When using Windows 95 (and other modern operating systems), you know that you can run several programs simultaneously. This capability is called *multitasking.* What you may not know is that many of today's operating systems also enable *threads,* which are separate processes that are not complete applications. A thread is a lot like a subprogram. An application can create several threads--several different flows of execution--and run them concurrently. Threads give you the ability to have multitasking inside multitasking. The user knows that he can run several applications at a time. The programmer knows that each application can run several threads at a time. In this chapter, you'll learn how to create and manage threads in your applications.

## **Understanding Simple Threads**

A thread is a path of execution through a program. In a multithreaded program, each thread has its own stack and operates independently of other threads running within the same program. MFC distinguishes between *UI threads*, which have a message pump and typically perform user interface tasks, and *worker threads*, which do not.

**NOTE:** Any application always has at least one thread, which is the program's primary or main thread. You can start and stop as many additional threads as you need, but the main thread keeps running as long as the application is active.

A thread is the smallest unit of execution, much smaller than a *process*. Generally each running application on your system is a process. If you start the same application (for example, Notepad) twice, there will be two processes, one for each instance. It is possible for several instances of an application to share a single process: for example, if you choose File, New Window in Internet Explorer, there are two applications on your taskbar, and they share a process. The unfortunate consequence is that if one instance crashes, they all do.

To create a worker thread using MFC, all you have to do is write a function that you want to run parallel with the rest of your application. Then call AfxBeginThread() to start a thread that will execute your function. The thread remains active as long as the thread's function is executing: When the thread function exits, the thread is destroyed. A simple call to AfxBeginThread() looks like this:

AfxBeginThread(ProcName, param, priority);

In the preceding line, ProcName is the name of the thread's function, param is any 32-bit value you want to pass to the thread, and priority is the thread's priority, which is represented by a number of predefined constants. Table 27.1 shows those constants and their descriptions.

#### **Table 27.1  Thread Priority Constants**

|  |  |
| --- | --- |
| **Constant** | **Description** |
| THREAD\_PRIORITY\_ABOVE\_NORMAL | Sets a priority one point higher than normal. |
| THREAD\_PRIORITY\_BELOW\_NORMAL | Sets a priority one point lower than normal. |
| THREAD\_PRIORITY\_HIGHEST | Sets a priority two points above normal. |
| THREAD\_PRIORITY\_IDLE | Sets a base priority of 1. For a REALTIME\_PRIORITY\_CLASS process, this sets a priority of 16. |
| THREAD\_PRIORITY\_LOWEST | Sets a priority two points below normal. |
| THREAD\_PRIORITY\_NORMAL | Sets normal priority. |
| THREAD\_PRIORITY\_TIME\_CRITICAL | Sets a base priority of 15. For a REALTIME\_PRIORITY\_CLASS process, this sets a priority of 30. |

**NOTE:** A thread's priority determines how often the thread takes control of the system, relative to the other running threads. Generally, the higher the priority, the more running time the thread gets, which is why the value of THREAD\_PRIORITY\_TIME\_CRITICAL is so high.

To see a simple thread in action, build the Thread application as detailed in the following steps.

**1.** Start a new AppWizard project workspace called **Thread**, as shown in Figure 27.1.

**FIG. 27.1** *Start an AppWizard project workspace called* **Thread***.*

**2.** Give the new project the following settings in the AppWizard dialog boxes. The New Project Information dialog box will then look like Figure 27.2.

Step 1: Single document

Step 2: Default settings

Step 3: Default settings

Step 4: Turn off all options

Step 5: Default settings

Step 6: Default settings

**FIG. 27.2** *These are the AppWizard settings for the Thread project.*

**3.** Use the resource editor to add a Thread menu to the application's IDR\_MAINFRAME menu. Give the menu one command called **Start Thread** with a command ID of **ID\_STARTTHREAD**, and enter a sensible prompt and ToolTip, as shown in Figure 27.3.

**FIG. 27.3** *Add a Thread menu with a Start Thread command.*

**4.** Use ClassWizard to associate the ID\_STARTTHREAD command with the OnStartthread() message-response function, as shown in Figure 27.4. Make sure that you have **CThreadView** selected in the Class Name box before you add the function.

**FIG. 27.4** *Add the OnStartthread() message-response function to the view class.*

**5.** Click the Edit Code button and then add the following lines to the new OnStartthread() function, replacing the TODO: Add your command handler code here comment:

HWND hWnd = GetSafeHwnd();

AfxBeginThread(ThreadProc, hWnd, THREAD\_PRIORITY\_NORMAL);

This code will call a function called ThreadProc within a worker thread of its own. Next, add ThreadProc, shown in Listing 27.1, to ThreadView.cpp, placing it right before the OnStartthread() function. Note that ThreadProc() is a global function and not a member function of the CThreadView class, even though it is in the view class's implementation file.

#### **Listing 27.1  ThreadView.cpp--ThreadProc()**

UINT ThreadProc(LPVOID param)

{

::MessageBox((HWND)param, "Thread activated.", "Thread", MB\_OK);

return 0;

}

This threaded function doesn't do much, just reports that it was started. The SDK function MessageBox() is very much like AfxMessageBox(), but because this isn't a member function of a class derived from CWnd, you can't use AfxMessageBox().

**TIP:** The double colons in front of a function name indicate a call to a global function, instead of an MFC class member function. For Windows programmers, this usually means an API or SDK call. For example, inside an MFC window class, you can call MessageBox("Hi, There!") to display *Hi, There!* to the user. This form of MessageBox() is a member function of the MFC window classes. To call the original Windows version, you write something like ::MessageBox(0, "Hi, There!", "Message", MB\_OK). Notice the colons in front of the function name and the additional arguments.

When you run the Thread program, the main window appears. Select the Thread, StartThread command, and the system starts the thread represented by the ThreadProc() function and displays a message box, as shown in Figure 27.5.

**FIG. 27.5** *The simple secondary thread in the Thread program displays a message box and then ends.*

## **Understanding Thread Communication**

Usually, a secondary thread performs some sort of task for the main program, which implies that there needs to be a channel of communication between the program (which is also a thread) and its secondary threads. There are several ways to accomplish these communications tasks: with global variables, event objects, and messages. In this section, you'll explore these thread-communication techniques.

### **Communicating with Global Variables**

Suppose you want your main program to be able to stop the thread. You need a way, then, to tell the thread when to stop. One method is to set up a global variable and then have the thread monitor the global variable for a value that signals the thread to end. Because the threads share the same address space, they have the same global variables. To see how this technique works, modify the Thread application as follows:

**1.** Use the resource editor to add a **Stop Thread** command to the application's Thread menu. Give this new command the **ID\_STOPTHREAD** ID, as shown in Figure 27.6.

**FIG. 27.6** *Add a Stop Thread command to the Thread menu.*

**2.** Use ClassWizard to associate the ID\_STOPTHREAD command with the OnStopthread() message-response function, as shown in Figure 27.7. Make sure that you have **CThreadView** selected in the Class Name box before you add the function. Add the following line to the OnStopthread() function, replacing the TODO: Add your command handler code here comment:

threadController = 0;

**FIG. 27.7** *Add the OnStopthread() message-response function.*

This refers to a new global variable you are about to declare.

**3.** Add the following line to the top of the ThreadView.cpp file, right after the endif directive:

volatile int threadController;

The volatile keyword means that you expect this variable will be changed from outside a thread that uses it. The keyword requests that the compiler not cache the variable in a register or in any way count on the value staying unchanged just because code in one thread doesn't seem to change it.

**4.** Add the following line to the OnStartthread() function, before the two lines you added earlier:

threadController = 1;

By now, perhaps, you've guessed that the value of threadController determines whether the thread will continue. Replace the ThreadProc() function with the one shown in Listing 27.2.

#### **Listing 27.2  The New ThreadProc() Function**

UINT ThreadProc(LPVOID param)

{

::MessageBox((HWND)param, "Thread activated.", "Thread", MB\_OK);

while (threadController == 1)

{

;

}

::MessageBox((HWND)param, "Thread stopped.", "Thread", MB\_OK);

return 0;

}

Now the thread first displays a message box, telling the user that the thread is starting. Then a while loop continues to check the threadController global variable, waiting for its value to change to 0. Although this while loop is trivial, it is here that you would place the code that performs whatever task you want the thread to perform, making sure not to tie things up for too long before rechecking the value of threadController.

Try a test: Build and run the program, and choose Thread, Start Thread to start the secondary thread. When you do, a message box appears, telling you that the new thread was started. To stop the thread, select the Thread, Stop Thread command. Again, a message box appears, this time telling you that the thread is stopping.

**CAUTION:** Using global variables to communicate between threads is, to say the least, an unsophisticated approach to thread communication and can be a dangerous technique if you're not sure how C++ handles variables from an assembly-language level. Other thread-communication techniques are safer and more elegant.

### **Communicating with User-Defined Messages**

Now you have a simple, albeit unsophisticated, method for communicating information from your main program to your thread. How about the reverse? That is, how can your thread communicate with the main program? The easiest method to accomplish this communication is to incorporate user-defined Windows messages into the program.

The first step is to define a user message, which you can do easily, like this:

const WM\_USERMSG = WM\_USER + 100;

The WM\_USER constant, defined by Windows, holds the first available user-message number. Because other parts of your program may use some user messages for their own purposes, the preceding line sets WM\_USERMSG to WM\_USER+100.

After defining the message, you call ::PostMessage() from the thread to send the message to the main program whenever you need to. (Message handling was discussed in Chapter 3, "Messages and Commands." Sending your own messages allows you to take advantage of the message-handling facility built into MFC.) A typical call to ::PostMessage() might look like this:

::PostMessage((HWND)param, WM\_USERMSG, 0, 0);

PostMessage()'s four arguments are the handle of the window to which the message should be sent, the message identifier, and the message's WPARAM and LPARAM parameters.

Modify the Thread application according to the next steps to see how to implement posting user messages from a thread.

**1.** Add the following line to the top of the ThreadView.h header file, right before the beginning of the class declaration:

const WM\_THREADENDED = WM\_USER + 100;

**2.** Still in the header file, add the following line to the message map, right after the //{{AFX\_MSG(CThreadView) comment and before DECLARE\_MESSAGE\_MAP:

afx\_msg LONG OnThreadended(WPARAM wParam, LPARAM lParam);

**3.** Switch to the ThreadView.cpp file and add the following line to the class's message map, making sure to place it right *after* the }}AFX\_MSG\_MAP comment:

ON\_MESSAGE(WM\_THREADENDED, OnThreadended)

**4.** Replace the ThreadProc() function with the one shown in Listing 27.3.

#### **Listing 27.3  The Message-Posting ThreadProc()**

UINT ThreadProc(LPVOID param)

{

::MessageBox((HWND)param, "Thread activated.", "Thread", MB\_OK);

while (threadController == 1)

{

;

}

::PostMessage((HWND)param, WM\_THREADENDED, 0, 0);

return 0;

}

**5.** Add the function shown in Listing 27.4 to the end of the ThreadView.cpp file.

#### **Listing 27.4  CThreadView::OnThreadended()**

LONG CThreadView::OnThreadended(WPARAM wParam, LPARAM lParam)

{

AfxMessageBox("Thread ended.");

return 0;

}

When you run the new version of the Thread program, select the Thread, Start Thread command to start the thread. When you do, a message box appears, telling you that the thread has started. To end the thread, select the Thread, Stop Thread command. Just as with the previous version of the program, a message box appears, telling you that the thread has ended.

Although this version of the Thread application seems to run identically to the previous version, there's a subtle difference. Now the program displays the message box that signals the end of the thread in the main program rather than from inside the thread. The program can do this because, when the user selects the Stop Thread command, the thread sends a WM\_THREADENDED message to the main program. When the program receives that message, it displays the final message box.

### **Communicating with Event Objects**

A slightly more sophisticated method of signaling between threads is to use *event objects,* which under MFC are represented by the CEvent class. An event object can be in one of two states: signaled and nonsignaled. Threads can watch for events to be signaled and so perform their operations at the appropriate time. Creating an event object is as easy as declaring a global variable, like this:

CEvent threadStart;

Although the CEvent constructor has a number of optional arguments, you can usually get away with creating the default object, as shown in the previous line of code. On creation, the event object is automatically in its nonsignaled state. To signal the event, you call the event object's SetEvent() member function, like this:

threadStart.SetEvent();

After the preceding line executes, the threadStart event object will be in its signaled state. Your thread should be watching for this signal so that the thread knows it's okay to get to work. How does a thread watch for a signal? By calling the Windows API function, WaitForSingleObject():

::WaitForSingleObject(threadStart.m\_hObject, INFINITE);

This function's two arguments are

* The handle of the event for which to check (stored in the event object's m\_hObject data member)
* The length of time the function should wait for the event

The predefined INFINITE constant tells WaitForSingleObject() not to return until the specified event is signaled. In other words, if you place the preceding line at the beginning of your thread, the system suspends the thread until the event is signaled. Even though you've started the thread execution, it's halted until whatever you need to happen happens. When your program is ready for the thread to perform its duty, you call the SetEvent() function, as previously described.

When the thread is no longer suspended, it can go about its business. However, if you want to signal the end of the thread from the main program, the thread must watch for this next event to be signaled. The thread can do this by polling for the event. To poll for the event, you again call WaitForSingleObject(), only this time you give the function a wait time of 0, like this:

::WaitForSingleObject(threadend.m\_hObject, 0);

In this case, if WaitForSingleObject() returns WAIT\_OBJECT\_0, the event has been signaled. Otherwise, the event is still in its nonsignaled state.

To better see how event objects work, follow these steps to further modify the Thread application:

**1.** Add the following line to the top of the ThreadView.cpp file, right after the line #include "ThreadView.h":

#include "afxmt.h"

**2.** Add the following lines near the top of the ThreadView.cpp file, after the volatile int threadController line that you placed there previously:

CEvent threadStart;

CEvent threadEnd;

**3.** Delete the volatile int threadController line from the file.

**4.** Replace the ThreadProc() function with the one shown in Listing 27.5.

#### **Listing 27.5  Yet Another ThreadProc()**

UINT ThreadProc(LPVOID param)

{

::WaitForSingleObject(threadStart.m\_hObject, INFINITE);

::MessageBox((HWND)param, "Thread activated.",

"Thread", MB\_OK);

BOOL keepRunning = TRUE;

while (keepRunning)

{

int result =

::WaitForSingleObject(threadEnd.m\_hObject, 0);

if (result == WAIT\_OBJECT\_0)

keepRunning = FALSE;

}

::PostMessage((HWND)param, WM\_THREADENDED, 0, 0);

return 0;

}

**5.** Replace all the code in the OnStartthread() function with the following line:

threadStart.SetEvent();

**6.** Replace the code in the OnStopthread() function with the following line:

threadEnd.SetEvent();

**7.** Use ClassWizard to add an OnCreate() function that handles the WM\_CREATE message, as shown in Figure 27.8. Make sure that you have **CThreadView** selected in the Class Name box before you add the function.

**FIG. 27.8** *Use ClassWizard to add the OnCreate() function.*

**8.** Add the following lines to the OnCreate() function, replacing the TODO: Add your specialized creation code here comment:

HWND hWnd = GetSafeHwnd();

AfxBeginThread(ThreadProc, hWnd);

Again, this new version of the program seems to run just like the preceding version. However, the program is now using both event objects and user-defined Windows messages to communicate between the main program and the thread. No more messing with clunky global variables.

One big difference from previous versions of the program is that the secondary thread is begun in the OnCreate() function, which is called when the application first runs and creates the view. However, because the first line of the thread function is the call to WaitForSingleObject(), the thread immediately suspends execution and waits for the threadStart event to be signaled.

When the threadStart event object is signaled, the thread is free to display the message box and then enter its while loop, where it polls the threadEnd event object. The while loop continues to execute until threadEnd is signaled, at which time the thread sends the WM\_THREADENDED message to the main program and exits. Because the thread is started in OnCreate(), after the thread ends, it can't be restarted.

## **Using Thread Synchronization**

Using multiple threads can lead to some interesting problems. For example, how do you prevent two threads from accessing the same data at the same time? What if, for example, one thread is in the middle of trying to update a data set when another thread tries to read that data? The second thread will almost certainly read corrupted data because only some of the data set will have been updated.

Trying to keep threads working together properly is called *thread synchronization*. Event objects, about which you just learned, are a form of thread synchronization. In this section, you'll learn about *critical sections, mutexes,* and *semaphores*--thread synchronization objects that make your thread programming even safer.

### **Using Critical Sections**

Critical sections are an easy way to ensure that only one thread at a time can access a data set. When you use a critical section, you give your threads an object that they have to share. Whichever thread possesses the critical-section object has access to the guarded data. Other threads have to wait until the first thread releases the critical section, after which another thread can grab the critical section to access the data in turn.

Because the guarded data is represented by a single critical-section object and because only one thread can own the critical section at any given time, the guarded data can never be accessed by more than a single thread at a time.

To create a critical-section object in an MFC program, you create an instance of the CCriticalSection class, like this:

CCriticalSection criticalSection;

Then, when program code is about to access the data that you want to protect, you call the critical-section object's Lock() member function, like this:

criticalSection.Lock();

If another thread doesn't already own the critical section, Lock() gives the object to the calling thread. That thread can then access the guarded data, after which it calls the critical-section object's Unlock() member function:

criticalSection.Unlock();

Unlock() releases the ownership of the critical-section object so that another thread can grab it and access the guarded data.

The best way to implement something like critical sections is to build the data you want to protect into a thread-safe class. When you do this, you no longer have to worry about thread synchronization in the main program; the class handles it all for you. As an example, look at Listing 27.6, which is the header file for a thread-safe array class.

#### **Listing 27.6  COUNTARRAY.H--The CCountArray Class Header File**

#include "afxmt.h"

class CCountArray

{

private:

int array[10];

CCriticalSection criticalSection;

public:

CCountArray() {};

~CCountArray() {};

void SetArray(int value);

void GetArray(int dstArray[10]);

};

The header file starts by including the MFC header file, afxmt.h, which gives the program access to the CCriticalSection class. Within the CCountArray class declaration, the file declares a 10-element integer array, which is the data that the critical section will guard, and declares the critical-section object, here called criticalSection. The CCountArray class's public member functions include the usual constructor and destructor, as well as functions for setting and reading the array. These latter two member functions must deal with the critical-section object because these functions access the array.

Listing 27.7 is the CCountArray class's implementation file. Notice that, in each member function, the class takes care of locking and unlocking the critical-section object. This means that any thread can call these member functions without worrying about thread synchronization. For example, if thread 1 calls SetArray(), the first thing SetArray() does is call criticalSection.Lock(), which gives the critical-section object to thread 1. The complete for loop then executes, without any fear of being interrupted by another thread. If thread 2 calls SetArray() or GetArray(), the call to criticalSection.Lock() suspends thread 2 until thread 1 releases the critical-section object, which it does when SetArray() finishes the for loop and executes the criticalSection.Unlock() line. Then the system wakes up thread 2 and gives it the critical-section object. In this way, all threads have to wait politely for their chance to access the guarded data.

#### **Listing 27.7  COUNTARRAY.CPP--The CCountArray Class Implementation File**

#include "stdafx.h"

#include "CountArray.h"

void CCountArray::SetArray(int value)

{

criticalSection.Lock();

for (int x=0; x<10; ++x)

array[x] = value;

criticalSection.Unlock();

}

void CCountArray::GetArray(int dstArray[10])

{

criticalSection.Lock();

for (int x=0; x<10; ++x)

dstArray[x] = array[x];

criticalSection.Unlock();

}

Now that you've had a chance to see what a thread-safe class looks like, it's time to put the class to work. Perform the following steps, which modify the Thread application to test the CCountArray class:

**1.** Use the File, New command to add a new C++ header file called **CountArray.h** to the project, as shown in Figure 27.9. Enter the code from Listing 27.6.

**2.** Again choose File, New, and create a new C++ source file called **CountArray.cpp** in this project. Enter the code from Listing 27.7.

**3.** Switch to ThreadView.cpp and add the following line near the top of the file, after the line #include "afxmt.h", which you placed there previously:

#include "CountArray.h"

**4.** Add the following line near the top of the file, after the CEvent threadEnd line you placed there previously:

CCountArray countArray;

**5.** Delete the CEvent threadStart and CEvent threadEnd lines from the file.

**FIG. 27.9** *Add CountArray.h to the Thread project.*

**6.** Delete the lines ON\_MESSAGE(WM\_THREADENDED, OnThreadended), ON\_COMMAND(ID\_STOPTHREAD, OnStopthread), and ON\_WM\_CREATE() from the message map.

**7.** Replace the ThreadProc() function with the thread functions shown in Listing 27.8.

#### **Listing 27.8  WriteThreadProc() and ReadThreadProc()**

UINT WriteThreadProc(LPVOID param)

{

for(int x=0; x<10; ++x)

{

countArray.SetArray(x);

::Sleep(1000);

}

return 0;

}

UINT ReadThreadProc(LPVOID param)

{

int array[10];

for (int x=0; x<20; ++x)

{

countArray.GetArray(array);

char str[50];

str[0] = 0;

for (int i=0; i<10; ++i)

{

int len = strlen(str);

wsprintf(&str[len], "%d ", array[i]);

}

::MessageBox((HWND)param, str, "Read Thread", MB\_OK);

}

return 0;

}

**8.** Replace all the code in the OnStartthread() function with the following lines:

HWND hWnd = GetSafeHwnd();

AfxBeginThread(WriteThreadProc, hWnd);

AfxBeginThread(ReadThreadProc, hWnd);

**9.** Delete the OnStopthread(), OnThreadended, and OnCreate() functions from the file.

**10.** Switch to the ThreadView.h file and delete the line const WM\_THREADENDED = WM\_USER + 100 from the listing.

**11.** Also, in ThreadView.h, delete the lines afx\_msg LONG OnThreadended(WPARAM wParam, LPARAM lParam), afx\_msg void OnStopthread(), and afx\_msg int OnCreate(LPCREATESTRUCT lpCreateStruct) from the message map.

**12.** Using the resource editor, remove the Stop Thread command from the Thread menu.

Now build and run the new version of the Thread application. When you do, the main window appears. Select the Thread, Start Thread command to get things hopping. The first thing you'll see is a message box (see Figure 27.10) displaying the current values in the guarded array. Each time you dismiss the message box, it reappears with the array's new contents. The message box will reappear 20 times. The values listed in the message box depend on how often you dismiss the message box. The first thread is writing new values into the array once a second, even as you're viewing the array's contents in the second thread.

**FIG. 27.10** *This message box displays the current contents of the guarded array.*

The important thing to notice is that at no time does the second thread interrupt when the first thread is changing the values in the array. You can tell that this is true because the array always contains 10 identical values. If the first thread were interrupted as it modified the array, the 10 values in the array would not be identical, as shown in Figure 27.11.

If you examine the source code carefully, you'll see that the first thread, named WriteThreadProc(), is calling the array class's SetArray() member function 10 times within a for loop. Each time through the loop, SetArray() gives the thread the critical-section object, changes the array contents to the passed number, and then takes the critical-section object away again. Note the call to the Sleep() function, which suspends the thread for the number of milliseconds given as the function's single argument.

**FIG. 27.11** *Without thread synchronization, you might see something like this in the message box.*

The second thread, named ReadThreadProc(), is also trying to access the same critical-section object to construct a display string of the values contained in the array. However, if WriteThreadProc() is currently trying to fill the array with new values, ReadThreadProc() has to wait. The inverse is also true. That is, WriteThreadProc() can't access the guarded data until it can regain ownership of the critical section from ReadThreadProc().

If you really want to prove that the critical-section object is working, remove the criticalSection.Unlock() line from the end of the CCountArray class's SetArray() member function. Then compile and run the program. This time when you start the threads, no message box appears. Why? Because WriteThreadProc() takes the critical-section object and never lets it go, which forces the system to suspend ReadThreadProc() forever (or at least until you exit the program).

### **Using Mutexes**

Mutexes are a lot like critical sections but a little more complicated because they enable safe sharing of resources, not only between threads in the same application but also between threads of different applications. Although synchronizing threads of different applications is beyond the scope of this chapter, you can get a little experience with mutexes by using them in place of critical sections.

Listing 27.9 is the CCountArray2 class's header file. Except for the new classname and the mutex object, this header file is identical to the original CountArray.h. Listing 27.10 is the modified class's implementation file. As you can see, the member functions look a lot different when they are using mutexes instead of critical sections, even though both objects provide essentially the same type of services.

#### **Listing 27.9  CCOUNTARRAY2.H--The CCountArray2 Class Header File**

#include "afxmt.h"

class CCountArray2

{

private:

int array[10];

CMutex mutex;

public:

CCountArray2() {};

~CCountArray2() {};

void SetArray(int value);

void GetArray(int dstArray[10]);

};

#### **Listing 27.10  COUNTARRAY2.CPP--The CCountArray2 Class Implementation File**

#include "stdafx.h"

#include "CountArray2.h"

void CCountArray2::SetArray(int value)

{

CSingleLock singleLock(&mutex);

singleLock.Lock();

for (int x=0; x<10; ++x)

array[x] = value;

}

void CCountArray2::GetArray(int dstArray[10])

{

CSingleLock singleLock(&mutex);

singleLock.Lock();

for (int x=0; x<10; ++x)

dstArray[x] = array[x];

}

To access a mutex object, you must create a CSingleLock or CMultiLock object, which performs the actual access control. The CCountArray2 class uses CSingleLock objects because this class is dealing with only a single mutex. When the code is about to manipulate guarded resources (in this case, the array), you create a CSingleLock object, like this:

CSingleLock singleLock(&mutex);

The constructor's argument is a pointer to the thread-synchronization object that you want to control. Then, to gain access to the mutex, you call the CSingleLock object's Lock() member function:

singleLock.Lock();

If the mutex is unowned, the calling thread becomes the owner. If another thread already owns the mutex, the system suspends the calling thread until the mutex is released, at which time the waiting thread is awakened and takes control of the mutex.

To release the mutex, you call the CSingleLock object's Unlock() member function. However, if you create your CSingleLock object on the stack (rather than on the heap, using the new operator) as shown in Listing 27.10, you don't have to call Unlock() at all. When the function exits, the object goes out of scope, which causes its destructor to execute. The destructor automatically unlocks the object for you.

To try out the new CCountArray2 class in the Thread application, add new CountArray2.h and CountArray2.cpp files to the Thread project and then delete the original CountArray.h and CountArray.cpp files. Finally, in ThreadView.cpp, change all references to CCountArray to CCountArray2. Because all the thread synchronization is handled in the CCountArray2 class, no further changes are necessary to use mutexes instead of critical sections.

### **Using Semaphores**

Although semaphores are used like critical sections and mutexes in an MFC program, they serve a slightly different function. Rather than enable only one thread to access a resource at a time, semaphores enable multiple threads to access a resource, but only to a point. That is, semaphores enable a maximum number of threads to access a resource simultaneously.

When you create the semaphore, you tell it how many threads should be allowed simultaneous access to the resource. Then, each time a thread grabs the resource, the semaphore decrements its internal counter. When the counter reaches 0, no further threads are allowed access to the guarded resource until another thread releases the resource, which increments the semaphore's counter.

You create a semaphore by supplying the initial count and the maximum count, like this:

CSemaphore Semaphore(2, 2);

Because in this section you'll be using a semaphore to create a thread-safe class, it's more convenient to declare a CSemaphore pointer as a data member of the class and then create the CSemaphore object dynamically in the class's constructor, like this:

semaphore = new CSemaphore(2, 2);

You should do this because you have to initialize a data member in the constructor rather than at the time you declare it. With the critical-section and mutex objects, you didn't have to supply arguments to the class's constructors, so you were able to create the object at the same time you declared it.

After you have created the semaphore object, it's ready to start counting resource access. To implement the counting process, you first create a CSingleLock object (or CMultiLock, if you're dealing with multiple thread-synchronization objects), giving it a pointer to the semaphore you want to use, like this:

CSingleLock singleLock(semaphore);

Then, to decrement the semaphore's count, you call the CSingleLock object's Lock() member function:

singleLock.Lock();

At this point, the semaphore object has decremented its internal counter. This new count remains in effect until the semaphore object is released, which you can do explicitly by calling the object's Unlock() member function:

singleLock.Unlock();

Alternatively, if you've created the CSingleLock object locally on the stack, you can just let the object go out of scope, which not only automatically deletes the object but also releases the hold on the semaphore. In other words, both calling Unlock() and deleting the CSingleLock object increment the semaphore's counter, enabling a waiting thread to access the guarded resource.

Listing 27.11 is the header file for a class called CSomeResource. CSomeResource is a mostly useless class whose only calling is to demonstrate the use of semaphores. The class has a single data member, which is a pointer to a CSemaphore object. The class also has a constructor and destructor, as well as a member function called UseResource(), which is where the semaphore will be used.

#### **Listing 27.11  SOMERESOURCE.H**

#include "afxmt.h"

class CSomeResource

{

private:

CSemaphore\* semaphore;

public:

CSomeResource();

~CSomeResource();

void UseResource();

};

Listing 27.12 shows the CSomeResource class's implementation file. You can see that the CSemaphore object is constructed dynamically in the class's constructor and deleted in the destructor. The UseResource() member function simulates accessing a resource by attaining a count on the semaphore and then sleeping for five seconds, after which the hold on the semaphore is released when the function exits and the CSingleLock object goes out of scope.

#### **Listing 27.12  SOMERESOURCE.CPP**

#include "stdafx.h"

#include "SomeResource.h"

CSomeResource::CSomeResource()

{

semaphore = new CSemaphore(2, 2);

}

CSomeResource::~CSomeResource()

{

delete semaphore;

}

void CSomeResource::UseResource()

{

CSingleLock singleLock(semaphore);

singleLock.Lock();

Sleep(5000);

}

If you modify the Thread application to use the CSomeResource object, you can watch semaphores at work. Follow these steps:

**1.** Delete any CountArray files that are still in the project. (In FileView, click the file once to select it; then press Del to delete the file from the project.)

**2.** Create the new empty **SomeResource.h** and **SomeResource.cpp** files in the project.

**3.** Add the code from Listings 27.11 and 27.12 to these empty files.

**4.** Load ThreadView.cpp and replace the line #include "CountArray2.h" with the following:

#include "SomeResource.h"

**5.** Replace the line CCountArray2 countArray with the following:

CSomeResource someResource;

**6.** Replace the WriteThreadProc() and ReadThreadProc() functions with the functions shown in Listing 27.13.

#### **Listing 27.13  ThreadProc1(), ThreadProc2(), and ThreadProc3()**

UINT ThreadProc1(LPVOID param)

{

someResource.UseResource();

::MessageBox((HWND)param,

"Thread 1 had access.", "Thread 1", MB\_OK);

return 0;

}

UINT ThreadProc2(LPVOID param)

{

someResource.UseResource();

::MessageBox((HWND)param,

"Thread 2 had access.", "Thread 2", MB\_OK);

return 0;

}

UINT ThreadProc3(LPVOID param)

{

someResource.UseResource();

::MessageBox((HWND)param,

"Thread 3 had access.", "Thread 3", MB\_OK);

return 0;

}

**7.** Replace the code in the OnStartthread() function with that shown in Listing 27.14.

#### **Listing 27.14  LST27\_14.TXT--New Code for the OnStartthread() Function**

HWND hWnd = GetSafeHwnd();

AfxBeginThread(ThreadProc1, hWnd);

AfxBeginThread(ThreadProc2, hWnd);

AfxBeginThread(ThreadProc3, hWnd);

Now compile and run the new version of the Thread application. When the main window appears, select the Thread, Start Thread command. In about five seconds, two message boxes will appear, informing you that thread 1 and thread 2 had access to the guarded resource. About five seconds after that, a third message box will appear, telling you that thread 3 also had access to the resource. Thread 3 took five seconds longer because thread 1 and thread 2 grabbed control of the resource first. The semaphore is set to allow only two simultaneous resource accesses, so thread 3 had to wait for thread 1 or thread 2 to release its hold on the semaphore.

**NOTE:** Although the sample programs in this chapter have demonstrated using a single thread-synchronization object, you can have as many synchronization objects as you need in a single program. You can even use critical sections, mutexes, and semaphores all at once to protect different data sets and resources in different ways.

For complex applications, threads offer the capability to maintain fast and efficient data processing. You no longer have to wait for one part of the program to finish its task before moving on to something else. For example, a spreadsheet application could use one thread to update the calculations while the main thread continues accepting entries from the user. Using threads, however, leads to some interesting problems, not the least of which is the need to control access to shared resources. Writing a threaded application requires thought and careful consideration of how the threads will be used and what resources they'll access.

Mutex Vs. Semaphore

-----------------------

1.

A mutex is essentially a binary semaphore, but with the additional property that it is logically "owned" by the executable entity (whether process or thread) that locked it. This property improves code structuring and allows some useful optimizations. You can't lock a mutex in one thread and then unlock it in another. You CAN, however, lock a binary semaphore in one thread and unlock it in another, because while a binary semaphore may be "locked" or "unlocked", it is never "owned".

A counting semaphore can be used to manage multiple resources, or for generalized blocking rather than "locking". That is, a semaphore initialized to the value 0 will always block when {waited, locked, decreased}, and the waiter will be unblocked when it is {increased, posted, unlocked}. If you initialize the semaphore to 1, the first "lock" will succeed, while a second will block. If you have 5-shared buffers, you can sometimes manage them simply by using a counting semaphore initialized to the value 5. (The first 5 "locks" succeed, and subsequent "locks" block until the semaphore is "unlocked".)

2.

I went through the same pains when I tried to learn synchronization

objects.

The easiest way to understand the difference is to look at it as a

real life situation::

Scenario 1:

There is a room with three (could be more) chairs for people to sit.

Any number of people can attempt to enter the room. How do you ensure

that all the people who enter the room do so only if a seat is

available. Answer:- You assign a gatekeeper who guards the door ??

Scenario 2:

Lets us say there is another room with only one chair, so only one

person can be in at any time. How do you ensure that. Same answer:-

assign a gatekeeper.

Mutexes and Semaphores are both gatekeepers. We now have to make a

choice between which gatekeeper to use.

The gatekeeper in the first scenario has to be an intelligent one, as

he has to do some math. He has to keep count of how many people are

currently in, how many are going out etc. So if ten people are waiting

to get in (because the room is currently full), the gatekeeper has to

keep all of them waiting. When two of them leave, he notes that and

allows two people to get in.

The gatekeeper in the second scenario can afford to be dumb, he just

checks if the room is full or empty and lets one person in if it is

empty. No math, simple.

Semaphore is the intelligent gatekeeper as it keeps track of number of

threads that are allowed to access the resource it protects.

Mutex is the dumb guy, he allows only one thread to access his

resource.

If you did understand this concept correctly, the first question on

your mind should be this.

Can't the intelligent guy do the dumb guys job. Can't you replace a

Mutex with a Semaphore?

The answer is yes. You can replace any Mutex with a Semaphore of

MaxCount 1.

3.

Indeed, mutex is logically equivalent to a binary semaphore, locking/unlocking is achieved by the Dijskra's P-V operations. However, using your metaphore, there is a price to be smart: efficiency. Obviously, to solve a hard problem that requires a lot of skills, one usually needs more time than to solve a trivial problem... To resume this thread about sema/mutex, the important point that have been discussed: - A mutex is logically equivalent to a binary semaphore, where locking/unlocking is achieved using the Dijskra's P-V operations. - mutexes are usually more efficient than binary semaphores. As a rule of thumb, don't use semaphores when you can use mutexes / conditions variables. This is especially true on SMP machines. - mutex has an owner concept: unlocking a mutex can be only done by the thread that locked (or, equivalently, "owns") the mutex. - (counting) semaphores are a more general synchronization device than mutexes. - Did I missed something?? I planned to write a technical article about the difference between semas and mutex, showing some issues like semaphore convoy on SMP machine. When it's finished, promized, I post it on c.p.t. ;-)