Модуль №7

Занятие №1

Версия 1.0.1

План занятия:

1. Повторение пройденного материала.
2. Потоки. Основные сведения.
3. Потоки и WinAPI.
4. Принципы работы с приоритетами потоков.
5. Локальная память потока.
6. Практическая часть.
7. Подведение итогов.
8. Домашнее задание.

1. Повторение пройденного материала

Данное занятие необходимо начать с краткого повторения материала предыдущего занятия. При общении со слушателями можно использовать следующие контрольные вопросы:

1. Что такое многозадачность?
2. Что такое процесс? Из каких компонент состоит процесс?
3. Какая функция API позволяет создать процесс?
4. Какие существуют способы завершения процесса?
5. Какой способ завершения процесса считается наиболее корректным?
6. Что такое объект ядра?
7. Для чего в объекте ядра необходим счётчик пользователей?
8. Что такое дочерний процесс?

9. Какой механизм позволяет нескольким процессам совместно использовать дескриптор объекта ядра?
10. Какая функция API позволяет закрыть описатель объекта ядра?
11. Что такое снимок системы?
12. Посредством какой функции библиотеки **Tool Help API** можно получить снимок системы?
13. Какие функции библиотеки **Tool Help API** позволяют получить список запущенных процессов из снимка системы?
14. Какая функция API используется для получения описателя процесса по его идентификатору?

2. Потоки. Основные сведения

Напомнить слушателям, что при запуске любого приложения операционная система Windows создает процесс. При этом процесс сам по себе не выполняется. При инициализации процесса система создает в нем единственный поток, который называется **первичным** или **основным** потоком. Первичный поток исполняет код программы, манипулируя данными в адресном пространстве процесса. Из основного потока при необходимости могут быть запущены один или несколько **вторичных** потоков, которые выполняются одновременно с основным потоком.

Акцентировать внимание слушателей на том, что в этой связи открывается возможность написания **многопоточных приложений**, что позволяет программисту более полно и тонко управлять ходом выполнения как программы в целом, так и отдельных ее частей. Это дает возможность создавать более эффективные приложения. Например, в одном потоке можно решать задачу сортировки данных, в другом – сбора информации от удаленных источников, а в третьем - задачу обработки информации, вводимой пользователем. Таким образом, написание многопоточных приложений может значительно повысить их производительность

за счет более эффективного распределения ресурсов компьютера. Это становится особенно актуальным при применении многопроцессорных систем или систем с несколькими ядрами в одном процессоре, поскольку позволяет выполнить несколько потоков одновременно.

Поскольку потоки всегда создаются в контексте какого-либо процесса, все они разделяют адресное пространство того процесса, в котором были созданы.

Следует отметить, что поток, также как и процесс, состоит из двух компонент:

1) **объект ядра «поток»** - структура данных Windows, через которую система управляет потоком. В ней хранится статистическая информация о потоке;
2) **стек потока**, который содержит параметры всех функций и локальные переменные, необходимые потоку для выполнения кода.

3. Потоки и WinAPI

Акцентировать внимание слушателей на том, что каждый поток начинает выполнение с некоторой **входной функции**. В первичном потоке используется функция **WinMain**. Если необходимо создать вторичный поток, в нем тоже должна быть входная функция, имеющая следующий прототип:

```
DWORD WINAPI ThreadProc(
      LPVOID lpParameter // аргумент, передаваемый в функцию при создании
      // вторичного потока
);
```

Следует подчеркнуть, что потоковая функция может выполнять различные задачи. После завершения своей работы функция потока вернет управление. В этот момент поток остановится, память, отведен-

ная под его стек, будет освобождена, а счетчик пользователей объекта ядра «поток» уменьшится на единицу. Когда счетчик обнулится, этот объект ядра будет разрушен.

Как отмечалось ранее, первичный поток создается системой при запуске исполняемого файла либо при создании дочернего процесса вызовом функции API **CreateProcess**. Для создания вторичных потоков необходимо воспользоваться функцией API **CreateThread**, которая создает новый поток в адресном пространстве процесса.

```
HANDLE CreateThread(
      LPSECURITY_ATTRIBUTES lpThreadAttributes, // указатель на структуру,
      // описывающую параметры защиты потока. Если параметру присвоено
      // значение NULL, то устанавливаются атрибуты по умолчанию.
      DWORD dwStackSize, // устанавливает размер стека, который отводится
      // потоку. Если параметр равен нулю, то устанавливается стек, равный
      // стеку первичного потока
      LPTHREAD_START_ROUTINE lpStartAddress,// адрес входной потоковой функции
      LPVOID lpParameter, // параметр, передаваемый в потоковую функцию
      DWORD dwCreationFlags, // флаг, который управляет созданием потока.
      // Если этот параметр равен CREATE_SUSPENDED, то поток после порождения
      // не  запускается на исполнение.
      LPDWORD lpThreadId // адрес переменной типа DWORD, в которую функция
      // возвращает идентификатор нового потока
);
```

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

- функция потока возвращает управление (рекомендуемый способ);
- поток самоуничтожается вызовом функции **ExitThread**;
- один из потоков данного или другого процесса вызывает функцию **TerminateThread**;
- завершается процесс, содержащий данный поток.

Акцентировать внимание слушателей на том, что при разработке приложения желательно, чтобы поток завершался только после возврата управления потоковой функцией. Это единственный способ, гарантирующий корректную очистку всех ресурсов, принадлежавших потоку.

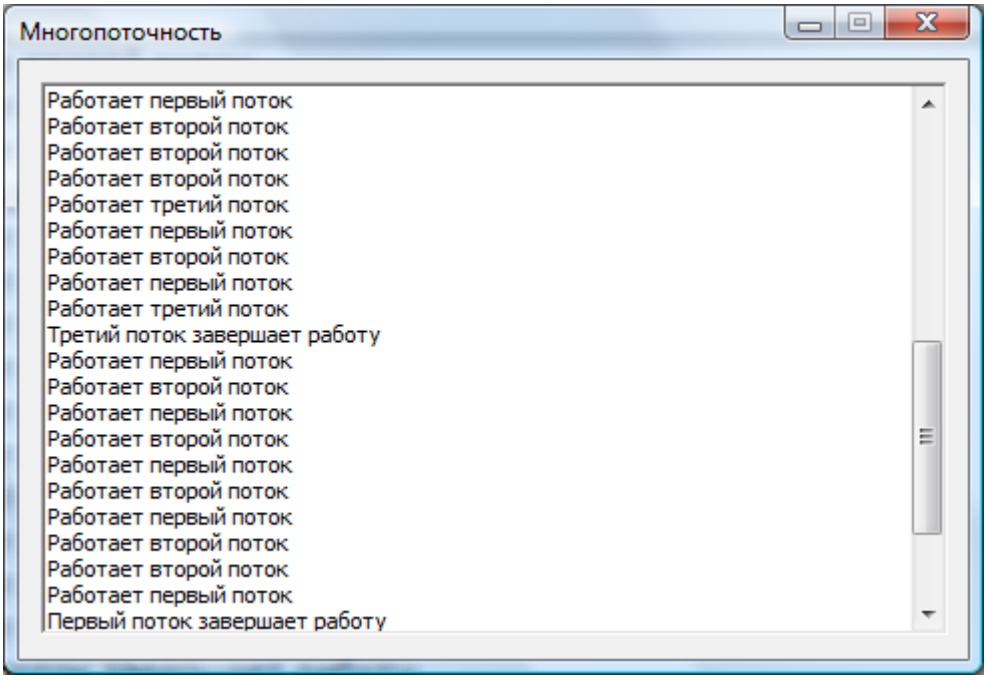
```
VOID ExitThread(
  UINT uExitCode // код завершения потока
);
```

```
BOOL TerminateThread(
  HANDLE hThread, // дескриптор завершаемого потока
  UINT uExitCode  // код завершения потока
 );
```

Поток можно приостановить на определенный период времени, вызвав функцию API **Sleep**.

```
VOID Sleep(
  DWORD dwMilliseconds // длительность временной задержки в миллисекундах
);
```

Привести слушателям пример многопоточного приложения.




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

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

class CMultithreadDlg
{
public:
      CMultithreadDlg(void);
public:
      ~CMultithreadDlg(void);
      static BOOL CALLBACK DlgProc(HWND hWnd, UINT mes, WPARAM wp, LPARAM lp);
      static CMultithreadDlg* ptr;
      void Cls_OnClose(HWND hwnd);
      BOOL Cls_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam);
 };


                                   // MultithreadDlg.cpp
#include "MultithreadDlg.h"

CMultithreadDlg* CMultithreadDlg::ptr = NULL;

CMultithreadDlg::CMultithreadDlg(void)
{
      ptr = this;
```

```
}

CMultithreadDlg::~CMultithreadDlg(void)
{

}

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

DWORD WINAPI Thread1(LPVOID lp)
{
	HWND hList = (HWND)lp;
	for(int i = 0; i < 15; i++)
	{
		SendMessage(hList, LB_ADDSTRING, 0,
				LPARAM(TEXT("Работает первый поток")));
		Sleep(10);
	}
	SendMessage(hList, LB_ADDSTRING, 0,
			LPARAM(TEXT("Первый поток завершает работу")));
	return 0;
}

DWORD WINAPI Thread2(LPVOID lp)
{
	HWND hList = (HWND)lp;
	for(int i = 0; i < 20; i++)
	{
		SendMessage(hList, LB_ADDSTRING, 0,
				LPARAM(TEXT("Работает второй поток")));
		Sleep(10);
	}
	SendMessage(hList, LB_ADDSTRING, 0,
			LPARAM(TEXT("Второй поток завершает работу")));
	return 0;
}

DWORD WINAPI Thread3(LPVOID lp)
{
	HWND hList = (HWND)lp;
	for(int i = 0; i < 10; i++)
	{
		SendMessage(hList, LB_ADDSTRING, 0,
				LPARAM(TEXT("Работает третий поток")));
		Sleep(10);
	}
	SendMessage(hList, LB_ADDSTRING, 0,
			LPARAM(TEXT("Третий поток завершает работу")));
	return 0;
}

BOOL CMultithreadDlg::Cls_OnInitDialog(HWND hwnd, HWND hwndFocus,
                                       LPARAM lParam)
{
	HWND hList = GetDlgItem(hwnd, IDC_LIST1);
	CreateThread(NULL, 0, Thread1, hList, 0, NULL);
	CreateThread(NULL, 0, Thread2, hList, 0, NULL);
```

```
      CreateThread(NULL, 0, Thread3, hList, 0, NULL);
      return TRUE;
}

BOOL CALLBACK CMultithreadDlg::DlgProc(HWND hwnd, UINT message,
                                       WPARAM wParam, LPARAM lParam)
{
      switch(message)
      {
            HANDLE_MSG(hwnd, WM_CLOSE, ptr->Cls_OnClose);
            HANDLE_MSG(hwnd, WM_INITDIALOG, ptr->Cls_OnInitDialog);
      }
      return FALSE;
 }
                              // MultithreadDlg.cpp
#include "MultithreadDlg.h"

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

Детально проанализировав вышеприведенный код, рассмотреть со слушателями механизм приостановки и возобновления работы потоков. Отметить, что в объекте ядра «поток» имеется **счетчик числа приостановок** данного потока. При вызове функции **CreateThread** он инициализируется значением, равным 1, которое запрещает системе выделять новому потоку процессорное время. Это обусловлено тем, что сразу же после создания поток ещё не готов к выполнению и ему необходимо время для инициализации. После того как поток полностью инициализирован, функция **CreateThread** проверяет, не передан ли ей флаг **CREATE_SUSPENDED**. Если данный флаг был указан, функция возвращает управление, оставив созданный поток в приостановленном состоянии. В ином случае счетчик приостановок обнуляется, и поток начинает выполняться.

Следует подчеркнуть, что выполнение отдельного потока можно приостанавливать несколько раз. Если поток был приостановлен 3 раза (в этом случае счётчик приостановок равен 3), то и возобновить его следует тоже 3 раза, чтобы обнулить счётчик приостановок — лишь тогда система выделит потоку процессорное время.

Для приостановки («усыпления») потока служит функция API **SuspendThread**:

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

В случае успешного выполнения данная функция увеличивает на единицу счетчик приостановок, возвращая его предыдущее значение.

Для возобновления работы потока служит функция API **ResumeThread**:

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

В случае успешного выполнения данная функция уменьшает на единицу счетчик приостановок, возвращая предыдущее значение счетчика.

Привести слушателям пример приложения, в котором демонстрируется работа вышеописанных функций.


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

#pragma comment(lib,"comctl32")
```

```
                                // MultithreadDlg.h
#pragma once
#include "header.h"


class CMultithreadDlg
{
public:
	CMultithreadDlg(void);
public:
	~CMultithreadDlg(void);
	static BOOL CALLBACK DlgProc(HWND hWnd, UINT mes, WPARAM wp, LPARAM lp);
	static CMultithreadDlg* ptr;
	void Cls_OnClose(HWND hwnd);
	BOOL Cls_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam);
	void Cls_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify);
	HWND hProgress1, hProgress2, hProgress3, hPlay1, hPlay2, hPlay3;
	HANDLE Th1, Th2, Th3;
};


                              // MultithreadDlg.cpp
#include "MultithreadDlg.h"

CMultithreadDlg* CMultithreadDlg::ptr = NULL;

CMultithreadDlg::CMultithreadDlg(void)
{
	ptr = this;
}

CMultithreadDlg::~CMultithreadDlg(void)
{

}

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

DWORD WINAPI Thread(LPVOID lp)
{
	HWND hProgress = (HWND)lp;
	while(TRUE)
	{
		SendMessage(hProgress, PBM_SETPOS, rand()%200+1, 0);
		Sleep(250);
	}
	return 0;
}

BOOL CMultithreadDlg::Cls_OnInitDialog(HWND hwnd, HWND hwndFocus,
                                       LPARAM lParam)
{
	srand(time(0));
	hProgress1 = GetDlgItem(hwnd, IDC_PROGRESS1);
	SendMessage(hProgress1, PBM_SETRANGE, 0, MAKELPARAM(0, 200));
	SendMessage(hProgress1, PBM_SETSTEP, 1, 0);
```

```
	SendMessage(hProgress1, PBM_SETPOS, 0, 0);
	SendMessage(hProgress1, PBM_SETBKCOLOR, 0, LPARAM(RGB(0, 0, 255)));
	SendMessage(hProgress1, PBM_SETBARCOLOR, 0, LPARAM(RGB(255, 255, 0)));

	hProgress2 = GetDlgItem(hwnd, IDC_PROGRESS2);
	SendMessage(hProgress2, PBM_SETRANGE, 0, MAKELPARAM(0, 200));
	SendMessage(hProgress2, PBM_SETSTEP, 1, 0);
	SendMessage(hProgress2, PBM_SETPOS, 0, 0);
	SendMessage(hProgress2, PBM_SETBKCOLOR, 0, LPARAM(RGB(0, 255, 0)));
	SendMessage(hProgress2, PBM_SETBARCOLOR, 0, LPARAM(RGB(255, 0, 255)));

	hProgress3 = GetDlgItem(hwnd, IDC_PROGRESS3);
	SendMessage(hProgress3, PBM_SETRANGE, 0, MAKELPARAM(0, 200));
	SendMessage(hProgress3, PBM_SETSTEP, 1, 0);
	SendMessage(hProgress3, PBM_SETPOS, 0, 0);
	SendMessage(hProgress3, PBM_SETBKCOLOR, 0, LPARAM(RGB(255, 0, 0)));
	SendMessage(hProgress3, PBM_SETBARCOLOR, 0,LPARAM(RGB(0, 255, 255)));

	hPlay1 = GetDlgItem(hwnd, IDC_PLAY1);
	hPlay2 = GetDlgItem(hwnd, IDC_PLAY2);
	hPlay3 = GetDlgItem(hwnd, IDC_PLAY3);

	Th1 = CreateThread(NULL, 0, Thread, hProgress1, CREATE SUSPENDED, NULL);
	Th2 = CreateThread(NULL, 0, Thread, hProgress2, 0, NULL);
	Th3 = CreateThread(NULL, 0, Thread, hProgress3, 0, NULL);


	return TRUE;
}


void CMultithreadDlg::Cls_OnCommand(HWND hwnd, int id, HWND hwndCtl,
                                    UINT codeNotify)
{
	if(id == IDC_PLAY1)
	{
		static BOOL flag = FALSE;
		if(flag)
		{
			SuspendThread(Th1);
			SetWindowText(hPlay1, TEXT("Пуск"));
		}
		else
		{
			ResumeThread(Th1);
			SetWindowText(hPlay1, TEXT("Пауза"));
		}
		flag = !flag;
	}
	else if(id == IDC_PLAY2)
	{
		static BOOL flag = TRUE;
		if(flag)
		{
			SuspendThread(Th2);
			SetWindowText(hPlay2, TEXT("Пуск"));
		}
		else
		{
			ResumeThread(Th2);
```

```
                  SetWindowText(hPlay2, TEXT("Пауза"));
            }
            flag = !flag;
      }
      else if(id == IDC_PLAY3)
      {
            static BOOL flag = TRUE;
            if(flag)
            {
                  SuspendThread(Th3);
                  SetWindowText(hPlay3, TEXT("Пуск"));
            }
            else
            {
                  ResumeThread(Th3);
                  SetWindowText(hPlay3, TEXT("Пауза"));
            }
            flag = !flag;
      }
      else if(id == IDC_STOP1)
      {
            TerminateThread(Th1, 0);
            CloseHandle(Th1);
            EnableWindow(hPlay1, FALSE);
      }


      else if(id == IDC_STOP2)
      {
            TerminateThread(Th2, 0);
            CloseHandle(Th2);
            EnableWindow(hPlay2, FALSE);
      }
      else if(id == IDC_STOP3)
      {
            TerminateThread(Th3, 0);
            CloseHandle(Th3);
            EnableWindow(hPlay3, FALSE);
      }
}

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


                              // MultithreadDlg.cpp
#include "MultithreadDlg.h"

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

```
    INITCOMMONCONTROLSEX icc = {sizeof(INITCOMMONCONTROLSEX)};
    icc.dwICC = ICC_WIN95_CLASSES;
    InitCommonControlsEx(&icc);
    CMultithreadDlg dlg;
    return DialogBox(hInst, MAKEINTRESOURCE(IDD_DIALOG1), NULL,
                     CMultithreadDlg::DlgProc);
}
```

4. Принципы работы с приоритетами потоков

Вначале рассмотрения данного вопроса следует вкратце ознакомить слушателей с принципами планирования потоков. Отметить, что операционная система выделяет каждому потоку определенное процессорное время. Тем самым для однопроцессорной системы создается иллюзия одновременного выполнения потоков. При этом в любой момент времени только один поток может находиться в состоянии выполнения. Все остальные потоки находятся либо в состоянии готовности, либо в состоянии блокировки.

Следует подчеркнуть, что важным моментом в реализации многозадачности является **организация очереди готовых к выполнению потоков**. Акцентировать внимание слушателей на том, что в операционной системе Windows реализована система вытесняющего планирования на основе приоритетов. Это означает, что освободившийся процессор продолжает обслуживать тот поток из очереди, который обладает наибольшим приоритетом. При этом **суммарный приоритет потока** складывается из двух составляющих: **класса приоритета процесса**, его создавшего, и **относительного приоритета потока** внутри этого класса.

Windows поддерживает 32 приоритета (от 0 до 31) — чем больше номер, тем выше приоритет.

Относительный приоритет потока	Idle	Класс приоритета процесса				Real-time
		Below normal	Normal	Above normal	High	
Time-critical						

(критичный по времени)	15	15	15	15	15	31
Highest (высший)	6	8	10	12	15	26
Above normal (выше обычного)	5	7	9	11	14	25
Normal (обычный)	4	6	8	10	13	24
Below normal (ниже обычного)	3	5	7	9	12	23
Lowest (низший)	2	4	6	8	11	22
Idle (простаивающий)	1	1	1	1	1	16

Рассматривая со слушателями вышеприведенную таблицу, следует дать краткий комментарий по каждому приоритету, приведя примеры использования. В частности, отметить, что классы **Below normal** и **Above normal** стали использоваться, начиная с Windows 2000. Класс **Idle** назначается процессу, который должен простаивать в случае активности других процессов (например, приложение «Хранитель экрана»).

Процессам, которые запускает пользователь, присваивается класс **Normal**. Это самые многочисленные процессы в системе, и, как правило, они интерактивны – требуют постоянного взаимодействия с пользователем (например, графические или текстовые редакторы). Процессы класса **Normal** делятся на **процессы переднего плана (foreground)** и **фоновые (background)**. Для процесса, с которым пользователь в данный момент работает, то есть для процесса переднего плана, уровень приоритета поднимается на две единицы. Это повышает уровень комфорта работы с программой.

Создавать процессы, относящиеся к классу **High**, следует с особой осторожностью. Если поток с классом приоритета **High** занимает процессор достаточно долго, то в это время другие потоки вообще не получат доступа к процессору. Обычно с классом **High** работают некоторые системные процессы, которые большую часть времени ожидают какого-либо события (например, winlogon.exe).

Акцентировать внимание слушателей на том, что пользовательским программам не желательно использовать класс приоритета **Realtime**, поскольку в этом случае приложение будет прерывать системные потоки, управляющие мышью, клавиатурой и дисковыми операциями – фактиче-

ски система будет парализована. Только в особых случаях, когда программа взаимодействует непосредственно с аппаратурой или решаются короткие подзадачи, для которых нужно гарантировать отсутствие прерываний, класс приоритета **Realtime** может быть кратковременно использован.

Для изменения класса приоритета процесса во время работы приложения может применяться функция API **SetPriorityClass**.

```
BOOL SetPriorityClass(
        HANDLE hProcess,// дескриптор процесса
        DWORD dwPriorityClass // класс приоритета процесса
);
```

Второй параметр может принимать следующие значения:

Класс приоритета процесса	Базовый уровень приоритета	Значение параметра
Idle	4	IDLE_PRIORITY_CLASS
Below normal	6	BELOW_NORMAL_PRIORITY_CLASS
Normal	8	NORMAL_PRIORITY_CLASS
Above normal	10	ABOVE_NORMAL_PRIORITY_CLASS
High	13	HIGH_PRIORITY_CLASS
Realtime	24	REALTIME_PRIORITY_CLASS

Для получения класса приоритета процесса используется функция API **GetPriorityClass**.

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

Следует отметить, что по умолчанию создаваемый поток получает базовый приоритет в соответствии с классом своего процесса. После создания потока его относительный приоритет может изменяться как операционной системой, так и приложением с помощью функции API **SetThreadPriority**.

```
BOOL SetThreadPriority(
         HANDLE hThread,// дескриптор потока
         int nPriority // относительный приоритет потока
 );
```

Второй параметр может принимать следующие значения:

Относительный приоритет потока	**Значение параметра**	**Описание**
Idle (простаивающий)	THREAD_PRIORITY_IDLE	Приоритет потока равен 1 для классов Idle, Normal и High и 16 для процессов Realtime
Lowest (самый низкий)	THREAD_PRIORITY_LOWEST	Приоритет потока на 2 меньше базового уровня приоритета класса процесса
Below normal(ниже нормального)	THREAD_PRIORITY_BELOW_NORMAL	Приоритет потока на 1 меньше базового уровня приоритета класса процесса
Normal (нормальный)	THREAD_PRIORITY_NORMAL	Приоритет потока равен базовому уровню приоритета класса процесса
Above normal(выше нормального)	THREAD_PRIORITY_ABOVE_NORMAL	Приоритет потока на 1 больше базового уровня приоритета класса процесса
Highest (наивысший)	THREAD_PRIORITY_HIGHEST	Приоритет потока на 2 больше базового уровня приоритета класса процесса
Time critical (критичный по времени)	THREAD_PRIORITY_TIME_CRITICAL	Приоритет потока равен 15 для классов Idle, Normal и High и 31 для процессов Realtime

Для получения относительного приоритета потока используется функция API **GetThreadPriority**.

```
    int GetThreadPriority(
            HANDLE hThread // дескриптор потока
    );
```

Для получения дескриптора текущего процесса используется функция API **GetCurrentProcess**.

```
HANDLE GetCurrentProcess(void);
```

Для получения дескриптора текущего потока используется функция API **GetCurrentThread**.

```
HANDLE GetCurrentThread(void);
```

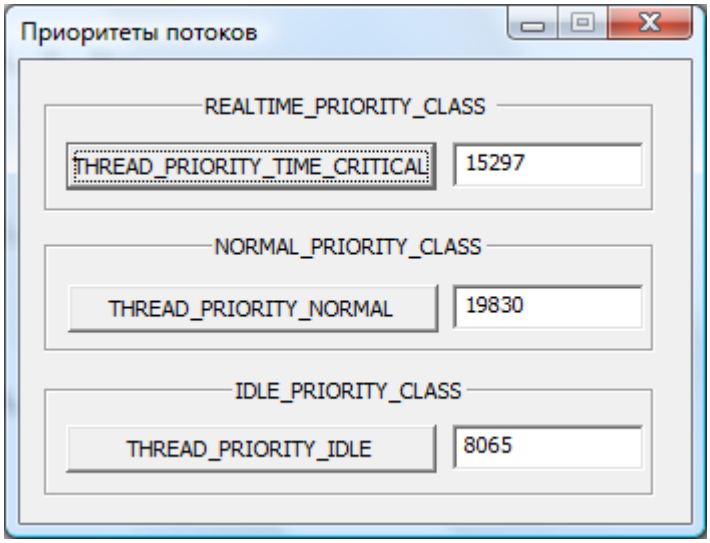
Для получения идентификатора текущего процесса используется функция API **GetCurrentProcessId**.

```
DWORD GetCurrentProcessId(void);
```

Для получения идентификатора текущего потока используется функция API **GetCurrentThreadId**.

```
DWORD GetCurrentThreadId(void);
```

В качестве примера, демонстрирующего принципы работы с приоритетами потоков, рекомендуется рассмотреть со слушателями следующее приложение.




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

```
#include <tchar.h>
#include "resource.h"

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


class CPriorityDlg
{
public:
      CPriorityDlg(void);
public:
      ~CPriorityDlg(void);
      static BOOL CALLBACK DlgProc(HWND hWnd, UINT mes, WPARAM wp, LPARAM lp);
      static CPriorityDlg* ptr;
      void Cls_OnClose(HWND hwnd);
      BOOL Cls_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam);
      void Cls_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify);
      HWND hEdit1, hEdit2, hEdit3;
};

                                    // PriorityDlg.cpp
#include "PriorityDlg.h"

CPriorityDlg* CPriorityDlg::ptr = NULL;


CPriorityDlg::CPriorityDlg(void)
{
      ptr = this;
}

CPriorityDlg::~CPriorityDlg(void)
{

}

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

BOOL CPriorityDlg::Cls_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam)
{
      hEdit1 = GetDlgItem(hwnd, IDC_EDIT1);
      hEdit2 = GetDlgItem(hwnd, IDC_EDIT2);
      hEdit3 = GetDlgItem(hwnd, IDC_EDIT3);
      SetPriorityClass(GetCurrentProcess(), NORMAL_PRIORITY_CLASS);
      SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_NORMAL);
      return TRUE;
}

DWORD WINAPI Thread1(LPVOID lp)
{
      DWORD pr = GetPriorityClass(GetCurrentProcess());
      SetPriorityClass(GetCurrentProcess(), REALTIME_PRIORITY_CLASS );
      SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_TIME_CRITICAL);
```

```
	HWND hWnd = HWND(lp);
	for (int i = 0; i <= 50000; i++)
	{
		TCHAR str[10];
		wsprintf(str, TEXT("%d"), i);
		SetWindowText(hWnd, str);
	}
	SetPriorityClass(GetCurrentProcess(), pr);
	return 0;
}

DWORD WINAPI Thread2(LPVOID lp)
{
	DWORD pr = GetPriorityClass(GetCurrentProcess());
	SetPriorityClass(GetCurrentProcess(), NORMAL_PRIORITY_CLASS );
	SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_NORMAL);
	HWND hWnd = HWND(lp);
	for (int i = 0; i <= 50000; i++)
	{
		TCHAR str[10];
		wsprintf(str, TEXT("%d"), i);
		SetWindowText(hWnd, str);
	}

	SetPriorityClass(GetCurrentProcess(), pr);
	return 0;
}

DWORD WINAPI Thread3(LPVOID lp)
{
	DWORD pr = GetPriorityClass(GetCurrentProcess());
	SetPriorityClass(GetCurrentProcess(), IDLE_PRIORITY_CLASS);
	SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_IDLE);
	HWND hWnd = HWND(lp);
	for (int i = 0; i <= 50000; i++)
	{
		TCHAR str[10];
		wsprintf(str, TEXT("%d"), i);
		SetWindowText(hWnd, str);
	}
	SetPriorityClass(GetCurrentProcess(), pr);
	return 0;
}

void CPriorityDlg::Cls_OnCommand(HWND hwnd, int id, HWND hwndCtl,
                                 UINT codeNotify)
{
	if(IDC_REALTIME == id)
	{
		HANDLE hThread = CreateThread(NULL, 0, Thread1, hEdit1, 0, NULL);
		CloseHandle(hThread);
	}
	else if(IDC_NORMAL == id)
	{
		HANDLE hThread = CreateThread(NULL, 0, Thread2, hEdit2, 0, NULL);
		CloseHandle(hThread);
	}
	else if(IDC_IDLE == id)
	{
		HANDLE hThread = CreateThread(NULL, 0, Thread3, hEdit3, 0, NULL);
```

```
            CloseHandle(hThread);
      }
}

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

                              // PriorityDlg.cpp
#include "PriorityDlg.h"

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


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

5. Локальная память потока

Продолжая рассмотрение механизма многопоточности, необходимо вкратце осветить вопрос, связанный с **локальной памятью потока** (**thread local storage, TLS**). Напомнить слушателям о том, что статические переменные, как глобальные, так и локальные по отношению к функциям, разделяются между потоками, поскольку они расположены в области памяти данных процесса. В то же время автоматические локальные переменные всегда уникальны для каждого потока, поскольку они располагаются в стеке, а каждый поток имеет свой стек.

Акцентировать внимание слушателей на том, что иногда бывает удобно использовать для двух и более потоков одну и ту же функцию, но при этом, чтобы у каждого потока были уникальные статические данные. В этом случае возникает необходимость иметь постоянную область памя-

ти, уникальную для каждого потока. Эта область памяти называется **локальной памятью потока**. Существуют два вида **TLS**:

- **динамическая** локальная память потока;
- **статическая** локальная память потока.

Следует отметить, что приложение работает с динамической локальной памятью потока, оперируя набором из четырех функций.

Каждый флаг выполняемого в системе процесса может находиться в состоянии **FREE** или **INUSE**, указывая, свободна или занята данная область локальной памяти потока (TLS-область). Чтобы воспользоваться динамической локальной памятью потока, первичный поток должен вызывать сначала функцию API **TlsAlloc** для получения значения индекса:

```
DWORD TlsAlloc();
```

Функция **TlsAlloc** заставляет систему сканировать битовые флаги в текущем процессе и искать флаг **FREE**. Отыскав, система меняет его на **INUSE**, а **TlsAlloc** возвращает индекс флага в битовом массиве. Индекс может храниться в глобальной переменной или может быть передан функции потока в параметре-структуре.

Создавая поток, система создает и массив из **TLS_MINIMUM_AVAILABLE** элементов - значений типа PVOID (идентификатор **TLS_MINIMUM_AVAILABLE** определен в файле **WinNT.h** как 64). При этом система инициализирует массив нулями и сопоставляет с потоком. Таким массивом, элементы которого могут принимать любые значения, располагает каждый поток. Прежде чем сохранить что-то в PVOID-массиве потока, необходимо выяснить, какой индекс в нем доступен, - для этой цели и служит предварительный вызов **TlsAlloc**.

Чтобы занести в массив потока значение, необходимо воспользоваться функцией API **TlsSetValue**.

```
BOOL TlsSetValue(
     DWORD dwTlsIndex,// TLS-индекс, возвращаемый функцией TlsAlloc
```

```
    PVOID pvTlsValue // значение типа PVOID
);
```

Данная функция помещает в элемент массива, индекс которого определяется параметром **dwTlsIndex**, значение типа PVOID, содержащееся в параметре **pvTlsValue**. Содержимое сопоставляется с потоком, вызвавшим **TlsSetValue**. В случае успеха возвращается **TRUE**.

Для чтения значений из массива потока служит функция API **TlsGetValue**.

```
PVOID TlsGetValue(
       DWORD dwTlsIndex // TLS-индекс, возвращаемый функцией TlsAlloc
 );
```

Данная функция возвращает значение, сопоставленное с TLS-областью под индексом **dwTlsIndex**. Как и **TlsSetValue**, функция **TlsGetValue** обращается только к массиву, который принадлежит вызывающему потоку.

Когда необходимость в TLS-области у всех потоков в процессе отпадет, необходимо вызвать функцию **TlsFree**.

```
BOOL TlsFree(
       DWORD dwTlsIndex // TLS-индекс, возвращаемый функцией TlsAlloc
  );
```

Эта функция просто сообщит системе, что данная область больше не нужна. Флаг **INUSE**, управляемый массивом битовых флагов процесса, установится как **FREE**.

Далее следует в общих чертах рассмотреть работу статической локальной памяти потока. Необходимо отметить, что статическая TLS основана на той же концепции, что и динамическая, - статическая локальная память потока предназначена для того, чтобы с потоком можно было сопоставить те или иные данные. Однако статическую локальную память потока использовать гораздо проще, так как при этом не нужно обращаться к каким-либо функциям.

Работа статической TLS-памяти строится на тесном взаимодействии с операционной системой. Загружая приложение в память, система отыскивает в EXE-файле раздел **.tls** (в данном разделе расположены все TLS-переменные) и динамически выделяет блок памяти для хранения всех статических TLS-переменных. Всякий раз, когда приложение ссылается на одну из таких переменных, ссылка переадресуется к участку, расположенному в выделенном блоке памяти. В итоге компилятору приходится генерировать дополнительный код для ссылок на статические TLS-переменные, что увеличивает размер приложения и замедляет скорость его работы.

Если в процессе создается другой поток, система выделяет еще один блок памяти для хранения статических переменных нового потока. Только что созданный поток имеет доступ лишь к своим статическим TLS-переменным, и не может обратиться к TLS-переменным любого другого потока.

6. Практическая часть

Дополнить приложение «Диспетчер задач» из предыдущего домашнего задания возможностью отображать информацию об активных потоках выбранного процесса. Для этого следует воспользоваться функциями библиотеки **Tool Help API Thread32First** и **Thread32Next**.

При этом сначала необходимо вызвать функцию **Thread32First** для получения информации о первом потоке в снимке системы.

```
BOOL WINAPI Thread32First(
  HANDLE hSnapshot, // дескриптор снимка системы
  LPTHREADENTRY32 lppe // указатель на структуру THREADENTRY32, которая
      // будет заполнена информацией о первом потоке в снимке системы
);
```

Чтобы получить информацию об остальных потоках в снимке системы необходимо многократно вызывать функцию **Thread32Next.**

```
BOOL WINAPI Thread32Next(
  HANDLE hSnapshot, // дескриптор снимка системы
  LPTHREADENTRY32  lppe  //  указатель  на  структуру  THREADENTRY32,  которая
      // будет заполнена информацией о следующем потоке в снимке системы
);
```

```
typedef struct tagTHREADENTRY32
{
      DWORD dwSize;  // размер структуры в байтах
      DWORD cntUsage;  // не используется и должен быть 0
      DWORD th32ThreadID; // идентификатор потока
      DWORD th32OwnerProcessID;  // идентификатор родительского процесса
      LONG tpBasePri;  // базовый приоритет потока

      LONG tpDeltaPri;  // не используется и должен быть 0
      DWORD dwFlags; // не используется и должен быть 0
} THREADENTRY32,  *PTHREADENTRY32;
```

7. Подведение итогов

Подвести общие итоги занятия. Ещё раз отметить, чем отличается многопоточность от многозадачности. Подчеркнуть, в чем заключается актуальность построения многопоточных приложений. Напомнить слушателям, в чем состоит идея вытесняющего планирования на основе приоритетов. Акцентировать внимание слушателей на наиболее тонких моментах изложенной темы.

8. Домашнее задание

Разработать приложение «Поиск файлов и папок», выполняющее поиск по заданной маске. В маске можно использовать «*» (любые символы в любом количестве), а также «?» (один любой символ). На форме диалога предусмотреть текстовое поле (Edit) для ввода маски, комбинированный список (ComboBox) для выбора диска, на котором производить поиск, а также список (ListBox) для отображения имён найденных файлов и папок. Поиск файлов и папок следует выполнить в отдельном потоке.