Multi-threading in .NET: Introduction and suggestions

Source: http://www.yoda.arachsys.com/csharp/threads/
## Inhalt

Multi-threading in .NET: Introduction and suggestions .......................................................... 4  
Introduction: What is multi-threading? .................................................................................... 4  
How does multi-threading work in .NET? .............................................................................. 5  
Multi-threaded "Hello, world" ................................................................................................. 5  
Passing Parameters to Threads ............................................................................................... 7  
Using the thread pool instead .................................................................................................. 7  
.NET 2.0 solution 1: anonymous methods (with and without the thread pool) ..................... 7  
.NET 2.0 solution 2: ParameterizedThreadStart ..................................................................... 8  
Data races .................................................................................................................................. 8  
Exclusive access - Monitor, Enter/Exit and the lock statement ........................................... 11  
Deadlocks ................................................................................................................................ 14  
More Monitor methods ............................................................................................................ 15  
WaitHandles - Auto/ManualResetEvent and Mutex ............................................................... 18  
Auto/ManualResetEvent ......................................................................................................... 18  
Mutex ..................................................................................................................................... 20  
Volutility, Atomicity and Interlocking ...................................................................................... 22  
A shortcut for some cases: the Interlocked class ................................................................. 25  
Threaded in Windows Forms ................................................................................................... 26  
The Thread Pool and Asynchronous Methods ......................................................................... 30  
ThreadPool.QueueUserWorkItem() ....................................................................................... 31  
Calling BeginInvoke on a delegate ....................................................................................... 31  
BeginRead (etc) ..................................................................................................................... 34  
Timers ..................................................................................................................................... 37  
System.Windows.Forms.Timer ................................................................................................. 37  
System.Timers.Timer ............................................................................................................. 37  
System.Threading.Timer .......................................................................................................... 37  
Examples ................................................................................................................................. 38  
Shutting Down Worker Threads Gracefully ............................................................................ 39  
Choosing What To Lock On ................................................................................................... 41  
An Alternative Approach To Monitors ................................................................................... 43  
When to use Threads .............................................................................................................. 45  
Thread-safe Types and Methods .............................................................................................. 47  
Thread-safe static methods, unsafe instance methods ............................................................ 47  
Type with thread affinity ......................................................................................................... 47  
Methods and properties using thread-local storage .............................................................. 47  
Totally thread-safe types ....................................................................................................... 48  
Types with limited thread safety ............................................................................................ 48  
Aborting and Interrupting Threads .......................................................................................... 48  
Aborting a thread .................................................................................................................... 48  
Interrupting a thread .............................................................................................................. 49  
A nasty bug in the framework .................................................................................................. 49  
Why Thread.Abort/Interrupt should be avoided ................................................................... 50  
Collected Hints and Tips ......................................................................................................... 51  
Resources ............................................................................................................................... 51
Multi-threading in .NET: Introduction and suggestions

One of the greatest understatements I've heard in a newsgroup was made by Patricia Shanahan, in a Java newsgroup in 2001: "Multi-threaded programming needs a little care." Multi-threading is probably one of the worst understood aspects of programming, and these days almost all application programmers need to understand it to some extent. This article acts as an introduction to multi-threading and gives some hints and tips for how to do it safely. Warning: I'm not an expert on the subject, and when the real experts start discussing it in detail, my head starts to spin somewhat. However, I've tried to pay attention to those who know what they're doing, and hopefully the contents of this article form at least part of a multi-threading "best practice".

This article uses the C# type shorthands throughout - int for Int32 etc. I hope this makes it easier for C# developers to read, and won't impede any other developers too much. It also only talks about the C# ways of declaring variables to be volatile and locking monitors. Developers using other languages can find the equivalents in their own preferred environment, I'm sure.

Introduction: What is multi-threading?

The fact that you're reading this article in the first place means you probably have at least some idea of what multi-threading is about: it's basically trying to do more than one thing at a time within a process.

So, what is a thread? A thread (or "thread of execution") is a sort of context in which code is running. Any one thread follows program flow for wherever it is in the code, in the obvious way. Before multi-threading, effectively there was always one thread running for each process in an operating system (and in many systems, there was only one process running anyway). If you think of processes running in parallel in an operating system (e.g. a browser downloading a file and a word processor allowing you to type, both "at the same time"), then apply the same kind of thinking within a single process, that's a reasonable way to visualise threading.

Multi-threading can occur in a "real" sense, in that a multi-processor box may have more than one processor executing instructions for a particular process at a time, or it may be effectively "simulated" by multiple threads executing in sequence: first some code for thread 1 is executed, then some code for thread 2, then back to thread 1 etc. In this situation, if both thread 1 and thread 2 are "compute bound" (all they're doing is computation, without waiting for any input from the network, or file system, or user etc) then that won't actually speed things up at all - in fact, it'll slow things down as the operating system has to switch between threads, and the memory cache probably won't be as effective. However, much of today's computing involves waiting for something to happen, and during that time the processor can be doing something else. Intel's "Hyper-Threading" technology which is on some of its more recent chips (bearing in mind that this article was written in early 2004!) is a sort of hybrid between this "real" and "simulated" threading - for more information, see Intel's web page on the subject [http://www.intel.com/technology/hyperthread/].
How does multi-threading work in .NET?

.NET has been designed from the start to support multi-threaded operation. There are two main ways of multi-threading which .NET encourages: starting your own threads with ThreadStart delegates, and using the ThreadPool class either directly (using ThreadPool.QueueUserWorkItem) or indirectly using asynchronous methods (such as Stream.BeginRead, or calling BeginInvoke on any delegate).

In general, you should create a new thread "manually" for long-running tasks, and use the thread pool only for brief jobs. The thread pool can only run so many jobs at once, and some framework classes use it internally, so you don't want to block it with a lot of tasks which need to block for other things. The examples in this article mostly use manual thread creation. On the other hand, for short-running tasks, particularly those created often, the thread pool is an excellent choice.

Multi-threaded "Hello, world"

Here is virtually the simplest threading example which actually shows something happening:

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

public class Test
{
    static void Main()
    {
        ThreadStart job = new ThreadStart(ThreadJob);
        Thread thread = new Thread(job);
        thread.Start();
        for (int i=0; i < 5; i++)
        {
            Console.WriteLine("Main thread: {0}", i);
            Thread.Sleep(1000);
        }
    }

    static void ThreadJob()
    {
        for (int i=0; i < 10; i++)
        {
            Console.WriteLine("Other thread: {0}", i);
            Thread.Sleep(500);
        }
    }
}
```

The code creates a new thread which runs the ThreadJob method, and starts it. That thread counts from 0 to 9 fairly fast (about twice a second) while the main thread counts from 0 to 4 fairly slowly (about once a second). The way they count at different speeds is by each of them including a call to Thread.Sleep, which just makes the current thread sleep (do nothing) for the specified period of time. Between each count in the main thread we sleep for 1000ms, and between each count in the other thread we sleep for 500ms. Here are the results from one test run on my machine:

```
Main thread: 0
Other thread: 0
```
One important thing to note here is that although the above is very regular, that's by chance. There's nothing to stop the first "Other thread" line coming first, or the pattern being slightly off - Thread.Sleep is always going to be somewhat approximate, and there's no guarantee that the sleeping thread will immediately start running as soon as the sleep finishes. (It will become able to run, but another thread may be currently running, and on a single processor machine that means the thread which has just "woken up" will have to wait until the thread scheduler decides to give it some processor time before it next does anything.)

As with all delegates, there's nothing to restrict you to static methods, or methods within the class that the delegate is used from. You need to have access to the method, of course, and if you want to specify an instance method, you have to use a particular instance. Here's another version of the program above, using an instance method in a different class. If the Count method had been static, the value of the job variable would have been new Thread.Start(Counter.Count). Most examples given in this article use methods within the same class, but that's just for brevity and simplicity.

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

public class Test
{
    static void Main()
    {
        Counter foo = new Counter();
        ThreadStart job = new ThreadStart(foo.Count);
        Thread thread = new Thread(job);
        thread.Start();
        for (int i=0; i < 5; i++)
        {
            Console.WriteLine("Main thread: {0}". Format(i));
            Thread.Sleep(1000);
        }
    }
}

public class Counter
{
    public void Count()
    {
        for (int i=0; i < 10; i++)
        {
            Console.WriteLine("Other thread: {0}". Format(i));
            Thread.Sleep(500);
        }
    }
}
```
Passing Parameters to Threads

Often when you start a thread, you want to give it some parameters - usually some data it has to process, a queue to wait on, etc. The ThreadStart delegate doesn't take any parameters, so the information has to be stored somewhere else if you are going to create a new thread yourself. Typically, this means creating a new instance of a class, and using that instance to store the information. Often the class itself can contain the delegate used for starting the thread. For example, we might have a program which needs to fetch the contents of various URLs, and wants to do it in the background. You could write code like this:

```csharp
public class UrlFetcher
{
    string url
    public UrlFetcher (string url)
    {
        this.url = url;
    }
    public void Fetch()
    {
        // use url here
    }
}
... in a different class ...
UrlFetcher fetcher = new UrlFetcher (myUrl);
new Thread (new ThreadStart (fetcher.Fetch)).Start();
```

In some cases, you actually just wish to call a method in some class (possibly the currently executing class) with a specific parameter. In that case, you may wish to use a nested class whose purpose is just to make this call - the state is stored in the class, and the delegate used to start the thread just calls the "real" method with the appropriate parameter. (Note that the object on which to call the method in the first place will also be required as state unless the method is static.)

Using the thread pool instead

One alternative to starting the thread using a ThreadStart delegate is to use the thread pool, either using ThreadPool.QueueUserWorkItem or by calling a delegate asynchronously. Both of these are covered later on in a more detailed discussion of the thread pool, which also contains examples. Note that calling a delegate asynchronously allows you to specify multiple parameters, and those parameters are strongly typed.

.NET 2.0 solution 1: anonymous methods (with and without the thread pool)

One of the enhancements to C# in version 2.0 is anonymous methods. These allow you to specify blocks of code as methods within other methods, and use those methods as delegates. You can access variables (including local variables and parameters of the "outside" method) within the anonymous method. For example, using an anonymous method to fetch a URL using a normal ThreadStart delegate (and using inference of delegate type too):
ThreadStart starter = delegate { Fetch (myUrl); };  
new Thread (starter).Start();  

(This could have all been expressed within a single step, creating both the thread and the delegate in the same line of code, but I believe the above is more readable.) Here's similar code to use a WaitCallback and queue the job in ThreadPool:

WaitCallback callback = delegate (object state) { Fetch ((string)state); };  
ThreadPool.QueueUserWorkItem (callback, myUrl);

Note the way that the state is declared.

.NET 2.0 solution 2: ParameterizedThreadStart

In .NET 2.0, there is a new delegate, ParameterizedThreadStart, which takes a parameter of type object. You can create a thread using an instance of this delegate instead of just ThreadStart, and a new overload to Thread.Start allows you to specify the value to be passed to the new thread. This is simple, but only accepts a single parameter and isn't type-safe (just like the options when using thread pool threads). The earlier code could then be rewritten as:

```
[In some method or other]
Thread t = new Thread (new ParameterizedThreadStart(FetchUrl));
t.Start (myUrl);

[And the actual method...]
static void FetchUrl(object url)
{
    // use url here, probably casting it to a known type before use
}
```

Data races

Well, that's shown you how to create a new thread and start it. For a very few cases, it really is as simple as that - just occasionally, you end up with a thread which doesn't need access to any data other than its own (the counters in this case). Far more commonly, however, you need threads to access the same data, sooner or later - and that's where the problems start. Let's take a very simple program to start with:

```
using System;
using System.Threading;

public class Test
{
    static int count = 0;

    static void Main()
    {
        ThreadStart job = new ThreadStart(ThreadJob);
        Thread thread = new Thread (job);
        thread.Start();

        for (int i = 0; i < 5; i++)
        {
            count++;
        }
```
This is very straightforward - each of the threads just increments the `count` variable, and then the main thread displays the final value of `count` at the end. The only really new thing here is the call in the main thread to `Thread.Join`, which basically pauses the main thread until the other thread has completed.

So, the result should always be `Final count: 10`, right? Well, no. In fact, chances are that that will be the result if you run the above code - but it isn't guaranteed to be. There are two reasons for this - one fairly simple, and one much subtler. We'll leave the subtle one for the moment, and just consider the simple one.

The statement `count++` actually does three things: it reads the current value of `count`, increments that number, and then writes the new value back to the `count` variable. Now, if one thread gets as far as reading the current value, then the other thread takes over, does the whole increment operation, and then the first thread gets control again, its idea of the value of `count` is out of date - so it will increment the old value, and write that newly incremented (but wrong) value back into the variable.

The easiest way of showing this is by separating the three operations and introducing some `Sleep` calls into the code, just to make it more likely that the threads will clash heads, as it were. Note that introducing `Sleep` calls should never change the correctness of a program, in terms of threading - any thread can go to sleep at any time, basically. In other words, you can never rely on two operations both happening without another thread doing stuff in between. I've also put some diagnostics in to make it clearer what's happening. The "main" thread's activities appear on the left, while the "other" thread's activities are on the right. Here's the code:

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

class Test
{
    static int count = 0;

    static void Main()
    {
        ThreadStart job = new ThreadStart(ThreadJob);
        Thread thread = new Thread(job);
        thread.Start();
        for (int i = 0; i < 5; i++)
        {
            int tmp = count;
            Console.WriteLine("Read count={0}", tmp);
            Thread.Sleep(50);
            count++;
        }
        thread.Join();
        Console.WriteLine("Final count: {0}", count);
    }

    static void ThreadJob()
    {
        for (int i = 0; i < 5; i++)
        {
            count++;
        }
    }
}
```

This code will sometimes give a result of `Final count: 10`, and sometimes it will give a result of `Final count: 5`. The reason for this is that the `count++` operation is not atomic - it does not guarantee that both the read and write will happen atomically. This is a common problem in threading, and it can be difficult to solve. One way to solve it is to use a lock, but there are other ways as well.
```csharp
tmp++;    
Console.WriteLine("Incremented tmp to {0}", tmp);    
Thread.Sleep(20);    
count = tmp;    
Console.WriteLine("Written count={0}", tmp);    
Thread.Sleep(30);
}

thread.Join();    
Console.WriteLine("Final count: {0}", count);
}

static void ThreadJob()
{
    for (int i=0; i < 5; i++)
    {
        int tmp = count;    
        Console.WriteLine("\t\t\t\tRead count={0}". tmp);    
        Thread.Sleep(20);    
        tmp++;    
        Console.WriteLine("\t\t\t\tIncremented tmp to {0}", tmp);    
        Thread.Sleep(10);    
        count = tmp;    
        Console.WriteLine("\t\t\t\tWritten count={0}". tmp);    
        Thread.Sleep(40);
    }
}
```

... and here's one set of results I saw ...

<table>
<thead>
<tr>
<th>Read count=0</th>
<th>Incremented tmp to 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Written count=1</td>
<td></td>
</tr>
<tr>
<td>Read count=1</td>
<td></td>
</tr>
<tr>
<td>Incremented tmp to 2</td>
<td></td>
</tr>
<tr>
<td>Written count=2</td>
<td></td>
</tr>
<tr>
<td>Read count=3</td>
<td></td>
</tr>
<tr>
<td>Incremented tmp to 4</td>
<td></td>
</tr>
<tr>
<td>Written count=4</td>
<td></td>
</tr>
<tr>
<td>Read count=4</td>
<td></td>
</tr>
<tr>
<td>Incremented tmp to 5</td>
<td></td>
</tr>
<tr>
<td>Written count=5</td>
<td></td>
</tr>
<tr>
<td>Read count=5</td>
<td></td>
</tr>
</tbody>
</table>

Just looking at the first few lines shows exactly the nasty behaviour described before the code: the main thread has read the value 0, the other thread has incremented count to 1, and
then the main thread has incremented its "stale" value from 0 to 1, and written that value to the variable. The same thing happens a few more times, and the end result is that count is 6, instead of 10.

**Exclusive access - Monitor.Enter/Exit and the lock statement**

What we need to fix the problem above is to make sure that while one thread is in a read/increment/write operation, no other threads can try to do the same thing. This is where monitors come in. Every object in .NET has a (theoretical) monitor associated with it. A thread can enter (or acquire) a monitor only if no other thread has currently "got" it. Once a thread has acquired a monitor, it can acquire it more times, or exit (or release) it. The monitor is only available to other threads again once it has been exited as many times as it was entered. If a thread tries to acquire a monitor which is owned by another thread, it will block until it is able to acquire it. (There may be more than one thread trying to acquire the monitor, in which case when the current owner thread releases it for the last time, only one of the threads will acquire it - the other one will have to wait for the new owner to release it too.)

In our example, we want exclusive access to the count variable while we're performing the increment operation. First we need to decide on an object to use for locking. I'll discuss this choice in more detail later, but for the moment we'll introduce a new variable just for the purposes of locking: countLock. This is initialised to be a reference a new object, and thereafter is never changed. It's important that it's not changed - otherwise one thread would be locking on one object's monitor, and another object might be locking on a different object's monitor, so they could interfere with each other just like they did before.

We then simply need to put each increment operation in a Monitor.Enter and Monitor.Exit pair:

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

public class Test
{
    static int count = 0;
    static readonly object countLock = new object();

    static void Main()
    {
        ThreadStart job = new ThreadStart(ThreadJob);
        Thread thread = new Thread(job);
        thread.Start();

        for (int i = 0; i < 5; i++)
        {
            Monitor.Enter(countLock);
            int tmp = count;
            Console.WriteLine("Read count={0}", tmp);
            Thread.Sleep(50);
            tmp++;
            Console.WriteLine("Incremented tmp to {0}", tmp);
            Thread.Sleep(20);
            count = tmp;
            Console.WriteLine("Written count={0}", tmp);
            Monitor.Exit(countLock);
            Thread.Sleep(30);
        }
        thread.Join();
    }
}
```

11
Console.WriteLine ("Final count: {0}", count);
}

static void ThreadJob()
{
    for (int i = 0; i < 5; i++)
    {
        Monitor.Enter (countLock);
        int tmp = count;
        Console.WriteLine ("\t\t\t\tRead count={0}", tmp);
        Thread.Sleep (20);
        tmp++;
        Console.WriteLine ("\t\t\t\tIncremented tmp to {0}", tmp);
        Thread.Sleep (10);
        count = tmp;
        Console.WriteLine ("\t\t\t\tWritten count={0}", tmp);
        Monitor.Exit (countLock);
        Thread.Sleep (40);
    }
}

The results look a lot better this time:

<table>
<thead>
<tr>
<th>Read count=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incremented tmp to 1</td>
</tr>
<tr>
<td>Written count=1</td>
</tr>
<tr>
<td>Read count=1</td>
</tr>
<tr>
<td>Incremented tmp to 2</td>
</tr>
<tr>
<td>Written count=2</td>
</tr>
<tr>
<td>Read count=2</td>
</tr>
<tr>
<td>Incremented tmp to 3</td>
</tr>
<tr>
<td>Written count=3</td>
</tr>
<tr>
<td>Read count=3</td>
</tr>
<tr>
<td>Incremented tmp to 4</td>
</tr>
<tr>
<td>Written count=4</td>
</tr>
<tr>
<td>Read count=4</td>
</tr>
<tr>
<td>Incremented tmp to 5</td>
</tr>
<tr>
<td>Written count=5</td>
</tr>
<tr>
<td>Read count=5</td>
</tr>
<tr>
<td>Incremented tmp to 6</td>
</tr>
<tr>
<td>Written count=6</td>
</tr>
<tr>
<td>Read count=6</td>
</tr>
<tr>
<td>Incremented tmp to 7</td>
</tr>
<tr>
<td>Written count=7</td>
</tr>
<tr>
<td>Read count=7</td>
</tr>
<tr>
<td>Incremented tmp to 8</td>
</tr>
<tr>
<td>Written count=8</td>
</tr>
<tr>
<td>Read count=8</td>
</tr>
<tr>
<td>Incremented tmp to 9</td>
</tr>
<tr>
<td>Written count=9</td>
</tr>
<tr>
<td>Read count=9</td>
</tr>
<tr>
<td>Incremented tmp to 10</td>
</tr>
<tr>
<td>Written count=10</td>
</tr>
<tr>
<td>Final count: 10</td>
</tr>
</tbody>
</table>

The fact that the increments were strictly alternating here is just due to the sleeps - in a more normal system there could be two increments in one thread, then three in another, etc. The important thing is that they would always be thread-safe: each increment would be isolated from each other increment, with only one being processed at a time.

There's a chance - a tiny chance, but a chance nonetheless - that the code above would hang, however. If part of the increment operation (one of the calls to Console.WriteLine, for in-
stance) threw an exception, the thread would still own the monitor, so the other thread would never be able to acquire it and move on. The obvious solution to this (if you're used to exception handling, at least) is to put the call to Monitor.Exit in a finally block, with everything after the call to Monitor.Enter in a try block. Just like the using statement which puts a call to Dispose in a finally block automatically, C# provides the lock statement to call Monitor.Enter and Monitor.Exit with a try/finally block automatically. This makes it much easier to get synchronization right, as you don't end up having to check for "balanced" calls to Enter and Exit everywhere. It also makes sure that you don't try to release a monitor you don't own: in the code we had above, if we changed the value of countLock to be a reference to a different object within the increment operation, we'd have failed to release the monitor we owned, and tried to release a monitor we didn't own - which would (in theory) have caused a SynchronizationLockException. (In fact, the exception wouldn't have been thrown because there's a bug in the framework in version 1.0/1.1, but that's another story.) The lock statement automatically takes a copy of the reference you specify, and calls both Enter and Exit with it. (In the example above, and everywhere else in this article, variables used to hold locks are declared as read-only. I have yet to come across a good reason to change what a particular piece of code locks on.)

So, we can rewrite our previous code into the somewhat clearer and more robust code below:

```csharp
using System;
using System.Threading;
public class Test
{
    static int count = 0;
    static readonly object countLock = new object();
    static void Main()
    {
        ThreadStart job = new ThreadStart(ThreadJob);
        Thread thread = new Thread(job);
        thread.Start();
        for (int i = 0; i < 5; i++)
        {
            lock (countLock)
            {
                int tmp = count;
                Console.WriteLine("Read count={0}", tmp);
                Thread.Sleep(50);
                tmp++;
                Console.WriteLine("Incremented tmp to {0}" , tmp);
                Thread.Sleep(20);
                count = tmp;
                Console.WriteLine("Written count={0}" , tmp);
            }
            Thread.Sleep(30);
        }
        thread.Join();
        Console.WriteLine("Final count: {0}", count);
    }
    static void ThreadJob()
    {
        for (int i = 0; i < 5; i++)
        {
            lock (countLock)
            {
                int tmp = count;
```
Deadlocks

The second major problem of multi-threading is that of deadlocks. Simply put, this is when two threads each hold a monitor that the other one wants. Each blocks, waiting for the monitor that it's waiting for to be released - and so the monitors are never released, and the application hangs (or at least those threads involved in the deadlock hang).

Here's an example of a program exhibiting deadlock:

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

public class Test
{
    static readonly object firstLock = new object();
    static readonly object secondLock = new object();

    static void Main()
    {
        new Thread(new ThreadStart(ThreadJob)).Start();
        // Wait until we're fairly sure the other thread
        // has grabbed firstLock
        Thread.Sleep(500);
        Console.WriteLine("Locking secondLock");
        lock (secondLock)
        {
            Console.WriteLine("Locked secondLock");
            Console.WriteLine("Locking firstLock");
            lock (firstLock)
            {
                Console.WriteLine("Locked firstLock");
                // Wait until we're fairly sure the first thread
                // has grabbed secondLock
                Thread.Sleep(1000);
                Console.WriteLine("Locking secondLock");
                lock (secondLock)
                {
                }
            }
            Console.WriteLine("Released firstLock");
        }
        Console.WriteLine("Released secondLock");
    }

    static void ThreadJob()
    {
        Console.WriteLine("\t\t\tLocking firstLock");
        lock (firstLock)
        {
            Console.WriteLine("\t\t\tLocked firstLock");
            // Wait until we're fairly sure the first thread
            // has grabbed secondLock
            Thread.Sleep(1000);
            Console.WriteLine("\t\t\tLocking secondLock");
            lock (secondLock)
            {
            } // If the second thread has released secondLock
        }
    }
}
```
And the results:

(You'll need to hit Ctrl-C or something similar to kill the program.) As you can see, each thread grabs one lock and then tries to grab the other. The calls to `Thread.Sleep` have been engineered so that they will try to do so at inopportune times, and deadlock.

Deadlocks can be a real pain in the neck to debug - they're hard to diagnose, they can crop up seemingly randomly (i.e. they're hard to reproduce) and once you've found out why there's a deadlock, it's not always obvious what the best course of action is. Locking strategies always need to be treated very carefully; you can't just start removing all the lock statements from your code, or you'll end up with data races etc.

The solution (which is easier to describe than to achieve) is to make sure that you always take out locks in the same order: in the code above, you might decide to never acquire `secondLock` unless you already had `firstLock`. Within one class, it's relatively straightforward to achieve this. It's when you have calls between classes (including delegates, where you don't really know what you're calling) that things get somewhat hairy. If possible, you should avoid making method calls outside your own class within a lock, unless you know pretty definitely that that method won't itself need to lock anything.

**More Monitor methods**

If you looked at the documentation for `Monitor.Enter` and `Monitor.Exit`, you will no doubt have seen that there are various other methods in the `Monitor` class, all of which can be useful at different times.

`Monitor.TryEnter` is the easiest one to describe - it simply attempts to acquire a lock, but doesn't block (or only blocks for a given period of time) if the lock cannot be acquired.

The other methods (`Wait`, `Pulse` and `PulseAll`) all go together. They're used to signal between threads. The idea is that one thread calls `Wait`, which makes it block until another thread calls `Pulse` or `PulseAll`. The difference between `Pulse` and `PulseAll` is how many threads are woken up: `Pulse` only wakes up a single waiting thread; `PulseAll` wakes up all threads waiting on that monitor. That doesn't mean they'll all instantly start running, however: in order to call any of these three methods, the thread has to own the monitor of the object reference it passes in as a parameter. When calling `Wait`, the monitor is released, but then
needs to be reacquired before the thread will actually run. This means blocking again until the thread which calls `Pulse` or `PulseAll` releases the monitor (which it must have in order to pulse the monitor in the first place) - and if multiple threads are woken up, they'll all try to acquire the monitor, which only one can have at a time, of course. Just to repeat: calling `Wait` unlocks the monitor you're waiting on. This is an important point, because otherwise the code looks like it'll just deadlock!

The most common use of these methods is in producer/consumer relationships, where one thread is putting work items on a queue, and another thread is taking them off. The consumer thread typically takes items off the list until it's empty, then waits on a lock. The producer thread pulses the lock when it adds an item to the list (or, if you're worried about efficiency, it can pulse the lock only when it adds an item to a previously-empty list). Here's a sample:

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

class Test
{
    public class ProducerConsumer
    {
        readonly object listLock = new object();
        Queue queue = new Queue();

        public void Produce(object o)
        {
            lock (listLock)
            {
                queue.Enqueue(o);
                // We always need to pulse, even if the queue wasn't
                // empty before. Otherwise, if we add several items
                // in quick succession, we may only pulse once, waking
                queue.Pulse();
            }
        }

        public object Consume()
        {
            return queue.Dequeue();
        }
    }

    static void Main()
    {
        queue = new ProducerConsumer();
        new Thread(new ThreadStart(ConsumerJob)).Start();

        Random rng = new Random(0);
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine("Producing {0}", i);
            queue.Produce(i);
            Thread.Sleep(rng.Next(1000));
        }
    }

    static void ConsumerJob()
    {
        // Make sure we get a different random seed from the
        // first thread
        Random rng = new Random(1);
        // We happen to know we've only got 10
        // items to receive
        for (int i = 0; i < 10; i++)
        {
            object o = queue.Consume();
            Console.WriteLine("Consuming {0}", o);
            Thread.Sleep(rng.Next(1000));
        }
    }
}
```

threads
   // a single thread up, even if there are multiple
   // waiting for items.
   Monitor.Pulse(listLock);
}
}

public object Consume()
{
   lock (listLock)
   {
      // If the queue is empty, wait for an item to be added
      // Note that this is a while loop, as we may be pulsed
      // but not wake up before another thread has come in
      and
      // consumed the newly added object. In that case, we'll
      // have to wait for another pulse.
      while (queue.Count==0)
      {
         // This releases listLock, only reacquiring it
         // after being woken up by a call to Pulse
         Monitor.Wait(listLock);
      }
      return queue.Dequeue();
   }
}

Here are some results I got:

| Producing 0 | Consuming 0 |
| Producing 1 | Consuming 1 |
| Producing 2 | Consuming 2 |
| Producing 3 | Consuming 3 |
| Producing 4 | Consuming 4 |
| Producing 5 | Consuming 5 |
| Producing 6 | Consuming 6 |
| Producing 7 | Consuming 7 |
| Producing 8 | Consuming 8 |
| Producing 9 | Consuming 9 |

Now, there's nothing stopping you from having more than one consumer or producer in the above. Everything will play nicely, and each produced object will only be consumed once, and will be consumed (almost) immediately if there are any consumers waiting for work.

The reason for having both **Pulse** and **PulseAll** is for different situations, where you're waiting on different conditions. If either there'll only be one thread waiting, or (as is the case above) any thread can consume any produced object, you can just use **Pulse**. If there are several threads waiting on the object, that ends up being more efficient than **PulseAll** - there's no point in waking up a bunch of threads if you know that only one of them is going to be able to make progress, and that it doesn't matter which you wake up. Sometimes, however, different threads are waiting on different conditions, but all waiting on the same monitor. In
that case, you need to use `PulseAll` so that you make sure that the thread which is waiting for whatever condition has just occurred is able to notice it and make progress.

Note that using these methods can easily lead to deadlock - if thread A holds locks X and Y, and waits on Y, but thread B needs to acquire lock X before acquiring and then pulsing Y, thread B won't be able to do anything. Only the lock which is waited on is released, not all the locks the waiting thread owns. Usually you should ensure that prior to waiting, a thread only owns the lock it's going to wait on. Sometimes this isn't possible, but in those cases you should think extra carefully about how everything is going to work.

**WaitHandles - Auto/ManualResetEvent and Mutex**

`Monitor.Wait/Pulse` isn't the only way of waiting for something to happen in one thread and telling that thread that it's happened in another. Win32 programmers have been using various other mechanisms for a long time, and these are exposed by the `AutoResetEvent`, `ManualResetEvent` and `Mutex` classes, all of which derive from `WaitHandle`. All of these classes are in the `System.Threading` namespace. (The Win32 Semaphore mechanism does not have a managed wrapper in .NET 1.1. It's present in .NET 2.0, but if you need to use it before then, you could either wrap it yourself using P/Invoke, or write your own counting semaphore class.)

Some people may be surprised to learn that using these classes can be significantly slower than using the various `Monitor` methods. I believe this is because going "out" of managed code into native Win32 calls and back "in" again is expensive compared with the entirely managed view of things which `Monitor` provides.

`WaitHandle` itself only exposes a few useful instance methods/properties:

- `WaitOne()` - used to wait for the handle to be free/signalled. The exact meaning of this depends on the concrete type being used (`Mutex`, `AutoResetEvent` or `ManualResetEvent`).
- `Close()`/`Dispose()` - used to release the resources used by the handle.
- `Handle` - used to get the native handle being wrapped. Most developers won't need to use this.

In addition, it has two useful static methods which deal with sets of `WaitHandles`:

- `WaitAny()` - used to wait for any of the handles in a set to be free/signalled.
- `WaitAll()` - used to wait for all of the handles in a set to be free/signalled.

All of the `WaitXXX()` methods have overloads allowing you to specify a timeout and whether or not to exit the synchronization domain. Frankly, it's unclear exactly what a synchronization domain is - there don't seem to be any clear and free articles about it on the net or in MSDN. If you know what they are, please mail me at skeet@pobox.com (skeet@pobox.com) so I can include the information in this article - or leave it as "definitely beyond the scope of this article".

**Auto/ManualResetEvent**

The two "event" classes (which are entirely different from .NET events - don't get the two confused) come as a sort of pair, and are very similar. You can think of them like doors -
when they're in the "signalled" (or "set") state they're open, and when they're in the "non-
signalled" (or "reset") state, they're closed. A call to \texttt{WaitOne()} waits for the door to be
opened so the thread can "go through it" in some sense. The difference between the two
classes is that an \texttt{AutoResetEvent} will reset itself to the non-signalled state immediately after
a call to \texttt{WaitOne()} - it's as if anyone going through the door closes it behind them. With a
\texttt{ManualResetEvent}, you have to tell the thread to reset it (close the door) when you want to
make calls to \texttt{WaitOne()} block again. Both classes can manually be set or reset at any time,
by any thread, using the \texttt{Set} and \texttt{Reset} methods, and can be created in the signalled/set or
non-signalled/reset state. (These methods return a boolean value saying whether or not they
were successful, but the documentation doesn't state why they might fail.)

Here's some sample code which simulates 10 runners. Each runner is passed a \texttt{ManualRe-
setEvent} which is initially non-signalled. When the runner completes the race, it signals the
event. The main thread uses \texttt{WaitHandle.WaitAny} to wait for the first runner to finish, and
uses the value returned by the method to say who won the race. It then uses \texttt{WaitHan-
dle.WaitAll} to wait for everyone to finish. Note that if we'd used \texttt{AutoResetEvent} instead,
we'd have to call \texttt{Set} on the event of the winner, as it would have been reset when we de-
tected it being set with the call to \texttt{WaitAny}.

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

class Test
{
    static void Main()
    {
        ManualResetEvent[] events = new ManualResetEvent[10];
        for (int i = 0; i < events.Length; i++)
        {
            events[i] = new ManualResetEvent(false);
            Runner r = new Runner(events[i], i);
            new Thread(new ThreadStart(r.Run)).Start();
        }
        int index = WaitHandle.WaitAny(events);
        Console.WriteLine("***** The winner is {0} *****", index);
        WaitHandle.WaitAll(events);
        Console.WriteLine("All finished!");
    }
}

class Runner
{
    static readonly object rngLock = new object();
    static Random rng = new Random();

    ManualResetEvent ev;
    int id;
    internal Runner (ManualResetEvent ev, int id)
    {
        this.ev = ev;
        this.id = id;
    }

    internal void Run()
    {
        for (int i = 0; i < 10; i++)
        {
```
Mutex

Whereas Auto/ManualResetEvent have a lot in common with using Monitor.Wait/Pulse, Mutex has even more in common with Monitor.Enter/Exit. A mutex has a count of the number of times it's been acquired, and a thread which is the current owner. If the count is zero, it has no owner and it can be acquired by anyone. If the count is non-zero, the current owner can acquire it however many times they like without blocking, but any other thread has to wait until the count becomes zero before they can acquire it. The WaitXXX() methods are used to acquire the mutex, and ReleaseMutex() is used by the owner thread to decrease the count by one. Only the owner can decrease the count.

So far, so much like Monitor. The difference is that a Mutex is a cross-process object - the same mutex can be used in many processes, if you give it a name. A thread in one process can wait for a thread in another process to release the mutex, etc. When you construct a named mutex, you should be careful about making assumptions as to whether or not you will be able to acquire initial ownership of it. Fortunately, there is a constructor which allows the code to detect whether the system has created a whole new mutex or whether it's used an existing one. If the constructor requested initial ownership, it will only have been granted it if it created a new mutex - even if the existing mutex can immediately be acquired.

Mutex names should start with either "Local\" or "Global\" to indicate whether they should be created in the local or global namespace respectively. (I believe that local is the default, but why take the risk? Make it explicit in the name.) If you create a mutex in the global namespace, it is shared with other users logged into the same machine. If you create a mutex in the local namespace, it is specific to the current user. Make sure you pick a suitably unique name so you don't clash with other programs.

To be honest, I think the principle use that mutexes will be put to in .NET is the one mentioned earlier - detecting that another instance of an application is already running. Most people don't need inter-process communication on this kind of level. The other use is to enable you to block until either one or all of a set of WaitHandles is released. For other purposes, where Monitor is good enough, I suggest using that - especially as C# has the lock statement specifically to support it. Here's an example of detecting a running application, however:

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

class Test
{
    static void Main()
    {
```
bool firstInstance;
using (Mutex mutex = new Mutex(true,
@"Global\Jon.Skeet.MutexTestApp",   out firstInstance))
{
    if (!firstInstance)
    {
        Console.WriteLine("Other instance detected; aborting.");
        return;
    }
    Console.WriteLine("We're the only instance running - yay!");
    for (int i = 0; i < 10; i++)
    {
        Console.WriteLine(i);
        Thread.Sleep(1000);
    }
}

Run the example in two different console windows - one will count to ten slowly; the other will abort after it detects that the other application instance is running. Note the using statement around the mutex: this should extend across the whole of the application's execution, otherwise another instance would be able to create a new mutex with the same name, after the old one had been destroyed. For instance, suppose you use a local variable without a using statement, like this:

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

class Test
{
    static void Main()
    {
        bool firstInstance;
        // Bad code - do not use!
        Mutex mutex = new Mutex(true,
            @"Global\Jon.Skeet.MutexTestApp",   out firstInstance);
        if (!firstInstance)
        {
            Console.WriteLine("Other instance detected; aborting.");
            return;
        }
        Console.WriteLine("We're the only instance running - yay!");
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine(i);
            Thread.Sleep(1000);
        }
    }
}
```

In that case, you'd probably find that everything would work fine under debug, where the GC is very conservative about what it collects. When not running under the debugger, however,
the GC can tell that the mutex variable isn't used after its initial assignment, so for the main duration of the app, it can be garbage collected at any time - and that destroys the mutex! The using statement shown earlier is only one way round this. You could make it a static variable instead, or use GC.KeepAlive(mutex); at the end of the method to make sure that the GC doesn't ignore the variable.

**Volatility, Atomicity and Interlocking**

Earlier, when introducing the topic of data races, I mentioned that there was a more subtle reason why the first attempt at the code wasn't thread-safe. It's to do with volatility and data caching. Here's a sample which will make explaining the topic somewhat easier:

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

class Test
{
    static bool stop;

    static void Main()
    {
        ThreadStart job = new ThreadStart(ThreadJob);
        Thread thread = new Thread(job);
        thread.Start();

        // Let the thread start running
        Thread.Sleep(2000);

        // Now tell it to stop counting
        stop = true;
    }

    static void ThreadJob()
    {
        int count = 0;
        while (!stop)
        {
            Console.WriteLine("Extra thread: count {0}", count);
            Thread.Sleep(100);
            count++;
        }
    }
}
```

Now, this code is fairly simple. We have a boolean variable (stop) which is polled by the new thread - it will keep counting until it notices that stop is true. In the main thread, we pause for a couple of seconds and then set stop to true.

So, the new thread should count for a couple of seconds and then stop, right? Well, in fact that's what will almost certainly happen if you run the code, but it's not guaranteed. The while loop in the new thread could keep running forever, never really checking whether or not the stop variable has been set to true. If that sounds bizarre to you, welcome to the weird and wonderful world of memory models.

Memory in modern computers is a very complicated business, with registers, multiple levels of cache, and multiple processors sharing main memory but possibly not caches, etc. The idea that there's just a single chunk of memory which is accessed in a simple way is very handy for programmers, but lousy for performance. In addition, if a processor knows it might have to read a bit of memory "soon", it could decide to read it early, etc. Hardware manufacturers and
compiler writers (including JIT-compiler writers) have worked very hard to make fast code easy to write. The memory model of a platform is the specification of what developers can do safely without knowing too much about the details of the hardware the platform is running on. This means (in our case) that you can run .NET code on any CPU which has a CLR, and so long as you follow the rules of the memory model, you should be okay - however "strong" or "weak" the memory model of the hardware itself is. (A "strong" memory model is one which guarantees a lot; a "weak" model is one which doesn't guarantee much at all, often giving better performance but requiring more work on the part of the developer. "x86" processors have a stronger memory model than the CLR itself, which is one reason problems such as seeing stale data are relatively hard to demonstrate.)

The memory model in .NET talks about when reads and writes "actually" happen compared with when they occur in the program's instruction sequence. Reads and writes can be reordered in any way which doesn't violate the rules given by the memory model. As well as "normal" reads and writes there are volatile reads and writes. Every read which occurs after a volatile read in the instruction sequence occurs after the volatile read in the memory model too - they can't be reordered to before the volatile read. A volatile write goes the other way round - every write which occurs before a volatile write in the instruction sequence occurs before the volatile write in the memory model too.

Don't worry if the above doesn't make much sense - the resource section at the end of this page contain a few links which should help you out a bit if you want to really understand it thoroughly, but the rule is pretty simple: when you have access to shared data, you need to make sure you read fresh data and write any changes back in a timely manner. There are two ways of doing this - volatile variables, and using lock again.

A variable which is declared volatile uses volatile reads and writes for all its accesses. You can only declare a variable to be volatile if it's one of the following types: a reference type, byte, sbyte, short, ushort, int, uint, char, float, or bool, or an enumeration with a base type of byte, sbyte, short, ushort, int, or uint. If you're only interested in sharing a single piece of data, and it's one of the above types, then using a volatile variable is probably the easiest way to go. Note, however, that for a reference type, only the access to the variable itself is volatile - if you write to something within the instance the reference refers to, that write won't be volatile. Personally I don't use volatile variables much, preferring the other approach: locking.

We've already seen how locking is used to limit access to a single thread at a time. It also has another side effect: a call to Monitor.Enter performs an implicit volatile read, and a call to Monitor.Exit performs an implicit volatile write. The two effects combine nicely: if you're reading, you perform a volatile read, so you know that your next read will be from main memory - and because you're then in a lock, you know that nothing else will be trying to change the value. Similarly, if you're writing, you know that nothing else will be trying to read the value between you writing it and the volatile write, so nothing will see an old value - assuming all access to the variable is covered with the same lock, of course. If you lock using one monitor for some access to a variable, and another monitor for other access to the same variable, the volatility and the locking won't mesh quite as nicely, and you won't get as strong a guarantee of freshness of data. Fortunately, there's very little reason why you'd even want to try this.

So, to get back to our sample program: it's currently flawed because the new thread could read the value of stop once (perhaps into a register) and then never bother reading it from main
memory. Alternatively, it could always read it from main memory, but the original thread
may never write it there. To fix it, we could either just make stop volatile, or we could use a
lock. The volatile solution is simple - just add the keyword volatile to the variable declaration,
and you're done. The locking solution requires a bit more effort, and I'd make things
slightly easier by introducing a property to do the locking. So long as you then only refer to
the variable via the property, you don't need to write the lock all over the place. Here's the full
code with a property which locks:

```csharp
using System.Threading;

public class Test
{
    static bool stop;
    static readonly object stopLock = new object();
    static bool Stop
    {
        get
        {
            lock (stopLock)
            {
                return stop;
            }
        }
        set
        {
            lock (stopLock)
            {
                stop = value;
            }
        }
    }

    static void Main()
    {
        ThreadStart job = new ThreadStart(ThreadJob);
        Thread thread = new Thread(job);
        thread.Start();

        // Let the thread start running
        Thread.Sleep(2000);

        // Now tell it to stop counting
        Stop = true;
    }

    static void ThreadJob()
    {
        int count = 0;
        while (!Stop)
        {
            Console.WriteLine("Extra thread: count {0}", count);
            Thread.Sleep(100);
            count++;
        }
    }
}
```

Unfortunately there's no way of getting the compiler to complain if you access stop directly,
so you do need to be careful to always use the property.

As of .NET 1.1, there is another way of achieving a memory barrier: Thread.MemoryBarrier(). In future versions there may well be separate method calls for
"write" memory barriers and "read" memory barriers. I would advise steering well clear of
these unless you're an expert - even the experts seem to argue amongst themselves about what's needed when. (Read the links in the resource section for more information.)

Just to reiterate: working things out to be as efficient as possible but still absolutely correct is hard. Fortunately, using locks whenever you want to access shared data is relatively easy and correct by the model. Stick to the simple way of doing things and you don't need to worry about all this too much.

**Atomicity**

This section is here almost as an aside - because if you're writing thread-safe code to start with, atomicity isn't particularly relevant to you. However, it's a good idea to clear up what atomicity is all about, because many people believe it's to do with volatility and the like.

An operation is *atomic* if it is indivisible - in other words, nothing else can happen in the middle. So, with an atomic write, you can't have another thread reading the value half way through the write, and ending up "seeing" half of the old value and half of the new value. Similarly, with an atomic read, you can't have another thread changing the value half way through the read, ending up (again) with a value which is neither the old value nor the new value.

The CLR guarantees that for types which are no bigger than the size of a *native* integer, if the memory is properly aligned (as it is by default - if you specify an explicit layout, that could change the alignment), reads and writes are atomic. In other words, if one thread is changing a properly aligned `int` variable's value from 0 to 5 and another thread is reading the variable's value, it will only ever see 0 or 5 - never 1 or 4, for instance. For a `long`, however, on a 32-bit machine, if one thread is changing the value from 0 to 0x0123456789abcdef, there's no guarantee that another thread won't see the value as 0x0123456700000000 or 0x0000000089abcdef. You'd have to be unlucky - but writing thread-safe code is all about taking luck out of the equation.

Fortunately, using the techniques I've already mentioned, you rarely need to worry about atomicity at all. Certainly if you use locking, you don't need to worry as you're already making sure that a read and a write can't overlap. If you use volatile variables there may be a slight chance of problems, as although every type which can be volatile *can* be atomically written and read, if the alignment of the variable is wrong, you could still get non-atomic reads and writes - the volatility doesn't provide any extra guarantees. Just another reason to use locking :)

**A shortcut for some cases: the `Interlocked` class**

Just occasionally, locking is a bit too much effort (and possibly too much of a performance hit) for doing very simple operations such as counting. The `Interlocked` class provides a set of methods for performing atomic changes: exchanges (optionally performing a comparison first), increments and decrements. The `Exchange` and `CompareExchange` methods act on variables of type `int`, `object` or `float`; the `Increment` and `Decrement` methods act on variables of type `int` or `long`.

Frankly I've never used the class myself in production code - I prefer to take the simple approach of using one tool (locking) to sort out all my volatility, atomicity and race-avoidance
problems. However, that does come at the cost of a bit of performance. While that's never bothered me overly, if you're writing code which needs to perform at its absolute fastest, you may want to consider using this class as a fast way of performing the very specific operations it provides. Here's a sample - the first example I used to illustrate data races, rewritten to be thread-safe using the `Interlocked` class:

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

public class Test
{
    static int count = 0;
    static void Main()
    {
        ThreadStart job = new ThreadStart(ThreadJob);
        Thread thread = new Thread(job);
        thread.Start();
        for (int i = 0; i < 5; i++)
        {
            Interlocked.Increment(ref count);
        }
        thread.Join();
        Console.WriteLine("Final count: {0}", count);
    }

    static void ThreadJob()
    {
        for (int i = 0; i < 5; i++)
        {
            Interlocked.Increment(ref count);
        }
    }
}
```

### Threading in Windows Forms

One of the issues which frequently comes up in newsgroups is how to handle threading in a UI. There are two golden rules for Windows Forms:

1) Never invoke any method or property on a control created on another thread other than `Invoke`, `BeginInvoke`, `EndInvoke` or `CreateGraphics`, and `InvokeRequired`.
   Each control is effectively bound to a thread which runs its message pump. If you try to access or change anything in the UI (for example changing the `Text` property) from a different thread, you run a risk of your program hanging or misbehaving in other ways. You may get away with it in some cases, but only by blind luck. Fortunately, the `Invoke`, `BeginInvoke` and `EndInvoke` methods have been provided so that you can ask the UI thread to call a method for you in a safe manner.

2) Never execute a long-running piece of code in the UI thread.
   If your code is running in the UI thread, that means no other code is running in that thread. That means you won't receive events, your controls won't be repainted, etc. This is a very Bad Thing. You can execute long-running code and periodically call `Application.DoEvents()`, and this is the natural thing for many VB programmers to wish to do - but I'd advise against it. It means you have to consider re-entrancy issues etc, which I believe are harder to diagnose and fix than "normal" threading problems. You have to judge when to call `DoEvents`, and you can't use anything which might
block (network access, for instance) without risking an unresponsive UI. I believe there are message pumping issues in terms of COM objects as well, but I don't have details of them (and I frankly wouldn't understand them fully anyway).

So, if you have a piece of long-running code which you need to execute, you need to create a new thread (or use a thread pool thread if you prefer) to execute it on, and make sure it doesn't directly try to update the UI with its results. The thread creation part is the same as any other threading problem, and we've addressed that before. The interesting bit is going the other way - invoking a method on the UI thread in order to update the UI.

There are two different ways of invoking a method on the UI thread, one synchronous (Invoke) and one asynchronous (BeginInvoke). They work in much the same way - you specify a delegate and (optionally) some arguments, and a message goes on the queue for the UI thread to process. If you use Invoke, the current thread will block until the delegate has been executed. If you use BeginInvoke, the call will return immediately. If you need to get the return value of a delegate invoked asynchronously, you can use EndInvoke with the IAsyncResult returned by BeginInvoke to wait until the delegate has completed and fetch the return value.

There are two options when working out how to get information between the various threads involved. The first option is to have state in the class itself, setting it in one thread, retrieving and processing it in the other (updating the display in the UI thread, for example). The second option is to pass the information as parameters in the delegate. Using state somewhere is necessary if you're creating a new thread rather than using the thread pool - but that doesn't mean you have to use state to return information to the UI. On the other hand, creating a delegate with lots of parameters often feels clumsy, and is in some ways less efficient than using a simple MethodInvoker or EventHandler delegate. These two delegates are treated in a special (fast) manner by Invoke and BeginInvoke. MethodInvoker is just a delegate which takes no parameters and returns no value (like ThreadStart), and EventHandler takes two parameters (a sender and an EventArgs parameter and returns no value. Note, however, that if you pass an EventHandler delegate to Invoke or BeginInvoke then even if you specify parameters yourself, they are ignored - when the method is invoked, the sender will be the control you have invoked it with, and the EventArgs will be EventArgs.Empty.

Here is an example which shows several of the above concepts. Notes are provided after the code.

```csharp
using System;
using System.Threading;
using System.Windows.Forms;
using System.Drawing;

class Test : Form
{
    delegate void StringParameterDelegate (string value);
    Label statusIndicator;
    Label counter;
    Button button;

    /// <summary>
    /// Lock around target and currentCount
    /// </summary>
    readonly object stateLock = new object();
    int target;
    int currentCount;
}
```
Random rng = new Random();

test()
{
    size = new Size (180, 120);
    text = "Test";

    label lbl = new Label();
    lbl.text = "Status:";
    lbl.size = new Size (50, 20);
    lbl.location = new Point (10, 10);
    controls.add(lbl);

    lbl = new Label();
    lbl.text = "Count:";
    lbl.size = new Size (50, 20);
    lbl.location = new Point (10, 34);
    controls.add(lbl);

    statusIndicator = new Label();
    statusIndicator.size = new Size (100, 20);
    statusIndicator.location = new Point (70, 10);
    controls.add(statusIndicator);

    counter = new Label();
    counter.size = new Size (100, 20);
    counter.location = new Point (70, 34);
    controls.add(counter);

    button = new Button();
    button.text = "Go";
    button.size = new Size (50, 20);
    button.location = new Point (10, 58);
    controls.add(button);
    button.click += new EventHandler (StartThread);
}

void StartThread (object sender, EventArgs e)
{
    button.enabled = false;
    lock (stateLock)
    {
        target = rng.next (100);
    }
    thread t = new Thread (new ThreadStart (ThreadJob));
    t.isBackground = true;
    t.start();
}

void ThreadJob()
{
    MethodInvoker updateCounterDelegate = new MethodInvoker (UpdateCount);
    int localTarget;
    lock (stateLock)
    {
        localTarget = target;
    }
    UpdateStatus ("Starting");
    lock (stateLock)
    {
        currentCount = 0;
    }
    invoke (updateCounterDelegate);
    // Pause before starting
    Thread.Sleep (500);
    UpdateStatus ("Counting");
    for (int i = 0; i < localTarget; i++)
    {
lock (stateLock)
{
    currentCount = i;
}
// Synchronously show the counter
Invoke (updateCounterDelegate);
Thread.Sleep(100);
}
UpdateStatus("Finished");
Invoke (new MethodInvoker(EnableButton));
}

void UpdateStatus(string value)
{
    if (InvokeRequired)
    {
        // We're not in the UI thread, so we need to call BeginInvoke
        BeginInvoke(new StringParameterDelegate(UpdateStatus),
                    new object[]{value});
        return;
    }
    // Must be on the UI thread if we've got this far
    statusIndicator.Text = value;
}

void UpdateCount()
{
    int tmpCount;
    lock (stateLock)
    {
        tmpCount = currentCount;
    }
    counter.Text = tmpCount.ToString();
}

void EnableButton()
{
    button.Enabled = true;
}

static void Main()
{
    Application.Run (new Test());
}

Notes:

- State is used to tell the worker thread what number to count up to.
- A delegate taking a parameter is used to ask the UI to update the status label. The worker thread's principal method actually just calls UpdateStatus, which uses InvokeRequired to detect whether or not it needs to "change thread". If it does, it then calls BeginInvoke to execute the same method again from the UI thread. This is quite a common way of making a method which interacts with the UI thread-safe. The choice of BeginInvoke rather than Invoke here was just to demonstrate how to invoke a method asynchronously. In real code, you would decide based on whether you needed to block to wait for the access to the UI to complete before continuing or not. In practice, I believe it's quite rare to actually require UI access to complete first, so I tend to use BeginInvoke instead of Invoke. Another approach might be to have a property which did the appropriate invoking when necessary. That's easier to use from the client code, but slightly harder work in that you would either have to have another method anyway, or get the MethodInfo for the property setter in order to construct the
delegate to invoke. In this case we actually know that `BeginInvoke` is required because we're running in the worker thread anyway, but I included the code for the sake of completeness.

- We don't call `EndInvoke` after the `BeginInvoke`. Unlike all other asynchronous methods (see the later section on the topic) you don't need to call `EndInvoke` unless you need the return value of the delegate's method. Of course, `BeginInvoke` is also different to all of the other asynchronous methods as it doesn't cause the delegate to be run on a thread pool thread - that would defeat the whole point in this case!

- State is used again to tell the UI thread how far we've counted so far. We use a `MethodInvoker` delegate to execute `UpdateCount`. We call this using `Invoke` to make sure that it executes on the UI thread. This time there's no attempt to detect whether or not an `Invoke` is required. I don't believe there's much harm in calling `Invoke` or `BeginInvoke` when it's not required - it'll just take a little longer than calling the method directly. (If you call `BeginInvoke` it will have a different effect than calling the method directly as it will occur later, rather than in the current execution flow, of course.) Again, we actually know that we need to call `Invoke` here anyway.

- A button is provided to let the user start the thread. It is disabled while the thread is running, and another `MethodInvoker` delegate is used to enable the button again afterwards.

- All state which is shared between threads (the current count and the target) is accessed in locks in the way described earlier. We spend as little time as possible in the lock, not updating the UI or anything else while holding the lock. This probably doesn't make too much difference here, but I believe it's worth getting into the habit of putting a lock around as little as possible - just as much as is needed. In particular, it would be disastrous to still have the lock in the worker thread when synchronously invoking `UpdateCount` - the UI thread would then try to acquire the lock as well, and you'd end up with deadlock.

- The worker thread is set to be a background thread (`IsBackground=true;`) so that when the UI thread exits, the whole application finishes. In other cases where you have a thread which should keep running even after the UI thread has quit, you need to be careful not to call `Invoke` or `BeginInvoke` when the UI thread is no longer running - you will either block permanently (waiting for the message to be taken off the queue, with nothing actually looking at messages) or receive an exception.

### The Thread Pool and Asynchronous Methods

The point of the thread pool is to avoid creating lots of threads for short tasks. Thread creation isn't particularly cheap, and if you start lots of threads, each doing only just enough work to warrant being run on a different thread in the first place, the cost of creation could significantly hamper performance. The thread pool solves that by having a "pool" of threads which have already been created and are just waiting for work items. When they've finished executing a work item, they then wait for the next one, etc. By default, the thread pool has 25 threads per processor. Note that the thread pool isn't just used for whatever asynchronous calls you make - the .NET framework libraries use it as well, and things can go badly wrong (usually resulting in a deadlock) if all the threads are used and some of them depend on other tasks which are scheduled. (For instance, if one thread is waiting for the results of another work item, but that work item is never run because there are no free threads.) This is a good reason to avoid using the thread pool for particularly long-running tasks. Personally I usually stick to creating a new thread for anything but pretty trivial tasks. If the thread's going to be running for more than a few seconds, the cost of creating the thread is going to be relatively insignificant.
Note that none of the samples below have any locking or volatility in to ensure that "fresh" values are seen. I haven't seen any code in any other samples, either. It's not a problem so long as there's a memory barrier in both the calling thread and the thread pool thread, but I haven't seen any guarantees of that. I would expect that there would be a memory barrier involved in each thread just to get the whole thing up and running in the first place, but as I say I haven't seen it guaranteed anywhere.

You can tell whether or not a thread is from the thread pool or not using Thread.IsThreadPoolThread. Thread pool threads are background threads - they don't prevent the runtime from exiting when all non-background threads have completed. There are various different ways of using the thread pool. Here's a brief description of each of them (except timers which have their own section, following this one):

**ThreadPool.QueueUserWorkItem()**

This is probably the simplest way of executing code in a thread pool thread. You simply provide a WaitCallback delegate (it doesn't return anything, and takes a single parameter of type object), and optionally the object to pass as the parameter to the callback when it is executed. If you don't specify a parameter, the callback will be passed null instead. Here's a sample program to demonstrate it:

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

public class Test
{
    static void Main()
    {        ThreadPool.QueueUserWorkItem(new WaitCallback(PrintOut), "Hello");

        // Give the callback time to execute - otherwise the app may terminate before it is called
        Thread.Sleep(1000);
    }

    static void PrintOut(object parameter)
    {
        Console.WriteLine(parameter);
    }
}
```

There is no built-in way of waiting for your callback to be executed, although you can of course signal the end of the callback using "normal" threading mechanisms (Monitor.Wait/Pulse etc).

**Calling BeginInvoke on a delegate**

It's relatively hard to find documentation on this topic, but when you create a delegate, the compiler generates three methods for you: Invoke, BeginInvoke and EndInvoke. Invoke is used to execute the delegate synchronously (i.e. a line of code such as myDelegate(); is actually compiled as myDelegate.Invoke());. The other two methods are for asynchronous execution, and must always be called as a pair - every BeginInvoke must be matched by a call to EndInvoke somewhere to guarantee that you don't leak resources. BeginInvoke takes the same parameters as the delegate itself does, plus another two parameters - an AsyncCallback which is called after the delegate has executed, and an object parameter which is made
available through the `AsyncState` property of the `IAsyncResult` parameter which is passed to the `AsyncCallback`. (This is typically used to pass the delegate which is being invoked, to make it easy to call `EndInvoke`.) The call to `EndInvoke` can be made to find the return value of the executed delegate. Don't worry if it sounds confusing - hopefully this example will make it somewhat simpler:

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

public class Test
{
    delegate int TestDelegate(string parameter);
    static void Main()
    {
        TestDelegate d = new TestDelegate(PrintOut);
        d.BeginInvoke("Hello", new AsyncCallback(Callback), d);
        // Give the callback time to execute - otherwise the app
        // may terminate before it is called
        Thread.Sleep(1000);
    }
    static int PrintOut (string parameter)
    {
        Console.WriteLine(parameter);
        return 5;
    }
    static void Callback (IAsyncResult ar)
    {
        TestDelegate d = (TestDelegate)ar.AsyncState;
        Console.WriteLine("Delegate returned {0}", d.EndInvoke(ar));
    }
}
```

The call to `BeginInvoke` returns an `IAsyncResult` which can be used to call `EndInvoke`, and you don't have to pass a callback delegate to be executed if you don't want to - just pass `null` as the last but one parameter to `BeginInvoke`. (You may still wish to pass in meaningful last parameter, as that will be available in the returned `IAsyncResult`'s `AsyncState` property, just as it would be in the callback case.)

The call to `EndInvoke` blocks until the delegate has finished executing - it's sort of like `Thread.Join`, but for a specific asynchronous delegate execution rather than a specific thread. Of course, when you call it from a callback delegate, it won't need to block as the callback will only execute after the delegate has finished anyway. Here's an example using `EndInvoke` from the original thread instead of using a callback:

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

public class Test
{
    delegate int TestDelegate(string parameter);
    static void Main()
    {
        TestDelegate d = new TestDelegate(PrintOut);
        IAsyncResult ar = d.BeginInvoke("Hello", null, null);
    }
    static int PrintOut (string parameter)
    {
        Console.WriteLine(parameter);
        return 5;
    }
    static void Callback (IAsyncResult ar)
    {
        TestDelegate d = (TestDelegate)ar.AsyncState;
        Console.WriteLine("Delegate returned {0}", d.EndInvoke(ar));
    }
}
```
Sometimes having to call `EndInvoke` is inconvenient - you often want "fire and forget" semantics where you don't care about the result or indeed when exactly the delegate has finished executing. Many articles suggest just calling `BeginInvoke` and not bothering with `EndInvoke`. This may work without leaking resources - but it's not guaranteed to, and even if it does now, it may not in the future. Here is a utility class (adapter from a mailing list post [http://discuss.develop.com/archives/wa.exe?A2=ind0302b&L=ADVANCED-DOTNET&D=0&P=2534] which allows you to call `FireAndForget` to execute a delegate asynchronously without worrying about `EndInvoke`:

```csharp
using System;
using System.Threading;
public class ThreadUtil
{
    /// <summary>
    /// Delegate to wrap another delegate and its arguments
    /// </summary>
    delegate void DelegateWrapper (Delegate d, object[] args);
    /// <summary>
    /// An instance of DelegateWrapper which calls InvokeWrappedDelegate,
    /// which in turn calls the DynamicInvoke method of the wrapped delegate.
    /// </summary>
    static DelegateWrapper wrapperInstance = new DelegateWrapper(InvokeWrappedDelegate);
    /// <summary>
    /// Callback used to call <code>EndInvoke</code> on the asynchronously invoked DelegateWrapper.
    /// </summary>
    static AsyncCallback callback = new AsyncCallback(EndWrapperInvoke);
    /// <summary>
    /// Executes the specified delegate with the specified arguments
    /// asynchronously on a thread pool thread.
    /// </summary>
    public static void FireAndForget (Delegate d, params object[] args)
    {
        // Invoke the wrapper asynchronously, which will then execute the wrapped delegate synchronously (in the thread pool thread)
        wrapperInstance.BeginInvoke(d, args, callback, null);
    }
}
```
When you provide `FireAndForget` with a delegate to execute, it actually invokes an internal delegate asynchronously, and that delegate executes the one you provided synchronously. This gives the effective result of the delegate you provided begin executed asynchronously, but allows the helper class to call `EndInvoke` on the delegate that it knows about - the `Delegate` class itself doesn't provide `BeginInvoke` or `EndInvoke` methods, otherwise this extra step would be unnecessary. Note the call to `ar.AsyncWaitHandle.Close()`. This prevents the `WaitHandle` leaking until garbage collection. The leak wouldn't cause any problems in most cases (unlike, for instance, file handles leaking), but in situations where `FireAndForget` would be called many, many times in quick succession, you could end up with a vast number of handles until the garbage collector started finalizing them. (This is also a bad thing in terms of performance - you shouldn't leave things to be finalised when it can be avoided.) The earlier sample code omitted this step for simplicity, but it's worth being aware of.

**BeginRead (etc)**

There are various methods in the standard library which come in `BeginXXX, EndXXX` pairs, such as `Stream.BeginRead` and `Stream.EndRead`. These almost all follow the same format, which is quite like calling `BeginInvoke` and `EndInvoke` on a delegate: you call `BeginXXX` with some "normal" parameters, an `AsyncCallback` parameter and a "state" parameter. The callback is executed asynchronously when the operation (such as reading from a stream) has completed. The callback can then use `EndXXX` to get the results of the operation. There has been some discussion on the newsgroups as to whether calls such as `Stream.BeginRead` use I/O completion ports, which are a very efficient way of performing I/O asynchronously without using one thread per operation. It seems likely that they do, so even after the first callback has started executing on a thread pool thread, if you need to do more of the same kind of operation (as you frequently will with something like a network stream, where you probably haven't read the whole thing in one go) it's a good idea to keep using asynchronous calls rather than the synchronous forms. Here's an example which downloads this page asynchronously:

```csharp
using System;
using System.IO;
using System.Net;
using System.Text;
using System.Threading;
```
```csharp
public class Test {
    static readonly object finishedLock = new object();

    static void Main()
    {
        WebRequest request = WebRequest.Create(PageUrl);
        RequestResponseState state = new RequestResponseState();
        state.request = request;

        // Lock the object we'll use for waiting now, to make
        // sure we don't (by some fluke) do everything in the other
        // threads // before getting to Monitor.Wait in this one. If we did,
        // the pulse
        // would effectively get lost!
        lock (finishedLock)
        {
            request.BeginGetResponse(new AsyncCallback(GetResponseCallback), state);
            Console.WriteLine("Waiting for response...");
            // Wait until everything's finished. Normally you'd
            // want to
            // carry on doing stuff here, of course.
            Monitor.Wait(finishedLock);
        }
    }

    static void GetResponseCallback (IAsyncResult ar)
    {
        RequestResponseState state = (RequestResponseState) ar.AsyncState;
        state.response = state.request.EndGetResponse(ar);
        state.stream = state.response.GetResponseStream();

        // Stash an Encoding for the text. I happen to know that
        // my web server returns text in ISO-8859-1 - which is
        // handy, as we don't need to worry about getting half
        // a character in one read and the other half in another.
        // (Use a Decoder if you want to cope with that.)
        state.encoding = Encoding.GetEncoding(28591);

        state.stream.BeginRead(state.buffer, 0, state.buffer.Length, new AsyncCallback(ReadCallback), state);
    }

    static void ReadCallback (IAsyncResult ar)
    {
        RequestResponseState state = (RequestResponseState) ar.AsyncState;

        int len = state.stream.EndRead(ar);

        // Have we finished now?
        if (len==0)
```

Note that there is no exception handling code. This is purely to keep the code simple here - it doesn't mean you don't need exception handling in your real code. Unfortunately it's harder to make sure that you don't fail to dispose of unmanaged resources in error situations when using asynchronous methods: the using statement is no use as it only works within one thread - if you did put using statements around the BeginRead (etc) method calls, you'd end up disposing of the stream before it had time to finish reading, which would be disastrous.

Of course, you could avoid asynchronous delegates entirely, not specifying any, and just call EndRead (etc) immediately after BeginRead in the main thread, just as with BeginInvoke and EndInvoke on delegates. Just like with delegates, you must call the matching EndXXX method to avoid potential resource leaks. It's less likely that you'll fail to do so in this case, of course, as the methods tend to return useful information. You should consider exceptions when devising your strategy for making these calls, however.
Timers

There are various different timers available in .NET, each of which basically calls a delegate after a certain amount of time has passed. This section is a very brief guide to the differences between them. See the resources section for more in depth links. All the timers implement IDisposable, so make sure you dispose of them when you're not using them any more.

System.Windows.Forms.Timer

This is (obviously) a forms-based timer. After creating it, you can set the interval which elapses between ticks (using the Interval property) and hook up a delegate to its Tick event. Calling Start and Stop methods (which effectively just change the value of the Enabled property) do the obvious things. While the timer is running, it generates ticks and fires the Tick event each time. Note, however, that because it runs entirely on the UI thread, if you have long-running UI operations, you may "miss" ticks if more than one would normally occur in the time taken by the long-running operation. Effectively, it keeps track of when its next tick is due, and when it next gets a chance to run, if the time is up, it fires the event. Personally, I find this the least useful of the timer classes.

System.Timers.Timer

This is a somewhat more powerful timer. Instead of a Tick event, it has the Elapsed event. As before, there are Start and Stop methods which are similar to changing the Enabled property. Changing the Interval or Enabled properties effectively reset the timer. (In other words, if the interval were set to two seconds, and every second you disabled and then re-enabled the timer, the event would never be fired.) The AutoReset property determines whether or not the timer should fire the event once and then stop, or keep going in a fire event / wait cycle.

By default, the event is fire on a thread pool thread. However, if you wish it to be fired on a particular thread, you can use the SynchronizingObject property which makes it invoke the event however the synchronizing object wishes it to. For instance, setting the synchronizing object to a UI control makes the event fire on that control's UI thread. Unlike the previous timer, the events are effectively queued - the timer doesn't wait for one event to have completed before starting to wait again and then firing off the next event.

System.Threading.Timer

This is the timer class I usually prefer, due to its simplicity. When constructing it, you need to pass in a TimerCallback delegate, a state object which is passed to the delegate when the timer fires, a "due" time and an interval. The timer will first fire after the due time has elapsed, and thereafter it will fire after each interval. Either value may be Timeout.Infinite - if the due time is infinite, the timer will never fire; if the interval is infinite, the timer will fire once (after the due time) and then it won't fire again. You can change the due time and the interval at any point using the Change method. (For instance, I sometimes find it useful to leave the interval as infinite, but every time the timer fires, call Change with a new due time.)

There's nothing fancy about this timer: the timer always fires on a thread pool thread. If you need to use it to update the UI, you need to use the techniques talked about in the Windows
Forms section. You don't start and stop it - if you don't want it to fire, just change the due time to be infinite.

Examples

Timers are simple enough that for the most part they don't really need examples, in my view. Just for kicks, here's a sample of System.Threading.Timer. If you really need other examples, please mail me (skeet@pobox.com) and let me know what exactly you're after. Note that the article in the resources section has some sample code too.

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

public class Test
{
    static void Main()
    {
        Console.WriteLine("Started at {0:HH:mm:ss.fff}", DateTime.Now);
        // Start in three seconds, then fire every one second
        using (Timer timer = new Timer(new TimerCallback(Tick), null, 3000, 1000))
        {
            // Wait for 10 seconds
            Thread.Sleep(10000);

            // Then go slow for another 10 seconds
            timer.Change(0, 2000);
            Thread.Sleep(10000);
        }

        // The timer will now have been disposed automatically due
to the using
        // statement, so there won't be any other threads running,
        and we'll quit.

        static void Tick(object state)
        {
            Console.WriteLine("Ticked at {0:HH:mm:ss.fff}" , DateTime.Now);
        }
    }
}
```

And here are the results of one run of that test:

```
Started at 15:32:07.473
Ticked at 15:32:10.520
Ticked at 15:32:11.520
Ticked at 15:32:12.520
Ticked at 15:32:13.520
Ticked at 15:32:14.520
Ticked at 15:32:15.520
Ticked at 15:32:16.520
Ticked at 15:32:17.520
Ticked at 15:32:17.536
Ticked at 15:32:19.552
Ticked at 15:32:21.552
Ticked at 15:32:23.552
Ticked at 15:32:25.552
Ticked at 15:32:27.552
```
Note the very small gap between the ticks at 15:32:17 - this was where Change was called with a due time of zero, which means "fire the delegate now".

**Shutting Down Worker Threads Gracefully**

This topic was mostly covered in the volatile section earlier, but as this is a particularly common scenario (with a typical question being "How can I shut down a worker thread?" I think it's worth presenting a working pattern. Here's a code skeleton which just needs the work for the worker thread to perform to be filled in (and any member it needs, of course):

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

/// <summary>
/// Skeleton for a worker thread. Another thread would typically set up an instance with some work to do, and invoke the Run method (eg with new Thread(new ThreadStart(job.Run)).Start())
/// </summary>
public class Worker
{
    /// <summary>
    /// Lock covering stopping and stopped
    /// </summary>
    readonly object stopLock = new object();
    /// <summary>
    /// Whether or not the worker thread has been asked to stop
    /// </summary>
    bool stopping = false;
    /// <summary>
    /// Whether or not the worker thread has stopped
    /// </summary>
    bool stopped = false;

    /// <summary>
    /// Returns whether the worker thread has been asked to stop. This continues to return true even after the thread has stopped.
    /// </summary>
    public bool Stopping
    {
        get
        {
            lock (stopLock)
            {
                return stopping;
            }
        }
    }

    /// <summary>
    /// Returns whether the worker thread has stopped.
    /// </summary>
    public bool Stopped
    {
        get
        {
            lock (stopLock)
            {
                return stopped;
            }
        }
    }

    /// <summary>
    /// Some work for the worker thread to do
    /// </summary>
    public void Run
    {
        // Work to be done here
    }
}
```
/// Tells the worker thread to stop, typically after completing
/// its
/// current work item. (The thread is *not* guaranteed to have
/// stopped
/// by the time this method returns.)
/// </summary>
public void Stop()
{
    lock (stopLock)
    {
        stopping = true;
    }
}

/// <summary>
/// Called by the worker thread to indicate when it has
/// stopped.
/// </summary>
void SetStopped()
{
    lock (stopLock)
    {
        stopped = true;
    }
}

/// <summary>
/// Main work loop of the class.
/// </summary>
public void Run()
{
    try
    {
        while (!Stopping)
        {
            // Insert work here. Make sure it doesn't tight
            // loop!
            // (If work is arriving periodically, use a queue
            // and Monitor.Wait, changing the Stop method to pulse the monitor as
            // well as setting
            // stopping.)

            // Note that you may also wish to break out
            *within* the loop
            // if work items can take a very long time but have
            points at which
            // it makes sense to check whether or not you've
            been asked to stop.
            // Do this with just:
            // if (Stopping)
            // {
            //     return;
            // }
            // The finally block will make sure that the
            stopped flag is set.
        }
    }
    finally
    {
        SetStopped();
    }
}
worker thread to stop, it pulses the monitor on the queue to make sure that the worker thread wakes up. (You can do this either by adding a "no-op" work item, or by modifying the the class implementing the queue to add a mechanism just tell worker threads to wake up and return a null work item.

In .NET v2, where properties can have different access for setters and getters, I'd recommend turning the `SetStopped` method into a setter for the `Stopped` property. I wouldn't recommend changing the `Stop` method into a setter, however. This is partly because it needs to be public, but should only go from `false` to `true`, and partly as a gut instinct in terms of the word stop being more forceful as a noun than just setting a property to true - it feels to me like properties shouldn't usually have as much of an effect on other threads as this one does.

The above code is fine for occasional use, but including it in several classes introduces a fair amount of redundancy. It's not very much work to abstract most of the above into a separate class and provide more functionality at the same time. The resulting class is a bit too long to include in an article, but can be found as part of my Miscellaneous Utilities [http://www.pobox.com/~skeet/csharp/miscutil] library.

**Choosing What To Lock On**

I said earlier that I'd explain why I created a new variable to lock on. Many books and articles recommend locking on `this` for instance methods and `typeof(MyTypeName)` (the Type object for whatever type you're writing code in) for static methods. I believe this is a bad idea, because it means you have less control over your locks. Other code may well end up locking on the same object as you do within your code, which makes it far harder to ensure that you only obtain locks in an appropriate order.

An alternative to locking on `this` is to lock on the reference of the object you're about to access - and indeed `ICollection` provides a `SyncRoot` property for exactly this purpose, providing a "shared" reference for different callers to lock on. This is a valid design in some situations, but should generally be avoided - if you keep your locks as private as possible, you should be able to write thread-safe objects when you wish to. The difficulty I believe `ICollection` is trying to solve is providing a single class which is fast in a single-threaded situation due to not locking internally but which allows easy thread-safe access when in a multi-threading situation. It also enables a common reference to be used for locking across several method calls, such as enumerating the whole collection. (Obviously `ICollection` itself doesn't provide any code for any of this as it's just an interface, but it encourages a consistent design for implementing classes to follow.)

In my experience, this level of complexity is unusual when developing applications - usually the ability to make each method thread-safe in itself is all that's required, and that can be achieved through entirely "private" locks, which no other object knows about. Even if you never expose your fields directly, just accessing them for callers, you don't usually know whether methods within those objects have decided to lock on `this`, so for absolute control you should create a new variable, and immediately instantiate a new object. This is the object you end up locking on. For "static locks" you should declare a private static variable (and immediately instantiate a new object) for the same reason - other classes can get a `Type` reference to your type too! While it is unlikely that they would lock on your type, it's not impossible - a private static variable keeps the lock invisible to other classes. In both cases, make the variable read-only to stop yourself from accidentally changing the value.
One other practice occasionally seen is that of locking on string literals, for example declaring

```
string someLock = "Lock guarding some data";
```

and then locking on `someLock`. This has the same problem as the other locking strategies, due to string interning - if two classes both use the same string literal, they'll end up using references to the same string, so they'll be sharing a lock without realising it. If you really want to make a lock with a description, you can always create your own `Lock` class. (This would also have the benefit of making it obvious to the rest of your code what the appropriate variable is used for - `Lock` is more descriptive than `object` in itself.)

In any one class you may have several locks, dealing with different bits of data which need to remain consistent. The more locks you have, the more finely grained your locking is - but the more complicated it gets making sure that you always take them out in a consistent order. (You also end up using more memory per instance, of course, but that's usually not an issue.) Of course, one of the benefits about having variables just for locking is that they can get their own XML documentation, making it easier to understand what pieces of state each lock is used to cover.

One final note on the issue: not only do many books and articles recommend locking on this:

```
the C# compiler does it for you automatically if you declare an event without specifying the
add/remove code. My recommendation is to explicitly write the add/remove code, following a
pattern something like this:
```

```csharp
/// <summary>
/// Delegate backing the SomeEvent event.
/// </summary>
SomeEventHandler someEvent;

/// <summary>
/// Lock for SomeEvent delegate access.
/// </summary>
readonly object someEventLock = new object();

/// <summary>
/// Description for the event
/// </summary>
public event SomeEventHandler SomeEvent
{
    add
    {
        lock (someEventLock)
        {
            someEvent += value;
        }
    }
    remove
    {
        lock (someEventLock)
        {
            someEvent -= value;
        }
    }
}

/// <summary>
/// Raises the SomeEvent event
/// </summary>
protected virtual OnSomeEvent(EventArgs e)
{
    SomeEventHandler handler;
    lock (someEventLock)
    {
        handler = someEvent;
    }
```

```csharp
```
if (handler != null)
{
    handler (this, e);
}

(This assumes that the SomeEventHandler delegate has been declared elsewhere.) Most of the time you can use a single lock for all your events, in my experience. Note the code for OnSomeEvent. It's important that you don't write it as:

```csharp
// Bad code! Do not use!
protected virtual OnSomeEvent (EventArgs e)
{
    lock (someEventLock)
    {
        if (someEvent != null)
        {
            someEvent (this, e);
        }
    }
}
```

or:

```csharp
// Bad code! Do not use!
protected virtual OnSomeEvent (EventArgs e)
{
    if (someEvent != null)
    {
        someEvent (this, e);
    }
}
```

The first ends up firing the event while still holding the lock, which is a bad idea - you don't know what code is going to be run at this stage, and it could well end up wanting to access the event from another thread, leading to deadlock.

The second doesn't lock at all, making it quite possible that the event delegate will change between the test for nullity and the invocation. That wouldn't be a problem most of the time, but if the delegate variable's value becomes null, then trying to invoke the delegate will lead to an exception being thrown. (If we didn't care whether the delegate variable's value was null or not, we wouldn't test it in the first place.)

**An Alternative Approach To Monitors**

Credit where credit is due - some of the ideas here are also embodied in Jeffrey Richter's Safe Thread Synchronization [http://msdn.microsoft.com/msdnmag/issues/03/01/NET/] article, and others are in Ian Griffiths' blog entry on locking with timeouts [http://www.interact-sw.co.uk/iangblog/2004/03/23/locking]. The C# team have also said that were they designing C# again now, they wouldn't have included the lock keyword, instead making sure that a mechanism for using the using statement for the same job would be available. (My main problem with the lock keyword is that it's the name I would almost always naturally use for the variable containing the monitor in a single-lock class.)
Both Java and .NET made the same mistake when it came to locking, I believe. At first it seems like having a monitor for every object is a good idea, but in practice if you care about monitor privacy, you need to have an extra field in classes which need to worry about thread safety anyway. This is basically the point Jeffrey Richter makes in the article referenced above, and the same thought had been going through my head before I saw his elegant explanation of it.

I then also heard about Ian Griffiths' quest for attempts to acquire monitors with a timeout, in a way which didn't make code really hard to follow. Using `Monitor.TryEnter` will work, but then you've got to put the `finally` block in yourself, and test the return value from the method. Ian reasoned that usually what you really want is just an exception if you can't acquire a lock within a reasonable time limit - it indicates that you've got deadlock, basically.

What intrigued me more than the idea was the implementation - specifically, using the `using` statement to neatly acquire the lock and release it at the end of a block. This page (and the referenced code) attempts to combine the two ideas, to give an alternative to normal locking which is clearer in intention than using a plain object, which still maintains the neatness of code, and which allows locking with timeouts.

Over time, I've expanded the initial concept to allow deadlock detection in terms of locks having an order imposed on them, and made a few other tweaks. The code is available as part of my Miscellaneous Utility library [http://www.pobox.com/~skeet/csharp/miscutil], and the usage page [http://www.pobox.com/~skeet/csharp/miscutil/usage/locking.html] has more details of exactly what's available. This page now just covers the basic principles.

Essentially, the three primary types involved are:

**SyncLock**
This is the main class callers will use. It basically encapsulates a monitor and a name for it. (If you can think of a nicer name than this which doesn't clash obviously with the framework, please mail me (skeet@pobox.com).)

**LockTimeoutException**
Obvious - thrown when an attempt to lock a monitor fails due to timeout. (I would have used just `TimeoutException`, but that seems to be specific to Windows Services, annoyingly enough - to my mind, it should have been in the `System` namespace.)

**LockToken**
This is a struct returned by `SyncLock` when the monitor has been successfully locked - it must be disposed in order to release the monitor. The reason for making this a struct rather than a class is that it then only takes up a bit of stack space for each lock operation, rather than needing to create a whole extra object each time, which would be nasty for performance.

The typical usage is very straightforward:

```csharp
// Create the lock - almost always as a field
// This constructor overload creates a named
// lock with a 20 second default timeout. Either the
// name or the default timeout can be omitted.
SyncLock syncLock = new SyncLock("Some name");

// Lock it using its default timeout using (syncLock.Lock())
{
   // Do stuff here
}
```
When to use Threads

I've written a lot now about how to use threads, but not when to use threads - when it's appropriate to and when it's better to keep everything in the same thread. This is partly because after a while it becomes fairly natural to work out what belongs where, and partly because it's quite tricky to actually describe.

There are a few times when there's absolutely no point in using multiple threads. For instance, if your application is bound by a single resource (e.g. the disk, or the CPU) and all the tasks you would use multiple threads for will all be trying to use that same resource, you'll just be adding contention. For instance, suppose you had an application which collected all the names of files on your disk. Splitting that job into multiple threads isn't likely to help - if anything, it'll make it worse, because it would be asking the file system for lots of different directories all at the same time, which could make the head seek all over the place instead of progressing steadily. Similarly, if you have workflow where each stage relies entirely on the results of the previous stage, you can't use threads effectively. For instance, if you have a program which loads an image, rotates it, then scales it, then turns it into black and white, then saves it again, each stage really needs the previous one to be finished before it can do anything.

Suppose, however, you wanted to read a bunch of files and then process their contents (e.g. calculating the cryptographic hash using an algorithm which takes a lot of processor power) then that might very well benefit from threading - either by having several threads doing both, or one thread dedicated to disk IO and another dedicated to hash calculation. The latter would probably be better, but would probably involve more work as the threads would need to be passing data to each other rather than just doing their own thing. Even though here there is still a dependency between the data being read from disk and the crypto processing, you don't have to read all the data from the disk before you can start processing it.

Similarly, in applications other than Windows Forms applications, it probably doesn't matter if you need to do something which will take a little while - a second or so, for instance. If you're writing a batch program, it doesn't matter in the slightest whether your code is always doing something different or whether some operations take longer than others. That's not to say that batch processing should always be single-threaded, but you don't need to worry about having a "main" thread which must be ready to react to events.

If you're writing an ASP.NET page, then unless you think the operation will take so long that you're willing to start another thread and send a page back to the user which tells them to wait and then refreshes itself (using an HTTP meta refresh tag, for instance) periodically to check whether the "real" page has finished, it's usually not worth changing to a different thread. It won't make the page come up any faster, as you'll just have to tell the "main" thread to wait for the other one to finish, and it'll be considerably more complicated to implement. Of course, the part about not being any faster isn't true if you genuinely can do two independent things at once - querying two different databases, for instance. In that case, threading can oc-
casionally be very useful even in ASP.NET scenarios, although you should consider using the thread pool for such tasks to avoid creating lots of threads which each only run for a short time. Don't forget that there may well be other requests which want to use the same resources, and you won't be doing less work in total by spreading it out over many threads. In short, while creating extra threads (or explicitly using the thread pool) is sometimes useful in ASP.NET, it's usually a last resort rather than a matter of course.

In Windows Forms applications, you really should put anything which takes any significant amount of time (even reading a short file) into a different thread, unless the code is really only for your own use and you don't mind an unresponsive UI. This is important, as while the UI thread is doing something else (reading a file, doing a heavy calculation, etc) it can't be reacting to events like the user trying to close it, or a previously hidden area now becoming exposed. The UI can easily become very unresponsive, which gives a horrible user experience. Here, threading isn't used to get the job done quickly - it's used to get the job done while keeping the user satisfied with responsiveness. You might be surprised just how quickly a user can notice an app becoming unresponsive. Even if the user can't actually do anything but close the application or move the window around while they wait, it gives a much more professional feel to a program if you don't end up with a big white box when you pass another window over it.

The choice between using a "new" thread and using one from the thread pool is a contentious one - I tend to like using new threads for most purposes, and others recommend almost always using thread pool threads. My concern about using the thread pool is that there are only a certain number of threads in it, and it's used in various ways by the framework itself - and it's not always obvious that it's doing so. If you accidentally end up with all the threads in the thread pool waiting for other work items which are scheduled to run in the thread pool, you'll have a very difficult deadlock to debug.

Using new threads, however, is relatively costly if they're only going to run for a short while - creating new threads isn't terribly cheap at the operating system level, whereas the thread pool will of course re-use threads to avoid repeating this cost. One happy medium is to use your own thread pool which is separate from the system one but which will still re-use threads. I have a fairly simple implementation in my Miscellaneous Utility Library [http://www.pobox.com/~skeet/csharp/miscutil] which you can tweak if it doesn't quite meet your requirements. It probably won't perform quite as well as the system thread pool which has been finely tuned - but you have a lot more control over what goes in it, how many threads it creates, etc.

Using multiple threads is almost always going to introduce complexity to your application, so the potential benefits should be carefully considered and weighed up before you start writing code. Work out the "boundaries" between threads in detail - what thread needs access to what data when, etc. This can make an enormous difference when it comes to the implementation. After making the decision to use threads and designing the threading scenarios carefully, keep taking care while writing the code - it's very easy to slip up, unfortunately, even with all the tools the .NET framework provides. The results of this work should be an elegant application which performs well and remains responsive whatever it's doing - something to feel justly proud of. Good luck!
Thread-safe Types and Methods

While you only occasionally need to make a decision about whether or not to use multiple threads, every time you design a type you need to consider what its thread safety model is going to be. This will very much depend on its intended usage. Some types naturally end up being used only within a single thread, others naturally end up being shared across threads, and some are used specifically for thread handling, and often have their own special characteristics. One of the most important things to do is to document the type's behaviour: which methods are thread-safe, which aren't, etc. Below are some common models. Note that when I talk about methods, I'm including properties as well.

Thread-safe static methods, unsafe instance methods

This is the most commonly found model in the framework. Basically, any static methods can be called by any thread at any time with no nasty side-effects. (There may be side-effects, but they shouldn't cause anything to get into an invalid state.) Instance methods, however, should only be called on a single thread at a time, and there should really be a memory barrier (in the calling code) before using an instance that has previous been used by another thread, and another memory barrier (again, in the calling code) after changing an instance that is thereafter going to be used by another thread. In practice, this is probably one of the areas most people are sloppiest in, just passing objects from one thread to another without making sure things are synchronized specifically - and in practice, it's usually going to be fine, because part of "handing over" an object usually involves a memory barrier in both threads anyway. None of that is the responsibility of the type in question with this model, however - instance methods should basically assuming they're running in a single thread, and not worry about locking in order to query/update values. Of course, if you know that one of the members of the type may be shared amongst threads, using that member may involve locking, depending on the thread-safety of the type of that member.

Type with thread affinity

If a type has a thread affinity, that usually means you can only use it (or most of it) from the thread it was created on. Controls are the most obvious examples of this - in the Windows Forms section I've already talked about the way that you're not meant to use any methods other than Invoke, BeginInvoke, EndInvoke, InvokeRequired and CreateGraphics on a control unless you're running in the thread responsible for it. Designing your own type like this should be a relatively rare occurrence, unless it's to do with the GUI, or uses something else with thread affinity.

Methods and properties using thread-local storage

Some methods and properties naturally use thread-local storage - that is to say, each thread has its own individual value for a variable. Thread.CurrentPrincipal is a good example of this - compare it with Thread.CurrentCulture, which is an instance property. Most methods and properties using thread-local storage are static, just like Thread.CurrentPrincipal - having one value per instance and per thread rarely makes sense, and indeed the simplest way of using thread-local storage is with ThreadStaticAttribute which can only be applied to static fields (which can then be returned by static properties, etc).
**Totally thread-safe types**

Sometimes, it's important for a type to be totally thread-safe, so that anyone can use any instance of it from any thread, with no internal inconsistencies or unwanted side-effects. This is particularly true of types which are typically accessed using the factory or singleton pattern. Encoding is a good example of this - it would be a pain if threads couldn't use Encoding.ASCII (or any other encoding) without carefully locking things.

Often, types like this are immutable - once they've been created, they don't change, so even without locking, it doesn't matter what you do with them. Assuming instances are acquired in a thread-safe way to start with (which usually involves a lock or a type initializer to ensure safety), immutable types are naturally thread-safe.

**Types with limited thread safety**

Some types can be accessed in some ways by multiple threads, but not in others. Many collections fall into this category. For instance, from the documentation for ArrayList:

> An ArrayList can support multiple readers concurrently, as long as the collection is not modified. To guarantee the thread safety of the ArrayList, all operations must be done through the wrapper returned by the Synchronized method.

So, you can populate an instance in one thread, and thereafter read the contents from multiple threads. The documentation doesn't specify anything about handing over an instance from one thread to another (such as whether the client code has to make sure there's a memory barrier after the last write and another before the first read in another thread), which is unfortunate, but probably means you can't rely on it. This kind of case is where clear documentation is absolutely crucial. If a type can support multiple threads doing one thing and another thread doing something else (multiple reader, single writer tending to be the most common form) then the documentation should state exactly what is entailed - is that level of safety available without any extra locking on the part of the client, what are the guarantees about how soon that any "new" data is seen by other threads, etc.

**Aborting and Interrupting Threads**

There are two methods in the Thread class which are often used for stopping threads - Abort and Interrupt. I don't recommend using either of these methods, but it's worth knowing about them - if only for why they should almost always be avoided.

**Aborting a thread**

Calling Thread.Abort aborts that thread as soon as possible. (Aborting a thread which is executing unmanaged code has no effect until the CLR gets control again.) A ThreadAbortException is thrown, which is a special exception which can be caught, but will automatically be rethrown at the end of the catch block. As it keeps being thrown, the exception will usually terminate the thread. Thread.ResetAbort can be called (if the caller has appropriate permissions) to stop the thread's abortion. Calling the method doesn't prevent stop the currently thrown exception, it just stops the exception from being rethrown at the end of a catch block. (Usually this distinction is irrelevant, as you'd almost always want to call ResetAbort from a catch block anyway).
Interrupting a thread

Calling `Thread.Interrupt` is similar, but somewhat less drastic. This causes a `ThreadInterruptedException` exception to be thrown the next time the thread enters the `WaitSleepJoin` state, or immediately if the thread is already in that state. There's nothing particularly special about the `ThreadInterruptedException` - it doesn't get rethrown like `ThreadAbortException` does. Note that threads can block without entering the `WaitSleepJoin` state, however. For example, reading from a blocking stream (a common situation where you'd like to interrupt a thread) doesn't make the thread enter the `WaitSleepJoin` state.

A nasty bug in the framework...

A post on a newsgroup drew my attention to a bug in version 1.0/1.1 of the framework. If a thread is aborted or interrupted while it is calling `Monitor.Wait`, and after the monitor has been pulsed but before the thread has been able to acquire it again, it returns from the call (with the appropriate exception) without reacquiring the monitor. This can lead to situations where the code makes it look like you'll certainly own the monitor, but you're executing it after an abort or interrupt and you no longer own it. Here's an example:

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

class Test
{
    static object someLock = new object();

    static void Main()
    {
        Console.WriteLine("Main thread starting");
        Thread secondThread = new Thread(new ThreadStart(ThreadJob));
        secondThread.Start();
        Console.WriteLine("Main thread sleeping");
        Thread.Sleep(500);
        lock (someLock)
        {
            Console.WriteLine("Main thread acquired lock - pulsing monitor");
            Monitor.Pulse(someLock);
            Console.WriteLine("Monitor pulsed; interrupting second thread");
            secondThread.Interrupt();
            Thread.Sleep(1000);
            Console.WriteLine("Main thread still owns lock...");
        }
    }

    static void ThreadJob()
    {
        Console.WriteLine("Second thread starting");
        lock (someLock)
        {
            Console.WriteLine("Second thread acquired lock - about to wait");
            try
            {
                Monitor.Wait(someLock);
            }
            catch (Exception e)
            {
                Console.WriteLine("Second thread caught an excep-
The results of the above are:

Main thread starting
Main thread sleeping
Second thread starting
Second thread acquired lock - about to wait
Main thread acquired lock - pulsing monitor
Monitor pulsed; interrupting second thread
Second thread caught an exception: Thread has been interrupted from a waiting state.
Main thread still owns lock...

Note the order of the last two lines - the line from the second thread has been written while the main thread owns the lock, despite being within the second thread's lock block.

In fact, the above code should throw a `SynchronizationLockException` when it implicitly calls `Monitor.Exit` at the end of the lock block. As it happens, `Monitor.Exit` doesn't throw the exception (despite the documentation's claims to the contrary).

It's hard to know exactly what should really happen here - if you've told a thread to be interrupted or aborted, you probably don't want it to hang around for a long time trying to reacquire a monitor (which is exactly what it does if the monitor hasn't been pulsed before the thread is interrupted). On the other hand, not being able to rely on a lock actually being owned within a lock block is nasty. I believe the behaviour has been changed for version 2 of the framework, but we'll have to wait to see exactly how it's changed.

**Why Thread.Abort/Interrupt should be avoided**

I don't use `Thread.Abort/Interrupt` routinely. I prefer a graceful shutdown which lets the thread do anything it wants to, and keeps things orderly. I dislike aborting or interrupting threads for the following reasons:

They aren't immediate

One of the reasons often given for not using the graceful shutdown pattern is that a thread could be waiting forever. Well, the same is true if you abort or interrupt it. If it's waiting for input from a stream of some description, you can abort or interrupt the thread and it will go right on waiting. If you only interrupt the thread, it could go right on processing other tasks, too - it won't actually be interrupted until it enters the `Wait-SleepJoin` state.

They can't be easily predicted

While they don't happen quite as quickly as you might sometimes want, aborts and interrupts do happen where you quite possibly don't want them to. If you don't know where a thread is going to be interrupted or aborted, it's hard to work out exactly how to get back to a consistent state. Although `finally` blocks will be executed, you don't want to have to put them all over the place just in case of an abort or interrupt. In almost all cases, the only time you don't mind a thread dying at any point in its operation is when the whole application is going down.
The bug described above

Getting your program into an inconsistent state is one problem - getting it into a state
which, on the face of it, shouldn't even be possible is even nastier.

Collected Hints and Tips

This is really just a collection of the various important parts of the rest of the article. An ex-
ecutive summary, if you like.

- Access all shared data from within locks, using the same lock for each access of the
  same collection of variables.
- Make sure you take out locks in a fixed order - if in one flow of execution you acquire
  lock A and then lock B (without first releasing A), there must be no flow of execution
  where you acquire lock B and then lock A, otherwise you risk deadlock.
- Lock on references which have no purpose other than locking. Unless they are to be
deliberately shared for the purposes of clients synchronizing themselves, keep these
locks private. They should also be read-only.
- Perform as small an amount of work as possible within a lock - in particular, don't use
  Control.Invoke within a lock that the UI thread will require.
- Where possible, call Monitor.Wait when have only acquired the lock you're waiting
  on. If you absolutely need to have another lock acquired at the same time, make sure
  that lock isn't required by the code which will pulse the monitor.
- Don't call event handlers from within a lock, but don't rely on the event delegate re-
  mainsing non-null after a single call: copy the delegate reference within the lock, and
  invoke it outside the lock.
- Never perform long-running operations on a UI thread.
- Never access a control's properties or methods (other than Invoke, BeginInvoke,
  EndInvoke, CreateGraphics or InvokeRequired) other than on its UI thread.
- Long-running operations should use newly created threads; short-running operations
can take advantage of the thread pool.
- Apart from Control.BeginInvoke, asynchronous BeginXXX method calls should al-
  ways make sure there is a matching EndXXX call.
- When passing an EventHandler delegate to Control.Invoke (or Con-
  trol.BeginInvoke), any parameters you specify are ignored - the sender is always
  set to the control it is executed for, and the EventArgs is always set to Even-
  tArgs.Empty.

Resources

Asynchronous Programming Design Pattern
MSDN pages with explanations for using and providing asynchronous methods.
Brad Abram's blog entry on volatility and memory barriers
[http://blogs.msdn.com/brada/archive/2004/05/12/130935.aspx]
Interesting as much for the discussion afterwards as the article itself, this discusses just
what is needed to make the double-checked lock algorithm thread-safe.
An email from Vance Morrison about the memory model
An excellent introduction to the .NET memory model, this email has one crucial cut-
and-paste problem in it (acknowledged later by Vance). The first ("broken") imple-
The implementation of the double-checked lock algorithm should not have a volatile variable. The volatility actually makes it thread-safe, against the intention of the author! Please bear that in mind while reading this, otherwise it makes a lot less sense.

"An Introduction to Programming with C# Threads" by Andrew Birrell

I haven't read all of this, and disagree with part of his "cheating" section (as he seems to assume that the .NET memory model is the same as the Java one, and that you therefore can't "fix" the double-check lock algorithm) but other than that, this looks very good.

Chris Brumme's blog entry on the .NET memory model
[http://blogs.gotdotnet.com/cbrumme/PermaLink.aspx/480d3a6d-1aa8-4694-96db-c69f01d7ff2b]

I disagree with some of the points Chris makes (as I believe it's far from impossible to write thread-safe code to the .NET memory model, so long as you don't try to do anything too clever). However, this is generally an excellent article describing some of the purpose and details of the .NET memory model.

Singleton implementation
[http://www.pobox.com/~skeet/csharp singleton.html]

My own page on writing a thread-safe singleton implementation. This tries to keep things somewhat simpler than some of the complicated and clever solutions given using volatile variables and explicit memory barriers.

Article by Justin Rogers about Control_Invoke
[http://weblogs.asp.net/justin_rogers/articles/126345.aspx]

Very detailed article about threading in Windows Forms.

Chris Brumme's blog entry on asynchronous operations
[http://weblogs.asp.net/cbrumme/archive/2003/05/06/51385.aspx]

Amongst other things, this article talks about the need to call EndXXX methods except in the case of Control_BeginInvoke.

Comparing the Timer Classes in the .NET Framework Class Library
[http://msdn.microsoft.com/msdnmag/issues/04/02/TimersinNET/default.aspx]

MSDN magazine article giving details of how to pick an appropriate timer, and how to use timers safely.

Advanced CLR mailing list post on threading principals

Details of a security oddity when it comes to invoking asynchronous operations.

Give Your .NET-based Application a Fast and Responsive UI with Multiple Threads
[http://msdn.microsoft.com/msdnmag/issues/03/02/Multithreading/]

Article on multi-threading which particularly concentrates on UI issues, but also contains "the basics".

Safe Thread Synchronization
[http://msdn.microsoft.com/msdnmag/issues/03/01/NET/]

Excellent article (as always) by Jeffrey Richter.

Timed Locks
[http://www.interact-sw.co.uk/iangblog/2004/03/23/locking]

The first of many blog entries by Ian Griffiths on his quest for locks with timeouts. (The other blog entries are linked from the first.)