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

2. Потоки.

2.1. Определение потоков.

Потоком в Windows называется объект ядра, которому операционная система выделяет процессорное время для выполнения приложения. Каждому потоку принадлежат следующие ресурсы:

– код исполняемой функции;
– набор регистров процессора;
– стек для работы приложения;
– стек для работы операционной системы;
– блок окружения, который содержит служебную информацию для работы потока.

Все эти ресурсы образуют *контекст потока в Windows*. В Windows различаются потоки двух типов:

– рабочие потоки (working threads);
– потоки интерфейса пользователя (user interface threads).

Рабочие потоки выполняют различные фоновые задачи в приложении. Потоки интерфейса пользователя связаны с окнами и выполняют обработку сообщений, поступающих этим окнам.

Каждое приложение имеет, по крайней мере, один поток, который называется *первичным* (*primary)* или *главным* (*main*) *потоком*. В консольных приложениях это поток, который исполняет функцию main. В приложениях с графическим интерфейсом это поток, который исполняет функцию WinMain.

2.2. Создание потоков в Windows.

Создается поток функцией *CreateThread*, которая имеет следующий прототип:

```
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES    lpThreadAttributes,    // атрибуты защиты
    DWORD                    dwStackSize,           // размер стека потока в байтах
    LPTHREAD_START_ROUTINE   lpStartAddress,        // адрес исполняемой функции
    LPVOID                   lpParameter,           // адрес параметра
    DWORD                    dwCreationFlags,       // флаги создания потока
    LPDWORD                  lpThreadId             // идентификатор потока
);
```

При успешном завершении функция *CreateThread* возвращает дескриптор созданного потока и его идентификатор, который является уникальным для всей системы. В противном случае эта функция возвращает значение NULL.

Кратко опишем назначение параметров функции *CreateThread*. Параметр lpThreadAttributes устанавливает атрибуты защиты создаваемого потока. До тех пор пока мы не изучим структуру системы безопасности в Windows, то есть раздел Windows NT Access Control из интерфейса программирования приложений Win32 API, мы будем устанавливать значения этого параметра в NULL при вызове почти всех функций ядра Windows. Это означает, что атрибуты защиты потока совпадают с атрибутами защиты создавшего его процесса. О процессах будет подробно рассказано в следующем разделе.

Параметр dwStackSize определяет размер стека, который выделяется потоку при запуске. Если этот параметр равен нулю, то потоку выделяется стек, размер которого равен по умолчанию 1 Мб. Это наименьший размер стека, который может быть выделен потоку. Если величина параметра dwStackSize меньше, значения, заданного по умолчанию, то все равно потоку выделяется стек размеров в 1Мб. Операционная система Windows округляет размер стека до одной страницы памяти, который обычно равен 4 Кб.

Параметр lpStartAddress указывает на исполняемую потоком функцию. Эта функция должна иметь следующий прототип:

```
DWORD WINAPI ThreadProc(LPVOID lpParameters);
```

Параметр lpParameter является единственным параметром, который будет передан функции потока.

Параметр dwCreationFlags определяет, в каком состоянии будет создан поток. Если значение этого параметра равно 0, то функция потока начинает выполняться сразу после создания потока. Если же значение этого параметра равно CREATE_SUSPENDED, то поток создается в подвешенном состоянии. В дальнейшем этот поток можно запустить вызовом функции ResumeThread.

Параметр lpThreadId является выходным, то есть его значение устанавливает Windows. Этот параметр должен указывать на переменную, в которую Windows поместит идентификатор потока, который уникален для всей системы и может в дальнейшем использоваться для ссылок на поток.

Приведем пример программы, которая использует функцию *CreateThread* для создания потока, и продемонстрируем способ передачи параметров исполняемой потоком функции.

Программа 2.1.

```
// Пример создания потока функцией CreateThread

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

volatile int n;

DWORD WINAPI Add(LPVOID iNum)
{
	cout << "Thread is started." << endl;
	n = n + (int)iNum;
	cout << "Thread is finished." << endl;

	return 0;
}

int main()
{
	int	inc = 10;
	HANDLE	hThread;
	DWORD	IDThread;

	cout << "n = " << n << endl;

	hThread = CreateThread(NULL, 0, Add, (void*)inc, 0, &IDThread);
	if (hThread == NULL)
		return GetLastError();

	// ждем пока поток Add закончит работу
	WaitForSingleObject(hThread, INFINITE);
	// закрываем дескриптор потока
	CloseHandle(hThread);

	cout << "n = " << n << endl;

	return 0;
}
```

Отметим, что в этой программе используется функция *WaitForSingleObject*, которая ждет завершения потока Add. Подробно эта функция будет рассмотрена далее в параграфе, посвященном объектам синхронизации и функциям ожидания.

Важное замечание. Отметим, что перед компиляцией многопоточных программ в консольном проекте необходимо установить режим отладки многопоточных приложений. Это делается следующим образом: в

пункте меню Project выбирается команда Settings, далее, в появившемся окне ProjectSettings выбирается вкладка C/C++. Теперь в списке Category выбираем строку Code Generation, а в списке Use run-time library выбираем строку Debug Multithreaded. После этого нажимаем OK и программа готова к компиляции, редактированию связей и выполнению.

2.3. Завершение потоков.

Поток завершается вызовом функции *ExitThread*, которая имеет следующий прототип:

```
VOID ExitThread(
        DWORD       dwExitCode          // код завершения потока
);
```

Эта функция может вызываться как явно, так и неявно при возврате из функции потока.

Узнать код завершения потока можно при помощи функции *GetExitCodeThread*.

Один поток может завершить другой поток, вызвав функцию *TerminateThread*, которая имеет следующий прототип:

```
BOOL TerminateThread(
        HANDLE      hThread,            // дескриптор потока
        DWORD       dwExitThread        // код завершения потока
);
```

В случае успешного завершения функция *TerminateThread* возвращает значение TRUE, в противном случае – значение FALSE. Функция *TerminateThread* завершает поток, но не освобождает все ресурсы, принадлежащие этому потоку. Поэтому эта функция должна вызываться только в аварийных ситуациях при зависании потока.

Приведем программу, которая демонстрируют работу функции *TerminateThread*.

Программа 2.3.

```
// Пример завершения потока функцией TerminateThread

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

volatile UINT count;

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

int main()
{
        HANDLE      hThread;
        DWORD       IDThread;
        char c;

        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 'n' to finish: ";
                cin >> c;
```

```
		if (c == 'y')
			cout << "count = " << count << endl;
		if (c == 'n')
			break;
	}

	// прерываем выполнение потока thread
	TerminateThread(hThread, 0);

	// закрываем дескриптор потока
	CloseHandle(hThread);

	return 0;
}
```

2.4. Приостановка и возобновление потоков.

Каждый созданный поток имеет счетчик приостановок, максимальное значение которого равно MAXIMUM_SUSPEND_COUNT. Счетчик приостановок показывает, сколько раз исполнение потока было приостановлено. Поток может исполняться только при условии, что значение счетчика приостановок равно нулю. В противном случае поток не исполняется или, как говорят, находится в подвешенном состоянии. Исполнение каждого потока может быть приостановлено вызовом функции *SuspendThread*, которая имеет следующий прототип:

```
DWORD SuspendThread(
	HANDLE		hThread		// дескриптор потока
);
```

Эта функция увеличивает значение счетчика приостановок на 1 и при успешном завершении возвращает текущее значение этого счетчика. В случае неудачи функция *SuspendThread* возвращает значение равное -1.

Для возобновления исполнения потока используется функция *ResumeThread*, которая имеет следующий прототип:

```
DWORD ResumeThread(
	HANDLE		hThread		// дескриптор потока
);
```

Функция *ResumeThread* уменьшает значение счетчика приостановок на 1 при условии, что это значение было больше нуля. Если полученное значение счетчика приостановок равно 0, то исполнение потока возобновляется, в противном случае поток остается в подвешенном состоянии. Если при вызове функции *ResumeThread* значение счетчика приостановок было равным 0, то это значит, что поток не находится в подвешенном состоянии. В этом случае функция не выполняет никаких действий. При успешном завершении функция *ResumeThread* возвращает текущее значение счетчика приостановок, в противном случае возвращаемое значение равно -1.

Поток может задержать свое исполнение вызовом функции *Sleep*, которая имеет следующий прототип:

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

Единственный параметр функции *Sleep* определяет количество миллисекунд, на которые поток, вызвавший эту функцию, приостанавливает свое исполнение. Если значение этого параметра равно 0, то выполнение потока просто прерывается, а затем возобновляется при условии, что нет других потоков, ждущих выделения процессорного времени. Если же значение этого параметра равно INFINITE, то поток приостанавливает свое исполнение навсегда, что приводит к блокированию работы приложения.

Ниже приведена программа, которая демонстрируют работу функций *SuspendThread*, *ResumeThread* и *Sleep*.

Программа 2.4.

```
// Пример работы функций SuspendThread, ResumeThread и Sleep

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

volatile UINT      nCount;
volatile DWORD dwCount;

void thread()
{
	for ( ; ; )
	{

		nCount++;
		// приостанавливаем поток на 100 миллисекунд
		Sleep(100);
	}
}

int main()
{
	HANDLE		hThread;
	DWORD		IDThread;
	char c;

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

	for ( ; ; )
	{
		cout << "Input :" << endl;
		cout << "\t'n' to exit" << endl;
		cout << "\t'y' to display the count" << endl;
		cout << "\t's' to suspend thread" << endl;
		cout << "\t'r' to resume thread" << endl;
		cin >> c;

		if (c == 'n')
			break;
		switch (c)
		{
		case 'y':
			cout << "count = " << nCount << endl;
			break;
		case 's':
			// приостанавливаем поток thread
			dwCount = SuspendThread(hThread);
			cout << "Thread suspend count = " << dwCount << endl;
			break;
		case 'r':
			// возобновляем поток thread
			dwCount = ResumeThread(hThread);
			cout << "Thread suspend count = " << dwCount << endl;
			break;
		}
	}
```

```
	// прерываем выполнение потока thread
	TerminateThread(hThread, 0);

	// закрываем дескриптор потока
	CloseHandle(hThread);

	return 0;
}
	// прерываем выполнение потока thread
	TerminateThread(hThread, 0);
	// закрываем дескриптор потока thread
	CloseHandle(hThread);

	return 0;
}
```

2.5. Обработка ошибок в Windows.

Большинство функций Win32 API возвращают код, по которому можно определить, как завершилась функция: успешно или нет. Если функция завершилась неудачей, то код возврата обычно равен FALSE, NULL или -1. В этом случае функция Win32 API также устанавливает внутренний код ошибки, который называется *код последней ошибки* (last-error code) и поддерживается отдельно для каждого потока. Чтобы получить код последней ошибки, нужно вызвать функцию *GetLastError*, которая имеет следующий прототип:

```
DWORD GetLastError(VOID);
```

Эта функция возвращает код последней ошибки, установленной в потоке. Установить код последней ошибки в потоке можно при помощи вызова функции *SetLastError*, которая имеет следующий прототип:

```
VOID SetLastError(
	DWORD		dwErrCode	// код ошибки
);
```

Чтобы получить сообщение об ошибке, соответствующее коду последней ошибки, необходимо использовать функцию *FormatMessage*, которая имеет следующий прототип:

```
DWORD FormatMessage(
	DWORD		dwFlags,		// режимы форматирования
	LPCVOID		lpSource,		// источник сообщения
	DWORD		dwMessageId,		// идентификатор сообщения
	DWORD		dwLanguageId,		// идентификатор языка
	LPTSTR		lpBuffer,		// буфер для сообщения
	DWORD		nSize,			// максимальный размер буфера для сообщения
	va_list		*Arguments		// список значений для вставки в сообщение
);
```

Мы не будем подробно рассматривать эту функцию, которая предназначена для форматирования символьных сообщений, приведем только пример её использования для вывода сообщения об ошибке в окне сообщений (message box). Для этого приведем сначала текст функции, которая выводит сообщение об ошибке, а затем программу, которая использует эту функцию.

Программа 2.5.

```
// Функция для вывода сообщения об ошибке в MessageBox

#include <windows.h>
```

```
void ErrorMessageBox()
{
	LPVOID lpMsgBuf;

	FormatMessage(
		FORMAT_MESSAGE_ALLOCATE_BUFFER |
		FORMAT_MESSAGE_FROM_SYSTEM |
		FORMAT_MESSAGE_IGNORE_INSERTS,
		NULL,
		GetLastError(),
		MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),	// язык по умолчанию
		(LPTSTR) &lpMsgBuf,
		0,
		NULL
	);
	// Показать ошибку в MessageBox.
	MessageBox(
		NULL,
		(LPCTSTR)lpMsgBuf,
		"Ошибка Win32 API",
		MB_OK | MB_ICONINFORMATION
	);
	// Освободить буфер.
	LocalFree( lpMsgBuf );
}
```

Программа 2.6.

```
// Пример вывода сообщения об ошибке в MessageBox.

#include <windows.h>

// прототип функции вывода сообщения об ошибке в MessageBox
void ErrorMessageBox();

// тест для функции вывода сообщения об ошибке на консоль
int main()
{
	HANDLE		hHandle=NULL;

	// неправильный вызов функции закрытия дескриптора
	if (!CloseHandle(hHandle))
		ErrorMessageBox();

	return 0;
}
```

Теперь, раз мы работаем с консольными приложениями, рассмотрим как выводить сообщение об ошибке на консоль. Для этого нам нужно научиться выводить русский текст на консоль. Это можно сделать при помощи функции *CharToOem*, которая имеет следующий прототип:

```
	BOOL CharToOem(
		LPCTSTR		lpszSrc,		// строка для перекодировки
		LPSTR		lpszDst		// перекодированная строка
	);
```

Эта функция перекодирует символы из кодировки Microsoft в кодировку, определенную производителем оборудования. Ниже приведен пример использования этой функции.

Программа 2.7.

```
// Пример перекодировки русских букв для вывода на консоль, используя функцию CharToOem.
// Обратная перекодировка выполняется функцией OemToChar.
// OEM=original equipment manufacturer или по-русски – настоящий производитель аппаратуры

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

int main()
{

	char big[] = "АБВГДЕЁЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯ";
	char sml[] = "абвгдеёжзийклмнопрстуфхцчшщъыьэюя";

	CharToOem(big,big);
	CharToOem(sml,sml);

	cout << big << endl;
	cout << sml << endl;

	return 0;
}
```

Теперь определим функцию, которая выводит сообщение об ошибке на русском языке на консоль. Текст этой функции приведен ниже.

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

// функция для вывода сообщения об ошибке на консоль на русском языке

void CoutErrorMessage()
{
	char prefix[] = "Ошибка Win32 API: ";
	LPVOID lpMsgBuf;

	CharToOem(prefix,prefix);		// перекодируем заголовок

	FormatMessage(
		FORMAT_MESSAGE_ALLOCATE_BUFFER |
		FORMAT_MESSAGE_FROM_SYSTEM |
		FORMAT_MESSAGE_IGNORE_INSERTS,
		NULL,
		GetLastError(),
		MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT), // язык по умолчанию
		(LPTSTR) &lpMsgBuf,
		0,
		NULL
	);
	// перекодируем русские буквы
	CharToOem((char*)lpMsgBuf,(char*)lpMsgBuf);
	// выводим сообщение об ошибке на консоль
	cout << prefix << (char*)lpMsgBuf << endl;
	// освобождаем буфер
	LocalFree(lpMsgBuf);
```

```
}
```

Теперь приведем пример использование функции *CoutErrorMessage* в консольном приложении.

Программа 2.8.

```
// Пример вывода сообщения об ошибке на консоль.

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

// прототип функции для вывода сообщения об ошибке на консоль
void CoutErrorMessage();

// тест для функции вывода сообщения об ошибке на консоль
int main()
{
	HANDLE		hHandle=NULL;

	// неправильный вызов функции закрытия дескриптора
	if (!CloseHandle(hHandle))
		CoutErrorMessage();

	return 0;
}
```

Важное замечание. Если при отладке приложения используется отладчик, то текст сообщения, соответствующий коду последней ошибки, можно посмотреть в окне watch, если набрать в строке «имя переменной» этого окна следующий текст: @err,hr.