Multi-threaded Programming

March 26, 2008

Multi-threaded Programming in C#

- Sometimes we want a program to do two or more things at once
  - E.g. Do a long computation without locking up the gui
  - Handle multiple requests from network clients simultaneously
- One solution: event driven programming
  - Slice long tasks into short-components
  - Manually decide when to execute components
  - Very very hard to do right.
- Another solution: *multi-threaded* programming
  - A thread is a delegate (e.g. function)
  - C#can schedule multiple *threads* “at the same time”
    - threads might share time on one processor
    - or threads may be run concurrently on multiple processors
  - Only very hard to do right.

A first multi-threaded program

```
using System;
using System.Threading;

// Some examples based on Joesph Albahari's notes
class ThreadTest{
  static void WriteY(){
    while (true){ Console.Write("Y"); }
  }

  static void Main(){
    // Start running WriteY while we run main
    Thread t = new Thread(WriteY);
    t.Start();

    while (true){ Console.Write("x"); }
  }
}
```

Program Output

YYYYYYYYYYxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
YYYYYYYYYYYYYYYYYYYYxxxxxxxxxxxxxxxxxxxxxxxxxxxx . . .

Basic types and operations in System.Threading

```
// Delegate types used to create threads
public delegate void ThreadStart();
public delegate void ParameterizedThreadStart
                                    (object obj);

// Thread class constructors
Thread(ThreadStart start)
Thread(ParameterizedThreadStart start)

// Starts t running.  Can only be called once.
t.Start()
t.Start(parameter)

// Kills t
t.Abort()
```

Basic types and operations in System.Threading

```
// Pauses/Restarts t
t.Suspend()
t.Resume()

// Calling thread waits for t to finish
t.Join()

// Calling thread waits for 42 milliseconds
Thread.Sleep(42)
```

Threads contain their own copies of local variables

```
using ...;

class ThreadTest2{
  static void Main(){
    Thread t = new Thread(Count);
    Thread s = new Thread(Count);
    t.Start();
    s.Start();
  }

  static void Count(){
    // s and t threads will have different
    // copies of i
    for(int i=0; i<100; i++){
      Console.Write("␣"+ i);
    }
  }
```

Program Output

```
0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17
18  19  20  21  22  23  24  25  26  27  28  0  1  2  3  4
5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20
21  22  23  24  25  26  27  28  29  30  31  32  33  34  35
36  37  38  39  40  41  42  43  44  45  46  47  48  49  50
51  52  53  54  55  56  57  58  59  60  61  62  63  64  65
66  67  68  69  70  71  72  73  74  75  76  77  78  79  80
81  82  83  84  85  86  87  88  89  90  91  92  93  94  95
96  97  98  99  29  30  31  32  33  34  35  36  37  38  39
40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
55  56  57  58  59  60  61  62  63  64  65  66  67  68  69
70  71  72  73  74  75  76  77  78  79  80  81  82  83  84
85  86  87  88  89  90  91  92  93  94  95  96  97  98  99
```

Program Output

```
0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17
18  19  20  21  22  23  24  25  26  27  28  0  1  2  3  4
5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20
21  22  23  24  25  26  27  28  29  30  31  32  33  34  35
36  37  38  39  40  41  42  43  44  45  46  47  48  49  50
51  52  53  54  55  56  57  58  59  60  61  62  63  64  65
66  67  68  69  70  71  72  73  74  75  76  77  78  79  80
81  82  83  84  85  86  87  88  89  90  91  92  93  94  95
96  97  98  99  29  30  31  32  33  34  35  36  37  38  39
40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
55  56  57  58  59  60  61  62  63  64  65  66  67  68  69
70  71  72  73  74  75  76  77  78  79  80  81  82  83  84
85  86  87  88  89  90  91  92  93  94  95  96  97  98  99
```

Program Output

```
0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17
18  19  20  21  22  23  24  25  26  27  28  0  1  2  3  4
5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20
21  22  23  24  25  26  27  28  29  30  31  32  33  34  35
36  37  38  39  40  41  42  43  44  45  46  47  48  49  50
51  52  53  54  55  56  57  58  59  60  61  62  63  64  65
66  67  68  69  70  71  72  73  74  75  76  77  78  79  80
81  82  83  84  85  86  87  88  89  90  91  92  93  94  95
96  97  98  99  29  30  31  32  33  34  35  36  37  38  39
40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
55  56  57  58  59  60  61  62  63  64  65  66  67  68  69
70  71  72  73  74  75  76  77  78  79  80  81  82  83  84
85  86  87  88  89  90  91  92  93  94  95  96  97  98  99
```

Program Output

```
0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17
18  19  20  21  22  23  24  25  26  27  28  0  1  2  3  4
5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20
21  22  23  24  25  26  27  28  29  30  31  32  33  34  35
36  37  38  39  40  41  42  43  44  45  46  47  48  49  50
51  52  53  54  55  56  57  58  59  60  61  62  63  64  65
66  67  68  69  70  71  72  73  74  75  76  77  78  79  80
81  82  83  84  85  86  87  88  89  90  91  92  93  94  95
96  97  98  99  29  30  31  32  33  34  35  36  37  38  39
40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
55  56  57  58  59  60  61  62  63  64  65  66  67  68  69
70  71  72  73  74  75  76  77  78  79  80  81  82  83  84
85  86  87  88  89  90  91  92  93  94  95  96  97  98  99
```

Program Output

```
0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17
18  19  20  21  22  23  24  25  26  27  28  0  1  2  3  4
5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20
21  22  23  24  25  26  27  28  29  30  31  32  33  34  35
36  37  38  39  40  41  42  43  44  45  46  47  48  49  50
51  52  53  54  55  56  57  58  59  60  61  62  63  64  65
66  67  68  69  70  71  72  73  74  75  76  77  78  79  80
81  82  83  84  85  86  87  88  89  90  91  92  93  94  95
96  97  98  99  29  30  31  32  33  34  35  36  37  38  39
40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
55  56  57  58  59  60  61  62  63  64  65  66  67  68  69
70  71  72  73  74  75  76  77  78  79  80  81  82  83  84
85  86  87  88  89  90  91  92  93  94  95  96  97  98  99
```

Threads can share state too

```
class ThreadTest3{
  static void Main(){
    Thread t = new Thread(Count);
    Thread s = new Thread(Count);
    t.Start();
    s.Start();
  }

  //Counter shared between threads
  static int i = 0;

  static void Count(){
    for( ; i<50; i++){
      Console.Write("␣"+ i);
    }
  }
}
```

Program Output

```
 0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15
16  17  18  19  20  21  22  23  24  25  26  27  28
28  29  30  31  32  33  34  35  36  37  38  39  40
41  42  43  44  45  46  47  48  49
```

Program Output

```
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
16 17 18 19 20 21 22 23 24 25 26 27 28
28 29 30 31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49
```

Program Output

```
 0  1  2  3  4  5  6  7  8  9  10  11  12  13  14  15
16  17  18  19  20  21  22  23  24  25  26  27  28
28  29  30  31  32  33  34  35  36  37  38  39  40
41  42  43  44  45  46  47  48  49
```

whoa.

Multi-threaded programming is non-deterministic

- Order that threads execute is unpredictable
- Control can transfer anywhere: between statements, or even within statements
- The compiler may move statements, add temporary variables, remove variables, etc.
  - Control may transfer between statements that don't even occur in the source code!
- Last program illustrates a *race condition*—two threads produce unpredictable behavior because of timing issues.

Let's explore the race condition

```
class ThreadTest4{
  static void Main(){
    Thread a = new Thread(Count);
    Thread b = new Thread(Count);
    t.Start("a");
    s.Start("b");
  }

  static int i = 0;

  static void Count(object id){
    for( ; i<500; i++){
      Console.WriteLine("␣("+id+":"+ i+")␣");
    }
  }
}
```

Program Output

```
(a:0)
(a:1)
⋮
(a:29)
(a:30)
(a:31)
(b:31)
(b:32)
(b:33)
⋮
(b:61)
(b:62)
(b:63)
(b:65)
(b:66)
(b:67)
⋮
```

```
⋮
(b:239)
(b:240)
(b:241)
(a:64)
(a:242)
(a:243)
(a:244)
⋮
```

```
⋮
(a:442)
(a:443)
(a:444)
(b:267)
(b:445)
(b:446)
⋮
(b:498)
(b:499)
(a:461)
```

Program Output

```
(a:0)
(a:1)
⋮
(a:29)
(a:30)
(a:31) ⇐
(b:31) ⇐
(b:32)
(b:33)
⋮
(b:61)
(b:62)
(b:63)
(b:65)
(b:66)
(b:67)
⋮
```

```
⋮
(b:239)
(b:240)
(b:241)
(a:64)
(a:242)
(a:243)
(a:244)
⋮
```

```
⋮
(a:442)
(a:443)
(a:444)
(b:267)
(b:445)
(b:446)
⋮
(b:498)
(b:499)
(a:461)
```

Program Output

(a:0)
(a:1)
⋮
(a:29)
(a:30)
(a:31)
(b:31)
(b:32)
(b:33)
⋮
(b:61)
(b:62)
(b:63) ⇐
(b:65) ⇐
(b:66)
(b:67)
⋮

⋮
(b:239)
(b:240)
(b:241)
(a:64)
(a:242)
(a:243)
(a:244)
⋮

⋮
(a:442)
(a:443)
(a:444)
(b:267)
(b:445)
(b:446)
⋮
(b:498)
(b:499)
(a:461)

Program Output

(a:0)
(a:1)
⋮
(a:29)
(a:30)
(a:31)
(b:31)
(b:32)
(b:33)
⋮
(b:61)
(b:62)
(b:63)
(b:65)
(b:66)
(b:67)
⋮

⋮
(b:239)
(b:240)
(b:241)
(a:64) ⇐
(a:242)
(a:243)
(a:244)
⋮

⋮
(a:442)
(a:443)
(a:444)
(b:267)
(b:445)
(b:446)
⋮
(b:498)
(b:499)
(a:461)

Program Output

```
(a:0)
(a:1)
⋮
(a:29)
(a:30)
(a:31)
(b:31)
(b:32)
(b:33)
⋮
(b:61)
(b:62)
(b:63)
(b:65)
(b:66)
(b:67)
⋮
```

```
⋮
(b:239)
(b:240)
(b:241)
(a:64)
(a:242)
(a:243)
(a:244)
⋮
```

```
⋮
(a:442)
(a:443)
(a:444)
(b:267) ⇐
(b:445)
(b:446)
⋮
(b:498)
(b:499)
(a:461)
```

Program Output

(a:0)
(a:1)
⋮
(a:29)
(a:30)
(a:31)
(b:31)
(b:32)
(b:33)
⋮
(b:61)
(b:62)
(b:63)
(b:65)
(b:66)
(b:67)
⋮

⋮
(b:239)
(b:240)
(b:241)
(a:64)
(a:242)
(a:243)
(a:244)
⋮

⋮
(a:442)
(a:443)
(a:444)
(b:267)
(b:445)
(b:446)
⋮
(b:498)
(b:499)
(a:461)⇐

Conclusion of analyzing anomalies

- Generated loop code must have following structure:

```
string s;
while(i < 500){
   s = "␣("+id+":"+ i+")␣";
   Console.WriteLine(s);
   i++;
}
```

- Threads can switch between the assignment to s and the call to WriteLine.

Preventing race conditions with locks.

- All C# reference objects can be used as locks.
- Entering a lock(o){ } changes the state of o to be “locked”.
- If o is already locked, lock(o){ } *blocks* the calling process.
- Blocked processes can continue when the locking process exists the lock region.

- A Gotcha: Using primitives (e.g. int, bool) as locks won’t work correctly, due to a bad interaction with auto-boxing.

Fixing the shared counter with locks

```
static int i = 0;
static object locker = new object();

static void Count(object id) {
  bool done = false; // Local state: used freely

  while(!done){
    lock(locker){   // Safe to access i here

      if(i < 500) {
        Console.WriteLine("␣("+id+":"+ i+")␣");
        i++;
      } else {
        done = true;
      }
    }
} }
```

More synchronization and locking

- Terminology

  critical section Code enclosed as with a lock.

  mutual exclusion Problem of preventing multiple threads from access a resource or running code simultaneously.

  deadlock A program is deadlocked if each thread is waiting for some other thread.

  synchronization General term for the coordination of threads in a system.

- Lock performs atomic test-and-set
  - Runtime simultaneously marks object as locked and returns it's previously value.
  - No chance for a bad race conditions.
  - Most mutual exclusion techniques based on a hardware instruction to do *test-and-set*, but not strictly necessary (i.e. the Baker's Algorithm by Lamport).

Other C# data structures for synchronization

Monitor Monitors act like locks. Manually call Monitor.Enter(locker) and Monitor.Exit(locker) to lock and unlock.

EventWaitHandle Threads suspend and are woken up by *events* (no relation to event delegates). Typically a thread handles one event at a time, and incoming events are queued if the thread is already busy.

Mutex Like Monitors, but can be assigned unique names. Names can be used to synchronize different operating system processes.

Semaphore Like locks, but can allow more than one thread to access a critical section at once.

And subclasses of the above...

The Synchronization attribute automatically prevents many race conditions.

- Derive your class from System.Runtime.Remoting.Contexts.ContextBoundObject.
- Mark it with a [Synchronization] attribute
- Compiler ensures that instance method code is treated as a single critical section.
- Compiled code uses a lock and a dummy class to mediate access to objects of the underlying class.

Fixing the shared counter: Synchronization attribute

```
using System.Runtime.Remoting.Contexts;

[Synchronization]
class ThreadTest2 : ContextBoundObject {
  static void Main(){ /* as usual */ }

  static int i = 0;

  // Recall: this is the naive Count
  void Count(object id){
    for( ; i<5000; i++){
      Console.WriteLine("␣("+id+":"+ i+")␣");
    }
  }
}
```

Program Output

```
(a:0)
(a:1)
(a:2)
(a:3)
(a:4)
(a:5)
(a:6)
(a:7)
(a:8)
(a:9)
(a:10)
(a:11)
(a:12)
(a:13)
(a:14)
(a:15)
(a:16)
(a:17)
(a:18)
(a:875)
(a:876)
(a:877)
(a:878)
(a:879)
(a:880)
(a:881)
(a:882)
(a:883)
(a:884)
(a:885)
(a:886)
(a:887)
(a:888)
(a:889)
(a:890)
(a:891)
(a:892)
(a:893)
(a:3285)
(a:3286)
(a:3287)
(a:3288)
(a:3289)
(a:3290)
(a:3291)
(a:3292)
(a:3293)
(a:3294)
(a:3295)
(a:3296)
(a:3297)
(a:3298)
(a:3299)
(a:3300)
(a:3301)
(a:3302)
(a:3303)
(a:4984)
(a:4985)
(a:4986)
(a:4987)
(a:4988)
(a:4989)
(a:4990)
(a:4991)
(a:4992)
(a:4993)
(a:4994)
(a:4995)
(a:4996)
(a:4997)
(a:4998)
(a:4999)
```

Limitations of the Synchronization attribute

- Compiler needs to be very conservative to prevent races $\Rightarrow$ lots of locking.
- Problems with over-locking
  - loss of parallelism (our problem)
  - increased potential for deadlock
- Level of indirection adds overhead to each method call.
- Automatic locking probably not suitable except in very simple cases

Where to go from here

- This lecture is not the whole story
- See Joseph Albahari's article Threading in C# for a readable account of C# concurrency.
- If necessary, consider using threads in your final projects.