Модуль №8

Занятие №3

Версия 1.0.1

План занятия:

1. Повторение пройденного материала.
2. События.
3. Таймер синхронизации.
4. Подведение итогов.
5. 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 мьютекс?
2. В чем заключается принцип синхронизации потоков с помощью мьютексов?
3. В чем состоит основное отличие мьютекса от критической секции?
4. Какая функция WinAPI предусмотрена для создания мьютекса?
5. Какая функция WinAPI позволяет получить дескриптор существующего мьютекса?
6. Какая функция WinAPI используется для получения доступа к разделяемому ресурсу?
7. Посредством какой функции WinAPI поток освобождает захваченный мьютекс?

8. Какая функция WinAPI позволяет ждать освобождения сразу нескольких объектов синхронизации?
9. Что такое семафор? Чем он отличается от мьютекса?
10. В чем состоит принцип синхронизации работы потоков с использованием семафоров?
11. Какая функция WinAPI предусмотрена для создания семафора?
12. Какая функция WinAPI позволяет получить дескриптор существующего семафора?
13. Какая функция WinAPI позволяет увеличить счетчик текущего числа ресурсов?

2. События.

Рассмотрение данного примитива синхронизации следует начать с определения.

Событие (event) — это объект ядра, предназначенный для синхронизации работы потоков. Чаще всего событие используется для уведомления об окончании какой-либо операции.

События бывают двух типов:

- со сбросом вручную (manual-reset events);
- с автоматическим сбросом (auto-reset events).

События с ручным сбросом позволяют возобновить выполнение сразу нескольких ждущих потоков, в то время как события с автоматическим сбросом возобновляют выполнение только одного потока.

Объект ядра «событие» создается функцией API **CreateEvent**.

```
HANDLE CreateEvent(
  LPSECURITY_ATTRIBUTES eventAttributes, // атрибуты доступа
  BOOL bManualReset, // тип сброса
  BOOL bInitialState, // начальное состояние
  LPCTSTR pszName // имя объекта
);
```

Параметр **bManualReset** определяет тип объекта-события. Значение **TRUE** создает событие со сбросом вручную, а значение **FALSE** — событие с автоматическим сбросом. Параметр **bInitialState** определяет начальное состояние события — свободное (**TRUE**) или занятое (**FALSE**). Параметр **pszName** содержит указатель на строку, в которой содержится имя объекта. Если **pszName** имеет значение NULL, то создается неименованный объект.

Синхронизация работы потоков с использованием событий состоит в следующем. Предположим, что некоторый поток, создавший объект ядра «событие», переводит его в занятое состояние и приступает к своим операциям. Закончив работу, поток сбрасывает событие в свободное состояние. В этот момент другой поток, который ждал перехода события в свободное состояние, пробуждается и становится планируемым.

Акцентировать внимание слушателей на том, что события могут использоваться для синхронизации потоков разных процессов.

Получить дескриптор существующего объекта-события можно вызовом функции **CreateEvent**, либо вызовом функции **OpenEvent**, указав в параметре **pszName** имя существующего объекта.

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

Создав событие, существует возможность напрямую управлять его состоянием. Для перевода события в свободное состояние используется функция API **SetEvent**.

```
BOOL SetEvent(
      HANDLE hEvent // дескриптор объекта ядра «событие»
);
```

Для перевода события в занятое состояние используется функция API **ResetEvent**.

```
BOOL ResetEvent(
      HANDLE hEvent // дескриптор объекта ядра «событие»
);
```

Следует отметить, что для событий с автоматическим сбросом действует следующее правило. Когда ожидание потоком освобождения события успешно завершается, то объект-событие автоматически сбрасывается в занятое состояние. Для событий со сбросом вручную автоматического сбрасывания не происходит, поэтому для возврата в занятое состояние необходимо вызвать функцию **ResetEvent**.

Необходимо ознакомить слушателей с функцией API **PulseEvent**, которая также может использоваться для управления состоянием события.

```
BOOL PulseEvent(
      HANDLE hEvent // дескриптор объекта ядра «событие»
);
```

Функция **PulseEvent** освобождает событие и тут же переводит его обратно в занятое состояние. Вызов данной функции равнозначен последовательному вызову функций **SetEvent** и **ResetEvent**. При вызове функции **PulseEvent** для события со сбросом вручную, любые потоки, ждущие этот объект, становятся планируемыми. При вызове этой функции применительно к событию с автоматическим сбросом пробуждается только один из ждущих потоков.

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

```
                                    // header.h
#pragma once

#include<windows.h>
```

```
#include <tchar.h>
#include <windowsX.h>

#include"resource.h"

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

class CEventDlg
{
public:
      CEventDlg(void);
public:
      ~CEventDlg(void);
      static BOOL CALLBACK DlgProc(HWND hWnd, UINT mes, WPARAM wp, LPARAM lp);
      static CEventDlg* 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, hEdit6;
};

                                    // EventDlg.cpp
#include "EventDlg.h"

CEventDlg* CEventDlg::ptr = NULL;

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

CEventDlg::~CEventDlg(void)
{
}

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

BOOL CEventDlg::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);
      hEdit6 = GetDlgItem(hwnd, IDC_EDIT6);
      return TRUE;
}

DWORD WINAPI Thread1(LPVOID lp)
{
      HWND hWnd = HWND (lp);
```

```
	// получим дескриптор существующего события
	HANDLE hEvent = OpenEvent(EVENT_ALL_ACCESS, 0,
				TEXT("{2BA7C99B-D9F7-4485-BB3F-E4735FFEF139}"));
	for(int i = 0; i < 5; i++)
	{
		//поток ожидает переход события в сигнальное состояние
		if(WaitForSingleObject(hEvent, INFINITE) == WAIT_OBJECT_0)
		{
			// Отпускаются все ждущие потоки и событие остаётся в
			// сигнальном состоянии, т.к. событие с ручным сбросом
			for(int i = 1; i <= 100; i++)
			{
				TCHAR str[10];
				wsprintf(str, TEXT("%d"), i);
				SetWindowText(hWnd, str);
				Sleep(10);
			}
		}
		ResetEvent(hEvent); // перевод события в несигнальное состояние
	}
	return 0;
}

DWORD WINAPI Thread2(LPVOID lp)
{
	HWND hWnd = HWND (lp);
	// получим дескриптор существующего события
	HANDLE hEvent = OpenEvent(EVENT_ALL_ACCESS, 0,
				TEXT("{ECA57A59-2BD7-4fb5-A132-7A00944F7CEF}"));
	for(int i = 0; i < 5; i++)
	{
		// поток ожидает переход события в сигнальное состояние
		if(WaitForSingleObject(hEvent, INFINITE) == WAIT_OBJECT_0)
		{
			// Отпускается один ждущий поток и
			// событие переводится в несигнальное состояние,
			// т.к. сброс автоматический
			for(int i = 1; i <= 100; i++)
			{
				TCHAR str[10];
				wsprintf(str, TEXT("%d"), i);
				SetWindowText(hWnd, str);
				Sleep(10);
			}
		}
	}
	return 0;
}

void CEventDlg::Cls_OnCommand(HWND hwnd, int id, HWND hwndCtl,
				UINT codeNotify)
{
	switch(id)
	{
		case IDC_BUTTON1:
			{
				CreateEvent(NULL, TRUE /* ручной сброс события */,
					FALSE /* несигнальное состояние */,
					TEXT("{2BA7C99B-D9F7-4485-BB3F-E4735FFEF139}"));
```

```
                        EnableWindow(GetDlgItem(hwnd, IDC_BUTTON1), 0);
                        HANDLE h = CreateThread(NULL, 0, Thread1, hEdit1, 0,
                                                NULL);
                        CloseHandle(h);
                        h = CreateThread(NULL, 0, Thread1, hEdit2, 0, NULL);
                        CloseHandle(h);
                        h = CreateThread(NULL, 0, Thread1, hEdit3, 0, NULL);
                        CloseHandle(h);
                  }
                  break;
            case IDC_BUTTON2:
                  {
                        HANDLE h = OpenEvent(EVENT_ALL_ACCESS, 0,
                              TEXT("{2BA7C99B-D9F7-4485-BB3F-E4735FFEF139}"));
                        SetEvent(h); // перевод события в сигнальное состояние
                  }
                  break;
            case IDC_BUTTON3:
                  {
                        HANDLE hEvent = CreateEvent(NULL,
                              FALSE /*автоматический сброс события */,
                              FALSE /* несигнальное состояние */,
                              TEXT("{ECA57A59-2BD7-4fb5-A132-7A00944F7CEF}"));
                        EnableWindow(GetDlgItem(hwnd, IDC_BUTTON3), FALSE);
                        HANDLE h;
                        h = CreateThread(NULL, 0, Thread2, hEdit4, 0, NULL);
                        CloseHandle(h);
                        h = CreateThread(NULL, 0, Thread2, hEdit5, 0, NULL);
                        CloseHandle(h);
                        h = CreateThread(NULL, 0, Thread2, hEdit6, 0, NULL);
                        CloseHandle(h);
                  }
            case IDC_BUTTON4:
                  {
                        HANDLE h = OpenEvent(EVENT_ALL_ACCESS, 0,
                              TEXT("{ECA57A59-2BD7-4fb5-A132-7A00944F7CEF}"));
                        SetEvent(h); // перевод события в сигнальное состояние
                  }
                  break;
      }
}

BOOL CALLBACK CEventDlg::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;
}
                                 // Event.cpp
#include "EventDlg.h"

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

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

3. Таймер синхронизации

Рассмотрение данного примитива синхронизации следует начать с определения.

Таймер синхронизации или ожидаемый таймер (waitable timer) - это объект ядра, который самостоятельно переходит в свободное состояние в определенное время или через регулярные промежутки времени.

Ожидаемый таймер аналогично событию бывают двух типов:

- со сбросом вручную;
- с автоматическим сбросом.

Таймер с ручным сбросом позволяют возобновить выполнение сразу нескольких ждущих потоков, в то время как таймер с автоматическим сбросом возобновляют выполнение только одного потока.

Для создания таймера синхронизации необходимо вызвать функцию API **CreateWaitableTimer**.

```
HANDLE CreateWaitableTimer(
        LPSECURITY_ATTRIBUTES psa, // атрибуты доступа
        BOOL bManualReset, // тип сброса
        LPCTSTR pszName // имя объекта
);
```

Параметр **bManualReset** определяет тип таймера синхронизации. Значение **TRUE** создает таймер со сбросом вручную, а значение **FALSE**

— таймер с автоматическим сбросом. Параметр **pszName** содержит указатель на строку, в которой содержится имя объекта синхронизации. Если **pszName** имеет значение NULL, то создается неименованный объект.

Получить дескриптор существующего таймера синхронизации можно вызовом функции API **OpenWaitableTimer**, указав в параметре **pszName** имя существующего объекта.

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

Объект «ожидаемый таймер» всегда создаются в занятом состоянии. Чтобы сообщить таймеру, в какой момент он должен перейти в свободное состояние, необходимо вызвать функцию API **SetWaitableTimer**.

```
BOOL SetWaitableTimer(
    HANDLE hTimer, // дескриптор таймера
    const LARGE_INTEGER pDueTime, // время первого срабатывания таймера
    LONG lPeriod, // период таймера в миллисекундах
    PTIMERAPCROUTINE pfnCompletionRoutine,// адрес асинхронно вызываемой
// APC-процедуры (asynchronous procedure call), которая вызывается в момент
// перехода таймера в сигнальное состояние
    LPVOID lpArgToCompletionRoutine,// указатель на структуру, передаваемую в
// APC-процедуру
    BOOL fResume // если данный параметр равен TRUE, при срабатывании таймера
// компьютер выйдет из режима сна (если он находился в спящем режиме), и
// возобновятся потоки, ожидавшие этот таймер.
);
```

Параметры **pDueTime** и **lPeriod** используются совместно: первый из них задает, когда таймер должен сработать в первый раз, второй определяет, насколько часто это должно происходить в дальнейшем.

Следует отметить, что чаще всего необходимо, чтобы таймер сработал только один раз - через определенное время перешел в свободное состояние и уже больше никогда не срабатывал. Для этого достаточно передать 0 в параметре **lPeriod**. Затем можно вызвать **CloseHandle**, чтобы закрыть таймер, либо перенастроить таймер повторным вызовом **SetWaitableTimer** с другими параметрами.

Для отмены ожидаемого таймера применяется функция API **CancelWaitableTimer**.

```
BOOL CancelWaitableTimer(
        HANDLE hTimer // дескриптор таймера синхронизации
 );
```

В качестве примера, демонстрирующего работу ожидаемого таймера, предлагается рассмотреть приложение «Будильник».

```
                                    // header.h
#pragma once

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


#pragma comment(lib,"comctl32")



                                // WaitableTimerDlg.h

#pragma once
#include "header.h"

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

```
	void Cls_OnSize(HWND hwnd, UINT state, int cx, int cy);
	// обработчик пользовательского сообщения
	void OnTrayIcon(WPARAM wp, LPARAM lp);
	HWND hButton, hEdit1, hSpin1, hEdit2, hSpin2, hEdit3, hSpin3, hDialog;
	HICON hIcon;
	PNOTIFYICONDATA pNID;

};

                          // WaitableTimerDlg.cpp

#define WM_ICON WM_APP
#define ID_TRAYICON WM_USER
#include "WaitableTimerDlg.h"

CWaitableTimerDlg* CWaitableTimerDlg::ptr = NULL;

CWaitableTimerDlg::CWaitableTimerDlg(void)
{
	ptr = this;
	pNID = new NOTIFYICONDATA;
}

CWaitableTimerDlg::~CWaitableTimerDlg(void)
{
	delete pNID;
}

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


BOOL CWaitableTimerDlg::Cls_OnInitDialog(HWND hwnd, HWND hwndFocus,
                                         LPARAM lParam)
{
	// Получим дескрипторы элементов управления
	hSpin1 = GetDlgItem(hwnd, IDC_SPIN2);
	hEdit1 = GetDlgItem(hwnd, IDC_EDIT1);
	hSpin2 = GetDlgItem(hwnd, IDC_SPIN3);
	hEdit2 = GetDlgItem(hwnd, IDC_EDIT2);
	hSpin3 = GetDlgItem(hwnd, IDC_SPIN4);
	hEdit3 = GetDlgItem(hwnd, IDC_EDIT3);
	hButton = GetDlgItem(hwnd, IDC_BUTTON1);
	hDialog = hwnd;
	// Установим необходимый диапазон для счётчиков
	SendMessage(hSpin1, UDM_SETRANGE32, 0, 23);
	SendMessage(hSpin2, UDM_SETRANGE32, 0, 59);
	SendMessage(hSpin3, UDM_SETRANGE32, 0, 59);
	// Получим дескриптор экземпляра приложения
	HINSTANCE hInst = GetModuleHandle(NULL);
	hIcon = LoadIcon(hInst, MAKEINTRESOURCE(IDI_ICON1)); // загружаем иконку
	// устанавливаем иконку в главном окне приложения
	SetClassLong(hDialog, GCL_HICON, LONG(hIcon));
	memset(pNID, 0, sizeof(NOTIFYICONDATA)); //Обнуление структуры
	pNID->cbSize = sizeof(NOTIFYICONDATA); //размер структуры
	// иконка, которая будет отображаться в области уведомлений
	pNID->hIcon = hIcon;
```

```
	// дескриптор окна, которое будет получать уведомляющие сообщения,
	// ассоциированные с иконкой в области уведомлений
	pNID->hWnd = hwnd;
	lstrcpy(pNID->szTip, TEXT("Будильник")); // Подсказка
	pNID->uCallbackMessage = WM_ICON; // Пользовательское сообщение
	pNID->uFlags = NIF_TIP | NIF_ICON | NIF_MESSAGE | NIF_INFO;
	// NIF_ICON - поле hIcon содержит корректное значение (позволяет создать
	// иконку в области уведомления).
	// NIF_MESSAGE - поле uCallbackMessage содержит корректное значение
	// (позволяет получать сообщения от иконки в трэе).
	// NIF_TIP - поле szTip содержит корректное значение (позволяет создать
	// всплывающую подсказку для иконки в области уведомления).
	// NIF_INFO - поле szInfo содержит корректное значение (позволяет
	// создать Balloon подсказку для иконки в области уведомления).
	lstrcpy(pNID->szInfo,
	TEXT("Приложение демонстрирует работу таймера синхронизации"));
	lstrcpy(pNID->szInfoTitle, TEXT("Будильник!"));
	pNID->uID = ID_TRAYICON; // предопределённый идентификатор иконки
	return TRUE;
}


DWORD WINAPI Thread(LPVOID lp)
{

	CWaitableTimerDlg* p = (CWaitableTimerDlg*)lp;
	// создаем таймер синхронизации
	HANDLE hTimer = CreateWaitableTimer(NULL, TRUE, NULL);
	TCHAR buf[10];
	int hours, minutes, seconds;
	GetWindowText(p->hEdit1, buf, 10);
	hours = _tstoi(buf);
	GetWindowText(p->hEdit2, buf, 10);
	minutes = _tstoi(buf);
	GetWindowText(p->hEdit3, buf, 10);
	seconds = _tstoi(buf);
	SYSTEMTIME st;
	GetLocalTime(&st); // получим текущее локальное время
	if(st.wHour > hours || st.wHour == hours && st.wMinute > minutes ||
	st.wHour == hours && st.wMinute == minutes && st.wSecond > seconds)
	{
		CloseHandle(hTimer);
		EnableWindow(p->hButton, TRUE);
		EnableWindow(p->hEdit1, TRUE);
		EnableWindow(p->hEdit2, TRUE);
		EnableWindow(p->hEdit3, TRUE);
		return 0;
	}
	st.wHour = hours;
	st.wMinute = minutes;
	st.wSecond = seconds;
	FILETIME ft;
	// преобразуем структуру SYSTEMTIME в FILETIME
	SystemTimeToFileTime(&st, &ft);
	// преобразуем местное время в UTC-время
	LocalFileTimeToFileTime(&ft, &ft);
	// устанавливаем таймер синхронизации
	SetWaitableTimer(hTimer, (LARGE_INTEGER *)&ft, 0, NULL, NULL, FALSE);
```

```
	// ожидаем переход таймера в сигнальное состояние
	if(WaitForSingleObject(hTimer, INFINITE) == WAIT_OBJECT_0)
	{
		Shell_NotifyIcon(NIM_DELETE, p->pNID); // Удаляем иконку из трэя
		ShowWindow(p->hDialog, SW_NORMAL); // Восстанавливаем окно
		// устанавливаем окно на передний план
		SetForegroundWindow(p->hDialog);
		for(int i = 0; i < 10; i++)
		{
			Beep(1000, 500);
			Sleep(1000);
		}
	}
	CancelWaitableTimer(hTimer); // отменяем таймер
	CloseHandle(hTimer); // закрываем дескриптор таймера
	EnableWindow(p->hButton, TRUE);
	EnableWindow(p->hEdit1, TRUE);
	EnableWindow(p->hEdit2, TRUE);
	EnableWindow(p->hEdit3, TRUE);
	return 0;
}


void CWaitableTimerDlg::Cls_OnCommand(HWND hwnd, int id, HWND hwndCtl,
                                      UINT codeNotify)
{
	if(id == IDC_BUTTON1)
	{
		HANDLE h;
		h = CreateThread(NULL, 0, Thread, this, 0, NULL);
		CloseHandle(h);
		EnableWindow(hButton, FALSE);
		EnableWindow(hEdit1, FALSE);
		EnableWindow(hEdit2, FALSE);
		EnableWindow(hEdit3, FALSE);
		ShowWindow(hwnd, SW_HIDE); // Прячем окно
		Shell_NotifyIcon(NIM_ADD, pNID); // Добавляем иконку в трэй
	}
}

void CWaitableTimerDlg::Cls_OnSize(HWND hwnd, UINT state, int cx, int cy)
{

	if(state == SIZE_MINIMIZED)
	{
		ShowWindow(hwnd, SW_HIDE); // Прячем окно
		Shell_NotifyIcon(NIM_ADD, pNID); // Добавляем иконку в трэй
	}
}

// обработчик пользовательского сообщения
void CWaitableTimerDlg::OnTrayIcon(WPARAM wp, LPARAM lp)
{
	// WPARAM - идентификатор иконки
	// LPARAM - сообщение от мыши или клавиатурное сообщение
	if(lp == WM_LBUTTONDBLCLK)
	{
		Shell_NotifyIcon(NIM_DELETE, pNID); // Удаляем иконку из трэя
```

```
            ShowWindow(hDialog, SW_NORMAL); // Восстанавливаем окно
            // устанавливаем окно на передний план
            SetForegroundWindow(hDialog);
      }
}

BOOL CALLBACK CWaitableTimerDlg::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);
            HANDLE_MSG(hwnd, WM_SIZE, ptr->Cls_OnSize);
      }
      // пользовательское сообщение
      if(message == WM_ICON)
      {
            ptr->OnTrayIcon(wParam, lParam);
            return TRUE;
      }
      return FALSE;
}


                              // WaitableTimerDlg.cpp

#include "WaitableTimerDlg.h"

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

Акцентировать внимание слушателей на том, что данное приложение обладает возможностью размещения иконки в **области уведомлений (System Tray)**.

Рассмотреть со слушателями описание структуры **NOTIFYICONDATA**, объекты которой используются для создания иконок в области уведомлений, а также для получения поступающих оттуда сообщений.

```
typedef struct _NOTIFYICONDATA{
   DWORD cbSize; // размер структуры в байтах
   HWND hWnd; // дескриптор окна, которое будет получать уведомляющие
   // сообщения, ассоциированные с иконкой в области уведомления
   UINT uID; // определенный приложением идентификатор иконки
   UINT uFlags; // массив флагов, которые указывают, какие из членов
   // структуры  задействованы, т.е. содержат корректные значения
   UINT uCallbackMessage; //определенный приложением идентификатор сообщения.
   // Система использует этот идентификатор для посылки уведомляющих
   // сообщений окну, дескриптор которого хранится в поле hWnd. Эти сообщения
   // посылаются, когда происходит "мышиное" сообщение в прямоугольнике, где
   // расположена иконка, или иконка выбирается или активизируется с помощью
   // клавиатуры. Параметр сообщения wParam содержит при этом идентификатор
   // иконки в трэе, где произошло событие, а параметр сообщения lParam -
   // "мышиное" или клавиатурное сообщение, ассоциированное с событием.
   // Пример события: щелчок мышки по иконке в области уведомления.
   HICON hIcon; // дескриптор иконки
   TCHAR szTip[64]; // указатель на завершающуюся нулем строку с текстом
   // стандартной подсказки. Максимальный размер подсказки 64 символа
} NOTIFYICONDATA, *PNOTIFYICONDATA;
```

Для добавления, удаления и изменения иконки в области уведомлений служит функция API **Shell_NotifyIcon**.

```
BOOL Shell_NotifyIcon(
   DWORD dwMessage, // выполняемое действие
      //(NIM_ADD - добавить иконку, NIM_DELETE - удалить иконку, NIM_MODIFY -
      // модифицировать  иконку)
   PNOTIFYICONDATA lpdata // указатель на структуру NOTIFYICONDATA
 );
```

4. Подведение итогов

Подвести общие итоги занятия. Напомнить слушателям, какие существуют два типа событий и в чём их отличие. Отметить, в чём состоит

сходство событий и таймеров синхронизации, а также в чем их различие. Акцентировать внимание слушателей на наиболее тонких моментах изложенной темы.

5. Домашнее задание

Разработать приложение «Планировщик задач». На форме диалога должен быть список задач и время их выполнения. И то и другое может добавляться, удаляться и модифицироваться. Когда подойдет заданное время, должна быть запущена ассоциированная с этим временем задача (для запуска задачи рекомендуется использовать функцию API ShellExecute). После выполнения задачи она должна добавляться в список выполненных задач с указанием её статуса (успешное или неудачное выполнение задачи).

Кроме того, данное приложение должно обладать возможностью размещения иконки в области уведомлений.