Thread Safety

Agenda

- Highlight issues with multi threaded programming
- Introduce thread synchronization primitives
- Introduce thread safe collections

Need for Synchronization

- Creating threads is easy
- When threads share data problems can occur
  - Inconsistent reads
  - State corruption
- Synchronization fixes these problems, but potentially creates a new problem
  - Over synchronization reduces scalability
- Lots of techniques to implement synchronization
  - Each have cost and benefit
- Developers role is to write an application that scales and is thread safe, by selecting the best synchronization technique

Simple Increment

- Two threads
  - Sharing an instance of Counter.
  - Both are calling Increment 1000 times
- Question
  - What is the value of count after both threads have completed?

```
public class Counter
{
  protected int count;

  public virtual void Increment()
  {
    count++;
  }
  public int Value { get { return count; }  }
}
```

Simple Increment, NOT Atomic

- Even a simple count++ is not an atomic operation.
  - Multiple CPU instructions that could be interweaved.
- Consider the possible execution below of two threads (T0, T1)
  - Assuming count=0 at the start
  - At the end of execution i would be 1 and not the desired 2.
- If two threads don't attempt to increment count at the same time not a problem. Spotting these kind of errors is hard

time

```
T0:  MOV R0,count
T0:  ADD R0,1              T1:  MOV R0,count
T0:  MOV count,R0          T1:  ADD R0,1
                           T1:  MOV count,R0
```

Interlocked

- Modern CPU’s expose special instruction set to perform various operations atomically
  - Cost more than non atomic variants.
- Access to these instructions via Interlocked class
  - Interlocked.Increment
  - Interlocked.Decrement
  - Interlocked.Add

```
public class InterlockedCounter : Counter
{
  public override void Increment()
  {
    // Atomic count++
    Interlocked.Increment(ref count);
  }
}
```

Multi step state transition

- What happens if
  - Thread A is inside ReceivePayment
  - Thread B is inside NetWorth
- Can Interlocked help ?

```
 class SmallBusiness {
   decimal Cash = 0;
   decimal Receivables = 1000;

   public void ReceivePayment(decimal amount){
     Cash += amount;
     Receivables -= amount;
   }

   public decimal NetWorth {
     get { return Cash + Receivables; }
   }
}
```

Sequential access

- To fix the problem
  - Sequentialise access to the object state
- How
  - Each instance of a reference type has a Monitor
  - CLR guarantees that only one thread can own the monitor
  - If a thread can't acquire the monitor it enters a wait state
  - When the monitor is available it is woken up and proceeds
- Critical areas of code can therefore be protected by using a monitor.

Monitor based solution

- Only one thread in any **critical region** at any point in time

```
private object _lock = new object();

public void ReceivePayment(decimal amount)
{
  Monitor.Enter(_lock);
    Cash += amount;
    Receivables -= amount;
  Monitor.Exit(_lock);
}
```

Could be an issue with exceptions

```
public decimal NetWorth
{  get {
    Monitor.Enter(_lock);
    try { return Cash + Receivables; }
    finally { Monitor.Exit(_lock);
    }
   }
}
```

Deals better with exceptions

Lock keyword

- Enter, try, finally , Exit common pattern
  - C# language offers lock keyword to assist
  - Compiler emits try, finally logic
- Use of Monitor.Enter and lock can lead to deadlocks
  - Prefer Montior.TryEnter which takes a timeout
- Avoid using lock(this) and lock(typeof(X))
  - Less control over objects use for synchronization.

```
public void ReceivePayment(decimal amount){
 lock (_lock)
 {
   Cash += amount;
   Receivables -= amount;
 }
}
```

High Read to Write Ratio

- Monitor provides mutual exclusion behaviour
  - Excluding readers and writers
- Thread safety not an issue if all threads read.
- Better throughput may be achieved with a Synchronization primitive that ensures
  - There can be Many Readers, Zero Writer
  - Or One Writer, Zero Readers
- This is known as a ReaderWriterLock
  - .NET 3.5 and above prefer ReaderWriterLockSlim
  - Pre 3.5, ReaderWriterLock
    - Not well implemented, can result in writer being denied access for long periods of time.
- Often used for caching, where most of the time is spent reading with occasional updates.

Reader Writer Lock

```
public class SimpleCache
{
 private Dictionary<int, string> cache=
              new Dictionary<int, string>();
 private ReaderWriterLockSlim _lock = new ReaderWriterLockSlim();

 public string Get(int key) {
   _lock.EnterReadLock();
    try { return cache[key]; }
    finally { _lock.ExitReadLock(); }
}

 public void Set(int key, string val)
 {
   _lock.EnterWriteLock();
   try { cache.Add(key,val) }
   finally { _lock.ExitWriteLock(); }
 }
}
```

Many threads can can read from the cache

When one thread has the write lock no other thread can obtain read or write lock.

Synchronization across app domains

- Managed synchronization primitives only allow synchronization inside a single app domain
- How to control access to a shared file ?
  - Requires Kernel based synchronization
- Kernel synchronization can be achieved via managed wrappers
  - Mutex
  - Semaphore
  - AutoResetEvent
  - ManualResetEvent
- These synchronization primitives are orders of magnitude more expensive than managed ones

Summary

- A variety of ways to perform synchronization, the skill is picking the correct one
- Concurrent collections make it simpler to write efficient thread safe code
- Only use kernel synchronization primitives when absolutely necessary
- Analyse code and imagine worse possible race conditions