Threading in C#

Joseph Albahari

Last updated 2011-4-27

Table of Contents

Part 1: Getting Started

Introduction and Concepts

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

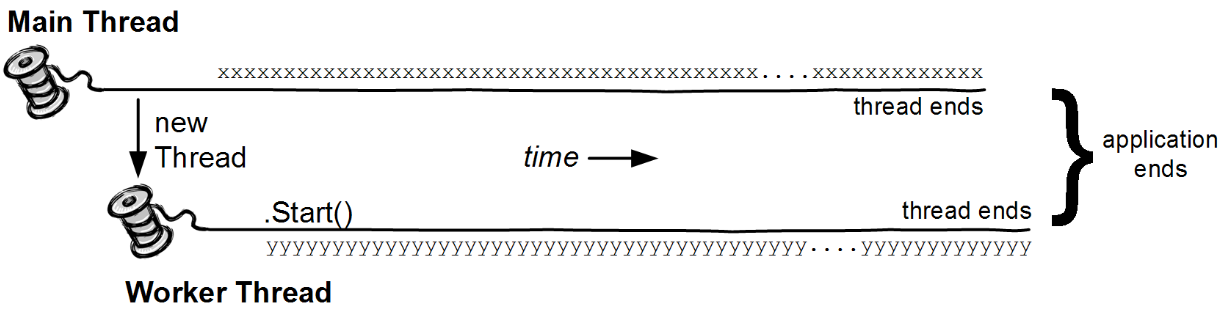
A C# *client* program (Console, WPF, or Windows Forms) starts in a single thread created automatically by the CLR and operating system (the “main” thread), and is made multithreaded by creating additional threads. Here’s a simple example and its output:

> All examples assume the following namespaces are imported:
>
> ```
> using System;
> using System.Threading;
> ```

```
class ThreadTest
{
  static void Main()
  {
    Thread t = new Thread (WriteY);          // Kick off a new thread
    t.Start();                               // running WriteY()

    // Simultaneously, do something on the main thread.
    for (int i = 0; i < 1000; i++) Console.Write ("x");
  }

  static void WriteY()
  {
    for (int i = 0; i < 1000; i++) Console.Write ("y");
  }
}
```

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”. Here’s the output:

```
xxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
...
```


Main Thread
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx....xxxxxxxxxxxxx
thread ends
new
Thread
time
.Start()
thread ends
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy....yyyyyyyyyyyyy
application
ends
Worker Thread

Once started, a thread’s `IsAlive` property returns `true`, until the point where the thread ends. A thread ends when the delegate passed to the `Thread`’s constructor finishes executing. Once ended, a thread cannot restart.

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:

```
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. For 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 (though unlikely) that "Done" could be printed twice. If, however, we swap the order of statements in the `Go` method, 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:

```
class ThreadSafe
{
  static bool done;
  static readonly 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.

> Shared data is the primary cause of complexity and obscure errors in multithreading. Although often essential, it pays to keep it as simple as possible.

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

Join and Sleep

You can wait for another thread to end by calling its `Join` method. For example:

```
static void Main()
{
  Thread t = new Thread (Go);
  t.Start();
  t.Join();
  Console.WriteLine ("Thread t has ended!");
}

static void Go()
{
  for (int i = 0; i < 1000; i++) Console.Write ("y");
}
```

This prints "y" 1,000 times, followed by "Thread t has ended!" immediately afterward. You can include a timeout when calling `Join`, either in milliseconds or as a `TimeSpan`. It then returns `true` if the thread ended or `false` if it timed out.

`Thread.Sleep` pauses the current thread for a specified period:

```
Thread.Sleep (TimeSpan.FromHours (1));  // sleep for 1 hour
Thread.Sleep (500);                     // sleep for 500 milliseconds
```

While waiting on a `Sleep` or `Join`, a thread is blocked and so does not consume CPU resources.

> `Thread.Sleep(0)` relinquishes the thread's current time slice immediately, voluntarily handing over the CPU to other threads. Framework 4.0's new `Thread.Yield()` method does the same thing—except that it relinquishes only to threads running on the *same* processor.
>
> `Sleep(0)` or `Yield` is occasionally useful in production code for advanced performance tweaks. It's also an excellent diagnostic tool for helping to uncover thread safety issues: if inserting `Thread.Yield()` anywhere in your code makes or breaks the program, you almost certainly have a bug.

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-processor computer, a thread scheduler performs *time-slicing*—rapidly switching execution between each of the active threads. Under Windows, a time-slice is typically in the tens-of-milliseconds region—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 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

A thread is analogous to the operating system process in which your application runs. Just as processes run in parallel on a computer, threads run in parallel *within a single process*. Processes are fully isolated from each other; threads have just a limited degree of isolation. In particular, threads share (heap) memory with other threads running in the same application. This, in part, is why threading is useful: one thread can fetch data in the background, for instance, while another thread can display the data as it arrives.

Threading’s Uses and Misuses

Multithreading has many uses; here are the most common:

Maintaining a responsive user interface

By running time-consuming tasks on a parallel “worker” thread, the main UI thread is free to continue processing keyboard and mouse events.

Making efficient use of an otherwise blocked CPU

Multithreading is useful when a thread is awaiting a response from another computer or piece of hardware. While one thread is blocked while performing the task, other threads can take advantage of the otherwise unburdened computer.

Parallel programming

Code that performs intensive calculations can execute faster on multicore or multiprocessor computers if the workload is shared among multiple threads in a “divide-and-conquer” strategy (see Part 5).

Speculative execution

On multicore machines, you can sometimes improve performance by predicting something that might need to be done, and then doing it ahead of time. LINQPad uses this technique to speed up the creation of new queries. A variation is to run a number of different algorithms in parallel that all solve the same task. Whichever one finishes first “wins”—this is effective when you can’t know ahead of time which algorithm will execute fastest.

Allowing requests to be processed simultaneously

On a server, client requests can arrive concurrently and so need to be handled in parallel (the .NET Framework creates threads for this automatically if you use ASP.NET, WCF, Web Services, or Remoting). This can also be useful on a client (e.g., handling peer-to-peer networking—or even multiple requests from the user).

With technologies such as ASP.NET and WCF, you may be unaware that multithreading is even taking place—unless you access shared data (perhaps via static fields) without appropriate locking, running afoul of thread safety.

Threads also come with strings attached. The biggest is that multithreading can increase complexity. Having lots of threads does not in and of itself create much complexity; it’s the interaction between threads (typically via shared data) that does. This applies whether or not the interaction is intentional, and can cause long development cycles and an ongoing susceptibility to intermittent and nonreproducible bugs. For this reason, it pays to keep interaction to a minimum, and to stick to simple and proven designs wherever possible. This article focuses largely on dealing with just these complexities; remove the interaction and there’s much less to say!

> A good strategy is to encapsulate multithreading logic into reusable classes that can be independently examined and tested. The Framework itself offers many higher-level threading constructs, which we cover later.

Threading also incurs a resource and CPU cost in scheduling and switching threads (when there are more active threads than CPU cores)—and there’s also a creation/tear-down cost. Multithreading will not always speed up your application—it can even slow it down if used excessively or inappropriately. For example, when heavy disk I/O is involved, it can be faster to have a couple of worker threads run tasks in sequence than to have 10 threads executing at once. (In Signaling with Wait and Pulse, we describe how to implement a producer/consumer queue, which provides just this functionality.)

Creating and Starting Threads

As we saw in the introduction, threads are created using the `Thread` class’s constructor, passing in a `ThreadStart` delegate which indicates 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 hellos.

A thread can be created more conveniently by specifying just a method group—and allowing C# to infer the `ThreadStart` delegate:

```
Thread t = new Thread (Go);    // No need to explicitly use ThreadStart
```

Another shortcut is to use a lambda expression or anonymous method:

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

Passing Data to a Thread

The easiest way to pass arguments to a thread's target method is to execute a lambda expression that calls the method with the desired arguments:

```
static void Main()
{
  Thread t = new Thread ( () => Print ("Hello from t!") );
  t.Start();
}

static void Print (string message)
{
  Console.WriteLine (message);
}
```

With this approach, you can pass in any number of arguments to the method. You can even wrap the entire implementation in a multi-statement lambda:

```
new Thread (() =>
{
  Console.WriteLine ("I'm running on another thread!");
  Console.WriteLine ("This is so easy!");
}).Start();
```

You can do the same thing almost as easily in C# 2.0 with anonymous methods:

```
new Thread (delegate()
{
  ...
}).Start();
```

Another technique is to pass an argument into `Thread`'s `Start` method:

```
static void Main()
{
  Thread t = new Thread (Print);
  t.Start ("Hello from t!");
}

static void Print (object messageObj)
{
  string message = (string) messageObj;   // We need to cast here
  Console.WriteLine (message);
}
```

This works because `Thread`’s constructor is overloaded to accept either of two delegates:

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

The limitation of `ParameterizedThreadStart` is that it accepts only one argument. And because it’s of type `object`, it usually needs to be cast.

Lambda expressions and captured variables

As we saw, a lambda expression is the most powerful way to pass data to a thread. However, you must be careful about accidentally modifying *captured variables* after starting the thread, because these variables are shared. For instance, consider the following:

```
for (int i = 0; i < 10; i++)
  new Thread (() => Console.Write (i)).Start();
```

The output is nondeterministic! Here’s a typical result:

```
0223557799
```

The problem is that the `i` variable refers to the *same* memory location throughout the loop’s lifetime. Therefore, each thread calls `Console.Write` on a variable whose value may change as it is running!

> This is analogous to the problem we describe in “Captured Variables” in Chapter 8 of C# 4.0 in a Nutshell. The problem is less about multithreading and more about C#'s rules for capturing variables (which are somewhat undesirable in the case of `for` and `foreach` loops).

The solution is to use a temporary variable as follows:

```
for (int i = 0; i < 10; i++)
{
  int temp = i;
  new Thread (() => Console.Write (temp)).Start();
}
```

Variable `temp` is now local to each loop iteration. Therefore, each thread captures a different memory location and there’s no problem. We can illustrate the problem in the earlier code more simply with the following example:

```
string text = "t1";
Thread t1 = new Thread ( () => Console.WriteLine (text) );

text = "t2";
Thread t2 = new Thread ( () => Console.WriteLine (text) );

t1.Start();
t2.Start();
```

Because both lambda expressions capture the same `text` variable, `t2` is printed twice:

```
t2
t2
```

Naming Threads

Each thread has a Name property that you can set for the benefit of debugging. This is particularly useful in Visual Studio, since the thread's name is displayed in the Threads Window and Debug Location toolbar. You can set a thread's name just once; attempts to change it later will throw an exception.

The static Thread.CurrentThread property gives you the currently executing thread. In the following example, we set the main thread's name:

```
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);
  }
}
```

Foreground and Background Threads

By default, threads you create explicitly are *foreground threads*. Foreground threads keep the application alive for as long as any one of them is running, whereas *background threads* do not. Once all foreground threads finish, the application ends, and any background threads still running abruptly terminate.

> A thread's foreground/background status has no relation to its priority or allocation of execution time.

You can query or change a thread's background status using its IsBackground property. Here's an example:

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

If this program is called with no arguments, the worker thread assumes foreground status and will wait on the ReadLine statement for the user to press Enter. Meanwhile, the main thread exits, but the application keeps running because a foreground thread is still alive.

On the other hand, if 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 process terminates in this manner, any finally blocks in the execution stack of background threads are circumvented. This is a problem if your program employs finally (or using) blocks to perform cleanup work such as releasing resources or deleting temporary files. To avoid this, you can explicitly wait out such background threads upon exiting an application. There are two ways to accomplish this:

- If you've created the thread yourself, call Join on the thread.
- If you're on a pooled thread, use an event wait handle.

In either case, you should specify a timeout, so you can abandon a renegade thread should it refuse to finish for some reason. This is your backup exit strategy: in the end, you want your application to close—without the user having to enlist help from the Task Manager!

Foreground threads don't require this treatment, but you must take care to avoid bugs that could cause the thread not to end. A common cause for applications failing to exit properly is the presence of active foreground threads.

> If a user uses the Task Manager to forcibly end a .NET process, all threads "drop dead" as though they were background threads. This is observed rather than documented behavior, and it could vary depending on the CLR and operating system version.

Thread Priority

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

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

This becomes relevant only when multiple threads are simultaneously active.

> Think carefully before elevating a thread's priority—it can lead to problems such as resource starvation for other threads.

Elevating a thread's priority doesn't make it capable of performing real-time work, because it's still throttled by the application's process priority. To perform real-time work, you must also elevate the process priority using the `Process` class in `System.Diagnostics` (we didn't tell you how to do this):

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

`ProcessPriorityClass.High` is actually one notch short of the highest priority: `Realtime`. Setting a process priority to `Realtime` instructs the OS that you never want the process to yield CPU time to another process. If your program enters an accidental infinite loop, you might find even the operating system locked out, with nothing short of the power button left to rescue you! For this reason, `High` is usually the best choice for real-time applications.

> If your real-time application has a user interface, elevating the process priority gives screen updates excessive CPU time, slowing down the entire computer (particularly if the UI is complex). Lowering the main thread's priority in conjunction with raising the process's priority ensures that the real-time thread doesn't get preempted by screen redraws, but doesn't solve the problem of starving other applications of CPU time, because the operating system will still allocate disproportionate resources to the process as a whole. An ideal solution is to have the real-time worker and user interface run as separate applications with different process priorities, communicating via Remoting or memory-mapped files. Memory-mapped files are ideally suited to this task; we explain how they work in Chapters 14 and 25 of C# 4.0 in a Nutshell.

Even with an elevated process priority, there's a limit to the suitability of the managed environment in handling hard real-time requirements. In addition to the issues of latency introduced by automatic garbage collection, the operating system may present additional challenges—even for unmanaged applications—that are best solved with dedicated hardware or a specialized real-time platform.

Exception Handling

Any `try`/`catch`/`finally` blocks in scope when a thread is created are of no relevance to the thread when it 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; }   // Throws a NullReferenceException
```

The `try`/`catch` statement in this example is ineffective, and the newly created thread will be encumbered with an unhandled `NullReferenceException`. This behavior makes sense when you consider that each thread has an independent execution path.

The remedy is to move the exception handler into the `Go` method:

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

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

You need an exception handler on all thread entry methods in production applications—just as you do (usually at a higher level, in the execution stack) on your main thread. An unhandled exception causes the whole application to shut down. With an ugly dialog!

In writing such exception handling blocks, rarely would you *ignore* the error: typically, you'd log the details of the exception, and then perhaps display a dialog allowing the user to automatically submit those details to your web server. You then might shut down the application—because it's possible that the error corrupted the program's state. However, the cost of doing so is that the user will lose his recent work—open documents, for instance.

The "global" exception handling events for WPF and Windows Forms applications (`Application.DispatcherUnhandledException` and `Application.ThreadException`) fire only for exceptions thrown on the main UI thread. You still must handle exceptions on worker threads manually.

`AppDomain.CurrentDomain.UnhandledException` fires on any unhandled exception, but provides no means of preventing the application from shutting down afterward.

There are, however, some cases where you don't need to handle exceptions on a worker thread, because the .NET Framework does it for you. These are covered in upcoming sections, and are:

- Asynchronous delegates
- BackgroundWorker
- The Task Parallel Library (conditions apply)

Thread Pooling

Whenever you start a thread, a few hundred microseconds are spent organizing such things as a fresh private local variable stack. Each thread also consumes (by default) around 1 MB of memory. The *thread pool* cuts these overheads by sharing and recycling threads, allowing multithreading to be applied at a very granular level without a performance penalty. This is useful when leveraging multicore processors to execute computationally intensive code in parallel in “divide-and-conquer” style.

The thread pool also keeps a lid on the total number of worker threads it will run simultaneously. Too many active threads throttle the operating system with administrative burden and render CPU caches ineffective. Once a limit is reached, jobs queue up and start only when another finishes. This makes arbitrarily concurrent applications possible, such as a web server. (The *asynchronous method pattern* is an advanced technique that takes this further by making highly efficient use of the pooled threads; we describe this in Chapter 23 of C# 4.0 in a Nutshell).

There are a number of ways to enter the thread pool:

- Via the Task Parallel Library (from Framework 4.0)
- By calling ThreadPool.QueueUserWorkItem
- Via asynchronous delegates
- Via BackgroundWorker

The following constructs use the thread pool *indirectly*:

- WCF, Remoting, ASP.NET, and ASMX Web Services application servers
- System.Timers.Timer and System.Threading.Timer
- Framework methods that end in *Async*, such as those on WebClient (the event-based asynchronous pattern), and most Begin*XXX* methods (the *asynchronous programming model* pattern)
- PLINQ

The *Task Parallel Library* (TPL) and PLINQ are sufficiently powerful and high-level that you’ll want to use them to assist in multithreading even when thread pooling is unimportant. We discuss these in detail in Part 5; right now, we'll look briefly at how you can use the Task class as a simple means of running a delegate on a pooled thread.

There are a few things to be wary of when using pooled threads:

- You cannot set the Name of a pooled thread, making debugging more difficult (although you can attach a description when debugging in Visual Studio’s Threads window).
- Pooled threads are always background threads (this is usually not a problem).
- Blocking a pooled thread may trigger additional latency in the early life of an application unless you call ThreadPool.SetMinThreads (see Optimizing the Thread Pool).

You are free to change the priority of a pooled thread—it will be restored to normal when released back to the pool.

You can query if you’re currently executing on a pooled thread via the property Thread.CurrentThread.IsThreadPoolThread.

Entering the Thread Pool via TPL

You can enter the thread pool easily using the `Task` classes in the Task Parallel Library. The `Task` classes were introduced in Framework 4.0: if you're familiar with the older constructs, consider the nongeneric `Task` class a replacement for `ThreadPool.QueueUserWorkItem`, and the generic `Task<TResult>` a replacement for asynchronous delegates. The newer constructs are faster, more convenient, and more flexible than the old.

To use the nongeneric `Task` class, call `Task.Factory.StartNew`, passing in a delegate of the target method:

```
static void Main()    // The Task class is in System.Threading.Tasks
{
  Task.Factory.StartNew (Go);
}

static void Go()
{
  Console.WriteLine ("Hello from the thread pool!");
}
```

`Task.Factory.StartNew` returns a `Task` object, which you can then use to monitor the task—for instance, you can wait for it to complete by calling its `Wait` method.

> Any unhandled exceptions are conveniently rethrown onto the host thread when you call a task's `Wait` method. (If you don't call `Wait` and abandon the task, an unhandled exception will shut down the process as with an ordinary thread.)

The generic `Task<TResult>` class is a subclass of the nongeneric `Task`. It lets you get a return value back from the task after it finishes executing. In the following example, we download a web page using `Task<TResult>`:

```
static void Main()
{
  // Start the task executing:
  Task<string> task = Task.Factory.StartNew<string>
    ( () => DownloadString ("http://www.linqpad.net") );

  // We can do other work here and it will execute in parallel:
  RunSomeOtherMethod();

  // When we need the task's return value, we query its Result property:
  // If it's still executing, the current thread will now block (wait)
  // until the task finishes:
  string result = task.Result;
}

static string DownloadString (string uri)
{
  using (var wc = new System.Net.WebClient())
    return wc.DownloadString (uri);
}
```

(The `<string>` type argument in boldface is for clarity: it would be *inferred* if we omitted it.)

Any unhandled exceptions are automatically rethrown when you query the task's `Result` property, wrapped in an `AggregateException`. However, if you fail to query its `Result` property (and don't call `Wait`) any unhandled exception will take the process down.

The Task Parallel Library has many more features, and is particularly well suited to leveraging multicore processors. We'll resume our discussion of TPL in Part 5.

Entering the Thread Pool Without TPL

You can't use the Task Parallel Library if you're targeting an earlier version of the .NET Framework (prior to 4.0). Instead, you must use one of the older constructs for entering the thread pool: `ThreadPool.QueueUserWorkItem` and asynchronous delegates. The difference between the two is that asynchronous delegates let you return data from the thread. Asynchronous delegates also marshal any exception back to the caller.

QueueUserWorkItem

To use `QueueUserWorkItem`, simply call this method with a delegate that you want to run on a pooled thread:

```
static void Main()
{
  ThreadPool.QueueUserWorkItem (Go);
  ThreadPool.QueueUserWorkItem (Go, 123);
  Console.ReadLine();
}

static void Go (object data)   // data will be null with the first call.
{
  Console.WriteLine ("Hello from the thread pool! " + data);
}

// Output:
Hello from the thread pool!
Hello from the thread pool! 123
```

Our target method, `Go`, must accept a single `object` argument (to satisfy the `WaitCallback` delegate). This provides a convenient way of passing data to the method, just like with `ParameterizedThreadStart`. Unlike with `Task`, `QueueUserWorkItem` doesn't return an object to help you subsequently manage execution. Also, you must explicitly deal with exceptions in the target code—unhandled exceptions will take down the program.

Asynchronous delegates

`ThreadPool.QueueUserWorkItem` doesn’t provide an easy mechanism for getting return values back from a thread after it has finished executing. Asynchronous delegate invocations (asynchronous delegates for short) solve this, allowing any number of typed arguments to be passed in both directions. Furthermore, unhandled exceptions on asynchronous delegates are conveniently rethrown on the original thread (or more accurately, the thread that calls `EndInvoke`), and so they don’t need explicit handling.

> Don’t confuse asynchronous delegates with asynchronous methods (methods starting with *Begin* or *End*, such as `File.BeginRead`/`File.EndRead`). Asynchronous methods follow a similar protocol outwardly, but they exist to solve a much harder problem, which we describe in Chapter 23 of C# 4.0 in a Nutshell.

Here’s how you start a worker task via an asynchronous delegate:

1. Instantiate a delegate targeting the method you want to run in parallel (typically one of the predefined `Func` delegates).
2. Call `BeginInvoke` on the delegate, saving its `IAsyncResult` return value.

   `BeginInvoke` returns immediately to the caller. You can then perform other activities while the pooled thread is working.
3. When you need the results, call `EndInvoke` on the delegate, passing in the saved `IAsyncResult` object.

In the following example, we use an asynchronous delegate invocation to execute concurrently with the main thread, a simple method that returns a string’s length:

```
static void Main()
{
  Func<string, int> method = Work;
  IAsyncResult cookie = method.BeginInvoke ("test", null, null);
  //
  // ... here's where we can do other work in parallel...
  //
  int result = method.EndInvoke (cookie);
  Console.WriteLine ("String length is: " + result);
}

static int Work (string s) { return s.Length; }
```

`EndInvoke` does three things. First, it waits for the asynchronous delegate to finish executing, if it hasn't already. Second, it receives the return value (as well as any `ref` or `out` parameters). Third, it throws any unhandled worker exception back to the calling thread.

> If the method you're calling with an asynchronous delegate has no return value, you are still (technically) obliged to call `EndInvoke`. In practice, this is open to debate; there are no `EndInvoke` police to administer punishment to noncompliers! If you choose not to call `EndInvoke`, however, you'll need to consider exception handling on the worker method to avoid silent failures.

You can also specify a callback delegate when calling `BeginInvoke`—a method accepting an `IAsyncResult` object that's automatically called upon completion. This allows the instigating thread to "forget" about the asynchronous delegate, but it requires a bit of extra work at the callback end:

```
static void Main()
{
  Func<string, int> method = Work;
  method.BeginInvoke ("test", Done, method);
  // ...
  //
}

static int Work (string s) { return s.Length; }

static void Done (IAsyncResult cookie)
{
  var target = (Func<string, int>) cookie.AsyncState;
  int result = target.EndInvoke (cookie);
  Console.WriteLine ("String length is: " + result);
}
```

The final argument to `BeginInvoke` is a user state object that populates the `AsyncState` property of `IAsyncResult`. It can contain anything you like; in this case, we're using it to pass the `method` delegate to the completion callback, so we can call `EndInvoke` on it.

Optimizing the Thread Pool

The thread pool starts out with one thread in its pool. As tasks are assigned, the pool manager "injects" new threads to cope with the extra concurrent workload, up to a maximum limit. After a sufficient period of inactivity, the pool manager may "retire" threads if it suspects that doing so will lead to better throughput.

You can set the upper limit of threads that the pool will create by calling `ThreadPool.SetMaxThreads`; the defaults are:

- 1023 in Framework 4.0 in a 32-bit environment
- 32768 in Framework 4.0 in a 64-bit environment
- 250 per core in Framework 3.5
- 25 per core in Framework 2.0

(These figures may vary according to the hardware and operating system.) The reason there are that many is to ensure progress should some threads be blocked (idling while awaiting some condition, such as a response from a remote computer).

You can also set a lower limit by calling `ThreadPool.SetMinThreads`. The role of the lower limit is subtler: it's an advanced optimization technique that instructs the pool manager not to *delay* in the allocation of threads until reaching the lower limit. Raising the minimum thread count improves concurrency when there are blocked threads (see sidebar).

> The default lower limit is one thread per processor core—the minimum that allows full CPU utilization. On server environments, though (such ASP.NET under IIS), the lower limit is typically much higher—as much as 50 or more.

How Does the Minimum Thread Count Work?

Increasing the thread pool's minimum thread count to *x* doesn't actually force *x* threads to be created right away—threads are created only on demand. Rather, it instructs the pool manager to create up to *x* threads the *instant* they are required. The question, then, is why would the thread pool otherwise delay in creating a thread when it's needed?

The answer is to prevent a brief burst of short-lived activity from causing a full allocation of threads, suddenly swelling an application's memory footprint. To illustrate, consider a quad-core computer running a client application that enqueues 40 tasks at once. If each task performs a 10 ms calculation, the whole thing will be over in 100 ms, assuming the work is divided among the four cores. Ideally, we'd want the 40 tasks to run on *exactly four threads*:

- Any less and we'd not be making maximum use of all four cores.
- Any more and we'd be wasting memory and CPU time creating unnecessary threads.

And this is exactly how the thread pool works. Matching the thread count to the core count allows a program to retain a small memory footprint without hurting performance—as long as the threads are efficiently used (which in this case they are).

But now suppose that instead of working for 10 ms, each task queries the Internet, waiting half a second for a response while the local CPU is idle. The pool manager's thread-economy strategy breaks down; it would now do better to create more threads, so all the Internet queries could happen simultaneously.

Fortunately, the pool manager has a backup plan. If its queue remains stationary for more than half a second, it responds by creating more threads—one every half-second—up to the capacity of the thread pool.

The half-second delay is a two-edged sword. On the one hand, it means that a one-off burst of brief activity doesn't make a program suddenly consume an extra unnecessary 40 MB (or more) of memory. On the other hand, it can needlessly delay things when a pooled thread blocks, such as when querying a database or calling `WebClient.DownloadFile`. For this reason, you can tell the pool manager not to delay in the allocation of the first *x* threads, by calling `SetMinThreads`, for instance:

```
ThreadPool.SetMinThreads (50, 50);
```

(The second value indicates how many threads to assign to I/O completion ports, which are used by the APM, described in Chapter 23 of C# 4.0 in a Nutshell.)

The default value is one thread per core.

Part 2: Basic Synchronization

Synchronization Essentials

So far, we’ve described how to start a task on a thread, configure a thread, and pass data in both directions. We’ve also described how local variables are private to a thread and how references can be shared among threads, allowing them to communicate via common fields.

The next step is *synchronization*: coordinating the actions of threads for a predictable outcome. Synchronization is particularly important when threads access the same data; it’s surprisingly easy to run aground in this area.

Synchronization constructs can be divided into four categories:

Simple blocking methods

> These wait for another thread to finish or for a period of time to elapse. `Sleep`, `Join`, and `Task.Wait` are simple blocking methods.

Locking constructs

> These limit the number of threads that can perform some activity or execute a section of code at a time. *Exclusive* locking constructs are most common—these allow just one thread in at a time, and allow competing threads to access common data without interfering with each other. The standard exclusive locking constructs are `lock` (`Monitor.Enter`/`Monitor.Exit`), `Mutex`, and `SpinLock`. The nonexclusive locking constructs are `Semaphore`, `SemaphoreSlim`, and the reader/writer locks.

Signaling constructs

> These allow a thread to pause until receiving a notification from another, avoiding the need for inefficient polling. There are two commonly used signaling devices: event wait handles and `Monitor`’s `Wait`/`Pulse` methods. Framework 4.0 introduces the `CountdownEvent` and `Barrier` classes.

Nonblocking synchronization constructs

> These protect access to a common field by calling upon processor primitives. The CLR and C# provide the following nonblocking constructs: `Thread.MemoryBarrier`, `Thread.VolatileRead`, `Thread.VolatileWrite`, the `volatile` keyword, and the `Interlocked` class.

Blocking is essential to all but the last category. Let’s briefly examine this concept.

Blocking

A thread is deemed *blocked* when its execution is paused for some reason, such as when `Sleep`ing or waiting for another to end via `Join` or `EndInvoke`. A blocked thread immediately *yields* its processor time slice, and from then on consumes no processor time until its blocking condition is satisfied. You can test for a thread being blocked via its `ThreadState` property:

```
bool blocked = (someThread.ThreadState & ThreadState.WaitSleepJoin) != 0;
```

(Given that a thread’s state may change in between testing its state and then acting upon that information, this code is useful only in diagnostic scenarios.)

When a thread blocks or unblocks, the operating system performs a *context switch*. This incurs an overhead of a few microseconds.

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.

Blocking Versus Spinning

Sometimes a thread must pause until a certain condition is met. Signaling and locking constructs achieve this efficiently by blocking until a condition is satisfied. However, there is a simpler alternative: a thread can await a condition by *spinning* in a polling loop. For example:

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

or:

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

In general, this is very wasteful on processor time: as far as the CLR and operating system are concerned, the thread is performing an important calculation, and so gets allocated resources accordingly!

Sometimes a hybrid between blocking and spinning is used:

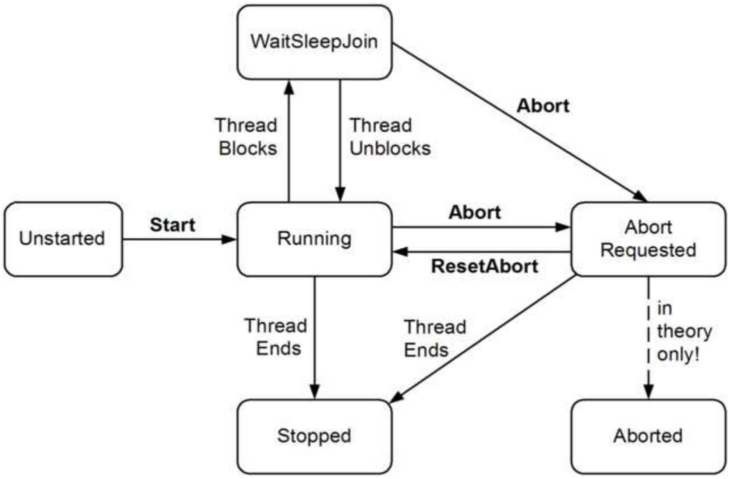
```
while (!proceed) Thread.Sleep (10);
```

Although inelegant, this is (in general) far more efficient than outright spinning. Problems can arise, though, due to concurrency issues on the `proceed` flag. Proper use of locking and signaling avoids this.

> Spinning *very briefly* can be effective when you expect a condition to be satisfied soon (perhaps within a few microseconds) because it avoids the overhead and latency of a context switch. The .NET Framework provides special methods and classes to assist—see “SpinLock and SpinWait”.

ThreadState

You can query a thread's execution status via its `ThreadState` property. This returns a flags enum of type `ThreadState`, which combines three “layers” of data in a bitwise fashion. Most values, however, are redundant, unused, or deprecated. The following diagram shows one “layer”:



The following code strips a `ThreadState` to one of the four most useful values: `Unstarted`, `Running`, `WaitSleepJoin`, and `Stopped`:

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

The ThreadState property is useful for diagnostic purposes, but unsuitable for synchronization, because a thread's state may change in between testing ThreadState and acting on that information.

Locking

Exclusive locking is used to ensure that only one thread can enter particular sections of code at a time. The two main exclusive locking constructs are lock and Mutex. Of the two, the lock construct is faster and more convenient. Mutex, though, has a niche in that its lock can span applications in different processes on the computer.

In this section, we'll start with the lock construct and then move on to Mutex and semaphores (for nonexclusive locking). Later, we'll cover reader/writer locks.

From Framework 4.0, there is also the SpinLock struct for high-concurrency scenarios (see final section).

Let's start with the following class:

```
class ThreadUnsafe
{
  static int _val1 = 1, _val2 = 1;

  static void Go()
  {
    if (_val2 != 0) Console.WriteLine (_val1 / _val2);
    _val2 = 0;
  }
}
```

This class 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 readonly 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 (a caveat is that nuances in the behavior of Windows and the CLR mean that the fairness of the queue can sometimes be violated). 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. In Interrupt and Abort, we describe how a blocked thread can be forcibly released via another thread. This is a fairly heavy-duty technique that might be used in ending a thread.

A Comparison of Locking Constructs

Construct	Purpose	Cross-process?	Overhead*
`lock` (`Monitor.Enter` / `Monitor.Exit`)	Ensures just one thread can access a resource (or section of code) at a time	-	20ns
`Mutex`		Yes	1000ns
`SemaphoreSlim` (introduced in Framework 4.0)	Ensures not more than a specified number of concurrent threads can access a resource	-	200ns
`Semaphore`		Yes	1000ns
`ReaderWriterLockSlim` (introduced in Framework 3.5)	Allows multiple readers to coexist with a single writer	-	40ns
`ReaderWriterLock` (effectively deprecated)		-	100ns

*Time taken to lock and unlock the construct once on the same thread (assuming no blocking), as measured on an Intel Core i7 860.

Monitor.Enter and Monitor.Exit

C#'s `lock` statement is in fact a syntactic shortcut for a call to the methods `Monitor.Enter` and `Monitor.Exit`, with a `try`/`finally` block. Here's (a simplified version of) what's actually happening within the `Go` method of the preceding 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.

The lockTaken overloads

The code that we just demonstrated is exactly what the C# 1.0, 2.0, and 3.0 compilers produce in translating a `lock` statement.

There's a subtle vulnerability in this code, however. Consider the (unlikely) event of an exception being thrown within the implementation of `Monitor.Enter`, or between the call to `Monitor.Enter` and the `try` block (due, perhaps, to `Abort` being called on that thread—or an `OutOfMemoryException` being thrown). In such a scenario, the lock may or may not be taken. If the lock *is* taken, it won't be released—because we'll never enter the `try`/`finally` block. This will result in a leaked lock.

To avoid this danger, CLR 4.0's designers added the following overload to `Monitor.Enter`:

```
public static void Enter (object obj, ref bool lockTaken);
```

`lockTaken` is false after this method if (and only if) the `Enter` method throws an exception and the lock was not taken.

Here's the correct pattern of use (which is exactly how C# 4.0 translates a `lock` statement):

```
bool lockTaken = false;
try
{
  Monitor.Enter (_locker, ref lockTaken);
  // Do your stuff...
}
finally { if (lockTaken) Monitor.Exit (_locker); }
```

TryEnter

Monitor also provides a TryEnter method that 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.

As with the Enter method, it's overloaded in CLR 4.0 to accept a lockTaken argument.

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. The synchronizing object is typically private (because this helps to encapsulate the locking logic) and is typically an instance or static field. The synchronizing object can double as the object it's protecting, as the _list field does in the following example:

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

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

A field dedicated for the purpose of locking (such as _locker, in the example prior) allows precise control over the scope and granularity of the lock. The containing object (this)—or even its type—can also be used as a synchronization object:

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

or:

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

The disadvantage of locking in this way is that you're not encapsulating the locking logic, so it becomes harder to prevent deadlocking and excessive blocking. A lock on a type may also seep through application domain boundaries (within the same process).

You can also lock on local variables captured by lambda expressions or anonymous methods.

> 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.

When to Lock

As a basic rule, you need to lock around accessing *any writable shared field*. Even in the simplest case—an assignment operation on a single field—you 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 ThreadSafe
{
  static readonly object _locker = new object();
  static int _x;

  static void Increment() { lock (_locker) _x++; }
  static void Assign()    { lock (_locker) _x = 123; }
}
```

In Nonblocking Synchronization, we explain how this need arises, and how the memory barriers and the `Interlocked` class can provide alternatives to locking in these situations.

Locking and Atomicity

If a group of variables are always read and written within the same lock, you can say the variables are read and written *atomically*. Let's suppose fields `x` and `y` are always read and 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 it 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.

The atomicity provided by a lock is violated if an exception is thrown within a `lock` block. For example, consider the following:

```
decimal _savingsBalance, _checkBalance;

void Transfer (decimal amount)
{
  lock (_locker)
  {
    _savingsBalance += amount;
    _checkBalance -= amount + GetBankFee();
  }
}
```

If an exception was thrown by `GetBankFee()`, the bank would lose money. In this case, we could avoid the problem by calling `GetBankFee` earlier. A solution for more complex cases is to implement "rollback" logic within a `catch` or `finally` block.

Instruction atomicity is a different, although analogous concept: an instruction is atomic if it executes indivisibly on the underlying processor (see Nonblocking Synchronization).

Nested Locking

A thread can repeatedly lock the same object in a nested (*reentrant*) fashion:

```
lock (locker)
  lock (locker)
    lock (locker)
    {
       // Do something...
    }
```

or:

```
Monitor.Enter (locker); Monitor.Enter (locker);  Monitor.Enter (locker);
// Do something...
Monitor.Exit (locker);  Monitor.Exit (locker);   Monitor.Exit (locker);
```

In these scenarios, the object is unlocked only when the outermost `lock` statement has exited—or a matching number of `Monitor.Exit` statements have executed.

Nested locking is useful when one method calls another within a lock:

```
static readonly object _locker = new object();

static void Main()
{
  lock (_locker)
  {
     AnotherMethod();
     // We still have the lock - because locks are reentrant.
  }
}

static void AnotherMethod()
{
  lock (_locker) { Console.WriteLine ("Another method"); }
}
```

A thread can block on only the first (outermost) lock.

Deadlocks

A deadlock happens when two threads each wait for a resource held by the other, so neither can proceed. The easiest way to illustrate this is with two locks:

```
object locker1 = new object();
object locker2 = new object();

new Thread (() => {
                    lock (locker1)
                    {
                      Thread.Sleep (1000);
                      lock (locker2);      // Deadlock
                    }
                  }).Start();
lock (locker2)
{
  Thread.Sleep (1000);
  lock (locker1);                          // Deadlock
}
```

More elaborate deadlocking chains can be created with three or more threads.

The CLR, in a standard hosting environment, is not like SQL Server and does not automatically detect and resolve deadlocks by terminating one of the offenders. A threading deadlock causes participating threads to block indefinitely, unless you’ve specified a locking timeout. (Under the SQL CLR integration host, however, deadlocks *are* automatically detected and a [catchable] exception is thrown on one of the threads.)

Deadlocking is one of the hardest problems in multithreading—especially when there are many interrelated objects. Fundamentally, the hard problem is that you can't be sure what locks your *caller* has taken out.

So, you might innocently lock private field `a` within your class `x`, unaware that your caller (or caller's caller) has already locked field `b` within class `y`. Meanwhile, another thread is doing the reverse—creating a deadlock. Ironically, the problem is exacerbated by (good) object-oriented design patterns, because such patterns create call chains that are not determined until runtime.

The popular advice, “lock objects in a consistent order to avoid deadlocks,” although helpful in our initial example, is hard to apply to the scenario just described. A better strategy is to be wary of locking around calling methods in objects that may have references back to your own object. Also, consider whether you really need to lock around calling methods in other classes (often you do—as we’ll see later—but sometimes there are other options). Relying more on declarative and data parallelism, immutable types, and nonblocking synchronization constructs, can lessen the need for locking.

Here is an alternative way to perceive the problem: when you call out to other code while holding a lock, the encapsulation of that lock subtly *leaks*. This is not a fault in the CLR or .NET Framework, but a fundamental limitation of locking in general. The problems of locking are being addressed in various research projects, including *Software Transactional Memory*.

Another deadlocking scenario arises when calling `Dispatcher.Invoke` (in a WPF application) or `Control.Invoke` (in a Windows Forms application) while in possession of a lock. If the UI happens to be running another method that's waiting on the same lock, a deadlock will happen right there. This can often be fixed simply by calling `BeginInvoke` instead of `Invoke`. Alternatively, you can release your lock before calling `Invoke`, although this won't work if your *caller* took out the lock. We explain `Invoke` and `BeginInvoke` in Rich Client Applications and Thread Affinity.

Performance

Locking is fast: you can expect to acquire and release a lock in as little as 20 nanoseconds on a 2010-era computer if the lock is uncontended. If it is contended, the consequential context switch moves the overhead closer to the microsecond region, although it may be longer before the thread is actually rescheduled. You can avoid the cost of a context switch with the `SpinLock` class—if you're locking very briefly (see final section).

Locking can degrade concurrency if locks are held for too long. This can also increase the chance of deadlock.

Mutex

A `Mutex` is like a C# `lock`, but it can work across multiple processes. In other words, `Mutex` can be *computer-wide* as well as *application-wide*.

Acquiring and releasing an uncontended `Mutex` takes a few microseconds—about 50 times slower than a `lock`.

With a `Mutex` class, you call the `WaitOne` method to lock and `ReleaseMutex` to unlock. Closing or disposing a `Mutex` automatically releases it. Just as with the `lock` statement, a `Mutex` can be released only from the same thread that obtained it.

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

```
class OneAtATimePlease
{
  static void Main()
  {
    // Naming a Mutex makes it available computer-wide. Use a name that's
    // unique to your company and application (e.g., include your URL).

    using (var mutex = new Mutex (false, "oreilly.com OneAtATimeDemo"))
    {
      // Wait a few seconds if contended, in case another instance
      // of the program is still in the process of shutting down.

      if (!mutex.WaitOne (TimeSpan.FromSeconds (3), false))
      {
        Console.WriteLine ("Another instance of the app is running. Bye!");
        return;
      }
      RunProgram();
    }
  }

  static void RunProgram()
  {
    Console.WriteLine ("Running. Press Enter to exit");
    Console.ReadLine();
  }
}
```

If running under Terminal Services, a computer-wide Mutex is ordinarily visible only to applications in the same terminal server session. To make it visible to all terminal server sessions, prefix its name with *Global*.

Semaphore

A semaphore is like a nightclub: it has a certain capacity, enforced by a bouncer. Once it's full, no more people can enter, and a queue builds up outside. Then, for each person that leaves, one person enters 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 club'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, whereas with Mutex and lock, only the thread that obtained the lock can release it.

There are two functionally similar versions of this class: Semaphore and SemaphoreSlim. The latter was introduced in Framework 4.0 and has been optimized to meet the low-latency demands of parallel programming. It's also useful in traditional multithreading because it lets you specify a cancellation token when waiting. It cannot, however, be used for interprocess signaling.

Semaphore incurs about 1 microsecond in calling WaitOne or Release; SemaphoreSlim incurs about a quarter of that.

Semaphores can be useful in limiting concurrency—preventing too many threads from executing a particular piece of code at once. In the following example, five threads try to enter a nightclub that allows only three threads in at once:

```
class TheClub      // No door lists!
{
  static SemaphoreSlim _sem = new SemaphoreSlim (3);    // Capacity of 3

  static void Main()
  {
    for (int i = 1; i <= 5; i++) new Thread (Enter).Start (i);
  }

  static void Enter (object id)
  {
    Console.WriteLine (id + " wants to enter");
    _sem.Wait();
    Console.WriteLine (id + " is in!");           // Only three threads
    Thread.Sleep (1000 * (int) id);               // can be here at
    Console.WriteLine (id + " is leaving");       // a time.
    _sem.Release();
  }
}
```

```
1 wants to enter
1 is in!
2 wants to enter
2 is in!
3 wants to enter
3 is in!
4 wants to enter
5 wants to enter
1 is leaving
4 is in!
2 is leaving
5 is in!
```

If the Sleep statement was instead performing intensive disk I/O, the Semaphore would improve overall performance by limiting excessive concurrent hard-drive activity.

A Semaphore, if named, can span processes in the same way as a Mutex.

Thread Safety

A program or method is thread-safe if it has no indeterminacy in the face of any multithreading scenario. Thread safety is achieved primarily with locking and by reducing the possibilities for thread interaction.

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 multithreaded 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 often the work involved in the latter makes 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 multithreaded environment. One is to sacrifice granularity by wrapping large sections of code—even access to an entire object—within a single exclusive lock, enforcing serialized access at a high level. This tactic is, in fact, essential if you want to use thread-unsafe third-party code (or most Framework types, for that matter) in a multithreaded context. The trick is simply to use the same exclusive lock to protect access to all properties, methods, and fields on the thread-unsafe object. The solution works well if the object's methods all execute quickly (otherwise, there will be a lot of blocking).

> Primitive types aside, 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 with exclusive locks. (The collections in `System.Collections.Concurrent` are an exception.)

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, the server methods they call must be thread-safe. A stateless design (popular for reasons of scalability) intrinsically limits the possibility of interaction, since classes do not persist data between requests. Thread interaction is then limited just to the static fields one may choose to create, for such purposes as caching commonly used data in memory and in providing infrastructure services such as authentication and auditing.

The final approach in implementing thread safety is to use an automatic locking regime. The .NET Framework does exactly this, if you subclass `ContextBoundObject` and apply the `Synchronization` attribute to the class. Whenever a method or property on such an object is then called, an object-wide lock is automatically taken for the whole execution of the method or property. Although this reduces the thread-safety burden, it creates problems of its own: deadlocks that would not otherwise occur, impoverished concurrency, and unintended reentrancy. For these reasons, manual locking is generally a better option—at least until a less simplistic automatic locking regime becomes available.

Thread Safety and .NET Framework Types

Locking can be used to convert thread-unsafe code into thread-safe code. A good application of this is the .NET Framework: nearly all of its nonprimitive types are not thread-safe (for anything more than read-only access) when instantiated, and yet they can be used in multithreaded code if all access to any given object is protected via a lock. Here's an example, where two threads simultaneously add an item to the same `List` collection, then enumerate the list:

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

  static void Main()
  {
    new Thread (AddItem).Start();
    new Thread (AddItem).Start();
  }
  static void AddItem()
  {
    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. If we had two interrelated lists, we would have to choose a common object upon which to lock (we could nominate one of the lists, or better: use an independent field).

Enumerating .NET collections is also thread-unsafe in the sense that an exception is thrown if the list is modified 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. (Another solution is to use a reader/writer lock.)

Locking around thread-safe objects

Sometimes you also need to lock around accessing thread-safe objects. To illustrate, imagine that the Framework's `List` class was, indeed, thread-safe, and we want to add an item to a list:

```
if (!_list.Contains (newItem)) _list.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 in order 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:

```
_list.Clear();
```

to ensure that it did not preempt the former statement. In other words, we would have to lock exactly as with our thread-unsafe collection classes (making the `List` class's hypothetical thread safety redundant).

> Locking around accessing a collection can cause excessive blocking in highly concurrent environments. To this end, Framework 4.0 provides a thread-safe queue, stack, and dictionary (see "Concurrent Collections").

Static methods

Wrapping access to an object around a custom lock works only if all concurrent threads are aware of—and use—the lock. This may not be the case if the object is widely scoped. The worst case is with static members in a public type. For instance, imagine if 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))`—before calling `DateTime.Now`. This would work only if all programmers agreed to do this (which is unlikely). Furthermore, locking a type creates problems of its own.

For this reason, static members on the `DateTime` struct have been carefully programmed to be thread-safe. This is a common pattern throughout the .NET Framework: *static members are thread-safe; instance members are not.* Following this pattern also makes sense when writing types for public consumption, so as not to create impossible thread-safety conundrums. In other words, by making static methods thread-safe, you're programming so as not to *preclude* thread safety for consumers of that type.

> Thread safety in static methods is something that you must explicitly code: it doesn't happen automatically by virtue of the method being static!

Read-only thread safety

Making types thread-safe for concurrent read-only access (where possible) is advantageous because it means that consumers can avoid excessive locking. Many of the .NET Framework types follow this principle: collections, for instance, are thread-safe for concurrent readers.

Following this principle yourself is simple: if you document a type as being thread-safe for concurrent read-only access, don't write to fields within methods that a consumer would expect to be read-only (or lock around doing so). For instance, in implementing a ToArray() method in a collection, you might start by compacting the collection's internal structure. However, this would make it thread-unsafe for consumers that expected this to be read-only.

Read-only thread safety is one of the reasons that enumerators are separate from "enumerables": two threads can simultaneously enumerate over a collection because each gets a separate enumerator object.

In the absence of documentation, it pays to be cautious in assuming whether a method is read-only in nature. A good example is the Random class: when you call Random.Next(), its internal implementation requires that it update private seed values. Therefore, you must either lock around using the Random class, or maintain a separate instance per thread.

Thread Safety in Application Servers

Application servers need to be multithreaded to handle simultaneous client requests. WCF, ASP.NET, and Web Services applications are implicitly multithreaded; the same holds true for Remoting server applications that use a network channel such as TCP or HTTP. This means that when writing code on the server side, you must consider thread safety if there's any possibility of interaction among the threads processing client requests. Fortunately, such a possibility is rare; a typical server class is either stateless (no fields) or has an activation model that creates a separate object instance for each client or each request. Interaction usually arises only through static fields, sometimes used for caching in memory parts of a database to improve performance.

For example, suppose you have a RetrieveUser method that queries a database:

```
// User is a custom class with fields for user data
internal User RetrieveUser (int id) { ... }
```

If this method was called frequently, you could improve performance by caching the results in a static Dictionary. Here's a solution that takes thread safety into account:

```
static class UserCache
{
  static Dictionary <int, User> _users = new Dictionary <int, User>();

  internal static User GetUser (int id)
  {
    User u = null;

    lock (_users)
      if (_users.TryGetValue (id, out u))
        return u;

    u = RetrieveUser (id);           // Method to retrieve from database;
    lock (_users) _users [id] = u;
    return u;
  }
}
```

We must, at a minimum, lock around reading and updating the dictionary to ensure thread safety. In this example, we choose a practical compromise between simplicity and performance in locking. Our design actually creates a very small potential for inefficiency: if two threads simultaneously called this method with the same previously unretrieved id, the RetrieveUser method would be called twice—and the dictionary would be updated unnecessarily. Locking once across the whole method would prevent this, but would create a worse inefficiency: the entire cache would be locked up for the duration of calling RetrieveUser, during which time other threads would be blocked in retrieving *any* user.

Rich Client Applications and Thread Affinity

Both the Windows Presentation Foundation (WPF) and Windows Forms libraries follow models based on thread affinity. Although each has a separate implementation, they are both very similar in how they function.

The objects that make up a rich client are based primarily on DependencyObject in the case of WPF, or Control in the case of Windows Forms. These objects have *thread affinity*, which means that only the thread that instantiates them can subsequently access their members. Violating this causes either unpredictable behavior, or an exception to be thrown.

On the positive side, this means you don't need to lock around accessing a UI object. On the negative side, if you want to call a member on object X created on another thread Y, you must marshal the request to thread Y. You can do this explicitly as follows:

- In WPF, call `Invoke` or `BeginInvoke` on the element's `Dispatcher` object.
- In Windows Forms, call `Invoke` or `BeginInvoke` on the control.

`Invoke` and `BeginInvoke` both accept a delegate, which references the method on the target control that you want to run. `Invoke` works *synchronously*: the caller blocks until the marshal is complete. `BeginInvoke` works *asynchronously*: the caller returns immediately and the marshaled request is queued up (using the same message queue that handles keyboard, mouse, and timer events).

Assuming we have a window that contains a text box called `txtMessage`, whose content we wish a worker thread to update, here's an example for WPF:

```
public partial class MyWindow : Window
{
  public MyWindow()
  {
    InitializeComponent();
    new Thread (Work).Start();
  }

  void Work()
  {
    Thread.Sleep (5000);           // Simulate time-consuming task
    UpdateMessage ("The answer");
  }

  void UpdateMessage (string message)
  {
    Action action = () => txtMessage.Text = message;
    Dispatcher.Invoke (action);
  }
}
```

The code is similar for Windows Forms, except that we call the (`Form`'s) `Invoke` method instead:

```
  void UpdateMessage (string message)
  {
    Action action = () => txtMessage.Text = message;
    this.Invoke (action);
  }
```

The Framework provides two constructs to simplify this process:

- `BackgroundWorker`
- `Task` continuations

Worker threads versus UI threads

It's helpful to think of a rich client application as having two distinct categories of threads: UI threads and worker threads. UI threads instantiate (and subsequently "own") UI elements; worker threads do not. Worker threads typically execute long-running tasks such as fetching data.

Most rich client applications have a single UI thread (which is also the main application thread) and periodically spawn worker threads—either directly or using `BackgroundWorker`. These workers then marshal back to the main UI thread in order to update controls or report on progress.

So, when would an application have multiple UI threads? The main scenario is when you have an application with multiple top-level windows, often called a *Single Document Interface* (SDI) application, such as Microsoft Word. Each SDI window typically shows itself as a separate "application" on the taskbar and is mostly isolated, functionally, from other SDI windows. By giving each such window its own UI thread, the application can be made more responsive.

Immutable Objects

An immutable object is one whose state cannot be altered—externally or internally. The fields in an immutable object are typically declared read-only and are fully initialized during construction.

Immutability is a hallmark of functional programming—where instead of *mutating* an object, you create a new object with different properties. LINQ follows this paradigm. Immutability is also valuable in multithreading in that it avoids the problem of shared writable state—by eliminating (or minimizing) the writable.

One pattern is to use immutable objects to encapsulate a group of related fields, to minimize lock durations. To take a very simple example, suppose we had two fields as follows:

```
int _percentComplete;
string _statusMessage;
```

and we wanted to read/write them atomically. Rather than locking around these fields, we could define the following immutable class:

```
class ProgressStatus    // Represents progress of some activity
{
  public readonly int PercentComplete;
  public readonly string StatusMessage;

  // This class might have many more fields...

  public ProgressStatus (int percentComplete, string statusMessage)
  {
    PercentComplete = percentComplete;
    StatusMessage = statusMessage;
  }
}
```

Then we could define a single field of that type, along with a locking object:

```
readonly object _statusLocker = new object();
ProgressStatus _status;
```

We can now read/write values of that type without holding a lock for more than a single assignment:

```
var status = new ProgressStatus (50, "Working on it");
// Imagine we were assigning many more fields...
// ...
lock (_statusLocker) _status = status;    // Very brief lock
```

To read the object, we first obtain a copy of the object (within a lock). Then we can read its values without needing to hold on to the lock:

```
ProgressStatus statusCpy;
lock (_locker ProgressStatus) statusCopy = _status;   // Again, a brief lock
int pc = statusCopy.PercentComplete;
string msg = statusCopy.StatusMessage;
...
```

Technically, the last two lines of code are thread-safe by virtue of the preceding lock performing an implicit memory barrier (see part 4).

Note that this lock-free approach prevents inconsistency within a group of related fields. But it doesn't prevent data from changing while you subsequently act on it—for this, you usually need a lock. In Part 5, we'll see more examples of using immutability to simplify multithreading—including PLINQ.

It's also possible to safely assign a new `ProgressStatus` object based on its preceding value (e.g., it's possible to "increment" the `PercentComplete` value)—without locking over more than one line of code. In fact, we can do this without using a single lock, through the use of explicit memory barriers, `Interlocked.CompareExchange`, and spin-waits. This is an advanced technique which we describe later (see "SpinLock and SpinWait").

Signaling with Event Wait Handles

Event wait handles are used for *signaling*. Signaling is when one thread waits until it receives notification from another. Event wait handles are the simplest of the signaling constructs, and they are unrelated to C# events. They come in three flavors: AutoResetEvent, ManualResetEvent, and (from Framework 4.0) CountdownEvent. The former two are based on the common EventWaitHandle class, where they derive all their functionality.

A Comparison of Signaling Constructs

Construct	Purpose	Cross-process?	Overhead*
AutoResetEvent	Allows a thread to unblock once when it receives a signal from another	Yes	1000ns
ManualResetEvent	Allows a thread to unblock indefinitely when it receives a signal from another (until reset)	Yes	1000ns
ManualResetEventSlim (introduced in Framework 4.0)		-	40ns
CountdownEvent (introduced in Framework 4.0)	Allows a thread to unblock when it receives a predetermined number of signals	-	40ns
Barrier (introduced in Framework 4.0)	Implements a thread execution barrier	-	80ns
Wait and Pulse	Allows a thread to block until a custom condition is met	-	120ns for a Pulse

*Time taken to signal and wait on the construct once on the same thread (assuming no blocking), as measured on an Intel Core i7 860.

AutoResetEvent

An AutoResetEvent is 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 steps 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. (As with locks, the fairness of the queue can sometimes be violated due to nuances in the operating system). 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.

You can create an AutoResetEvent in two ways. The first is via its constructor:

```
var auto = new AutoResetEvent (false);
```

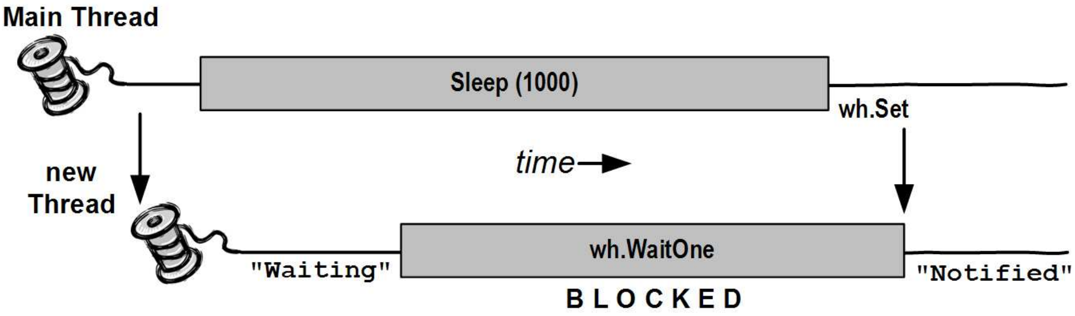
(Passing true into the constructor is equivalent to immediately calling Set upon it.) The second way to create an AutoResetEvent is as follows:

```
var auto = new EventWaitHandle (false, EventResetMode.AutoReset);
```

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

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

  static void Main()
  {
    new Thread (Waiter).Start();
    Thread.Sleep (1000);                  // Pause for a second...
    _waitHandle.Set();                    // Wake up the Waiter.
  }
```

```
    static void Waiter()
    {
      Console.WriteLine ("Waiting...");
      _waitHandle.WaitOne();                // Wait for notification
      Console.WriteLine ("Notified");
    }
  }

  // Output:
  Waiting... (pause) Notified.
```



If `Set` is called when no thread is waiting, the handle stays open for as long as it takes until some thread calls `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."

Calling `Reset` on an `AutoResetEvent` closes the turnstile (should it be open) without waiting or blocking.

`WaitOne` accepts an optional timeout parameter, returning `false` if the wait ended because of a timeout rather than obtaining the signal.

> Calling `WaitOne` with a timeout of `0` tests whether a wait handle is "open," without blocking the caller. Bear in mind, though, that doing this resets the `AutoResetEvent` if it's open.

Disposing Wait Handles

Once you've finished with a wait handle, you can call its `Close` method to release the operating system resource. Alternatively, you can simply drop all references to the wait handle and allow the garbage collector to do the job for you sometime later (wait handles implement the disposal pattern whereby the finalizer calls `Close`). This is one of the few scenarios where relying on this backup is (arguably) acceptable, because wait handles have a light OS burden (asynchronous delegates rely on exactly this mechanism to release their `IAsyncResult`'s wait handle).

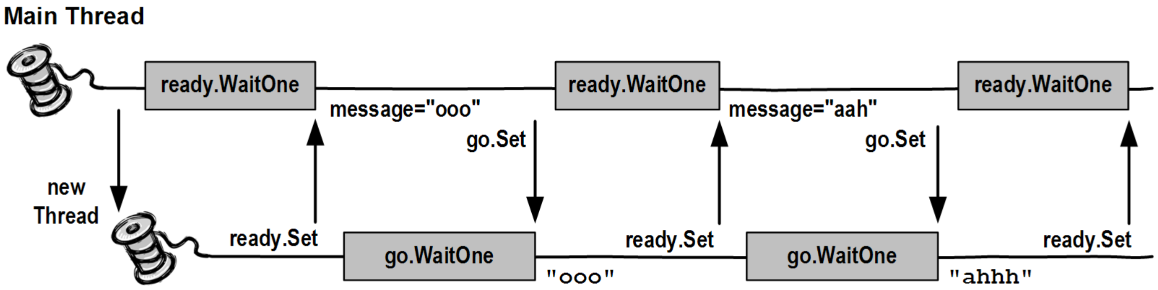
Wait handles are released automatically when an application domain unloads.

Two-way signaling

Let's say we want the main thread to signal a worker thread three times in a row. If the main thread simply calls `Set` on a wait handle several times in rapid succession, the second or third signal may get lost, since the worker may take time to process each signal.

The solution is for the main thread to wait until the worker's ready before signaling it. This can be done with another `AutoResetEvent`, as follows:

```
class TwoWaySignaling
{
  static EventWaitHandle _ready = new AutoResetEvent (false);
  static EventWaitHandle _go = new AutoResetEvent (false);
  static readonly object _locker = new object();
  static string _message;

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

    _ready.WaitOne();                  // First wait until worker is ready
    lock (_locker) _message = "ooo";
    _go.Set();                         // Tell worker to go

    _ready.WaitOne();
    lock (_locker) _message = "ahhh";  // Give the worker another message
    _go.Set();

    _ready.WaitOne();
    lock (_locker) _message = null;    // Signal the worker to exit
    _go.Set();
  }

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

// Output:
ooo
ahhh
```



Here, we're using a null message to indicate that the worker should end. With threads that run indefinitely, it's important to have an exit strategy!

Producer/consumer queue

A producer/consumer queue is a common requirement in threading. Here's how it works:

- A queue is set up to describe work items—or data upon which work is performed.
- When a task needs executing, it's enqueued, allowing the caller to get on with other things.
- One or more worker threads plug away in the background, picking off and executing queued items.

The advantage of this model is that you have precise control over how many worker threads execute at once. This can allow you to limit consumption of not only CPU time, but other resources as well. If the tasks perform intensive disk I/O, for instance, you might have just one worker thread to avoid starving the operating system and other applications. Another type of application may have 20. You can also dynamically add and remove workers throughout the queue's life. The CLR's thread pool itself is a kind of producer/consumer queue.

A producer/consumer queue typically holds items of data upon which (the same) task is performed. For example, the items of data may be filenames, and the task might be to encrypt those files.

In the example below, we use a single `AutoResetEvent` to signal a worker, which waits when it runs out of tasks (in other words, when the queue is empty). We end the worker by enqueing a null task:

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

class ProducerConsumerQueue : IDisposable
{
  EventWaitHandle _wh = new AutoResetEvent (false);
  Thread _worker;
  readonly 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
    }
  }
}
```

To ensure thread safety, we used a lock to protect access to the Queue<string> collection. We also explicitly closed the wait handle in our Dispose method, since we could potentially create and destroy many instances of this class within the life of the application.

Here's a main method to test the queue:

```
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!
```

> Framework 4.0 provides a new class called BlockingCollection<T> that implements the functionality of a producer/consumer queue (see “Concurrent Collections”).
>
> Our manually written producer/consumer queue is still valuable—not only to illustrate AutoResetEvent and thread safety, but also as a basis for more sophisticated structures. For instance, if we wanted a *bounded blocking queue* (limiting the number of enqueued tasks) and also wanted to support cancellation (and removal) of enqueued work items, our code would provide an excellent starting point. We’ll take the producer/consume queue example further in our discussion of Wait and Pulse.

ManualResetEvent

A ManualResetEvent functions like an ordinary gate. Calling Set opens the gate, allowing *any* number of threads calling WaitOne to be let through. Calling Reset closes the gate. Threads that call WaitOne on a closed gate will block; when the gate is next opened, they will be released all at once. Apart from these differences, a ManualResetEvent functions like an AutoResetEvent.

As with AutoResetEvent, you can construct a ManualResetEvent in two ways:

```
var manual1 = new ManualResetEvent (false);
var manual2 = new EventWaitHandle (false, EventResetMode.ManualReset);
```

> From Framework 4.0, there's another version of ManualResetEvent called ManualResetEventSlim. The latter is optimized for short waiting times—with the ability to opt into spinning for a set number of iterations. It also has a more efficient managed implementation and allows a Wait to be canceled via a CancellationToken. It cannot, however, be used for interprocess signaling. ManualResetEventSlim doesn’t subclass WaitHandle; however, it exposes a WaitHandle property that returns a WaitHandle-based object when called (with the performance profile of a traditional wait handle).

Signaling Constructs and Performance

Waiting or signaling an AutoResetEvent or ManualResetEvent takes about one microsecond (assuming no blocking).

ManualResetEventSlim and CountdownEvent can be up to 50 times faster in short-wait scenarios, because of their nonreliance on the operating system and judicious use of spinning constructs.

In most scenarios, however, the overhead of the signaling classes themselves doesn't create a bottleneck, and so is rarely a consideration. An exception is with highly concurrent code, which we'll discuss in Part 5.

A ManualResetEvent is useful in allowing one thread to unblock many other threads. The reverse scenario is covered by CountdownEvent.

CountdownEvent

CountdownEvent lets you wait on more than one thread. The class is new to Framework 4.0 and has an efficient fully managed implementation.

> If you're running on an earlier version of the .NET Framework, all is not lost! Later on, we show how to write a CountdownEvent using Wait and Pulse.

To use CountdownEvent, instantiate the class with the number of threads or "counts" that you want to wait on:

```
var countdown = new CountdownEvent (3);  // Initialize with "count" of 3.
```

Calling Signal decrements the "count"; calling Wait blocks until the count goes down to zero. For example:

```
static CountdownEvent _countdown = new CountdownEvent (3);

static void Main()
{
  new Thread (SaySomething).Start ("I am thread 1");
  new Thread (SaySomething).Start ("I am thread 2");
  new Thread (SaySomething).Start ("I am thread 3");

  _countdown.Wait();   // Blocks until Signal has been called 3 times

  Console.WriteLine ("All threads have finished speaking!");
}

static void SaySomething (object thing)
{
  Thread.Sleep (1000);
  Console.WriteLine (thing);
  _countdown.Signal();
}
```

> Problems for which CountdownEvent is effective can sometimes be solved more easily using the *structured parallelism* constructs that we'll cover in Part 5 (PLINQ and the Parallel class).

You can reincrement a CountdownEvent's count by calling AddCount. However, if it has already reached zero, this throws an exception: you can't "unsignal" a CountdownEvent by calling AddCount. To avoid the possibility of an exception being thrown, you can instead call TryAddCount, which returns false if the countdown is zero.

To unsignal a countdown event, call Reset: this both unsignals the construct and resets its count to the original value.

Like ManualResetEventSlim, CountdownEvent exposes a WaitHandle property for scenarios where some other class or method expects an object based on WaitHandle.

Creating a Cross-Process EventWaitHandle

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

Wait Handles and the Thread Pool

If your application has lots of threads that spend most of their time blocked on a wait handle, you can reduce the resource burden by calling ThreadPool.RegisterWaitForSingleObject. This method accepts a delegate that is executed when a wait handle is signaled. While it's waiting, it doesn't tie up a thread:

```
static ManualResetEvent _starter = new ManualResetEvent (false);

public static void Main()
{
  RegisteredWaitHandle reg = ThreadPool.RegisterWaitForSingleObject
   (_starter, Go, "Some Data", -1, true);
  Thread.Sleep (5000);
  Console.WriteLine ("Signaling worker...");
  _starter.Set();
  Console.ReadLine();
  reg.Unregister (_starter);    // Clean up when we're done.
}

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

// Output:
(5 second delay)
Signaling worker...
Started - Some Data
```

When the wait handle is signaled (or a timeout elapses), the delegate runs on a pooled thread.

In addition to the wait handle and delegate, RegisterWaitForSingleObject accepts a "black box" object that it passes to your delegate method (rather like ParameterizedThreadStart), as well as a timeout in milliseconds (–1 meaning no timeout) and a boolean flag indicating whether the request is one-off rather than recurring.

RegisterWaitForSingleObject is particularly valuable in an application server that must handle many concurrent requests. Suppose you need to block on a ManualResetEvent and simply call WaitOne:

```
void AppServerMethod()
{
  _wh.WaitOne();
  // ... continue execution
}
```

If 100 clients called this method, 100 server threads would be tied up for the duration of the blockage. Replacing _wh.WaitOne with RegisterWaitForSingleObject allows the method to return immediately, wasting no threads:

```
void AppServerMethod
{
  RegisteredWaitHandle reg = ThreadPool.RegisterWaitForSingleObject
   (_wh, Resume, null, -1, true);
  ...
}

static void Resume (object data, bool timedOut)
{
  // ... continue execution
}
```

The data object passed to `Resume` allows continuance of any transient data.

WaitAny, WaitAll, and SignalAndWait

In addition to the `Set`, `WaitOne`, and `Reset` methods, there are static methods on the `WaitHandle` class to crack more complex synchronization nuts. The `WaitAny`, `WaitAll`, and `SignalAndWait` methods perform atomic signaling and waiting operations on multiple handles. The wait handles can be of differing types (including `Mutex` and `Semphore`, since these also derive from the abstract `WaitHandle` class). `ManualResetEventSlim` and `CountdownEvent` can also partake in these methods via their `WaitHandle` properties.

> `WaitAll` and `SignalAndWait` have a weird connection to the legacy COM architecture: these methods require that the caller be in a multithreaded apartment, the model least suitable for interoperability. The main thread of a WPF or Windows application, for example, is unable to interact with the clipboard in this mode. We'll discuss alternatives shortly.

`WaitHandle.WaitAny` waits for any one of an array of wait handles; `WaitHandle.WaitAll` waits on all of the given handles, atomically. This means that if you wait on two `AutoResetEvents`:

- `WaitAny` will never end up "latching" both events.
- `WaitAll` will never end up "latching" only one event.

`SignalAndWait` calls `Set` on one `WaitHandle`, and then calls `WaitOne` on another `WaitHandle`. The atomicity guarantee is that after signaling the first handle, it will jump to the head of the queue in waiting on the second handle: you can think of it as "swapping" one signal for another. You can use this method on a pair of `EventWaitHandle`s to set up two threads to rendezvous or "meet" at the same point in time. Either `AutoResetEvent` or `ManualResetEvent` will do the trick. The first thread executes the following:

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

whereas the second thread does the opposite:

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

Alternatives to WaitAll and SignalAndWait

`WaitAll` and `SignalAndWait` won't run in a single-threaded apartment. Fortunately, there are alternatives. In the case of `SignalAndWait`, it's rare that you need its atomicity guarantee: in our rendezvous example, for instance, you could simply call `Set` on the first wait handle, and then `WaitOne` on the other. In The Barrier Class, we'll explore yet another option for implementing a thread rendezvous.

In the case of `WaitAll`, an alternative in some situations is to use the `Parallel` class's `Invoke` method, which we'll cover in Part 5. (We'll also cover `Task`s and continuations, and see how `Task.ContinueWhenAny` provides an alternative to `WaitAny`.)

In all other scenarios, the answer is to take the low-level approach that solves all signaling problems: `Wait` and `Pulse`.

Synchronization Contexts

An alternative to locking manually is to lock *declaratively*. By deriving from ContextBoundObject and applying the Synchronization attribute, you instruct the CLR to apply locking automatically. For 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!

Further, 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. For 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

The Event-Based Asynchronous Pattern

The event-based asynchronous pattern (EAP) provides a simple means by which classes can offer multithreading capability without consumers needing to explicitly start or manage threads. It also provides the following features:

- A cooperative cancellation model
- The ability to safely update WPF or Windows Forms controls when the worker completes
- Forwarding of exceptions to the completion event

The EAP is just a pattern, so these features must be written by the implementer. Just a few classes in the Framework follow this pattern, most notably `BackgroundWorker` (which we'll cover next), and `WebClient` in `System.Net`. Essentially the pattern is this: a class offers a family of members that internally manage multithreading, similar to the following.

```
// These members are from the WebClient class:

public byte[] DownloadData (Uri address);    // Synchronous version
public void DownloadDataAsync (Uri address);
public void DownloadDataAsync (Uri address, object userToken);
public event DownloadDataCompletedEventHandler DownloadDataCompleted;

public void CancelAsync (object userState);  // Cancels an operation
public bool IsBusy { get; }                  // Indicates if still running
```

The `*Async` methods execute asynchronously: in other words, they start an operation on another thread and then return immediately to the caller. When the operation completes, the `*Completed` event fires—automatically calling `Invoke` if required by a WPF or Windows Forms application. This event passes back an event arguments object that contains:

- A flag indicating whether the operation was canceled (by the consumer calling `CancelAsync`)
- An `Error` object indicating an exception that was thrown (if any)
- The `userToken` object if supplied when calling the `Async` method

Here's how we can use `WebClient`'s EAP members to download a web page:

```
var wc = new WebClient();
wc.DownloadStringCompleted += (sender, args) =>
{
  if (args.Cancelled)
    Console.WriteLine ("Canceled");
  else if (args.Error != null)
    Console.WriteLine ("Exception: " + args.Error.Message);
  else
  {
    Console.WriteLine (args.Result.Length + " chars were downloaded");
    // We could update the UI from here...
  }
};
wc.DownloadStringAsync (new Uri ("http://www.linqpad.net"));  // Start it
```

A class following the EAP may offer additional groups of asynchronous methods. For instance:

```
public string DownloadString (Uri address);
public void DownloadStringAsync (Uri address);
public void DownloadStringAsync (Uri address, object userToken);
public event DownloadStringCompletedEventHandler DownloadStringCompleted;
```

However, these will share the same CancelAsync and IsBusy members. Therefore, only one asynchronous operation can happen at once.

> The EAP offers the *possibility* of economizing on threads, if its internal implementation follows the APM (this is described in Chapter 23 of C# 4.0 in a Nutshell).

We'll see in Part 5 how Tasks offer similar capabilities—including exception forwarding, continuations, cancellation tokens, and support for synchronization contexts. This makes *implementing* the EAP less attractive—except in simple cases where BackgroundWorker will do.

BackgroundWorker

BackgroundWorker is a helper class in the System.ComponentModel namespace for managing a worker thread. It can be considered a general-purpose implementation of the EAP, and provides the following features:

- A cooperative cancellation model
- The ability to safely update WPF or Windows Forms controls when the worker completes
- Forwarding of exceptions to the completion event
- A protocol for reporting progress
- An implementation of IComponent allowing it to be sited in Visual Studio's designer

BackgroundWorker uses the thread pool, which means you should never call Abort on a BackgroundWorker thread.

Using BackgroundWorker

Here are the minimum steps in using BackgroundWorker:

1. Instantiate BackgroundWorker and handle the DoWork event.
2. 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 has a RunWorkerCompleted event that fires after the DoWork event handler has done its job. Handling RunWorkerCompleted is not mandatory, but you usually do so in order to query any exception that was thrown in DoWork. Further, code within a RunWorkerCompleted event handler is able to update user interface controls without explicit marshaling; code within the DoWork event handler cannot.

To add support for progress reporting:

1. Set the WorkerReportsProgress property to true.
2. Periodically call ReportProgress from within the DoWork event handler with a “percentage complete” value, and optionally, a user-state object.
3. Handle the ProgressChanged event, querying its event argument’s ProgressPercentage property.
4. 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:

1. Set the WorkerSupportsCancellation property to true.
2. Periodically check the CancellationPending property from within the DoWork event handler. If it’s true, set the event argument’s Cancel property to true, and return. (The worker can also set Cancel and exit without CancellationPending being true if it decides that the job is too difficult and it can’t go on.)
3. Call CancelAsync to request cancellation.

Here’s an example that implements all the preceding features:

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

class Program
{
  static BackgroundWorker _bw;

  static void Main()
  {
    _bw = new BackgroundWorker
    {
      WorkerReportsProgress = true,
      WorkerSupportsCancellation = true
    };
    _bw.DoWork += bw_DoWork;
    _bw.ProgressChanged += bw_ProgressChanged;
    _bw.RunWorkerCompleted += bw_RunWorkerCompleted;

    _bw.RunWorkerAsync ("Hello to worker");

    Console.WriteLine ("Press Enter in the 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);      // Just for the demo... don't go sleeping
    }                           // for real in pooled threads!

    e.Result = 123;    // This gets passed to RunWorkerCompleted
  }
  static void bw_RunWorkerCompleted (object sender,
                                     RunWorkerCompletedEventArgs e)
  {
    if (e.Cancelled)
      Console.WriteLine ("You canceled!");
    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 + "%");
  }
}

// Output:
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 canceled!
```

Subclassing BackgroundWorker

Subclassing BackgroundWorker is an easy way to implement the EAP, in cases when you need to offer only one asynchronously executing method.

BackgroundWorker is not sealed and provides a virtual OnDoWork method, suggesting another pattern for its use. In writing a potentially long-running method, you could write an additional version returning a subclassed BackgroundWorker, preconfigured to perform the job concurrently. The consumer then needs to handle only the RunWorkerCompleted and ProgressChanged events. For instance, suppose we wrote a time-consuming method called GetFinancialTotals:

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

We could refactor it as follows:

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

public class FinancialWorker : BackgroundWorker
{
  public Dictionary <string,int> Result;   // You can add typed fields.
  public readonly int Foo, Bar;

  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, can be canceled, is friendly with WPF and Windows Forms applications, and handles exceptions well.

Interrupt and Abort

All blocking methods (such as `Sleep`, `Join`, `EndInvoke`, and `Wait`) block forever if the unblocking condition is never met and no timeout is specified. Occasionally, it can be useful to release a blocked thread prematurely; for instance, when ending an application. Two methods accomplish this:

- `Thread.Interrupt`
- `Thread.Abort`

The `Abort` method is also capable of ending a nonblocked thread—stuck, perhaps, in an infinite loop. `Abort` is occasionally useful in *niche* scenarios; `Interrupt` is almost never needed.

`Interrupt` and `Abort` can cause considerable trouble: it's precisely because they *seem* like obvious choices in solving a range of problems that it's worth examining their pitfalls.

Interrupt

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

```
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();
}

// Output:
Forcibly Woken!
```

Interrupting a thread does not cause the thread to end, unless 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 preemption 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 isn't designed to be interrupted (with appropriate cleanup code in `finally` blocks), objects could be left in an unusable state or resources incompletely released.

Moreover, `Interrupt` is unnecessary: if you are writing the code that blocks, you can achieve the same result more safely with a signaling construct—or Framework 4.0's cancellation tokens. And if you want to "unblock" someone else's code, `Abort` is nearly always more useful.

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 rethrown 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`.

An unhandled `ThreadAbortException` is one of only two types of exception that does not cause application shutdown (the other is `AppDomainUnloadException`).

The big difference between `Interrupt` and `Abort` is what happens when it's called on a thread that is not blocked. Whereas `Interrupt` waits until the thread next blocks before doing anything, `Abort` throws an exception on the thread right where it's executing (unmanaged code excepted). This is a problem because .NET Framework code might be aborted—code that is not abort-safe. For example, if an abort occurs while a `FileStream` is being constructed, it's possible that an unmanaged file handle will remain open until the application domain ends. This rules out using `Abort` in almost any nontrivial context.

For more detail on why Abort is unsafe, see Aborting Threads in Part 4.

There are two cases, though, where you can safely use `Abort`. One is if you are willing to tear down a thread's application domain after it is aborted. A good example of when you might do this is in writing a unit-testing framework. Another case where you can call `Abort` safely is on your own thread (because you know exactly where you are).

Aborting your own thread throws an “unswallowable” exception: one that gets rethrown after each catch block. ASP.NET does exactly this when you call `Redirect`.

LINQPad aborts threads when you cancel a runaway query. After aborting, it dismantles and re-creates the query’s application domain to avoid the potentially polluted state that could otherwise occur.

Safe Cancellation

As we saw in the preceding section, calling `Abort` on a thread is dangerous in most scenarios. The alternative, then, is to implement a *cooperative* pattern whereby the worker periodically checks a flag that indicates whether it should abort (like in `BackgroundWorker`). To cancel, the instigator simply sets the flag, and then waits for the worker to comply. This `BackgroundWorker` helper class implements such a flag-based cancellation pattern, and you easily implement one yourself.

The obvious disadvantage is that the worker method must be written explicitly to support cancellation. Nonetheless, this is one of the few safe cancellation patterns. To illustrate this pattern, we’ll first write a class to encapsulate the cancellation flag:

```
class RulyCanceler
{
  object _cancelLocker = new object();
  bool _cancelRequest;
  public bool IsCancellationRequested
  {
    get { lock (_cancelLocker) return _cancelRequest; }
  }

  public void Cancel() { lock (_cancelLocker) _cancelRequest = true; }

  public void ThrowIfCancellationRequested()
  {
    if (IsCancellationRequested) throw new OperationCanceledException();
  }
}
```

`OperationCanceledException` is a Framework type intended for just this purpose. Any exception class will work just as well, though.

We can use this as follows:

```
class Test
{
  static void Main()
  {
    var canceler = new RulyCanceler();
    new Thread (() => {
                        try { Work (canceler); }
                        catch (OperationCanceledException)
                        {
                          Console.WriteLine ("Canceled!");
                        }
                      }).Start();
    Thread.Sleep (1000);
    canceler.Cancel();               // Safely cancel worker.
  }
```

```
    static void Work (RulyCanceler c)
    {
      while (true)
      {
        c.ThrowIfCancellationRequested();
        // ...
        try      { OtherMethod (c); }
        finally  { /* any required cleanup */ }
      }
    }

    static void OtherMethod (RulyCanceler c)
    {
      // Do stuff...
      c.ThrowIfCancellationRequested();
    }
  }
```

We could simplify our example by eliminating the RulyCanceler class and adding the static boolean field _cancelRequest to the Test class. However, doing so would mean that if several threads called Work at once, setting _cancelRequest to true would cancel all workers. Our RulyCanceler class is therefore a useful abstraction. Its only inelegance is that when we look at the Work method's signature, the intention is unclear:

```
static void Work (RulyCanceler c)
```

Might the Work method itself intend to call Cancel on the RulyCanceler object? In this instance, the answer is no, so it would be nice if this could be enforced in the type system. Framework 4.0 provides *cancellation tokens* for this exact purpose.

Cancellation Tokens

Framework 4.0 provides two types that formalize the cooperative cancellation pattern that we just demonstrated: CancellationTokenSource and CancellationToken. The two types work in tandem:

- A CancellationTokenSource defines a Cancel method.
- A CancellationToken defines an IsCancellationRequested property and ThrowIfCancellationRequested method.

Together, these amount to a more sophisticated version of the RulyCanceler class in our previous example. But because the types are separate, you can isolate the ability to cancel from the ability to check the cancellation flag.

To use these types, first instantiate a CancellationTokenSource object:

```
var cancelSource = new CancellationTokenSource();
```

Then, pass its Token property into a method for which you'd like to support cancellation:

```
new Thread (() => Work (cancelSource.Token)).Start();
```

Here's how Work would be defined:

```
void Work (CancellationToken cancelToken)
{
  cancelToken.ThrowIfCancellationRequested();
  ...
}
```

When you want to cancel, simply call Cancel on cancelSource.

> CancellationToken is actually a struct, although you can treat it like a class. When implicitly copied, the copies behave identically and reference the original CancellationTokenSource.

The CancellationToken struct provides two additional useful members. The first is WaitHandle, which returns a wait handle that's signaled when the token is canceled. The second is Register, which lets you register a callback delegate that will be fired upon cancellation.

Cancellation tokens are used within the .NET Framework itself, most notably in the following classes:

- ManualResetEventSlim and SemaphoreSlim
- CountdownEvent
- Barrier
- BlockingCollection
- PLINQ and the Task Parallel Library

Most of these classes' use of cancellation tokens is in their Wait methods. For example, if you Wait on a ManualResetEventSlim and specify a cancellation token, another thread can Cancel its wait. This is much tidier and safer than calling Interrupt on the blocked thread.

Lazy Initialization

A common problem in threading is how to lazily initialize a shared field in a thread-safe fashion. The need arises when you have a field of a type that's expensive to construct:

```
class Foo
{
  public readonly Expensive Expensive = new Expensive();
  ...
}
class Expensive {  /* Suppose this is expensive to construct */  }
```

The problem with this code is that instantiating Foo incurs the performance cost of instantiating Expensive—whether or not the Expensive field is ever accessed. The obvious answer is to construct the instance *on demand*:

```
class Foo
{
  Expensive _expensive;
  public Expensive Expensive       // Lazily instantiate Expensive
  {
    get
    {
      if (_expensive == null) _expensive = new Expensive();
      return _expensive;
    }
  }
  ...
}
```

The question then arises, is this thread-safe? Aside from the fact that we're accessing _expensive outside a lock without a memory barrier, consider what would happen if two threads accessed this property at once. They could both satisfy the if statement's predicate and each thread end up with a *different* instance of Expensive. As this may lead to subtle errors, we would say, in general, that this code is not thread-safe.

The solution to the problem is to lock around checking and initializing the object:

```
Expensive _expensive;
readonly object _expenseLock = new object();

public Expensive Expensive
{
  get
  {
    lock (_expenseLock)
    {
      if (_expensive == null) _expensive = new Expensive();
      return _expensive;
    }
  }
}
```

Lazy<T>

Framework 4.0 provides a new class called Lazy<T> to help with lazy initialization. If instantiated with an argument of true, it implements the thread-safe initialization pattern just described.

> Lazy<T> actually implements a slightly more efficient version of this pattern, called *double-checked locking*. Double-checked locking performs an additional volatile read to avoid the cost of obtaining a lock if the object is already initialized.

To use Lazy<T>, instantiate the class with a value factory delegate that tells it how to initialize a new value, and the argument true. Then access its value via the Value property:

```
Lazy<Expensive> _expensive = new Lazy<Expensive>
  (() => new Expensive(), true);

public Expensive Expensive { get { return _expensive.Value; } }
```

If you pass false into Lazy<T>'s constructor, it implements the thread-unsafe lazy initialization pattern that we described at the start of this section—this makes sense when you want to use Lazy<T> in a single-threaded context.

LazyInitializer

LazyInitializer is a static class that works exactly like Lazy<T> except:

- Its functionality is exposed through a static method that operates directly on a field in your own type. This avoids a level of indirection, improving performance in cases where you need extreme optimization.
- It offers another mode of initialization that has multiple threads race to initialize.

To use LazyInitializer, call EnsureInitialized before accessing the field, passing a reference to the field and the factory delegate:

```
Expensive _expensive;
public Expensive Expensive
{
  get          // Implement double-checked locking
  {
    LazyInitializer.EnsureInitialized (ref _expensive,
                                      () => new Expensive());
    return _expensive;
  }
}
```

You can also pass in another argument to request that competing threads *race* to initialize. This sounds similar to our original thread-unsafe example, except that the first thread to finish always wins—and so you end up with only one instance. The advantage of this technique is that it's even faster (on multicores) than double-checked locking—because it can be implemented entirely without locks. This is an extreme optimization that you rarely need, and one that comes at a cost:

- It's slower when more threads race to initialize than you have cores.
- It potentially wastes CPU resources performing redundant initialization.
- The initialization logic must be thread-safe (in this case, it would be thread-unsafe if Expensive's constructor wrote to static fields, for instance).
- If the initializer instantiates an object requiring disposal, the "wasted" object won't get disposed without additional logic.

For reference, here's how double-checked locking is implemented:

```
volatile Expensive _expensive;
public Expensive Expensive
{
  get
  {
    if (_expensive == null)
    {
      var expensive = new Expensive();
      lock (_expenseLock) if (_expensive == null) _expensive = expensive;
    }
    return _expensive;
  }
}
```

And here’s how the race-to-initialize pattern is implemented:

```
volatile Expensive _expensive;
public Expensive Expensive
{
  get
  {
    if (_expensive == null)
    {
      var instance = new Expensive();
      Interlocked.CompareExchange (ref _expensive, instance, null);
    }
    return _expensive;
  }
}
```

Thread-Local Storage

Much of this article has focused on synchronization constructs and the issues arising from having threads concurrently access the same data. Sometimes, however, you want to keep data isolated, ensuring that each thread has a separate copy. Local variables achieve exactly this, but they are useful only with transient data.

The solution is *thread-local storage*. You might be hard-pressed to think of a requirement: data you’d want to keep isolated to a thread tends to be transient by nature. Its main application is for storing “out-of-band” data—that which supports the execution path’s infrastructure, such as messaging, transaction, and security tokens. Passing such data around in method parameters is extremely clumsy and alienates all but your own methods; storing such information in ordinary static fields means sharing it among all threads.

> Thread-local storage can also be useful in optimizing parallel code. It allows each thread to exclusively access its own version of a thread-unsafe object without needing locks—and without needing to reconstruct that object between method calls.

There are three ways to implement thread-local storage.

[ThreadStatic]

The easiest approach to thread-local storage is to mark a static field with the `ThreadStatic` attribute:

```
[ThreadStatic] static int _x;
```

Each thread then sees a separate copy of `_x`.

Unfortunately, `[ThreadStatic]` doesn’t work with instance fields (it simply does nothing); nor does it play well with field initializers—they execute only *once* on the thread that's running when the static constructor executes. If you need to work with instance fields—or start with a nondefault value—`ThreadLocal<T>` provides a better option.

ThreadLocal<T>

ThreadLocal<T> is new to Framework 4.0. It provides thread-local storage for both static and instance fields—and allows you to specify default values.

Here's how to create a ThreadLocal<int> with a default value of 3 for each thread:

```
static ThreadLocal<int> _x = new ThreadLocal<int> (() => 3);
```

You then use _x's Value property to get or set its thread-local value. A bonus of using ThreadLocal is that values are lazily evaluated: the factory function evaluates on the first call (for each thread).

ThreadLocal<T> and instance fields

ThreadLocal<T> is also useful with instance fields and captured local variables. For example, consider the problem of generating random numbers in a multithreaded environment. The Random class is not thread-safe, so we have to either lock around using Random (limiting concurrency) or generate a separate Random object for each thread. ThreadLocal<T> makes the latter easy:

```
var localRandom = new ThreadLocal<Random>(() => new Random());
Console.WriteLine (localRandom.Value.Next());
```

Our factory function for creating the Random object is a bit simplistic, though, in that Random's parameterless constructor relies on the system clock for a random number seed. This may be the same for two Random objects created within ~10 ms of each other. Here's one way to fix it:

```
var localRandom = new ThreadLocal<Random>
 ( () => new Random (Guid.NewGuid().GetHashCode()) );
```

We'll use this in Part 5 (see the parallel spellchecking example in "PLINQ").

GetData and SetData

The third approach is to use two methods in the Thread class: GetData and SetData. These store data in thread-specific "slots". 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. The same slot can be used across all threads and they'll still get separate values. Here's an example:

```
class Test
{
  // 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); }
  }
  ...
```

In this instance, we called Thread.GetNamedDataSlot, which creates a named slot—this allows sharing of that slot across the application. Alternatively, you can control a slot's scope yourself by instantiating a LocalDataStoreSlot explicitly—without providing any name:

```
class Test
{
  LocalDataStoreSlot _secSlot = new LocalDataStoreSlot();
  ...
```

Thread.FreeNamedDataSlot will release a named data slot across all threads, but only once all references to that LocalDataStoreSlot have dropped out of scope and have been garbage-collected. This ensures that threads don't get data slots pulled out from under their feet, as long as they keep a reference to the appropriate `LocalDataStoreSlot` object while the slot is needed.

Timers

If you need to execute some method repeatedly at regular intervals, the easiest way is with a *timer*. Timers are convenient and efficient in their use of memory and resources—compared with techniques such as the following:

```
new Thread (delegate() {
                         while (enabled)
                         {
                           DoSomeAction();
                           Thread.Sleep (TimeSpan.FromHours (24));
                         }
                       }).Start();
```

Not only does this permanently tie up a thread resource, but without additional coding, `DoSomeAction` will happen at a later time each day. Timers solve these problems.

The .NET Framework provides four timers. Two of these are general-purpose multithreaded timers:

- `System.Threading.Timer`
- `System.Timers.Timer`

The other two are special-purpose single-threaded timers:

- `System.Windows.Forms.Timer` (Windows Forms timer)
- `System.Windows.Threading.DispatcherTimer` (WPF timer)

The multithreaded timers are more powerful, accurate, and flexible; the single-threaded timers are safer and more convenient for running simple tasks that update Windows Forms controls or WPF elements.

Multithreaded Timers

`System.Threading.Timer` is the simplest multithreaded timer: it has just a constructor and two methods (a delight for minimalists, as well as book authors!). In the following example, a timer calls the `Tick` method, which writes "tick..." after five seconds have elapsed, and then every second after that, until the user presses Enter:

```
using System;
using System.Threading;

class Program
{
  static void Main()
  {
    // First interval = 5000ms; subsequent intervals = 1000ms
    Timer tmr = new Timer (Tick, "tick...", 5000, 1000);
    Console.ReadLine();
    tmr.Dispose();         // This both stops the timer and cleans up.
  }

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

You can change a timer's interval later by calling its `Change` method. If you want a timer to fire just once, specify `Timeout.Infinite` in the constructor's last argument.

The .NET Framework provides another timer class of the same name in the `System.Timers` namespace. This simply wraps the `System.Threading.Timer`, providing additional convenience while using the identical underlying engine. Here's a summary of its added features:

- A `Component` implementation, allowing it to be sited in Visual Studio's designer
- An `Interval` property instead of a `Change` method
- An `Elapsed` *event* instead of a callback delegate
- An `Enabled` property to start and stop 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 is `true`)
- A `SynchronizingObject` property with `Invoke` and `BeginInvoke` methods for safely calling methods on WPF elements and Windows Forms controls

Here's an example:

```
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();                    // Stop the timer
    Console.ReadLine();
    tmr.Start();                   // Restart the timer
    Console.ReadLine();
    tmr.Dispose();                 // Permanently stop the timer
  }

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

Multithreaded timers use the thread pool to allow a few threads to serve many timers. This means that the callback method or `Elapsed` event may fire on a different thread each time it is called. Furthermore, `Elapsed` always fires (approximately) on time—regardless of whether the previous `Elapsed` has finished executing. Hence, callbacks or event handlers must be thread-safe.

The precision of multithreaded timers depends on the operating system, and is typically in the 10–20 ms region. If you need greater precision, you can use native interop and call the Windows multimedia timer. This has precision down to 1 ms and it is defined in *winmm.dll*. First call `timeBeginPeriod` to inform the operating system that you need high timing precision, and then call `timeSetEvent` to start a multimedia timer. When you're done, call `timeKillEvent` to stop the timer and `timeEndPeriod` to inform the OS that you no longer need high timing precision. You can find complete examples on the Internet that use the multimedia timer by searching for the keywords *dllimport winmm.dll timesetevent*.

Single-Threaded Timers

The .NET Framework provides timers designed to eliminate thread-safety issues for WPF and Windows Forms applications:

- `System.Windows.Threading.DispatcherTimer` (WPF)
- `System.Windows.Forms.Timer` (Windows Forms)

> The single-threaded timers are not designed to work outside their respective environments. If you use a Windows Forms timer in a Windows Service application, for instance, the `Timer` event won't fire!

Both are like `System.Timers.Timer` in the members that they expose (`Interval`, `Tick`, `Start`, and `Stop`) and are used in a similar manner. However, they differ in how they work internally. Instead of using the thread pool to generate timer events, the WPF and Windows Forms timers rely on the message pumping mechanism of their underlying user interface model. This means that the `Tick` event always fires on the same thread that originally created the timer—which, in a normal application, is the same thread used to manage all user interface elements and controls. This has a number of benefits:

- You can forget about thread safety.
- A fresh `Tick` will never fire until the previous `Tick` has finished processing.
- You can update user interface elements and controls directly from `Tick` event handling code, without calling `Control.Invoke` or `Dispatcher.Invoke`.

It sounds too good to be true, until you realize that a program employing these timers is not really multithreaded—*there is no parallel execution.* One thread serves all timers—as well as the processing UI events. This brings us to the disadvantage of single-threaded timers:

- Unless the `Tick` event handler executes quickly, the user interface becomes unresponsive.

This makes the WPF and Windows Forms timers suitable for only small jobs, typically those that involve updating some aspect of the user interface (e.g., a clock or countdown display). Otherwise, you need a multithreaded timer.

In terms of precision, the single-threaded timers are similar to the multithreaded timers (tens of milliseconds), although they are typically less *accurate*, because they can be delayed while other user interface requests (or other timer events) are processed.

Part 4: Advanced Topics

Nonblocking Synchronization

Earlier, we said that the need for synchronization arises even in 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 of a context switch and the latency of being descheduled, which can be undesirable in highly concurrent and performance-critical scenarios. The .NET Framework's *nonblocking* synchronization constructs can perform simple operations without ever blocking, pausing, or waiting.

> Writing nonblocking or lock-free multithreaded code properly is tricky! Memory barriers, in particular, are easy to get wrong (the `volatile` keyword is even easier to get wrong). Think carefully whether you really need the performance benefits before dismissing ordinary locks. Remember that acquiring and releasing an uncontended lock takes as little as 20ns on a 2010-era desktop.

The nonblocking approaches also work across multiple processes. An example of where this might be useful is in reading and writing process-shared memory.

Memory Barriers and Volatility

Consider the following example:

```
class Foo
{
  int _answer;
  bool _complete;

  void A()
  {
    _answer = 123;
    _complete = true;
  }

  void B()
  {
    if (_complete) Console.WriteLine (_answer);
  }
}
```

If methods `A` and `B` ran concurrently on different threads, might it be possible for `B` to write "0"? The answer is yes—for the following reasons:

- The compiler, CLR, or CPU may *reorder* your program's instructions to improve efficiency.
- The compiler, CLR, or CPU may introduce caching optimizations such that assignments to variables won't be visible to other threads right away.

C# and the runtime are very careful to ensure that such optimizations don't break ordinary single-threaded code—or multithreaded code that makes proper use of locks. Outside of these scenarios, you must explicitly defeat these optimizations by creating *memory barriers* (also called *memory fences*) to limit the effects of instruction reordering and read/write caching.

Full fences

The simplest kind of memory barrier is a *full memory barrier* (*full fence*) which prevents any kind of instruction reordering or caching around that fence. Calling `Thread.MemoryBarrier` generates a full fence; we can fix our example by applying four full fences as follows:

```
class Foo
{
  int _answer;
  bool _complete;

  void A()
  {
    _answer = 123;
    Thread.MemoryBarrier();    // Barrier 1
    _complete = true;
    Thread.MemoryBarrier();    // Barrier 2
  }

  void B()
  {
    Thread.MemoryBarrier();    // Barrier 3
    if (_complete)
    {
      Thread.MemoryBarrier();       // Barrier 4
      Console.WriteLine (_answer);
    }
  }
}
```

Barriers 1 and 4 prevent this example from writing "0". Barriers 2 and 3 provide a *freshness* guarantee: they ensure that if B ran after A, reading `_complete` would evaluate to `true`.

A full fence takes around ten nanoseconds on a 2010-era desktop.

The following implicitly generate full fences:

- C#'s `lock` statement (`Monitor.Enter`/`Monitor.Exit`)
- All methods on the `Interlocked` class (we'll cover these soon)
- Asynchronous callbacks that use the thread pool—these include asynchronous delegates, APM callbacks, and `Task` continuations
- Setting and waiting on a signaling construct

Anything that relies on signaling, such as starting or waiting on a `Task`

By virtue of that last point, the following is thread-safe:

```
int x = 0;
Task t = Task.Factory.StartNew (() => x++);
t.Wait();
Console.WriteLine (x);    // 1
```

You don't necessarily need a full fence with every individual read or write. If we had three *answer* fields, our example would still need only four fences:

```
class Foo
{
  int _answer1, _answer2, _answer3;
  bool _complete;

  void A()
  {
    _answer1 = 1; _answer2 = 2; _answer3 = 3;
    Thread.MemoryBarrier();
    _complete = true;
    Thread.MemoryBarrier();
  }

  void B()
  {
    Thread.MemoryBarrier();
    if (_complete)
    {
      Thread.MemoryBarrier();
      Console.WriteLine (_answer1 + _answer2 + _answer3);
    }
  }
}
```

A good approach is to start by putting memory barriers before and after every instruction that reads or writes a shared field, and then strip away the ones that you don't need. If you're uncertain of any, leave them in. Or better: switch back to using locks!

Do We Really Need Locks and Barriers?

Working with *shared writable fields* without locks or fences is asking for trouble. There's a lot of misleading information on this topic—including the MSDN documentation which states that `MemoryBarrier` is required only on multiprocessor systems with weak memory ordering, such as a system employing multiple Itanium processors. We can demonstrate that memory barriers are important on ordinary Intel Core-2 and Pentium processors with the following short program. You'll need to run it with optimizations enabled and without a debugger (in Visual Studio, select Release Mode in the solution's configuration manager, and then start without debugging):

```
static void Main()
{
  bool complete = false;
  var t = new Thread (() =>
  {
    bool toggle = false;
    while (!complete) toggle = !toggle;
  });
  t.Start();
  Thread.Sleep (1000);
  complete = true;
  t.Join();        // Blocks indefinitely
}
```

This program *never terminates* because the `complete` variable is cached in a CPU register. Inserting a call to `Thread.MemoryBarrier` inside the `while` loop (or locking around reading `complete`) fixes the error.

The volatile keyword

Another (more advanced) way to solve this problem is to apply the volatile keyword to the _complete field:

```
volatile bool _complete;
```

The volatile keyword instructs the compiler to generate an *acquire-fence* on every read from that field, and a *release-fence* on every write to that field. An acquire-fence prevents other reads/writes from being moved *before* the fence; a release-fence prevents other reads/writes from being moved *after* the fence. These “half-fences” are faster than full fences because they give the runtime and hardware more scope for optimization.

> As it happens, Intel’s X86 and X64 processors always apply acquire-fences to reads and release-fences to writes—whether or not you use the volatile keyword—so this keyword has no effect on the *hardware* if you’re using these processors. However, volatile *does* have an effect on optimizations performed by the compiler and the CLR—as well as on 64-bit AMD and (to a greater extent) Itanium processors. This means that you cannot be more relaxed by virtue of your clients running a particular type of CPU.
>
> (And even if you *do* use volatile, you should still maintain a healthy sense of anxiety, as we’ll see shortly!)

The effect of applying volatile to fields can be summarized as follows:

First instruction	**Second instruction**	**Can they be swapped?**
Read	Read	No
Read	Write	No
Write	Write	No (The CLR ensures that write-write operations are never swapped, even without the volatile keyword)
Write	Read	**Yes!**

Notice that applying volatile doesn’t prevent a write followed by a read from being swapped, and this can create brainteasers. Joe Duffy illustrates the problem well with the following example: if Test1 and Test2 run simultaneously on different threads, it’s possible for a and b to both end up with a value of 0 (despite the use of volatile on both x and y):

```
class IfYouThinkYouUnderstandVolatile
{
  volatile int x, y;

  void Test1()        // Executed on one thread
  {
    x = 1;            // Volatile write (release-fence)
    int a = y;        // Volatile read (acquire-fence)
    ...
  }

  void Test2()        // Executed on another thread
  {
    y = 1;            // Volatile write (release-fence)
    int b = x;        // Volatile read (acquire-fence)
    ...
  }
}
```

> The MSDN documentation states that use of the volatile keyword ensures that the most up-to-date value is present in the field at all times. This is incorrect, since as we’ve seen, a write followed by a read *can* be reordered.

This presents a strong case for avoiding volatile: even if you understand the subtlety in this example, will other developers working on your code also understand it? A full fence between each of the two assignments in Test1 and Test2 (or a lock) solves the problem.

The volatile keyword is not supported with pass-by-reference arguments or captured local variables: in these cases you must use the VolatileRead and VolatileWrite methods.

VolatileRead and VolatileWrite

The static `VolatileRead` and `VolatileWrite` methods in the `Thread` class read/write a variable while enforcing (technically, a superset of) the guarantees made by the `volatile` keyword. Their implementations are relatively inefficient, though, in that they actually generate full fences. Here are their complete implementations for the integer type:

```
public static void VolatileWrite (ref int address, int value)
{
  MemoryBarrier(); address = value;
}

public static int VolatileRead (ref int address)
{
  int num = address; MemoryBarrier(); return num;
}
```

You can see from this that if you call `VolatileWrite` followed by `VolatileRead`, no barrier is generated in the middle: this enables the same brainteaser scenario that we saw earlier.

Memory barriers and locking

As we said earlier, `Monitor.Enter` and `Monitor.Exit` both generate full fences. So if we ignore a lock’s mutual exclusion guarantee, we could say that this:

```
lock (someField) { … }
```

is equivalent to this:

```
Thread.MemoryBarrier(); { … } Thread.MemoryBarrier();
```

Interlocked

Use of memory barriers is not always enough when reading or writing fields in lock-free code. Operations on 64-bit fields, increments, and decrements require the heavier approach of using the `Interlocked` helper class. `Interlocked` also provides the `Exchange` and `CompareExchange` methods, the latter enabling lock-free read-modify-write operations, with a little additional coding.

A statement is intrinsically *atomic* if it executes as a single indivisible instruction on the underlying processor. Strict atomicity precludes any possibility of preemption. A simple read or write on a field of 32 bits or less is always atomic. Operations on 64-bit fields are guaranteed to be atomic only in a 64-bit runtime environment, and statements that combine more than one read/write operation are never atomic:

```
class Atomicity
{
  static int _x, _y;
  static long _z;

  static void Test()
  {
    long myLocal;
    _x = 3;             // Atomic
    _z = 3;             // Nonatomic on 32-bit environs (_z is 64 bits)
    myLocal = _z;       // Nonatomic on 32-bit environs (_z is 64 bits)
    _y += _x;           // Nonatomic (read AND write operation)
    _x++;               // Nonatomic (read AND write operation)
  }
}
```

Reading and writing 64-bit fields is nonatomic on 32-bit environments because it requires two separate instructions: one for each 32-bit memory location. So, if thread X reads a 64-bit value while thread Y is updating it, thread X may end up with a bitwise combination of the old and new values (a *torn read*).

The compiler implements unary operators of the kind `x++` by reading a variable, processing it, and 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--; }
}
```

Putting aside the issue of memory barriers, you might expect that if 10 threads concurrently run `Go`, `_x` would end up as `0`. However, this is not guaranteed, because a *race condition* is possible whereby one thread preempts another in between retrieving `_x`'s current value, decrementing it, and writing it back (resulting in an out-of-date value being written).

Of course, you can address these issues by wrapping the nonatomic operations in a `lock` statement. Locking, in fact, simulates atomicity if consistently applied. The `Interlocked` class, however, provides an easier and faster solution for such simple 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):
    Console.WriteLine (Interlocked.CompareExchange (ref _sum,
                                                    123, 10);      // 123
  }
}
```

All of `Interlocked`'s methods generate a full fence. Therefore, fields that you access via `Interlocked` don't need additional fences—unless they're accessed in other places in your program without `Interlocked` or a `lock`.

`Interlocked`'s mathematical operations are restricted to `Increment`, `Decrement`, and `Add`. If you want to multiply—or perform any other calculation—you can do so in lock-free style by using the `CompareExchange` method (typically in conjunction with spin-waiting). We give an example in "SpinLock and SpinWait".

`Interlocked` works by making its need for atomicity known to the operating system and virtual machine.

`Interlocked`'s methods have a typical overhead of 10 ns—half that of an uncontended `lock`. Further, they can never suffer the additional cost of context switching due to blocking. The flip side is that using `Interlocked` within a loop with many iterations can be less efficient than obtaining a single lock *around* the loop (although `Interlocked` enables greater *concurrency*).

Signaling with 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 the static methods `Wait` and `Pulse` (and `PulseAll`). The principle is that you write the signaling logic yourself using custom flags and fields (enclosed in `lock` statements), and then introduce `Wait` and `Pulse` commands to prevent spinning. With just these methods and the `lock` statement, you can achieve the functionality of `AutoResetEvent`, `ManualResetEvent`, and `Semaphore`, as well as (with some caveats) `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.

`Wait` and `Pulse` signaling, however, has some disadvantages over event wait handles:

- `Wait`/`Pulse` cannot span application domains or processes on a computer.
- You must remember to protect all variables related to the signaling logic with locks.
- `Wait`/`Pulse` programs may confuse developers relying on Microsoft's documentation.

The documentation problem arises because it's not obvious how `Wait` and `Pulse` are supposed to be used, even when you've read up on how they work. `Wait` and `Pulse` also have a peculiar aversion to dabblers: they will seek out any holes in your understanding and then delight in tormenting you! Fortunately, there is a simple pattern of use that tames `Wait` and `Pulse`.

In terms of performance, calling `Pulse` takes around a hundred nanoseconds on a 2010-era desktop—about a third of the time it takes to call `Set` on a wait handle. The overhead for waiting on uncontended signal is entirely up to you—because you implement the logic yourself using ordinary fields and variables. In practice, this is very simple and amounts purely to the cost of taking a `lock`.

How to Use Wait and Pulse

Here's how to use `Wait` and `Pulse`:

1. Define a single field for use as the synchronization object, such as:

   ```
   readonly object _locker = new object();
   ```

2. Define field(s) for use in your custom blocking condition(s). For example:

   ```
   bool _go;  or:  int _semaphoreCount;
   ```

3. Whenever you want to block, include the following code:

   ```
   lock (_locker)
     while ( <blocking-condition> )
       Monitor.Wait (_locker);
   ```

4. Whenever you change (or potentially change) a blocking condition, include this code:

   ```
   lock (_locker)
   {
     < alter the field(s) or data that might
       impact the blocking condition(s) >
     Monitor.Pulse (_locker);   // or: Monitor.PulseAll (_locker)
   }
   ```

   (If you change a blocking condition *and* want to block, you can incorporate steps 3 and 4 in a single `lock` statement.)

This pattern allows any thread to wait at any time for any condition. Here's a simple example, where a worker thread waits until the `_go` field is set to `true`:

```
class SimpleWaitPulse
{
  static readonly object _locker = new object();
  static bool _go;

  static void Main()
  {                                // The new thread will block
    new Thread (Work).Start();     // because _go==false.

    Console.ReadLine();            // Wait for user to hit Enter

    lock (_locker)                 // Let's now wake up the thread by
    {                              // setting _go=true and pulsing.
      _go = true;
      Monitor.Pulse (_locker);
    }
  }

  static void Work()
  {
    lock (_locker)
      while (!_go)
        Monitor.Wait (_locker);    // Lock is released while we’re waiting

    Console.WriteLine ("Woken!!!");
  }
}

// Output
Woken!!!   (after pressing Enter)
```

For thread safety, we ensure that all shared fields are accessed within a lock. Hence, we add `lock` statements around both reading and updating the `_go` flag. This is essential (unless you’re willing to follow the nonblocking synchronization principles).

The `Work` method is where we block, waiting for the `_go` flag to become `true`. The `Monitor.Wait` method does the following, in order:

1. Releases the lock on `_locker`.
2. Blocks until `_locker` is “pulsed.”
3. Reacquires the lock on `_locker`. If the lock is contended, then it blocks until the lock is available.

> This means that despite appearances, *no* lock is held on the synchronization object while `Monitor.Wait` awaits a pulse:
>
> ```
> lock (_locker)
> {
>   while (!_go)
>     Monitor.Wait (_locker);  // _lock is released
>   // lock is regained
>   ...
> }
> ```

Execution then continues at the next statement. `Monitor.Wait` is designed for use within a `lock` statement; it throws an exception if called otherwise. The same goes for `Monitor.Pulse`.

In the `Main` method, we signal the worker by setting the `_go` flag (within a lock) and calling `Pulse`. As soon as we *release the lock*, the worker resumes execution, reiterating its `while` loop.

The `Pulse` and `PulseAll` methods release threads blocked on a `Wait` statement. `Pulse` releases a maximum of one thread; `PulseAll` releases them all. In our example, just one thread is blocked, so their effects are identical. If more than one thread is waiting, calling `PulseAll` is *usually* safest with our suggested pattern of use.

In order for Wait to communicate with Pulse or PulseAll, the synchronizing object (_locker, in our case) must be the same.

In our pattern, pulsing indicates that *something might have changed*, and that waiting threads should recheck their blocking conditions. In the Work method, this check is accomplished via the while loop. The *waiter* then decides whether to continue, *not the notifier*. If pulsing by itself is taken as instruction to continue, the Wait construct is stripped of any real value; you end up with an inferior version of an AutoResetEvent.

If we abandon our pattern, removing the while loop, the _go flag, and the ReadLine, we get a bare-bones Wait/Pulse example:

```
static void Main()
{
  new Thread (Work).Start();
  lock (_locker) Monitor.Pulse (_locker);
}
static void Work()
{
  lock (_locker) Monitor.Wait (_locker);
  Console.WriteLine ("Woken!!!");
}
```

It's not possible to display the output, because it's nondeterministic! A race ensues between the main thread and the worker. If Wait executes first, the signal works. If Pulse executes first, the *pulse is lost* and the worker remains forever stuck. This differs from the behavior of an AutoResetEvent, where its Set method has a memory or "latching" effect, so it is still effective if called before WaitOne.

Pulse has no latching effect because you're expected to write the latch yourself, using a "go" flag as we did before. This is what makes Wait and Pulse versatile: with a boolean flag, we can make it function as an AutoResetEvent; with an integer field, we can write a CountdownEvent Semaphore. With more complex data structures, we can go further and write such constructs as a producer/consumer queue.

Producer/Consumer Queue

Earlier, we described the concept of a producer/consumer queue, and how to write one with an AutoResetEvent. We're now going to write a more powerful version with Wait and Pulse. This time, we'll allow an arbitrary number of workers, each with its own thread. We'll keep track of the threads in an array:

```
Thread[] _workers;
```

This gives us the option of Joining those threads later when we shut down the queue.

Each worker thread will execute a method called Consume. We can create the threads and start them in a single loop as follows:

```
public PCQueue (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();
}
```

Rather than using a simple string to describe a task, we'll take the more flexible approach of using a delegate. We'll use the System.Action delegate in the .NET Framework, which is defined as follows:

```
public delegate void Action();
```

This delegate matches any parameterless method—rather like the ThreadStart delegate. We can still represent tasks that call method with parameters, though—by wrapping the call in an anonymous delegate or lambda expression:

```
Action myFirstTask = delegate      // Anonymous method
{
    Console.WriteLine ("foo");
};
```

```
Action mySecondTask = () => Console.WriteLine ("foo");    // Lambda expression
```

To represent a queue of tasks, we'll use the `Queue<T>` collection as we did before:

```
Queue<Action> _itemQ = new Queue<Action>();
```

Before going into the `EnqueueItem` and `Consume` methods, let's look first at the complete code:

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

public class PCQueue
{
  readonly object _locker = new object();
  Thread[] _workers;
  Queue<Action> _itemQ = new Queue<Action>();

  public PCQueue (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 Shutdown (bool waitForWorkers)
  {
    // Enqueue one null item per worker to make each exit.
    foreach (Thread worker in _workers)
      EnqueueItem (null);

    // Wait for workers to finish
    if (waitForWorkers)
      foreach (Thread worker in _workers)
        worker.Join();
  }

  public void EnqueueItem (Action item)
  {
    lock (_locker)
    {
      _itemQ.Enqueue (item);           // We must pulse because we're
      Monitor.Pulse (_locker);         // changing a blocking condition.
    }
  }

  void Consume()
  {
    while (true)                        // Keep consuming until
    {                                   // told otherwise.
      Action item;
      lock (_locker)
      {
        while (_itemQ.Count == 0) Monitor.Wait (_locker);
        item = _itemQ.Dequeue();
      }
      if (item == null) return;         // This signals our exit.
      item();                           // Execute item.
    }
  }
}
```

Again we have an exit strategy: enqueuing a null item signals a consumer to finish after completing any outstanding items (if we want it to quit sooner, we could use an independent "cancel" flag). Because we're supporting multiple consumers, we must enqueue one null item per consumer to completely shut down the queue.

Here's a `Main` method that starts a producer/consumer queue, specifying two concurrent consumer threads, and then enqueues 10 delegates to be shared among the two consumers:

```
static void Main()
{
  PCQueue q = new PCQueue (2);

  Console.WriteLine ("Enqueuing 10 items...");

  for (int i = 0; i < 10; i++)
  {
    int itemNumber = i;      // To avoid the captured variable trap
    q.EnqueueItem (() =>
    {
      Thread.Sleep (1000);          // Simulate time-consuming work
      Console.Write (" Task" + itemNumber);
    });
  }

  q.Shutdown (true);
  Console.WriteLine();
  Console.WriteLine ("Workers complete!");
}

// Output:
Enqueuing 10 items...
 Task1 Task0 (pause...) Task2 Task3 (pause...) Task4 Task5 (pause...)
 Task6 Task7 (pause...) Task8 Task9 (pause...)
Workers complete!
```

Let's look now at the `EnqueueItem` method:

```
public void EnqueueItem (Action item)
{
  lock (_locker)
  {
    _itemQ.Enqueue (item);           // We must pulse because we're
    Monitor.Pulse (_locker);         // changing a blocking condition.
  }
}
```

Because the queue is used by multiple threads, we must wrap all reads/writes in a lock. And because we're modifying a blocking condition (a consumer may kick into action as a result of enqueuing a task), we must pulse.

For the sake of efficiency, we call `Pulse` instead of `PulseAll` when enqueuing an item. This is because (at most) one consumer need be woken per item. If you had just one ice cream, you wouldn't wake a class of 30 sleeping children to queue for it; similarly, with 30 consumers, there's no benefit in waking them all—only to have 29 spin a useless iteration on their `while` loop before going back to sleep. We wouldn't break anything functionally, however, by replacing `Pulse` with `PulseAll`.

Let's now look at the `Consume` method, where a worker picks off and executes an item from the queue. We want the worker to block while there's nothing to do; in other words, when there are no items on the queue. Hence, our blocking condition is `_itemQ.Count == 0`:

```
Action item;
lock (_locker)
{
  while (_itemQ.Count == 0) Monitor.Wait (_locker);
  item = _itemQ.Dequeue();
}
if (item == null) return;         // This signals our exit
item();                           // Perform task.
```

The while loop exits when _itemQ.Count is nonzero, meaning that (at least) one item is outstanding. We must dequeue the item *before* releasing the lock—otherwise, the item may not be there for us to dequeue (the presence of other threads means things can change while you blink!). In particular, another consumer just finishing a previous job could sneak in and dequeue our item if we didn't hold onto the lock, and did something like this instead:

```
Action item;
lock (_locker)
{
  while (_itemQ.Count == 0) Monitor.Wait (_locker);
}
lock (_locker)    // WRONG!
{
  item = _itemQ.Dequeue();    // Item may not longer be there!
}
...
```

After the item is dequeued, we release the lock immediately. If we held on to it while performing the task, we would unnecessarily block other consumers and producers. We don't pulse after dequeuing, as no other consumer can ever unblock by there being fewer items on the queue.

Locking briefly is advantageous when using Wait and Pulse (and in general) as it avoids unnecessarily blocking other threads. Locking across many lines of code is fine—providing they all execute quickly. Remember that you're helped by Monitor.Wait's releasing the underlying lock while awaiting a pulse!

Wait Timeouts

You can specify a timeout when calling Wait, either in milliseconds or as a TimeSpan. The Wait method then returns false if it gave up because of a timeout. The timeout applies only to the *waiting phase*. Hence, a Wait with a timeout does the following:

1. Releases the underlying lock
2. Blocks until pulsed, *or the timeout elapses*
3. Reacquires the underlying lock

Specifying a timeout is like asking the CLR to give you a "virtual pulse" after the timeout interval. A timed-out Wait will still perform step 3 and reacquire the lock—just as if pulsed.

Should Wait block in step 3 (while reacquiring the lock), any timeout is ignored. This is rarely an issue, though, because other threads will lock only briefly in a well-designed Wait/Pulse application. So, reacquiring the lock should be a near-instant operation.

Wait timeouts have a useful application. 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—you 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 rechecked at the interval specified by the timeout, as well as when pulsed. The simpler the blocking condition, the smaller the timeout can be without creating inefficiency. In this case, we don't care whether the Wait was pulsed or timed out, so we ignore its return value.

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!

Monitor.Wait returns a bool value indicating whether it got a "real" pulse. If this returns false, it means that it timed out: sometimes it can be useful to log this or throw an exception if the timeout was unexpected.

Two-Way Signaling and Races

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 unblocked. Otherwise the pulse has no effect and is silently ignored.

Hence Pulse provides one-way communication: a pulsing thread (potentially) signals a waiting thread. There is no intrinsic acknowledgment mechanism: Pulse does not return a value indicating whether or not its pulse was received. Further, when a notifier pulses and releases its lock, there's no guarantee that an eligible waiter will kick into life *immediately*. There can be a small delay, at the discretion of the thread scheduler, during which time neither thread has a lock. This means that the pulser cannot know if or when a waiter resumes—unless you code something specifically (for instance with another flag and another reciprocal, Wait and Pulse).

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

To illustrate, let's say we want to signal a thread five times in a row:

```
class Race
{
  static readonly 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.PulseAll (_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 and demonstrates a *race condition*: the for loop in the main thread can freewheel right through its five iterations anytime the worker doesn't hold the lock, and 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 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.

We can solve this by adding a _ready flag to the class, controlled by the worker. The main thread then waits until the worker's ready before setting the _go flag.

This is analogous to a previous example that performed the same thing using two AutoResetEvents, except more extensible.

Here it is:

```
class Solved
{
  static readonly object _locker = new object();
  static bool _ready, _go;

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

    for (int i = 0; i < 5; i++)
      lock (_locker)
      {
        while (!_ready) Monitor.Wait (_locker);
        _ready = false;
        _go = true;
        Monitor.PulseAll (_locker);
      }
  }

  static void SaySomething()
  {
    for (int i = 0; i < 5; i++)
      lock (_locker)
      {
        _ready = true;
        Monitor.PulseAll (_locker);           // Remember that calling
        while (!_go) Monitor.Wait (_locker);  // Monitor.Wait releases
        go = false;                           // and reacquires the lock.
        Console.WriteLine ("Wassup?");
      }
  }
}

// Output:
Wassup? (repeated five times)
```

In the Main method, we clear the _ready flag, set the _go flag, and pulse, all in the same lock statement. The benefit of doing this is that it offers robustness if we later introduce a third thread into the equation. Imagine another thread trying to signal the worker at the same time. Our logic is watertight in this scenario; in effect, we're clearing _ready and setting _go, *atomically*.

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 the basis of a ManualResetEvent.

Let’s flesh out the complete code for a ManualResetEvent using Wait and Pulse:

```
readonly object _locker = new object();
bool _signal;

void WaitOne()
{
  lock (_locker)
  {
    while (!_signal) Monitor.Wait (_locker);
  }
}

void Set()
{
  lock (_locker) { _signal = true; Monitor.PulseAll (_locker); }
}

void Reset() { lock (_locker) _signal = false; }
```

We used PulseAll because there could be any number of blocked waiters.

Writing an AutoResetEvent is simply a matter of replacing the code in WaitOne with this:

```
lock (_locker)
{
  while (!_signal) Monitor.Wait (_locker);
  _signal = false;
}
```

and replacing PulseAll with Pulse in the Set method:

```
  lock (_locker) { _signal = true; Monitor.Pulse (_locker); }
```

> Use of PulseAll would forgo fairness in the queuing of backlogged waiters, because each call to PulseAll would result in the queue breaking and then re-forming.

Replacing _signal with an integer field would form the basis of a Semaphore.

Simulating the static methods that work across a set of wait handles is easy in simple scenarios. The equivalent of calling WaitAll is nothing more than a blocking condition that incorporates all the flags used in place of the wait handles:

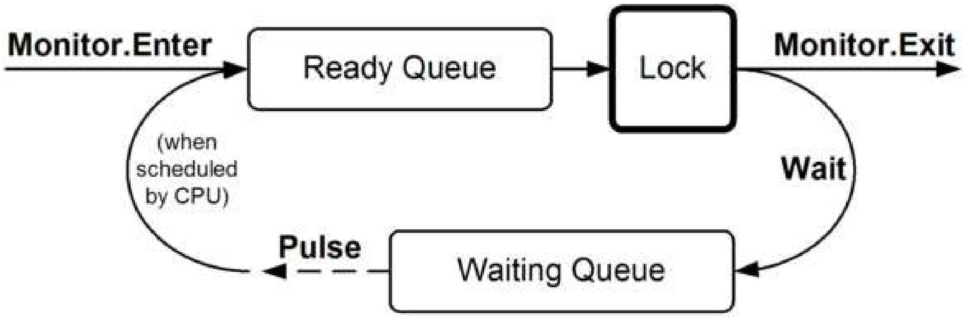
```
lock (_locker)
  while (!_flag1 && !_flag2 && !_flag3...)
    Monitor.Wait (_locker);
```

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

> If you have dozens of flags, this approach becomes less efficient because they must all share a single synchronizing object in order for the signaling to work atomically. This is where wait handles have an advantage.

Waiting queues and PulseAll

When more than one thread Waits upon the same object, 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).



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.

PulseAll releases the entire queue, or pool, of waiting threads. 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.

Writing a CountdownEvent

With Wait and Pulse, we can implement the essential functionality of a CountdownEvent as follows:

```
public class Countdown
{
  object _locker = new object ();
  int _value;

  public Countdown() { }
  public Countdown (int initialCount) { _value = initialCount; }

  public void Signal() { AddCount (-1); }

  public void AddCount (int amount)
  {
    lock (_locker)
    {
      _value += amount;
      if (_value <= 0) Monitor.PulseAll (_locker);
    }
  }

  public void Wait()
  {
    lock (_locker)
      while (_value > 0)
        Monitor.Wait (_locker);
  }
}
```

The pattern is like what we’ve seen previously, except that our blocking condition is based on an integer field.

Thread Rendezvous

We can use the Countdown class that we just wrote to rendezvous a pair of threads—as we did earlier with WaitHandle.SignalAndWait:

```
class Rendezvous
{
  static object _locker = new object();

  // In Framework 4.0, we could instead use the built-in CountdownEvent class.
  static Countdown _countdown = new Countdown(2);

  public 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));

    _countdown.Signal();
    _countdown.Wait();

    Console.Write ("Mate! ");
  }

  static void Mate (object delay)
  {
    Thread.Sleep ((int) delay);

    _countdown.Signal();
    _countdown.Wait();

    Console.Write ("Mate! ");
  }
}
```

In this example, each thread sleeps a random amount of time, and then waits for the other thread, resulting in them both writing "Mate" at (almost) the same time. This is called a *thread execution barrier* and can be extended to any number of threads (by adjusting the initial countdown value).

Thread execution barriers are useful when you want to keep several threads in step as they process a series of tasks. However, our current solution is limited in that we can't re-use the same Countdown object to rendezvous threads a second time—at least not without additional signaling constructs. To address this, Framework 4.0 provides a new class called Barrier.

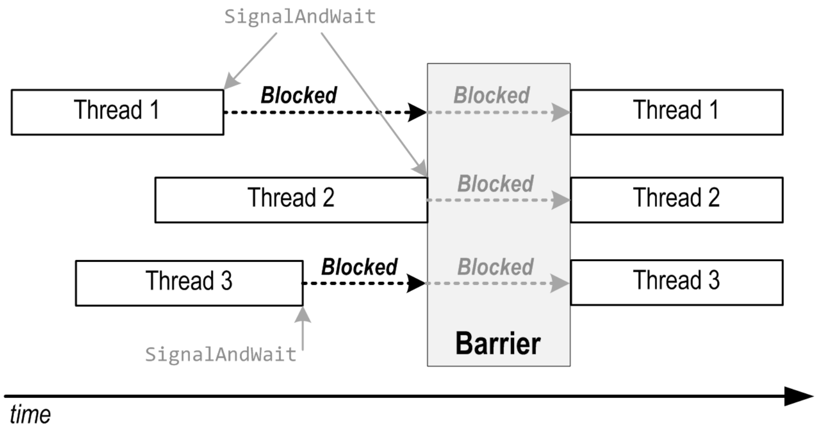
The Barrier Class

The Barrier class is a signaling construct new to Framework 4.0. It implements a *thread execution barrier*, which allows many threads to rendezvous at a point in time. The class is very fast and efficient, and is built upon Wait, Pulse, and spinlocks.

To use this class:

1. Instantiate it, specifying how many threads should partake in the rendezvous (you can change this later by calling AddParticipants/RemoveParticipants).
2. Have each thread call SignalAndWait when it wants to rendezvous.

Instantiating Barrier with a value of 3 causes SignalAndWait to block until that method has been called three times. But unlike a CountdownEvent, it then automatically starts over: calling SignalAndWait again blocks until called another three times. This allows you to keep several threads "in step" with each other as they process a series of tasks.



In the following example, each of three threads writes the numbers 0 through 4, while keeping in step with the other threads:

```
static Barrier _barrier = new Barrier (3);

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

static void Speak()
{
  for (int i = 0; i < 5; i++)
  {
    Console.Write (i + " ");
    _barrier.SignalAndWait();
  }
}

OUTPUT:  0 0 0 1 1 1 2 2 2 3 3 3 4 4 4
```

A really useful feature of `Barrier` is that you can also specify a *post-phase action* when constructing it. This is a delegate that runs after `SignalAndWait` has been called *n* times, but *before* the threads are unblocked. In our example, if we instantiate our barrier as follows:

```
static Barrier _barrier = new Barrier (3, barrier => Console.WriteLine());
```

then the output is:

```
0 0 0
1 1 1
2 2 2
3 3 3
4 4 4
```

A post-phase action can be useful for coalescing data from each of the worker threads. It doesn't have to worry about preemption, because all workers are blocked while it does its thing.

Reader/Writer Locks

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` was introduced in Framework 3.5 and is a replacement for the older “fat” `ReaderWriterLock` class. The latter is similar in functionality, but it is several times slower and has an inherent design fault in its mechanism for handling lock upgrades.
>
> When compared to an ordinary `lock` (`Monitor.Enter`/`Exit`), `ReaderWriterLockSlim` is twice as slow.

With both classes, there are two basic kinds of lock—a read lock and a write lock:

- A write lock is universally exclusive.
- 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 that 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 that 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 a simple lock. We can illustrate this by inserting the following line in 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 of 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; }
```

Upgradeable Locks and Recursion

Sometimes it's useful to swap a read lock for a write lock in a single atomic operation. For instance, suppose you want 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 (e.g., adding the same item) 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 whether the item is already present in the list).
3. Call `EnterWriteLock` (this converts the upgradeable lock to a write lock).
4. Perform write-based activities (e.g., add the item to the 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 upgradeable 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 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 create 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 be 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.

Suspend and Resume

A thread can be explicitly suspended and resumed via the deprecated methods `Thread.Suspend` and `Thread.Resume`. This mechanism is completely separate to that of blocking. 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 results 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 you can implement a simple synchronization mechanism—ith 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 determining whether 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 Wait and Pulse. This makes `Suspend` and `Resume` useless on all counts.

> The deprecated `Suspend` and `Resume` methods have two modes: dangerous and useless!

Aborting Threads

You can end a thread 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 aborted 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`, you 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 call to 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 see that `File.CreateText` 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.

This raises the question on how to go about writing an abort-friendly method. The most common workaround is not to abort another thread at all—but to implement a cooperative cancellation pattern, as described previously.

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`, you tear 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:

```
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:

```
class Program
{
  static void Main ()
  {
    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 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 is of 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`.

Part 5: Parallel Programming

Parallel Programming

In this section, we cover the multithreading APIs new to Framework 4.0 for leveraging multicore processors:

- Parallel LINQ or PLINQ
- The `Parallel` class
- The task parallelism constructs
- The *concurrent collections*
- `SpinLock` and `SpinWait`

These APIs are collectively known (loosely) as PFX (Parallel Framework). The `Parallel` class together with the task parallelism constructs is called the *Task Parallel Library* or TPL.

Framework 4.0 also adds a number of lower-level threading constructs that are aimed equally at traditional multithreading. We covered these previously:

- The low-latency signaling constructs (`SemaphoreSlim`, `ManualResetEventSlim`, `CountdownEvent` and `Barrier`)
- Cancellation tokens for cooperative cancellation
- The lazy initialization classes
- `ThreadLocal<T>`

You'll need to be comfortable with the fundamentals in Parts 1-4 before continuing—particularly locking and thread safety.

Why PFX?

In recent times, CPU clock speeds have stagnated and manufacturers have shifted their focus to increasing core counts. This is problematic for us as programmers because our standard single-threaded code will not automatically run faster as a result of those extra cores.

Leveraging multiple cores is easy for most server applications, where each thread can independently handle a separate client request, but is harder on the desktop—because it typically requires that you take your computationally intensive code and do the following:

1. *Partition* it into small chunks.
2. Execute those chunks in parallel via multithreading.
3. *Collate* the results as they become available, in a thread-safe and performant manner.

Although you can do all of this with the classic multithreading constructs, it's awkward—particularly the steps of partitioning and collating. A further problem is that the usual strategy of locking for thread safety causes a lot of contention when many threads work on the same data at once.

The PFX libraries have been designed specifically to help in these scenarios.

> Programming to leverage multicores or multiple processors is called *parallel programming*. This is a subset of the broader concept of multithreading.

PFX Concepts

There are two strategies for partitioning work among threads: *data parallelism* and *task parallelism*.

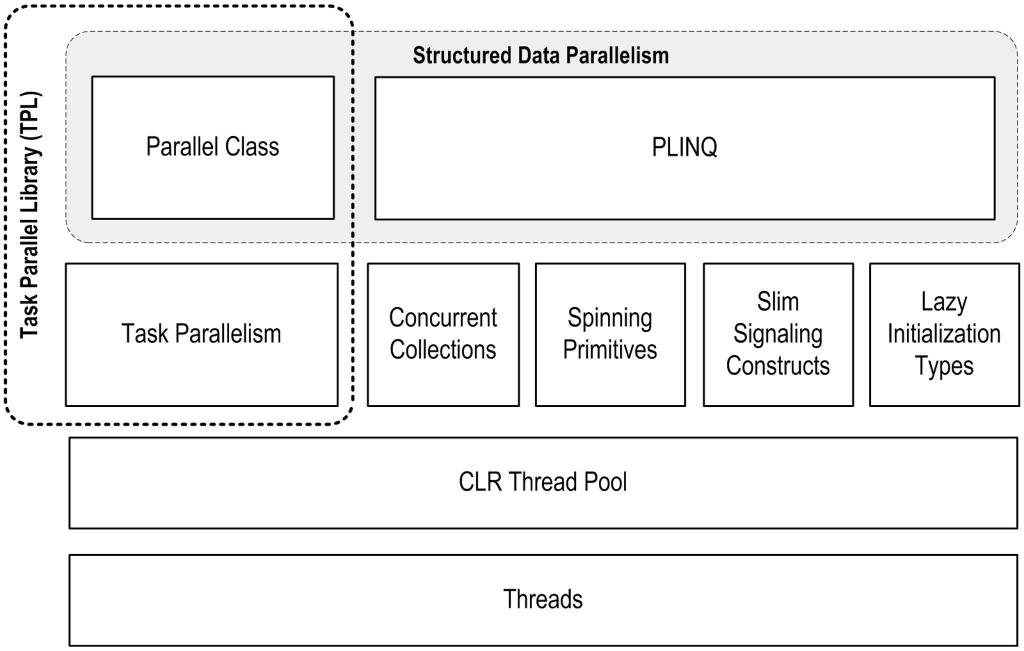
When a set of tasks must be performed on many data values, we can parallelize by having each thread perform the (same) set of tasks on a subset of values. This is called *data parallelism* because we are partitioning the *data* between threads. In contrast, with *task parallelism* we partition the *tasks*; in other words, we have each thread perform a different task.

In general, data parallelism is easier and scales better to highly parallel hardware, because it reduces or eliminates shared data (thereby reducing contention and thread-safety issues). Also, data parallelism leverages the fact that there are often more data values than discrete tasks, increasing the parallelism potential.

Data parallelism is also conducive to *structured parallelism*, which means that parallel work units start and finish in the same place in your program. In contrast, task parallelism tends to be unstructured, meaning that parallel work units may start and finish in places scattered across your program. Structured parallelism is simpler and less error-prone and allows you to farm the difficult job of partitioning and thread coordination (and even result collation) out to libraries.

PFX Components

PFX comprises two layers of functionality. The higher layer consists of two *structured data parallelism* APIs: PLINQ and the `Parallel` class. The lower layer contains the task parallelism classes—plus a set of additional constructs to help with parallel programming activities.


Task Parallel Library (TPL)
Structured Data Parallelism
Parallel Class
PLINQ
Task Parallelism
Concurrent Collections
Spinning Primitives
Slim Signaling Constructs
Lazy Initialization Types
CLR Thread Pool
Threads

PLINQ offers the richest functionality: it automates all the steps of parallelization—including partitioning the work into tasks, executing those tasks on threads, and collating the results into a single output sequence. It's called *declarative*—because you simply declare that you want to parallelize your work (which you structure as a LINQ query), and let the Framework take care of the implementation details. In contrast, the other approaches are *imperative*, in that you need to explicitly write code to partition or collate. In the case of the `Parallel` class, you must collate results yourself; with the task parallelism constructs, you must partition the work yourself, too:

	Partitions work	**Collates results**
PLINQ	Yes	Yes
The `Parallel` class	Yes	No
PFX's task parallelism	No	No

The concurrent collections and spinning primitives help you with lower-level parallel programming activities. These are important because PFX has been designed to work not only with today's hardware, but also with future generations of processors with far more cores. If you want to move a pile of chopped wood and you have 32 workers to do the job, the biggest challenge is moving the wood without the workers getting in each other's way. It's the same with dividing an algorithm among 32 cores: if ordinary locks are used to protect common resources, the resultant blocking may mean that only a fraction of those cores are ever actually busy at once. The concurrent collections are tuned specifically for highly concurrent access, with the focus on minimizing or eliminating blocking. PLINQ and the `Parallel` class themselves rely on the concurrent collections and on spinning primitives for efficient management of work.

PFX and Traditional Multithreading

A traditional multithreading scenario is one where multithreading can be of benefit even on a single-core machine—with no true *parallelization* taking place. We covered these previously: they include such tasks as maintaining a responsive user interface and downloading two web pages at once.

Some of the constructs that we'll cover in the parallel programming sections are also sometimes useful in traditional multithreading. In particular:

- PLINQ and the `Parallel` class are useful whenever you want to execute operations in parallel and then wait for them to complete (*structured* parallelism). This includes non-CPU-intensive tasks such as calling a web service.
- The task parallelism constructs are useful when you want to run some operation on a pooled thread, and also to manage a task's workflow through continuations and parent/child tasks.
- The concurrent collections are sometimes appropriate when you want a thread-safe queue, stack, or dictionary.
- `BlockingCollection` provides an easy means to implement producer/consumer structures.

When to Use PFX

The primary use case for PFX is *parallel programming*: leveraging multicore processors to speed up computationally intensive code.

A challenge in leveraging multicores is Amdahl's law, which states that the maximum performance improvement from parallelization is governed by the portion of the code that must execute sequentially. For instance, if only two-thirds of an algorithm's execution time is parallelizable, you can never exceed a threefold performance gain—even with an infinite number of cores.

So, before proceeding, it's worth verifying that the bottleneck is in parallelizable code. It's also worth considering whether your code *needs* to be computationally intensive—optimization is often the easiest and most effective approach. There's a trade-off, though, in that some optimization techniques can make it harder to parallelize code.

The easiest gains come with what's called *embarrassingly parallel* problems—where a job can be divided easily into tasks that execute efficiently on their own (structured parallelism is very well suited to such problems). Examples include many image processing tasks, ray tracing, and brute force approaches in mathematics or cryptography. An example of a nonembarrassingly parallel problem is implementing an optimized version of the quicksort algorithm—a good result takes some thought and may require unstructured parallelism.

PLINQ

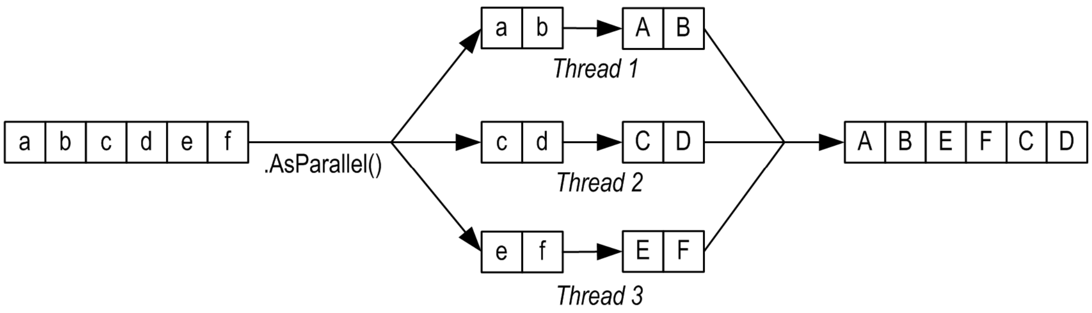
PLINQ automatically parallelizes local LINQ queries. PLINQ has the advantage of being easy to use in that it offloads the burden of both work partitioning and result collation to the Framework.

To use PLINQ, simply call `AsParallel()` on the input sequence and then continue the LINQ query as usual. The following query calculates the prime numbers between 3 and 100,000—making full use of all cores on the target machine:

```
// Calculate prime numbers using a simple (unoptimized) algorithm.

IEnumerable<int> numbers = Enumerable.Range (3, 100000-3);

var parallelQuery =
  from n in numbers.AsParallel()
  where Enumerable.Range (2, (int) Math.Sqrt (n)).All (i => n % i > 0)
  select n;

int[] primes = parallelQuery.ToArray();
```

AsParallel is an extension method in System.Linq.ParallelEnumerable. It wraps the input in a sequence based on ParallelQuery<TSource>, which causes the LINQ query operators that you subsequently call to bind to an alternate set of extension methods defined in ParallelEnumerable. These provide parallel implementations of each of the standard query operators. Essentially, they work by partitioning the input sequence into chunks that execute on different threads, collating the results back into a single output sequence for consumption:

ParallelEnumerable.Select



```
"abcdef".AsParallel().Select (c => char.ToUpper(c)).ToArray()
```

Calling AsSequential() unwraps a ParallelQuery sequence so that subsequent query operators bind to the standard query operators and execute sequentially. This is necessary before calling methods that have side effects or are not thread-safe.

For query operators that accept two input sequences (Join, GroupJoin, Concat, Union, Intersect, Except, and Zip), you must apply AsParallel() to both input sequences (otherwise, an exception is thrown). You don't, however, need to keep applying AsParallel to a query as it progresses, because PLINQ's query operators output another ParallelQuery sequence. In fact, calling AsParallel again introduces inefficiency in that it forces merging and repartitioning of the query:

```
mySequence.AsParallel()           // Wraps sequence in ParallelQuery<int>
          .Where (n => n > 100)   // Outputs another ParallelQuery<int>
          .AsParallel()           // Unnecessary - and inefficient!
          .Select (n => n * n)
```

Not all query operators can be effectively parallelized. For those that cannot, PLINQ implements the operator sequentially instead. PLINQ may also operate sequentially if it suspects that the overhead of parallelization will actually slow a particular query.

PLINQ is only for local collections: it doesn't work with LINQ to SQL or Entity Framework because in those cases the LINQ translates into SQL which then executes on a database server. However, you *can* use PLINQ to perform additional local querying on the result sets obtained from database queries.

If a PLINQ query throws an exception, it's rethrown as an AggregateException whose InnerExceptions property contains the real exception (or exceptions). See Working with AggregateException for details.

Why Isn't .AsParallel() the Default?

Given that AsParallel transparently parallelizes LINQ queries, the question arises, "Why didn't Microsoft simply parallelize the standard query operators and make PLINQ the default?"

There are a number of reasons for the *opt-in* approach. First, for PLINQ to be useful there has to be a reasonable amount of computationally intensive work for it to farm out to worker threads. Most LINQ to Objects queries execute very quickly, and not only would parallelization be unnecessary, but the overhead of partitioning, collating, and coordinating the extra threads may actually slow things down.

Additionally:

- The output of a PLINQ query (by default) may differ from a LINQ query with respect to element ordering.
- PLINQ wraps exceptions in an AggregateException (to handle the possibility of multiple exceptions being thrown).
- PLINQ will give unreliable results if the query invokes thread-unsafe methods.

Finally, PLINQ offers quite a few hooks for tuning and tweaking. Burdening the standard LINQ to Objects API with such nuances would add distraction.

Parallel Execution Ballistics

Like ordinary LINQ queries, PLINQ queries are lazily evaluated. This means that execution is triggered only when you begin consuming the results—typically via a foreach loop (although it may also be via a conversion operator such as ToArray or an operator that returns a single element or value).

As you enumerate the results, though, execution proceeds somewhat differently from that of an ordinary sequential query. A sequential query is powered entirely by the consumer in a "pull" fashion: each element from the input sequence is fetched exactly when required by the consumer. A parallel query ordinarily uses independent threads to fetch elements from the input sequence slightly *ahead* of when they're needed by the consumer (rather like a teleprompter for newsreaders, or an antiskip buffer in CD players). It then processes the elements in parallel through the query chain, holding the results in a small buffer so that they're ready for the consumer on demand. If the consumer pauses or breaks out of the enumeration early, the query processor also pauses or stops so as not to waste CPU time or memory.

> You can tweak PLINQ's buffering behavior by calling WithMergeOptions after AsParallel. The default value of AutoBuffered generally gives the best overall results. NotBuffered disables the buffer and is useful if you want to see results as soon as possible; FullyBuffered caches the entire result set before presenting it to the consumer (the OrderBy and Reverse operators naturally work this way, as do the element, aggregation, and conversion operators).

PLINQ and Ordering

A side effect of parallelizing the query operators is that when the results are collated, it's not necessarily in the same order that they were submitted, as illustrated in the previous diagram. In other words, LINQ's normal order-preservation guarantee for sequences no longer holds.

If you need order preservation, you can force it by calling AsOrdered() after AsParallel():

```
myCollection.AsParallel().AsOrdered()...
```

Calling AsOrdered incurs a performance hit with large numbers of elements because PLINQ must keep track of each element's original position.

You can negate the effect of AsOrdered later in a query by calling AsUnordered: this introduces a "random shuffle point" which allows the query to execute more efficiently from that point on. So if you wanted to preserve input-sequence ordering for just the first two query operators, you'd do this:

```
inputSequence.AsParallel().AsOrdered()
  .QueryOperator1()
  .QueryOperator2()
  .AsUnordered()       // From here on, ordering doesn't matter
  .QueryOperator3()
  ...
```

AsOrdered is not the default because for most queries, the original input ordering doesn't matter. In other words, if AsOrdered was the default, you'd have to apply AsUnordered to the majority of your parallel queries to get the best performance, which would be burdensome.

PLINQ Limitations

There are currently some practical limitations on what PLINQ can parallelize. These limitations may loosen with subsequent service packs and Framework versions.

The following query operators prevent a query from being parallelized, unless the source elements are in their original indexing position:

- Take, TakeWhile, Skip, and SkipWhile
- The indexed versions of Select, SelectMany, and ElementAt

Most query operators change the indexing position of elements (including those that remove elements, such as Where). This means that if you want to use the preceding operators, they'll usually need to be at the start of the query.

The following query operators are parallelizable, but use an expensive partitioning strategy that can sometimes be slower than sequential processing:

- Join, GroupBy, GroupJoin, Distinct, Union, Intersect, and Except

The Aggregate operator's *seeded* overloads in their standard incarnations are not parallelizable—PLINQ provides special overloads to deal with this.

All other operators are parallelizable, although use of these operators doesn't guarantee that your query will be parallelized. PLINQ may run your query sequentially if it suspects that the overhead of parallelization will slow down that particular query. You can override this behavior and force parallelism by calling the following after AsParallel():

```
.WithExecutionMode (ParallelExecutionMode.ForceParallelism)
```

Example: Parallel Spellchecker

Suppose we want to write a spellchecker that runs quickly with very large documents by leveraging all available cores. By formulating our algorithm into a LINQ query, we can very easily parallelize it.

The first step is to download a dictionary of English words into a HashSet for efficient lookup:

```
if (!File.Exists ("WordLookup.txt"))    // Contains about 150,000 words
  new WebClient().DownloadFile (
    "http://www.albahari.com/ispell/allwords.txt", "WordLookup.txt");

var wordLookup = new HashSet<string> (
  File.ReadAllLines ("WordLookup.txt"),
  StringComparer.InvariantCultureIgnoreCase);
```

We'll then use our word lookup to create a test "document" comprising an array of a million random words. After building the array, we'll introduce a couple of spelling mistakes:

```
var random = new Random();
string[] wordList = wordLookup.ToArray();

string[] wordsToTest = Enumerable.Range (0, 1000000)
  .Select (i => wordList [random.Next (0, wordList.Length)])
  .ToArray();

wordsToTest [12345] = "woozsh";     // Introduce a couple
wordsToTest [23456] = "wubsie";     // of spelling mistakes.
```

Now we can perform our parallel spellcheck by testing `wordsToTest` against `wordLookup`. PLINQ makes this very easy:

```
var query = wordsToTest
  .AsParallel()
  .Select  ((word, index) => new IndexedWord { Word=word, Index=index })
  .Where   (iword => !wordLookup.Contains (iword.Word))
  .OrderBy (iword => iword.Index);

foreach (var mistake in query)
  Console.WriteLine (mistake.Word + " - index = " + mistake.Index);

// OUTPUT:
// woozsh - index = 12345
// wubsie - index = 23456
```

`IndexedWord` is a custom struct that we define as follows:

```
struct IndexedWord { public string Word; public int Index; }
```

The `wordLookup.Contains` method in the predicate gives the query some “meat” and makes it worth parallelizing.

> We could simplify the query slightly by using an anonymous type instead of the `IndexedWord` struct. However, this would degrade performance because anonymous types (being classes and therefore reference types) incur the cost of heap-based allocation and subsequent garbage collection.
>
> The difference might not be enough to matter with sequential queries, but with parallel queries, favoring stack-based allocation can be quite advantageous. This is because stack-based allocation is highly parallelizable (as each thread has its own stack), whereas all threads must compete for the same heap—managed by a single memory manager and garbage collector.

Using ThreadLocal<T>

Let’s extend our example by parallelizing the creation of the random test-word list itself. We structured this as a LINQ query, so it should be easy. Here’s the sequential version:

```
string[] wordsToTest = Enumerable.Range (0, 1000000)
  .Select (i => wordList [random.Next (0, wordList.Length)])
  .ToArray();
```

Unfortunately, the call to `random.Next` is not thread-safe, so it’s not as simple as inserting `AsParallel()` into the query. A potential solution is to write a function that locks around `random.Next`; however, this would limit concurrency. The better option is to use `ThreadLocal<Random>` to create a separate `Random` object for each thread. We can then parallelize the query as follows:

```
var localRandom = new ThreadLocal<Random>
 ( () => new Random (Guid.NewGuid().GetHashCode()) );

string[] wordsToTest = Enumerable.Range (0, 1000000).AsParallel()
  .Select (i => wordList [localRandom.Value.Next (0, wordList.Length)])
  .ToArray();
```

In our factory function for instantiating a `Random` object, we pass in a `Guid`’s hashcode to ensure that if two `Random` objects are created within a short period of time, they’ll yield different random number sequences.

When to Use PLINQ

It’s tempting to search your existing applications for LINQ queries and experiment with parallelizing them. This is usually unproductive, because most problems for which LINQ is obviously the best solution tend to execute very quickly and so don’t benefit from parallelization. A better approach is to find a CPU-intensive bottleneck and then consider, “Can this be expressed as a LINQ query?” (A welcome side effect of such restructuring is that LINQ typically makes code smaller and more readable.)

PLINQ is well suited to embarrassingly parallel problems. It also works well for structured blocking tasks, such as calling several web services at once (see Calling Blocking or I/O-Intensive Functions).

PLINQ can be a poor choice for imaging, because collating millions of pixels into an output sequence creates a bottleneck. Instead, it’s better to write pixels directly to an array or unmanaged memory block and use the `Parallel` class or task parallelism to manage the multithreading. (It is possible, however, to defeat result collation using `ForAll`. Doing so makes sense if the algorithm naturally lends itself to LINQ.)

Functional Purity

Because PLINQ runs your query on parallel threads, you must be careful not to perform thread-unsafe operations. In particular, writing to variables is *side-effecting* and therefore thread-unsafe:

```
// The following query multiplies each element by its position.
// Given an input of Enumerable.Range(0,999), it should output squares.
int i = 0;
var query = from n in Enumerable.Range(0,999).AsParallel() select n * i++;
```

We could make incrementing `i` thread-safe by using locks or `Interlocked`, but the problem would still remain that `i` won’t necessarily correspond to the position of the input element. And adding `AsOrdered` to the query wouldn’t fix the latter problem, because `AsOrdered` ensures only that the elements are output in an order consistent with them having been processed sequentially—it doesn’t actually *process* them sequentially.

Instead, this query should be rewritten to use the indexed version of `Select`:

```
var query = Enumerable.Range(0,999).AsParallel().Select ((n, i) => n * i);
```

For best performance, any methods called from query operators should be thread-safe by virtue of not writing to fields or properties (non-side-effecting, or *functionally pure*). If they’re thread-safe by virtue of locking, the query’s parallelism potential will be limited—by the duration of the lock divided by the total time spent in that function.

Calling Blocking or I/O-Intensive Functions

Sometimes a query is long-running not because it’s CPU-intensive, but because it *waits* on something—such as a web page to download or some hardware to respond. PLINQ can effectively parallelize such queries, providing that you hint it by calling `WithDegreeOfParallelism` after `AsParallel`. For instance, suppose we want to ping six websites simultaneously. Rather than using clumsy asynchronous delegates or manually spinning up six threads, we can accomplish this effortlessly with a PLINQ query:

```
from site in new[]
{
  "www.albahari.com",
  "www.linqpad.net",
  "www.oreilly.com",
  "www.google.com",
  "www.takeonit.com",
  "stackoverflow.com"
}
.AsParallel().WithDegreeOfParallelism(6)
let p = new Ping().Send (site)
select new
{
  site,
  Result = p.Status,
  Time = p.RoundtripTime
}
```

WithDegreeOfParallelism forces PLINQ to run the specified number of tasks simultaneously. This is necessary when calling blocking functions such as Ping.Send because PLINQ otherwise assumes that the query is CPU-intensive and allocates tasks accordingly. On a two-core machine, for instance, PLINQ may default to running only two tasks at once, which is clearly undesirable in this situation.

> PLINQ typically serves each task with a thread, subject to allocation by the thread pool. You can accelerate the initial ramping up of threads by calling ThreadPool.SetMinThreads.

To give another example, suppose we were writing a surveillance system and wanted to repeatedly combine images from four security cameras into a single composite image for display on a CCTV. We'll represent a camera with the following class:

```
class Camera
{
  public readonly int CameraID;
  public Camera (int cameraID) { CameraID = cameraID; }

  // Get image from camera: return a simple string rather than an image
  public string GetNextFrame()
  {
    Thread.Sleep (123);       // Simulate time taken to get snapshot
    return "Frame from camera " + CameraID;
  }
}
```

To obtain a composite image, we must call GetNextFrame on each of four camera objects. Assuming the operation is I/O-bound, we can quadruple our frame rate with parallelization—even on a single-core machine. PLINQ makes this possible with minimal programming effort:

```
Camera[] cameras = Enumerable.Range (0, 4)    // Create 4 camera objects.
  .Select (i => new Camera (i))
  .ToArray();

while (true)
{
  string[] data = cameras
    .AsParallel().AsOrdered().WithDegreeOfParallelism (4)
    .Select (c => c.GetNextFrame()).ToArray();

  Console.WriteLine (string.Join (", ", data));   // Display data...
}
```

GetNextFrame is a blocking method, so we used WithDegreeOfParallelism to get the desired concurrency. In our example, the blocking happens when we call Sleep; in real life it would block because fetching an image from a camera is I/O- rather than CPU-intensive.

> Calling AsOrdered ensures the images are displayed in a consistent order. Because there are only four elements in the sequence, this would have a negligible effect on performance.

Changing the degree of parallelism

You can call WithDegreeOfParallelism only once within a PLINQ query. If you need to call it again, you must force merging and repartitioning of the query by calling AsParallel() again within the query:

```
"The Quick Brown Fox"
  .AsParallel().WithDegreeOfParallelism (2)
  .Where (c => !char.IsWhiteSpace (c))
  .AsParallel().WithDegreeOfParallelism (3)   // Forces Merge + Partition
  .Select (c => char.ToUpper (c))
```

Cancellation

Canceling a PLINQ query whose results you're consuming in a foreach loop is easy: simply break out of the foreach and the query will be automatically canceled as the enumerator is implicitly disposed.

For a query that terminates with a conversion, element, or aggregation operator, you can cancel it from another thread via a cancellation token. To insert a token, call WithCancellation after calling AsParallel, passing in the Token property of a CancellationTokenSource object. Another thread can then call Cancel on the token source, which throws an OperationCanceledException on the query's consumer:

```
IEnumerable<int> million = Enumerable.Range (3, 1000000);

var cancelSource = new CancellationTokenSource();

var primeNumberQuery =
  from n in million.AsParallel().WithCancellation (cancelSource.Token)
  where Enumerable.Range (2, (int) Math.Sqrt (n)).All (i => n % i > 0)
  select n;

new Thread (() => {
                    Thread.Sleep (100);      // Cancel query after
                    cancelSource.Cancel();   // 100 milliseconds.
                  }
           ).Start();
try
{
  // Start query running:
  int[] primes = primeNumberQuery.ToArray();
  // We'll never get here because the other thread will cancel us.
}
catch (OperationCanceledException)
{
  Console.WriteLine ("Query canceled");
}
```

PLINQ doesn't preemptively abort threads, because of the danger of doing so. Instead, upon cancellation it waits for each worker thread to finish with its current element before ending the query. This means that any external methods that the query calls will run to completion.

Optimizing PLINQ

Output-side optimization

One of PLINQ's advantages is that it conveniently collates the results from parallelized work into a single output sequence. Sometimes, though, all that you end up doing with that sequence is running some function once over each element:

```
foreach (int n in parallelQuery)
  DoSomething (n);
```

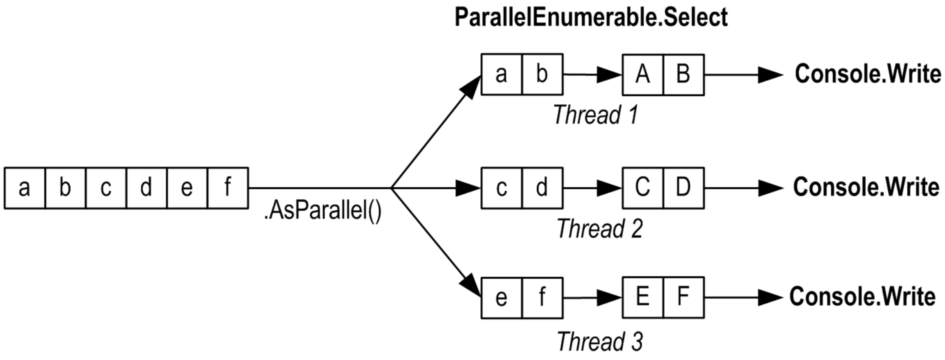
If this is the case—and you don't care about the order in which the elements are processed—you can improve efficiency with PLINQ's ForAll method.

The ForAll method runs a delegate over every output element of a ParallelQuery. It hooks right into PLINQ's internals, bypassing the steps of collating and enumerating the results.

> Collating and enumerating results is not a massively expensive operation, so the ForAll optimization yields the greatest gains when there are large numbers of quickly executing input elements.

To give a trivial example:

```
"abcdef".AsParallel().Select (c => char.ToUpper(c)).ForAll (Console.Write);
```



```
"abcdef".AsParallel().Select (c => char.ToUpper(c)).ForAll (Console.Write)
```

Input-side optimization

PLINQ has three partitioning strategies for assigning input elements to threads:

Strategy	**Element allocation**	**Relative performance**
Chunk partitioning	Dynamic	Average
Range partitioning	Static	Poor to excellent
Hash partitioning	Static	Poor

For query operators that require comparing elements (`GroupBy`, `Join`, `GroupJoin`, `Intersect`, `Except`, `Union`, and `Distinct`), you have no choice: PLINQ always uses *hash partitioning*. Hash partitioning is relatively inefficient in that it must precalculate the hashcode of every element (so that elements with identical hashcodes can be processed on the same thread). If you find this too slow, your only option is to call `AsSequential` to disable parallelization.

For all other query operators, you have a choice as to whether to use range or chunk partitioning. By default:

- If the input sequence is *indexable* (if it's an array or implements `IList<T>`), PLINQ chooses *range partitioning*.
- Otherwise, PLINQ chooses *chunk partitioning*.

In a nutshell, range partitioning is faster with long sequences for which every element takes a similar amount of CPU time to process. Otherwise, chunk partitioning is usually faster.

To force *range partitioning*:

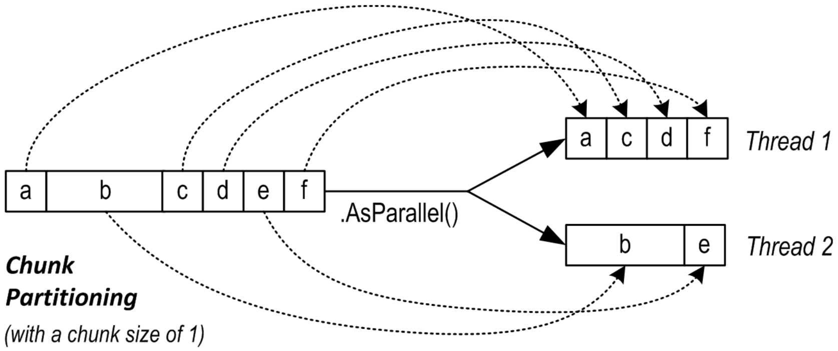
- If the query starts with `Enumerable.Range`, replace the latter with `ParallelEnumerable.Range`.
- Otherwise, simply call `ToList` or `ToArray` on the input sequence (obviously, this incurs a performance cost in itself which you should take into account).

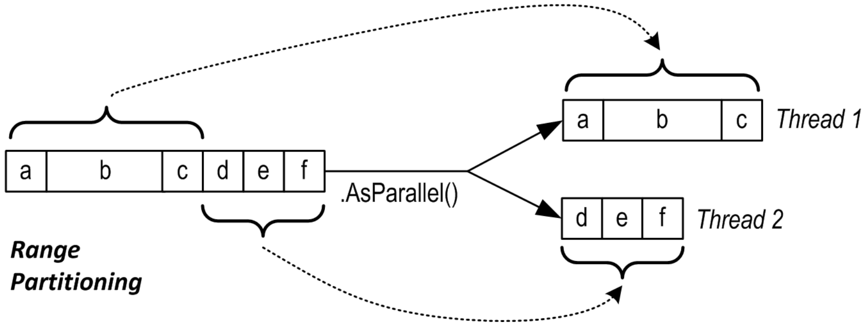
> `ParallelEnumerable.Range` is not simply a shortcut for calling `Enumerable.Range(…).AsParallel()`. It changes the performance of the query by activating range partitioning.

To force *chunk partitioning*, wrap the input sequence in a call to `Partitioner.Create` (in `System.Collection.Concurrent`) as follows:

```
int[] numbers = { 3, 4, 5, 6, 7, 8, 9 };
var parallelQuery =
  Partitioner.Create (numbers, true).AsParallel()
  .Where (...)
```

The second argument to `Partitioner.Create` indicates that you want to *load-balance* the query, which is another way of saying that you want chunk partitioning.





Chunk partitioning works by having each worker thread periodically grab small "chunks" of elements from the input sequence to process. PLINQ starts by allocating very small chunks (one or two elements at a time), then increases the chunk size as the query progresses: this ensures that small sequences are effectively parallelized and large sequences don't cause excessive round-tripping. If a worker happens to get "easy" elements (that process quickly) it will end up getting more chunks. This system keeps every thread equally busy (and the cores "balanced"); the only downside is that fetching elements from the shared input sequence requires synchronization (typically an exclusive lock)—and this can result in some overhead and contention.

Range partitioning bypasses the normal input-side enumeration and preallocates an equal number of elements to each worker, avoiding contention on the input sequence. But if some threads happen to get easy elements and finish early, they sit idle while the remaining threads continue working. Our earlier prime number calculator might perform poorly with range partitioning. An example of when range partitioning would do well is in calculating the sum of the square roots of the first 10 million integers:

```
ParallelEnumerable.Range (1, 10000000).Sum (i => Math.Sqrt (i))
```

`ParallelEnumerable.Range` returns a `ParallelQuery<T>`, so you don't need to subsequently call `AsParallel`.

> Range partitioning doesn't necessarily allocate element ranges in *contiguous* blocks—it might instead choose a "striping" strategy. For instance, if there are two workers, one worker might process odd-numbered elements while the other processes even-numbered elements. The `TakeWhile` operator is almost certain to trigger a striping strategy to avoid unnecessarily processing elements later in the sequence.

Parallelizing Custom Aggregations

PLINQ parallelizes the `Sum`, `Average`, `Min`, and `Max` operators efficiently without additional intervention. The `Aggregate` operator, though, presents special challenges for PLINQ.

If you're unfamiliar with this operator, you can think of `Aggregate` as a generalized version of `Sum`, `Average`, `Min`, and `Max`—in other words, an operator that lets you plug in a custom accumulation algorithm for implementing unusual aggregations. The following demonstrates how `Aggregate` can do the work of `Sum`:

```
int[] numbers = { 2, 3, 4 };
int sum = numbers.Aggregate (0, (total, n) => total + n);   // 9
```

The first argument to `Aggregate` is the *seed*, from which accumulation starts. The second argument is an expression to update the accumulated value, given a fresh element. You can optionally supply a third argument to project the final result value from the accumulated value.

> Most problems for which `Aggregate` has been designed can be solved as easily with a `foreach` loop—and with more familiar syntax. The advantage of `Aggregate` is precisely that large or complex aggregations can be parallelized declaratively with PLINQ.

Unseeded aggregations

You can omit the seed value when calling `Aggregate`, in which case the first element becomes the *implicit* seed, and aggregation proceeds from the second element. Here's the preceding example, *unseeded*:

```
int[] numbers = { 1, 2, 3 };
int sum = numbers.Aggregate ((total, n) => total + n);   // 6
```

This gives the same result as before, but we're actually doing a *different calculation*. Before, we were calculating 0+1+2+3; now we're calculating 1+2+3. We can better illustrate the difference by multiplying instead of adding:

```
int[] numbers = { 1, 2, 3 };
int x = numbers.Aggregate (0, (prod, n) => prod * n);   // 0*1*2*3 = 0
int y = numbers.Aggregate (   (prod, n) => prod * n);   //   1*2*3 = 6
```

As we'll see shortly, unseeded aggregations have the advantage of being parallelizable without requiring the use of special overloads. However, there is a trap with unseeded aggregations: the unseeded aggregation methods are intended for use with delegates that are *commutative* and *associative*. If used otherwise, the result is either *unintuitive* (with ordinary queries) or *nondeterministic* (in the case that you parallelize the query with PLINQ). For example, consider the following function:

```
(total, n) => total + n * n
```

This is neither commutative nor associative. (For example, 1+2*2 != 2+1*1). Let's see what happens when we use it to sum the square of the numbers 2, 3, and 4:

```
int[] numbers = { 2, 3, 4 };
int sum = numbers.Aggregate ((total, n) => total + n * n);    // 27
```

Instead of calculating:

```
2*2 + 3*3 + 4*4    // 29
```

it calculates:

```
2 + 3*3 + 4*4      // 27
```

We can fix this in a number of ways. First, we could include 0 as the first element:

```
int[] numbers = { 0, 2, 3, 4 };
```

Not only is this inelegant, but it will still give incorrect results if parallelized—because PLINQ leverages the function's assumed associativity by selecting *multiple* elements as seeds. To illustrate, if we denote our aggregation function as follows:

```
f(total, n) => total + n * n
```

then LINQ to Objects would calculate this:

```
f(f(f(0, 2),3),4)
```

whereas PLINQ may do this:

```
f(f(0,2),f(3,4))
```

with the following result:

```
First partition:   a = 0 + 2*2  (= 4)
Second partition:  b = 3 + 4*4  (= 19)
Final result:          a + b*b  (= 365)
OR EVEN:               b + a*a  (= 35)
```

There are two good solutions. The first is to turn this into a seeded aggregation—with zero as the seed. The only complication is that with PLINQ, we'd need to use a special overload in order for the query not to execute sequentially (as we'll see soon).

The second solution is to restructure the query such that the aggregation function is commutative and associative:

```
int sum = numbers.Select (n => n * n).Aggregate ((total, n) => total + n);
```

> Of course, in such simple scenarios you can (and should) use the `Sum` operator instead of `Aggregate`:
>
> ```
> int sum = numbers.Sum (n => n * n);
> ```
>
> You can actually go quite far just with `Sum` and `Average`. For instance, you can use `Average` to calculate a root-mean-square:
>
> ```
> Math.Sqrt (numbers.Average (n => n * n))
> ```
>
> and even standard deviation:
>
> ```
> double mean = numbers.Average();
> double sdev = Math.Sqrt (numbers.Average (n =>
>               {
>                 double dif = n - mean;
>                 return dif * dif;
>               }));
> ```
>
> Both are safe, efficient and fully parallelizable.

Parallelizing Aggregate

We just saw that for *unseeded* aggregations, the supplied delegate must be associative and commutative. PLINQ will give incorrect results if this rule is violated, because it draws *multiple seeds* from the input sequence in order to aggregate several partitions of the sequence simultaneously.

Explicitly seeded aggregations might seem like a safe option with PLINQ, but unfortunately these ordinarily execute sequentially because of the reliance on a single seed. To mitigate this, PLINQ provides another overload of `Aggregate` that lets you specify multiple seeds—or rather, a *seed factory function*. For each thread, it executes this function to generate a separate seed, which becomes a *thread-local* accumulator into which it locally aggregates elements.

You must also supply a function to indicate how to combine the local and main accumulators. Finally, this `Aggregate` overload (somewhat gratuitously) expects a delegate to perform any final transformation on the result (you can achieve this as easily by running some function on the result yourself afterward). So, here are the four delegates, in the order they are passed:

`seedFactory`

Returns a new local accumulator

`updateAccumulatorFunc`

Aggregates an element into a local accumulator

`combineAccumulatorFunc`

Combines a local accumulator with the main accumulator

`resultSelector`

Applies any final transformation on the end result

> In simple scenarios, you can specify a *seed value* instead of a seed factory. This tactic fails when the seed is a reference type that you wish to mutate, because the same instance will then be shared by each thread.

To give a very simple example, the following sums the values in a `numbers` array:

```
numbers.AsParallel().Aggregate (
 () => 0,                                      // seedFactory
  (localTotal, n) => localTotal + n,           // updateAccumulatorFunc
  (mainTot, localTot) => mainTot + localTot,   // combineAccumulatorFunc
  finalResult => finalResult)                  // resultSelector
```

This example is contrived in that we could get the same answer just as efficiently using simpler approaches (such as an unseeded aggregate, or better, the Sum operator). To give a more realistic example, suppose we wanted to calculate the frequency of each letter in the English alphabet in a given string. A simple sequential solution might look like this:

```
string text = "Let’s suppose this is a really long string";
var letterFrequencies = new int[26];
foreach (char c in text)
{
  int index = char.ToUpper (c) - 'A';
  if (index >= 0 && index <= 26) letterFrequencies [index]++;
};
```

> An example of when the input text might be very long is in gene sequencing. The “alphabet” would then consist of the letters *a*, *c*, *g*, and *t*.

To parallelize this, we could replace the foreach statement with a call to Parallel.ForEach (as we’ll cover in the following section), but this will leave us to deal with concurrency issues on the shared array. And locking around accessing that array would all but kill the potential for parallelization.

Aggregate offers a tidy solution. The accumulator, in this case, is an array just like the letterFrequencies array in our preceding example. Here’s a sequential version using Aggregate:

```
int[] result =
  text.Aggregate (
    new int[26],                // Create the "accumulator"
    (letterFrequencies, c) =>   // Aggregate a letter into the accumulator
    {
      int index = char.ToUpper (c) - 'A';
      if (index >= 0 && index <= 26) letterFrequencies [index]++;
      return letterFrequencies;
    });
```

And now the parallel version, using PLINQ’s special overload:

```
int[] result =
  text.AsParallel().Aggregate (
   () => new int[26],             // Create a new local accumulator

    (localFrequencies, c) =>       // Aggregate into the local accumulator
    {
      int index = char.ToUpper (c) - 'A';
      if (index >= 0 && index <= 26) localFrequencies [index]++;
      return localFrequencies;
    },
                                   // Aggregate local->main accumulator
    (mainFreq, localFreq) =>
      mainFreq.Zip (localFreq, (f1, f2) => f1 + f2).ToArray(),

    finalResult => finalResult     // Perform any final transformation
  );                               // on the end result.
```

Notice that the local accumulation function *mutates* the localFrequencies array. This ability to perform this optimization is important—and is legitimate because localFrequencies is local to each thread.

The Parallel Class

PFX provides a basic form of structured parallelism via three static methods in the `Parallel` class:

`Parallel.Invoke`

Executes an array of delegates in parallel

`Parallel.For`

Performs the parallel equivalent of a C# `for` loop

`Parallel.ForEach`

Performs the parallel equivalent of a C# `foreach` loop

All three methods block until all work is complete. As with PLINQ, after an unhandled exception, remaining workers are stopped after their current iteration and the exception (or exceptions) are thrown back to the caller—wrapped in an `AggregateException`.

Parallel.Invoke

`Parallel.Invoke` executes an array of `Action` delegates in parallel, and then waits for them to complete. The simplest version of the method is defined as follows:

```
public static void Invoke (params Action[] actions);
```

Here's how we can use `Parallel.Invoke` to download two web pages at once:

```
Parallel.Invoke (
 () => new WebClient().DownloadFile ("http://www.linqpad.net", "lp.html"),
 () => new WebClient().DownloadFile ("http://www.jaoo.dk", "jaoo.html"));
```

On the surface, this seems like a convenient shortcut for creating and waiting on two `Task` objects (or asynchronous delegates). But there's an important difference: `Parallel.Invoke` still works efficiently if you pass in an array of a million delegates. This is because it *partitions* large numbers of elements into batches which it assigns to a handful of underlying `Task`s—rather than creating a separate `Task` for each delegate.

As with all of `Parallel`'s methods, you're on your own when it comes to collating the results. This means you need to keep thread safety in mind. The following, for instance, is thread-unsafe:

```
var data = new List<string>();
Parallel.Invoke (
 () => data.Add (new WebClient().DownloadString ("http://www.foo.com")),
 () => data.Add (new WebClient().DownloadString ("http://www.far.com")));
```

Locking around adding to the list would resolve this, although locking would create a bottleneck if you had a much larger array of quickly executing delegates. A better solution is to use the thread-safe collections that we cover in C# 4.0 in a Nutshell—`ConcurrentBag` would be ideal in this case (see "Concurrent Collections").

`Parallel.Invoke` is also overloaded to accept a `ParallelOptions` object:

```
public static void Invoke (ParallelOptions options,
                           params Action[] actions);
```

With `ParallelOptions`, you can insert a cancellation token, limit the maximum concurrency, and specify a custom task scheduler. A cancellation token is relevant when you're executing (roughly) more tasks than you have cores: upon cancellation, any unstarted delegates will be abandoned. Any already-executing delegates will, however, continue to completion. See Cancellation for an example of how to use cancellation tokens.

Parallel.For and Parallel.ForEach

`Parallel.For` and `Parallel.ForEach` perform the equivalent of a C# `for` and `foreach` loop, but with each iteration executing in parallel instead of sequentially. Here are their (simplest) signatures:

```
public static ParallelLoopResult For (
  int fromInclusive, int toExclusive, Action<int> body)

public static ParallelLoopResult ForEach<TSource> (
  IEnumerable<TSource> source, Action<TSource> body)
```

The following sequential `for` loop:

```
for (int i = 0; i < 100; i++)
  Foo (i);
```

is parallelized like this:

```
Parallel.For (0, 100, i => Foo (i));
```

or more simply:

```
Parallel.For (0, 100, Foo);
```

And the following sequential `foreach`:

```
foreach (char c in "Hello, world")
  Foo (c);
```

is parallelized like this:

```
Parallel.ForEach ("Hello, world", Foo);
```

To give a practical example, if we import the `System.Security.Cryptography` namespace, we can generate six public/private key-pair strings in parallel as follows:

```
var keyPairs = new string[6];

Parallel.For (0, keyPairs.Length,
              i => keyPairs[i] = RSA.Create().ToXmlString (true));
```

As with `Parallel.Invoke`, we can feed `Parallel.For` and `Parallel.ForEach` a large number of work items and they'll be efficiently partitioned onto a few tasks.

> The latter query could also be done with PLINQ:
>
> ```
> string[] keyPairs =
>   ParallelEnumerable.Range (0, 6)
>   .Select (i => RSA.Create().ToXmlString (true))
>   .ToArray();
> ```

Outer versus inner loops

`Parallel.For` and `Parallel.ForEach` usually work best on outer rather than inner loops. This is because with the former, you're offering larger chunks of work to parallelize, diluting the management overhead. Parallelizing both inner and outer loops is usually unnecessary. In the following example, we'd typically need more than 100 cores to benefit from the inner parallelization:

```
Parallel.For (0, 100, i =>
{
  Parallel.For (0, 50, j => Foo (i, j));   // Sequential would be better
});                                        // for the inner loop.
```

Indexed Parallel.ForEach

Sometimes it's useful to know the loop iteration index. With a sequential `foreach`, it's easy:

```
int i = 0;
foreach (char c in "Hello, world")
  Console.WriteLine (c.ToString() + i++);
```

Incrementing a shared variable, however, is not thread-safe in a parallel context. You must instead use the following version of `ForEach`:

```
public static ParallelLoopResult ForEach<TSource> (
  IEnumerable<TSource> source, Action<TSource,ParallelLoopState,long> body)
```

We'll ignore `ParallelLoopState` (which we'll cover in the following section). For now, we're interested in `Action`'s third type parameter of type `long`, which indicates the loop index:

```
Parallel.ForEach ("Hello, world", (c, state, i) =>
{
   Console.WriteLine (c.ToString() + i);
});
```

To put this into a practical context, we'll revisit the spellchecker that we wrote with PLINQ. The following code loads up a dictionary along with an array of a million words to test:

```
if (!File.Exists ("WordLookup.txt"))    // Contains about 150,000 words
  new WebClient().DownloadFile (
    "http://www.albahari.com/ispell/allwords.txt", "WordLookup.txt");

var wordLookup = new HashSet<string> (
  File.ReadAllLines ("WordLookup.txt"),
  StringComparer.InvariantCultureIgnoreCase);

var random = new Random();
string[] wordList = wordLookup.ToArray();

string[] wordsToTest = Enumerable.Range (0, 1000000)
  .Select (i => wordList [random.Next (0, wordList.Length)])
  .ToArray();

wordsToTest [12345] = "woozsh";     // Introduce a couple
wordsToTest [23456] = "wubsie";     // of spelling mistakes.
```

We can perform the spellcheck on our `wordsToTest` array using the indexed version of `Parallel.ForEach` as follows:

```
var misspellings = new ConcurrentBag<Tuple<int,string>>();

Parallel.ForEach (wordsToTest, (word, state, i) =>
{
  if (!wordLookup.Contains (word))
    misspellings.Add (Tuple.Create ((int) i, word));
});
```

Notice that we had to collate the results into a thread-safe collection: having to do this is the disadvantage when compared to using PLINQ. The advantage over PLINQ is that we avoid the cost of applying an indexed `Select` query operator—which is less efficient than an indexed `ForEach`.

ParallelLoopState: Breaking early out of loops

Because the loop body in a parallel `For` or `ForEach` is a delegate, you can't exit the loop early with a `break` statement. Instead, you must call `Break` or `Stop` on a `ParallelLoopState` object:

```
public class ParallelLoopState
{
  public void Break();
  public void Stop();

  public bool IsExceptional { get; }
  public bool IsStopped { get; }
  public long? LowestBreakIteration { get; }
  public bool ShouldExitCurrentIteration { get; }
}
```

Obtaining a `ParallelLoopState` is easy: all versions of `For` and `ForEach` are overloaded to accept loop bodies of type `Action<TSource,ParallelLoopState>`. So, to parallelize this:

```
foreach (char c in "Hello, world")
  if (c == ',')
    break;
  else
    Console.Write (c);
```

do this:

```
Parallel.ForEach ("Hello, world", (c, loopState) =>
{
  if (c == ',')
    loopState.Break();
  else
    Console.Write (c);
});

// OUTPUT: Hlloe
```

You can see from the output that loop bodies may complete in a random order. Aside from this difference, calling `Break` yields *at least* the same elements as executing the loop sequentially: this example will always output *at least* the letters *H*, *e*, *l*, *l*, and *o* in some order. In contrast, calling `Stop` instead of `Break` forces all threads to finish right after their current iteration. In our example, calling `Stop` could give us a subset of the letters *H*, *e*, *l*, *l*, and *o* if another thread was lagging behind. Calling `Stop` is useful when you've found something that you're looking for—or when something has gone wrong and you won't be looking at the results.

> The `Parallel.For` and `Parallel.ForEach` methods return a `ParallelLoopResult` object that exposes properties called `IsCompleted` and `LowestBreakIteration`. These tell you whether the loop ran to completion, and if not, at what cycle the loop was broken.
>
> If `LowestBreakIteration` returns null, it means that you called `Stop` (rather than `Break`) on the loop.

If your loop body is long, you might want other threads to break partway through the method body in case of an early `Break` or `Stop`. You can do this by polling the `ShouldExitCurrentIteration` property at various places in your code; this property becomes true immediately after a `Stop`—or soon after a `Break`.

> `ShouldExitCurrentIteration` also becomes true after a cancellation request—or if an exception is thrown in the loop.

`IsExceptional` lets you know whether an exception has occurred on another thread. Any unhandled exception will cause the loop to stop after each thread's current iteration: to avoid this, you must explicitly handle exceptions in your code.

Optimization with local values

`Parallel.For` and `Parallel.ForEach` each offer a set of overloads that feature a generic type argument called `TLocal`. These overloads are designed to help you optimize the collation of data with iteration-intensive loops. The simplest is this:

```
public static ParallelLoopResult For <TLocal> (
  int fromInclusive,
  int toExclusive,
  Func <TLocal> localInit,
  Func <int, ParallelLoopState, TLocal, TLocal> body,
  Action <TLocal> localFinally);
```

These methods are rarely needed in practice because their target scenarios are covered mostly by PLINQ (which is fortunate because these overloads are somewhat intimidating!).

Essentially, the problem is this: suppose we want to sum the square roots of the numbers 1 through 10,000,000. Calculating 10 million square roots is easily parallelizable, but summing their values is troublesome because we must lock around updating the total:

```
object locker = new object();
double total = 0;
Parallel.For (1, 10000000,
              i => { lock (locker) total += Math.Sqrt (i); });
```

The gain from parallelization is more than offset by the cost of obtaining 10 million locks—plus the resultant blocking.

The reality, though, is that we don't actually *need* 10 million locks. Imagine a team of volunteers picking up a large volume of litter. If all workers shared a single trash can, the travel and contention would make the process extremely inefficient. The obvious solution is for each worker to have a private or “local” trash can, which is occasionally emptied into the main bin.

The TLocal versions of For and ForEach work in exactly this way. The volunteers are internal worker threads, and the *local value* represents a local trash can. In order for Parallel to do this job, you must feed it two additional delegates that indicate:

1. How to initialize a new local value
2. How to combine a local aggregation with the master value

Additionally, instead of the body delegate returning void, it should return the new aggregate for the local value. Here’s our example refactored:

```
object locker = new object();
double grandTotal = 0;

Parallel.For (1, 10000000,

  () => 0.0,                        // Initialize the local value.

  (i, state, localTotal) =>         // Body delegate. Notice that it
     localTotal + Math.Sqrt (i),    // returns the new local total.

  localTotal =>                                    // Add the local value
    { lock (locker) grandTotal += localTotal; }    // to the master value.
);
```

We must still lock, but only around aggregating the local value to the grand total. This makes the process dramatically more efficient.

As stated earlier, PLINQ is often a good fit in these scenarios. Our example could be parallelized with PLINQ simply like this:

```
ParallelEnumerable.Range (1, 10000000)
                  .Sum (i => Math.Sqrt (i))
```

(Notice that we used ParallelEnumerable to force *range partitioning*: this improves performance in this case because all numbers will take equally long to process.)

In more complex scenarios, you might use LINQ’s Aggregate operator instead of Sum. If you supplied a local seed factory, the situation would be somewhat analogous to providing a local value function with Parallel.For.

Task Parallelism

Task parallelism is the lowest-level approach to parallelization with PFX. The classes for working at this level are defined in the `System.Threading.Tasks` namespace and comprise the following:

Class	Purpose
`Task`	For managing a unit for work
`Task<TResult>`	For managing a unit for work with a return value
`TaskFactory`	For creating tasks
`TaskFactory<TResult>`	For creating tasks and continuations with the same return type
`TaskScheduler`	For managing the scheduling of tasks
`TaskCompletionSource`	For manually controlling a task's workflow

Essentially, a task is a lightweight object for managing a parallelizable unit of work. A task avoids the overhead of starting a dedicated thread by using the CLR's thread pool: this is the same thread pool used by `ThreadPool.QueueUserWorkItem`, tweaked in CLR 4.0 to work more efficiently with `Task`s (and more efficiently in general).

Tasks can be used whenever you want to execute something in parallel. However, they're *tuned* for leveraging multicores: in fact, the `Parallel` class and PLINQ are internally built on the task parallelism constructs.

Tasks do more than just provide an easy and efficient way into the thread pool. They also provide some powerful features for managing units of work, including the ability to:

- Tune a task's scheduling
- Establish a parent/child relationship when one task is started from another
- Implement cooperative cancellation
- Wait on a set of tasks—without a signaling construct
- Attach "continuation" task(s)
- Schedule a continuation based on multiple antecedent tasks
- Propagate exceptions to parents, continuations, and task consumers

Tasks also implement *local work queues*, an optimization that allows you to efficiently create many quickly executing child tasks without incurring the contention overhead that would otherwise arise with a single work queue.

The Task Parallel Library lets you create hundreds (or even thousands) of tasks with minimal overhead. But if you want to create millions of tasks, you'll need to partition those tasks into larger work units to maintain efficiency. The `Parallel` class and PLINQ do this automatically.

Visual Studio 2010 provides a new window for monitoring tasks (Debug | Window | Parallel Tasks). This is equivalent to the Threads window, but for tasks. The Parallel Stacks window also has a special mode for tasks.

Creating and Starting Tasks

As we described in Part 1 in our discussion of thread pooling, you can create and start a `Task` by calling `Task.Factory.StartNew`, passing in an `Action` delegate:

```
Task.Factory.StartNew (() => Console.WriteLine ("Hello from a task!"));
```

The generic version, `Task<TResult>` (a subclass of `Task`), lets you get data back from a task upon completion:

```
Task<string> task = Task.Factory.StartNew<string> (() =>    // Begin task
{
  using (var wc = new System.Net.WebClient())
    return wc.DownloadString ("http://www.linqpad.net");
});

RunSomeOtherMethod();         // We can do other work in parallel...

string result = task.Result;  // Wait for task to finish and fetch result.
```

`Task.Factory.StartNew` creates and starts a task in one step. You can decouple these operations by first instantiating a `Task` object, and then calling `Start`:

```
var task = new Task (() => Console.Write ("Hello"));
...
task.Start();
```

A task that you create in this manner can also be run synchronously (on the same thread) by calling `RunSynchronously` instead of `Start`.

> You can track a task's execution status via its `Status` property.

Specifying a state object

When instantiating a task or calling `Task.Factory.StartNew`, you can specify a *state* object, which is passed to the target method. This is useful should you want to call a method directly rather than using a lambda expression:

```
static void Main()
{
  var task = Task.Factory.StartNew (Greet, "Hello");
  task.Wait();  // Wait for task to complete.
}

static void Greet (object state) { Console.Write (state); }   // Hello
```

Given that we have lambda expressions in C#, we can put the *state* object to better use, which is to assign a meaningful name to the task. We can then use the `AsyncState` property to query its name:

```
static void Main()
{
  var task = Task.Factory.StartNew (state => Greet ("Hello"), "Greeting");
  Console.WriteLine (task.AsyncState);   // Greeting
  task.Wait();
}

static void Greet (string message) { Console.Write (message); }
```

> Visual Studio displays each task's `AsyncState` in the Parallel Tasks window, so having a meaningful name here can ease debugging considerably.

TaskCreationOptions

You can tune a task's execution by specifying a `TaskCreationOptions` enum when calling `StartNew` (or instantiating a `Task`). `TaskCreationOptions` is a flags enum with the following (combinable) values:

```
LongRunning
PreferFairness
AttachedToParent
```

`LongRunning` suggests to the scheduler to dedicate a thread to the task. This is beneficial for long-running tasks because they might otherwise "hog" the queue, and force short-running tasks to wait an unreasonable amount of time before being scheduled. `LongRunning` is also good for blocking tasks.

> The task queuing problem arises because the task scheduler ordinarily tries to keep just enough tasks active on threads at once to keep each CPU core busy. Not *oversubscribing* the CPU with too many active threads avoids the degradation in performance that would occur if the operating system was forced to perform a lot of expensive time slicing and context switching.

PreferFairness tells the scheduler to try to ensure that tasks are scheduled in the order they were started. It may ordinarily do otherwise, because it internally optimizes the scheduling of tasks using local work-stealing queues. This optimization is of practical benefit with very small (fine-grained) tasks.

AttachedToParent is for creating *child tasks*.

Child tasks

When one task starts another, you can optionally establish a parent-child relationship by specifying TaskCreationOptions.AttachedToParent:

```
Task parent = Task.Factory.StartNew (() =>
{
  Console.WriteLine ("I am a parent");

  Task.Factory.StartNew (() =>        // Detached task
  {
    Console.WriteLine ("I am detached");
  });

  Task.Factory.StartNew (() =>        // Child task
  {
    Console.WriteLine ("I am a child");
  }, TaskCreationOptions.AttachedToParent);
});
```

A child task is special in that when you wait for the *parent* task to complete, it waits for any children as well. This can be particularly useful when a child task is a continuation, as we'll see shortly.

Waiting on Tasks

You can explicitly wait for a task to complete in two ways:

- Calling its Wait method (optionally with a timeout)
- Accessing its Result property (in the case of Task<TResult>)

You can also wait on multiple tasks at once—via the static methods Task.WaitAll (waits for all the specified tasks to finish) and Task.WaitAny (waits for just one task to finish).

WaitAll is similar to waiting out each task in turn, but is more efficient in that it requires (at most) just one context switch. Also, if one or more of the tasks throw an unhandled exception, WaitAll still waits out every task—and then rethrows a single AggregateException that accumulates the exceptions from each faulted task. It's equivalent to doing this:

```
// Assume t1, t2 and t3 are tasks:
var exceptions = new List<Exception>();
try { t1.Wait(); } catch (AggregateException ex) { exceptions.Add (ex); }
try { t2.Wait(); } catch (AggregateException ex) { exceptions.Add (ex); }
try { t3.Wait(); } catch (AggregateException ex) { exceptions.Add (ex); }
if (exceptions.Count > 0) throw new AggregateException (exceptions);
```

Calling WaitAny is equivalent to waiting on a ManualResetEventSlim that's signaled by each task as it finishes.

As well as a timeout, you can also pass in a cancellation token to the Wait methods: this lets you cancel the wait—*not the task itself*.

Exception-Handling Tasks

When you wait for a task to complete (by calling its Wait method or accessing its Result property), any unhandled exceptions are conveniently rethrown to the caller, wrapped in an AggregateException object. This usually avoids the need to write code *within* task blocks to handle unexpected exceptions; instead we can do this:

```
int x = 0;
Task<int> calc = Task.Factory.StartNew (() => 7 / x);
try
{
  Console.WriteLine (calc.Result);
}
catch (AggregateException aex)
{
  Console.Write (aex.InnerException.Message);  // Attempted to divide by 0
}
```

You still need to exception-handle detached autonomous tasks (unparented tasks that are not waited upon) in order to prevent an unhandled exception taking down the application when the task drops out of scope and is garbage-collected (subject to the following note). The same applies for tasks waited upon with a timeout, because any exception thrown *after* the timeout interval will otherwise be "unhandled."

The static TaskScheduler.UnobservedTaskException event provides a final last resort for dealing with unhandled task exceptions. By handling this event, you can intercept task exceptions that would otherwise end the application—and provide your own logic for dealing with them.

For parented tasks, waiting on the parent implicitly waits on the children—and any child exceptions then bubble up:

```
TaskCreationOptions atp = TaskCreationOptions.AttachedToParent;
var parent = Task.Factory.StartNew (() =>
{
  Task.Factory.StartNew (() =>   // Child
  {
    Task.Factory.StartNew (() => { throw null; }, atp);   // Grandchild
  }, atp);
});

// The following call throws a NullReferenceException (wrapped
// in nested AggregateExceptions):
parent.Wait();
```

Interestingly, if you check a task's Exception property after it has thrown an exception, the act of reading that property will prevent the exception from subsequently taking down your application. The rationale is that PFX's designers don't want you *ignoring* exceptions—as long as you acknowledge them in some way, they won't punish you by terminating your program.

An unhandled exception on a task doesn't cause *immediate* application termination: instead, it's delayed until the garbage collector catches up with the task and calls its finalizer. Termination is delayed because it can't be known for certain that you don't plan to call Wait or check its Result or Exception property until the task is garbage-collected. This delay can sometimes mislead you as to the original source of the error (although Visual Studio's debugger can assist if you enable breaking on first-chance exceptions).

As we'll see soon, an alternative strategy for dealing with exceptions is with continuations.

Canceling Tasks

You can optionally pass in a cancellation token when starting a task. This lets you cancel tasks via the cooperative cancellation pattern described previously:

```
var cancelSource = new CancellationTokenSource();
CancellationToken token = cancelSource.Token;

Task task = Task.Factory.StartNew (() =>
{
  // Do some stuff...
  token.ThrowIfCancellationRequested();  // Check for cancellation request
  // Do some stuff...
}, token);
...
cancelSource.Cancel();
```

To detect a canceled task, catch an `AggregateException` and check the inner exception as follows:

```
try
{
  task.Wait();
}
catch (AggregateException ex)
{
  if (ex.InnerException is OperationCanceledException)
    Console.Write ("Task canceled!");
}
```

> If you want to explicitly throw an `OperationCanceledException` (rather than calling `token.ThrowIfCancellationRequested`), you must pass the cancellation token into `OperationCanceledException`’s constructor. If you fail to do this, the task won’t end up with a `TaskStatus.Canceled` status and won’t trigger `OnlyOnCanceled` continuations.

If the task is canceled before it has started, it won’t get scheduled—an `OperationCanceledException` will instead be thrown on the task immediately.

Because cancellation tokens are recognized by other APIs, you can pass them into other constructs and cancellations will propagate seamlessly:

```
var cancelSource = new CancellationTokenSource();
CancellationToken token = cancelSource.Token;

Task task = Task.Factory.StartNew (() =>
{
  // Pass our cancellation token into a PLINQ query:
  var query = someSequence.AsParallel().WithCancellation (token)...
  ... enumerate query ...
});
```

Calling `Cancel` on `cancelSource` in this example will cancel the PLINQ query, which will throw an `OperationCanceledException` on the task body, which will then cancel the task.

> The cancellation tokens that you can pass into methods such as `Wait` and `CancelAndWait` allow you to cancel the *wait* operation and not the task itself.

Continuations

Sometimes it's useful to start a task right after another one completes (or fails). The ContinueWith method on the Task class does exactly this:

```
Task task1 = Task.Factory.StartNew (() => Console.Write ("antecedant.."));
Task task2 = task1.ContinueWith (ant => Console.Write ("..continuation"));
```

As soon as task1 (the *antecedent*) finishes, fails, or is canceled, task2 (the *continuation*) automatically starts. (If task1 had completed before the second line of code ran, task2 would be scheduled to execute right away.) The ant argument passed to the continuation's lambda expression is a reference to the antecedent task.

Our example demonstrated the simplest kind of continuation, and is functionally similar to the following:

```
Task task = Task.Factory.StartNew (() =>
{
  Console.Write ("antecedent..");
  Console.Write ("..continuation");
});
```

The continuation-based approach, however, is more flexible in that you could first wait on task1, and then later wait on task2. This is particularly useful if task1 returns data.

Another (subtler) difference is that by default, antecedent and continuation tasks may execute on different threads. You can force them to execute on the same thread by specifying TaskContinuationOptions.ExecuteSynchronously when calling ContinueWith: this can improve performance in very fine-grained continuations by lessening indirection.

Continuations and Task<TResult>

Just like ordinary tasks, continuations can be of type Task<TResult> and return data. In the following example, we calculate Math.Sqrt(8*2) using a series of chained tasks and then write out the result:

```
Task.Factory.StartNew<int> (() => 8)
  .ContinueWith (ant => ant.Result * 2)
  .ContinueWith (ant => Math.Sqrt (ant.Result))
  .ContinueWith (ant => Console.WriteLine (ant.Result));   // 4
```

Our example is somewhat contrived for simplicity; in real life, these lambda expressions would call computationally intensive functions.

Continuations and exceptions

A continuation can find out if an exception was thrown by the antecedent via the antecedent task's Exception property. The following writes the details of a NullReferenceException to the console:

```
Task task1 = Task.Factory.StartNew (() => { throw null; });
Task task2 = task1.ContinueWith (ant => Console.Write (ant.Exception));
```

If an antecedent throws and the continuation fails to query the antecedent's Exception property (and the antecedent isn't otherwise waited upon), the exception is considered unhandled and the application dies (unless handled by TaskScheduler.UnobservedTaskException).

A safe pattern is to rethrow antecedent exceptions. As long as the continuation is Waited upon, the exception will be propagated and rethrown to the Waiter:

```
Task continuation = Task.Factory.StartNew     (()  => { throw null; })
                                .ContinueWith (ant =>
  {
    if (ant.Exception != null) throw ant.Exception;
    // Continue processing...
  });

continuation.Wait();    // Exception is now thrown back to caller.
```

Another way to deal with exceptions is to specify different continuations for exceptional versus nonexceptional outcomes. This is done with `TaskContinuationOptions`:

```
Task task1 = Task.Factory.StartNew (() => { throw null; });

Task error = task1.ContinueWith (ant => Console.Write (ant.Exception),
                                 TaskContinuationOptions.OnlyOnFaulted);

Task ok = task1.ContinueWith (ant => Console.Write ("Success!"),
                              TaskContinuationOptions.NotOnFaulted);
```

This pattern is particularly useful in conjunction with child tasks, as we'll see very soon.

The following extension method "swallows" a task's unhandled exceptions:

```
public static void IgnoreExceptions (this Task task)
{
  task.ContinueWith (t => { var ignore = t.Exception; },
    TaskContinuationOptions.OnlyOnFaulted);
}
```

(This could be improved by adding code to log the exception.) Here's how it would be used:

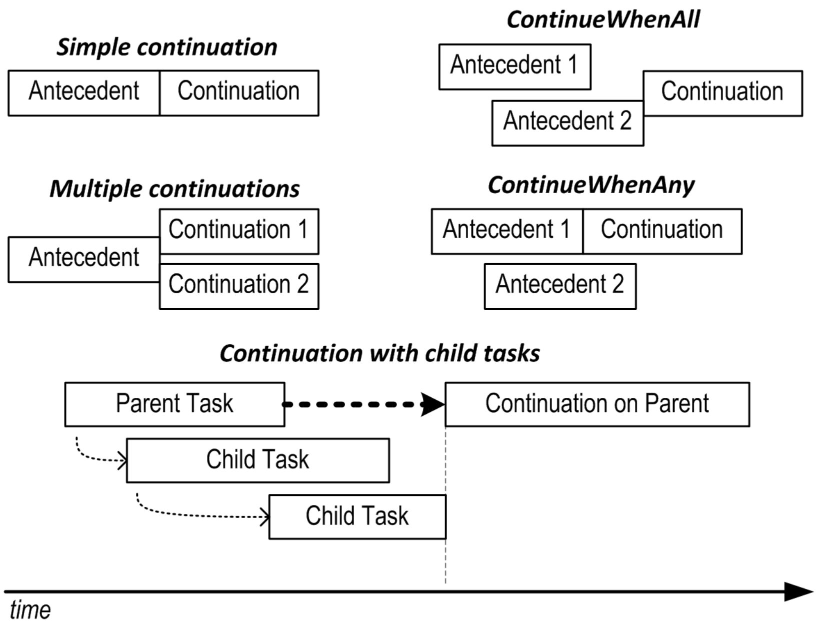
```
Task.Factory.StartNew (() => { throw null; }).IgnoreExceptions();
```

Continuations and child tasks

A powerful feature of continuations is that they kick off only when all child tasks have completed. At that point, any exceptions thrown by the children are marshaled to the continuation.

In the following example, we start three child tasks, each throwing a `NullReferenceException`. We then catch all of them in one fell swoop via a continuation on the parent:

```
TaskCreationOptions atp = TaskCreationOptions.AttachedToParent;
Task.Factory.StartNew (() =>
{
  Task.Factory.StartNew (() => { throw null; }, atp);
  Task.Factory.StartNew (() => { throw null; }, atp);
  Task.Factory.StartNew (() => { throw null; }, atp);
})
.ContinueWith (p => Console.WriteLine (p.Exception),
                    TaskContinuationOptions.OnlyOnFaulted);
```



Conditional continuations

By default, a continuation is scheduled *unconditionally*—whether the antecedent completes, throws an exception, or is canceled. You can alter this behavior via a set of (combinable) flags included within the `TaskContinuationOptions` enum. The three core flags that control conditional continuation are:

```
NotOnRanToCompletion = 0x10000,
NotOnFaulted = 0x20000,
NotOnCanceled = 0x40000,
```

These flags are subtractive in the sense that the more you apply, the less likely the continuation is to execute. For convenience, there are also the following precombined values:

```
OnlyOnRanToCompletion = NotOnFaulted | NotOnCanceled,
OnlyOnFaulted = NotOnRanToCompletion | NotOnCanceled,
OnlyOnCanceled = NotOnRanToCompletion | NotOnFaulted
```

(Combining all the `Not*` flags [`NotOnRanToCompletion`, `NotOnFaulted`, `NotOnCanceled`] is nonsensical, as it would result in the continuation always being canceled.)

"RanToCompletion" means the antecedent succeeded—without cancellation or unhandled exceptions.

"Faulted" means an unhandled exception was thrown on the antecedent.

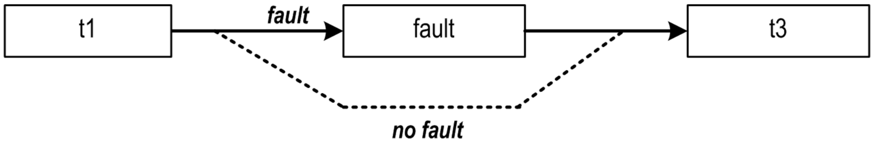
"Canceled" means one of two things:

- The antecedent was canceled via its cancellation token. In other words, an `OperationCanceledException` was thrown on the antecedent—whose `CancellationToken` property matched that passed to the antecedent when it was started.
- The antecedent was implicitly canceled because *it* didn't satisfy a conditional continuation predicate.

It's essential to grasp that when a continuation doesn't execute by virtue of these flags, the continuation is not forgotten or abandoned—it's *canceled*. This means that any continuations on the continuation itself *will then run*—unless you predicate them with `NotOnCanceled`. For example, consider this:

```
Task t1 = Task.Factory.StartNew (...);

Task fault = t1.ContinueWith (ant => Console.WriteLine ("fault"),
                              TaskContinuationOptions.OnlyOnFaulted);

Task t3 = fault.ContinueWith (ant => Console.WriteLine ("t3"));
```

As it stands, `t3` will always get scheduled—even if `t1` doesn't throw an exception. This is because if `t1` succeeds, the `fault` task will be *canceled*, and with no continuation restrictions placed on `t3`, `t3` will then execute unconditionally.



If we want `t3` to execute only if `fault` actually runs, we must instead do this:

```
Task t3 = fault.ContinueWith (ant => Console.WriteLine ("t3"),
                              TaskContinuationOptions.NotOnCanceled);
```

(Alternatively, we could specify `OnlyOnRanToCompletion`; the difference is that `t3` would not then execute if an exception was thrown within `fault`.)

Continuations with multiple antecedents

Another useful feature of continuations is that you can schedule them to execute based on the completion of multiple antecedents. `ContinueWhenAll` schedules execution when all antecedents have completed; `ContinueWhenAny` schedules execution when one antecedent completes. Both methods are defined in the `TaskFactory` class:

```
var task1 = Task.Factory.StartNew (() => Console.Write ("X"));
var task2 = Task.Factory.StartNew (() => Console.Write ("Y"));
var continuation = Task.Factory.ContinueWhenAll (
  new[] { task1, task2 }, tasks => Console.WriteLine ("Done"));
```

This writes “Done” after writing “XY” or “YX”. The `tasks` argument in the lambda expression gives you access to the array of completed tasks, which is useful when the antecedents return data. The following example adds together numbers returned from two antecedent tasks:

```
// task1 and task2 would call complex functions in real life:
Task<int> task1 = Task.Factory.StartNew (() => 123);
Task<int> task2 = Task.Factory.StartNew (() => 456);

Task<int> task3 = Task<int>.Factory.ContinueWhenAll (
  new[] { task1, task2 }, tasks => tasks.Sum (t => t.Result));

Console.WriteLine (task3.Result);           // 579
```

We’ve included the `<int>` type argument in our call to `Task.Factory` in this example to clarify that we’re obtaining a generic task factory. The type argument is unnecessary, though, as it will be inferred by the compiler.

Multiple continuations on a single antecedent

Calling `ContinueWith` more than once on the same task creates multiple continuations on a single antecedent. When the antecedent finishes, all continuations will start together (unless you specify `TaskContinuationOptions.ExecuteSynchronously`, in which case the continuations will execute sequentially).

The following waits for one second, and then writes either “XY” or “YX”:

```
var t = Task.Factory.StartNew (() => Thread.Sleep (1000));
t.ContinueWith (ant => Console.Write ("X"));
t.ContinueWith (ant => Console.Write ("Y"));
```

Task Schedulers and UIs

A *task scheduler* allocates tasks to threads. All tasks are associated with a task scheduler, which is represented by the abstract `TaskScheduler` class. The Framework provides two concrete implementations: the *default scheduler* that works in tandem with the CLR thread pool, and the *synchronization context scheduler*. The latter is designed (primarily) to help you with the threading model of WPF and Windows Forms, which requires that UI elements and controls are accessed only from the thread that created them. For example, suppose we wanted to fetch some data from a web service in the background, and then update a WPF label called `lblResult` with its result. We can divide this into two tasks:

1. Call a method to get data from the web service (antecedent task).
2. Update `lblResult` with the results (continuation task).

If, for a continuation task, we specify the *synchronization context scheduler* obtained when the window was constructed, we can safely update `lblResult`:

```
public partial class MyWindow : Window
{
  TaskScheduler _uiScheduler;   // Declare this as a field so we can use
                                // it throughout our class.
  public MyWindow()
  {
    InitializeComponent();

    // Get the UI scheduler for the thread that created the form:
    _uiScheduler = TaskScheduler.FromCurrentSynchronizationContext();

    Task.Factory.StartNew<string> (SomeComplexWebService)
      .ContinueWith (ant => lblResult.Content = ant.Result, _uiScheduler);
  }

  string SomeComplexWebService() { ... }
}
```

It’s also possible to write our own task scheduler (by subclassing `TaskScheduler`), although this is something you’d do only in very specialized scenarios. For custom scheduling, you’d more commonly use `TaskCompletionSource`, which we’ll cover soon.

TaskFactory

When you call `Task.Factory`, you’re calling a static property on `Task` that returns a default `TaskFactory` object. The purpose of a task factory is to create tasks—specifically, three kinds of tasks:

- “Ordinary” tasks (via `StartNew`)
- Continuations with multiple antecedents (via `ContinueWhenAll` and `ContinueWhenAny`)
- Tasks that wrap methods that follow the asynchronous programming model (via `FromAsync`)

> Interestingly, `TaskFactory` is the *only* way to achieve the latter two goals. In the case of `StartNew`, `TaskFactory` is purely a convenience and technically redundant in that you can simply instantiate `Task` objects and call `Start` on them.

Creating your own task factories

`TaskFactory` is not an *abstract* factory: you can actually instantiate the class, and this is useful when you want to repeatedly create tasks using the same (nonstandard) values for `TaskCreationOptions`, `TaskContinuationOptions`, or `TaskScheduler`. For example, if we wanted to repeatedly create long-running *parented* tasks, we could create a custom factory as follows:

```
var factory = new TaskFactory (
  TaskCreationOptions.LongRunning | TaskCreationOptions.AttachedToParent,
  TaskContinuationOptions.None);
```

Creating tasks is then simply a matter of calling `StartNew` on the factory:

```
Task task1 = factory.StartNew (Method1);
Task task2 = factory.StartNew (Method2);
...
```

The custom continuation options are applied when calling `ContinueWhenAll` and `ContinueWhenAny`.

TaskCompletionSource

The `Task` class achieves two distinct things:

- It schedules a delegate to run on a pooled thread.
- It offers a rich set of features for managing work items (continuations, child tasks, exception marshaling, etc.).

Interestingly, these two things are not joined at the hip: you can leverage a task’s features for managing work items without scheduling anything to run on the thread pool. The class that enables this pattern of use is called `TaskCompletionSource`.

To use `TaskCompletionSource` you simply instantiate the class. It exposes a `Task` property that returns a task upon which you can wait and attach continuations—just like any other task. The task, however, is entirely controlled by the `TaskCompletionSource` object via the following methods:

```
public class TaskCompletionSource<TResult>
{
  public void SetResult (TResult result);
  public void SetException (Exception exception);
  public void SetCanceled();

  public bool TrySetResult (TResult result);
  public bool TrySetException (Exception exception);
  public bool TrySetCanceled();
  ...
}
```

If called more than once, `SetResult`, `SetException`, or `SetCanceled` throws an exception; the `Try*` methods instead return `false`.

> `TResult` corresponds to the task's result type, so `TaskCompletionSource<int>` gives you a `Task<int>`. If you want a task with no result, create a `TaskCompletionSource` of `object` and pass in `null` when calling `SetResult`. You can then cast the `Task<object>` to `Task`.

The following example prints 123 after waiting for five seconds:

```
var source = new TaskCompletionSource<int>();

new Thread (() => { Thread.Sleep (5000); source.SetResult (123); })
  .Start();

Task<int> task = source.Task;      // Our "slave" task.
Console.WriteLine (task.Result);   // 123
```

In "Concurrent Collections," we show how `BlockingCollection` can be used to write a producer/consumer queue. We then demonstrate how `TaskCompletionSource` improves the solution by allowing queued work items to be waited upon and canceled.

Working with AggregateException

As we've seen, PLINQ, the `Parallel` class, and `Task`s automatically marshal exceptions to the consumer. To see why this is essential, consider the following LINQ query, which throws a `DivideByZeroException` on the first iteration:

```
try
{
  var query = from i in Enumerable.Range (0, 1000000)
              select 100 / i;
  ...
}
catch (DivideByZeroException) { ... }
```

If we asked PLINQ to parallelize this query and it ignored the handling of exceptions, a `DivideByZeroException` would probably be thrown on a *separate thread*, bypassing our `catch` block and causing the application to die.

Hence, exceptions are automatically caught and rethrown to the caller. But unfortunately, it's not quite as simple as catching a `DivideByZeroException`. Because these libraries leverage many threads, it's actually possible for two or more exceptions to be thrown simultaneously. To ensure that all exceptions are reported, exceptions are therefore wrapped in an `AggregateException` container, which exposes an `InnerExceptions` property containing each of the caught exception(s):

```
try
{
  var query = from i in ParallelEnumerable.Range (0, 1000000)
              select 100 / i;
  // Enumerate query
  ...
}
catch (AggregateException aex)
{
  foreach (Exception ex in aex.InnerExceptions)
    Console.WriteLine (ex.Message);
}
```

> Both PLINQ and the `Parallel` class end the query or loop execution upon encountering the first exception—by not processing any further elements or loop bodies. More exceptions might be thrown, however, before the current cycle is complete. The first exception in `AggregateException` is visible in the `InnerException` property.

Flatten and Handle

The AggregateException class provides a couple of methods to simplify exception handling: Flatten and Handle.

Flatten

AggregateExceptions will quite often contain other AggregateExceptions. An example of when this might happen is if a child task throws an exception. You can eliminate any level of nesting to simplify handling by calling Flatten. This method returns a new AggregateException with a simple flat list of inner exceptions:

```
catch (AggregateException aex)
{
  foreach (Exception ex in aex.Flatten().InnerExceptions)
    myLogWriter.LogException (ex);
}
```

Handle

Sometimes it’s useful to catch only specific exception types, and have other types rethrown. The Handle method on AggregateException provides a shortcut for doing this. It accepts an exception predicate which it runs over every inner exception:

```
public void Handle (Func<Exception, bool> predicate)
```

If the predicate returns true, it considers that exception “handled.” After the delegate has run over every exception, the following happens:

- If all exceptions were “handled” (the delegate returned true), the exception is not rethrown.
- If there were any exceptions for which the delegate returned false (“unhandled”), a new AggregateException is built up containing those exceptions, and is rethrown.

For instance, the following ends up rethrowing another AggregateException that contains a single NullReferenceException:

```
var parent = Task.Factory.StartNew (() =>
{
  // We’ll throw 3 exceptions at once using 3 child tasks:

  int[] numbers = { 0 };

  var childFactory = new TaskFactory
   (TaskCreationOptions.AttachedToParent, TaskContinuationOptions.None);

  childFactory.StartNew (() => 5 / numbers[0]);   // Division by zero
  childFactory.StartNew (() => numbers [1]);      // Index out of range
  childFactory.StartNew (() => { throw null; });  // Null reference
});

try { parent.Wait(); }
catch (AggregateException aex)
{
  aex.Flatten().Handle (ex =>   // Note that we still need to call Flatten
  {
    if (ex is DivideByZeroException)
    {
      Console.WriteLine ("Divide by zero");
      return true;                           // This exception is "handled"
    }
    if (ex is IndexOutOfRangeException)
    {
      Console.WriteLine ("Index out of range");
      return true;                           // This exception is "handled"
    }
    return false;    // All other exceptions will get rethrown
  });
}
```

Concurrent Collections

Framework 4.0 provides a set of new collections in the System.Collections.Concurrent namespace. All of these are fully thread-safe:

Concurrent collection	Nonconcurrent equivalent
ConcurrentStack<T>	Stack<T>
ConcurrentQueue<T>	Queue<T>
ConcurrentBag<T>	(none)
BlockingCollection<T>	(none)
ConcurrentDictionary<TKey,TValue>	Dictionary<TKey,TValue>

The concurrent collections can sometimes be useful in general multithreading when you need a thread-safe collection. However, there are some caveats:

- The concurrent collections are tuned for *parallel programming*. The conventional collections outperform them in all but highly concurrent scenarios.
- A thread-safe collection doesn't guarantee that the code using it will be thread-safe (see "Thread Safety").
- If you enumerate over a concurrent collection while another thread is modifying it, no exception is thrown—instead, you get a mixture of old and new content.
- There's no concurrent version of List<T>.
- The concurrent stack, queue, and bag classes are implemented internally with linked lists. This makes them less memory-efficient than the nonconcurrent Stack and Queue classes, but better for concurrent access because linked lists are conducive to lock-free or low-lock implementations. (This is because inserting a node into a linked list requires updating just a couple of references, while inserting an element into a List<T>-like structure may require moving thousands of existing elements.)

In other words, these collections don't merely provide shortcuts for using an ordinary collection with a lock. To demonstrate, if we execute the following code on a *single* thread:

```
var d = new ConcurrentDictionary<int,int>();
for (int i = 0; i < 1000000; i++) d[i] = 123;
```

it runs three times more slowly than this:

```
var d = new Dictionary<int,int>();
for (int i = 0; i < 1000000; i++) lock (d) d[i] = 123;
```

(*Reading* from a ConcurrentDictionary, however, is fast because reads are lock-free.)

The concurrent collections also differ from conventional collections in that they expose special methods to perform atomic test-and-act operations, such as TryPop. Most of these methods are unified via the IProducerConsumerCollection<T> interface.

IProducerConsumerCollection<T>

A producer/consumer collection is one for which the two primary use cases are:

- Adding an element ("producing")
- Retrieving an element while removing it ("consuming")

The classic examples are stacks and queues. Producer/consumer collections are significant in parallel programming because they're conducive to efficient lock-free implementations.

The IProducerConsumerCollection<T> interface represents a thread-safe producer/consumer collection. The following classes implement this interface:

```
ConcurrentStack<T>
ConcurrentQueue<T>
ConcurrentBag<T>
```

`IProducerConsumerCollection<T>` extends `ICollection`, adding the following methods:

```
void CopyTo (T[] array, int index);
T[] ToArray();
bool TryAdd (T item);
bool TryTake (out T item);
```

The `TryAdd` and `TryTake` methods test whether an add/remove operation can be performed, and if so, they perform the add/remove. The testing and acting are performed atomically, eliminating the need to lock as you would around a conventional collection:

```
int result;
lock (myStack) if (myStack.Count > 0) result = myStack.Pop();
```

`TryTake` returns `false` if the collection is empty. `TryAdd` always succeeds and returns `true` in the three implementations provided. If you wrote your own concurrent collection that prohibited duplicates, however, you'd make `TryAdd` return `false` if the element already existed (an example would be if you wrote a concurrent *set*).

The particular element that `TryTake` removes is defined by the subclass:

- With a stack, `TryTake` removes the most recently added element.
- With a queue, `TryTake` removes the least recently added element.
- With a bag, `TryTake` removes whatever element it can remove most efficiently.

The three concrete classes mostly implement the `TryTake` and `TryAdd` methods explicitly, exposing the same functionality through more specifically named public methods such as `TryDequeue` and `TryPop`.

ConcurrentBag<T>

`ConcurrentBag<T>` stores an *unordered* collection of objects (with duplicates permitted). `ConcurrentBag<T>` is suitable in situations when you *don't care* which element you get when calling `Take` or `TryTake`.

The benefit of `ConcurrentBag<T>` over a concurrent queue or stack is that a bag's `Add` method suffers almost *no* contention when called by many threads at once. In contrast, calling `Add` in parallel on a queue or stack incurs *some* contention (although a lot less than locking around a *nonconcurrent* collection). Calling `Take` on a concurrent bag is also very efficient—as long as each thread doesn't take more elements than it `Add`ed.

Inside a concurrent bag, each thread gets it own private linked list. Elements are added to the private list that belongs to the thread calling `Add`, eliminating contention. When you enumerate over the bag, the enumerator travels through each thread's private list, yielding each of its elements in turn.

When you call `Take`, the bag first looks at the current thread's private list. If there's at least one element,[1] it can complete the task easily and without contention. But if the list is empty, it must "steal" an element from another thread's private list and incur the potential for contention.

So, to be precise, calling `Take` gives you the element added most recently on that thread; if there are no elements on that thread, it gives you the element added most recently on another thread, chosen at random.

Concurrent bags are ideal when the parallel operation on your collection mostly comprises `Add`ing elements—or when the `Add`s and `Take`s are balanced on a thread. We saw an example of the former previously, when using `Parallel.ForEach` to implement a parallel spellchecker:

```
var misspellings = new ConcurrentBag<Tuple<int,string>>();

Parallel.ForEach (wordsToTest, (word, state, i) =>
{
  if (!wordLookup.Contains (word))
    misspellings.Add (Tuple.Create ((int) i, word));
});
```

A concurrent bag would be a poor choice for a producer/consumer queue, because elements are added and removed by *different* threads.

[1] Due to an implementation detail, there actually needs to be at least two elements to avoid contention entirely.

BlockingCollection<T>

If you call TryTake on any of the producer/consumer collections we discussed previously:

```
ConcurrentStack<T>
ConcurrentQueue<T>
ConcurrentBag<T>
```

and the collection is empty, the method returns false. Sometimes it would be more useful in this scenario to *wait* until an element is available.

Rather than overloading the TryTake methods with this functionality (which would have caused a blowout of members after allowing for cancellation tokens and timeouts), PFX's designers encapsulated this functionality into a wrapper class called BlockingCollection<T>. A blocking collection wraps any collection that implements IProducerConsumerCollection<T> and lets you Take an element from the wrapped collection—blocking if no element is available.

A blocking collection also lets you limit the total size of the collection, blocking the *producer* if that size is exceeded. A collection limited in this manner is called a *bounded blocking collection*.

To use BlockingCollection<T>:

1. Instantiate the class, optionally specifying the IProducerConsumerCollection<T> to wrap and the maximum size (bound) of the collection.
2. Call Add or TryAdd to add elements to the underlying collection.
3. Call Take or TryTake to remove (consume) elements from the underlying collection.

If you call the constructor without passing in a collection, the class will automatically instantiate a ConcurrentQueue<T>. The producing and consuming methods let you specify cancellation tokens and timeouts. Add and TryAdd may block if the collection size is bounded; Take and TryTake block while the collection is empty.

Another way to consume elements is to call GetConsumingEnumerable. This returns a (potentially) infinite sequence that yields elements as they become available. You can force the sequence to end by calling CompleteAdding: this method also prevents further elements from being enqueued.

Previously, we wrote a producer/consumer queue using Wait and Pulse (see "Signaling with Wait and Pulse"). Here's the same class refactored to use BlockingCollection<T> (exception handling aside):

```
public class PCQueue : IDisposable
{
  BlockingCollection<Action> _taskQ = new BlockingCollection<Action>();

  public PCQueue (int workerCount)
  {
    // Create and start a separate Task for each consumer:
    for (int i = 0; i < workerCount; i++)
      Task.Factory.StartNew (Consume);
  }

  public void Dispose() { _taskQ.CompleteAdding(); }

  public void EnqueueTask (Action action) { _taskQ.Add (action); }

  void Consume()
  {
    // This sequence that we're enumerating will block when no elements
    // are available and will end when CompleteAdding is called.

    foreach (Action action in _taskQ.GetConsumingEnumerable())
      action();     // Perform task.
  }
}
```

Because we didn't pass anything into BlockingCollection's constructor, it instantiated a concurrent queue automatically. Had we passed in a ConcurrentStack, we'd have ended up with a producer/consumer stack.

BlockingCollection also provides static methods called AddToAny and TakeFromAny, which let you add or take an element while specifying several blocking collections. The action is then honored by the first collection able to service the request.

Leveraging TaskCompletionSource

The producer/consumer that we just wrote is inflexible in that we can't track work items after they've been enqueued. It would be nice if we could:

- Know when a work item has completed.
- Cancel an unstarted work item.
- Deal elegantly with any exceptions thrown by a work item.

An ideal solution would be to have the EnqueueTask method return some object giving us the functionality just described. The good news is that a class already exists to do exactly this—the Task class. All we need to do is to hijack control of the task via TaskCompletionSource:

```
public class PCQueue : IDisposable
{
  class WorkItem
  {
    public readonly TaskCompletionSource<object> TaskSource;
    public readonly Action Action;
    public readonly CancellationToken? CancelToken;

    public WorkItem (
      TaskCompletionSource<object> taskSource,
      Action action,
      CancellationToken? cancelToken)
    {
      TaskSource = taskSource;
      Action = action;
      CancelToken = cancelToken;
    }
  }

  BlockingCollection<WorkItem> _taskQ = new BlockingCollection<WorkItem>();

  public PCQueue (int workerCount)
  {
    // Create and start a separate Task for each consumer:
    for (int i = 0; i < workerCount; i++)
      Task.Factory.StartNew (Consume);
  }

  public void Dispose() { _taskQ.CompleteAdding(); }

  public Task EnqueueTask (Action action)
  {
    return EnqueueTask (action, null);
  }

  public Task EnqueueTask (Action action, CancellationToken? cancelToken)
  {
    var tcs = new TaskCompletionSource<object>();
    _taskQ.Add (new WorkItem (tcs, action, cancelToken));
    return tcs.Task;
  }
  void Consume()
  {
    foreach (WorkItem workItem in _taskQ.GetConsumingEnumerable())
      if (workItem.CancelToken.HasValue &&
          workItem.CancelToken.Value.IsCancellationRequested)
      {
        workItem.TaskSource.SetCanceled();
      }
      else
        try
        {
          workItem.Action();
          workItem.TaskSource.SetResult (null);   // Indicate completion
        }
        catch (Exception ex)
        {
          workItem.TaskSource.SetException (ex);
        }
  }
}
```

In `EnqueueTask`, we enqueue a work item that encapsulates the target delegate and a task completion source—which lets us later control the task that we return to the consumer.

In `Consume`, we first check whether a task has been canceled after dequeuing the work item. If not, we run the delegate and then call `SetResult` on the task completion source to indicate its completion.

Here's how we can use this class:

```
var pcQ = new PCQueue (1);
Task task = pcQ.EnqueueTask (() => Console.WriteLine ("Easy!"));
...
```

We can now wait on `task`, perform continuations on it, have exceptions propagate to continuations on parent tasks, and so on. In other words, we've got the richness of the task model while, in effect, implementing our own scheduler.

SpinLock and SpinWait

In parallel programming, a brief episode of spinning is often preferable to blocking, as it avoids the cost of context switching and kernel transitions. `SpinLock` and `SpinWait` are designed to help in such cases. Their main use is in writing custom synchronization constructs.

> `SpinLock` and `SpinWait` are structs and not classes! This design decision was an extreme optimization technique to avoid the cost of indirection and garbage collection. It means that you must be careful not to unintentionally *copy* instances—by passing them to another method without the `ref` modifier, for instance, or declaring them as `readonly` fields. This is particularly important in the case of `SpinLock`.

SpinLock

The `SpinLock` struct lets you lock without incurring the cost of a context switch, at the expense of keeping a thread spinning (uselessly busy). This approach is valid in high-contention scenarios when locking will be very brief (e.g., in writing a thread-safe linked list from scratch).

> If you leave a spinlock contended for too long (we're talking milliseconds at most), it will yield its time slice, causing a context switch just like an ordinary lock. When rescheduled, it will yield again—in a continual cycle of "spin yielding." This consumes far fewer CPU resources than outright spinning—but more than blocking.
>
> On a single-core machine, a spinlock will start "spin yielding" immediately if contended.

Using a SpinLock is like using an ordinary lock, except:

- Spinlocks are structs (as previously mentioned).
- Spinlocks are not reentrant, meaning that you cannot call Enter on the same SpinLock twice in a row on the same thread. If you violate this rule, it will either throw an exception (if *owner tracking* is enabled) or deadlock (if owner tracking is disabled). You can specify whether to enable owner tracking when constructing the spinlock. Owner tracking incurs a performance hit.
- SpinLock lets you query whether the lock is taken, via the properties IsHeld and, if owner tracking is enabled, IsHeldByCurrentThread.
- There's no equivalent to C#'s lock statement to provide SpinLock syntactic sugar.

Another difference is that when you call Enter, you *must* follow the robust pattern of providing a lockTaken argument (which is nearly always done within a try/finally block).

Here's an example:

```
var spinLock = new SpinLock (true);   // Enable owner tracking
bool lockTaken = false;
try
{
  spinLock.Enter (ref lockTaken);
  // Do stuff...
}
finally
{
  if (lockTaken) spinLock.Exit();
}
```

As with an ordinary lock, lockTaken will be false after calling Enter if (and only if) the Enter method throws an exception and the lock was not taken. This happens in very rare scenarios (such as Abort being called on the thread, or an OutOfMemoryException being thrown) and lets you reliably know whether to subsequently call Exit.

SpinLock also provides a TryEnter method which accepts a timeout.

> Given SpinLock's ungainly value-type semantics and lack of language support, it's almost as if they *want* you to suffer every time you use it! Think carefully before dismissing an ordinary lock.

A SpinLock makes the most sense when writing your own reusable synchronization constructs. Even then, a spinlock is not as useful as it sounds. It still limits concurrency. And it wastes CPU time doing *nothing useful*. Often, a better choice is to spend some of that time doing something *speculative*—with the help of SpinWait.

SpinWait

SpinWait helps you write lock-free code that spins rather than blocks. It works by implementing safeguards to avoid the dangers of resource starvation and priority inversion that might otherwise arise with spinning.

> Lock-free programming with SpinWait is as *hardcore* as multithreading gets and is intended for when none of the higher-level constructs will do. A prerequisite is to understand "Nonblocking Synchronization".

Why we need SpinWait

Suppose we wrote a spin-based signaling system based purely on a simple flag:

```
bool _proceed;
void Test()
{
  // Spin until another thread sets _proceed to true:
  while (!_proceed) Thread.MemoryBarrier();
  ...
}
```

This would be highly efficient if Test ran when _proceed was already true—or if _proceed became true within a few cycles. But now suppose that _proceed remained false for several seconds—and that four threads called Test at once.

The spinning would then fully consume a quad-core CPU! This would cause other threads to run slowly (resource starvation)—including the very thread that might eventually set _proceed to true (priority inversion). The situation is exacerbated on single-core machines, where spinning will nearly *always* cause priority inversion. (And although single-core machines are rare nowadays, single-core *virtual machines* are not.)

SpinWait addresses these problems in two ways. First, it limits CPU-intensive spinning to a set number of iterations, after which it yields its time slice on every spin (by calling Thread.Yield and Thread.Sleep), lowering its resource consumption. Second, it detects whether it's running on a single-core machine, and if so, it yields on every cycle.

How to use SpinWait

There are two ways to use SpinWait. The first is to call its static method, SpinUntil. This method accepts a predicate (and optionally, a timeout):

```
bool _proceed;
void Test()
{
  SpinWait.SpinUntil (() => { Thread.MemoryBarrier(); return _proceed; });
  ...
}
```

The other (more flexible) way to use SpinWait is to instantiate the struct and then to call SpinOnce in a loop:

```
bool _proceed;
void Test()
{
  var spinWait = new SpinWait();
  while (!_proceed) { Thread.MemoryBarrier(); spinWait.SpinOnce(); }
  ...
}
```

The former is a shortcut for the latter.

How SpinWait works

In its current implementation, SpinWait performs CPU-intensive spinning for 10 iterations before yielding. However, it doesn't return to the caller *immediately* after each of those cycles: instead, it calls Thread.SpinWait to spin via the CLR (and ultimately the operating system) for a set time period. This time period is initially a few tens of nanoseconds, but doubles with each iteration until the 10 iterations are up. This ensures some predictability in the total time spent in the CPU-intensive spinning phase, which the CLR and operating system can tune according to conditions. Typically, it's in the few-tens-of-microseconds region—small, but more than the cost of a context switch.

On a single-core machine, SpinWait yields on every iteration. You can test whether SpinWait will yield on the next spin via the property NextSpinWillYield.

If a SpinWait remains in "spin-yielding" mode for long enough (maybe 20 cycles) it will periodically *sleep* for a few milliseconds to further save resources and help other threads progress.

Lock-free updates with SpinWait and Interlocked.CompareExchange

SpinWait in conjunction with Interlocked.CompareExchange can atomically update fields with a value calculated from the original (read-modify-write). For example, suppose we want to multiply field *x* by 10. Simply doing the following is not thread-safe:

```
x = x * 10;
```

for the same reason that incrementing a field is not thread-safe, as we saw in "Nonblocking Synchronization".

The correct way to do this without locks is as follows:

4. Take a "snapshot" of *x* into a local variable.
5. Calculate the new value (in this case by multiplying the snapshot by 10).
6. Write the calculated value back *if* the snapshot is still up-to-date (this step must be done atomically by calling Interlocked.CompareExchange).
7. If the snapshot was stale, *spin* and return to step 1.

For example:

```
int x;

void MultiplyXBy (int factor)
{
  var spinWait = new SpinWait();
  while (true)
  {
    int snapshot1 = x;
    Thread.MemoryBarrier();
    int calc = snapshot1 * factor;
    int snapshot2 = Interlocked.CompareExchange (ref x, calc, snapshot1);
    if (snapshot1 == snapshot2) return;   // No one preempted us.
    spinWait.SpinOnce();
  }
}
```

We can improve performance (slightly) by doing away with the call to `Thread.MemoryBarrier`. We can get away with this because `CompareExchange` generates a memory barrier anyway—so the worst that can happen is an extra spin if `snapshot1` happens to read a stale value in its first iteration.

`Interlocked.CompareExchange` updates a field with a specified value *if* the field's current value matches the third argument. It then returns the field's old value, so you can test whether it succeeded by comparing that against the original snapshot. If the values differ, it means that another thread preempted you, in which case you spin and try again.

`CompareExchange` is overloaded to work with the `object` type too. We can leverage this overload by writing a lock-free update method that works with all reference types:

```
static void LockFreeUpdate<T> (ref T field, Func <T, T> updateFunction)
  where T : class
{
  var spinWait = new SpinWait();
  while (true)
  {
    T snapshot1 = field;
    T calc = updateFunction (snapshot1);
    T snapshot2 = Interlocked.CompareExchange (ref field, calc, snapshot1);
    if (snapshot1 == snapshot2) return;
    spinWait.SpinOnce();
  }
}
```

Here's how we can use this method to write a thread-safe event without locks (this is, in fact, what the C# 4.0 compiler now does by default with events):

```
EventHandler _someDelegate;
public event EventHandler SomeEvent
{
  add    { LockFreeUpdate (ref _someDelegate, d => d + value); }
  remove { LockFreeUpdate (ref _someDelegate, d => d - value); }
}
```

SpinWait Versus SpinLock

We could solve these problems instead by wrapping access to the shared field around a `SpinLock`. The problem with spin locking, though, is that it allows only one thread to proceed at a time—even though the spinlock (usually) eliminates the context-switching overhead. With `SpinWait`, we can proceed speculatively and *assume* no contention. If we do get preempted, we simply try again. Spending CPU time doing something that *might* work is better than wasting CPU time in a spinlock!

Finally, consider the following class:

```
class Test
{
  ProgressStatus _status = new ProgressStatus (0, "Starting");

  class ProgressStatus    // Immutable class
  {
    public readonly int PercentComplete;
    public readonly string Message;

    public ProgressStatus (int percentComplete, string message)
    {
      PercentComplete = percentComplete;
      Message = message;
    }
  }
}
```

We can use our `LockFreeUpdate` method to “increment” the `PercentComplete` field in `_status` as follows:

```
LockFreeUpdate (ref _status,
  s => new ProgressStatus (s.PercentComplete + 1, s.Message));
```

Notice that we’re creating a new `ProgressStatus` object based on existing values. Thanks to the `LockFreeUpdate` method, the act of reading the existing `PercentComplete` value, incrementing it, and writing it back can’t get *unsafely* preempted: any preemption is reliably detected, triggering a spin and retry.