Threading in C#

Joseph Albahari

Interested in a book on C# and .NET by the same author?
See *www.albahari.com/nutshell/*

Table of Contents

URI: http://www.albahari.com/threading/

Overview and Concepts

C# supports parallel execution of code through multithreading. A thread is an independent execution path, able to run simultaneously with other threads.

A C# program starts in a single thread created automatically by the CLR and operating system (the "main" thread), and is made multi-threaded by creating additional threads. Here's a simple example and its output:

All examples assume the following namespaces are imported, unless otherwise specified:

using System;
using System.Threading;

```csharp
class ThreadTest {
  static void Main() {
    Thread t = new Thread (WriteY);
    t.Start();                        // Run WriteY on new thread
    while (true) Console.Write ("x");  // Write 'x' forever
  }

  static void WriteY() {
    while (true) Console.Write ("y");  // Write 'y' forever
  }
}
```

```
xxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
...
```

The main thread creates a new thread **t** on which it runs a method that repeatedly prints the character **y**. Simultaneously, the main thread repeatedly prints the character **x**.

The CLR assigns each thread its own memory stack so that local variables are kept separate. In the next example, we define a method with a local variable, then call the method simultaneously on the main thread and a newly created thread:

```csharp
static void Main() {
  new Thread (Go).Start();      // Call Go() on a new thread
  Go();                         // Call Go() on the main thread
}

static void Go() {
  // Declare and use a local variable - 'cycles'
  for (int cycles = 0; cycles < 5; cycles++) Console.Write ('?');
}
```

```
??????????
```

A separate copy of the **cycles** variable is created on each thread's memory stack, and so the output is, predictably, ten question marks.

Threads share data if they have a common reference to the same object instance. Here's an example:

```
class ThreadTest {
 bool done;

 static void Main() {
   ThreadTest tt = new ThreadTest();   // Create a common instance
   new Thread (tt.Go).Start();
   tt.Go();
 }

 // Note that Go is now an instance method
 void Go() {
   if (!done) { done = true; Console.WriteLine ("Done"); }
 }
}
```

Because both threads call **Go()** on the same **ThreadTest** instance, they share the **done** field. This results in "Done" being printed once instead of twice:

```
Done
```

Static fields offer another way to share data between threads. Here's the same example with **done** as a static field:
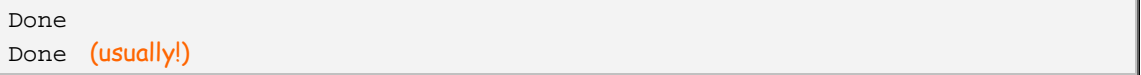
```
class ThreadTest {
 static bool done;     // Static fields are shared between all threads

 static void Main() {
   new Thread (Go).Start();
   Go();
 }

 static void Go() {
   if (!done) { done = true; Console.WriteLine ("Done"); }
 }
}
```

Both of these examples illustrate another key concept – that of *thread safety* (or, rather, lack of it!) The output is actually indeterminate: it's possible (although unlikely) that "Done" could be printed twice. If, however, we swap the order of statements in the **Go** method, then the odds of "Done" being printed twice go up dramatically:

```
static void Go() {
   if (!done) { Console.WriteLine ("Done"); done = true; }
}
```

```
Done
Done  (usually!)
```

The problem is that one thread can be evaluating the **if** statement right as the other thread is executing the **WriteLine** statement – before it's had a chance to set **done** to **true**.

The remedy is to obtain an exclusive lock while reading and writing to the common field. C# provides the lock statement for just this purpose:

4

```
class ThreadSafe {
  static bool done;
  static object locker = new object();

  static void Main() {
    new Thread (Go).Start();
    Go();
  }

  static void Go() {
    lock (locker) {
      if (!done) { Console.WriteLine ("Done"); done = true; }
    }
  }
}
```

When two threads simultaneously contend a lock (in this case, **locker**), one thread waits, or *blocks*, until the lock becomes available. In this case, it ensures only one thread can enter the critical section of code at a time, and "Done" will be printed just once. Code that's protected in such a manner – from indeterminacy in a multithreading context – is called thread-safe.

Temporarily pausing, or *blocking*, is an essential feature in coordinating, or *synchronizing* the activities of threads. Waiting for an exclusive lock is one reason for which a thread can block. Another is if a thread wants to pause, or Sleep for a period of time:

```
Thread.Sleep (TimeSpan.FromSeconds (30));     // Block for 30 seconds
```

A thread can also wait for another thread to end, by calling its Join method:

```
Thread t = new Thread (Go);      // Assume Go is some static method
t.Start();
t.Join();                        // Wait (block) until thread t ends
```

A thread, while blocked, doesn't consume CPU resources.

How Threading Works

Multithreading is managed internally by a *thread scheduler*, a function the CLR typically delegates to the operating system. A thread scheduler ensures all active threads are allocated appropriate execution time, and that threads that are waiting or blocked – for instance – on an exclusive lock, or on user input – do not consume CPU time.

On a single-core computer, a thread scheduler performs *time-slicing* – rapidly switching execution between each of the active threads. This results in "choppy" behavior, such as in the very first example, where each block of a repeating **X** or **Y** character corresponds to a time-slice allocated to the thread. Under Windows, a time-slice is typically in the tens-of-milliseconds region – chosen such as to be much larger than the CPU overhead in actually switching context between one thread and another (which is typically in the few-microseconds region).

On a multicore or multi-processor computer, multithreading is implemented with a mixture of time-slicing and genuine concurrency – where different threads run code simultaneously on different CPUs. It's almost certain there will still be some time-slicing, because of the operating system's need to service its own threads – as well as those of other applications.

A thread is said to be preempted when its execution is interrupted due to an external factor such as time-slicing. In most situations, a thread has no control over when and where it's preempted.

Threads vs. Processes

All threads within a single application are logically contained within a *process* – the operating system unit in which an application runs.

Threads have certain similarities to processes – for instance, processes are typically time-sliced with other processes running on the computer in much the same way as threads within a single C# application. The key difference is that processes are fully isolated from each other; threads share (heap) memory with other threads running in the same application. This is what makes threads useful: one thread can be fetching data in the background, while another thread is displaying the data as it arrives.

When to Use Threads

A common application for multithreading is performing time-consuming tasks in the background. The main thread keeps running, while the *worker thread* does its background job. With Windows Forms or WPF applications, if the main thread is tied up performing a lengthy operation, keyboard and mouse messages cannot be processed, and the application becomes unresponsive. For this reason, it's worth running time-consuming tasks on worker threads even if the main thread has the user stuck on a "Processing… please wait" modal dialog in cases where the program can't proceed until a particular task is complete. This ensures the application doesn't get tagged as "Not Responding" by the operating system, enticing the user to forcibly end the process in frustration! The modal dialog approach also allows for implementing a "Cancel" button, since the modal form will continue to receive events while the actual task is performed on the worker thread. The BackgroundWorker class assists in just this pattern of use.

In the case of non-UI applications, such as a Windows Service, multithreading makes particular sense when a task is potentially time-consuming because it's awaiting a response from another computer (such as an application server, database server, or client). Having a worker thread perform the task means the instigating thread is immediately free to do other things.

Another use for multithreading is in methods that perform intensive calculations. Such methods can execute faster on a multi-processor computer if the workload is divided amongst multiple threads. (One can test for the number of processors via the **Environment.ProcessorCount** property).

A C# application can become multi-threaded in two ways: either by explicitly creating and running additional threads, or using a feature of the .NET framework that implicitly creates threads – such as BackgroundWorker, thread pooling, a threading timer, a Remoting server, or a Web Services or ASP.NET application. In these latter cases, one has no choice but to embrace multithreading. A single-threaded ASP.NET web server would not be cool – even if such a thing were possible! Fortunately, with stateless application servers, multithreading is usually fairly simple; one's only concern perhaps being in providing appropriate locking mechanisms around data cached in static variables.

When Not to Use Threads

Multithreading also comes with disadvantages. The biggest is that it can lead to vastly more complex programs. Having multiple threads does not in itself create complexity; it's the *interaction between the threads* that creates complexity. This applies whether or not the interaction is intentional, and can result long development cycles, as well as an ongoing susceptibility to intermittent and non-reproducable bugs. For this reason, it pays to keep such interaction in a multi-threaded design simple – or not use multithreading at all – unless you have a peculiar penchant for re-writing and debugging!

Multithreading also comes with a resource and CPU cost in allocating and switching threads if used excessively. In particular, when heavy disk I/O is involved, it can be faster to have just one or two workers thread performing tasks in sequence, rather than having a multitude of threads each executing a task at the same time. Later we describe how to implement a Producer/Consumer queue, which provides just this functionality.

Creating and Starting Threads

Threads are created using the **Thread** class's constructor, passing in a **ThreadStart** delegate – indicating the method where execution should begin. Here's how the **ThreadStart** delegate is defined:

```
public delegate void ThreadStart();
```

Calling **Start** on the thread then sets it running. The thread continues until its method returns, at which point the thread ends. Here's an example, using the expanded C# syntax for creating a **TheadStart** delegate:

```
class ThreadTest {
  static void Main() {
    Thread t = new Thread (new ThreadStart (Go));
    t.Start();   // Run Go() on the new thread.
    Go();        // Simultaneously run Go() in the main thread.
  }

  static void Go() { Console.WriteLine ("hello!"); }
```

In this example, thread **t** executes **Go()** – at (much) the same time the main thread calls **Go()**. The result is two near-instant **hello**s:

```
hello!
hello!
```

A thread can be created more conveniently using C#'s shortcut syntax for instantiating delegates:

```
static void Main() {
  // No need to explicitly use ThreadStart:
  Thread t = new Thread (Go);
  t.Start();
  ...
}

static void Go() { ... }
```

In this case, a **ThreadStart** delegate is inferred automatically by the compiler. Another shortcut is to use an anonymous method to start the thread:

```
static void Main() {
  Thread t = new Thread (delegate() { Console.Write ("Hello!"); });
  t.Start();
}
```

A thread has an **IsAlive** property that returns true after its **Start()** method has been called, up until the thread ends. A thread, once ended, cannot be re-started.

Passing Data to ThreadStart

Let's say, in the example above, we wanted to better distinguish the output from each thread, perhaps by having one of the threads write in upper case. We could achieve this by passing a flag to the **Go** method: but then we couldn't use the **ThreadStart** delegate because it doesn't accept arguments. Fortunately, the .NET framework defines another version of the delegate called **ParameterizedThreadStart**, which accepts a single object argument as follows:

```
public delegate void ParameterizedThreadStart (object obj);
```

The previous example then looks like this:

```
class ThreadTest {
  static void Main() {
    Thread t = new Thread (Go);
    t.Start (true);              // == Go (true)
    Go (false);
  }

  static void Go (object upperCase) {
    bool upper = (bool) upperCase;
    Console.WriteLine (upper ? "HELLO!" : "hello!");
  }
```

```
hello!
HELLO!
```

In this example, the compiler automatically infers a **ParameterizedThreadStart** delegate because the Go method accepts a single object argument. We could just as well have written:

```
Thread t = new Thread (new ParameterizedThreadStart (Go));
t.Start (true);
```

A feature of using **ParameterizedThreadStart** is that we must cast the **object** argument to the desired type (in this case **bool**) before use. Also, there is only a single-argument version of this delegate.

An alternative is to use an anonymous method to call an ordinary method as follows:

```
static void Main() {
  Thread t = new Thread (delegate() { WriteText ("Hello"); });
  t.Start();
}

static void WriteText (string text) { Console.WriteLine (text); }
```

The advantage is that the target method (in this case **WriteText**) can accept any number of arguments, and no casting is required. However one must take into account the outer-variable semantics of anonymous methods, as is apparent in the following example:

```
static void Main() {
  string text = "Before";
  Thread t = new Thread (delegate() { WriteText (text); });
  text = "After";
  t.Start();
}

static void WriteText (string text) { Console.WriteLine (text); }
```

```
After
```

> Anonymous methods open the grotesque possibility of unintended interaction via outer variables if they are modified by either party subsequent to the thread starting. Intended interaction (usually via fields) is generally considered more than enough! Outer variables are best treated as ready-only once thread execution has begun – unless one's willing to implement appropriate locking semantics on both sides.

Another common system for passing data to a thread is by giving **Thread** an instance method rather than a static method. The instance object's properties can then tell the thread what to do, as in the following rewrite of the original example:

```
class ThreadTest {
  bool upper;

  static void Main() {
    ThreadTest instance1 = new ThreadTest();
    instance1.upper = true;
    Thread t = new Thread (instance1.Go);
    t.Start();
    ThreadTest instance2 = new ThreadTest();
    instance2.Go();          // Main thread – runs with upper=false
  }

  void Go() { Console.WriteLine (upper ? "HELLO!" : "hello!"); }
```

Naming Threads

A thread can be named via its **Name** property. This is of great benefit in debugging: as well as being able to **Console.WriteLine** a thread's name, Microsoft Visual Studio picks up a thread's name and displays it in the *Debug Location* toolbar. A thread's name can be set at any time – but only once – attempts to subsequently change it will throw an exception.

The application's main thread can also be assigned a name – in the following example the main thread is accessed via the **CurrentThread** static property:

```
class ThreadNaming {
  static void Main() {
    Thread.CurrentThread.Name = "main";
    Thread worker = new Thread (Go);
    worker.Name = "worker";
    worker.Start();
    Go();
  }

  static void Go() {
    Console.WriteLine ("Hello from " + Thread.CurrentThread.Name);
  }
}
```

```
Hello from main
Hello from worker
```

Foreground and Background Threads

By default, threads are foreground threads, meaning they keep the application alive for as long as any one of them is running. C# also supports background threads, which don't keep the application alive on their own – terminating immediately once all foreground threads have ended.

> Changing a thread from foreground to background doesn't change its priority or status within the CPU scheduler in any way.

A thread's **IsBackground** property controls its background status, as in the following example:

```
class PriorityTest {
  static void Main (string[] args) {
    Thread worker = new Thread (delegate() { Console.ReadLine(); });
    if (args.Length > 0) worker.IsBackground = true;
    worker.Start();
  }
}
```

If the program is called with no arguments, the worker thread runs in its default foreground mode, and will wait on the **ReadLine** statement, waiting for the user to hit **Enter**. Meanwhile, the main thread exits, but the application keeps running because a foreground thread is still alive.

If on the other hand an argument is passed to **Main()**, the worker is assigned background status, and the program exits almost immediately as the main thread ends – terminating the **ReadLine.**

When a background thread terminates in this manner, any **finally** blocks are circumvented. As circumventing **finally** code is generally undesirable, it's good practice to explicitly wait for any background worker threads to finish before exiting an application – perhaps with a timeout (this is achieved by calling Thread.Join). If for some reason a renegade worker thread never finishes, one can then attempt to abort it, and if that fails, abandon the thread, allowing it to die with the process (logging the conundrum at this stage would also make sense!)

Having worker threads as background threads can then beneficial, for the very reason that it's always possible to have the last say when it comes to ending the application. Consider the alternative – foreground thread that won't die – preventing the application from exiting. An abandoned foreground worker thread is particularly insidious with a Windows Forms application, because the application will appear to exit when the main thread ends (at least to the user) but its process will remain running. In the Windows Task Manager, it will have disappeared from the *Applications* tab, although its executable filename still be visible in the Processes tab. Unless the user explicitly locates and ends the task, it will continue to consume resources and perhaps prevent a new instance of the application from starting or functioning properly.

> A common cause for an application failing to exit properly is the presence of "forgotten" foregrounds threads.

Thread Priority

A thread's **Priority** property determines how much execution time it gets relative to other active threads in the same process, on the following scale:

```
public enum ThreadPriority
   { Lowest, BelowNormal, Normal, AboveNormal, Highest }
```

This becomes relevant only when multiple threads are simultaneously active.

Setting a thread's priority to high doesn't mean it can perform real-time work, because it's still limited by the application's process priority. To perform real-time work, the **Process** class in **System.Diagnostics** must also be used to elevate the process priority as follows (I didn't tell you how to do this):

```
Process.GetCurrentProcess().PriorityClass =
   ProcessPriorityClass.High;
```

ProcessPriorityClass.High is actually one notch short of the highest process priority: **Realtime**. Setting one's process priority to **Realtime** instructs the operating system that you never want your process to be preempted. If your program enters an accidental infinite loop you can expect even the operating system to be locked out. Nothing short of the power button will rescue you! For this reason, **High** is generally considered the highest usable process priority.

If the real-time application has a user interface, it can be undesirable to elevate the process priority because screen updates will be given excessive CPU time – slowing the entire computer, particularly if the UI is complex. (Although at the time of writing, the Internet telephony program Skype gets away with doing just this, perhaps because its UI is fairly simple). Lowering the main thread's priority – in conjunction with raising the process's priority – ensures the real-time thread doesn't get preempted by screen redraws, but doesn't prevent the computer from slowing, because the operating system will still allocate excessive CPU to the process as a whole. The ideal solution is to have the real-time work and user interface in separate processes (with different priorities), communicating via Remoting or shared memory. Shared memory requires P/Invoking the Win32 API (web-search *CreateFileMapping* and *MapViewOfFile*).

Exception Handling

Any **try/catch/finally** blocks in scope when a thread is created are of no relevance once the thread starts executing. Consider the following program:

```
public static void Main() {
  try {
    new Thread (Go).Start();
  }
  catch (Exception ex) {
    // We'll never get here!
    Console.WriteLine ("Exception!");
  }

  static void Go() { throw null; }
}
```

The **try/catch** statement in this example is effectively useless, and the newly created thread will be encumbered with an unhandled **NullReferenceException**. This behavior makes sense when you consider a thread has an independent execution path. The remedy is for thread entry methods to have their own exception handlers:

```
public static void Main() {
  new Thread (Go).Start();
}

static void Go() {
  try {
    ...
    throw null;        // this exception will get caught below
    ...
  }
  catch (Exception ex) {
    Typically log the exception, and/or signal another thread
    that we've come unstuck
    ...
  }
```

From .NET 2.0 onwards, an unhandled exception on any thread shuts down the whole application, meaning ignoring the exception is generally not an option. Hence a **try/catch** block is required in every thread entry method – at least in production applications – in order to avoid unwanted application shutdown in case of an unhandled exception. This can be somewhat cumbersome – particularly for Windows Forms programmers, who commonly use the "global" exception handler, as follows:

```
static class Program {
  static void Main() {
    Application.ThreadException += HandleError;
    Application.Run (new MainForm());
  }

  static void HandleError (object sender,
                           ThreadExceptionEventArgs e) {
    Log exception, then either exit the app or continue...
  }
}
```

The **Application.ThreadException** event fires when an exception is thrown from code that was ultimately called as a result of a Windows message (for example, a keyboard, mouse or "paint" message) – in short, nearly all code in a typical Windows Forms application. While this works perfectly, it lulls one into a false sense of security – that all exceptions will be caught by the central exception handler. Exceptions thrown on worker threads are a good example of exceptions not caught by **Application.ThreadException** (the code inside the **Main** method is another – including the main form's constructor, which executes before the Windows message loop begins).

The .NET framework provides a lower-level event for global exception handling: **AppDomain.UnhandledException**. This event fires when there's an unhandled exception in any thread, and in any type of application (with or without a user interface). However, while it offers a good last-resort mechanism for logging untrapped exceptions, it provides no means of preventing the application from shutting down – and no means to suppress the .NET unhandled exception dialog.

In production applications, explicit exception handling is required on all thread entry methods. One can cut the work by using a wrapper or helper class to perform the job, such as **BackgroundWorker** (discussed in Part 3).

PART 2
BASIC SYNCHRONIZATION

Synchronization Essentials

The following summarize the .NET tools for coordinating or synchronizing the actions of threads:

Simple Blocking Methods

Construct	Purpose
Sleep	Blocks for a given time period.
Join	Waits for another thread to finish.

Locking Constructs

Construct	Purpose	Cross-Process?	Speed
lock	Ensures just one thread can access a resource, or section of code.	no	fast
Mutex	Ensures just one thread can access a resource, or section of code. Can be used to prevent multiple instances of an application from starting.	yes	moderate
Semaphore	Ensures not more than a specified number of threads can access a resource, or section of code.	yes	moderate

Signaling Constructs

Construct	Purpose	Cross-Process?	Speed
EventWaitHandle	Allows a thread to wait until it receives a signal from another thread.	yes	moderate
Wait and Pulse*	Allows a thread to wait until a custom blocking condition is met.	no	moderate

Non-Blocking Synchronization Constructs*

Construct	Purpose	Cross-Process?	Speed
Interlocked*	To perform simple non-blocking atomic operations.	yes (assuming shared memory)	very fast
volatile*	To allow safe non-blocking access to individual fields outside of a lock.		very fast

*Covered in Part 4

Blocking

When a thread waits or pauses as a result of using the constructs listed in the tables above, it's said to be *blocked*. Once blocked, a thread immediately relinquishes its allocation of CPU time, adds **WaitSleepJoin** to its ThreadState property, and doesn't get re-scheduled until unblocked. Unblocking happens in one of four ways (the computer's power button doesn't count!):

- by the blocking condition being satisfied

- by the operation timing out (if a timeout is specified)

- by being interrupted via Thread.Interrupt

- by being aborted via Thread.Abort

A thread is not deemed blocked if its execution is paused via the (deprecated) Suspend method.

Sleeping and Spinning

Calling **Thread.Sleep** blocks the current thread for the given time period (or until interrupted):

```
static void Main() {
  Thread.Sleep (0);                        // relinquish CPU time-slice
  Thread.Sleep (1000);                     // sleep for 1000 ms
  Thread.Sleep (TimeSpan.FromHours (1));   // sleep for 1 hour
  Thread.Sleep (Timeout.Infinite);         // sleep until interrupted
}
```

More precisely, **Thread.Sleep** relinquishes the CPU, requesting that the thread is not re-scheduled until the given time period has elapsed. Thread.Sleep(0) relinquishes the CPU just long enough to allow any other active threads present in a time-slicing queue (should there be one) to be executed.

> **Thread.Sleep** is unique amongst the blocking methods in that suspends Windows message pumping within a Windows Forms application, or COM environment on a thread for which the single-threaded apartment model is used. This is of little consequence with Windows Forms applications, in that any lengthy blocking operation on the main UI thread will make the application unresponsive – and is hence generally avoided – regardless of the whether or not message pumping is "technically" suspended. The situation is more complex in a legacy COM hosting environment, where it can sometimes be desirable to sleep while keeping message pumping alive. Microsoft's Chris Brumme discusses this at length in his web log (search: **'COM "Chris Brumme"'**).

The Thread class also provides a **SpinWait** method, which doesn't relinquish any CPU time, instead looping the CPU – keeping it "uselessly busy" for the given number of iterations. 50 iterations might equate to a pause of around a microsecond, although this depends on CPU speed and load. Technically, **SpinWait** is not a blocking method: a spin-waiting thread does not have a **ThreadState** of **WaitSleepJoin** and can't be prematurely **Interrupt**ed by another thread. **SpinWait** is rarely used – its primary purpose being to wait on a resource that's expected to be ready very soon (inside maybe a microsecond) without calling **Sleep** and wasting CPU time by forcing a thread change. However this technique is advantageous only on multi-processor computers: on single-processor computers, there's no opportunity for a resource's status to change until the spinning thread ends its time-slice – which defeats the purpose of spinning to begin with. And calling **SpinWait** often or for long periods of time itself is wasteful on CPU time.

Blocking vs. Spinning

A thread can wait for a certain condition by explicitly spinning using a polling loop, for example:

```
while (!proceed);
```

or:

```
while (DateTime.Now < nextStartTime);
```

This is very wasteful on CPU time: as far as the CLR and operating system is concerned, the thread is performing an important calculation, and so gets allocated resources accordingly! A thread looping in this state is not counted as blocked, unlike a thread waiting on an EventWaitHandle (the construct usually employed for such signaling tasks).

A variation that's sometimes used is a hybrid between blocking and spinning:

```
while (!proceed) Thread.Sleep (x);    // "Spin-Sleeping!"
```

The larger **x**, the more CPU-efficient this is; the trade-off being in increased latency. Anything above 20ms incurs a negligible overhead – unless the condition in the while-loop is particularly complex.

Except for the slight latency, this combination of spinning and sleeping can work quite well (subject to concurrency issues on the **proceed** flag, discussed in Part 4). Perhaps its biggest use is when a programmer has given up on getting a more complex signaling construct to work!

Joining a Thread

You can block until another thread ends by calling **Join**:

```
class JoinDemo {
  static void Main() {
    Thread t = new Thread (delegate() { Console.ReadLine(); });
    t.Start();
    t.Join();    // Wait until thread t finishes
    Console.WriteLine ("Thread t's ReadLine complete!");
  }
}
```

The **Join** method also accepts a timeout argument – in milliseconds, or as a **TimeSpan**, returning false if the **Join** timed out rather than found the end of the thread. **Join** with a timeout functions rather like **Sleep** – in fact the following two lines of code are almost identical:

```
Thread.Sleep (1000);
Thread.CurrentThread.Join (1000);
```

(Their difference is apparent only in single-threaded apartment applications with COM interoperability, and stems from the subtleties in Windows message pumping semantics described previously: Join keeps message pumping alive while blocked; Sleep suspends message pumping).

Locking and Thread Safety

Locking enforces exclusive access, and is used to ensure only one thread can enter particular sections of code at a time. For example, consider following class:

```
class ThreadUnsafe {
  static int val1, val2;

  static void Go() {
    if (val2 != 0) Console.WriteLine (val1 / val2);
    val2 = 0;
  }
}
```

This is not thread-safe: if **Go** was called by two threads simultaneously it would be possible to get a division by zero error – because **val2** could be set to zero in one thread right as the other thread was in between executing the **if** statement and **Console.WriteLine**.

Here's how **lock** can fix the problem:

```
class ThreadSafe {
  static object locker = new object();
  static int val1, val2;

  static void Go() {
    lock (locker) {
      if (val2 != 0) Console.WriteLine (val1 / val2);
      val2 = 0;
    }
  }
}
```

Only one thread can lock the synchronizing object (in this case **locker**) at a time, and any contending threads are blocked until the lock is released. If more than one thread contends the lock, they are queued – on a "ready queue" and granted the lock on a first-come, first-served basis as it becomes available. Exclusive locks are sometimes said to enforce serialized access to whatever's protected by the lock, because one thread's access cannot overlap with that of another. In this case, we're protecting the logic inside the **Go** method, as well as the fields **val1** and **val2**.

A thread blocked while awaiting a contended lock has a ThreadState of **WaitSleepJoin**. Later we discuss how a thread blocked in this state can be forcibly released via another thread calling its Interrupt or Abort method. This is a fairly heavy-duty technique that might typically be used in ending a worker thread.

C#'s **lock** statement is in fact a syntactic shortcut for a call to the methods **Monitor.Enter** and **Monitor.Exit**, within a try-finally block. Here's what's actually happening within the **Go** method of the previous example:

```
Monitor.Enter (locker);
try {
  if (val2 != 0) Console.WriteLine (val1 / val2);
  val2 = 0;
}
finally { Monitor.Exit (locker); }
```

Calling **Monitor.Exit** without first calling **Monitor.Enter** on the same object throws an exception.

Monitor also provides a **TryEnter** method allows a timeout to be specified – either in milliseconds or as a **TimeSpan**. The method then returns true – if a lock was obtained – or false – if no lock was obtained because the method timed out. **TryEnter** can also be called with no argument, which "tests" the lock, timing out immediately if the lock can't be obtained right away.

Choosing the Synchronization Object

Any object visible to each of the partaking threads can be used as a synchronizing object, subject to one hard rule: it must be a reference type. It's also highly recommended that the synchronizing object be privately scoped to the class (i.e. a private instance field) to prevent an unintentional interaction from external code locking the same object. Subject to these rules, the synchronizing object can double as the object it's protecting, such as with the **list** field below:

```csharp
class ThreadSafe {
  List <string> list = new List <string>();

  void Test() {
    lock (list) {
      list.Add ("Item 1");
      ...
```

A dedicated field is commonly used (such as **locker**, in the example prior), because it allows precise control over the scope and granularity of the lock. Using the object or type itself as a synchronization object, i.e.:

```csharp
lock (this) { ... }
```

or:

```csharp
lock (typeof (Widget)) { ... }    // For protecting access to statics
```

is discouraged because it potentially offers public scope to the synchronization object.

> Locking doesn't restrict access to the synchronizing object itself in any way. In other words, **x.ToString()** will not block because another thread has called **lock(x)** – both threads must call **lock(x)** in order for blocking to occur.

Nested Locking

A thread can repeatedly lock the same object, either via multiple calls to **Monitor.Enter**, or via nested **lock** statements. The object is then unlocked when a corresponding number of **Monitor.Exit** statements have executed, or the outermost **lock** statement has exited. This allows for the most natural semantics when one method calls another as follows:

```csharp
static object x = new object();

static void Main() {
  lock (x) {
    Console.WriteLine ("I have the lock");
    Nest();
    Console.WriteLine ("I still have the lock");
  }
  Here the lock is released.
}

static void Nest() {
  lock (x) {
    ...
  }
  Released the lock? Not quite!
}
```

A thread can block only on the first, or outermost lock.

When to Lock

As a basic rule, any field accessible to multiple threads should be read and written within a lock. Even in the simplest case – an assignment operation on a single field – one must consider synchronization. In the following class, neither the **Increment** nor the **Assign** method is thread-safe:

```
class ThreadUnsafe {
  static int x;
  static void Increment() { x++; }
  static void Assign()    { x = 123; }
}
```

Here are thread-safe versions of **Increment** and **Assign**:

```
class ThreadUnsafe {
  static object locker = new object();
  static int x;

  static void Increment() { lock (locker) x++; }
  static void Assign()    { lock (locker) x = 123; }
}
```

As an alternative to locking, one can use a non-blocking synchronization construct in these simple situations. This is discussed in Part 4 (along with the reasons that such statements require synchronization).

Locking and Atomicity

If a group of variables are always read and written within the same lock, then one can say the variables are read and written *atomically*. Let's suppose fields **x** and **y** are only ever read or assigned within a **lock** on object **locker**:

```
lock (locker) { if (x != 0) y /= x; }
```

One can say **x** and **y** are accessed atomically, because the code block cannot be *divided* or *preempted* by the actions of another thread in such a way that will change **x** or **y** and *invalidate its outcome*. You'll never get a division-by-zero error, providing **x** and **y** are always accessed within this same exclusive lock.

Performance Considerations

Locking itself is very fast: a lock is typically obtained in tens of nanoseconds assuming no blocking. If blocking occurs, the consequential task-switching moves the overhead closer to the microseconds-region, although it may be milliseconds before the thread's actually rescheduled. This, in turn, is dwarfed by the hours of overhead – or overtime – that can result from not locking when you should have!

Locking can have adverse effects if improperly used – impoverished concurrency, deadlocks and lock races. Impoverished concurrency occurs when too much code is placed in a lock statement, causing other threads to block unnecessarily. A deadlock is when two threads each wait for a lock held by the other, and so neither can proceed. A lock race happens when it's possible for either of two threads to obtain a lock first, the program breaking if the "wrong" thread wins.

Deadlocks are most commonly a syndrome of too many synchronizing objects. A good rule is to start on the side of having fewer objects on which to lock, increasing the locking granularity when a plausible scenario involving excessive blocking arises.

Thread Safety

Thread-safe code is code which has no indeterminacy in the face of any multithreading scenario. Thread-safety is achieved primarily with locking, and by reducing the possibilities for interaction between threads.

- A method which is thread-safe in any scenario is called reentrant. General-purpose types are rarely thread-safe in their entirety, for the following reasons:the development burden in full thread-safety can be significant, particularly if a type has many fields (each field is a potential for interaction in an arbitrarily multi-threaded context)

- thread-safety can entail a performance cost (payable, in part, whether or not the type is actually used by multiple threads)

- a thread-safe type does not necessarily make the program using it thread-safe – and sometimes the work involved in the latter can make the former redundant.

Thread-safety is hence usually implemented just where it needs to be, in order to handle a specific multithreading scenario.

There are, however, a few ways to "cheat" and have large and complex classes run safely in a multi-threaded environment. One is to sacrifice granularity by wrapping large sections of code – even access to an entire object – around an exclusive lock – enforcing serialized access at a high level. This tactic is also crucial in allowing a thread-unsafe object to be used within thread-safe code – and is valid providing the same exclusive lock is used to protect access to all properties, methods and fields on the thread-unsafe object.

Primitive types aside, very few .NET framework types when instantiated are thread-safe for anything more than concurrent read-only access. The onus is on the developer to superimpose thread-safety – typically using exclusive locks.

Another way to cheat is to minimize thread interaction by minimizing shared data. This is an excellent approach and is used implicitly in "stateless" middle-tier application and web page servers. Since multiple client requests can arrive simultaneously, each request comes in on its own thread (by virtue of the ASP.NET, Web Services or Remoting architectures), and this means the methods they call must be thread-safe. A stateless design (popular for reasons of scalability) intrinsically limits the possibility of interaction, since classes are unable to persist data between each request. Thread interaction is then limited just to static fields one may choose to create – perhaps for the purposes of
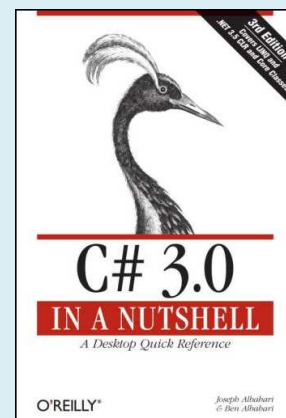
caching commonly used data in memory – and in providing infrastructure services such as authentication and auditing.

Thread-Safety and .NET Framework Types

Locking can be used to convert thread-unsafe code into thread-safe code. A good example is with the .NET framework – nearly all of its non-primitive types are not thread safe when instantiated, and yet they can be used in multi-threaded code if all access to any given object is protected via a lock. Here's an example, where two threads simultaneously add items to the same **List** collection, then enumerate the list:

```
class ThreadSafe {
  static List <string> list = new List <string>();

  static void Main() {
    new Thread (AddItems).Start();
    new Thread (AddItems).Start();
  }

  static void AddItems() {
    for (int i = 0; i < 100; i++)
      lock (list)
        list.Add ("Item " + list.Count);

    string[] items;
    lock (list) items = list.ToArray();
    foreach (string s in items) Console.WriteLine (s);
  }
}
```

In this case, we're locking on the **list** object itself, which is fine in this simple scenario. If we had two interrelated lists, however, we would need to lock upon a common object – perhaps a separate field, if neither list presented itself as the obvious candidate.

Enumerating .NET collections is also thread-unsafe in the sense that an exception is thrown if another thread alters the list during enumeration. Rather than locking for the duration of enumeration, in this example, we first copy the items to an array. This avoids holding the lock excessively if what we're doing during enumeration is potentially time-consuming.

Here's an interesting supposition: imagine if the **List** class was, indeed, thread-safe. What would it solve? Potentially, very little! To illustrate, let's say we wanted to add an item to our hypothetical thread-safe list, as follows:

```
if (!myList.Contains (newItem)) myList.Add (newItem);
```

Whether or not the list was thread-safe, this statement is certainly not! The whole **if** statement would have to be wrapped in a lock – to prevent preemption in between testing for containership and adding the new item. This same lock would then need to be used everywhere we modified that list. For instance, the following statement would also need to be wrapped – in the identical lock:

```
myList.Clear();
```

to ensure it did not preempt the former statement. In other words, we would have to lock almost exactly as with our thread-unsafe collection classes. Built-in thread safety, then, can actually be a waste of time!

One could argue this point when writing custom components – why build in thread-safety when it can easily end up being redundant?

There is a counter-argument: wrapping an object around a custom lock works only if all concurrent threads are aware of, and use, the lock – which may not be the case if the object is widely scoped. The worst scenario crops up with static members in a public type. For instance, imagine the static property on the **DateTime** struct, **DateTime.Now**, was not thread-safe, and that two concurrent calls could result in garbled output or an exception. The only way to remedy this with external locking might be to lock the type itself – **lock(typeof(DateTime))** – around calls to **DateTime.Now** – which would work only if all programmers agreed to do this. And this is unlikely, given that locking a type is considered by many, a Bad Thing!

For this reason, static members on the **DateTime** struct are guaranteed to be thread-safe. This is a common pattern throughout the .NET framework – static members are thread-safe, while instance members are not. Following this pattern also makes sense when writing custom types, so as not to create impossible thread-safety conundrums!

> When writing components for public consumption, a good policy is to program at least such as not to preclude thread-safety. This means being particularly careful with static members – whether used internally or exposed publicly.

Interrupt and Abort

A blocked thread can be released prematurely in one of two ways:

- via Thread.Interrupt

- via Thread.Abort

This must happen via the activities of another thread; the waiting thread is powerless to do anything in its blocked state.

Interrupt

Calling **Interrupt** on a blocked thread forcibly releases it, throwing a **ThreadInterruptedException**, as follows:

```
class Program {
  static void Main() {
    Thread t = new Thread (delegate() {
      try {
        Thread.Sleep (Timeout.Infinite);
      }
      catch (ThreadInterruptedException) {
        Console.Write ("Forcibly ");
      }
      Console.WriteLine ("Woken!");
    });

    t.Start();
    t.Interrupt();
  }
}
```

```
Forcibly Woken!
```

Interrupting a thread only releases it from its current (or next) wait: it does not cause the thread to end (unless, of course, the **ThreadInterruptedException** is unhandled!)

If **Interrupt** is called on a thread that's not blocked, the thread continues executing until it next blocks, at which point a **ThreadInterruptedException** is thrown. This avoids the need for the following test:

```
if ((worker.ThreadState & ThreadState.WaitSleepJoin) > 0)
  worker.Interrupt();
```

which is not thread-safe because of the possibility of being preempted in between the **if** statement and **worker.Interrupt**.

Interrupting a thread arbitrarily is dangerous, however, because any framework or third-party methods in the calling stack could unexpectedly receive the interrupt rather than your intended code. All it would take is for the thread to block briefly on a simple lock or synchronization resource, and any pending interruption would kick in. If the method wasn't designed to be interrupted (with appropriate cleanup code in finally blocks) objects could be left in an unusable state, or resources incompletely released.

Interrupting a thread is safe when you know exactly where the thread is. Later we cover signaling constructs, which provide just such a means.

Abort

A blocked thread can also be forcibly released via its **Abort** method. This has an effect similar to calling **Interrupt**, except that a **ThreadAbortException** is thrown instead of a **ThreadInterruptedException**. Furthermore, the exception will be re-thrown at the end of the catch block (in an attempt to terminate the thread for good) unless **Thread.ResetAbort** is called within the catch block. In the interim, the thread has a **ThreadState** of **AbortRequested**.

The big difference, though, between **Interrupt** and **Abort**, is what happens when it's called on a thread that is not blocked. While **Interrupt** waits until the thread next blocks before doing anything, **Abort** throws an exception on the thread right where it's executing – maybe not even in your code. Aborting a non-blocked thread can have significant consequences, the details of which are explored in the later section "Aborting Threads".

Thread State
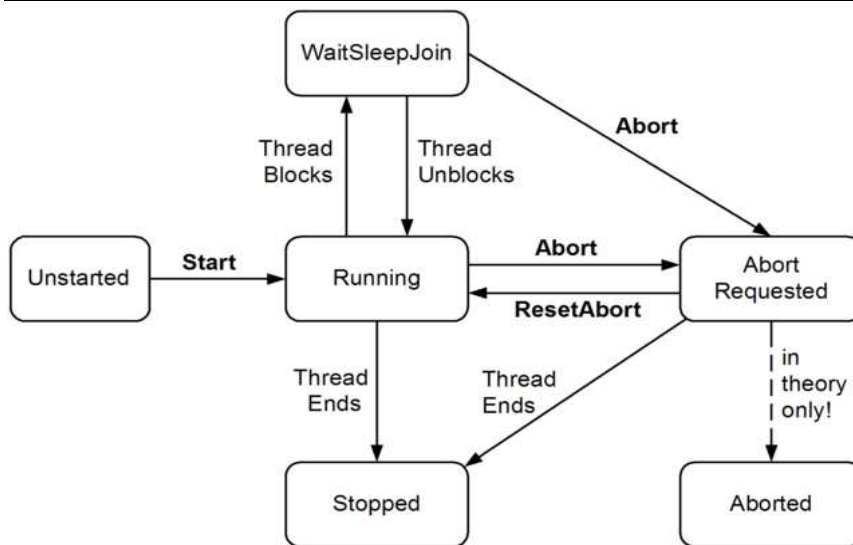
Figure 1: Thread State Diagram

One can query a thread's execution status via its **ThreadState** property. Figure 1 shows one "layer" of the **ThreadState** enumeration. **ThreadState** is horribly designed, in that it combines three "layers" of state using bitwise flags, the members within each layer being themselves mutually exclusive. Here are all three layers:

- the running / blocking / aborting status (as shown in Figure 1)

- the background/foreground status (**ThreadState.Background**)

- the progress towards suspension via the deprecated Suspend method (**ThreadState.SuspendRequested** and **ThreadState.Suspended**)

In total then, **ThreadState** is a bitwise combination of zero or one members from each layer! Here are some sample **ThreadState**s:

```
Unstarted
Running
WaitSleepJoin
Background, Unstarted
SuspendRequested, Background, WaitSleepJoin
```

(The enumeration has two members that are never used, at least in the current CLR implementation: **StopRequested** and **Aborted**.)

To complicate matters further, **ThreadState.Running** has an underlying value of 0, so the following test does not work:

```
if ((t.ThreadState & ThreadState.Running) > 0) ...
```

and one must instead test for a running thread by exclusion, or alternatively, use the thread's **IsAlive** property. **IsAlive**, however, might not be what you want. It returns true if the thread's blocked or suspended (the only time it returns false is before the thread has started, and after it has ended).

Assuming one steers clear of the deprecated **Suspend** and **Resume** methods, one can write a helper method that eliminates all but members of the first layer, allowing simple equality tests to be

performed. A thread's background status can be obtained independently via its IsBackground property, so only the first layer actually has useful information:

```
public static ThreadState SimpleThreadState (ThreadState ts)
{
  return ts & (ThreadState.Aborted | ThreadState.AbortRequested |
               ThreadState.Stopped | ThreadState.Unstarted |
               ThreadState.WaitSleepJoin);
}
```

ThreadState is invaluable for debugging or profiling. It's poorly suited, however, to coordinating multiple threads, because no mechanism exists by which one can test a **ThreadState** and then act upon that information, without the **ThreadState** potentially changing in the interim.

Wait Handles

The lock statement (aka **Monitor.Enter** / **Monitor.Exit**) is one example of a thread synchronization construct. While **lock** is suitable for enforcing exclusive access to a particular resource or section of code, there are some synchronization tasks for which it's clumsy or inadequate, such as signaling a waiting worker thread to begin a task.

The Win32 API has a richer set of synchronization constructs, and these are exposed in the .NET framework via the **EventWaitHandle**, **Mutex** and **Semaphore** classes. Some are more useful than others: the Mutex class, for instance, mostly doubles up on what's provided by **lock**, while **EventWaitHandle** provides unique signaling functionality.

All three classes are based on the abstract **WaitHandle** class, although behaviorally, they are quite different. One of the things they do all have in common is that they can, optionally, be "named", allowing them to work across all operating system processes, rather than across just the threads in the current process.

EventWaitHandle has two subclasses: AutoResetEvent and ManualResetEvent (neither being related to a C# event or delegate). Both classes derive all their functionality from their base class: their only difference being that they call the base class's constructor with a different argument.

In terms of performance, the overhead with all Wait Handles typically runs in the few-microseconds region. Rarely is this of consequence in the context in which they are used.

> **AutoResetEvent** is the most useful of the **WaitHandle** classes, and is a staple synchronization construct, along with the **lock statement**.

AutoResetEvent

An **AutoResetEvent** is much like a ticket turnstile: inserting a ticket lets exactly one person through. The "auto" in the class's name refers to the fact that an open turnstile automatically closes or "resets" after someone is let through. A thread waits, or blocks, at the turnstile by calling **WaitOne** (wait at this "one" turnstile until it opens) and a ticket is inserted by calling the **Set** method. If a number of threads call **WaitOne**, a queue builds up behind the turnstile. A ticket can come from any thread – in other words, any (unblocked) thread with access to the **AutoResetEvent** object can call **Set** on it to release one blocked thread.

If **Set** is called when no thread is waiting, the handle stays open for as long as it takes until some thread to call **WaitOne.** This behavior helps avoid a race between a thread heading for the turnstile, and a thread inserting a ticket ("oops, inserted the ticket a microsecond too soon, bad luck, now you'll have to wait indefinitely!") However calling Set repeatedly on a turnstile at which no-one is waiting doesn't allow a whole party through when they arrive: only the next single person is let through and the extra tickets are "wasted".

WaitOne accepts an optional timeout parameter – the method then returns false if the wait ended because of a timeout rather than obtaining the signal. **WaitOne** can also be instructed to exit the current synchronization context for the duration of the wait (if an automatic locking regime is in use) in order to prevent excessive blocking.

A **Reset** method is also provided that closes the turnstile – should it be open, without any waiting or blocking.

An **AutoResetEvent** can be created in one of two ways. The first is via its constructor:

```
EventWaitHandle wh = new AutoResetEvent (false);
```

If the boolean argument is true, the handle's **Set** method is called automatically, immediately after construction. The other method of instantiatation is via its base class, **EventWaitHandle:**

```
EventWaitHandle wh = new EventWaitHandle (false,
                                          EventResetMode.Auto);
```

EventWaitHandle's constructor also allows a ManualResetEvent to be created (by specifying **EventResetMode.Manual**).

One should call **Close** on a Wait Handle to release operating system resources once it's no longer required. However, if a Wait Handle is going to be used for the life of an application (as in most of the examples in this section), one can be lazy and omit this step as it will be taken care of automatically during application domain tear-down.

In the following example, a thread is started whose job is simply to wait until signaled by another thread:

```
class BasicWaitHandle {
  static EventWaitHandle wh = new AutoResetEvent (false);

  static void Main() {
    new Thread (Waiter).Start();
    Thread.Sleep (1000);                  // Wait for some time...
    wh.Set();                             // OK - wake it up
  }
  static void Waiter() {
    Console.WriteLine ("Waiting...");
    wh.WaitOne();                         // Wait for notification
    Console.WriteLine ("Notified");
  }
}
```

```
Waiting... (pause) Notified.
```

Creating a Cross-Process EventWaitHandle

EventWaitHandle's constructor also allows a "named" **EventWaitHandle** to be created – capable of operating across multiple processes. The name is simply a string – and can be any value that doesn't unintentionally conflict with someone else's! If the name is already in use on the computer, one gets a

reference to the same underlying **EventWaitHandle**, otherwise the operating system creates a new one. Here's an example:

```
EventWaitHandle wh = new EventWaitHandle (false, EventResetMode.Auto,
  "MyCompany.MyApp.SomeName");
```

If two applications each ran this code, they would be able to signal each other: the wait handle would work across all threads in both processes.

Acknowledgement

Supposing we wish to perform tasks in the background without the overhead of creating a new thread each time we get a task. We can achieve this with a single worker thread that continually loops – waiting for a task, executing it, and then waiting for the next task. This is a common multithreading scenario. As well as cutting the overhead in creating threads, task execution is serialized, eliminating the potential for unwanted interaction between multiple workers and excessive resource consumption.

We have to decide what to do, however, if the worker's already busy with previous task when a new task comes along. Suppose in this situation we choose to block the caller until the previous task is complete. Such a system can be implemented using two **AutoResetEvent** objects: a "ready" **AutoResetEvent** that's **Set** by the worker when it's ready, and a "go" **AutoResetEvent** that's **Set** by the calling thread when there's a new task. In the example below, a simple string field is used to describe the task (declared using the volatile keyword to ensure both threads always see the same version):

```
class AcknowledgedWaitHandle {
  static EventWaitHandle ready = new AutoResetEvent (false);
  static EventWaitHandle go = new AutoResetEvent (false);
  static volatile string task;

  static void Main() {
    new Thread (Work).Start();

    // Signal the worker 5 times
    for (int i = 1; i <= 5; i++) {
      ready.WaitOne();                  // First wait until worker is ready
      task = "a".PadRight (i, 'h');   // Assign a task
      go.Set();                         // Tell worker to go!
    }

    // Tell the worker to end using a null-task
    ready.WaitOne(); task = null; go.Set();
  }

  static void Work() {
    while (true) {
      ready.Set();                          // Indicate that we're ready
      go.WaitOne();                         // Wait to be kicked off...
      if (task == null) return;             // Gracefully exit
      Console.WriteLine (task);
    }
  }
}
```

```
ah
ahh
ahhh
ahhhh
```

Notice that we assign a null task to signal the worker thread to exit. Calling Interrupt or Abort on the worker's thread in this case would work equally well – providing we first called **ready.WaitOne**. This is because after calling **ready.WaitOne** we can be certain on the location of the worker – either on or just before the **go.WaitOne** statement – and thereby avoid the complications of interrupting arbitrary code. Calling **Interrupt** or **Abort** would also also require that we caught the consequential exception in the worker.

Producer/Consumer Queue

Another common threading scenario is to have a background worker process tasks from a queue. This is called a Producer/Consumer queue: the producer enqueues tasks; the consumer dequeues tasks on a worker thread. It's rather like the previous example, except that the caller doesn't get blocked if the worker's already busy with a task.

A Producer/Consumer queue is scaleable, in that multiple consumers can be created – each servicing the same queue, but on a separate thread. This is a good way to take advantage of multi-processor systems while still restricting the number of workers so as to avoid the pitfalls of unbounded concurrent threads (excessive context switching and resource contention).

In the example below, a single **AutoResetEvent** is used to signal the worker, which waits only if it runs out of tasks (when the queue is empty). A generic collection class is used for the queue, whose access must be protected by a lock to ensure thread-safety. The worker is ended by enqueing a null task:

```
using System;
using System.Threading;
using System.Collections.Generic;

class ProducerConsumerQueue : IDisposable {
  EventWaitHandle wh = new AutoResetEvent (false);
  Thread worker;
  object locker = new object();
  Queue<string> tasks = new Queue<string>();

  public ProducerConsumerQueue() {
    worker = new Thread (Work);
    worker.Start();
  }

  public void EnqueueTask (string task) {
    lock (locker) tasks.Enqueue (task);
    wh.Set();
  }

  public void Dispose() {
    EnqueueTask (null);     // Signal the consumer to exit.
    worker.Join();          // Wait for the consumer's thread to finish.
    wh.Close();             // Release any OS resources.
  }

  void Work() {
    while (true) {
      string task = null;
      lock (locker)
        if (tasks.Count > 0) {
          task = tasks.Dequeue();
          if (task == null) return;
        }
      if (task != null) {
```

```
          Console.WriteLine ("Performing task: " + task);
          Thread.Sleep (1000);   // simulate work...
        }
        else
          wh.WaitOne();                // No more tasks - wait for a signal
      }
    }
  }
}
```

Here's a main method to test the queue:

```
class Test {
  static void Main() {
    using (ProducerConsumerQueue q = new ProducerConsumerQueue()) {
      q.EnqueueTask ("Hello");
      for (int i = 0; i < 10; i++) q.EnqueueTask ("Say " + i);
      q.EnqueueTask ("Goodbye!");
    }
    // Exiting the using statement calls q's Dispose method, which
    // enqueues a null task and waits until the consumer finishes.
  }
}
```

```
Performing task: Hello
Performing task: Say 1
Performing task: Say 2
Performing task: Say 3

…

…
Performing task: Say 9
Goodbye!
```

Note that in this example we explicitly close the Wait Handle when our **ProducerConsumerQueue** is disposed – since we could potentially create and destroy many instances of this class within the life of the application.

ManualResetEvent

A **ManualResetEvent** is a variation on AutoResetEvent. It differs in that it doesn't automatically reset after a thread is let through on a **WaitOne** call, and so functions like a gate: calling **Set** opens the gate, allowing any number of threads that WaitOne at the gate through; calling **Reset** closes the gate, causing, potentially, a queue of waiters to accumulate until its next opened.

One could simulate this functionality with a boolean "gateOpen" field (declared with the volatile **keyword)** in combination with **"spin-sleeping" –** repeatedly checking the flag, and then sleeping for a short period of time.

ManualResetEvents are sometimes used to signal that a particular operation is complete, or that a thread's completed initialization and is ready to perform work.

Mutex

Mutex provides the same functionality as C#'s lock statement, making **Mutex** mostly redundant. Its one advantage is that it can work across multiple processes – providing a computer-wide lock rather than an application-wide lock.

> While **Mutex** is reasonably fast, **lock** is a hundred times faster again. Acquiring a **Mutex** takes a few microseconds; acquiring a **lock** takes tens of nanoseconds (assuming no blocking).

With a **Mutex** class, the **WaitOne** method obtains the exclusive lock, blocking if it's contended. The exclusive lock is then released with the **ReleaseMutex** method. Just like with C#'s **lock** statement, a **Mutex** can only be released from the same thread that obtained it.

A common use for a cross-process **Mutex** is to ensure that only instance of a program can run at a time. Here's how it's done:

```csharp
class OneAtATimePlease {
  // Use a name unique to the application (eg include your company URL)
  static Mutex mutex = new Mutex (false, "oreilly.com OneAtATimeDemo");

  static void Main() {
    // Wait 5 seconds if contended – in case another instance
    // of the program is in the process of shutting down.

    if (!mutex.WaitOne (TimeSpan.FromSeconds (5), false)) {
      Console.WriteLine ("Another instance of the app is running. Bye!");
      return;
    }
    try {
      Console.WriteLine ("Running - press Enter to exit");
      Console.ReadLine();
    }
    finally { mutex.ReleaseMutex(); }
  }
}
```

A good feature of **Mutex** is that if the application terminates without **ReleaseMutex** first being called, the CLR will release the **Mutex** automatically.

Semaphore

A **Semaphore** is like a nightclub: it has a certain capacity, enforced by a bouncer. Once full, no more people can enter the nightclub and a queue builds up outside. Then, for each person that leaves, one person can enter from the head of the queue. The constructor requires a minimum of two arguments – the number of places currently available in the nightclub, and the nightclub's total capacity.

A **Semaphore** with a capacity of one is similar to a **Mutex** or **lock**, except that the **Semaphore** has no "owner" – it's *thread-agnostic*. Any thread can call **Release** on a **Semaphore**, while with **Mutex** and **lock**, only the thread that obtained the resource can release it.

In this following example, ten threads execute a loop with a **Sleep** statement in the middle. A **Semaphore** ensures that not more than three threads can execute that **Sleep** statement at once:

```
class SemaphoreTest {
  static Semaphore s = new Semaphore (3, 3);  // Available=3; Capacity=3

  static void Main() {
    for (int i = 0; i < 10; i++) new Thread (Go).Start();
  }

  static void Go() {
    while (true) {
      s.WaitOne();
      Thread.Sleep (100);   // Only 3 threads can get here at once
      s.Release();
    }
  }
}
```

WaitAny, WaitAll and SignalAndWait

In addition to the **Set** and **WaitOne** methods, there are static methods on the **WaitHandle** class to crack more complex synchronization nuts.

The **WaitAny**, **WaitAll** and **SignalAndWait** methods facilitate waiting across multiple Wait Handles, potentially of differing types.

SignalAndWait is perhaps the most useful: it calls **WaitOne** on one **WaitHandle**, while calling **Set** on another **WaitHandle** – in an atomic operation. One can use method this on a pair of **EventWaitHandle**s to set up two threads so they "meet" at the same point in time, in a textbook fashion. Either **AutoResetEvent** or **ManualResetEvent** will do the trick. The first thread does the following:

```
WaitHandle.SignalAndWait (wh1, wh2);
```

while the second thread does the opposite:

```
WaitHandle.SignalAndWait (wh2, wh1);
```

WaitHandle.WaitAny waits for any one of an array of wait handles; **WaitHandle.WaitAll** waits on all of the given handles. Using the ticket turnstile analogy, these methods are like simultaneously queuing at all the turnstiles – going through at the first one to open (in the case of **WaitAny**), or waiting until they all open (in the case of **WaitAll**).

WaitAll is actually of dubious value because of a weird connection to apartment threading – a throwback from the legacy COM architecture. **WaitAll** requires that the caller be in a multi-threaded apartment – which happens to be the apartment model least suitable for interoperability – particularly for Windows Forms applications, which need to perform tasks as mundane as interacting with the clipboard!

Fortunately, the .NET framework provides a more advanced signaling mechanism for when Wait Handles are awkward or unsuitable – **Monitor.Wait** and **Monitor.Pulse**.

Synchronization Contexts

Rather than locking manually, one can lock declaratively. By deriving from **ContextBoundObject** and applying the **Synchronization** attribute, one instructs the CLR to apply locking automatically. Here's an example:

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

[Synchronization]
public class AutoLock : ContextBoundObject {
  public void Demo() {
    Console.Write ("Start...");
    Thread.Sleep (1000);          // We can't be preempted here
    Console.WriteLine ("end");    // thanks to automatic locking!
  }
}

public class Test {
  public static void Main() {
    AutoLock safeInstance = new AutoLock();
    new Thread (safeInstance.Demo).Start();    // Call the Demo
    new Thread (safeInstance.Demo).Start();    // method 3 times
    safeInstance.Demo();                        // concurrently.
  }
}
```

```
Start... end
Start... end
Start... end
```

The CLR ensures that only one thread can execute code in **safeInstance** at a time. It does this by creating a single synchronizing object – and locking it around every call to each of **safeInstance**'s methods or properties. The scope of the lock – in this case – the **safeInstance** object – is called a *synchronization context*.

So, how does this work? A clue is in the **Synchronization** attribute's namespace: **System.Runtime.Remoting.Contexts**. A **ContextBoundObject** can be thought of as a "remote" object – meaning all method calls are intercepted. To make this interception possible, when we instantiate **AutoLock**, the CLR actually returns a proxy – an object with the same methods and properties of an **AutoLock** object, which acts as an intermediary. It's via this intermediary that the automatic locking takes place. Overall, the interception adds around a microsecond to each method call.

> Automatic synchronization cannot be used to protect static type members, nor classes not derived from **ContextBoundObject** (for instance, a Windows **Form**).

The locking is applied internally in the same way. You might expect that the following example will yield the same result as the last:

```
[Synchronization]
public class AutoLock : ContextBoundObject {
  public void Demo() {
    Console.Write ("Start...");
    Thread.Sleep (1000);
    Console.WriteLine ("end");
  }

  public void Test() {
    new Thread (Demo).Start();
    new Thread (Demo).Start();
    new Thread (Demo).Start();
    Console.ReadLine();
  }

  public static void Main() {
    new AutoLock().Test();
  }
}
```

(Notice that we've sneaked in a **Console.ReadLine** statement). Because only one thread can execute code at a time in an object of this class, the three new threads will remain blocked at the **Demo** method until the **Test** method finishes – which requires the **ReadLine** to complete. Hence we end up with the same result as before, but only after pressing the **Enter** key. This is a thread-safety hammer almost big enough to preclude any useful multithreading within a class!

Furthermore, we haven't solved a problem described earlier: if **AutoLock** were a collection class, for instance, we'd still require a lock around a statement such as the following, assuming it ran from another class:

```
if (safeInstance.Count > 0) safeInstance.RemoveAt (0);
```

unless this code's class was itself a synchronized **ContextBoundObject**!

A synchronization context can extend beyond the scope of a single object. By default, if a synchronized object is instantiated from within the code of another, both share the same context (in other words, one big lock!) This behavior can be changed by specifying an integer flag in **Synchronization** attribute's constructor, using one of the constants defined in the **SynchronizationAttribute** class:

Constant	Meaning
NOT_SUPPORTED	Equivalent to not using the **Synchronized** attribute
SUPPORTED	Joins the existing synchronization context if instantiated from another synchronized object, otherwise remains unsynchronized
REQUIRED (default)	Joins the existing synchronization context if instantiated from another synchronized object, otherwise creates a new context
REQUIRES_NEW	Always creates a new synchronization context

So if object of class *SynchronizedA* instantiates an object of class *SynchronizedB*, they'll be given separate synchronization contexts if *SynchronizedB* is declared as follows:

```
[Synchronization (SynchronizationAttribute.REQUIRES_NEW)]
public class SynchronizedB : ContextBoundObject { ...
```

The bigger the scope of a synchronization context, the easier it is to manage, but the less the opportunity for useful concurrency. At the other end of the scale, separate synchronization contexts invite deadlocks. Here's an example:

```
[Synchronization]
public class Deadlock : ContextBoundObject {
  public DeadLock Other;
  public void Demo() { Thread.Sleep (1000); Other.Hello(); }
  void Hello()      { Console.WriteLine ("hello");          }
}

public class Test {
  static void Main() {
    Deadlock dead1 = new Deadlock();
    Deadlock dead2 = new Deadlock();
    dead1.Other = dead2;
    dead2.Other = dead1;
    new Thread (dead1.Demo).Start();
    dead2.Demo();
  }
}
```

Because each instance of **Deadlock** is created within **Test** – an unsynchronized class – each instance will gets its own synchronization context, and hence, its own lock. When the two objects call upon each other, it doesn't take long for the deadlock to occur (one second, to be precise!) The problem would be particularly insidious if the **Deadlock** and **Test** classes were written by different programming teams. It may be unreasonable to expect those responsible for the **Test** class to be even aware of their transgression, let alone know how to go about resolving it. This is in contrast to explicit locks, where deadlocks are usually more obvious.

Reentrancy

A thread-safe method is sometimes called *reentrant*, because it can be preempted part way through its execution, and then called again on another thread without ill effect. In a general sense, the terms *thread-safe* and *reentrant* are considered either synonymous or closely related.

Reentrancy, however, has another more sinister connotation in automatic locking regimes. If the **Synchronization** attribute is applied with the **reentrant** argument true:

```
[Synchronization(true)]
```

then the synchronization context's lock will be temporarily released when execution leaves the context. In the previous example, this would prevent the deadlock from occurring; obviously desirable. However, a side effect is that during this interim, any thread is free to call any method on the original object ("re-entering" the synchronization context) and unleashing the very complications of multithreading one is trying to avoid in the first place. This is the problem of *reentrancy*.

Because **[Synchronization(true)]** is applied at a class-level, this attribute turns every out-of-context method call made by the class into a Trojan for reentrancy.

While reentrancy can be dangerous, there are sometimes few other options. For instance, suppose one was to implement multithreading internally within a synchronized class, by delegating the logic to workers running objects in separate contexts. These workers may be unreasonably hindered in communicating with each other or the original object without reentrancy.

This highlights a fundamental weakness with automatic synchronization: the extensive scope over which locking is applied can actually manufacture difficulties that may never have otherwise arisen. These difficulties – deadlocking, reentrancy, and emasculated concurrency – can make manual locking more palatable in anything other than simple scenarios.

PART 3
USING THREADS

Apartments and Windows Forms

Apartment threading is an automatic thread-safety regime, closely allied to COM – Microsoft's legacy Component Object Model. While .NET largely breaks free of legacy threading models, there are times when it still crops up because of the need to interoperate with older APIs. Apartment threading is most relevant to Windows Forms, because much of Windows Forms uses or wraps the long-standing Win32 API – complete with its apartment heritage.

An apartment is a logical "container" for threads. Apartments come in two sizes – "single" and "multi". A single-threaded apartment contains just one thread; multi-threaded apartments can contain any number of threads. The single-threaded model is the more common and interoperable of the two.

As well as containing threads, apartments contain objects. When an object is created within an apartment, it stays there all its life, forever house-bound along with the resident thread(s). This is similar to an object being contained within a .NET synchronization context, except that a synchronization context does not own or contain threads. Any thread can call upon an object in any synchronization context – subject to waiting for the exclusive lock. But objects contained within an apartment can only be called upon by a thread within the apartment.

Imagine a library, where each book represents an object. Borrowing is not permitted – books created in the library stay there for life. Furthermore, let's use a person to represent a thread.

A synchronization context library allows any person to enter, as long as only one person enters at a time. Any more, and a queue forms outside the library.

An apartment library has resident staff – a single librarian for a single-threaded library, and whole team for a multi-threaded library. No-one is allowed in other than members of staff – a patron wanting to perform research must signal a librarian, then ask the librarian to do the job! Signaling the librarian is called *marshalling* – the patron *marshals* the method call over to a member of staff (or, the member of staff!) Marshalling is automatic, and is implemented at the librarian-end via a message pump – in Windows Forms, this is the mechanism that constantly checks for keyboard and mouse events from the operating system. If messages arrive too quickly to be processed, they enter a message queue, so they can be processed in the order they arrive.

Specifying an Apartment Model

A .NET thread is automatically assigned an apartment upon entering apartment-savvy Win32 or legacy COM code. By default, it will be allocated a multi-threaded apartment, unless one requests a single-threaded apartment as follows:

```
Thread t = new Thread (...);
t.SetApartmentState (ApartmentState.STA);
```

One can also request that the main thread join a single-threaded apartment using the **STAThread** attribute on the main method:

```
class Program {
  [STAThread]
  static void Main() {
    ...
```

Apartments have no effect while executing pure .NET code. In other words, two threads with an apartment state of **STA** can simultaneously call the same method on the same object, and no automatic marshalling or locking will take place. Only when execution hits unmanaged code can they kick in.

The types in the **System.Windows.Forms** namespace extensively call Win32 code designed to work in a single-threaded apartment. For this reason, a Windows Forms program should have have the **[STAThread]** attribute on its main method, otherwise one of two things will occur upon reaching Win32 UI code:

- it will marshal over to a single-threaded apartment

- it will crash

Control.Invoke

In a multi-threaded Windows Forms application, it's illegal to call a method or property on a control from any thread other than the one that created it. All cross-thread calls must be explicitly marshalled to the thread that created the control (usually the main thread), using the **Control.Invoke** or **Control.BeginInvoke** method. One cannot rely on automatic marshalling because it takes place too late – only when execution gets well into unmanaged code, by which time plenty of internal .NET code may already have run on the "wrong" thread – code which is not thread-safe.

> WPF is similar to Windows Forms in that elements can be accessed only from the thread that originally created them. The equivalent to **Control.Invoke** in WPF is **Dispatcher.Invoke**.

An excellent solution to managing worker threads in Windows Forms and WPF applications is to use BackgroundWorker. This class wraps worker threads that need to report progress and completion, and automatically calls **Control.Invoke** or **Dispatcher.Invoke** as required.

BackgroundWorker

BackgroundWorker is a helper class in the **System.ComponentModel** namespace for managing a worker thread. It provides the following features:

- A "cancel" flag for signaling a worker to end without using Abort

- A standard protocol for reporting progress, completion and cancellation

- An implementation of **IComponent** allowing it be sited in the Visual Studio Designer

- Exception handling on the worker thread

- The ability to update Windows Forms and WPF controls in response to worker

progress or completion.

The last two features are particularly useful – it means you don't have to include a **try/catch** block in your worker method, and can update Windows Forms and WPF controls without needing to call **Control.Invoke**.

BackgroundWorker uses the thread-pool, which recycles threads to avoid recreating them for each new task. This means one should never call Abort on a **BackgroundWorker** thread.

Here are the minimum steps in using **BackgroundWorker**:

- Instantiate **BackgroundWorker**, and handle the **DoWork** event

- Call **RunWorkerAsync**, optionally with an object argument.

This then sets it in motion. Any argument passed to **RunWorkerAsync** will be forwarded to **DoWork**'s event handler, via the event argument's **Argument** property. Here's an example:

```
class Program {
  static BackgroundWorker bw = new BackgroundWorker();
  static void Main() {
    bw.DoWork += bw_DoWork;
    bw.RunWorkerAsync ("Message to worker");
    Console.ReadLine();
  }

  static void bw_DoWork (object sender, DoWorkEventArgs e) {
    // This is called on the worker thread
    Console.WriteLine (e.Argument);        // writes "Message to worker"
    // Perform time-consuming task...
  }
```

BackgroundWorker also provides a **RunWorkerCompleted** event which fires after the **DoWork** event handler has done its job. Handling **RunWorkerCompleted** is not mandatory, but one usually does so in order to query any exception that was thrown in DoWork. Furthermore, code within a **RunWorkerCompleted** event handler is able to update Windows Forms and WPF controls without explicit marshalling; code within the **DoWork** event handler cannot.

To add support for progress reporting:

- Set the **WorkerReportsProgress** property to true

- Periodically call **ReportProgress** from within the **DoWork** event handler with a "percentage complete" value, and optionally, a user-state object

- Handle the **ProgressChanged** event, quering its event argument's **ProgressPercentage** property

Code in the **ProgressChanged** event handler is free to interact with UI controls just as with **RunWorkerCompleted**. This is typically where you will update a progress bar.

To add support for cancellation:

- Set the **WorkerSupportsCancellation** property to true

- Periodically check the **CancellationPending** property from within the **DoWork** event handler – if true, set the event argument's **Cancel** property true, and return. (The worker can set **Cancel** true and exit without prompting via **CancellationPending** – if it decides the job's too difficult and it can't go on).

- Call **CancelAsync** to request cancellation.

Here's an example that implements all the above features:

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

class Program {
  static BackgroundWorker bw;
  static void Main() {
    bw = new BackgroundWorker();
    bw.WorkerReportsProgress = true;
    bw.WorkerSupportsCancellation = true;
    bw.DoWork += bw_DoWork;
    bw.ProgressChanged += bw_ProgressChanged;
    bw.RunWorkerCompleted += bw_RunWorkerCompleted;

    bw.RunWorkerAsync ("Hello to worker");

    Console.WriteLine ("Press Enter in next 5 seconds to cancel");
    Console.ReadLine();
    if (bw.IsBusy) bw.CancelAsync();
    Console.ReadLine();
  }

  static void bw_DoWork (object sender, DoWorkEventArgs e) {
    for (int i = 0; i <= 100; i += 20) {
      if (bw.CancellationPending) {
        e.Cancel = true;
        return;
      }
      bw.ReportProgress (i);
      Thread.Sleep (1000);
    }
    e.Result = 123;    // This gets passed to RunWorkerCopmleted
  }

  static void bw_RunWorkerCompleted (object sender,
  RunWorkerCompletedEventArgs e) {
    if (e.Cancelled)
      Console.WriteLine ("You cancelled!");
    else if (e.Error != null)
      Console.WriteLine ("Worker exception: " + e.Error.ToString());
    else
      Console.WriteLine ("Complete - " + e.Result);  // from DoWork
  }

  static void bw_ProgressChanged (object sender,
                                  ProgressChangedEventArgs e) {
    Console.WriteLine ("Reached " + e.ProgressPercentage + "%");
  }
}
```

```
Press Enter in the next 5 seconds to cancel
Reached 0%
Reached 20%
Reached 40%
Reached 60%
Reached 80%
Reached 100%
Complete – 123

Press Enter in the next 5 seconds to cancel
Reached 0%
Reached 20%
Reached 40%

You cancelled!
```

Subclassing BackgroundWorker

BackgroundWorker is not sealed and provides a virtual **OnDoWork** method, suggesting another pattern for its use. When writing a potentially long-running method, one could instead – or as well – write a version returning a subclassed **BackgroundWorker**, pre-configured to perform the job asynchronously. The consumer then only need handle the **RunWorkerCompleted** and **ProgressChanged** events. For instance, suppose we wrote a time-consuming method called **GetFinancialTotals**:

```csharp
public class Client {
  Dictionary <string,int> GetFinancialTotals (int foo, int bar) { ... }
  ...
}
```

We could refactor it as follows:

```csharp
public class Client {
  public FinancialWorker GetFinancialTotalsBackground (int foo, int bar) {
    return new FinancialWorker (foo, bar);
  }
}

public class FinancialWorker : BackgroundWorker {
  public Dictionary <string,int> Result;   // We can add typed fields.
  public volatile int Foo, Bar;            // We could even expose them
                                           // via properties with locks!
  public FinancialWorker() {
    WorkerReportsProgress = true;
    WorkerSupportsCancellation = true;
  }

  public FinancialWorker (int foo, int bar) : this() {
    this.Foo = foo; this.Bar = bar;
  }
```

```
protected override void OnDoWork (DoWorkEventArgs e) {
    ReportProgress (0, "Working hard on this report...");
    Initialize financial report data

    while (!finished report ) {
        if (CancellationPending) {
            e.Cancel = true;
            return;
        }
        Perform another calculation step
        ReportProgress (percentCompleteCalc, "Getting there...");
    }
    ReportProgress (100, "Done!");
    e.Result = Result = completed report data;
    }
}
```

Whoever calls **GetFinancialTotalsBackground** then gets a **FinancialWorker** – a wrapper to manage the background operation with real-world usability. It can report progress, be cancelled, and is compatible with Windows Forms without **Control.Invoke**. It's also exception-handled, and uses a standard protocol (in common with that of anyone else using **BackgroundWorker**!)

This usage of **BackgroundWorker** effectively deprecates the old "event-based asynchronous pattern".

ReaderWriterLockSlim / ReaderWriterLock

Quite often, instances of a type are thread-safe for concurrent read operations, but not for concurrent updates (nor for a concurrent read and update). This can also be true with resources such as a file. Although protecting instances of such types with a simple exclusive lock for all modes of access usually does the trick, it can unreasonably restrict concurrency if there are many readers and just occasional updates. An example of where this could occur is in a business application server, where commonly used data is cached for fast retrieval in static fields. The **ReaderWriterLockSlim** class is designed to provide maximum-availability locking in just this scenario.

> **ReaderWriterLockSlim** is new to Framework 3.5 and is a replacement for the older "fat" **ReaderWriterLock** class. The latter is similar in functionality, but is several times slower and has an inherent design fault in its mechanism for handling lock upgrades.

With both classes, there are two basic kinds of lock: a read lock and a write lock. A write lock is universally exclusive, whereas a read lock is compatible with other read locks.

So, a thread holding a write lock blocks all other threads trying to obtain a read or write lock (and vice versa). But if no thread holds a write lock, any number of threads may concurrently obtain a read lock.

ReaderWriterLockSlim defines the following methods for obtaining and releasing read/write locks:

```
public void EnterReadLock();
public void ExitReadLock();
public void EnterWriteLock();
public void ExitWriteLock();
```

Additionally, there are "Try" versions of all **EnterXXX** methods which accept timeout arguments in the style of **Monitor.TryEnter** (timeouts can occur quite easily if the resource is heavily contended). **ReaderWriterLock** provides similar methods, named **AcquireXXX** and **ReleaseXXX**. These throw an **ApplicationException** if a timeout occurs rather than returning false.

The following program demonstrates **ReaderWriterLockSlim**. Three threads continually enumerate a list, while two further threads append a random number to the list every second. A read lock protects the list readers and a write lock protects the list writers:

```
class SlimDemo
{
  static ReaderWriterLockSlim rw = new ReaderWriterLockSlim();
  static List<int> items = new List<int>();
  static Random rand = new Random();

  static void Main()
  {
    new Thread (Read).Start();
    new Thread (Read).Start();
    new Thread (Read).Start();

    new Thread (Write).Start ("A");
    new Thread (Write).Start ("B");
  }

  static void Read()
  {
    while (true)
    {
      rw.EnterReadLock();
      foreach (int i in items) Thread.Sleep (10);
      rw.ExitReadLock();
    }
  }

  static void Write (object threadID)
  {
    while (true)
    {
      int newNumber = GetRandNum (100);
      rw.EnterWriteLock();
      items.Add (newNumber);
      rw.ExitWriteLock();
      Console.WriteLine ("Thread " + threadID + " added " + newNumber);
      Thread.Sleep (100);
    }
  }

  static int GetRandNum (int max) { lock (rand) return rand.Next (max); }
}
```

> In production code, you'd typically add **try/finally** blocks to ensure locks were released if an exception was thrown.

Here's the result:

```
Thread B added 61
Thread A added 83
```

```
Thread B added 55
Thread A added 33
...
```

ReaderWriterLockSlim allows more concurrent **Read** activity than would a simple lock. We can illustrate this by inserting the following line to the **Write** method, at the start of the **while** loop:

```
Console.WriteLine (rw.CurrentReadCount + " concurrent readers");
```

This nearly always prints "3 concurrent readers" (the **Read** methods spend most their time inside the **foreach** loops). As well as **CurrentReadCount**, **ReaderWriterLockSlim** provides the following properties for monitoring locks:

```
public bool IsReadLockHeld            { get; }
public bool IsUpgradeableReadLockHeld { get; }
public bool IsWriteLockHeld           { get; }

public int  WaitingReadCount          { get; }
public int  WaitingUpgradeCount       { get; }
public int  WaitingWriteCount         { get; }

public int  RecursiveReadCount        { get; }
public int  RecursiveUpgradeCount     { get; }
public int  RecursiveWriteCount       { get; }
```

Sometimes it's useful to swap a read lock for a write lock in a single atomic operation. For instance, suppose you wanted to add an item to a list only if the item wasn't already present. Ideally, you'd want to minimize the time spent holding the (exclusive) write lock, so you might proceed as follows:

1. Obtain a read lock

2. Test if the item is already present in the list, and if so, release the lock and **return**

3. Release the read lock

4. Obtain a write lock

5. Add the item

The problem is that another thread could sneak in and modify the list (adding the same item, for instance) between steps 3 and 4. **ReaderWriterLockSlim** addresses this through a third kind of lock called an *upgradeable lock*. An upgradeable lock is like a read lock except that it can later be promoted to a write lock in an atomic operation. Here's how you use it:

1. Call **EnterUpgradeableReadLock**

2. Perform read-based activities (e.g. test if item already present in list)

3. Call **EnterWriteLock** (this converts the upgradeable lock to a write lock)

4. Perform write-based activities (e.g. add item to list)

5. Call **ExitWriteLock** (this converts the write lock back to an upgradeable lock)

6. Perform any other read-based activities

7. Call **ExitUpgradeableReadLock**

From the caller's perspective, it's rather like nested or recursive locking. Functionally, though, in step 3, **ReaderWriterLockSlim** releases your read-lock and obtains a fresh write-lock, atomically.

There's another important difference between upgradeable locks and read locks. While an upgradable lock can coexist with any number of *read* locks, only one upgradeable lock can itself be taken out at a time. This prevents conversion deadlocks by *serializing* competing conversions—just as update locks do in SQL Server:

SQL Server	ReaderWriterLockSlim
Share lock	Read lock
Exclusive lock	Write lock
Update lock	Upgradeable lock

We can demonstrate an upgradeable lock by by changing the **Write** method in the preceding example such that it adds a number to list only if not already present:

```
while (true)
{
  int newNumber = GetRandNum (100);
  rw.EnterUpgradeableReadLock();
  if (!items.Contains (newNumber))
  {
    rw.EnterWriteLock();
    items.Add (newNumber);
    rw.ExitWriteLock();
    Console.WriteLine ("Thread " + threadID + " added " + newNumber);
  }
  rw.ExitUpgradeableReadLock();
  Thread.Sleep (100);
}
```

> **ReaderWriterLock** can also do lock conversions—but unreliably because it doesn't support the concept of upgradeable locks. This is why the designers of **ReaderWriterLockSlim** had to start afresh with a new class.

Lock recursion

Ordinarily, nested or recursive locking is prohibited with ReaderWriterLockSlim. Hence, the following throws an exception:

```
var rw = new ReaderWriterLockSlim();
rw.EnterReadLock();
rw.EnterReadLock();
rw.ExitReadLock();
rw.ExitReadLock();
```

It runs without error, however, if you construct **ReaderWriterLockSlim** as follows:

```
var rw = new ReaderWriterLockSlim (LockRecursionPolicy.SupportsRecursion);
```

This ensures that recursive locking can happen only if you plan for it. Recursive locking can bring undesired complexity because it's possible to acquire more than one kind of lock:

```
rw.EnterWriteLock();
rw.EnterReadLock();
Console.WriteLine (rw.IsReadLockHeld);     // True
Console.WriteLine (rw.IsWriteLockHeld);    // True
```

```
      rw.ExitReadLock();
      rw.ExitWriteLock();
```

The basic rule is that once you've acquired a lock, subsequent recursive locks can less, but not greater, on the following scale:

Read Lock --> Upgradeable Lock --> Write Lock

A request to promote an upgradeable lock to a write lock, however, is always legal.

Thread Pooling

If your application has lots of threads that spend most of their time blocked on a Wait Handle, you can reduce the resource burden via *thread pooling*. A thread pool economizes by coalescing many Wait Handles onto a few threads.

To use the thread pool, you register a Wait Handle along with a delegate to be executed when the Wait Handle is signaled. This is done by calling **ThreadPool.RegisterWaitForSingleObject**, such in this example:

```
class Test {
  static ManualResetEvent starter = new ManualResetEvent (false);

  public static void Main() {
    ThreadPool.RegisterWaitForSingleObject (starter, Go,
                                            "hello", -1, true);
    Thread.Sleep (5000);
    Console.WriteLine ("Signaling worker...");
    starter.Set();
    Console.ReadLine();
  }

  public static void Go (object data, bool timedOut) {
    Console.WriteLine ("Started " + data);
    // Perform task...
  }
}
```

```
(5 second delay)
Signaling worker...
Started hello
```

In addition to the Wait Handle and delegate, **RegisterWaitForSingleObject** accepts a "black box" object which it passes to your delegate method (rather like with a ParameterizedThreadStart), as well as a timeout in milliseconds (-1 meaning no timeout) and a boolean flag indicating if the request is one-off rather than recurring.

All pooled threads are *background* threads, meaning they terminate automatically when the application's foreground thread(s) end. However if one wanted to wait until any important jobs running on pooled threads completed before exiting an application, calling Join on the threads would not be an option, since pooled threads never finish! The idea is that they are instead recycled, and end only when the parent process terminates. So in order to know when a job running on a pooled thread has finished, one must signal – for instance, with another Wait Handle.

Calling **Abort** on a pooled thread is Bad Idea. The threads need to be recycled for the life of the application domain.

You can also use the thread pool without a Wait Handle by calling the **QueueUserWorkItem** method – specifying a delegate for immediate execution. You don't then get the saving of sharing threads amongst multiple jobs, but do get another benefit: the thread pool keeps a lid on the total number of threads (25, by default), automatically enqueuing tasks when the job count goes above this. It's rather like an application-wide producer-consumer queue with 25 consumers! In the following example, 100 jobs are enqueued to the thread pool, of which 25 execute at a time. The main thread then waits until they're all complete using Wait and Pulse:

```csharp
class Test {
  static object workerLocker = new object ();
  static int runningWorkers = 100;

  public static void Main() {
    for (int i = 0; i < runningWorkers; i++) {
      ThreadPool.QueueUserWorkItem (Go, i);
    }
    Console.WriteLine ("Waiting for threads to complete...");
    lock (workerLocker) {
      while (runningWorkers > 0) Monitor.Wait (workerLocker);
    }
    Console.WriteLine ("Complete!");
    Console.ReadLine();
  }

  public static void Go (object instance) {
    Console.WriteLine ("Started: " + instance);
    Thread.Sleep (1000);
    Console.WriteLine ("Ended: " + instance);
    lock (workerLocker) {
      runningWorkers--; Monitor.Pulse (workerLocker);
    }
  }
}
```

In order to pass more than a single object to the target method, one can either define a custom object with all the required properties, or call via an anonmymous method. For instance, if the **Go** method accepted two integer parameters, it could be started as follows:

```csharp
ThreadPool.QueueUserWorkItem (delegate (object notUsed) { Go (23,34); });
```

Another way into the thread pool is via *asynchronous delegates*.

Asynchronous Delegates

In Part 1 we described how to pass data to a thread, using ParameterizedThreadStart. Sometimes you need to go the other way, and get return values back from a thread when it finishes executing. Asynchronous delegates offer a convenient mechanism for this, allowing any number of typed arguments to be passed in both directions. Furthermore, unhandled exceptions on asynchronous delegates are conveniently re-thrown on the original thread, and so don't need explicit handling. Asynchronous delegates also provide another way into the thread pool.

The price you must pay for all this is in following its asynchronous model. To see what this means, we'll first discuss the more usual, synchronous, model of programming. Let's say we want to compare two web pages. We could achieve this by downloading each page in sequence, then comparing their output as follows:

```
static void ComparePages() {
  WebClient wc = new WebClient ();
  string s1 = wc.DownloadString ("http://www.oreilly.com");
  string s2 = wc.DownloadString ("http://oreilly.com");
  Console.WriteLine (s1 == s2 ? "Same" : "Different");
}
```

Of course it would be faster if both pages downloaded at once. One way to view the problem is to blame **DownloadString** for blocking the calling method while the page is downloading. It would be nice if we could call **DownloadString** in a non-blocking asynchronous fashion, in other words:

1. We tell **DownloadString** to start executing.

2. We perform other tasks while it's working, such as downloading another page.

3. We ask **DownloadString** for its results.

> The **WebClient** class actually offers a built-in method called **DownloadStringAsync** which provides asynchronous-like functionality. For now, we'll ignore this and focus on the mechanism by which *any* method can be called asynchronously.

The third step is what makes asynchronous delegates useful. The caller rendezvous with the worker to get results and to allow any exception to be re-thrown. Without this step, we have normal multithreading. While it's possible to use asynchronous delegates without the rendezvous, you gain little over calling ThreadPool.QueueWorkerItem or using BackgroundWorker.

Here's how we can use asynchronous delegates to download two web pages, while simultaneously performing a calculation:

```
delegate string DownloadString (string uri);

static void ComparePages() {

  // Instantiate delegates with DownloadString's signature:
  DownloadString download1 = new WebClient().DownloadString;
  DownloadString download2 = new WebClient().DownloadString;

  // Start the downloads:
  IAsyncResult cookie1 = download1.BeginInvoke (uri1, null, null);
  IAsyncResult cookie2 = download2.BeginInvoke (uri2, null, null);

  // Perform some random calculation:
  double seed = 1.23;
  for (int i = 0; i < 1000000; i++) seed = Math.Sqrt (seed + 1000);

  // Get the results of the downloads, waiting for completion if necessary.
  // Here's where any exceptions will be thrown:
  string s1 = download1.EndInvoke (cookie1);
  string s2 = download2.EndInvoke (cookie2);

  Console.WriteLine (s1 == s2 ? "Same" : "Different");
}
```

We start by declaring and instantiating delegates for methods we want to run asynchronously. In this example, we need two delegates so that each can reference a separate **WebClient** object (**WebClient** does not permit concurrent access—if it did, we could use a single delegate throughout).

We then call **BeginInvoke**. This begins execution while immediately returning control to the caller. In accordance with our delegate, we must pass a string to **BeginInvoke** (the compiler enforces this, by manufacturing typed **BeginInvoke** and **EndInvoke** methods on the delegate type).

BeginInvoke requires two further arguments—an optional callback and data object; these can be left null as they're usually not required. **BeginInvoke** returns an **IASynchResult** object which acts as a cookie for calling **EndInvoke**. The **IASynchResult** object also has the property **IsCompleted** which can be used to check on progress.

We then call **EndInvoke** on the delegates, as their results are needed. **EndInvoke** waits, if necessary, until its method finishes, then returns the method's return value as specified in the delegate (string, in this case). A nice feature of **EndInvoke** is that if the **DownloadString** method had any ref or out parameters, these would be added into **EndInvoke**'s signature, allowing multiple values to be sent back by to the caller.

If at any point during an asynchronous method's execution an unhandled exception is encountered, it's re-thrown on the caller's thread upon calling **EndInvoke**. This provides a tidy mechanism for marshaling exceptions back to the caller.

> If the method you're calling asynchronously has no return value, you are still (technically) obliged to call **EndInvoke**. In a practical sense this is open to interpretation; the MSDN is contradictory on this issue. If you choose not to call **EndInvoke**, however, you'll need to consider exception handling on the worker method.

Asynchronous Methods

Some types in the .NET Framework offer asynchronous versions of their methods, with names starting with "Begin" and "End". These are called asynchronous methods and have signatures similar to those of asynchronous delegates, but exist to solve a much harder problem: *to allow more concurrent activities than you have threads*. A web or TCP sockets server, for instance, can process several hundred concurrent requests on just a handful of pooled threads if written using **NetworkStream.BeginRead** and **NetworkStream.BeginWrite**.

Unless you're writing a high concurrency application, however, you should avoid asynchronous methods for a number of reasons:

- Unlike asynchronous delegates, asynchronous methods may not actually execute in parallel with the caller

- The benefits of asynchronous methods erodes or disappears if you fail to follow the pattern meticulously

- Things can get complex pretty quickly when you do follow the pattern correctly

If you're simply after parallel execution, you're better off calling the synchronous version of the method (e.g. **NetworkStream.Read**) via an asynchronous delegate. Another option is to use **ThreadPool.QueueUserWorkItem** or **BackgroundWorker**—or simply create a new thread. Chapter 20 of C# 3.0 in a Nutshell explains asynchronous methods in detail.

Asynchronous Events

Another pattern exists whereby types can provide asynchronous versions of their methods. This is called the "event-based asynchronous pattern" and is distinguished by a method whose name ends with "Async", and a corresponding event whose name ends in "Completed". The **WebClient** class employs this pattern in its **DownloadStringAsync** method. To use it, you first handle the "Completed" event (e.g. **DownloadStringCompleted**) and then call the "Async" method (e.g. **DownloadStringAsync**). When the method finishes, it calls your event handler. Unfortunately, **WebClient**'s implementation is flawed: methods such as **DownloadStringAsync** block the caller for a portion of the download time.

The event-based pattern also offers events for progress reporting and cancellation, designed to be friendly with Windows applications that update forms and controls. If you need these features in a type that doesn't support the event-based asynchronous model (or doesn't support it correctly!) you don't have to take on the burden of implementing the pattern yourself, however (and you wouldn't want to!) All of this can be achieved more simply with the BackgroundWorker helper class.

Timers

The easiest way to execute a method periodically is using a *timer* – such as the **Timer** class provided in the **System.Threading** namespace. The threading timer takes advantage of the thread pool, allowing many timers to be created without the overhead of many threads. **Timer** is a fairly simple class, with a constructor and just two methods (a delight for minimalists, as well as book authors!)

```
public sealed class Timer : MarshalByRefObject, IDisposable
{
  public Timer (TimerCallback tick, object state, 1st, subsequent);
  public bool Change (1st, subsequent);   // To change the interval
  public void Dispose();                   // To kill the timer
}
1st = time to the first tick in milliseconds or a TimeSpan
subsequent = subsequent intervals in milliseconds or a TimeSpan
  (use Timeout.Infinite for a one-off callback)
```

In the following example, a timer calls the **Tick** method which writes "tick..." after 5 seconds have elapsed, then every second after that – until the user presses **Enter**:

```
using System;
using System.Threading;

class Program {
  static void Main() {
    Timer tmr = new Timer (Tick, "tick...", 5000, 1000);
    Console.ReadLine();
    tmr.Dispose();           // End the timer
  }

  static void Tick (object data) {
    // This runs on a pooled thread
    Console.WriteLine (data);           // Writes "tick..."
  }
}
```

The .NET framework provides another timer class of the same name in the **System.Timers** namespace. This simply wraps **System.Threading.Timer**, providing additional convenience while

using the same thread pool – and the identical underlying engine. Here's a summary of its added features:

- A **Component** implementation, allowing it to be sited in the Visual Studio Designer

- An **Interval** property instead of a **Change** method

- An **Elapsed** *event* instead of a callback delegate

- An **Enabled** property to start and pause the timer (its default value being false)

- **Start** and **Stop** methods in case you're confused by **Enabled**

- an **AutoReset** flag for indicating a recurring event (default value true)

Here's an example:

```csharp
using System;
using System.Timers;    // Timers namespace rather than Threading

class SystemTimer {
  static void Main() {
    Timer tmr = new Timer();        // Doesn't require any args
    tmr.Interval = 500;
    tmr.Elapsed += tmr_Elapsed;     // Uses an event instead of a delegate
    tmr.Start();                    // Start the timer
    Console.ReadLine();
    tmr.Stop();                     // Pause the timer
    Console.ReadLine();
    tmr.Start();                    // Resume the timer
    Console.ReadLine();
    tmr.Dispose();                  // Permanently stop the timer
  }

  static void tmr_Elapsed (object sender, EventArgs e) {
    Console.WriteLine ("Tick");
  }
}
```

The .NET framework provides yet a third timer – in the **System.Windows.Forms** namespace. While similar to **System.Timers.Timer** in its interface, it's radically different in the functional sense. A Windows Forms timer does not use the thread pool, instead firing its "Tick" event always on the same thread that originally created the timer. Assuming this is the main thread – also responsible for instantiating all the forms and controls in the Windows Forms application – the timer's event handler is then able to interact with the forms and controls without violating thread-safety – or the impositions of apartment-threading. **Control.Invoke** is not required. The Windows timer is, in effect, a *single-threaded timer*.

> There's an equivalent single-threaded timer for WPF, called **DispatcherTimer**.

Windows Forms and WPF timers are intended for jobs that may involve updating the user interface and which execute quickly. Quick execution is important because the **Tick** event is called on the main thread – which if tied up, will make the user interface unresponsive.

Local Storage

Each thread gets a data store isolated from all other threads. This is useful for storing "out-of-band" data – that which supports the execution path's infrastructure, such as messaging, transaction or security tokens. Passing such data around via method parameters would be extremely clumsy and would alienate all but your own methods; storing such information in static fields would mean sharing it between all threads.

Thread.GetData reads from a thread's isolated data store; **Thread.SetData** writes to it. Both methods require a **LocalDataStoreSlot** object to identify the slot – this is just a wrapper for a string that names the slot – the same one can be used across all threads and they'll still get separate values. For example:

```
class ... {
  // The same LocalDataStoreSlot object can be used
  // across all threads.
  LocalDataStoreSlot secSlot = Thread.GetNamedDataSlot
                               ("securityLevel");

  // This property has a separate value on each thread.
  int SecurityLevel {
    get {
      object data = Thread.GetData (secSlot);
      return data == null ? 0 : (int) data; // null == uninitialized
    }
    set {
      Thread.SetData (secSlot, value);
    }
  }
}
...
```

Thread.FreeNamedDataSlot will release a given data slot across all threads – but only once all **LocalDataStoreSlot** objects of the same name have dropped out of scope and been garbage collected. This ensures threads don't get data slots pulled out from under their feet – as long as they keep a reference to the appropriate **LocalDataStoreSlot** object for as long as it's in use.

PART 4
ADVANCED TOPICS

Non-Blocking Synchronization

Earlier, we said that the need for synchronization arises even the simple case of assigning or incrementing a field. Although locking can always satisfy this need, a contended lock means that a thread must block, suffering the overhead and latency of being temporarily descheduled. The .NET framework's *non-blocking* synchronization constructs can perform simple operations without ever blocking, pausing, or waiting. These involve using instructions that are strictly atomic, and instructing the compiler to use "volatile" read and write semantics. At times these constructs can also be simpler to use than locks.

Atomicity and Interlocked

A statement is *atomic* if it executes as a single indivisible instruction. Strict atomicity precludes any possibility of preemption. In C#, a simple read or assignment on a field of 32 bits or less is atomic (assuming a 32-bit CPU). Operations on larger fields are non-atomic, as are statements that combine more than one read/write operation:

```
class Atomicity {
  static int x, y;
  static long z;

  static void Test() {
    long myLocal;
    x = 3;              // Atomic
    z = 3;              // Non-atomic (z is 64 bits)
    myLocal = z;        // Non-atomic (z is 64 bits)
    y += x;             // Non-atomic (read AND write operation)
    x++;                // Non-atomic (read AND write operation)
  }
}
```

Reading and writing 64-bit fields is non-atomic on 32-bit CPUs in the sense that two separate 32-bit memory locations are involved. If thread A reads a 64-bit value while thread B is updating it, thread A may end up with a bitwise combination of the old and new values.

Unary operators of the kind **x++** require first reading a variable, then processing it, then writing it back. Consider the following class:

```
class ThreadUnsafe {
  static int x = 1000;
  static void Go () { for (int i = 0; i < 100; i++) x--; }
}
```

You might expect that if 10 threads concurrently ran **Go, then x** would end up 0. However this is not guaranteed, because it's possible for one thread to preempt another in between retrieving **x**'s current value, decrementing it, and writing it back (resulting in an out-of-date value being written).

One way to solve to these problems is to wrap the non-atomic operations around a lock statement. Locking, in fact, simulates atomicity. The **Interlocked** class, however, provides a simpler and faster solution for simple atomic operations:

```
class Program {
  static long sum;

  static void Main() {                                          // sum

    // Simple increment/decrement operations:
    Interlocked.Increment (ref sum);                            // 1
    Interlocked.Decrement (ref sum);                            // 0

    // Add/subtract a value:
    Interlocked.Add (ref sum, 3);                               // 3

    // Read a 64-bit field:
    Console.WriteLine (Interlocked.Read (ref sum));             // 3

    // Write a 64-bit field while reading previous value:
    // (This prints "3" while updating sum to 10)
    Console.WriteLine (Interlocked.Exchange (ref sum, 10));     // 10

    // Update a field only if it matches a certain value (10):
    Interlocked.CompareExchange (ref sum, 123, 10);            // 123
  }
}
```

Using Interlocked is generally more efficient that obtaining a **lock**, because it can never block and suffer the overhead of its thread being temporarily descheduled.

Interlocked is also valid across multiple processes – in contrast to the **lock** statement, which is effective only across threads in the current process. An example of where this might be useful is in reading and writing into shared memory.

Memory Barriers and Volatility

Consider this class:

```
class Unsafe {
  static bool endIsNigh, repented;

  static void Main() {
    new Thread (Wait).Start();     // Start up the spinning waiter
    Thread.Sleep (1000);           // Give it a second to warm up!
    repented = true;
    endIsNigh = true;
    Console.WriteLine ("Going...");
  }

  static void Wait() {
    while (!endIsNigh);            // Spin until endIsNigh
    Console.WriteLine ("Gone, " + repented);
  }
}
```

Here's a question: can a significant delay separate "Going..." from "Gone" – in other words, is it possible for the **Wait** method to continue spinning in its **while** loop after the **endIsNigh** flag has been set to true? Furthermore, is it possible for the **Wait** method to write "Gone, false"?

The answer to both questions is, theoretically, yes, on a multi-processor machine, if the thread scheduler assigns the two threads different CPUs. The **repented** and **endIsNigh** fields can be cached in CPU registers to improve performance, with a potential delay before their updated values are

written back to memory. And when the CPU registers are written back to memory, it's not necessarily in the order they were originally updated.

This caching can be circumvented by using the static methods **Thread.VolatileRead** and **Thread.VolatileWrite** to read and write to the fields. **VolatileRead** means "read the latest value"; **VolatileWrite** means "write immediately to memory". The same functionality can be achieved more elegantly by declaring the field with the **volatile** modifier:

```
class ThreadSafe {
  // Always use volatile read/write semantics:
  volatile static bool endIsNigh, repented;
  ...
```

> If the **volatile** keyword is used in preference to the **VolatileRead** and **VolatileWrite** methods, one can think in the simplest terms, that is, "do not thread-cache this field!"

The same effect can be achieved by wrapping access to **repented** and **endIsNigh** in **lock** statements. This works because an (intended) side effect of locking is to create a memory barrier – a guarantee that the volatility of fields used within the **lock** statement will not extend outside the **lock** statement's scope. In other words, the fields will be fresh on entering the lock (*volatile read*) and be written to memory before exiting the lock (*volatile write*).

Using a **lock** statement would in fact be necessary if we needed to access the fields **end** and **endIsNigh** atomically, for instance, to run something like this:

```
lock (locker) { if (endIsNigh) repented = true; }
```

A **lock** may also be preferable where a field is used many times in a loop (assuming the lock is held for the duration of the loop). While a volatile read/write beats a lock in performance, it's unlikely that a thousand volatile read/write operations would beat one lock!

Volatility is relevant only to primitive integral (and unsafe pointer) types – other types are not cached in CPU registers and cannot be declared with the **volatile** keyword. Volatile read and write semantics are applied automatically when fields are accessed via the Interlocked class.

> If one has a policy always of accessing fields accessible by multiple threads in a **lock** statement, than **volatile** and **Interlocked** are unnecessary.

Wait and Pulse

Earlier we discussed Event Wait Handles – a simple signaling mechanism where a thread blocks until it receives notification from another.

A more powerful signaling construct is provided by the **Monitor** class, via two static methods – **Wait** and **Pulse**. The principle is that you write the signaling logic yourself using custom flags and fields (in conjunction with lock statements), then introduce **Wait** and **Pulse** commands to mitigate CPU spinning. This advantage of this low-level approach is that with just **Wait**, **Pulse** and the **lock** statement, you can achieve the functionality of AutoResetEvent, ManualResetEvent and Semaphore, as well as **WaitHandle'**s static methods **WaitAll** and **WaitAny**. Furthermore, **Wait** and **Pulse** can be amenable in situations where all of the Wait Handles are parsimoniously challenged.

A problem with **Wait** and **Pulse** is their poor documentation – particularly with regard their *reason-to-be*. And to make matters worse, the **Wait** and **Pulse** methods have a peculiar aversion to dabblers: if you call on them without a full understanding, they will know – and will delight in seeking you out and tormenting you! Fortunately, there is a simple pattern one can follow that provides a fail-safe solution in every case.

Wait and Pulse Defined

The purpose of **Wait** and **Pulse** is to provide a simple signaling mechanism: **Wait** blocks until it receives notification from another thread; **Pulse** provides that notification.

Wait must execute before **Pulse** in order for the signal to work. If **Pulse** executes first, its pulse is lost, and the late waiter must wait for a fresh pulse, or remain forever blocked. This differs from the behavior of an AutoResetEvent, where its **Set** method has a "latching" effect and so is effective if called before **WaitOne**.

One must specify a *synchronizing object* when calling **Wait** or **Pulse. If** two threads use the same object, then they are able to signal each other. The synchronizing object must be locked prior to calling Wait or Pulse.

For example, if **x** has this declaration:

```
class Test {
  // Any reference-type object will work as a synchronizing object
  object x = new object();
}
```

then the following code blocks upon entering **Monitor.Wait**:

```
lock (x) Monitor.Wait (x);
```

The following code (if executed later on another thread) releases the blocked thread:

```
lock (x) Monitor.Pulse (x);
```

Lock toggling

To make this work, **Monitor.Wait** temporarily releases, or *toggles* the underlying lock while waiting, so another thread (such as the one performing the **Pulse**) can obtain it. The **Wait** method can be thought of as expanding into the following pseudo-code:

```
Monitor.Exit (x);                    // Release the lock
wait for a pulse on x
Monitor.Enter (x);                   // Regain the lock
```

Hence a **Wait** can block twice: once in waiting for a pulse, and again in regaining the exclusive lock. This also means that **Pulse** by itself does not fully unblock a waiter: *only when the pulsing thread exits its lock statement* can the waiter actually proceed.

Wait's lock toggling is effective regardless of the lock nesting level. If **Wait** is called inside two nested **lock** statements:

```
lock (x)
  lock (x)
    Monitor.Wait (x);
```

then **Wait** logically expands into the following:

```
Monitor.Exit (x); Monitor.Exit (x);    // Exit twice to release the lock
wait for a pulse on x
Monitor.Enter (x); Monitor.Enter (x);  // Restore previous nesting level
```

Consistent with normal locking semantics, only the first call to **Monitor.Enter** affords a blocking opportunity.

Why the lock?

Why have **Wait** and **Pulse** been designed such that they will only work within a lock? The primary reason is so that **Wait** can be called conditionally – without compromising thread-safety. To take a simple example, suppose we want to **Wait** only if a boolean field called **available** is false. The following code is thread-safe:

```
lock (x) {
  if (!available) Monitor.Wait (x);
  available = false;
}
```

Several threads could run this concurrently, and none could preempt another in between checking the **available** field and calling **Monitor.Wait**. The two statements are effectively atomic. A corresponding notifier would be similarly thread-safe:

```
lock (x)
  if (!available) {
    available = true;
    Monitor.Pulse (x);
  }
```

Specifying a timeout

A timeout can be specified when calling **Wait**, either in milliseconds or as a **TimeSpan**. **Wait** then returns false if it gave up because of a timeout. The timeout applies only to the "waiting" phase (waiting for a pulse): a timed out **Wait** will still subsequently block in order to re-acquire the lock, no matter how long it takes. Here's an example:

```
lock (x) {
  if (!Monitor.Wait (x, TimeSpan.FromSeconds (10)))
    Console.WriteLine ("Couldn't wait!");
  Console.WriteLine ("But hey, I still have the lock on x!");
}
```

This rationale for this behavior is that in a well-designed **Wait/Pulse** application, the object on which one calls **Wait** and **Pulse** is locked just briefly. So re-acquiring the lock should be a near-instant operation.

Pulsing and acknowledgement

An important feature of **Monitor.Pulse** is that it executes asynchronously, meaning that it doesn't itself block or pause in any way. If another thread is waiting on the pulsed object, it's notified, otherwise the pulse has no effect and is silently ignored.

Pulse provides one-way communication: a pulsing thread signals a waiting thread. There is no intrinsic acknowledgment mechanism: **Pulse** does not return a value indicating whether or not its pulse was received. Furthermore, when a notifier pulses and releases its lock, there's no guarantee that an eligible waiter will kick into life immediately. There can be an arbitrary delay, at the discretion of the thread scheduler – during which time neither thread has the lock. This makes it difficult to know when a waiter has actually resumed, unless the waiter specifically acknowledges, for instance via a custom flag.

> If reliable acknowledgement is required, it must be explicitly coded, usually via a flag in conjunction with another, reciprocal, **Pulse** and **Wait**.

Relying on timely action from a waiter with no custom acknowledgement mechanism counts as "messing" with **Pulse** and **Wait**. You'll lose!

Waiting queues and PulseAll

More than one thread can simultaneously **Wait** upon the same object – in which case a "waiting queue" forms behind the synchronizing object (this is distinct from the "ready queue" used for granting access to a lock). Each **Pulse** then releases a single thread at the head of the waiting-queue, so it can enter the ready-queue and re-acquire the lock. Think of it like an automatic car park: you queue first at the pay station to validate your ticket (the waiting queue); you queue again at the barrier gate to be let out (the ready queue).
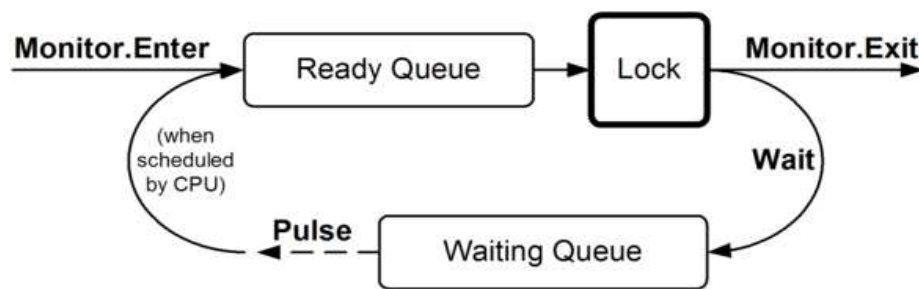
Figure 2: Waiting Queue vs. Ready Queue

The order inherent in the queue structure, however, is often unimportant in **Wait/Pulse** applications, and in these cases it can be easier to imagine a "pool" of waiting threads. Each pulse, then, releases one waiting thread from the pool.

Monitor also provides a **PulseAll** method that releases the entire queue, or pool, of waiting threads in a one-fell swoop. The pulsed threads won't all start executing exactly at the same time, however, but rather in an orderly sequence, as each of their **Wait** statements tries to re-acquire the same lock. In effect, **PulseAll** moves threads from the waiting-queue to the ready-queue, so they can resume in an orderly fashion.

How to use Pulse and Wait

Here's how we start. Imagine there are two rules:

- the only synchronization construct available is the lock statement, aka **Monitor.Enter** and **Monitor.Exit**

- there are no restrictions on spinning the CPU!

With those rules in mind, let's take a simple example: a worker thread that pauses until it receives notification from the main thread:

```csharp
class SimpleWaitPulse {
  bool go;
  object locker = new object();

  void Work() {
    Console.Write ("Waiting... ");
    lock (locker) {                         // Let's spin!
      while (!go) {
        // Release the lock so other threads can change the go flag
        Monitor.Exit (locker);
        // Regain the lock so we can re-test go in the while loop
        Monitor.Enter (locker);
      }
    }
    Console.WriteLine ("Notified!");
  }

  void Notify()// called from another thread
  {
    lock (locker) {
      Console.Write ("Notifying... ");
      go = true;
    }
  }
```

```
    }
```

Here's a main method to set things in motion:

```
static void Main() {
  SimpleWaitPulse test = new SimpleWaitPulse();

  // Run the Work method on its own thread
  new Thread (test.Work).Start();           // "Waiting..."

  // Pause for a second, then notify the worker via our main thread:
  Thread.Sleep (1000);
  test.Notify();                    // "Notifying... Notified!"
}
```

The **Work** method is where we spin – extravagantly consuming CPU time by looping constantly until the **go** flag is true! In this loop we have to keep toggling the lock – releasing and re-acquiring it via **Monitor**'s **Exit** and **Enter** methods – so that another thread running the **Notify** method can itself get the lock and modify the **go** flag. The shared **go** field must always be accessed from within a lock to avoid volatility issues (remember that all other synchronization constructs, such as the volatile keyword, are out of bounds in this stage of the design!)

The next step is to run this and test that it actually works. Here's the output from the test **Main** method:

```
Waiting... (pause) Notifying... Notified!
```

Now we can introduce **Wait** and **Pulse**. We do this by:

- replacing lock toggling (**Monitor.Exit** followed by **Monitor.Enter**) with **Monitor.Wait**

- inserting a call to **Monitor.Pulse** when a blocking condition is changed (i.e. the **go** flag is modified).

Here's the updated class, with the **Console** statements omitted for brevity:

```
class SimpleWaitPulse {
  bool go;
  object locker = new object();

  void Work() {
    lock (locker)
      while (!go) Monitor.Wait (locker);
  }

  void Notify() {
    lock (locker) {
      go = true;
      Monitor.Pulse (locker);
    }
  }
}
```

The class behaves as it did before, but with the spinning eliminated. The **Wait** command implicitly performs the code we removed – **Monitor.Exit** followed by **Monitor.Exit**, but with one extra step in the middle: while the lock is released, it waits for another thread to call **Pulse**. The Notifier method does just this, after setting the **go** flag true. The job is done.

Pulse and Wait Generalized

Let's now expand the pattern. In the previous example, our blocking condition involved just one boolean field – the **go** flag. We could, in another scenario, require an additional flag set by the waiting thread to signal that's it's ready or complete. If we extrapolate by supposing there could be any number of fields involved in any number of blocking conditions, the program can be generalized into the following pseudo-code (in its spinning form):

```
class X {
   Blocking Fields:  one or more objects involved in blocking condition(s), eg
     bool go;   bool ready;   int semaphoreCount;   Queue <Task> consumerQ...

   object locker = new object();      // protects all the above fields!

   ... SomeMethod {
     ... whenever I want to BLOCK based on the blocking fields:
     lock (locker)
       while (! blocking fields to my liking ) {
         // Give other threads a chance to change blocking fields!
         Monitor.Exit (locker);
         Monitor.Enter (locker);
       }

     ... whenever I want to ALTER one or more of the blocking fields:
     lock (locker) { alter blocking field(s) }
   }
}
```

We then apply Pulse and Wait as we did before:

- In the waiting loops, lock toggling is replaced with Monitor.Wait

- Whenever a blocking condition is changed, **Pulse** is called before releasing the lock.

Here's the updated pseudo-code:

Wait/Pulse Boilerplate #1: Basic Wait/Pulse Usage

```
class X {
  < Blocking Fields ... >
  object locker = new object();

  ... SomeMethod {
    ...
    ... whenever I want to BLOCK based on the blocking fields:
    lock (locker)
      while (! blocking fields to my liking )
        Monitor.Wait (locker);

    ... whenever I want to ALTER one or more of the blocking fields:
    lock (locker) {
      alter blocking field(s)
      Monitor.Pulse (locker);
    }
  }
}
```

This provides a robust pattern for using **Wait** and **Puls**e. Here are the key features to this pattern:

- Blocking conditions are implemented using custom fields (capable of functioning without **Wait** and **Pulse**, albeit with spinning)

- **Wait** is always called within a **while** loop that checks its blocking condition (itself within a **lock** statement)

- A single synchronization object (in the example above, **locker**) is used for all **Wait**s and **Pulse**s, and to protect access to all objects involved in all blocking conditions

- Locks are held only briefly

Most importantly, with this pattern, pulsing does not force a waiter to continue. Rather, it notifies a waiter that something has changed, advising it to re-check its blocking condition. The waiter then determines whether or not it should proceed (via another iteration of its **while** loop) – and not the pulser. The benefit of this approach is that it allows for sophisticated blocking conditions, without sophisticated synchronization logic.

Another benefit of this pattern is immunity to the effects of a missed pulse. A missed pulse happens when **Pulse** is called before **Wait** – perhaps due to a race between the notifier and waiter. But because in this pattern a pulse means "re-check your blocking condition" (and not "continue"), an early pulse can safely be ignored since the blocking condition is always checked before calling Wait, thanks to the **while** statement.

With this design, one can define multiple blocking fields, and have them partake in multiple blocking conditions, and yet still use a single synchronization object throughout (in our example, **locker**). This is usually better than having separate synchronization objects on which to **lock**, **Pulse** and **Wait**, in that one avoids the possibility of deadlock. Furthermore, with a single locking object, all blocking fields are read and written to as a unit, avoiding subtle atomicity errors. It's a good idea, however, not to use the synchronization object for purposes outside of the necessary scope (this can be assisted by declaring **private** the synchronization object, as well as all blocking fields).

Producer/Consumer Queue

A simple **Wait**/**Pulse** application is a producer-consumer queue – the structure we wrote earlier using an **AutoResetEvent**. A producer enqueues tasks (typically on the main thread), while one or more consumers running on worker threads pick off and execute the tasks one by one.

In this example, we'll use a string to represent a task. Our task queue then looks like this:

```
Queue<string> taskQ = new Queue<string>();
```

Because the queue will be used on multiple threads, we must wrap all statements that read or write to the queue in a **lock**. Here's how we enqueue a task:

```
lock (locker) {
  taskQ.Enqueue ("my task");
  Monitor.PulseAll (locker);   // We're altering a blocking condition
}
```

Because we're modifying a potential blocking condition, we must pulse. We call **PulseAll** rather than **Pulse** because we're going to allow for multiple consumers. More than one thread may be waiting.

We want the workers to block while there's nothing to do, in other words, when there are no items on the queue. Hence our blocking condition is **taskQ.Count==0**. Here's a **Wait** statement that performs exactly this:

```
lock (locker)
  while (taskQ.Count == 0) Monitor.Wait (locker);
```

The next step is for the worker to dequeue the task and execute it:

```
lock (locker)
  while (taskQ.Count == 0) Monitor.Wait (locker);

string task;
lock (locker)
  task = taskQ.Dequeue();
```

This logic, however, is not thread-safe: we've basing a decision to dequeue upon stale information – obtained in a prior **lock** statement. Consider what would happen if we started two consumer threads concurrently, with a single item already on the queue. It's possible that neither thread would enter the **while** loop to block – both seeing a single item on the queue. They'd both then attempt to dequeue the same item, throwing an exception in the second instance! To fix this, we simply hold the **lock** a bit longer – until we've finished interacting with the queue:

```
string task;
lock (locker) {
  while (taskQ.Count == 0) Monitor.Wait (locker);
  task = taskQ.Dequeue();
}
```

(We don't need to call **Pulse** after dequeuing, as no consumer can ever unblock by there being fewer items on the queue).

Once the task is dequeued, there's no further requirement to keep the lock. Releasing it at this point allows the consumer to perform a possibly time-consuming task without unnecessary blocking other threads.

Here's the complete program. As with the AutoResetEvent version, we enqueue a null task to signal a consumer to exit (after finishing any outstanding tasks). Because we're supporting multiple consumers, we must enqueue one null task per consumer to completely shut down the queue:

Wait/Pulse Boilerplate #2: Producer/Consumer Queue

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

public class TaskQueue : IDisposable {
  object locker = new object();
  Thread[] workers;
  Queue<string> taskQ = new Queue<string>();

  public TaskQueue (int workerCount) {
    workers = new Thread [workerCount];

    // Create and start a separate thread for each worker
    for (int i = 0; i < workerCount; i++)
      (workers [i] = new Thread (Consume)).Start();
  }

  public void Dispose() {
    // Enqueue one null task per worker to make each exit.
    foreach (Thread worker in workers) EnqueueTask (null);
    foreach (Thread worker in workers) worker.Join();
  }

  public void EnqueueTask (string task) {
    lock (locker) {
      taskQ.Enqueue (task);
      Monitor.PulseAll (locker);
    }
  }

  void Consume() {
    while (true) {
      string task;
      lock (locker) {
        while (taskQ.Count == 0) Monitor.Wait (locker);
        task = taskQ.Dequeue();
      }
      if (task == null) return;          // This signals our exit
      Console.Write (task);
      Thread.Sleep (1000);               // Simulate time-consuming task
    }
  }
}
```

Here's a **Main** method that starts a task queue, specifying two concurrent consumer threads, and then enqueues ten tasks to be shared amongst the two consumers:

```
static void Main() {
  using (TaskQueue q = new TaskQueue (2)) {
    for (int i = 0; i < 10; i++)
      q.EnqueueTask (" Task" + i);

    Console.WriteLine ("Enqueued 10 tasks");
    Console.WriteLine ("Waiting for tasks to complete...");
  }
  // Exiting the using statement runs TaskQueue's Dispose method,
  // which shuts down the consumers, after all outstanding tasks
  // have completed.
  Console.WriteLine ("\r\nAll tasks done!");
}
```

```
Enqueued 10 tasks
Waiting for tasks to complete...
 Task1 Task0 (pause…) Task2 Task3 (pause…) Task4 Task5 (pause…)
 Task6 Task7 (pause…) Task8 Task9 (pause…)
All tasks done!
```

Consistent with our design pattern, if we remove **PulseAll** and replace **Wait** with lock toggling, we'll get the same output.

Pulse Economy

Let's revisit the producer enqueuing a task:

```
lock (locker) {
  taskQ.Enqueue (task);
  Monitor.PulseAll (locker);
}
```

Strictly speaking, we could economize by pulsing only when there's a possibility of a freeing a blocked worker:

```
lock (locker) {
  taskQ.Enqueue (task);
  if (taskQ.Count <= workers.Length) Monitor.PulseAll (locker);
}
```

We'd be saving very little, though, since pulsing typically takes under a microsecond, and incurs no overhead on busy workers – since they ignore it anyway! It's a good policy with multi-threaded code to cull any unnecessary logic: an intermittent bug due to a silly mistake is a heavy price to pay for a one-microsecond saving! To demonstrate, this is all it would take to introduce an intermittent "stuck worker" bug that would most likely evade initial testing (spot the difference):

```
lock (locker) {
  taskQ.Enqueue (task);
  if (taskQ.Count < workers.Length) Monitor.PulseAll (locker);
}
```

Pulsing unconditionally protects us from this type of bug.

> If in doubt, **Pulse**. Rarely can you go wrong by pulsing, within this design pattern.

Pulse or PulseAll?

This example comes with further pulse economy potential. After enqueuing a task, we could call **Pulse** instead of **PulseAll** and nothing would break.

Let's recap the difference: with **Pulse**, a maximum of one thread can awake (and re-check its while-loop blocking condition); with **PulseAll**, all waiting threads will awake (and re-check their blocking conditions). If we're enqueuing a single task, only one worker can handle it, so we need only wake up one worker with a single **Pulse**. It's rather like having a class of sleeping children – if there's just one ice-cream there's no point in waking them all to queue for it!

In our example we start only two consumer threads, so we would have little to gain. But if we started ten consumers, we might benefit slightly in choosing **Pulse** over **PulseAll**. It would mean, though, that if we enqueued multiple tasks, we would need to **Pulse** multiple times. This can be done within a single **lock** statement, as follows:

```
lock (locker) {
  taskQ.Enqueue ("task 1");
  taskQ.Enqueue ("task 2");
  Monitor.Pulse (locker);    // "Signal up to two
  Monitor.Pulse (locker);    //  waiting threads."
}
```

The price of one **Pulse** too few is a stuck worker. This will usually manifest as an intermittent bug, because it will crop up only when a consumer is in a **Waiting** state. Hence one could extend the previous maxim "if in doubt, **Pulse**", to "if in doubt, **PulseAll**!"

A possible exception to the rule might arise if evaluating the blocking condition was unusually time-consuming.

Using Wait Timeouts

Sometimes it may be unreasonable or impossible to **Pulse** whenever an unblocking condition arises. An example might be if a blocking condition involves calling a method that derives information from periodically querying a database. If latency is not an issue, the solution is simple: one can specify a timeout when calling **Wait**, as follows:

```
lock (locker) {
  while ( blocking condition )
    Monitor.Wait (locker, timeout);
```

This forces the blocking condition to be re-checked, at a minimum, at a regular interval specified by the timeout, as well as immediately upon receiving a pulse. The simpler the blocking condition, the smaller the timeout can be without causing inefficiency.

The same system works equally well if the pulse is absent due to a bug in the program! It can be worth adding a timeout to all **Wait** commands in programs where synchronization is particularly complex – as an ultimate backup for obscure pulsing errors. It also provides a degree of bug-immunity if the program is modified later by someone not on the **Pulse**!

Races and Acknowledgement

Let's say we want a signal a worker five times in a row:

```
class Race {
  static object locker = new object();
  static bool go;

  static void Main() {
    new Thread (SaySomething).Start();

    for (int i = 0; i < 5; i++) {
      lock (locker) { go = true; Monitor.Pulse (locker); }
    }
  }

  static void SaySomething() {
    for (int i = 0; i < 5; i++) {
      lock (locker) {
        while (!go) Monitor.Wait (locker);
        go = false;
      }
      Console.WriteLine ("Wassup?");
    }
  }
}
```

Expected Output:

```
Wassup?
Wassup?
Wassup?
Wassup?
Wassup?
```

Actual Output:

```
Wassup?
 (hangs)
```

This program is flawed: the **for** loop in the main thread can free-wheel right through its five iterations any time the worker doesn't hold the lock. Possibly before the worker even starts! The Producer/Consumer example didn't suffer from this problem because if the main thread got ahead of the worker, each request would simply queue up. But in this case, we need the main thread to block at each iteration if the worker's still busy with a previous task.

A simple solution is for the main thread to wait after each cycle until the **go** flag is cleared by the worker. This, then, requires that the worker call **Pulse** after clearing the **go** flag:

```
class Acknowledged {
 static object locker = new object();
  static bool go;

  static void Main() {
    new Thread (SaySomething).Start();

    for (int i = 0; i < 5; i++) {
      lock (locker) { go = true; Monitor.Pulse (locker); }
      lock (locker) { while (go) Monitor.Wait (locker); }
    }
  }

  static void SaySomething() {
    for (int i = 0; i < 5; i++) {
      lock (locker) {
        while (!go) Monitor.Wait (locker);
        go = false; Monitor.Pulse (locker);    // Worker must Pulse
      }
      Console.WriteLine ("Wassup?");
    }
  }
}
```

```
Wassup? (repeated five times)
```

An important feature of such a program is that the worker releases its lock before performing its potentially time-consuming job (this would happen in place of where we're calling **Console.WriteLine**). This ensures the instigator is not unduly blocked while the worker performs the task for which it has been signaled (and is blocked only if the worker is busy with a previous task).

In this example, only one thread (the main thread) signals the worker to perform a task. If multiple threads were to signal the worker – using our **Main** method's logic – we would come unstuck. Two signaling threads could each execute the following line of code in sequence:

```
lock (locker) { go = true; Monitor.Pulse (locker); }
```

resulting in the second signal being lost if the worker didn't happen to have finish processing the first. We can make our design robust in this scenario by using a pair of flags – a "ready" flag as well as a "go" flag. The "ready" flag indicates that the worker is able to accept a fresh task; the "go" flag is an instruction to proceed, as before. This is analogous to a previous example that performed the same thing using two **AutoResetEvent**s, except more extensible. Here's the pattern, refactored with instance fields:

Wait/Pulse Boilerplate #3: Two-way Signaling

```csharp
public class Acknowledged {
  object locker = new object();
  bool ready;
  bool go;

  public void NotifyWhenReady() {
    lock (locker) {
      // Wait if the worker's already busy with a previous job
      while (!ready) Monitor.Wait (locker);
      ready = false;
      go = true;
      Monitor.PulseAll (locker);
    }
  }

  public void AcknowledgedWait() {
    // Indicate that we're ready to process a request
    lock (locker) { ready = true; Monitor.Pulse (locker); }

    lock (locker) {
      while (!go) Monitor.Wait (locker);      // Wait for a "go" signal
      go = false; Monitor.PulseAll (locker);  // Acknowledge signal
    }

    Console.WriteLine ("Wassup?");            // Perform task
  }
}
```

To demonstrate, we'll start two concurrent threads, each that will notify the worker five times. Meanwhile, the main thread will wait for ten notifications:

```csharp
public class Test {
  static Acknowledged a = new Acknowledged();

  static void Main() {
    new Thread (Notify5).Start();     // Run two concurrent
    new Thread (Notify5).Start();     // notifiers...
    Wait10();                         // ... and one waiter.
  }

  static void Notify5() {
    for (int i = 0; i < 5; i++)
      a.NotifyWhenReady();
  }

  static void Wait10() {
    for (int i = 0; i < 10; i++)
      a.AcknowledgedWait();
  }
}
```

```
Wassup?
Wassup?
Wassup?
 (repeated ten times)
```

In the **Notify** method, the **ready** flag is cleared before exiting the **lock** statement. This is vitally important: it prevents two notifiers signaling sequentially without re-checking the flag. For the sake of simplicity, we also set the **go** flag and call **PulseAll** in the same **lock** statement – although we could just as well put this pair of statements in a separate **lock** and nothing would break.

Simulating Wait Handles

You might have noticed a pattern in the previous example: both waiting loops have the following structure:

```
lock (locker) {
  while (!flag) Monitor.Wait (locker);
  flag = false;
  ...
}
```

where *flag* is set to **true** in another thread. This is, in effect, mimicking an AutoResetEvent. If we omitted *flag*=**false**, we'd then have a ManualResetEvent. Using an integer field, **Pulse** and **Wait** can also be used to mimic a Semaphore. In fact the only Wait Handle we can't mimic with **Pulse** and **Wait** is a Mutex, since this functionality is provided by the **lock** statement.

Simulating the static methods that work across multiple Wait Handles is in most cases easy. The equivalent of calling WaitAll across multiple **EventWaitHandle**s is nothing more than a blocking condition that incorporates all the flags used in place of the Wait Handles:

```
lock (locker) {
  while (!flag1 && !flag2 && !flag3...) Monitor.Wait (locker);
```

This can be particularly useful given that **WaitAll** is in most cases unusable due to COM legacy issues. Simulating **WaitAny** is simply a matter of replacing the **&&** operator with the || operator.

SignalAndWait is trickier. Recall that this method signals one handle while waiting on another in an atomic operation. We have a situation analogous to a distributed database transaction – we need a two-phase commit! Assuming we wanted to signal **flagA** while waiting on **flagB**, we'd have to divide each flag into two, resulting in code that might look something like this:

```
lock (locker) {
  flagAphase1 = true;
  Monitor.Pulse (locker);
  while (!flagBphase1) Monitor.Wait (locker);

  flagAphase2 = true;
  Monitor.Pulse (locker);
  while (!flagBphase2) Monitor.Wait (locker);
}
```

perhaps with additional "rollback" logic to retract **flagAphase1** if the first **Wait** statement threw an exception as a result of being interrupted or aborted. This is one situation where Wait Handles are way easier! True atomic signal-and-waiting, however, is actually an unusual requirement.

Wait Rendezvous

Just as WaitHandle.SignalAndWait can be used to rendezvous a pair of threads, so can **Wait** and **Pulse**. In the following example, one could say we simulate two **ManualResetEvent**s (in other words, we define two boolean flags!) and then perform reciprocal signal-and-waiting by setting one flag while waiting for the other. In this case we don't need true atomicity in signal-and-waiting, so can avoid the need for a "two-phase commit". As long as we set our flag true and Wait in the same **lock** statement, the rendezvous will work:

```
class Rendezvous {
  static object locker = new object();
  static bool signal1, signal2;

  static void Main() {
    // Get each thread to sleep a random amount of time.
    Random r = new Random();
    new Thread (Mate).Start (r.Next (10000));
    Thread.Sleep (r.Next (10000));

    lock (locker) {
      signal1 = true;
      Monitor.Pulse (locker);
      while (!signal2) Monitor.Wait (locker);
    }
    Console.Write ("Mate! ");
  }

  // This is called via a ParameterizedThreadStart
  static void Mate (object delay) {
    Thread.Sleep ((int) delay);
    lock (locker) {
      signal2 = true;
      Monitor.Pulse (locker);
      while (!signal1) Monitor.Wait (locker);
    }
    Console.Write ("Mate! ");
  }
}
```

```
Mate! Mate! (almost in unison)
```

Wait and Pulse vs. Wait Handles

Because **Wait** and **Pulse** are the most flexible of the synchronization constructs, they can be used in almost any situation. Wait Handles, however, have two advantages:

- they have the capability of working across multiple processes

- they are simpler to understand, and harder to break

Additionally, Wait Handles are more interoperable in the sense that they can be passed around via method arguments. In *thread pooling*, this technique is usefully employed.

In terms of performance, **Wait** and **Pulse** have a slight edge, if one follows the suggested design pattern for waiting, that is:

```
lock (locker)
  while ( blocking condition ) Monitor.Wait (locker);
```

and the blocking condition happens to false from the outset. The only overhead then incurred is that of taking out the lock (tens of nanoseconds) versus the few microseconds it would take to call **WaitHandle.WaitOne**. Of course, this assumes the lock is uncontended; even the briefest lock contention would be more than enough to even things out; frequent lock contention would make a Wait Handle faster!

> Given the potential for variation through different CPUs, operating systems, CLR versions, and program logic; and that in any case a few microseconds is unlikely to be of any consequence before a **Wait** statement, performance may be a dubious reason to choose **Wait** and **Pulse** over Wait Handles, or vice versa.

A sensible guideline is to use a Wait Handle where a particular construct lends itself naturally to the job, otherwise use **Wait** and **Pulse**.

Suspend and Resume

A thread can be explicitly suspended and resumed via the methods **Thread.Suspend** and **Thread.Resume**. This mechanism is completely separate to that of blocking discussed previously. Both systems are independent and operate in parallel.

A thread can suspend itself or another thread. Calling **Suspend** results in the thread briefly entering the **SuspendRequested** state, then upon reaching a point safe for garbage collection, it enters the **Suspended** state. From there, it can be resumed only via another thread that calls its **Resume** method. **Resume** will work only on a suspended thread, not a blocked thread.

From .NET 2.0, **Suspend** and **Resume** have been deprecated, their use discouraged because of the danger inherent in arbitrarily suspending another thread. If a thread holding a lock on a critical resource is suspended, the whole application (or computer) can deadlock. This is far more dangerous than calling Abort – which would result in any such locks being released – at least theoretically – by virtue of code in **finally** blocks.

It is, however, safe to call **Suspend** on the current thread – and in doing so one can implement a simple synchronization mechanism – with a worker thread in a loop – performing a task, calling **Suspend** on itself, then waiting to be resumed ("woken up") by the main thread when another task is ready. The difficulty, though, is in testing whether or not the worker is suspended. Consider the following code:

```
worker.NextTask = "MowTheLawn";
if ((worker.ThreadState & ThreadState.Suspended) > 0)
  worker.Resume;
else
  // We cannot call Resume as the thread's already running.
  // Signal the worker with a flag instead:
  worker.AnotherTaskAwaits = true;
```

This is horribly thread-unsafe – the code could be preempted at any point in these five lines – during which the worker could march on in and change its state. While it can be worked around, the solution is more complex than the alternative – using a synchronization construct such as an AutoResetEvent or Monitor.Wait. This makes **Suspend** and **Resume** useless on all counts.

> The deprecated **Suspend** and **Resume** methods have two modes – dangerous and useless!

71

Aborting Threads

A thread can be ended forcibly via the **Abort** method:

```
class Abort {
  static void Main() {
    Thread t = new Thread (delegate() {while(true);});   // Spin forever
    t.Start();
    Thread.Sleep (1000);       // Let it run for a second...
    t.Abort();                 // then abort it.
  }
}
```

The thread upon being aborted immediately enters the **AbortRequested** state. If it then terminates as expected, it goes into the **Stopped** state. The caller can wait for this to happen by calling Join:

```
class Abort {
  static void Main() {
    Thread t = new Thread (delegate() { while (true); });
    Console.WriteLine (t.ThreadState);     // Unstarted

    t.Start();
    Thread.Sleep (1000);
    Console.WriteLine (t.ThreadState);     // Running

    t.Abort();
    Console.WriteLine (t.ThreadState);     // AbortRequested

    t.Join();
    Console.WriteLine (t.ThreadState);     // Stopped
  }
}
```

Abort causes a **ThreadAbortException** to be thrown on the target thread, in most cases right where the thread's executing at the time. The thread being aborted can choose to handle the exception, but the exception then gets automatically re-thrown at the end of the **catch** block (to help ensure the thread, indeed, ends as expected). It is, however, possible to prevent the automatic re-throw by calling **Thread.ResetAbort** within the **catch** block. Then thread then re-enters the **Running** state (from which it can potentially be a**borted** again). In the following example, the worker thread comes back from the dead each time an **Abort** is attempted:

```
class Terminator {
  static void Main() {
    Thread t = new Thread (Work);
    t.Start();
    Thread.Sleep (1000); t.Abort();
    Thread.Sleep (1000); t.Abort();
    Thread.Sleep (1000); t.Abort();
  }

  static void Work() {
    while (true) {
      try { while (true); }
      catch (ThreadAbortException) { Thread.ResetAbort(); }
      Console.WriteLine ("I will not die!");
    }
  }
}
```

ThreadAbortException is treated specially by the runtime, in that it doesn't cause the whole application to terminate if unhandled, unlike all other types of exception.

Abort will work on a thread in almost any state – running, blocked, suspended, or stopped. However if a suspended thread is aborted, a **ThreadStateException** is thrown – this time on the calling thread – and the abortion doesn't kick off until the thread is subsequently resumed. Here's how to abort a suspended thread:

```
try { suspendedThread.Abort(); }
catch (ThreadStateException) { suspendedThread.Resume(); }
// Now the suspendedThread will abort.
```

Complications with Thread.Abort

Assuming an aborted thread doesn't call **ResetAbort**, one might expect it to terminate fairly quickly. But as it happens, with a good lawyer the thread may remain on death row for quite some time! Here are a few factors that may keep it lingering in the **AbortRequested** state:

- Static class constructors are never aborted part-way through (so as not to potentially poison the class for the remaining life of the application domain)

- All catch/finally blocks are honored, and never aborted mid-stream

- If the thread is executing unmanaged code when aborted, execution continues until the next managed code statement is reached

The last factor can be particularly troublesome, in that the .NET framework itself often calls unmanaged code, sometimes remaining there for long periods of time. An example might be when using a networking or database class. If the network resource or database server dies or is slow to respond, it's possible that execution could remain entirely within unmanaged code, for perhaps minutes, depending on the implementation of the class. In these cases, one certainly wouldn't want to Join the aborted thread – at least not without a timeout!

Aborting pure .NET code is less problematic, as long as **try/finally** blocks or **using** statements are incorporated to ensure proper cleanup takes place should a **ThreadAbortException** be thrown. However, even then, one can still be vulnerable to nasty surprises. For example, consider the following:

```
using (StreamWriter w = File.CreateText ("myfile.txt"))
  w.Write ("Abort-Safe?");
```

C#'s **using** statement is simply a syntactic shortcut, which in this case expands to the following:

```
StreamWriter w;
w = File.CreateText ("myfile.txt");
try    { w.Write ("Abort-Safe"); }
finally { w.Dispose();          }
```

It's possible for an **Abort** to fire after the **StreamWriter is** created, but before the **try** block begins. In fact, by digging into the IL, one can see that it's also possible for it to fire in between the **StreamWriter** being created and assigned to **w**:

```
IL_0001:  ldstr       "myfile.txt"
IL_0006:  call        class [mscorlib]System.IO.StreamWriter
                      [mscorlib]System.IO.File::CreateText(string)
IL_000b:  stloc.0
.try
{
```

```
    ...
```

Either way, the **Dispose** method in the **finally** block is circumvented, resulting in an abandoned open file handle – preventing any subsequent attempts to create **myfile.txt** until the application domain ends.

In reality, the situation in this example is worse still, because an **Abort** would most likely take place within the implementation of **File.CreateText**. This is referred to as opaque code – that which we don't have the source. Fortunately, .NET code is never truly opaque: we can again wheel in ILDASM, or better still, Lutz Roeder's Reflector – and looking into the framework's assemblies, see that it calls **StreamWriter'**s constructor, which has the following logic:

```
public StreamWriter (string path, bool append, ...)
{
  ...
  ...
  Stream stream1 = StreamWriter.CreateFile (path, append);
  this.Init (stream1, ...);
}
```

Nowhere in this constructor is there a **try/catch** block, meaning that if the **Abort** fires anywhere within the (non-trivial) **Init** method, the newly created stream will be abandoned, with no way of closing the underlying file handle.

Because disassembling every required CLR call is obviously impractical, this raises the question on how one should go about writing an abort-friendly method. The most common workaround is not to abort another thread at all – but rather add a custom boolean field to the worker's class, signaling that it should abort. The worker checks the flag periodically, exiting gracefully if true. Ironically, the most graceful exit for the worker is by calling **Abort on its own thread** – although explicitly throwing an exception also works well. This ensures the thread's backed right out, while executing any **catch/finally** blocks – rather like calling **Abort** from another thread, except the exception is thrown only from designated places:

```
class ProLife {
  public static void Main() {
    RulyWorker w = new RulyWorker();
    Thread t = new Thread (w.Work);
    t.Start();
    Thread.Sleep (500);
    w.Abort();
  }
```

```csharp
public class RulyWorker {
  // The volatile keyword ensures abort is not cached by a thread
  volatile bool abort;

  public void Abort() { abort = true; }

  public void Work() {
    while (true) {
      CheckAbort();
      // Do stuff...
      try     { OtherMethod(); }
      finally { /* any required cleanup */ }
    }
  }

  void OtherMethod() {
    // Do stuff...
    CheckAbort();
  }

  void CheckAbort() { if (abort) Thread.CurrentThread.Abort(); }
}
```

> Calling **Abort** on one's own thread is one circumstance in which **Abort** is totally safe. Another is when you can be certain the thread you're aborting is in a particular section of code, usually by virtue of a synchronization mechanism such as a Wait Handle or Monitor.Wait. A third instance in which calling **Abort** is safe is when you subsequently tear down the thread's application domain or process.

Ending Application Domains

Another way to implement an abort-friendly worker is by having its thread run in its own application domain. After calling **Abort**, one simply tears down the application domain, thereby releasing any resources that were improperly disposed.

Strictly speaking, the first step – aborting the thread – is unnecessary, because when an application domain is unloaded, all threads executing code in that domain are automatically aborted. However, the disadvantage of relying on this behavior is that if the aborted threads don't exit in a timely fashion (perhaps due to code in **finally** blocks, or for other reasons discussed previously) the application domain will not unload, and a **CannotUnloadAppDomainException** will be thrown on the caller. For this reason, it's better to explicitly abort the worker thread, then call Join with some timeout (over which you have control) before unloading the application domain.

In the following example, the worker enters an infinite loop, creating and closing a file using the abort-unsafe **File.CreateText** method. The main thread then repeatedly starts and aborts workers. It usually fails within one or two iterations, with **CreateText** getting aborted part way through its internal implementation, leaving behind an abandoned open file handle:

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

class Program {
  static void Main() {
    while (true) {
      Thread t = new Thread (Work);
      t.Start();
      Thread.Sleep (100);
      t.Abort();
      Console.WriteLine ("Aborted");
    }
  }

  static void Work() {
    while (true)
      using (StreamWriter w = File.CreateText ("myfile.txt")) { }
  }
}
```

```
Aborted
Aborted
IOException: The process cannot access the file 'myfile.txt' because it
is being used by another process.
```

Here's the same program modified so the worker thread runs in its own application domain, which is unloaded after the thread is aborted. It runs perpetually without error, because unloading the application domain releases the abandoned file handle:

```csharp
class Program {
  static void Main (string [] args) {
    while (true) {
      AppDomain ad = AppDomain.CreateDomain ("worker");
      Thread t = new Thread (delegate() { ad.DoCallBack (Work); });
      t.Start();
      Thread.Sleep (100);
      t.Abort();
      if (!t.Join (2000)) {
        // Thread won't end – here's where we could take further action,
        // if, indeed, there was anything we could do. Fortunately in
        // this case, we can expect the thread *always* to end.
      }
      AppDomain.Unload (ad);             // Tear down the polluted domain!
      Console.WriteLine ("Aborted");
    }
  }

  static void Work() {
    while (true)
      using (StreamWriter w = File.CreateText ("myfile.txt")) { }
  }
}
```

```
Aborted
Aborted
Aborted
Aborted
…
…
```

Creating and destroying an application domain is classed as relatively time-consuming in the world of threading activities (taking a few milliseconds) so it's something conducive to being done irregularly rather than in a loop! Also, the separation introduced by the application domain introduces another element that can be either of benefit or detriment, depending on what the multi-threaded program is setting out to achieve. In a unit-testing context, for instance, running threads on separate application domains can be of great benefit.

Ending Processes

Another way in which a thread can end is when the parent process terminates. One example of this is when a worker thread's IsBackground property is set to true, and the main thread finishes while the worker is still running. The background thread is unable to keep the application alive, and so the process terminates, taking the background thread with it.

When a thread terminates because of its parent process, it stops dead, and no **finally** blocks are executed.

The same situation arises when a user terminates an unresponsive application via the Windows Task Manager, or a process is killed programmatically via **Process.Kill**.