Модуль №8

Занятие №1

Версия 1.0.1

План занятия:

1. Повторение пройденного материала.
2. Проблематика синхронизации потоков.
3. Блокированные вызовы.
4. Критические секции.
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. Что такое поток? Какие бывают потоки?
2. Что такое многопоточность? Чем отличается многопоточность от многозадачности?
3. В чем заключается актуальность построения многопоточных приложений?
4. Из каких компонент состоит поток?
5. Что такое потоковая функция?
6. Какая функция API позволяет создать вторичный поток?
7. Какие существуют способы завершения потока?
8. Какой способ завершения потока считается наиболее корректным?
9. Какая функция API позволяет приостановить поток на определенный период времени?

10. Посредством какой функции API можно «усыпить» поток?
11. Посредством какой функции API можно возобновить работу потока?
12. В чем состоит идея вытесняющего планирования на основе приоритетов?
13. Как определяется суммарный приоритет потока?
14. Какая функция API предназначена для изменения класса приоритета процесса?
15. Какая функция API предназначена для получения класса приоритета процесса?
16. Какая функция API позволяет изменить относительный приоритет потока?
17. Какая функция API позволяет получить относительный приоритет потока?
18. Какая функция API используется для получения дескриптора текущего процесса?
19. Какая функция API используется для получения дескриптора текущего потока?
20. Какая функция API используется для получения идентификатора текущего процесса?
21. Какая функция API используется для получения идентификатора текущего потока?
22. Что такое локальная память потока? Какие существуют виды TLS?
23. Какие функции библиотеки **Tool Help API** позволяют получить информацию о потоках в снимке системы?

2. Проблематика синхронизации потоков

В ходе рассмотрения данного вопроса необходимо обозначить проблему, для решения которой применяются различные механизмы

синронизации потоков. В частности, отметить, что все потоки в системе должны иметь доступ к различным системным ресурсам — кучам, последовательным портам, файлам, окнам и т.д. Если один из потоков запросит монопольный доступ к какому-либо ресурсу, другим потокам, которым тоже нужен этот ресурс, не удастся выполнить свои задачи. С другой стороны, просто недопустимо, чтобы потоки бесконтрольно пользовались ресурсами. Например, если один поток записывает информацию в файл, все другие потоки не должны в этот момент времени использовать данный файл. Таким образом, возникает необходимость синхронизировать работу потоков, желающих получить доступ к одному и тому же ресурсу.

Другой причиной может быть ожидание одним потоком некоторого события, которое может наступить лишь при выполнении другого потока. Для таких случаев должны быть предусмотрены специальные средства, с помощью которых первый поток будет переведен в состояние ожидания до возникновения соответствующего события, а после этого продолжит выполнение.

Таким образом, следует выделить два основных случая, когда потоки должны взаимодействовать друг с другом:

- совместное использование разделяемого ресурса (чтобы не разрушить его);
- необходимость уведомления других потоков о завершении каких-либо операций.

Акцентировать внимание слушателей на том, что синхронизацию потоков обычно осуществляют с использованием **примитивов синхронизации**, таких как:

- атомарные операции API-уровня;
- критические секции;
- события;
- ожидаемые таймеры;
- семафоры;

- мьютексы.

На этом и последующих занятиях необходимо детально ознакомить слушателей с каждым из вышеперечисленных примитивов синхронизации.

3. Блокированные вызовы

Большая часть синхронизации потоков связана с **атомарным доступом** (**atomic access**) — монопольным захватом ресурса обращающимся к нему потоком. Привести слушателям следующий пример, обозначив проблемную ситуацию.

```
LONG g = 0; // определяем глобальную переменную

DWORD WINAPI Thread1( LPVOID lp )
{
  ++g;
  return 0;
}

DWORD WINAPI Thread2( LPVOID lp )
{
  ++g;
  return 0;
}
```

Как видно из вышеприведенного примера, код обеих потоковых функций идентичен: обе функции увеличивают значение глобальной переменной **g** на 1. Поскольку начальное значение переменной **g** равно 0, то, когда оба потока завершат свою работу, значение данной переменной станет равным 2. Однако в реальности ситуация может быть несколько иной.

Далее следует представить слушателям код потоковых функций, записанный на ассемблере.

```
; поток1
mov ax, g; в регистр помещается значение переменной g
inc ax; значение регистра увеличивается на 1
```

```
    mov g, ax; в переменную g копируется значение регистра
    ; поток2
    mov ax, g; в регистр помещается значение переменной g
    inc ax; значение регистра увеличивается на 1
    mov g, ax; в переменную g копируется значение регистра
```

Следует отметить, что при параллельной работе потоков вполне допустима следующая последовательность выполнения ассемблерного кода.

```
MOV EAX, g; выполняется поток 1 - в регистр помещается 0
INC EAX; выполняется поток 1 - значение регистра увеличивается на 1
MOV EAX, g; выполняется поток 2 - в регистр помещается 0
INC EAX; выполняется поток 2 - значение регистра увеличивается на 1
MOV g, EAX; выполняется поток 1 - значение 1 помещается в переменную g
MOV g, EAX; выполняется поток 2 - значение 1 помещается в переменную g
```

В итоге вместо ожидаемого результата в переменной **g** окажется 1. Акцентировать внимание слушателей, что для решения данной проблемы необходим механизм, гарантирующий приращение значения переменной **g** на уровне **атомарного доступа**, т. е. без прерывания другими потоками. Следует подчеркнуть, что Win32 API предоставляет семейство **Interlocked-функций** для реализации взаимно блокированных операций. При этом все Interlocked-функции работают корректно только при условии, что их аргументы выровнены по границе двойного слова (**DWORD**).

Для увеличения значения переменной на единицу предназначена функция API **InterlockedIncrement**.

```
LONG InterlockedIncrement(
    LPLONG lpAddend // адрес переменной, значение которой инкрементируется
);
```

Для уменьшения значения переменной на единицу предназначена функция API **InterlockedDecrement**.

```
LONG InterlockedDecrement(
    LPLONG lpAddend // адрес переменной, значение которой декрементируется
);
```

Функция API **InterlockedExchange** монопольно заменяет текущее значение переменной типа **LONG**, адрес которой передается в первом параметре, значением, передаваемым во втором параметре.

```
LONG InterlockedExchange(
        PLONG plTarget,// адрес переменной, значение которой заменяется
        LONG lValue // значение для замены
  );
```

Функция API **InterlockedExchangeAdd** добавляет к значению переменной, адрес которой передается в первом параметре, значение, передаваемое во втором параметре:

```
LONG InterlockedExchangeAdd(
      PLONG plAddend, // адрес переменной, значение которой изменяется
      LONG lIncrement // значение, на которое увеличивается переменная
      );
```

Функция API **InterlockedCompareExchange** выполняет операцию сравнения и присваивания по результату сравнения:

```
LONG InterlockedCompareExchange(
      LPLONG  plDestination, // адрес переменной, значение которой изменяется
      LONG lExchange, // значение для замены
      LONG lComperand // значение для сравнения
   );
```

Если значение переменной, адрес которой передается в первом параметре, совпадает со значением, передаваемым в третьем параметре, то оно заменяется значением, передаваемым во втором параметре.

Возвращаясь к примеру, рассмотренному ранее, следует привести слушателям исправленную версию, в которой происходит приращение переменной **g** на уровне атомарного доступа посредством функции **InterlockedIncrement**.

```
LONG g = 0; // определяем глобальную переменную

DWORD WINAPI Thread1( LPVOID lp )
{
   InterlockedIncrement(&g);
   return 0;
}

DWORD WINAPI Thread2( LPVOID lp )
{
   InterlockedIncrement(&g);
   return 0;
}
```

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

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

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

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

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

```
CInterlockDlg* CInterlockDlg::ptr = NULL;

LONG counter = 0;

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

CInterlockDlg::~CInterlockDlg(void)
{

}

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


BOOL CInterlockDlg::Cls_OnInitDialog(HWND hwnd, HWND hwndFocus,
                                     LPARAM lParam)
{
	hEdit1 = GetDlgItem(hwnd, IDC_EDIT1);
	hEdit2 = GetDlgItem(hwnd, IDC_EDIT2);
	return TRUE;
}


DWORD WINAPI Thread1(LPVOID lp)
{
	HWND hWnd = HWND(lp);
	for (int i = 0; i < 50000000; i++)
	{
		++counter;
	}
	TCHAR str[10];
	wsprintf(str, TEXT("%d"), counter);
	SetWindowText(hWnd, str);
	return 0;
}

DWORD WINAPI Thread2(LPVOID lp)
{
	HWND hWnd = HWND(lp);
	for (int i = 0; i < 50000000; i++)
	{
		--counter;
	}
	TCHAR str[10];
	wsprintf(str, TEXT("%d"), counter);
	SetWindowText(hWnd, str);
	return 0;
}

DWORD WINAPI Thread3(LPVOID lp)
{
	HWND hWnd = HWND(lp);
```

```
	for (int i = 0; i < 50000000; i++)
	{
		InterlockedIncrement(&counter);
	}

	TCHAR str[10];
	wsprintf(str, TEXT("%d"), counter);
	SetWindowText(hWnd, str);
	return 0;
}


DWORD WINAPI Thread4(LPVOID lp)
{
	HWND hWnd = HWND(lp);
	for (int i = 0; i < 50000000; i++)
	{
		InterlockedDecrement(&counter);
	}

	TCHAR str[10];
	wsprintf(str, TEXT("%d"), counter);
	SetWindowText(hWnd, str);
	return 0;
}

void CInterlockDlg::Cls_OnCommand(HWND hwnd, int id, HWND hwndCtl,
                                  UINT codeNotify)
{
	if(id == IDC_BUTTON1)
	{
		counter = 0;
		HANDLE hThread = CreateThread(NULL, 0, Thread1, hEdit1, 0, NULL);
		CloseHandle(hThread);
		hThread = CreateThread(NULL, 0, Thread2, hEdit1, 0, NULL);
		CloseHandle(hThread);
	}
	else if(id == IDC_BUTTON2)
	{
		counter = 0;
		HANDLE hThread = CreateThread(NULL, 0, Thread3, hEdit2, 0, NULL);
		CloseHandle(hThread);
		hThread = CreateThread(NULL, 0, Thread4, hEdit2, 0, NULL);
		CloseHandle(hThread);
	}
}

BOOL CALLBACK CInterlockDlg::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;
}
```

```
                                    // Interlock.cpp
#include "InterlockDlg.h"

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

4. Критические секции

Рассмотрение данного примитива синхронизации следует начать с определения. **Критическая секция** (**critical section**) — это небольшой участок кода, требующий монопольного доступа к каким-то общим данным. Она позволяет сделать так, чтобы единовременно только один поток получал доступ к определенному ресурсу. Если в определенный момент времени несколько потоков попытаются получить доступ к критической секции, то контроль над ним будет предоставлен только одному из потоков, а все остальные будут переведены в состояние ожидания до тех пор, пока участок не освободится.

Для использования критической секции необходимо определить переменную типа **CRITICAL_SECTION**. Поскольку эта переменная должна находиться в области видимости для каждого использующего ее потока, обычно ее объявляют глобальной. Кроме того, эту переменную следует инициализировать до ее первого применения с помощью функции API **InitializeCriticalSection**:

```
void InitializeCriticalSection(
      LPCRITICAL_SECTION lpCriticalSection // указатель на объект критической
      // секции
);
```

Чтобы завладеть критической секцией, поток должен вызвать функцию API **EnterCriticalSection**.

```
void EnterCriticalSection(
      LPCRITICAL_SECTION lpCriticalSection // указатель на объект критической
      // секции
);
```

Важно отметить, что если критическая секция не используется в данный момент другим потоком, то текущий поток захватывает ее и выполняет инструкции этой секции, которые следуют сразу после вызова функции **EnterCriticalSection**. При этом критическая секция обозначается системой как занятая.

Если критическая секция в данный момент уже используется другим потоком, то текущий поток блокируется до тех пор, пока секция не будет освобождена функцией API **LeaveCriticalSection**.

```
void LeaveCriticalSection(
      LPCRITICAL_SECTION lpCriticalSection // указатель на объект критической
      // секции
);
```

Следует подчеркнуть, что вызов функции **LeaveCriticalSection** определяет конец критической секции. При этом критическая секция обозначается системой как доступная.

Ещё раз акцентировать внимание слушателей на том, что как только поток получает контроль над критической секцией, доступ других потоков к этой секции блокируется. При этом очень важно, чтобы время выполнения критической секции было минимальным. Это позволит добиться наилучших результатов работы приложения.

Следует отметить, если критическая секция больше не нужна, то используемые ей ресурсы необходимо освободить вызовом функции **DeleteCriticalSection**.

```
void DeleteCriticalSection(
      LPCRITICAL_SECTION lpCriticalSection // указатель на объект критической
      // секции
);
```

Кроме перечисленных выше функций **WinAPI** предоставляет функцию **TryEnterCriticalSection**, которая позволяет осуществить попытку захватить критическую секцию.

```
void TryEnterCriticalSection(
      LPCRITICAL_SECTION lpCriticalSection // указатель на объект критической
      // секции
);
```

Например:

```
CRITICAL_SECTION cs;
      ...
BOOL bAcquired = TryEnterCriticalSection(&cs);
if(bAcquired)
{
      // Выполнение кода критической секции (монопольный доступ к ресурсу)
}
else
{
      // Контроль над критической секцией недоступен
}
```

Как следует из вышеприведенного примера, если критическая секция доступна, то поток входит в эту секцию и выполняет ее код, получая монопольной доступ к какому-то ресурсу. Если секция недоступна, то поток не блокируется, как в случае применения функции **EnterCriticalSection**, а занимается другой работой, за что отвечает ветвь **else**.

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

```
                                    // header.h
#pragma once

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

using namespace std;

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

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


                               // CriticalSectionDlg.cpp

#include "CriticalSectionDlg.h"

CriticalSectionDlg* CriticalSectionDlg::ptr = NULL;
CRITICAL_SECTION cs;

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

CriticalSectionDlg::~CriticalSectionDlg(void)
{
      DeleteCriticalSection(&cs);
}

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

BOOL CriticalSectionDlg::Cls_OnInitDialog(HWND hwnd, HWND hwndFocus,
                                          LPARAM lParam)
{
      InitializeCriticalSection(&cs);
      return TRUE;
}
```

```
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);
	}
}

DWORD WINAPI Write(LPVOID lp)
{
	srand(time(0));
	EnterCriticalSection(&cs);
	ofstream out(TEXT("array.txt"));
	if(!out.is_open())
	{
		MessageAboutError(GetLastError());
		return 1;
	}
	int A[100];
	for(int i = 0; i < 100; i++)
	{
		A[i] = rand()%50;
		out << A[i] << ' ';
	}
	out.close();
	LeaveCriticalSection(&cs);
	MessageBox(0, TEXT("Поток записал информацию в файл"),
			TEXT("Критическая секция"), MB_OK);
	return 0;
}

DWORD WINAPI Read(LPVOID lp)
{
	EnterCriticalSection(&cs);
	ifstream in(TEXT("array.txt"));
	if(!in.is_open())
	{
		MessageAboutError(GetLastError());
		return 1;
	}
	int B[100];
	int sum = 0;
	for(int i = 0; i < 100; i++)
	{
		in >> B[i];
		sum += B[i];
	}
	in.close();
```

```
	LeaveCriticalSection(&cs);
	MessageBox(0, TEXT("Поток прочитал информацию из файла"),
		TEXT("Критическая секция"), MB_OK);
	TCHAR str[30];
	wsprintf(str, TEXT("Сумма чисел равна %d"), sum);
	MessageBox(0, str, TEXT("Критическая секция"), MB_OK);
	return 0;
}


void CriticalSectionDlg::Cls_OnCommand(HWND hwnd, int id, HWND hwndCtl,
                                       UINT codeNotify)
{
	if(id == IDC_BUTTON1)
	{
		HANDLE hThread = CreateThread(NULL, 0, Write, 0, 0, NULL);
		CloseHandle(hThread);
		hThread = CreateThread(NULL, 0, Read, 0, 0, NULL);
		CloseHandle(hThread);
	}
}

BOOL CALLBACK CriticalSectionDlg::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;
}

                          // CriticalSection.cpp
#include "CriticalSectionDlg.h"

int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPrev, LPSTR lpszCmdLine,
                   int nCmdShow)
{
	CriticalSectionDlg dlg;
	return DialogBox(hInst, MAKEINTRESOURCE(IDD_DIALOG1), NULL,
                     CriticalSectionDlg::DlgProc);
}
```

Акцентировать внимание слушателей на том, что работая с критическими секциями или применяя **Interlocked**-функции, программа не переключается в режим ядра. Поэтому эти виды синхронизации называют **синхронизацией в пользовательском режиме**. Она отличается высокой скоростью реализации. Однако главным недостатком такой синхронизации является невозможность ее применения для потоков, принадлежащих разным процессам.

5. Подведение итогов

Подвести общие итоги занятия. Ещё раз обозначить причины, по которым возникает необходимость синхронизировать работу потоков. Подчеркнуть преимущество использования **Interlocked**-функций вместо операторов C++ в потоковых функциях. Отметить достоинства и недостатки применения критических секций для синхронизации потоков. Акцентировать внимание слушателей на наиболее тонких моментах изложенной темы.

6. Домашнее задание

Разработать приложение, в котором создаются два вторичных потока **WriteToFiles** и **ReadFromFiles**. Поток **WriteToFiles** открывает на чтение исходный текстовый файл и создает **N** копий этого файла. Поток **ReadFromFiles** копирует содержимое всех **N** копий в результирующий текстовый файл.

При параллельной работе обоих потоков, поток **ReadFromFiles** не должен опережать в работе поток **WriteToFiles**, т.е. **n**-я копия, не будучи созданной потоком **WriteToFiles,** не может быть открыта на чтение потоком **ReadFromFiles.**

Для синхронизации работы потоков необходимо использовать механизм критических секций.