》》》》》》》》》》》》》》》》》》多线程问题

一、问题的提出

编写一个耗时的单线程程序：
新建一个基于对话框的应用程序 SingleThread，在主对话框 IDD_SINGLETHREAD_DIALOG 添加一个按钮，ID 为 IDC_SLEEP_SIX_SECOND，标题为“延时 6 秒”，添加按钮的响应函数，代码如下：

```
void CSingleThreadDlg::OnSleepSixSecond()
{
    Sleep(6000); //延时 6 秒
}
```

编译并运行应用程序，单击“延时 6 秒”按钮，你就会发现在这 6 秒期间程序就象“死机”一样，不在响应其它消息。为了更好地处理这种耗时的操作，我们有必要学习——多线程编程。

二、多线程概述

进程和线程都是操作系统的概念。进程是应用程序的执行实例，每个进程是由私有的虚拟地址空间、代码、数据和其它各种系统资源组成，进程在运行过程中创建的资源随着进程的终止而被销毁，所使用的系统资源在进程终止时被释放或关闭。

线程是进程内部的一个执行单元。系统创建好进程后，实际上就启动执行了该进程的主执行线程，主执行线程以函数地址形式，比如说 main 或 WinMain 函数，将程序的启动点提供给 Windows 系统。主执行线程终止了，进程也就随之终止。

每一个进程至少有一个主执行线程，它无需由用户去主动创建，是由系统自动创建的。用户根据需要在应用程序中创建其它线程，多个线程并发地运行于同一个进程中。一个进程中的所有线程都在该进程的虚拟地址空间中，共同使用这些虚拟地址空间、全局变量和系统资源，所以线程间的通讯非常方便，多线程技术的应用也较为广泛。

多线程可以实现并行处理，避免了某项任务长时间占用 CPU 时间。要说明的一点是，目前大多数的计算机都是单处理器（CPU）的，为了运行所有这些线程，操作系统为每个独立线程安排一些 CPU 时间，操作系统以轮换方式向线程提供时间片，这就给人一种假象，好象这些线程都在同时运行。由此可见，如果两个非常活跃的线程为了抢夺对 CPU 的控制权，在线程切换时会消耗很多的 CPU 资源，反而会降低系统的性能。这一点在多线程编程时应该注意。

Win32 SDK 函数支持进行多线程的程序设计，并提供了操作系统原理中的各种同步、互斥和临界区等操作。Visual C++ 6.0 中，使用 MFC 类库也实现了多线程的程序设计，使得

多线程编程更加方便。

三、Win32 API对多线程编程的支持

Win32 提供了一系列的 API 函数来完成线程的创建、挂起、恢复、终结以及通信等工作。下面将选取其中的一些重要函数进行说明。

1、HANDLE CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes,
DWORD dwStackSize,
LPTHREAD_START_ROUTINE lpStartAddress,
LPVOID lpParameter,
DWORD dwCreationFlags,
LPDWORD lpThreadId);

该函数在其调用进程的进程空间里创建一个新的线程，并返回已建线程的句柄，其中各参数说明如下：

lpThreadAttributes：指向一个 SECURITY_ATTRIBUTES 结构的指针，该结构决定了线程的安全属性，一般置为 NULL；
dwStackSize：指定了线程的堆栈深度，一般都设置为 0；
lpStartAddress：表示新线程开始执行时代码所在函数的地址，即线程的起始地址。一般情况为(LPTHREAD_START_ROUTINE)ThreadFunc，ThreadFunc 是线程函数名；
lpParameter：指定了线程执行时传送给线程的 32 位参数，即线程函数的参数；
dwCreationFlags：控制线程创建的附加标志，可以取两种值。如果该参数为 0，线程在被创建后就会立即开始执行；如果该参数为 CREATE_SUSPENDED,则系统产生线程后，该线程处于挂起状态，并不马上执行，直至函数 ResumeThread 被调用；
lpThreadId：该参数返回所创建线程的 ID；

如果创建成功则返回线程的句柄，否则返回 NULL。

2、DWORD SuspendThread(HANDLE hThread);

该函数用于挂起指定的线程，如果函数执行成功，则线程的执行被终止。

3、DWORD ResumeThread(HANDLE hThread);

该函数用于结束线程的挂起状态，执行线程。

4、VOID ExitThread(DWORD dwExitCode);

该函数用于线程终结自身的执行，主要在线程的执行函数中被调用。其中参数 dwExitCode

用来设置线程的退出码。

5、BOOL TerminateThread(HANDLE hThread,DWORD dwExitCode);
一般情况下，线程运行结束之后，线程函数正常返回，但是应用程序可以调用 TerminateThread 强行终止某一线程的执行。各参数含义如下：

hThread：将被终结的线程的句柄；
dwExitCode：用于指定线程的退出码。

使用 TerminateThread()终止某个线程的执行是不安全的，可能会引起系统不稳定；虽然该函数立即终止线程的执行，但并不释放线程所占用的资源。因此，一般不建议使用该函数。

6、BOOL PostThreadMessage(DWORD idThread,
UINT Msg,
WPARAM wParam,
LPARAM lParam);

该函数将一条消息放入到指定线程的消息队列中，并且不等到消息被该线程处理时便返回。
idThread：将接收消息的线程的 ID；
Msg：指定用来发送的消息；
wParam：同消息有关的字参数；
lParam：同消息有关的长参数；
调用该函数时，如果即将接收消息的线程没有创建消息循环，则该函数执行失败。

四、Win32 API多线程编程例程

例程 1 MultiThread1

建立一个基于对话框的工程 MultiThread1，在对话框 IDD_MULTITHREAD1_DIALOG 中加入两个按钮和一个编辑框，两个按钮的 ID 分别是 IDC_START，IDC_STOP ，标题分别为“启动”，“停止”，IDC_STOP 的属性选中 Disabled；编辑框的 ID 为 IDC_TIME ，属性选中 Read-only;在 MultiThread1Dlg.h 文件中添加线程函数声明：

void ThreadFunc();

注意，线程函数的声明应在类 CMultiThread1Dlg 的外部。 在类 CMultiThread1Dlg 内部添加 protected 型变量：

HANDLE hThread;
DWORD ThreadID;

分别代表线程的句柄和 ID。
在 MultiThread1Dlg.cpp 文件中添加全局变量 m_bRun ：

volatile BOOL m_bRun;

m_bRun 代表线程是否正在运行。

你要留意到全局变量 m_bRun 是使用 volatile 修饰符的，volatile 修饰符的作用是告诉编译器无需对该变量作任何的优化，即无需将它放到一个寄存器中，并且该值可被外部改变。对于多线程引用的全局变量来说，volatile 是一个非常重要的修饰符。

编写线程函数：

```
void ThreadFunc()
{
    CTime time;
    CString strTime;
    m_bRun=TRUE;
    while(m_bRun)
    {
        time=CTime::GetCurrentTime();
        strTime=time.Format("%H:%M:%S");
        ::SetDlgItemText(AfxGetMainWnd()->m_hWnd,IDC_TIME,strTime);
        Sleep(1000);
    }
}
```

该线程函数没有参数，也不返回函数值。只要 m_bRun 为 TRUE，线程一直运行。

双击 IDC_START 按钮，完成该按钮的消息函数：

```
void CMultiThread1Dlg::OnStart()
{
    // TODO: Add your control notification handler code here
    hThread=CreateThread(NULL,
        0,
        (LPTHREAD_START_ROUTINE)ThreadFunc,
        NULL,
        0,
        &ThreadID);
    GetDlgItem(IDC_START)->EnableWindow(FALSE);
    GetDlgItem(IDC_STOP)->EnableWindow(TRUE);

}
```

双击 IDC_STOP 按钮，完成该按钮的消息函数：

```
void CMultiThread1Dlg::OnStop()
{
    // TODO: Add your control notification handler code here
    m_bRun=FALSE;
    GetDlgItem(IDC_START)->EnableWindow(TRUE);
    GetDlgItem(IDC_STOP)->EnableWindow(FALSE);
}
```

编译并运行该例程，体会使用 Win32 API 编写的多线程。

例程 2 MultiThread2

该线程演示了如何传送一个一个整型的参数到一个线程中，以及如何等待一个线程完成处理。

建立一个基于对话框的工程 MultiThread2，在对话框 IDD_MULTITHREAD2_DIALOG 中加入一个编辑框和一个按钮，ID 分别是 IDC_COUNT，IDC_START ，按钮控件的标题为“开始”； 在 MultiThread2Dlg.h 文件中添加线程函数声明：

```
void ThreadFunc(int integer);
```

注意，线程函数的声明应在类 CMultiThread2Dlg 的外部。

在类 CMultiThread2Dlg 内部添加 protected 型变量:

```
HANDLE hThread;
DWORD ThreadID;
```

分别代表线程的句柄和 ID。

打开 ClassWizard，为编辑框 IDC_COUNT 添加 int 型变量 m_nCount。在 MultiThread2Dlg.cpp 文件中添加：

```
void ThreadFunc(int integer)
{
    int i;
    for(i=0;i<integer;i++)
    {
        Beep(200,50);
        Sleep(1000);
    }
}
```

双击 IDC_START 按钮，完成该按钮的消息函数：

```
void CMultiThread2Dlg::OnStart()
{
    UpdateData(TRUE);
    int integer=m_nCount;
    hThread=CreateThread(NULL,
        0,
        (LPTHREAD_START_ROUTINE)ThreadFunc,
        (VOID*)integer,
        0,
        &ThreadID);
    GetDlgItem(IDC_START)->EnableWindow(FALSE);
```

```
    WaitForSingleObject(hThread,INFINITE);
    GetDlgItem(IDC_START)->EnableWindow(TRUE);
}
```

顺便说一下 WaitForSingleObject 函数，其函数原型为：
DWORD WaitForSingleObject(HANDLE hHandle,DWORD dwMilliseconds);

hHandle 为要监视的对象（一般为同步对象，也可以是线程）的句柄；
dwMilliseconds 为 hHandle 对象所设置的超时值，单位为毫秒；

当在某一线程中调用该函数时，线程暂时挂起，系统监视 hHandle 所指向的对象的状态。如果在挂起的 dwMilliseconds 毫秒内，线程所等待的对象变为有信号状态，则该函数立即返回；如果超时时间已经到达 dwMilliseconds 毫秒，但 hHandle 所指向的对象还没有变成有信号状态，函数照样返回。参数 dwMilliseconds 有两个具有特殊意义的值：0 和 INFINITE。若为 0，则该函数立即返回；若为 INFINITE，则线程一直被挂起，直到 hHandle 所指向的对象变为有信号状态时为止。

本例程调用该函数的作用是按下 IDC_START 按钮后，一直等到线程返回，再恢复 IDC_START 按钮正常状态。编译运行该例程并细心体会。

例程 3 MultiThread3 将演示如何传送一个指向结构体的指针参数。

建立一个基于对话框的工程 MultiThread3，在对话框 IDD_MULTITHREAD3_DIALOG 中加入一个编辑框 IDC_MILLISECOND，一个按钮 IDC_START，标题为“开始”，一个进度条 IDC_PROGRESS1；
打开 ClassWizard，为编辑框 IDC_MILLISECOND 添加 int 型变量 m_nMilliSecond，为进度条 IDC_PROGRESS1 添加 CProgressCtrl 型变量 m_ctrlProgress；
在 MultiThread3Dlg.h 文件中添加一个结构的定义：

```
struct threadInfo
{
    UINT nMilliSecond;
    CProgressCtrl* pctrlProgress;
};
```

线程函数的声明：

```
UINT ThreadFunc(LPVOID lpParam);
```

注意，二者应在类 CMultiThread3Dlg 的外部。
在类 CMultiThread3Dlg 内部添加 protected 型变量:

```
HANDLE hThread;
DWORD ThreadID;
```

分别代表线程的句柄和 ID。
在 MultiThread3Dlg.cpp 文件中进行如下操作：
定义公共变量

```
struct threadInfo Info;
```

双击按钮 IDC_START，添加相应消息处理函数：

```
void CMultiThread3Dlg::OnStart()
```

```
{
    // TODO: Add your control notification handler code here

    UpdateData(TRUE);
    Info.nMilliSecond=m_nMilliSecond;
    Info.pctrlProgress=&m_ctrlProgress;

    hThread=CreateThread(NULL,
        0,
        (LPTHREAD_START_ROUTINE)ThreadFunc,
        &Info,
        0,
        &ThreadID);
/*
    GetDlgItem(IDC_START)->EnableWindow(FALSE);
    WaitForSingleObject(hThread,INFINITE);
    GetDlgItem(IDC_START)->EnableWindow(TRUE);
*/
}
```

在函数 BOOL CMultiThread3Dlg::OnInitDialog()中添加语句： {

```
    ……

    // TODO: Add extra initialization here
    m_ctrlProgress.SetRange(0,99);
    m_nMilliSecond=10;
    UpdateData(FALSE);
    return TRUE;  // return TRUE  unless you set the focus to a control
}
```

添加线程处理函数：

```
UINT ThreadFunc(LPVOID lpParam) {
    threadInfo* pInfo=(threadInfo*)lpParam;
    for(int i=0;i<100;i++)
    {
        int nTemp=pInfo->nMilliSecond;

        pInfo->pctrlProgress->SetPos(i);

        Sleep(nTemp);
    }
    return 0;
}
```

顺便补充一点，如果你在 void CMultiThread3Dlg::OnStart() 函数中添加/* */语句，编译运行你就会发现进度条不进行刷新，主线程也停止了反应。什么原因呢？这是因为 WaitForSingleObject 函数等待子线程（ThreadFunc）结束时，导致了线程死锁。因为 WaitForSingleObject 函数会将主线程挂起（任何消息都得不到处理），而子线程 ThreadFunc 正在设置进度条，一直在等待主线程将刷新消息处理完毕返回才会检测通知事件。这样两个线程都在互相等待，死锁发生了，编程时应注意避免。

五、MFC对多线程编程的支持

MFC 中有两类线程，分别称之为工作者线程和用户界面线程。二者的主要区别在于工作者线程没有消息循环，而用户界面线程有自己的消息队列和消息循环。

工作者线程没有消息机制，通常用来执行后台计算和维护任务，如冗长的计算过程，打印机的后台打印等。用户界面线程一般用于处理独立于其他线程执行之外的用户输入，响应用户及系统所产生的事件和消息等。但对于 Win32 的 API 编程而言，这两种线程是没有区别的，它们都只需线程的启动地址即可启动线程来执行任务。

在 MFC 中，一般用全局函数 AfxBeginThread()来创建并初始化一个线程的运行，该函数有两种重载形式，分别用于创建工作者线程和用户界面线程。两种重载函数原型和参数分别说明如下：

(1) CWinThread* AfxBeginThread(AFX_THREADPROC pfnThreadProc,
LPVOID pParam,
nPriority=THREAD_PRIORITY_NORMAL,
UINT nStackSize=0,
DWORD dwCreateFlags=0,
LPSECURITY_ATTRIBUTES lpSecurityAttrs=NULL);

PfnThreadProc:指向工作者线程的执行函数的指针，线程函数原型必须声明如下：
UINT ExecutingFunction(LPVOID pParam);
请注意，ExecutingFunction()应返回一个 UINT 类型的值，用以指明该函数结束的原因。一般情况下，返回 0 表明执行成功。
pParam：传递给线程函数的一个 32 位参数，执行函数将用某种方式解释该值。它可以是数值，或是指向一个结构的指针，甚至可以被忽略；
nPriority：线程的优先级。如果为 0，则线程与其父线程具有相同的优先级；
nStackSize:：线程为自己分配堆栈的大小，其单位为字节。如果 nStackSize 被设为 0，则线程的堆栈被设置成与父线程堆栈相同大小；
dwCreateFlags：如果为 0，则线程在创建后立刻开始执行。如果为 CREATE_SUSPEND，则线程在创建后立刻被挂起；
lpSecurityAttrs：线程的安全属性指针，一般为 NULL；

(2) CWinThread* AfxBeginThread(CRuntimeClass* pThreadClass,
int nPriority=THREAD_PRIORITY_NORMAL,
UINT nStackSize=0,

DWORD dwCreateFlags=0,
LPSECURITY_ATTRIBUTES lpSecurityAttrs=NULL);

pThreadClass 是指向 CWinThread 的一个导出类的运行时类对象的指针，该导出类定义了被创建的用户界面线程的启动、退出等；其它参数的意义同形式 1。使用函数的这个原型生成的线程也有消息机制，在以后的例子中我们将发现同主线程的机制几乎一样。
下面我们对 CWinThread 类的数据成员及常用函数进行简要说明。
m_hThread：当前线程的句柄；
m_nThreadID:当前线程的 ID；
m_pMainWnd：指向应用程序主窗口的指针
BOOL CWinThread::CreateThread(DWORD dwCreateFlags=0,
UINT nStackSize=0,
LPSECURITY_ATTRIBUTES lpSecurityAttrs=NULL);
该函数中的 dwCreateFlags、nStackSize、lpSecurityAttrs 参数和 API 函数 CreateThread 中的对应参数有相同含义，该函数执行成功，返回非 0 值，否则返回 0。
一般情况下，调用 AfxBeginThread()来一次性地创建并启动一个线程，但是也可以通过两步法来创建线程：首先创建 CWinThread 类的一个对象，然后调用该对象的成员函数 CreateThread()来启动该线程。
virtual BOOL CWinThread::InitInstance();
重载该函数以控制用户界面线程实例的初始化。初始化成功则返回非 0 值，否则返回 0。用户界面线程经常重载该函数，工作者线程一般不使用 InitInstance()。
virtual int CWinThread::ExitInstance();
在线程终结前重载该函数进行一些必要的清理工作。该函数返回线程的退出码，0 表示执行成功，非 0 值用来标识各种错误。同 InitInstance()成员函数一样，该函数也只适用于用户界面线程。

例程 6 MultiThread6

建立一个基于对话框的工程 MultiThread6，在对话框 IDD_MULTITHREAD6_DIALOG 中加入一个按钮 IDC_UI_THREAD，标题为“用户界面线程” 右击工程并选中“New Class…”为工程添加基类为 CWinThread 派生线程类 CUIThread。 给工程添加新对话框 IDD_UITHREADDLG，标题为“线程对话框”。
为对话框 IDD_UITHREADDLG 创建一个基于 CDialog 的类 CUIThreadDlg。使用 ClassWizard 为 CUIThreadDlg 类添加 WM_LBUTTONDOWN 消息的处理函数
OnLButtonDown，如下：

```
void CUIThreadDlg::OnLButtonDown(UINT nFlags, CPoint point)
{
    AfxMessageBox("You Clicked The Left Button!");
    CDialog::OnLButtonDown(nFlags, point);
}
```

在 UIThread.h 中添加 **#include "UIThreadDlg.h"**
并在 CUIThread 类中添加 protected 变量 **CUIThread m_dlg；**

```
class CUIThread : public CWinThread
{
```

```
	DECLARE_DYNCREATE(CUIThread)
protected:
	CUIThread();			// protected constructor used by dynamic creation

// Attributes
public:

// Operations
public:

// Overrides
	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(CUIThread)
	public:
	virtual BOOL InitInstance();
	virtual int ExitInstance();
	//}}AFX_VIRTUAL

// Implementation
protected:
	CUIThreadDlg m_dlg;
	virtual ~CUIThread();

	// Generated message map functions
	//{{AFX_MSG(CUIThread)
		// NOTE - the ClassWizard will add and remove member functions here.
	//}}AFX_MSG

	DECLARE_MESSAGE_MAP()
};
```

分别重载 InitInstance()函数和 ExitInstance()函数：

```
BOOL CUIThread::InitInstance()
{
	m_dlg.Create(IDD_UITHREADDLG);
	m_dlg.ShowWindow(SW_SHOW);
	m_pMainWnd=&m_dlg;
	return TRUE;
}

int CUIThread::ExitInstance()
{
	m_dlg.DestroyWindow();
	return CWinThread::ExitInstance();
```

}

双击按钮 IDC_UI_THREAD，添加消息响应函数：

```
void CMultiThread6Dlg::OnUiThread()
{
    CWinThread *pThread=AfxBeginThread(RUNTIME_CLASS(CUIThread));
}
```

并在 MultiThread6Dlg.cpp 的开头添加：

```
#include "UIThread.h"
```

好了，编译并运行程序吧。每单击一次“用户界面线程”按钮，都会弹出一个线程对话框，在任何一个线程对话框内按下鼠标左键，都会弹出一个消息框。

六、线程间通讯

一般而言,应用程序中的一个次要线程总是为主线程执行特定的任务,这样,主线程和次要线程间必定有一个信息传递的渠道,也就是主线程和次要线程间要进行通信。这种线程间的通信不但是难以避免的，而且在多线程编程中也是复杂和频繁的，下面将进行说明。

1）使用全局变量进行通信

由于属于同一个进程的各个线程共享操作系统分配该进程的资源，故解决线程间通信最简单的一种方法是使用全局变量。对于标准类型的全局变量，我们建议使用 volatile 修饰符，它告诉编译器无需对该变量作任何的优化，即无需将它放到一个寄存器中，并且该值可被外部改变。如果线程间所需传递的信息较复杂，我们可以定义一个结构，通过传递指向该结构的指针进行传递信息。

2）使用自定义消息

我们可以在一个线程的执行函数中向另一个线程发送自定义的消息来达到通信的目的。一个线程向另外一个线程发送消息是通过操作系统实现的。利用 Windows 操作系统的消息驱动机制，当一个线程发出一条消息时，操作系统首先接收到该消息，然后把该消息转发给目标线程，接收消息的线程必须已经建立了消息循环。

例程 7 MultiThread7

该例程演示了如何使用自定义消息进行线程间通信。首先，主线程向 CCalculateThread 线程发送消息 WM_CALCULATE，CCalculateThread 线程收到消息后进行计算，再向主线程发送 WM_DISPLAY 消息，主线程收到该消息后显示计算结果。

建立一个基于对话框的工程 MultiThread7，在对话框 IDD_MULTITHREAD7_DIALOG 中加入三个单选按钮 IDC_RADIO1，IDC_RADIO2，IDC_RADIO3，标题分别为 1+2+3+4+.....+10，1+2+3+4+.....+50，1+2+3+4+.....+100。加入按钮 IDC_SUM，标题为“求和”。加入标签框 IDC_STATUS，属性选中“边框”；
在 MultiThread7Dlg.h 中定义如下变量：

```
protected:
    int nAddend;
```

代表加数的大小。

分别双击三个单选按钮，添加消息响应函数：

```
void CMultiThread7Dlg::OnRadio1()
{
    nAddend=10;
}

void CMultiThread7Dlg::OnRadio2()
{
    nAddend=50;

}

void CMultiThread7Dlg::OnRadio3()
{
    nAddend=100;

}
```

并在 OnInitDialog 函数中完成相应的初始化工作：

```
BOOL CMultiThread7Dlg::OnInitDialog()
{
……
    ((CButton*)GetDlgItem(IDC_RADIO1))->SetCheck(TRUE);
    nAddend=10;
……
```

在 MultiThread7Dlg.h 中添加：

```
#include "CalculateThread.h"
#define WM_DISPLAY WM_USER+2
class CMultiThread7Dlg : public CDialog
{
// Construction
public:
    CMultiThread7Dlg(CWnd* pParent = NULL);    // standard constructor
```

```
    CCalculateThread* m_pCalculateThread;
……
protected:
    int nAddend;
    LRESULT OnDisplay(WPARAM wParam,LPARAM lParam);
……
```

在 MultiThread7Dlg.cpp 中添加：

```
BEGIN_MESSAGE_MAP(CMultiThread7Dlg, CDialog)
……
    ON_MESSAGE(WM_DISPLAY,OnDisplay)
END_MESSAGE_MAP()

LRESULT CMultiThread7Dlg::OnDisplay(WPARAM wParam,LPARAM lParam)
{
    int nTemp=(int)wParam;
    SetDlgItemInt(IDC_STATUS,nTemp,FALSE);
    return 0;

}
```

以上代码使得主线程类 CMultiThread7Dlg 可以处理 WM_DISPLAY 消息，即在 IDC_STATUS 标签框中显示计算结果。

双击按钮 IDC_SUM，添加消息响应函数：

```
void CMultiThread7Dlg::OnSum()
{
    m_pCalculateThread=
        (CCalculateThread*)AfxBeginThread(RUNTIME_CLASS(CCalculateThread));

    Sleep(500);

    m_pCalculateThread->PostThreadMessage(WM_CALCULATE,nAddend,NULL);
}
```

OnSum()函数的作用是建立 CalculateThread 线程，延时给该线程发送 WM_CALCULATE 消息。

右击工程并选中“New Class…”为工程添加基类为 CWinThread 派生线程类 CCalculateThread。

在文件 CalculateThread.h 中添加

```
#define WM_CALCULATE WM_USER+1
class CCalculateThread : public CWinThread
{
……
protected:
    afx_msg LONG OnCalculate(UINT wParam,LONG lParam);
```

……

在文件 CalculateThread.cpp 中添加

```
LONG CCalculateThread::OnCalculate(UINT wParam,LONG lParam)
{
    int nTmpt=0;
    for(int i=0;i<=(int)wParam;i++)
    {
        nTmpt=nTmpt+i;
    }

    Sleep(500);
    ::PostMessage((HWND)(GetMainWnd()->GetSafeHwnd()),WM_DISPLAY,nTmpt,NU
LL);

    return 0;
}
BEGIN_MESSAGE_MAP(CCalculateThread, CWinThread)
    //{{AFX_MSG_MAP(CCalculateThread)
        // NOTE - the ClassWizard will add and remove mapping macros here.
    //}}AFX_MSG_MAP
    ON_THREAD_MESSAGE(WM_CALCULATE,OnCalculate)
//和主线程对比，注意它们的区别
END_MESSAGE_MAP()
```

在 CalculateThread.cpp 文件的开头添加一条：

```
#include "MultiThread7Dlg.h"
```

以上代码为 CCalculateThread 类添加了 WM_CALCULATE 消息，消息的响应函数是 OnCalculate，其功能是根据参数 wParam 的值，进行累加，累加结果在临时变量 nTmpt 中，延时 0.5 秒，向主线程发送 WM_DISPLAY 消息进行显示，nTmpt 作为参数传递。
编译并运行该例程,体会如何在线程间传递消息。

七、线程的同步

虽然多线程能给我们带来好处，但是也有不少问题需要解决。例如，对于像磁盘驱动器这样独占性系统资源，由于线程可以执行进程的任何代码段，且线程的运行是由系统调度自动完成的，具有一定的不确定性，因此就有可能出现两个线程同时对磁盘驱动器进行操作，从而出现操作错误；又例如，对于银行系统的计算机来说，可能使用一个线程来更新其用户

数据库，而用另外一个线程来读取数据库以响应储户的需要，极有可能读数据库的线程读取的是未完全更新的数据库，因为可能在读的时候只有一部分数据被更新过。

使隶属于同一进程的各线程协调一致地工作称为线程的同步。MFC 提供了多种同步对象，下面我们只介绍最常用的四种：

临界区（CCriticalSection）
事件（CEvent）
互斥量（CMutex）
信号量（CSemaphore）

通过这些类，我们可以比较容易地做到线程同步。

1）使用 CCriticalSection 类

当多个线程访问一个独占性共享资源时,可以使用“临界区”对象。任一时刻只有一个线程可以拥有临界区对象，拥有临界区的线程可以访问被保护起来的资源或代码段，其他希望进入临界区的线程将被挂起等待，直到拥有临界区的线程放弃临界区时为止，这样就保证了不会在同一时刻出现多个线程访问共享资源。
CCriticalSection 类的用法非常简单，步骤如下：
定义 CCriticalSection 类的一个全局对象（以使各个线程均能访问），如
CCriticalSection critical_section;
在访问需要保护的资源或代码之前，调用 CCriticalSection 类的成员 Lock（）获得临界区对象：
critical_section.Lock();

在线程中调用该函数来使线程获得它所请求的临界区。如果此时没有其它线程占有临界区对象，则调用 Lock()的线程获得临界区；否则，线程将被挂起，并放入到一个系统队列中等待，直到当前拥有临界区的线程释放了临界区时为止。
访问临界区完毕后，使用 CCriticalSection 的成员函数 Unlock()来释放临界区：
critical_section.Unlock();

再通俗一点讲，就是线程 A 执行到 critical_section.Lock();语句时，如果其它线程(B)正在执行 critical_section.Lock();语句后且 critical_section. Unlock();语句前的语句时，线程 A 就会等待，直到线程 B 执行完 critical_section. Unlock();语句，线程 A 才会继续执行。
下面再通过一个实例进行演示说明。

例程 8 MultiThread8

建立一个基于对话框的工程 MultiThread8，在对话框 IDD_MULTITHREAD8_DIALOG 中加入两个按钮和两个编辑框控件，两个按钮的 ID 分别为

IDC_WRITEW 和 IDC_WRITED，标题分别为“写‘W’”和“写‘D’”；两个编辑框的 ID

分别为 IDC_W 和 IDC_D，属性都选中 Read-only；
在 MultiThread8Dlg.h 文件中声明两个线程函数：

```
UINT WriteW(LPVOID pParam);
UINT WriteD(LPVOID pParam);
```

使用 ClassWizard 分别给 IDC_W 和 IDC_D 添加 CEdit 类变量 m_ctrlW 和 m_ctrlD；
在 MultiThread8Dlg.cpp 文件中添加如下内容：
为了文件中能够正确使用同步类，在文件开头添加：

```
#include "afxmt.h"
```

定义临界区和一个字符数组，为了能够在不同线程间使用，定义为全局变量：

```
CCriticalSection critical_section;
char g_Array[10];
```

添加线程函数：

```
UINT WriteW(LPVOID pParam)
{
    CEdit *pEdit=(CEdit*)pParam;
    pEdit->SetWindowText("");
    critical_section.Lock();
    //锁定临界区，其它线程遇到 critical_section.Lock();语句时要等待
    //直至执行 critical_section.Unlock();语句
    for(int i=0;i<10;i++)
    {
        g_Array[i]=''W'';
        pEdit->SetWindowText(g_Array);
        Sleep(1000);
    }
    critical_section.Unlock();
    return 0;

}

UINT WriteD(LPVOID pParam)
{
    CEdit *pEdit=(CEdit*)pParam;
    pEdit->SetWindowText("");
    critical_section.Lock();
    //锁定临界区，其它线程遇到 critical_section.Lock();语句时要等待
    //直至执行 critical_section.Unlock();语句
    for(int i=0;i<10;i++)
    {
        g_Array[i]=''D'';
        pEdit->SetWindowText(g_Array);
        Sleep(1000);
```

```
    }
    critical_section.Unlock();
    return 0;

}
```

分别双击按钮 IDC_WRITEW 和 IDC_WRITED，添加其响应函数：

```
void CMultiThread8Dlg::OnWritew()
{
    CWinThread *pWriteW=AfxBeginThread(WriteW,
        &m_ctrlW,
        THREAD_PRIORITY_NORMAL,
        0,
        CREATE_SUSPENDED);
    pWriteW->ResumeThread();
}

void CMultiThread8Dlg::OnWrited()
{
    CWinThread *pWriteD=AfxBeginThread(WriteD,
        &m_ctrlD,
        THREAD_PRIORITY_NORMAL,
        0,
        CREATE_SUSPENDED);
    pWriteD->ResumeThread();

}
```

由于代码较简单，不再详述。编译、运行该例程，您可以连续点击两个按钮，观察体会临界类的作用。

2）使用 CEvent 类

CEvent 类提供了对事件的支持。事件是一个允许一个线程在某种情况发生时，唤醒另外一个线程的同步对象。例如在某些网络应用程序中，一个线程（记为 A）负责监听通讯端口，另外一个线程（记为 B）负责更新用户数据。通过使用 CEvent 类，线程 A 可以通知线程 B 何时更新用户数据。每一个 CEvent 对象可以有两种状态：**有信号状态**和**无信号状态**。线程监视位于其中的 CEvent 类对象的状态，并在相应的时候采取相应的操作。

在 MFC 中，CEvent 类对象有两种类型：**人工事件**和**自动事件**。一个自动 CEvent 对象在被至少一个线程释放后会自动返回到无信号状态；而人工事件对象获得信号后，释放可利用线程，但直到调用成员函数 ReSetEvent()才将其设置为无信号状态。在创建 CEvent 类的对象时，默认创建的是自动事件。 CEvent 类的各成员函数的原型和参数说明如下：

1、
```
CEvent(BOOL bInitiallyOwn=FALSE,
        BOOL bManualReset=FALSE,
        LPCTSTR lpszName=NULL,
```

LPSECURITY_ATTRIBUTES lpsaAttribute=NULL);

bInitiallyOwn:指定事件对象初始化状态，TRUE 为有信号，FALSE 为无信号；
bManualReset：指定要创建的事件是属于人工事件还是自动事件。TRUE 为人工事件，FALSE 为自动事件；后两个参数一般设为 NULL，在此不作过多说明。

2、BOOL CEvent：：SetEvent();

将 CEvent 类对象的状态设置为**有信号状态**。如果事件是人工事件，则 CEvent 类对象保持为有信号状态，直到调用成员函数 ResetEvent()将 其重新设为无信号状态时为止。如果 CEvent 类对象为自动事件，则在 SetEvent()将事件设置为有信号状态后，CEvent 类对象由系统自动重置为无信号状态。如果该函数执行成功，则返回非零值，否则返回零。

3、BOOL CEvent：：ResetEvent();

该函数将事件的状态设置为**无信号状态**，并保持该状态直至 SetEvent()被调用时为止。由于自动事件是由系统自动重置，故自动事件不需要调用该函数。如果该函数执行成功，返回非零值，否则返回零。我们一般通过调用 WaitForSingleObject 函数来监视事件状态。前面我们已经介绍了该函数。由于语言描述的原因，CEvent 类的理解确实有些难度，但您只要通过仔细玩味下面例程，多看几遍就可理解。

例程 9 MultiThread9

建立一个基于对话框的工程 MultiThread9，在对话框 IDD_MULTITHREAD9_DIALOG 中加入一个按钮和两个编辑框控件，按钮的 ID 为 IDC_WRITEW，标题为“写‘W’”；两个编辑框的 ID 分别为 IDC_W 和 IDC_D，属性都选中 Read-only; 在 MultiThread9Dlg.h 文件中声明两个线程函数：

UINT WriteW(LPVOID pParam);
UINT WriteD(LPVOID pParam);

使用 ClassWizard 分别给 IDC_W 和 IDC_D 添加 CEdit 类变量 m_ctrlW 和 m_ctrlD；
在 MultiThread9Dlg.cpp 文件中添加如下内容：
为了文件中能够正确使用同步类，在文件开头添加

#include "afxmt.h"

定义事件对象和一个字符数组，为了能够在不同线程间使用，定义为全局变量。

CEvent eventWriteD;
char g_Array[10];

添加线程函数：

```
UINT WriteW(LPVOID pParam)
{
    CEdit *pEdit=(CEdit*)pParam;
    pEdit->SetWindowText("");
    for(int i=0;i<10;i++)
    {
```

```
            g_Array[i]=''W'';
            pEdit->SetWindowText(g_Array);
            Sleep(1000);
        }
        eventWriteD.SetEvent();
        return 0;

}

UINT WriteD(LPVOID pParam)
{
        CEdit *pEdit=(CEdit*)pParam;
        pEdit->SetWindowText("");
        WaitForSingleObject(eventWriteD.m_hObject,INFINITE);
        for(int i=0;i<10;i++)
        {
            g_Array[i]=''D'';
            pEdit->SetWindowText(g_Array);
            Sleep(1000);
        }
        return 0;

}
```

仔细分析这两个线程函数，您就会正确理解 CEvent 类。线程 WriteD 执行到 WaitForSingleObject(eventWriteD.m_hObject,INFINITE);处等待，直到事件 eventWriteD 为有信号该线程才往下执行，因为 eventWriteD 对象是自动事件，则当 WaitForSingleObject()返回时，系统自动把 eventWriteD 对象重置为无信号状态。

双击按钮 IDC_WRITEW，添加其响应函数：

```
void CMultiThread9Dlg::OnWritew()
{
        CWinThread *pWriteW=AfxBeginThread(WriteW,
            &m_ctrlW,
            THREAD_PRIORITY_NORMAL,
            0,
            CREATE_SUSPENDED);
        pWriteW->ResumeThread();
}
void CMultiThread9Dlg::OnWrited()
{
        CWinThread *pWriteD=AfxBeginThread(WriteD,
            &m_ctrlD,
            THREAD_PRIORITY_NORMAL,
            0,
```

```
        CREATE_SUSPENDED);
    pWriteD->ResumeThread();

}
```

编译并运行程序，单击“写‘W’”按钮，体会事件对象的作用。

3)使用 CMutex 类

互斥对象与临界区对象很像.互斥对象与临界区对象的不同在于:互斥对象可以在进程间使用,而临界区对象只能在同一进程的各线程间使用。当然，互斥对象也可以用于同一进程的各个线程间，但是在这种情况下，使用临界区会更节省系统资源，更有效率。

4)使用 CSemaphore 类

当需要一个计数器来限制可以使用某个线程的数目时，可以使用“信号量”对象。CSemaphore 类的对象保存了对当前访问某一指定资源的线程的计数值，该计数值是当前还可以使用该资源的线程的数目。如果这个计数达到了零，则所有对这个 CSemaphore 类对象所控制的资源的访问尝试都被放入到一个队列中等待，直到超时或计数值不为零时为止。一个线程被释放已访问了被保护的资源时，计数值减 1；一个线程完成了对被控共享资源的访问时，计数值增 1。这个被 CSemaphore 类对象所控制的资源可以同时接受访问的最大线程数在该对象的构建函数中指定。

CSemaphore 类的构造函数原型及参数说明如下：

```
CSemaphore (LONG lInitialCount=1,
            LONG lMaxCount=1,
            LPCTSTR pstrName=NULL,
            LPSECURITY_ATTRIBUTES lpsaAttributes=NULL);
```

lInitialCount:信号量对象的初始计数值，即可访问线程数目的初始值；

lMaxCount：信号量对象计数值的最大值，该参数决定了同一时刻可访问由信号量保护的资源的线程最大数目；

后两个参数在同一进程中使用一般为 NULL，不作过多讨论；

在用 CSemaphore 类的构造函数创建信号量对象时要同时指出允许的最大资源计数和当前可用资源计数。一般是将当前可用资源计数设置为最大资源计数，每增加一个线程对共享资源的访问，当前可用资源计数就会减 1，只要当前可用资源计数是大于 0 的，就可以发出信号量信号。但是当前可用计数减小到 0 时，则说明当前占用资源的线程数已经达到了所允许的最大数目，不能再允许其它线程的进入，此时的信号量信号将无法发出。线程在处理完共享资源后，应在离开的同时通过 ReleaseSemaphore()函数将当前可用资源数加 1。

下面给出一个简单实例来说明 CSemaphore 类的用法。

例程 10 MultiThread10

建立一个基于对话框的工程 MultiThread10，在对话框 IDD_MULTITHREAD10_DIALOG 中加入一个按钮和三个编辑框控件，按钮的 ID 为 IDC_START，

标题为“同时写‘A’、‘B’、‘C’”；三个编辑框的 ID 分别为 IDC_A、IDC_B 和 IDC_C，

属性都选中 Read-only；
在 MultiThread10Dlg.h 文件中声明三个线程函数：

```
UINT WriteA(LPVOID pParam);
UINT WriteB(LPVOID pParam);
UINT WriteC(LPVOID pParam);
```

使用 ClassWizard 分别给 IDC_A、IDC_B 和 IDC_C 添加 CEdit 类变量 m_ctrlA、m_ctrlB 和 m_ctrlC；
在 MultiThread10Dlg.cpp 文件中添加如下内容：
为了文件中能够正确使用同步类，在文件开头添加：

```
#include "afxmt.h"
```

定义信号量对象和一个字符数组，为了能够在不同线程间使用，定义为全局变量：

```
CSemaphore semaphoreWrite(2,2); //资源最多访问线程 2 个，当前可访问线程数 2 个
char g_Array[10];
```

添加三个线程函数：

```
UINT WriteA(LPVOID pParam)
{
    CEdit *pEdit=(CEdit*)pParam;
    pEdit->SetWindowText("");
    WaitForSingleObject(semaphoreWrite.m_hObject,INFINITE);
    CString str;
    for(int i=0;i<10;i++)
    {
        pEdit->GetWindowText(str);
        g_Array[i]=''A'';
        str=str+g_Array[i];
        pEdit->SetWindowText(str);
        Sleep(1000);
    }
    ReleaseSemaphore(semaphoreWrite.m_hObject,1,NULL);
    return 0;

}
UINT WriteB(LPVOID pParam)
{
    CEdit *pEdit=(CEdit*)pParam;
    pEdit->SetWindowText("");
    WaitForSingleObject(semaphoreWrite.m_hObject,INFINITE);
    CString str;
    for(int i=0;i<10;i++)
    {
```

```
            pEdit->GetWindowText(str);
            g_Array[i]=''B'';
            str=str+g_Array[i];
            pEdit->SetWindowText(str);
            Sleep(1000);
        }
        ReleaseSemaphore(semaphoreWrite.m_hObject,1,NULL);
        return 0;

}
UINT WriteC(LPVOID pParam)
{
        CEdit *pEdit=(CEdit*)pParam;
        pEdit->SetWindowText("");
        WaitForSingleObject(semaphoreWrite.m_hObject,INFINITE);
        for(int i=0;i<10;i++)
        {
            g_Array[i]=''C'';
            pEdit->SetWindowText(g_Array);
            Sleep(1000);
        }
        ReleaseSemaphore(semaphoreWrite.m_hObject,1,NULL);
        return 0;

}
```

这三个线程函数不再多说。在信号量对象有信号的状态下，线程执行到 WaitForSingleObject 语句处继续执行，同时可用线程数减 1；若线程执行到 WaitForSingleObject 语句时信号量对象无信号，线程就在这里等待，直到信号量对象有信号线程才往下执行。

双击按钮 IDC_START，添加其响应函数：

```
void CMultiThread10Dlg::OnStart()
{
        CWinThread *pWriteA=AfxBeginThread(WriteA,
            &m_ctrlA,
            THREAD_PRIORITY_NORMAL,
            0,
            CREATE_SUSPENDED);
        pWriteA->ResumeThread();

        CWinThread *pWriteB=AfxBeginThread(WriteB,
            &m_ctrlB,
            THREAD_PRIORITY_NORMAL,
            0,
            CREATE_SUSPENDED);
```

```
pWriteB->ResumeThread();

CWinThread *pWriteC=AfxBeginThread(WriteC,
    &m_ctrlC,
    THREAD_PRIORITY_NORMAL,
    0,
    CREATE_SUSPENDED);
pWriteC->ResumeThread();

}
```

八、线程安全

如果你的代码所在的进程中有多个线程在同时运行，而这些线程可能会同时运行这段代码。如果每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，就是线程安全的。

一个类或者程序所提供的接口对于线程来说是原子操作或者多个线程之间的切换不会导致该接口的执行结果存在二义性,也就是说我们不用考虑同步的问题。

即：线程安全： 在多线程中使用时,不用自己做同步处理.

线程不安全：在多线程中使用时, 必须做线程同步,不然会有未知后果.

线程安全问题都是由**全局变量**及**静态变量**引起的。

若每个线程中对全局变量、静态变量只有读操作，而无写操作，一般来说，这个全局变量是线程安全的；若有多个线程同时执行写操作，一般都需要**考虑线程同步**，否则就可能影响线程安全。