Белорусский государственный университет
Факультет прикладной математики и информатики
Кафедра технологии программирования
доцент Побегайло А.П.

3. Процессы.

3.1. Процессы и их создание в Windows.

В Windows под *процессом* понимается объект ядра, которому принадлежат системные ресурсы, используемые приложением. Поэтому можно сказать, что в Windows процессом является приложение. Выполнение каждого процесса начинается с первичного потока. В процессе своего исполнения процесс может создавать другие потоки. Исполнение процесса заканчивается при завершении работы всех его потоков. Процесс может быть также завершен вызовом функций *ExitProcess* и *TerminateProcess*, которые будут рассмотрены в следующем параграфе.

Новый процесс в Windows создается вызовом функции *CreateProcess*, которая имеет следующий прототип:

```
BOOL CreateProcess(
        LPCTSTR                 lpApplicationName,          // имя исполняемого модуля
        LPTSTR                  lpCommandLine,              // командная строка
        LPSECURITY_ATTRIBUTES   lpProcessAttributes,        // атрибуты защиты процесса
        LPSECURITY_ATTRIBUTES   lpThreadAttributes,         // атрибуты защиты потока
        BOOL                    bInheritHandle,             // наследуемый ли дескриптор
        DWORD                   dwCreationFlags,            // флаги создания процесса
        LPVOID                  lpEnvironment,              // блок новой среды окружения
        LPCTSTR                 lpCurrentDirectory,         // текущий каталог
        LPSTARTUPINFO           lpStartUpInfo,              // вид главного окна
        LPPROCESS_INFORMATION   lpProcessInformation        // информация о процессе
);
```

Функция *CreateProcess* возвращает значение TRUE, если процесс был создан успешно. В противном случае эта функция возвращает значение FALSE. Процесс, который создает новый процесс, называется *родительским процессом* (parent process) по отношению к создаваемому процессу. Новый же процесс, который создается другим процессом, называется *дочерним процессом* (child process) по отношению к процессу родителю.

Сейчас мы опишем только назначение некоторых параметров функции *CreateProcess*. Остальные параметры этой функции будут описываться по мере их использования. Первый параметр lpApplicationName определяет строку с именем exe-файла, который будет запускаться при создании нового процесса. Эта строка должна заканчиваться нулем и содержать полный путь к запускаемому файлу. Для примера рассмотрим следующую программу, которая выводит на консоль свое имя и параметры.

Программа 3.1.

```
// Консольный процесс, который выводит на консоль свое имя и параметры

#include <conio.h>

int main(int argc, char *argv[])
{
        int i;

        _cputs("I am created.");

        _cputs("\nMy name is: ");
        _cputs(argv[0]);

        for (i = 1; i < argc; i++)
                _cprintf ("\n My %d parameter =  %s", i, argv[i]);
```

```
	_cputs("\nPress any key to finish.\n");
	_getch();

	return 0;
}
```

Создадим из этой программы exe-файл, который расположим на диске C и назовем ConsoleProcess.exe. Тогда этот exe-файл может быть запущен из другого приложения следующим образом.

Программа 3.2.

```
// Пример консольного процесса, который создает другое консольное приложение
// с новой консолью и ждет завершения работы этого приложения.

#include <windows.h>
#include <conio.h>

int main()
{
	char lpszAppName[] = "C:\\ConsoleProcess.exe";

	STARTUPINFO si;
	PROCESS_INFORMATION piApp;

	ZeroMemory(&si, sizeof(STARTUPINFO));
	si.cb = sizeof(STARTUPINFO);

	// создаем новый консольный процесс
	if (!CreateProcess(lpszAppName, NULL, NULL, NULL, FALSE,
			CREATE_NEW_CONSOLE, NULL, NULL, &si, &piApp))
	{
		_cputs("The new process is not created.\n");
		_cputs("Check a name of the process.\n");
		_cputs("Press any key to finish.\n");
		_getch();
		return 0;
	}

	_cputs("The new process is created.\n");
	// ждем завершения созданного прцесса
	WaitForSingleObject(piApp.hProcess, INFINITE);
	// закрываем дескрипторы этого процесса в текущем процессе
	CloseHandle(piApp.hThread);
	CloseHandle(piApp.hProcess);

	return 0;
}
```

Отметим в последней программе два момента. Во-первых, перед запуском консольного процесса ConsoleProcess.exe все поля структуры si типа STARTUPINFO должны заполняться нулями. Это делается при помощи вызова функции *ZeroMemory*, которая предназначена для этой цели и имеет следующий прототип:

```
	VOID ZeroMemory(
		PVOID		Destination,		// адрес блока памяти
		SIZE_T		Length			// длина блока памяти
	);
```

В этом случае вид главного окна запускаемого приложения определяется по умолчанию самой операционной системой Windows. Во-вторых, в параметре dwCreationFlags устанавливается флаг CREATE_NEW_CONSOLE. Это говорит системе о том, что для нового создаваемого процесса должна быть создана новая консоль. Если этот параметр будет равен NULL, то новая консоль для запускаемого процесса не создается и весь консольный вывод нового процесса будет направляться в консоль родительского процесса.

Структура piApp типа PROCESS_INFORMATION содержит идентификаторы и дескрипторы нового создаваемого процесса и его главного потока. Мы не используем эти дескрипторы в нашей программе и поэтому закрываем их. Значение FALSE параметра bInheritHandle говорит о том, что эти дескрипторы не являются наследуемыми. О наследовании дескрипторов мы поговорим подробнее в одном из следующих параграфов этой главы.

Теперь запустим наш новый консольный процесс другим способом, используя второй параметр функции *CreateProcess*. Это можно сделать при помощи следующей программы.

Программа 3.3.

```
// Пример процесса, который создает новое консольное приложение с новой консолью

#include <windows.h>
#include <conio.h>

int main()
{
	char lpszCommandLine[] = "C:\\01-1-ConsoleProcess.exe p1 p2 p3";

	STARTUPINFO si;
	PROCESS_INFORMATION piCom;

	ZeroMemory(&si, sizeof(STARTUPINFO));
	si.cb = sizeof(STARTUPINFO);

	// создаем новый консольный процесс
	CreateProcess(NULL, lpszCommandLine, NULL, NULL, FALSE,
			CREATE_NEW_CONSOLE, NULL, NULL, &si, &piCom);
	// закрываем дескрипторы этого процесса
	CloseHandle(piCom.hThread);
	CloseHandle(piCom.hProcess);

	_cputs("The new process is created.\n");
	_cputs("Press any key to finish.\n");
	_getch();

	return 0;
}
```

Отличие этой программы от предыдущей состоит в том, что мы передаем системе имя нового процесса и его параметры через командную строку. В этом случае имя нового процесса может и не содержать полный путь к exe-файлу, а только имя самого exe-файла. При использовании параметра lpCommandLine система для запуска нового процесса осуществляет поиск требуемого exe-файла в следующей последовательности каталогов:

– каталог из которого запущено приложение;
– текущий каталог родительского процесса;
– системный каталог Windows;
– каталог Windows;
– каталоги, которые перечислены в переменной PATH среды окружения.

Для иллюстрации сказанного запустим приложение Notepad.exe, используя командную строку. Программа, запускающая блокнот из командной строки, выглядит следующим образом.

Программа 3.4.

```
// Пример запуска процесса Notepad

#include <windows.h>
#include <iostream>
using namespace std;

int main()
{
	STARTUPINFO si;
	PROCESS_INFORMATION pi;

	// заполняем значения структуры STARTUPINFO по умолчанию
	ZeroMemory(&si, sizeof(STARTUPINFO));
	si.cb = sizeof(STARTUPINFO);

	// запускаем процесс Notepad
	if (!CreateProcess(
			NULL,		// имя не задаем
			"Notepad.exe",	// командная строка, первая лексема указывает имя программы
			NULL,		// атрибуты защиты процесса устанавливаем по умолчанию
			NULL,		// атрибуты защиты первичного потока по умолчанию
			FALSE,		// дескрипторы текущего процесса не наследуются новым процессом
			0,		// по умолчанию NORMAL_PRIORITY_CLASS
			NULL,		// используем среду окружения вызывающего процесса
			NULL,		// текущий диск и каталог, как и в вызывающем процессе
			&si,		// вид главного окна - по умолчанию
			&pi		// здесь будут дескрипторы и идентификаторы
					// нового процесса и его первичного потока
			)
		)
	{
		cout << "The mew process is not created." << endl
			<< "Check a name of the process." << endl;
		return 0;
	}

	Sleep(1000);	// немного подождем и закончим свою работу

	// закроем дескрипторы запущенного процесса в текущем процессе
	CloseHandle(pi.hThread);
	CloseHandle(pi.hProcess);

	return 0;
}
```

3.2. Завершение процессов.

Процесс может завершить свою работу вызовом функции *ExitProcess*, которая имеет следующий прототип:

```
VOID ExitProcess(
	UINT		uExitCode			// код возврата для всех потоков
);
```

При вызове функции *ExitProcess* завершаются все потоки процесса с кодом возврата, который является параметром этой функции. Приведем пример программы, которая завершает свою работу вызовом функции *ExitProcess*.

Программа 3.5.

```
// Пример завершения процесса функцией ExitProcess

#include <windows.h>
#include <iostream>
using namespace std;

volatile UINT count;
volatile char c;

void thread()
{
	for ( ; ; )
	{
		count++;
		Sleep(100);
	}
}

int main()
{
	HANDLE	hThread;
	DWORD	IDThread;

	hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)thread, NULL, 0, &IDThread);
	if (hThread == NULL)
		return GetLastError();

	for ( ; ; )
	{
		cout << "Input 'y' to display the count or 'e' to exit: ";
		cin >> (char)c;
		if (c == 'y')
			cout << "count = " << count << endl;
		if (c == 'e')
			ExitProcess(1);
	}
}
```

Один процесс может завершить другой процесс при помощи вызова функции *TerminateProcess*, которая имеет следующий прототитп:

```
BOOL TerminateProcess(
	HANDLE	hProcess,
	UINT	uExitCode
);
```

Если функция *TerminateProcess* выполнилась успешно, то она возвращает значение равно TRUE. В противном случае возвращаемое значение равно FALSE. Функция *TerminateProcess* завершает работу процесса, но не освобождает все ресурсы, принадлежащие этому процессу. Поэтому эта функция должна вызываться только в аварийных ситуациях при зависании процесса.

Приведем программу, которая демонстрируют работу функции *TerminateProcess*. Для этого сначала создадим бесконечный процесс-счетчик, который назовем ConsoleProcess.exe и расположим на диске C.

Программа 3.6.

```
// Пример бесконечного процесса

#include <windows.h>
#include <iostream>
using namespace std;
int count;

void main()
{
	for ( ; ; )
	{
		count++;
		Sleep(1000);
		cout << "count = " << count << endl;
	}
}
```

Ниже приведена программа, которая создает этот процесс, а потом завершает его по требованию пользователя.

Программа 3.7.

```
// Пример процесса, который создает другое консольное приложение с новой консолью,
// а потом завершает его при помощи функции TerminateProcess

#include <windows.h>
#include <conio.h>

int main()
{
	char lpszAppName[] = "C:\\ConsoleProcess.exe";

	STARTUPINFO si;
	PROCESS_INFORMATION pi;

	ZeroMemory(&si, sizeof(STARTUPINFO));
	si.cb=sizeof(STARTUPINFO);

	// создаем новый консольный процесс
	if (!CreateProcess(lpszAppName, NULL, NULL, NULL, FALSE,
			CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi))
	{
		_cputs("The new process is not created.\n");
		_cputs("Check a name of the process.\n");
		_cputs("Press any key to finish.\n");
		_getch();
		return 0;
	}

	_cputs("The new process is created.\n");

	while (true)
	{
		char c;

		_cputs("Input 't' to terminate the new console process: ");
```

```
                c = _getch();
                if (c == 't')
                {
                        _cputs("t\n");
                                // завершаем новый процесс
                        TerminateProcess(pi.hProcess,1);
                        break;
                }
        }


        // закрываем дескрипторы нового процесса в текущем процессе
        CloseHandle(pi.hThread);
        CloseHandle(pi.hProcess);

        return 0;
}
```

3.3. Обслуживание потоков

Отметим, что операционная система Windows обслуживает потоки, то есть процессорное время выделяется потокам в соответствии с их приоритетами. Приоритеты потоков в Windows определяются относительно приоритета процесса, в рамках которого они исполняются, и изменяются от 0 (низший приоритет) до 31 (высший приоритет). Приоритет процессов устанавливается при их создании функцией *CreateProcess*, используя параметр dwCreationFlags этой функции. Для установки приоритета процесса, в этом параметре нужно установить один из следующих флагов.

IDLE_PRIORITY_CLASS
BELOW_NORMAL_PRIORITY_CLASS (используется только в Windows 2000)
NORMAL_PRIORITY_CLASS
ABOVE_NORMAL_PRIORITY_CLASS (используется только в Windows 2000)
HIGH_PRIORITY_CLASS
REAL_TIME_PRIORITY_CLASS

Рассмотрим правила для назначения приоритетов процессам в Windows. Предполагается, что операционная система Windows различает четыре типа процессов в соответствии с их приоритетами: фоновые процессы, процессы с нормальным приоритетом, процессы с высоким приоритетом и процессы реального времени. Рассмотрим каждый из этих типов процессов чуть подробнее.

Фоновые процессы выполняют свою работу, когда нет активных пользовательских процессов. Обычно, эти процессы следят за состоянием системы. Приоритет таких процессов устанавливается флагом IDLE_PRIORITY_CLASS.

Процессы с нормальным приоритетом это обычные пользовательские процессы. Приоритет таких процессов устанавливается флагом NORMAL_PRIORITY_CLASS. Этот приоритет также назначается пользовательским процессам по умолчанию. В Windows 2000 приоритет обычных пользовательских процессов может также устанавливаться флагами BELOW_NORMAL_PRIORITY_CLASS или ABOVE_NORMAL_PRIORITY_CLASS, которые соответственно немного повышают или понижают приоритет пользовательского процесса.

Процессы с высоким приоритетом это такие пользовательские процессы, для которых требуется быстрая реакция на некоторые события. Приоритет таких процессов устанавливается флагом HIGH_PRIORITY_CLASS. Эти процессы должны содержать небольшой программный код и выполняться очень быстро, чтобы не замедлять работу системы.

К последнему типу процессов относятся процессы реального времени. Приоритет таких процессов устанавливается флагом REAL_TIME_PRIORITY_CLASS. Работа таких процессов обычно происходит в масштабе реального времени и связана с реакцией на внешние события. Эти процессы должны работать непосредственно с аппаратурой компьютера.

Приоритет процесса можно изменить при помощи функции *SetPriorityClass*, которая имеет следующий прототип:

```
BOOL SetPriorityClass(
```

```
        HANDLE      hProcess,           // дескриптор процесса
        DWORD       dwPriorityClass     // приоритет
);
```

При успешном завершении функция *SetPriorityClass* возвращает значение TRUE, в противном случае значение – FALSE. Параметр dwPriorityClass этой функции должен быть равен одному из флагов, которые приведены выше.

Узнать приоритет процесса можно посредством вызова функции *GetPriorityClass*, которая имеет следующий прототип:

```
DWORD GetPriorityClass(
        HANDLE      hProcess            // дескриптор процесса
);
```

При успешном завершении эта функция возвращает флаг установленного приоритета процесса, в противном случае возвращаемое значение равно нулю.

Ниже приведена программа, которая демонстрирует работу функций *SetPriorityClass* и *GetPriorityClass*.

Программа 3.8.

```
// Пример работы функций SetPriorityClass и GetPriorityClass

#include <windows.h>
#include <conio.h>

int main()
{
        HANDLE      hProcess;
        DWORD       dwPriority;

        // получаем псевдодескриптор текущего потока
        hProcess = GetCurrentProcess();

        // узнаем приоритет текущего процесса
        dwPriority = GetPriorityClass(hProcess);
        _cprintf("The priority of the process = %d.\n", dwPriority);

        // устанавливаем фоновый приоритет текущего процесса
        if (!SetPriorityClass(hProcess, IDLE_PRIORITY_CLASS))
        {
                _cputs("Set priority class failed.\n");
                _cputs("Press any key to exit.\n");
                _getch();
                return GetLastError();
        }
        dwPriority = GetPriorityClass(hProcess);
        _cprintf("The priority of the process = %d.\n", dwPriority);

        // устанавливаем высокий приоритет текущего процесса
        if (!SetPriorityClass(hProcess, HIGH_PRIORITY_CLASS))
        {
                _cputs("Set priority class failed.\n");
                _cputs("Press any key to exit.\n");
                _getch();
                return GetLastError();
        }
        dwPriority = GetPriorityClass(hProcess);
        _cprintf("The priority of the process = %d.\n", dwPriority);
```

```
    _cputs("Press any key to exit.\n");
    _getch();

    return 0;
}
```

Отметим в связи с этой программой два момента. Во-первых, числовые значения флагов не соответствуют числовым значениям приоритетов процессов. Так, например, числовое значение флага IDLE_PRIORITY_CLASS больше чем числовое значение флага NORMAL_PRIORITY_CLASS. Но система считает, что приоритет нормального процесса выше, чем приоритет фонового процесса. Во-вторых, в этой программе мы использовали функцию *GetCurrentProcess*, которая имеет следующий прототип:

```
HANLDE GetCurrentProcess(VOID);
```

и возвращает псевдодескриптор текущего процесса. *Псевдодескриптор текущего процесса* отличается от настоящего дескриптора процесса тем, что он может использоваться только самим текущим процессом и, следовательно, не может наследоваться другими процессами.

Теперь перейдем к приоритетам потоков, задание которых в Windows довольно запутанное. Приоритет потока, который учитывается системой при выделении потокам процессорного времени, называется *базовым* (base) или *основным приоритетом потока*. Всего существует 32 базовых приоритета потока, значения которых изменяются от 0 до 31. Для каждого базового приоритета существует очередь потоков. При диспетчеризации потоков квант процессорного времени выделяется потоку, который стоит первым в очереди с наивысшим базовым приоритетом. Базовый приоритет потока определяется как сумма приоритета процесса и *уровня приоритета потока*, который может принимать одно из следующих значений, которые мы разобьем на две группы.

```
THREAD_PRIORITY_LOWEST
THREAD_PRIORITY_BELOW_NORMAL
THREAD_PRIORITY_NORMAL
THREAD_PRIORITY_ABOVE_NORMAL
THREAD_PRIORITY_HIGHEST

THREAD_PRIORITY_IDLE
THREAD_PRIORITY_TIME_CRITICAL
```

Значения уровня приоритета потока из первой группы в сумме с приоритетом процесса, в рамках которого этот поток выполняется, уменьшают, оставляют неизменным или увеличивают значение базового приоритета потока соответственно на –2, -1, 0, 1, 2. Уровень приоритета потока THREAD_PRIORITY_IDLE устанавливает базовый приоритет потока равным 16, если приоритет процесса, в рамках которого выполняется поток, равен REAL_TIME_PRIORITY_CLASS и 1 – в остальных случаях. Уровень приоритета потока THREAD_PRIORITY_TIME_CRITICAL устанавливает базовый приоритет потока равным 31, если приоритет процесса, в рамках которого выполняется поток, равен REAL_TIME_PRIORITY_CLASS и 15 – в остальных случаях.

При создании потока его базовый приоритет устанавливается как сумма приоритета процесса, в рамках которого этот поток выполняется, и уровня приоритета потока THREAD_PRIORITY_NORMAL. Для изменения приоритета потока используется функция *SetThreadPriority*, которая имеет следующий прототип:

```
BOOL SetThreadPriority(
        HANDLE      hThread,        // дескриптор потока
        Int         nPriority       // уровень приоритета потока
);
```

При удачном завершении функция *SetThreadPriority* возвращает значение TRUE, в противном случае – FALSE. Параметр nPriority этой функции должен быть равен одному из перечисленных уровней приоритетов.

Узнать уровень приоритета потока можно посредством вызова функции *GetThreadPriority*, которая имеет следующий прототип:

```
DWORD GetThreadPriority(
```

```
        HANDLE      hThread                 // дескриптор потока
    );
```

При успешном завершении эта функция возвращает одно из значений уровня приоритета, в противном случае функция *GetThreadPriority* возвращает значение THREAD_PRIORITY_ERROR_RETURN.

Ниже приведена программа, которая демонстрирует работу функций *SetThreadPriority* и *GetThreadPriority*.

Программа 3.9.

```
// Пример работы функций SetThreadPriority и GetThreadPriority

#include <windows.h>
#include <conio.h>

int main()
{
    HANDLE      hThread;
    DWORD       dwPriority;

    // получаем псевдодескриптор текущего потока
    hThread = GetCurrentThread();

    // узнаем уровень приоритета текущего процесса
    dwPriority = GetThreadPriority(hThread);
    _cprintf("The priority level of the thread = %d.\n", dwPriority);

    // понижаем приоритет текущего потока
    if (!SetThreadPriority(hThread, THREAD_PRIORITY_LOWEST))
    {
        _cputs("Set thread priority failed.\n");
        _cputs("Press any key to exit.\n");
        _getch();
        return GetLastError();
    }
    // узнаем уровень приоритет текущего потока
    dwPriority = GetThreadPriority(hThread);
    _cprintf("The priority level of the thread = %d.\n", dwPriority);

    // повышаем приоритет текущего потока
    if (!SetThreadPriority(hThread, THREAD_PRIORITY_HIGHEST))
    {
        _cputs("Set thread priority failed.\n");
        _cputs("Press any key to exit.\n");
        _getch();
        return GetLastError();
    }
    // узнаем уровень приоритета текущего потока
    dwPriority = GetThreadPriority(hThread);
    _cprintf("The priority level of the thread = %d.\n", dwPriority);

    _cputs("Press any key to exit.\n");
    _getch();

    return 0;
}
```

Отметим в этой программе использование функции *GetCurrentThread*, которая имеет следующий прототип:

```
HANLDE GetCurrentThread(VOID);
```

и возвращает псевдодескриптор текущего потока. *Псевдодескриптор текущего потока* отличается от настоящего дескриптора потока тем, что он может использоваться только самим текущим потоком и поэтому не может наследоваться другими процессами.

Базовый приоритет потока может динамически изменяться системой, если этот приоритет находится в пределах между уровнями 0 и 15. При получении потоком сообщения или при его переходе в состояние готовности, система повышает базовый приоритет этого потока на 2. В процессе выполнения базовый приоритет такого потока понижается на 1, с каждым отработанным квантом времени, но никогда не опускается ниже исходного базового приоритета. В операционной системе Windows 2000 возможно программное управление режимом динамического изменения базовых приоритетов потоков. Для отмены или возобновления режима динамического изменения базового приоритета всех потоков процесса используется функция *SetProcessPriorityBoost*, которая имеет следующий прототип:

```
BOOL SetProcessPriorityBoost(
        HANDLE      hProcess,                   // дескриптор процесса
        BOOL        DisablePriorityBoost        // состояние повышения приоритета
);
```

Если функция *SetProcessPriorityBoost* завершается успешно, то она возвращает значение TRUE, в противном случае возвращаемое значение равно FALSE. Значение параметра DisablePriorityBoost устанавливает состояние режима динамического повышения базовых приоритетов потоков. Если это значение равно TRUE, то режим динамического повышения базовых приоритетов потоков, выполняемых в рамках процесса с дескриптором hProcess, запрещается. Если же значение этого параметра равно FALSE, то, наоборот, режим динамического повышения базовых приоритетов этих потоков разрешается.

Узнать, разрешен ли режим динамического повышения базовых приоритетов потоков, можно посредством вызова функции *GetProcessPriorityBoost*, которая имеет следующий прототип:

```
BOOL GetProcessPriorityBoost(
        HANDLE      hProcess,                   // дескриптор процесса
        PBOOL       pDisablePriorityBoost       // состояние повышения приоритета
);
```

Если функция *GetProcessPriorityBoost* завершается успешно, то она возвращает значение TRUE, в противном случае возвращаемое значение равно FALSE. Значение булевой переменной, на которую указывает параметр pDisablePriorityBoost, определяет состояние режима динамического повышения базовых приоритетов потоков. Если это значение равно TRUE, то режим динамического повышения базовых приоритетов потоков, выполняемых в рамках процесса с дескриптором hProcess, запрещен. Если же значение этого параметра равно FALSE, то, наоборот, режим динамического повышения базовых приоритетов этих потоков разрешен.

Для отмены или возобновления режима динамического изменения базового приоритета только одного потока используется функция *SetThreadPriorityBoost*, которая имеет следующий прототип:

```
BOOL  SetThreadPriorityBoost (
        HANDLE      hThread,                    // дескриптор потока
        BOOL        DisablePriorityBoost        // состояние повышения приоритета
);
```

Эта функция работает анологично функции *SetProcessPriorityBoost*, но только для одного потока с дескриптором hThread.

Чтобы определить, разрешен ли режим динамического повышения базового приоритета для какого-то конкретного потока, используется функция *GetThreadPriorityBoost*, которая имеет следующий прототип:

```
BOOL GetThreadPriorityBoost(
        HANDLE      hThread,                    // дескриптор потока
        PBOOL       pDisablePriorityBoost       // состояние повышения приоритета
);
```

Эта функция работает так же, как и функция *GetProcessPriorityBoost*, но только для одного потока с дескриптором hThread.

Ниже приведена программа, которая демонстрирует работу функций *SetProcessPriorityBoost*, *GetProcessPriorityBoost*, *SetThreadPriorityBoost* и *GetThreadPriorityBoost*. Ещё раз отметим, что эти функции работают корректно только в операционной системе Windows 2000. В операционной системе Windows 98 эти функции всегда возвращают значение FALSE.

Программа 3.10.

```
// Пример работы функций
// SetProcessPriorityBoost и GetProcessPriorityBoost,
// SetThreadPriorityBoost и GetThreadPriorityBoost.

#include <windows.h>
#include <conio.h>

int main()
{
	HANDLE		hProcess, hThread;
	BOOL		bPriorityBoost;

	// получаем псевдодескриптор текущего процесса
	hProcess = GetCurrentProcess();

	// узнаем режим динамического повышения приоритетов для процесса
	if (!GetProcessPriorityBoost(hProcess, &bPriorityBoost))
	{
		_cputs("Get process priority boost failed.\n");
		_cputs("Press any key to exit.\n");
		_getch();
		return GetLastError();
	}
	_cprintf("The process priority boost = %d.\n", bPriorityBoost);

	// выключаем режим динамического повышения приоритетов для процесса
	if (!SetProcessPriorityBoost(hProcess, TRUE))
	{
		_cputs("Set process priority boost failed.\n");
		_cputs("Press any key to exit.\n");
		_getch();
		return GetLastError();
	}

	// получаем псевдодескриптор текущего потока
	hThread = GetCurrentThread();
	// узнаем режим динамического повышения приоритетов для потока
	if (!GetThreadPriorityBoost(hThread, &bPriorityBoost))
	{
		_cputs("Get process priority boost failed.\n");
		_cputs("Press any key to exit.\n");
		_getch();
		return GetLastError();
	}
	_cprintf("The thread priority boost = %d.\n", bPriorityBoost);

	// включаем режим динамического повышения приоритетов для потока
	if (!SetThreadPriorityBoost(hThread, FALSE))
	{
		_cputs("Set process priority boost failed.\n");
		_cputs("Press any key to exit.\n");
		_getch();
```

```
            return GetLastError();
        }
        _cputs("Press any key to exit.\n");
        _getch();

        return 0;
}
```

В заключении данного параграфа приведем таблицу базовых приоритетов потоков в зависимости от приоритета процесса и уровня приоритета потока.

Таблица 3.1. Базовые приоритеты потоков.

	Класс приоритета процесса	**Уровень приоритета потока**
1	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
1	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
1	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
1	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
1	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
2	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
3	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
4	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
4	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
5	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
5	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
5	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
6	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
6	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
6	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
7	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
7	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
7	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
8	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
8	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
8	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
8	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
9	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST

9	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
9	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
10	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
10	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
11	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
11	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
11	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
12	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
12	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
13	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
14	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
15	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
15	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL
15	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL
15	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL
15	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL
15	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL
16	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
17	REALTIME_PRIORITY_CLASS	-7
18	REALTIME_PRIORITY_CLASS	-6
19	REALTIME_PRIORITY_CLASS	-5
20	REALTIME_PRIORITY_CLASS	-4
21	REALTIME_PRIORITY_CLASS	-3
22	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
23	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
24	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
25	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
26	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
27	REALTIME_PRIORITY_CLASS	3
28	REALTIME_PRIORITY_CLASS	4
29	REALTIME_PRIORITY_CLASS	5
30	REALTIME_PRIORITY_CLASS	6

31	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL

Отметим, что уровни приоритетов потоков от -7 до -3 и от 3 до 6 поддерживаются только операционной системой Windows 2000 и не имеют символьных обозначений.

3.4. Наследование дескрипторов.

Большинство объектов Windows может быть *наследуемыми* или *ненаследуемыми*. Исключением из этого правила являются объекты, использование которых несколькими процессами невозможно главным образом по причине нарушения доступа к памяти процесса, которая изолирована от других процессов. Свойство наследования объекта означает, что если наследуемый объект создан или открыт в некотором процессе, то к этому объекту будут также иметь доступ все процессы, которые создаются этим процессом, то есть являются его потомками. Свойство наследования объекта определяется его дескриптором, который также может быть *наследуемым* или *ненаследуемым*. Для того чтобы сделать объект наследуемым, необходимо сделать наследуемым его дескриптор и наоборот.

Однако, для того чтобы дочерний процесс имел доступ к наследуемому объекту в родительском процессе, недостаточно просто сделать дескриптор этого объекта наследуемым. Наследуемый дескриптор передается системой дочернему процессу, но этот дескриптор скрыт от программ, которые выполняются в дочернем процессе. То есть программа дочернего процесса должна знать этот дескриптор и передать его этой программе должна программа родительского процесса. Одним из способов передачи дескрипторов дочернему процессу является использование командной строки, которая позволяет передавать дескрипторы как параметры.

Для пояснения сказанного приведем пример двух процессов, в которых используются наследуемые дескрипторы. Первая программа описывает дочерний процесс, который получает дескриптор потока от родительского процесса и по требованию пользователя прекращает выполнение этого потока в родительском процессе.

```
// Пример процесса, который завершает поток в родительском процессе.
// Дескриптор потока передается через командную строку.

#include <windows.h>
#include <conio.h>

int main(int argc, char *argv[])
{
	HANDLE	hThread;
	char	c;

	// преобразуем символьное представление дескриптора в число
	hThread = (HANDLE)atoi(argv[1]);
	// ждем команды о завершении потока
	while (true)
	{
		_cputs("Input 't' to terminate the thread: ");
		c = _getch();
		if (c == 't')
		{
			_cputs("t\n");
			break;
		}
	}
	// завершаем поток
	TerminateThread(hThread,0);
	// закрываем дескриптор потока
	CloseHandle(hThread);

	_cputs("Press any key to exit.\n");
```

```
	_getch();

	return 0;
}
```

Программа 3.11.

Вторая программа описывает родительский процесс, который создает дочерний процесс и передает ему через командную строку наследуемый дескриптор потока.

```
// Пример процесса, который передает наследуемый дескриптор потока
// дочернему процессу через командную строку.

#include <windows.h>
#include <conio.h>

volatile int count;

void thread()
{
	for ( ; ; )
	{
		count++;
		Sleep(500);
		_cprintf ("count = %d\n", count);
	}
}

int main()
{
	char lpszComLine[80];	// для командной строки

	STARTUPINFO si;
	PROCESS_INFORMATION pi;
	SECURITY_ATTRIBUTES sa;

	HANDLE		hThread;
	DWORD		IDThread;

	_cputs("Press any key to start the count-thread.\n");
	_getch();

	// устанавливает атрибуты защиты потока
	sa.nLength = sizeof(SECURITY_ATTRIBUTES);
	sa.lpSecurityDescriptor = NULL;		// защита по умолчанию
	sa.bInheritHandle = TRUE;		// дескриптор потока наследуемый

	// запускаем поток-счетчик
	hThread = CreateThread(&sa, 0, (LPTHREAD_START_ROUTINE)thread, NULL, 0, &IDThread);
	if (hThread == NULL)
		return GetLastError();

	// устанавливаем атрибуты нового процесса
	ZeroMemory(&si, sizeof(STARTUPINFO));
	si.cb=sizeof(STARTUPINFO);
	// формируем командную строку
	wsprintf(lpszComLine, "C:\\ConsoleProcess.exe %d", (int)hThread);
	// запускаем новый консольный процесс
```

```
	if (!CreateProcess(
			NULL,		// имя процесса
			lpszComLine,	// адрес командной строки
			NULL,		// атрибуты защиты процесса по умолчанию
			NULL,		// атрибуты защиты первичного потока по умолчанию
			TRUE,		// наследуемые дескрипторы текущего процесса
					// наследуются новым процессом
			CREATE_NEW_CONSOLE,	// новая консоль
			NULL,		// используем среду окружения процесса предка
			NULL,		// текущий диск и каталог, как и в процессе предке
			&si,		// вид главного окна - по умолчанию
			&pi		// здесь будут дескрипторы и идентификаторы
					// нового процесса и его первичного потока
			)
		)
	{
		_cputs("The new process is not created.\n");
		_cputs("Press any key to finish.\n");
		_getch();
		return GetLastError();
	}
	// закрываем дескрипторы нового процесса
	CloseHandle(pi.hProcess);
	CloseHandle(pi.hThread);

	// ждем закрытия потока-счетчика
	WaitForSingleObject(hThread, INFINITE);
	_cputs("Press any key to exit.\n");
	_getch();
	// закрываем дескриптор потока
	CloseHandle(hThread);

	return 0;
}
```

Программа 3.12.

В этой программе особенно нужно обратить внимание на два момента: значение параметра bInheritHandle функции *CreateProcess* и использование структуры sa типа SECURITY_ATTRIBUTES, адрес которой является первым параметром функции *CreateThread*. Если значение параметра bInheritHandle равно TRUE, то наследуемые дескрипторы родительского процесса передаются дочернему процессу. Поле bInheritHandle структуры sa имеет тип BOOL. Если значение этого поля установлено в TRUE, то дескриптор создаваемого потока является наследуемым, в противном случае – ненаследуемым.

Теперь рассмотрим следующую ситуацию. Предположим, что дескриптор созданного объекта является ненаследуемым, а нам необходимо сделать его наследуемым. Для решения этой проблемы в среде операционной системы Windows 2000 можно использовать функцию *SetHandleInformation*, которая используется для изменения свойств дескрипторов и имеет следующий прототип:

```
	BOOL SetHandleInformation(
		HANDLE	hObject,	// дескриптор объекта
		DWORD	dwMask,		// флаги, которые изменяем
		DWORD	dwFlags		// новые значения флагов
	);
```

В случае успешного завершения эта функция возвращает значение TRUE, в противном случае – FALSE. Для иллюстрации работы этой функции изменим последнюю программу следующим образом.

Программа 3.13.

```
// Пример изменения свойства наследуемости дескриптора функцией SetHandleInformation

#include <windows.h>
#include <conio.h>

volatile int count;

void thread()
{
	for ( ; ; )
	{
		count++;
		Sleep(500);
		_cprintf ("count = %d\n", count);
	}
}

int main()
{
	// имя нового процесса с пробелом
	char lpszComLine[80]="C:\\ConsoleProcess.exe ";
	// для символьного представления дескриптора
	char lpszHandle[20];

	STARTUPINFO si;
	PROCESS_INFORMATION pi;

	HANDLE		hThread;
	DWORD		IDThread;

	_cputs("Press any key to start the count-thread.\n");
	_cputs("After terminating the thread press any key to exit.\n");
	_getch();

	// запускаем поток-счетчик
	hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)thread, NULL, 0, &IDThread);
	if (hThread == NULL)
		return GetLastError();

	// делаем дескриптор потока наследуемым
	if(!SetHandleInformation(
			hThread,			// дескриптор потока
			HANDLE_FLAG_INHERIT,		// изменяем наследование дескриптора
			HANDLE_FLAG_INHERIT))		// делаем дескриптор наследуемым
	{
		_cputs("The inheritance is not changed.\n");
		_cputs("Press any char to finish.\n");
		_getch();
		return GetLastError();
	}

	// устанавливаем атрибуты нового процесса
	ZeroMemory(&si, sizeof(STARTUPINFO));
	si.cb=sizeof(STARTUPINFO);
	// преобразуем дескриптор в символьную строку
	_itoa((int)hThread,lpszHandle,10);
	// создаем командную строку
	strcat(lpszComLine,lpszHandle);
	// запускаем новый консольный процесс
```

```
    if (!CreateProcess(
                        NULL,           // имя процесса
                        lpszComLine,    // адрес командной строки
                        NULL,           // атрибуты защиты процесса по умолчанию
                        NULL,           // атрибуты защиты первичного потока по умолчанию
                        TRUE,           // наследуемые дескрипторы текущего процесса
                                        // наследуются новым процессом
                        CREATE_NEW_CONSOLE,     // новая консоль
                        NULL,           // используем среду окружения процесса предка
                        NULL,           // текущий диск и каталог как и в процессе предке
                        &si,            // вид главного окна - по умолчанию
                        &pi             // здесь будут дескрипторы и идентификаторы
                                        // нового процесса и его первичного потока
                        )
            )
    {
            _cputs("The new process is not created.\n");
            _cputs("Press any key to finish.\n");
            _getch();
            return GetLastError();
    }
    // закрываем дескрипторы нового процесса
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);

    _getch();

    // закрываем дескриптор потока
    CloseHandle(hThread);

    return 0;
}
```

Для определения свойств дескриптора используется функция *GetHandleInformation*, которая имеет следующий прототип:

```
BOOL GetHandleInformation(
        HANDLE      hObject,        // дескриптор объекта
        LPDWORD     lpdwFlags       // свойства дескриптора
);
```

Эта функция также в случае успешного завершения возвращает значение TRUE. В противном случае возвращаемое значение равно FALSE.

Отметим важную деталь, функции *SetHandleInformation* и *GetHandleInformation* правильно работают только операционной системой Windows 2000. В операционной системе Windows 98 эти функции всегда возвращают значение FALSE. Для решения рассматриваемой проблемы в операционной системе Windows 98, нужно использовать дублирование дескрипторов, которое мы рассмотрим в следующем параграфе.

3.5. Дублирование дескрипторов.

Дублирование дескрипторов необходимо для решения следующей задачи. Иногда, при передаче дескриптора из одного процесса в другой необходимо изменить не только свойство наследования дескриптора, но и другие свойства этого дескриптора, которые управляют доступом к объекту. Для решения этой проблемы предназначена функция *DuplicateHandle*, которая имеет следующий прототип:

```
BOOL DuplicateHandle(
        HANDLE      hSourceProcessHandle,       // дескриптор процесса источника
        HANDLE      hSourceHandle,              // исходный дескриптор для дублирования
```

```
        HANDLE      hTargetProcessHandle,    // дескриптор процесса приемника
        LPHANDLE    lpTargetHandle,          // дубликат исходного дескриптора
        DWORD       dwDesiredAccess,         // флаги доступа к объекту
        BOOL        bInheritHandle,          // наследрвание дескриптора
        DWORD       dwOptions                // дополнительные необязательные флаги
);
```

Если функция *DuplicateHandle* завершается успешно, то возвращаемое значение равно TRUE, в противном случае эта функция возвращает значение FALSE.

Отметим назначение трех последних параметров функция *DuplicateHandle*. Начнем с последнего параметра dwOptions, который может быть равен любой комбинации из двух флагов: DUPLICATE_CLOSE_SOURCE и DUPLICATE_SAME_ACCESS. Если указан флаг DUPLICATE_CLOSE_SOURCE, то при любом своем завершении функция *DuplicateHandle* закрывает исходный дескриптор. Если указан флаг DUPLICATE_SAME_ACCESS, то доступ к объекту через дублированный дескриптор совпадает с доступом к объекту через исходный дескриптор. Совместное использование этих флагов обеспечивает выполнение двух указанных действий.

Теперь перейдем к параметру dwDesiredAccess. Этот параметр определяет возможный доступ к объекту через дубликат исходного дескриптора, используя определенную комбинацию флагов. Значения этих флагов отличаются для объектов разных типов и будут описаны в процессе работы с объектами. Если доступ к объекту не изменяется, что определяется значением последнего параметра dwOptions, то система игнорирует значение параметра dwDesiredAccess.

Параметр bInheritHandle функции *DuplicateHandle* устанавливает свойство наследования нового дескриптора. Если значение этого параметра равно TRUE, то создаваемый дубликат исходного дескриптора является наследуемым, в случае FALSE – ненаследуемым.

Приведем пример программы, которая использует функцию *DuplicateHandle* для разрешения дочернему процессу завершить поток в родительском процессе. Эта программа является другим решением задачи, решаемой программой 3.10.

Программа 3.14.

```
// Пример создания наследуемого дескриптора функцией DuplicateHandle

#include <windows.h>
#include <conio.h>

volatile int count;

void thread()
{
        for ( ; ; )
        {
                count++;
                Sleep(500);
                _cprintf ("count = %d\n", count);
        }
}

int main()
{
        // имя нового процесса с пробелом
        char lpszComLine[80]="C:\\ConsoleProcess.exe ";
        // для символьного представления дескриптора
        char lpszHandle[20];

        STARTUPINFO si;
        PROCESS_INFORMATION pi;

        HANDLE      hThread, hInheritThread;
        DWORD       IDThread;
```

```
_cputs("Press any key to start the count-thread.\n");
_cputs("After terminating the thread press any key to exit.\n");
_getch();

// запускаем поток-счетчик
hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)thread, NULL, 0, &IDThread);
if (hThread == NULL)
	return GetLastError();

// делаем наследуемый дубликат дескриптора потока
if(!DuplicateHandle(
		GetCurrentProcess(),	// дескриптор текущего процесса
		hThread,		// исходный дескриптор потока
		GetCurrentProcess(),	// дескриптор текущего процесса
		&hInheritThread,	// новый дескриптор потока
		0,			// этот параметр игнорируется
		TRUE,			// новый дескриптор наследуемый
		DUPLICATE_SAME_ACCESS))  // доступ не изменяем
{
	_cputs("The handle is not duplicated.\n");
	_cputs("Press any key to finish.\n");
	_getch();
	return GetLastError();
}

// устанавливаем атрибуты нового процесса
ZeroMemory(&si, sizeof(STARTUPINFO));
si.cb=sizeof(STARTUPINFO);
// преобразуем наследуемый дескриптор в символьную строку
_itoa((int)hInheritThread,lpszHandle,10);
// создаем командную строку
strcat(lpszComLine,lpszHandle);
// запускаем новый консольный процесс
if (!CreateProcess(
		NULL,		// имя процесса
		lpszComLine,	// адрес командной строки
		NULL,		// атрибуты защиты процесса по умолчанию
		NULL,		// атрибуты защиты первичного потока по умолчанию
		TRUE,		// наследуемые дескрипторы текущего процесса
				// наследуются новым процессом
		CREATE_NEW_CONSOLE,	// новая консоль
		NULL,		// используем среду окружения процесса предка
		NULL,		// текущий диск и каталог как и в процессе предке
		&si,		// вид главного окна - по умолчанию
		&pi		// здесь будут дескрипторы и идентификаторы
				// нового процесса и его первичного потока
		)
	)
{
	_cputs("The new process is not created.\n");
	_cputs("Press any key to finish.\n");
	_getch();
	return GetLastError();
}
// закрываем дескрипторы нового процесса
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);

_getch();
```

```
	// закрываем дескриптор потока
	CloseHandle(hThread);

	return 0;
}
```

Теперь приведем пример программы, которая дублирует дескриптор, но при этом изменяет доступ к нему. Конкретно, разрешим дочернему процессу только прекращать выполнение потока в родительском процессе. В этом случае остальные функции над потоками, такие как, например, *SuspendThread* и *ResumeThread*, будут недоступны для выполнения в дочернем процессе. Обратите внимание на изменение значений параметров функции *DuplicateHandle*.

Программа 3.15.

```
// Пример создания наследуемого дескриптора функцией DuplicateHandle

#include <windows.h>
#include <conio.h>

volatile int count;

void thread()
{
	for ( ; ; )
	{
		count++;
		Sleep(500);
		_cprintf ("count = %d\n", count);
	}
}

int main()
{
	// имя нового процесса с пробелом
	char lpszComLine[80]="C:\\ConsoleProcess.exe ";
	// для символьного представления дескриптора
	char lpszHandle[20];

	STARTUPINFO si;
	PROCESS_INFORMATION pi;

	HANDLE		hThread, hInheritThread;
	DWORD		IDThread;

	_cputs("Press any key to start the count-thread.\n");
	_cputs("After terminating the thread press any key to exit.\n");
	_getch();

	// запускаем поток-счетчик
	hThread = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)thread, NULL, 0, &IDThread);
	if (hThread == NULL)
		return GetLastError();

	// делаем наследуемый дубликат дескриптора потока
	if(!DuplicateHandle(
			GetCurrentProcess(),	// дескриптор текущего процесса
			hThread,		// исходный дескриптор потока
```

```
                        GetCurrentProcess(),        // дескриптор текущего процесса
                        &hInheritThread,            // новый дескриптор потока
                        THREAD_TERMINATE, // только завершение потока
                        TRUE,                       // новый дескриптор наследуемый
                        0))                         // не используем
        {
                _cputs("The handle is not duplicated.\n");
                _cputs("Press any key to finish.\n");
                _getch();
                return GetLastError();
        }

        // устанавливаем атрибуты нового процесса
        ZeroMemory(&si, sizeof(STARTUPINFO));
        si.cb=sizeof(STARTUPINFO);
        // преобразуем наследуемый дескриптор в символьную строку
        _itoa((int)hInheritThread,lpszHandle,10);
        // создаем командную строку
        strcat(lpszComLine,lpszHandle);
        // запускаем новый консольный процесс
        if (!CreateProcess(
                        NULL,               // имя процесса
                        lpszComLine,        // адрес командной строки
                        NULL,               // атрибуты защиты процесса по умолчанию
                        NULL,               // атрибуты защиты первичного потока по умолчанию
                        TRUE,               // наследуемые дескрипторы текущего процесса
                                            // наследуются новым процессом
                        CREATE_NEW_CONSOLE,         // новая консоль
                        NULL,               // используем среду окружения процесса предка
                        NULL,               // текущий диск и каталог, как и в процессе предке
                        &si,                // вид главного окна - по умолчанию
                        &pi                 // здесь будут дескрипторы и идентификаторы
                                            // нового процесса и его первичного потока
                        )
                )
        {
                _cputs("The new process is not created.\n");
                _cputs("Press any key to finish.\n");
                _getch();
                return GetLastError();
        }
        // закрываем дескрипторы нового процесса
        CloseHandle(pi.hProcess);
        CloseHandle(pi.hThread);

        _getch();

        // закрываем дескриптор потока
        CloseHandle(hThread);

        return 0;
}
```