Multithreading

Source http://www.albahari.com/threading/part2.html

Blocking

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

- by the blocking condition being satisfied
- by the operation timing out (if a timeout is specified)
- by being interrupted via Thread.Interrupt
- by being aborted via Thread.Abort

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

Sleeping and Spinning

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

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

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

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

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

Blocking vs. Spinning

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

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

or:

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

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

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

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

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

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

Joining a Thread

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

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

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

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

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

Locking and Thread Safety

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

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

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

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

Here's how lock can fix the problem:

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

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

Only one thread can lock the synchronizing object (in this case locker) at a time, and any contending threads are blocked until the lock is released. If more than one thread contends the lock, they are queued – on a "ready queue" and granted the lock on a first-come, first-served basis as it becomes available. Exclusive locks are sometimes said to enforce serialized access to whatever's protected by the lock, because one thread's access cannot

overlap with that of another. In this case, we're protecting the logic inside the Go method, as well as the fields val1 and val2.

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

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

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

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

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

Choosing the Synchronization Object

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

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

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

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

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

or:

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

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

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

Nested Locking

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

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

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

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

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

When to Lock

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

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

Here are thread-safe versions of Increment and Assign:

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

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

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

Locking and Atomicity

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

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

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

Performance Considerations

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

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

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

Thread Safety

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

A method which is thread-safe in any scenario is called reentrant. General-purpose types are rarely thread-safe in their entirety, for the following reasons:

- the development burden in full thread-safety can be significant, particularly if a type has many fields (each field is a potential for interaction in an arbitrarily multi-threaded context)
- thread-safety can entail a performance cost (payable, in part, whether or not the type is actually used by multiple threads)
- a thread-safe type does not necessarily make the program using it thread-safe – and sometimes the work involved in the latter can make the former redundant.

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

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

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

Thread-Safety and .NET Framework Types

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Interrupt and Abort

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

- **via Thread.Interrupt**
- **via Thread.Abort**

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

Interrupt

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

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

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

```
Forcibly Woken!
```

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

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

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

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

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

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

Abort

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

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

Thread State

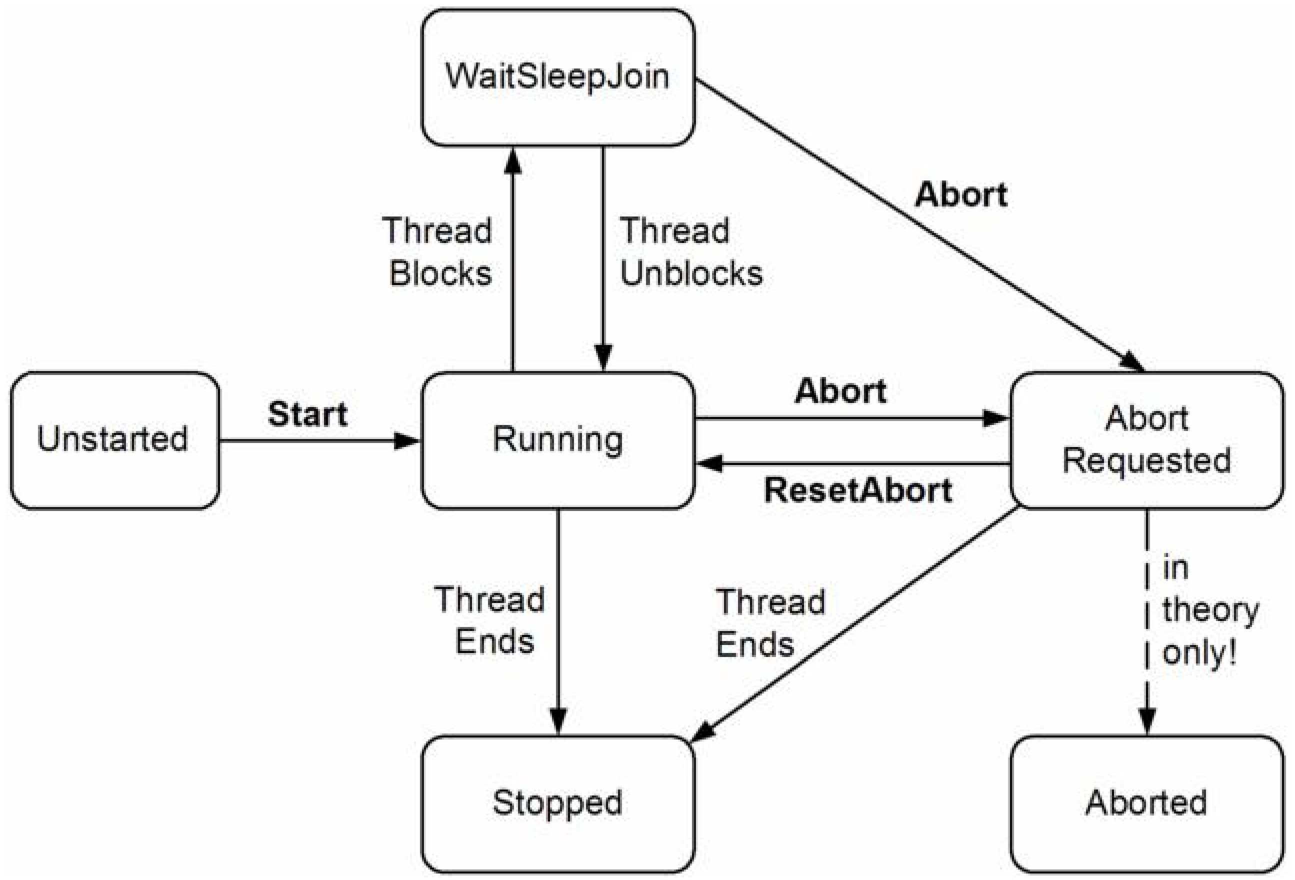


Figure 1: Thread State Diagram

One can query a thread's execution status via its ThreadState property. Figure 1 shows one "layer" of the ThreadState enumeration. ThreadState is horribly designed, in that it

combines three "layers" of state using bitwise flags, the members within each layer being themselves mutually exclusive. Here are all three layers:

- **the running / blocking / aborting status (as shown in Figure 1)**
- **the background/foreground status (ThreadState.Background)**
- **the progress towards suspension via the deprecated Suspend method (ThreadState.SuspendRequested and ThreadState.Suspended)**

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

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

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

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

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

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

Assuming one steers clear of the deprecated Suspend and Resume methods, one can write a helper method that eliminates all but members of the first layer, allowing simple equality tests to be performed. A thread's background status can be obtained independently via its IsBackground property, so only the first layer actually has useful information:

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

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

Wait Handles

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

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

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

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

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

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

AutoResetEvent

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Creating a Cross-Process EventWaitHandle

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

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

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

Acknowledgement

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

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

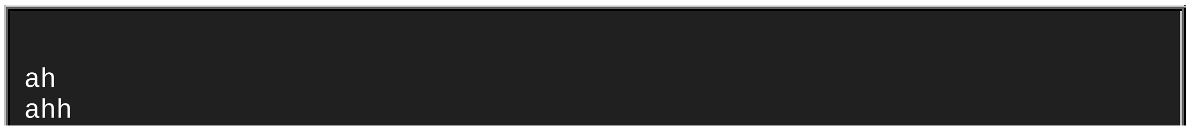
```
class AcknowledgedWaitHandle {
  static EventWaitHandle ready = new AutoResetEvent (false);
  static EventWaitHandle go = new AutoResetEvent (false);
  static volatile string task;

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

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

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

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



```
ah
ahh
```

```
ahhh
ahhhh
```

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

Producer/Consumer Queue

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

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

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

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

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

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

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

  public void Dispose() {
    EnqueueTask (null);     // Signal the consumer to exit.
    worker.Join();          // Wait for the consumer's thread to finish.
    wh.Close();             // Release any OS resources.
  }
  void Work() {
    while (true) {
      string task = null;
      lock (locker)
        if (tasks.Count > 0) {
          task = tasks.Dequeue();
          if (task == null) return;
        }
      if (task != null) {
        Console.WriteLine ("Performing task: " + task);
        Thread.Sleep (1000);  // simulate work...
      }
      else
        wh.WaitOne();         // No more tasks - wait for a signal
    }
  }
}
```

Here's a main method to test the queue:

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

```
Performing task: Hello
Performing task: Say 1
Performing task: Say 2
Performing task: Say 3
…
…
Performing task: Say 9
Goodbye!
```

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

ManualResetEvent

A ManualResetEvent is a variation on AutoResetEvent. It differs in that it doesn't automatically reset after a thread is let through on a WaitOne call, and so functions like a gate: calling Set opens the gate, allowing any number of threads that WaitOne at the gate

through; calling Reset closes the gate, causing, potentially, a queue of waiters to accumulate until its next opened.

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

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

Mutex

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

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

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

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

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

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

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

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

Semaphore

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

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

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

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

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

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

WaitAny, WaitAll and SignalAndWait

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

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

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

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

while the second thread does the opposite:

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

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

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

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

Synchronization Contexts

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A synchronization context can extend beyond the scope of a single object. By default, if a synchronized object is instantiated from within the code of another, both share the same context (in other words, one big lock!) This behavior can be changed by specifying an

integer flag in Synchronization attribute's constructor, using one of the constants defined in the SynchronizationAttribute class:

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

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

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

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

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

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

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

Reentrancy

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

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

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

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

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

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

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

***~~~ End of Article ~~~***