANDLE hThr;
unsigned long uThrID;
void Thread( void* pParams )
{
	int i, num = 0;
	while (1)
	{
		for (i=0; i<5; i++) a[i] = num;
		num++;
	}
}

int main( void )
{
	hThr=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)Thread,N
ULL,0,&uThrID);
	while(1)
	printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]);
	return 0;
}
```

Именно поэтому необходим механизм, позволяющий потокам согласовывать свою работу с общими ресурсами. Этот механизм получил название механизма синхронизации нитей (thread synchronization).

Этот механизм представляет собой набор объектов операционной системы, которые создаются и управляются программно, являются общими для всех нитей в системе (некоторые - для нитей, принадлежащих одному процессу) и используются для координирования доступа к ресурсам. В качестве ресурсов может выступать все, что может быть общим для двух и более нитей - файл на диске, порт, запись в базе данных, объект GDI, и даже глобальная переменная программы (которая может быть доступна из нитей, принадлежащих одному процессу).

Объектов синхронизации существует несколько, самые важные из них - это взаимоисключение (mutex), критическая секция (critical section), событие

(event) и семафор (semaphore). Каждый из этих объектов реализует свой способ синхронизации. Также в качестве объектов синхронизации могут использоваться сами процессы и нити (когда одна нить ждет завершения другой нити или процесса); а также файлы, коммуникационные устройства, консольный ввод и уведомления об изменении.

Любой объект синхронизации может находиться в так называемом сигнальном состоянии. Для каждого типа объектов это состояние имеет различный смысл. Нити могут проверять текущее состояние объекта и/или ждать изменения этого состояния и таким образом согласовывать свои действия. При этом гарантируется, что когда нить работает с объектами синхронизации (создает их, изменяет состояние) система не прервет ее выполнения, пока она не завершит это действие. Таким образом, все конечные операции с объектами синхронизации являются атомарными (неделимыми.

Работа с объектами синхронизации

Чтобы создать тот или иной объект синхронизации, производится вызов специальной функции WinAPI типа Create... (напр. CreateMutex). Этот вызов возвращает дескриптор объекта (HANDLE), который может использоваться всеми нитями, принадлежащими данному процессу. Есть возможность получить доступ к объекту синхронизации из другого процесса - либо унаследовав дескриптор этого объекта, либо, что предпочтительнее, воспользовавшись вызовом функции открытия объекта (Open...). После этого вызова процесс получит дескриптор, который в дальнейшем можно использовать для работы с объектом. Объекту, если только он не предназначен для использования внутри одного процесса, обязательно присваивается имя. Имена всех объектов должны быть различны (даже если они разного типа). Нельзя, например, создать событие и семафор с одним и тем же именем.

По имеющемуся дескриптору объекта можно определить его текущее состояние. Это делается с помощью т.н. ожидающих функций. Чаще всего используется функция WaitForSingleObject. Эта функция принимает два параметра, первый из которых - дескриптор объекта, второй - время ожидания в мсек. Функция возвращает WAIT_OBJECT_0, если объект находится в сигнальном состоянии, WAIT_TIMEOUT - если истекло время ожидания, и WAIT_ABANDONED, если объект-взаимоисключение не был

освобожден до того, как владеющая им нить завершилась. Если время ожидания указано равным нулю, функция возвращает результат немедленно, в противном случае она ждет в течение указанного промежутка времени. В случае, если состояние объекта станет сигнальным до истечения этого времени, функция вернет WAIT_OBJECT_0, в противном случае функция вернет WAIT_TIMEOUT. Если в качестве времени указана символическая константа INFINITE, то функция будет ждать неограниченно долго, пока состояние объекта не станет сигнальным.

Очень важен тот факт, что обращение к ожидающей функции блокирует текущую нить, т.е. пока нить находится в состоянии ожидания, ей не выделяется процессорного времени.

Критические секции

Объект-критическая секция помогает программисту выделить участок кода, где нить получает доступ к разделяемому ресурсу, и предотвратить одновременное использование ресурса. Перед использованием ресурса нить входит в критическую секцию (вызывает функцию EnterCriticalSection). Если после этого какая-либо другая нить попытается войти в ту же самую критическую секцию, ее выполнение приостановится, пока первая нить не покинет секцию с помощью вызова LeaveCriticalSection. Используется только для нитей одного процесса. Порядок входа в критическую секцию не определен.

Существует также функция TryEnterCriticalSection, которая проверяет, занята ли критическая секция в данный момент. С ее помощью нить в процессе ожидания доступа к ресурсу может не блокироваться, а выполнять какие-то полезные действия.

Пример. Синхронизация нитей с помощью критических секций.

```
#include <windows.h>
#include <stdio.h>
CRITICAL_SECTION cs;
int a[5];
HANDLE hThr;
unsigned long uThrID;
```

```
void Thread( void* pParams )
{
	int i, num = 0;
	while (1)
	{
		EnterCriticalSection( &cs );
		for (i=0; i<5; i++) a[i] = num;
		num++;
		LeaveCriticalSection( &cs );
	}
}

int main( void )
{
	InitializeCriticalSection( &cs );
	hThr=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)Thread,N
ULL,0,&uThrID);
	while(1)
	{
		EnterCriticalSection( &cs );
		printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]);
		LeaveCriticalSection( &cs );
	}
	return 0;
}
```

Взаимоисключения

Объекты-взаимоисключения (мьютексы, mutex - от MUTual EXclusion) позволяют координировать взаимное исключение доступа к разделяемому ресурсу. Сигнальное состояние объекта (т.е. состояние "установлен") соответствует моменту времени, когда объект не принадлежит ни одной нити и его можно "захватить". И наоборот, состояние "сброшен" (не сигнальное) соответствует моменту, когда какая-либо нить уже владеет этим объектом. Доступ к объекту разрешается, когда нить, владеющая объектом, освободит его.

Две (или более) нити могут создать мьютекс с одним и тем же именем, вызвав функцию CreateMutex. Первая нить действительно создает мьютекс, а следующие - получают дескриптор уже существующего объекта. Это дает

возможность нескольким нитям получить дескриптор одного и того же мьютекса, освобождая программиста от необходимости заботиться о том, кто в действительности создает мьютекс. Если используется такой подход, желательно установить флаг bInitialOwner в FALSE, иначе возникнут определенные трудности при определении действительного создателя мьютекса.

Несколько нитей могут получить дескриптор одного и того же мьютекса, что делает возможным взаимодействие между процессами. Можно использовать следующие механизмы такого подхода:

Дочерний процесс, созданный при помощи функции CreateProcess может наследовать дескриптор мьютекса в случае, если при создании мьютекса функцией CreateMutex был указан параметр lpMutexAttributes.

Нить может получить дубликат существующего мьютекса с помощью функции DuplicateHandle.

Нить может указать имя существующего мьютекса при вызове функций OpenMutex или CreateMutex.

Для того чтобы объявить взаимоисключение принадлежащим текущей нити, надо вызвать одну из ожидающих функций. Нить, которой принадлежит объект, может его "захватывать" повторно сколько угодно раз (это не приведет к самоблокировке), но столько же раз она должна будет его освобождать с помощью функции ReleaseMutex.

Для синхронизации нитей одного процесса более эффективно использование критических секций.

Пример. Синхронизация нитей с помощью мьютексов.

```
#include <windows.h>
#include <stdio.h>
HANDLE hMutex;
int a[5];
HANDLE hThr;
unsigned long uThrID;

void Thread( void* pParams )
```

```
{
	int i, num = 0;
	while (1)
	{
		WaitForSingleObject( hMutex, INFINITE );
		for (i=0; i<5; i++) a[i] = num;
		num++;
		ReleaseMutex( hMutex );
	}
}

int main( void )
{
	hMutex=CreateMutex( NULL, FALSE, NULL );
	hThr=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)Thread,N
ULL,0,&uThrID);
	while(1)
	{
		WaitForSingleObject( hMutex, INFINITE );
		printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]);
		ReleaseMutex( hMutex );
	}
return 0;
}
```

События

Объекты-события используются для уведомления ожидающих нитей о наступлении какого-либо события. Различают два вида событий - с ручным и автоматическим сбросом. Ручной сброс осуществляется функцией ResetEvent. События с ручным сбросом используются для уведомления сразу нескольких нитей. При использовании события с автосбросом уведомление получит и продолжит свое выполнение только одна ожидающая нить, остальные будут ожидать дальше.

Функция CreateEvent создает объект-событие, SetEvent - устанавливает событие в сигнальное состояние, ResetEvent - сбрасывает событие. Функция PulseEvent устанавливает событие, а после возобновления ожидающих это событие нитей (всех при ручном сбросе и только одной при автоматическом), сбрасывает его. Если ожидающих нитей нет, PulseEvent просто сбрасывает событие.

Пример. Синхронизация нитей с помощью событий.

```
#include <windows.h>
#include <stdio.h>
HANDLE hEvent1, hEvent2;
int a[5];
HANDLE hThr;
unsigned long uThrID;
void Thread( void* pParams )
{
	int i, num = 0;
	while (1)
	{
		WaitForSingleObject( hEvent2, INFINITE );
		for (i=0; i<5; i++) a[i] = num;
		num++;
		SetEvent( hEvent1 );
	}
}

int main( void )
{
	hEvent1=CreateEvent( NULL, FALSE, TRUE, NULL );
	hEvent2=CreateEvent( NULL, FALSE, FALSE, NULL );
	hThr=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)Thread,N
ULL,0,&uThrID);
	while(1)
	{
		WaitForSingleObject( hEvent1, INFINITE );
		printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]);
		SetEvent( hEvent2 );
	}
	return 0;
}
```

Семафоры

Объект-семафор - это фактически объект-взаимоисключение со счетчиком. Данный объект позволяет "захватить" себя определенному количеству нитей. После этого "захват" будет невозможен, пока одна из ранее "захвативших" семафор нитей не освободит его. Семафоры применяются для ограничения количества нитей, одновременно работающих с ресурсом. Объекту при инициализации передается максимальное число нитей, после каждого "захвата" счетчик семафора уменьшается. Сигнальному состоянию

соответствует значение счетчика больше нуля. Когда счетчик равен нулю, семафор считается не установленным (сброшенным).

Функция CreateSemaphore создает объект-семафор с указанием и максимально возможного начального его значения, OpenSemaphore – возвращает дескриптор существующего семафора, захват семафора производится с помощью ожидающих функций, при этом значение семафора уменьшается на единицу, ReleaseSemaphore - освобождение семафора с увеличением значения семафора на указанное в параметре число.

Пример. Синхронизация нитей с помощью семафоров.

```
#include <windows.h>
#include <stdio.h>
HANDLE hSem;
int a[5];
HANDLE hThr;
unsigned long uThrID;

void Thread( void* pParams )
{
	int i, num = 0;
	while (1)
	{
		WaitForSingleObject( hSem, INFINITE );
		for (i=0; i<5; i++) a[i] = num;
		num++;
		ReleaseSemaphore( hSem, 1, NULL );
	}
}

int main( void )
{
	hSem=CreateSemaphore( NULL, 1, 1, "MySemaphore1" );
	hThr=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)Thread,N
ULL,0,&uThrID);
	while(1)
	{
		WaitForSingleObject( hSem, INFINITE );
		printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]);
		ReleaseSemaphore( hSem, 1, NULL );
	}
return 0;
}
```

Защищенный доступ к переменным

Существует ряд функций, позволяющих работать с глобальными переменными из всех нитей, не заботясь о синхронизации, т.к. эти функции сами за ней следят – их выполнение атомарно. Это функции InterlockedIncrement, InterlockedDecrement, InterlockedExchange, InterlockedExchangeAdd и InterlockedCompareExchange. Например, функция InterlockedIncrement атомарно увеличивает значение 32-битной переменной на единицу, что удобно использовать для различных счетчиков.

Для получения полной информации о назначении, использовании и синтаксисе всех функций WIN32 API необходимо воспользоваться системой помощи MS SDK, входящей в состав сред программирования Borland Delphi или CBuilder, а также MSDN, поставляемым в составе системы программирования Visual C.

Самостоятельная работа 9

Процессом (process) называется экземпляр программы, загруженной в память. Этот экземпляр может создавать нити (thread), которые представляют собой последовательность инструкций на выполнение. Важно понимать, что выполняются не процессы, а именно нити.

Причем любой процесс имеет хотя бы одну нить. Эта нить называется главной (основной) нитью приложения.

Так как практически всегда нитей гораздо больше, чем физических процессоров для их выполнения, то нити на самом деле выполняются не одновременно, а по очереди (распределение процессорного времени происходит именно между нитями). Но переключение между ними происходит так часто, что кажется, будто они выполняются параллельно.

В зависимости от ситуации нити могут находиться в трех состояниях. Во-первых, нить может выполняться, когда ей выделено процессорное время, т.е. она может находиться в состоянии активности. Во-вторых, она может быть неактивной и ожидать выделения процессора, т.е. быть в состоянии готовности. И есть еще третье, тоже очень важное состояние - состояние блокировки. Когда нить заблокирована, ей вообще не выделяется время. Обычно блокировка ставится на время ожидания какого-либо события. При возникновении этого события нить автоматически переводится из состояния

блокировки в состояние готовности. Например, если одна нить выполняет вычисления, а другая должна ждать результатов, чтобы сохранить их на диск. Вторая могла бы использовать цикл типа "while(!isCalcFinished) continue;", но легко убедиться на практике, что во время выполнения этого цикла процессор занят на 100 % (это называется активным ожиданием). Таких вот циклов следует по возможности избегать, в чем оказывает неоценимую помощь механизм блокировки. Вторая нить может заблокировать себя до тех пор, пока первая не установит событие, сигнализирующее о том, что чтение окончено.

Синхронизация нитей в ОС Windows

В Windows реализована вытесняющая многозадачность - это значит, что в любой момент система может прервать выполнение одной нити и передать управление другой. Ранее, в Windows 3.1, использовался способ организации, называемый кооперативной многозадачностью: система ждала, пока нить сама не передаст ей управление и именно поэтому в случае зависания одного приложения приходилось перезагружать компьютер.

Все нити, принадлежащие одному процессу, разделяют некоторые общие ресурсы - такие, как адресное пространство оперативной памяти или открытые файлы. Эти ресурсы принадлежат всему процессу, а значит, и каждой его нити. Следовательно, каждая нить может работать с этими ресурсами без каких-либо ограничений. Но... Если одна нить еще не закончила работать с каким-либо общим ресурсом, а система переключилась на другую нить, использующую этот же ресурс, то результат работы этих нитей может чрезвычайно сильно отличаться от задуманного. Такие конфликты могут возникнуть и между нитями, принадлежащими различным процессам. Всегда, когда две или более нитей используют какой-либо общий ресурс, возникает эта проблема.

Пример. Несинхронизированная работа нитей: если временно приостановить выполнение нити вывода на экран (пауза), фоновая нить заполнения массива будет продолжать работать.

```
#include <windows.h>
#include <stdio.h>
int a[5];
HANDLE hThr;
unsigned long uThrID;
void Thread( void* pParams )
{
	int i, num = 0;
	while (1)
	{
		for (i=0; i<5; i++) a[i] = num;
		num++;
	}
}

int main( void )
{
	hThr=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)Thread,N
ULL,0,&uThrID);
	while(1)
	printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]);
	return 0;
}
```

Именно поэтому необходим механизм, позволяющий потокам согласовывать свою работу с общими ресурсами. Этот механизм получил название механизма синхронизации нитей (thread synchronization).

Этот механизм представляет собой набор объектов операционной системы, которые создаются и управляются программно, являются общими для всех нитей в системе (некоторые - для нитей, принадлежащих одному процессу) и используются для координирования доступа к ресурсам. В качестве ресурсов может выступать все, что может быть общим для двух и более нитей - файл на диске, порт, запись в базе данных, объект GDI, и даже глобальная переменная программы (которая может быть доступна из нитей, принадлежащих одному процессу).

Объектов синхронизации существует несколько, самые важные из них - это взаимоисключение (mutex), критическая секция (critical section), событие (event) и семафор (semaphore). Каждый из этих объектов реализует свой способ синхронизации. Также в качестве объектов синхронизации могут использоваться сами процессы и нити (когда одна нить ждет завершения другой нити или процесса); а также файлы, коммуникационные устройства, консольный ввод и уведомления об изменении.

Любой объект синхронизации может находиться в так называемом сигнальном состоянии. Для каждого типа объектов это состояние имеет различный смысл. Нити могут проверять текущее состояние объекта и/или ждать изменения этого состояния и таким образом согласовывать свои действия. При этом гарантируется, что когда нить работает с объектами синхронизации (создает их, изменяет состояние) система не прервет ее выполнения, пока она не завершит это действие. Таким образом, все конечные операции с объектами синхронизации являются атомарными (неделимыми.

Работа с объектами синхронизации

Чтобы создать тот или иной объект синхронизации, производится вызов специальной функции WinAPI типа Create... (напр. CreateMutex). Этот вызов возвращает дескриптор объекта (HANDLE), который может использоваться всеми нитями, принадлежащими данному процессу. Есть возможность получить доступ к объекту синхронизации из другого процесса - либо унаследовав дескриптор этого объекта, либо, что предпочтительнее, воспользовавшись вызовом функции открытия объекта (Open...). После этого вызова процесс получит дескриптор, который в дальнейшем можно использовать для работы с объектом. Объекту, если только он не предназначен для использования внутри одного процесса, обязательно присваивается имя. Имена всех объектов должны быть различны (даже если

они разного типа). Нельзя, например, создать событие и семафор с одним и тем же именем.

По имеющемуся дескриптору объекта можно определить его текущее состояние. Это делается с помощью т.н. ожидающих функций. Чаще всего используется функция WaitForSingleObject. Эта функция принимает два параметра, первый из которых - дескриптор объекта, второй - время ожидания в мсек. Функция возвращает WAIT_OBJECT_0, если объект находится в сигнальном состоянии, WAIT_TIMEOUT - если истекло время ожидания, и WAIT_ABANDONED, если объект-взаимоисключение не был освобожден до того, как владеющая им нить завершилась. Если время ожидания указано равным нулю, функция возвращает результат немедленно, в противном случае она ждет в течение указанного промежутка времени. В случае, если состояние объекта станет сигнальным до истечения этого времени, функция вернет WAIT_OBJECT_0, в противном случае функция вернет WAIT_TIMEOUT. Если в качестве времени указана символическая константа INFINITE, то функция будет ждать неограниченно долго, пока состояние объекта не станет сигнальным.

Очень важен тот факт, что обращение к ожидающей функции блокирует текущую нить, т.е. пока нить находится в состоянии ожидания, ей не выделяется процессорного времени.

Критические секции

Объект-критическая секция помогает программисту выделить участок кода, где нить получает доступ к разделяемому ресурсу, и предотвратить одновременное использование ресурса. Перед использованием ресурса нить входит в критическую секцию (вызывает функцию EnterCriticalSection). Если после этого какая-либо другая нить попытается войти в ту же самую критическую секцию, ее выполнение приостановится, пока первая нить не покинет секцию с помощью вызова LeaveCriticalSection. Используется только

для нитей одного процесса. Порядок входа в критическую секцию не определен.

Существует также функция TryEnterCriticalSection, которая проверяет, занята ли критическая секция в данный момент. С ее помощью нить в процессе ожидания доступа к ресурсу может не блокироваться, а выполнять какие-то полезные действия.

Пример. Синхронизация нитей с помощью критических секций.

```
#include <windows.h>
#include <stdio.h>
CRITICAL_SECTION cs;
int a[5];
HANDLE hThr;
unsigned long uThrID;

void Thread( void* pParams )
{
	int i, num = 0;
	while (1)
	{
		EnterCriticalSection( &cs );
		for (i=0; i<5; i++) a[i] = num;
		num++;
		LeaveCriticalSection( &cs );
	}
}

int main( void )
{
```

```
	InitializeCriticalSection( &cs );
	hThr=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)Thread,NULL,0,&uThrID);
	while(1)
	{
		EnterCriticalSection( &cs );
		printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]);
		LeaveCriticalSection( &cs );
	}
	return 0;
}
```

Взаимоисключения

Объекты-взаимоисключения (мьютексы, mutex - от MUTual EXclusion) позволяют координировать взаимное исключение доступа к разделяемому ресурсу. Сигнальное состояние объекта (т.е. состояние "установлен") соответствует моменту времени, когда объект не принадлежит ни одной нити и его можно "захватить". И наоборот, состояние "сброшен" (не сигнальное) соответствует моменту, когда какая-либо нить уже владеет этим объектом. Доступ к объекту разрешается, когда нить, владеющая объектом, освободит его.

Две (или более) нити могут создать мьютекс с одним и тем же именем, вызвав функцию CreateMutex. Первая нить действительно создает мьютекс, а следующие - получают дескриптор уже существующего объекта. Это дает возможность нескольким нитям получить дескриптор одного и того же мьютекса, освобождая программиста от необходимости заботиться о том, кто в действительности создает мьютекс. Если используется такой подход, желательно установить флаг bInitialOwner в FALSE, иначе возникнут определенные трудности при определении действительного создателя мьютекса.

Несколько нитей могут получить дескриптор одного и того же мьютекса, что делает возможным взаимодействие между процессами. Можно использовать следующие механизмы такого подхода:

Дочерний процесс, созданный при помощи функции CreateProcess может наследовать дескриптор мьютекса в случае, если при создании мьютекса функцией CreateMutex был указан параметр lpMutexAttributes.

Нить может получить дубликат существующего мьютекса с помощью функции DuplicateHandle.

Нить может указать имя существующего мьютекса при вызове функций OpenMutex или CreateMutex.

Для того чтобы объявить взаимоисключение принадлежащим текущей нити, надо вызвать одну из ожидающих функций. Нить, которой принадлежит объект, может его "захватывать" повторно сколько угодно раз (это не приведет к самоблокировке), но столько же раз она должна будет его освобождать с помощью функции ReleaseMutex.

Для синхронизации нитей одного процесса более эффективно использование критических секций.

Пример. Синхронизация нитей с помощью мьютексов.

```
#include <windows.h>
#include <stdio.h>
HANDLE hMutex;
int a[5];
HANDLE hThr;
unsigned long uThrID;

void Thread( void* pParams )
{
	int i, num = 0;
	while (1)
```

```
        {
                WaitForSingleObject( hMutex, INFINITE );
                for (i=0; i<5; i++) a[i] = num;
                num++;
                ReleaseMutex( hMutex );
        }
}

int main( void )
{
        hMutex=CreateMutex( NULL, FALSE, NULL );
        hThr=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)Thread,N
ULL,0,&uThrID);
        while(1)
        {
                WaitForSingleObject( hMutex, INFINITE );
                printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]);
                ReleaseMutex( hMutex );
        }
return 0;
}
```

События

Объекты-события используются для уведомления ожидающих нитей о наступлении какого-либо события. Различают два вида событий - с ручным и автоматическим сбросом. Ручной сброс осуществляется функцией ResetEvent. События с ручным сбросом используются для уведомления сразу нескольких нитей. При использовании события с автосбросом уведомление получит и продолжит свое выполнение только одна ожидающая нить, остальные будут ожидать дальше.

Функция CreateEvent создает объект-событие, SetEvent - устанавливает событие в сигнальное состояние, ResetEvent - сбрасывает событие. Функция PulseEvent устанавливает событие, а после возобновления ожидающих это событие нитей (всех при ручном сбросе и только одной при автоматическом), сбрасывает его. Если ожидающих нитей нет, PulseEvent просто сбрасывает событие.

Пример. Синхронизация нитей с помощью событий.

```
#include <windows.h>
#include <stdio.h>
HANDLE hEvent1, hEvent2;
int a[5];
HANDLE hThr;
unsigned long uThrID;
void Thread( void* pParams )
{
	int i, num = 0;
	while (1)
	{
		WaitForSingleObject( hEvent2, INFINITE );
		for (i=0; i<5; i++) a[i] = num;
		num++;
		SetEvent( hEvent1 );
	}
}

int main( void )
{
	hEvent1=CreateEvent( NULL, FALSE, TRUE, NULL );
```

```
	hEvent2=CreateEvent( NULL, FALSE, FALSE, NULL );
	hThr=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)Thread,N
ULL,0,&uThrID);
	while(1)
	{
		WaitForSingleObject( hEvent1, INFINITE );
		printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]);
		SetEvent( hEvent2 );
	}
	return 0;
}
```

Семафоры

Объект-семафор - это фактически объект-взаимоисключение со счетчиком. Данный объект позволяет "захватить" себя определенному количеству нитей. После этого "захват" будет невозможен, пока одна из ранее "захвативших" семафор нитей не освободит его. Семафоры применяются для ограничения количества нитей, одновременно работающих с ресурсом. Объекту при инициализации передается максимальное число нитей, после каждого "захвата" счетчик семафора уменьшается. Сигнальному состоянию соответствует значение счетчика больше нуля. Когда счетчик равен нулю, семафор считается не установленным (сброшенным).

Функция CreateSemaphore создает объект-семафор с указанием и максимально возможного начального его значения, OpenSemaphore – возвращает дескриптор существующего семафора, захват семафора производится с помощью ожидающих функций, при этом значение семафора уменьшается на единицу, ReleaseSemaphore - освобождение семафора с увеличением значения семафора на указанное в параметре число.

Пример. Синхронизация нитей с помощью семафоров.

```
#include <windows.h>
```

```
#include <stdio.h>
HANDLE hSem;
int a[5];
HANDLE hThr;
unsigned long uThrID;

void Thread( void* pParams )
{
	int i, num = 0;
	while (1)
	{
		WaitForSingleObject( hSem, INFINITE );
		for (i=0; i<5; i++) a[i] = num;
		num++;
		ReleaseSemaphore( hSem, 1, NULL );
	}
}

int main( void )
{
	hSem=CreateSemaphore( NULL, 1, 1, "MySemaphore1" );
	hThr=CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)Thread,N
ULL,0,&uThrID);
	while(1)
	{
		WaitForSingleObject( hSem, INFINITE );
		printf("%d %d %d %d %d\n", a[0], a[1], a[2], a[3], a[4]);
		ReleaseSemaphore( hSem, 1, NULL );
	}
return 0;
```

}

Защищенный доступ к переменным

Существует ряд функций, позволяющих работать с глобальными переменными из всех нитей, не заботясь о синхронизации, т.к. эти функции сами за ней следят – их выполнение атомарно. Это функции InterlockedIncrement, InterlockedDecrement, InterlockedExchange, InterlockedExchangeAdd и InterlockedCompareExchange. Например, функция InterlockedIncrement атомарно увеличивает значение 32-битной переменной на единицу, что удобно использовать для различных счетчиков.

Для получения полной информации о назначении, использовании и синтаксисе всех функций WIN32 API необходимо воспользоваться системой помощи MS SDK, входящей в состав сред программирования Borland Delphi или CBuilder, а также MSDN, поставляемым в составе системы программирования Visual C.

Самостоятельная работа 10

В операционной системе Linux поддержка потоков обеспечена определенным набором типов языка программирования C и набором функций для выполнения операций над потоками. Поддержка потоков выполнения реализована в виде набора заголовочных файлов и библиотеки (libpthreads.so), подключаемой к программе на этапе ее компоновки. Прототипы функций для манипуляции с потоками описываются в файле pthread.h (#include <pthreads.h>). Ниже приводятся прототипы наиболее часто используемых функций вместе с пояснением их синтаксиса и выполняемых ими действий.

int pthread_create(pthread_t* thread, const pthread_attr_t* attr, void* (*routine) (void*), void* arg);

– создает поток в процессе с атрибутами, указанными в attr. При указании 0 в качестве attr будут использованы атрибуты по умолчанию, которые подходят для большинства случаев. В выходной параметр thread заносится дескриптор созданного потока. Созданный поток выполняет указанную функцию routine,

которой при вызове будет передан указанный аргумент arg. Функция routine может выглядеть следующим образом:

```
void * DoSomeWork(void *pointer)

{

int arg = (int) pointer;

arg++;

cout<<”Значение arg = ”<<arg<<endl;

return NULL;

}
```

void pthread_exit(void *value); – завершает вызывающий поток и возвращает значение value потоку, ожидающему завершения данного потока.

pthread_t pthread_self(); – возвращает дескриптор вызвавшего потока, вызывается в самом потоке.

int pthread_join(pthread_t thread, void** value_ptr); – переводит вызывающий поток (т.е. поток в котором она была вызвана) в состояние ожидания завершения указанного потока thread. В параметр value_ptr (если он отличен от 0, т.е. то что возвращает функция routine) заносится результат выполнения завершенного потока.

Средства синхронизации потоков

Как известно, в рамках приложения все потоки выполняются в одном адресном пространстве. В связи с этим встает проблема совместного использования общих переменных. Для ее решения требуются средства, позволяющие разграничить доступ потоков к таким разделяемым переменным, или к разделяемым ресурсам, поскольку в один момент времени только единственный поток должен работать с определенным разделяемым ресурсом. Сформулированная задача имеет название обеспечение взаимоисключения, а участки программного кода, в которых

потоки выполняют операции с разделяемыми ресурсами, называются критическими секциями.

С другой стороны, потокам может потребоваться кооперация не по данным, а по выполняемым действиям. Примером такой кооперации является ситуация, при которой потоку для продолжения своей работы требуется результат выполнения другого потока. В приведенном случае поток должен синхронизировать свои действия с другим(и) потоками по готовности данных. Другим примером кооперации по действиям является необходимость выполнения некоторой операции только одним из многих потоков, причем каким из них априорно неизвестно. Для решения рассмотренных задач и других, подобных им, используются специальные средства синхронизации потоков. Основные из них – это мьютексы, семафоры и условные переменные. Синхронизация и взаимоисключение обеспечиваются за счет атомарности выполняемых операций над мьютексами и семафорами. Атомарной называют операцию, которая не может быть прервана в ходе своего выполнения.

Мьютекс

Мьютекс позволяет потокам управлять доступом к данным. При использовании мьютекса только один поток в определенный момент времени может заблокировать мьютекс и получить доступ к разделяемому ресурсу («лицензию» на его использование). При завершении работы с ресурсом поток должен возвратить «лицензию», разблокировав мьютекс. Если какой-либо поток обратится к уже заблокированному мьютексу, то он будет вынужден ждать разблокировки мьютекса потоком, владеющим им.

Прототипы функций для выполнения операций над мьютексами описываются в файле pthread.h. Ниже приводятся прототипы наиболее часто используемых функций вместе с пояснением их синтаксиса и выполняемых ими действий.

pthread_mutex_init(pthread_mutex_t* mutex, const pthread_mutexattr_t* attr); – инициализирует мьютекс mutex с указанными атрибутами attr или с атрибутами по умолчанию (при указании 0 в качестве attr).

int pthread_mutex_destroy(pthread_mutex_t* mutex); – уничтожает мьютекс mutex.

int pthread_mutex_lock(pthread_mutex_t* mutex); – выполняет блокировку мьютекса mutex.

Если мьютекс уже заблокирован, то вызвавший поток будет заблокирован до разблокировки мьютекса.

int pthread_mutex_unlock(pthread_mutex_t* mutex); – разблокировка мьютекса mutex.

Семафор

Семафор предназначен для синхронизации потоков по действиям и по данным. Семафор – это защищенная переменная, значения которой можно опрашивать и менять только при помощи специальных операций P и V и операции инициализации. Семафор может принимать целое неотрицательное значение. При выполнении потоком операции P над семафором S значение семафора уменьшается на 1 при S>0 или поток блокируется, «ожидая на семафоре», при S=0. При выполнении операции V(S) происходит пробуждение одного из потоков, ожидающих на семафоре S, а если таковых нет, то значение семафора увеличивается на 1. Как следует из вышесказанного, при входе в критическую секцию поток должен выполнять операцию P, а при выходе из критической секции операцию V.

Прототипы функций для манипуляции с семафорами описываются в файле semaphore.h. Ниже приводятся прототипы функций вместе с пояснением их синтаксиса и выполняемых ими действий.

int sem_init(sem_t* sem, int pshared, unsigned int value); – инициализация семафора sem значением value. В качестве pshared всегда необходимо указывать 0.

int sem_wait(sem_t* sem); – «ожидание на семафоре». Выполнение потока блокируется до тех пор, пока значение семафора не станет положительным. При этом значение семафора уменьшается на 1.

int sem_post(sem_t* sem); – увеличивает значение семафора sem.

int sem_destroy(sem_t* sem); – уничтожает семафор sem.

int sem_trywait(sem_t* sem); – неблокирующий вариант функции sem_wait. При этом вместо блокировки вызвавшего потока функция возвращает управление с кодом ошибки в качестве результата работы.

Условная переменная

Условная переменная позволяет потокам ожидать выполнения некоторого условия (события), связанного с разделяемыми данными. Над условными переменными определены две основные операции: информирование о наступлении события и ожидание события. При выполнении операции «информирование» один из потоков, ожидающих на условной переменной, возобновляет свою работу.

Условная переменная всегда используется совместно с мьютексом. Перед выполнением операции «ожидание» поток должен заблокировать мьютекс. При выполнении операции «ожидание» указанный мьютекс автоматически разблокируется. Перед возобновлением ожидающего потока выполняется автоматическая блокировка мьютекса, позволяющая потоку войти в критическую секцию, после критической секции рекомендуется разблокировать мьютекс. При подачи сигнала другим потокам рекомендуется так же функцию «сигнализации» защитить мьютексом.

Прототипы функций для работы с условными переменными содержатся в файле pthread.h. Ниже приводятся прототипы функций вместе с пояснением их синтаксиса и выполняемых ими действий.

pthread_cond_init(pthread_cond_t* cond, const pthread_condattr_t* attr); – инициализирует условную переменную cond с указанными атрибутами attr или с атрибутами по умолчанию (при указании 0 в качестве attr).

int pthread_cond_destroy(pthread_cond_t* cond); – уничтожает условную переменную cond.

int pthread_cond_signal(pthread_cond_t* cond); – информирование о наступлении события потоков, ожидающих на условной переменной cond.

int pthread_cond_broadcast(pthread_cond_t* cond); – информирование о наступлении события потоков, ожидающих на условной переменной cond. При этом возобновлены будут все ожидающие потоки.

int pthread_cond_wait(pthread_cond_t* cond , pthread_mutex_t* mutex); – ожидание события на условной переменной cond.

Рассмотренных средств достаточно для решения разнообразных задач синхронизации потоков. Вместе с тем они обеспечивают взаимоисключение на низком уровне и не наполнены семантическим смыслом. При

непосредственном их использовании легко допустить ошибки различного вида: забыть выйти из критической секции, использовать примитив не по назначению, вложенное использование примитива и т. д. При этом операции с мьютексами, семафорам и условными переменными оказываются разбросанными по всему программному коду приложения, что повышает вероятность появления ошибки и усложняет ее поиск и устранение.