第10章　线程同步工具包

多年来，我对线程的同步问题进行了许多开发工作，编写了一些C++类和组件。本章将介绍这些内容。希望这些代码有用，能够使你节省许多编程时间。

本章首先要介绍如何实现关键代码段和将各种特性添加给它的方法。尤其是，要学习如何在多个进程中使用关键代码段。然后要学习如何将数据类型包装在C++类中，使对象成为对线程安全的对象。使用这些类，将展示一种其行为特性与信标相反的对象。

接着要介绍如何解决一个常见的编程问题，即当多个线程读取一种资源但是只有一个线程写入资源时如何进行编程。Windows没有预先内置能够容易地实现这种类型的同步的特性，因此我编写了一个C++类以便实现这个特性。

最后要介绍如何实现WaitForMultipleExpressions函数，该函数可以用来创建复杂的表达式，以便指明应该何时唤醒线程（它的作用很像WaitForMultipleObjects函数，该函数可以用来等待任何单个对象变成已通知状态，或者使所有对象处于已通知状态）。

10.1　实现关键代码段：Optex

关键代码段始终对我有着巨大的吸引力。但是，如果它们只是用户方式对象，为什么不能自己来实现它们呢？为什么需要操作系统的支持特性才能使关键代码段运行呢？另外，如果编写自己的关键代码段，可能需要将各种特性添加给它，并用某种方法来增强它的性能。至少想要让它跟踪目前究竟哪个线程拥有该资源。如果有一个关键代码段能够实现这些操作，就能帮助解决代码中的死锁问题；可以使用一个调试程序来发现哪个线程没有释放该资源。

在进一步的说明之前，让我们来看一看究竟如何来实现关键代码段。我反复说，关键代码段属于用户方式对象。实际上，这种说法并不是百分之百的正确。如果一个线程试图进入另一个线程拥有的关键代码段，那么该线程就会被置于等待状态。如果要使它进入等待状态，唯一的办法是从用户方式转入内核方式。用户方式线程通过循环运行，就能够停止执行有用的操作，但是这不是个有效的等待方式，因此应该避免使用它。

关键代码段必须包含某个内核对象，以便使线程进入有效的等待状态。关键代码段的运行速度很快，因为只有当争用该关键代码段的时候，才使用该内核对象。只要线程能够立即获得对资源的访问权，并且使用该资源，然后释放该资源，而不与其他线程争用该资源，那么就不使用该内核对象，而且该线程决不会退出用户方式。在大多数应用程序中，两个线程很少会同时争用关键代码段。

Optex.h和Optex.cpp文件（见后面清单10-1）说明了关键代码段的实现方法。这里称关键代码段是一个Optex（这是optimized互斥对象（优化互斥对象）的缩略词），并将它作为一个C++类来实现。一旦理解了这个代码，就会懂得关键代码段的运行速度为什么比互斥对象内核对象快。

由于实现了关键代码段，因此可以将有用的特性添加给它。例如，COptex类使得不同进程中的线程能够实现同步。这是个令人叫绝的附加特性，这样就得到了一个高速运行的机制，使得不同进程中的线程之间能够互相进行通信。

若要使用optex，只需要声明一个COptex对象。该对象有3个构造函数：

```
COptex::(DWORD dwSpinCount = 4000);
```

```
COptex::(PCSTR  pszName, DWORD dwSpinCount = 4000);
COptex::(PCWSTR pszName, DWORD dwSpinCount = 4000);
```

第一个构造函数用于创建只能用来对单个进程中的各个线程进行同步的COptex对象。这种类型的optex占用的开销比跨进程的optex要少得多。另外两个构造函数可以用来创建在多个进程中的线程之间实现同步的optex。对于pszName参数，必须传递一个ANSI或Unicode字符串，该字符串用于对每个共享的optex进行标识。若要使两个或多个进程共享一个optex，那么两个进程必须建立一个COptex对象的实例，并且传递相同的字符串名字。

如果线程要进入和退出COptex对象，请调用Enter和leave方法：

```
void COptex::Enter();
void COptex::Leave();
```

我甚至列入了关键代码段的TryEnterCriticalSection和SetCriticalSectionSpinCount函数的等价方法：

```
BOOL COptex::TryEnter();
void COptex::SetSpinCount(DWORD dwSpinCount);
```

如果需要知道optex是属于单进程optex还是跨进程optex，可以调用下面所示的最后一种方法（很少需要调用这个方法，但是内部方法函数有时要调用它）。

```
BOOL COptex::IsSingleProcessOptex() const;
```

这些就是在使用optex时需要知道的所有（公用）函数。现在我准备介绍optex是如何运行的。一般来说，optex（和关键代码段）包含了许多成员变量。这些变量反映了optex的状态。在Optex.h文件中，大多数成员变量采用SHAREDINFO结构，少数成员变量属于类本身的成员。表10-1描述了每个成员的作用。

表10-1 成员变量描述

成 员	描 述
m_lLockCount	指明线程试图进入optex的次数。如果没有线程进入optex，那么这个值是0
m_dwThreadId	指明拥有optex的线程的唯一ID。如果没有线程拥有optex，那么这个值是0
m_lRecurseCount	指明线程拥有optex的次数。如果optex没有被线程所拥有，则这个值是0
m_hevt	这是个事件内核对象的句柄，只有当一个线程试图在另一个线程拥有optex时进入该optex，才使用这个句柄。内核对象句柄是与进程相关的句柄，这就是该成员为什么不使用SHAREDINFO结构的原因
m_dwSpinCount	指明试图进入optex的线程在等待事件内核对象之前应该尝试进入的次数。在单处理器计算机上，这个值总是0
m_hfm	这是文件映象内核对象的句柄，当多进程共享一个optex时，便使用这个句柄。内核对象句柄属于与进程相关的句柄，这就是为什么该成员不是SHAREDINFO结构的原因。对于单进程optex来说，这个值总是NULL
m_psi	这是指向潜在的共享optex数据成员的指针。内存地址是与进程相关的地址，这就是为什么该成员不使用SHAREDINFO结构的原因。对于单进程optex来说，它指向一个堆栈分配的内存块。对于多进程optex来说，它指向一个内存映射文件

该源代码已经作了充分的说明，因此要理解optex如何来运行，不会遇到什么困难。需要特别指出的是，optex之所以能够获得较快的运行速度，原因是它大量使用了互锁函数家族。这使得该代码始终能够以用户方式来运行，并且避免了方式转换操作。

Optex示例应用程序

清单10-1列出的Optex（“10 Optex.exe”）应用程序用于测试COptex类，以便确保它能够正

确地运行。该应用程序的源代码和源文件位于本书所附光盘上的 10-Optex目录下。我总是在调试程序中运行该应用程序，因此能够密切注意所有的成员函数和变量。

当运行该应用程序时，它首先要检测它是不是运行该应用程序的第一个实例。我的做法是创建一个带名字的事件内核对象。在该应用程序的任何地方，实际上并不使用事件对象，创建这个对象只是为了观察GetLastError是否返回ERROR_ALREADY_EXISTS。如果它返回了这个值，那么我就知道它是运行的该应用程序的第二个实例。下面说明为什么要运行该应用程序的两个实例。

如果这是第一个实例，那么我创建一个单进程COptex对象，并且调用FirstFunc函数。该函数对该optex对象执行一系列的操作。这时创建第二个线程，它也对同一个 optex对象进行操作。这时，对该optex进行操作的两个线程都在同一个进程中。可以观察源代码，以便了解我执行的是什么测试。我设法说明所有可能出现的情况，这样， Coptex类中的所有代码都能够有机会运行。

测试了单进程optex后，我又测试了跨进程optex。在_tWinMain中，当首次调用的FirstFunc返回时，我创建了另一个 COptex optex对象，但是这一次我给该 optex赋予了一个字符串名字CrossOptexTest。只需创建一个带名字的optex，就可以使它成为一个跨进程optex。接着，我第二次调用FirstFunc函数，给它传递了跨进程optex的地址。这时FirstFunc基本上执行与以前一样的代码。但是这次它不是产生第二个线程，而是产生了一个子进程。

该子进程只是同一个应用程序的另一个实例。但是，当它启动运行时，它创建了一个事件内核对象，并且发现该事件对象已经存在。这就是第二个应用程序实例如何知道它是第二个实例并且执行与第一个实例不同的代码的方法。第二个实例首先做的事情是调用 DebugBreak函数：

```
VOID DebugBreak();
```

这个使用方便的函数能够强制一个调试程序运行，并且将它自己与该进程连接起来。这样就可以方便地调试应用程序的这两个实例。然后第二个实例创建一个跨进程 optex，传递相同的字符串名字。由于字符串名字是相同的，因此两个进程共享该 optex。顺便要说明的是，两个以上的进程可以共享同一个optex。

接着，应用程序的第二个实例调用 SecondFunc函数，为它传递跨进程optex的地址。这时，进行同一组测试，但是对optex进行操作的两个线程不在同一进程中。

清单10-1 optex示例应用程序

Optex.cpp

```
/******************************************************************************
Module:  Optex.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include "Optex.h"


//////////////////////////////////////////////////////////////////////////////
```

```
// 0=multi-CPU, 1=single-CPU, -1=not set yet
BOOL COptex::sm_fUniprocessorHost = -1;


//////////////////////////////////////////////////////////////////////////////


PSTR COptex::ConstructObjectName(PSTR pszResult,
   PCSTR pszPrefix, BOOL fUnicode, PVOID pszName) {

   pszResult[0] = 0;
   if (pszName == NULL)
      return(NULL);

   wsprintfA(pszResult, fUnicode ? "%s%S" : "%s%s", pszPrefix, pszName);
   return(pszResult);
}


//////////////////////////////////////////////////////////////////////////////
void COptex::CommonConstructor(DWORD dwSpinCount,
   BOOL fUnicode, PVOID pszName) {

   if (sm_fUniprocessorHost == -1) {
      // This is the 1st object constructed, get the number of CPUs
      SYSTEM_INFO sinf;
      GetSystemInfo(&sinf);
      sm_fUniprocessorHost = (sinf.dwNumberOfProcessors == 1);
   }

   m_hevt = m_hfm = NULL;
   m_psi  = NULL;

   if (pszName == NULL) {  // Creating a single-process optex

      m_hevt = CreateEventA(NULL, FALSE, FALSE, NULL);
      chASSERT(m_hevt != NULL);

      m_psi = new SHAREDINFO;
      chASSERT(m_psi != NULL);
      ZeroMemory(m_psi, sizeof(*m_psi));

   } else {                // Creating a cross-process optex

      // Always use ANSI so that this works on Win9x and Windows 2000
      char szResult[100];
      ConstructObjectName(szResult, "Optex_Event_", fUnicode, pszName);
      m_hevt = CreateEventA(NULL, FALSE, FALSE, szResult);
      chASSERT(m_hevt != NULL);

      ConstructObjectName(szResult, "Optex_MMF_", fUnicode, pszName);
      m_hfm = CreateFileMappingA(INVALID_HANDLE_VALUE, NULL,
         PAGE_READWRITE, 0, sizeof(*m_psi), szResult);
      chASSERT(m_hfm != NULL);
```

```
   m_psi = (PSHAREDINFO) MapViewOfFile(m_hfm,
      FILE_MAP_WRITE, 0, 0, 0);
   chASSERT(m_psi != NULL);

   // Note: SHAREDINFO's m_lLockCount, m_dwThreadId, and m_lRecurseCount
   // members need to be initialized to 0. Fortunately, a new pagefile
   // MMF sets all of its data to 0 when created. This saves us from
   // some thread synchronization work.
}

   SetSpinCount(dwSpinCount);
}


//////////////////////////////////////////////////////////////////////////////


COptex::~COptex() {

#ifdef _DEBUG
   if (IsSingleProcessOptex() && (m_psi->m_dwThreadId != 0)) {
      // A single-process optex shouldn't be destroyed if any thread owns it
      DebugBreak();
   }

   if (!IsSingleProcessOptex() &&
      (m_psi->m_dwThreadId == GetCurrentThreadId())) {

      // A cross-process optex shouldn't be destroyed if our thread owns it
      DebugBreak();
   }
#endif

   CloseHandle(m_hevt);

   if (IsSingleProcessOptex()) {
      delete m_psi;
   } else {
      UnmapViewOfFile(m_psi);
      CloseHandle(m_hfm);
   }
}


//////////////////////////////////////////////////////////////////////////////


void COptex::SetSpinCount(DWORD dwSpinCount) {

   // No spinning on single CPU machines
   if (!sm_fUniprocessorHost)
      InterlockedExchangePointer((PVOID*) &m_psi->m_dwSpinCount,
        (PVOID) (DWORD_PTR) dwSpinCount);
}
```

```
//////////////////////////////////////////////////////////////////////////////


void COptex::Enter() {

   // Spin, trying to get the optex
   if (TryEnter())
      return;  // We got it, return

   // We couldn't get the optex, wait for it.
   DWORD dwThreadId = GetCurrentThreadId();

   if (InterlockedIncrement(&m_psi->m_lLockCount) == 1) {

      // Optex is unowned, let this thread own it once
      m_psi->m_dwThreadId = dwThreadId;
      m_psi->m_lRecurseCount = 1;

   } else {

      if (m_psi->m_dwThreadId == dwThreadId) {

         // If optex is owned by this thread, own it again
         m_psi->m_lRecurseCount++;

      } else {

         // Optex is owned by another thread, wait for it
         WaitForSingleObject(m_hevt, INFINITE);

         // Optex is unowned, let this thread own it once
         m_psi->m_dwThreadId = dwThreadId;
         m_psi->m_lRecurseCount = 1;
      }
   }
}


//////////////////////////////////////////////////////////////////////////////


BOOL COptex::TryEnter() {

   DWORD dwThreadId = GetCurrentThreadId();
BOOL fThisThreadOwnsTheOptex = FALSE;     // Assume a thread owns the optex
DWORD dwSpinCount = m_psi->m_dwSpinCount; // How many times to spin

do {
   // If lock count = 0, optex is unowned, we can own it
   fThisThreadOwnsTheOptex = (0 ==
      InterlockedCompareExchange(&m_psi->m_lLockCount, 1, 0));

   if (fThisThreadOwnsTheOptex) {

      // Optex is unowned, let this thread own it once
```

```
         m_psi->m_dwThreadId = dwThreadId;
         m_psi->m_lRecurseCount = 1;

      } else {

         if (m_psi->m_dwThreadId == dwThreadId) {

            // If optex is owned by this thread, own it again
            InterlockedIncrement(&m_psi->m_lLockCount);
            m_psi->m_lRecurseCount++;
            fThisThreadOwnsTheOptex = TRUE;
         }
      }

   } while (!fThisThreadOwnsTheOptex && (dwSpinCount-- > 0));

   // Return whether or not this thread owns the optex
   return(fThisThreadOwnsTheOptex);
}


//////////////////////////////////////////////////////////////////////////////


void COptex::Leave() {

#ifdef _DEBUG
   // Only the owning thread can leave the optex
   if (m_psi->m_dwThreadId != GetCurrentThreadId())
      DebugBreak();
#endif

   // Reduce this thread's ownership of the optex
   if (--m_psi->m_lRecurseCount > 0) {
      // We still own the optex
      InterlockedDecrement(&m_psi->m_lLockCount);

   } else {

      // We don't own the optex anymore
      m_psi->m_dwThreadId = 0;

      if (InterlockedDecrement(&m_psi->m_lLockCount) > 0) {

         // Other threads are waiting, the auto-reset event wakes one of them
         SetEvent(m_hevt);
      }
   }
}


//////////////////////////////// End of File //////////////////////////////////
```

Optex.h

```
/******************************************************************************
Module name: Optex.h
Written by:  Jeffrey Richter
******************************************************************************/


#pragma once


//////////////////////////////////////////////////////////////////////////////


class COptex {
public:
   COptex(DWORD dwSpinCount = 4000);
   COptex(PCSTR  pszName, DWORD dwSpinCount = 4000);
   COptex(PCWSTR pszName, DWORD dwSpinCount = 4000);
   ~COptex();

   void SetSpinCount(DWORD dwSpinCount);
   void Enter();
   BOOL TryEnter();
   void Leave();
   BOOL IsSingleProcessOptex() const;

private:
   typedef struct {
      DWORD m_dwSpinCount;
      long  m_lLockCount;
      DWORD m_dwThreadId;
      long  m_lRecurseCount;
   } SHAREDINFO, *PSHAREDINFO;

   HANDLE      m_hevt;
   HANDLE      m_hfm;
   PSHAREDINFO m_psi;

private:
   static BOOL sm_fUniprocessorHost;

private:
   void CommonConstructor(DWORD dwSpinCount, BOOL fUnicode, PVOID pszName);
   PSTR ConstructObjectName(PSTR pszResult,
      PCSTR pszPrefix, BOOL fUnicode, PVOID pszName);
};


//////////////////////////////////////////////////////////////////////////////

inline COptex::COptex(DWORD dwSpinCount) {

   CommonConstructor(dwSpinCount, FALSE, NULL);
}

//////////////////////////////////////////////////////////////////////////////
```

```
inline COptex::COptex(PCSTR pszName, DWORD dwSpinCount) {

   CommonConstructor(dwSpinCount, FALSE, (PVOID) pszName);
}


//////////////////////////////////////////////////////////////////////////////
inline COptex::COptex(PCWSTR pszName, DWORD dwSpinCount) {

   CommonConstructor(dwSpinCount, TRUE, (PVOID) pszName);
}


//////////////////////////////////////////////////////////////////////////////


inline COptex::IsSingleProcessOptex() const {

   return(m_hfm == NULL);
}


//////////////////////////////// End of File /////////////////////////////////
```

Optex.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

//////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

//////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32
//////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
```

```
// remains consistent on all systems.
IDI_OPTEX               ICON    DISCARDABLE     "Optex.ICO"

#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED

#endif    // English (U.S.) resources
//////////////////////////////////////////////////////////////////////////////



#ifndef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//

//////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

OptexTest.cpp

```
/******************************************************************************
Module name: OptexTest.cpp
Written by:  Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <tchar.h>
#include <process.h>
#include "Optex.h"
```

```
//////////////////////////////////////////////////////////////////////////////

DWORD WINAPI SecondFunc(PVOID pvParam) {

   COptex& optex = * (COptex*) pvParam;

   // The primary thread should own the optex here, this should fail
   chVERIFY(optex.TryEnter() == FALSE);

   // Wait for the primary thread to give up the optex
   optex.Enter();

   optex.Enter();  // Test recursive ownership
   chMB("Secondary: Entered the optex\n(Dismiss me 2nd)");

   // Leave the optex but we still own it once
   optex.Leave();
   chMB("Secondary: The primary thread should not display a box yet");
   optex.Leave();  // The primary thread should be able to run now

   return(0);
}

//////////////////////////////////////////////////////////////////////////////
VOID FirstFunc(BOOL fLocal, COptex& optex) {

   optex.Enter();  // Gain ownership of the optex

   // Since this thread owns the optex, we should be able to get it again
   chVERIFY(optex.TryEnter());

   HANDLE hOtherThread = NULL;
   if (fLocal) {
      // Spawn a secondary thread for testing purposes (pass it the optex)

      DWORD dwThreadId;
      hOtherThread = chBEGINTHREADEX(NULL, 0,
         SecondFunc, (PVOID) &optex, 0, &dwThreadId);

   } else {
      // Spawn a secondary process for testing purposes
      STARTUPINFO si = { sizeof(si) };
      PROCESS_INFORMATION pi;
      TCHAR szPath[MAX_PATH];
      GetModuleFileName(NULL, szPath, chDIMOF(szPath));
      CreateProcess(NULL, szPath, NULL, NULL,
         FALSE, 0, NULL, NULL, &si, &pi);
      hOtherThread = pi.hProcess;
      CloseHandle(pi.hThread);
   }

   // Wait for the second thread to own the optex
   chMB("Primary: Hit OK to give optex to secondary");
```

```
   // Let the second thread gain ownership of the optex
   optex.Leave();
   optex.Leave();

   // Wait for the second thread to own the optex
   chMB("Primary: Hit OK to wait for the optex\n(Dismiss me 1st)");

   optex.Enter();  // Try to gain ownership back

   WaitForSingleObject(hOtherThread, INFINITE);
   CloseHandle(hOtherThread);
   optex.Leave();
}

//////////////////////////////////////////////////////////////////////////////


int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   // This event is just used to determine which instance this is.
   HANDLE hevt = CreateEvent(NULL, FALSE, FALSE, TEXT("OptexTest"));

   if (GetLastError() != ERROR_ALREADY_EXISTS) {

      // This is the first instance of this test application

      // First, let's test the single-process optex
      COptex optexSingle;   // Create a single-process optex
      FirstFunc(TRUE, optexSingle);

      // Now, let's test the cross-process optex
      COptex optexCross("CrossOptexTest");   // Create a cross-process optex
      FirstFunc(FALSE, optexCross);

   } else {

      // This is the second instance of this test application
      DebugBreak();  // Force debugger connection for tracing

      // Test the cross-process optex
      COptex optexCross("CrossOptexTest");   // Create a cross-process optex
      SecondFunc((PVOID) &optexCross);
   }

   CloseHandle(hevt);
   return(0);
}


//////////////////////////////// End of File ////////////////////////////////
```

10.2 创建线程安全的数据类型和反信标

有一天我正在编写一些代码，这时需要一个内核对象，这个对象的行为特性必须与信标对

象的行为特性相反。当它的当前资源数量变为0时，便通知该对象，当它的当前资源数量大于0时，就不向该对象发出通知。

我发现这种类型的对象有许多用途。例如，有一个线程，当将某个操作运行100次时，需要将该线程唤醒。为了实现这个要求，需要一个能够将它初始化为100的内核对象。当该内核对象的数量大于0时，该对象不应该被通知。每当执行某个操作时，必须递减该内核对象中的数量。当该数量降到0时，该对象应该得到通知，这样，其他线程就能够醒来，以便处理某些操作。这是个常见的问题，我不知道Windows为什么没有提供这个内置特性。

实际上，Microsoft只要让一个信标的当前资源数量变为负值，就能很容易解决这个问题。可以将该信标的数量初始化为-99，然后在执行每个操作后调用ReleaseSemaphore函数。当该信标的数量到达1时，该对象就得到通知，并且其他线程能够醒来执行它的操作。可惜Microsoft不允许信标的数量变为负值，在可以预见的将来他们不会修改这个代码。

本节将介绍一组C++模板类，它们具有反信标的行为特性和一整套其他特性。这些类的代码都在Interlocked.h文件中（参见后面的清单10-2）。

当着手解决这个问题的时候，我认识到这个解决方案的核心是需要一个进行变量操作时线程安全的方法。我想设计一个巧妙的解决方案，以便能够非常容易地编写引用变量的代码。显然，要使资源成为线程安全的资源，最容易的办法是用关键代码段来保护它。使用C++类，就能够非常容易地为数据对象赋予线程安全的特性。要做的工作只不过是创建一个C++类，它包含想要保护的变量和一个CRITICAL_SECTION数据结构。然后，在该构造函数中，调用InitializeCriticalSection，在析构函数中，调用DeleteCriticalSection。对于所有其他成员变量，可以调用EnterCriticalSection，然后对变量进行操作，调用LeaveCriticalSection。如果用这种方法实现一个C++类，就可以很容易编写能用线程安全方式访问数据结构的代码。这是本节介绍的所有C++类的基本原则（当然可以使用前一节中讲述的optex，而不使用关键代码段）。

第一个类是个资源保护类，称为CResGuard。它包含一个CRITICAL_SECTION数据成员和一个LONG数据成员。LONG数据成员用于跟踪线程进入关键代码段的次数。这个信息可以用于调试。CResGuard对象的构造函数和析构函数分别调用InitializeCriticalSection和DeleteCriticalSection。由于只有单个线程能够创建对象，因此C++对象的构造函数和析构函数不必是线程安全的函数。IsGuarded成员函数只是返回是否为该对象至少调用了一次EnterCriticalSection。如前所述，这是用于调试目的的。将CRITICAL_SECTION放入一个C++对象，可以确保关键代码段被正确地初始化和删除。

CResGuard类也提供了一个嵌套的公用C++类CGuard。CGuard对象包含了一个对CResGuard对象的引用，并且只提供一个构造函数和析构函数。构造函数调用CResGuard的Guard成员函数，而该成员函数则调用EnterCriticalSection。CGuard的析构函数调用CResGuard的Unguard成员函数，而该成员函数则调用LeaveCriticalSection。有这样的方法来建立这些类，就可以更加容易地操作CRITICAL_SECTION。下面是使用这些类的一个小代码段：

```
struct SomeDataStruct {
   ...
} g_SomeSharedData;

// Create a CResGuard object that protects g_SomeSharedData.
// Note: The constructor initializes the critical section and
//       the destructor deletes the critical section.
CResGuard g_rgSomeSharedData;
```

```
void AFunction() {

   // This function touches the shared data.

   // Protect the resource from being accessed from multiple threads.
   CResGuard::CGuard gDummy(g_rgSomeSharedData); //Enter critical section
   // Touch the g_SomeSharedData resource.
   ...

}  // Note: LeaveCriticalSection is called when gDummy goes out of scope.
```

下一个C++类CInterlockedType包含了创建线程安全的数据对象时必要的全部元素。我将CInterlockedType类做成了一个模板类，这样，它就可以用于使任何数据类型成为线程安全的数据类型。例如，可以使用这个类建立一个线程安全的整数，一个线程安全的字符串 或一个线程安全的数据结构。

CInterlockedType对象的每个实例都包含两个数据成员。第一个数据成员是你想要使之成为线程安全的模板数据类型的实例。该数据成员是私有的，只能使用 CInterlockedType的成员函数进行操作。第二个数据成员是CResGuard对象的一个实例，它负责保护对数据成员的访问。CResGuard对象是个受保护的数据成员，因此，由CInterlockedType派生的类能够很容易地保护它的数据。

总是应该将 CInterlockedType类用作一个基类，以便派生一个新类。前面说过，CInterlockedType类提供了创建线程安全的对象时需要的所有元素，但是该派生类负责使用这些元素，以便用线程安全的方式来正确地访问数据值。

CInterlockedType类只提供了4个公有函数，一个是不对数据值进行初始化的构造函数，一个是对数据值进行初始化的构造函数，一个是什么也不做的虚拟析构函数，还有一个是类型转换操作符函数。类型转换操作符函数只是负责保证对数据值的线程安全访问，方法是保护资源并返回对象的当前值（当局部变量x超出规定的范围时，资源将自动失去保护）。类型转换操作符函数使得你能够更加容易地观察线程安全方式中的这个类包含的数据对象的值。

CInterlockedType类还提供了3个派生类将要调用的非虚拟保护的函数。两个 GetVal函数返回数据对象的当前值。在文件的调试代码中，这两个函数首先要查看数据对象是否得到了保护。如果该对象没有被保护，GetVal可以为该对象返回一个值，然后在原始调用线程查看该值之前，允许另一个线程修改该对象的值。

我假设调用线程获得了对象的值，因此它能够以某种方法来修改该值。根据这个假设，GetVal函数要求调用线程拥有对数据值的受保护的访问权。如果 GetVal函数确定该数据受到了保护，那么就返回数据的当前值。这两个 GetVal函数是相同的，不同之处在于其中一个是在对象的固定版本上运行的。这两个版本允许编写可以对常量数据类型和非常量数据类型进行操作的代码而不需要编译器生成警告信息。

第三个非虚拟受保护成员函数是 SetVal。当一个派生类的成员函数想要修改数据值时，该派生类的函数应该保护对该数据的访问权，然后调用 SetVal函数。与GetVal函数一样，SetVal函数首先要进行调试检查，以确保派生类的代码不会忘记对数据值访问权的保护。然后， SetVal要查看数据值是否真的正在被修改。如果正在被修改， SetVal就将老的值保存起来，将对象改为它的新值，然后调用一个虚拟的、受保护的成员函数 OnValChanged，并将老的和新的数据值传递给它。CInterlockedType类实现了一个OnValChanged成员函数，但是该函数并不做任何工作。可以使用OnValChanged成员函数将一些强大的功能添加给该派生类，就像下面我们介

绍CWhenZero类时的情况那样。

到现在为止，已经介绍了许多抽象类和它们的概念。下面来说明如何使用所有这些结构。首先介绍CInterlockedScalar类，这是由CInterlockedType派生的一个模板类。可以使用这个类创建线程安全的标量数据类型，如字节、字符、 16位整数、32位整数、64位整数和浮点值等。由于CInterlockedScalar类是CInterlockedType类派生而来的，因此它并不拥有它自己的数据类型。CInterlockedScalar的构造函数只是调用 CInterlockedType的构造函数，为该标量传递一个初始值。由于CInterlockedScalar类总是使用数字值，因此将默认构造函数的参数设置为 0，这样，对象总是在已知状态中创建。 CInterlockedScalar的构造函数根本就不进行任何操作。

CInterlockedScalar的其余成员函数全部用于修改标量值。可以用于对标量值进行的每个操作都有一个成员函数。为了使该CInterlockedScalar类能够用线程安全的方式对它的数据对象进行操作，所有这些成员函数都在操作前对数据值进行了保护。这些成员函数很简单，因此不对它们进行详细的说明。可以观察其代码，以便了解它们能够做些什么。不过要介绍一下如何使用这些类。下面的代码声明了线程安全的BYTE数据类型，并可以对它进行操作：

```
CInterlockedScalar<BYTE> b = 5;   // A thread-safe BYTE
BYTE b2 = 10;                     // A non-thread-safe BYTE
b2 = b++;                         // b2=5, b=6
b *= 4;                           // b=24
b2 = b;                           // b2=24, b=24
b += b;                           // b=48
b %= 2;                           // b=0
```

对线程安全的标量值进行操作就像对非线程安全的标量值的操作一样简单。实际上，由于C++的操作符被重载，它们的代码是相同的。运用迄今为止介绍的 C++类，可以很容易地将非线程安全的变量转换成线程安全的变量，只需要对源代码做很少的修改即可。

当我开始设计所有这些类的时候，我的脑子里就已经有了一个特定的目的：想要创建一个对象，它的行为特性与信标的行为特性正好相反。提供这个行为特性的C++类是CWhenZero类。CWhenZero类是由CInterlockedScalar类派生而来的。当标量值是0时，便通知CWhenZero对象。当该数据值不是0时，便不通知CWhenZero对象。这与信标的行为特性正好相反。

如你所知，C++对象不能被通知，只有内核对象才能被通知，并且可以用于线程的同步。因此，CWhenZero对象必须包含某些别的数据成员，这些成员是事件内核对象的句柄。一个CWhenZero对象包含两个数据成员，一个是 m_hevtZero，这是当数据值是0时得到通知的事件内核对象的句柄，另一个成员是m_hevtNotZero，这是当数据值不是0时得到通知的事件内核对象的句柄。

CWhenZero的构造函数接受数据对象的初始值，并且也让你设定这两个事件内核对象是人工重置的对象（默认值）还是自动重置的对象。然后该构造函数调用 CreateEvent，创建两个事件内核对象，并且根据数据的初始值是 0还是非0，将它们设置为已通知状态或未通知状态。CWhenZero的析构函数仅仅负责关闭两个事件的句柄。由于 CWhenZero的类公开继承了CInterlockedScalar类，因此CWhenZero对象的用户可以使用重载操作符函数的所有成员函数。

还记得OnValChanged能够保护CInterlockedType类中声明的成员函数吗？CWhenZero类重载了该虚拟函数。该函数负责根据数据对象的值，使事件内核对象保持已通知状态或者未通知状态，每当数据值变更时，便调用 OnValChanged函数。CWhenZero对该函数的实现代码查看新的值是否是0。如果是0，它就设置m_hevtZero事件，并且重置m_hevtNotZero事件。如果新值不是0，OnValChanged就不反转。

现在，当想要使线程等待数据值变为0时，只需要编写下面的代码：

```
CWhenZero<BYTE> b = 0;              // A thread-safe BYTE

// Returns immediately because b is 0
WaitForSingleObject(b, INFINITE);

b = 5;

//Returns only if another thread sets b to 0
WaitForSingleObject(b, INFINITE);
```

可以像前面那样编写对WaitForSingleObject的调用代码，因为CWhenZero类也包含一个类型转换操作符成员函数，它能够将CWhenZero对象转换成一个内核对象句柄。换句话说，如果将一个CWhenZero C++对象传递给期望一个HANDLE对象的函数，该类型转换操作符函数就会被调用，它的返回值被传递给该函数。 CWhenZero的HANDLE类型转换操作符函数返回了m_hevtNotZero事件内核对象的句柄。

CWhenZero类中的m_hevtNotZero事件句柄使你能够编写等待数据值不是0的代码。不幸的是，我已经拥有一个HANDLE类型转换操作符函数，因此不能拥有另一个返回 m_hevtNotZero句柄的函数。为了获得该句柄，我添加了 GetNotZeroHandle成员函数。使用该函数，可以编写下面的代码：

```
CWhenZero<BYTE> b = 5;      // A thread-safe BYTE

// Returns immediately because b is not 0
WaitForSingleObject(b.GetNotZeroHandle(), INFINITE);

b = 0;

// Returns only if another thread sets b to not 0
WaitForSingleObject(b.GetNotZeroHandle(), INFINITE);
```

InterlockedType示例应用程序

清单10-2所示的InterlockedType（“10 InterlockedType.exe”）应用程序用于测试我刚刚介绍的C++类。该应用程序的源代码和资源文件在本书所附光盘上的10- InterlockedType目录下。我总是在调试程序中运行该应用程序，因此能够清楚地观察所有的类成员函数和变量的变化。

该代码显示一种常见的编程环境，这个环境是，一个线程产生若干个工作线程，并对内存块进行初始化。然后主线程唤醒工作线程，这样它们就能够开始处理该内存块。这时，主线程必须终止自己的运行，直到所有工作线程运行完成。然后主线程用新数据重新对内存块进行初始化，并唤醒工作线程，以便重新启动整个进程。

通过观察这个代码，可以看到，要用便于阅读和维护的 C++代码来解决这个常见的编程问题是多么烦琐。如你所见，CWhenZero类为我们提供的行为特性大大超过了信标的相反行为特性。现在我们有一个线程安全的数字，当它的值是 0 时，它就被通知。虽然你可以递增或递减信标的值，但是你也可以用CWhenZero对象对该值进行加、减、乘、除、求模，并且将它设置为任何值，甚至运行位操作。CWhenZero对象的功能实际上比信标内核对象的功能大得多。

了解这些C++模板类的概念是非常有用的。例如，可以创建一个由 CInterlockedType类派生的CInterlockedString类。可以使用CInterlockedString类以线程安全的方式对字符串进行操作。然后可以用CInterlockedString类派生一个CWhenCertainString类，当字符串变成某个值或某几

个值时，它就能通知一个事件内核对象。这种可能性是无限的。

清单10-2 InterlockedType示例应用程序

IntLockTest.cpp

```
/******************************************************************************
Module:  IntLockTest.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <tchar.h>
#include "Interlocked.h"


//////////////////////////////////////////////////////////////////////////////


// Set to TRUE when worker threads should terminate cleanly.
volatile BOOL g_fQuit = FALSE;


//////////////////////////////////////////////////////////////////////////////


DWORD WINAPI WorkerThread(PVOID pvParam) {

   CWhenZero<BYTE>& bVal = * (CWhenZero<BYTE> *) pvParam;

   // Should worker thread terminate?
   while (!g_fQuit) {

      // Wait for something to do
      WaitForSingleObject(bVal.GetNotZeroHandle(), INFINITE);

      // If we should quit, quit
      if (g_fQuit)
         continue;

      // Do something
      chMB("Worker thread: We have something to do");

      bVal--;     // We're done

      // Wait for all worker threads to stop
      WaitForSingleObject(bVal, INFINITE);
   }

   chMB("Worker thread: terminating");
   return(0);
}
```

```
//////////////////////////////////////////////////////////////////////////////


int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   // Initialize to indicate that NO worker threads have anything to do
   CWhenZero<BYTE> bVal = 0;

   // Create the worker threads
   const int nMaxThreads = 2;
   HANDLE hThreads[nMaxThreads];
   for (int nThread = 0; nThread < nMaxThreads; nThread++) {
      DWORD dwThreadId;
      hThreads[nThread] = CreateThread(NULL, 0,
         WorkerThread, (PVOID) &bVal, 0, &dwThreadId);
   }

   int n;
   do {
      // Do more work or stop running?
      n = MessageBox(NULL,
         TEXT("Yes: Give worker threads something to do\nNo: Quit"),
         TEXT("Primary thread"), MB_YESNO);

      // Tell worker threads that we're quitting
      if (n == IDNO)
      InterlockedExchangePointer((PVOID*) &g_fQuit, (PVOID) TRUE);

   bVal = nMaxThreads;  // Wake the worker threads

   if (n == IDYES) {

      // There is work to do, wait for the worker threads to finish
      WaitForSingleObject(bVal, INFINITE);
   }

} while (n == IDYES);

// There is no more work to do, the process wants to die.
// Wait for the worker threads to terminate
WaitForMultipleObjects(nMaxThreads, hThreads, TRUE, INFINITE);

// Close the worker thread handles.
for (nThread = 0; nThread < nMaxThreads; nThread++)
   CloseHandle(hThreads[nThread]);

// Tell the user that the process is dying
chMB("Primary thread: terminating");

return(0);
}

//////////////////////////////// End of File //////////////////////////////////
```

Interlocked.h

```
/******************************************************************************
Module:  Interlocked.h
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#pragma once

////////////////////////////////////////////////////////////////////////////////

// Instances of this class will be accessed by multiple threads. So,
// all members of this class (except the constructor and destructor)
// must be thread-safe.
class CResGuard {
public:
   CResGuard()  { m_lGrdCnt = 0; InitializeCriticalSection(&m_cs); }
   ~CResGuard() { DeleteCriticalSection(&m_cs); }

   // IsGuarded is used for debugging
   BOOL IsGuarded() const { return(m_lGrdCnt > 0); }

public:
   class CGuard {
   public:
      CGuard(CResGuard& rg) : m_rg(rg) { m_rg.Guard(); };
      ~CGuard() { m_rg.Unguard(); }

   private:
      CResGuard& m_rg;
   };

private:
   void Guard()   { EnterCriticalSection(&m_cs); m_lGrdCnt++; }
   void Unguard() { m_lGrdCnt--; LeaveCriticalSection(&m_cs); }

   // Guard/Unguard can only be accessed by the nested CGuard class.
   friend class CResGuard::CGuard;

private:
   CRITICAL_SECTION m_cs;
   long m_lGrdCnt;   // # of EnterCriticalSection calls
};


////////////////////////////////////////////////////////////////////////////////


// Instances of this class will be accessed by multiple threads. So,
// all members of this class (except the constructor and destructor)
// must be thread-safe.
template <class TYPE>
class CInterlockedType {

public:     // Public member functions
```

```
   // Note: Constructors & destructors are always thread-safe
   CInterlockedType() { }
   CInterlockedType(const TYPE& TVal) { m_TVal = TVal; }
   virtual ~CInterlockedType()  { }

   // Cast operator to make writing code that uses
   // thread-safe data type easier
   operator TYPE() const {
      CResGuard::CGuard x(m_rg);
      return(GetVal());
   }

protected:  // Protected function to be called by derived class
   TYPE& GetVal() {
      chASSERT(m_rg.IsGuarded());
      return(m_TVal);
   }

   const TYPE& GetVal() const {
      assert(m_rg.IsGuarded());
      return(m_TVal);
   }

   TYPE SetVal(const TYPE& TNewVal) {
      chASSERT(m_rg.IsGuarded());
      TYPE& TVal = GetVal();
      if (TVal != TNewVal) {
         TYPE TPrevVal = TVal;
         TVal = TNewVal;
         OnValChanged(TNewVal, TPrevVal);
      }
      return(TVal);
   }

protected:  // Overridable functions
   virtual void OnValChanged(
      const TYPE& TNewVal, const TYPE& TPrevVal) const {
      // Nothing to do here
   }

protected:
   // Protected guard for use by derived class functions
      mutable CResGuard m_rg;

private:    // Private data members
   TYPE m_TVal;
};

///////////////////////////////////////////////////////////////////////////////

// Instances of this class will be accessed by multiple threads. So,
// all members of this class (except the constructor and destructor)
// must be thread-safe.
template <class TYPE>
class CInterlockedScalar : protected CInterlockedType<TYPE> {

public:
```

```
    CInterlockedScalar(TYPE TVal = 0) : CInterlockedType<TYPE>(TVal) {
    }

    ~CInterlockedScalar() { /* Nothing to do */ }

    // C++ does not allow operator cast to be inherited.
    operator TYPE() const {
       return(CInterlockedType<TYPE>::operator TYPE());
    }

    TYPE operator=(TYPE TVal) {
       CResGuard::CGuard x(m_rg);
       return(SetVal(TVal));
    }

    TYPE operator++(int) {    // Postfix increment operator
       CResGuard::CGuard x(m_rg);
       TYPE TPrevVal = GetVal();
       SetVal((TYPE) (TPrevVal + 1));
       return(TPrevVal);      // Return value BEFORE increment
    }

    TYPE operator--(int) {    // Postfix decrement operator.
       CResGuard::CGuard x(m_rg);
       TYPE TPrevVal = GetVal();
       SetVal((TYPE) (TPrevVal - 1));
       return(TPrevVal);      // Return value BEFORE decrement
    }

    TYPE operator += (TYPE op)
       { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() +  op)); }
    TYPE operator++()
       { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() +   1)); }
    TYPE operator -= (TYPE op)
       { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() -  op)); }
    TYPE operator--()
       { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() -   1)); }
    TYPE operator *= (TYPE op)
       { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() *  op)); }
    TYPE operator /= (TYPE op)
       { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() /  op)); }
    TYPE operator %= (TYPE op)
       { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() %  op)); }
    TYPE operator ^= (TYPE op)
       { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() ^  op)); }
    TYPE operator &= (TYPE op)
       { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() &  op)); }
    TYPE operator |= (TYPE op)
       { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() |  op)); }
    TYPE operator <<=(TYPE op)
       { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() << op)); }
    TYPE operator >>=(TYPE op)
       { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() >> op)); }
};
```

```
//////////////////////////////////////////////////////////////////////////////


// Instances of this class will be accessed by multiple threads. So,
// all members of this class (except the constructor and destructor)
// must be thread-safe.
template <class TYPE>
class CWhenZero : public CInterlockedScalar<TYPE> {
public:
   CWhenZero(TYPE TVal = 0, BOOL fManualReset = TRUE)
      : CInterlockedScalar<TYPE>(TVal) {

      // The event should be signaled if TVal is 0
      m_hevtZero = CreateEvent(NULL, fManualReset, (TVal == 0), NULL);

      // The event should be signaled if TVal is NOT 0
      m_hevtNotZero = CreateEvent(NULL, fManualReset, (TVal != 0), NULL);
   }

   ~CWhenZero() {
      CloseHandle(m_hevtZero);
      CloseHandle(m_hevtNotZero);
   }
   // C++ does not allow operator= to be inherited.
   TYPE operator=(TYPE x) {
      return(CInterlockedScalar<TYPE>::operator=(x));
   }

   // Return handle to event signaled when value is zero
   operator HANDLE() const { return(m_hevtZero); }

   // Return handle to event signaled when value is not zero
   HANDLE GetNotZeroHandle() const { return(m_hevtNotZero); }

   // C++ does not allow operator cast to be inherited.
   operator TYPE() const {
      return(CInterlockedScalar<TYPE>::operator TYPE());
   }

protected:
   void OnValChanged(const TYPE& TNewVal, const TYPE& TPrevVal) const {
      // For best performance, avoid jumping to
      // kernel mode if we don't have to
      if ((TNewVal == 0) && (TPrevVal != 0)) {
         SetEvent(m_hevtZero);
         ResetEvent(m_hevtNotZero);
      }
      if ((TNewVal != 0) && (TPrevVal == 0)) {
         ResetEvent(m_hevtZero);
         SetEvent(m_hevtNotZero);
      }
   }

private:
   HANDLE m_hevtZero;      // Signaled when data value is 0
```

```
    HANDLE m_hevtNotZero;   // Signaled when data value is not 0
};

//////////////////////////////// End of File /////////////////////////////////
```

InterlockedType.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

//////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

//////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

//////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_INTERLOCKEDTYPE     ICON    DISCARDABLE     "InterLockedType.ICO"

#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END
```

```
3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED

#endif    // English (U.S.) resources
/////////////////////////////////////////////////////////////////////////////


#ifndef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


/////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

10.3 单个写入程序/多个阅读程序的保护

许多应用程序存在一个基本的同步问题，这个问题称为单个写入程序 /多个阅读程序环境。该问题涉及到试图访问共享资源的任意数量的线程。这些线程中，有些线程（写入程序）需要修改数据的内容，而有些线程（阅读程序）则需要读取数据。由于下面 4个原则，它们之间的同步是必要的：

1) 当一个线程正在写入数据时，其他任何线程不能写入数据。

2) 当一个线程正在写入数据时，其他任何线程不能读取数据。

3) 当一个线程正在读取数据时，其他任何线程不能写入数据。

4) 当一个线程正在读取数据时，其他线程也能够读取数据。

让我们观察一下数据库应用程序环境中的这个问题。比如说，有 5个最终用户，他们都要访问同一个数据库。两个员工将一些记录输入该数据库，3个员工则从该数据库中检索记录。

在这个环境中，必须使用原则 1，因为我们不能让员工 1和员工2同时更新同一个记录。如果两个员工都试图修改同一个记录，那么员工 1的修改和员工2的修改就会在同一时间内进行，该记录中的信息就会被破坏。

原则2用于在某个员工更新数据库中的记录时禁止另一个员工访问该记录。如果不能防止这种情况的发生，那么员工 4就能够在员工1修改一个记录时读取该记录的内容。当员工 4的计算机显示该记录时，该记录将包含某些老的信息和某些更新过的信息，这当然是不行的。原则3可以用来解决同样的问题。无论谁首先拥有对数据库记录的访问权，即无论是试图写入的员工还是试图读取的员工，原则2和原则3用词上的差别都可以防止出现上面所说的情况。

原则4用于解决性能上的问题。如果没有员工试图修改数据库中的记录，那么数据库中的内容就不会变更，因此凡是只想从数据库中检索记录的员工都应该被允许进行检索。它的另一个前提是阅读者多于写入者。

好了，你已经掌握了问题的要领。现在的问题是，如何来解决这个问题。

注意 这里介绍的代码是新编写的。以前我提出的对这个问题的解决办法受到了一些人的批评，原因有二。首先，我以前编写的代码运行速度太慢，因为它们的设计初衷是要在许多环境中都能运行。例如，它们使用了较多的内核对象，因此不同进程中的线程能够同步它们对数据库的访问。当然，我的实现代码仍然可以在单进程环境中运行，但是，大量使用内核对象后，当所有线程都在单进程中运行时，就会增加很大的开销。必须承认，单进程的情况是更加常见的一种情况。

第二个批评是说，我的实现代码有可能完全把写入线程锁在外面。从前面介绍的几个原则来看，如果有许多阅读线程访问数据库，那么写入线程就永远无法访问该资源。

这里展示的实现代码解决了上面所说的两个问题。它尽可能避免使用内核对象，而是使用关键代码段，以便实现大多数同步问题。

为了简化操作，我将解决方案封装在一个C++类中，它称为CSWNRG，是“单个写入程序/多个阅读程序的保护”的英文缩写。SWMRG.h和SWMRG.cpp文件（参见后面的清单10-3）显示了实现代码。

CSWMRG的使用是再容易不过的了。只需要创建CSWMRG C++类的一个对象，然后按照应用程序的指令调用相应的成员函数。C++类中只有下面的3个方法（不包括构造函数和析构函数）：

```
VOID CSWMRG::WaitToRead();  // Call this to gain shared read access.
VOID CSWMRG::WaitToWrite(); // Call this to gain exclusive write access.
VOID CSWMRG::Done();        // Call this when done accessing the resource.
```

在执行读取共享资源的代码之前，调用第一个方法WaitToRead。在执行读取或写入共享资源的代码之前，调用第二个方法WaitToWrite。当代码不再访问共享资源时，调用第三个方法Done。这不是很简单吗？

一般来说，CSWMRG对象包含许多成员变量，这些变量反映了线程访问共享资源时的状态。表10-2描述了每个成员变量的作用，并且综合说明了它的整个工作情况。详细信息参见源代码。

表10-2 CSWMRG对象中成员变量的作用

成 员	描 述
m_cs	用于保护所有的其他成员变量，这样，对它们的操作就能够以原子操作方式来完成
m_nActive	用于反映共享资源的当前状态。如果该值是0，那么没有线程在访问资源。如果该值大于0，这个值用于表示当前读取该资源的线程的数量。如果这个数量是负值，那么写入程序正在将数据写入该资源。唯一有效的负值是-1
m_nWaitingReaders	表示想要访问资源的阅读线程的数量。该值被初始化为0，当m_nActive是-1时，每当线程调用一次WaitToRead，该值就递增1
m_nWaitingWriters	表示想要访问资源的写入线程的数量。该值被初始化为0，当m_nActive大于0时，每当线程调用一次WaitToWrite，该值就递增1
m_hsemWriters	当线程调用WaitToWrite，但是由于m_nActive大于0而被拒绝访问时，所有写入线程均等待该信标。当一个线程正在等待时，新阅读线程将被拒绝访问该资源。这可以防止阅读线程垄断该资源。当最后一个拥有资源访问权的阅读线程调用Done时，该信标就被释放，其数量是1，从而唤醒一个正在等待的写入线程
m_hsemReaders	当许多线程调用WaitToRead，但是由于m_nActive是-1而被拒绝访问时，所有阅读线程均等待该信标。当最后一个正在等待的阅读线程调用Done时，该信标被释放，其数量是m_nWaitingReaders，从而唤醒所有正在等待的阅读线程

SWMRG示例应用程序

清单10-3列出的SWMRG应用程序（“10 SWMRG.exe”）用于测试CSWMRG C++类。该应用程序的源代码和资源文件均存放在本书所附光盘上的10 SWMRG目录下。我总是在调试程序中运行该应用程序，这样，就能清楚地观察所有的类成员函数和变量的变化。

当运行该应用程序时，主线程就会产生若干个线程，它们全部运行相同的线程函数。然后，主线程通过调用 WaitForMultipleObjects函数，等待所有这些线程终止运行，当所有的线程终止运行后，它们的句柄被关闭，该进程退出。

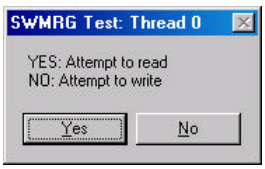

图10-1 辅助线程显示的消息

每个辅助线程均显示一条类似图10-1所示的消息。

如果想要该线程模拟阅读该资源，单击 Yes按钮。如果想要该线程模拟写入该资源，单击 No。这些操作只是使线程分别调用 SWMRG对象的WaitToRead或WaitToWrite函数。

当调用两个函数中的一个之后，该线程就显示另一个消息框，类似图 10-2或图10-3所示的消息框。

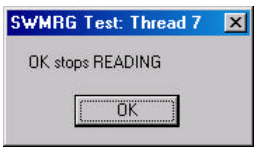

图10-2 调用SWMRG对象的WaitToRead后显示的消息框

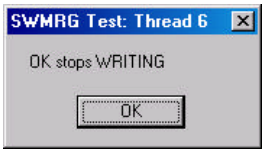

图10-3 调用SWMRG对象的WaitToWrite后显示的消息框

该消息框将暂停线程的运行，暂停的时间相当于拥有资源访问权的线程对该资源进行操作所用的时间。

当然，如果一个线程当前正在读取资源，而你又命令另一个线程写入该资源，那么写入线程的消息框就不会出现，因为该线程正在它对 WaitToWrite的调用中等待。同样，如果命令一个线程读取资源，而另一个线程的消息框已经显示，那么想要读取资源的线程就会在对WaitToRead的调用中暂停运行，它的消息框将不会出现，直到任何一个或所有写入线程完成它们对资源的模拟访问为止。

当单击OK按钮，以退出任何一个消息框时，拥有对资源访问权的线程就调用 Done。而SWMRG对象便终止任何正在等待的线程的运行。

清单10-3 SWMRG示例应用程序

SWMRG.cpp

```
/*****************************************************************************
Module:  SWMRG.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
*****************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include "SWMRG.h"
```

```
//////////////////////////////////////////////////////////////////////////////
CSWMRG::CSWMRG() {

   // Initially no readers want access, no writers want access, and
   // no threads are accessing the resource
   m_nWaitingReaders = m_nWaitingWriters = m_nActive = 0;
   m_hsemReaders = CreateSemaphore(NULL, 0, MAXLONG, NULL);
   m_hsemWriters = CreateSemaphore(NULL, 0, MAXLONG, NULL);
   InitializeCriticalSection(&m_cs);
}


//////////////////////////////////////////////////////////////////////////////


CSWMRG::~CSWMRG() {

#ifdef _DEBUG
   // A SWMRG shouldn't be destroyed if any threads are using the resource
   if (m_nActive != 0)
      DebugBreak();
#endif

   m_nWaitingReaders = m_nWaitingWriters = m_nActive = 0;
   DeleteCriticalSection(&m_cs);
   CloseHandle(m_hsemReaders);
   CloseHandle(m_hsemWriters);
}


//////////////////////////////////////////////////////////////////////////////


VOID CSWMRG::WaitToRead() {

   // Ensure exclusive access to the member variables
   EnterCriticalSection(&m_cs);

   // Are there writers waiting or is a writer writing?
   BOOL fResourceWritePending = (m_nWaitingWriters || (m_nActive < 0));

   if (fResourceWritePending) {

      // This reader must wait, increment the count of waiting readers
      m_nWaitingReaders++;
  } else {

     // This reader can read, increment the count of active readers
     m_nActive++;
  }

  // Allow other threads to attempt reading/writing
  LeaveCriticalSection(&m_cs);

  if (fResourceWritePending) {
```

```
      // This thread must wait
      WaitForSingleObject(m_hsemReaders, INFINITE);
   }
}

//////////////////////////////////////////////////////////////////////////////

VOID CSWMRG::WaitToWrite() {

   // Ensure exclusive access to the member variables
   EnterCriticalSection(&m_cs);

   // Are there any threads accessing the resource?
   BOOL fResourceOwned = (m_nActive != 0);

   if (fResourceOwned) {

      // This writer must wait, increment the count of waiting writers
      m_nWaitingWriters++;
   } else {

      // This writer can write, decrement the count of active writers
      m_nActive = -1;
   }

   // Allow other threads to attempt reading/writing
   LeaveCriticalSection(&m_cs);

   if (fResourceOwned) {

      // This thread must wait
      WaitForSingleObject(m_hsemWriters, INFINITE);
   }
}

//////////////////////////////////////////////////////////////////////////////

VOID CSWMRG::Done() {

   // Ensure exclusive access to the member variables
   EnterCriticalSection(&m_cs);

   if (m_nActive > 0) {

      // Readers have control so a reader must be done
      m_nActive--;
   } else {

      // Writers have control so a writer must be done
      m_nActive++;
   }

   HANDLE hsem = NULL;  // Assume no threads are waiting
   LONG lCount = 1;     // Assume only 1 waiter wakes; always true for writers

   if (m_nActive == 0) {
```

```
      // No thread has access, who should wake up?
      // NOTE: It is possible that readers could never get access
      //       if there are always writers wanting to write

      if (m_nWaitingWriters > 0) {

         // Writers are waiting and they take priority over readers
         m_nActive = -1;          // A writer will get access
         m_nWaitingWriters--;     // One less writer will be waiting
         hsem = m_hsemWriters;    // Writers wait on this semaphore
         // NOTE: The semaphore will release only 1 writer thread

      } else if (m_nWaitingReaders > 0) {

         // Readers are waiting and no writers are waiting
         m_nActive = m_nWaitingReaders;   // All readers will get access
         m_nWaitingReaders = 0;           // No readers will be waiting
         hsem = m_hsemReaders;            // Readers wait on this semaphore
         lCount = m_nActive;              // Semaphore releases all readers
      } else {

         // There are no threads waiting at all; no semaphore gets released
      }
   }
   // Allow other threads to attempt reading/writing
   LeaveCriticalSection(&m_cs);

   if (hsem != NULL) {

      // Some threads are to be released
      ReleaseSemaphore(hsem, lCount, NULL);
   }
}

//////////////////////////////// End of File //////////////////////////////////
```

SWMRG.h

```
/******************************************************************************
Module:  SWMRG.h
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#pragma once

//////////////////////////////////////////////////////////////////////////////

class CSWMRG {
public:
   CSWMRG();                  // Constructor
   ~CSWMRG();                 // Destructor

   VOID WaitToRead();         // Call this to gain shared read access
   VOID WaitToWrite();        // Call this to gain exclusive write access
   VOID Done();               // Call this when done accessing the resource
```

```
private:
   CRITICAL_SECTION m_cs;    // Permits exclusive access to other members
   HANDLE m_hsemReaders;     // Readers wait on this if a writer has access
   HANDLE m_hsemWriters;     // Writers wait on this if a reader has access
   int    m_nWaitingReaders; // Number of readers waiting for access
   int    m_nWaitingWriters; // Number of writers waiting for access
   int    m_nActive;         // Number of threads currently with access
                             //   (0=no threads, >0=# of readers, -1=1 writer)
};

////////////////////////////////// End of File //////////////////////////////////
```

SWMRG.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

////////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

////////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

////////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_SWMRG               ICON    DISCARDABLE     "SWMRG.ico"

#ifdef APSTUDIO_INVOKED
////////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END
```

```
2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED

#endif    // English (U.S.) resources
/////////////////////////////////////////////////////////////////////////////



#ifndef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


/////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

SWMRGTest.cpp

```
/******************************************************************************
Module:  SWMRGTest.Cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <tchar.h>
#include <process.h>       // for _beginthreadex
#include "SWMRG.h"
//////////////////////////////////////////////////////////////////////////////


// Global Single-Writer/Multiple-Reader Guard synchronization object
CSWMRG g_swmrg;


//////////////////////////////////////////////////////////////////////////////


DWORD WINAPI Thread(PVOID pvParam) {

   TCHAR sz[50];
   wsprintf(sz, TEXT("SWMRG Test: Thread %d"), PtrToShort(pvParam));
   int n = MessageBox(NULL,
```

```
      TEXT("YES: Attempt to read\nNO: Attempt to write"), sz, MB_YESNO);

   // Attempt to read or write
   if (n == IDYES)
      g_swmrg.WaitToRead();
   else
      g_swmrg.WaitToWrite();

   MessageBox(NULL,
      (n == IDYES) ? TEXT("OK stops READING") : TEXT("OK stops WRITING"),
      sz, MB_OK);

   // Stop reading/writing
   g_swmrg.Done();
   return(0);
}


//////////////////////////////////////////////////////////////////////////


int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   // Spawn a bunch of threads that will attempt to read/write
   HANDLE hThreads[MAXIMUM_WAIT_OBJECTS];
   for (int nThreads = 0; nThreads < 8; nThreads++) {
      DWORD dwThreadId;
      hThreads[nThreads] =
         chBEGINTHREADEX(NULL, 0, Thread, (PVOID) (DWORD_PTR) nThreads,
            0, &dwThreadId);
   }
   // Wait for all the threads to exit
   WaitForMultipleObjects(nThreads, hThreads, TRUE, INFINITE);
   while (nThreads--)
      CloseHandle(hThreads[nThreads]);

   return(0);
}


//////////////////////////////// End of File //////////////////////////////////
```

10.4 实现一个WaitForMultipleExpressions函数

不久以前，我正在编写一个应用程序，我必须解决复杂的线程同步问题。 WaitForMultipleObjects**函数虽然能够让线程等待单个对象或多个对象，但是它不能满足我的需要。我需要的是一个能够表达更丰富的等待环境的函数。我有** 3**个内核对象，即一个进程，一个信标和一个事件。我需要一种方法，使我的线程能够等待进程和信标都得到通知，或者进程和事件都得到通知为止。**

通过创造性地运用现有的一些 Windows**函数，我创建了所需要的函数** WaitForMultipleExpressions**。我建立了下面这个函数原型：**

```
DWORD WINAPI WaitForMultipleExpressions(
   DWORD nExpObjects,
   CONST HANDLE* phExpObjects,
   DWORD dwMilliseconds);
```

若要调用该函数，首先必须指定一个HANDLE数组，并对该数组的所有项目进行初始化。nExpObjects参数用于指明phExpObjects参数指向的数组中的项目数量。该数组包含多组内核对象句柄，每组句柄之间用一个NULL句柄项分开，WaitForMultipleExpressions将单组句柄中的对象视为用AND组合起来的对象组，而各个句柄组则是用OR组合起来的句柄组。因此，调用WaitForMultipleExpressions函数就可以暂停调用线程的运行，直到单组句柄中的所有对象均已同时得到通知为止。

下面是个例子。假设使用表10-3中的4个内核对象。

表10-3 内核对象的句柄值

对 象	句 柄 值
线程	0x1111
信标	0x2222
事件	0x3333
进程	0x4444

像下面这样对句柄数组进行初始化，就可以命令WaitForMultipleObjects函数暂停调用线程的运行，直到线程与信标得到通知，或者信标与事件与进程得到通知，或者线程与进程得到通知。如表10-4所示。

表10-4 句柄数组

索 引	句 柄 值	组
0	0x1111（线程）	0
1	0x2222（信标）	
2	0x0000（OR）	
3	0x2222（信标）	1
4	0x3333（事件）	
5	0x4444（进程）	
6	0x0000（OR）	
7	0x1111（线程）	2
8	0x4444（进程）	

也许你还记得，不能调用WaitForMultipleObjects来传递超过64个（MAXIMUM_WAIT_OBJECTS）项目的句柄数组。运用WaitForMultipleExpressions，该句柄数组的项目可以大大超过64。然而你不得拥有64个以上的表达式，并且每个表达式可以包含63个以上的句柄。另外，如果将一个互斥对象的句柄传递给它，那么WaitForMultipleExpressions将不能正确运行。

表10-5显示了WaitForMultipleExpressions可能的返回值。如果一个表达式真的实现了，那么WaitForMultipleExpressions将返回基于WAIT_OBJECT_0的该表达式的索引。使用该例子，如果线程和进程对象变为已通知状态，WaitForMultipleExpressions就返回索引WAIT_OBJECT_0+2。

表10-5 WaitForMultipleExpressions 的返回值

返 回 值	描 述
WAIT_OBJECT_0至（WAIT_OBJECT_0+表达式－1的号码）	用于指明哪个表达式被选定了
WAIT_TIMEOUT	在指定的时间内没有选定表达式
WAIT_FAILED	产生一个错误。若要了解详细信息，调用 GetLastError。如果产生的错误是ERROR_TOO_MANY_SECRETS，那么意味着设定的表达式超过了 64个。如果产生的错误是ERROR_SECRET_TOO_LONG，则意味着至少有一个表达式设定的对象超过了63个。也可能返回别的错误代码（为了我自己的目的，我不得不使用这两个错误代码）

WaitForMultipleExpressions示例应用程序

清单10-4列出的WaitForMultipleExpressions应用程序（“10 WaitForMultExp.exe”）用于测试WaitForMultipleExpressions函数。该应用程序的源代码和资源文件均存放在本书所附光盘上10-WaitForMultExp目录下。当运行该应用程序时，就会出现图10-4所示的对话框。

如果不改变任何设置，并且单击Wait For Multiple Expressions按钮，就会出现图10-5所示对话框。

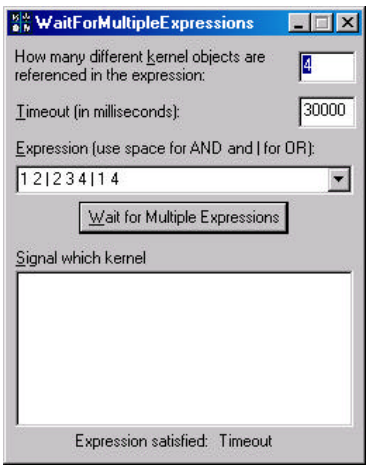

图10-4 WaitForMultiple Expressions 对话框

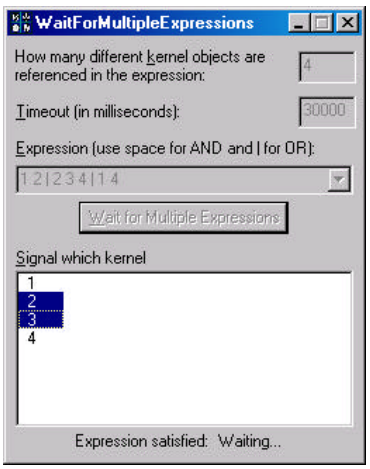

图10-5 单击WaitForMultipleExpressions按钮后出现的对话框

在内部，该应用程序创建了4个事件内核对象，开始时它们都处于未通知状态，同时，它为每个内核对象将一个项目放入这个多列、多节列表框中。然后，该应用程序对表达式的各个域进行分析，并创建句柄数组。我选择了与前面例子相吻合的4个内核对象和一个表达式。

由于我设定的超时为30000ms，因此有30s时间可以用来对列表框中的项目进行打开和关闭的切换，以便选定和取消有关的事件对象。如果选定了一个项目，则调用 SetEvent，给对象发送通知，如果取消一个项目，则调用 ResetEvent，使该事件取消已通知状态。当切换了足够的项目以便满足表达式中的某一个后， WaitForMultipleExpressions函数就返回，并在对话框的底部显示哪个表达式的条件得到了满足。如果在 30s内没有表达式的条件得到满足，则出现“Timeout”（超时）字样。

下面要介绍如何来实现WaitForMultipleExpressions函数。这是个不容易实现的函数，当使用这个函数的时候，必须注意某些开销问题。如你所知，Windows提供了WaitForMultipleObjects函数，它使得线程可以等待单个AND表达式：

```
DWORD WaitForMultipleObjects(
   DWORD dwObjects,
   CONST HANDLE* phObjects,
   BOOL fWaitAll,
   DWORD dwMilliseconds);
```

为了扩展该函数，使之包含使用OR的表达式，必须生成多个线程：每个OR表达式需要一个线程。这些线程中的每一个都使用WaitForMultipleObjectsEx来等待AND表达式（我使用WaitForMultipleObjectsEx，而不是使用更常用的WaitForMultipleObjects函数，其原因将在下面说明）。当其中的一个表达式被选定时，生成的线程中就有一个被唤醒并终止运行。

调用WaitForMultipleExpressions函数的线程（它与产生所有OR线程的线程相同）必须等待，直到其中的一个OR表达式得以实现。它是通过调用WaitForMultipleObjectsEx来实现的。生成的线程（OR表达式）的数量被传递给dwObjects参数，phObjects参数则指向一个数组，该数组包含生成的线程句柄的列表。对于fWaitAll参数，则传递FALSE，这样，一旦任何一个表达式得以实现，主线程就立即醒来。最后，传递给WaitForMultipleExpressions的dwTimeout值被传递给WaitForMultipleObjectsEx。

如果在规定的时间内没有任何表达式得以实现，WaitForMultipleObjectsEx就返回WAIT_TIMEOUT，而WaitForMultipleExpressions也返回WAIT_TIMEOUT。如果有一个表达式得以实现，那么WaitForMultipleExpressions就返回一个索引值，指明哪个线程已经终止运行。由于每个线程都是一个独立的表达式，因此该索引值也指明哪个表达式已经得以实现，并且WaitForMultipleExpressions返回了相同的索引值。

这就是WaitForMultipleExpressions函数如何运行的基本情况。但是还有3个较小的问题需要具体加以说明。首先，我们不希望多个OR线程在调用WaitForMultipleExpressions的时候同时醒来，因为成功地等待某个内核对象会导致该对象改变其状态。例如，等待一个信标会导致它的数量递减1。WaitForMultipleExpressions只等待一个表达式得以实现，因此必须防止对象多次改变它的状态。

对这个问题的解决方案实际上很简单。在生成OR线程之前，我创建了一个我自己的信标对象，其初始数量是1。然后，OR线程对WaitForMultipleObjectsEx的每次调用都包含该信标的句柄和表达式中的其他句柄。这说明每组句柄可以设定的句柄不得超过63个。为了使一个OR线程醒来，它等待的所有对象，包括我的私有信标对象，都必须得到通知。由于我给信标设定的初始数量是1，因此唤醒的OR线程不能超过1个，同时其他对象也不会不小心改变其状态。

第二个需要说明的具体问题是如何强制正在等待的线程停止等待，以便正确地撤消。添加信标可以确保醒来的线程不会超过1个，但是一旦我知道哪个表达式等待实现，我就必须强制剩余的线程醒来，这样，它们就能够明确地终止运行，释放它们的内存堆栈。应该始终避免调用TerminateThread函数，因此需要另一个机制。在思考了一会儿后，我想到，当一个项目进入异步过程调用（APC）队列时，如果等待线程处于待命状态，那么它们就会被强制唤醒。

我的WaitForMultipleExpressions实现代码使用QueueUserAPC来强制等待线程醒来。当主线程调用的WaitForMultipleObjects返回时，我将一个APC项放入每个还在等待的OR线程的队列：

```
// Break all the waiting expression threads out of their
// wait state so that they can terminate cleanly.
for (dwExpNum = 0; dwExpNum < dwNumExps; dwExpNum++) {

   if ((WAIT_TIMEOUT == dwWaitRet) ||
      (dwExpNum != (dwWaitRet - WAIT_OBJECT_0))) {

      QueueUserAPC(WFME_ExpressionAPC, ahThreads[dwExpNum], 0);
   }
}
```

回调函数WFME_ExpressionAPC之所以采用这种形式，原因是我实际上并没有什么事情要做，只是想要让线程停止等待。

```
// This is the APC callback routine function.
VOID WINAPI WFME_ExpressionAPC(DWORD dwData) {

   // This function intentionally left blank
}
```

第三个需要说明的具体问题是如何正确处理超时。如果线程在等待时没有任何表达式等待实现，那么主线程调用的WaitForMultipleObjects就返回一个WAIT_TIMEOUT值。如果出现这种情况，我就想要防止任何表达式得以实现，这可能导致对象改变它们的状态。下面这个代码能够做到这一点：

```
   // Wait for an expression to come TRUE or for a timeout.
   dwWaitRet = WaitForMultipleObjects(dwExpNum, ahThreads,
      FALSE, dwMilliseconds);

   if (WAIT_TIMEOUT == dwWaitRet) {
   // We timed out; check if any expressions were satisfied by
   // checking the state of the hsemOnlyOne semaphore.
   dwWaitRet = WaitForSingleObject(hsemOnlyOne, 0);

   if (WAIT_TIMEOUT == dwWaitRet) {

      // If the semaphore was not signaled, some thread expression
      // was satisfied; we need to determine which expression.
      dwWaitRet = WaitForMultipleObjects(dwExpNum,
         ahThreads, FALSE, INFINITE);

   } else {

      // No expression was satisfied and WaitForSingleObject just gave
      // us the semaphore, so we know that no expression can ever be
      // satisfied now -- waiting for an expression has timed out.
      dwWaitRet = WAIT_TIMEOUT;
   }
}
```

我用等待信标的办法来防止其他表达式的实现。这将使信标的数量递减为 0，同时所有的OR线程都不会醒来。但是，在主线程调用WaitForMultipleObjects和它调用WaitForSingleObject之后的某个位置上，一个表达式可能得以实现。这就是为什么要检查调用 WaitForSingleObject的返回值的原因。如果它返回WAIT_OBJECT_0，那么主线程得到了信标对象，并且没有一个

表达式得以实现。但是，如果返回WAIT_TIMEOUT，那么在主线程得到信标前，一个表达式就会得以实现。若要确定哪个表达式得到了实现，主线程要再次调用超时为INFINITE的WaitForMultipleObjects，这是可行的，因为我知道一个OR线程得到了信标，并且将要终止运行。这时，必须强制OR线程醒来，这样它们才能彻底退出。调用QueueUserAPC的循环能够进行这项操作。

由于WaitForMultipleExpressions是通过使用不同的线程来等待每组用AND连接起来的对象而在内部实现的，因此可以非常容易地了解为什么不能使用互斥对象。与其他内核对象不同，互斥对象可以被线程所拥有。因此，如果AND线程之一获得对互斥对象的所有权，那么当线程终止运行时，它就会放弃该互斥对象。如果Microsoft给Windows增加了一个函数，使得一个线程能够将互斥对象的所有权转交给另一个线程，那么WaitForMultipleExpressions函数就能够很容易地调整以便相应地支持互斥对象。在出现这个WaitForMultipleExpressions函数之前，一直没有支持互斥对象的出色方法。

清单10-4 WaitForMultipleExpressions示例应用程序

WaitForMultExp.cpp

```
/******************************************************************************
Module:  WaitForMultExp.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <malloc.h>
#include <process.h>
#include "WaitForMultExp.h"


//////////////////////////////////////////////////////////////////////////////


// Internal data structure representing a single expression.
// Used to tell OR-threads what objects to wait on.
typedef struct {
   PHANDLE m_phExpObjects;   // Points to set of handles
   DWORD   m_nExpObjects;    // Number of handles
} EXPRESSION, *PEXPRESSION;


//////////////////////////////////////////////////////////////////////////////


// The OR-thread function
DWORD WINAPI WFME_ThreadExpression(PVOID pvParam) {

   // This thread function just waits for an expression to come true.
   // The thread waits in an alertable state so that it can be forced
```

```
   // to stop waiting by queuing an entry to its APC queue.
   PEXPRESSION pExpression = (PEXPRESSION) pvParam;
   return(WaitForMultipleObjectsEx(
      pExpression->m_nExpObjects, pExpression->m_phExpObjects,
      TRUE, INFINITE, TRUE));
}


//////////////////////////////////////////////////////////////////////////////
// This is the APC callback routine function
VOID WINAPI WFME_ExpressionAPC(ULONG_PTR dwData) {

   // This function intentionally left blank
}


//////////////////////////////////////////////////////////////////////////////


// Function to wait on multiple Boolean expressions
DWORD WINAPI WaitForMultipleExpressions(DWORD nExpObjects,
   CONST HANDLE* phExpObjects, DWORD dwMilliseconds) {

   // Allocate a temporary array because we modify the passed array and
   // we need to add a handle at the end for the hsemOnlyOne semaphore.
   PHANDLE phExpObjectsTemp = (PHANDLE)
      _alloca(sizeof(HANDLE) * (nExpObjects + 1));
   CopyMemory(phExpObjectsTemp, phExpObjects, sizeof(HANDLE) * nExpObjects);
   phExpObjectsTemp[nExpObjects] = NULL;  // Put sentinel at end

   // Semaphore to guarantee that only one expression gets satisfied
   HANDLE hsemOnlyOne = CreateSemaphore(NULL, 1, 1, NULL);

   // Expression information: 1 per possible thread
   EXPRESSION Expression[MAXIMUM_WAIT_OBJECTS];

   DWORD dwExpNum  = 0;    // Current expression number
   DWORD dwNumExps = 0;    // Total number of expressions

   DWORD dwObjBegin = 0;   // First index of a set
   DWORD dwObjCur   = 0;   // Current index of object in a set

   DWORD dwThreadId, dwWaitRet = 0;

   // Array of thread handles for threads: 1 per expression
   HANDLE ahThreads[MAXIMUM_WAIT_OBJECTS];

   // Parse the callers handle list by initializing a structure for
   // each expression and adding hsemOnlyOne to each expression.
   while ((dwWaitRet != WAIT_FAILED) && (dwObjCur <= nExpObjects)) {

      // While no errors, and object handles are in the caller's list...
   // Find next expression (OR-expressions are separated by NULL handles)
   while (phExpObjectsTemp[dwObjCur] != NULL)
      dwObjCur++;
```

```
      // Initialize Expression structure which an OR-thread waits on
      phExpObjectsTemp[dwObjCur] = hsemOnlyOne;
      Expression[dwNumExps].m_phExpObjects =  &phExpObjectsTemp[dwObjBegin];
      Expression[dwNumExps].m_nExpObjects  =  dwObjCur - dwObjBegin + 1;

      if (Expression[dwNumExps].m_nExpObjects > MAXIMUM_WAIT_OBJECTS) {
         // Error: Too many handles in single expression
         dwWaitRet = WAIT_FAILED;
         SetLastError(ERROR_SECRET_TOO_LONG);
      }

      // Advance to the next expression
      dwObjBegin = ++dwObjCur;
      if (++dwNumExps == MAXIMUM_WAIT_OBJECTS) {
         // Error: Too many expressions
         dwWaitRet = WAIT_FAILED;
         SetLastError(ERROR_TOO_MANY_SECRETS);
      }
   }

   if (dwWaitRet != WAIT_FAILED) {

      // No errors occurred while parsing the handle list

      // Spawn thread to wait on each expression
      for (dwExpNum = 0; dwExpNum < dwNumExps; dwExpNum++) {

         ahThreads[dwExpNum] = chBEGINTHREADEX(NULL,
            1, // We only require a small stack
            WFME_ThreadExpression, &Expression[dwExpNum],
            0, &dwThreadId);
      }

      // Wait for an expression to come TRUE or for a timeout
      dwWaitRet = WaitForMultipleObjects(dwExpNum, ahThreads,
         FALSE, dwMilliseconds);

      if (WAIT_TIMEOUT == dwWaitRet) {

         // We timed-out, check if any expressions were satisfied by
         // checking the state of the hsemOnlyOne semaphore.
         dwWaitRet = WaitForSingleObject(hsemOnlyOne, 0);

         if (WAIT_TIMEOUT == dwWaitRet) {

            // If the semaphore was not signaled, some thread expressions
            // was satisfied; we need to determine which expression.
            dwWaitRet = WaitForMultipleObjects(dwExpNum,
               ahThreads, FALSE, INFINITE);

         } else {

            // No expression was satisfied and WaitForSingleObject just gave
            // us the semaphore so we know that no expression can ever be
            // satisfied now -- waiting for an expression has timed-out.
```

```
            dwWaitRet = WAIT_TIMEOUT;
         }
      }

      // Break all the waiting expression threads out of their
      // wait state so that they can terminate cleanly.
      for (dwExpNum = 0; dwExpNum < dwNumExps; dwExpNum++) {

         if ((WAIT_TIMEOUT == dwWaitRet) ||
             (dwExpNum != (dwWaitRet - WAIT_OBJECT_0))) {

            QueueUserAPC(WFME_ExpressionAPC, ahThreads[dwExpNum], 0);
         }
      }

#ifdef _DEBUG
      // In debug builds, wait for all of expression threads to terminate
      // to make sure that we are forcing the threads to wake up.
      // In non-debug builds, we'll assume that this works and
      // not keep this thread waiting any longer.
      WaitForMultipleObjects(dwExpNum, ahThreads, TRUE, INFINITE);
#endif

      // Close our handles to all the expression threads
      for (dwExpNum = 0; dwExpNum < dwNumExps; dwExpNum++) {
         CloseHandle(ahThreads[dwExpNum]);
      }
   }  // error occurred while parsing
      CloseHandle(hsemOnlyOne);
      return(dwWaitRet);
   }


   ////////////////////////////////// End of File //////////////////////////////////////
```

WaitForMultExp.h

```
   /*****************************************************************************
   Module:  WaitForMultExp.h
   Notices: Copyright (c) 2000 Jeffrey Richter
   *****************************************************************************/


   #pragma once


   ////////////////////////////////////////////////////////////////////////////////


   DWORD WINAPI WaitForMultipleExpressions(DWORD nExpObjects,
      CONST HANDLE* phExpObjects, DWORD dwMilliseconds);


   ////////////////////////////////// End of File //////////////////////////////////////
```

WfMETest.cpp

```
/******************************************************************************
Module: WfMETest.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
******************************************************************************/


#include "..\CmnHdr.h"     /* See Appendix A. */
#include <windowsx.h>
#include <tchar.h>
#include <process.h>
#include "resource.h"
#include "WaitForMultExp.h"
//////////////////////////////////////////////////////////////////////////////


// g_ahObjs contains the list of event kernel object
// handles referenced in the Boolean expression.
#define MAX_KERNEL_OBJS     1000
HANDLE g_ahObjs[MAX_KERNEL_OBJS];


// ahExpObjs contains all the expressions. A single expression
// consists of a contiguous set of kernel object handles that
// is TRUE when all the objects are signaled at the same time.
// A NULL handle is used to separate OR expressions.

// A handle value may NOT appear multiple times within an AND
// expression but the same handle value may appear in
// different OR expressions.

// An expression can have a maximum of 64 sets with no more
// than 63 handles/set plus a NULL handle to separate each set
#define MAX_EXPRESSION_SIZE   ((64 * 63) + 63)


// m_nExpObjects is the number of entries used in the ahExpObjects array.
typedef struct {
   HWND     m_hwnd;                              // Where to send results
   DWORD    m_dwMilliseconds;                    // How long before timeout
   DWORD    m_nExpObjects;                       // # of entries in object list
   HANDLE   m_ahExpObjs[MAX_EXPRESSION_SIZE];    // List of objs
} AWFME, *PAWFME;
AWFME g_awfme;


// This message is posted to the UI thread when an expression
// comes true or when we timeout while waiting for an
// expression to come TRUE.
#define WM_WAITEND      (WM_USER + 101)


//////////////////////////////////////////////////////////////////////////////
```

```
   BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {
   chSETDLGICONS(hwnd, IDI_WFMETEXT);

   // Initialize the controls in the dialog box
   SetDlgItemInt(hwnd,  IDC_NUMOBJS, 4, FALSE);
   SetDlgItemInt(hwnd,  IDC_TIMEOUT, 30000, FALSE);
   SetDlgItemText(hwnd, IDC_EXPRESSION,
      _T("1 2 | 2 3 4 | 1 4"));

   // Set the multicolumn listbox's column size
   ListBox_SetColumnWidth(GetDlgItem(hwnd, IDC_OBJLIST),
      LOWORD(GetDialogBaseUnits()) * 4);

   return(TRUE);  // Accept default focus window.
}


//////////////////////////////////////////////////////////////////////////////


DWORD WINAPI AsyncWaitForMultipleExpressions(PVOID pvParam) {

   PAWFME pawfme = (PAWFME) pvParam;

   DWORD dw = WaitForMultipleExpressions(pawfme->m_nExpObjects,
      pawfme->m_ahExpObjs, pawfme->m_dwMilliseconds);
   PostMessage(pawfme->m_hwnd, WM_WAITEND, dw, 0);
   return(0);
}


//////////////////////////////////////////////////////////////////////////////


LRESULT Dlg_OnWaitEnd(HWND hwnd, WPARAM wParam, LPARAM lParam) {

   // Close all the event kernel object handles
   for (int n = 0; g_ahObjs[n] != NULL; n++)
      CloseHandle(g_ahObjs[n]);

   // Tell the user the result of running the test
   if (wParam == WAIT_TIMEOUT)
      SetDlgItemText(hwnd, IDC_RESULT, __TEXT("Timeout"));
   else
      SetDlgItemInt(hwnd, IDC_RESULT, (DWORD) wParam - WAIT_OBJECT_0, FALSE);
   // Allow the user to change values and run the test again
   EnableWindow(GetDlgItem(hwnd, IDC_NUMOBJS),    TRUE);
   EnableWindow(GetDlgItem(hwnd, IDC_TIMEOUT),    TRUE);
   EnableWindow(GetDlgItem(hwnd, IDC_EXPRESSION), TRUE);
   EnableWindow(GetDlgItem(hwnd, IDOK),           TRUE);
   SetFocus(GetDlgItem(hwnd, IDC_EXPRESSION));

   return(0);
}
```

```
//////////////////////////////////////////////////////////////////////////////

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

   // Obtain the user's settings from the dialog box controls.
   TCHAR szExpression[100];
   ComboBox_GetText(GetDlgItem(hwnd, IDC_EXPRESSION), szExpression,
      sizeof(szExpression) / sizeof(szExpression[0]));

   int nObjects = GetDlgItemInt(hwnd, IDC_NUMOBJS, NULL, FALSE);

   switch (id) {
   case IDCANCEL:
      EndDialog(hwnd, id);
      break;

   case IDC_OBJLIST:
      switch (codeNotify) {
      case LBN_SELCHANGE:
         // An item changed state, reset all items and set the selected ones.
         for (int n = 0; n < nObjects; n++)
            ResetEvent(g_ahObjs[n]);

         for (n = 0; n < nObjects; n++) {
            if (ListBox_GetSel(GetDlgItem(hwnd, IDC_OBJLIST), n))
               SetEvent(g_ahObjs[n]);
         }
         break;
      }
      break;
     case IDOK:
        // Prevent the user from changing values while the test is running
        SetFocus(GetDlgItem(hwnd, IDC_OBJLIST));
        EnableWindow(GetDlgItem(hwnd, IDC_NUMOBJS),    FALSE);
        EnableWindow(GetDlgItem(hwnd, IDC_TIMEOUT),    FALSE);
        EnableWindow(GetDlgItem(hwnd, IDC_EXPRESSION), FALSE);
        EnableWindow(GetDlgItem(hwnd, IDOK),           FALSE);

        // Notify the user that the test is running
        SetDlgItemText(hwnd, IDC_RESULT, TEXT("Waiting..."));

        // Create all of the desired kernel objects
        ZeroMemory(g_ahObjs, sizeof(g_ahObjs));
        g_awfme.m_nExpObjects = 0;
        ZeroMemory(g_awfme.m_ahExpObjs, sizeof(g_awfme.m_ahExpObjs));
        g_awfme.m_hwnd = hwnd;
        g_awfme.m_dwMilliseconds = GetDlgItemInt(hwnd, IDC_TIMEOUT, NULL, FALSE);

        ListBox_ResetContent(GetDlgItem(hwnd, IDC_OBJLIST));
        for (int n = 0; n < nObjects; n++) {
           TCHAR szBuf[20];
           g_ahObjs[n] = CreateEvent(NULL, FALSE, FALSE, NULL);

           wsprintf(szBuf, TEXT("  %d"), n + 1);
```

```
            ListBox_AddString(GetDlgItem(hwnd, IDC_OBJLIST),
               &szBuf[lstrlen(szBuf) - 3]);
         }

         PTSTR p = _tcstok(szExpression, TEXT(" "));
         while (p != NULL) {
            g_awfme.m_ahExpObjs[g_awfme.m_nExpObjects++] =
               (*p == TEXT('|')) ? NULL : g_ahObjs[_ttoi(p) - 1];
            p = _tcstok(NULL, TEXT(" "));
         }

         DWORD dwThreadId;
         CloseHandle(chBEGINTHREADEX(NULL, 0,
            AsyncWaitForMultipleExpressions, &g_awfme,
            0, &dwThreadId));
         break;
      }
   }

//////////////////////////////////////////////////////////////////////////////
INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);

      case WM_WAITEND:
         return(Dlg_OnWaitEnd(hwnd, wParam, lParam));
   }

   return(FALSE);
}


//////////////////////////////////////////////////////////////////////////////


int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   DialogBox(hinstExe, MAKEINTRESOURCE(IDD_TESTW4ME), NULL, Dlg_Proc);
   return(0);
}


//////////////////////////////// End of File /////////////////////////////////
```

WfMETest.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
//////////////////////////////////////////////////////////////////////////////
//
```

```
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

/////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

/////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

/////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_WFMETEST DIALOGEX 0, 0, 168, 185
STYLE DS_3DLOOK | DS_CENTER | WS_MINIMIZEBOX | WS_VISIBLE | WS_CAPTION |
    WS_SYSMENU
EXSTYLE WS_EX_APPWINDOW
CAPTION "WaitForMultipleExpressions"
FONT 8, "MS Sans Serif", 0, 0, 0x1
BEGIN
    LTEXT           "How many different &kernel objects are referenced in the expression:"
                    IDC_STATIC,3,4,121,17
    EDITTEXT        IDC_NUMOBJS,138,6,27,14,ES_AUTOHSCROLL
    LTEXT           "&Timeout (in milliseconds):",IDC_STATIC,4,28,83,8
    EDITTEXT        IDC_TIMEOUT,138,26,27,14,ES_AUTOHSCROLL
    LTEXT           "&Expression (use space for AND and | for OR):",
                    IDC_STATIC,4,44,143,8
    COMBOBOX        IDC_EXPRESSION,4,56,160,76,CBS_DROPDOWN | WS_VSCROLL |
                    WS_TABSTOP
    DEFPUSHBUTTON   "&Wait for Multiple Expressions",IDOK,34,72,99,14
    LTEXT           "&Signal which kernel objects:",IDC_STATIC,4,92,83,8
    LISTBOX         IDC_OBJLIST,4,102,160,68,LBS_MULTIPLESEL |
                    LBS_NOINTEGRALHEIGHT | LBS_MULTICOLUMN | WS_VSCROLL |
                    WS_HSCROLL | WS_TABSTOP
    LTEXT           "Expression satisfied:",IDC_STATIC,32,172,63,8
    LTEXT           "Timeout",IDC_RESULT,100,172,36,8
END

#ifdef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//
1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END
```

```
2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED


/////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_WFMETEXT            ICON    DISCARDABLE     "WaitForMultExp.ico"
#endif    // English (U.S.) resources
/////////////////////////////////////////////////////////////////////////////



#ifndef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//


/////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```