- `SpinLock` and `SpinWait` for high-concurrency scenarios

Synchronization Overview

Synchronization is the act of coordinating concurrent actions for a predictable outcome. Synchronization is particularly important when multiple threads access the same data; it's surprisingly easy to run aground in this area.

The simplest and most useful synchronization tools are arguably the continuations and task combinators described in Chapter 14. By formulating concurrent programs into asynchronous operations strung together with continuations and combinators, you lessen the need for locking and signaling. However, there are still times when the lower-level constructs come into play.

The synchronization constructs can be divided into three categories:

Exclusive locking

> Exclusive locking constructs allow just one thread to perform some activity or execute a section of code at a time. Their primary purpose is to let threads access shared writing state without interfering with one another. The exclusive locking constructs are `lock`, `Mutex`, and `SpinLock`.

Nonexclusive locking

> Nonexclusive locking lets you *limit* concurrency. The nonexclusive locking constructs are `Semaphore(Slim)` and `ReaderWriterLock(Slim)`.

Signaling

These allow a thread to block until receiving one or more notifications from other thread(s). The signaling constructs include `ManualResetEvent(Slim)`, `AutoResetEvent`, `CountdownEvent`, and `Barrier`. The former three are referred to as *event wait handles*.

It's also possible (and tricky) to perform certain concurrent operations on shared state without locking through the use of *nonblocking synchronization constructs*. These are `Thread.MemoryBarrier`, `Thread.VolatileRead`, `Thread.VolatileWrite`, the `volatile` keyword, and the `Interlocked` class. We cover this topic online, along with `Monitor`'s `Wait`/`Pulse` methods, which you can use to write custom signaling logic.

Exclusive Locking

There are three exclusive locking constructs: the `lock` statement, `Mutex`, and `SpinLock`. The `lock` construct is the most convenient and widely used, whereas the other two target niche scenarios:

- `Mutex` lets you span multiple processes (computer-wide locks).
- `SpinLock` implements a micro-optimization that can lessen context switches in high-concurrency scenarios (see *http://albahari.com/threading*).

The lock Statement

To illustrate the need for locking, consider 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` were 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` fixes the problem:

```
class ThreadSafe
{
  static readonly object _locker = new object();
  static int _val1 = 1, _val2 = 1;

  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.[1] 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`.

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 has a subtle vulnerability. Consider the (unlikely) event of an exception being thrown between the call to `Monitor.Enter` and the `try` block (due, perhaps, to an `OutOfMemoryException` or, in .NET Framework, if the thread is aborted). In such a scenario, the lock might or might 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, `Monitor.Enter` defines the following overload:

```
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 more robust pattern of use (which is exactly how C# 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 immediately. As with the `Enter` method, `TryEnter` is overloaded to accept a `lockTaken` argument.

Choosing the Synchronization Object

You can use any object visible to each of the partaking threads 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. You can also use the containing object (`this`) as a synchronization object:

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

Or even its type:

```
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 more difficult to prevent deadlocking and excessive blocking.

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

NOTE

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`:

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

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

Without locks, two problems can arise:

- Operations such as incrementing a variable (or even reading/writing a variable, under certain conditions) are not atomic.
- The compiler, CLR, and processor are entitled to reorder instructions and cache variables in CPU registers to improve performance—as long as such optimizations don't change the behavior of a *single*-threaded program (or a multithreaded program that uses locks).

Locking mitigates the second problem because it creates a *memory barrier* before and after the lock. A memory barrier is a "fence" through which the effects of reordering and caching cannot penetrate.

NOTE

This applies not just to locks but to all synchronization constructs. So, if your use of a *signaling* construct, for instance, ensures that just one thread reads/writes a variable at a time, you don't need to lock. Hence, the following code is thread-safe without locking around `x`:

```
var signal = new ManualResetEvent (false);
int x = 0;
new Thread (() => { x++; signal.Set(); }).Start();
signal.WaitOne();
Console.WriteLine (x);    // 1 (always)
```

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 that the variables are read and written *atomically*. Let's suppose that fields `x` and `y` are always read and assigned within a `lock` on object `locker`:

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

We can say that `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, provided that `x` and `y` are always accessed within this same exclusive lock.

NOTE

The atomicity provided by a lock is violated if an exception is thrown within a `lock` block (whether or not multithreading is involved). For example, consider the following:

```
decimal _savingsBalance, _checkBalance;

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

If an exception were 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, albeit analogous, concept: an instruction is atomic if it executes indivisibly on the underlying processor.

Nested Locking

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

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

Alternatively:

```
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 from within a lock:

```
object locker = new object();

lock (locker)
{
  AnotherMethod();
  // We still have the lock - because locks are reentrant.
}

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
}
```

You can create more elaborate deadlocking chains with three or more threads.

NOTE

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 most difficult 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 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 prevent deadlocks," although helpful in our initial example, is difficult to apply to the scenario just described. A better strategy is to be wary of locking around calls to methods in objects that might have references back to your own object. Also, consider whether you really need to lock around calls to methods in other classes (often you do—as you'll see in "Locking and Thread Safety"—but sometimes there are other options). Relying more on higher-level synchronization options such as task continuations/combinators, data parallelism and immutable types (later in this chapter) can lessen the need for locking.

NOTE

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; it's 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 user interface happens to be running another method that's waiting on the same lock, a deadlock will happen right there. You often can fix this simply by calling `BeginInvoke` instead of `Invoke` (or relying on asynchronous functions that do this implicitly when a synchronization context is present). Alternatively, you can release your lock before calling `Invoke`, although this won't work if your *caller* took out the lock.

Performance

Locking is fast: you can expect to acquire and release a lock in less than 20 nanoseconds on a 2020-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 can be longer before the thread is actually rescheduled.

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 around half a microsecond—more than 20 times slower than a `lock`.

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

NOTE

If you forget to call `ReleaseMutex` and simply call `Close` or `Dispose`, an `AbandonedMutexException` will be thrown upon anyone else waiting upon that mutex.

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:

```
// 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 (true, @"Global\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;
}
```

```
try { RunProgram(); }
finally { mutex.ReleaseMutex (); }

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

NOTE

If you're running under Terminal Services or in separate Unix consoles, a computer-wide `Mutex` is ordinarily visible only to applications in the same session. To make it visible to all terminal server sessions, prefix its name with *Global*, as shown in the example.

Locking and Thread Safety

A program or method is thread-safe if it can work correctly in 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 thus 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 .NET 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).

NOTE

Primitive types aside, few .NET 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` that we cover in Chapter 22 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. Because 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 because classes do not save data between requests. Thread interaction is then limited just to the static fields that you might choose to create, for such purposes as caching commonly used data in memory and in providing infrastructure services such as authentication and auditing.

Yet another solution (in rich-client applications) is to run code that accesses shared state on the UI thread. As we saw in Chapter 14, asynchronous functions make this easy.

Thread Safety and .NET Types

You can use locking to convert thread-unsafe code into thread-safe code. A good application of this is .NET: nearly all of its nonprimitive types are not thread-safe (for anything more than read-only access) when instantiated, and yet you can use them in multithreaded code if all access to any given object is protected via a lock. Here's an example in which two threads simultaneously add an item to the same `List` collection and 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 need 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; see "Reader/Writer Locks".)

Locking around thread-safe objects

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

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

Regardless of whether the list was thread-safe, this statement is certainly not! The whole `if` statement would need 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 to ensure that it did not preempt the former statement:

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

In other words, we would need to lock exactly as with our thread-unsafe collection classes (making the `List` class's hypothetical thread safety redundant).

> **NOTE**
>
> Locking around accessing a collection can cause excessive blocking in highly concurrent environments. To this end, .NET provides a thread-safe queue, stack, and dictionary, which we discuss in Chapter 22.

Static members

Wrapping access to an object around a custom lock works only if all concurrent threads are aware of—and use—the lock. This might 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 .NET: *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.

NOTE

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 .NET 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 begin 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.

NOTE

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. ASP.NET Core and Web API applications are implicitly multithreaded. 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 that 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 were called frequently, you could improve performance by caching the results in a static `Dictionary`. Here's a conceptually simple 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 creates a 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 it 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.

For an ideal solution, we need to use the strategy we described in "Completing synchronously". Instead of caching `User`, we cache `Task<User>`, which the caller then awaits:

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

  internal static Task<User> GetUserAsync (int id)
  {
    lock (_userTasks)
      if (_userTasks.TryGetValue (id, out var userTask))
        return userTask;
      else
        return _userTasks [id] = Task.Run (() => RetrieveUser (id));
  }
}
```

Notice that we now have a single lock that covers the entire method's logic. We can do this without hurting concurrency because all we're doing inside the lock is accessing the dictionary and (potentially) *initiating* an asynchronous operation (by calling `Task.Run`). Should two threads call this method at the same time with the same ID, they'll both end up awaiting the *same task*, which is exactly the outcome we want.

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 that we had two fields, as follows:

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

Now let's assume that we want to read and 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 and 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 reference (within a lock). Then, we can read its values without needing to hold onto the lock:

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

Nonexclusive Locking

The nonexclusive locking constructs serve to *limit* concurrency. In this section, we cover semaphores and read/writer locks, and also illustrate how the `SemaphoreSlim` class can limit concurrency with asynchronous operations.

Semaphore

A semaphore is like a nightclub with a limited capacity, enforced by a bouncer. When the club is full, no more people can enter, and a queue builds up outside.

A semaphore's *count* corresponds to the number of spaces in the nightclub. *Releasing* a semaphore *increases* the count; this typically happens when somebody leaves the club (corresponding to a resource being released), and also when the semaphore is initialized (to set its starting capacity). You can also call `Release` at any time to increase capacity.

Waiting on a semaphore *decrements* the count, and typically occurs prior to a resource being obtained. Calling `Wait` on a semaphore whose current count is greater than `0` completes immediately.

A semaphore can optionally have a maximum count that serves as a hard limit. Increasing the count beyond this limit throws an exception. When constructing a semaphore, you specify the initial count (starting capacity), and optionally, a maximum limit.

A semaphore with an initial count 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.

NOTE

There are two functionally similar versions of this class: `Semaphore` and `SemaphoreSlim`. The latter 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 (see "Cancellation"), and it exposes a `WaitAsync` method for asynchronous programming. You cannot use it, however, for interprocess signaling.

`Semaphore` incurs about one microsecond in calling `WaitOne` and `Release`; `SemaphoreSlim` incurs about one-tenth 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!
```

It’s also legal to instantiate a semaphore with an initial count (capacity) of 0 and then call `Release` to increase its count. The following two semaphores are equivalent:

```
var semaphore1 = new SemaphoreSlim (3);
var semaphore2 = new SemaphoreSlim (0); semaphore2.Release (3);
```

A `Semaphore`, if named, can span processes in the same way as a `Mutex` (named `Semaphore`s are available only on Windows, whereas named `Mutex` also work on Unix platforms).

Asynchronous semaphores and locks

It is illegal to lock across an `await` statement:

```
lock (_locker)
{
  await Task.Delay (1000);    // Compilation error
  ...
}
```

Doing so would make no sense, because locks are held by a thread, which typically changes when returning from an await. Locking also *blocks*, and blocking for a potentially long period of time is exactly what you're *not* trying to achieve with asynchronous functions.

It's still sometimes desirable, however, to make asynchronous operations execute sequentially—or limit the parallelism such that not more than *n* operations execute at once. For example, consider a web browser: it needs to perform asynchronous downloads in parallel, but it might want to impose a limit such that a maximum of 10 downloads happen at a time. We can achieve this by using a `SemaphoreSlim`:

```
SemaphoreSlim _semaphore = new SemaphoreSlim (10);

async Task<byte[]> DownloadWithSemaphoreAsync (string uri)
{
    await _semaphore.WaitAsync();
    try { return await new WebClient().DownloadDataTaskAsync (uri); }
    finally { _semaphore.Release(); }
}
```

Reducing the semaphore's `initialCount` to `1` reduces the maximum parallelism to 1, turning this into an asynchronous lock.

Writing an EnterAsync extension method

The following extension method simplifies the asynchronous use of `SemaphoreSlim` by using the `Disposable` class that we wrote in "Anonymous Disposal":

```
public static async Task<IDisposable> EnterAsync (this SemaphoreSlim ss)
{
  await ss.WaitAsync().ConfigureAwait (false);
  return Disposable.Create (() => ss.Release());
}
```

With this method, we can rewrite our `DownloadWithSemaphoreAsync` method as follows:

```
async Task<byte[]> DownloadWithSemaphoreAsync (string uri)
{
  using (await _semaphore.EnterAsync())
    return await new WebClient().DownloadDataTaskAsync (uri);
}
```

Parallel.ForEachAsync

From .NET 6, another approach to limit asynchronous concurrency is to use the `Parallel.ForEachAsync` method. Assuming `uris` in an array of URIs that you wish to download, here's how to download them in parallel, while limiting the concurrency to a maximum of 10 parallel downloads:

```
await Parallel.ForEachAsync (uris,
  new ParallelOptions { MaxDegreeOfParallelism = 10 },
  async (uri, cancelToken) =>
   {
    var download = await new HttpClient().GetByteArrayAsync (uri);
    Console.WriteLine ($"Downloaded {download.Length} bytes");
  });
```

The other methods in the Parallel class are intended for (compute-bound) parallel programming scenarios, which we describe in Chapter 22.

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, for which 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.

NOTE

`ReaderWriterLockSlim` 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 still twice as slow, though. The trade-off is less contention (when there's a lot of reading and minimal writing).

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 `Enter`*`XXX`* 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 `Acquire`*`XXX`* and `Release`*`XXX`*. 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 100 ms. 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); }
}
```

NOTE

In production code, you'd typically add `try`/`finally` blocks to ensure that locks were released if an exception were 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

Sometimes, it's useful to swap a read lock for a write lock in a single atomic operation. For instance, suppose that 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 whether the item is already present in the list; 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. Although 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 the list only if it's 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);
  }
```

NOTE

`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 after 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.

Signaling with Event Wait Handles

The simplest kind of signaling constructs are called *event wait handles* (unrelated to C# events). Event wait handles come in three flavors: `AutoResetEvent`, `ManualResetEvent(Slim)`, and `CountdownEvent`. The former two are based on the common `EventWaitHandle` class from which they derive all their functionality.

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[2] 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.

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 (see Figure 21-1):

```
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.
```

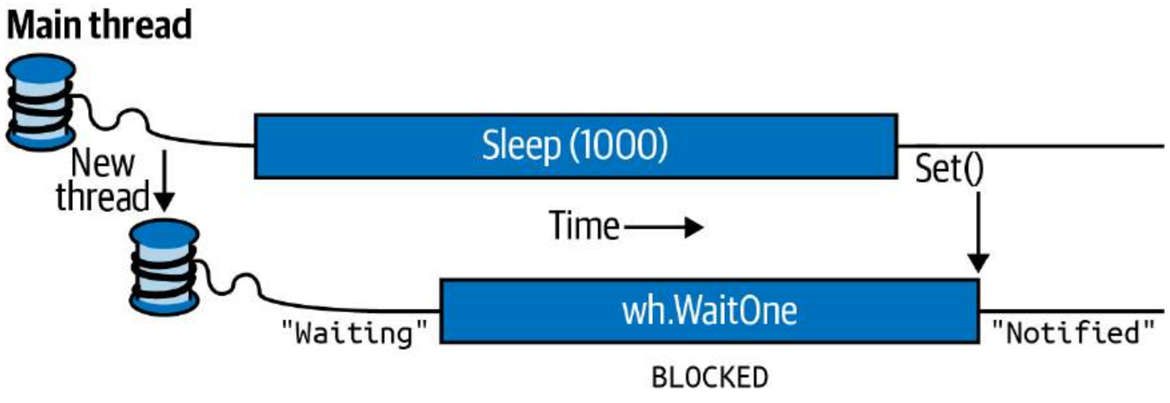

Figure 21-1. Signaling with an `EventWaitHandle`

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 prevent a race between a thread heading for the turnstile and a thread inserting a ticket ("Oops, inserted the ticket a microsecond too soon; now you'll have to wait indefinitely!”). However, calling `Set` repeatedly on a turnstile at which no one is waiting doesn’t allow an entire party through when they arrive: only the next single person is let through, and the extra tickets are “wasted.”

DISPOSING WAIT HANDLES

After you’ve finished with a wait handle, you can call its `Close` method to release the OS 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 for which relying on this backup is (arguably) acceptable, because wait handles have a light OS burden.

Wait handles are released automatically when a process exits.

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.

NOTE

Calling `WaitOne` with a timeout of `0` tests whether a wait handle is “open,” without blocking the caller. Keep in mind, though, that doing this resets the `AutoResetEvent` if it’s open.

Two-way signaling

Suppose that 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 can become lost because the worker might take time to process each signal.

The solution is for the main thread to wait until the worker's ready before signaling it. We can do this by using 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
```

Figure 21-2 shows this process.

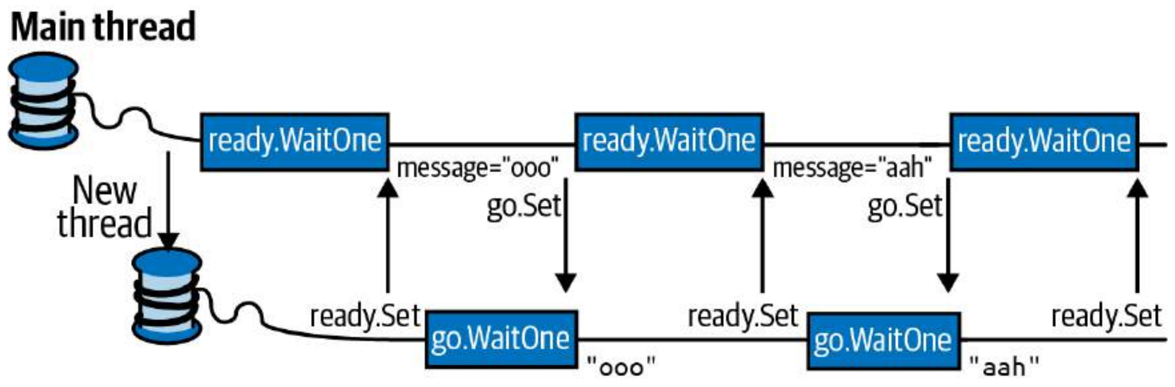

Figure 21-2. Two-way signaling

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!

ManualResetEvent

As we described in Chapter 14, a `ManualResetEvent` functions like a simple 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);
```

NOTE

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`. `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 OS and judicious use of spinning constructs. In most scenarios, however, the overhead of the signaling classes themselves doesn’t create a bottleneck; thus, it is rarely a consideration.

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 has an efficient, fully managed implementation. To use the class, instantiate it 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:

```
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!");

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

NOTE

You can sometimes more easily solve problems for which `CountdownEvent` is effective by using the *structured parallelism* constructs that we describe in Chapter 22 (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 prevent 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 in which 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 OS creates a new one. Here's an example:

```
EventWaitHandle wh = new EventWaitHandle (false, EventResetMode.AutoReset,
                                      @"Global\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.

Named event wait handles are available only on Windows.

Wait Handles and Continuations

Rather than waiting on a wait handle (and blocking your thread), you can attach a "continuation" to it by calling `ThreadPool.RegisterWaitForSingleObject`. This method accepts a delegate that is executed when a wait handle is signaled:

```
var starter = new ManualResetEvent (false);

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.

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. You are then supposed to call `Unregister` to release the unmanaged handle to the callback.

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 a one-off rather than recurring.

NOTE

You can reliably call `RegisterWaitForSingleObject` only once per wait handle. Calling this method again on the same wait handle causes an intermittent failure, whereby an unsignaled wait handle fires a callback as though it were signaled.

This limitation makes (the nonslim) wait handles poorly suited to asynchronous programming.

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 signaling and waiting operations on multiple handles. The wait handles can be of differing types (including `Mutex` and `Semaphore` given that these also derive from the abstract `WaitHandle` class). `ManualResetEventSlim` and `CountdownEvent` can also partake in these methods via their `WaitHandle` properties.

NOTE

`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 Forms application, for example, is unable to interact with the clipboard in this mode. We 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`. After signaling the first handle, it will jump to the head of the queue in waiting on the second handle; this helps it succeed (although the operation is not truly atomic). You can think of this method as "swapping" one signal for another, and use it 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);
```

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 queue-jumping semantics: in our rendezvous example, for instance, it would be valid simply to call `Set` on the first wait handle, and then `WaitOne` on the other, if wait handles were used solely for that rendezvous. In the following section, we explore yet another option for implementing a thread rendezvous.

In the case of `WaitAny` and `WaitAll`, if you don't need atomicity, you can use the code we wrote in the previous section to convert the wait handles to tasks and then use `Task.WhenAny` and `Task.WhenAll` (Chapter 14).

If you need atomicity, you can take the lowest-level approach to signaling and write the logic yourself with `Monitor`'s `Wait` and `Pulse` methods. We describe `Wait` and `Pulse` in detail in *http://albahari.com/threading*.

The Barrier Class

The `Barrier` class implements a *thread execution barrier*, allowing many threads to rendezvous at a point in time (not to be confused with `Thread.MemoryBarrier`). 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. It then starts over: calling `SignalAndWait` again blocks until called another three times. This keeps each thread "in step" with every other thread.

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

```
var barrier = new Barrier (3);

new Thread (Speak).Start();
new Thread (Speak).Start();
new Thread (Speak).Start();

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 (as shown in the shaded area in Figure 21-3). In our example, if we instantiate our barrier as follows:

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

the output is this:

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

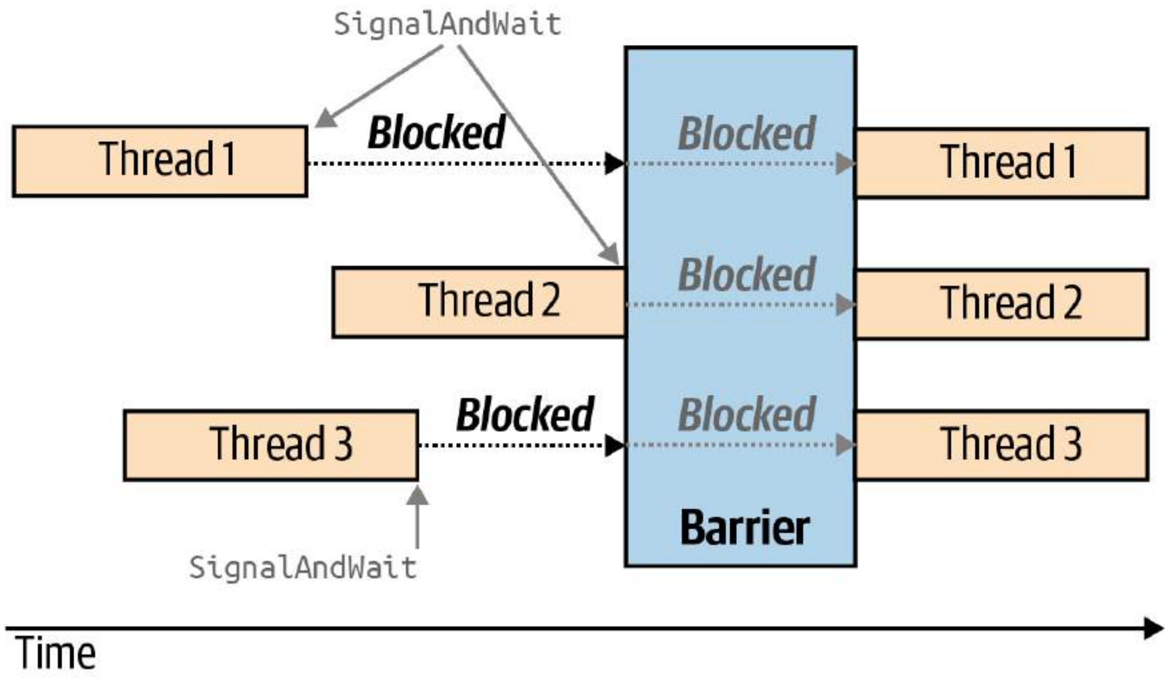

Figure 21-3. Barrier

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

Lazy Initialization

A frequent 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`—regardless of whether 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`. Because this can 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>

The `Lazy<T>` class is available to help with lazy initialization. If instantiated with an argument of `true`, it implements the thread-safe initialization pattern just described.

NOTE

`Lazy<T>` actually implements a micro-optimized 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 beginning 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 prevents a level of indirection, improving performance in cases where you need extreme optimization.
- It offers another mode of initialization in which multiple threads can 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 using advanced techniques that we describe in “Nonblocking Synchronization” and “Lazy Initialization” at *http://albahari.com/threading*. This is an extreme (and rarely needed) optimization 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 be disposed without additional logic.

Thread-Local Storage

Much of this chapter 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 can be clumsy and can alienate 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 four ways to implement thread-local storage. We take a look at them in the following subsections.

[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>` 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 use this in Chapter 22 (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. You can use the same slot 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 with an unnamed slot, obtained by calling `Thread.AllocateDataSlot`:

```
class Test
{
  LocalDataStoreSlot _secSlot = Thread.AllocateDataSlot();
  ...
```

`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 have 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.

AsyncLocal<T>

The approaches to thread-local storage that we've discussed so far are incompatible with asynchronous functions, because after an `await`, execution can resume on a different thread. The `AsyncLocal<T>` class solves this by preserving its value across an `await`:

```
static AsyncLocal<string> _asyncLocalTest = new AsyncLocal<string>();

async void Main()
```

```
{
  _asyncLocalTest.Value = "test";
  await Task.Delay (1000);
  // The following works even if we come back on another thread:
  Console.WriteLine (_asyncLocalTest.Value);   // test
}
```

`AsyncLocal<T>` is still able to keep operations started on separate threads apart, whether initiated by `Thread.Start` or `Task.Run`. The following writes “one one” and “two two”:

```
static AsyncLocal<string> _asyncLocalTest = new AsyncLocal<string>();

void Main()
{
  // Call Test twice on two concurrent threads:
  new Thread (() => Test ("one")).Start();
  new Thread (() => Test ("two")).Start();
}

async void Test (string value)
{
  _asyncLocalTest.Value = value;
  await Task.Delay (1000);
  Console.WriteLine (value + " " + _asyncLocalTest.Value);
}
```

`AsyncLocal<T>` has an interesting and unique nuance: if an `AsyncLocal<T>` object already has a value when a thread is started, the new thread will “inherit” that value:

```
static AsyncLocal<string> _asyncLocalTest = new AsyncLocal<string>();

void Main()
{
  _asyncLocalTest.Value = "test";
  new Thread (AnotherMethod).Start();
}

void AnotherMethod() => Console.WriteLine (_asyncLocalTest.Value);  // test
```

The new thread, however, gets a *copy* of the value, so any changes that it makes will not affect the original:

```
static AsyncLocal<string> _asyncLocalTest = new AsyncLocal<string>();

void Main()
{
  _asyncLocalTest.Value = "test";
  var t = new Thread (AnotherMethod);
  t.Start(); t.Join();
  Console.WriteLine (_asyncLocalTest.Value);   // test  (not ha-ha!)
}

void AnotherMethod() => _asyncLocalTest.Value = "ha-ha!";
```

Keep in mind that the new thread gets a *shallow* copy of the value. So, if you were to replace `Async<string>` with `Async<StringBuilder>` or `Async<List<string>>`, the new thread could clear the `StringBuilder` or add/remove items to the `List<string>`, and this would affect the original.

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.

.NET provides five 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.

Finally, from .NET 6, there's the `PeriodicTimer`, which we will cover first.

PeriodicTimer

`PeriodicTimer` is not really a timer; it's a class to help with asynchronous looping. It's important to consider that since the advent of `async` and `await`, traditional timers are not usually necessary. Instead, the following pattern works nicely:

```
StartPeriodicOperation();

async void StartPeriodicOperation()
{
  while (true)
  {
    await Task.Delay (1000);
    Console.WriteLine ("Tick");   // Do some action
  }
 }
```

NOTE

If you call `StartPeriodicOperation` from a UI thread, it will behave as a single-threaded timer, because the await will always return on the same synchronization context.

You can make it behave as a multithreaded timer simply by adding `.ConfigureAwait(false)` to the await.

`PeriodicTimer` is a class to simplify this pattern:

```
var timer = new PeriodicTimer (TimeSpan.FromSeconds (1));
StartPeriodicOperation();
// Optionally dispose timer when you want to stop looping.

async void StartPeriodicOperation()
{
  while (await timer.WaitForNextTickAsync())
    Console.WriteLine ("Tick");    // Do some action
}
```

PeriodicTimer also allows you to stop the timer by disposing the timer instance. This results in WaitForNextTickAsync returning false, allowing the loop to end.

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;

// 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.
```

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

NOTE

See “Timers” for a discussion on disposing multithreaded timers.

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.

.NET 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:

- An `IComponent` implementation, allowing it to be sited in Visual Studio’s Designer’s component tray
- 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

var 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

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 can fire on a different thread each time it is called. Furthermore, the `Elapsed` event always fires (approximately) on time—regardless of whether the previous `Elapsed` event finished executing. Hence, callbacks or event handlers must be thread-safe.

The precision of multithreaded timers depends on the OS, and is typically in the 10- to 20-millisecond region. If you need greater precision, you can use native interop and call the Windows multimedia timer. This has precision down to one millisecond, and it is defined in *winmm.dll*. First call `timeBeginPeriod` to inform the OS 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. Chapter 24 demonstrates calling external methods with P/Invoke. 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

.NET 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)

> **NOTE**
>
> 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`, `Start`, and `Stop` (and `Tick`, which is equivalent to `Elapsed`)—and are used in a similar manner. However, they differ in how they work internally. Instead of firing timer events on pooled threads, they post the events to the WPF or Windows Forms message loop. 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.BeginInvoke` or `Dispatcher.BeginInvoke`.

Thus, a program employing these timers is not really multithreaded: you end up with the same kind of pseudo-concurrency that's described in Chapter 14 with asynchronous functions that execute on a UI thread. One thread serves all timers as well as the processing UI events, which means that the `Tick` event handler must execute quickly, otherwise the UI becomes unresponsive.

This makes the WPF and Windows Forms timers suitable for small jobs, typically updating some aspect of the UI (e.g., a clock or countdown display).

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 UI requests (or other timer events) are processed.

---

[1] Nuances in the behavior of Windows and the CLR mean that the fairness of the queue can sometimes be violated.

[2] As with locks, the fairness of the queue can sometimes be violated due to nuances in the operating system.