

Министерство образования и науки Российской Федерации
Магнитогорский государственный технический университет
им. Г.И. Носова

А.Н. Калитаев, В.Д. Тутарова, Д.Н. Мазнин, Ю.В. Кочержинская

**ПРАКТИКУМ ПО ДИСЦИПЛИНЕ
«ТЕОРИЯ ВЫЧИСЛИТЕЛЬНЫХ ПРОЦЕССОВ»**

Утверждено Редакционно-издательским советом университета
в качестве практикума

Магнитогорск
2016

УДК 004.421.2
ББК 32.973-018
К 172

Рецензенты:

Заместитель директора по учебно-методической работе
Новотроицкого филиала ФГАОУ ВО «НИТУ МИСИС»,
кандидат технических наук, доцент
С.Н. Басков

Начальник бюро автоматизированных систем управления
АНО ДПО «КЦПК «Персонал»
А.Ю. Тяжельников

Калитаев А.Н.

К 172 Практикум по дисциплине «Теория вычислительных процессов»: практикум / А.Н. Калитаев, В.Д. Тутарова, Д.Н. Мазнин, Ю.В. Кочержинская. Магнитогорск: Изд-во Магнитогорск. гос. техн. ун-та им. Г.И. Носова, 2016. 80 с.

Данное учебное издание представляет собой практикум по дисциплине «Теория вычислительных процессов». Пособие включает в себя теоретические аспекты разработки системного программного обеспечения, а также практические приемы и навыки в области разработки системного программного обеспечения в операционных системах семейства *Windows NT* (32- и 64-разрядных версиях) и *Windows 9x* с использованием функций *Windows API*.

Предназначено для студентов направления 09.03.01 – «Информатика и вычислительная техника».

УДК 004.421.2
ББК 32.973-018

© Магнитогорский государственный
технический университет
им. Г.И. Носова, 2016
© Калитаев А.Н., Тутарова В.Д.,
Мазнин Д.Н., Кочержинская Ю.В. 2016

ОГЛАВЛЕНИЕ

Оглавление	3
Введение	4
Лабораторная работа №1. Чтение карты процессов и потоков	5
Лабораторная работа №2. Чтение карты памяти	11
Лабораторная работа №3. Многопоточковая обработка	22
Лабораторная работа №4. Средства межпроцессного взаимодействия – каналы (pipe).....	33
Лабораторная работа №5. Файлы данных, проецируемые в память	41
Курсовая работа. Организация взаимодействия и синхронизации параллельных процессов и потоков	52
Контрольные вопросы	73
Заключение	74
Библиографический список	75
Приложение А	76

ВВЕДЕНИЕ

В операционной системе *Windows* существуют средства, позволяющие использовать системные ресурсы непосредственно в прикладных программах. Эти средства объединены в совокупность системных процедур и функций, принадлежащих ядру ОС и ее надстройкам. Множество этих процедур и функций получило название интерфейса прикладного программирования *API* (*Application Program Interface*). Полное название интерфейса – *Win32 API* означает, что эти средства поддерживаются семейством 32-разрядных ОС *Windows: Windows NT* и *Windows 9x*. В настоящее время разработан и используется 64-разрядный интерфейс *Win64 API*, который ориентирован в основном на большие серверные приложения и некоторые рабочие станции. Одной из наиболее интересных особенностей системы программирования *Borland Developer Studio* является предоставление, наряду с высокоуровневыми функциями *VCL*, простого доступа к функциям *Windows API*. Программист в любой момент имеет возможность (в зависимости от стоящей перед ним задачи) выбрать для ее решения простые в использовании компоненты, либо реализовать алгоритм, требующий компактности и быстродействия, при помощи прямых вызовов *API*. Более того, как правило, можно без прекращения использования компонентов и визуального программирования внести в программу небольшие дополнения при помощи средств *API* и добиться максимальной точности решения задачи и быстродействия. Применение процедур и функций *Win32 API* в прикладных программах мало, чем отличается от использования обычных процедур и функций пользователя. Все процедуры и функции *Win32 API* имеют общепринятые имена и списки параметров. Часть параметров задается в виде именованных констант, другие описываются в программе обычным образом. При вызове процедуры или функции указываются ее имя и в скобках значения фактических параметров, порядок размещения и типы которых совпадают со значениями формальных параметров. В настоящее время интерфейс *Win32 API* насчитывает несколько сотен процедур и функций, которые можно разбить на несколько групп [1].

ЛАБОРАТОРНАЯ РАБОТА №1. ЧТЕНИЕ КАРТЫ ПРОЦЕССОВ И ПОТОКОВ

Цель работы

Изучить функции и процедуры семейства *ToolHelp32*, составляющих подмножество *Win32 API*, которые позволяют получить сведения о некоторых низкоуровневых аспектах работы ОС. В частности, получить информацию обо всех процессах, выполняющихся в системе в данный момент, а также потоках, модулях, принадлежащих каждому процессу.

Информация

Большинство данных, получаемых от функций *ToolHelp32*, используется, главным образом, приложениями, которые должны заглядывать «внутри» ОС [2].

Моментальные снимки

Благодаря многозадачной природе ОС, такие объекты, как процессы, потоки, модули и т.п., постоянно создаются, разрушаются и модифицируются. И поскольку состояние компьютера непрерывно изменяется, системная информация, которая, возможно, будет иметь значение в данный момент, через секунду уже никого не заинтересует. Например, предположим, что необходимо написать программу для регистрации всех модулей, загруженных в систему. Поскольку операционная система в любое время может прервать выполнение потока, отработавшего программу, чтобы предоставить какие-то кванты времени другому потоку в системе, модули теоретически могут создаваться и разрушаться даже в момент выборки информации о них.

В этой динамической среде имеет смысл сделать «снимок» системы в заданный момент времени. Данный снимок делается с помощью функции *CreateToolhelp32Snapshot*.

```
HANDLE WINAPI CreateToolhelp32Snapshot(  
    DWORD dwFlags,  
    DWORD th32ProcessID);
```

Параметр *dwFlags* означает тип информации, подлежащий включению в моментальный снимок. Этот параметр может иметь одно из значений, перечисленных в табл. 1.

Второй параметр, *th32ProcessID*, задает идентификатор процесса. Для текущего процесса данный параметр принимает значение 0. Этот параметр используется в том случае, если параметр *dwFlags* принимает значения *TH32CS_SNAPHEAPLIST* или *TH32CS_SNAPMODULE*. В остальных случаях игнорируется (принимает значение 0).

Таблица 1

Значение параметра *dwFlags*

Значение	Описание
TH32CS_INHERIT	Означает, что дескриптор снимка будет наследуемым
TH32CS_SNAPALL	Эквивалентно заданию значений: TH32CS_SNAPHEAPLIST, TH32CS_SNAPMODULE, TH32CS_SNAPPROCESS, TH32CS_SNAPTHREAD
TH32CS_SNAPHEAPLIST	Включает в снимок список куч заданного процесса
TH32CS_SNAPMODULE	Включает в снимок список модулей заданного процесса
TH32CS_SNAPPROCESS	Включает в снимок список процессов
TH32CS_SNAPTHREAD	Включает в снимок список потоков

Функция *CreateToolhelp32Snapshot* возвращает дескриптор созданного снимка или *-1* в случае ошибки. Возвращаемый дескриптор работает подобно другим дескрипторам относительно процессов и потоков, для которых он действителен. По завершении работы с созданным функцией *CreateToolhelp32Snapshot* дескриптором, для освобождения связанных с ним ресурсов используйте функцию *CloseHandle*.

Обработка информации о процессах

Имея дескриптор снимка, содержащий информацию о процессах, можно воспользоваться двумя функциями, которые позволяют последовательно просмотреть сведения обо всех процессах в системе. Функции *Process32First* и *Process32Next* определены следующим образом:

```
BOOL WINAPI Process32First(
    HANDLE hSnapshot,
    LPPROCESSENTRY32 lppe);
```

```
BOOL WINAPI Process32Next(
    HANDLE hSnapshot,
    LPPROCESSENTRY32 lppe);
```

Первый параметр, *hSnapshot*, у обеих функций является дескриптором снимка, возвращаемым функцией *CreateToolhelp32Snapshot*.

Второй параметр, *lppe*, представляет собой структуру *PROCESSENTRY32*, которая передается по ссылке. По мере прохождения

по элементам перечисления функции будут заполнять эту структуру информацией о следующем процессе. Запись `PROCESSENTRY32` определяется так:

```
typedef struct tagPROCESSENTRY32 {
    DWORD dwSize;
    DWORD cntUsage;
    DWORD th32ProcessID;
    DWORD th32DefaultHeapID;
    DWORD th32ModuleID;
    DWORD cntThreads;
    DWORD th32ParentProcessID;
    LONG  pcPriClassBase;
    DWORD dwFlags;
    char  szExeFile[MAX_PATH];
} PROCESSENTRY32;
typedef PROCESSENTRY32 * PPROCESSENTRY32;
typedef PROCESSENTRY32 * LPPROCESSENTRY32;
```

Поля структуры:

- `dwSize` – размер структуры `PROCESSENTRY32`. До использования этой записи поле `dwSize` должно быть инициализировано значением `sizeof(PROCESSENTRY32)`;
- `cntUsage` – значение счетчика ссылок процесса. Когда это значение станет равным нулю, операционная система выгрузит процесс;
- `th32ProcessID` – идентификационный номер процесса.
- `th32DefaultHeapID` – идентификатор *ID* для кучи процесса, действующей по умолчанию. Этот *ID* имеет значение только для функций *ToolHelp32*, и его нельзя использовать с другими функциями *Win32*;
- `th32ModuleID` – идентифицирует модуль, связанный с процессом. Это поле имеет значение только для функций *ToolHelp32*;
- `cntThreads` – количество потоков начало выполняться в данном процессе;
- `th32ParentProcessID` – идентифицирует родительский процесс для данного процесса;
- `pcPriClassBase` – базовый приоритет процесса. Операционная система использует это значение для управления работой потоков;
- `dwFlags` – зарезервировано (не используется);
- `szExeFile` содержит строку с ограничивающим нуль-символом, которая представляет собой путь и имя файла EXE-программы или драйвера, связанного с данным процессом.

После создания снимка, содержащего информацию о процессах, для опроса данных по каждому процессу следует вызвать сначала функцию *Process32First*, а затем вызывать функцию *Process32Next* до тех пор, пока она не вернет значение false.

Обработка информации о потоках

Для составления списка потоков некоторого процесса в *ToolHelp32* предусмотрены две функции, которые аналогичны функциям, предназначенным для регистрации процессов: *Thread32First* и *Thread32Next*, и объявляются следующим образом:

```
BOOL WINAPI Thread32First(  
    HANDLE hSnapshot,  
    LPTHREADENTRY32 lpTe);  
  
BOOL WINAPI Thread32Next(  
    HANDLE hSnapshot,  
    LPTHREADENTRY32 lpTe);
```

Помимо обычного параметра *hSnapshot* (дескриптор снимка, возвращаемым функцией *CreateToolhelp32Snapshot*), этим функциям также передается по ссылке параметр типа *THREADENTRY32*. Как и в случае функций, работающих с процессами, каждая из них заполняет запись *THREADENTRY32*, объявление которой имеет вид:

```
typedef struct tagTHREADENTRY32{  
    DWORD dwSize;  
    DWORD cntUsage;  
    DWORD th32ThreadID;  
    DWORD th32OwnerProcessID;  
    LONG tpBasePri;  
    LONG tpDeltaPri;  
    DWORD dwFlags;  
} THREADENTRY32;  
typedef THREADENTRY32 * PTHREADENTRY32;  
typedef THREADENTRY32 * LPTHREADENTRY32;
```

Поля структуры:

- *dwSize* – размер структуры, и поэтому оно должно быть инициализировано значением *sizeof(THREADENTRY32)* до использования этой структуры;
- *cntUsage* – счетчик ссылок данного потока. При обнулении этого счетчика поток выгружается операционной системой;
- *th32ThreadID* – идентификационный номер потока, который имеет значение только для функций *ToolHelp32*;

- `th32OwnerProcessID` – идентификатор *ID* процесса, которому принадлежит данный поток. Этот *ID* можно использовать с другими функциями *Win32*;
- `tpBasePri` – базовый класс приоритета потока. Это значение одинаково для всех потоков данного процесса. Описания этих значений приведены в табл. 2.
- `dwFlags` – зарезервировано (не используется).

Таблица 2

Значение констант приоритетов (параметр *tpBasePri*)

Константа	Значение
THREAD_PRIORITY_IDLE	-15
THREAD_PRIORITY_LOWEST	-2
THREAD_PRIORITY_BELOW_NORMAL	-1
THREAD_PRIORITY_NORMAL	0
THREAD_PRIORITY_ABOVE_NORMAL	1
THREAD_PRIORITY_HIGHEST	2
THREAD_PRIORITY_TIME_CRITICAL	15

Списки потоков, полученные с помощью функций *ToolHelp32*, не связываются с определенным потоком. Поэтому при сканировании потоков нужно обязательно проверять результат так, чтобы потоки были связаны с интересующим вас потоком.

Обработка информации о модулях

Опрос модулей выполняется практически так же, как опрос процессов или потоков. Для этого в *ToolHelp32* предусмотрены функции *Module32First* и *Module32Next*, которые определяются следующим образом:

```
BOOL WINAPI Module32First(
    HANDLE hSnapshot,
    LPMODULEENTRY32 lpme);
```

```
BOOL WINAPI Module32Next(
    HANDLE hSnapshot,
    LPMODULEENTRY32 lpme);
```

Пример. С помощью функций и процедур семейства *ToolHelp32* считать информацию о процессах и потоках, запущенных в системе в определенный момент времени (выполнить «моментальный снимок - snapshot»).

```
#include <vcl.h>
#include <tlhelp32.h>
```

```

int main()
{
    // Читаем карту процессов
    HANDLE WINAPI SnapShot_Pr;
    SnapShot_Pr=CreateToolhelp32Snapshot (TH32CS_SNAPPROCESS,0)
    ;
    if ((int)SnapShot_Pr==-1)
        ShowMessage("Не могу прочитать карту процессов");
    tagPROCESSENTRY32 ProcEntry;
    ProcEntry.dwSize=sizeof(ProcEntry);
    // Считываем первый процесс из списка
    Process32First (SnapShot_Pr,&ProcEntry);
    // Читаем его идентификатор
    DWORD ProcID=ProcEntry.th32ProcessID;
    // Имя исполняемого файла
    AnsiString ProcName=ProcEntry.szExeFile;
    // Считываем количество потоков
    DWORD ProcThreadcount=ProcEntry.cntThreads;
    int i=1;

    ...

    // Пока не опустеет список, читаем процессы
    while (Process32Next (SnapShot_Pr,&ProcEntry))
    {
        i++;
        ProcID=ProcEntry.th32ProcessID;
        ProcName=ProcEntry.szExeFile;
        ProcThreadcount=ProcEntry.cntThreads;

        ...
    } // С потоками поступаем так же
    return 0;
}

```

Задание. Требуется создать программу, позволяющую прочитать список запущенных в системе процессов и потоков. На рис. 1 приведен фрагмент работы программы.

Процессы:			Потоки:	
ID процесса	Имя исполняемого файла	Количество потоков	ID потока	ID родительского процесса
0	[System Process]	1	4076	4
4	System	71	2152	4
976	SMSS.EXE	3	2496	4
1068	CSRSS.EXE	15	2492	4
1096	WINLOGON.EXE	23	3972	4
1140	SERVICES.EXE	15	980	976
1152	LSASS.EXE	18	984	976
1300	ATI2EVXX.EXE	5	988	976
1312	SVCHOST.EXE	17	1076	1068
1396	SVCHOST.EXE	10	1080	1068

Процессы: 63 Потоков: 601

Карта процессов и потоков

Рис. 1. Карта процессов и потоков

ЛАБОРАТОРНАЯ РАБОТА №2. ЧТЕНИЕ КАРТЫ ПАМЯТИ

Цель работы

Получение практических навыков по использованию Win32 API для исследования памяти Windows.

Информация

Виртуальное адресное пространство процесса

Каждому процессу выделяется собственное виртуальное адресное пространство [3]. Для 32-разрядных процессов его размер составляет 4 Гб. Соответственно 32-битный указатель может быть любым числом от 0x00000000 до 0xFFFFFFFF. Всего, таким образом, указатель может принимать 4 294 967 296 значений, что как раз и перекрывает четырехгигабайтовый диапазон. Для 64-разрядных процессов размер адресного пространства равен 16 экзабайтам, поскольку 64-битный указатель может быть любым числом от 0x00000000 00000000 до 0xFFFFFFFF FFFFFFFF.

Поскольку каждому процессу отводится закрытое адресное пространство, то когда в процессе выполняется какой-нибудь поток, он получает доступ только к той памяти, которая принадлежит его процессу. Память, отведенная другим процессам, скрыта от этого потока и недоступна ему.

Виртуальное адресное пространство каждого процесса разбивается на разделы. Их размер и назначение в какой-то мере зависят от конкретного ядра Windows (табл. 3).

В разделе «Для кода и данных пользовательского режима» располагается закрытая (неразделяемая) часть адресного пространства процесса. Ни один процесс не может получить доступ к данным другого процесса, размещенным в этом разделе. Основной объем данных, принадлежащих процессу, хранится именно в этом разделе (это касается всех приложений). Поэтому приложения менее зависимы от взаимных «капризов», и вся система функционирует устойчивее.

Windows-функции, сообщающие о состоянии системной памяти и виртуального адресного пространства в процессах

Многие параметры операционной системы (размер страницы, гранулярность выделения памяти и др.) зависят от используемого в компьютере процессора. Поэтому нельзя жестко «зашивать» их значения в исходный код программ. Эту информацию необходимо считывать в момент инициализации процесса с помощью функции *GetSystemInfo*:

```
VOID GetSystemInfo(LPSYSTEM_INFO lpSystemInfo);
```

Таблица 3

Разделы адресного пространства процесса

Раздел	32-разрядная Windows 2000 (на x86 и Alpha)	64-разрядная Windows 2000 (на Alpha и IA-64)	Windows 98
Для выявления нулевых указателей	0x00000000 0x0000FFFF	0x00000000 00000000 0x00000000 0000FFFF	0x00000000 0x000000FF
Для совместимости с программами DOS и 16-разрядной Windows	Нет	Нет	0x00001000 0x003FFFFF
Для кода и данных пользовательского режима	0x00010000 0x7FFEFFFF	0x00000000 00010000 0x000003FF FFFEFFFF	0x00400000 0x7FFFFFFF
Закрытый, размером 64 Кб	0x7FFF0000 0x7FFFFFFF	0x000003FF FFFF0000 0x000003FF FFFFFFFF	Нет
Для общих MMF (файлов, проецируемых в память)	Нет	Нет	0x80000000 0xBFFFFFFF F
Для кода и данных режима ядра	0x80000000 0xFFFFFFFF	0x00000400 00000000 0xFFFFFFFF FFFFFFFF	0xC0000000 0xFFFFFFFF

В функцию *GetSystemInfo* передается адрес структуры `SYSTEM_INFO`, и функция инициализирует элементы этой структуры:

```
typedef struct _SYSTEM_INFO {
    union {
        DWORD dwOemId;
        struct {
            WORD wProcessorArchitecture;
            WORD wReserved;
        };
    };
    DWORD dwPageSize;
    LPVOID lpMinimumApplicationAddress;
    LPVOID lpMaximumApplicationAddress;
    DWORD dwActiveProcessorsMask;
    DWORD dwNumberOfProcessors;
    DWORD dwProcessorType;
    DWORD dwAllocationGranularity;
    WORD wProcessorLevel;
    WORD wProcessorRevision;
} SYSTEM_INFO;
```

При загрузке система определяет значения элементов этой структуры; для конкретной системы их значения постоянны. Функция *GetSystemInfo* предусмотрена специально для того, чтобы и приложения могли получать эту информацию (рис. 2). Из всех элементов структуры SYSTEM_INFO лишь четыре имеют отношение к памяти (табл. 4).

Таблица 4

Элементы структуры SYSTEM_INFO

Элемент	Описание
dwPageSize	Размер страницы памяти. На процессорах x86 это значение равно 4096, а на процессорах Alpha – 8 192 байтам
lpMinimumApplicationAddress	Минимальный адрес памяти доступного адресного пространства для каждого процесса. В Windows 98 это значение равно 4 194 304, или 0x00400000, поскольку нижние 4 Мб адресного пространства каждого процесса недоступны. В Windows 2000 это значение равно 65536, или 0x00010000, так как в этой системе резервируются лишь первые 64 Кб адресного пространства каждого процесса
lpMaximumApplicationAddress	Максимальный адрес памяти доступного адресного пространства, отведенного в «личное пользование» каждому процессу. В Windows 98 этот адрес равен 2 147 483 647, или 0x7FFFFFFF, так как верхние 2 Гб занимают общие файлы, проецируемые в память, и разделяемый код операционной системы. В Windows 2000 этот адрес соответствует началу раздела для кода и данных режима ядра за вычетом 64 Кб
dwAllocationGranularity	Гранулярность резервирования регионов адресного пространства.

Остальные элементы структуры SYSTEM_INFO приведены в табл. 5.

Таблица 5

Элементы структуры SYSTEM_INFO

Элемент	Описание
dwOemId	Устарел; больше не используется
wReserved	Зарезервирован на будущее; пока не

Элемент	Описание
	используется
dwNumberOfProcessors	Число процессоров в компьютере
dwActiveProcessorMask	Битовая маска, которая сообщает, какие процессоры активны (выполняют потоки)
dwProcessorType	Используется только в Windows 98; сообщает тип процессора, например Intel 386, 486 или Pentium
wProcessorArchitecture	Используется только в Windows 2000; сообщает тип архитектуры процессора, например Intel, Alpha, 64-разрядный Intel или 64-разрядная Alpha
wProcessorLevel	Используется только в Windows 2000; сообщает дополнительные подробности об архитектуре процессора, например Intel Pentium Pro или Pentium II
wProcessorRevision	Используется только в Windows 2000; сообщает дополнительные подробности об уровне данной архитектуры процессора

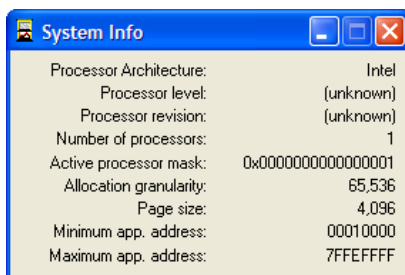


Рис. 2. Системная информация [Дж. Рихтер. Windows], полученная при вызове функции *GetSystemInfo*

В Windows имеется функция, позволяющая запрашивать определенную информацию об участке памяти по заданному адресу (в пределах адресного пространства вызывающего процесса): размер, тип памяти и атрибуты защиты. Описание функции *VirtualQuery*:

```
DWORD VirtualQuery(
    LPCVOID lpAddress,
    PMEMORY_BASIC_INFORMATION lpBuffer,
    DWORD dwLength);
```

Парная ей функция, *VirtualQueryEx*, сообщает ту же информацию о памяти, но в другом процессе. Описание функции *VirtualQueryEx*:

```
DWORD VirtualQueryEx(
    HANDLE hProcess,
```

```
LPCVOID lpAddress,
PMEMORY_BASIC_INFORMATION lpBuffer,
DWORD dwLength);
```

Эти функции идентичны с тем исключением, что *VirtualQueryEx* принимает описатель процесса, об адресном пространстве которого необходимо получить информацию. Чаще всего функцией *VirtualQueryEx* пользуются отладчики и системные утилиты – остальные приложения обращаются к *VirtualQuery*. При вызове *VirtualQuery(Ex)* параметр *lpAddress* должен содержать адрес виртуальной памяти, о которой необходимо получить информацию. Параметр *lpBuffer* – это адрес структуры *MEMORY_BASIC_INFORMATION*, которую надо создать перед вызовом функции. Данная структура определена в следующем виде:

```
typedef struct _MEMORY_BASIC_INFORMATION { // mbi
    PVOID BaseAddress;
    PVOID AllocationBase;
    DWORD AllocationProtect;
    DWORD RegionSize;
    DWORD State;
    DWORD Protect;
    DWORD Type;
} MEMORY_BASIC_INFORMATION;
typedef
    MEMORY_BASIC_INFORMATION *PMEMORY_BASIC_INFORMATION;
```

Параметр *dwLength* задает размер структуры *MEMORY_BASIC_INFORMATION*. Функция *VirtualQuery(Ex)* возвращает число байтов, скопированных в буфер.

Используя адрес, указанный в параметре *lpAddress*, функция *VirtualQuery(Ex)* заполняет структуру информацией о диапазоне смежных страниц, имеющих одинаковое состояние, атрибуты защиты и тип. Описание элементов структуры приведено в табл. 6.

Отдельным страницам физической памяти можно присвоить свои атрибуты защиты, представленные в табл. 7.

Так как функция *VirtualQueryEx* принимает описатель процесса, возникает проблема получения описателя (дескриптора) процесса по известному идентификатору процесса. По идентификатору можно определить дескриптор любого процесса с помощью функции *OpenProcess*:

```
HANDLE OpenProcess(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    DWORD dwProcessId);
```

Таблица 6

Элементы структуры MEMORY_BASIC_INFORMATION

Элемент	Описание
BaseAddress	Сообщает то же значение, что и параметр <i>lpAddress</i> , но округленное до ближайшего меньшего размера, кратного размеру страницы
AllocationBase	Идентифицирует базовый адрес региона, включающего в себя адрес, указанный в параметре <i>lpAddress</i>
AllocationProtect	Идентифицирует атрибут защиты, присвоенный региону при его резервировании (табл. 7)
RegionSize	Сообщает суммарный размер (в байтах) группы страниц, которые начинаются с базового адреса <i>BaseAddress</i> и имеют те же атрибуты защиты, состояние и тип, что и страница, расположенная по адресу, указанному в параметре <i>lpAddress</i>
State	Сообщает состояние (MEM_FREE, MEM_RESERVE или MEM_COMMIT) всех смежных страниц, которые имеют те же атрибуты защиты, состояние и тип, что и страница, расположенная по адресу, указанному в параметре <i>lpAddress</i> . При MEM_FREE элементы <i>AllocationBase</i> , <i>AllocationProtect</i> , <i>Protect</i> и <i>Type</i> содержат неопределенные значения, а при MEM_RESERVE неопределенное значение содержит элемент <i>Protect</i>
Protect	Идентифицирует атрибут защиты (PAGE_*) всех смежных страниц, которые имеют те же атрибуты защиты, состояние и тип, что и страница, расположенная по адресу, указанному в параметре <i>lpAddress</i> (табл. 7)
Type	Идентифицирует тип физической памяти (MEM_IMAGE, MEM_MAPPED или MEM_PRIVATE) (табл. 8), связанной с группой смежных страниц, которые имеют те же атрибуты защиты, состояние и тип, что и страница, расположенная по адресу, указанному в параметре <i>lpAddress</i> .

Атрибуты защиты

Атрибут защиты	Описание
PAGE_NOACCESS	Попытки чтения, записи или исполнения содержимого памяти на этой странице вызывают нарушение доступа
PAGE_READONLY	Попытки записи или исполнения содержимого памяти на этой странице вызывают нарушение доступа
PAGE_READWRITE	Попытки исполнения содержимого памяти на этой странице вызывают нарушение доступа
PAGE_EXECUTE	Попытки чтения или записи на этой странице вызывают нарушение доступа
PAGE_EXECUTE_READ	Попытки записи на этой странице вызывают нарушение доступа
PAGE_EXECUTE_READWRITE	На этой странице возможны любые операции
PAGE_WRITECOPY	Попытки исполнения содержимого памяти на этой странице вызывают нарушение доступа; попытка записи приводит к тому, что процессу предоставляется «личная» копия данной страницы
PAGE_EXECUTE_WRITECOPY	На этой странице возможны любые операции; попытка записи приводит к тому, что процессу предоставляется «личная» копия данной страницы
Специальные флаги атрибутов защиты	
PAGE_NOCACHE	Отключает кэширование переданных страниц. Данный флаг предусмотрен главным образом для разработчиков драйверов устройств при манипулировании буферами памяти
PAGE_GUARD	Позволяет приложениям получать уведомление (через механизм исключений) в тот момент, когда на страницу записывается какой-нибудь байт
PAGE_WRITECOMBINE	Предназначен для разработчиков драйверов устройств. Позволяет объединять несколько операций записи на устройство в один пакет, что увеличивает скорость передачи данных

Типы регионов памяти приведены в табл. 8.

Таблица 8

Типы регионов памяти

Тип	Описание
Free	Этот диапазон виртуальных адресов не сопоставлен ни с каким типом физической памяти. Его адресное пространство не зарезервировано; приложение может зарезервировать регион по указанному базовому адресу или в любом месте в границах свободного региона
Private	Этот диапазон виртуальных адресов сопоставлен со страничным файлом
Image	Этот диапазон виртуальных адресов изначально был сопоставлен с образом EXE- или DLL-файла, проецируемого в память, но теперь, возможно, уже нет. Например, при записи в глобальную переменную из образа модуля механизм поддержки «копирования при записи» выделяет соответствующую страницу памяти из страничного файла, а не исходного образа файла
Mapped	Этот диапазон виртуальных адресов изначально был сопоставлен с файлом данных, проецируемым в память, но теперь, возможно, уже нет. Например, файл данных мог быть спроецирован с использованием механизма поддержки «копирование при записи». Любые операции записи в этот файл приведут к тому, что соответствующие страницы памяти будут выделены из страничного файла, а не из исходного файла данных

Параметр *dwDesiredAccess* имеет отношение к правам доступа и может принимать различные значения (табл. 9).

Таблица 9

Значения параметра

Значение	Описание
PROCESS_ALL_ACCESS	Эквивалентно установке флагов полного доступа
PROCESS_CREATE_PROCESS	Для внутреннего использования
PROCESS_CREATE_THREAD	Позволяет использовать дескриптор процесса в функции <i>CreateRemoteThread</i> для создания потоков в процессе
PROCESS_DUP_HANDLE	Использует дескриптор, как исходного процесса, так и принимающего в функции <i>DuplicateHandle</i> для копирования (дублирования) дескриптора
PROCESS_QUERY_INFORMATION	Задействует дескриптор процесса для чтения информации из объекта <i>Process</i>
PROCESS_SET_INFORMATION	Позволяет использовать дескриптор

Значение	Описание
	процесса в <i>SetPriorityClass</i> функцию, чтобы установить класс приоритета процесса
PROCESS_TERMINATE	Работает для завершения процесса с его дескриптором в функции <i>TerminateProcess</i>
PROCESS_VM_OPERATION	Использует дескриптор процесса для модификации виртуальной памяти процесса
PROCESS_VM_READ	Применяет для чтения из виртуальной памяти процесса его дескриптора в функции <i>ReadProcessMemory</i>
PROCESS_VM_WRITE	Использует для записи в виртуальную память процесса его дескриптора в функции <i>WriteProcessMemory</i>
SYNCHRONIZE	Windows NT: работает с дескриптором процесса в любой из функций ожидания, таких как <i>WaitForSingleObject</i> , для ожидания завершения процесса

Параметр *bInheritHandle* – установлен в значение TRUE, для того чтобы позволить порожденным процессам наследовать дескриптор. Иначе говоря, порожденный процесс получает дескриптор родительского процесса. Отметим, что значение дескриптора может изменяться.

Параметр *dwProcessID* – должен иметь значение идентификатора того процесса, дескриптор которого нужно узнать.

Функция *OpenProcess* возвращает дескриптор указанного процесса.

Пример. Определить информацию о первом регионе памяти адресного пространства (суммарный размер (в байтах) группы страниц, которые начинаются с базового адреса (минимальный адрес памяти доступного адресного пространства) и имеют те же атрибуты защиты, состояние и тип, что и страница, расположенная по данному адресу) вызываемого процесса.

```
{
    _SYSTEM_INFO sysinfo;
    GetSystemInfo(&sysinfo);
    LPVOID minAddress=sysinfo.lpMinimumApplicationAddress,
           maxAddress=sysinfo.lpMaximumApplicationAddress;
    LabelEdit1->Text=IntToHex((int)minAddress,8);
    LabelEdit2->Text=IntToHex((int)maxAddress,8);
    _MEMORY_BASIC_INFORMATION meminfo;
    VirtualQuery(minAddress,&meminfo,sizeof(meminfo));
}
```

```

LabelEdit3->Text=IntToStr(meminfo.RegionSize);
}

```

Результат работы программы представлен на рис. 3.

Минимальный адрес	00010000
Максимальный адрес	7FFEFFFF
Размер первого региона памяти, байт	4096

Рис. 3. Информация об адресном пространстве и первом регионе памяти

Адрес следующего региона получаем так:

<Текущий адрес региона> + <объем текущего региона>

Далее продолжаем итерации, пока не доберемся до максимального из доступных адресов.

Задание. Требуется создать программу, позволяющую прочесть информацию (размер, тип памяти и атрибуты защиты) о регионах памяти адресного пространства процесса, как для вызываемого процесса (рис. 4), так и для любого процесса, запущенного в системе (рис. 5).

Карта памяти_Пример

Current process... Минимальный адрес 00010000 Максимальный адрес 7FFEFFFF

BaseAddress	AllocationBase	AllocationProtect	RegionSize	State	Protect	Type
00010000	00010000	PAGE_READWRITE	4096	MEM_COMMIT	PAGE_READWRITE	MEM_PRIVATE
00011000	00000000	-	61440	MEM_FREE	PAGE_NOACCESS	-
00020000	00020000	PAGE_READWRITE	4096	MEM_COMMIT	PAGE_READWRITE	MEM_PRIVATE
00021000	00000000	-	61440	MEM_FREE	PAGE_NOACCESS	-
00030000	00030000	PAGE_READWRITE	1032192	MEM_RESERVE	-	MEM_PRIVATE
0012C000	00030000	PAGE_READWRITE	4096	MEM_COMMIT	-	MEM_PRIVATE
0012D000	00030000	PAGE_READWRITE	12288	MEM_COMMIT	PAGE_READWRITE	MEM_PRIVATE
00130000	00130000	PAGE_READWRITE	12288	MEM_COMMIT	PAGE_READWRITE	MEM_MAPPED
00133000	00000000	-	53248	MEM_FREE	PAGE_NOACCESS	-
00140000	00140000	PAGE_READWRITE	192512	MEM_COMMIT	PAGE_READWRITE	MEM_PRIVATE
0016F000	00140000	PAGE_READWRITE	856064	MEM_RESERVE	-	MEM_PRIVATE
00240000	00240000	PAGE_READWRITE	24576	MEM_COMMIT	PAGE_READWRITE	MEM_PRIVATE
00246000	00240000	PAGE_READWRITE	40960	MEM_RESERVE	-	MEM_PRIVATE
00250000	00250000	PAGE_READWRITE	12288	MEM_COMMIT	PAGE_READWRITE	MEM_MAPPED
00253000	00250000	PAGE_READWRITE	53248	MEM_RESERVE	-	MEM_MAPPED
00260000	00260000	PAGE_READWRITE	90112	MEM_COMMIT	PAGE_READWRITE	MEM_MAPPED
00276000	00000000	-	40960	MEM_FREE	PAGE_NOACCESS	-
00280000	00280000	PAGE_READWRITE	249856	MEM_COMMIT	PAGE_READWRITE	MEM_MAPPED

Регионов памяти: 256

Карта памяти

Рис. 4. Карта памяти вызываемого процесса

Карты памяти_Пример						
EXPLORER.EXE			Минимальный адрес: 00010000		Максимальный адрес: 7FFFFFFF	
BaseAddress	AllocationBase	AllocationProtect	RegionSize	State	Protect	Type
00010000	00010000	PAGE_READWRITE	4096	MEM_COMMIT	PAGE_READWRITE	MEM_PRIVATE
00011000	00000000	.	61440	MEM_FREE	PAGE_NOACCESS	.
00020000	00020000	PAGE_READWRITE	4096	MEM_COMMIT	PAGE_READWRITE	MEM_PRIVATE
00021000	00000000	.	61440	MEM_FREE	PAGE_NOACCESS	.
00030000	00030000	PAGE_READWRITE	200704	MEM_RESERVE	.	MEM_PRIVATE
00061000	00030000	PAGE_READWRITE	4096	MEM_COMMIT	.	MEM_PRIVATE
00062000	00030000	PAGE_READWRITE	57344	MEM_COMMIT	PAGE_READWRITE	MEM_PRIVATE
00070000	00070000	PAGE_READONLY	12288	MEM_COMMIT	PAGE_READONLY	MEM_MAPPED
00073000	00000000	.	53248	MEM_FREE	PAGE_NOACCESS	.
00080000	00080000	PAGE_READWRITE	1048576	MEM_COMMIT	PAGE_READWRITE	MEM_PRIVATE
00180000	00180000	PAGE_READWRITE	32768	MEM_COMMIT	PAGE_READWRITE	MEM_PRIVATE
00188000	00180000	PAGE_READWRITE	32768	MEM_RESERVE	.	MEM_PRIVATE
00190000	00190000	PAGE_READWRITE	12288	MEM_COMMIT	PAGE_READWRITE	MEM_MAPPED
00193000	00190000	PAGE_READWRITE	53248	MEM_RESERVE	.	MEM_MAPPED
001A0000	001A0000	PAGE_READONLY	90112	MEM_COMMIT	PAGE_READONLY	MEM_MAPPED
001B6000	00000000	.	40960	MEM_FREE	PAGE_NOACCESS	.
001C0000	001C0000	PAGE_READONLY	249856	MEM_COMMIT	PAGE_READONLY	MEM_MAPPED
001FD000	00000000	.	12288	MEM_FREE	PAGE_NOACCESS	.
Регионов памяти: 1333						
Карты памяти						

Рис. 5. Карта памяти процесса (explorer.exe), запущенного в системе

ЛАБОРАТОРНАЯ РАБОТА №3. МНОГОПОТОКОВАЯ ОБРАБОТКА

Цель работы

Изучить функции, предназначенные для создания (порождения) дополнительных потоков в системе, принципы использования системой объектов ядра «поток» для управления потоками.

Информация

Любой поток состоит из двух компонентов:

- объекта ядра, через который операционная система управляет потоком. Там же хранится статистическая информация о потоке;
- стека потока, который содержит параметры всех функций и локальные переменные, необходимые потоку для выполнения кода.

Процесс ничего не исполняет, он просто служит контейнером потоков. Потоки всегда создаются в контексте какого-либо процесса, и вся их жизнь проходит только в его границах. На практике это означает, что потоки исполняют код и манипулируют данными в адресном пространстве процесса. Поэтому, если два и более потока выполняется в контексте одного процесса, все они делят одно адресное пространство. Потоки могут исполнять один и тот же код и манипулировать одними и теми же данными, а также совместно использовать описатели объектов ядра, поскольку таблица описателей создается не в отдельных потоках, а в процессах [3].

Поток (*thread*) определяет последовательность исполнения кода в процессе. При инициализации процесса система всегда создает первичный поток. Большинство приложений обходится единственным, первичным потоком. Однако процессы могут создавать дополнительные потоки, что позволяет им эффективнее выполнять свою работу.

Каждый поток начинает выполнение с некоей входной функции. В первичном потоке таковым является *main*, *wmain*, *WinMain* или *wWinMain*. Если необходимо создать вторичный поток, в нем должна быть входная функция, которая выглядит примерно так:

```
DWORD WINAPI ThreadFunc(PVOID lpParam) {  
    DWORD dwResult=0;  
    ...  
    return(dwResult)  
}
```

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 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.

В основе реализации функции потока заложены следующие требования.

- В отличие от входной функции первичного потока, у которой должно быть одно из четырех имен: *main*, *wmain*, *WinMain*, *wWinMain*, – функцию потока можно назвать как угодно. Однако, если в программе несколько функций потоков, необходимо присвоить им разные имена, иначе компилятор или компоновщик решит, что создается несколько реализаций единственной функции.
- Поскольку входным функциям первичного потока передаются строковые параметры, они существуют в ANSI- и Unicode-версиях: *main* – *wmain* и *WinMain* – *wWinMain*. Но функциям потока передается единственный параметр, смысл которого определяется программистом, а не операционной системой. Поэтому проблем с ANSI/Unicode нет.
- Функция потока должна возвращать значение, которое будет использоваться как код завершения потока. Полная аналогия с библиотекой C/C++: код завершения первичного потока становится кодом завершения процесса.
- Функции потоков (да и все функции) должны по мере возможности обходиться своими параметрами и локальными переменными. Так как к статической или глобальной переменной могут одновременно обратиться несколько потоков, есть повредить ее содержимое. Однако параметры и локальные переменные создаются в стеке потока, поэтому они в гораздо меньшей степени подвержены влиянию другого потока.

Реализовав функцию потока, необходимо, чтобы бы операционная система создала поток, который выполнит эту функцию.

Создание потока

Для создания дополнительных потоков необходимо вызвать из первичного потока функцию *CreateThread*:

```
HANDLE CreateThread(  
    LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    DWORD dwStackSize,  
    LPTHREAD_START_ROUTINE lpStartAddress,  
    LPVOID lpParameter,  
    DWORD dwCreationFlags,
```

```
LPDWORD lpThreadId);
```

При каждом вызове этой функции система создает объект ядра «поток». Это не сам поток, а компактная структура данных, которая используется операционной системой для управления потоком и хранит статистическую информацию о потоке.

ПРИМЕЧАНИЕ *CreateThread* — это Windows-функция, создающая поток. Если Вы пишете код на C/C++ эффективнее использовать функцию *_beginthreadex* из библиотеки Visual C++.

Параметр *lpThreadAttributes* является указателем на структуру SECURITY_ATTRIBUTES. Если необходимо, чтобы объекту ядра «поток» были присвоены атрибуты защиты по умолчанию (что чаще всего и бывает), передается в этом параметре NULL. А чтобы дочерние процессы смогли наследовать описатель этого объекта, необходимо определить структуру SECURITY_ATTRIBUTES и инициализировать ее элемент *binheritHandle* значением TRUE.

Параметр *dwStackSize* определяет, какую часть адресного пространства поток сможет использовать под свой стек. Каждому потоку выделяется отдельный стек. Если при обращении к функции *CreateThread*, передается в параметре *dwStackSize* ненулевое значение, функция резервирует всю указанную память. Ее объем определяется либо значением параметра *dwStackSize*, либо значением, заданным в ключе /STACK (/STACK:[*reserve*][, *commit*]) — аргумент *reserve* определяет объем адресного пространства, который система должна зарезервировать под стек потока (по умолчанию — 1Мб); аргумент *commit* задает объем физической памяти, который изначально передается области, зарезервированной под стек (по умолчанию — 1 страница) компоновщика (выбирается большее из них). Но передается стеку лишь тот объем памяти, который соответствует значению в *dwStackSize*. Если же в параметре *dwStackSize* передается нулевое значение, *CreateThread* создает стек для нового потока, используя информацию, встроенную компоновщиком в EXE-файл.

Параметр *lpStartAddress* определяет адрес функции потока, с которой должен будет начать работу создаваемый поток, а параметр *lpParameter* идентичен параметру *lpParameter* функции потока. *CreateThread* лишь передает этот параметр по эстафете той функции, с которой начинается выполнение создаваемого потока. Таким образом, данный параметр позволяет передавать функции потока какое-либо инициализирующее значение. Оно может быть или просто числовым значением, или указателем на структуру данных с дополнительной информацией. Вполне допустимо и даже полезно создавать несколько потоков, у которых в качестве входной точки используется адрес одной и

той же функции. Например, можно реализовать Web-сервер, который обрабатывает каждый клиентский запрос в отдельном потоке. При создании каждому потоку передается свое значение *lpParameter*. Так как Windows – операционная система с вытесняющей многозадачностью, следовательно, новый поток и поток, вызвавший *CreateThread*, могут выполняться одновременно, что может привести к определенным проблемам.

Параметр *dwCreationFlags* определяет дополнительные флаги, управляющие созданием потока. Он принимает одно из двух значений: 0 (исполнение потока начинается немедленно) или *CREATE_SUSPENDED*. В последнем случае система создает поток, инициализирует его и приостанавливает до последующих указаний. Флаг *CREATE_SUSPENDED* позволяет программе изменить какие-либо свойства потока перед тем, как он начнет выполнять код.

Последний параметр *lpThreadId* функции *CreateThread* – это адрес переменной типа *DWORD*, в которой функция возвращает идентификатор, приписанный системой новому потоку.

ПРИМЕЧАНИЕ В Windows 2000 и Windows NT в этом параметре можно передавать *NULL*. Тем самым сообщается функции, что программиста не интересует идентификатор потока. Но в Windows 95/98 это приведет к ошибке, так как функция попытается записать идентификатор потока по нулевому адресу, что недопустимо. И поток не будет создан.

Завершение потока

Поток можно завершить четырьмя способами [3]:

- 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 (нежелательный способ).

Возврат управления функцией потока

Функцию потока следует проектировать так, чтобы поток завершался только после того, как она возвращает управление. Это единственный способ, гарантирующий корректную очистку всех ресурсов, принадлежащих потоку. При этом:

- любые C++-объекты, созданные данным потоком, уничтожаются соответствующими деструкторами;

- система корректно освобождает память, которую занимал стек потока;
- система устанавливает код завершения данного потока (поддерживаемый объектом ядра «поток») – его и возвращает функция потока;
- счетчик пользователей данного объекта ядра «поток» уменьшается на 1.

Функция ExitThread

Поток можно завершить принудительно, вызвав:

```
VOID ExitThread(DWORD dwExitCode);
```

При этом освобождаются все ресурсы операционной системы, выделенные данному потоку, но C/C++ ресурсы (например, объекты, созданные из C++-классов) не очищаются. Именно поэтому лучше возвращать управление из функции потока, чем самому вызывать функцию *ExitThread*.

В параметре *dwExitCode* помещается значение, которое система рассматривает как код завершения потока. Возвращаемого значения у этой функции нет, так как после ее вызова поток перестает существовать.

ПРИМЕЧАНИЕ *ExitThread* – это Windows-функция, которая уничтожает поток. Если Вы пишете код на C/C++ эффективнее использовать функцию *_endthreadex* из библиотеки Visual C++.

Функция TerminateThread

Вызов этой функции также завершает поток.

```
BOOL TerminateThread(  
    HANDLE hThread,  
    DWORD dwExitCode);
```

В отличие от *ExitThread*, которая уничтожает только вызывающий поток, эта функция завершает поток, указанный в параметре *hThread*. В параметре *dwExitCode* указывается значение, которое система рассматривает как код завершения потока. После того как поток будет уничтожен, счетчик пользователей его объекта ядра «поток» уменьшится на 1. Корректно написанное приложение не должно вызывать эту функцию, поскольку поток не получает никакого уведомления о завершении; из-за этого он не может выполнить должную очистку ресурсов.

ПРИМЕЧАНИЕ Уничтожение потока при вызове *ExitThread* или возврате управления из функции потока приводит к разрушению стека. Но если он завершен функцией *TerminateThread*, система не уничтожает стек, пока он не завершится и процесс, которому принадлежал этот

поток. Так сделано потому, что другие потоки могут использовать указатели, ссылающиеся на данные в стеке заверщенного потока. Если бы они обратились к несуществующему стеку, произошло бы нарушение доступа. Кроме того, при завершении потока система уведомляет об этом все DLL, подключенные к процессу – владельцу заверщенного потока. Но при вызове *TerminateThread* такого не происходит, и процесс может быть завершен некорректно.

Завершение процесса, содержащего данный поток

Функции *ExitProcess* и *TerminateProcess* тоже завершают потоки. Единственное отличие в том, что они прекращают выполнение всех потоков, принадлежащих завершённому процессу. При этом гарантируется высвобождение любых выделенных процессу ресурсов, в том числе стеков потоков. Однако эти две функции уничтожают потоки принудительно – так, будто для каждого из них вызывается функция *TerminateThread*. А это означает, что очистка проводится некорректно: деструкторы C++-объектов не вызываются, данные на диск не сбрасываются и т.д.

При завершении потока сопоставленный с ним объект ядра «поток» не освобождается до тех пор, пока не будут закрыты все внешние ссылки на этот объект.

Для проверки завершен ли поток, идентифицируемый описателем *hThread*, из других потоков, запущенных в системе, используется функция *GetExitCodeThread*.

```
BOOL GetExitCodeThread(  
    HANDLE hThread,  
    LPDWORD lpExitCode);
```

Код завершения возвращается в переменной типа **DWORD**, на которую указывает *lpExitCode*. Если поток не завершен на момент вызова *GetExitCodeThread*, функция записывает в эту переменную идентификатор **STILL_ACTIVE** (0x103). При успешном вызове функция возвращает **TRUE**.

Планирование потоков

Операционная система с вытесняющей многозадачностью должна использовать тот или иной алгоритм, позволяющий ей распределять процессорное время между потоками. Каждые 20 мс (или около того) Windows просматривает все существующие объекты ядра «поток» и отмечает те из них, которые могут получать процессорное время. Далее она выбирает один из таких объектов и загружает в регистры процессора значения из его контекста. Эта операция называется *переключением контекста* (context switching). По каждому потоку Windows ведет учет

того, сколько раз он подключался к процессору. Поток выполняет код и манипулирует данными в адресном пространстве своего процесса. Примерно через 20 мс Windows сохранит значения регистров процессора в контексте потока и приостановит его выполнение. Далее система просмотрит остальные объекты ядра «поток», подлежащие выполнению, выберет один из них, загрузит его контекст в регистры процессора, и все повторится. Этот цикл операций – выбор потока, загрузка его контекста, выполнение и сохранение контекста – начинается с момента запуска системы и продолжается до ее выключения. Таков вкратце механизм планирования работы множества потоков.

Приостановка и возобновление потоков

В объекте ядра «поток» имеется переменная – счетчик числа простоев данного потока. При вызове *CreateProcess* или *CreateThread* он инициализируется значением, равным 1, которое запрещает системе выделять новому потоку процессорное время. Такая схема весьма разумна: сразу после создания поток не готов к выполнению, ему нужно время для инициализации.

После того как поток полностью инициализирован, *CreateProcess* или *CreateThread* проверяет, не передан ли ей флаг `CREATE_SUSPENDED`, и, если да, возвращает управление, оставив поток в приостановленном состоянии. В ином случае счетчик простоев обнуляется, и поток включается в число планируемых – если только он не ждет какого-то события (например, ввода с клавиатуры).

Создав поток в приостановленном состоянии, можно настроить некоторые его свойства (например, приоритет). Закончив настройку, необходимо разрешить выполнение потока. Для этого вызывается функция *ResumeThread* и передается описатель потока *hThread*, возвращенный функцией *CreateThread* (описатель можно взять и из структуры, на которую указывает параметр *lpProcessInformation*, передаваемый в *CreateProcess*).

```
DWORD ResumeThread(HANDLE hThread);
```

Если вызов *ResumeThread* прошел успешно, она возвращает предыдущее значение счетчика простоев данного потока; в противном случае – `0xFFFFFFFF`.

Выполнение отдельного потока можно приостанавливать несколько раз. Если поток приостановлен 3 раза, то и возобновлен он должен быть тоже 3 раза – лишь тогда система выделит ему процессорное время. Выполнение потока можно приостановить не только при его создании с флагом `CREATE_SUSPENDED`, но и вызовом функции *SuspendThread*:

```
DWORD SuspendThread(HANDLE hThread);
```

Любой поток может вызвать эту функцию и приостановить выполнение другого потока (если его описатель известен). Приостановить свое выполнение поток способен сам, а возобновить себя – нет. Как и функция *ResumeThread*, функция *SuspendThread* возвращает предыдущее значение счетчика простоев данного потока. Поток можно приостанавливать не более чем **MAXIMUM_SUSPEND_COUNT** раз. Функция *SuspendThread* в режиме ядра работает асинхронно, но в пользовательском режиме не выполняется, пока поток остается в приостановленном состоянии.

Дополнительные функции для работы с потоками

Функция Sleep

Поток может сообщить системе о невыделении ему процессорного времени на определенный период, вызвав:

```
VOID Sleep(DWORD dwMilliseconds);
```

Эта функция приостанавливает поток на *dwMilliseconds* миллисекунд.

Функция SwitchToThread

Функция *SwitchToThread* позволяет подключить к процессору другой поток (если он есть):

```
BOOL SwitchToThread(VOID)
```

SwitchToThread позволяет потоку, которому не хватает процессорного времени, отнять этот ресурс у потока с более низким приоритетом. Она возвращает **FALSE**, если на момент ее вызова в системе нет ни одного потока, готового к исполнению; в ином случае – ненулевое значение.

Вызов функции *SwitchToThread* аналогичен вызову функции *Sleep* с передачей в *dwMilliseconds* нулевого значения. Разница лишь в том, что функция *SwitchToThread* дает возможность выполнять потоки с более низким приоритетом, которым не хватает процессорного времени, а функция *Sleep* действует без оглядки на «голодающие» потоки.

Функции изменения приоритетов потоков

Windows поддерживает шесть классов приоритета: *idle* (простаивающий), *below normal* (ниже обычного), *normal* (обычный), *above normal* (выше обычного), *high* (высокий) и *realtime* (реального времени). Самый распространенный класс приоритета – *normal*; его используют 99% приложений. Для потоков Windows поддерживает семь относительных приоритетов: *idle* (простаивающий), *lowest* (низший), *below normal* (ниже обычного), *normal* (обычный), *above normal* (выше

обычного), *highest* (высший) и *time-critical* (критический по времени). Эти приоритеты относительно класса приоритета процесса. Большинство потоков использует обычный приоритет. Поэтому только что созданный поток получает относительный приоритет *normal*. При этом функция *CreateThread* не позволяет задать относительный приоритет. Операция изменения относительного приоритета потока осуществляется вызовом функции:

```
BOOL SetThreadPriority(
    HANDLE hThread,
    int nPriority);
```

Параметр *hThread* указывает на поток, чей приоритет необходимо изменить, а через параметр *nPriority* передается один из идентификаторов, соответствующий определенному приоритету потока (табл. 10).

Таблица 10

Значения параметра *nPriority* в соответствии с приоритетом потока

Относительный приоритет потока	Идентификатор
Time-critical	THREAD_PRIORITY_TIME_CRITICAL
Highest	THREAD_PRIORITY_HIGHEST
Above normal	THREAD_PRIORITY_ABOVE_NORMAL
Normal	THREAD_PRIORITY_NORMAL
Below normal	THREAD_PRIORITY_BELOW_NORMAL
Lowest	THREAD_PRIORITY_LOWEST
Idle	THREAD_PRIORITY_IDLE

Функция *GetThreadPriority*, парная *SetThreadPriority*, позволяет узнать относительный приоритет потока:

```
int GetThreadPriority(HANDLE hThread);
```

Она возвращает один из идентификаторов, приведенных в табл. 10.

Пример. Создать поток с относительным приоритетом *idle* и функцию потока, с которой должен будет начать работу создаваемый поток. В качестве параметра из первичного (главного) потока процесса в функцию потока передается текущее время, а функция создаваемого потока выводит переданную информацию на экран.

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

DWORD WINAPI ThreadFunc(PVOID lpParam)
{
    struct time tt=((struct time*)lpParam);
```

```

    printf("\n The current time is: %2d:%02d:%02d.%02d\n",
           tt.ti_hour, tt.ti_min, tt.ti_sec, tt.ti_hund);
    return 0;
}

int main()
{
    struct time t;
    gettimeofday(&t);
    DWORD dwThreadId;
    HANDLE hThread = CreateThread(NULL, 0, ThreadFunc, &t,
                                  CREATE_SUSPENDED, &dwThreadId);
    SetThreadPriority(hThread, THREAD_PRIORITY_IDLE);
    ResumeThread(hThread);
    CloseHandle(hThread);
    getchar();
    return 0;
}

```

Результат работы программы представлен на рис. 6.

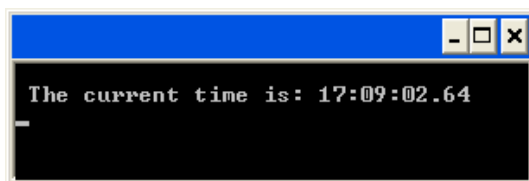


Рис. 6. Результат работы функции потока

Задание: Выполнить имитатор гонок (в просторечии эта задача известна как «тараканьи бега») при помощи создания нескольких потоков (рис. 7). Каждый поток обслуживает свою «беговую дорожку». На исполнение все потоки запускаются одновременно, после чего потоки произвольным образом приостанавливаются и запускаются вновь. На исполнение каждому потоку выделяется квант времени (например, 500 мс или 1 с). За этот период поток производит выполнение задачи, например, увеличивает позицию гонщика на некоторую величину. После истечения кванта времени поток приостанавливается на произвольный период времени, определяемый при помощи генератора случайных чисел. После завершения гонки производится выдача результатов (очередность завершения).

Использование класса *TThread*, включенного в поставку *Borland Developer Studio*, допускается только в ознакомительных целях.

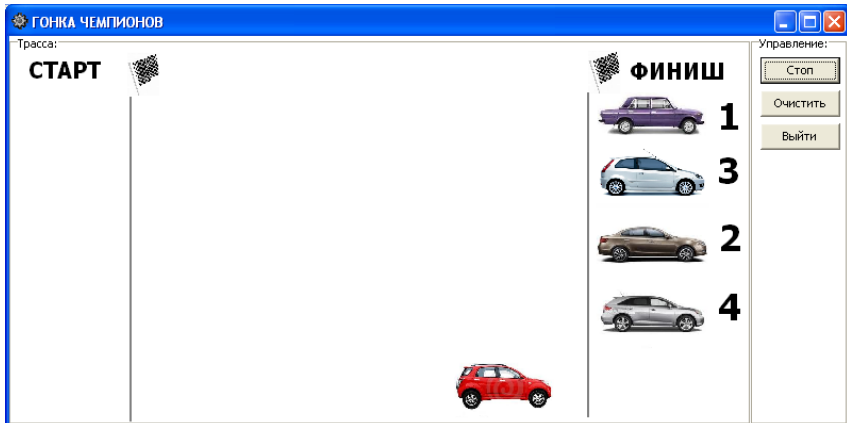
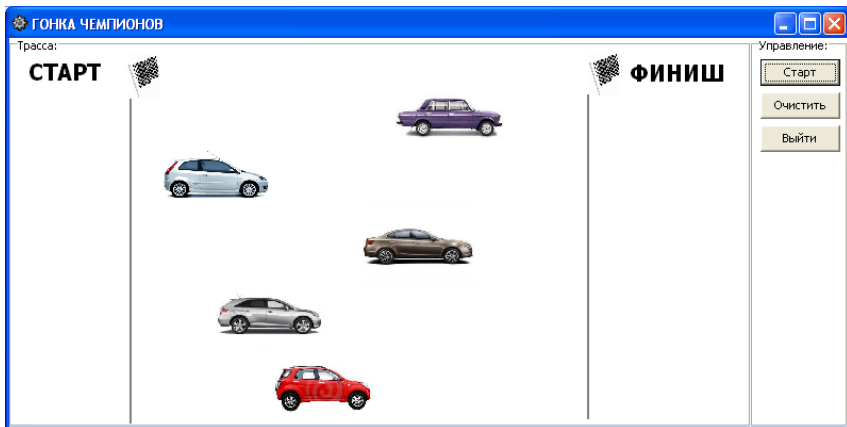


Рис. 7. Фрагменты работы программы «Имитатор гонки»

ЛАБОРАТОРНАЯ РАБОТА №4. СРЕДСТВА МЕЖПРОЦЕССНОГО ВЗАИМОДЕЙСТВИЯ – КАНАЛЫ (PIPE)

Цель работы

Изучение механизмов межпроцессного взаимодействия (InterProcess Communication) в Windows, получение практических навыков по использованию Win32 API для программирования механизмов IPC.

Информация

К механизмам межпроцессного обмена относятся [2]:

- файлы, проецируемые в память (*file mapping*);
- почтовые ящики (*mailslot*).

Почтовые ящики обеспечивают только однонаправленные соединения. Каждый процесс, который создает почтовый ящик, является «сервером почтовых ящиков» (*mailslot server*). Другие процессы, называемые «клиентами почтовых ящиков» (*mailslot clients*), посылают сообщения серверу, записывая их в почтовый ящик. Входящие сообщения всегда дописываются в почтовый ящик и сохраняются до тех пор, пока сервер их не прочтет. Каждый процесс может одновременно быть и сервером, и клиентом почтовых ящиков, создавая, таким образом, двунаправленные коммуникации между процессами.

Клиент может посылать сообщения на почтовый ящик, расположенный на том же компьютере, на компьютере в сети, или на все почтовые ящики с одним именем всем компьютерам выбранного домена. При этом широковещательное сообщение, транслируемое по домену, не может быть более 400 байт. В остальных случаях размер сообщения ограничивается только при создании почтового ящика сервером.

Почтовые ящики предлагают легкий путь для обмена короткими сообщениями, позволяя при этом вести передачу и по локальной сети, в том числе и по всему домену.

Mailslot является псевдофайлом, находящимся в памяти, и следует использовать стандартные функции для работы с файлами, чтобы получить доступ к нему. Данные в почтовом ящике могут быть в любой форме – их интерпретацией занимается прикладная программа, но их общий объем не должен превышать 64 Кб. Однако, в отличие от дисковых файлов, *mailslot* являются временными – когда все дескрипторы почтового ящика закрыты, он и все его данные удаляются. Все почтовые ящики являются локальными по отношению к создавшему их процессу; процесс не может создать удаленный *mailslot*.

Сообщения меньше чем *425 байт* передаются с использованием дитаграмм. Сообщения, больше чем *426 байт*, используют передачу с установлением логического соединения на основе *SMB*-сеансов. Передачи с установлением соединения допускают только индивидуальную передачу от одного клиента к одному серверу. Следовательно, теряется возможность широковещательной трансляции сообщений от одного клиента ко многим серверам. Windows не поддерживает сообщения размером в *425* или *426 байт*.

Когда процесс создает почтовый ящик, имя последнего должно иметь следующую форму:

```
\\.\mailslot\[path]name
```

Например:

```
\\.\mailslot\taxes\bobs_comments
\\.\mailslot\taxes\petes_comments
\\.\mailslot\taxes\sues_comments
```

Если необходимо отправить сообщение в почтовый ящик на удаленный компьютер, то следует воспользоваться NETBIOS-именем:

```
\\ComputerName\mailslot\[path]name
```

Чтобы передать сообщение всем mailslot с указанным именем внутри домена, понадобится NETBIOS-имя домена:

```
\\DomainName\mailslot\[path]name
```

Для главного домена операционной системы (домен, в котором находится рабочая станция):

```
\\*\mailslot\[path]name
```

Клиенты и серверы, использующие почтовые ящики, при работе с ними должны пользоваться функциями, представленными в табл. 11.

Таблица 11

Функции почтовых ящиков

Функция	Описание
<i>Серверов</i>	
CreateMailslot	Создает почтовый ящик и возвращает его дескриптор
GetMailslotInfo	Извлекает максимальный размер сообщения, размер почтового ящика, размер следующего сообщения в ящике, количество сообщений и время ожидания сообщения при выполнении операции чтения
SetMailslotInfo	Изменение таймаута при чтении из почтового

Функция	Описание
	ящика
DuplicateHandle	Дублирование дескриптора почтового ящика
ReadFile ReadFileEx	Считывание сообщений из почтового ящика
GetFileTime	Получение даты и времени создания почтового ящика
SetFileTime	Установка даты и времени создания, модификации почтового ящика
GetHandleInformation	Получение свойств дескриптора почтового ящика
SetHandleInformation	Установка свойств дескриптора почтового ящика
<i>Клиентов</i>	
CreateFile	Создает дескриптор почтового ящика для клиентского процесса
DuplicateHandle	Дублирование дескриптора почтового ящика
WriteFile WriteFileEx	Запись сообщений в почтовый ящик
CloseHandle	Закрывает дескриптор почтового ящика для клиентского процесса

Рассмотрим последовательно все операции, необходимые для корректной работы с почтовыми ящиками.

1. Создание почтового ящика. Операция выполняется процессом сервера с использованием функции *CreateMailslot*:

```
HANDLE CreateMailslot(
    LPCTSTR lpName,           // Имя почтового ящика
    DWORD nMaxMessageSize,    // Максимальный размер сообщения
    DWORD lReadTimeout,       // Таймаут операции чтения
    LPSECURITY_ATTRIBUTES     // Опции наследования и
    lpSecurityAttributes       // безопасности
);
```

2. Запись сообщений в почтовый ящик производится аналогично записи в стандартный дисковый файл с помощью функции *WriteFile*.

3. Чтение сообщений из почтового ящика. Создавший почтовый ящик процесс получает право считывания сообщений, из него используя дескриптор *mailslot* в вызове функции *ReadFile*.

Почтовый ящик существует до тех пор, пока не вызвана функция *CloseHandle* на сервере или пока существует сам процесс сервера. В обоих случаях все непрочитанные сообщения удаляются из почтового ящика, уничтожаются все клиентские дескрипторы, и *mailslot* удаляется из памяти.

Функция считывает параметры почтового ящика:

```

BOOL GetMailslotInfo(
HANDLE hMailslot,          // Дескриптор почтового ящика.
LPDWORD lpMaxMessageSize, // Максимальный размер сообщения.
LPDWORD lpNextSize,       // Размер следующего
                           // непрочитанного сообщения.
LPDWORD lpMessageCount,   // Количество сообщений.
LPDWORD lpReadTimeout     // Таймаут операции чтения.
);

```

Функция устанавливает таймаут операции чтения:

```

BOOL SetMailslotInfo(
HANDLE hMailslot,          // Дескриптор почтового ящика.
DWORD lReadTimeout        // Новый таймаут операции чтения.
);

```

Каналы (pipe)

Существует два способа организовать двустороннее соединение с помощью следующих типов каналов [4]:

1. *Безымянные (анонимные) каналы* позволяют связанным процессам передавать информацию друг другу. Обычно, безымянные каналы используются для перенаправления стандартного ввода/вывода дочернего процесса так, чтобы он мог обмениваться данными с родительским процессом. Чтобы производить обмен данными в обоих направлениях, следует создать два безымянных канала. Родительский процесс записывает данные в первый канал, используя его дескриптор записи, в то время как дочерний процесс считывает данные из канала, используя дескриптор чтения. Аналогично, дочерний процесс записывает данные во второй канал и родительский процесс считывает из него данные. Безымянные каналы не могут быть использованы для передачи данных по сети и для обмена между несвязанными процессами.

2. *Именованные каналы* используются для передачи данных между независимыми процессами или между процессами, работающими на разных компьютерах. Обычно, процесс сервера именованных каналов создает именованный канал с известным именем или с именем, которое будет передано клиентам. Процесс клиента именованных каналов, зная имя созданного канала, открывает его на своей стороне с учетом ограничений, указанных процессом сервера. После этого между сервером и клиентом создается соединение, по которому может производиться обмен данными в обоих направлениях. В организации межпроцессного обмена наибольший интерес представляют именованные каналы.

Общие принципы работы именованных и неименованных каналов:

1. При чтении меньшего числа байт, чем находится в канале, возвращается требуемое число байт, остаток сохраняется для последующих чтений.
2. При чтении большего числа байт, чем находится в канале, возвращается доступное число байт. Процесс, читающий из канала, должен эту ситуацию отработать.
3. Если канал пуст и ни один процесс не открыл его на запись, при чтении из канала будет получено 0 байт. Если один или более процессов открыли канал для записи, вызов на чтение будет заблокирован до появления данных в канале.
4. Запись числа байт, меньшего емкости канала, гарантирована атомарно. В случае, когда несколько процессов одновременно записывают в канал, порции данных от них не перемешиваются.
5. При записи большего числа байт, чем это позволяет канал, вызов на запись блокируется до освобождения требуемого места в канале. Атомарность при этом не гарантируется.

При создании и получении доступа к существующему каналу необходимо придерживаться следующего стандарта имен каналов:

`\\.\pipe\pipename`

Если канал находится на удаленном компьютере, то потребуется NETBIOS-имя компьютера:

`\\ComputerName\pipe\pipename`

Клиентам и серверам для работы с каналами допускается использовать функции, представленные в табл. 12.

Кроме того, для работы с каналами используется функция *CreateFile* (для подключения к каналу со стороны клиента) и функции *WriteFile* и *ReadFile* для записи и чтения данных в/из канала соответственно.

Таблица 12

Функции работы с каналами

Функция	Описание
<code>CallNamedPipe</code>	Выполняет подключение к каналу, записывает в канал сообщение, считывает из канала сообщение и затем закрывает канал
<code>ConnectNamedPipe</code>	Позволяет серверу именованных каналов ожидать подключения одного или нескольких клиентских процессов к экземпляру именованного канала
<code>CreateNamedPipe</code>	Создает экземпляр именованного канала и возвращает дескриптор для последующих операций с каналом

Функция	Описание
CreatePipe	Создает безымянный канал
DisconnectNamedPipe	Отсоединяет серверную сторону экземпляра именованного канала от клиентского процесса
GetNamedPipeHandleState	Получает информацию о работе указанного именованного канала
GetNamedPipeInfo	Извлекает свойства указанного именованного канала
PeekNamedPipe	Копирует данные из именованного или безымянного канала в буфер без удаления их из канала
SetNamedPipeHandleState	Устанавливает режим чтения и режим блокировки вызова функций (синхронный или асинхронный) для указанного именованного канала
TransactNamedPipe	Комбинирует операции записи сообщения в канал и чтения сообщения из канала в одну сетевую транзакцию
WaitNamedPipe	Ожидает, пока истечет время ожидания или пока экземпляр указанного именованного канала не будет доступен для подключения к нему

Пример реализации межпроцессного взаимодействия (клиент-серверного приложения) [5] представлен в приложении А. Результат работы программ межпроцессного взаимодействия приведен на рис. 8.

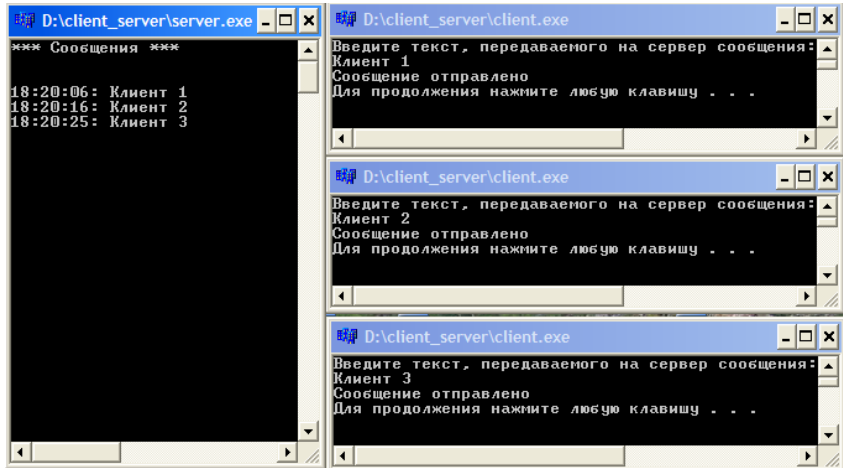


Рис. 8. Результат работы программ межпроцессного взаимодействия

Задание. Организовать работу программы-сервера и нескольких программ-клиентов следующим образом.

Сервер предоставляет клиентам какой-либо из своих ресурсов (например, собственное окно), причем сервер может быть запущен только один.

Клиенты подключаются к серверу и начинают запись в окно, причем первый клиент пишет только «1», второй – только «2», и т.д. в каждый момент времени. Клиентов может быть произвольное количество, но не менее пяти. Предусмотреть возможность отправки на сервер произвольного сообщения.

Если клиент подключается к серверу в монопольном режиме, он получает исключительные права на использование ресурса сервера. Все остальные клиенты, пытающиеся подключиться в данный момент, не должны получить доступ к ресурсу сервера и должны оказаться в очереди на обслуживание.

В разделяемом режиме каждому из подключенных клиентов предоставляется квант времени на исполнение (например, $1c$). Если клиент записывает символы в окно сервера с частотой 1 символ в секунду, то в случае, когда к серверу подключены пять клиентов, окно сервера должно содержать примерно следующую информацию, представленную на рис. 9.

Обмен данными между клиентами и сервером организовать при помощи именованных каналов.

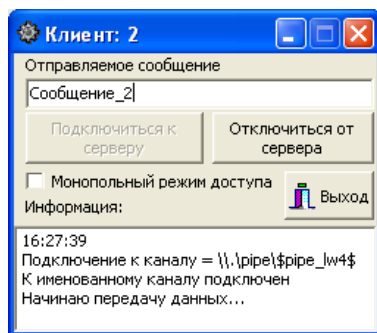
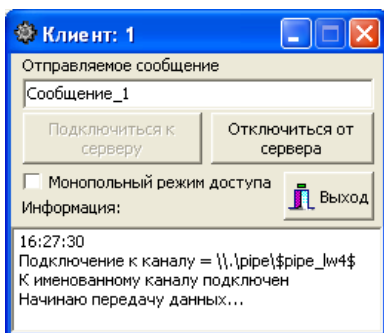
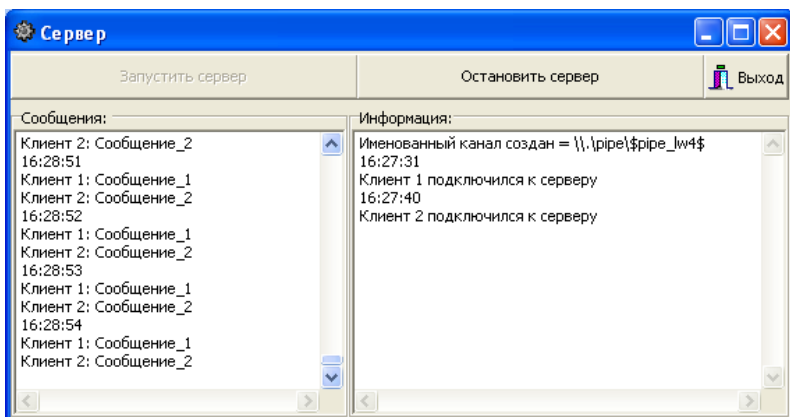


Рис. 9. Результат работы программы-сервера и программ-клиентов

ЛАБОРАТОРНАЯ РАБОТА №5. ФАЙЛЫ ДАННЫХ, ПРОЕЦИРУЕМЫЕ В ПАМЯТЬ

Цель работы

Изучить принципы работы с файлами, проецируемыми в память.

Информация

Как и виртуальная память, проецируемые файлы позволяют резервировать регион адресного пространства и передавать ему физическую память.

Операционная система позволяет проецировать на адресное пространство процесса и файл данных. Это очень удобно при манипуляциях с большими потоками данных [3].

Проецируемые файлы применяются для:

- загрузки и выполнения EXE- и DLL-файлов. Это позволяет существенно экономить как на размере страничного файла, так и на времени, необходимом для подготовки приложения к выполнению;
- доступа к файлу данных, размещенному на диске. Это позволяет обойтись без операций файлового ввода-вывода и буферизации его содержимого;
- разделения данных между несколькими процессами, выполняемыми на одной машине.

Чтобы представить всю мощь такого применения механизма проецирования файлов, рассмотрим четыре возможных метода реализации программы, меняющей порядок следования всех байтов в файле на обратный.

Метод 1: один файл, один буфер

Первый (и теоретически простейший) метод – выделение блока памяти, достаточного для размещения всего файла. Открываем файл, считываем его содержимое в блок памяти, закрываем. Располагая в памяти содержимым файла, можно поменять первый байт с последним, второй – с предпоследним и т.д. Этот процесс будет продолжаться, пока мы не поменяем местами два смежных байта, находящихся в середине файла. Закончив эту операцию, вновь открываем файл и перезаписываем его содержимое.

Метод 2: два файла, один буфер

Открываем существующий файл и создаем на диске новый – нулевой длины. Затем выделяем небольшой внутренний буфер размером, скажем, 8 Кб. Устанавливаем указатель файла в позицию 8 Кб от конца, считываем в буфер последние 8 Кб содержимого файла, меняем в нем порядок следования байтов на обратный и переписываем буфер в только

что созданный файл. Повторяем эти операции, пока не дойдем до начала исходного файла. Конечно, если длина файла не будет кратна 8 Кб, операции придется немного усложнить. Закончив обработку, закрываем оба файла и удаляем исходный файл.

Метод 3: один файл, два буфера

Программа инициализирует два отдельных буфера, допустим, по 8 Кб и считываем первые 8 Кб файла в один буфер, а последние 8 Кб – в другой. Далее содержимое обоих буферов обмениваются в обратном порядке, и первый буфер записывается в конец, а второй – в начало того же файла. На каждой итерации программа перемещает восьмиклобитные блоки из одной половины файла в другую. В данном методе следует предусмотреть обработку на случай, если длина файла не кратна 16 Кб.

Метод 4: один файл и никаких буферов

Открывается файл с указанием системе зарезервировать регион виртуального адресного пространства. Далее, первый байт файла проецируется на первый байт этого региона и производится обращение к региону так, будто он на самом деле содержит файл. Если в конце файла есть отдельный нулевой байт, можно вызвать библиотечную функцию `_strrev` и поменять порядок следования байтов на обратный.

Использование проецируемых в память файлов

Для этого нужно выполнить три операции:

1. Создать или открыть объект ядра «файл», идентифицирующий дисковый файл, который используется как проецируемый в память.
2. Создать объект ядра «проекция файла», сообщив системе размер файла и способ доступа к нему.
3. Указать системе, как спроецировать в адресное пространство процесса объект «проекция файла» - целиком или частично.

Закончив работу с проецируемым в память файлом, следует выполнить следующие операции:

1. Сообщить системе об отмене проецирования на адресное пространство процесса объекта ядра «проекция файла».
2. Закрыть этот объект.
3. Закрыть объект ядра «файл».

Этап 1: создание или открытие объекта ядра «файл»

Для создания и открытия объекта ядра «файл» используется функция `CreateFile`:

```
HANDLE CreateFile(  
    PCSTR    lpFileName,
```

```

DWORD    dwDesiredAccess,
DWORD    dwShareMode,
PSECURITY_ATTRIBUTES  lpSecurityAttributes,
DWORD    dwCreationDisposition,
DWORD    dwFlagsAndAttributes,
HANDLE    hTemplateFile);

```

Рассмотрим три первых параметра *lpFileName*, *dwDesiredAccess* и *dwShareMode*.

Параметр *lpFileName* идентифицирует имя создаваемого или открываемого файла (при необходимости вместе с путем). Второй параметр, *dwDesiredAccess*, указывает способ доступа к содержимому файла. Здесь задается одно из четырех значений (табл. 13):

Таблица 13

Значение параметра *dwDesiredAccess*

Значение	Описание
0	Содержимое файла нельзя считывать или записывать; указывается это значение, если требуется получить атрибуты файла
GENERIC_READ	Чтение файла разрешено
GENERIC_WRITE	Запись в файл разрешена
GENERIC_READ GENERIC_WRITE	Разрешено и чтение и запись

Создавая или открывая файл данных, используемый в качестве проецируемого в память, устанавливается флаг **GENERIC_READ** (только для чтения), либо комбинированный флаг **GENERIC_READ | GENERIC_WRITE** (чтение/запись).

Третий параметр, *dwShareMode*, указывает тип совместного доступа к данному файлу (табл. 14).

Таблица 14

Значение параметра *dwShareMode*

Значение	Описание
0	Другие попытки открыть файл закончатся неудачно
FILE_SHARE_READ	Попытка постороннего процесса открыть файл с флагом GENERIC_WRITE не удастся
FILE_SHARE_WRITE	Попытка постороннего процесса открыть файл с флагом GENERIC_READ не удастся
FILE_SHARE_READ FILE_SHARE_WRITE	Посторонний процесс может открыть файл без ограничений

Создав или открыв указанный файл, *CreateFile* возвращает его дескриптор, в ином случае – идентификатор `INVALID_HANDLE_VALUE`, определенный как `((HANDLE)-1)`.

Этап 2: создание объекта ядра «проекция файла»

Указав операционной системе, где находится физическая память для проекции файла: на жестком диске, в сети, на CD-ROM или в другом месте (вызов функции *CreateFile*), необходимо указать системе какой объем физической памяти нужен проекции файла. Для этого необходимо вызвать функцию *CreateFileMapping*:

```
HANDLE CreateFileMapping(  
    HANDLE    hFile,  
    LPSECURITY_ATTRIBUTES    lpFileMappingAttributes,  
    DWORD     flProtect,  
    DWORD     dwMaximumSizeHigh,  
    DWORD     dwMaximumSizeLow,  
    LPCTSTR   lpName);
```

Первый параметр, *hFile*, идентифицирует дескриптор файла, проецируемого на адресное пространство процесса. Этот дескриптор получили после вызова функции *CreateFile*. Параметр *lpFileMappingAttributes* – указатель на структуру `SECURITY_ATTRIBUTES`, которая относится к объекту ядра «проекция файла»; для установки защиты по умолчанию ему присваивается `NULL`.

Создание файла, проецируемого в память, аналогично резервированию региона адресного пространства с последующей передачей ему физической памяти. Разница лишь в том, что физическая память для проецируемого файла – сам файл на диске, и для него не нужно выделять пространство в страничном файле. При создании объекта «проекция файла» система не резервирует регион адресного пространства и не увязывает его с физической памятью из файла. Но, как только дело дойдет до отображения физической памяти на адресное пространство процесса, системе понадобится точно знать атрибут защиты, присваиваемый страницам физической памяти. Поэтому в параметре *flProtect* надо указать желательные атрибуты защиты (табл. 15).

Таблица 15

Значение параметра *flProtect*

Атрибут защиты	Описание
<code>PAGE_READONLY</code>	Отобразив объект «проекция файла» на адресное пространство, можно считывать данные из файла. При этом необходимо было передать в функцию

Атрибут защиты	Описание
	<i>CreateFile</i> флаг <code>GENERIC_READ</code> .
<code>PAGE_READWRITE</code>	Отобразив объект «проекция файла» на адресное пространство, можно считывать данные из файла и записывать их. При этом необходимо было передать в функцию <i>CreateFile</i> комбинацию флагов <code>GENERIC_READ GENERIC_WRITE</code> .
<code>PAGE_WRITECOPY</code>	Отобразив объект «проекция файла» на адресное пространство, можно считывать данные из файла и записывать их. Запись приведет к созданию закрытой копии страницы. При этом необходимо было передать в функцию <i>CreateFile</i> либо <code>GENERIC_READ</code> , либо <code>GENERIC_READ GENERIC_WRITE</code> .

Кроме рассмотренных выше атрибутов защиты страницы, существует еще четыре атрибута раздела: `SEC_NO_CACHE`, `SEC_IMAGE`, `SEC_RESERVE` и `SEC_COMMIT`.

Следующие два параметра этой функции (*dwMaximumSizeHigh* и *dwMaximumSizeLow*) самые важные. Основное назначение *CreateFileMapping* – гарантировать, что объекту «проекция файла» доступен нужный объем физической памяти. Через эти параметры сообщается системе максимальный размер файла в байтах. Так как Windows позволяет работать с файлами, размеры которых выражаются 64-разрядными числами, в параметре *dwMaximumSizeHigh* указываются старшие 32 бита, а в *dwMaximumSizeLow* – младшие 32 бита этого значения. Для файлов размером менее 4 Гб *dwMaximumSizeHigh* всегда равен 0. Наличие 64-разрядного значения подразумевает, что Windows способна обрабатывать файлы длиной до 16 экзбайтов.

Для создания объекта «проекция файла» таким, чтобы он отражал текущий размер файла, необходимо передавать в обоих параметрах нули. Так же следует поступить, если необходимо ограничиться считыванием или как-то обработать файл, не меняя его размер. Для дозаписи данных в файл выбирается его размер максимальным, чтобы оставить пространство для «маневра». Если в данный момент файл на диске имеет нулевую длину, в параметрах *dwMaximumSizeHigh* и *dwMaximumSizeLow* нельзя передавать нули. Иначе система решит, что необходима проекция файла с объемом памяти, равным 0. А это ошибка, и *CreateFileMapping* вернет `NULL`.

Последний параметр функции *CreateFileMapping* – *lpName* – строка с нулевым байтом в конце; в ней указывается имя объекта «проекция файла», которое используется для доступа к данному объекту из другого процесса. В случае если совместное использование

проецируемого в память файла не требуется, в данном параметре передают NULL.

Чтобы получить доступ к существующему объекту ядра «проекция файла» необходимо вызвать функцию *OpenFileMapping* с указанием операций, которые будут проводиться над объектом:

```
HANDLE OpenFileMapping(
    DWORD    dwDesiredAccess,
    BOOL     bInheritHandle,
    LPCTSTR  lpName);
```

Первый параметр, *dwDesiredAccess*, идентифицирует вид доступа к данным. Необходимо указать, как именно мы хотим обращаться к файловым данным, задавая одно из четырех значений (табл. 16).

Таблица 16

Значение параметра *dwDesiredAccess*

Значение	Описание
FILE_MAP_WRITE	Файловые данные можно считывать и записывать; при этом в функцию <i>CreateFileMapping</i> должен быть передан атрибут PAGE_READWRITE.
FILE_MAP_READ	Файловые данные можно только считывать; при этом в функцию <i>CreateFileMapping</i> должен быть передан любой из следующих атрибутов PAGE_READONLY, PAGE_READWRITE или PAGE_WRITECOPY.
FILE_MAP_ALL_ACCESS	То же, что и FILE_MAP_WRITE.
FILE_MAP_COPY	Файловые данные можно считывать и записывать, но запись приводит к созданию закрытой копии страницы; при этом в функцию <i>CreateFileMapping</i> должен быть передан любой из следующих атрибутов PAGE_READONLY, PAGE_READWRITE или PAGE_WRITECOPY.

Второй параметр, *bInheritHandle*, указывает, наследовал ли новый процесс дескрипторы от процесса запроса. Если TRUE, каждый наследственный открытый дескриптор в процессе запроса унаследован новым процессом.

Последний параметр функции *OpenFileMapping* – *lpName* – строка с нулевым байтом в конце; в ней указывается имя объекта «проекция файла», которое используется для доступа к объекту из данного процесса.

Функция *OpenFileMapping*, прежде чем вернуть действительный описатель, проверяет тип защиты объекта. Если есть доступ к существующему объекту ядра «проекция файла», функция *OpenFileMapping* возвращает действительный описатель. Но если отказано в доступе, функция *OpenFileMapping* возвращает NULL, а вызов *GetLastError* дает код ошибки 5 (или `ERROR_ACCESS_DENIED`).

Этап 3: проецирование файловых данных на адресное пространство процесса

Когда объект «проекция файла» создан, необходимо, чтобы система, зарезервировав регион адресного пространства под данные файла, передала их как физическую память, отображенную на регион. Это делает функция *MapViewOfFile*:

```
LPVOID MapViewOfFile(  
    HANDLE    hFileMappingObject,  
    DWORD     dwDesiredAccess,  
    DWORD     dwFileOffsetHigh,  
    DWORD     dwFileOffsetLow,  
    DWORD     dwNumberOfBytesToMap);
```

Параметр *hFileMappingObject* идентифицирует описатель объекта «проекция файла», возвращаемый предшествующим вызовом либо функцией *CreateFileMapping*, либо функцией *OpenFileMapping*. Параметр *dwDesiredAccess* идентифицирует вид доступа к данным (табл. 16).

Остальные три параметра относятся к резервированию региона адресного пространства и к отображению на него физической памяти. При этом необязательно проецировать на адресное пространство весь файл сразу. Можно спроецировать лишь малую его часть, которая в таком случае называется представлением (*view*).

Проецируя на адресное пространство процесса представление файла, необходимо сделать две вещи. Во-первых, сообщить системе, какой байт файла данных считать в представлении первым. Для этого предназначены параметры *dwFileOffsetHigh* и *dwFileOffsetLow*. Во-вторых, потребуется указать размер представления, т.е. сколько байтов файла данных должно быть спроецировано на адресное пространство. Размер указывается в параметре *dwNumberOfBytesToMap*. Если этот параметр равен 0, система попытается спроецировать представление, начиная с указанного смещения и до конца файла.

Если при вызове *MapViewOfFile* указан флаг `FILE_MAP_COPY`, система передаст физическую память из страничного файла. Размер передаваемого пространства определяется параметром *dwNumberOfBytesToMap*. Пока данные считываются из представления файла, страницы, переданные из страничного файла, не используются. Но

стоит какому-нибудь потоку в процессе совершить попытку записи по адресу, попадающему в границы представления файла, как система тут же берет из страничного файла одну из перечисленных страниц, копирует на нее исходные данные и проецирует ее на адресное пространство процесса. С этого момента потоки процесса начинают обращаться к локальной копии данных и теряют доступ к исходным данным. Создав копию исходной страницы, система меняет ее атрибут защиты с `PAGE_WRITECOPY` на `PAGE_READWRITE`.

Этап 4: отключение файла данных от адресного пространства процесса

Когда необходимость в данных файла (спроецированного на регион адресного пространства) отпадает, требуется освободить регион вызовом функции *UnmapViewOfFile*:

```
BOOL UnmapViewOfFile(LPCVOID lpBaseAddress);
```

Параметр, *lpBaseAddress*, указывает базовый адрес возвращаемого системе региона. Он должен совпадать со значением, полученным после вызова функции *MapViewOfFile*. Если не вызвать функцию *MapViewOfFile* регион не освободится до завершения процесса. Повторный вызов *MapViewOfFile* приводит к резервированию нового региона в пределах адресного пространства процесса, но ранее выделенные регионы *не освобождаются*.

У функции *UnmapViewOfFile* есть одна особенность. Если первоначально представление было спроецировано с флагом `FILE_MAP_COPY`, любые изменения, внесенные в файловые данные, на самом деле производятся над копией этих данных, хранящихся в страничном файле. Вызванной в этом случае функции *UnmapViewOfFile* нечего обновлять в дисковом файле, и она просто иницирует возврат системе страниц физической памяти, выделенных из страничного файла. Все изменения в данных на этих страницах теряются. Поэтому о сохранении измененных данных придется заботиться самостоятельно.

Этапы 5 и 6: закрытие объектов «проекция файлов» и «файл»

Закончив работу с любым открытым объектом ядра, необходимо его закрыть, иначе в процессе начнется утечка ресурсов. Для закрытия объектов «проекция файлов» и «файл» требуется дважды вызвать функцию *CloseHandle*.

При операциях с проецируемыми файлами обычно открывают файл, создают объект «проекция файла» и с его помощью проецируют представление файловых данных на адресное пространство процесса. Поскольку система увеличивает внутренние счетчики объектов «файл» и

«проекция файла», их можно закрыть в начале кода, тем самым, исключив возможность утечки ресурсов.

Если требуется создавать из одного файла несколько объектов «проекция файла» или проецировать несколько представлений этого объекта, применить функцию *CloseHandle* в начале кода не удастся – описатели еще понадобятся для дополнительных вызовов *CreateFileMapping* и *MapViewOfFile*.

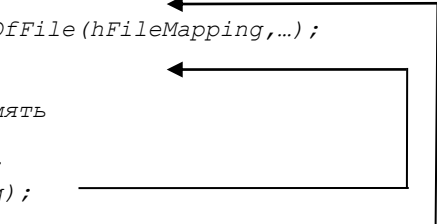
Рассмотрим это подробнее на фрагменте псевдокода:

```
HANDLE hFile = CreateFile(...);
HANDLE hFileMapping = CreateFileMapping(hFile,...);

LPVOID pvFile = MapViewOfFile(hFileMapping,...);

// работаем с файлом,
// спроецированным в память

UnmapViewOfFile(pvFile);
CloseHandle(hFileMapping);
CloseHandle(hFile);
```

A diagram with three horizontal arrows pointing left. The top arrow starts from the right margin and points to the `CloseHandle(hFile);` line. The middle arrow starts from the right margin and points to the `CloseHandle(hFileMapping);` line. The bottom arrow starts from the right margin and points to the `UnmapViewOfFile(pvFile);` line.

Пример. Создаем файл “*File-mapping.txt*”, пишем в него строчку “*The named or unnamed file-mapping object.*”, закрываем, открываем для чтения, проецируем в память и копируем строку из спроецированного региона памяти.

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

int main()
{
    PDWORD lpNumberOfBytesWritten=new DWORD;
    char str[]="The named or unnamed file-mapping object.";
    HANDLE MyFile=CreateFile("File-mapping.txt",
                             GENERIC_WRITE,
                             FILE_SHARE_WRITE, NULL,
                             CREATE_ALWAYS,
                             FILE_ATTRIBUTE_NORMAL,0);
    WriteFile(MyFile,str,strlen(str),
              lpNumberOfBytesWritten,NULL);
    SetEndOfFile(MyFile);
    CloseHandle(MyFile);
    MyFile=CreateFile("File-mapping.txt", GENERIC_READ,
                     FILE_SHARE_READ,NULL,OPEN_EXISTING,
                     FILE_ATTRIBUTE_NORMAL,0);
    HANDLE MappedFile=CreateFileMapping(MyFile,NULL,
                                         PAGE_READONLY,0,0,
                                         NULL);
```

```

LPVOID Map=MapViewOfFile(MappedFile,FILE_MAP_READ,
                        0,0,0);
char * strin =(char *) Map;
cout << strin << endl;
UnmapViewOfFile(Map);
CloseHandle(MappedFile);
CloseHandle(MyFile);
return 0;
}

```

Задание. Требуется создать программу «Писатель» и программу «Читатель». Запустить программу «Писатель» (обеспечить запуск только одного экземпляра программы) и несколько экземпляров программы «Читатель». Программа «Писатель» постоянно обновляет содержимое некоторого файла (физический файл на диске) и проецируют его на собственное адресное пространство (рис. 10). Программы «Читатель» получают доступ к объекту ядра «проекция файла», созданному программой «Писатель». При обновлении файла происходит автоматическое обновление содержимого файла в окнах программ «Читатель» (рис. 11).

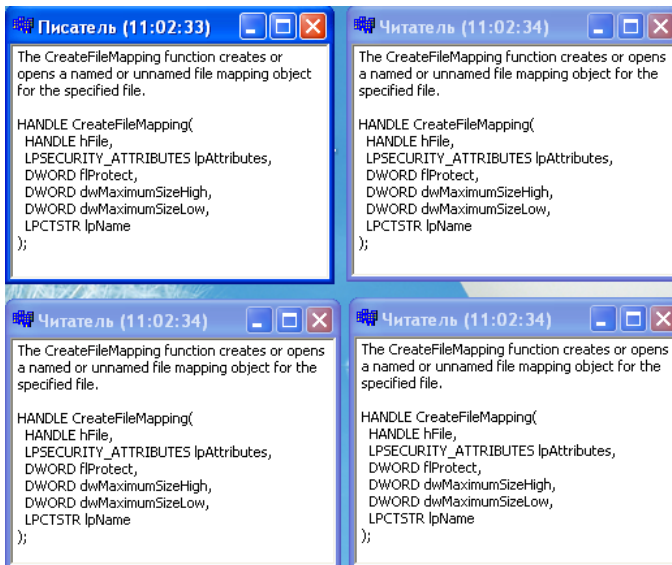


Рис. 10. Результаты работы программ

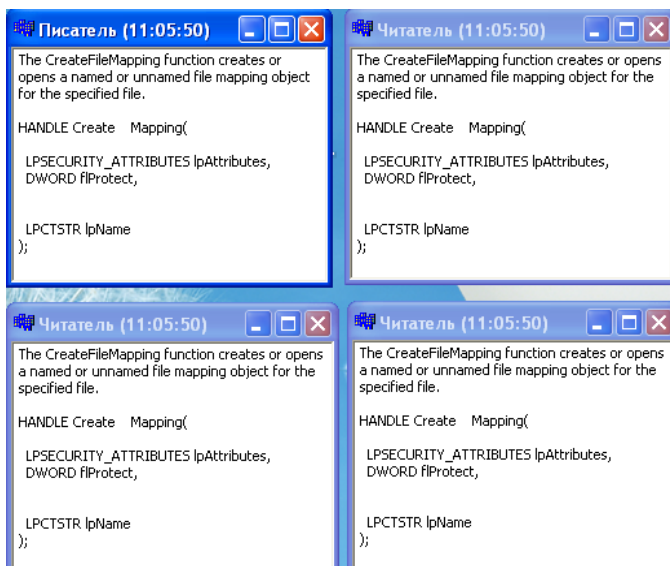


Рис. 11. Результаты работы программ
(при изменении информации писателем)

КУРСОВАЯ РАБОТА.

ОРГАНИЗАЦИЯ ВЗАИМОДЕЙСТВИЯ И СИНХРОНИЗАЦИИ ПАРАЛЛЕЛЬНЫХ ПРОЦЕССОВ И ПОТОКОВ

Цель работы

Получение практических навыков по использованию Win32 API для организации взаимодействия и синхронизации параллельных процессов и потоков при выполнении курсовой работы.

Информация

При многопоточной обработке могут возникнуть следующие ситуации [6]:

- рассинхронизация (*race conditions*);
- тупиковая ситуация (*deadlock* – «смертельное объятие»).

Рассмотрим ситуацию, когда успех одной операции зависит от успеха другой, но обе они не синхронизированы друг с другом.

Пусть *Поток1* процесса *A* подготавливает принтер, а *Поток2* ставит задание на печать в очередь. Если потоки не синхронизированы и *Поток1* не успеет выполнить свою работу до того, как начнется печать, то получим сбой.

Из истории вопроса. Голландский профессор математики Э. Дейкстра в начале 70-х годов XX века, рассматривая ситуации с многопользовательским доступом, ввел следующие понятия:

- критический интервал (*critical section*);
- семафоры (*semaphore*);
- взаимные исключения (*mutex*).

1. Механизмы синхронизации операционной системы Windows

В Win32 существуют средства синхронизации двух типов:

- реализованные на уровне пользователя (критические секции – *Critical section*);
- реализованные на уровне ядра (мьютексы – *Mutex*, события – *Event*, семафоры – *Semaphore*).

Общие черты механизмов синхронизации:

- используют примитивы ядра при выполнении, что сказывается на производительности;
- могут быть именованными и неименованными;
- работают на уровне системы, то есть могут служить механизмом межпроцессного взаимодействия;
- используют для ожидания и захвата примитива единую функцию: *WaitForSingleObject/WaitForMultipleObjects*.

Существуют несколько стратегий, которые могут применяться, чтобы разрешать проблемы, связанные с взаимодействием потоков.

Наиболее распространенным способом является синхронизация потоков, суть которой состоит в том, чтобы вынудить один поток ждать, пока другой не закончит какую-то определенную заранее операцию. Для этой цели существуют специальные синхронизирующие объекты ядра операционной системы *Windows*.

Они исключают возможность одновременного доступа к тем данным, которые с ними связаны. Их реализация зависит от конкретной ситуации и предпочтений программиста.

Общие положения использования объектов ядра системы:

- однажды созданный объект ядра можно открыть в любом приложении, если оно имеет соответствующие права доступа к нему;
- каждый объект ядра имеет счетчик числа своих пользователей. Как только он станет равным нулю, система уничтожит объект ядра;
- обращаться к объекту ядра надо через описатель (*handle*), который система дает при создании объекта;
- каждый объект может находиться в одном из двух состояний: свободном (*signaled*) или занятом (*nonsignaled*).

Работа с объектом Критическая секция (Critical section)

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. Естественно, система может в любой момент вытеснить Ваш поток и подключить к процессору другой, но ни один из потоков, которым нужен занятый Вами ресурс, не получит процессорное время до тех пор, пока Ваш поток не выйдет за границы критической секции. Критические секции являются простыми объектами ядра *Windows*, которые не снижают общей эффективности приложения.

Для работы с критическими секциями есть ряд функций API и тип данных `CRITICAL_SECTION`. Алгоритм использования следующий:

1. Объявить глобальную структуру `CRITICAL_SECTION cs`.
2. Инициализировать (обычно это делается один раз, перед тем как начнется работа с разделяемым ресурсом) глобальную структуру вызовом функции *InitializeCriticalSection (&cs)*.
3. Поместить охраняемую часть программы внутрь блока, который начинается вызовом функции *EnterCriticalSection* и заканчивается вызовом *LeaveCriticalSection*:

```
EnterCriticalSection (&cs);  
{  
  // охраняемый блок кодов  
}  
LeaveCriticalSection (&cs);
```

Функция *EnterCriticalSection*, анализируя поле структуры *cs*, которое является счетчиком ссылок, выясняет, вызвана ли она в первый раз.

Если да, то функция увеличивает значение счетчика и разрешает выполнение потока дальше. При этом выполняется блок, модифицирующий критические данные. Допустим, в это время истекает квант времени, отпущенный данному потоку, или он вытесняется более приоритетным потоком, использующим те же данные.

Новый поток выполняется, пока не встречает функцию *EnterCriticalSection*, которая помнит, что объект *cs* уже занят. Новый поток приостанавливается (засыпает), а остаток процессорного времени передается другому потоку.

Функция *LeaveCriticalSection* уменьшает счетчик ссылок на объект *cs*.

Как только поток покидает критическую секцию, счетчик ссылок обнуляется и система будит ожидающий поток, снимая защиту секции кодов.

Критические секции применяются для синхронизации потоков лишь в пределах одного процесса. Они управляют доступом к данным так, что в каждый конкретный момент времени только один поток может их изменять.

4. Когда надобность в синхронизации потоков отпадает, следует вызвать функцию, освобождающую все ресурсы, включенные в критическую секцию: *DeleteCriticalSection (&cs)*.

ПРИМЕЧАНИЕ Функция *TryEnterCriticalSection()* позволяет проверить критическую секцию на занятость.

Таким образом, поток, который желает обезопасить определенные данные от *race conditions*, вызывает функцию *EnterCriticalSection / TryEnterCriticalSection*:

- если критическая секция свободна, поток занимает ее;
- если же нет, поток блокируется до тех пор, пока секция не будет освобождена другим потоком с помощью вызова функции *LeaveCriticalSection*.

Данные функции – атомарные, то есть целостность данных нарушена не будет.

Пример. Разграничение доступа к общему ресурсу с использованием критической секции

```
int g_nNums[100]; // разделяемый ресурс
CRITICAL_SECTION g_cs; // защита ресурса

DWORD WINAPI ThreadFunc(PVOID pvParam) {
    EnterCriticalSection(&g_cs);
    for (int x = 0; x < 100; x++)
    {
        g_nNums[x] = 0;
    }
    LeaveCriticalSection(&g_cs);
    return(0);
}
```

Работа с объектом Семафор (Semaphore)

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.

Для семафоров определены следующие правила:

- когда счетчик текущего числа ресурсов становится больше 0, семафор переходит в свободное состояние;
- если этот счетчик равен 0, семафор занят;
- система не допускает присвоения отрицательных значений счетчику текущего числа ресурсов;
- счетчик текущего числа ресурсов не может быть больше максимального числа ресурсов.

Для работы с объектом *Semaphore* существует ряд функций:

- функция *CreateSemaphore()* создает семафор с заданным начальным значением счетчика и максимальным значением, которое ограничивает доступ;

```
HANDLE CreateSemaphore(
    PSECURITY_ATTRIBUTES psa,
    LONG lInitialCount,
    LONG lMaximumCount,
    PCTSTR pszName);
```

Параметр *psa* является указателем на структуру *SECURITY_ATTRIBUTES*, в большинстве случаев параметр принимает значение *NULL*, при этом создается объект с защитой по умолчанию. Параметр *lMaximumCount* сообщает системе

максимальное число ресурсов, обрабатываемое приложением. Поскольку 32-битное значение со знаком, предельное число ресурсов может достигать 2147483647. Параметр *InitialCount* указывает, сколько из этих ресурсов доступно изначально (на данный момент). Параметр *pszName* определяет имя объекта ядра Windows (если параметр принимает значение NULL, то создается безымянный объект ядра).

- функция *OpenSemaphore()* осуществляет доступ к семафору;

```
HANDLE OpenSemaphore(  
    DWORD fdwAccess,  
    BOOL bInheritHandle,  
    PCTSTR pszName);
```

- функция *ReleaseSemaphore()* увеличивает значение счетчика. Счетчик может меняться от 0 до максимального значения;

```
BOOL ReleaseSemaphore(  
    HANDLE hSem,  
    LONG lReleaseCount,  
    PLONG plPreviousCount);
```

- после завершения работы надо вызвать *CloseHandle()*.

```
BOOL CloseHandle(HANDLE hSem);
```

Работа с объектом Мьютекс (Mutex)

Объекты ядра «мьютексы» (mutual exclusion, mutex) гарантируют потокам взаимоисключающий доступ к единственному ресурсу. Они содержат счетчик числа пользователей, счетчик рекурсии и переменную, в которой запоминается идентификатор потока. Мьютексы ведут себя точно так же, как и критические секции. Однако, если последние являются объектами пользовательского режима, то мьютексы – объектами ядра. Кроме того, единственный объект-мьютекс позволяет синхронизировать доступ к ресурсу нескольких потоков из разных процессов; при этом можно задать максимальное время ожидания доступа к ресурсу.

Для работы с этим объектом есть ряд функций:

- функция создания объекта *Mutex* – *CreateMutex()*;

```
HANDLE CreateMutex(  
    PSECURITY_ATTRIBUTES psa,  
    BOOL fInitialOwner,  
    PCTSTR pszName);
```

Параметр *fInitialOwner* определяет начальное состояние мьютекса. Если в нем передается FALSE, объект-мьютекс не

принадлежит ни одному из потоков и поэтому находится в свободном состоянии. При этом его идентификатор потока и счетчик рекурсии равны 0. Если же в нем передается TRUE, идентификатор потока, принадлежащий мьютексу, приравнивается идентификатору вызывающего потока, а счетчик рекурсии получает значение 1. Поскольку теперь идентификатор потока отличен от 0, мьютекс изначально находится в занятом состоянии.

- для доступа – *OpenMutex()*;

```
HANDLE OpenMutex(  
    DWORD fdwAccess,  
    BOOL bInheritHandle,  
    PCTSTR pszName);
```

- для освобождения ресурса – *ReleaseMutex()*;

```
BOOL ReleaseMutex(HANDLE hMutex);
```

Функция уменьшает счетчик рекурсии в объекте-мьютексе на 1. Если данный объект передавался во владение потоку неоднократно, поток обязан вызвать *ReleaseMutex* столько раз, сколько необходимо для обнуления счетчика рекурсии. Как только счетчик станет равен 0, переменная, хранящая идентификатор потока, тоже обнулится, и объект-мьютекс освободится.

- для доступа к объекту *Mutex* используется ожидающая функция *WaitForSingleObject()*.

Каждая программа создает объект *Mutex* по имени, то есть *Mutex* – это именованный объект.

ПРИМЕЧАНИЕ Если такой объект синхронизации уже создала другая программа, то по вызову *CreateMutex()* можно получить указатель на объект, который уже создала первая программа, то есть у обеих программ будет один и тот же объект, что и позволяет производить синхронизацию.

С любым объектом ядра сопоставляется счетчик, фиксирующий, сколько раз данный объект передавался во владение потокам.

Объект *Mutex* отличается от других синхронизирующих объектов ядра тем, что занявшему его потоку передаются права на владение им.

Прочие синхронизирующие объекты могут быть либо свободны, либо заняты и только, а *Mutex* способны еще и запоминать, какому потоку они принадлежат.

Отказ от *Mutex* происходит, когда ожидавший его поток захватывает этот объект, переводя его в занятое состояние, а потом завершается. В таком случае получается, что *Mutex* занят и никогда не освободится, поскольку другой поток не сможет этого сделать. Система

не допускает подобных ситуаций и, заметив, что произошло, автоматически переводит *Mutex* в свободное состояние.

Работа с объектом Событие (Event)

События – самая примитивная 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.

Для работы с объектом *Event* есть ряд функций:

- функция *CreateEvent()* используется для создания события;

```
HANDLE CreateEvent (
    PSECURITY_ATTRIBUTES psa,
    BOOL fManualReset,
    BOOL fInitialState,
    PCTSTR pszName);
```

Параметр *fManualReset* (булева переменная) сообщает системе, хотите Вы создать событие со сбросом вручную (TRUE) или с автосбросом (FALSE). Параметр *fInitialState* определяет начальное состояние события – свободное (TRUE) или занятое (FALSE).

- функция *OpenEvent()* – для доступа;

```
HANDLE OpenEvent (
    DWORD fdwAccess,
    BOOL fInherit,
    PCTSTR pszName);
```

- функция *SetEvent()* – для установки события в свободное состояние;

```
BOOL SetEvent (HANDLE hEvent);
```

- функция *ResetEvent()* используется для сброса события.

```
BOOL ResetEvent (HANDLE hEvent);
```

- функция *PulseEvent()* освобождает событие и тут же переводит его обратно в занятое состояние; ее вызов равнозначен последовательному вызову *SetEvent()* и *ResetEvent()*.

```
BOOL PulseEvent (HANDLE hEvent);
```

Дескриптор события после окончания работы нужно закрыть. Класс *CEvent* представляет функциональность синхронизирующего объект ядра (события). Он позволяет одному потоку уведомить (*notify*) другой поток о том, что произошло событие, которое тот поток, возможно, ждал.

Существуют два типа объектов: ручной (*manual*) и автоматический (*automatic*):

- ручной объект начинает сигнализировать, когда будет вызван метод *SetEvent*. Вызов *ResetEvent* переводит его в противоположное состояние;
- автоматический объект класса *CEvent* не нуждается в сбросе. Он сам переходит в состояние *nonsignaled*, и охраняемый код при этом недоступен, когда хотя бы один поток был уведомлен о наступлении события.

Пример. Использование объектов ядра «событие» для синхронизации потоков.

```
// глобальный описатель события со сбросом вручную (в занятом состоянии)
HANDLE g_hEvent;
int WINAPI WinMain(...) {
// создаем объект «событие со сбросом вручную» (в занятом состоянии)
g_hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
// порождаем три новых потока
HANDLE hThread[3];
DWORD dwThreadId;
hThread[0] = _beginthreadex(NULL, 0, WordCount, NULL, 0, &dwThreadId);
hThread[1] = _beginthreadex(NULL, 0, SpellCheck, NULL, 0, &dwThreadId);
hThread[2] = _beginthreadex(NULL, 0, GrammarCheck, NULL, 0, &dwThreadId);
OpenFileAndReadContentsIntoMemory(...);
// разрешаем всем трем потокам обращаться к памяти
SetEvent(g_hEvent);

}

DWORD WINAPI WordCount(PVOID pvParam) {
// ждем, когда в память будут загружены данные из файла
WaitForSingleObject(g_hEvent, INFINITE);
// обращаемся к блоку памяти

return(0);
}

DWORD WINAPI SpellCheck(PVOID pvParam) {
// ждем, когда в память будут загружены данные из файла
WaitForSingleObject(g_hEvent, INFINITE);
// обращаемся к блоку памяти

return(0);
}

DWORD WINAPI GrammarCheck(PVOID pvParam) {
```

```
// ждем, когда в память будут загружены данные из файла
WaitForSingleObject(g_hEvent, INFINITE);
// обращаемся к блоку памяти

return(0);
}
```

При запуске этот процесс создает занятое событие со сбросом вручную и записывает его описатель в глобальную переменную. Это упрощает другим потокам процесса доступ к тому же объекту-событию. Затем порождается три потока. Они ждут, когда в память будут загружены данные (текст) из некоего файла, и потом обращаются к этим данным: например, один поток подсчитывает количество слов, другой проверяет орфографические ошибки, третий – грамматические. Все три функции потоков начинают работать одинаково: каждый поток вызывает *WaitForSingleObject()*, которая приостанавливает его до тех пор, пока первичный поток не считает в память содержимое файла.

Функции ряда Wait

Функции ряда Wait блокируют выполнение потока до наступления какого-то события или тайм-аута. Для того чтобы пользоваться этими функциями, должен быть объект синхронизации, который проверяет эти функции. У этих объектов два состояния: «установлен» и «сброшен».

Алгоритм использования ожидающих функций:

- вызов функций (им передается указатель на объект синхронизации);
- объект проверяется;
- если объект не установлен, то функция будет ждать, пока не истечет тайм-аут, все это время поток будет блокирован.

Сценарий синхронизации потоков с использованием ожидающих функций:

- прежде чем заснуть, поток сообщает системе то особое событие, которое должно разбудить его;
- при засыпании потока операционная система перестает выделять ему кванты процессорного времени, приостанавливая его выполнение;
- как только указанное потоком событие произойдет, система возобновит выдачу ему квантов процессорного времени и поток вновь может развиваться.

Win32 API поддерживает целый ряд функций, которые начинаются с *Wait*:

- *WaitForMultipleObjects*;
- *WaitForMultipleObjectsEx*;
- *WaitForSingleObject*;

- *WaitForSingleObjectEx*;
- *MsgWaitForMultipleObjects*;
- *MsgWaitForMultipleObjectsEx*.

Также существует функция *WaitCommEvent()*, предназначенная для работы с данными в последовательных портах.

Функции, у которых в имени есть *Single*, предназначены для установки одного синхронизирующего объекта.

Функции, у которых в имени есть *Multiple*, используются для установки ожидания сразу нескольким объектам.

Функции с префиксами *Msg* предназначены для ожидания события определенного типа, например, ввода с клавиатуры.

Функции с окончанием *Ex* расширены опциями по управлению асинхронным вводом-выводом.

ПРИМЕЧАНИЕ При необходимости захвата нескольких ресурсов проблему решает использование *WaitForMultipleObject()*, эта функция пока не захватит все объекты, менять состояние одного из них не будет. *WaitForSingleObject()* в этом случае использовать нельзя, так как это приведет к *deadlock* (тупиковая ситуация).

Использование функций ряда Wait для синхронизации потоков

Потоки «усыпляют» себя до освобождения какого-либо синхронизирующего объекта с помощью следующих функций ряда *Wait*:

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

DWORD WaitForMultipleObjects (
    DWORD nCount,
    CONST HANDLE* lpHandles,
    BOOL bWaitAll,
    DWORD dwTimeout).
```

Функция *WaitForSingleObject* приостанавливает поток до тех пор, пока:

- заданный параметром *hObject* синхронизирующий объект не освободится;
- не истечет интервал времени, задаваемый параметром *dwTimeout*. Если указанный объект в течение заданного интервала не перейдет в свободное состояние, то система вновь активизирует поток, и он продолжит свое выполнение.

В качестве параметра *dwTimeout* могут выступать два особых значения:

- 0 – функция только проверяет состояние объекта (занят или свободен) и сразу же возвращается;

- INFINITE – время ожидания бесконечно; если объект так и не освободится, поток останется в неактивном состоянии и никогда не получит процессорного времени.

Функция *WaitForSingleObject* в соответствии с причинами, по которым поток продолжает выполнение, может возвращать одно из следующих значений:

- WAIT_TIMEOUT – объект не перешел в свободное состояние, но интервал времени истек;
- WAIT_ABANDONED – ожидаемый объект является *Mutex*, который не был освобожден владеющим им потоком перед окончанием этого потока. Объект *Mutex* автоматически переводится системой в состояние свободен. Такая ситуация называется «отказ от *Mutex*»;
- WAIT_OBJECT_0 – объект перешел в свободное состояние;
- WAIT_FAILED – произошла ошибка, причину которой можно узнать, вызвав *GetLastError*.

Пример. Работа функции *WaitForSingleObject* со значением таймаута, отличным от INFINITE:

```
DWORD dw = WaitForSingleObject(hProcess, 5000);
switch (dw) {
case WAIT_OBJECT_0:
    // процесс завершается
    break;
case WAIT_TIMEOUT:
    // процесс не завершился в течение 5000 мс
    break;
case WAIT_FAILED:
    // неправильный вызов функции (неверный описатель?)
    break;
}
```

Функция *WaitForMultipleObjects* задерживает поток и, в зависимости от значения флага *bWaitAll*, ждет одного из следующих событий:

- освобождение хотя бы одного синхронизирующего объекта из заданного списка;
- освобождение всех указанных объектов;
- истечение заданного интервала времени.

Пример. Работа функции *WaitForMultipleObjects* с тремя процессами:

```
HANDLE h[3];
h[0] = hProcess1;
h[1] = hProcess2;
h[2] = hProcess3;
DWORD dw = WaitForMultipleObjects(3, h, FALSE, 5000);
```

```

switch (dw) {
case WAIT_FAILED:
// неправильный вызов функции (неверный описатель?)
break;
case WAIT_TIMEOUT:
// ни один из объектов не освободился в течение 5000 мс
break;
case WAIT_OBJECT_0 + 0:
// завершился процесс, идентифицируемый h[0], т.е.
описателем (hProcess1)
break;
case WAIT_OBJECT_0 + 1:
// завершился процесс, идентифицируемый h[1], т.е.
описателем (hProcess2)
break;
case WAIT_OBJECT_0 + 2:
// завершился процесс, идентифицируемый h[2], т.е.
описателем (hProcess3)
break;
}

```

2. Приоритеты в Windows

Приоритеты процессов

Часть ОС, называемая системным планировщиком (*system scheduler*), управляет переключением заданий, определяя, какому из конкурирующих потоков следует выделить следующий квант времени процессора.

Решение принимается с учетом приоритетов конкурирующих потоков.

Множество приоритетов, определенных в операционной системе для потоков, занимает диапазон от 0 (низший приоритет) до 31 (высший приоритет).

Нулевой уровень приоритета система присваивает особому потоку обнуления свободных страниц. Он работает при отсутствии других потоков, требующих внимания со стороны операционной системы. Ни один поток, кроме него, не может иметь нулевой уровень.

Приоритет каждого потока определяется в два этапа, исходя из:

- класса приоритета процесса, в контексте которого выполняется поток;
- уровня приоритета потока внутри класса приоритета потока.

Комбинация этих параметров определяет базовый приоритет (*base priority*) потока. Существуют шесть классов приоритетов для процессов:

- **IDLE_PRIORITY_CLASS;**
- **BELOW_NORMAL_PRIORITY_CLASS;**

- `NORMAL_PRIORITY_CLASS`;
- `ABOVE_NORMAL_PRIORITY_CLASS`;
- `HIGH_PRIORITY_CLASS`;
- `REALTIME_PRIORITY_CLASS`.

Работа с приоритетами процесса:

- по умолчанию процесс получает класс приоритета `NORMAL_PRIORITY_CLASS`;
- программист может задать класс приоритета создаваемому им процессу, указав его в качестве одного из параметров функции *CreateProcess*;
- кроме того, существует возможность динамически, во время выполнения потока, изменять класс приоритета процесса с помощью API-функции *SetPriorityClass*;
- выяснить класс приоритета какого-либо процесса можно с помощью API-функции *GetPriorityClass*.

Процессы, осуществляющие мониторинг системы, а также хранители экрана (*screen savers*) должны иметь низший класс (`IDLE...`), чтобы не мешать другим полезным потокам.

Процессы самого высокого класса (`REALTIME...`) способны прервать даже те системные потоки, которые обрабатывают сообщения мыши, ввод с клавиатуры и фоновую работу с диском. Этот класс должны иметь только те процессы, которые выполняют короткие обменные операции с аппаратурой.

Для написания драйвера какого-либо устройства, используя API-функции из набора *Device Driver Kit* (DDK), следует использовать для процесса класс `REALTIME...`

С осторожностью следует использовать класс `HIGH_PRIORITY_CLASS`, так как если поток процесса этого класса подолгу занимает процессор, то другие потоки не имеют шанса получить свой квант времени. Если несколько потоков имеют высокий приоритет, то эффективность работы каждого из них, а также всей системы резко падает. Этот класс зарезервирован для реакций на события, критичные ко времени их обработки.

Пример. С помощью функции *SetPriorityClass* процессу временно присваивают значение `HIGH...`, затем, после завершения *CriticalSection* кода, его снижают.

Пример. Создается процесс с высоким классом приоритета и тотчас же блокируется – погружается в сон с помощью функции *Sleep*. При возникновении критической ситуации поток или потоки этого процесса пробуждаются только на то время, которое необходимо для обработки события.

Приоритеты потоков

Рассмотрим уровни приоритета, которые могут быть присвоены потокам процесса. Внутри каждого процесса, которому присвоен какой-либо класс приоритета, могут существовать потоки, где уровень приоритета принимает одно из семи возможных значений:

- `THREAD_PRIORITY_IDLE`;
- `THREAD_PRIORITY_LOWEST`;
- `THREAD_PRIORITY_BELOW_NORMAL`;
- `THREAD_PRIORITY_NORMAL`;
- `THREAD_PRIORITY_ABOVE_NORMAL`;
- `THREAD_PRIORITY_HIGHEST`;
- `THREAD_PRIORITY_TIME_CRITICAL`.

Работа с приоритетами потока следующая:

- все потоки сначала создаются с уровнем `THREAD_PRIORITY_NORMAL`;
- программист может изменить этот начальный уровень, вызвав функцию *SetThreadPriority*;
- для определения текущего уровня приоритета потока существует функция *GetThreadPriority*, которая возвращает один из семи рассмотренных уровней.

Типичной стратегией является повышение уровня до ...`ABOVE_NORMAL` или ...`HIGHEST` для потоков, которые должны быстро реагировать на действия пользователя по вводу информации.

Потоки, которые интенсивно используют процессор для вычислений, часто относят к фоновым. Им дают уровень приоритета ...`BELOW_NORMAL` или ...`LOWEST`, чтобы при необходимости они могли быть вытеснены.

Иногда возникает ситуация, когда поток с более высоким приоритетом должен ждать поток с низким приоритетом, пока тот не закончит какую-либо операцию. В этом случае не следует программировать ожидание завершения операции в виде цикла, так как львиная доля времени процессора уйдет на выполнение команд этого цикла. Возможно даже заикливание – ситуация типа *deadlock*, так как поток с более низким приоритетом не имеет шанса получить управление и завершить операцию.

Обычной практикой в таких случаях является использование:

- одной из функций ожидания (*wait functions*);
- вызов функции *Sleep (sleepEx)*;
- вызов функции *SwitchToThread*;
- использование объекта типа *Critical section* (Критическая секция).

Базовый приоритет потока является комбинацией класса приоритета процесса и уровня приоритета потока.

Ознакомьтесь с таблицей приоритетов в справке (*Help*), в разделе *Platform SDK-Scheduling Priorities* (Платформа, *SDK* Планирование приоритетов).

Пример. Считая, что класс приоритета процесса не изменяется и остается равным `HIGH_PRIORITY_CLASS`, сведем все семь возможных вариантов в табл. 17.

Таблица 17

Приоритеты потоков

Уровень приоритета потока	Базовый уровень
<code>THREAD_PRIORITY_IDLE</code>	1
<code>THREAD_PRIORITY_LOWEST</code>	11
<code>THREAD_PRIORITY_BELOW_NORMAL</code>	12
<code>THREAD_PRIORITY_NORMAL</code>	13
<code>THREAD_PRIORITY_ABOVE_NORMAL</code>	14
<code>THREAD_PRIORITY_HIGHEST</code>	15
<code>THREAD_PRIORITY_TIME_CRITICAL</code>	15

Переключение потоков

Планировщик операционной системы поддерживает для каждого из базовых уровней приоритета функционирование очереди выполняемых или готовых к выполнению потоков (*ready threads queue*). Когда процессор становится доступным, то планировщик производит переключение контекстов.

Алгоритм переключения следующий:

- сохранение контекста потока, завершающего выполнение;
- перемещение этого потока в конец своей очереди;
- поиск очереди с высшим приоритетом, которая содержит потоки, готовые к выполнению;
- выбор первого потока из этой очереди, загрузка его контекста и запуск на выполнение.

ПРИМЕЧАНИЕ Если в системе за каждым процессором закреплен хотя бы один поток с приоритетом 31, то остальные потоки с более низким приоритетом не смогут получить доступ к процессору и поэтому не будут выполняться. Такая ситуация называется *starvation*.

Различают потоки, неготовые к выполнению:

- потоки, которые при создании имели флаг `CREATE_SUSPENDED`;
- потоки, выполнение которых было прервано вызовом функции *SuspendThread* или *SwitchToThread*;
- потоки, которые ожидают ввода или синхронизирующего события.

Блокированные таким образом потоки, или приостановленные (*suspended*) потоки, не получают кванта времени независимо от величины их приоритета.

Типичные причины переключения контекстов:

- истек квант времени;
- в очереди с более высоким приоритетом появился поток, готовый к выполнению;
- текущий поток вынужден ждать.

В последнем случае система не ждет завершения кванта времени и отбирает управление, как только поток впадает в ожидание. Каждый поток обладает динамическим приоритетом, кроме рассмотренного базового уровня. Под этим понятием скрываются временные колебания уровня приоритета, которые вызваны планировщиком. Он намеренно вызывает такие колебания для того, чтобы убедиться в управляемости и реактивности потока, а также, чтобы дать шанс выполниться потокам с низким приоритетом (система никогда не подстегивает потоки, приоритет которых и так высок (от 16 до 31)).

Когда пользователь работает с каким-то процессом, то он считается активным (*foreground*), а остальные процессы – фоновыми (*background*). При ускорении потока (*priority boost*) система действует следующим образом: когда процесс с нормальным классом приоритета «выходит на передний план» (*is brought to the foreground*), он получает ускорение.

ПРИМЕЧАНИЕ Термин «*foreground*» обозначает качество процесса, которое характеризует его с точки зрения связи с активным окном *Windows*. *Foreground window* – это окно, которое в данный момент находится в фокусе и, следовательно, расположено поверх остальных. Это состояние может быть получено как программным способом (вызов функции *SetFocus*), так и аппаратно (пользователь щелкнул окно).

Планировщик изменяет класс процесса, связанного с этим окном, так, чтобы он был больше или равен классу любого процесса, связанного с *background*-окном. Класс приоритета вновь восстанавливается при потере процессом статуса *foreground*. Пользователь может управлять величиной ускорения всех процессов класса *NORMAL_PRIORITY* с помощью панели управления (команда *System*, вкладка *Performance*, ползунок *Boost Application Performance*).

Когда окно получает сообщение типа *WM_TIMER*, *WM_LBUTTONDOWN* или *WM_KEYDOWN*, планировщик также ускоряет (*boosts*) поток, владеющий этим окном.

Существуют еще ситуации, когда планировщик временно повышает уровень приоритета потока. Довольно часто потоки ожидают

возможности обратиться к диску. Когда диск освобождается, заблокированный поток просыпается, и в этот момент система повышает его уровень приоритета. После ускорения потока планировщик постепенно снижает уровень приоритета до базового значения. Уровень снижается на одну единицу после завершения очередного кванта времени.

Иногда система инвертирует приоритеты, чтобы разрешить конфликты типа *deadlock*. Благодаря динамике изменения приоритетов потоки активного процесса вытесняют потоки фонового процесса, а потоки с низким приоритетом все-таки имеют шанс получить управление.

Пример. Поток1 с высоким приоритетом вынужден ждать, пока Поток2 с низким приоритетом выполняет код в критической секции. В это же время готов к выполнению Поток3 со средним значением приоритета. Он получает время процессора, а два других потока застревают на неопределенное время, так как Поток2 не в состоянии вытеснить Поток3, а Поток1 помнит, что надо ждать, когда Поток2 выйдет из критической секции.

Операционная система *Windows* разрешает эту ситуацию так: планировщик увеличивает приоритеты готовых потоков на величину, выбранную случайным образом. В нашем примере это приводит к тому, что поток с низким приоритетом получает шанс на время процессора и, в течение, может быть, нескольких квантов закончит выполнение кодов критической секции. Как только это произойдет, Поток1 с высоким приоритетом сразу получит управление и сможет, вытеснив Поток3, начать выполнение кодов критической секции.

Программист имеет возможность управлять процессом ускорения потоков с помощью API-функций *SetProcessPriorityBoost* (все потоки данного процесса) или *SetThreadPriorityBoost* (данный поток). Функции *GetProcessPriorityBoost* и *GetThreadPriorityBoost* позволяют выяснить текущее состояние флага.

При наличии нескольких процессоров *Windows* применяет симметричную модель распределения потоков по процессорам *symmetric multiprocessing* (SMP). Это означает, что любой поток может быть направлен на любой процессор. Программист может ввести некоторые коррективы в эту модель равноправного распределения. Функции *SetProcessAffinityMask* и *SetThreadAffinityMask* позволяют указать предпочтения в смысле выбора процессора для всех потоков процесса или для одного определенного потока. Потоковое предпочтение (*thread affinity*) вынуждает систему выбирать процессоры только из множества, указанного в маске.

Существует также возможность для каждого потока указать один предпочтительный процессор. Это делается с помощью функции

SetThreadidealProcessor. Это указание служит подсказкой для планировщика заданий.

Варианты тем теоретической части курсовой работы:

1. Семантика и семантические схемы программ.
2. Модели автоматов. Детерминированные и недетерминированные автоматы.
3. Конечные автоматы. Двоичные автоматы.
4. Формальная спецификация программ.
5. Структурные отношения процессов и отношения между процессами.
6. Модели вычислительных процессов.
7. Организация работы процессов и потоков в системах реального времени.
8. Инициализация, работа и уничтожение процессов в *WinAPI*.
9. Виды и свойства алгоритмов.
10. Критические секции, интервалы, ресурсы и механизмы разрешения проблемы критических ресурсов.
11. Ядро ОС. Виды, характеристики ядра различных видов ОС.
12. Системные процессы: свойства, использование.
13. Пользовательские процессы в ОС.
14. Модели памяти компьютера.
15. Процессы и потоки в ОС *Windows* и ОС *Unix*.
16. Алгоритм Деккера и алгоритм Петерсона и их применение для разрешения проблемы критических интервалов
17. Архитектура памяти компьютера. Виды памяти.
18. Способы адресации в виртуальном адресном пространстве.
19. Блокировка и механизмы разрешения блокировок. Клинич.
20. Виртуальная память. LDT, GDT. Реализация механизма доступа в различных ОС.
21. Средства межпрограммного обмена.
22. Интерфейсы и протоколы для организации межпрограммного обмена.
23. Сети Петри: построение, способы реализации, область применения, ограничения.
24. Применение семафорных механизмов в решении задач синхронизации.
25. Управление процессами и потоками в различных ОС.
26. Реализация семафорных механизмов в 32-х и 64-х разрядных ОС. Общее и различия.
27. Файлы, проецируемые в память.
28. Реализация механизма мониторов Хоара в мультипрограммных системах.

29. Средства межзадачного (межпрограммного) обмена и способы их реализации в различных ОС.
30. Эволюция подсистемы безопасности в линейке ОС Windows.

Варианты тем практической части курсовой работы:

1. Задача о спящем парикмахере

Задача формулируется следующим образом. Представим парикмахерскую, состоящую из кресла и зала ожидания. Посетитель может войти в зал ожидания только в случае наличия в нем свободных мест. Если кресло парикмахера свободно, посетитель может его занять. Если длительное время в парикмахерской нет посетителей, парикмахер засыпает в кресле. Требуется решить задачу таким образом, чтобы исключить попадание «лишних» посетителей в зал ожидания и засыпания парикмахера в момент нахождения посетителя в кресле.

Решение задачи при помощи сетей Петри:

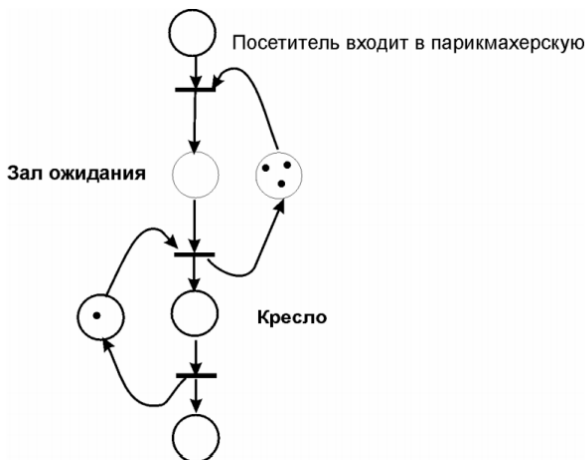


Рис. 12. Решение задачи о спящем парикмахере при помощи сети Петри

2. Задача о пяти обедающих философах

Представим себе парк, по аллеям которого прогуливаются пять философов. В центре парка расположена столовая, в которой накрыт круглый стол. На столе стоит миска со спагетти, пять тарелок и пять вилок. Если философ проголодался, он входит в столовую, занимает свободное место за столом, берет две (обязательное условие!!!) вилки и накладывает на тарелку спагетти. Утолив голод, философ возвращает вилки на стол и покидает столовую. В случае если все пять философов одновременно придут в столовую, займут места за столом и возьмут по

вилке, система окажется заблокированной, т.к. ни один из философов не сможет приступить к еде.



Рис. 13. Обедаящие философы

Требуется организовать систему таким образом, чтобы пять философов не могли одновременно оказаться за столом. Данная задача иллюстрирует конкуренцию между задачами за право монопольного обладания ресурсами. Важным моментом в решении задачи является предотвращение ситуации, когда каждый из философов взял по вилке и, удерживая ее, продолжает ожидать, когда освободится следующая вилка.

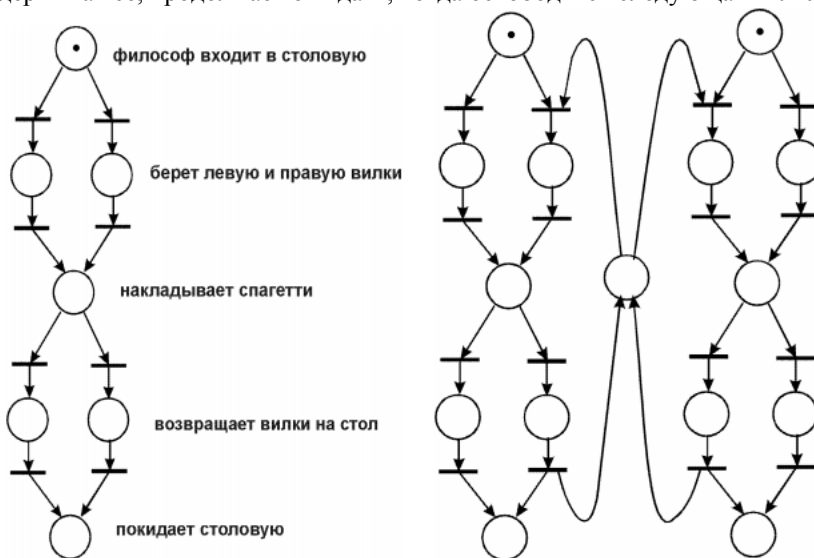


Рис. 14. Решение данной задачи при помощи семафорного механизма и сети Петри.

3. Задача Майхилла (задача о стрелках)

Имеется цепь стрелков и офицер. Каждый находящийся в цепи солдат может общаться только со своими соседями справа и слева. Офицер размещается на фланге цепи и может подавать команды только крайнему в цепи стрелку. Общее количество стрелков в цепи каждому из стрелков неизвестно. Общаться каждый из стрелков может только со своими соседями справа и слева. Требуется обеспечить одновременный залп всех стрелков цепи после подачи команды офицером. Решение задачи выполняется следующим образом: первым команду офицера слышит, естественно, крайний в цепи солдат. Получив команду, он передает ее своему соседу с указанием своего номера в цепи («первый») и начинает считать: один, два, три и т.д. Получивший команду вычисляет свой номер в цепи («второй», «третий» и т.д.) и передает его следующему. Последний в цепи солдат будет знать общее количество стрелков в цепи. Получив команду, он сообщает об этом соседу, ее сообщившему («команду принял»), и начинает отсчет от значения, соответствующего количеству стрелков в цепи, до нуля. Каждый из стрелков цепи, получив обратный сигнал, начинает обратный отсчет от значения, до которого он успел добраться при прямом счете (то есть от своего фактического номера в цепи), до нуля. Досчитав до нуля, стрелок открывает огонь. Как только каждый из стрелков доберется до нуля, цепь выстрелит одновременно. На передачу сигнала от стрелка к стрелку тратится 1 с. Значение счетчика изменяется на 1 за 1 с.

Данная задача решается при помощи автоматной модели поведения стрелка. В течение работы программа-стрелок может находиться в одном из следующих состояний:

- ожидание;
- прямой счет;
- обратный счет;
- открытие огня.

В состояние прямого счета и обратного счета стрелок переходит в случае изменения состояния соседних стрелков.

В состояние прямого счета стрелка переводит изменение состояния соседа слева, в состояние обратного счета переводит изменение состояния соседа справа.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Приведите понятие моментального снимка. С помощью какой функции его можно получить?
2. Что такое адресное пространство процесса?
3. Перечислите области, которые присутствуют в адресном пространстве ОС Windows.
4. Перечислите основные функции для работы с потоками.
5. Возможно, ли создать поток в приостановленном состоянии (не запускается на исполнение)?
6. Перечислите способы завершения потока. Какой из них является наиболее безопасным?
7. Какой относительный приоритет получает поток при создании?
8. Позволяет ли функция *CreateThread* задать относительный приоритет потока?
9. Перечислите средства межпроцессного взаимодействия.
10. Для решения каких задач может применяться межпроцессное взаимодействие?
11. В каких случаях необходимо использовать почтовые ящики?
12. Приведите классификацию каналов (pipe).
13. Как можно организовать параллельные вычисления при помощи именованных каналов?
14. Перечислите области применения файлов, проецируемых в память.
15. Как можно осуществить межпроцессное взаимодействие, используя файлы, проецируемые в память?

ЗАКЛЮЧЕНИЕ

Данное учебное издание представляет собой практикум по дисциплине «Теория вычислительных процессов» для студентов направления 09.03.01 – «Информатика и вычислительная техника».

В практикуме приведены лабораторные работы, рассматривающие чтение карты процессов и потоков, чтение карты памяти, многопоточковую обработку, средства межпроцессного взаимодействия, файлы данных, проецируемые в память. При выполнении лабораторных работ студенты получают практические приемы и навыки в области разработки системного программного обеспечения в операционных системах семейства *Windows NT* (32- и 64-разрядных версиях) и *Windows 9x* с использованием функций *Windows API*.

Также в учебном издании содержатся задания для курсовой работы по организации взаимодействия и синхронизации параллельных процессов и потоков в виде тем теоретической и практической частей курсовой работы и контрольные вопросы по данной дисциплине.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Создание приложений Windows с использованием средств прикладного программирования Win32 API. [Электронный ресурс]: Программирование по-русски на Delphi, C++, PHP, Prolog, GPSS. – Режим доступа к ресурсу: http://www.condingras.ru/readarticle.php?article_id=2001.
2. Майстренко Н.В., Майстренко А.В. Программное обеспечение САПР. Операционные системы. Учебное пособие. – Тамбов: Изд-во ТГТУ, 2007. – 76 с.
3. Рихтер Дж. Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows / Пер. с англ. – 4-е изд. – СПб.: Питер; М., Издательство «Русская редакция»; 2008. – 720 стр.: ил.
4. Олифер В.Г., Олифер Н.А. Сетевые операционные системы. – СПб.: Питер, 2002. – 544 с.
5. MSDN Library. [Электронный ресурс] – Режим доступа к ресурсу: <http://www.msdn.microsoft.com/en-us/library/>.
6. Громова Л.Н. Теория вычислительных процессов и структур: лаб. практикум для студентов специальности 1-08 01 01-07 «Профессиональное обучение (Информатика)». – Мн.: МГВРК, 2009. – 72 с. ISBN 978-985-526-026-5.

Многопоточный сервер именованных каналов

В следующем примере представлен многопоточный сервер, использующий именованные каналы. В нём существует основной поток, в котором выполняется цикл, в котором, в свою очередь, создаётся именованный канал и выполняется ожидание подсоединения клиента. Когда клиент подсоединится, сервер создаёт поток для обслуживания именно этого клиента и продолжает цикл. Успешное соединение клиента с каналом возможно в интервале между вызовами функций *CreateNamedPipe* и *ConnectNamedPipe*. В случае неудачи, *ConnectNamedPipe* вернёт нуль, и функция *GetLastError* вернёт статус `ERROR_PIPE_CONNECTED`.

Поток, созданный для обслуживания каждого канала, считывает из него запросы и записывает в него ответы до того момента, как клиент закроет дескриптор канала. В этом случае поток сбрасывает содержимое буферов канала, отсоединяется от него, закрывает дескриптор канала и завершается.

```
#include <vcl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>
#include <iostream.h>
#define BUFSIZE 512
#define PIPE_TIMEOUT NMPWAIT_USE_DEFAULT_WAIT

int xx = 0;
VOID InstanceThread(LPVOID);
VOID GetAnswerToRequest(LPTSTR, LPTSTR, LPDWORD);
VOID PrintInfo(CHAR *);

int main()
{
    BOOL fConnected;
    DWORD dwThreadId;
    HANDLE hPipe, hThread;
    LPTSTR lpszPipeName = "\\.\pipe\mynamedpipe";

    /*
Основной цикл создаёт именованный канал и ожидает
соединения клиента с ним. При подсоединении клиента
создаётся поток для общения с этим клиентом и происходит
новая итерация цикла
```

```

*/
PrintInfo("*** Сообщения ***\n\n");

for (;;)
{
    hPipe = CreateNamedPipe(
        lpzPipename,          // имя канала
        PIPE_ACCESS_DUPLEX,    // доступ на
                               // чтение/запись
        PIPE_TYPE_MESSAGE |    // канал передачи
                               // сообщений
        PIPE_READMODE_MESSAGE | // режим чтения
                               // сообщений
        PIPE_WAIT,             // блокирующий режим
        PIPE_UNLIMITED_INSTANCES, // максимальное число
                               // экземпляров этого канала в системе
        BUFSIZE,               // размер выходного буфера
        BUFSIZE,               // размер входного буфера
        PIPE_TIMEOUT,          // таймаут для клиента
        NULL);                 // атрибуты
                               //безопасности отсутствуют
    if (hPipe == INVALID_HANDLE_VALUE)
        PrintInfo("Ошибка в функции CreatePipe");

/*
Ожидать подсоединения клиента. В случае успеха будет
возвращено ненулевое значение. Иначе - функция возвращает
нуль, а функция GetLastError вернёт статус
ERROR_PIPE_CONNECTED.
*/

    fConnected = ConnectNamedPipe(hPipe, NULL) ?
        TRUE : (GetLastError() == ERROR_PIPE_CONNECTED);

    if (fConnected)
    {
        // Создать поток для работы с этим клиентом
        hThread = CreateThread(
            NULL,              // Отсутствие атрибутов безопасности
            0,                 // размер стека потока выбрать по
                               // умолчанию
            (LPTHREAD_START_ROUTINE) InstanceThread,
            (LPVOID) hPipe,     // параметр потока
            0,                 // не приостановленный
            &dwThreadId);       // возврат идентификатора
                               // потока

        if (hThread == NULL)

```

```

        PrintInfo("Ошибка в функции CreateThread");
    else
        CloseHandle(hThread);

}
else
    // Клиент не смог соединиться - закрываем поток
    CloseHandle(hPipe);
}
return 1;
}

VOID InstanceThread(LPVOID lpvParam)
{
    CHAR chRequest[BUFSIZE];
    CHAR chReply[BUFSIZE];
    DWORD cbBytesRead, cbReplyBytes, cbWritten;
    BOOL fSuccess;
    HANDLE hPipe;

    // Параметр потока - дескриптор именованного канала

    hPipe = (HANDLE) lpvParam;

    while (1)
    {
        //Считывание запросов клиента из канала
        fSuccess = ReadFile(
            hPipe,          // дескриптор канала
            chRequest,      // выходной буфер
            BUFSIZE,        // размер буфера
            &cbBytesRead,    // сколько байт считывать
            NULL);          // синхронный ввод/вывод

        if (! fSuccess || cbBytesRead == 0)
            break;

        GetAnswerToRequest(chRequest, chReply, &cbReplyBytes);

        // Запись ответа в канал
        fSuccess = WriteFile(
            hPipe,          // дескриптор канала
            chReply,        // входной буфер
            cbReplyBytes,    // сколько байт писать
            &cbWritten,      // сколько байт записано
            NULL);          // синхронный ввод/вывод

        if (! fSuccess || cbReplyBytes != cbWritten) break;
    }
}

```

```

    }

    /*
    Сбросить содержимое канала перед разъединением, чтобы
    клиент мог считать эти данные. Затем разъединить канал и
    закрыть его дескриптор
    */

    FlushFileBuffers(hPipe);
    DisconnectNamedPipe(hPipe);
    CloseHandle(hPipe);
}

VOID GetAnswerToRequest(LPTSTR chRequ, LPTSTR chRepl,
LPDWORD cbRepByte)
{
    cout<< TimeToStr(Time()).c_str()<<": "<<chRequ<<endl;
}

VOID PrintInfo(CHAR *texterr)
{
    CHAR buffer[80];
    CharToOem(texterr,buffer);
    cout<<buffer<<endl;
}

```

Клиент именованных каналов

Клиент именованных каналов использует функцию *CreateFile* для получения дескриптора канала. Если канала существует, но число клиентов равно максимально допустимому значению, функция *CreateFile* возвращает нуль и функция *GetLastError* вернёт статус `ERROR_PIPE_BUSY`. В этом случае клиент использует функцию *WaitNamedPipe* для ожидания освобождения канала.

Функция *CreateFile* завершается неудачей, если параметры доступа несовместимы с параметрами, задаваемыми сервером при создании канала (чтение, запись или чтение/запись).

Дескриптор, возвращаемый функцией *CreateFile*, по умолчанию установлен в режим чтения байт, блокировки при ожидании, с отключенными асинхронным режимом и режимом прямой записи (буферизация операций отключена). Для изменения поведения по умолчанию можно воспользоваться флагами `FILE_FLAG_OVERLAPPED` и `FILE_FLAG_WRITE_THROUGH` для включения асинхронного ввода/вывода и отключения буферизации операций соответственно. Клиент может использовать функцию *SetNamedPipeHandleState* для включения режима чтения/записи без блокировки (задавая флаг `PIPE_WAIT`) и для включения режима чтения сообщений (флаг `PIPE_READMODE_MESSAGE`).

В следующем примере показан клиент именованного канала, который открывает канал, задаёт режим чтения сообщений, использует функцию *WriteFile* для отправки запросов серверу и функцию *ReadFile* для чтения ответов от сервера. Этот клиент именованных каналов может быть использован с любым типом серверов, обслуживающих сообщения, представленных в предыдущих примерах. Однако при попытке соединения с сервером, работающим в режиме потоков байт, работа клиента завершится неудачей при попытке смены режима общения с каналом (*SetNamedPipeHandleState*). Так как клиент считывает данные в режиме чтения сообщений, возможна ситуация, когда *ReadFile* вернёт нуль (считалась часть сообщения). Это происходит, когда размер сообщения превышает размер буфера. В этой ситуации *GetLastError* вернёт статус `ERROR_MORE_DATA`, и клиент может считать оставшуюся часть сообщения, используя дополнительные вызовы *ReadFile*.

```
#include <vcl.h>
#include <windows.h>
#include <iostream.h>
VOID PrintInfo(CHAR *);
int main()
{
    HANDLE hPipe;
```



```

LPVOID lpvMessage;
CHAR chBuf[512];
BOOL fSuccess;
DWORD cbRead, cbWritten, dwMode;
LPTSTR lpszPipename = "\\.\pipe\mynamedpipe";
// Ввод текста передаваемого сообщения
CHAR buffer[80], text[80];
CharToOem("Введите текст, передаваемого на сервер
сообщения: \n", buffer);
cout<<buffer;
cin.getline(text, 80);
lpvMessage=text;
// Попытаться открыть канал; ждать его освобождения в
//случае необходимости
while (1)
{
    hPipe = CreateFile(
        lpszPipename,    // имя канала
        GENERIC_READ |  // доступ на чтение/запись
        GENERIC_WRITE,
        0,               // без разделения
        NULL,            // без атрибутов безопасности
        OPEN_EXISTING,   // открыть уже существующий
        0,               // атрибуты по умолчанию
        NULL);           // не файл шаблона
    //завершиться, если получен неправильный дескриптор канала
    if (hPipe != INVALID_HANDLE_VALUE)
        break;
    // Exit if an error other than ERROR_PIPE_BUSY occurs.
    //завершиться, если произошла любая ошибка, кроме
    //ERROR_PIPE_BUSY

    if (GetLastError() != ERROR_PIPE_BUSY)
        PrintInfo("Не могу открыть канал");
    // Превышено максимальное число соединений с каналом;
    //подождать 20 секунд

    if (! WaitNamedPipe(lpszPipename, 20000) )
        PrintInfo("Не могу открыть канал");
}
//Соединились с каналом. Устанавливаем его в режим чтения
//сообщений
dwMode = PIPE_READMODE_MESSAGE;
fSuccess = SetNamedPipeHandleState(
    hPipe,           // дескриптор канала
    &dwMode,        // новый режим
    NULL,           // не задавать максимальные байты
    NULL,           // (максимальный размер буфера)

```

```

        NULL);    // не задавать максимальный таймаут
    if (!fSuccess)
        PrintInfo("Ошибка в функции
SetNamedPipeHandleState");
    //отправить сообщение серверу именованных каналов
    fSuccess = WriteFile(
        hPipe,                // дескриптор канала
        lpvMessage,           // сообщение
        strlen((char*)lpvMessage) + 1, // длина сообщение
        &cbWritten,            // число записанных байт
        NULL);                // синхронно
    if (! fSuccess)
        PrintInfo("Ошибка в функции WriteFile");
do
{
    // Считывание данных из канала
    fSuccess = ReadFile(
        hPipe,    // дескриптор канала
        chBuf,    // буфер для получения данных
        512,      // его размер
        &cbRead,  // число считанных байт
        NULL);    // синхронно

    if (! fSuccess && GetLastError() != ERROR_MORE_DATA)
        break;
    //Ответ из канала выводится на стандартное устройство
    //вывода
    if (! WriteFile(GetStdHandle(STD_OUTPUT_HANDLE),
        chBuf, cbRead, &cbWritten, NULL))
    {
        break;
    }
} while (! fSuccess); // повторить цикл, если
//произошла ошибка ERROR_MORE_DATA

    PrintInfo("Сообщение отправлено");
    CloseHandle(hPipe);
    system("pause");
    return 0;
}

VOID PrintInfo(CHAR *texterr)
{
    char buffer[80];
    CharToOem(texterr,buffer);
    cout<<buffer<<endl;
}

```