Министерство образования
Республики Беларусь

БЕЛОРУССКИЙ НАЦИОНАЛЬНЫЙ
ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Кафедра «Программное обеспечение вычислительной техники
и автоматизированных систем»

СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Лабораторный практикум

Минск
БНТУ
2012

Министерство образования Республики Беларусь
БЕЛОРУССКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Кафедра «Программное обеспечение вычислительной техники и автоматизированных систем»

СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Лабораторный практикум
для студентов специальностей 1-40 01 02
«Информационные системы и технологии» и 1-40 01 01
«Программное обеспечение информационных технологий»

Минск
БНТУ
2012

УДК 004.451 (076.5)
ББК 32.973– 018у7
С 40

Составитель *Н.А. Разорёнов*

Рецензенты:
А.А. Москаленко, О.В. Бугай

С 40 Системное программирование: лабораторный практикум для студентов специальностей 1-40 01 02 «Информационные системы и технологии» и 1-40 01 01 «Программное обеспечение информационных технологий» / сост. Н.А. Разорёнов. – Минск: БНТУ, 2012. – 81 с.

Приведен теоретический материал по выполнению лабораторных работ по дисциплине «Системное программирование». Рассматриваются вопросы организации и принципы программирования в операционных системах (ОС) семейства Windows, графический оконный интерфейс, использование аппаратных и программных средств современных ОС, предназначенных для управления памятью, динамические библиотеки, управление файлами, создание, управление и взаимодействие процессов и потоков.

Указания могут быть полезны студентам специальностей, связанных с программированием, и лицам, которые занимаются разработкой программного обеспечения на базе прикладного программного интерфейса операционных систем.

ISBN 978-985-525-767-8

© БНТУ, 2012

СОДЕРЖАНИЕ

Основные требования к оформлению и содержанию отчета о лабораторной работе. 4

Лабораторная работа № 1.
Управление потоками в Windows. 6

Лабораторная работа № 2.
Синхронизация потоков Windows. 14

Лабораторная работа № 3.
Управление памятью в Windows 19

Лабораторная работа № 4.
Создание и использование DLL 36

Лабораторная работа № 5.
Графика Windows. Основы управления выводом графической и текстовой информации на базе библиотеки GDI. 43

Лабораторная работа № 6.
Растровая графика. 49

Лабораторная работа № 7.
Передача информации между процессами 54

Лабораторная работа № 8.
Буфер обмена. 62

Лабораторная работа № 9.
Межпроцессорное взаимодействие. 68

Литература 72

Приложение. 73

Основные требования к оформлению и содержанию отчета о лабораторной работе

При оформлении отчета о работе следует соблюдать следующие требования:

1. Шрифт – Times New Roman, 12–14 пт, через полтора интервала. Параметры страницы: формат А4, левое поле 30 мм, правое поле 10 мм, верхнее и нижнее поля 20 мм. Абзацы 15–17 мм, одинаковые по всему тексту. Страницы следует нумеровать в верхнем правом углу. Номер страницы на титульном листе не ставится, но включается в общую нумерацию страниц.

2. Отчёт оформляется персонально и самостоятельно, представляется к защите в установленный срок в бумажном/электронном виде перед защитой лабораторной работы. Форма отчета устанавливается преподавателем. Выполненные лабораторные работы и отчеты сохраняются до конца семестра. Объем отчета 5–7 листов формата А4. Отчет может содержать приложения.

3. Отчет должен содержать следующие листы и пункты:

1-й лист – титульный лист (пример оформления титульного листа отчета приведен на рис. 1);

2-й и последующие листы отчета содержат пункты:

1) Цель работы.
2) Изучаемые вопросы. } берутся из описания работы
3) Постановка задачи.

4) Ход выполнения работы (содержит подпункты, комментирующие фрагменты кода разработанной программы, которые раскрывают изучаемый вопрос. В подпунктах допускается приводить краткие теоретические сведения, схемы, рисунки, фрагменты дампов изучаемых объектов и т. д.).

5) Результаты работы программного обеспечения.

6) Выводы (не менее шести пунктов).

7) Приложения (при необходимости).

БЕЛОРУССКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Факультет информационных технологий и робототехники

Кафедра «Программное обеспечение вычислительной техники и автоматизированных систем»

О Т Ч Е Т

о лабораторной работе № N

Дисциплина
«Системное программирование»

Тема
«Управление памятью в Windows»

Выполнил: студент гр.1073XX   Сидоров Н.Н.

Проверил: Разорёнов Н.А.

Минск 201x

Рис. 1. Пример оформления титульного листа отчета

Лабораторная работа № 1

УПРАВЛЕНИЕ ПОТОКАМИ В WINDOWS

Цель работы: изучить основы создания и управления потоками в ОС Windows.

Изучаемые вопросы

1. Виды потоков, состояния потока.
2. Структура CONTEXT.
3. Создание потока.
4. Относительный приоритет потока.
5. Потоковая функция.
6. Функции WinAPI для управления потоками.
7. Окончание потока.
8. Время выполнения потока.

Постановка задачи

Разработать многопоточное Win32-приложение, которое использует диалоговое окно для управления потоками процессов. Дизайн диалогового окна задается вариантом, преподавателем или самостоятельно. Для визуализации работы потоков использовать соответствующие элементы управления диалога, графику. Приложение должно содержать три потока. Предусмотреть вывод системной информации о потоках (например полей CONTEXT , временны параметры и т.д.). В отчете привести диаграмму состояния потоков, копии окон.

Теоретические сведения

Виды потоков

Поток – последовательность команд, обрабатываемых CPU. В рамках одного процесса может находиться один или несколько потоков. Процесс предоставляет ресурсы, поток – команды и данные для

обработки. Процесс содержащий один поток называется однопоточным, в противном случае – многопоточным.

Многопоточная модель охватывает 2 категории потоков и их комбинацию:

- потоки на уровне пользователя ULT (User Level Thread);
- потоки на уровне ядра KLT (Kernel Level Thread);
- комбинированная модель UKLT.

ULT управляются самим приложением. KLT управляется самим ядром через интерфейс прикладного программирования средств ядра ОС.

Структура CONTEXT

В структуре CONTEXT хранятся данные о состоянии регистров с учетом специфики конкретного процессора. Она используется системой для выполнения различных внутренних операций. В настоящее время такие структуры определены для процессоров Intel, MIPS, Alpha и PowerPC. Эта структура разбита на несколько разделов.

Раздел CONTEXT_CONTROL содержит управляющие регистры процессора: указатель команд, указатель стека, флаги и адрес возврата функции.

Раздел CONTEXT_INTEGER соответствует целочисленным регистрам процессора, CONTEXT_FLOATING_POINT – регистрам с плавающей точкой, CONTEXT_SEGMENTS – сегментным регистрам (только для x86), CONTEXT_DEBUG_REGISTERS – регистрам, предназначенным для отладки (только для x86), а CONTEXT_ EXTENDED_REGISTERS – дополнительным регистрам (только для x86).

Получить сведения о текущем состоянии регистров процессора. можно с помощью функции:

```
BOOL GetThreadContext( HANDLE hThread, PCONTEXT pContext);
```

Пример.

Объявляем структуру ct:

```
CONTEXT ct; //
```

Заполняем структуру CONTEXT с помощью функции ***GetThreadContext***:

```
GetThreadContext(hThread[num], &ct);
```

Описание структуры CONTEXT:

```
typedef struct _CONTEXT {
DWORD ContextFlags;
DWORD    Dr0; DWORD    Dr1;
DWORD    Dr2; DWORD    Dr3;
DWORD    Dr6; DWORD    Dr7;
DWORD    SegGs;       DWORD    SegFs;
DWORD    SegEs;       DWORD    SegDs;
DWORD    Edi; DWORD    Esi;
DWORD    Ebx; DWORD    Edx;
DWORD    Ecx; DWORD    Eax;
DWORD    Ebp; DWORD    EFlags;
DWORD    Esp; DWORD    SegSs;
BYTE   ExtendedRegisters[MAXIMUM_SUPPORTED_EXTENSION];
} CONTEXT;
```

Некоторые поля данной структуры отображаются в окне программы, приведенной на рисунке 1.1.

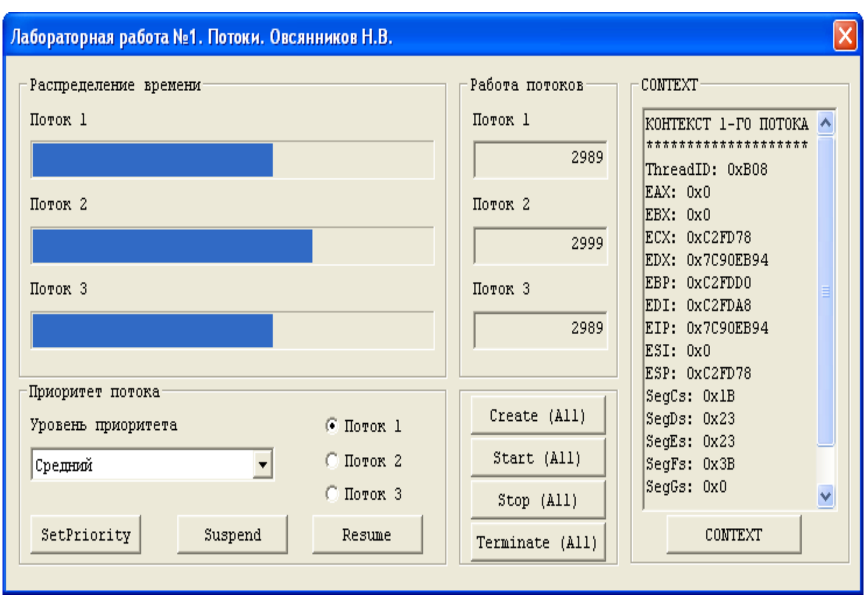


Рис. 1.1. Главное окно программы

Создание потока

Ниже приведен фрагмент кода, демонстрирующий создание потока в приостановленном состоянии.

```
hThread[0] = CreateThread(
  NULL,      // указатель на структуру  SECURITY_ATTRIBUTES
  0,         // определяет размер стека по умолчанию
  ThreadOneProc, //определяет адрес потоковой функции
  NULL,      // указатель на аргумент потоковой функции
  CREATE_SUSPENDED, // поток остановлен
  &dwThreadId[0]);   // возвращает идентификатор потока
```

Относительный приоритет потока

При создании потока функцией ***CreateThread***() существует возможность задать относительный приоритет потока. Если не указывать данный флаг, то по умолчанию потоку присвоится нормальный приоритет. Возможные приоритеты перечислены в таблице 1.1. Также задать приоритет потока можно с помощью функции ***SetThreadPriority***:

```
SetThreadPriority(hThread[i], THREAD_PRIORITY_NORMAL);
```

Таблица 1.1

Относительные приоритеты потоков

Приоритет	Флаговый идентификатор
Below normal (ниже обычного)	THREAD_PRIORITY_BELOW_NORMAL
Normal (обычный)	THREAD_PRIORITY_NORMAL
Above normal (выше обычного)	THREAD_PRIORITY_ABOVE_NORMAL
Highest (высокий)	THREAD_PRIORITY_HIGHEST
Realtime (реального времени)	THREAD_PRIORITY_TIME_CRITICAL
Lowest (низкий)	THREAD_PRIORITY_LOWEST
Idle (простаивающий)	THREAD_PRIORITY_IDLE

Потоковая функция

Функция потока принимает указатель на структуру с параметрами типа LPVOIDи возвращает значение типа DWORD .

```
DWORD WINAPI ThreadOneProc(LPVOID lpParam)
      {
            …
            ExitThread(0); // завершение потока
            return 0;
      }
```

Функция потока может выполнять любые задачи. Рано или поздно она закончит свою работу и вернет управление. В этот момент поток остановится, память, отведенная под его стек, будет освобождена, а счетчик пользователей его объекта ядра "поток" уменьшится на 1. Когда счетчик обнулится, этот объект ядра будет разрушен. Но, как и объект ядра "процесс", он может жить гораздо дольше, чем сопоставленный с ним поток.

Пример функции потока, визуализирующий свою работу трек баром, приведен ниже.

```
DWORD WINAPI TrackBarThread(PVOID pvParam)
{
   static int count=0;
   for(;;)//for(int i=0;i<=100;i++)
   {
         for(int i=0;i<=10000000;i++)
         {

         }
         count++;
         if(count==100) count=0;
         SendMessage(hTB, TBM_SETPOS,(WPARAM)
TRUE,(LPARAM) count);
   }
   return 0;
}
```

Функции Win32 для управления потоками, состояния потока

Создание потока:

```
CreateThread(NULL,0,ThreadOneProc,NULL, CREATE_SUSPENDED,
&dwThreadId[0]);.
```

Назначение потоку относительного приоритета:

```
SetThreadPriority(hThread[i], THREAD_PRIORITY_NORMAL);.
```

Возобновление потока: `ResumeThread(hThread[i]);.`

Приостановка потока: `SuspendThread(hThread[i]);.`

Получение времени создания, окончания потока и др.:

```
GetThreadTimes(hThread[i],&creationTime,&exitTime,
&kernelTime, &userTime);.
```

Ожидание завершения множества потоков:

```
WaitForMultipleObjects(3, hThread, TRUE, INFINITE);.
```

Получение контекста потока:

```
GetThreadContext(hThread[num], &ct);.
```

Завершение потока: `ExitThread(0);.`

Завершение потока извне: `TerminateThread(hThread[i],0);.`

Окончание потока

Поток можно завершить способами:

- функция потока возвращает управление (рекомендуемый способ);
- поток самоуничтожается вызовом функции ***ExitThread*** (нежелательный способ);
- один из потоков данного или стороннего процесса вызывает функцию ***TerminateThread*** (нежелательный способ);
- завершается процесс, содержащий данный поток (тоже нежелательно);
- какой-либо поток процесса вызывает функцию ***ExitProcess***();
- какой-либо поток вызывает функцию ***TerminateThread***() с дескриптором потока;
- какой-либо поток вызывает функцию ***TerminateProcess***() с дескриптором процесса.

Время выполнения потока

Временные показатели работы потока определяются функцией ***GetThreadTimes***, которая возвращает четыре временных параметра, приведенных в таблице 1.2.

```
BOOL GetThreadTimes(HANDLE hThread,
```

```
PFILETIME pftCreationTime,  PFILETIME pftExitTime,
PFILETIME pftKernelTime,    PFILETIME pftUserTime);
```

Таблица 1.2

Временные параметры *GetThreadTimes*

Показатель времени	Описание
Время создания (creation time)	Абсолютная величина, выраженная в интервалах по 100 нс. Отсчитывается с полуночи 1 января 1601 года по Гринвичу до момента создания потока
Время завершения (exit time)	Абсолютная величина, выраженная в интервалах по 100 нс. Отсчитывается с полуночи 1 января 1601 года по Гринвичу до момента завершения потока. Если поток все еще выполняется, этот показатель имеет неопределенное значение
Время выполнения ядра (kernel time)	Относительная величина, выраженная в интервалах по 100 нс. Сообщает время, затраченное этим потоком на выполнение кода операционной системы
Время выполнения User (User time)	Относительная величина, выраженная в интервалах по 100 нс. Сообщает время, затраченное потоком на выполнение кода приложения.

В программе определение и вывод времени создания потоков можно выполнить следующим образом:

```
GetThreadTimes(hThread[i],&creationTime,
&exitTime,&kernelTime,&userTime);
  // Дата и время создания процесса по Гринвичу
FileTimeToSystemTime(&creationTime, &stUTC);
  // Конвертируем время создания процесса в местное время
SystemTimeToTzSpecificLocalTime(NULL, &stUTC, &stLocal);
j+=wsprintf(buf+j, L"Время создания %d-го потока:
%02d/%02d/%d  %02d:%02d:%02d\r\n\n", i+1, stLocal.wDay,
stLocal.wMonth, stLocal.wYear, stLocal.wHour,
stLocal.wMinute, stLocal.wSecond);
  // Дата и время завершения процесса по Гринвичу
FileTimeToSystemTime(&exitTime, &stUTC);
  // Конвертируем время завершения процесса в местное время
```

```
SystemTimeToTzSpecificLocalTime(NULL, &stUTC, &stLocal);
j+=wsprintf(buf+j, L"Время завершения %d-го потока:
%02d/%02d/%d  %02d:%02d:%02d\r\n\n", i+1, stLocal.wDay,
stLocal.wMonth, stLocal.wYear, stLocal.wHour,
stLocal.wMinute, stLocal.wSecond);
```

На рисунке 1.2 приведен вывод временных показателей потоков.

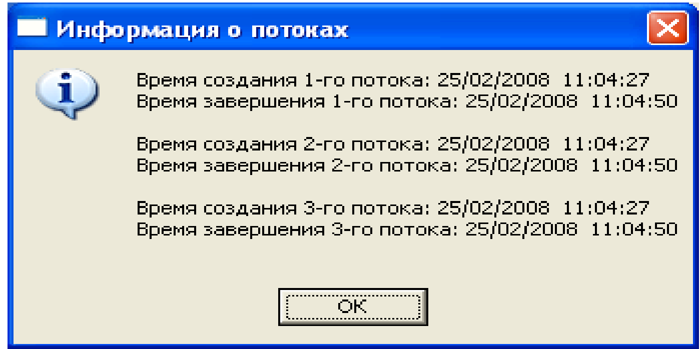


Рис. 1.2. Информация о времени создания потоков

Лабораторная работа № 2

СИНХРОНИЗАЦИЯ ПОТОКОВ В WINDOWS

Цель работы: Изучить основы синхронизации потоков в ОС Windows.

Изучаемые вопросы

1. Блокирующие функции. Защищённый доступ к переменным.
2. Критические секции.
3. Ожидающие функции.
4. Взаимоисключения.
5. События.
6. Семафоры.

Постановка задачи

Разработать многопоточное Win32-приложение, синхронизирующее работу трех вторичных потоков:

– первый помещает данные типа Type в хранилище на n элементов;
– второй сортирует данные в хранилище;
– третий извлекает данные из хранилища и визуализирует их.

Тип хранилища и тип данных задается преподавателем.

Теоретические сведения

Блокирующие функции

Большая часть синхронизации потоков связана с *атомарным доступом* (atomic access) – монопольным захватом ресурса обращающимся к нему потоком.

```
DWORD WINAPI FirstThread(LPVOID)
{
```

```
while (InterlockedExchange((PLONG)&g_fResourceInUse, TRUE)
== TRUE) Sleep(X);
	char buffer[3];
	GetWindowText(hEdit3,buffer,GetWindowTextLength(hEdit
3)+1);
	n=atoi(buffer);
	GetWindowText(hEdit4,buffer,GetWindowTextLength(hEdit
4)+1);
	m=atoi(buffer);
	for(short i=0;i<n;i++)
	{
		for(short j=0;j<m;j++)
		{
			sMatr[i][j]=rand()%32000-16000;
		}
	}
	InterlockedExchange((PLONG)&g_fResourceInUse, FALSE);
	return 0;
}
```

В этой функции постоянно "крутится" цикл *while*, в котором переменной *g_fResourceInUse* присваивается значение TRUE и проверяется ее предыдущее значение. Если оно было равно FALSE, значит, ресурс не был занят, но вызывающий поток только что занял его, на этом цикл завершается. В ином случае (значение было равно TRUE) ресурс занимал другой поток, и цикл повторяется.

Ожидающие функции

Wait-функции позволяют потоку в любой момент приостановиться и ждать освобождения какого-либо объекта ядра. Из всего семейства этих функций чаще всего используется ***WaitForSingleObject****:*

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

Первый параметр, *hObject,* идентифицирует объект ядра, поддерживающий состояния «свободен-занят». Второй параметр, *dwMilliseconds,* указывает, сколько времени (в миллисекундах) поток готов ждать освобождения объекта.

```
hThreadI=CreateThread(NULL,0,ThreadFuncI,0,0,&dwThreadIDI);
```

```
DWORD dw = WaitForSingleObject(hThreadI, 500);
switch (dw)
 {
   case WAIT_OBJECT_0:
       SendMessage(hResult,LB_RESETCONTENT,0,0L);
       for (i=0; i<=5;i++)
       SendMessage(hResult,LB_ADDSTRING,0,
(LPARAM)(LPSTR)szBuffer[i]);
   break;
   case WAIT_TIMEOUT:
   MessageBox(hWnd,"Не успели...","Предупреждение",MB_OK);
    break;
   case WAIT_FAILED:
   MessageBox(hWnd,"Нет такого описателя.",
"Предупреждение",MB_OK);
   break;
}
```

Критические секции

Критическая секция (critical section) – это небольшой участок кода, требующий монопольного доступа к каким-то общим данным. Она позволяет сделать так, чтобы единовременно только один поток получал доступ к определенному ресурсу.

```
CRITICAL_SECTION g_cs;
//инициализация структуры CRITICAL_SECTION
InitializeCriticalSection(&g_cs);

DWORD WINAPI FunctionThread(LPVOID)
   {
      . . .

   //начало работы с критической секцией
EnterCriticalSection(&g_cs);
      . . .

   //конец работы с критической секцией
LeaveCriticalSection(&g_cs);
      . . .

   }
```

Взаимоисключения

Объекты ядра "мьютексы" гарантируют потокам взаимоисключающий доступ к единственному ресурсу.
Мьютекс создается функцией ***CreateMutex***():

```
m_hMutex1 = CreateMutex(
NULL, //указатель на структуру SECURITY_ATTRIBUTES
FALSE,        // определяет начальное состояние мьютекса
NULL   // адрес строки (с нулевым символом в конце),
              // идентифицирующей мьютекс
       );
```

Функция ***ReleaseMutex***() переводит объект-мьютекс из занятого состояния в свободное.

События

События – разновидность объектов ядра. Они содержат счетчик числа пользователей (как и все объекты ядра) и две булевы переменные: одна сообщает тип данного объекта-события, другая – его состояние (свободен или занят).

События просто уведомляют об окончании какой-либо операции. Объекты-события бывают двух типов: со сбросом вручную (manual-reset events) и с автосбросом (auto-reset events). Первые позволяют возобновлять выполнение сразу нескольких ждущих потоков, вторые – только одного.

Пример создания и использования событий приведен ниже.

```
HANDLE g_hEvent1,g_hEvent2,g_hEvent3;
       //создание события с автосбросом, свободное
g_hEvent1 = CreateEvent(NULL, FALSE, TRUE, NULL);
       //создание события с автосбросом, занятое
g_hEvent2 = CreateEvent(NULL, FALSE, FALSE, NULL);
       . . .
       //ожидание освобождения события
WaitForSingleObject(g_hEvent2, INFINITE);
       . . .
       //установить g_hEvent2 в занятое состояние
ResetEvent(g_hEvent2);
    //установить g_hEvent3 в свободное состояние
SetEvent(g_hEvent3);
```

Семафоры

Объекты ядра "семафор" используются для учета ресурсов. Как и все объекты ядра, они содержат счетчик числа пользователей, но, кроме того, поддерживают два 32-битных значения со знаком: одно определяет максимальное число ресурсов (контролируемое семафором), другое используется как счетчик текущего числа ресурсов.

Пример создания и использования семафора приведен ниже.

```
HANDLE hSem1,hSem2,hSem3;
hSem1 = CreateSemaphore(
NULL, // указатель на структуру SECURITY_ATTRIBUTES
1,     // сколько из ресурсов доступно изначально
1,     //максимальное значение счетчика
NULL   // адрес строки, идентифицирующей семафор);

hSem2 = CreateSemaphore(NULL, 0, 1, NULL);
//создание занятого семофора
       . . .
WaitForSingleObject(hSem2, INFINITE);
       . . .
//увеличение значение счетчика текущего числа ресурсов
ReleaseSemaphore(
       hSem3, //указатель на объекта ядра «семафор»
       1,      // на сколько изменить счетчик
       NULL); // исходное значение счетчика ресурсов
```

ЗАДАНИЕ[1]

Постановка задачи

В буфере имеется N записей типа Type. К этим записям на чтение и на запись обращаются m потоков. Каждый из потоков может изменить содержимое записи. Существует поток I, который отображает на мониторе изменённые записи. Каждую запись может изменить только I-тый поток. Когда поток I отобразит последнюю измененную запись, остальные потоки могут продолжать модифицировать записи. Поток I запускается по нажатию клавиш ALT+I.

Реализовать задачу. Тип записи задается преподавателем.

---

[1] Данное задание выполняется студентами, которые претендуют на оценку «отлично»

Лабораторная работа № 3

УПРАВЛЕНИЕ ПАМЯТЬЮ В WINDOWS

Цель работы: Изучить основы распределения и управления памятью в Windows.

Изучаемые вопросы

1. Виртуальное адресное пространство:
   a. Виртуальное адресное пространство и физическая память.
   b. Состояние страниц.
   c. Атрибуты защиты.
   d. Границы выделения памяти.
   e. Регионы в адресном пространстве.
2. Работа с виртуальной памятью:
   a. Выделение памяти.
   b. Освобождение памяти.
   c. Изменение атрибутов защиты.
   d. Блокировка физической памяти RAM.
   e. Стек потока.
3. Кучи.
4. Стандартные библиотечные функции языка С.

Постановка задачи

Разработать приложение, которое предоставляет пользователю возможность управлять памятью в Windows, получать системную информацию о состоянии физической и виртуальной памяти, а также управлять выделением и освобождением виртуальной памяти.

В программе предусмотреть использование различных видов памяти:

1. Статически распределяемой.
2. Стековой памяти.
3. Динамически распределяемой памяти.
4. Регионов виртуальной памяти.

Теоретические сведения

Виртуальное адресное пространство и физическая память

Виртуальное адресное пространство каждого процесса значительно больше всей физической памяти, доступной для всех процессов. Общий объем памяти, доступной для всех исполняемых процессов, является суммой физической памяти и свободного места на диске, доступного для файла подкачки (paging file) – файла на диске, используемого для увеличения объема физической памяти. Реальная физическая память и виртуальное адресное пространство для каждого процесса организованны в страницы (pages) – модули памяти, чей размер зависит от процессора. Например, для процессоров семейства x86 размер страницы памяти равен 4 Кбайт.

Состояние страниц и атрибуты защиты

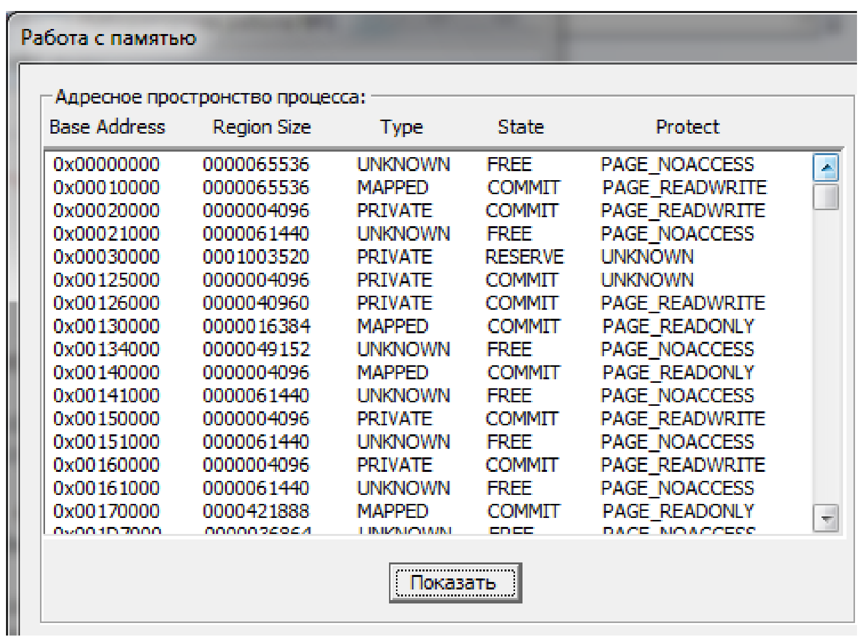
Страницы виртуального адресного пространства процессов могут находиться в одном из трех состояний:

— *свободное* (*free*) – свободная страница в данный момент не доступна, но ей можно передать физическую память или зарезервировать;

— *зарезервированное* (*reserved*) – зарезервированная страница – это блок виртуального адресного пространства процесса, который был зарезервирован для более позднего использования. Процесс не имеет доступа к зарезервированной странице и с ней не связана физическая память. Зарезервированная страница резервирует область виртуального адресного пространства, которая в дальнейшем не может быть использована другой функцией размещения памяти.

— *фиксированное* (*committed*) – фиксированная страница – это такая страница, для которой выделена физическая память (в ОЗУ или на диске). Она может быть защищена от доступа других процессов, может быть доступна только для чтения или для чтения и записи.

На рисунке 3.1 продемонстрировано окно приложения по распределению памяти процесса в Windows. Ниже приведен код для определения состояния страниц.



Base Address	Region Size	Type	State	Protect
0x00000000	0000065536	UNKNOWN	FREE	PAGE_NOACCESS
0x00010000	0000065536	MAPPED	COMMIT	PAGE_READWRITE
0x00020000	0000004096	PRIVATE	COMMIT	PAGE_READWRITE
0x00021000	0000061440	UNKNOWN	FREE	PAGE_NOACCESS
0x00030000	0001003520	PRIVATE	RESERVE	UNKNOWN
0x00125000	0000004096	PRIVATE	COMMIT	UNKNOWN
0x00126000	0000040960	PRIVATE	COMMIT	PAGE_READWRITE
0x00130000	0000016384	MAPPED	COMMIT	PAGE_READONLY
0x00134000	0000049152	UNKNOWN	FREE	PAGE_NOACCESS
0x00140000	0000004096	MAPPED	COMMIT	PAGE_READONLY
0x00141000	0000061440	UNKNOWN	FREE	PAGE_NOACCESS
0x00150000	0000004096	PRIVATE	COMMIT	PAGE_READWRITE
0x00151000	0000061440	UNKNOWN	FREE	PAGE_NOACCESS
0x00160000	0000004096	PRIVATE	COMMIT	PAGE_READWRITE
0x00161000	0000061440	UNKNOWN	FREE	PAGE_NOACCESS
0x00170000	0000421888	MAPPED	COMMIT	PAGE_READONLY

Рис. 3.1. Адресное пространство процесса

Пример.

```
//состояние страниц
switch(mem_info2.State)
{
case MEM_COMMIT:
        i+=sprintf(buf+i,"  COMMIT  ");
        break;
case MEM_FREE:
        i+=sprintf(buf+i," FREE    ");
        break;
case MEM_RESERVE:
        i+=sprintf(buf+i,"  RESERVE");
        break;
default:
        i+=sprintf(buf+i,"  UNKNOWN");
break;
        }
```

На рисунке 3.2 представлен интерфейс для выбора атрибутов защиты страниц памяти, где:

— PAGE_NOACCESS – попытки чтения, записи или исполнения содержимого памяти в этом регионе вызывают нарушение доступа;

— PAGE_READONLY – попытки записи или исполнения содержимого памяти в этом регионе вызывают нарушение доступа;

— PAGE_READWRITE – попытки исполнения содержимого памяти в этом регионе вызывают нарушение доступа;

— PAGE_EXECUTE – попытки чтения или записи в память этого региона вызывают нарушение доступа;

— PAGE_EXECUTE_READ – попытки записи в память этого региона вызывают нарушение доступа;

— PAGE_EXECUTE_READWRITE – данный регион допускает любые операции;

— PAGE_GUARD – страницы в этом регионе становятся защитными. Попытки прочитать или записать в защитную страницу приведут к возникновению исключительной ситуации и к сбросу защитного состояния. Таким образом защитная страница может служить одноразовым сигналом. Комбинируется с любыми атрибутами (кроме PAGE_NOACCESS);

— PAGE_NOCACHE – устанавливает невозможность кэширования зафиксированной области страниц. Комбинируется с любыми атрибутами (кроме PAGE_NOACCESS);

— PAGE_WRITECOPY (для функции ***VirtualProtect***) – попытки исполнения содержимого памяти в этом регионе вызывают нарушение доступа. Запись в память этого региона приводит к тому, что процессу предоставляется «личная» копия данной страницы физической памяти;

— PAGE_EXECUTE_WRITECOPY (для функции ***VirtualProtect***) – данный регион допускает любые операции. Запись в память этого региона приводит к тому, что процессу предоставляется «личная» копия данной страницы физической памяти.

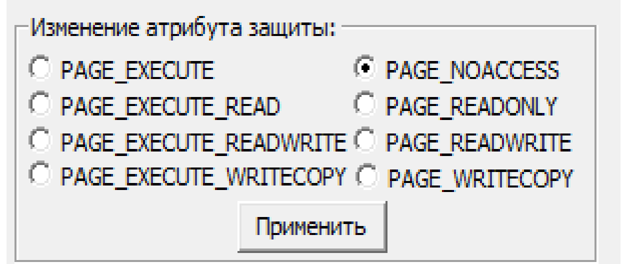


Рис. 3.2.. Выбор атрибута защиты памяти

Отдельным страницам физической памяти, выделяемым функцией ***VirtualAlloc***, можно определить атрибуты защиты, что приведено в нижележащем коде.

```
VirtualQuery(pMemory, &memInfo,
sizeof(MEMORY_BASIC_INFORMATION));
switch(memInfo.Protect)
{
case PAGE_EXECUTE:
        SendDlgItemMessage(hDlg, IDC_RADIO1, BM_SETCHECK,
BST_CHECKED, NULL);
break;
case PAGE_EXECUTE_READ:
        SendDlgItemMessage(hDlg, IDC_RADIO2, BM_SETCHECK,
BST_CHECKED, NULL);
break;
case PAGE_EXECUTE_READWRITE:
        SendDlgItemMessage(hDlg, IDC_RADIO3, BM_SETCHECK,
BST_CHECKED, NULL);
break;
case PAGE_EXECUTE_WRITECOPY:
        SendDlgItemMessage(hDlg, IDC_RADIO4, BM_SETCHECK,
BST_CHECKED, NULL);
break;
case PAGE_NOACCESS:
        SendDlgItemMessage(hDlg, IDC_RADIO5, BM_SETCHECK,
BST_CHECKED, NULL);
break;
case PAGE_READONLY:
        SendDlgItemMessage(hDlg, IDC_RADIO6, BM_SETCHECK,
BST_CHECKED, NULL);
break;
case PAGE_READWRITE:
```

```
      SendDlgItemMessage(hDlg, IDC_RADIO7, BM_SETCHECK,
BST_CHECKED, NULL);
break;
case PAGE_WRITECOPY:
      SendDlgItemMessage(hDlg, IDC_RADIO8, BM_SETCHECK,
BST_CHECKED, NULL);
break;
}
```

Границы выделения памяти

Граница выделения памяти обуславливаются гранулярностью обращения к памяти через страницы. Так как страница имеет кратность 4Кб, то граница выделения любого региона кратна этому значению. Возможные значения определяются граничными значениями доступных участков.

Для отслеживания состояния памяти можно использовать интерфейс приложения, приведенного на рисунке 3.3 и приведенный ниже код.

```
INT_PTR CALLBACK MemoryInfo(HWND hDlg, UINT message,
WPARAM wParam, LPARAM lParam)
{
   UNREFERENCED_PARAMETER(lParam);
   SYSTEM_INFO sys_info;//структура, содержащая
 //системную информацию (размер страницы, гранулярность
// выделения памяти и др.0
   MEMORYSTATUS mem_status = { sizeof(mem_status) };
   char buff[256];
   string str1="Размер страницы памяти:";
   string str2="Байт свободно в страничном файле:";

   switch (message)
   {
   case WM_INITDIALOG:
SetTimer(hDlg,TIMER,1*500,NULL);
SendMessage(hDlg, WM_TIMER, NULL, NULL);
GetSystemInfo(&sys_info);

   str1=str1+itoa(sys_info.dwPageSize,buff,10)+"\r\n";
   sprintf(buff,"Максимальный адрес доступной памяти:
%p",sys_info.lpMaximumApplicationAddress);
   str1=str1+buff+"\r\n";
```

```
   sprintf(buff,"Минимальный адрес доступной памяти:
%p",sys_info.lpMinimumApplicationAddress);
   str1=str1+buff+"\r\n";
   sprintf(buff, TEXT("0x%016I64X"), (__int64)
sys_info.dwActiveProcessorMask);
   str1=str1+"Число процессоров:
"+itoa(sys_info.dwNumberOfProcessors,buff,10)+"\r\n";
   str1=str1+"Гранулярность резервирования:
"+buff+"\r\n";
   SetDlgItemText(hDlg,IDC_SYSTEM_INFO,str1.c_str());

   return (INT_PTR)TRUE;

   case WM_TIMER:
   GlobalMemoryStatus(&mem_status);//позволяет отслежи-
вать текущее состояние памяти
   sprintf(buff,"%I64d",(__int64)
mem_status.dwAvailPageFile);
   str2=str2+buff+"\r\n";
   sprintf(buff,"%d", mem_status.dwTotalPhys);
   str2=str2+"Объем физической памяти в байтах:
"+buff+"\r\n";
   sprintf(buff,"%I64d",(__int64)
mem_status.dwAvailPhys);
   str2=str2+"Число байтов свободной физической памяти:
"+buff+"\r\n";
   sprintf(buff,"%I64d",(__int64)
mem_status.dwTotalVirtual);
   str2=str2+"Объем виртуальной памяти в байтах):
"+buff+"\r\n";
   sprintf(buff,"%I64d",(__int64)
mem_status.dwAvailVirtual);
   str2=str2+"Число байтов свободной виртуальной памяти:
"+buff+"\r\n";
   sprintf(buff,"%I64d",(__int64)
mem_status.dwTotalPageFile);
   str2=str2+"Макс. кол-во байт в стран. файле на hdd:
"+buff+"\r\n";
   sprintf(buff,"%d",mem_status.dwMemoryLoad);
   str2=str2+"Насколько занята подсистема управления па-
мятью: "+buff+"\r\n";

   SetDlgItemText(hDlg,IDC_MEMORY_STATUS,str2.c_str());
   return TRUE;
   }
   return (INT_PTR)FALSE;
```

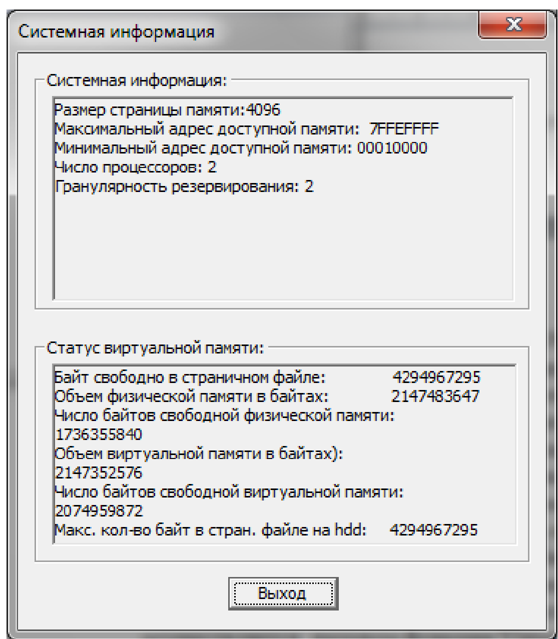


Рис. 3.3. Информация о распределении памяти

Регионы в адресном пространстве

Адресное пространство, выделяемое процессу в момент создания, практически все свободно (незарезервировано). Поэтому, чтобы воспользоваться какой-нибудь его частью, нужно выделить в нем определенные регионы через функцию ***VirtualAlloc***.

Операция выделения региона называется *резервированием*. При резервировании система обязательно выравнивает начало региона с учетом так называемой гранулярности выделения памяти. На практике необходимый объем памяти как правило не кратен размеру страницы и программист пытается выделить память не определенному количеству страниц, а некому региону памяти.

Регионом является любая часть памяти внутри адресного пространства потока. Когда зарезервированный регион адресного пространства становится не нужным, его следует вернуть в общие ресурсы системы. Эта операция – *освобождение* региона – осуществляется вызовом функции ***VirtualFree***.

Выделение памяти

Функции для работы с виртуальной памятью манипулируют страницами памяти. Эти функции используют размер страницы на компьютере для округления определенного размера и адреса.

Функция ***VirtualAlloc*** выполняет одну из операций:

— резервирует определенное количество страниц;

— передает физическую память зарезервированным страницам;

— резервирует и передает физическую память страницам.

Доступ к области, которой передана физическая память, осуществляется по обычным ссылкам.

```
case IDC_ALLOC:
   pMemory1= (char*)GetDlgItemInt(hDlg,IDC_ADRESS,
&bResult, FALSE);
   if ( pMemory == NULL )
{
  size = GetDlgItemInt(hDlg,IDC_SIZE, &bResult, FALSE);
   if ( bResult && size>0)
      pMemory = (char*)VirtualAlloc(
//возвращает виртуальный адрес  региона
      pMemory1,
//система должна зарезервировать регион там, где,
//по ее мнению, будет лучше
      size ,//размер резервируемого региона в байтах
      MEM_RESERVE |
//зарезервировать регион адресного пространства
      MEM_COMMIT | //Для передачи физической памяти
      MEM_TOP_DOWN,
//зарезервировать регион по самым старшим адресам
      PAGE_READWRITE);//указывает атрибут защиты
      if( pMemory == NULL )
      MessageBox( hDlg, "Нельзя выделить память", "Ошибка",
MB_ICONERROR);
      else
       {
      SendDlgItemMessage(hDlg, IDC_LIST, LB_ADDSTRING,
NULL, (LPARAM)"Память выделена");
VirtualQuery(//заполняет структуру MEMORY_BASIC_INFORMATION
      //информацией о диапазоне смежных страниц, имеющих
      //одинаковые состояние, атрибуты защиты и тип.
      pMemory,//виртуальный адрес  региона
      &memInfo,//адрес структуры MEMORY_BASIC_INFORMATION
      sizeof(MEMORY_BASIC_INFORMATION));//размер структуры
                        //    MEMORY_BASIC_INFORMATION
```

На рисунке 3.4 приведен интерфейс программы, демонстрирующей работу с виртуальной памятью.

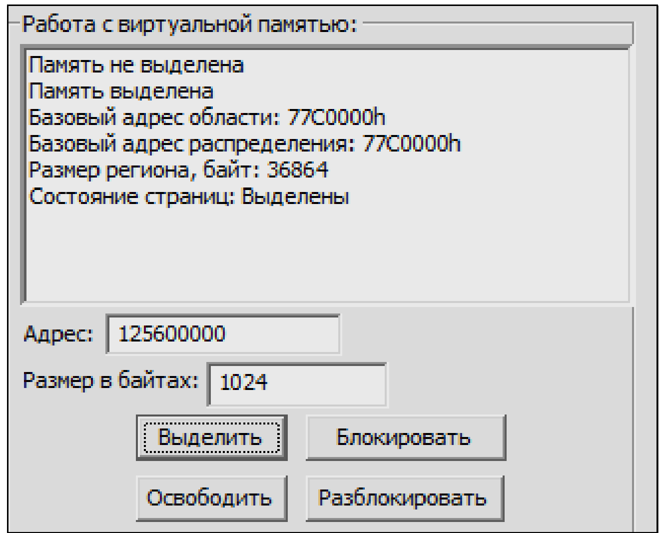


Рис. 3.4. Работа с виртуальной памятью

Освобождение памяти

Функция ***VirtualFree*** выполняет одну из операций:

— возврат физической памяти;

— освобождение региона.

```
// освобождение памяти
case IDC_FREE_MEMORY:
if(pMemory != NULL)
{
 VirtualFree (pMemory,//базовый адрес региона
0, // вернет системе весь диапазон выделенных страниц
MEM_RELEASE);//возврат системе всей физической памяти,
   //отображенной на регион, и освобождение самого региона
int count = SendDlgItemMessage(hDlg, IDC_LIST, LB_GETCOUNT,
NULL, NULL);
for (int i = 0; i< count; i++ )
SendDlgItemMessage(hDlg, IDC_LIST, LB_DELETESTRING, 0, NULL);
SendDlgItemMessage(hDlg, IDC_LIST, LB_ADDSTRING, NULL,
(LPARAM)"Память не выделена");
```

```
pMemory = NULL;
}
else   MessageBox(hDlg,"Блок памяти не выделен","Ошибка",
MB_ICONEXCLAMATION);
   return TRUE;
```

Работа со страницами

Функции ***VirtualQuery*** и ***VirtualQueryEx*** возвращают информацию об области последовательно расположенных страниц, начиная с определенного адреса в адресном пространстве процесса. ***VirtualQuery*** возвращает информацию о памяти в текущем процессе.

```
VirtualQuery(pMemory,&memInfo,sizeof(MEMORY_BASIC_INFORMATION));
```

Изменение атрибутов защиты

Атрибуты защиты страниц можно поменять функцией ***VirtualProtect***.

```
 // Изменение атрибута защиты
case IDC_ATTRIBYTE:
if (pMemory == NULL)
{
      MessageBox(hDlg, "Память не выделена", "Ошибка",
MB_ICONEXCLAMATION);
}
if( SendDlgItemMessage(hDlg, IDC_RADIO1, BM_GETCHECK, NULL,
NULL) == BST_CHECKED)
      dwProtect = PAGE_EXECUTE;
if( SendDlgItemMessage(hDlg, IDC_RADIO2, BM_GETCHECK, NULL,
NULL) == BST_CHECKED)
      dwProtect = PAGE_EXECUTE_READ;
if( SendDlgItemMessage(hDlg, IDC_RADIO3, BM_GETCHECK, NULL,
NULL) == BST_CHECKED)
      dwProtect = PAGE_EXECUTE_READWRITE;
if( SendDlgItemMessage(hDlg, IDC_RADIO4, BM_GETCHECK, NULL,
NULL) == BST_CHECKED)
      dwProtect = PAGE_EXECUTE_WRITECOPY;
if( SendDlgItemMessage(hDlg, IDC_RADIO5, BM_GETCHECK, NULL,
NULL) == BST_CHECKED)
      dwProtect = PAGE_NOACCESS;
```

```
if( SendDlgItemMessage(hDlg, IDC_RADIO6, BM_GETCHECK, NULL,
NULL) == BST_CHECKED)
       dwProtect = PAGE_READONLY;
if( SendDlgItemMessage(hDlg, IDC_RADIO7, BM_GETCHECK, NULL,
NULL) == BST_CHECKED)
       dwProtect = PAGE_READWRITE;
if( SendDlgItemMessage(hDlg, IDC_RADIO8, BM_GETCHECK, NULL,
NULL) == BST_CHECKED)
      dwProtect = PAGE_WRITECOPY;

if( !VirtualProtect( pMemory,// базовый адрес памяти
size,//число байтов, для которых  изменяете атрибут защиты
dwProtect, //новый атрибут защиты
&dwOldProtect))//адрес переменной типа DWORD,
             //в которую VirtualProtect заносит
             //старое значение атрибута защиты
MessageBox( hDlg, "Не удалось изменить атрибуты защиты",
"Ошибка", MB_OK);
break;
```

Блокировка физической памяти RAM

Функция ***VirtualLockc*** заблокирует в оперативной памяти блок.

```
// блокировка памяти
case IDC_LOCK_MEMORY:
if(pMemory != NULL)
 {
  if ( bLock == false)
       {
       VirtualQuery(pMemory,&memInfo,sizeof(MEMORY_BASIC_INF
ORMATION));
        if (memInfo.Protect != PAGE_NOACCESS)
          {
if (VirtualLock( memInfo.BaseAddress, size))
// Указатель на базовый адрес региона и размер региона
       {
        SendDlgItemMessage(hDlg, IDC_LIST, LB_ADDSTRING,
NULL, (LPARAM)"Блокировка памяти: ");
        sprintf(buf, _TEXT("%d%s%Xh"), size, " байт начиная
с адреса  : ", memInfo.BaseAddress );
        SendDlgItemMessage(hDlg, IDC_LIST, LB_ADDSTRING,
NULL, (LPARAM)buf);
bLock = true;
       }
```

```
       else MessageBox(hDlg, "Слишком большой размер блока
памяти",  "Ошибка", MB_ICONEXCLAMATION);
       }
 else MessageBox(hDlg, "Нет доступа к памяти",  "Ошибка",
MB_ICONEXCLAMATION);
     }
else MessageBox(hDlg, "Память уже заблокирована", "Ошибка",
MB_ICONEXCLAMATION);
}
else   MessageBox(hDlg, "Память не выделена", "Ошибка",
MB_ICONEXCLAMATION);
return TRUE;
```

Функция ***VirtualUnlock*** разблокирует в оперативной памяти блок.

```
// разблокировка памяти
case IDC_UNLOCK_MEMORY:
if(pMemory != NULL)
{
 if ( bLock )
 {
VirtualQuery(pMemory,&memInfo,sizeof(MEMORY_BASIC_INFORMATION));
       if (memInfo.Protect != PAGE_NOACCESS)
       {
         if (VirtualUnlock( memInfo.BaseAddress, size))
// Указатель на базовый адрес региона и размер региона
            {
       SendDlgItemMessage(hDlg, IDC_LIST, LB_ADDSTRING,
NULL, (LPARAM)"Разблокировка памяти: ");
       sprintf(buf, _TEXT("%d%s%Xh"), size, " байт начиная с
адреса  : ", memInfo.BaseAddress );
       SendDlgItemMessage(hDlg, IDC_LIST, LB_ADDSTRING,
NULL, (LPARAM)buf);
       bLock = false;
       }
   }
       else MessageBox(hDlg, "Нет доступа к памяти", "Ошибка",
MB_ICONEXCLAMATION);
}
 else MessageBox(hDlg, "Память не заблокирована", "Ошибка",
MB_ICONEXCLAMATION);
}
else   MessageBox(hDlg, "Память не выделена", "Ошибка",
MB_ICONEXCLAMATION);
return TRUE;
```

Стек потока

В ряде случаев система сама резервирует некоторые регионы в адресном пространстве процесса. Это делается, например, для размещения блоков переменных окружения процесса и его потоков, а также для размещения стека потока.

Когда процесс создает поток, система резервирует регион адресного пространства для стека потока и передает этому региону некоторый объем физической памяти. По умолчанию система резервирует 1Мбайт адресного пространства и передает ему 2 страницы физической памяти. Регион стека и вся физическая память, переданная ему, имеют атрибут защиты PAGE_READWRITE. Непосредственно перед тем, как приступить к исполнению потока, система настраивает регистр потока – указатель стека так, чтобы он указывал на конец верхней страницы региона стека. Это страница, на которой поток начинает пользоваться своим стеком. Следующая страница недоступна и считается защитной (guard page).

По мере разрастания дерева вызовов (одновременного обращения ко все большему числу функций) потоку требуется все больший объем стека. Как только он обращается к следующей (защищенной) странице, система уведомляет о происшедшей попытке. Тогда система в ответ на эту попытку передает память еще одной странице, располагая ее прямо под защитной и устанавливая для нее флаг защитной страницы. Благодаря такому механизму, размер стека потока будет расти по мере необходимости.

Кучи

Куча (***heap***) представляет собой часть памяти, зарезервированную для программы для использования в качестве временного запоминающего устройства для структур данных, чей размер не может быть определен, пока программа не запущена. Программа может затребовать память из кучи для помещения туда подобных элементов, использовать эту память и освобождать ее.

При инициализации процесса система создает кучу в его адресном пространстве. Куча, предоставляемая процессу по умолчанию, необходима многим Win32 функциям. Поскольку приложение мо-

жет вызвать одновременно несколько таких функций, доступ к куче разрешается только по очереди.

Функции работы с кучей позволяют процессу создавать свою собственную кучу. Далее процесс может использовать ряд функций для управления памятью кучи. Нет разницы между выделением памяти из собственной кучи и использованием других фукнций для выделения памяти.

Функция ***HeapCreate*** создает объект собственной кучи. Интерфейс работы с кучей приведен на рисунке 3.5.

```
case   IDC_CREATE_HEAP:
if (hHeap == NULL)
{
  hsize = GetDlgItemInt(hDlg, IDC_EDIT_HEAP, &bResult,
FALSE);
       if ( bResult && hsize>0 )
        {
        hHeap = HeapCreate( HEAP_NO_SERIALIZE, hsize, hsize);
       SendDlgItemMessage(hDlg, IDC_LIST2, LB_ADDSTRING,
NULL, (LPARAM)"Создана новая куча ");
       lpHeap = NULL;
        }
       else MessageBox(hDlg, "Неверно задан размер", "Ошиб-
ка", MB_ICONEXCLAMATION);
        }
       else MessageBox(hDlg, "Куча уже создана", "Ошибка",
MB_ICONEXCLAMATION);
return true;
```

Работа с кучей

Размер в байтах | 1024

Новая куча не создана
Создана новая куча

Создать | Уничтожить | Выделить | Освободить

Рис. 3.5. Создание кучи

При создании функции задается как начальный размер, так и максимальный размер кучи. Начальный размер определяет число зафиксированных, доступных для чтения и записи страниц, размещенных в памяти. Максимальный размер определяет наибольшее количество зарезервированных страниц. Эти страницы создают непрерывный блок в виртуальном адресном пространстве процесса, куда куча может расти. Дополнительным страницам автоматически передается физическая память из этого зарезервированного пространства, если запросы функции ***HeapAlloc*** превышают текущий размер зафиксированных страниц, полагая, что физическая память доступна для этого. Страницы, которым однажды была передана физическая память, не могут вернуть ее, пока процесс не завершится или куча не будет уничтожена вызовом функции ***HeapDestroy***.

```
case IDC_DESTROY_HEAP:
  if (hHeap != NULL)
       {
        HeapDestroy(hHeap);
        int count = SendDlgItemMessage(hDlg, IDC_LIST,
LB_GETCOUNT, NULL, NULL);
        for (int i = 0; i< count; i++ )
        SendDlgItemMessage(hDlg, IDC_LIST2, LB_DELETESTRING,
0, NULL);
        SendDlgItemMessage(hDlg, IDC_LIST2, LB_ADDSTRING,
NULL, (LPARAM)"Новая куча не создана");
       hHeap = NULL;

       }
       else MessageBox(hDlg, "Куча еще не создана", "Ошиб-
ка", MB_ICONEXCLAMATION);
return true;
```

Память собственной кучи доступна только для создавшего ее процесса. Если DLL создает свою кучу, она делает это в адресном пространстве вызывающего процесса, и куча будет доступна только ему.

Память, распределенная функцией ***HeapAlloc***, нельзя перемещать. Поскольку система не может упаковать собственную кучу, куча может стать фрагментируемой. Память, запрошенная функцией ***HeapCreate***, не обязательно будет непрерывной. Память, выделенная в пределах кучи фукнцией ***HeapAlloc*** – непрерывная.

Стандартные библиотечные функции языка Си

Win32 приложения могут благополучно использовать возможности управления памятью библиотеки программ этапа исполнения языка Си (malloc, free и т.д.) и Си++ (new, delete и т.д.). Библиотечные функции Си не имеют тех проблем, которые могли возникнуть в 16-ти разрядной Windows. Управление памятью больше не проблема, потому что система вольна управлять памятью, перемещая страницы физической памяти без задействования виртуальных адресов. В связи с этим различие между ближними (near) и дальними (far) указателями больше не важно. Поэтому можно использовать стандартные библиотечные средства работы с памятью. При этом Win32 функции управления памятью предоставляют больше функциональных возможностей.

Лабораторная работа № 4

СОЗДАНИЕ И ИСПОЛЬЗОВАНИЕ DLL

Цель работы: рассмотреть ряд аспектов создания и использования динамических библиотек DLL в операционной среде Win32.

Изучаемые вопросы

1. Функция DllMain. Последовательность вызовов в многопоточном приложении. Назначение и классификация диалоговых окон.
2. Экспорт/импорт функций.
3. Экспорт/импорт ресурсов.
4. Согласование интерфейсов.
5. Загрузка неявно подключаемой DLL.
6. Динамическая загрузка и выгрузка DLL.
7. Вызов функции по номеру.
8. Список динамических библиотек процесса.

Постановка задачи

Разработать многопоточное Win32-приложение и две DLL-библиотеки, одна из которых загружается явно, а вторая неявно. Первая динамическая библиотека содержит код обработки информации (например, код доступа к системной информации файловых систем FAT или NTFS), вторая – ресурсы типа диалог. Индивидуальное задание получить у преподавателя.

Теоретические сведения

Функция DllMain. Последовательность вызовов в многопоточном приложении.

Большинство библиотек DLL – просто коллекции практически независимых друг от друга функций, экспортируемых в приложения и используемых в них. Кроме функций, предназначенных для экспортирования, в каждой библиотеке DLL есть функция ***DllMain***. Эта функция предназначена для инициализации и очистки DLL.

Структура простейшей функции ***DllMain*** может выглядеть, например, так:

```
BOOL APIENTRY DllMain(HMODULE hModule,//дескриптор модуля DLL
DWORD  ul_reason_for_call, // причина вызова функции
LPVOID lpReserved
 )
{
	switch (ul_reason_for_call)
	{
	case DLL_PROCESS_ATTACH: // Инициализация процесса.
	case DLL_THREAD_ATTACH: // Инициализация потока.
	case DLL_THREAD_DETACH: // Очистка структур потока.
	case DLL_PROCESS_DETACH: // Очистка структур процесса.
		break;
	}
	return TRUE;
}
```

Функция ***DllMain*** вызывается в нескольких случаях. Причина ее вызова определяется параметром ul_reason_for_call, который может принимать одно из следующих значений. При первой загрузке библиотеки DLL процессом вызывается функция ***DllMain*** с dwReason, равным DLL_PROCESS_ATTACH. Каждый раз при создании процессом нового потока ***DllMain*** вызывается с ul_reason_for_call, равным DLL_THREAD_ATTACH (кроме первого потока, потому что в этом случае ul_reason_for_call равен DLL_PROCESS_ATTACH).

По окончании работы процесса с DLL функция ***DllMain*** вызывается с параметром ul_reason_for_call, равным DLL_PROCESS_DETACH. При уничтожении потока (кроме первого) ul_reason_for_call будет равен DLL_THREAD_DETACH.

В случае успешного завершения функция ***DllMain*** должна возвращать TRUE. В случае возникновения ошибки возвращается FALSE, и дальнейшие действия прекращаются.

Экспорт/импорт функций

Чтобы приложение могло обращаться к функциям динамической библиотеки, каждая из них должна занимать строку в таблице экспортируемых функций DLL.

Для того, чтобы сделать функцию или переменную экспортируемой, нужно определить их с модификатором extern “C” или квалификатором _declspec(dllexport).

Так же для определения экспортируемых из Dll функций и переменных может использоваться файл определения модуля, который имеет расширение .def. Такой def-файл содержит имя и описание библиотеки, а также список экспортируемых функций. Например,

```
LIBRARY    "InfoDLL"
EXPORTS
ProcesInfo     @1
ModuleInfo     @2
```

В строке экспорта функции можно указать ее порядковый номер, поставив перед ним символ @. Этот номер будет затем использоваться при обращении к ***GetProcAddress ()***.

Когда используется *статическая* компоновка, то необходимо включать в файл проекта приложения соответствующий lib-файл, содержащий нужную вам библиотеку объектных модулей. Такая библиотека содержит исполняемый код модулей, который на этапе статической компоновки включается в exe-файл загрузочного модуля. Если же используется *динамическая* компоновка, в загрузочный exe-файл приложения записывается не исполнимый код функций, а ссылка на соответствующую DLL-библиотеку и функцию внутри нее. Как мы уже говорили, эта ссылка может быть организована с использованием либо имени функции, либо ее порядкового номера в DLL-библиотеке.

Для того, чтобы редактор связей мог создать ссылку, в файл проекта приложения вы должны включить так называемую библиотеку импорта *(import library*). Эта библиотека создается автоматически системой разработки Microsoft Visual C++.

Экспорт/импорт ресурсов

Динамическая загрузка применима и к ресурсам DLL. Для этого сначала необходимо вызвать функцию ***LoadLibrary***:

```
   hDllResource=LoadLibraryEx("DllResource.dll",0,
LOAD_LIBRARY_AS_DATAFILE);
```

Устанавливаем флаг LOAD_LIBRARY_AS_DATAFILE,так как DLL содержит только ресурсы и никаких функций и а также его нужно указывать для загрузки EXE.

Для экспортирования ресурсов требуется переименовать resource.h, а затем добавить данный файл к коду нашего приложения, например, как приведено ниже.

```
#include "DllResource.h"
```

Согласование интерфейсов

При использовании собственных библиотек или библиотек независимых разработчиков придется обратить внимание на согласование вызова функции с ее прототипом.

По умолчанию в Visual C++ интерфейсы функций согласуются по правилам C++. Это значит, что параметры заносятся в стек справа налево, вызывающая программа отвечает за их удаление из стека при выходе из функции и расширении ее имени. Расширение имен (name mangling) позволяет редактору связей различать перегруженные функции, т.е. функции с одинаковыми именами, но разными списками аргументов. Однако в старой библиотеке С функции с расширенными именами отсутствуют.

Хотя все остальные правила вызова функции в С идентичны правилам вызова функции в C++, в библиотеках С имена функций не расширяются. К ним только добавляется впереди символ подчеркивания (_).

Если необходимо подключить библиотеку на С к приложению на C++, все функции из этой библиотеки придется объявить как внешние в формате С:

```
extern "C" int MyOldCFunction(int myParam);
```

Объявления функций библиотеки обычно помещаются в файле заголовка этой библиотеки, хотя заголовки большинства библиотек С не рассчитаны на применение в проектах на C++. В этом случае необходимо создать копию файла заголовка и включить в нее модификатор extern "C" к объявлению всех используемых функций библиотеки. Модификатор extern "C" можно применить и к целому блоку, к которому с помощью директивы #include подключен файл

старого заголовка С. Таким образом, вместо модификации каждой функции в отдельности можно обойтись всего тремя строками:

```
extern "C"
{
      #include "MyCLib.h"
}
```

В программах для старых версий Windows использовались также соглашения о вызове функций языка PASCAL для функций Windows API. В новых программах следует использовать модификатор WINAPI, преобразуемый в _stdcall. Хотя это и не стандартный интерфейс функций С или С++, но именно он используется для обращений к функциям Windows API. Однако обычно все это уже учтено в стандартных заголовках Windows.

Загрузка явно подключаемой DLL

При запуске приложение пытается найти все файлы DLL, неявно подключенные к приложению, и поместить их в область оперативной памяти, занимаемую данным процессом. Поиск файлов DLL операционной системой осуществляется в следующей последовательности:

1) Каталог, в котором находится EXE-файл;

2) Текущий каталог процесса;

3) Системный каталог Windows.

Если библиотека DLL не обнаружена, система выводит диалоговое окно с сообщением о ее отсутствии и путях, по которым осуществлялся поиск. Объект-процесс не создается.

Если нужная библиотека найдена, она помещается в оперативную память процесса, где и остается до его окончания. Теперь приложение может обращаться к функциям, содержащимся в DLL.

Для неявного подключения библиотеки нужно в папку приложения поместить lib- файл и заголовочный файл. А так же, подключить к линковщику скопированный lib-файл.

```
#include "UsageDll.h"
#pragma comment(lib, "UsageDll.lib")
```

Динамическая загрузка и выгрузка DLL

Для экономии ресурсов и увеличения быстродействия программы можно использовать динамическую загрузку и выгрузку DLL во время выполнения программы.

Первое, что необходимо сделать при динамической загрузке DLL, – это поместить модуль библиотеки в память процесса. Данная операция выполняется с помощью функции ***LoadLibrary***, имеющей единственный аргумент – имя загружаемого модуля. Соответствующий фрагмент программы должен выглядеть так:

```
   hInfoDLL=LoadLibrary("InfoDLL.dll");
   if(!hInfoDLL)
       {
             MessageBox(NULL,"Невозможно загрузить
InfoDll.dll!","Ошибка!",MB_OK);
             TerminateThread(hTh,0);
             TerminateThread(hTh1,0);
       }
```

После завершения работы с библиотекой динамической компоновки, ее можно выгрузить из памяти процесса с помощью функции ***FreeLibrary***:

```
   if (!FreeLibrary(hInfoDLL))
       {      MessageBox(NULL,"Невозможно выгрузить
InfoDll.dll","Ошибка!",MB_OK);              }
```

Вызов функции по номеру

Для того чтобы вызвать функцию из библиотеки, зная ее идентификатор, необходимо получить значение дальнего указателя на эту функцию, вызвав функцию GetProcAddress(HINSTANCE hLibrary, LPCSTR lpszProcName). Через параметр hLibrary необходимо передать функции идентификатор DLL-библиотеки, полученный ранее от функции LoadLibrary.

Параметр lpszProcName является дальним указателем на строку, содержащую имя функции или ее порядковый номер, преобразованный макрокомандой MAKEINTRESOURCE.

Для упрощения вызова функции может осуществляться ее вызов не по имени, а по номеру. Для этого в строке экспорта

функции можно указать ее порядковый номер, поставив перед ним символ @.

```
LIBRARY   "InfoDLL"
EXPORTS
   ProcesInfo   @1
   ModuleInfo   @2
```

Этот номер будет затем использоваться при обращении к ***GetProcAddress***, как показано ниже:

```
   ProcesInfo = (MYPROC) GetProcAddress(hInfoDLL,
MAKEINTRESOURCE(1));

   ModuleInfo = (MYPROC)
GetProcAddress(hInfoDLL,MAKEINTRESOURCE(2) );
```

На самом деле компилятор присваивает порядковые номера всем экспортируемым объектам. Однако способ, которым он это делает, отчасти непредсказуем, если не присвоить эти номера явно.

В строке экспорта можно использовать параметр NONAME. Он запрещает компилятору включать имя функции в таблицу экспортирования DLL:

```
MyFunction   @1 NONAME
```

Приложения, использующие библиотеку импортирования для неявного подключения DLL, не “заметят” разницы, поскольку при неявном подключении порядковые номера используются автоматически.

Список динамических библиотек процесса

В комплекте системы разработки Microsoft Visual C++ входит программа dumpbin.exe, предназначенная для запуска из командной строки. С помощью этой утилиты вы сможете проанализировать содержимое любого загрузочного файла в формате COFF, в том числе DLL-библиотеки, определив имена экспортируемых функций, их порядковые номера, имена DLL-библиотек и номера функций, импортируемых из этих библиотек и т. д.

Лабораторная работа № 5

ГРАФИКА WINDOWS. ОСНОВЫ УПРАВЛЕНИЯ ВЫВОДОМ ГРАФИЧЕСКОЙ И ТЕКСТОВОЙ ИНФОРМАЦИИ НА БАЗЕ БИБЛИОТЕКИ GDI

Цель работы: Изучить основы управления выводом текстовой и графической информации на базе библиотеки GDI.

Изучаемые вопросы

1. Графическое устройство и его контекст.
2. Атрибуты системы координат, их влияние на вывод информации.
3. Шрифты, классификация, параметры шрифта, установка в контекст устройства.
4. Атрибуты контекста устройства, влияющие на вывод текста.
5. Методы GDI для вывода текста и векторной графики.

Постановка задачи

1. Нарисовать геометрическую фигуру в заданной области.
2. Осуществить вывод текста в заданной области (по контуру фигуры) согласно индивидуальному заданию из приложения (выдаётся преподавателем).

Теоретические сведения

Графическое устройство и его контекст

Взаимодействие приложения с GDI осуществляется при непременном участии посредника – так называемого контекста устройства. ***Контекст устройства*** (device context) – это внутренняя структура данных, которая определяет набор графических объектов и ассоциированных с ними атрибутов, а также графических режимов, влияющих на вывод. Контекст устройства содержит много атрибутов, определяющих поведение функций GDI.

Если необходимо рисовать на устройстве графического вывода (экране дисплея или принтере), то сначала нужно получить дескриптор контекста устройства. Возвращая этот дескриптор после

вызова соответствующих функций, Windows тем самым предоставляет разработчику право на использование данного устройства. После этого дескриптор контекста устройства передается как параметр в функции GDI, чтобы идентифицировать устройство, на котором должно выполняться рисование.

Для создания и освобождения контекста применяются пары функций: ***BeginPaint*** и ***EndPaint*** или ***GetDC*** и ***ReleaseDC***:

Для первых двух функций требуется дескриптор окна (передаваемый в оконную процедуру как параметр) и адрес переменной типа структуры PAINTSTRUCT, определяемой в оконной процедуре. Вызов ***BeginPaint*** заполняет поля этой структуры, а также возвращает дескриптор контекста рабочей области окна, который должен запоминаться в переменной типа HDC. Вызов функции ***EndPaint*** освобождает дескриптор контекста рабочей области окна. Метод используется при обработке сообщения WM_PAINT.

Если контекст был получен с помощью функции ***GetDC***, то после завершения процедуры рисования перед выходом из обработчика сообщений следует освободить полученный контекст, вызвав функцию ***ReleaseDC***. Эту пару функций можно использовать при обработке других, не WM_PAINT сообщений.

Атрибуты системы координат, их влияние на вывод информации

Система координат для окна базируется на координатной системе дисплея. Основной единицей измерения служит пиксел. Точки на экране задаются парой координат (x, y). При этом x-координаты возрастают слева направо, а y-координаты – сверху вниз. Направление осей системы координат зависит от режима отображения.

Режим отображения контекста устройства выбирается следующей функцией:

```
SetMapMode(hdc,MM_HIMETRIC);
```

Вместо MM_HIMETRIC можно задать следующие режимы отображения, приведенные в таблице 5.1

Таблица 5.1

Режимы отображения

Режим отображения	Направление по оси x	Направление по оси y	Логические единицы
MM_TEXT	Вправо	Вниз	Пиксель
MM_LOMETRIC	Вправо	Вверх	0,1 мм
MMJHIMETRIC	Вправо	Вверх	0,01 мм
MMJ.OENGLISH	Вправо	Вверх	0,01 дюйма
MM_HIENGLISH	Вправо	Вверх	0,001 дюйма
MM_TWIPS	Вправо	Вверх	1 / 1440 дюйма
MM_ISOTROPIC	Любое	Любое	Произвольные единицы (x = = y)
MM_ANISOTROPIC	Любое	Любое	Произвольные единицы (x! = y)

Если функция SetMapMode не вызывалась, то по умолчанию используется режим отображения MM_TEXT.

Шрифты, классификация, параметры шрифта, установка в контекст устройства

Шрифты GDI подразделяются на три типа:

- растровые шрифты;
- векторные шрифты;
- шрифты типа TrueType.

Основными параметрами шрифтов являются высота, ширина, наклон шрифта, толщина шрифта и его тип.

Создать шрифт можно функцией ***CreateFontInderect***:

```
LOGFONT Font;

Font.lfHeight = 500;//высота буквы
    Font.lfWidth = 0;//ширина буквы
// (0-по умолчанию подходящий по масштабу)
Font.lfEscapement =3600-atan(y/x)*1800/3.14;
 //угол наклона
    Font.lfOrientation=3600-atan(y/x)*1800/3.14;
                        //atan(y/x)*10;
    Font.lfWeight = 500; //толщина шрифта 500 =FW_MEDIUM
    Font.lfItalic = TRUE; //курсив
    Font.lfUnderline = FALSE;
    Font.lfStrikeOut = FALSE;
//определяет кодировку шрифта
    Font.lfCharSet = DEFAULT_CHARSET;
//тип и семейство шрифта
    Font.lfPitchAndFamily = DEFAULT_PITCH;
    Font.lfClipPrecision = CLIP_LH_ANGLES;

    hFont = CreateFontIndirect(&Font);
```

Для установки шрифта в контекст устройства используется функция:

```
    HFONT hOldFont= (HFONT)SelectObject(hdc,hFont);
```

где hdc – это контекст устройства, hFont – описатель созданного шрифта.

Атрибуты контекста устройства, влияющие на вывод текста

На вывод текста могут повлиять такие параметры контекста как режим отображения, цвет текста, цвет фона, на котором рисуется текст.

Цвет текста устанавливается с помощью функции:

```
SetTextColor(hdc,color);
```

где hdc – контекст устройства , color – цвет выводимых символов.

Для установки фона текста используется функция

```
SetBkColor(hdc,RGB(255,0,0));
```

Методы GDI для вывода текста и векторной графики

GDI поддерживает множество объектов рисования: перья, кисти, шрифты, палитры, растровые изображения.

Перья используются для рисования линий, кривых и контуров других фигур.

Создать перо можно функцией:

```
    hPen = CreatePen(PS_SOLID, 3, RGB(0,200,0));
```

где первый параметр PS_SOLID указывает на стиль линий-сплошной,второй параметр указывает на толщину линии, третий параметр указывает на цвет линии.

Для создания кисти используется функция:

```
hBrush = CreateSolidBrush(RGB(255,0,0));
```

Для того, чтобы добавить созданный нами объект в контекст необходимо применить функцию:

```
hOldPen = (HPEN) SelectObject(hdc1, hPen);
```

После создания и выбора объекта для рисования могут использоваться следующие функции:

```
   // эллипс
   BOOL Ellipse(HDC hdc, int nLeftRect, int nTopRect, int
nRightRect, int nBottomRec);

   // отрезок
   BOOL LineTo(HDC hdc,  int nXEnd, int nYEnd,);

   // серия отрезков
   BOOL Polyline(HDC hdc, CONST POINT *lppt, int cPoints);
```

Для того, чтобы использовать корректно функцию ***LineTo,*** мы должны установить текущую точку в нужные нам координаты, то есть то место, откуда мы хотим нарисовать линию. Для этого есть

соответствующая функция ***MoveToEx***, которая помещает текущую точку в нужные нам координаты:

```
MoveToEx(hdc,ThirdCoord.x,ThirdCoord.y,NULL);
LineTo(hdc,FirstCoord.x,FirstCoord.y);
Ellipse(hdc,ThirdEllipse.x,ThirdEllipse.y,ThirdEllipse.x1
,ThirdEllipse.y1);
```

Для вывода текста можно использовать следующие функции:

```
BOOL TextOut (
HDC hdc, // дескриптор контекста устройства
int nXStart. // x-координата стартовой позиции
int nYStart. // y-координата стартовой позиции
LPCTSTR lpString, // указатель на символьную строку
int cbString // число символов в строке
);
```

Функция обеспечивает вывод строки с адресом lpString, размещая текст в заданной позиции с учетом текущего режима выравнивания. При выводе используются текущие значения атрибутов контекста устройства – шрифт, цвет текста и цвет фона графических элементов.

```
TextOut(hdc1,ThirdCoord.x+150,ThirdCoord.y+600,STR,
wcslen(STR));
```

Лабораторная работа № 6

РАСТРОВАЯ ГРАФИКА

Цель работы: изучить структуру и основные операции по обработке растровых изображений в Windows.

Изучаемые вопросы

1. Структура файлов *.bmp.
   a) Заголовки.
   b) Палитра цветов.
   c) Битовый массив образа.
2. Структура BITMAP.
   a) Создание битовой карты.
   b) Заполнение битовой карты.
   c) Вывод битовой карты.
3. Растровые операции.

Постановка задачи

1. Создать в графическом редакторе растровые изображения размером 12×12 (16 бит), 16×16 (4 бита), 32×32 (8 бит). Рисунки и дамп включить в отчет.
2. Расшифровку дампов файлов рисунков привести в виде таблицы.
3. Написать программу, где, в соответствии с вариантом из приложения, фигуры из каждого сектора экрана через Δt после запуска программы начинают двигаться вверх. Схема окна приложения приведена на рисунке 6.1. Движением каждой фигуры управляет отдельный поток.

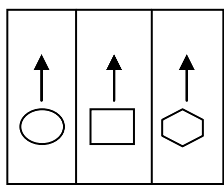


Рис. 6.1. Схема окна приложения

Теоретические сведения

*Структура файлов *.bmp. Заголовки*

BMP-файл состоит из четырёх частей:

1) Заголовок файла (BITMAPFILEHEADER);

2) Заголовок изображения (BITMAPINFOHEADER, может отсутствовать). BITMAPV4HEADER (Win95, NT4.0) BITMAPV5HEADER (Win98/Me, 2000/XP);

3) Палитра (может отсутствовать);

4) Само изображение.

Файл DIB начинается с секции заголовка, определенной структурой BITMAPFILEHEADER. Эта структура имеет пять полей:

Поле	***Размер***	***Описание***
bfType	WORD	Байты"BM" для битовых образов
bfSize	DWORD	Общий размер файла
bfReserved1	WORD	Установлено в 0
bfReserved2	WORD	Установлено в 0
bfOffBits	DWORD	Смещение битов битового образа от начала файла

За этой информацией следует другой заголовок, определенный структурой BITMAPINFOHEADER. Структура имеет 11 полей:

Поле	***Размер***	***Описание***
biSize	*DWORD*	*Размер структуры в байтах*
biWidth	*LONG*	*Ширина битового образа в пикселях*
biHeight	*LONG*	*Высота битового образа в пикселях*
biPlanes	*WORD*	*Установлено в 1*
biBitCount	*WORD*	*Число битов цвета на пиксель (1, 4, 8, 24)*
biCompression	*DWORD*	*Схема компрессии (если нет — 0)*
biSizeImage	*DWORD*	*Размер битов битового образа в байтах (нужен только при компрессии)*
biXPelsPerMeter	*LONG*	*Разрешение в пикселях на метр по горизонтали*
biYPelsPerMeter	*LONG*	*Разрешение в пикселях на метр по вертикали*
biClrUsed	*DWORD*	*Число цветов, используемых в изображении*
biClrImportant	*DWORD*	*Число важных цветов в изображении*

Все поля, следующие за полем biBitCount, могут быть по умолчанию установлены в 0 (или их может вообще не быть в файле). В этом случае длина структуры будет равна 16 байтам. Кроме описанных выше полей, она может также содержать дополнительные поля.

Палитра цветов

Если biClrUsed установлено в 0 и число битов цвета на пиксель равно 1, 4 или 8, то за структурой BITMAPINFOHEADER следует таблица цветов, состоящая из двух или более структур RGBQUAD. Структура RGBQUAD определяет значение RGB цвета:

Поле	***Размер***	***Описание***
rgbBlue	*BYTE*	*Интенсивность голубого*
rgbGreen	*BYTE*	*Интенсивность зеленого*
rgbRed	*BYTE*	*Интенсивность красного*
rgbReserved	*BYTE*	*Равно 0*

Число структур RGBQUAD обычно определяется значением поля biBitCount : 2 структуры RGBQUAD при 1 цветовом бите, 16 при 4 цветовых битах, 256 при 8 битах цвета. Однако, если значение в поле biClrUsed не равно нулю, то в нем содержится число структур RGBQUAD, входящих в таблицу цветов.

Битовый массив образа

За таблицей цветов следует массив битов, определяющих битовый образ. Этот массив начинается с нижней строки пикселей. Каждая строка начинается с самого левого пикселя. Каждый пиксель представлен 1, 4, 8 или 256 битами.

Для монохромных битовых образов с 1 битом цвета на пиксель первый пиксель в каждой строке представляется наиболее значащим битом первого байта в каждой строке. Если этот бит равен 0, то цвет пикселя определяется из первой структуры RGBQUAD таблицы цветов. Если он равен 1, то цвет пикселя определяется из второй структуры RGBQUAD таблицы цветов.

В случае 16-цветного битового образа с 4 битами на пиксель первый пиксель каждой строки представляется четырьмя самыми

значащими битами первого байта в каждой строке. Цвет каждого пикселя определяется путем использования этого 4-х битного значения как индекса для любого из 16 входов таблицы цветов.

В случае 256-цветного битового образа каждый байт соответствует одному пикселю. Цвет каждого пикселя определяется путем использования этого 8-ми битного значения как индекса для любого из 256 входов таблицы цветов. Если битовый образ содержит 24 бита для представления цвета одного пикселя, то каждый набор из 3-х байтов – это RGB-цвет пикселя. Таблица цветов тсутствует, если значение поля biClrUsed структуры BITMAPINFOHEADER не равно 0.

В любом случае, каждая строка данных битового образа имеет размер, кратный 4 байтам. Для удовлетворения этого требования, если необходимо, строка расширяется вправо.

Структура BITMAP. Создание и заполнение битовой карты

Для создания битовой карты можно использовать функции CreateBitmap(), CreateBitmapInderect(), CreateCompatibleBitmap().

Для создания битового образа виртуального контекста устройства использовалась функция:

```
hdcMem = CreateCompatibleDC(hdc);
```

Эта функция позволяет создать виртуальный контекст устройства, которое связано с заданным контекстом устройства. Полученный контекст используется дальше для вывода изображения на экран.

Следующая функция создает пустое растровое изображение в памяти размером 50*50,совместимое с контекстом устройства и получаем его хэндл:

```
hbmMem = CreateCompatibleBitmap(hdc, 1000, 1000);
```

Выбираем изображение в совместимый контекст в памяти:

```
SelectObject(hdcMem, hBitmFon);
```

Для заполнения битовой карты можно использовать функцию:

```
SetBitmapBits(HBITMAP hbmp, DWORD cBytes,CONST VOID
*lpBits);
```

Эта функция устанавливает биты цветов для побитового отображения в hbmp из области памяти lpBits, cBytes – количество копируемых байтов.

Растровые операции

Растровые операции предназначены для определения, как данные цвета для прямоугольной области источника должны быть объединены с данными цветами прямоугольной области назначения для получения окончательно цвета. В таблице 6.1 приведены наиболее широко применяемые операции и их коды.

Таблица 6.1

Коды растровых операций

Коды	Описание
DSTINVERT	Инвертирует целевое растровое изображение
MERGECOPY	Объединяет растр рисунка с исходным посредством логического оператора AND
MERGEPAINT	Инвертирует исходное растровое изображение и объединяет его с исходным логич. оператором OR
NOTSRCCOPY	Инвертирует изображение и копирует его
NOTSRCERASE	Объединяет целевое и исходное растровые изображения, которые затем инвертирует
PATCOPY	Копирует рисунок в целевое растровое изображение
PATINVERT	Объединяет целевое растровое изображение с рисунком, используя логический оператор XOR
PATPAINT	Инвертирует исходное изображение объединяет его с рисунком операцией OR, затем объединяет результат с целевым изображением операцией OR
SRCAND	Объединяет целевое и исходное растровое изображения, используя логический оператор AND
SRCCOPY	Копирует исходное растровое изображение
SRCERASE	Объединяет инвертированное целевое изображение с исходным, используя логический оператор AND
SRCINVERT	Объединяет целевое и исходное растровое изображения, используя логический оператор XOR
SRCPAINT	Объединяет целевое и исходное растровое изображения, используя логический оператор OR

Лабораторная работа № 7

ПЕРЕДАЧА ИНФОРМАЦИИ МЕЖДУ ПРОЦЕССАМИ

Цель работы: изучить основы взаимодействия локальных процессов на базе файлов, проецируемых в память.

Изучаемые вопросы

1. Функции для работы с файлами, отображаемыми в память.
2. Файлы данных, проецируемые в память.
3. Совместный доступ процессов к данным через механизм проецирования.
4. Обработка текстового и графического файлов двумя процессами
5. Работа с файлами больших размеров.
6. Взаимодействие процессов через Page файл.

Постановка задачи

1. Осуществить обмен текстовыми и графическими данными между двумя приложениямиP1 и P2 с выводом информации в заданную область окна программы.
2. Реализовать обработку процессом файлов больших размеров.
3. Осуществить обмен данными между двумя приложениями P1 и P2 через Page файл с выводом информации в заданную область окна программы.

Теоретические сведения

Функции для работы с файлами, отображаемыми в память

Функция ***CreateFile*** позволяет процессу открывать файл, проецируемый в память другим процессом:

```
hTxtFile =
CreateFileW(L"kate.txt",GENERIC_READ,FILE_SHARE_READ,
NULL,OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
```

Функция ***CreateFileMapping*** создаёт уже другие, «проецирующие» внутренние структуры и связывает «сущность на диске» с «сущностью в адресном пространстве»:

```
//Создание объекта ядра «проекция файла»
hTxtMappingFile=CreateFileMappingA(hTxtFile,NULL,PAGE_READO
NLY,0,0,"kate");
```

По своему месту в последовательности действий она создаёт что-то наподобие «адресного пространства из файла». Этот объект поддерживает соответствие между содержимым файла и адресным пространством процесса, использующего этот файл (“kate” – имя объекта файлового отображения). Так как мы хотим отобразить весь файл, то четвертый и пятый параметры должны быть равны нулю.

Функция ***MapViewOfFile*** делает "проекцию в память":

```
//Проецирование данных файла на адресное пространство процесса
hTxtMapFileStartAddr =
MapViewOfFile(hTxtMappingFile,FILE_MAP_READ,0,0,0);
```

Здесь hTxtMappingFile – дескриптор созданного объекта файлового отображения. Второй параметр определяет режим доступа к файлу. Значение, возвращаемое функцией MapViewOfFile, имеет тип "указатель". Если функция отработала успешно, то она вернет начальный адрес данных объекта файлового отображения.

Функция ***OpenFileMapping*** используется для открытия объекта ядра «проекция файла»:

```
//Открытие объекта ядра «проекция файла»
hTxtMappingFile =
OpenFileMappingA(FILE_MAP_READ,FALSE,"kate");
```

Функция ***UnmapViewOfFile*** прекращает отображение в адресное пространство процесса того файла, который перед этим был отображен при помощи функции ***MapViewOfFile***:

```
UnmapViewOfFile(hTxtMapFileStartAddr);
```

Файлы данных, проецируемые в память

Операционная система позволяет проецировать на адресное пространство процесса и файл данных. Это очень удобно при манипуляциях с большими потоками данных.

Для проецирования файла данных нужно выполнить три операции:

1. Создать или открыть объект ядра "файл", идентифицирующий дисковый файл, который Вы хотите использовать как проецируемый в память. Для создания объекта “файл” используется функция ***CreateFile*** .

2. С помощью функции ***CreateFileMapping*** создается объект ядра “проецируемый файл”, чтобы сообщить системе размер файла и способ доступа к нему. При этом используется описатель файла (handle), возвращенный функцией ***CreateFile***. Теперь файл готов к проецированию.

3. Производится отображение объекта “проецируемый файл” или его части на адресное пространство процесса. Для этого применяется функция ***MapViewOfFile***.

Для открепления файла от адресного пространства процесса используется функция ***UnmapViewOfFile***, а для уничтожения объектов “файл” и “проецируемый файл” – функция ***CloseHandle***.

Общая схема работы с проецированными файлами такова:

```
//Открытие объекта ядра «файл»
hTxtFile = CreateFileW(L"kate.txt",…);

//Создание объекта ядра «проекция файла»
hTxtMappingFile=CreateFileMappingA(hTxtFile,…,"kate");

//Проецирование данных файла на адресное пространство
процесса
hTxtMapFileStartAddr = MapViewOfFile(hTxtMappingFile,…);
```

Совместный доступ процессов к данным через механизм проецирования

Совместное использование данных в этом случае происходит так: два или более процесса проецируют в память представления одного и того же объекта "проекция файла", т.е. одни и те же стра-

ницы физической памяти. В результате, когда один процесс записывает данные в представление общего объекта "проекция файла", изменения немедленно отражаются на представлениях в других процессах. Но при этом все процессы должны использовать одинаковое имя объекта "проекция файла".

1.exe

```
case WM_INITDIALOG:
hFileMapping = CreateFileMapping(INVALID_HANDLE_VALUE,
NULL, PAGE_READWRITE, 0, 1000, TEXT("katarina"));
   if(hFileMapping == NULL)
     MessageBox(hDlg,TEXT("Невозможно создать проекцию
файла в Page-файле..."),TEXT("Error"),MB_ICONERROR);
     else
   {
    sharedBuffer  = (LPTSTR) MapViewOfFile(hFileMapping,
FILE_MAP_ALL_ACCESS,0,0,0);
     }
```

2.exe

```
hFileMapping =
OpenFileMapping(FILE_MAP_READ,FALSE,TEXT("katarina"));

  if(hFileMapping == NULL)
  MessageBox(hDlg,TEXT("Невозможно создать проекцию
файла..."),TEXT("Error"),MB_ICONERROR);
   else
  {
   sharedBuffer = (LPTSTR) MapViewOfFile(hFileMapping,
FILE_MAP_READ,0,0,0);
   DWORD dwError = GetLastError();
   }
```

Обработка текстового и графического файлов двумя процессами

При обработке текстовых и графических файлов небольшого размера можно спроецировать весь файл в память. Затем произвести его обработку. По завершении обработки файла необходимо прекратить отображение на адресное пространство представления объекта «проекция файла» и закрыть описатель всех объектов ядра.

Текстовый файл:

```
//Открытие объекта ядра «проекция файла»
hTxtMappingFile =
OpenFileMappingA(FILE_MAP_READ,FALSE,"kate");
if(hTxtMappingFile != NULL)
{
//Проецирование данных файла на адресное пространство процесса
hTxtMapFileStartAddr = MapViewOfFile(hTxtMappingFile,
FILE_MAP_READ,0,0,0);
//Вывод данных из текстового файла в эдит
SetDlgItemTextA(hDlg,
IDC_EDIT_TEXT,(char*)hTxtMapFileStartAddr);
//Отключение файла данных от адресного пространства
процесса
UnmapViewOfFile(hTxtMapFileStartAddr);
//Закрытие объекта «проекция файла»
CloseHandle(hTxtMappingFile);
```

Графический файл:

```
//Открытие объекта ядра «файл»
hBmpFile =
CreateFileA("LOVE.bmp",GENERIC_READ,FILE_SHARE_READ,NULL,
OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,NULL);
	GetClientRect(GetDlgItem(hDlg,IDC_BMPFRAME),&rect);
//Создание объекта ядра «проекция файла»
	hBmpMappingFile = CreateFileMappingA(
	hBmpFile,		//описатель файла
	NULL,			//аттрибуты защиты объекта ядра
	PAGE_READONLY,		//атрибут защиты, присваиваемый
				// страницам физической памяти
	0,0,			//максимальный размер файла в
				// байтах (0 -- размер меньше
			4Гб)
	"BmpFile");		//имя объекта ядра

//Проецирование данных файла на адресное пространство процесса
	hBmpMapFileAddr = MapViewOfFile(
	hBmpMappingFile,	// описатель объекта «проекция файла»
	FILE_MAP_READ,			// вид доступа к данным
	0,0,		// смещение в файле до байта файла
данных,
		//который нужно считать в представлении первым
```

```
	0);	// сколько байтов файла данных должно быть
		//спроецировано на адресное пространство
		//(0 -- от смещения и до конца файла)
BITMAPFILEHEADER *bFileHeader=
	(BITMAPFILEHEADER*)hBmpMapFileAddr;
BITMAPINFO *bInfo=(BITMAPINFO*)((char*)hBmpMapFileAddr+14);
	hdc=GetDC(GetDlgItem(hDlg,IDC_BMPFRAME));
	hBmpFile=CreateDIBitmap(hdc,&(bInfo->bmiHeader),
CBM_INIT,(char*)hBmpMapFileAddr+bFileHeader->bfOffBits,
bInfo,DIB_PAL_COLORS);
	hMemDC=CreateCompatibleDC(hdc);
	SelectObject(hMemDC,hBmpFile);
	StretchBlt(hdc,0,0,rect.right,rect.bottom,hMemDC,
0,0,bInfo->bmiHeader.biWidth,bInfo->bmiHeader.biHeight,
SRCCOPY);
	ReleaseDC(GetDlgItem(hDlg,IDC_BMPFRAME),hdc);
	DeleteDC(hMemDC);
	DeleteObject(hBmpFile);
return (INT_PTR)TRUE;
```

Работа с файлами больших размеров

```
//*****Работа с большим файлом******************
if (LOWORD(wParam) == IDC_HUGE)
{
// начальные границы представлений всегда начинаются по
// адресам, кратным гранулярности выделения памяти
	SYSTEM_INFO sinf;
	GetSystemInfo(&sinf);

// открываем файл данных
HANDLE hBigFile = CreateFileA("TECT.txt", GENERIC_READ,
FILE_SHARE_READ, NULL,OPEN_EXISTING,
FILE_FLAG_SEQUENTIAL_SCAN,NULL);

// создаем объект проекции файла
HANDLE hHugeFileMapping = CreateFileMapping(hBigFile,
NULL,PAGE_READONLY, 0, 0, NULL);

	DWORD dwFileSizeHigh;
	__int64 dwFileSize = GetFileSize(hBigFile,
&dwFileSizeHigh);
	dwFileSize += (((__int64) dwFileSizeHigh) << 32);
// доступ к описателю объекта файл нам больше не нужен
```

```
       CloseHandle(hBigFile);
       __int64 dwFileOffset = 0;
       DWORD dwNumOfOs = 0;
       char symbol[100] = "";
DWORD dwPrBarRange =
dwFileSize/sinf.dwAllocationGranularity;

while (dwFileSize > 0)
{
// определяем, сколько байтов надо спроецировать
DWORD dwBytesInBlock = sinf.dwAllocationGranularity;
       if (dwFileSize < sinf.dwAllocationGranularity)
       dwBytesInBlock = (DWORD)dwFileSize;

//Проецирование данных файла на адресное
//пространство процесса
PBYTE hHugeMapFileStartAddr = (PBYTE)
MapViewOfFile(hHugeFileMapping,FILE_MAP_READ,
       (DWORD) (dwFileOffset >> 32), // начальный байт
       (DWORD) (dwFileOffset & 0xFFFFFFFF), // в файле
       dwBytesInBlock); // число проецируемых байтов

       SetDlgItemTextA(hDlg, IDC_EDIT_HUGEFILE, (char*)
hHugeMapFileStartAddr);
       SendMessage(GetDlgItem(hDlg, IDC_PROGRESS),
(UINT)PBM_STEPIT, 0, 0);

// прекращаем проецирование представления,
// чтобы в адресном пространстве
// не образовалось несколько представлений одного файла

       UnmapViewOfFile(hHugeMapFileStartAddr);

// переходим к следующей группе байтов в файле
       dwFileOffset += dwBytesInBlock;
       dwFileSize -= dwBytesInBlock;
}
CloseHandle(hHugeFileMapping);
return (INT_PTR)TRUE;
       }
```

Пример взаимодействия процессов через Page файл.

1.exe

```
case WM_INITDIALOG:
hFileMapping = CreateFileMapping(INVALID_HANDLE_VALUE,
NULL, PAGE_READWRITE, 0, 1000, TEXT("katarina"));
if(hFileMapping == NULL)
MessageBox(hDlg,TEXT("Невозможно создать проекцию файла в
Page-файле..."),TEXT("Error"),MB_ICONERROR);
  else
{
sharedBuffer  = (LPTSTR) MapViewOfFile(hFileMapping,
FILE_MAP_ALL_ACCESS,0,0,0);
}
```

2.exe

```
hFileMapping =
OpenFileMapping(FILE_MAP_READ,FALSE,TEXT("katarina"));

if(hFileMapping == NULL)
    MessageBox(hDlg,TEXT("Невозможно создать проекцию
файла..."),TEXT("Error"),MB_ICONERROR);
  else
  {
sharedBuffer = (LPTSTR) MapViewOfFile(hFileMapping,
FILE_MAP_READ,0,0,0);
 DWORD dwError = GetLastError();
  }
```

Лабораторная работа № 8

БУФЕР ОБМЕНА

Цель работы: изучить основы работы с объектом – буфер обмена.

Изучаемые вопросы

1. Форматы данных.
2. Запись информации в буфер обмена.
3. Чтение информации из буфера обмена.
4. Передача информации пользовательского типа *.

Постановка задачи

Создать приложение, состоящее из двух процессов:

- первый процесс записывает текстовый файл и растровый рисунок в буфер обмена;
- второй процесс считывает информацию из буфера обмена и отображает в окне процесса.

Текстовый файл и файл с растровым рисунком взять из предыдущих лабораторных работ.

Теоретические сведения

Форматы данных

Ниже в таблице 8.1 представлены типы данных и соответствующие им форматы данных.

Таблица 8.1

Форматы данных

Формат	Тип данных
CF_BITMAP	Растр (bitmap) в чистом виде
CF_DIB	Растр (bitmap) с заголовком BITMAPINFO
CF_DIF	Универсальный формат обмена (Data Interchange Format)
CF_DSPBITMAP	Пользовательское растровое изображение
CF_DSPENHMETAFILE	Пользовательский расширенный метафайл
CF_DSPMETAFILEPICT	Пользовательский метафайл
CF_DSPTEXT	Пользовательский текст
CF_ENHMETAFILE	Расширенный метафайл
CF_METAFILEPICT	Метафайл в стиле METAFILEPICT
CF_OEMTEXT	Текст в кодировке OEM
CF_OWNERDISPLAY	Пользовательский формат данных
CF_PALETTE	Цветовая палитра
CF_PENDATA	Формат для данных, связанных с электронным пером
CF_RIFF	Файл ресурсов (Resource Interchange File Format)
CF_SYLK	Символическая ссылка
CF_TEXT	Текст
CF_TIFF	Графика в формате TIFF
CF_WAVE	Звук в формате WAVE
CF_UNICODETEXT	Текст в кодировке UNICODE

Запись информации в буфер обмена

Общая процедура записи данных в буфер обмена состоит из следующих шагов:

a) Прежде чем поместить в буфер обмена какую-либо информацию, ваша программа (далее просто окно) должна его открыть, используя функцию ***OpenClipboard***:

```
if (OpenClipboard(hDlg))  //открываем буфер обмена
```

b) После того, как программа открыла буфер обмена, она должна его очистить от предыдущего содержания, для чего следует вызвать функцию ***EmptyClipboard***.

```
if (EmptyClipboard())     //очищение буфера обмена
```

c) Выделяем блок глобальной памяти, достаточный для того, чтобы хранить в нем данные, которые необходимо поместить в буфер обмена.

```
HGLOBAL hGl;
hGl = GlobalAlloc(GMEM_DDESHARE, strlen(buffer2));
               //заказываем блок памяти
```

Функция ***GlobalAlloc***() выделяет память и возвращает дескриптор выделенного блока.

Для получения указателя на область памяти, выделенную при помощи ***GlobalAlloc***(), следует использовать функцию ***GlobalLock***():

```
LPVOID lpstr = (char *) GlobalLock(hGl); //блокируем его
```

Функция ***GlobalLock***() фиксирует в памяти объект (блок), дескриптор которого передается в параметре hGl. Зафиксированный объект не перемещается в памяти и не выгружается. Функция ***GlobalLock***() возвращает адрес начала блока в случае успешного завершения или NULL при возникновении ошибки.

После получения указателя на глобальный блок памяти необходимо скопировать в этот блок данные, которые Вы хотите поместить в буфер обмена.

```
memcpy(lpstr, buffer2, strlen(buffer2)); //записываем данные
```

Когда копирование завершится, блок памяти можно разблокировать, вызвав функцию GlobalUnlock():

```
GlobalUnlock(hGl);  //разблокировать блок
```

Теперь мы имеем полное право помещать в него свои данные в различных форматах, используя функцию ***SetClipboardData.***

```
SetClipboardData(CF_TEXT, hGl); //помещаем данные в буфер
                                 обмена
```

Чтение информации из буфера обмена

Для чтения данных из буфера обмена используется следующая последовательность шагов:

a) Необходимо открыть буфер обмена;

```
if (OpenClipboard(hDlg))
```

Чтобы получить доступ к данным, хранящимся в буфере обмена, последний должен быть открыт.

b) Чтобы извлечь информацию из буфера обмена, окно должно вызвать функцию ***GetClipboardData***.

```
HANDLE hData = GetClipboardData(CF_TEXT);
     //извлекаем информацию из буфера
```

Данная функция в качестве параметра принимает формат буфера обмена, для того чтобы извлечь данные в этом формате.

Если в буфер обмена данные поместила другая программа, вы можете проверить доступные форматы данных перед их непосредственным извлечением, используя функцию ***IsClipboardFormatAvailable***.

```
if (IsClipboardFormatAvailable(CF_TEXT))
     //проверка на доступные файлы
```

c) Копируем данные из буфера обмена
d) Закрываем буфер обмена

```
CloseClipboard();
```

Пользовательский формат данных

Возможно ситуация, когда приложению необходимо поместить данные в буфер обмена, а стандартные форматы для этого не подходят. Выходом из такой ситуации является возможность регистрации собственного формата данных.

Для того чтобы зарегистрировать новый формат буфера обмена, используется функция ***RegisterClipboardFormat***. В качестве параметра этой функции следует передать указатель на текстовую строку, закрытую двоичным нулем и содержащую имя регистрируемого нестандартного формата данных для буфера обмена.

Функция возвращает нулевое значение при ошибке или идентификатор зарегистрированного формата данных, который можно использовать аналогично идентификаторам стандартных форматов в качестве параметра функции ***SetClipboardData***.

Два различных приложения или две копии одного приложения могут зарегистрировать формат с одним и тем же именем, при этом функция RegisterClipboardFormat вернет один и тот же идентификатор формата. Поэтому два приложения всегда смогут "договориться", если они знают имя нестандартного формата данных.

В приведенном ниже коде регистрируется новый формат данных, представленный структурой MyClipboardData, а затем заполненная структура помещается в буфер обмена:

```
if (LOWORD(wParam) == IDC_BUTTON1)
{
       UINT format =
RegisterClipboardFormat(L"MyClipboardData");
//регистрируем наш формат данных
       MyClipboardData MCD;
       SendDlgItemMessageA(hDlg, IDC_EDIT2, WM_GETTEXT, 50,
(LPARAM)MCD.InfData);
       if (OpenClipboard(hDlg))
//для работы с буфером обмена его нужно открыть
{
if (EmptyClipboard())
{
       HGLOBAL hGl;
       EmptyClipboard();  //очищаем буфер
       hGl = GlobalAlloc(GMEM_DDESHARE,
sizeof(MyClipboardData)); //выделим память
```

```
MyClipboardData * buffer = (MyClipboardData *)
GlobalLock(hGl); //запишем данные в память
       *buffer = MCD;
  //поместим данные в буфер обмена
       GlobalUnlock(hGl);
       SetClipboardData(format, hGl);
 //помещаем данные в буфер обмена
       CloseClipboard();
 //после работы с буфером, его нужно закрыть
}
       return (INT_PTR)TRUE;
}}
```

В приведенном ниже коде извлекается из буфера обмена данные, представленные структурой MyClipboardData:

```
case IDC_BUTTON1:
{
       if (OpenClipboard(hDlg))
       {
       UINT format =
RegisterClipboardFormat(L"MyClipboardData");
 //вызываем второй раз, чтобы просто получить формат
       if (IsClipboardFormatAvailable(format))
       {
        MyClipboardData MCD;
    //извлекаем данные из буфера
       HANDLE hData = GetClipboardData(format);
       MyClipboardData* buffer = (MyClipboardData *)
GlobalLock(hData);
   //заполняем нашу структуру полученными данными
    MCD = *buffer;
       GlobalUnlock(hData)
     SetDlgItemTextA(hDlg, IDC_EDIT1, MCD.In    fData);
       CloseClipboard();
}
}
CloseClipboard();
return TRUE;
}
```

Лабораторная работа № 9

МЕЖПРОЦЕССОРНОЕ ВЗАИМОДЕЙСТВИЕ

Цель работы: изучить основы передачи между процессами информации на базе сообщения WM_COPYDATA.

Изучаемые вопросы

1. Структура COPYDATASTRUCT.
2. Передача текстовой информации.
3. Прием текстовой информации.
4. Передача структурированной информации.
5. Прием структурированной информации.

Постановка задачи

Создать приложение, состоящее из двух процессов:

– первый процесс посылает текстовую и структурированную информацию;
– второй процесс принимает информацию и отображает её в окне процесса.

Теоретические сведения

Структура COPYDATASTRUCT

Структура COPYDATASTRUCT содержит данные, которые будут переданы в другую прикладную программу в соответствии с сообщением WM_COPYDATA.

Синтаксис:

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

где:

dwData – устанавливает до 32 битов данных, которые будут переданы в принимающую прикладную программу;

cbData – устанавливает размер, в байтах, данных, указанных элементом структуры lpData;

lpData – указывает на данные, которые будут переданы в принимающую прикладную программу. Этот элемент структуры может быть значением ПУСТО (NULL).

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

Синтаксис:

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

Параметры:

hwnd – идентифицирует окно, которое передает данные;

pcds – указывает на структуру COPYDATASTRUCT, которая содержит данные для передачи.

Если принимающая программа обрабатывает это сообщение, она должна возвратить значение ИСТИНА (TRUE); в противном случае она должна возвратить – ЛОЖЬ (FALSE).

Для передачи этого сообщения программа должна использовать функцию ***SendMessage***. Данные, предназначенные для передачи, не должны содержать указателей или других ссылок на объекты, не доступные для программы, принимающей эти данные.

До тех пор, пока это сообщение действует, вызванные данные не должны быть изменены другим потоком процесса пересылки. Принимающая программа должна принимать во внимание данные только для чтения. Параметр pcds правилен только в течение обработки сообщения. Принимающая программа не должна освобождать память, вызванную pcds. Если принимающая программа обратилась к данным после возврата значения функцией SendMessage, она должно копировать данные в локальный буфер.

Передача текстовой информации

При передаче текстовой информации подготавливается строка, которая отправляется в выбранное окно (находимое

функцией ***FindWindow***) функцией ***SendMessage*** с параметром WM_COPYDATA:

```
SetDlgItemText(hDlg,IDC_EDIT_TEXT,L"Разоренов Николай");
HWND hWnd1 = FindWindowA(NULL,"2");

	cds.dwData = MYMEM;
	cds.cbData = 50;
	char n1[50];
	n1[0] = 0;
	GetDlgItemTextA(hDlg,IDC_EDIT_TEXT,n1,sizeof(n1)*2);
	cds.lpData = n1;
	if (hWnd1 != NULL)
	SendMessage(hWnd1, WM_COPYDATA, (WPARAM)(HWND) hDlg,
(LPARAM) (LPVOID) &cds);
```

Прием текстовой информации

При получении текстовой информации во втором процессе проверяется, не было ли использовано WM_COPYDATA. При правильном заполнении структуры COPYDATASTRUCT происходит примерно следующее:

```
case WM_COPYDATA:
{
	pcds = (PCOPYDATASTRUCT) lParam;
	switch( pcds->dwData )
	{
case MYMEM:
	{
SetDlgItemTextA(hDlg,IDC_EDIT_TEXT,(CHAR *)pcds->lpData);
	break;
}
```

Передача структурированной информации

Передача структурированной информации отличается от передачи строки тем, что в параметр lpData структуры COPYDATASTRUCT помещается структура с данными:

```
HWND hWnd1 = FindWindowA(NULL,"2");


	BMPStruct.biSize    = BmpInfoHeader->biSize ;
	BMPStruct.biWidth = BmpInfoHeader->biWidth;
```

```
      BMPStruct.biHeight  = BmpInfoHeader->biHeight ;
      BMPStruct.biBitCount  = BmpInfoHeader->biBitCount ;
      BMPStruct.biClrUsed  = BmpInfoHeader->biClrUsed ;

      cds.dwData = MYSTRUCT;
//устанавливает до 32 битов данные, которые будут переданы
      cds.cbData = sizeof(BITMAPFH_STRUCT); //размер данных
      cds.lpData = &BMPStruct;
      //данные, которые будут переданы
      if (hWnd1 != NULL)
      SendMessage(hWnd1, WM_COPYDATA, (WPARAM)(HWND) hDlg,
(LPARAM) (LPVOID) &cds);
```

Прием структурированной информации.

При приеме структурированной информации из параметра lpData извлекается переданная структура и происходит (в нашем случае) вывод полученной информации на экран:

```
   case MYSTRUCT:
{
      char *sBuf;
      DWORD n;
      BMPStruct = (BITMAPFH_STRUCT *) pcds->lpData;

      n = (DWORD) &BMPStruct->biSize;
      SetDlgItemInt(hDlg,IDC_EDIT2,n,FALSE);
      n = (DWORD) BMPStruct->biWidth;
      SetDlgItemInt(hDlg,IDC_EDIT3,n,FALSE);
      n = (DWORD) BMPStruct->biHeight;
      SetDlgItemInt(hDlg,IDC_EDIT4,n,FALSE);
      n = (DWORD) BMPStruct->biBitCount;
      SetDlgItemInt(hDlg,IDC_EDIT1,n,FALSE);
      n = (DWORD) BMPStruct->biClrUsed;
      SetDlgItemInt(hDlg,IDC_EDIT5,n,FALSE);
      break;
   }
```

Литература

1. Рихтер, Дж. Windows для профессионалов: создание эффективных Win32 приложений с учетом специфики 64-разрядной версии Windows / Дж. Рихтер; пер. с англ. – 4-е изд. – СПб.: Питер, М.: Издательско-торговый дом «Русская редакция», 2001. – 752 с.

2. Шилдт, Г. Полный справочник по С++ / Г. Шилдт. – 4-е изд. – М.: Вильямс, 2006. – 796 с.

3. Петзолд, Ч. Программирование для Windows 95: в 2 т. / Ч. Петзолд; пер. с англ. – СПб.: BHV – Санкт-Петербург, 1997. – Т. 2. – 368 с.

4. Гордеев, А.В. Системное программное обеспечение / А.В. Гордеев, А.Ю. Молчанов. – СПб.: Питер, 2003. – 736 с.

5. Румянцев, П.В. Азбука программирования в WIN32 API / П.В. Румянцев. – СПб.: Питер, 2004. – 310 с.

6. Разработка приложений на Microsoft Visual C++ 6.0. Учебный кур: Официальное пособие Microsoft для самостоятельной подготовки / пер. с англ. – М.: Издательско-торговый дом «Русская Редакция», 2000. – 576 с.

9. Джонсон, М. Харт. Системное программирование в среде Windows / М. Харт Джонсон. – М.: Издательский дом «Вильямс», 2001. – 464 с.

10. Рихтер, Дж. Программирование серверных приложений для Microsoft Windows 2000. Мастер-класс. / Дж. Рихтер, Дж. Д. Кларк: пер. с англ. – СПб.: Питер, М.: Издательско-торговый дом «Русская редакция», 2001. – 592 с.

ПРИЛОЖЕНИЕ

Варианты заданий лабораторной работы № 5

№	Система координат	Рисунок	Пояснения
1	LOENGLISH		Рисунок нарисованный сплошной черной линией (———) появляется на экране при запуске программы, нарисованный пунктирной линией (· · · · · · · · · ·) появляется через Δt1 и является зеркальным отражением начального рисунка. По направлению указанному стрелками выводится вводимый с клавиатуры текст.
2	HIMETRIC		На экране по трем точкам (координатам курсора мыши при щелчке) рисуется треугольник. Его размеры изменяются как показано на рисунке при «оттягивании» одного из его углов мышью. По направлению указанному стрелками выводится вводимый с клавиатуры текст.
3	LOENGLISH		На экране по двум точкам (координатам курсора мыши при щелчке) рисуется прямоугольник. Его размеры изменяются как показано на рисунке при «оттягивании» нижнего правого угла мышью. По направлению указанному стрелками выводится вводимый с клавиатуры текст.
4	TWIPS		На экране нарисовать данную фигуру. По направлению указанному стрелками выводится вводимый с клавиатуры текст.

№	Система координат	Рисунок	Пояснения
5	LOMETRIC		На экране нарисовать данную фигуру. По направлению указанному стрелками выводится вводимый с клавиатуры текст.
6	ANISOTROPIC		На экране нарисовать данную фигуру. По направлению указанному стрелками выводится вводимый с клавиатуры текст.
7	ISOTROPIC		На экране нарисовать данную фигуру. По направлению указанному стрелками выводится вводимый с клавиатуры текст.
8	LOENGLISH		На экране нарисовать данную фигуру. По направлению указанному стрелками выводится вводимый с клавиатуры текст.
9	HIMETRIC		На экране нарисовать данную фигуру. По направлению указанному стрелками выводится вводимый с клавиатуры текст.
10	LOENGLISH		На экране нарисовать данную фигуру. По направлению указанному стрелками выводится вводимый с клавиатуры текст.

№	Си-стема коор-динат	Рисунок	Пояснения
11	TWIPS		На экране нарисовать данную фигуру. По направлению указанному стрелками выводится вводимый с клавиатуры текст. По щелку мыши текст начинает выводиться с другой стороны от вертикальной оси.
12	LOMETRIC		На экране нарисовать данную фигуру. По направлению указанному стрелками выводится вводимый с клавиатуры текст.
13	ISOTROPIC		На экране нарисовать данную фигуру. По направлению указанному стрелками выводится вводимый с клавиатуры текст.
14	LOENGLISH		На экране нарисовать карандаш. По щелчку мыши изменяется цвет стержня. По направлению указанному стрелками выводится вводимый с клавиатуры текст.
15	HIMETRIC		На экране по двум точкам (координатам курсора мыши при щелчке) рисуется прямоугольник. Его размеры изменяются, как показано на рисунке при «оттягивании» нижнего правого угла мышью. По направлению указанному стрелками выводится вводимый с клавиатуры текст.

№	Система координат	Рисунок	Пояснения
16	LOENGLISH		На экране нарисовать N прямоугольников высоты h_1, h_2, h_3,… Каждый прямоугольник имеет свою штриховку. Высота прямоугольника выводится над прямоугольником.
17			На экране изобразить диаграмму по значениям T1, T2, T3, T4 вводимым с клавиатуры.
18	ANISOTROPIC		На экране изобразить график заданной функции. Сделать подписи к осям и над опущенными на ось X перпендикулярами указать значения функции при данном X.
19			Написать игру «Крестики-нолики». По щелку левой кнопкой мыши в данной клеточке появляется крестик, правой — нолик. По окончании игры предусмотреть объявление победителя.

Варианты заданий лабораторной работы № 6

Фигуры, на которые на рисунке указывает курсор мыши, двигаются по экрану при нажатой левой клавиши мыши по траектории движения мыши.

Переменные С1, С2, С3…и т.д. обозначают цвета фигур и участков экрана. Если индексы в переменных разные, то и цвета должны быть разные.

Вывод всех рисунков на экран осуществляется методом двойной буферизации.

№	Рисунок	Пояснения
1		По экрану по показанной траектории двигается круг, изменяя свой радиус.
2		По экрану по показанной траектории двигается круг, изменяя свой радиус и цвет.
3	С3 С2 С1	На экране цвета С3 через Δt1 появляется фигура цвета С2, а через Δt2 фигура цвета С1.
4	С1 С2 С3 С4 С5 С6	Фигуры из каждого сектора экрана через Δt1 после запуска программы начинают двигаться вверх.

№	Рисунок	Пояснения
5		Через Δt1 после запуска программы на экране получить данную картинку.
6		По экрану по показанной траектории двигается круг, изменяя свой радиус и цвет.
7		В программе имитировать восход солнца.
8		В программе имитировать приближение автомобиля к плоскости экрана.
9	C1 C3 C2 C4	Фигуры появляются на экране друг за другом последовательно через Δt1.

№	Рисунок	Пояснения
10	C1 C2 C3	На экране имитировать движение маятника. Левая кнопка мыши колебания начинаются, правая – заканчиваются.
11	C1 C2 C3	На экране имитировать движение маятника. Правая кнопка мыши колебания начинаются, левая – заканчиваются.
12	C1 C2	Фигура появляется на экране через Δt1.
13		По траектории показанной на рисунке движется круг, изменяя свой цвет в зависимости от своего местоположения.
14	C1 C2 C3	На экране цвета C1 через Δt1 появляется круг цвета C2, а через Δt2 круг цвета C3, который начинает хаотично двигаться внутри другого круга.
15	C1 C3 C2 C4	Фигуры появляются на экране друг за другом последовательно через Δt1.

№	Рисунок	Пояснения
16		По экрану по показанной траектории двигается эллипс, изменяя свой цвет.
17		По экрану по показанной траектории двигается круг, изменяя свой радиус.
18	С3 С1 С2 С1 С2 С1 С2 С1 С2 С4	На экране получить разноцветную фигуру, по которой может двигаться круг.
19		На экране сымитировать мерцание звезд на ночном небе
20		На экране проиллюстрировать одновременное движение автомобиля и полет воздушного шара в указанных направлениях. Предметы начинают движение по левому щелчку мыши.

Учебное издание

СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Лабораторный практикум
для студентов специальностей 1-40 01 02
«Информационные системы и технологии» и 1-40 01 01
«Программное обеспечение информационных технологий»

Составитель
РАЗОРЁНОВ Николай Александрович

Технический редактор О.В. Песенько

Подписано в печать 20.02.2012.
Формат $60\times84^{1}/_{16}$. Бумага офсетная.
Отпечатано на ризографе. Гарнитура Таймс.
Усл. печ. л. 4,71. Уч.-изд. л. 3,68. Тираж 100. Заказ 1065.

Издатель и полиграфическое исполнение:
Белорусский национальный технический университет.
ЛИ № 02330/0494349 от 16.03.2009.
Проспект Независимости, 65. 220013, Минск.