Модуль №8

Занятие №2

Версия 1.0.1

План занятия:

1. Повторение пройденного материала.
2. Мьютексы.
3. Семафоры.
4. Практическая часть.
5. Подведение итогов.
6. 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. В каких случаях возникает необходимость синхронизации потоков?
2. Какие существуют примитивы синхронизации?
3. Что такое атомарный доступ?
4. Какие преимущества имеют **Interlocked**-функции перед операторами C++? В каких случаях желательно их применять?
5. Что такое критическая секция?
6. Какая функция служит для инициализации объекта критической секции?
7. Какая функция позволяет потоку завладеть критической секцией?
8. Посредством какой функции поток освобождает критическую секцию?

9. Какая функция позволяет освободить ресурсы, используемые критической секцией?
10. Каким преимуществом обладает критическая секция?
11. Какой недостаток у синхронизации в пользовательском режиме?

2. Мьютексы

Рассмотрение данного примитива синхронизации традиционно следует начать с определения.

Мьютекс (**mutual exclusion** – **mutex**) – это объект ядра, который гарантируют потокам взаимоисключающий доступ к единственному ресурсу.

Мьютекс содержит следующие поля:

- счетчик числа пользователей;
- счетчик рекурсии;
- идентификатор потока - владельца.

Акцентировать внимание слушателей на том, что мьютексы очень похожи на критические секции. Однако если критические секции являются объектами пользовательского режима, то мьютексы — это объекты ядра. Поэтому они позволяют синхронизировать доступ к ресурсу нескольких потоков из разных процессов.

Идентификатор потока-владельца определяет, какой поток захватил мьютекс, а **счетчик рекурсий** показывает, сколько раз это произошло.

Следует подчеркнуть, что мьютексы – это объекты ядра, наиболее часто используемые для синхронизации потоков. Как правило, с их помощью защищают блок памяти, к которому обращается множество потоков. Если бы потоки одновременно использовали какой-то блок памяти, то данные в нем были бы повреждены. Мьютексы гарантируют, что любой поток получает монопольный доступ к блоку памяти, и тем самым обеспечивают целостность данных.

Для использования объекта-мьютекса один из потоков должен сначала создать его вызовом функции API **CreateMutex**.

```
HANDLE  CreateMutex(
      LPSECURITY_ATTRIBUTES lpMutexAttributes, // атрибуты доступа
      BOOL bInitialOwner, // флаг наличия потока-владельца
      LPCTSTR pszName // имя объекта
);
```

Параметр **bInitialOwner** определяет начальное состояние мьютекса. Если он имеет значение **FALSE**, то объект-мьютекс не принадлежит ни одному из потоков и поэтому находится в свободном состоянии. При этом его идентификатор потока-владельца и счетчик рекурсии равны нулю. Если же в этом параметре передается значение **TRUE**, то идентификатор потока-владельца в мьютексе приравнивается идентификатору вызывающего потока, а счетчик рекурсии получает единичное значение. Если идентификатор потока-владельца не равен нулю, мьютекс считается занятым.

Необходимо отметить, что всякий раз, когда поток захватывает объект-мьютекс, счетчик рекурсии в этом объекте увеличивается на 1. Единственная ситуация, в которой значение счетчика рекурсии может быть больше 1, - поток захватывает один и тот же мьютекс несколько раз.

Любой поток может получить дескриптор существующего объекта-мьютекса, вызвав функцию API **OpenMutex**.

```
HANDLE OpenMutex(
    DWORD dwDesiredAccess,// права доступа (MUTEX_ALL_ACCESS – полный доступ)
    BOOL blnheritHandle, // параметр определяет, будет ли наследоваться
      // дескриптор мьютекса (если TRUE – дескриптор наследуемый)
    LPCTSTR pszName // имя мьютекса
);
```

Для получения доступа к разделяемому ресурсу поток обычно вызывает функцию **WaitForSingleObject** и передает ей дескриптор мьютекса, охраняющего этот ресурс.

```
DWORD WaitForSingleObject(
      HANDLE hHandle, // дескриптор объекта ядра «мьютекс»
      DWORD dwMilliseconds // время ожидания
);
```

Когда некоторый поток вызывает функцию **WaitForSingleObject**, параметр **hHandle** идентифицирует объект ядра «мьютекс», который может находится либо в свободном, либо в занятом состоянии. Параметр **dwMilliseconds** задает тайм-аут — временной интервал, спустя который функция возвращает управление, даже если объект остается в занятом состоянии. Если параметр **dwMilliseconds** имеет нулевое значение, то функция только проверяет состояние объекта и возвращает управление немедленно. Константа **INFINITE** в качестве значения **dwMilliseconds** задает бесконечное значение тайм-аута.

Возвращаемым значением функции **WaitForSingleObject** чаще всего является одна из следующих констант:

Константа	Интерпретация
WAIT_OBJECT_0	Контролируемый объект ядра перешел в свободное (сигнальное) состояние, т.е. произошел захват мьютекса
WAIT_TIMEOUT	Истек интервал тайм-аута, а контролируемый объект ядра остался в занятом (несигнальном) состоянии
WAIT_FAILED	Функция завершилась с ошибкой.
WAIT_ABANDONED	Объект мьютекса не был освобожден тем потоком, который им владел, по причине того, что поток был некорректно завершен. В этом случае мьютекс переводится в сигнальное состояние и им овладевает тот поток, который его ожидал.

Функция **WaitForSingleObject** проверяет у мьютекса идентификатор потока-владельца. Если идентификатор равен нулю, то ресурс свободен и вызывающий поток может продолжить выполнение. В

этом случае перед возвратом из функции идентификатор потока-владельца в мьютексе становится равным идентификатору вызывающего потока, а счетчику рекурсии присваивается единичное значение.

Если функция **WaitForSingleObject** определяет, что идентификатор потока-владельца не равен нулю, то вызывающий поток переходит в состояние ожидания до тех пор, пока мьютекс не перейдет в сигнальное состояние, т.е. пока захваченный мьютекс не будет освобожден.

Когда поток, занимающий ресурс, заканчивает с ним работать, он должен освободить мьютекс вызовом функции API **ReleaseMutex**.

```
BOOL ReleaseMutex(
    HANDLE hMutex // дескриптор мьютекса
);
```

Функция **ReleaseMutex** уменьшает значение счетчика рекурсии в мьютексе на единицу. Если счетчик рекурсии становится равным нулю, то обнуляется также идентификатор потока-владельца.

Акцентировать внимание слушателей на том, что **все проверки и изменения состояния объекта-мьютекса выполняются на уровне атомарного доступа**.

Как было отмечено ранее, система допускает рекурсивный многократный захват мьютекса одним потоком. В этом случае счетчик рекурсии в мьютексе каждый раз увеличивается на единицу. Важно отметить, что в этой ситуации для освобождения мьютекса количество вызовов потоком функции **ReleaseMutex** должно совпадать с количеством предыдущих захватов мьютекса.

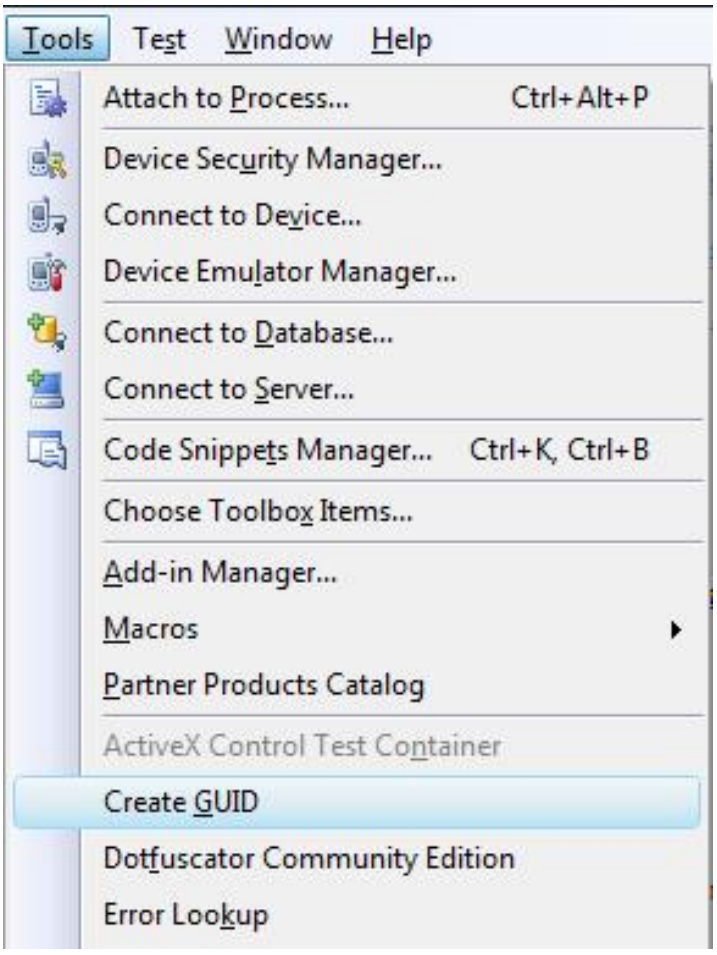
В качестве примера синхронизации потоков с помощью мьютексов привести слушателям следующее приложение, которое позволяет себя запустить только в одной копии.

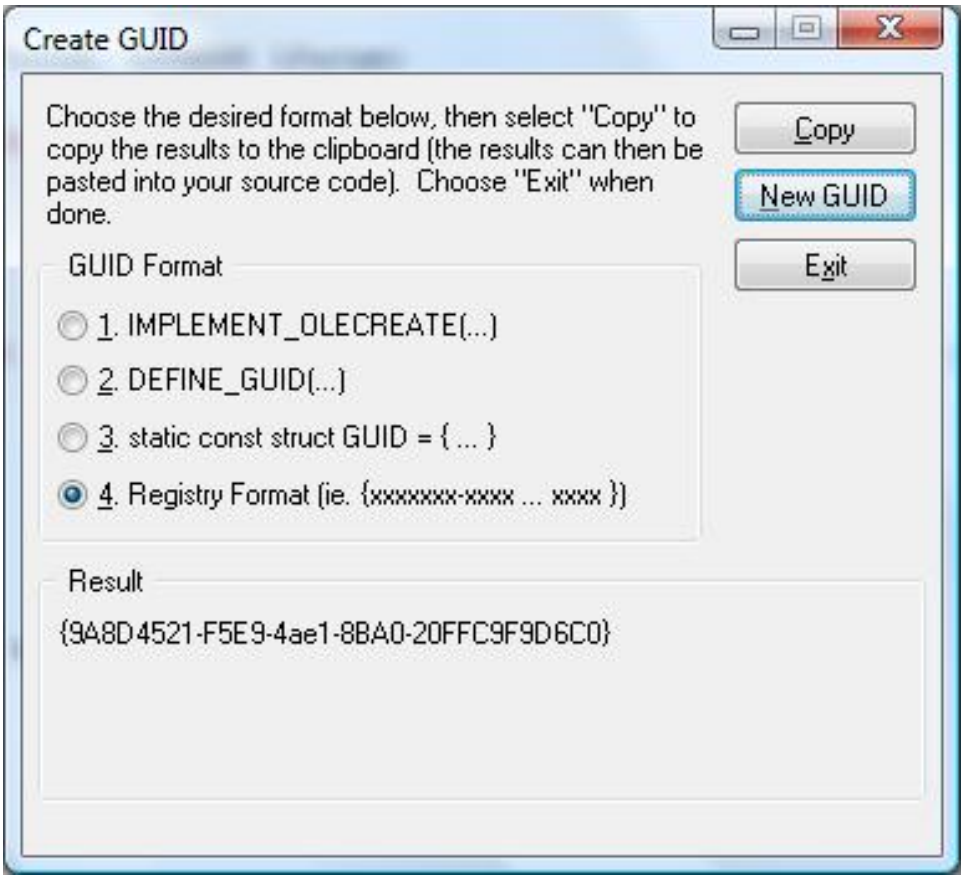
```
                                    // header.h
#pragma once

#include<windows.h>
#include <windowsX.h>
#include <tchar.h>
#include"resource.h"


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

class OnlyOneCopyDlg
{
public:
      OnlyOneCopyDlg(void);
public:
      ~OnlyOneCopyDlg(void);
      static BOOL CALLBACK DlgProc(HWND hWnd, UINT mes, WPARAM wp, LPARAM lp);
      static OnlyOneCopyDlg* 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);
      HANDLE hMutex;

};

                                  // OnlyOneCopyDlg.cpp


#include "OnlyOneCopyDlg.h"

OnlyOneCopyDlg* OnlyOneCopyDlg::ptr = NULL;

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

OnlyOneCopyDlg::~OnlyOneCopyDlg(void)
{
      ReleaseMutex(hMutex);
}

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


BOOL OnlyOneCopyDlg::Cls_OnInitDialog(HWND hwnd, HWND hwndFocus,
                                      LPARAM lParam)
{
      TCHAR GUID[] = TEXT("{D99CD3E0-670D-4def-9B74-99FD7E793DFB}");
      hMutex = CreateMutex(NULL, FALSE, GUID);
      DWORD dwAnswer = WaitForSingleObject(hMutex, 0);
```

```
      if (dwAnswer == WAIT_TIMEOUT)
      {
            MessageBox(hwnd,
            TEXT("Нельзя запускать более одной копии приложения!!!"),
            TEXT("Мьютекс"), MB_OK | MB_ICONINFORMATION);
            EndDialog(hwnd, 0);
      }
      return TRUE;
}


void OnlyOneCopyDlg::Cls_OnCommand(HWND hwnd, int id, HWND hwndCtl,
                                   UINT codeNotify)
{
      if(id == IDOK || id == IDCANCEL)
            EndDialog(hwnd, 0);
}


BOOL CALLBACK OnlyOneCopyDlg::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;

}

                              // OnlyOneCopy.cpp


#include "OnlyOneCopyDlg.h"

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

}
```

Как видно из вышеприведенного кода, в качестве имени мьютекса используется **GUID** (**Globally Unique Identifier**) — глобальный уникальный идентификатор. Это рекомендуется для того, чтобы гарантировать уникальность имени мьютекса.

Создать **GUID** можно средствами **Micrisoft Visual Studio**.


Tools
Test
Window
Help
Attach to Process... Ctrl+Alt+P
Device Security Manager...
Connect to Device...
Device Emulator Manager...
Connect to Database...
Connect to Server...
Code Snippets Manager... Ctrl+K, Ctrl+B
Choose Toolbox Items...
Add-in Manager...
Macros
Partner Products Catalog
ActiveX Control Test Container
Create GUID
Dotfuscator Community Edition
Error Lookup


Create GUID
Choose the desired format below, then select "Copy" to copy the results to the clipboard (the results can then be pasted into your source code). Choose "Exit" when done.
Copy
New GUID
Exit
GUID Format
1. IMPLEMENT_OLECREATE(...)
2. DEFINE_GUID(...)
3. static const struct GUID = { ... }
4. Registry Format (ie. {xxxxxxx-xxxx ... xxxx })
Result
{9A8D4521-F5E9-4ae1-8BA0-20FFC9F9D6C0}

Привести слушателям ещё один пример синхронизации потоков с помощью мьютексов.

В данном примере создаются два потока **CodingThread** и **DecodingThread**. Поток **CodingThread** открывает на чтение исходный текстовый файл и выполняет его шифрование, сохраняя эту информацию в файл **coding.txt** (**разделяемый ресурс**). Поток **DecodingThread** декодирует файл **coding.txt**, тем самым создавая копию исходного текстового файла.

Потоки **CodingThread** и **DecodingThread** принадлежат разным процессам, которые, в свою очередь, создаются некоторым родительским процессом.

```
/* В приложении CodingThread создается поток, который открывает на чтение исходный текстовый файл и выполняет его шифрование, сохраняя эту информацию в файл coding.txt */

                                    // header.h
#pragma once

#include<windows.h>
#include <windowsX.h>
#include <fstream>
#include <tchar.h>
#include"resource.h"

using namespace std;
                                // CodingThreadDlg.h
#pragma once
#include "header.h"

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

                                // CodingThreadDlg.cpp
#include "CodingThreadDlg.h"
```

```
CodingThreadDlg* CodingThreadDlg::ptr = NULL;

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

CodingThreadDlg::~CodingThreadDlg(void)
{

}

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

DWORD WINAPI Coding_Thread(LPVOID lp)
{
	CodingThreadDlg *ptr = (CodingThreadDlg *)lp;
	char buf[4096];
	ifstream in(TEXT("music.txt"), ios::binary | ios::in);
	if(!in)
	{
		MessageBox(ptr->hDialog, TEXT("Ошибка открытия файла!"),
				TEXT("Мьютекс"), MB_OK | MB_ICONINFORMATION);
		return 1;
	}
	HANDLE hMutex = OpenMutex(MUTEX_ALL_ACCESS, false,
				TEXT("{B8A2C367-10FE-494d-A869-841B2AF972E0}"));
	DWORD dwAnswer = WaitForSingleObject(hMutex, INFINITE);
	if(dwAnswer == WAIT_OBJECT_0)
	{
		ofstream out("coding.txt", ios::binary | ios::out | ios::trunc);
		const int KEY = 100;
		while(!in.eof())
		{
			in.read(buf, 4096);
			int n = in.gcount();
			for(int i = 0; i < n; i++)
			{
				buf[i] ^= KEY;
			}
			out.write(buf, n);
		}
		out.close();
		ReleaseMutex(hMutex);
		MessageBox(ptr->hDialog,
				TEXT("Запись данных в файл coding.txt завершена!"),
				TEXT("Мьютекс"), MB_OK | MB_ICONINFORMATION);
	}
	in.close();
	return 0;
}

BOOL CodingThreadDlg::Cls_OnInitDialog(HWND hwnd, HWND hwndFocus,
					LPARAM lParam)
{
	hDialog = hwnd;
```

```
	CreateThread(NULL, 0, Coding_Thread, this, 0, NULL);
	return TRUE;
}

BOOL CALLBACK CodingThreadDlg::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;
}
```

```
                              // CodingThread.cpp
#include "CodingThreadDlg.h"

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

```
/* В приложении DecodingThread создается поток, который получает доступ к
разделяемому ресурсу coding.txt и выполняет его декодирование */
```

```
                                  // header.h
#pragma once

#include<windows.h>
#include <windowsX.h>
#include <tchar.h>
#include <fstream>
#include"resource.h"

using namespace std;
```

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

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

```
	HWND hDialog;
};
                              // DecodingThreadDlg.h


#include "DecodingThreadDlg.h"

DecodingThreadDlg* DecodingThreadDlg::ptr = NULL;

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

DecodingThreadDlg::~DecodingThreadDlg(void)
{

}

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

DWORD WINAPI Decoding_Thread(LPVOID lp)
{
	DecodingThreadDlg *ptr = (DecodingThreadDlg *)lp;
	char buf[4096];
	HANDLE hMutex = OpenMutex(MUTEX_ALL_ACCESS, false,
			TEXT("{B8A2C367-10FE-494d-A869-841B2AF972E0}"));
	DWORD dwAnswer = WaitForSingleObject(hMutex, INFINITE);
	if(dwAnswer == WAIT_OBJECT_0)
	{
		ifstream in(TEXT("coding.txt"), ios::binary | ios::in);
		if(!in)
		{
			MessageBox(ptr->hDialog, TEXT("Ошибка открытия файла!"),
				TEXT("Мьютекс"), MB_OK | MB_ICONINFORMATION);
			return 1;
		}
		ofstream out("copymusic.txt",ios::binary | ios::out | ios::trunc);
		const int KEY = 100;
		while(!in.eof())
		{
			in.read(buf, 4096);
			int n = in.gcount();
			for(int i = 0; i < n; i++)
			{
				buf[i] ^= KEY;
			}
			out.write(buf, n);
		}
		out.close();
		in.close();
		ReleaseMutex(hMutex);
		MessageBox(ptr->hDialog,
			TEXT("Чтение данных из файла coding.txt завершено!"),
			TEXT("Мьютекс"), MB_OK | MB_ICONINFORMATION);
	}
```

```
	return 0;
}

BOOL DecodingThreadDlg::Cls_OnInitDialog(HWND hwnd, HWND hwndFocus,
                                         LPARAM lParam)
{
	hDialog = hwnd;
	CreateThread(NULL, 0, Decoding_Thread, this, 0, NULL);
	return TRUE;
}

BOOL CALLBACK DecodingThreadDlg::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;
}
```

```
                              // DecodingThread.h
#include "DecodingThreadDlg.h"

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


/* В приложении ParentProcess создаются два дочерних процесса, исполняющие
код приложений, представленных выше */
                                    // header.h
#pragma once

#include<windows.h>
#include <windowsX.h>
#include <tchar.h>
#include"resource.h"

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

class ParentProcessDlg
{
public:
	ParentProcessDlg(void);
public:
	~ParentProcessDlg(void);
	static BOOL CALLBACK DlgProc(HWND hWnd, UINT mes, WPARAM wp, LPARAM lp);
	static ParentProcessDlg* ptr;
```

```
      void Cls_OnClose(HWND hwnd);
      void Cls_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify);
};

                              // ParentProcessDlg.cpp
#include "ParentProcessDlg.h"

ParentProcessDlg* ParentProcessDlg::ptr = NULL;

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

ParentProcessDlg::~ParentProcessDlg(void)
{

}

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

void MessageAboutError(DWORD dwError)
{
      LPVOID lpMsgBuf = NULL;
      TCHAR szBuf[300];
      BOOL fOK = FormatMessage(
            FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_ALLOCATE_BUFFER,
            NULL, dwError, MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
            (LPTSTR)&lpMsgBuf, 0, NULL);
      if(lpMsgBuf != NULL)
      {
            wsprintf(szBuf, TEXT("Ошибка %d: %s"), dwError, lpMsgBuf);
            MessageBox(0, szBuf, TEXT("Сообщение об ошибке"),
                        MB_OK | MB_ICONSTOP);
            LocalFree(lpMsgBuf);
      }
}

void ParentProcessDlg::Cls_OnCommand(HWND hwnd, int id, HWND hwndCtl,
                                     UINT codeNotify)
{
      if(IDC_BUTTON1 == id)
      {
            CreateMutex(0, FALSE,
                        TEXT("{B8A2C367-10FE-494d-A869-841B2AF972E0}"));
            STARTUPINFO st = {sizeof(st)};
            PROCESS_INFORMATION pr;
            TCHAR filename[20];
            wsprintf(filename, TEXT("%s"), TEXT("CodingThread.exe"));
            if (!CreateProcess(NULL, filename, NULL, NULL, 0, 0, NULL, NULL,
                              &st, &pr))
            {
                  MessageAboutError(GetLastError());
                  return;
            }
```

```
            CloseHandle(pr.hThread);
            CloseHandle(pr.hProcess);
            ZeroMemory(&st,sizeof(st));
            st.cb = sizeof(st);
            wsprintf(filename, TEXT("%s"), TEXT("DecodingThread.exe"));
            if (!CreateProcess(NULL, filename, NULL, NULL, 0, 0, NULL, NULL,
                              &st, &pr))
            {
                  MessageAboutError(GetLastError());
                  return;
            }
            CloseHandle(pr.hThread);
            CloseHandle(pr.hProcess);
      }
}

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

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

Необходимо также отметить, что помимо рассмотренной выше функции **WaitForSingleObject** иногда используется функция **WaitForMultipleObjects**. Она работает так же, как и функция **WaitForSingleObject**, но при этом позволяет ждать освобождения сразу нескольких объектов или какого-то одного объекта из заданного списка.

```
DWORD WaitForMultipleObjects(
  DWORD nCount, // количество объектов ядра - это значение должно быть в
// пределах от 1 до MAXIMUM_WAIT_OBJECTS (64)
  CONST HANDLE* lpHandles, // указатель на массив описателей объектов ядра
```

```
  BOOL fWaitAll, // Значение TRUE задает режим ожидания  освобождения всех
// указанных объектов ядра, а FALSE — только одного из них.
  DWORD dwMilliseconds // время ожидания
);
```

Возвращаемое функцией значение сообщает, почему возобновилось выполнение вызвавшего ее потока. Значения **WAIT_TIMEOUT** и **WAIT_FAILED** интерпретируются по аналогии с функцией **WaitForSingleObject**. Если параметр **fWaitAll** равен **TRUE** и все объекты перешли в свободное состояние, то функция возвращает значение **WAIT_OBJECT_0**. Если же **fWaitAll** имеет значение **FALSE**, то функция возвращает управление, как только освобождается любой из объектов. При этом ее код возврата лежит в интервале от **WAIT_OBJECT_0** до **WAIT_OBJECT_0** + **nCount** - 1. Иными словами, если из кода возврата вычесть константу **WAIT_OBJECT_0**, то получится индекс освободившегося объекта в массиве **lpHandles**.

Акцентировать внимание слушателей на том, что обе **WAIT**-функции не тратят процессорное время, пока ждут освобождения объекта или наступления тайм-аута.

3. Семафоры

Рассмотрение данного примитива синхронизации традиционно следует начать с определения.

Семафор – это объект синхронизации, который предоставляет доступ к разделяемому ресурсу ограниченному количеству потоков.

Семафор содержит:

- счетчик числа пользователей;
- максимальное количество ресурсов, контролируемых семафором;
- счетчик текущего числа ресурсов.

Объект ядра «семафор» создается вызовом функции API **CreateSemaphore**.

```
  HANDLE CreateSemaphore(
         PSECURITY_ATTRIBUTE psa,// атрибуты безопасности
         LONG lInitialCount, // текущее количество доступных ресурсов
         LONG lMaximumCount, // максимальное количество ресурсов
         PCSTR pszName // имя семафора
);
```

Синхронизация работы потоков с использованием семафоров состоит в следующем. Предположим, что необходимо предоставить доступ к какому-то ресурсу трем потокам. Сначала объект ядра «семафор» инициализируется и ему передается количество потоков (в нашем случае 3), которые к нему могут обратиться. Далее при каждом обращении к семафору его счетчик ресурсов уменьшается. И когда счетчик уменьшиться до 0, то к семафору нельзя будет больше обратиться. При отсоединении потоков от семафора его счетчик ресурсов увеличивается, что позволяет другим потокам обращаться к нему.

Следует подчеркнуть, что сигнальному состоянию семафора соответствует значение счетчика ресурсов, большее нуля. Когда счетчик равен нулю, семафор считается не установленным (сброшенным).

Любой поток может получить дескриптор существующего объекта-семафора, вызвав функцию API **OpenSemaphore**.

```
HANDLE OpenSemaphore(
      DWORD dwDesiredAccess, // права доступа
      // (SEMAPHORE_ALL_ACCESS – полный доступ)
    BOOL blnheritHandle, // параметр определяет, будет ли наследоваться
      // дескриптор семафора (если TRUE – дескриптор наследуемый)
    LPCTSTR pszName // имя семафора
);
```

Поток может увеличить значение счетчика текущего числа доступных ресурсов на величину **lReleaseCount**, вызывая функцию API **ReleaseSemaphore**.

```
BOOL ReleaseSemaphore(
      HANDLE hSemaphore, // дескриптор семафора
      LONG lReleaseCount, // приращение количества доступных ресурсов
      LPLONG lpPreviousCount // предыдущее значение счетчика ресурсов
);
```

Следует отметить, что для получения доступа к разделяемому ресурсу поток вызывает функцию **WaitForSingleObject** и передает ей дескриптор семафора, охраняющего этот ресурс. Функция **WaitForSingleObject** проверяет у семафора значение счетчика текущего числа ресурсов. Если оно больше нуля и семафор свободен, то значение счетчика уменьшается на единицу, а вызывающий поток продолжает выполнение. При этом важно, что эта **операция выполняется для семафора на уровне атомарного доступа**. Иначе говоря, пока wait-функция не вернет управление, операционная система не позволит прервать эту операцию никакому другому потоку.

Если wait-функция определяет, что счетчик текущего числа ресурсов равен нулю (семафор занят), то система переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не увеличит значение этого счетчика.

В качестве примера синхронизации потоков с помощью семафоров привести слушателям следующее приложение.

```
                                    // header.h
#pragma once

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

```
                                    // SemaphoreDlg.h

#pragma once
#include "header.h"

class CSemaphoreDlg
{
public:
      CSemaphoreDlg(void);
public:
      ~CSemaphoreDlg(void);
      static BOOL CALLBACK DlgProc(HWND hWnd, UINT mes, WPARAM wp, LPARAM lp);
      static CSemaphoreDlg* 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, hEdit4, hEdit5;
};

                                  // SemaphoreDlg.cpp
#include "SemaphoreDlg.h"

CSemaphoreDlg* CSemaphoreDlg::ptr = NULL;

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

CSemaphoreDlg::~CSemaphoreDlg(void)
{

}

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

BOOL CSemaphoreDlg::Cls_OnInitDialog(HWND hwnd, HWND hwndFocus,
                                     LPARAM lParam)
{
      hEdit1 = GetDlgItem(hwnd, IDC_EDIT1);
      hEdit2 = GetDlgItem(hwnd, IDC_EDIT2);
      hEdit3 = GetDlgItem(hwnd, IDC_EDIT3);
      hEdit4 = GetDlgItem(hwnd, IDC_EDIT4);
      hEdit5 = GetDlgItem(hwnd, IDC_EDIT5);
      return TRUE;
}

DWORD WINAPI Thread(LPVOID lp)
{
      HWND hEdit = (HWND) lp;
      HANDLE h = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE,
                               TEXT("{41B4DBD4-F00A-4999-BFA9-1A20D12591B1}"));
      if (WaitForSingleObject(h, INFINITE) == WAIT_OBJECT_0)
      {
```

```
            for (int i = 0; i <= 50; i++)
            {
                  TCHAR str[5];
                  wsprintf(str, TEXT("%d"), i);
                  Sleep(100);
                  SetWindowText(hEdit, str);
            }
            ReleaseSemaphore(h, 1, NULL);
      }
      CloseHandle(h);
      return 1;
}

void CSemaphoreDlg::Cls_OnCommand(HWND hwnd, int id, HWND hwndCtl,
                                  UINT codeNotify)
{
      if(id == IDC_BUTTON1)
      {
            HANDLE hSemaphore = CreateSemaphore(NULL, 3, 3,
                              TEXT("{41B4DBD4-F00A-4999-BFA9-1A20D12591B1}"));
            HANDLE hThread = CreateThread(NULL, 0, Thread, hEdit1, 0, NULL);
            CloseHandle(hThread);
            hThread = CreateThread(NULL, 0, Thread, hEdit2, 0, NULL);
            CloseHandle(hThread);
            hThread = CreateThread(NULL, 0, Thread, hEdit3, 0, NULL);
            CloseHandle(hThread);
            hThread = CreateThread(NULL, 0, Thread, hEdit4, 0, NULL);
            CloseHandle(hThread);
            hThread = CreateThread(NULL, 0, Thread, hEdit5, 0, NULL);
            CloseHandle(hThread);
      }
}

BOOL CALLBACK CSemaphoreDlg::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;
}
```

```
                                  // Semaphore.cpp
#include "SemaphoreDlg.h"

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

Привести слушателям ещё один пример синхронизации потоков с помощью семафоров.

В данном примере создается поток **CodingThread**, а также три потока **DecodingThread**. Поток **CodingThread** открывает на чтение исходный текстовый файл и выполняет его шифрование, сохраняя эту информацию в файл **coding.txt** (**разделяемый ресурс**). Параллельно работающие потоки **DecodingThread** декодируют файл **coding.txt**, тем самым создавая копии исходного текстового файла.

Все потоки принадлежат разным процессам, которые, в свою очередь, создаются некоторым родительским процессом.

```
/* В приложении CodingThread создается поток, который открывает на чтение исходный текстовый файл и выполняет его шифрование, сохраняя эту информацию в файл coding.txt */

                                    // header.h

#pragma once

#include<windows.h>
#include <windowsX.h>
#include <fstream>
#include <tchar.h>
#include"resource.h"

using namespace std;


                                // CodingThreadDlg.h

#pragma once
#include "header.h"

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


                                // CodingThreadDlg.cpp

#include "CodingThreadDlg.h"
```

```
CodingThreadDlg* CodingThreadDlg::ptr = NULL;

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

CodingThreadDlg::~CodingThreadDlg(void)
{

}

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

DWORD WINAPI Coding_Thread(LPVOID lp)
{
	CodingThreadDlg *ptr = (CodingThreadDlg *)lp;
	char buf[4096];
	ifstream in(TEXT("music.txt"), ios::binary | ios::in);
	if(!in)
	{
		MessageBox(ptr->hDialog, TEXT("Ошибка открытия файла!"),
				TEXT("Семафор"), MB_OK | MB_ICONINFORMATION);
		return 1;
	}
	HANDLE hSemaphore = OpenSemaphore(SEMAPHORE_ALL_ACCESS, false,
				TEXT("{2525FD5F-12E6-47c0-838A-7C5CA1EBD169}"));
	DWORD dwAnswer = WaitForSingleObject(hSemaphore, INFINITE);
	if(dwAnswer == WAIT_OBJECT_0)
	{
		ofstream out("coding.txt", ios::binary | ios::out | ios::trunc);
		const int KEY = 100;
		while(!in.eof())
		{
			in.read(buf, 4096);
			int n = in.gcount();
			for(int i = 0; i < n; i++)
			{
				buf[i] ^= KEY;
			}
			out.write(buf, n);
		}
		out.close();
		ReleaseSemaphore(hSemaphore, 3, NULL);
		MessageBox(ptr->hDialog,
				TEXT("Запись данных в файл coding.txt завершена!"),
				TEXT("Семафор"), MB_OK | MB_ICONINFORMATION);
	}
	in.close();
	return 0;
}

BOOL CodingThreadDlg::Cls_OnInitDialog(HWND hwnd, HWND hwndFocus,
					LPARAM lParam)
{
```

```
	hDialog = hwnd;
	CreateThread(NULL, 0, Coding_Thread, this, 0, NULL);
	return TRUE;
}


BOOL CALLBACK CodingThreadDlg::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;
}
                              // CodingThread.cpp
#include "CodingThreadDlg.h"

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


/* В приложении DecodingThread создается поток, который получает доступ к
разделяемому ресурсу coding.txt и выполняет его декодирование */


                                  // header.h
#pragma once

#include<windows.h>
#include <windowsX.h>
#include <tchar.h>
#include <fstream>
#include"resource.h"

using namespace std;


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

class DecodingThreadDlg
{
public:
	DecodingThreadDlg(void);
public:
	~DecodingThreadDlg(void);
	static BOOL CALLBACK DlgProc(HWND hWnd, UINT mes, WPARAM wp, LPARAM lp);
	static DecodingThreadDlg* ptr;
```

```
	void Cls_OnClose(HWND hwnd);
	BOOL Cls_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam);
	HWND hDialog;
};

                            // DecodingThreadDlg.cpp
#include "DecodingThreadDlg.h"

DecodingThreadDlg* DecodingThreadDlg::ptr = NULL;

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

DecodingThreadDlg::~DecodingThreadDlg(void)
{

}

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

DWORD WINAPI Decoding_Thread(LPVOID lp)
{
	DecodingThreadDlg *ptr = (DecodingThreadDlg *)lp;
	char buf[4096];
	TCHAR str[MAX_PATH];
	_tcscpy(str, GetCommandLine());
	TCHAR seps[] = TEXT(" ");
	TCHAR *token, *last;
	token = _tcstok(str, seps);
	while(token != NULL)
	{
		token = _tcstok(NULL, seps);
		if(token)
			last = token;
	}
	wsprintf(str, TEXT("copymusic_%s.txt"), last);
	HANDLE hSemaphore = OpenSemaphore(SEMAPHORE_ALL_ACCESS, false,
				TEXT("{2525FD5F-12E6-47c0-838A-7C5CA1EBD169}"));
	DWORD dwAnswer = WaitForSingleObject(hSemaphore, INFINITE);
	if(dwAnswer == WAIT_OBJECT_0)
	{
		ifstream in(TEXT("coding.txt"), ios::binary | ios::in);
		if(!in)
		{
			MessageBox(ptr->hDialog, TEXT("Ошибка открытия файла!"),
				TEXT("Семафор"), MB_OK | MB_ICONINFORMATION);
			return 1;
		}
		ofstream out(str, ios::binary | ios::out | ios::trunc);
		const int KEY = 100;
		while(!in.eof())
		{
			in.read(buf, 4096);
```

```
                int n = in.gcount();
                for(int i = 0; i < n; i++)
                {
                        buf[i] ^= KEY;
                }
                out.write(buf, n);
            }
            out.close();
            in.close();
            ReleaseSemaphore(hSemaphore, 1, NULL);
            MessageBox(ptr->hDialog,
                        TEXT("Чтение данных из файла coding.txt завершено!"),
                        TEXT("Семафор"), MB_OK | MB_ICONINFORMATION);
      }
      return 0;
}

BOOL DecodingThreadDlg::Cls_OnInitDialog(HWND hwnd, HWND hwndFocus,
                                         LPARAM lParam)
{
      hDialog = hwnd;
      CreateThread(NULL, 0, Decoding Thread, this, 0, NULL);
      return TRUE;
}

BOOL CALLBACK DecodingThreadDlg::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;
}
                              // DecodingThread.cpp
#include "DecodingThreadDlg.h"

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


/* В приложении ParentProcess создаются четыре дочерних процесса, исполняющие
код приложений, представленных выше */
                                    // header.h
#pragma once

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

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

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

                              // ParentProcessDlg.cpp
#include "ParentProcessDlg.h"


ParentProcessDlg* ParentProcessDlg::ptr = NULL;

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

ParentProcessDlg::~ParentProcessDlg(void)
{

}

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

void MessageAboutError(DWORD dwError)
{
      LPVOID lpMsgBuf = NULL;
      TCHAR szBuf[300];
      BOOL fOK = FormatMessage(
            FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_ALLOCATE_BUFFER,
            NULL, dwError, MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
            (LPTSTR)&lpMsgBuf, 0, NULL);
      if(lpMsgBuf != NULL)
      {
            wsprintf(szBuf, TEXT("Ошибка %d: %s"), dwError, lpMsgBuf);
            MessageBox(NULL, szBuf, TEXT("Сообщение об ошибке"),
                        MB_OK | MB_ICONSTOP);
            LocalFree(lpMsgBuf);
      }
}

void ParentProcessDlg::Cls_OnCommand(HWND hwnd, int id, HWND hwndCtl,
                                     UINT codeNotify)
{
```

```
	if(IDC_BUTTON1 == id)
	{
		CreateSemaphore(NULL, 1, 3,
					TEXT("{2525FD5F-12E6-47c0-838A-7C5CA1EBD169}"));
		STARTUPINFO st = {sizeof(st)};
		PROCESS_INFORMATION pr;
		TCHAR filename[20];
		wsprintf(filename, TEXT("%s"), TEXT("CodingThread.exe"));
		if (!CreateProcess(NULL, filename, NULL, NULL, 0, 0, NULL, NULL,
					&st, &pr))
		{
			MessageAboutError(GetLastError());
			return;
		}
		CloseHandle(pr.hThread);
		CloseHandle(pr.hProcess);
		ZeroMemory(&st, sizeof(st));
		st.cb = sizeof(st);
		for(int i = 1; i <= 3; i++)
		{
			TCHAR buf[30];
			ZeroMemory(&st, sizeof(st));


			st.cb = sizeof(st);
			wsprintf(buf, TEXT("DecodingThread.exe %d"), i);
			if (!CreateProcess(NULL, buf, NULL, NULL, 0, 0, NULL, NULL,
						&st, &pr))
			{
				MessageAboutError(GetLastError());
			}
			CloseHandle(pr.hThread);
			CloseHandle(pr.hProcess);
		}
	}
}

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

```
// ParentProcess.cpp
#include "ParentProcessDlg.h"

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

4. Практическая часть

Разработать приложение, которое можно будет запускать не более чем в трех копиях.

5. Подведение итогов

Подвести общие итоги занятия. Подчеркнуть основное отличие мьютекса от критической секции. Еще раз вкратце повторить основные принципы синхронизации потоков с помощью мьютексов и семафоров. Отметить достоинства и недостатки механизма синхронизации потоков в режиме ядра. Акцентировать внимание слушателей на наиболее тонких моментах изложенной темы.

6. Домашнее задание

1) Реализовать задание из предыдущего занятия, используя мьютексы для синхронизации работы потоков. При этом необходимо, чтобы потоки **WriteToFiles** и **ReadFromFiles** принадлежали разным процессам, которые, в свою очередь, должны быть созданы некоторым родительским процессом. Таким образом, данное задание предполагает наличие трех процессов, и, следовательно, трех приложений.

2) Реализовать это же задание, используя семафоры для синхронизации работы потоков. В этом случае предусмотреть четыре потока – один записывающий поток **WriteToFiles** и три читающих потока **ReadFromFiles,** каждый из которых создаёт свой результирующий файл из копий. При этом необходимо, чтобы все потоки принадле-

жали разным процессам, которые, в свою очередь, должны быть созданы некоторым родительским процессом.