Microsoft

5

Fifth Edition

WINDOWS VIA C/C++

Jeffrey Richter
Christophe Nasarre

Wintellect
Know how.

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2007939306

ISBN: 978-0-7356-6377-0

1 2 3 4 5 6 7 8 9 10 QGT 6 5 4 3 2 1

Printed and bound in the United States of America.

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to mspinput@microsoft.com.

Acquisitions Editor: Ben Ryan
Developmental and Project Editor: Lynn Finnel
Editorial Production: Publishing.com
Technical Reviewer: Scott Seely; Technical Review services provided by Content Master, a member of CM Group, Ltd.

Body Part No. X14-25709

insert a produced element for a writer thread. The difference between these two signaling functions is not obvious:

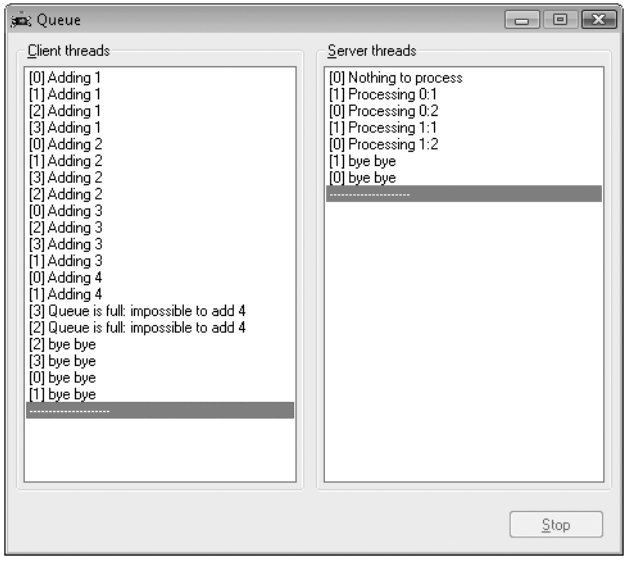
```
VOID WakeConditionVariable(
   PCONDITION_VARIABLE ConditionVariable);

VOID WakeAllConditionVariable(
   PCONDITION_VARIABLE ConditionVariable);
```

When you call **WakeConditionVariable**, one thread waiting for the same condition variable to be signaled inside a **SleepConditionVariable*** function will return with the lock acquired. When this thread releases the same lock, no other thread waiting on the same condition variable will be awakened. When you call **WakeAllConditionVariable**, one or several threads waiting for the same condition variable inside a **SleepConditionVariable*** function might wake up and return. Having multiple threads wake up is OK because you are assured that the lock can be acquired by only one writer thread at a time if you requested an exclusive lock or by several reader threads if you passed **CONDITION_VARIABLE_LOCKMODE_SHARED** to the **Flag** parameter. So, sometimes, all reader threads will wake up, or one reader and then one writer, and so on until each blocked thread has acquired the lock. If you work with the Microsoft .NET Framework, you might find similarities between the **Monitor** class and a condition variable. Both provide synchronized access through **SleepConditionVariable** / **Wait** and a signal feature with **Wake*ConditionVariable** / **Pulse(All)**. You can find more details about the **Monitor** class in the page dedicated to that topic on MSDN (*http://msdn2.microsoft.com/en-us/library/hf5de04k.aspx*) or in my book *CLR via C#, Second Edition* (Microsoft Press, 2006).

The Queue Sample Application

A condition variable always works in conjunction with a lock: either a critical section or an **SRWLock**. The Queue (08-Queue.exe) application uses an **SRWLock** and two condition variables to control a queue of request elements. The source code and resource files for the application are in the 08-Queue directory on the companion content Web page mentioned earlier. When you run the application and click the Stop button, after a while the following dialog box appears:



When Queue initializes, it creates four client threads (writers) and two server threads (readers). Each client thread appends a request element to a queue before sleeping for a period of time, and then it tries to add a request again. As an element is queued, the Client Threads list box is updated. Each entry indicates which client thread added the request element with its number. For example, the first entry in the list box indicates that client thread 0 appended its first request. Then client threads 1, 2, and 3 appended their first requests, followed by client thread 0 appending its second request, and so on.

Each server thread is responsible for processing requests—represented by an even number for thread 0 and an odd number for thread 1—but both have nothing to do until at least one element appears in the queue. When an element appears, a server thread wakes up to process the request. If the request number is even or odd as expected, the server thread processes the request, marks it as read, notifies the client's threads that it is possible for them to add a new request into the queue, and finally goes back to sleep until a new request is available. If there is no compatible request element to process, it goes back to sleep until there is something to read.

The Server Threads list box shows the status of the server threads. The first entry shows that server thread 0 is trying to find a request with an even number in the queue but does not find any. The second entry shows that server thread 1 is processing the first request from client thread 0. The third entry shows server thread 0 processing client thread 0's second request, and so on. The Stop button has been clicked, so the threads are notified to stop their processing and say "bye bye" in their related list box.

In this example, the server threads cannot process the client's requests quickly enough and the queue fills to maximum capacity. I initialized the queue data structure so that it can hold no more than 10 elements at a time; this causes the queue to fill quickly. Plus, there are four client threads and only two server threads. We see that the queue is full when client threads 3 and 2 attempt to append their fourth request to the queue without success.

The Queue Implementation Details

OK, so that's what you see—what's more interesting is how it works. The queue is managed by a C++ class, **CQueue**:

```
class CQueue
{
public:
   struct ELEMENT {
      int   m_nThreadNum;
      int   m_nRequestNum;
      // Other element data should go here
   };
   typedef ELEMENT* PELEMENT;

private:
   struct INNER_ELEMENT {
      int      m_nStamp;  // 0 means empty
      ELEMENT  m_element;
   };
   typedef INNER_ELEMENT* PINNER_ELEMENT;
```

```
private:
   PINNER_ELEMENT m_pElements;      // Array of elements to be processed
   int            m_nMaxElements;   // Maximum # of elements in the array
   int            m_nCurrentStamp;  // Keep track of the # of added elements

private:
   int GetFreeSlot();
   int GetNextSlot(int nThreadNum);

public:
   CQueue(int nMaxElements);
   ~CQueue();
   BOOL IsFull();
   BOOL IsEmpty(int nThreadNum);
   void AddElement(ELEMENT e);
   BOOL GetNewElement(int nThreadNum, ELEMENT& e);
};
```

The public **ELEMENT** structure inside this class defines what a queue data element looks like. The actual content is not particularly important. For this sample application, clients store their client thread number and their request number in this request element so that the servers can display this information in their list box when they process the retrieved even or odd request element. A real-life application generally does not require this information. This **ELEMENT** structure is wrapped by an **INNER_ELEMENT** structure that keeps track of the insertion order through the **m_nStamp** field, which is incremented each time an element is added.

For the other private members, we have **m_pElements**, which points to a fixed-size array of **INNER_ELEMENT** structures. This is the data that needs to be protected from the multiple client/server threads accesses. The **m_nMaxElements** member indicates how large this array is initialized to when the **CQueue** object is constructed. The next member, **m_nCurrentStamp**, is an integer that is incremented each time a new element is added to the queue. The **GetFreeSlot** private function returns the index of the first **INNER_ELEMENT** of **m_pElements** with an **m_nStamp** of **0** (meaning its content has been already read or is empty). If no such element is found, **Ð1** is returned.

```
int CQueue::GetFreeSlot() {

   // Look for the first element with a 0 stamp
   for (int current = 0; current < m_nMaxElements; current++) {
      if (m_pElements[current].m_nStamp == 0)
         return(current);
   }

   // No free slot was found
   return(-1);
}
```

The **GetNextSlot** private helper functions return the index in **m_pElements** of the **INNER_ELEMENT** with the lowest stamp (which means that it was added first) but different from 0 (which means free or read). If all elements have been read (their stamp is equal to 0), **Ð1** is returned.

```
int CQueue::GetNextSlot(int nThreadNum) {

   // By default, there is no slot for this thread
   int firstSlot = -1;

   // The element can't have a stamp higher than the last added
   int firstStamp = m_nCurrentStamp+1;

   // Look for the even (thread 0) / odd (thread 1) element that is not free
   for (int current = 0; current < m_nMaxElements; current++) {

      // Keep track of the first added (lowest stamp) in the queue
      // --> so that "first in first out" behavior is ensured
      if ((m_pElements[current].m_nStamp != 0) &&  // free element
          ((m_pElements[current].m_element.m_nRequestNum % 2) == nThreadNum) &&
          (m_pElements[current].m_nStamp < firstStamp)) {

         firstStamp = m_pElements[current].m_nStamp;
         firstSlot = current;
      }
   }

   return(firstSlot);
}
```

You should now have no trouble understanding **CQueue**'s constructor, destructor, and **IsFull** and **IsEmpty** methods, so let's turn our attention to the **AddElement** function, which is called by client threads to add a request element into the queue:

```
void CQueue::AddElement(ELEMENT e) {

   // Do nothing if the queue is full
   int nFreeSlot = GetFreeSlot();
   if (nFreeSlot == -1)
      return;

   // Copy the content of the element
   m_pElements[nFreeSlot].m_element = e;

   // Mark the element with the new stamp
   m_pElements[nFreeSlot].m_nStamp = ++m_nCurrentStamp;
}
```

If there is a free slot in **m_pElements**, it is used to store the **ELEMENT** passed as a parameter and the current stamp is incremented to count the number of added request elements. When a server

thread wants to process a request, it calls **GetNewElement**, passing the thread number (0 or 1) and the **ELEMENT** to be filled with the details of a corresponding new request:

```
BOOL CQueue::GetNewElement(int nThreadNum, ELEMENT& e) {

   int nNewSlot = GetNextSlot(nThreadNum);
   if (nNewSlot == -1)
      return(FALSE);

   // Copy the content of the element
   e = m_pElements[nNewSlot].m_element;

   // Mark the element as read
   m_pElements[nNewSlot].m_nStamp = 0;

   return(TRUE);
}
```

The **GetNextSlot** helper function does most of the job of finding the first element corresponding to the given reader thread. If there is one in the queue, **GetNewElement** copies this request's details back to the caller before stamping it as read with an **m_nStamp** value of **0**.

There is nothing really complicated here, and you must be thinking that the **CQueue** is not thread safe. You're right. In Chapter 9, I explain how other synchronization kernel objects can be used to build a thread-safe version of a queue. However, in the 08-Queue.exe application, it is the responsibility of the client and server thread to synchronize their access to the global instance of the queue:

```
CQueue              g_q(10);    // The shared queue
```

In the 08-Queue.exe application, three global variables are used to let client (writer) threads and server (reader) threads work in harmony, without corrupting the queue:

```
SRWLOCK             g_srwLock;  // Reader/writer lock to protect the queue
CONDITION_VARIABLE  g_cvReadyToConsume;  // Signaled by writers
CONDITION_VARIABLE  g_cvReadyToProduce;  // Signaled by readers
```

Each time a thread wants access to the queue, the **SRWLock** must be acquired, either in share mode by the server (reader) threads or in exclusive mode by the client (writer) threads.

The Client Thread Is the *WriterThread*

Let's see the client thread implementation:

```
DWORD WINAPI WriterThread(PVOID pvParam) {

   int nThreadNum = PtrToUlong(pvParam);
   HWND hWndLB = GetDlgItem(g_hWnd, IDC_CLIENTS);

   for (int nRequestNum = 1; !g_fShutdown; nRequestNum++) {

      CQueue::ELEMENT e = { nThreadNum, nRequestNum };
```

```
      // Require access for writing
      AcquireSRWLockExclusive(&g_srwLock);

      // If the queue is full, fall asleep as long as the condition variable
      // is not signaled
      // Note: During the wait for acquiring the lock,
      //       a stop might have been received
      if (g_q.IsFull() & !g_fShutdown) {
         // No more room in the queue
         AddText(hWndLB, TEXT("[%d] Queue is full: impossible to add %d"),
            nThreadNum, nRequestNum);

         // --> Need to wait for a reader to empty a slot before acquiring
         //     the lock again
         SleepConditionVariableSRW(&g_cvReadyToProduce, &g_srwLock,
            INFINITE, 0);
      }

      // Other writer threads might still be blocked on the lock
      // --> Release the lock and notify the remaining writer threads to quit
      if (g_fShutdown) {
         // Show that the current thread is exiting
         AddText(hWndLB, TEXT("[%d] bye bye"), nThreadNum);

         // No need to keep the lock any longer
         ReleaseSRWLockExclusive(&g_srwLock);

         // Signal other blocked writer threads that it is time to exit
         WakeAllConditionVariable(&g_cvReadyToProduce);

         // Bye bye
         return(0);
      } else {
         // Add the new ELEMENT into the queue
         g_q.AddElement(e);

         // Show result of processing element
         AddText(hWndLB, TEXT("[%d] Adding %d"), nThreadNum, nRequestNum);

         // No need to keep the lock any longer
         ReleaseSRWLockExclusive(&g_srwLock);

         // Signal reader threads that there is an element to consume
         WakeAllConditionVariable(&g_cvReadyToConsume);

         // Wait before adding a new element
         Sleep(1500);
      }
   }

   // Show that the current thread is exiting
   AddText(hWndLB, TEXT("[%d] bye bye"), nThreadNum);

   return(0);
}
```