Lecture 19: Introduction to Multithreading

Prof. Aaron Lanterman
School of Electrical and Computer Engineering
Georgia Institute of Technology

References (1)

by Ben Albahari, Peter Drayton, and Brad Merrill, 2001

by Joseph Hall, 2008

Reference (2)

Francisco Balena
2006
Microsoft Press

Threading example

```csharp
using System;
using System.Threading;

class ThreadTest
{
    static void Main()
    {
        Thread t = new Thread(new ThreadStart(Go));
        t.Start();
        Go();
    }

    static void Go()
    {
        for (char c = 'a'; c <= 'z'; c++)
            Console.Write(c);
    }
}
```

Threading example output

```csharp
using System;
using System.Threading;

class ThreadTest
{
    static void Main()
    {
        Thread t = new Thread(new ThreadStart(Go));
        t.Start();
        Go();
    }
    static void Go()
    {
        for (char c='a'; c <= 'z'; c++)
        Console.Write(c);
    }
}
```

Output:

```
abcdefghijklmnopqrstuvwxyz
```


Lock example

```csharp
using System;
using System.Threading;

class LockTest
{
    static void Main()
    {
        LockTest lt = new LockTest();
        Thread t = new Thread(new ThreadStart(lt.Go));
        t.Start();
        lt.Go();
    }
    void Go()
    {
        lock (this)
        {
            for (char c='a'; c <= 'z'; c++)
            Console.Write(c);
        }
    }
}
```

Output:

```
abcdefghijklmnopqrstuvwxyz
```

Example from "C# Essentials," p. 108

Locks example output

```csharp
using System;
using System.Threading;

class LockTest
{
    static void Main()
    {
        LockTest lt = new LockTest();
        Thread t = new Thread(new ThreadStart(lt.Go));
        t.Start();
        lt.Go();
    }
    void Go()
    {
        lock (this)
        {
            for (char c='a'; c <= 'z'; c++)
                Console.Write(c);
            Monitor.Pulse(this);
            Monitor.Wait(this);
        }
    }
}
```

Example from "C# Essentials," p. 108

Pulse and wait

```csharp
using System;
using System.Threading;

class MonitorTest
{
    static void Main()
    {
        MonitorTest mt = new MonitorTest();
        Thread t = new Thread(new ThreadStart(mt.Go));
        t.Start();
        mt.Go();
    }
    void Go()
    {
        for (char c='a'; c <= 'z'; c++)
            lock (this)
            {
                Console.Write(c);
                Monitor.Pulse(this);
                Monitor.Wait(this);
            }
        Monitor.Wait(this);
    }
}
```

Output:

```
abcdefghijklmnopqrstuvwxyz
```

Example from "C# Essentials," p. 109

```
```

```
Example from "C# Essentials," p. 108
```

```
Example from "C# Essentials," p. 108
```

```
Example from "C# Essentials," p. 109
```
Lock: behind the curtain

```csharp
lock(expression)
{
    //mycode
}
```

is syntactic sugar for

```csharp
System.Threading.Monitor.Enter(expression);
try {
    // mycode
} finally {
    System.Threading.Monitor.Exit(expression);
}
```

From "C# Essentials," pp. 108-109

Pulse and wait example output

```csharp
using System;
using System.Threading;
class MonitorTest {
    static void Main() {
        MonitorTest mt = new MonitorTest();
        Thread t = new Thread(new ThreadStart(mt.Go));
        t.Start();
        mt.Go();
    }
    void Go() {
        for (char c = 'a'; c <= 'z'; c++)
            lock(this) {
                Console.Write(c);
                Monitor.Pulse(this);
                if (c < 'z')
                    Monitor.Wait(this);
            }
    }
}
```

Output:

```
aabbccddeeffghhiijjkkllmmnnnooppqqrsttuuvvwwxxyyzz
```

Example from "C# Essentials," p. 108

What's the problem?

```csharp
using System;
using System.Threading;
class MonitorTest {
    static void Main() {
        MonitorTest mt = new MonitorTest();
        Thread t = new Thread(new ThreadStart(mt.Go));
        t.Start();
        mt.Go();
    }
    void Go() {
        for (char c = 'a'; c <= 'z'; c++)
            lock(this) {
                Console.Write(c);
                Monitor.Pulse(this);
                Monitor.Wait(this);
            }
    }
}
```

Release lock temporarily; go to sleep until another thread pulses me

Breaking the deadlock

```csharp
void Go() {
    for (char c = 'a'; c <= 'z'; c++)
        lock(this) {
            Console.Write(c);
            Monitor.Pulse(this);
            if (c < 'z')
                Monitor.Wait(this);
        }
}
```

Example from "C# Essentials," p. 110
Impatient Wait

```csharp
public static bool Wait(object obj, int millisecondsTimeout);
```

- If another thread doesn’t pulse me within millisecondsTimeout, reacquire the lock and wake myself up
- Return true if reactivated by monitor being pulsed
- Return false if wait timed out

Grrrrrrrrrr!!!!!

- XNA on Xbox 360 uses Compact Framework, not full .NET like on Windows
- Compact Framework has a Monitor class (so can use locks), but it doesn’t implement Pulse/Wait and their variations 😞
- Not sure about “pro Xbox 360 development,” i.e. C++ XDK

Lock advice from MSDN

- “In general, avoid locking on a public type, or instances beyond your code’s control…
  - `lock(this)` is a problem if the instance can be accessed publicly.
  - `lock(typeof(MyType))` is a problem if MyType is publicly accessible.
  - `lock("myLock")` is a problem since any other code in the process using the same string, will share the same lock.”


Lock advice from Rico Mariani

```csharp
class MyClass {
    private static String myLock = “MyLock“;
    public void Foo() {
        lock(myLock) { ... }
    }
}
```

- “This is bad because string literals are normally interned, meaning that there is one instance of any given string literal for the entire program. The exact same object represents the literal… on all threads. So if someone else comes along and locks a literal named “MyLock” his literal will interfere with yours.
- Recommendation:

```csharp
private static Object myLock = new Object();
```

Don’t lock on value types

• Value types can be “boxed” to act as reference types…

• …but each lock construct will create a different box

Polling

• Main thread checks flag variables set by the worker threads when they finish

• Useful if main thread can do some stuff (e.g., eye-candy animation in a turn-based strategy game) independently of the worker threads (e.g. AI), but needs worker threads to finish before continuing (e.g. making the computer’s move)

Polling example

```csharp
bool done = false;
while (!done)
{
    Thread.Sleep(0);
    done = true;
    for int(i = 0; i < ThreadDone.Length; i++)
    {
        done &= m_ThreadDone[i];
    }
    Worker thread i sets
    m_ThreadDone[i]=true before it exits
}
```

Code from Joseph Hall, “XNA Game Studio Express,” p. 608

The problem with polling

• Polling takes up “C# cycles”

• If your main thread only needs to wait until its worker threads are done, the Wait/Pulse approach is better
  – Let the .NET runtime handle it!
  – Uh… oh, but only on Windows. 😞
One Mutex

```csharp
// 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

```csharp
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 (available on Windows)

```csharp
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)


Semaphores

- Semaphores are good for restricting the number of threads accessing a resource to some maximum number
- As far as I can tell, in XNA, semaphores only available on Windows 😞
  - Seems to be missing in Compact Framework and hence XNA on Xbox 360
  - Not sure about “pro Xbox 360 development,” i.e. C++ XDK

Semaphore sem = new Semaphore(2,2);

sem.WaitOne(); // count from 2 to 1
sem.Release(); // count from 1 to 2
sem.Release(); // tries to bring to 3,
  // but throws a // SemaphoreFull // exception
  // count from 2
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 (makes sense on Windows)
• 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 locks


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

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, not in L2 or main memory

volatile fields

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
• Can still be problematic when doing “real C++ XDK” Xbox 360 programming (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):
    Interlocked.CompareExchange(ref a, b, c)
  ```


Out-of-order execution (1)

• “CPUs employ performance optimizations that can result in out-of-order execution, including memory load and store operations.”

• “Memory operation reordering normally goes unnoticed within a single thread of execution, but causes unpredictable behaviour in concurrent programs and device drivers unless carefully controlled.”

http://en.wikipedia.org/wiki/Memory_barrier
### Out-of-order execution (2)

- “When a program runs on a single CPU, the hardware performs the necessary book-keeping to ensure that programs execute as if all memory operations were performed in program order, hence memory barriers are not necessary.”
- “However, when the memory is shared with multiple devices, such as other CPUs in a multiprocessor system, or memory mapped peripherals, out-of-order access may affect program behavior.
- “For example a second CPU may see memory changes made by the first CPU in a sequence which differs from program order.”

http://en.wikipedia.org/wiki/Memory_barrier

### Memory barriers

- “a class of instructions which cause a central processing unit (CPU) to enforce an ordering constraint on memory operations issued before and after the barrier instruction.”

- PowerPC: sync, lwsync, eieio assembly instructions

http://en.wikipedia.org/wiki/Memory_barrier

### C#: MemoryBarrier()

- “Synchronizes memory access as follows: The processor executing the current thread cannot reorder instructions in such a way that memory accesses prior to the call to MemoryBarrier execute after memory accesses that follow the call to MemoryBarrier.”


### Notes on MemoryBarrier()

- “MemoryBarrier is required only on multiprocessor systems with weak memory ordering (for example, a system employing multiple Intel Itanium processors).”
- “For most purposes, the C# lock statement…the Monitor class provide easier ways to synchronize data.”

Dangers in the Xbox 360 CPU

- **Interlocked.X & volatile-type operations are very fast**
- **Safe on Windows (because of Intel memory model)**
- **When doing "real X++ XDK" Xbox 360 development, Interlocked.X and volatile keyword will prevent compiler from reordering reads and writes, but not the CPU!**
  - Xbox 360 CPU may reorder writes to L2 cache!
  - 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, but can be a problem with multithreaded
  - Allowed in PowerPC’s “relaxed memory consistency” model


---

Compact Framework to the rescue? (1)

- “Now, we have access to a fair few Interlocked.xxx methods in the framework, which would do fine if I were programming on Windows, however on the 360 I need to be sure that I am not going to be caught out by write-reordering by the CLR or CPU. (i.e the reading thread spins until Interlocked.xxx sees a flag change, but the writing thread's CPU hasn't finished writing out its data to its cache, causing the reading thread to see old data).”
  - CosmicFlux, 7/9/2007


---

Dangers in the Xbox 360 CPU

- **Can still do native lockless programming in on the Xbox 360, but you have to really know what you’re doing**


---

Compact Framework to the rescue? (2)

“From the CF guys who implemented these methods: The Interlocked functions in NETCF provide a memory barrier on both sides of the interlocked operation. (This is different than native Xbox360 programming.) In addition, we provide the Thread.MemoryBarrier api if the customer needs to place an explicit memory barrier. Also, the Monitor functions are generally a higher performance operation than using a Mutex unless there are many many collisions on the lock. They were quite impressed that someone actually understood the issues involved :-)
  - Shawn Hargreaves, 7/10/2007

Take home message

• Xbox 360 CPU may cause different threads to see writes happening out-of-order (not a problem on Windows)
• Lockless techniques (Interlocked.X, etc.) can give faster performance
  – Only for ninja kung-fu when doing native Xbox 360 development; gains must justify complexity
  – Should be “safer” in XNA (but lockless programming is still tricky)
• Monitor locks and polling (with volatile declarations where needed) are probably easiest/safest at this stage in your career

Setting thread priority in C#

```csharp
t.Priority = ThreadPriority.Normal;
```

• Defaults to normal
• OS may ignore you
• Be careful about boosting thread priority
  – If the priority is too high, you could cause the system to hang and become unresponsive
  – If the priority is too low, the thread may starve

Locating your threads on the Xbox 360

```csharp
Thread.CurrentThread.SetProcessorAffinity(new int[] {index});
```

• Set thread affinity **within** the worker thread immediately after starting it
  – Don’t forget to call it, or your worker thread will be running on the same hardware thread as your main thread
• Only available on Xbox 360 XNA

Check to see if you’re on an Xbox 360

```csharp
#if XBOX360
Thread.CurrentThread.SetProcessorAffinity(new int[] {index});
#endif
```

• No way I know of in C# to manually set processor affinity in Windows like on the Xbox 360
• Windows decides what threads run where
### Xbox 360 hardware threads

<table>
<thead>
<tr>
<th>Ind</th>
<th>CPU</th>
<th>Thr</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Not available in XNA</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>Available; main thread; game runs here by default</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>Not available in XNA</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>Available; parts of the Guide and Dashboard live here</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>Available; Xbox Live Marketplace downloads</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>2</td>
<td>Available; parts of the Guide and Dashboard live here</td>
</tr>
</tbody>
</table>

Table from Joseph Hall, “XNA Game Studio Express,” p. 608

### Xbox 360 specific notes

- "If a program holds a lock for too long—because of poor design or because the thread has been swapped out by a higher priority thread—then other threads may be blocked for a long time.
- "This risk is particularly great on Xbox 360, because the software threads are assigned a hardware thread by the developer, and the operating system won't move them to another hardware thread, even if one is idle."


### Xbox 360 specific notes (2)

- The Xbox 360 also has no protection against priority inversion, where a high-priority thread spins in a loop while waiting for a low-priority thread to release a lock.


### Advice

- More than one thread per core isn’t bad...
- …but more than one processor-intensive task per core is!
- Put most intensive tasks on separate cores, and some less-demanding tasks on those same cores (threads that work in short bursts, disk I/O, etc.)

Advice from Joseph Hall, “XNA Game Studio Express,” p. 610
More advice

- Limit number of synchronization points
- Don’t lock resources longer than necessary
- Avoid sharing data when possible
- Profile your code before and after to make sure you’re getting the performance benefits you expect
  - Very easy to write multithreaded code that performs worse than single threaded!

Advice from Joseph Hall, “XNA Game Studio Express,” p. 611