МИНОБРНАУКИ РОССИИ
федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Самарский государственный технический университет»

Кафедра "Вычислительная техника"

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Методические указания

Самара
Самарский государственный технический университет
2013

Печатается по решению методического совета факультета АИТ

УДК 681.324

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ: Метод. указ. к лаб. работам / Самар. гос. техн. ун-т; Сост. *А.А. Тихомиров, А.М.Кистанов*. Самара, 2013. 80 с.

Изложены методы получения информации о состоянии ресурсов вычислительной системы, описаны используемые функции интерфейса Win32.

Приведен материал, необходимый для выполнения семи лабораторных работ по курсу «Системное программное обеспечение» в среде MS Visual Studio 2010. Методические указания рассчитаны на бакалавров направлений 230100 «Информатика и вычислительная техника» и 231000 «Программная инженерия» и родственных направлений, изучающих организацию и функционирование операционных систем MS Windows на системном уровне.

Составители: А. А. ТИХОМИРОВ, А. М. КИСТАНОВ
Рецензент к.т.н., доцент каф. «Информационные технологии» СамГТУ Чернышев С.В.

ЛАБОРАТОРНАЯ РАБОТА № 1

МОНИТОР ПРОЦЕССОВ И ПОТОКОВ

Цель работы – практическое знакомство с методикой использования функций Win32 API для получения информации о процессах, потоках, модулях и кучах ОС Windows в консольном приложении на языке C++.

1. КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.1 Получение списка процессов, выполняющихся в системе

Задача получения списка выполняющихся в системе процессов является одной из основных при выполнении мониторинга ресурсов, как отдельного ПК, так и локальной вычислительной сети в целом, поэтому для ее решения имеется встроенное системное средство – диспетчер задач. Кроме того, разработано значительное количество утилит – Process Explorer, System Explorer и т. п. [4].

Все перечисленные программные средства используют как функции Win32 API (Application Program Interface – прикладной программный интерфейс), так и функции еще одного базового интерфейса, называемого *Native API* (естественный API). Внешняя часть Native API пользовательского режима содержится в модуле ntdll.dll, «настоящий» интерфейс реализован в ntoskernel.exe – ядре операционной системы NT (NT operation system kernel). Функции Win32 API, как правило, обращаются к функциям Native API, отбрасывая часть полученной от них информации. Поэтому использование функций Native API позволяет получить более эффективное ПО.

К функциям Win32 API для получения информации о выполняющихся в системе процессах относятся функции

CreateToolhelp32Snapshot(), Process32First(), Process32Next(), Thread32First (), Thread32Next(), Module32First(), Module32Next(), Heap32ListFirst(), Heap32ListNext() и некоторые другие [2]. Самая известная из функций Native API для доступа к содержимому многих важных внутренних структур операционной системы, таких как списки процессов, потоков, дескрипторов, драйверов и т. п. – функция NtQuerySystemInformation ().

1.1.1 Использование функций CreateToolhelp32Snapshot () и Process32xxxx() для получения списка имен процессов

Первый этап получения информации о выполняющихся в системе процессах - создание *снимка* (snapshot) системы, который содержит информацию о состоянии системы в момент создания снимка.

Снимок создается с помощью вызова функции CreateToolhelp32Snapshot (dwFlags, th32ProcessID), первый аргумент определяет, какая информация будет записана в снимок - возможные значения dwFlags приведены в таблице 1 [1].

Таблица 1

Значения флагов функции CreateToolhelp32Snapshot

Флаг - dwFlags	Описание
TH32CS_SNAPHEAPLIST	В снимок включается информация о динамически распределяемой памяти указанного процесса
TH32CS_SNAPPROCESS	В снимок включается список процессов, присутствующих в системе
TH32CS_SNAPTHREAD	В снимок включается список потоков
TH32CS_SNAPMODULE	В снимок включается список модулей, принадлежащих указанному процессу
TH32CS_SNAPALL	В снимок включается список куч, процессов, потоков и модулей

Второй аргумент определяет процесс (указанием его идентификатора), информация о котором необходима (если требуется список куч или модулей). В остальных случаях он игнорируется.

Второй этап - извлечение из снимка списка процессов. Для выполнения этой операции служат функции:

BOOL Process32First (HANDLE hSnapshot, LPPROCESSENTRY32 lppe);

BOOL Process32Next (HANDLE hSnapshot, LPPROCESSENTRY32 lppe);

Первый аргумент - хэндл созданного снимка (возвращает функция CreateToolhelp32Snapshot).

Второй аргумент- структура, содержащая 7 полей:

1. Первое поле этой структуры dwSize - должно перед вызовом функции содержать размер структуры PROCESSENTRY32 в байтах, т.е. sizeof (PROCESSENTRY32).
2. Второе поле cntUsage - содержит число ссылок на процесс, то есть число потоков, которые в настоящий момент используют какие-либо данные процесса.
3. Третье поле th32ProcessID - является идентификатором процесса.
4. Поле pcPriClassBase содержит базовый приоритет процесса.
5. Поле szExeFile[MAX_PATH] содержит имя файла, создавшего процесс.
6. Шестое поле cntThreads - определяет число потоков, принадлежащих процессу.
7. Седьмое поле th32ParentProcessID - является идентификатором родительского по отношению к текущему процесса.

Для того, чтобы получить информацию о первом процессе в снимке, необходимо вызвать функцию Process32First. В случае успешного завершения функция возвращает TRUE. Для того, чтобы просмотреть все оставшиеся процессы, нужно вызывать функцию Process32Next до тех пор, пока она не возвратит FALSE.

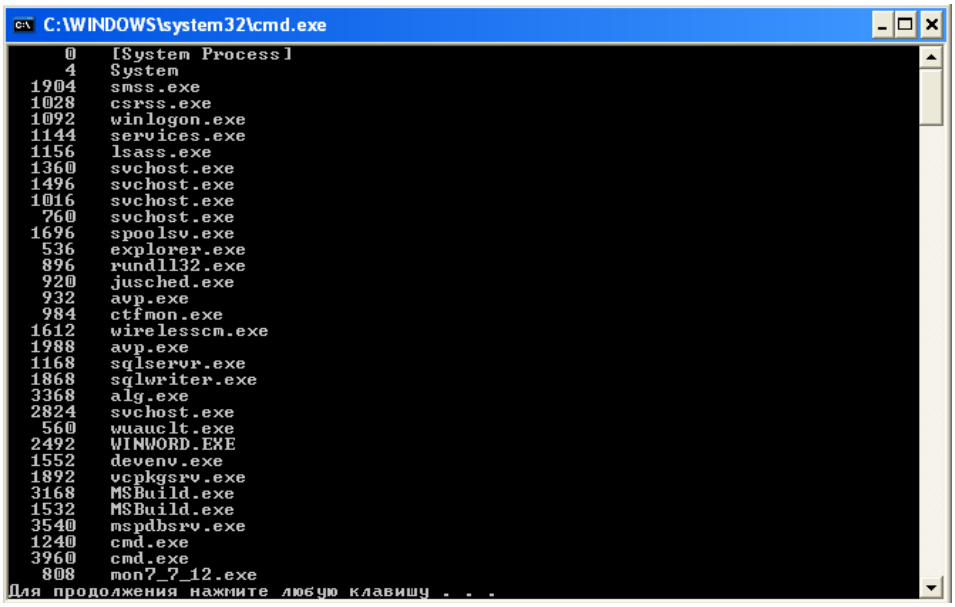
Пример 1. Получить список имен выполняющихся в системе процессов, используя рассмотренные выше функции. В список директив

приложения следует добавить директиву #include <tlhelp32.h>

Исходный текст приложения для примера 1 приведен ниже.

```
#include "stdafx.h"
#include <Windows.h>
#include <tlhelp32.h>
#include <clocale>
#include <iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
setlocale(LC_CTYPE, "RUS"); // русификация вывода
// получение снимка процессов
HANDLE Hs=CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS,0);
PROCESSENTRY32 P;
P.dwSize = sizeof(PROCESSENTRY32);
if (Process32First(Hs,&P))
{
	do
	{
		printf ("%6d",P.th32ProcessID);
		wprintf(L"%1s %-30s \n"," ",P.szExeFile);
	}
	while (Process32Next(Hs,&P));
	CloseHandle(Hs);
	}
}
```

Результат выполнения примера 1 показан на рис. 1.



Р и с. 1. Список выполняющихся процессов в ОС Windows

1.1.2 Использование функций CreateToolhelp32Snapshot () и Thread32xxxx() для получения сведений о приоритетах потоков процессов

Для получения сведений о приоритетах потоков необходимо извлечь из снимка состояния системы с помощью функций Thread32First() и Thtead32Next () значения соответствующих полей.

Обращение к функциям имеет вид:

BOOL Thread32First (HANDLE hSnapshot, LPTHREADENTRY32 lpte);

BOOL Thtead32Next (HANDLE hSnapshot, LPTHREADENTRY32 lpte);

Первый аргумент - хэндл созданного снимка (возвращает функция CreateToolHelp32Snapshot).

Второй аргумент - структура, содержащая 7 полей, наиболее важные из которых перечислены ниже:

1. Поле dwSize должно перед вызовом функции содержать размер структуры в байтах - sizeof (THREADEntry32).
2. Поле th32OwnerProcessID хранит идентификатор процесса – владельца потока.
3. Поле tpBasePri содержит текущий приоритет потока.

4. Поле tpDeltaPri содержит разность между текущим уровнем приоритета потока и базовым уровнем, то есть тем, который присваивается при создании потока.

1.1.3 Использование функций CreateToolhelp32Snapshot () и Module32xxxx() для получения списка модулей

Для получения сведений о модулях, используемых одним из процессов, необходимо:

– получить снимок выполняющихся процессов;
– извлечь из снимка имена процессов и их идентификаторы, вывести их на экран;
– ввести с клавиатуры идентификатор процесса, для которого требуется получить список принадлежащих ему модулей (можно использовать идентификатор любого прикладного процесса);
– используя введенный идентификатор в качестве второго аргумента функции CreateToolhelp32Snapshot (), получить снимок списка модулей выбранного процесса;
– извлечь из полученного снимка с помощью функций Module32First() и Module32Next() значения соответствующих полей.

Обращение к функциям имеет вид:

BOOL Module32First (HANDLE hSnapshot, MODULEENTRY32 lpte);
BOOL Module32Next (HANDLE hSnapshot, MODULEENTRY32 lpte).

Первый аргумент - хэндл созданного снимка, возвращаемый функцией CreateToolhelp32Snapshot).

Второй аргумент – структура, содержащая 10 полей:

unsigned long dwSize – размер структуры MODULEENTRY32;
unsigned long th32ModuleID – не используется, заполняется значением 1;
unsigned long th32ProcessID – идентификатор процесса, владеющего модулем;

unsigned long GlblcntUsage – счетчик глобальных пользователей модуля;
unsigned long ProccntUsage – счетчик процессов – пользователей;
unsigned char* modBaseAddr – базовый адрес модуля;
unsigned long modBaseSize – базовый размер модуля;
void * hModule – хэндл модуля;
char[256] szModule – имя модуля;
char[260] szExePath – путь размещения модуля.

1.2. Завершение выбранного процесса

Для завершения процесса используется функция TerminateProcess (HandleProc, ExitCode). Первый аргумент функции – описатель или хэндл процесса типа Handle – возвращается функцией, создавшей процесс, второй аргумент – код возврата типа DWord.

Значение описателя необходимо получить с помощью функции OpenProcess (unsigned long dwDesiredAcccess,

// флаг доступа – например, PROCESS_TERMINATE
int bInheritHandle, // handle inheritance flag
unsigned long dwProcessId); // идентификатор процесса

Некоторые значения первого параметра функции OpenProcess() приведены в таблице 2 [1].

Таблица 2

Некоторые значения первого параметра функции OpenProcess()

Вид доступа	Описание
PROCESS_ALL_ACCESS	Задает все возможные флаги доступа к объекту «процесс»
PROCESS_SET_INFORMATION	Разрешает использование описателя процесса в функции SetPriorityClass для задания класса приоритета процесса
PROCESS_TERMINATE	Разрешает использование описателя процесса в функции TerminateProcess() для завершения процесса.

Алгоритм завершения процесса включает следующие шаги:

1. Вывести на экран список имен процессов и их идентификаторов, используя описанную в п.1.1.1 методику.
2. Ввести с клавиатуры идентификатор завершаемого прикладного процесса.
3. По идентификатору процесса получить его описатель, используя функцию OpenProcess().
4. Если описатель получен (его значение не равно 0), завершить процесс, используя функцию TerminateProcess (HandleProc, ExitCode) и полученный описатель.

Для процессов с небольшими значениями ProcID – системных процессов – функция OpenProcess () не возвращает описатель, так как обычное приложения не должно иметь возможности останавливать системные процессы (службы). В то же время иногда необходимо иметь средство для удаления из системы зависшей службы.

Уровень привилегий приложения можно повысить (но для этого необходимо иметь привилегии администратора). Поэтому данная часть работы может быть выполнена только на компьютере с привилегиями администратора. В частности, отладчик для выполнения своих функций должен обладать самыми широкими полномочиями в отношении всех процессов системы. Изменение уровня привилегий процесса выполняется при помощи следующих действий [1]:

1. Открыть токен доступа процесса с помощью функции OpenProcessToken() из библиотеки advapi32.dll.
2. Подготовить структуру TOKEN_PRIVILEGES, в которой разместить информацию о требуемом уровне привилегий.
3. Обратиться к функции AdjustTokenPrivilages().

Пример 2 функции для изменений уровня привилегий процесса.

```
void IAmaDebugger(bool debug)
{
```

```
HANDLE hCp;
if (OpenProcessToken(GetCurrentProcess(),
TOKEN_ADJUST_PRIVILEGES, &hCp))
{
TOKEN_PRIVILEGES tp;
tp.PrivilegeCount = 1;
LookupPrivilegeValue(0, SE_DEBUG_NAME, &tp.Privileges[0].Luid);
tp.Privileges[0].Attributes = debug ? SE_PRIVILEGE_ENABLED : 0;
AdjustTokenPrivileges(hCp, 0, &tp, sizeof(tp), 0, 0);
CloseHandle(hCp);
}
}
```

1.3. Получение дополнительной информации о процессах и потоках

Для получения информации о времени работы процессов и их потоков, используемой памяти и других ресурсов служат функции Win32 API GetProcessTimes(), GetProcessIoCounters(), GetProcessHandleCount(), EnumDeviceDrivers(), GetProcessMemoryInfo(), GetProcessWorkingSetSize() [1].

1.3.1 Получение информации о времени выполнения процессов

Функция BOOL WINAPI GetProcessTimes(__in HANDLE hProcess, __out LPFILETIME lpCreationTime, __out LPFILETIME lpExitTime, __out LPFILETIME lpKernelTime,__out LPFILETIME lpUserTime); используется для получения времени запуска (создания), времени завершения, времени работы процесса в режиме ядра и пользователя. Процесс задается описателем hProcess, время возвращается ОС в переменных типа LPFILETIME. Время отсчитывается в 100

наносекундных интервалах с 1.01.1601 по Гринвичу. Для представления времени старта и завершения в привычном формате используются функции

BOOL FileTimeToLocalFileTime (const FILETIME* lpFileTime, LPFILETIME lpLocalFileTime);

BOOL WINAPI FileTimeToSystemTime (__in const FILETIME *lpFileTime, __out LPSYSTEMTIME lpSystemTime);

Используя поля структуры lpSystemTime, можно получить дату и время старта или завершения процесса с точностью до миллисекунды.

Пример 3 использования перечисленных функций:

```
GetProcessTimes(hProc,&CreationTime,&ExitTime,&KernelTime,&UserTime);
  FileTimeToLocalFileTime(&CreationTime,&LocCreationTime);
  FileTimeToSystemTime(&LocCreationTime,&sysCreation);
  FileTimeToSystemTime(&KernelTime,&sysKernel);
```

Для получения значений описателей процесса в общем случае следует использовать функцию OpenProcess():

```
hProc=OpenProcess(PROCESS_QUERY_INFORMATION, FALSE, P.th32ProcessID);
```

В частном случае, когда интерес представляет текущий процесс, описатель процесса возвращается функцией GetCurrentProcess().

1.3.2 Получение информации счетчиков ввода-вывода и количества описателей (дескрипторов)

Содержимое счетчиков ввода-вывода – количество прочитанных или записанных байт может быть получено функцией

BOOL WINAPI GetProcessIoCounters(__in HANDLE hProcess, __out PIO_COUNTERS lpIoCounters);

Первый аргумент определяет процесс, второй – указатель на структуру

typedef struct _IO_COUNTERS {
ULONGLONG ReadOperationCount; ULONGLONG WriteOperationCount;
ULONGLONG OtherOperationCount; ULONGLONG ReadTransferCount;
ULONGLONG WriteTransferCount; ULONGLONG OtherTransferCount;
} IO_COUNTERS, *PIO_COUNTERS;

Для использования описанной функции необходимо объявить экземпляр структуры IO_COUNTERS IoCounters и вызвать функцию GetProcessIoCounters(hProc, &IoCounters);

Функция
BOOL WINAPI GetProcessHandleCount(__in HANDLE hProcess,
__inout PDWORD pdwHandleCount);
возвращает указатель на переменную типа Dword, содержащую количество созданных указанных процессом описателей (дескрипторов). Для использования функции необходимо описать переменную DWORD HandCount и указать вторым аргументом функции адрес этой переменной, например

GetProcessHandleCount (GetCurrentProcess, &HandCount);

1.3.3 Получение информации об используемой процессом памяти

Функция BOOL WINAPI GetProcessMemoryInfo (__in HANDLE Process, __out PPROCESS_MEMORY_COUNTERS ppsmemCounters,
__in DWORD cb
);
из библиотеки Psapi.lib содержит различную информацию об используемой процессом памяти.

В приложение необходимо добавить директиву #pragma comment (lib,"psapi.lib") для подключения библиотеки psapi.lib.

Процесс задается описателем Process, структура, содержащая информацию об использовании памяти

```
typedef struct _PROCESS_MEMORY_COUNTERS {
DWORD cb;  // размер структуры
DWORD  PageFaultCount; // количество страничных ошибок
SIZE_T PeakWorkingSetSize;// пиковый размер используемой ОП
SIZE_T WorkingSetSize;// используемый объем ОП
SIZE_T QuotaPeakPagedPoolUsage; //максимальный размер страничного пула памяти
SIZE_T QuotaPagedPoolUsage; // размер страничного пула памяти
SIZE_T QuotaPeakNonPagedPoolUsage; //макс размер невыгружаемого пула памяти
SIZE_T QuotaNonPagedPoolUsage;// размер невыгружаемого пула памяти
SIZE_T PagefileUsage; // использование страничного файла
SIZE_T PeakPagefileUsage;
} PROCESS_MEMORY_COUNTERS,
*PPROCESS_MEMORY_COUNTERS;
```

задается указателем.

Функция

```
BOOL WINAPI GetProcessWorkingSetSize(__in HANDLE hProcess,
__out PSIZE_T lpMinimumWorkingSetSize,
__out PSIZE_T lpMaximumWorkingSetSize);
```

возвращает минимальный и максимальный размеры рабочего множества страниц указанного процесса (в байтах).

1.3.4 Оценка загрузки процессора процессом с использованием счетчиков производительности

Один из способов оценки загрузки процессора процессом – использование данных счетчиков производительности. Счетчики производительности создаются и поддерживаются ОС Windows, их

значения заносятся в реестр. Для доступа к значениям счетчиков, как правило, используется библиотека PDH.

Различают счетчики локального компьютера и счетчики компьютеров локальной сети. Каждый счетчик производительности относится к определенной категории – например, имеются такие категории, как Процессор, Процесс, Система, Память и т.д. Для некоторых категорий счетчиков имеется несколько экземпляров счетчиков – например, счетчики категории Процессор или Процесс. Каждый счетчик имеет определенное функциональное назначение, например, имеются счетчики категории Процесс - % загруженности процессора, % работы в пользовательском режиме и т.д. Проще всего ознакомиться с категориями, экземплярами и назначением поддерживаемых счетчиков можно, ознакомившись с графическим интерфейсом утилиты perfmon.

Программный интерфейс (API) библиотеки PDH включает следующие функции:

- PdhAddCounter(hQuery, string, 0, &hCounter);
- PdhCollectQueryData(hQuery);
- PdhGetFormattedCounterValue(hCounter, PDH_FMT_DOUBLE, 0, &value).

Первая функция используется для добавления нового счетчика. Добавляемый счетчик задается строкой string вида

\\Процесс(name)\\% загруженности процессора".

Вторая функция служит для многократного обращения к выбранному счетчику для съема измерений. Третья функция выполняет форматирование полученных данных в соответствии с заданным форматом – в примере PDH_FMT_DOUBLE – числа с плавающей точкой. Подробное описание перечисленных выше функций и примеры их использования можно найти в MSDN.

2. МЕТОДИКА ВЫПОЛНЕНИЯ

2.1. Выполнить базовые задания для всех бригад:

Вывести на экран список процессов, выполняющихся в системе.

Для выбранного процесса (ввести его идентификатор) вывести сведения о его приоритете и количестве потоков.

Для выбранного прикладного процесса (ввести его идентификатор) вывести время его работы в режиме ядра и в режиме пользователя.

2.2. Вывести на экран информацию о загрузке каждого ядра процессора или о загрузке процессора каждым процессом.

2.3. Выполнить индивидуальное задание 1 согласно таблице 3.

2.4. Выполнить индивидуальное задание 2 согласно таблице 4.

Таблица 3

Варианты индивидуального задания 1 для бригад

№№ бригад	Варианты задания
1, 3	Для выбранного процесса вывести сведения об используемых им модулях (имя модуля и его размер). Процесс выбирать с помощью ввода с клавиатуры его идентификатора.
2, 4	Вывести список имен выполняющихся процессов с указанием ID и количества потоков, упорядочить список по количеству потоков процессов
5, 6	Для выбранного процесса вывести список имен дочерних процессов. Процесс выбирать с помощью ввода с клавиатуры его идентификатора.
7, 9	Вывести список выполняющихся процессов с указанием ID, имени, базового приоритета, количества потоков и используемых модулей
9, 10	Вывести список имен выполняющихся процессов, у которых есть потомки.

Таблица 4

Варианты индивидуального задания 2 для бригад

№№ бригад	Варианты задания
1	Для выбранного прикладного процесса вывести количество страничных ошибок. Процесс выбирать путем ввода его PID.
2	Для выбранного процесса вывести количество введенных и записанных байтов. Процесс выбирать путем ввода его PID.
3	Для выбранного прикладного процесса вывести размер используемой ОП и его максимальное значение. Процесс выбирать путем ввода его PID.
4	Для выбранного прикладного процесса вывести размер рабочего множества. Процесс выбирать путем ввода его PID.
5	Для выбранного прикладного процесса вывести время его старта – мин и сек. Процесс выбирать путем ввода его PID. Результат сравнить с данными ОС на панели задач.
6	Для выбранного прикладного процесса вывести время его пребывания в системе с момента запуска – мин и сек. Процесс выбирать путем ввода его PID. Результат сравнить с данными ОС на панели задач.
7	Для выбранного прикладного процесса вывести размер выгружаемого пула страниц и его максимальное значение. Процесс выбирать путем ввода его PID. Результат сравнить с данными диспетчера задач.
8	Для выбранного прикладного процесса вывести размер невыгружаемого пула страниц и его максимальное значение. Процесс выбирать путем ввода его PID.
9	Вывести список имен прикладных процессов, использовавших более T мсек времени ЦП. Значение T вводить с клавиатуры после запуска программы
10	Вывести список имен прикладных процессов, использовавших более M байт оперативной памяти. Значение M вводить с клавиатуры

Полученные результаты сравнить данными диспетчера задач.

2.4. Ответить на контрольные вопросы и подготовить отчет о работе в соответствии с п. 3.

2.5. Распечатать отчет на принтере любого типа.

2.6. Готовый отчет представить преподавателю для отметки о выполнении работы

3. СОДЕРЖАНИЕ ОТЧЕТА

Отчет о работе должен содержать:

1. Титульный лист с номером, названием работы, номером группы, бригады, фамилиями студентов и преподавателей
2. Перечень использованных при выполнении работы функций Win32 API и их назначение в виде таблицы
3. Описание алгоритма и текст программной реализации индивидуальных заданий.
4. Результаты, полученные при выполнении индивидуальных заданий.
5. Выводы.

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Функции Win32 API для получения сведений о выполняющихся в системе процессах и используемых ими ресурсах.
2. Атрибуты (описатели, характеристики) процесса Win32.
3. Алгоритм работы приложения для получения списка имен выполняющихся процессов.
4. Алгоритм работы приложения для получения списка модулей, используемых процессом.
5. Алгоритм принудительного завершения процесса.
6. Функции Win32 API для получения информации о времени выполнения процессов и потоков.
7. Функции Win32 API для преобразования времени из формата filetime в

общепринятый формат и методика их использования.
8. Функции Win32 API для получения информации счетчиков ввода-вывода.
9. Функции Win32 API для получения информации об используемой процессом памяти
10.Назначение и использование функции OpenProcess.

ЛАБОРАТОРНАЯ РАБОТА № 2

ПРОЦЕССЫ И ПОТОКИ. ИССЛЕДОВАНИЕ ДИСПЕТЧЕРИЗАЦИИ ПОТОКОВ

Цель работы - знакомство со средствами создания и управления процессами и потоками, исследование алгоритма диспетчеризации потоков ОС Windows.

1. КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.1 Функции Win32 для создания и управления процессами

Процесс (Process) в ОС Windows в момент создания получает в собственное владение единственный программный *поток* (Thread), который при исполнении может создавать другие потоки.

Для **создания** нового процесса можно использовать две функции: Win32 – WinExec() – (устаревшая) или CreateProcess().

UINT WINAPI WinExec(__in LPCSTR lpCmdLine,
__in UINT uCmdShow);

BOOL WINAPI CreateProcess(__in_opt LPCTSTR lpApplicationName,
__inout_opt LPTSTR lpCommandLine,
__in_opt LPSECURITY_ATTRIBUTES lpProcessAttributes,

__in_opt LPSECURITY_ATTRIBUTES lpThreadAttributes,
__in BOOL bInheritHandles,
__in DWORD dwCreationFlags,
__in_opt LPVOID lpEnvironment,
__in_opt LPCTSTR lpCurrentDirectory,
__in LPSTARTUPINFO lpStartupInfo,
__out LPPROCESS_INFORMATION lpProcessInformation);.

Для **завершения** процесса используются функции VOID WINAPI ExitProcess(__in UINT uExitCode) или BOOL WINAPI TerminateProcess(__in HANDLE hProcess, __in UINT uExitCode).

Для **управления выполнением** процесса используются две функции:

– HANDLE WINAPI GetCurrentProcess (void); - получение описателя текущего процесса;
– BOOL WINAPI SetPriorityClass (__in HANDLE hProcess,
__in DWORD dwPriorityClass); - изменение класса приоритета процесса

Параметр dwPriorityClass может принимать ТОЛЬКО, перечисленные ниже значения [4]:

– HIGH_PRIORITY_CLASS - для процесса, выполняющего критичные по времени задачи;
– IDLE_PRIORITY_CLASS - для процесса, потоки которого должны выполняться, когда нет более приоритетных задач – например, хранитель экрана;
– NORMAL_PRIORITY_CLASS - для большинства процессов (значение по умолчанию);
– REALTIME_PRIORITY_CLASS - наибольшее возможное значение класса приоритета.

1.2 Функции Win32 для создания и управления потоками

Для создания потока служит функция HANDLE CreateThread (
__in SEC_ATTRS SecurityAttributes,
__in ULONG StackSize,
__in SEC_THREAD_START StartFunction,
__in PVOID ThreadParameter,
__in ULONG CreationFlags,
__out PULONG ThreadId);

Для завершения потока служат функции:

- VOID WINAPI ExitThread(__in DWORD dwExitCode);
- BOOL WINAPI TerminateThread(__inout HANDLE hThread,
__in DWORD dwExitCode);

Функция ExitThread () используется для обычного (штатного) завершения потока, функция TerminateThread() завершает поток аварийно, без освобождения принадлежащих ему ресурсов.

Для **управления приоритетом** потока используются функции

- HANDLE WINAPI GetCurrentThread(void) – получение описателя текущего потока;
- BOOL WINAPI SetThreadPriority(__in HANDLE hThread, __in int nPriority) - установка относительного приоритета потока в пределах процесса; hThread – описатель потока, nPriority – константа, определяющая величину относительного приоритета потока, которая может принимать следующие 7 значений (в порядке возрастания приоритета):
- THREAD_PRIORITY_IDLE;
- THREAD_PRIORITY_LOWEST;
- THREAD_PRIORITY_BELOW_NORMAL;
- THREAD_PRIORITY_NORMAL;
- THREAD_PRIORITY_ABOVE_NORMAL;

– THREAD_PRIORITY_HIGHEST;
– THREAD_PRIORITY_TIME_CRITICAL.

1.3 Методика исследования алгоритма диспетчеризации потоков

Для исследования алгоритма диспетчеризации потоков следует создать тестовое приложение, содержащее три потока - главный и два дополнительных, приоритеты которых нужно будет изменять в ходе выполнения работы (при запуске каждого потока). Дополнительные потоки должны выполнять вычисление какого-либо математического выражения в бесконечном цикле (для создания нагрузки на ЦП), а главный - отображать количество выполненных итераций циклов за фиксированное время, например, за последнюю секунду. Приложение должно выводить эти значения раз в секунду и сбрасывать счетчики итераций.

Если запущенное приложение – единственное, выполняемое на компьютере, и загрузкой ЦП другими приложениями и системными потоками можно пренебречь, то количество выполненных итераций дает возможность оценить время ЦП, затраченное на выполнение каждого дополнительного потока и сделать выводы о влиянии приоритета потока на количество получаемых этим потоком квантов времени ЦП.

2. МЕТОДИКА ВЫПОЛНЕНИЯ

1. Разработать тестовое приложение в соответствии с методикой, рассмотренной в п.1.3, проверить его работу.
2. Построить зависимость количества итераций от приоритета потока при 3 значениях приоритета – ниже нормального, нормальный, выше нормального. При выполнении задания изменять приоритет только

ОДНОГО потока. Повторить эксперимент, установив для приложения ограничение на использование только одного ядра ЦП. Оценить результат влияния приоритета на количество выполняемых итераций в зависимости от доступности всех ядер процессора или только одного ядра.

3. Запустить две копии тестового приложения. Оценить зависимость количества итераций в потоках от активности окна приложения (изменять активное окно вручную). При выполнении задания изменять приоритет ОДНОГО потока. Ограничивать ядра не обязательно.

4. С целью исследования влияния загрузки ЦП на зависимость количества вычислений потока от его приоритета добавить задержку на 5 мсек в цикл каждого потока. Выполнить эксперимент с ограничением на использование только одного ядра процессора и без ограничения. Сравнить влияние приоритета на количество выполняемых итераций в зависимости от доступности ядер процессора.

5. Повторить предыдущий эксперимент, уменьшив задержку до 1 мсек.

6. Для каждого эксперимента построить диаграммы зависимости количества итераций от приоритета. На одной диаграмме отображать как случай, в котором доступны все ядра, так и тот, где на приложение наложено ограничение выполнения только на одном ядре.

7. С помощью диспетчера задач получить количество потоков тестового приложения и оценить загрузку ЦП.

3. СОДЕРЖАНИЕ ОТЧЕТА

Отчет о работе должен содержать:

1. Исходный код тестового приложения, разработанный в п.1.
2. Таблицы и построенные по ним графики (п.п. 2-3 и 6, п.7).

3.Таблица оценки влияния активности окна приложения на производительность его потоков для 3 значений приоритета одного потока.

4. Скриншот диспетчера задач с загрузкой процессора и количеством потоков приложения (п. 7).

5. 5. Вывод. Обобщить результаты экспериментов и сформулировать критерий влияния приоритета потоков на их производительность/

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Функции Win32, используемые для создания процессов.
2. Функции Win32, используемые для управления выполнением процессов
3. Функции Win32, используемые для завершения процессов.
4. Функции Win32, используемые для создания и завершения потоков.
5. Составляющие приоритета потока.
6. Функции Win32, используемые для управления приоритетом потока.
7. Характер влияния приоритета потока на количество выполняемых вычислений (производительность).
8. Характер влияния активности окна приложения на производительность потоков приложения.
9. Влияние загрузки ЦП на характер зависимости производительности потока от его приоритета.
10. Голодание потока с низким приоритетом.
11. Каким образом можно наглядно показать динамическое повышение приоритета голодающего потока?

ЛАБОРАТОРНАЯ РАБОТА № 3

СРЕДСТВА СИНХРОНИЗАЦИИ ПОТОКОВ

Цель работы - практическое знакомство с методами синхронизации двух потоков одного процесса с помощью критических

секций и объектов ядра ОС MS Windows - мьютексов, семафоров и событий, тупиками и их распознаванием средствами ОС Windows.

1. КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

В многопоточной многозадачной ОС при одновременной модификации неразделяемого ресурса (например, глобальной переменной) двумя и более потоками возможна потеря выполненных изменений. Для правильной работы многопоточных приложений необходимо обеспечить поочередный доступ потоков к участкам кода, выполняющим изменение и запись значений переменной в память (критическим участкам).

Только один процесс должен иметь возможность обращаться к общему неразделяемому ресурсу с целью его модификации, например, *изменять значения глобальных переменных.*

Участок программного кода, в котором поток обращается к общему неразделяемому ресурсу с целью его модификации, называется *критическим участком.*

Правильная работа программы, содержащей критические участки, возможна только при поочередном выполнении потоками программы критических участков.

Для решения этой задачи могут использоваться простые средства – критические секции, обеспечивающие поочередный доступ к критическим участкам потоков одного процесса, и более сложные средства – объекты ядра ОС – Mutex (mutually exclusive – взаимно - исключающий) - *мьютексы*, решающие такую же задачу для потоков, созданных *различными* процессами. Кроме того, задача обеспечения поочередного доступа потоков к критическим участкам может быть решена с помощью универсальных средств синхронизации - *семафоров и событий.*[2, 3].

В рассмотренном ниже приложении два потока увеличивают значение глобальной переменной Global от начального значения 100 на 1 при каждом выполнении цикла. Число повторений цикла каждого потока равно 12, результаты увеличения переменной Global выводятся на экран.

При правильной работе приложения конечным значением переменной должно быть число 100+12+12=124. Однако без использования средств синхронизации такого значения получить в общем случае не удается, так как потоки могут перезаписывать значения переменной Global.

1.1 Критические секции

Критические секции – средство синхронизации потоков одного приложения. Критические секции обеспечивают поочередный доступ потоков к критическим участкам программы. Для использования критической секции ее необходимо *создать*, то есть объявить глобальную переменную CSect: CRITICAL_SECTION CSect и инициализировать ее с помощью вызова функции

```
void InitializeCriticalSection(
LPCRITICAL_SECTION lpCriticalSection);
```

Для обеспечения поочередного доступа потоков к критическому участку в начале критического участка вызывается функция

```
void WINAPI EnterCriticalSection(
__inout LPCRITICAL_SECTION lpCriticalSection);
```

а в конце – функция

```
void WINAPI LeaveCriticalSection(
__inout LPCRITICAL_SECTION lpCriticalSection);
```

После окончания использования критической секции она должна быть уничтожена вызовом функции

void WINAPI DeleteCriticalSection (__inout LPCRITICAL_SECTION lpCriticalSection);

1.2 Мьютексы

Мьютекс – объект ядра, используемый как средство синхронизации доступа потоков одного или нескольких приложений к критическому участку. Мьютекс создается с помощью функции [1]

```
HANDLE WINAPI CreateMutex (
__in_opt LPSECURITY_ATTRIBUTES lpMutexAttributes,// обычно NULL
__in BOOL bInitialOwner, // обычно FALSE
__in_opt LPCTSTR lpName);
```

Перед входом в критический участок поток должен проверить возможность входа с помощью вызова функции

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

а в конце участка – сообщить ОС о завершении выполнения критического участка с помощью функции

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

Удаление объекта ядра Mutex выполняет функция

```
BOOL WINAPI CloseHandle(__in HANDLE hObject);
```

1.3 Семафоры

Семафор – объект ядра, используемый как универсальное средство управления ресурсами. Как правило, семафор используется для учета неразделяемых ресурсов. В данной работе семафор должен использоваться для синхронизации двух конкурирующих потоков одного приложения, поэтому создается безымянный объект.

Семафор создается с помощью функции [1]

HANDLE CreateSemaphore(
LPSECURITY_ATTRIBUTES lpSemaphoreAttributes, // обычно равен NULL
LONG lInitialCount, // начальное значение счетчика свободных ресурсов
LONG lMaximumCount, // максимальное количество свободных ресурсов
LPCTSTR lpName // имя семафора - обычно NULL);

В начале критического участка вызывается функция

DWORD WINAPI WaitForSingleObject(__in HANDLE hHandle,
__in DWORD dwMilliseconds);

а в конце – функция BOOL ReleaseSemaphore(
HANDLE hSemaphore,
LONG lReleaseCount,
LPLONG lpPreviousCount);

Удаление объекта ядра Semsphore выполняет функция

BOOL WINAPI CloseHandle(__in HANDLE hObject);

1.4 События

Событие **с автоматическим сбросом** создается с помощью функции

HANDLE CreateEvent(
LPSECURITY_ATTRIBUTES
lpEventAttributes, // обычно равен NULL
BOOL bManualReset, // тип события - manual-reset event - False
BOOL InitialState, // начальное состояние initial state
LPTSTR lpName // имя события - обычно равен NULL);

В начале критического участка вызывается функция

DWORD WINAPI WaitForSingleObject (__in HANDLE hHandle,

__in DWORD dwMilliseconds);
а в конце – функция BOOL SetEvent (HANDLE hEvent);
для освобождения критического участка (события).

Удаление объекта ядра Event выполняет функция
BOOL WINAPI CloseHandle(__in HANDLE hObject);

2. МЕТОДИКА ВЫПОЛНЕНИЯ

1. Ознакомиться с приведенным текстом консольного приложения, в котором создаются 2 потока одного процесса, выполняющие увеличение глобальной переменной. Для синхронизации доступа конкурирующих потоков к переменной используется неименованный объект ядра – mutex.

```
#include "stdafx.h"
#include <windows.h>
#include <iostream>
using namespace std;
// объявление глобальных переменных
int global=100; // глобальная переменная, значение которой
 // увеличивают потоки
DWORD tid1, tid2, res;
HANDLE ht1, ht2;
HANDLE hm;
// функция первого потока
DWORD WINAPI ThreadProc1( LPVOID lpParameter )
{
int i, j;
SetThreadPriority (ht1,THREAD_PRIORITY_TIME_CRITICAL);
```

```
for (j=1; j <= 12; j++)
  {
WaitForSingleObject(hm,INFINITE);
// начало критического участка
i=global;
i++;
Sleep (1);
global =i;
// конец критического участка
printf ( "%4s %4d \n", " 1 th", i );
ReleaseMutex(hm);
  }
return 0;
}
// функция второго потока
DWORD WINAPI ThreadProc2 (LPVOID lpParameter)
{
int i, j;
for (j=1; j <= 12; j++)
{
WaitForSingleObject (hm, INFINITE);
// начало критического участка
i=global;
i++;
Sleep (1);
global =i;
// конец критического участка
  printf ( "%4s %4d %4d \n", " 2 th", i, j );
ReleaseMutex(hm);
}
```

```
return 0;
}

int _tmain(int argc, _TCHAR* argv[])
{
HANDLE msh[2];
int m; // основной поток приложения
hm = CreateMutex(NULL,FALSE,NULL);
ht1 = CreateThread(NULL,0,&ThreadProc1,NULL,0,&tid1);
ht2 = CreateThread(NULL,0,&ThreadProc2,NULL,0,&tid2);
msh[0] = ht1;
msh[1] = ht2;
// ожидание завершения работы потоков в течение 100 мс
if (WaitForMultipleObjects(2,msh,TRUE,100)==WAIT_TIMEOUT)
  printf ("over time");
return 0;
}
```

2. Выполнить приложение и убедиться в том, что после его выполнения значение переменной global будет равно 124.

3. Закомментировать операторы, выполняющие вызов функций

```
WaitForSingleObject (hm, INFINITE);
ReleaseMutex(hm);
```

(отключить синхронизацию).

4. Выполнить приложение и убедиться в том, что после его выполнения значение переменной global будет менее 124. Объяснить причину изменений.

5. Закомментировать вызовы функций Sleep (1).

6. Выполнить приложение и убедиться в том, что после его выполнения значение переменной global изменилось. Объяснить причину изменений

7. Завершить работу первого потока после выполнения 5 циклов. Для этого вставить в цикл функции первого потока строку

if (j==5) TerminateThread(GetCurrentThread(),0);

8. Выполнить приложение и убедиться в том, что первый поток выполнился 5 раз, а второй -12 раз. Объяснить причины такого поведения потоков.

9. Выполнить индивидуальные задания для бригад согласно табл. 3.

Таблица 3

Индивидуальные задания для бригад

№№ бригад	Средства синхронизации потоков
1, 2	Событие с автоматическим сбросом
3, 4	Критическая секция
5, 6	Семафор
7,8, 9,10	Событие с ручным сбросом

Изменить текст исходного приложения, удалив из него обращения к средству синхронизации Mutex и добавив указанные в табл. 3 средства синхронизации (согласно вариантам). Затем повторно выполнить пункты 2 – 8.

10. Используя указанные в табл. 3 средства синхронизации (в соответствии с номером бригады), создать тупик с участием двух потоков, использующих по два неразделяемых ресурса каждый. Первый поток должен занять первый ресурс, затем второй ресурс. Далее этот поток должен освободить ресурсы в любом порядке.

Второй поток должен занять сначала второй ресурс, затем первый. После этого он должен освободить ресурсы.

11. Используя функции распознавания тупиковых ситуаций WCT [1, 3], проверить возможность распознавания созданной тупиковой ситуации.

3. СОДЕРЖАНИЕ ОТЧЕТА

Отчет о работе должен содержать:

1. тексты разработанных в п.1, 3, 7 и 9-11 приложений;
2. письменные ответы на вопросы п.п. 4, 6, 8;
3. письменный ответ на вопрос: в чем состоит отличие поведения разработанных приложений, использующих для доступа потоков к критическим участкам мьютексы, критические секции, семафоры и события при досрочном завершении одного из потоков?

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. В чем состоит отличие критического участка от критической секции?
2. Синхронизация потоков с помощью критических секций.
3. Синхронизация потоков с помощью мьютексов.
4. Синхронизация потоков с помощью семафоров.
5. Синхронизация потоков с помощью событий.
6. В чем состоит отличие поведения разработанных приложений, использующих для доступа потоков к критическим участкам мьютексы, критические секции, семафоры и события?
7. Функции Win32, использованные при выполнении работы.
8. Понятие тупика двух потоков.
9. Возможности функций WCT по распознаванию тупиковых ситуаций.

ЛАБОРАТОРНАЯ РАБОТА № 4

УПРАВЛЕНИЕ ВИРТУАЛЬНОЙ ПАМЯТЬЮ

Цель работы - знакомство с функциями Win32 и структурами данных, используемыми для управления виртуальной памятью.

1 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.1 Механизмы управления виртуальной памятью

В Win32 четыре механизма управления виртуальной памятью [3]:

- виртуальная память, используемая для операций с большими массивами объектов или структур – размер элемента данных превышает размер страницы;
- кучи (heaps), применяемые для операций с большим количеством малых объектов – размер объекта меньше размера страницы;
- файлы, проецируемые в память - средства для операций с интенсивными потоками данных (обычно из файлов) и для обеспечения совместного доступа приложений к данным;
- AWE.

В данной работе рассматриваются первые ТРИ механизма.

1.2 Функции и структуры данных, используемые для управления виртуальной памятью

1. Сведения о конкретной платформе (совокупности аппаратно-программных средств) и параметрах виртуального адресного пространства предоставляет процедура:

VOID WINAPI GetSystemInfo (OUT LPSYSTEM_INFO lpSystemInfo);

Структура данных SystemInfo описана в файле winbase.h следующим образом:

```
typedef struct _SYSTEM_INFO { // sinf
union {
DWORD  dwOemId;
struct {
WORD wProcessorArchitecture;
```

```
WORD wReserved;
};
};
DWORD  dwPageSize; // размер страницы
LPVOID lpMinimumApplicationAddress; //мин адрес доступного
адресного пространства
LPVOID lpMaximumApplicationAddress; //макс адрес доступного
адресного пространства
DWORD dwActiveProcessorMask;
DWORD dwNumberOfProcessors; // число процессоров
WORD wProcessorArchitecture;
DWORD dwProcessorType;
DWORD dwAllocationGranularity; // гранулярность размещения
регионов адресного пространства
WORD wProcessorLevel;
WORD wProcessorRevision;
} SYSTEM_INFO;
```

Поле wProcessorArchitecture указывает архитектуру процессора. Значение 0 соответствует архитектуре Intel.

Поле wProcessorLevel определяет уровень процессора.

Поле wProcessorRevision указывает версию процессора.

Поле dwAllocationGranularity определяет гранулярность размещения регионов в адресном пространстве.

Пример 4. Вызов функции GetSystemInfo() для получения размера страницы виртуальной памяти

```
{
SYSTEM_INFO sysinf;
      GetSystemInfo(&sysinf);
      cout<<sysinf.dwPageSize;
 }
```

2. Сведения о ресурсах памяти и ее текущем состоянии можно получить с помощью функции

VOID GlobalMemoryStatus (IN OUT LPMEMORYSTATUS lpbuffer);

Параметр функции является указателем на структуру типа MEMORYSTATUS, назначение полей которой приведено в табл. 1 [1].

Таблица 4

Назначение полей этой структуры MEMORYSTATUS

Наименование поля	Содержимое поля
dwLength	Размер структуры в байтах
dwMemoryLoad	Оценка занятости системы управления памятью(0-100)
dwTotalPhys	Общий размер физической памяти RAM-памяти в байтах
dwAvailPhys	Общий размер физической памяти RAM-памяти в байтах, доступной для выделения
dwTotalPageFile	максимальное количество байтов, которое может содержаться в страничном файле на жестком диске (или дисках)
dwAvailPageFile	максимальное количество байтов, которое может быть передано процессу из страничного файла
dwTotalVirtual	количество байтов в адресном пространстве, принадлежащих лично данному процессу
dwAvailVirtual	суммарный объем всех свободных регионов в адресном пространстве процесса, вызывающего процедуру GlobalMemoryStatus

Перед вызовом процедуры необходимо занести в поле dwLength размер структуры MEMORYSTATUS в байтах с помощью функции sizeof().

Пример 5. Вызов функции GlobalMemoryStatus() для получения объема оперативной памяти

```
MemoryStatus MemInfo;
MemInfo.dwLength = sizeof (MemInfo);
```

GlobalMemoryStatus (&MemInfo);

printf (" Размер ОП %d Кбайт ", MemInfo.dwTotalPhys / 1024);

3. Для запроса информации о состоянии области виртуальной памяти текущего процесса (размер, тип памяти, атрибуты защиты) служит функция VirtualQuery, которая имеет следующий прототип:

DWORD VirtualQuery (

LPCVOID lpAddress, // адрес области

PMEMORY_BASIC_INFORMATION lpBuffer, // буфер для информации о состоянии области

DWORD dwLength // длина буфера

);

При вызове функции первый параметр должен указывать на область виртуальной памяти, информацию о которой нужно получить. При этом размер области определяется количеством последовательных виртуальных страниц, имеющих одинаковые атрибуты (состояние и тип защиты). Параметр lpBuffer указывает на структуру типа MEMORY_BASIC_INFORMATION, в которую функция VirtualQuery поместит информацию об указанной параметром lpAddress области виртуальной памяти. Параметр dwLength должен содержать длину структуры, на которую указывает параметр lpBuffer.

Функция возвращает действительное количество байтов, записанных в структуру по адресу lpBuffer. Если возвращено нулевое значение, информация о запрошенном участке НЕ ПОЛУЧЕНА.

Структура MEMORY_BASIC_INFORMATION состоит из следующих полей:

typedef struct _MEMORY_BASIC_INFORMATION { // mbi

PVOID BaseAddress; // базовый адрес региона

PVOID AllocationBase; // адрес размещения

DWORD AllocationProtect; // начальное значение защиты

DWORD RegionSize; // размер региона в байтах

DWORD State; // состояние региона (свободен, занят, зарезервирован)
DWORD Protect; // текущее значение защиты
DWORD Type; // тип страниц региона
} MEMORY_BASIC_INFORMATION;
typedef MEMORY_BASIC_INFORMATION *PMEMORY_BASIC_INFORMATION;

Назначение полей этой структуры приведено в табл. 5 [1].

Таблица 5

Назначение полей структуры MEMORY_BASIC_INFORMATION

Наименование поля	Содержимое поля
BaseAddress	Базовый адрес области виртуальной памяти - значение параметра lpAddress, округленное до значения, кратного размеру страницы
AllocationBase	Базовый адрес распределенной памяти - базовый адрес региона, включающего адрес запроса
AllocationProtect	Атрибут защиты региона
RegionSize	Суммарный размер (в байтах) группы страниц, начинающихся с базового адреса и имеющих те же атрибуты защиты, состояние и тип, что и страница, обнаруженная по адресу lpAddress
State	Указывает состояние (MEM_FREE, MEM_RESERVE, MEM_COMMIT) всех смежных страниц, имеющих те же атрибуты защиты, состояние и тип, что и страница, расположенная по адресу lpAddress
Protect	Содержит атрибут защиты (PAGE_*), общий для всех смежных страниц, имеющих те же атрибуты защиты, состояние и тип, что и страница, расположенная по адресу lpAddress
Type	Тип страниц в области виртуальной памяти

Поле Type может принимать одно из следующих значений:

– MEM_IMAGE – исполняемый код
– MEM_MAPPED –файл, проецируемый в память
– MEM_PRIVATE – память, принадлежащая процессу

Сканируя виртуальное адресное пространство в диапазоне от минимального до максимального адреса, можно построить карту виртуальной памяти процесса.

Более широкими возможностями обладает функция VirtualQueryEx (), отличающаяся от предыдущей тем, что дает возможность получить информацию о виртуальном адресном пространстве любого процесса, для которого известен его описатель.

Пример 6. Использование функции VirtualQuery () для просмотра виртуального адресного пространства текущего процесса и вывода адресов и размеров зарезервированных регионов.

```
MEMORY_BASIC_INFORMATION mbi;
DWORD pb; // базовый адрес
pb = 4096;
while (VirtualQuery (LPCVOID(pb), &mbi, sizeof
(MEMORY_BASIC_INFORMATION)) == sizeof (mbi))
{
      if (mbi.State == MEM_RESERVE)
      printf ("%8x %8d\n", mbi.BaseAddress, mbi.RegionSize);
      pb = pb + mbi.RegionSize;
}
```

4. Для резервирования региона в виртуальном адресном пространстве (первый механизм управления памятью) используется функция

```
    LPVOID VirtualAlloc (
LPVOID lpAddress, // начальный адрес резервируемого региона виртуальной памяти,
SIZE_T dwSize, // размер региона
DWORD flAllocationType, // вид операции
DWORD flProtect // тип защиты доступа
    );
```

Первый параметр определяет положение резервируемого региона в адресном пространстве. Если параметр равен NULL, система определяет положение региона самостоятельно. Регионы всегда резервируются на границе 64К (гранулярность размещения).

Если запрос выполнен, функция возвращает базовый адрес зарезервированного региона.

Второй параметр задает размер резервируемого региона в байтах. Заданный размер увеличивается до величины, кратной размеру страницы.

Третий параметр определяет вид операции – MEM_RESERVE – зарезервировать регион, MEM_COMMIT – передать региону физическую память (сначала нужно зарезервировать регион, затем передать ему память). В принципе можно одновременно зарезервировать регион и передать ему физическую память – например

```
VirtualAlloc (NULL, 100*1024, MEM_RESERVE | MEM_COMMIT,
                                PAGE_READWRITE);
```

Параметр MEM_TOP_DOWN указывает системе на необходимость резервирования региона в верхних адресах виртуального адресного пространства с целью уменьшения фрагментации виртуальной памяти.

Последний параметр определяет атрибут защиты.

5. Для возврата физической памяти, спроецированной на регион, или освобождения всего региона адресного пространства используется функция

```
BOOL VirtualFree ( LPVOID lpAddress,    // адрес региона
DWORD dwSize,              // размер региона
DWORD dwFreeType        // тип операции освобождения
);
```

В простейшем случае использования этой функции – для освобождения всей переданной региону физической памяти – в параметр lpAddress необходимо передать базовый адрес региона, в

параметр dwSize=0, так как системе известен размер региона. В третьем параметре следует передать MEM_RELEASE – это приведет к возврату системе всей физической памяти, спроецированной на регион, и освобождению самого региона.

1.3 КУЧИ

Процессу разрешено создавать несколько куч, являющихся частью его адресного пространства.

Куча – это регион зарезервированного адресного пространства [3]. Первоначально большей его части физическая память не передается. После того, как программа занимает эту область под данные, диспетчер, управляющий кучами, передает ей страницы физической памяти. При освобождении страниц в куче диспетчер возвращает физическую память системе.

Одна куча предоставляется процессу *по умолчанию*. Ее описатель возвращает функция HANDLE GetProcessHeap (VOID);

Дополнительные кучи создаются вызовом функции

```
HANDLE HeapCreate(DWORD flOptions, // флаг операции
DWORD dwInitialSize,      // начальный размер кучи
DWORD dwMaximumSize       // максимальный размер кучи
);
```

Первый параметр определяет характер операций, выполняемых над кучей. По умолчанию (значение 0) действует принцип последовательного доступа к куче (как к критическому участку).

Второй параметр определяет количество байт, первоначально передаваемых куче.

Третий параметр указывает максимальный размер кучи. Может быть равен 0, в этом случае система резервирует регион,

самостоятельно, определяя его размер, и при необходимости расширяет его до максимально возможного объема.

Выделение блока памяти из кучи выполняет функция

```
LPVOID HeapAlloc(
HANDLE hHeap,      // описатель блока
DWORD dwFlags,     // флаги управления доступом
DWORD dwBytes      // количество размещаемых байтов
);
```

Первый параметр – описатель кучи, возвращаемый функциями GetProcessHeap() или HeapCreate()

Второй параметр указывает на характер выделения памяти. При его значении, равном HEAP_ZERO_MEMORY = 8, выделяемый блок заполняется нулями.

Третий параметр определяет число выделяемых в куче байт.

Освобождение блока памяти выполняет функция

```
BOOL HeapFree(
HANDLE hHeap,        // handle to the heap
DWORD dwFlags,       // heap freeing flags
LPVOID lpMem // pointer to the memory to free
);
```

Уничтожение кучи выполняет функция

```
BOOL HeapDestroy(HANDLE hHeap);
```

1.4. Файлы, проецируемые в память

Проецируемые в память файлы позволяют резервировать регион адресного пространства и передавать ему физическую память. При использовании проецируемых файлов физическая память не выделяется из системного страничного файла, а берется из файла, уже находящегося на диске. Как только файл спроецирован в память, к нему

можно обращаться так, будто он целиком в нее загружен [4]. Описанный механизм управления памятью применяется для:

– загрузки и исполнения EXE- и DLL-файлов;

– доступа к файлу данных, размещенному на диске;

– обеспечения совместного доступа разных процессов к общим данным.

Для подготовки к использованию файлов, проецируемых в память, нужно выполнить три операции:

1. Создать или открыть объект ядра "файл".
2. Создать объект ядра "проецируемый файл", чтобы сообщить системе размер файла и способ доступа к нему.
3. Указать системе, как спроецировать в адресное пространство процесса созданный в п.2 объект - целиком или только его часть.

Для завершения работы с файлом, проецируемым в память, следует выполнить три операции:

1. Сообщить системе об отмене проецирования на адресное пространство процесса объекта ядра "проецируемый файл".
2. Закрыть этот объект.
3. Закрыть объект ядра "файл".

2. МЕТОДИКА ВЫПОЛНЕНИЯ

1. Написать ДВА приложения командной строки, реализующие следующие функции - см. табл. 6:

Таблица 6

Индивидуальные задания для бригад

Номер бригады	Содержание задания
1, 3	1. Вывод на экран информации о параметрах и состоянии виртуальной памяти (размер страницы, сведения о процессоре), количество и общий объем свободных регионов выполняющегося процесса. Вывод на экран карты виртуальной памяти выполняющегося процесса в диапазоне 0-1

	Гб (адрес, размер региона, статус). 2. Операции с памятью – резервирование региона в ВАП и передача физической памяти, размер региона (в байтах) задавать путем ввода с клавиатуры. Проанализировать изменение параметров приложения с помощью диспетчера задач.
Номер бригады	Содержание задания
2, 4	1. Вывод на экран информации о параметрах и состоянии виртуальной памяти (гранулярность, размер файла подкачки (страничного файла) и суммарный объем всех зарезервированных регионов в адресном пространстве процесса), количество зарезервированных регионов выполняющегося процесса. Вывод на экран карты виртуальной памяти выполняющегося процесса в диапазоне 1-2 Гб (адрес, размер региона, атрибут защиты). 2. Операции с памятью – резервирование региона в верхних адресах ВАП, размер региона (в байтах) задавать путем ввода с клавиатуры. Проанализировать изменение параметров приложения с помощью диспетчера задач.
5, 6	1. Вывод на форму информации о параметрах и состоянии виртуальной памяти (гранулярность, диапазон адресов ВАП, занятость системы управления памятью), количество и общий объем зарезервированных регионов выполняющегося процесса. 2. Операции с памятью – резервирование региона в ВАП и передача физической памяти, размер региона (в байтах) задавать путем ввода с клавиатуры. Проанализировать изменение параметров приложения с помощью диспетчера задач.
7, 8	1. Вывод на форму информации о параметрах и состоянии виртуальной памяти (гранулярность, диапазон адресов ВАП, занятость системы управления памятью), количество и общий объем зарезервированных регионов выполняющегося процесса (при нажатии на кнопку). 2. Операции с кучами – создать новую кучу процесса и в созданной куче разместить N блоков по K байт в каждом блоке. Значения N и K

	вводить с клавиатуры. Проанализировать изменение параметров приложения с помощью диспетчера задач.
Номер бригады	Содержание задания
9, 10	1. Вывод на форму информации о параметрах и состоянии виртуальной памяти (гранулярность, диапазон адресов ВАП, занятость системы управления памятью), количество и общий объем зарезервированных регионов выполняющегося процесса. 2. Операции с памятью – резервирование региона памяти в верхних адресах ВАП, размер региона (в байтах) задавать путем ввода с клавиатуры, и передавать память с задержкой 10 сек. Проанализировать изменение параметров приложения с помощью диспетчера задач.

Для анализа изменений при резервировании памяти следует использовать имеющийся в Диспетчере задач Монитор ресурсов (раздел Память).

2. Проверить работу приложений. Обратить внимание на изменение состояния виртуального адресного пространства процесса после резервирования и передачи региону физической памяти.

3. Ответить на контрольные вопросы.

3. СОДЕРЖАНИЕ ОТЧЕТА

Отчет о работе должен содержать:

1. Тексты приложений, разработанных в соответствии с п.1;
2. Выводы.

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Функции, используемые для получения общей информации о состоянии виртуальной памяти.
2. Функции, используемые для получения подробной информации о состоянии виртуальной памяти.
3. Функции управления виртуальной памятью.
4. Функции размещения информации в куче.
5. Функции создания и удаления дополнительных куч процесса.
6. Необходимость создания дополнительных куч процесса.
7. Использование куч при разработке приложений.
8. Что такое гранулярность выделения памяти?
9. Каков диапазон адресов виртуального адресного пространства, возвращаемый функцией VirtualQuery()?

ЛАБОРАТОРНАЯ РАБОТА № 5

ДИНАМИЧЕСКИ ЗАГРУЖАЕМЫЕ БИБЛИОТЕКИ (DLL)

Цель работы - освоение методики создания и использования динамически компонуемых библиотек (DLL) средствами системы VS10.

1. КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.1. DLL и их роль в Win32

DLL составляют основу архитектуры всех версий операционных систем (ОС) MS Windows. Области применения DLL:

– CPL - файлы (компоненты панели управления);

– серверы OLE и т.д.

В Win32, получив команду загрузить программу, ОС создает процесс, который располагает 2 Гбайт виртуальной памяти, в которой выполняется программа. После создания процесса создается объект отображения файла, который отображает выполняемый файл в адресное пространство процесса. После отображения выполняемого файла в память ОС по заданному смещению от начала образа переходит к адресу, в котором записаны имена всех DLL, используемых данным EXE-модулем. Все DLL разыскиваются в системе и отображаются в тот же процесс. При этом для каждой DLL создается свой объект отображения файла. Если некоторые DLL обращаются, в свою очередь, к другим DLL, последние также отображаются в адресное пространство программы.

После прекращения выполнения программы образ каждого DLL-файла удаляется из памяти. Процесс продолжается до удаления образа EXE-файла и объекта главного процесса.

DLL могут использоваться совместно разными программами.

1.2. Создание проекта DLL

Для того чтобы создать DLL в Visual Studio 2012, необходимо: Файл - Создать - Проект - Приложение Win32 (консольное либо проект Win32). После выбора, ввести имя и нажать ОК, затем далее и выбрать Тип приложения "библиотека DLL". После этого открывается файл, в который необходимо ввести текст DLL модуля.

Пример 7 написания файла DLL.

Чтобы в результате компиляции была создана DLL, а не exe-файл, достаточно указать в настройках компиляции, что необходимо получить именно Dynamic Link Library. Для этого функцию надо объявить как экспортируемую из DLL. Делается это с помощью добавления слева от функции модификатора extern «C».

Самый простой вариант кода:

```
#include "stdafx.h"
#include <iostream>
extern "C" __declspec(dllexport) void LetterList()
{ std::cout << "This function was called from LetterList() " << std::endl; }
extern "C" __declspec(dllexport) int PutInt(int param) { return param; }
```

Одна функция выводит сообщение, другая функция возвращает принятое значение. После компиляции DLL будут получены два файла: ***.dll и ***.lib.

Затем в этом же проекте необходимо создать файл заголовка (***.h), где описываются прототипы функций, находящихся в этом DLL файле (Меню: Проект/Добавить новый элемент).

Пример 8 написания файла заголовка (Sozd.h):

```
void LetterList();
int PutInt(int param);
```

Этот файл следует добавить в программу, в которой планируется использовать созданный DLL файл.

1.3. Вызов функций из DLL

В результате выполнения описанных выше действий будут получены откомпилированный DLL файл, файл Lib и файл заголовка. Эти файлы следует скопировать в программу, использующую созданный DLL файл. Для этого необходимо создать новый проект: Файл - Создать - Проект - Консольное Приложение Win32 - Готово. В этот проект нужно добавить информацию о месте расположении файлов exe, dll и lib, а файл заголовка вставить среди заголовочных файлов проекта. В полученный файл вписываем код программы, в котором вызываются функции из DLL.

Пример 9. Вызов функций PutInt и LetterList

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

```
# include "Sozd.h"
#include "conio.h"
#include "iostream"
int _tmain(int argc, _TCHAR* argv[])
 {
        int x = PutInt(5);
        LetterList();
        std::cout<<x;
        _getch();
        return 0;
}
```

1.4. Подключение dll файла в Visual Studio

В Visual Studio подключение dll файла выполняется с помощью меню в следующем порядке: Проект - Свойства - Компоновщик - Ввод - Дополнительные зависимости. Туда добавляется путь к файлу .lib.

1.5. Экспортирование

Записывая DLL на диск, компоновщик записывает также имена всех экспортируемых функций или процедур вместе с их смещениями в образе файла [3]. Когда во время выполнения программы DLL отображается в адресное пространство загружающего процесса, загрузчик находит внутри образа файла DLL имена и адреса всех функций и процедур, необходимых загружающей программе или DLL. Если какую-либо из процедур или функций найти не удается, загрузка DLL прекращается.

К экспортируемым DLL функциям или процедурам можно обращаться из ОС, прикладной программы или другой DLL. Сами эти функции могут служить интерфейсом связи с объектами внутри DLL, либо независимо выполнять нужные операции. Простейшее применение

DLL - объединение некоторого числа функций сходного назначения, которые затем могут использоваться разными программами.

1.6. Загрузка DLL

Существует два способа загрузки DLL: с ***явной*** и ***неявной компоновкой***.

При ***неявной компоновке*** функции загружаемой DLL добавляются в секцию импорта вызывающего файла. При запуске такого файла загрузчик операционной системы анализирует секцию импорта и подключает все указанные библиотеки. Ввиду своей простоты этот способ пользуется большой популярностью, однако неявной компоновке присущи определенные недостатки и ограничения:

1. все подключенные DLL загружаются *всегда*, даже если в течение всего сеанса работы программа ни разу не обратится ни к одной из них;
2. если хотя бы одна из требуемых DLL отсутствует, загрузка исполняемого файла прерывается сообщением "*Dynamic link library could not be found*";
3. поиск DLL происходит в следующем порядке: в каталоге, содержащем вызывающий файл; в текущем каталоге процесса; в системном каталоге %Windows%System%; в основном каталоге %Windows%; в каталогах, указанных в переменной PATH. Задать другой путь поиска невозможно.

Это наиболее простой метод подключения DLL к программе. Все, что нужно - это передать компоновщику имя библиотеки импорта, чтобы он использовал ее в процессе сборки.

Явная компоновка устраняет все эти недостатки ценой некоторого усложнения кода. Программисту самому придется позаботиться о загрузке DLL и подключении экспортируемых функций. Зато явная компоновка позволяет подгружать DLL по мере необходимости и дает

программисту возможность самостоятельно обрабатывать ситуации с отсутствием DLL.

При явном подключении DLL программист должен сам позаботиться о загрузке библиотеки перед ее использованием. Для этого используется функция LoadLibrary, которая получает имя библиотеки и возвращает ее дескриптор. Дескриптор необходимо сохранить в переменной, так как он будет использоваться всеми остальными функциями, предназначенными для работы с DLL.

Пример 10 использования явной загрузки DLL.

```
HMODULE hLib;
hLib = LoadLibrary(L"MyDll.dll");
if (hLib)
{ cout << "Library load." << endl;
MYPROC MyFunction = (MYPROC)GetProcAddress(hLib, "MyFunction");
// Загружаем функцию
 if (MyFunction)
{
 cout << "Function load." << endl;
MyFunction(); // Вызов функции из dll
}
FreeLibrary(hLib);
}
```

2. МЕТОДИКА ВЫПОЛНЕНИЯ

1. Создать DLL, содержащую набор функций в соответствии с таблицей 7.

2. Написать приложение, использующее функции (ю) из разработанной библиотеки. Библиотеку подключить, используя неявную компоновку.
3. Написать приложение, использующее функции (ю) из разработанной библиотеки. Библиотеку подключить, используя явную компоновку

Таблица 7

Варианты заданий

Номер бригады	Набор функций, s, s1 и s2 – аргументы функции
1	s=s1+s2 (конкатенация строк); y=sin(x)*cos(x);
2	s=s1+s2+s1 (конкатенация строк); y=min (x1, x2, x3); x1, x2, x3- целые числа
3	нахождение количества вхождений строки s1 в строку s2
4	s=s1+s1+s1… - конкатенация аргумента s1 N раз
5	длина s1+s2 (длина конкатенации строк); m=N! (факториал аргумента)
6	количество слов в строке s
7	количество гласных букв в строке s
8, 9, 10	s1 – строка символов, s2 – класс строки: 1– идентификатор; 2 – целое число; 3 –произвольная строка

3. СОДЕРЖАНИЕ ОТЧЕТА

Отчет о работе должен содержать:

1. текст разработанной DLL;

2. тексты приложений, использующих DLL с помощью неявной и явной компоновок;

3. выводы.

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Области применения DLL.
2. Основные DLL, используемые приложениями MS Windows.
3. Создание DLL средствами Visual Studio 2010.
4. Структура DLL.
5. Достоинства и недостатки неявного способа загрузки DLL.

6. Достоинства и недостатки явного способа загрузки DLL.
7. Программная реализация неявного способа загрузки DLL.
8. Программная реализация явного способа загрузки DLL.
9. Функции Win32, используемые при явной загрузке DLL.

ЛАБОРАТОРНАЯ РАБОТА № 6

ОБМЕН ДАННЫМИ МЕЖДУ ПРОЦЕССАМИ

Цель работы - практическое знакомство со средствами передачи данных между процессами, (Interprocess Communications - IPC), выполняющимися на одном компьютере.

1. КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.1. Способы передачи данных между процессами

Под обменом данными между параллельными процессами понимается пересылка данных от одного потока к другому потоку, предполагая, что эти потоки выполняются в контекстах разных процессов. Поток, который посылает данные другому потоку, называется *отправителем.* Поток, который получает данные от другого потока, называется *адресатом* или *получателем.*

Если потоки выполняются в одном процессе, то для обмена данными между ними можно использовать глобальные переменные и средства синхронизации потоков. Сложнее обстоит дело в случае, если потоки выполняются в разных процессах — потоки не могут обращаться к общим переменным и для обмена данными между ними

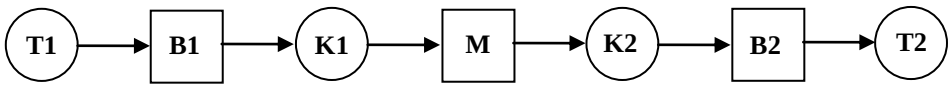
существуют специальные средства операционной системы. В этом случае для обмена данными между процессами создается *канал передачи данных,* организация которого схематически показана на рис. 2.

Кратко поясним устройство и работу канала передачи данных. Канал данных включает входной и выходной буферы памяти, потоки ядра операционной системы и общую память, доступ к которой имеют оба потока ядра. Канал передачи данных работает следующим образом:

– первый поток ядра операционной системы читает данные из входного буфера *B1* и записывает их в общую память М;
– второй поток ядра читает данные из общей памяти *M* и записывает их в буфер В2;

Пользовательские потоки *T1* и *T2* посредством вызова функций ядра операционной системы имеют доступ к буферам *B1* и *B2* соответственно. Поэтому пересылка данных из потока *T1* в поток *T2* происходит следующим образом:

– пользовательский поток *T1* записывает данные в буфер *B1,* используя специальную функцию ядра операционной системы;
– поток *K1* ядра операционной системы читает данные из буфера *B1* и записывает их в общую память *M*;
– поток *K2* ядра операционной системы читает данные из общей памяти *M* и записывает их в буфер *B2*;
– пользовательский поток *T2* читает данные из буфера *B2*.



Р и с. 2 - Схема канала передачи данных

Т1, Т2 — пользовательские потоки
В1, В2 — буферы
К1, К2 — потоки ядра операционной системы
М — общая память

Из рис. 2 видно, что в любом случае обмен данными может быть организован только через цепочку взаимодействующих потоков,

которые обмениваются между собой данными через общую только для них память.

На схему канала, показанную на рис. 2, можно взглянуть и шире. Она также подходит для организации канала передачи данных по сети.

Канал передачи данных между процессами можно организовать программным способом без поддержки операционной системы. Для этого нужно потоки ядра операционной системы и общую память, используемую для обмена данными, заменить файлом. В результате получим простейший канал передачи данных между потоками, выполняющимися в контекстах разных процессов. Таким образом, обмен данными между процессами через общий файл представляет собой организацию простейшего канала передачи данных между процессами.

При обмене данными между параллельными процессами различают два способа передачи данных:

– потоком;

– сообщением.

Если данные передаются непрерывной последовательностью байтов, такая пересылка данных называется *передача данных потоком.* В этом случае общая память *M,* доступная потокам ядра операционной системы, может и отсутствовать, а пересылка данных выполняется одним потоком ядра непосредственно из буфера B1 в буфер B2.

Если данные пересылаются группами байтов, то такая группа байтов называется *сообщением,* а сама пересылка данных называется *передачей данных сообщениями.*

1.2. Виды связей между процессами

Прежде чем передавать данные между процессами, нужно установить между этими процессами связь. Связь между процессами

может устанавливаться как на физическом (или аппаратном), так и на логическом (или программном) уровнях. С точки зрения направления передачи данных различают следующие виды связей:

– *полудуплексная связь,* т. е. данные по этой связи могут передаваться только в одном направлении;
– *дуплексная связь,* т. е. данные по этой связи могут передаваться в обоих направлениях.

Теперь, предполагая, что рассматриваются только полудуплексные связи, определим возможные топологии связей. Под *топологией связи* будем понимать конфигурацию связей между процессами-отправителями и адресатами. Различают следующие виды связей:

– 1 -> 1 – между собой связаны только два процесса;
– 1 -> *N* – один процесс связан с *N* процессами;
– N-> 1 – каждый из *N* процессов связан с одним процессом;
– N-> *M* – каждый из N процессов связан с каждым из *M* процессов.

1.3 Синхронный и асинхронный обмен данными

При передаче данных различают синхронный и асинхронный обмен данными. Если поток-отправитель, отправив сообщение функцией send, блокируется до получения этого сообщения потоком-адресатом, то такое отправление сообщения называется *синхронным.* В противном случае отправление сообщения называется *асинхронным.* Если поток-адресат, вызвавший функцию receive, блокируется до тех пор, пока не получит сообщение, то такое получение сообщения называется *синхронным.* В противном случае получение сообщения называется *асинхронным.*

Обмен сообщениями называется *синхронным,* если поток-отправитель синхронно передает сообщения, а поток-адресат синхронно принимает эти сообщения. В противном случае обмен сообщениями

называется *асинхронным.* Предполагается, что обмен данными потоком всегда происходит синхронным образом, т. к. в этом случае между отправителем и адресатом устанавливается непосредственная связь.

1.4 Использование буфера обмена

Буфер обмена (clipboard) в основном используется для удобства пользователей и редко используется как метод IPC. Канал передачи данных между процессами с использованием буфера обмена следует рассматривать как организованный программным способом без поддержки операционной системы.

1.4.1 Копирование информации в буфер обмена

Процесс копирования информации в буфер обмена осуществляется следующим образом:

a. Открыть буфер обмена функцией BOOL OpenClipboard(hWndNewOwner);

b. hWndNewOwner – дескриптор окна, которое связывается с открытым буфером обмена. Если этот параметр имеет значение ПУСТО (NULL), открытый буфер связан с текущей задачей.

c. Очистить буфер обмена функцией BOOL EmptyClipboard(VOID);

d. Вызвать функцию HANDLE SetClipboardData(UINT uFormat, HANDLE hMem); для каждого формата буфера обмена, которые поддерживает приложение.

uFormat задает формат буфера обмена. Этим параметром может быть зарегистрированный или любой стандартный формат буфера обмена. hMem - дескриптор данных в заданном формате. Этим параметром может быть значение ПУСТО (NULL), обозначая, что окно после запроса обеспечивает данные в заданном формате буфера обмена

(предоставляет формат). Если окно задерживает предоставление формата, оно должно обрабатывать сообщения WM_RENDERFORMAT и WM_RENDERALLFORMATS.

e. Закрыть буфер обмена функцией BOOL CloseClipboard(VOID);

1.4.2 Вставка данных из буфера обмена

Процесс вставки информации из буфера обмена осуществляется следующим образом:

a. Открыть буфер обмена функцией OpenClipboard.
b. Определить форматы данных, хранящихся в буфере обмена.
c. Получить хэндл данных нужного формата при помощи функции GetClipboardData.
d. Вставить копию данных в документ.
e. Владельцем хэндла, возвращенного GetClipboardData, остается все еще буфер обмена, поэтому приложение не должно его освобождать.
f. Закрыть буфер обмена функцией CloseClipboard.

1.5 Использование файлов, проецируемых в память

Способ передачи данных между приложениями - использование области в страничном файле, доступ к которой могут иметь несколько процессов – так же как и предыдущий, организует канал передачи данных между процессами программным способом.

Для создания области в страничном файле можно использовать способ проецирования файла. Проецируемый в память файл может применяться для совместного использования файла или памяти между двумя или несколькими процессами. Чтобы совместно использовать файл или память, все процессы должны использовать имя или дескриптор одного и того же объекта "проецируемый файл".

Чтобы совместно использовать файл, первый процесс создает или открывает файл, используя функцию CreateFile. Затем, он создает объект "проецируемый файл", используя функцию CreateFileMapping, определяя дескриптор файла и имя для объекта "проецируемый файл". Имена объектов "проецируемый файл" совместно используют одно и то же пространство имен. Поэтому функции CreateFileMapping и OpenFileMapping завершаются ошибкой, если они устанавливают имя, которое используется объектом другого типа.

HANDLE CreateFileMapping(HANDLE *hFile*, // дескриптор файла
LPSECURITY_ATTRIBUTES *lpAttributes*, // защита файла
DWORD *flProtect*, // атрибуты защиты файла
DWORD *dwMaximumSizeHigh*, // старшее слово размера файла
DWORD *dwMaximumSizeLow*, // младшее слово размера файла
LPCTSTR *lpName* // имя объекта
);
HANDLE OpenFileMapping(
DWORD *dwDesiredAccess*, // режим доступа
BOOL *bInheritHandle*, // флажок наследования
LPCTSTR *lpName* // имя объекта
);

Чтобы совместно использовать память, которая не связана с файлом, процесс должен использовать функцию CreateFileMapping, а вместо существующего дескриптора файла установить INVALID_HANDLE_VALUE в качестве параметра hFile. Соответствующий объект "проецируемый файл" получает доступ к памяти, который поддерживается системным файлом подкачки. Необходимо установить размер объекта больше нуля при установке в параметре hFile значения INVALID_HANDLE_VALUE при вызове функции CreateFileMapping. [3]

Самый простой способ получения дескриптора объекта "проецируемый файл", созданного первым процессом, другими процессами состоит в том, чтобы использовать функцию OpenFileMapping и установить имя объекта. Оно указывается как имя совместно используемой памяти (named shared memory). Если объект "проецируемый файл" не имеет имени, процесс должен получить его дескриптор через посредство наследования или дублирования.

Процессы, которые совместно используют файлы или память, должны создавать представления файла, используя функцию MapViewOfFile. Они должны координировать свой доступ, используя семафоры, мьютексы или события. Прототип функции имеет вид [1]:

LPVOID MapViewOfFile(
HANDLE *hFileMappingObject*, // дескрипт. объекта проецируемый файл
DWORD *dwDesiredAccess*, // режим доступа
DWORD *dwFileOffsetHigh*, // старшее DWORD смещения
DWORD *dwFileOffsetLow*, // младшее DWORD смещения
SIZE_T *dwNumberOfBytesToMap* // число отображаемых байтов
);
LPVOID MapViewOfFileEx(
HANDLE *hFileMappingObject*, // дескриптор отображаемого объекта
DWORD *dwDesiredAccess*, // режим доступа
DWORD *dwFileOffsetHigh*, // старшее DWORD смещения
DWORD *dwFileOffsetLow*, // младшее DWORD смещения
SIZE_T *dwNumberOfBytesToMap*, // число отображаемых байтов
LPVOID *lpBaseAddress* // начальный адрес
);

Совместно используемый объект "проецируемый файл" не будет разрушен до тех пор, пока все процессы, которые используют его, не закроют свои дескрипторы связанные с этим объектом, используя функцию CloseHandle.

1.6. Использование именованного канала

Именованным каналом называется объект ядра операционной системы, который обеспечивает передачу данных между процессами, выполняющимися на компьютерах одной локальной сети. Процесс, который создает именованный канал, называется сервером именованного канала. Процессы, которые связываются с именованным каналом, называются клиентами именованного канала. Характеристики именованных каналов:

– имеют имя, которое используется клиентами для связи с именованным каналом;
– могут быть как полудуплескные, так и дуплексные;
– передача данных может осуществляться как потоком, так и сообщением;
– обмен данными может быть как синхронным, так и асинхронным.

Порядок работы с именованными каналами:

– создание именованного канала сервером;
– соединение сервера с экземпляром именованного канала;
– соединение клиента с экземпляром именованного канала;
– обмен данными по именованному каналу;
– отсоединение сервера от экземпляра именованного канала;
– закрытие именованного канала клиентом и сервером.

Серверы назначают именованным каналам имена в соответствии с универсальными правилами именования. Формат имени канала выглядит так:

\\Сервер\Pipe\ИмяКанала

Элемент Сервер указывает компьютер, на котором работает сервер именованного канала. Элемент Pipe должен быть строкой “Pipe”, а ИмяКанала – уникальным именем, назначенным именованному каналу.

Для сервера, работающего на локальном компьютере, используется псевдоним локального компьютера \\.\

Для создания именованного канала Pipe следует использовать функцию CreateNamedPipe. Прототип этой функции имеет вид [1]:

```
HANDLE CreateNamedPipe (
LPCTSTR lpName, // адрес строки имени канала
DWORD dwOpenMode, // режим открытия канала
DWORD dwPipeMode, // режим работы канала
DWORD nMaxInstances, // максим. количество экземпляров канала
DWORD nOutBufferSize, // размер выходного буфера в байтах
DWORD nInBufferSize, // размер входного буфера в байтах
DWORD nDefaultTimeOut, // время ожидания в миллисекундах
LPSECURITY_ATTRIBUTES lpSecurityAttributes); // адрес
// переменной для атрибутов защиты
```

1.6.1 Создание сервером именованного канала

Для создания именованного канала сервер использует функцию CreateNamedPipe(). Пример вызова данной функции для создания серверной части канала на локальной машине:

```
hPipe = CreateNamedPipe ('\\.\PIPE\test', // Имя канала = 'test'.
PIPE_ACCESS_DUPLEX | // Двусторонний канал
FILE_FLAG_OVERLAPPED, // Асинхронный ввод-вывод
PIPE_WAIT | // Ожидать сообщений
PIPE_READMODE_MESSAGE | // Обмен в канале производится
// пакетами (сообщениями)
PIPE_TYPE_MESSAGE,
MAX_PIPE_INSTANCES, // Максимальное число экземпляров канала.
OUT_BUF_SIZE, // Размеры буферов чтения/записи. 0 – размер по
// умолчанию
```

IN_BUF_SIZE,
TIME_OUT, // Тайм-аут.
NULL); // Атрибуты безопасности.

Именованные каналы, работающие в режиме передачи сообщений, упрощают реализацию приемника, поскольку в этом случае число передач и приемов одинаково, а приемник, разом получая целое сообщение, не должен заботиться об отслеживании фрагментов сообщений.

При первом вызове CreateNamedPipe с указанием какого-либо имени создается первый экземпляр именованного канала с этим именем и задается поведение всех последующих экземпляров этого канала. Повторно вызывая CreateNamedPipe, сервер может создавать дополнительные экземпляры именованного канала, максимальное число которых указывается при первом вызове CreateNamedPipe.

1.6.2 Соединение сервера с клиентом

Создав минимум один экземпляр именованного канала, сервер вызывает функцию ConnectNamedPipe, после чего именованный канал позволяет устанавливать соединения с клиентами. Функция ConnectNamedPipe может выполняться как синхронно, так и асинхронно, и она не завершится, пока клиент не установит соединение через данный экземпляр именованного канала (или не возникнет ошибка).

В случае успешного завершения эта функция возвращает ненулевое значение.

BOOL ConnectNamedPipe(
HANDLE hNamedPipe, // идентификатор именованного канала
LPOVERLAPPED lpOverlapped); // адрес структуры OVERLAPPED

1.6.3 Соединение клиентов с именованным каналом

Прежде чем соединяться с именованным каналом, клиент должен определить, доступен ли какой-либо экземпляр этого канала для соединения. С этой целью клиент должен вызвать функцию WaitNamedPipe(). Если заранее известно, что сервер вызвал функцию ConnectNamedPipe(), функцию WaitNamedPipe() можно не вызывать и сразу подключаться к именованному каналу.

Для подключения к серверу клиенты именованного канала используют функцию CreateFile() или CallNamedPipe(), указывая при вызове имя созданного сервером канала. Если клиенту разрешен доступ к именованному каналу, он получает описатель, представляющий клиентскую сторону именованного канала, и функция ConnectNamedPipe(), вызванная сервером, завершается.

После того, как соединение по именованному каналу установлено, клиент и сервер могут использовать его для чтения и записи данных с помощью функций ReadFile() и WriteFile(). Именованные каналы поддерживают как синхронную, так и асинхронную передачу сообщений. Пример соединения:

```
HANDLE CreateFile(
LPCTSTR lpFileName, // адрес строки имени файла
DWORD dwDesiredAccess, // режим доступа
DWORD dwShareMode, // режим совместного использования файла
LPSECURITY_ATTRIBUTES lpSecurityAttributes // дескриптор защиты
DWORD dwCreationDistribution, // параметры создания
DWORD dwFlagsAndAttributes, // атрибуты файла
HANDLE hTemplateFile); // идентификатор файла с атрибутами
```

В случае успешного завершения эта функция возвращает дескриптор именованного канала.

1.6.4 Обмен данными по именованному каналу

Для обмена данными по именованному каналу используются функции ReadFile() и WriteFile().

Запись данных в открытый канал выполняется с помощью функции WriteFile, аналогично записи в обычный файл:

```
HANDLE hNamedPipe;
DWORD cbWritten;
char szBuf [256];
WriteFile(hNamedPipe, szBuf, strlen(szBuf) + 1, &cbWritten, NULL);
```

Через первый параметр функции WriteFile передается идентификатор реализации канала. Через второй параметр передается адрес буфера, данные из которого будут записаны в канал. Размер этого буфера указывается при помощи третьего параметра. Предпоследний параметр используется для определения количества байт данных, действительно записанных в канал. И, наконец, последний параметр задан как NULL, поэтому запись будет выполняться в синхронном режиме.

Для чтения данных из канала можно воспользоваться функцией ReadFile, например, так:

```
HANDLE hNamedPipe;
DWORD cbRead;
char szBuf[256];
ReadFile(hNamedPipe, szBuf, 512, &cbRead, NULL);
```

Данные, прочитанные из канала hNamedPipe, будут записаны в буфер szBuf, имеющий размер 512 байт. Количество действительно прочитанных байт данных будет сохранено функцией ReadFile в переменной cbRead. Так как последний параметр функции указан как NULL, используется синхронный режим работы без перекрытия.

Для разрушения экземпляра канала используется вызов

DisconnectNamedPipe (hPipe);

2. МЕТОДИКА ВЫПОЛНЕНИЯ

1. Ознакомиться с описаниями (MSDN) функций, упомянутых в тексте данного руководства и доступными примерами их использования.
2. Написать приложение, выполняющее занесение в буфер обмена текста и графики и приложение, получающее из буфера обмена находящиеся в нем данные. Для занесения в буфер новых данных можно использовать стандартные приложения MS Windows.
3. Написать приложение, использующее для передачи данных общую область в страничном файле.
4. Написать приложения для передачи данных через именованный канал. Метод передачи – любой, передавать можно любую строку из одного приложения в другое.

3. СОДЕРЖАНИЕ ОТЧЕТА

Отчет о работе должен содержать:

1. Описание алгоритмов работы приложений, разработанных в п. 2, и п.3 задания на выполнение работы;
2. Тексты программ для их реализации;
3. Результаты выполнения;
4. Выводы.

4. КОНТРОЛЬНЫЕ ВОПРОСЫ:

1. Способы передачи данных между процессами.
2. Виды связей между процессами.
3. Синхронный обмен данными.
4. Асинхронный обмен данными.

5. Буфер обмена.
6. Файлы, проецируемые в память.
7. Создание именованного канала.
8. Принцип работы функций ReadFile и WriteFile.

ЛАБОРАТОРНАЯ РАБОТА № 7

ДРАЙВЕРЫ РЕЖИМА ЯДРА

Цель работы – знакомство со структурой драйвера режима ядра, средствами и процедурой инсталляции драйвера и функцией управления драйвером DeviceIOControl.

1. КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

1.1 Структура драйвера

Драйвер режима ядра реализуется как *набор функций*, каждая из которых предназначена для реализации отдельного типа обращений к драйверу со стороны Диспетчера ввода-вывода. Экспорт этих функций выполняется путем их регистрации в процедуре, стандартной для всех драйверов, - DriverEntry. Драйвер может быть загружен и выгружен, а для выполнения действий по инициализации и освобождению ресурсов драйвер должен зарегистрировать соответствующие рабочие функции [5]. Описание прототипов рабочих процедур драйвера можно найти в документации DDK(2000, XP) – файл kmarch.chm.

Рассматриваемый в работе драйвер содержит следующие функции:

– процедуру DriverEntry;

– вспомогательную функцию CompleteIrp, реализующую действия по завершению обработки IRP пакета с кодом завершения status;

– рабочую процедуру обработки запросов read/write, предназначенную для обработки запросов Диспетчера ввода/вывода, которые он формирует в виде IRP пакетов с кодами IRP_MJ_READ/ IRP_MJ_WRITE;

– рабочую процедуру обработки запросов открытия драйвера Create_File_IRPprocessing, которая предназначена для обработки запросов Диспетчера ввода/вывода с кодами IPR_MJ_CREATE. Без регистрации данной процедуры система не позволяет выполнять вызов драйвера из пользовательского режима функцией CreateFile;

– рабочую процедуру обработки запросов закрытия драйвера Close_File_IRPprocessing, которая предназначена для обработки запросов Диспетчера ввода/вывода с кодами IPR_MJ_CLOSE;

– рабочую процедуру обработки IOCTL запросов. Она предназначена для обработки запросов Диспетчера ввода/вывода, которые он формирует в виде IRP пакетов с кодами IRP_MJ_DEVICE_CONTROL по результатам обращения к драйверу из пользовательского приложения с помощью функций DeviceIOControl [5].

В примере последняя функция реализует обработку пяти IOCTL запросов:

– IOCTL _PRINT_DEBUG_MESS – вывод отладочного сообщения;

– IOCTL_CHANGE_IRQL CTL – поднятие уровня IRQL;

– IOCTL_MAKE_SYSTEM_CRASH – попытка обрушить ОС;

– IOCTL_TOUCH_PORT_378H – обращение к аппаратным ресурсам системы;

– IOCTL_SEND_BYTE_TO_USER – отправить байт данных в пользовательское приложение.

Данные коды определены с помощью макроса CTL_CODE следующим образом:

```
#define IOCTL_PRINT_DEBUG_MESS CTL_CODE( \
FILE_DEVICE_UNKNOWN,        0x801,        METHOD_BUFFERED,
FILE_ANY_ACCESS)
#define IOCTL_CHANGE_IRQL CTL_CODE(\
FILE_DEVICE_UNKNOWN,        0x802,        METHOD_BUFFERED,
FILE_ANY_ACCESS)
#define IOCTL_MAKE_SYSTEM_CRASH CTL_CODE( \
FILE_DEVICE_UNKNOWN,        0x803,        METHOD_BUFFERED,
FILE_ANY_ACCESS)
#define IOCTL_TOUCH_PORT_378H CTL_CODE( \
FILE_DEVICE_UNKNOWN,        0x804,        METHOD_BUFFERED,
FILE_ANY_ACCESS)
#define IOCTL_SEND_BYTE_TO_USER CTL_CODE( \
FILE_DEVICE_UNKNOWN,        0x805,        METHOD_BUFFERED,
FILE_ANY_ACCESS)
#define FILE_DEVICE_UNKNOWN 0x00000022
#define METHOD_IN_DIRECT 1
#define METHOD_OUT_DIRECT 2
#define METHOD_BUFFERED 0
#define FILE_ANY_ACCESS 0
CTL_CODE (DevType, Func, Method, Access) = (DevType<<16) or
(Access<<14) or (Func<<2) or (Method)
```

Текст драйвера [5]

```
//      Инициализация драйвера
//      DriverEntry       Главная точка входа в драйвер
//      UnloadRoutine  Процедура выгрузки драйвера
//      DeviceControlRoutine Обработчик DeviceIoControl IRP пакетов
#include "Driver.h"
// Предварительные объявления функций:
```

```
NTSTATUS DeviceControlRoutine( IN PDEVICE_OBJECT fdo, IN PIRP Irp );
VOID UnloadRoutine(IN PDRIVER_OBJECT DriverObject);
NTSTATUS ReadWrite_IRPhandler( IN PDEVICE_OBJECT fdo, IN PIRP Irp );
NTSTATUS Create_File_IRPprocessing(IN PDEVICE_OBJECT fdo, IN PIRP Irp);
NTSTATUS Close_HandleIRPprocessing(IN PDEVICE_OBJECT fdo, IN PIRP Irp);
KSPIN_LOCK MySpinLock;
#pragma code_seg("INIT") // начало секции INIT
//      DriverEntry - инициализация драйвера и необходимых объектов
extern "C"
NTSTATUS DriverEntry(  IN PDRIVER_OBJECT DriverObject,
            IN PUNICODE_STRING RegistryPath )
{
NTSTATUS status = STATUS_SUCCESS;
PDEVICE_OBJECT fdo;
UNICODE_STRING devName;
// Экспорт точек входа в драйвер
//DriverObject->DriverExtension->AddDevice= OurAddDeviceRoutine;
DriverObject->DriverUnload = UnloadRoutine;
DriverObject->MajorFunction[IRP_MJ_CREATE]=
Create_File_IRPprocessing;
DriverObject->MajorFunction[IRP_MJ_CLOSE]                    =
Close_HandleIRPprocessing;
DriverObject->MajorFunction[IRP_MJ_READ] = ReadWrite_IRPhandler;
DriverObject->MajorFunction[IRP_MJ_WRITE] = ReadWrite_IRPhandler;
DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]=
DeviceControlRoutine;
```

```
// Действия по созданию символьной ссылки
RtlInitUnicodeString( &devName, L"\\Device\\EXAMPLE" );
// Создаем Functional Device Object (FDO) и получаем
// указатель на созданный FDO в переменной fdo.
// EXAMPLE_DEVICE_EXTENSION (для этого передаем в вызов ее размер, вычисляемый оператором sizeof):
status = IoCreateDevice(DriverObject,
        sizeof (EXAMPLE_DEVICE_EXTENSION),
        &devName, // может быть и NULL
        FILE_DEVICE_UNKNOWN,
        0,
        FALSE, // без эксклюзивного доступа
        &fdo);
        if (!NT_SUCCESS (status)) return status;
      // Получаем указатель на область, предназначенную под структуру расширение устройства
PEXAMPLE_DEVICE_EXTENSION dx =
 (PEXAMPLE_DEVICE_EXTENSION)fdo->DeviceExtension;
dx->fdo = fdo; // Сохраняем обратный указатель
// Действия по созданию символьной ссылки
UNICODE_STRING symLinkName; // Сформировать символьное имя:
// Создаем символьную ссылку
status = IoCreateSymbolicLink( &symLinkName, &devName );
        } // Теперь можно вызывать CreateFile("\\\\.\\Example",…);
        // в пользовательских приложениях
// Объект спин-блокировки, который будем использовать для разнесения во времени выполнения кода обработчика IOCTL запросов. Инициализируем его:
KeInitializeSpinLock(&MySpinLock);
#pragma code_seg() // end INIT section
```

```
// CompleteIrp: Устанавливает IoStatus и завершает обработку IRP
// Первый аргумент – указатель на объект FDO.
      NTSTATUS CompleteIrp( PIRP Irp, NTSTATUS status, ULONG info)
  {
    Irp->IoStatus.Status = status;
    Irp->IoStatus.Information = info;
    IoCompleteRequest(Irp,IO_NO_INCREMENT);
  return status;
  }
```

// ReadWrite_IRPhandler: Берет на себя обработку запросов чтения/записи и завершает обработку IRP вызовом CompleteIrp с числом переданных/полученных байт (BytesTxd) равным нулю.

```
// Аргументы: указатель на объект FDO
// Указатель на структуру IRP, поступившего от Диспетчера ввода вывода
NTSTATUS ReadWrite_IRPhandler( IN PDEVICE_OBJECT fdo, IN PIRP Irp )
{
ULONG BytesTxd = 0;
NTSTATUS status = STATUS_SUCCESS; //Завершение с кодом status
NTSTATUS Create_File_IRPprocessing(IN PDEVICE_OBJECT fdo,IN PIRP Irp)
{
PIO_STACK_LOCATION IrpStack = IoGetCurrentIrpStackLocation(Irp);
```

// Close_File_IRPprocessing: Берет на себя обработку запросов с кодом IRP_MJ_CLOSE.

```
// Аргументы: Указатель на объект FDO
// Указатель на структуру IRP, поступившего от Диспетчера ввода/вывода
```

```
NTSTATUS Close_HandleIRPprocessing(IN PDEVICE_OBJECT fdo, IN PIRP Irp)
// DeviceControlRoutine: обработчик IRP_MJ_DEVICE_CONTROL запросов // Аргументы: Указатель на объект FDO
// Указатель на структуру IRP, поступившего от Диспетчера ВВ
//      Возвращает: STATUS_XXX
NTSTATUS DeviceControlRoutine( IN PDEVICE_OBJECT fdo, IN PIRP Irp )
{
NTSTATUS status = STATUS_SUCCESS;
ULONG BytesTxd =0; // Число переданных/полученных байт (пока 0)
PIO_STACK_LOCATION IrpStack=IoGetCurrentIrpStackLocation(Irp);
// Получение указателя на расширение устройства
PEXAMPLE_DEVICE_EXTENSION dx =
      (PEXAMPLE_DEVICE_EXTENSION)fdo->DeviceExtension;
     // Выделяем из IRP собственно значение IOCTL кода, по поводу которого случился вызов:
   ULONG ControlCode =
   IrpStack->Parameters.DeviceIoControl.IoControlCode;
   ULONG method = ControlCode & 0x03;
// Получение текущего значения уровня IRQL
   KIRQL irql,
   currentIrql = KeGetCurrentIrql();
     // Запрос владения объектом спин-блокировки. Пока потоку выделен объект спин-блокировки –  никакой другой поток не сможет войти в оператор switch:
       KeAcquireSpinLock(&MySpinLock,&irql);
       // Диспетчеризация по IOCTL кодам:
       switch( ControlCode) {
       #ifndef SMALL_VERSION
```

```
case IOCTL_PRINT_DEBUG_MESS:
case IOCTL_CHANGE_IRQL:
case IOCTL_MAKE_SYSTEM_CRASH:
{
int errDetected=0;
char x = (char)0xFF;
// Вызываем системный сбой обращением по нулевому адресу
__try {
x = *(char*)0x0L; // ошибочная ситуация
        //здесь случится сбой NT
}
__except(EXCEPTION_EXECUTE_HANDLER)
{ // Перехват исключения не работает!
        errDetected=1;
        };
        break;
}

#ifndef SMALL_VERSION
case IOCTL_TOUCH_PORT_378H:
{
        unsigned short ECRegister = 0x378+0x402;
// Пробуем программно перевести параллельный порт 378, сконфигурированный средствами BIOS // как ECP+EPP, в режим EPP.
_asm {
        mov dx,ECRegister ;
        xor al,al ;
        out dx,al ; Установить EPP mode 000
        mov al,095h ; Биты 7:5 = 100
        out dx,al ; Установить EPP mode 100
```

```
 }
// Эти строки демонстрируют использование LPT порта
        break;
 }

 case IOCTL_SEND_BYTE_TO_USER:
{       // Размер данных, поступивших от пользователя:
 ULONG InputLength = // для примера
 IrpStack->Parameters.DeviceIoControl.InputBufferLength;
 // Размер буфера для данных, ожидаемых пользователем
 ULONG OutputLength =
 IrpStack->Parameters.DeviceIoControl.OutputBufferLength;
 if( OutputLength<1 )
 {// Если не предоставлен буфер – завершить IRP с ошибкой
 status = STATUS_INVALID_PARAMETER;
 break;
 }
 UCHAR *buff; // unsigned char, привыкаем к новой нотации
 if (method==METHOD_BUFFERED)
 {
        buff = (PUCHAR)Irp->AssociatedIrp.SystemBuffer;
 }
 else
 if (method==METHOD_NEITHER)
        {
        buff=(unsigned char*)Irp->UserBuffer;
        }
        else
        {
        status = STATUS_INVALID_DEVICE_REQUEST;
```

```
            break;
            }
            *buff=33; //
            BytesTxd = 1; // Передан 1 байт
            break;
        }
        #endif // SMALL_VERSION
        // Ошибочный запрос (код IOCTL, который не обрабатывается):
        default: status = STATUS_INVALID_DEVICE_REQUEST;
        }
        // Освобождение спин-блокировки
        KeReleaseSpinLock(&MySpinLock,irql);
        return CompleteIrp(Irp,status,BytesTxd); // Завершение IRP
      }
      // UnloadRoutine: Выгружает драйвер
      // Arguments:  указатель на объект драйвера
      #pragma code_seg("PAGE")
      // Допускает размещение в странично организованной памяти
      VOID UnloadRoutine(IN PDRIVER_OBJECT pDriverObject)
      {
        PDEVICE_OBJECT  pNextDevObj;
        int i;
        // Проход по всем объектам, контролируемым  драйвером
        pNextDevObj = pDriverObject->DeviceObject;
        for(i=0; pNextDevObj!=NULL; i++)
        {
        PEXAMPLE_DEVICE_EXTENSION dx =
        (PEXAMPLE_DEVICE_EXTENSION)pNextDevObj-
>DeviceExtension;
        // Удаление символьной ссылки и уничтожение FDO:
```

```
  UNICODE_STRING *pLinkName = & (dx->ustrSymLinkName);
  // !!! сохраняем указатель:
  pNextDevObj = pNextDevObj->NextDevice;
  IoDeleteSymbolicLink(pLinkName);
  IoDeleteDevice( dx->fdo);
  }
}
#pragma code_seg() // end PAGE section
```

1.2 Инсталляция драйвера

Существует несколько способов инсталляции и запуска драйвера [5]:

- инсталляция путем внесения записей в реестр;
- инсталляция с использованием INF файла;
- инсталляция с использованием сервисов Service Control Manager (SCM).

Для установки драйвера ***вторым способом*** необходимо создать текстовый файл, содержащий информацию, необходимую для работы Мастера установки нового оборудования. Составление inf-файлов описано в документации DDK, файл install.chm, а также в главе 11 [5]. В данной работе необходимая для установки информация находится в подкаталоге \install. Используя указанную информацию, можно установить драйвер с помощью мастера установки нового оборудования. На панели управления выбрать Установка оборудования, в списке оборудования выбрать Добавление нового устройства, затем – показать все устройства и Установить с диска.

После инсталляции и запуска драйвера следует убедиться в его присутствии в системе с помощью утилиты DeviceTree и просмотреть

список поддерживаемых им функций (в окне Major Function Codes Supported).

1.3 Тестирование драйвера

1. Открыть драйвер с помощью функции CreateFile

```
hHandle = CreateFile( “\\.\Example”,
      GENERIC_READ | GENERIC_WRITE,
      FILE_SHARE_READ | FILE_SHARE_WRITE, NULL,
OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0 );
    if (hHandle == INVALID_HANDLE_VALUE)
    printf (“ERR: can not access driver Example.sys”);
```

2. Последовательно выполнить обращения к драйверу с различными кодами IOCTL с помощью функции DeviceIOControl

```
if (!DeviceIoControl ( hHandle, ioctlCode,
                    NULL, 0,  // Input
                    NULL, 0,  // Output
                    BytesReturned,
                    NULL ))
                    printf ( “ Error in IOCTL_”);
```

2. МЕТОДИКА ВЫПОЛНЕНИЯ

1. Ознакомиться с п. 1.1, 1.2, 1.3.

2. Установить драйвер на виртуальной машине, используя второй способ.

3. Написать и выполнить приложение для тестирования драйвера путем выполнения следующих операций:

– IOCTL_MAKE_SYSTEM_CRASH – попытка обрушить ОС

– IOCTL_TOUCH_PORT_378H – обращение к аппаратным ресурсам системы
– IOCTL_SEND_BYTE_TO_USER – отправить байт данных в пользовательское приложение

Для выполнения всех операции использовать вызов DeviceIOControl

3. СОДЕРЖАНИЕ ОТЧЕТА

Отчет о работе должен содержать текст тестирующего приложения с комментариями, описание, объяснение и оценку полученных результатов.

4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Назначение драйверов. Виды драйверов.
2. Взаимодействие Диспетчера ввода-вывода с драйвером.
3. Структура драйвера.
4. Способы инсталляции драйверов.
5. Функция DeviceIOControl – назначение, параметры, использование

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. http://msdn.microsoft.com/ru-ru/ [Электронный ресурс]
2. Рихтер Дж. Windows для профессионалов. 4-е изд. – СПб: Питер, 2008- 720с. – ISBN 978-5-7502-0360-4.
3. Рихтер Дж., Назар К. Windows via C/C++. Программирование на языке Visual C++.- СПб.: Питер, 2009- 896 с.-ISBN 978-5-7502-0367-3.
4. Руссинович.М. Внутреннее устройство Microsoft Windows [Текст] : Windows Server 2003,Windows XP и Windows 2000:[Пер.с англ.] / М.Руссинович, Д.Соломон. - 4-е изд. - М. : Рус.Ред. ; СПб. : Питер, 2008. - 968 с. - (Мастер-класс). - ISBN 978-5-469-011 74-3(в пер.). - ISBN 0-7356-1917-4.
5. Солдатов В.П. Программирование драйверов Windows. – М.: ООО «Бином-Пресс», 2004 г. - 432 с. – ISBN 5-9518-0099-4.

ОГЛАВЛЕНИЕ

Системное программное обеспечение

Составители*: ТИХОМИРОВ Алексей Александрович*
КИСТАНОВ Алексей Михайлович

В авторской редакции

Подписано в печать 19.06.2013
Формат 60x84 $^{1}/_{16}$. Бум. офсетная.
Печать офсетная.
Усл. п. л. 4,6
Тираж 50 экз.

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
"Самарский государственный технический университет"
443100, г. Самара, ул. Молодогвардейская, 244.
Главный корпус

Отпечатано в типографии
Самарского государственного технического университета
443100, г. Самара, ул. Молодогвардейская, 244. Корпус № 8