Compound Synchronization Objects

Ruediger R. Asche
Microsoft Developer Network Technology Group

Created: July 21, 1994

Abstract

This article describes strategies to combine the Microsoft Windows synchronization primitives into more complex synchronization objects. It presents several solutions for implementing advanced synchronization objects along with a comparison of how they perform. The correctness of the implementations will be shown in a future article.

Introduction

Multithreading is a little bit like Zen Buddhism: The more you learn about it, the more it escapes you. Although the multithreading application programming interface (API) is fairly small, the hard part is to use it right. The problems you can run into when not correctly using synchronization objects can be extremely hard to track down, and formal analysis methods cannot always be applied.

The Motivation

The synchronization primitives that the Windows API provides are fairly basic: There are mutexes (or critical sections, which are similar in semantics), semaphores, and events. When you look at the specifications of other operating systems and programming languages, you will notice that some of them provide fairly intricate synchronization mechanisms, such as reader/writer locks and conditional critical sections. Modeling one synchronization mechanism with another is part of the "high art" of multithreading, as we will see, and the purpose of this article is to teach you a little bit of that high art.

Group Locks

The most basic synchronization primitive that the Windows API offers is a mutex object that can be owned by only one thread at a time, period. What if we want several threads to be able to claim the mutex, but only under certain conditions? A semaphore, for example, is a predefined synchronization object that restricts access to shared resources to a certain number of threads. The first variation we will look at are group locks, which provide another quality of restriction.

In the multithreaded world, it is hard to come up with good examples; most sample problems seem to be rather goofy or artificial, and it is up to the designer of a multithreaded application to determine if a particular sample is a good place to start for his or her application. Here is another artificial problem to demonstrate group locks: Assume a surgical suite in a hospital. There are two entries to the operating room: one for the medical staff (and their patients) and one for the cleaning staff.

For hygienic reasons, no member of the cleaning staff can enter the room as long as there are nurses or physicians in the room, but as soon as there is one janitor in the room, other janitors can enter at will. Conversely, no medical person can enter the room while there is any member of the cleaning staff (or more than one) in the room. In other

words, there can be an arbitrary number of members of either one of the two groups in the room at any time, but there can never be a mix of cleaning and medical personnel in the room. If we represent each employee by one thread, the surgical suite problem could be formulated as "we need a synchronization object that allows an arbitrary number of threads of exactly one group to claim it."

This problem can also be stated in a more general way: Let each thread belong to one of *n* groups of threads. The new synchronization object would allow, at a maximum, *m* threads of exactly one of the *n* groups of threads to claim it.

You will notice that there is no synchronization primitive to accomplish this behavior, but we can employ a mix of synchronization objects to perform the task.

For the sake of the upcoming discussion, let us assume that this primitive exists, and let us define an API for how we would use it. I will present a short test application that shows how to use the API, and then we can go about implementing the object.

Here is my suggestion for an API that could be used to access a group lock. I will, in fact, define the lock as a C++ class with the following definition:

```
class CompoundSynch
{
public:
virtual BOOL Initialize(int) = 0;
virtual void Cleanup() = 0;
virtual void Claim(int i) = 0;
virtual void Release(int i) = 0;
};
```

The different implementations of the lock will be derived from this base class.

*BOOL **Initialize** (int iNumberofGroups)* would initialize a group lock. Any application that intends to use a group lock must call this function before using the lock. The function returns TRUE if the lock could be initialized correctly, FALSE otherwise.

*void **Cleanup**(void)* is the counterpart to **Initialize** and destroys all data structures associated with the object.

*void **Claim**(int iGroup)* can be called from any thread to claim the lock, telling the system what group it belongs to. *iGroup* must be smaller than the integer passed to **Initialize** as the second parameter. This function will suspend the calling thread if the group lock is claimed by at least one thread that identified itself as belonging to another group; it will return if the lock is already claimed by a thread of the same group or if the lock is currently unowned. If the calling thread gets suspended on the lock, it will be resumed later on when the lock becomes available.

Let us further define that this function cannot be called recursively by the same thread; that is, it is not feasible for a thread that has claimed the lock to claim the lock again without first releasing it.

*void **Release**(GROUP_LOCK *pLock, int iGroup)* will release a lock previously claimed. It is an error to call this function with a different *iGroup* than the one passed to the corresponding previous call to **Claim**, and likewise, it is an error to call this function without previously calling **Claim**.

Note that this API is specific to C++. In C, it is perfectly fine, for example, to define a GROUPLOCK data structure that is being passed to global functions (similar to the way critical section access is defined).

Let us look at a sample application that would employ a group lock. The sample does not do anything meaningful; it just creates three threads each from the two groups that utilize the lock. Each thread goes through 200 loops, during which it tries to acquire the lock, goes to sleep for a few milliseconds, and then releases the lock. To keep the sample application from getting too academic, I designed it to keep track of the time that is spent while owning the lock; this way, we can later compare the efficiency of different approaches.

The sample also has a "sanity check" installed; that is, it keeps track of the number of threads of each group that owns the lock. If any thread has successfully claimed the lock and finds that a thread of another group already owns the lock, it prints a warning on the screen. This is a quick-and-dirty way to see if the lock implementation works correctly. Note, however, that in a real-life scenario, a sanity check like this might not be sufficient to show the correctness of the implementation. Due to the asynchronous nature of multithreading, a faulty behavior might not necessarily show up in any test run. You are very well advised to employ a formal correctness analysis such as the one presented in "Synchronization on The Fly" to make sure that your implementation works correctly. (I will discuss the correctness of all of these solutions in a future article.)

The code sample LOCKTEST consists of two C++ files: LOCKTEST.CPP contains the code for the lock test as shown below, and LOCKS.CPP contains the implementation of the locks. LOCKS.H contains the definition of the structure of the lock classes.

```
#define GROUPS 2    // We have 2 groups of objects here.
#define NUMBEROFTHREADS 3  // We want three of each group.
#define DELAY 50
#define DELAYBIAS 50
#define TESTLOOPS 200
signed long iSanity[GROUPS];

HANDLE hResumeEvent;

class ThreadInfo
{public:
 int iID;
 CompoundSynch *csSynch;
 ThreadInfo(int i,CompoundSynch *csArg)
 { iID = i;
   csSynch = csArg;
 };
};

long WINAPI ThreadFn(ThreadInfo *lpArg)
{ int iID, iLoop, iTest;
  long lWaitTime=0;
  long lWaitSum=0;
  long lTurnAround;
  long lMaxWait=0;
  long lMinWait=10000;
  long lGotItRightAway=0;
  int  iGoofUp=0;
  iID = lpArg->iID;
```

```
  CompoundSynch *csSynch;
  csSynch = lpArg->csSynch;
  WaitForSingleObject(hResumeEvent,INFINITE);  // to synch up all the threads...
  lTurnAround= timeGetTime();
  for (iLoop = 0; iLoop < TESTLOOPS; iLoop++)
  { lWaitTime = timeGetTime();

// Beginning of critical code!

    csSynch->Claim(iID);    // Identify yourself.
    lWaitTime = timeGetTime() - lWaitTime;
    InterlockedIncrement(&iSanity[iID]);  // Do the sanity check.
   if (!lWaitTime) lGotItRightAway++;
    lMaxWait= lMaxWait > lWaitTime ? lMaxWait : lWaitTime;
   lMinWait= lMinWait < lWaitTime ? lMinWait : lWaitTime;
    Sleep(DELAY-DELAYBIAS+(rand()%(2*DELAYBIAS)));
   for (iTest = 0; iTest < GROUPS; iTest++)
    { if (iTest == iID) continue;
       if (iSanity[iTest] > 0)
        iGoofUp++;
     };
   lWaitSum+=lWaitTime;
    InterlockedDecrement(&iSanity[iID]);
    csSynch->Release(iID);

// Critical code is done with here.

    Sleep(DELAY-DELAYBIAS+(rand()%(2*DELAYBIAS)));
 };   // End of for loop
lTurnAround= (timeGetTime() - lTurnAround)/TESTLOOPS;
if (iGoofUp >0)
  printf ("goofup!!! Threads of more than one group in
      the lock %d times\n\r",iGoofUp);
printf ("Avg. turnaround for thread of grp %d : %8ld ms;
      avg wait: %lf ms, min %ld, max %ld, aces:
      %ld\n\r",iID,lTurnAround,(float)lWaitSum/
      (float)TESTLOOPS,lMinWait,lMaxWait,lGotItRightAway);
delete lpArg;
return(0);
}

void RunTestForOneObjectType(CompoundSynch *csObject, char *szTitleOfTest)
{
HANDLE hThreads[NUMBEROFTHREADS][GROUPS];
int iLoop, iInnerLoop;
long lOldElapseTime, lNewElapseTime;
unsigned long iDummyID;
printf(szTitleOfTest);
printf("\n\r");
if (!csObject->Initialize(GROUPS)) return;
hResumeEvent = CreateEvent(NULL,TRUE,FALSE,NULL);
// Manual, initially blocking
// This samples the time before start of the test suite.
```

```
lOldElapseTime = timeGetTime();
// Create the threads.
for (iLoop = 0; iLoop <GROUPS; iLoop++)
{  iSanity[iLoop] = 0;
   for (iInnerLoop = 0; iInnerLoop < NUMBEROFTHREADS; iInnerLoop++)
   hThreads[iInnerLoop][iLoop] =
         CreateThread(NULL,0,(LPTHREAD_START_ROUTINE)ThreadFn,
     (void *)new ThreadInfo(iLoop,csObject),0,&iDummyID);
};
// Start all threads.
SetEvent(hResumeEvent);
// Now wait for termination.
WaitForMultipleObjects(GROUPS*NUMBEROFTHREADS,(const
         HANDLE *)hThreads,TRUE,INFINITE);
// Measure elapsed time.
lNewElapseTime = timeGetTime();
lNewElapseTime -= lOldElapseTime;
// Clean up.
for (iLoop = 0; iLoop < GROUPS; iLoop++)
  for (iInnerLoop; iInnerLoop< NUMBEROFTHREADS; iInnerLoop++)
   CloseHandle(hThreads[iInnerLoop][iLoop]);
printf ("Total elapsed time: %8ld ms; per loop:
         %ld\n\r",lNewElapseTime,lNewElapseTime/TESTLOOPS);
CloseHandle(hResumeEvent);
csObject->Cleanup();
delete csObject;
}
```

Now all we need to do is implement the lock API. (Famous last words!)

Just for the heck of it, let us try the whole thing with the API functions stubbed out: This is the implementation provided in **CUnsafe**.

```
BOOL CUnsafe::Initialize(int i) {return TRUE;};
void CUnsafe::Cleanup() {};
void CUnsafe::Claim(int i) {};
void CUnsafe::Release(int i) {};
```

When compiling and running the program, you will see a number of goofup messages on the screen, indicating that, as expected, the dummy implementation of our group lock synchronization object does not work correctly.

The Weenie Solution: Mutexes

For a second, let us pretend that we are satisfied with letting the group lock behave like a mutex (that is, we let only one thread at any time claim the lock). In that case, we can in fact implement the lock in terms of a mutex. Let us do this quickly. (It will be interesting later on to compare the performance figures to the "real" solution.) Here is an implementation of the lock that uses a mutex only (as provided in the **CMutex** object):

```
BOOL CMutex::Initialize(int i)
{ hMutex = CreateMutex(NULL,FALSE,NULL);
  if (!hMutex) return FALSE;
  return (TRUE);
};
void CMutex::Cleanup()
{ CloseHandle(hMutex);
};
void CMutex::Claim(int i)
{ WaitForSingleObject(hMutex,INFINITE);
};
void CMutex::Release(int i)
{ ReleaseMutex(hMutex);
};
```

After compiling and running the test application with this implementation, you will see that no goofup warnings will be printed. Recall, however, that this does not necessarily mean that the solution really *does* protect the data appropriately; our test run might just have been a coincidentally successful execution.

First Attempt: Critical Sections

I want to introduce one fairly intuitive and simple implementation of the group lock rather quickly. The problem with this solution is that it cannot be easily extended to more specific forms, such as the reader/writer lock. Anyway, here is the code:

```
BOOL CCritSect::Initialize(int i)
{ // Note: We don't do any error checking here. Better do that in real life.
  int iLoop;
  iGroupNo = i;
  csShields = (CRITICAL_SECTION *)
    VirtualAlloc(0,sizeof(CRITICAL_SECTION)*i,MEM_COMMIT,PAGE_READWRITE);
  pCounters = (long *)
    VirtualAlloc(0,sizeof(long)*i,MEM_COMMIT,PAGE_READWRITE);
  for (iLoop = 0; iLoop < i; iLoop++)
    {InitializeCriticalSection(&csShields[iLoop]);
     pCounters[iLoop] = -1;
    };
  hMutex = CreateEvent(NULL,FALSE,TRUE,NULL);
  return (TRUE);
};

void CCritSect::Cleanup()
{ int iLoop;
  for (iLoop = 0;iLoop<iGroupNo;iLoop++)
    DeleteCriticalSection(&csShields[iLoop]);
  CloseHandle(hMutex);
  VirtualFree((LPVOID)csShields,sizeof(CRITICAL_SECTION)*iGroupNo ,
              MEM_DECOMMIT);
```

```
  VirtualFree((LPVOID)pCounters,sizeof(long)*iGroupNo ,
              MEM_DECOMMIT);
};


void CCritSect::Claim(int i)
{ EnterCriticalSection(&csShields[i]);
  if (InterlockedIncrement(&pCounters[i]) == 0)
    WaitForSingleObject(hMutex, INFINITE);
  LeaveCriticalSection(&csShields[i]);
};

void CCritSect::Release(int i)
{EnterCriticalSection(&csShields[i]);
  if (InterlockedDecrement(&pCounters[i]) < 0)
     SetEvent(hMutex);
  LeaveCriticalSection(&csShields[i]);
};
```

In this simple solution, each group of threads is associated with a counter that keeps track of how many threads of each group have attempted to claim the lock. Each group is also associated with a critical section that regulates access to the lock within a group.

There is also one mutex object per lock that regulates the access to the group lock. Here's the way this works: The first thread of each group to attempt to claim the lock must wait for the mutex object; after the first thread has successfully entered the lock, many threads of the same group can arbitrarily enter without waiting for the mutex.

The per-group critical sections are necessary to ensure two things: First, they make the functions **Claim** and **Release** uninterruptible so that no two threads of the same group can garble the order of instructions due to interleaved incrementing/decrementing. Second, they "propagate" the wait from the first thread of each group to the other ones: Because the **WaitForSingleObject** call that the first thread submits to obtain the lock will not return until the lock is available, no other thread of the same group can obtain the critical section for that group and therefore "sneak into" the lock. We will see later on that this "cascaded waiting," although helping us out this time, may complicate matters in other instances.

Events

There are several ways to implement group locks, some of which are terribly complicated and require a high number of synchronization primitives. I looked into several solutions and liked the following one best; it is fairly small and intuitive—as implemented in **CEvents**:

```
BOOL CEvents::Initialize(int i)
{ // Note: We don't do any error checking here. Better do that in real life.
  int iLoop;
  iGroupNo = i;
  hEvents = (HANDLE *)
    VirtualAlloc(0,sizeof(HANDLE)*i,MEM_COMMIT,PAGE_READWRITE);
  pCounters = (long *)
    VirtualAlloc(0,sizeof(long)*i,MEM_COMMIT,PAGE_READWRITE);
```

```
  for (iLoop = 0; iLoop < i; iLoop++)
    {hEvents[iLoop]=CreateEvent(NULL,TRUE,FALSE,NULL);
     pCounters[iLoop] = -1;
    };
  hMutex = CreateEvent(NULL,FALSE,TRUE,NULL);
  return (TRUE);
};

void CEvents::Cleanup()
{ int iLoop;
  for (iLoop = 0;iLoop<iGroupNo;iLoop++)
    CloseHandle(hEvents[iLoop]);
  CloseHandle(hMutex);
  VirtualFree((LPVOID)hEvents,sizeof(HANDLE)*iGroupNo ,
              MEM_DECOMMIT);
  VirtualFree((LPVOID)pCounters,sizeof(int)*iGroupNo ,
              MEM_DECOMMIT);
};

void CEvents::Claim(int i)
{ if (InterlockedIncrement(&pCounters[i]) == 0)
   { WaitForSingleObject(hMutex, INFINITE);
     SetEvent(hEvents[i]);
   };
  WaitForSingleObject(hEvents[i],INFINITE);
};

void CEvents::Release(int i)
{ if (InterlockedDecrement(&pCounters[i]) < 0)
   { ResetEvent(hEvents[i]);
     SetEvent(hMutex);
   };
};
```

The most confusing part of this code is probably the initialization—if we implemented the lock for a fixed number of thread groups or limited the number to a predefined maximum, the code would even be smaller. Note also that I left out the error checking on the system calls (for readability reasons); you should always make sure to test every API call return value for failure.

Each thread group is associated with one event and a counter. What happens here, roughly, is that there is a two-level approach to synchronization: There is one "global" (that is, per-lock) synchronization object (CEvents.hMutex), which is used to regulate access from the first thread of each respective group to the lock, and one "local" (that is, per thread group) synchronization object, CEvents.hEvents[iGroup]. The first thread of any group to request access to the lock will either be granted access or will block on the global synchronizer, and as soon as the first thread acquires the lock, it signals the other threads of the same group that they can enter as well. The last thread of each group to release the lock will block other threads of the same group and also release the global lock so that the first thread of another thread group may acquire the lock. We could informally say that the first thread of each group tries to claim the lock "on behalf" of all of the other threads in its group.

This code shows a case for the usefulness of events. Although the global synchronization object looks semantically pretty much like a mutex in that it regulates the access of exactly one thread—the first of each group—to the lock, it

cannot be implemented as a mutex because a mutex object can only be released by the same thread that owns it. However, the group lock semantics imply that the last thread to release a group lock is not necessarily the same as the first thread that claimed the lock. Thus, the global synchronization object is implemented as an event object that can be signaled by a different thread than the one by which it was claimed. Because the global synchronizer can only be claimed by threads of one group at any time, I chose an auto-reset event, which will become unsignaled automatically once a wait for the object has been satisfied.

On the other hand, the local event can be claimed by as many threads of one group as desired, so I made it a manual-reset event that is manually set to unsignaled by the last thread to leave the lock.

Notice the use of **InterlockedIncrement** and **InterlockedDecrement**. If we had used the standard C operators ++ and — instead, it might have happened that two threads would mess up the value of the shared variable (see "Synchronization on the Fly" for details); thus the use of the inseparable instructions. Notice also that the variable is initialized to –1 instead of 0 because the **InterlockedIncrement** and **InterlockedDecrement** instructions are not guaranteed to return the exact results of the increment and decrement operations; instead, all they can do is test for 0, less than zero, or greater than zero. Because the crucial threads to perform the operations are the first one to enter and the last one to leave, an increment from –1 to 0 indicates the first thread to enter, and the decrement from 0 to -1 is performed by the last thread to leave, so the checks for 0 and less than zero are sufficient to determine which thread has claimed or released the lock. Note that the very same thing could be achieved by initializing the counter to 1, swapping the **InterlockedIncrement** and **InterlockedDecrement** instructions, and changing the test predicate in the **Release** function from a "<" to a ">".

My confession for today: This solution was stolen in part from Jeffrey Richter's BUCKET application. I took the code from Jeffrey, modified it a little bit, and found out (much to my surprise) that I received about 1 percent goofup messages; that is, of 200 loops, each thread found itself in only 2 loops where threads of other groups had claimed the lock already. It turned out that this was a known problem in his code; the version of BUCKET on which I modelled my lock used a preemption event instead of a local event. The solution looked good, but it turned out that in a particular "race" scenario between two threads of the same group, it was possible for those two threads to "sneak" into the lock and both believe that they were not the first threads of their group to claim the lock, which made the situation unsafe. The moral of the story: The problems with synchronization may be so subtle that they may slip by even somebody with a very strong background and profound knowledge of synchronization.

Variations on a Theme

Now that we have a working implementation of a group lock, let us look at some common variations. First, it might be useful to restrict the number of threads *per group* that can claim the lock. Doing that is fairly easy: Just add a semaphore to each group of threads, and right after waiting for the local event, each thread waits for the semaphore. The code is right here (and in the object **CSemaphore**), and as usual, I left out a lot of error checking:

```
#define SEMCOUNT 2

BOOL CSemaphore::Initialize(int i)
{ // Note: We don't do any error checking here. Better do that in real life.
  int iLoop;
  iGroupNo = i;
  hEvents = (HANDLE *)
    VirtualAlloc(0,sizeof(HANDLE)*i,MEM_COMMIT,PAGE_READWRITE);
  hSemaphores = (HANDLE *)
    VirtualAlloc(0,sizeof(HANDLE)*i,MEM_COMMIT,PAGE_READWRITE);
  pCounters = (long *)
```

```
      VirtualAlloc(0,sizeof(long)*i,MEM_COMMIT,PAGE_READWRITE);
    for (iLoop = 0; iLoop < i; iLoop++)
      {hEvents[iLoop]=CreateEvent(NULL,TRUE,FALSE,NULL);
       hSemaphores[iLoop]=CreateSemaphore(NULL,SEMCOUNT,SEMCOUNT,NULL);
       pCounters[iLoop] = -1;
      };

    hMutex = CreateEvent(NULL,FALSE,TRUE,NULL);
    return (TRUE);
  };

  void CSemaphore::Cleanup()
  { int iLoop;
    for (iLoop = 0;iLoop<iGroupNo;iLoop++)
      {CloseHandle(hEvents[iLoop]);
      CloseHandle(hSemaphores[iLoop]);
     };
    CloseHandle(hMutex);
    VirtualFree((LPVOID)hEvents,sizeof(HANDLE)*iGroupNo ,
                MEM_DECOMMIT);
    VirtualFree((LPVOID)hSemaphores,sizeof(HANDLE)*iGroupNo ,
                MEM_DECOMMIT);
    VirtualFree((LPVOID)pCounters,sizeof(int)*iGroupNo ,
                MEM_DECOMMIT);
  };

  void CSemaphore::Claim(int i)
  { if (InterlockedIncrement(&pCounters[i]) == 0)
     { WaitForSingleObject(hMutex, INFINITE);
       SetEvent(hEvents[i]);
     };
    WaitForSingleObject(hEvents[i],INFINITE);
    WaitForSingleObject(hSemaphores[i],INFINITE);
  };

  void CSemaphore::Release(int i)
  { ReleaseSemaphore(hSemaphores[i],1,0);
    if (InterlockedDecrement(&pCounters[i]) < 0)
     { ResetEvent(hEvents[i]);
       SetEvent(hMutex);
     };
  };
```

I hard code to two the number of threads to own each lock, but the code is easy to change for arbitrary counts; indeed, it may be possible to restrict each group of threads to a different maximum number of threads that may own the lock. (We will see a possible application later on.)

The Power of WaitForMultipleObjects

Finally, let us look at how we would implement the group lock using **WaitForMultipleObjects**, instead of using two separate **WaitForSingleObject** calls. Although this code looks similar to the "pure" group lock solution, it works somewhat differently, especially with regard to the semantics of the counter:

```
BOOL CWFMO::Initialize(int i)
{ // Note: We don't do any error checking here. Better do that in real life.
  int iLoop;
  iGroupNo = i;
  hEvents = (HANDLE *)
    VirtualAlloc(0,sizeof(HANDLE)*i,MEM_COMMIT,PAGE_READWRITE);
  pCounters = (long *)
    VirtualAlloc(0,sizeof(long)*i,MEM_COMMIT,PAGE_READWRITE);
  pHandleArrays = (PHANDLE *)VirtualAlloc(0,sizeof(PHANDLE)*i,MEM_COMMIT,PAGE_READWRITE
  hMutex = CreateEvent(NULL,FALSE,TRUE,NULL);
  for (iLoop = 0; iLoop < i; iLoop++)
    {hEvents[iLoop]=CreateEvent(NULL,TRUE,FALSE,NULL);
     pCounters[iLoop] = -1;
     (pHandleArrays)[iLoop] =
         (PHANDLE)VirtualAlloc(0,sizeof(HANDLE)*2,MEM_COMMIT,PAGE_READWRITE);
     ((pHandleArrays)[iLoop])[0]= hMutex;
    ((pHandleArrays)[iLoop])[1]= hEvents[iLoop];
    };
  return (TRUE);
};

void CWFMO::Cleanup()
{ int iLoop;
  for (iLoop = 0;iLoop<iGroupNo;iLoop++)
    CloseHandle(hEvents[iLoop]);
  CloseHandle(hMutex);
  VirtualFree((LPVOID)hEvents,sizeof(HANDLE)*iGroupNo ,
            MEM_DECOMMIT);
  VirtualFree((LPVOID)pCounters,sizeof(int)*iGroupNo ,
            MEM_DECOMMIT);
// Clean up the rest here, too.
};

void CWFMO::Claim(int i)
{  if (WaitForMultipleObjects(2,pHandleArrays[i],FALSE,INFINITE)
          == WAIT_OBJECT_0)
    SetEvent(hEvents[i]);
     InterlockedIncrement(&pCounters[i]);
};

void CWFMO::Release(int i)
{ if (InterlockedDecrement(&pCounters[i]) < 0)
   {
     ResetEvent(hEvents[i]);
     SetEvent(hMutex);
   };
};
```

The idea is pretty simple: Each group is associated with an array of two events, the first of which is our global event (the one that is shared between all groups), and the second of which is the local (group-specific) event. When no thread owns the lock, the global event is signaled, and the local event is unsignaled (because all local events are initialized to unsignaled). The first thread of each group will reverse this scenario because a call to **WaitForMultipleObjects** with the "wait for any" option enabled returns the index of the first object that was signaled. Thus, the first thread of a group to claim the lock will implicitly unsignal the global event, thereby blocking all other threads, and when the object for which the wait was satisfied was indeed the global event, that first thread will set the local event before incrementing the counter. Thus, in effect, the original situation (in which the first event in the object was signaled and the second one was unsignaled) has been reversed by the first thread. Every other thread of the same group will now pass because the second handle in the array has signaled; threads of other groups will be blocked because both the global and their respective local events are not yet signaled.

When a thread leaves its lock, it again flips the array by first unsignaling its local event (thereby blocking other threads from entering the lock) and then signaling the global event. Thus, at any time, at least one of the two objects in each array is unsignaled, and if one object is signaled, it will allow only threads of the same group to pass.

That's the theory. If you still don't believe me that synchronization problems can be horribly subtle and complicated, get this: I ran this test a bazillion times (this is no joke) with no problems. Then, one day, after having ported the application to C++ (originally everything was plain C), I ran the debug version of the lock, and much to my surprise, there was one single goofup message on the screen. Only in the debug version, and only one out of 200 times did the lock leak! That's what you get for not analyzing your code!

Using a ball-point pen and a lot of paper (recycled, of course!), I figured out the problem. It may happen that the last thread of one group leaves the lock, but in between the time it decrements its counter and flips the signal states in its array, another thread of the same group comes along, sneaks into the lock (because the local event of its group is still signaled), and parties on the critical data, during which time the thread that believed itself to be the last of its group closes down the local event of its group and opens up the global event, thereby possibly leading threads of other groups into the lock.

Looking more closely at the problem, we see that the leak is cause by a lack of atomicity in the flipping of the states in the event array. While in the call to **Claim**, the sequence that leads from <signaled, unsignaled> to <unsignaled, signaled> in the array goes through <unsignaled, unsignaled> in an uninterruptable fashion (while **WaitForMultipleObjects** executes); the sequence in **Release** allows another thread to satisfy its respective **WaitForMultipleObjects** call while the array is flipped from <unsignaled, signaled> to <unsignaled, unsignaled>. If we can enforce some degree of atomicity so that the transition is made without the chance of being interrupted by a fresh thread of the same group, we're in business.

Thus, the solution is to switch the local event from signaled to unsignaled *before* the group counter is decremented. This way, another thread of the same group that could otherwise sneak in will block on the array and, therefore, allow the releasing thread to clean up appropriately.

We achieve this by having a releasing thread first preventively unsignal the local event (this will block out all other threads from the same group) and then later on, if it turns out that the thread was not the last, signal it again. Here is the revised code:

```
void CWFMO::Release(int i)
{ ResetEvent(hEvents[i]);
  if (InterlockedDecrement(&pCounters[i]) < 0)
     SetEvent(hMutex);
     else SetEvent(hEvents[i]);
```

```
};
```

Reader/Writer Locks

Probably the most widely used application of a group lock is a special restricted case, the reader/writer lock. In fact, many programming languages and operating systems define a reader/writer lock as a required system primitive. A reader/writer lock is a lock that only serves two groups of threads: readers and writers. An arbitrary number of readers can own the lock at any time, but no more than one writer can own it. Thus, a reader/writer lock is some sort of an asymmetric group lock, which for one group (the readers) behaves like the standard group lock, but for the other group (the writers) adds a restriction—the limitation to only one at a time.

In order to implement a reader/writer lock, we could simply take the code for the restricted group lock, remove the semaphore for the reader group, and create the writer semaphore with a count of one. This will leave us with a few redundancies: First of all, since we do not allow more than one writer in the lock at any time, we do not need to keep count of the writers; furthermore, the local writer's event and the semaphore with count 1 can be collapsed into one object, which might as well become a mutex because the thread to release the lock will by definition be the one that claimed it in the first place. Look at the new implementation (found in **CRWLock**):

```
#define WRITER 0
#define READER 1


BOOL CRWLock::Initialize(int i)
{ // Note: We don't do any error checking here. Better do that in real life.
  hReaderEvent=CreateEvent(NULL,TRUE,FALSE,NULL);
  hMutex = CreateEvent(NULL,FALSE,TRUE,NULL);
  hWriterMutex = CreateMutex(NULL,FALSE,NULL);
  iCounter = -1;
  return (TRUE);
};

void CRWLock::Cleanup()
{ CloseHandle(hReaderEvent);
  CloseHandle(hMutex);
  CloseHandle(hWriterMutex);
};

void CRWLock::Claim(int i)
{
 if (i== WRITER)
 {
    WaitForSingleObject(hWriterMutex,INFINITE);
   WaitForSingleObject(hMutex, INFINITE);
 }
 else
 {
 if (InterlockedIncrement(&iCounter) == 0)
      { WaitForSingleObject(hMutex, INFINITE);
        SetEvent(hReaderEvent);
```

```
      };
   WaitForSingleObject(hReaderEvent,INFINITE);
   };
  };

  void CRWLock::Release(int i)
  { if (i == WRITER)
    {
       SetEvent(hMutex);
       ReleaseMutex(hWriterMutex);
    }
    else
    if (InterlockedDecrement(&iCounter) < 0)
       { ResetEvent(hReaderEvent);
         SetEvent(hMutex);
       };
  };
```

The implementation of the writer's claiming and releasing the lock is fairly simple: Any writer must first pass the mutex that the writers share to make sure that no two writers can possibly share the lock. As soon as a writer passes this point (that is, the one writer that is granted exclusive writing access to the lock), it goes through roughly the same steps as before, competing with the first reader for the global event.

Once the happy writer has passed both synchronization objects and has finished writing, it first signals the global event, thereby allowing one (that is, the first) waiting reader to claim the lock on behalf of all readers. Finally, the writer releases the writer's mutex, thereby allowing the next writer to compete with the readers for the global synchronizer.

The implementation of the reader has not changed.

Conditional Critical Sections

All solutions presented so far have one thing in common: Their implementations assume that one synchronization object will be used by the threads of all groups to guard access to the lock, and on top of that, the threads of one group may decide to cooperatively use that synchronization object. Thus, the first thread of each group to claim the synchronizer does so on behalf of other waiting threads of the same group. Those other threads then "sneak around" the synchronizer by not trying to claim it once they know that they are not the first arrivals.

It is not always desirable to enforce such a close degree of cooperation among threads. Here is another approach to implementing group (or reader/writer) locks: Let each group of threads be associated with one event object. If the event that belongs to a certain group is reset, a thread of that group is in the lock; thus, in order to enter the lock, a thread should check to see whether any of the events associated with the other groups are reset, and if yes, wait. This new solution is implemented in a version of the group lock to be found under CCR.C and CCR.H, which stands for *conditional critical region*. (We will see later on where this name comes from.)

This scheme is different from the other approaches in that there is not one global synchronization object, but one event object for each group of threads. Note that the event solution of the group lock uses both a global and a group-specific synchronization object, but the event solution works sort of "inside out" compared to the conditional critical region solution. In other words, in the event solution each group-specific synchronization object is tested by the threads of the group that the object is associated with, whereas in the conditional critical region implementation, the group-specific events are used by the threads of the other groups.

Conditional critical regions were proposed rather early in the history of multithreaded applications, and they are required synchronization primitives in several existing programming languages and operating systems. The idea is fairly simple: A conditional critical region is a section of code that can be executed by only one thread (just like a section of code shielded by a mutex object), but a thread can enter the region only if a given condition—an arbitrary Boolean expression that is evaluated at run time and can differ from thread to thread—is true.

Let us assume a hypothetical conditional critical region object with the following syntax:

```
ExecuteCriticalSectionConditionally(&CCRObject,condition,statement)
```

where *condition* is an arbitrary Boolean expression and *statement* is any elementary or compound C statement. Then the functions **Claim** and **Release** could be rewritten as follows (this is pseudocode, of course):

```
void ClaimGroupLock(PGROUPLOCK gLock, int iGroup)
{ ExecuteCriticalSectionConditionally(&CCR,<events of all other groups set>,
                                      if (iCounter[iGroup]++ == 0) ResetEvent(event of t
}
void ReleaseGroupLock(PGROUPLOCK gLock, int iGroup)
{ ExecuteCriticalSectionConditionally(&CCR,TRUE,
                                      if (iCounter[iGroup]-- < 0)
                                      SetEvent(event of this group));
};
```

It would, in fact, be very easy to show that this solution is correct, given the atomicity of the operation. Each thread that tries to claim the group lock must check to see that no thread from any other group has done so; the first thread to do so will reset the event for its group, and the last one to return will set the event. Because all code is executed in an uninterruptible context (using the imaginary conditional critical section object CCR), no two threads of any groups can interfere with each other.

The problem in implementing conditional critical sections is that you can easily fall into a deadlock situation. My first thought in implementing conditional critical regions was to use a mutex variable or a "normal" critical section, like this:

```
ExecuteCriticalSectionConditionally(&CCR, condition, statements)
```

which expands into:

```
EnterCriticalSection(&someCriticalSection);
```

```
if (condition)
   statements;
LeaveCriticalSection(&someCriticalSection);
```

However, if either the condition or the statements contain any wait for a synchronization object (as is the case with our imaginary implementation of the group lock), we inevitably fail because the wait cannot be satisfied unless another thread signals the object, which requires that the other thread enter the critical section that is owned by the waiting thread—classical deadlock. Note that this very scenario helped us earlier in the critical section solution!

Here is a working implementation of conditional critical regions. I use my favorite Windows API function, **WaitForMultipleObjects**, to overcome the potential deadlock. The wait that a thread executes when checking for the event objects of the other groups will always return because it waits for either the events or one global event that is always set to TRUE. If the return of the wait indicates that the events of the other groups are signaled, the lock is claimed; if not, the attempted wait is repeated, but in any case, the critical section that the wait is wrapped in is left so that other threads can enter their respective conditional critical sections. Note that this scheme works because the semantics of **WaitForMultipleObjects** implies that the list of objects is scanned from left to right; that is, objects that come earlier in the array of objects to wait for have a higher priority than later ones. (Remember that we also used this behavior earlier when we implemented the **WaitForMultipleObjects** solution to the group lock.)

This implementation is hard-coded to two groups of threads; it can be extended to more than two, but the data structures to do so would be a bit more complicated. Note also that with an additional mutex object, this solution can easily be adapted to a reader/writer lock as before.

```
BOOL CCCR::Initialize(int i)
{ // Note: We don't do any error checking here. Better do that in real life.
  int iLoop;
  iGroupNo = i;
  if (i>2) return FALSE;   // We currently hard code the thing for two groups.
  hAlwaysTrue = CreateEvent(NULL,TRUE,TRUE,NULL);
  hEvents = (HANDLE *)
    VirtualAlloc(0,sizeof(HANDLE)*i*2,MEM_COMMIT,PAGE_READWRITE);
  pCounters = (long *)
    VirtualAlloc(0,sizeof(long)*i,MEM_COMMIT,PAGE_READWRITE);
  for (iLoop = 0; iLoop < i; iLoop++)
    {hEvents[iLoop*2]=CreateEvent(NULL,TRUE,TRUE,NULL);
    hEvents[iLoop*2+1]=hAlwaysTrue;
     pCounters[iLoop] = -1;
    };
  InitializeCriticalSection(&csAtomizer);
  return (TRUE);
};

void CCCR::Cleanup()
{ int iLoop;
  DeleteCriticalSection(&csAtomizer);
  CloseHandle(hAlwaysTrue);
  for (iLoop = 0;iLoop<iGroupNo;iLoop++)
    CloseHandle(hEvents[iLoop*2]);
  VirtualFree((LPVOID)hEvents,sizeof(HANDLE)*iGroupNo*2 ,
```

```
                MEM_DECOMMIT);
    VirtualFree((LPVOID)pCounters,sizeof(int)*iGroupNo ,
                MEM_DECOMMIT);
  };

  void CCCR::Claim(int i)
  {
    TryAgain:
   EnterCriticalSection(&csAtomizer);
    switch(WaitForMultipleObjects(2,&hEvents[((i+1)%2)*2],FALSE,INFINITE))
      { case WAIT_OBJECT_0:     // the event of the other group
          if (InterlockedIncrement(&pCounters[i]) == 0)
            ResetEvent(hEvents[i*2]);
          LeaveCriticalSection(&csAtomizer);
        break;
       case WAIT_OBJECT_0+1:     // hAlwaysTrue
         LeaveCriticalSection(&csAtomizer);
        goto TryAgain;
       case WAIT_FAILED:
         printf("Error returning from WFMO: %d",GetLastError());
        break;
     };
  };

  void CCCR::Release(int i)
  { EnterCriticalSection(&csAtomizer);
    if (InterlockedDecrement(&pCounters[i]) < 0)
       SetEvent(hEvents[i*2]);
    LeaveCriticalSection(&csAtomizer);
  };
```

Note that instead of calling **WaitForMultipleObjects** on an event that is always signaled, we may as well do a **WaitForSingleObject** on the object with a timeout of 0. The behavior would be the same.

Performance Issues

You may remember that one of the motivations for worrying about compound synchronization objects in the first place was to allow for a higher throughput of threads (that is, allow several threads to do their critical computations at the same time if we know that they don't interfere with each other). Let us see if all the worrying was worth it.

For your convenience, I have provided the implementations for all the lock variations in the LOCKS.CPP file in the code sample and the test set in the LOCKTEST.CPP file. By running the LOCKTEST.EXE application and redirecting the output to a file, you can see for yourself how the different implementations are doing. I have included the PROTOCOL output file for you to scrutinize. If you wish to run the test yourself, here is a word of caution: The test suite runs for a long time. Don't think it has hung after 10 or more minutes.

There are eight tests altogether: **CUnsafe** (the synchronization functions stubbed out), **CMutex** (with the group lock implemented simply as a mutex), **CCritSect** (the first variation we looked at), **CEvent** (the generic group lock implementation), **CSemaphore** (the restricted group lock that allows at most two threads of each group to be in the lock at any time), **CWFMO** (a variation of **CEvent** in which **WaitForMultipleObjects** is used to wait simultaneously for the local and global synchronizers), **CRWLock** (the implementation of the reader/writer lock), and **CCCR** (the

implementation using conditional critical sections).

The test code (located in LOCKTEST.C) creates three threads each from two groups. Each thread executes 200 loops of the following sequence of statements:

1. Acquire the lock.
2. Try to verify the correctness of the algorithm by seeing if a thread from another group has claimed the lock already. (As I said before, this sanity check may not be sufficient to prove correctness, but is fairly reliable at making us aware of "leaks.")
3. Sleep for a random amount of time from 0 to 100 milliseconds (simulate a critical computation).
4. Release the lock.
5. Sleep for another random amount of time from 0 to 100 milliseconds (ms).

The following chart tells us how well the different implementations of the locks perform. I ran the test suite on a preliminary version of Microsoft Windows NT™ version 3.5 running on a uniprocessor 486/33 machine. Some optimizations have been added over Windows NT 3.1, which makes the overall performance a little bit better, but the relative performance figures are the same for both versions of the operating system. (In fact, I even ran the test suite on a preliminary version of the next version of Microsoft Windows® version 3.1, called Windows 95, obtaining relatively the same results.)

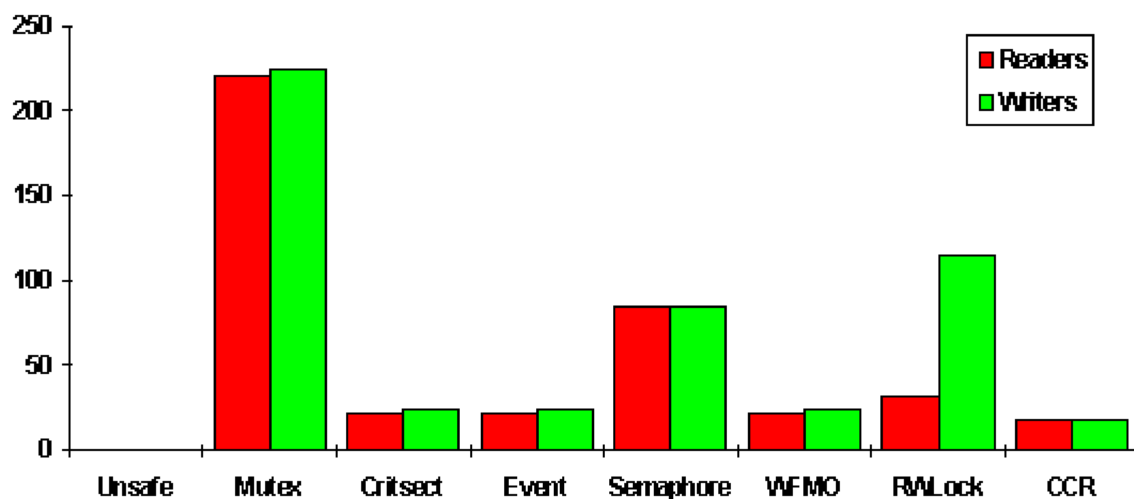

Figure 1. Relative performance of synchronization objects

The bars depict the absolute time (in milliseconds) that a thread of each group spends waiting per loop (that is, the time that a thread is blocked on the lock). Note that in all but the RWLock case, the distinction between readers and writers is invalid because by definition all groups are treated equally in the group lock version.

In the unsafe version, we see that no thread ever waits, because the lock is implemented as a nonworking stub. In the mutex version, all six threads compete for the one mutex, and consequently, the time each thread spends waiting for the lock is about five times the time a thread spends owning the lock (43 ms * 5 = 215 ms).

When we implement the group lock as an event (using critical sections, **WaitForSingleObject**, or **WaitForMultipleObjects**), the time spent waiting decreases from over 200 ms to about 20 ms, which is a 900 percent win. Wow! How can that happen if any thread at some time has to wait at least the time that a thread of the other group claims the lock (which is more often than not over 20 ms)?

If you look at the output protocol, you will find that I also sample the minimum and maximum waiting time for each

thread, and in my test run, the maximum wait time I sampled for both the EVENT and WFMO solution is 92 ms. However, the average wait time, 20 ms, is way below this peak value, so we can safely assume that in most cases, the time that a thread spends waiting is well below 20 ms. This is surprising because we would assume that a group lock is typically owned by more than one thread, so that the time a thread from the other group has to wait actually adds up. You will notice that, in most cases, a thread obtains the lock without waiting at all. Because it is not the first thread of its group to obtain the lock, this scenario is marked in the log file as an "ace." This "ace"-ing behavior is what makes the group lock so efficient.

The semaphore solution is somewhat surprising: The average wait time for a semaphore with count 2 is four times as long as for the nonrestricted group lock, even though all threads quite frequently manage to obtain the lock without waiting. However, we see that the maximum wait time per thread is much higher.

The reader/writer lock shows that a reader thread loses 50 percent throughput over the group lock scenario (the average wait time goes up to 30 ms as opposed to 20 ms in the group lock case). That is because, as you can see in the log, a reader obtains the lock without waiting ("an ace") far less than in other cases. That makes sense because the writers, being separated from each other, do not overlap their time in the lock anymore; thus, a reader does not get to obtain the lock as often as it did before. Each writer has to compete against three other threads: two other writers and the collective reader team; thus, a writer's throughput is about 1:3—that is, it spends about three times as long waiting for the lock as it spends in the lock itself.

Finally, the conditional critical section solution gives us another surprise: Although we force a claiming thread to poll the lock to see if it is taken by other threads, the performance of the group lock implementation is very compatible with the performance of the event solution. We can see that the number of "aces" (that is, times during which the lock is obtained without waiting) is fairly high.

Conclusion

As the performance chart shows, it is well worth the effort to fine-tune synchronization in multithreaded applications by building less restricted synchronization objects. Due to the very subtle nature of synchronization, it is absolutely crucial that compound synchronization objects be analyzed and tested carefully to ensure that they work as expected, both in terms of correctness and liveness. Aside from the sanity check technique that I introduced here, a formal analysis should also be employed.

Bibliography

Asche, Ruediger. "Multithreading for Rookies." September 1993. (MSDN Library, Technical Articles)

Ben-Ari, M. *Principles of Concurrent Programming*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1982.

Bic, L. and A. Shaw. *The Logical Design of Operating Systems*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1988.

Richter, Jeffrey. *Advanced Windows NT Programming*. Redmond, WA: Microsoft Press, 1993.

© 2015 Microsoft