Модуль №6

Занятие №1

Версия 1.0.1

План занятия:

1. Основные сведения о процессах.
2. Основные сведения об объекте ядра.
3. Дочерние процессы.
   3.1. Понятие дочернего процесса.
   3.2. Наследование дескрипторов объектов ядра.
   3.3. Запуск обособленных дочерних процессов.
4. Получение списка запущенных в системе процессов.
5. Подведение итогов.
6. Домашнее задание.

1. Основные сведения о процессах

Начать рассмотрение данного вопроса следует с определения понятия многозадачности.

Многозадачность – это способность операционной системы выполнять одновременно несколько программ. При этом каждой программе предоставляется квант процессорного времени и создается иллюзия одновременного выполнения программ. Разумеется, для многопроцессорных систем возможен истинный параллелизм.

Далее следует пояснить слушателям термин «**процесс**». Отметить, что **процесс – это программа, запущенная на выполнение**. Две

запущенные копии одной и той же программы - это два отдельных процесса. Отметить, что процесс состоит из двух компонент:

1) **Объект ядра «процесс»** – структура данных Windows, через которую система управляет процессом. В ней хранится статистическая информация о процессе.
2) **Адресное пространство**, в котором содержится код и данные всех EXE- и DLL модулей.

Следует подчеркнуть, что процессы инертны. Чтобы процесс что-нибудь выполнил, в нем нужно создать поток. Именно потоки отвечают за исполнение кода, содержащегося в адресном пространстве процесса.

Далее привести функцию API **CreateProcess**, позволяющую создать процесс:

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,
  LPTSTR lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL bInheritHandles,
  DWORD dwCreationFlags,
  LPVOID lpEnvironment,
  LPCTSTR lpCurrentDirectory,
  LPSTARTUPINFO lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

Подчеркнуть, когда поток в приложении вызывает функцию **CreateProcess**, система создает **объект ядра «процесс»** с начальным значением счётчика числа его пользователей, равным 1. Объект ядра «процесс» — это структура данных, через которую операционная система управляет процессом. Затем система создает для нового процесса виртуальное адресное пространство и загружает в него код и данные, как для исполняемого файла, так и для любых DLL. После этого система формирует **объект ядра «поток»** (со счетчиком, равным 1) для пер-

вичного потока нового процесса. Как и в первом случае, объект ядра «поток» — это структура данных, через которую система управляет потоком. Если системе удастся создать новый процесс и его первичный поток, **CreateProcess** вернет **TRUE**.

Далее следует подробно прокомментировать каждый параметр вышеприведенной функции.

Параметры	Описание
lpApplicationName	Указатель на строку, содержащую имя исполняемой программы. Имя может быть полное. Если оно не полное, то поиск файла производится в текущем каталоге. Параметру может быть присвоено значение NULL. В этом случае в качестве имени файла выступает первая выделенная пробелами лексема из строки lpCommandLine.
lpCommandLine	Указатель командной строки. Если параметр lpApplicationName имеет значение NULL, то имя исполняемого файла выделяется из lpCommandLine, а поиск исполняемого файла производится в следующем порядке: 1. Каталог, содержащий EXE-файл вызывающего процесса. 2. Текущий каталог вызывающего процесса. 3. Системный каталог Windows. 4. Основной каталог Windows. 5. Каталоги, перечисленные в переменной окружения PATH.
lpProcessAttributes	Указатель на структуру, описывающую параметры защиты процесса. Если параметру присвоено значение NULL, то устанавливаются атрибуты по умолчанию.
lpThreadAttributes	Указатель на структуру, описывающую параметры защиты первичного потока. Если параметру присвоено значение NULL, то устанавливаются атрибуты по умолчанию.
bInheritHandles	Параметр определяет, будет ли порожденный процесс наследовать описатели (дескрипторы) объектов ядра родительского процесса.
dwCreationFlags	Параметр определяет некоторые дополнительные условия создания процесса и его класс приоритета.
lpEnvironment	Указатель на блок переменных среды порожденного процесса. Если этот параметр равен NULL, то порожденный процесс наследует среду родителя.
lpCurrentDirectory	Указатель на строку, содержащую полное имя текущего каталога порожденного процесса. Если этот параметр равен

Параметры	Описание
	NULL, то порожденный процесс наследует каталог родителя.
lpStartupInfo	Указатель на структуру STARTUPINFO, которая определяет параметры главного окна порожденного процесса.
lpProcessInformation	Указатель на структуру PROCESS_INFORMATION, которая будет заполнена информацией о порожденном процессе после возврата из функции.

Для демонстрации работы функции привести следующий пример:

```
STARTUPINFO s = {sizeof(STARTUPINFO)};
PROCESS_INFORMATION p;
TCHAR buffer[] = TEXT("Calc.exe");
//функция CreateProcess запускает калькулятор
CreateProcess(NULL, buffer, NULL, NULL, FALSE, 0, NULL, NULL, &s, &p);
```

Следует подробно рассмотреть поля структуры **STARTUPINFO**.

```
typedef struct _STARTUPINFO{
DWORD cb; // содержит количество байтов, занимаемых структурой STARTUPINFO
LPTSTR lpReserved; // зарезервирован
LPTSTR lpDesktop; /* Идентифицирует имя рабочего стола, на котором запускается приложение. Если присвоить NULL, процесс связывается с текущим рабочим столом. */
LPTSTR lpTitle; // определяет заголовок консольного окна
DWORD dwX; // указывает x - координату окна приложения
DWORD dwY; // указывают y - координаты окна приложения
DWORD dwXSize; //определяют ширину (в пикселях) окна приложения
DWORD dwYSize; //определяют высоту (в пикселях) окна приложения
DWORD dwXCountChars; // определяют ширину (в символах) консольных окон
// дочернего процесса
DWORD dwYCountChars; // определяют высоту (в символах) консольных окон
// дочернего процесса
DWORD dwFillAttribute; // задает цвет текста и фона в консольных окнах
// дочернего процесса
DWORD dwFlags; // содержит набор флагов, позволяющих управлять созданием
// дочернего процесса.
```

```
WORD wShowWindow; /* Определяет, как именно должно выглядеть первое перекры-
ваемое окно дочернего процесса, если приложение при первом вызове функции
ShowWindow передает в параметре nCmdShow идентификатор SW_SHOWDEFAULT. */
WORD cbReserved2; // зарезервирован
LPBYTE lpReserved2; // зарезервирован
HANDLE hStdInput;
HANDLE hStdOutput;
HANDLE hStdError;
/* Последние три поля определяют описатели буферов для консольного ввода-
вывода. По умолчанию hStdInput идентифицирует буфер клавиатуры, а hStdOutput
и hStdError — буфер консольного окна. */
}STARTUPINFO, *LPSTARTUPINFO;
```

Следует отметить, что большинство приложений порождает процессы с атрибутами по умолчанию. Но даже в этом случае необходимо инициализировать все элементы структуры хотя бы нулевыми значениями, а в элемент **cb** - заносить размер этой структуры.

Далее следует рассмотреть структуру **PROCESS_INFORMATION**.

```
typedef struct _PROCESS_INFORMATION {
  HANDLE hProcess; // дескриптор созданного процесса
  HANDLE hThread; // дескриптор первичного потока процесса
  DWORD dwProcessId; // идентификатор процесса
  DWORD dwThreadId; // идентификатор первичного потока
}PROCESS_INFORMATION, *LPPROCESS_INFORMATION;
```

Детально ознакомив слушателей с функцией создания процесса, необходимо рассмотреть способы завершения процесса. Подчеркнуть, что процесс будет завершён в следующих случаях:

- входная функция первичного потока возвращает управление (рекомендуемый способ);
- поток процесса вызывает функцию API **ExitProcess**;
- поток другого процесса вызывает функцию API **TerminateProcess.**

Акцентировать внимание слушателей на том, что при разработке приложения желательно, чтобы его процесс завершался только после возврата управления входной функцией первичного потока. Это единственный способ, гарантирующий корректную очистку всех ресурсов, принадлежавших первичному потоку.

```
VOID ExitProcess(
  UINT uExitCode // код завершения процесса
);
```

```
BOOL TerminateProcess(
  HANDLE hProcess, // дескриптор завершаемого процесса
  UINT uExitCode  // код завершения процесса
 );
```

Главное отличие функции **TerminateProcess** от **ExitProcess** состоит в том, что ее может вызвать любой поток и завершить любой процесс.

2. Основные сведения об объекте ядра

Объекты ядра используются системой и приложениями для управления самыми разнообразными ресурсами, например, процессами, потоками, файлами, семафорами, событиями и многими другими. Каждый **объект ядра — это блок памяти, выделенный ядром и доступный только ему. Этот блок представляет собой структуру данных, в полях которой содержится информация об объекте**. Некоторые поля (дескриптор защиты, счетчик числа пользователей и др.) присутствуют во всех объектах, но большая их часть специфична для объектов ядра конкретного типа. Например, у объекта ядра «процесс» есть идентификатор, базовый приоритет и код завершения, а у объекта ядра «файл» — смещение в байтах, режим разделения и режим открытия.

Акцентировать внимание слушателей на том, что структуры объектов ядра доступны только ядру, и приложение не может самостоятельно найти эти структуры в памяти и напрямую модифицировать их содержимое. Такое ограничение введено намеренно, чтобы ни одна программа не могла нарушить целостность структур объектов ядра. При этом в Windows предусмотрен набор функций, обрабатывающих структуры объектов ядра по строго определенным правилам. Таким образом, программа может получить доступ к объектам ядра только через эти функции.

Важно отметить, что **объекты ядра принадлежат ядру, а не процессу**. Иначе говоря, если некоторый процесс вызывает функцию, создающую объект ядра, а затем завершается, объект ядра может быть не разрушен, в случае если созданный объект ядра используется другим процессом. Ядро запретит разрушение объекта до тех пор, пока от него не откажется и тот процесс.

Акцентировать внимание слушателей на том, что **ядру всегда известно, сколько процессов использует конкретный объект ядра, поскольку в каждом объекте есть счетчик числа его пользователей**. В момент создания объекта счетчику присваивается единичное значение. Когда к существующему объекту ядра обращается другой процесс, счетчик увеличивается на 1. А когда какой-то процесс завершается, счетчики всех используемых им объектов ядра автоматически уменьшаются на 1. Как только счетчик какого-либо объекта обнуляется, ядро уничтожает этот объект.

Объекты ядра можно защитить специальным дескриптором защиты, который описывает, кто создал объект и кто имеет права на доступ к нему. Дескрипторы защиты обычно используется при написании серверных приложений. При разработке клиентского приложения объект ядра чаще всего создаётся с защитой по умолчанию.

Следует отметить, что любая функция, создающая объект ядра (например, **CreateFile**), возвращает дескриптор созданного объекта. Для

большей надежности операционной системы Microsoft сделала так, что **значения дескрипторов объектов ядра действительны только в адресном пространстве процесса, их создавшего**. Поэтому все попытки передачи такого дескриптора другому процессу с помощью какого-либо механизма межпроцессной связи и его использования в другом процессе приводят к ошибкам. Это обусловлено тем, что для каждого процесса создаётся таблица дескрипторов, в которой каждая строка идентифицирует объект ядра, используемый данным процессом.

Индекс (дескриптор)	Указатель на блок памяти (объект ядра)	Маска доступа (DWORD)	Флаги (DWORD)
1	0x???	0x???	0x???
2	0x???	0x???	0x???
3	0x???	0x???	0x???

Указатель на блок памяти (объект ядра) – содержит адрес системной памяти, где расположен объект ядра.

Маска доступа (DWORD) – определяет, кто имеет доступ к объекту. Если маска доступа не указывается при создании объекта, то доступ к нему имеют только члены группы администраторов.

Флаги (DWORD) – опции для созданного объекта. Например, наследуется ли дескриптор или нет.

Таким образом, любая функция, возвращающая дескриптор объекта ядра, вписывает новую строку в таблицу дескрипторов. Дескриптор служит в качестве индекса, соответствующего строке таблицы. При вызове функции, принимающей в качестве аргумента дескриптор объекта ядра, происходит обращение в таблицу дескрипторов, принадлежащих конкретному процессу, и считывается адрес нужного объекта ядра. Если же передать функции дескриптор, относящийся к «чужому» процессу (т.е. индекс, которого нет в таблице), функция завершится с ошибкой и функция **GetLastError** возвратит значение **ERROR_INVALID_HANDLE**.

При вызове функции API **CloseHandle** происходит уничтожение строки из таблицы дескрипторов.

```
BOOL CloseHandle(
  HANDLE hObject //дескриптор объекта ядра
);
```

Эта функция сначала проверяет таблицу дескрипторов данного процесса, чтобы убедиться, что процесс имеет доступ к объекту **hObject**. Если доступ разрешен, то система получает адрес структуры данных объекта **hObject** и уменьшает в ней счетчик количества пользователей. Как только счетчик обнулится, ядро удаляет объект из памяти.

3. Дочерние процессы

3.1. Понятие дочернего процесса

При разработке приложения часто бывает нужно, чтобы какую-то операцию выполнял другой блок кода. Поэтому приходится вызывать функции или подпрограммы. Но вызов функции приводит к приостановке выполнения основного кода программы до возврата из вызванной функции. Один из альтернативных способов состоит в том, что процесс порождает дочерний при помощи уже рассмотренной функции **CreateProcess** и возлагает на него выполнение части операций.

3.2. Наследование дескрипторов объектов ядра

Следует отметить, что время от времени возникает необходимость в разделении объектов ядра между потоками, исполняемыми в разных процессах. Причин тому может быть несколько:

- объекты «проекции файлов» позволяют двум процессам, исполняемым на одной машине, совместно использовать одни и те же блоки данных;
- почтовые ящики и именованные каналы дают возможность программам обмениваться данными с процессами, исполняемыми на других машинах в сети;
- мьютексы, семафоры и события позволяют синхронизировать потоки, исполняемые в разных процессах, чтобы одно приложение могло уведомить другое об окончании той или иной операции.

Как отмечалось ранее, значения дескрипторов объектов ядра действительны только в адресном пространстве процесса, их создавшего. Однако существуют механизмы, позволяющие процессам совместно использовать одни и те же объекты ядра. Один из таких механизмов заключается в наследовании описателя (дескриптора) объекта ядра в дочернем процессе. Другими словами, родительский процесс может передавать по наследству дочернему процессу дескрипторы объектов ядра. Важно отметить, что дескрипторы при этом должны быть наследуемыми, что можно явно указать при их создании. В качестве иллюстрации к сказанному привести следующий фрагмент кода:

```
SECURITY_ATTRIBUTES sa;
sa.nLength = sizeof(SECURITY_ATTRIBUTES); // размер структуры
sa.lpSecurityDescriptor = 0; // система защиты по умолчанию
sa.bInheritHandle = TRUE; // дескриптор объекта ядра "Файл" наследуемый

HANDLE hHandle = CreateFile(
TEXT("Test.txt"), // путь к файлу
GENERIC_READ, // только чтение
0, // запрет совместного использования файла
&sa, //атрибуты защиты объекта ядра «файл»
OPEN_EXISTING, // открытие существующего файла
FILE_ATTRIBUTE_NORMAL, // атрибутов нет
NULL // обычно используется в клиент-серверном приложении
);
```

```
if ( hHandle == INVALID_HANDLE_VALUE ) //ошибка открытия файла
      return;

STARTUPINFO startin = {sizeof(STARTUPINFO)};
PROCESS_INFORMATION info = {0};


TCHAR cmd[20];
//дескриптор объекта ядра передается дочернему процессу через командную
//строку
wsprintf(cmd, TEXT("%s %d"), TEXT("child.exe"), hHandle);

CreateProcess(
NULL, // в качестве имени файла выступает первая выделенная пробелами лексема
//из 2-го параметра
cmd, // Указатель командной строки, содержащей, в том числе, имя исполняемого
//файла
NULL, // атрибуты защиты процесса по умолчанию
NULL, // атрибуты защиты потока по умолчанию
TRUE, // наследование дескрипторов объектов ядра разрешено
0, // класс приоритета по умолчанию - NORMAL_PRIORITY_CLASS
NULL, // среда окружения наследуется у родительского процесса
NULL, // текущий каталог наследуется у родительского процесса
&startin, //способ отображения главного окна, а также размер и заголовок окна
&info // информация о порождённом процессе
);
CloseHandle(hHandle); // закрывается описатель объекта ядра «файл»
CloseHandle(info.hThread); // закрывается описатель объекта ядра «поток»
CloseHandle(info.hProcess); // закрывается описатель объекта ядра «процесс»
```

В приведенном фрагменте кода с помощью функции API **CreateFile** создаётся **объект ядра «файл»**.

```
HANDLE CreateFile(
  LPCTSTR lpFileName, // имя создаваемого или открываемого файла
  DWORD dwDesiredAccess, // способ доступа к содержимому файла
  DWORD dwShareMode, // тип совместного доступа к данному файлу
  LPSECURITY_ATTRIBUTES lpSecurityAttributes, // атрибуты защиты
  DWORD dwCreationDisposition, /* действие, которое необходимо выполнить в
том случае, если файл существует, и в том случае, если файл не существует */
  DWORD dwFlagsAndAttributes, //атрибуты файла
  HANDLE hTemplateFile /* дескриптор с правами доступа GENERIC_READ к шаблону
файла, который предоставляет расширенные атрибуты для создаваемого файла */
);
```

```
    typedef struct _SECURITY_ATTRIBUTES {
    DWORD nLength, // размер структуры в байтах
    LPVOID lpSecurityDescriptor; // дескриптор безопасности
```

```
    BOOL bInheritHandle; /* данный параметр указывает, будет ли дескриптор
    объекта ядра наследуемым */
    } SECURITY_ATTRIBUTES, *PSECURITY_ATTRIBUTES;
```

Обратить внимание слушателей на 4-й параметр **lpSecurityAttributes** функции **CreateFile**, представляющий наибольший интерес. Этот параметр – указатель на структуру **SECURITY_ATTRIBUTES**, определяющую атрибуты безопасности, с которыми создаётся объект ядра. В данном примере объект ядра создаётся с защитой по умолчанию, о чём свидетельствует нулевое значение поля **lpSecurityDescriptor**. Значение **TRUE** поля **bInheritHandle** указывает на то, что описатель объекта ядра «файл» станет наследуемым и в дальнейшем может быть унаследован дочерним процессом.

Для того чтобы дочерний процесс мог наследовать дескрипторы объектов ядра родительского процесса необходимо в 5-м параметре **bInheritHandles** функции **CreateProcess** указать **TRUE.** При инициализации дочернего процесса система сформирует в нем пустую таблицу описателей объектов ядра. Затем в эту таблицу из таблицы родительского процесса будут скопированы записи, соответствующие объектам ядра, у которых наследуемые дескрипторы. Причем копирование произойдёт в те же позиции. Последний факт чрезвычайно важен, так как означает, что дескрипторы будут идентичны в обоих процессах (родительском и дочернем). Помимо копирования записей из таблицы описателей, система увеличивает значения счетчиков соответствующих объектов ядра, поскольку эти объекты теперь используются обоими процессами. Чтобы уничтожить какой-то объект ядра, его описатель должны закрыть (вызовом **CloseHandle**) оба процесса. Кстати, сразу после возврата управления функцией **CreateProcess** родительский процесс может закрыть свой описатель объекта, и это никак не отразится на способности дочернего процесса манипулировать этим объектом.

Акцентировать внимание слушателей на следующей особенности: **дочерний процесс не знает о том, что он унаследовал какие-то дескрипторы**. Поэтому наследование дескрипторов объектов ядра по-

лезно только когда дочерний процесс сообщает, что при его создании родительским процессом он ожидает доступа к какому-нибудь объекту ядра. Для этого в дочерний процесс обычно передают значение ожидаемого им дескриптора объекта ядра как аргумент в командной строке. Инициализирующий код дочернего процесса анализирует командную строку, извлекает из нее значение дескриптора, и дочерний процесс получает неограниченный доступ к объекту. При этом механизм наследования срабатывает только потому, что значение дескриптора разделяемого объекта ядра в родительском и дочернем процессах одинаково.

Рекомендуется рассмотреть со слушателями следующий пример, в котором демонстрируется наследование описателя объекта ядра.

Исходный код приложения, в котором родительский процесс создает три дочерних процесса, представлен ниже.

```
                                    // header.h
#pragma once
#include <windows.h>
#include <windowsX.h>
#include <tchar.h>
#include "resource.h"


                                // InheritHandleDlg.h
#pragma once
#include "header.h"

class CInheritHandleDlg
{
public:
      CInheritHandleDlg(void);
public:
      ~CInheritHandleDlg(void);
      static BOOL CALLBACK DlgProc(HWND hWnd, UINT mes, WPARAM wp, LPARAM lp);
      static CInheritHandleDlg* ptr;
      void Cls_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify);
      void Cls_OnClose(HWND hwnd);
};


                               // InheritHandleDlg.cpp
#include "InheritHandleDlg.h"

CInheritHandleDlg* CInheritHandleDlg::ptr = NULL;

CInheritHandleDlg::CInheritHandleDlg(void)
{
```

```
	ptr = this;
}



CInheritHandleDlg::~CInheritHandleDlg(void)
{

}

void CInheritHandleDlg::Cls_OnClose(HWND hwnd)
{
	EndDialog(hwnd, 0);
}

void CInheritHandleDlg::Cls_OnCommand(HWND hwnd, int id, HWND hwndCtl,
                                      UINT codeNotify)
{
	if(id == IDC_BUTTON1)
	{
		SECURITY_ATTRIBUTES sa;
		sa.nLength = sizeof(SECURITY_ATTRIBUTES); // размер структуры
		sa.lpSecurityDescriptor = 0; // система защиты по умолчанию

		// дескриптор объекта ядра "Файл" наследуемый !!!
		sa.bInheritHandle = TRUE;
		HANDLE hHandle = CreateFile(
			TEXT("Test.txt"), // путь к файлу
			GENERIC_READ, // только чтение
			0, // запрет совместного использования файла
			&sa, // система защиты
			OPEN_EXISTING, // открытие существующего файла
			FILE_ATTRIBUTE_NORMAL, // атрибутов нет
			NULL // обычно используется в клиент-серверном приложении
			);
		if ( hHandle == INVALID_HANDLE_VALUE ) // ошибка открытия файла
			return;
		STARTUPINFO startin = {sizeof(STARTUPINFO)};
		PROCESS_INFORMATION info={0};
		TCHAR cmd[20];
		// дескриптор объекта ядра передается дочернему процессу через
		// командную строку
		wsprintf(cmd, TEXT("%s %d"), TEXT("child.exe"), hHandle);
		CreateProcess(
			NULL, // в качестве имени файла выступает первая выделенная
				// пробелами лексема из 2-го параметра
			cmd, // Указатель командной строки, содержащей, в том числе,
				// имя исполняемого файла
			NULL, // атрибуты защиты процесса по умолчанию
			NULL, // атрибуты защиты потока по умолчанию
			TRUE, // наследование разрешено !!!
			0, // класс приоритета по умолчанию - NORMAL_PRIORITY_CLASS
			NULL, // среда окружения наследуется у родительского
				// процесса
			NULL, // текущий каталог наследуется у родительского
				// процесса
			&startin, // способ отображения главного окна, а также
					// размер и заголовок окна
			&info // информация о порождённом процессе
			);
```

```
		CloseHandle(hHandle); // закрывается описатель объекта ядра «файл»
		CloseHandle(info.hThread); // закрывается описатель объекта ядра
					// «поток»


		CloseHandle(info.hProcess); // закрывается описатель объекта ядра
					// «процесс»
	}
	else if(id == IDC_BUTTON2)
	{
		SECURITY_ATTRIBUTES sa;
		sa.nLength = sizeof(SECURITY_ATTRIBUTES); // размер структуры
		sa.lpSecurityDescriptor = 0; // система защиты по умолчанию
		sa.bInheritHandle = FALSE; // дескриптор объекта ядра "Файл"
					// ненаследуемый !!!
		HANDLE hHandle = CreateFile(
			TEXT("Test.txt"), // путь к файлу
			GENERIC_READ, //только чтение
			0, // запрет совместного использования файла
			&sa, // система защиты
			OPEN_EXISTING, // открытие существующего файла
			FILE_ATTRIBUTE_NORMAL, // атрибутов нет
			NULL // обычно используется в клиент-серверном приложении
			);
		if ( hHandle == INVALID_HANDLE_VALUE ) //ошибка открытия файла
			return;
		STARTUPINFO startin = {sizeof(STARTUPINFO)};
		PROCESS_INFORMATION info = {0};
		TCHAR cmd[20];
		// дескриптор объекта ядра передается дочернему процессу через
		// командную строку
		wsprintf(cmd, TEXT("%s %d"), TEXT("child.exe"), hHandle);
		CreateProcess(
			NULL, // в качестве имени файла выступает первая выделенная
				// пробелами лексема из 2-го параметра
			cmd, // Указатель командной строки, содержащей, в том числе,
				// имя исполняемого файла
			NULL, // атрибуты защиты процесса по умолчанию
			NULL, // атрибуты защиты потока по умолчанию
			TRUE, // наследование разрешено !!!
			0, // класс приоритета по умолчанию - NORMAL_PRIORITY_CLASS
			NULL, // среда окружения наследуется у родительского
				// процесса
			NULL, // текущий каталог наследуется у родительского
				// процесса
			&startin, // способ отображения главного окна, а также
					// размер и заголовок окна
			&info // информация о порождённом процессе
			);
		CloseHandle(hHandle); // закрывается описатель объекта ядра «файл»
		CloseHandle(info.hThread); // закрывается описатель объекта ядра
					// «поток»
		CloseHandle(info.hProcess); // закрывается описатель объекта ядра
					// «процесс»
	}
	else if(id == IDC_BUTTON3)
	{
		SECURITY_ATTRIBUTES sa;
		sa.nLength = sizeof(SECURITY_ATTRIBUTES); // размер структуры
		sa.lpSecurityDescriptor = 0; // система защиты по умолчанию
```

```
            sa.bInheritHandle = TRUE; // дескриптор объекта ядра "Файл"
                                      // наследуемый !!!



            HANDLE hHandle = CreateFile(
                  TEXT("Test.txt"), // путь к файлу
                  GENERIC_READ, // только чтение
                  0, // запрет совместного использования файла
                  &sa, // система защиты
                  OPEN_EXISTING, // открытие существующего файла
                  FILE_ATTRIBUTE_NORMAL, // атрибутов нет
                  NULL // обычно используется в клиент-серверном приложении
                  );
            if ( hHandle == INVALID_HANDLE_VALUE ) // ошибка открытия файла
                  return;
            STARTUPINFO startin = {sizeof(STARTUPINFO)};
            PROCESS_INFORMATION info={0};
            TCHAR cmd[20];
            // дескриптор объекта ядра передается дочернему процессу через
            // командную строку
            wsprintf(cmd, TEXT("%s %d"), TEXT("child.exe"), hHandle);
            CreateProcess(
                  NULL, // в качестве имени файла выступает первая выделенная
                        // пробелами лексема из 2-го параметра
                  cmd, // Указатель командной строки, содержащей, в том числе,
                        // имя исполняемого файла
                  NULL, // атрибуты защиты процесса по умолчанию
                  NULL, // атрибуты защиты потока по умолчанию
                  FALSE, // наследование запрещено !!!
                  0, // класс приоритета по умолчанию - NORMAL_PRIORITY_CLASS
                  NULL, // среда окружения наследуется у родительского
                        // процесса
                  NULL, // текущий каталог наследуется у родительского
                        // процесса
                  &startin, // способ отображения главного окна, а также
                              // размер и заголовок окна
                  &info // информация о порождённом процессе
                  );
            CloseHandle(hHandle); // закрывается описатель объекта ядра «файл»
            CloseHandle(info.hThread); // закрывается описатель объекта ядра
                                       // «поток»
            CloseHandle(info.hProcess); // закрывается описатель объекта ядра
                                        // «процесс»
      }
}

BOOL CALLBACK CInheritHandleDlg::DlgProc(HWND hwnd, UINT message,
                                         WPARAM wParam, LPARAM lParam)
{
      switch(message)
      {
            HANDLE_MSG(hwnd, WM_CLOSE, ptr->Cls_OnClose);
            HANDLE_MSG(hwnd, WM_COMMAND, ptr->Cls_OnCommand);
      }
      return FALSE;
}

                              // InheritHandle.cpp

#include "InheritHandleDlg.h"
```

```
int WINAPI _tWinMain(HINSTANCE hInst, HINSTANCE hPrev, LPTSTR lpszCmdLine,
                     int nCmdShow)
{

      CInheritHandleDlg dlg;
      return DialogBox(hInst, MAKEINTRESOURCE(IDD_DIALOG1), NULL,
                           CInheritHandleDlg::DlgProc);
}
```

Исходный код приложения, в котором дочерний процесс получает наследуемый дескриптор объекта ядра «файл», представлен ниже.

```
                                    // header.h
#pragma once
#include <windows.h>
#include <windowsX.h>
#include <tchar.h>
#include "resource.h"

                                // InheritHandleDlg.h
#pragma once
#include "header.h"

class CInheritHandleDlg
{
public:
      CInheritHandleDlg(void);
public:
      ~CInheritHandleDlg(void);
      static BOOL CALLBACK DlgProc(HWND hWnd, UINT mes, WPARAM wp, LPARAM lp);
      static CInheritHandleDlg* ptr;
      void Cls_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify);
      void Cls_OnClose(HWND hwnd);
      TCHAR CommandLine[MAX_PATH];
      HANDLE hHandle;
};

                               // InheritHandleDlg.cpp

#include "InheritHandleDlg.h"

CInheritHandleDlg* CInheritHandleDlg::ptr = NULL;

CInheritHandleDlg::CInheritHandleDlg(void)
{
      ptr = this;
      _tcscpy(CommandLine, GetCommandLine()); // получим командную строку
      TCHAR seps[]   = TEXT(" ");
      TCHAR *token, *last;
      token = _tcstok(CommandLine, seps); // разбиваем командную строку на
                                          // лексемы (разделитель "пробел")
      while( token != NULL )
```

```
	{
		token = _tcstok(NULL, seps); // разбиваем командную строку на
					     // лексемы (разделитель "пробел")


		if(token)
			last = token; // указатель на последнюю лексему, т.е. на
				      // дескриптор объекта ядра
	}
	hHandle = HANDLE(_tstoi(last));
}

CInheritHandleDlg::~CInheritHandleDlg(void)
{

}

void CInheritHandleDlg::Cls_OnClose(HWND hwnd)
{
	EndDialog(hwnd, 0);
}

void MessageAboutError(DWORD dwError)
{
	LPVOID lpMsgBuf = NULL;
	TCHAR szBuf[300];
	//Функция FormatMessage форматирует строку сообщения
	BOOL fOK = FormatMessage(
		FORMAT_MESSAGE_FROM_SYSTEM /* флаг сообщает функции, что нужна
		строка, соответствующая коду ошибки, определённому в системе */
		| FORMAT_MESSAGE_ALLOCATE_BUFFER, // нужно выделить
			// соответствующий блок памяти для хранения текста
		NULL, // указатель на строку, содержащую текст сообщения
		dwError, // код ошибки
		MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // язык, на котором
		// выводится описание ошибки (язык пользователя по умолчанию)
		(LPTSTR)&lpMsgBuf, // указатель на буфер, в который будет записан
				   // текст сообщения
		0, // минимальный размер буфера для выделения памяти - память
			// выделяет система
		NULL // список аргументов форматирования
		);
	if(lpMsgBuf != NULL)
	{
		wsprintf(szBuf, TEXT("Ошибка %d: %s"), dwError, lpMsgBuf);
		MessageBox(NULL,  szBuf, TEXT("Сообщение об ошибке"),
				MB_OK | MB_ICONSTOP);
		LocalFree(lpMsgBuf); // освобождаем память, выделенную системой
	}
}

void CInheritHandleDlg::Cls_OnCommand(HWND hwnd, int id, HWND hwndCtl,
					UINT codeNotify)
{
	if(id == IDC_BUTTON1)
	{
		TCHAR buf[100] = {0};
		DWORD dwRead = 0;
		// читаем строку из файла, дескриптор которого был унаследован от
		// "родителя"
```

```
            ReadFile(hHandle, buf, sizeof(buf), &dwRead, NULL);
            DWORD dwError = GetLastError();



            if ( dwError != 0 ) // Ошибка чтения из файла
                  MessageAboutError(dwError);
            else
            {
                  MessageBox(hwnd, buf, TEXT("Наследование описателя"),
                              MB_OK | MB_ICONINFORMATION);
                  // закрываем описатель объекта ядра «файл»
                  CloseHandle(hHandle);
            }
      }
}

BOOL CALLBACK CInheritHandleDlg::DlgProc(HWND hwnd, UINT message,
                                          WPARAM wParam, LPARAM lParam)
{
      switch(message)
      {
            HANDLE MSG(hwnd, WM CLOSE, ptr->Cls OnClose);
            HANDLE_MSG(hwnd, WM_COMMAND, ptr->Cls_OnCommand);
      }
      return FALSE;
}

                              // InheritHandleDlg.cpp
#include "InheritHandleDlg.h"

int WINAPI _tWinMain(HINSTANCE hInst, HINSTANCE hPrev, LPTSTR lpszCmdLine,
                  int nCmdShow)
{
      CInheritHandleDlg dlg;
      return DialogBox(hInst, MAKEINTRESOURCE(IDD_DIALOG1), NULL,
                        CInheritHandleDlg::DlgProc);
}
```

Как видно из вышеприведенного кода, дочерний процесс получает наследуемый описатель объекта ядра «файл», переданный родительским процессом посредством командной строки. Наличие описателя позволяет читать данные из файла, используя функцию API **ReadFile**.

```
BOOL WINAPI ReadFile(
      HANDLE hFile, // дескриптор файла
      LPVOID lpBuffer, // указатель на буфер, в который будут записаны данные,
      // прочтённые из файла
      DWORD nNumberOfBytesToRead, // максимальное количество байт для чтения
      LPDWORD lpNumberOfBytesRead, // указатель на переменную, в которую будет
```

```
      // записано фактическое количество прочтённых байт
      LPOVERLAPPED lpOverlapped // указатель на структуру OVERLAPPED
);
```

3.3. Запуск обособленных дочерних процессов

Чаще всего приложение создает другие процессы как обособленные. Это означает, что после создания и запуска нового процесса родительскому процессу нет нужды с ним взаимодействовать или ждать, пока тот закончит работу. Чтобы создать обособленный дочерний процесс необходимо вызовом **CloseHandle** закрыть свои описатели, связанные с новым процессом и его первичным потоком. Приведенный ниже фрагмент кода демонстрирует вышесказанное.

```
STARTUPINFO s = {sizeof(STARTUPINFO)};
PROCESS_INFORMATION p;
TCHAR buffer[] = TEXT("Calc.exe");
BOOL fs = CreateProcess(NULL, buffer, NULL, NULL, FALSE, 0,NULL, NULL, &s,
&p);
if (fs)
{
      CloseHandle(p.hThread);
      CloseHandle(p.hProcess);
}
```

4. Получение списка запущенных в системе процессов

Ознакомить слушателей с понятием **снимка системы** (**snapshot**). Отметить, что снимок системы необходим для того, чтобы получить информацию обо всех запущенных в системе процессах, их потоках, используемой процессами памяти и т.д.

Для получения снимка системы необходимо воспользоваться функцией **CreateToolhelp32Snapshot**.

```
HANDLE WINAPI CreateToolhelp32Snapshot(
      DWORD dwFlags, // набор флагов, определяющих, какие параметры будут
      // записаны в снимок системы

      DWORD th32ProcessID // идентификатор процесса, о котором необходимо
      // получить информацию. Если данный параметр равен 0, то запрашивается
      // информация о текущем процессе. Этот параметр имеет смысл, когда
      // необходимо получить список куч и модулей, используемых конкретным
      // процессом, в противном случае параметр игнорируется
);
```

Первый параметр данной функции может принимать следующие значения:

Макрос	Описание
TH32CS_INHERIT	Снимок может наследоваться дочерними процессами
TH32CS_SNAPALL	Комбинация флагов TH32CS_SNAPHEAPLIST, TH32CS_SNAPMODULE, TH32CS_SNAPPROCESS и TH32CS_SNAPTHREAD
TH32CS_SNAPHEAPLIST	В снимок включается список куч, принадлежащих процессу
TH32CS_SNAPMODULE	В снимок включается список модулей, принадлежащих процессу
TH32CS_SNAPPROCESS	В снимок включается список процессов, запущенных в системе
TH32CS_SNAPTHREAD	В снимок включается список потоков

Для получения списка запущенных процессов из снимка системы следует воспользоваться функциями **Process32First** и **Process32Next**.

При этом сначала необходимо вызвать функцию **Process32First** для получения информации о первом процессе в списке.

```
BOOL WINAPI Process32First(
  HANDLE hSnapshot, // дескриптор снимка системы
  LPPROCESSENTRY32 lppe // указатель на структуру PROCESSENTRY32, которая
      // будет заполнена информацией о первом процессе в снимке
);
```

Чтобы получить информацию об остальных процессах в снимке системы необходимо многократно вызывать функцию **Process32Next.**

```
BOOL WINAPI Process32Next(
  HANDLE hSnapshot, // дескриптор снимка системы

  LPPROCESSENTRY32 lppe // указатель на структуру PROCESSENTRY32, которая
      // будет заполнена информацией о следующем процессе в снимке
);
```

```
typedef struct tagPROCESSENTRY32
{
      DWORD dwSize;  // размер структуры в байтах
      DWORD cntUsage;  // не используется и должен быть 0
      DWORD th32ProcessID;  // идентификатор процесса
      ULONG_PTR th32DefaultHeapID;  // не используется и должен быть 0
      DWORD th32ModuleID;  // не используется и должен быть 0
      DWORD cntThreads;  // количество потоков в процессе
      DWORD th32ParentProcessID;  // идентификатор родительского процесса
      LONG pcPriClassBase;  // базовый приоритет всех потоков, созданных в
                            // процессе
      DWORD dwFlags;  // не используется и должен быть 0
      TCHAR szExeFile[MAX_PATH]; // название exe-файла для данного процесса
} PROCESSENTRY32, *PPROCESSENTRY32;
```

Следующий фрагмент кода демонстрирует механизм получения списка запущенных процессов.

```
HANDLE h = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
if (h == NULL)
{
      MessageBox(hDialog, TEXT("Error!"), 0, MB_OK);
            return;
}

PROCESSENTRY32 process;
bool flag = Process32First(h, &process);
if (flag)
{
      if (!strcmp(process.szExeFile, TEXT("calc.exe")))
      {

            HANDLE hPR = OpenProcess(PROCESS_ALL_ACCESS, FALSE,
                        process.th32ProcessID) ;
            TerminateProcess(hPR,0);
      }
      while (Process32Next(h,&process))
```

```
	{
		MessageBox(hDialog, process.szExeFile,
				TEXT("Перечисление процессов"), MB_OK);


		if (!strcmp(process.szExeFile, TEXT("calc.exe")))
		{
			HANDLE hPR = OpenProcess(PROCESS_ALL_ACCESS, FALSE,
					process.th32ProcessID) ;
			TerminateProcess(hPR,0);
		}
	}
}
```

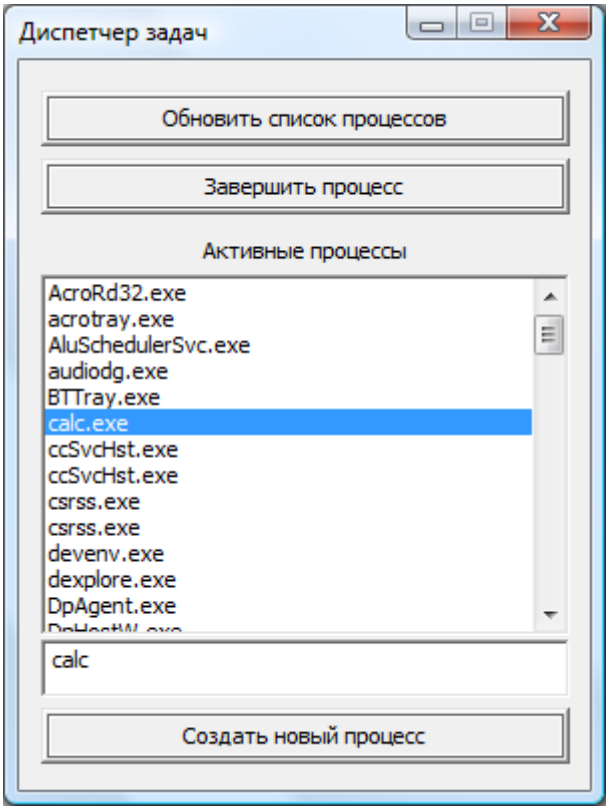
В вышеприведенном примере используется функция **OpenProcess** для получения описателя процесса по его идентификатору.

```
HANDLE OpenProcess(
  DWORD dwDesiredAccess, // желаемый уровень доступа (PROCESS_ALL_ACCESS -
      // полный доступ)
  BOOL bInheritHandle, // если значение параметра равно TRUE, тогда
      // полученный дескриптор можно передавать между процессами, в противном
      // случае дескриптор ненаследуемый
  DWORD dwProcessId // идентификатор процесса
);
```

5. Подведение итогов

Подвести общие итоги занятия. Ещё раз акцентировать внимание слушателей на многозадачности. Отметить, из каких компонент состоит процесс. Повторить понятие дочернего процесса. Вкратце напомнить слушателям механизм наследования описателей объектов ядра. Подчеркнуть, для каких целей необходимо создавать снимок системы. Акцентировать внимание слушателей на наиболее тонких моментах изложенной темы.

6. Домашнее задание



Написать приложение, позволяющее:

- сформировать список запущенных процессов системы;
- завершить процесс, выбранный в списке;
- обновить список процессов;
- создать новый процесс (путь к исполняемому файлу вводится в текстовое поле ввода).