Lecture 15: More on Multithreading

Prof. Aaron Lanterman
School of Electrical and Computer Engineering
Georgia Institute of Technology
Reference (for most of this talk)

Francisco Balena

2006

Microsoft Press
One Mutex

// This Mutex object must be accessible to all threads.
Mutex m = new Mutex();

public void WaitOneExample();
{
    // Attempt to enter the synchronized section,
    // but give up after 0.1 seconds
    if (m.WaitOne(100, false))
    {
        // Enter the synchronized section.
        ...
        // Exit the synchronized section, and release the Mutex.
        m.ReleaseMutex();
    }
}

A mutex is called “signalled” if no thread currently owns it

Many Mutexes - WaitAny

```csharp
static Mutex[] mutexes =
    { new Mutex(), new Mutex(), new Mutex() };

public void WaitAnyExample();
{
    // Wait until a resource becomes available.
    // (Returns the index of the available resource.)
    int mutexNdx = Mutex.WaitAny(mutexes);
    // Enter the synchronized section.
    // (This code should use only the
    // resource corresponding to mutexNdx.)
    ...
    // Exit the synchronized section, and release the Mutex.
    mutexes[mutexNdx].ReleaseMutex();
}
```

Many Mutexes - WaitAll

Mutex.WaitAll(mutexes)

• Wait until all resources have been released

• Useful if you can’t proceed until all the other threads are done

Naming a Mutex

Mutex m = new Mutex(false, "mutexname");

• If a Mutex with that name already exists, caller gets a reference to it; otherwise a new Mutex is created
• Lets you share Mutex objects among different applications
  – Not too relevant to video game programming
Mutexes vs. Monitor locks

- Mutexes slower than locks (around 20 times slower!)
  - Monitor locks operating at the level of the CLR
  - Mutexes operate at the OS level
- Mutexes generally reserved for interprocess communications (vs. interthread)

Info from B. Dawson, “Coding For Multiple Cores on Xbox 360 and Microsoft Windows,”
Semaphores

- Semaphores are good for restricting the number of threads accessing a resource to some maximum number.
Semaphores

Semaphore sem = new Semaphore(2, 2);

Initial count  Max count

sem.WaitOne(); // count from 2 to 1
sem.Release(); // count from 1 to 2
sem.Release(); // tries to bring
    // count from 2 to 3,
    // but throws a
    // SemaphoreFull
    // exception

Semaphores

Semaphore sem = new Semaphore(2, 2);

Initial count  Max count

sem.WaitOne(); // count from 2 to 1
sem.WaitOne(); // count from 1 to 0
sem.WaitOne(); // Blocks until
    // another thread
    // calls sem.Release();

Typical use of a semaphore

Semaphore sem = new Semaphore(2, 2);

void SemaphoreExample()
{
    // Wait until a resource becomes available.
    sem.WaitOne();
    // Enter the synchronized section

    // Exit the synchronized section, and
    // release the resource
    sem.Release();
}

Semaphores: Naming and timeouts

• Semaphores can be named like mutexes
• Like event waits (pulse/wait from previous lecture), semaphore and mutex waits can be given timeout parameter, and return a boolean indicating whether they acquired the resource “naturally” or timed out
Semaphore with max count 1 vs. Mutex

• A mutex or Monitor lock is owned by a thread; only that thread can release it

• Semaphores can be released by anyone
Thread safety

• Some .NET objects are thread-safe
• Some aren’t
• Some .NET objects have some method that are thread safe and some that aren’t
• Check the documentation
Synchronized types

• Some .NET types that aren’t ordinarily thread-safe offer thread-safe version

    // Create an ArrayList object, and add some values to it
    ArrayList al = new ArrayList();
    Al.Add(1); al.Add(2); al.Add(3);
    // Create a synchronized, thread-safe version
    ArrayList syncAl = ArrayList.Synchronized(al);
    // Prove that the new object is thread-safe
    Console.WriteLine(al.IsSynchronized);  // => False;
    Console.WriteLine(syncAl.IsSynchronized); // => True;
    // You can share the syncAl object among different
    // threads

Synchronized types - disadvantages

• Accessing synchronized objects is slower than accessing the original nonsynchronized object

• Generally better (in terms of speed) to use regular types and synchronize via lock blocks

True or False?

“If all you are doing is reading or writing a shared integer variable, nothing can go wrong and you don’t need any lock blocks, since reads and writes correspond to a single CPU instruction… right?”

Beware enregistering

```csharp
private bool Done = false;

void TheTask();
{
    // Exit the loop when another thread has set the Done
    // flag or when the task being performed is complete.
    while (this.Done == false)
    {
        // Do some stuff
        if (nothingMoreToDo)
        {
            this.Done = true;
            break;
        }
    }
}
```

Enregistering: compiler caches variable in a register (or local L1 cache on a multicore system), not in L2 or main memory.

volatile fields

```csharp
private volatile bool Done = false;
```

- volatile tells compiler other threads may be reading or writing to the variable, so don’t enregister it
- Does not ensure operations are carried out atomically for classes, structs, arrays…
- Does not ensure atomic read+write for anything
  - Increment, decrement
  - Test & Set
- Problematic on Xbox 360 (we’ll return to this later)

Interlocked.X (1)

Atomic increment and decrement:

```csharp
int lockCounter = 0;

// Increment the counter and execute some code if
// its previous value was zero
if (Interlocked.Increment(ref lockCounter) == 1)
{
    ...
    // Decrement the shared counter.
    Interlocked.Decrement(ref lockCounter);
}
```

Can also increment or decrement by an arbitrary amount with a second argument

Interlocked.X (2)

- Can assign a value and return its previous value as an atomic operation:

```csharp
string s1 = "123";
string s2 = Interlocked.Exchange(ref s1, "abc");
```

After execution, s2 = "123", s1 = "abc"

- Variation to the assignment if a and c are equal (reference equality in the case of objects):

```csharp
Interlocked.CompareExchange(ref a, b, c)
```

Dangers in the Xbox 360 CPU

- Interlocked.X & volatile-type operations are very fast
- Safe on Windows
- On Xbox 360, Interlocked.X and volatile keyword will prevent compiler from reordering reads and writes, but not the CPU!
  - Xbox 360 CPU won’t reorder instructions, but it may reorder writes!
  - Writes go to one of 8 store-gather buffers first, not L2 cache
  - 64 bytes can be transferred from a buffer to L2 in one op
  - Reads are an issue too
  - None of this is a problem with single-threaded code
- Can still do lockless programming on the Xbox 360, but you have to really know what you’re doing

Take home message

• Xbox 360 CPU may rearrange reads and writes to the L2 cache (not a problem on Windows)
• Lockless techniques (Interlocked.X, etc.) can give faster performance
  – Gains must justify complexity
  – Only for ninja kung-fu
• Monitor locks and Events (pulse, wait) are easiest/safest at this stage in your career
  – .NET will flush cache as needed