CSE451 More Synchronization
Spring 2001

Gary Kimura

Lecture #9

April 13, 2001

Today

- So far we've talked about semaphore in a rather general terms
- Semaphores can be used both in user and kernel level code. Kernel level is usual direct support, user level code often requires some kernel calls to support synchronized access to the data structures
- Today we'll delve into some additional kernel level synchronization routines. Why? Because semaphores a nice but are not the right tool for all occasions
- In particular we're going to look at Windows 2000

Additional Synchronization Routines

- First two very specialized primitives
  - Spinlocks
  - Interlocked Operations
- Three general kernel synchronization objects
  - Events
  - Semaphores
  - Mutex
- Lastly one general purpose synchronization package
  - Eresource
- Some are kernel mode only
- Some are implemented in kernel mode and exported to the user as a system call
- Some are implemented in both kernel and user mode

Spinlocks

- A spinlock is a quick mechanism for acquiring exclusive access to a critical section of code
- It uses Interrupt Request Levels (IRQL) to disable other code from running
- In the Windows 2000 Kernel the spinlock calls are:
  - **KeAcquireSpinLock**: Enter critical section
  - **KeReleaseSpinLock**: Leave critical section
- On UP systems spinlocks are implemented by raising the Interrupt Request Level
- On MP systems spinlocks are implemented using a shared variable that contains zero or one (or the owners thread ID)
- Look at handout for coding example

Interlocked Operations

- Interlocked operations are atomic operations that can be used to synchronize access to a single variable or as a building block for more complex synchronization routines
- Some of the supported interlocked operations are
  - **InterlockedIncrement**
  - **InterlockedDecrement**
  - **InterlockedCompareExchange**
  - and there are more

X86 Interlocked Examples

lock prefix is MP only

```
__inline
LONG FASTCALL
InterlockedIncrement(
    IN PLONG Addend )
{
    __asm {
        mov     eax, 1
        mov     ecx, Addend
   lock xadd    [ecx], eax
        inc     eax
    }
}
```

```
where xadd atomically does
    temp := eax + [ecx]
    eax := [ecx]
    [ecx] := temp
```

```
__inline
LONG FASTCALL
InterlockedCompareExchange(
    IN OUT PLONG Destination,
    IN LONG Exchange,
    IN LONG Comperand )
{
    __asm {
        mov     eax, Comperand
        mov     ecx, Destination
        mov     edx, Exchange
   lock cmpxchg [ecx], edx
    }
}
```

```
where cmpxchg atomically does
    if eax == [ecx]
        [ecx] := edx
    else
        eax := [ecx]
```

Events and Semaphores

- Semaphores in Windows 2000 are plain vanilla counted semaphores
- Event are something new for our class. There are two types of events
  - Synchronization events: used to allow one thread at a time to enter a critical region of code
  - Notification events: used to signal all the threads about an event

Events

- An event variable's state is either signaled or not-signaled.
- Threads can wait for an event to become signaled
- Kernel level event calls are
  - **KeClearEvent**: Event becomes not-signaled
  - **KePulseEvent**: Atomically signal and then not-signal event
  - **KeReadStateEvent**: Returns current event state
  - **KeResetEvent**: Event becomes not-signaled and returns previous state
  - **KeSetEvent** and **KeSetEventBoostPriority**: Signal an event and optionally boost the priority of the waiting thread
- Similar calls are also available at the user level

Semaphores

- A semaphore variable is just a plain old vanilla counted semaphore, but its implementation shares much of the event logic
- Kernel semaphore calls are
  - **KeReadStateSemaphore**: Returns the signal state of the semaphore
  - **KeReleaseSemaphore**: Takes as input a release count and a priority boost of the newly released threads
- This is also a set of calls available at user level

Wait for routine

- In Windows there are many types of objects that a thread can wait on in the kernel. There are semaphores, events, timers, queues, processes, threads, etc.
- There are two routines that threads use to wait on these objects. In kernel mode the wait calls are
  - **KeWaitForSingleObject**: Wait for one event or semaphore to become signaled
  - **KeWaitForMultipleObjects**: Wait for one or more events or semaphores (WaitAny or WaitAll)
- There is an optional timeout on the wait calls that will return from the wait call even if the object is not signaled
- Two similar calls are also available at the user level

An example using events and waits

```
KEVENT Event;
KSEMAPHORE Semaphore;

Main:
    KeInitializeEvent( &Event, SynchronizationEvent);
    KeInitializeSemaphore(&Semaphore, 0, MAXLONG);

Thread One:
    Status = KeWaitForSingleObject( Event,…,Timeout);
    if (Status != STATUS_TIMEOUT) {
        // The event was signaled and its ours
    }

Thread Two:
    Objects[2] = {&Event, &Semaphor};
    Status = KeWaitForMultipleObjects( 2, Objects, WaitAny,…, Timeout);
    Switch (Status) {
    0: // event went off
    1: // Semaphore went off
    Default: // something else happened
    }

Thread Three:
    KeSetEvent(Event,…)
```

Executive Mutex

- A mutex looks like a spinlock from the programmers viewpoint but a mutex allow recursion and page faults
- Operations are
  - **ExAcquireFastMutex**: Enter critical region
  - **ExReleaseFastMutex**: Exit critical region
  - **ExTryToAcquireFastMutex**: Like acquire but return immediately without blocking and indicate if we acquired the mutex
- The executive mutex is implemented using kernel synchronization events
- Similar calls are also available at user level

Recap of what we have so far

- Spinlocks – in theory this can be done at the user level however for practical purposes it rarely is done at that level. Great for protecting short little critical sections. Usually we want everything in memory (I.e., we don't really want to wait while holding a spinlock)
- Interlocked Operations – usable in both user and kernel mode. Sometimes difficult to use but a great building block none the less
- Semaphore – Both user and kernel mode. Pretty good for general synchronization
- Events – Both user and kernel mode. More specialized than semaphores
- Mutex – Both user and kernel mode. More usable than spinlocks but also more expensive to use

Executive Resource

- The Executive Resource (Eresource) package is used by Windows 2000 kernel level code to handle the multi-reader/single-writer access to protected data structures.
- A version is also available in user mode
- The package exports a typedef called an Eresource
- Operations are:
  - **ExAcquireResourceShared**
  - **ExAcquireResourceExclusive**
  - **ExAcquireSharedStarveExclusive**
  - **ExReleaseResource** (**ExReleaseResourceForThread**)
  - **ExConvertExclusiveToShared**

Eresource features

- A resource can be acquired recursively
- The ownership of a resource is tied to a thread ID
- The users get to decide whether exclusive waiters starve
- The package automatically handles priority inversion
- The routines are implemented using spinlocks, events, and semaphores
- More details at a later lecture if time permits

Next Time

- Review and a midterm
- The midterm will be
  - Closed book
  - Closed notes
  - Closed neighbor
  - Open mind