Primitives

03/30/2017 • 8 minutes to read • Contributors all

In this article

The .NET Framework provides a range of synchronization primitives for controlling the interactions of threads and avoiding race conditions. These can be roughly divided into three categories: locking, signaling, and interlocked operations.

The categories are not tidy nor clearly defined: Some synchronization mechanisms have characteristics of multiple categories; events that release a single thread at a time are functionally like locks; the release of any lock can be thought of as a signal; and interlocked operations can be used to construct locks. However, the categories are still useful.

It is important to remember that thread synchronization is cooperative. If even one thread bypasses a synchronization mechanism and accesses the protected resource directly, that synchronization mechanism cannot be effective.

This overview contains the following sections:

- Locking
- Signaling
- Lightweight Synchronization Types

- SpinWait
- Interlocked Operations

Locking

Locks give control of a resource to one thread at a time, or to a specified number of threads. A thread that requests an exclusive lock when the lock is in use blocks until the lock becomes available.

Exclusive Locks

The simplest form of locking is the `lock` statement in C# and the `SyncLock` statement in Visual Basic, which controls access to a block of code. Such a block is frequently referred to as a critical section. The `lock` statement is implemented by using the Monitor.Enter and Monitor.Exit methods, and it uses `try…catch…finally` block to ensure that the lock is released.

In general, using the `lock` or `SyncLock` statement to protect small blocks of code, never spanning more than a single method, is the best way to use the Monitor class. Although powerful, the Monitor class is prone to orphan locks and deadlocks.

Monitor Class

The Monitor class provides additional functionality, which can be used in conjunction with the `lock` statement:

- The TryEnter method allows a thread that is blocked waiting for the resource to give up after a specified interval. It returns a Boolean value indicating success or failure, which can be used to detect and avoid potential deadlocks.

- The Wait method is called by a thread in a critical section. It gives up control of the resource and blocks until the resource is available again.
- The Pulse and PulseAll methods allow a thread that is about to release the lock or to call Wait to put one or more threads into the ready queue, so that they can acquire the lock.

Timeouts on Wait method overloads allow waiting threads to escape to the ready queue.

The Monitor class can provide locking in multiple application domains if the object used for the lock derives from MarshalByRefObject.

Monitor has thread affinity. That is, a thread that entered the monitor must exit by calling Exit or Wait.

The Monitor class is not instantiable. Its methods are static (`Shared` in Visual Basic), and act on an instantiable lock object.

For a conceptual overview, see Monitors.

Mutex Class

Threads request a Mutex by calling an overload of its WaitOne method. Overloads with timeouts are provided, to allow threads to give up the wait. Unlike the Monitor class, a mutex can be either local or global. Global mutexes, also called named mutexes, are visible throughout the operating system, and can be used to synchronize threads in multiple application domains or

processes. Local mutexes derive from MarshalByRefObject, and can be used across application domain boundaries.

In addition, Mutex derives from WaitHandle, which means that it can be used with the signaling mechanisms provided by WaitHandle, such as the WaitAll,

WaitAny, and SignalAndWait methods.

Like Monitor, Mutex has thread affinity. Unlike Monitor, a Mutex is an instantiable object.

For a conceptual overview, see Mutexes.

SpinLock Class

Starting with the .NET Framework 4, you can use the SpinLock class when the overhead required by Monitor degrades performance. When SpinLock encounters a locked critical section, it simply spins in a loop until the lock becomes available. If the lock is held for a very short time, spinning can provide better performance than blocking. However, if the lock is held for more than a few tens of cycles, SpinLock performs just as well as Monitor, but will use more CPU cycles and thus can degrade the performance of other threads or processes.

Other Locks

Locks need not be exclusive. It is often useful to allow a limited number of threads concurrent access to a resource. Semaphores and reader-writer locks are designed to control this kind of pooled resource access.

ReaderWriterLock Class

The ReaderWriterLockSlim class addresses the case where a thread that changes data, the writer, must have exclusive access to a resource. When the writer is not active, any number of readers can access the resource (for example, by calling the EnterReadLock method). When a thread requests exclusive access, (for example, by calling the EnterWriteLock method), subsequent reader requests block until all existing readers have exited the lock, and the writer has entered and exited the lock.

ReaderWriterLockSlim has thread affinity.

For a conceptual overview, see Reader-Writer Locks.

Semaphore Class

The Semaphore class allows a specified number of threads to access a resource. Additional threads requesting the resource block until a thread releases the semaphore.

Like the Mutex class, Semaphore derives from WaitHandle. Also like Mutex, a Semaphore can be either local or global. It can be used across application domain boundaries.

Unlike Monitor, Mutex, and ReaderWriterLock, Semaphore does not have thread affinity. This means it can be used in scenarios where one thread acquires the semaphore and another releases it.

For a conceptual overview, see Semaphore and SemaphoreSlim.

System.Threading.SemaphoreSlim is a lightweight semaphore for synchronization within a single process boundary.

Back to top

Signaling

The simplest way to wait for a signal from another thread is to call the Join method, which blocks until the other thread completes. Join has two overloads that allow the blocked thread to break out of the wait after a specified interval has elapsed.

Wait handles provide a much richer set of waiting and signaling capabilities.

Wait Handles

Wait handles derive from the WaitHandle class, which in turn derives from MarshalByRefObject. Thus, wait handles can be used to synchronize the activities of threads across application domain boundaries.

Threads block on wait handles by calling the instance method WaitOne or one of the static methods WaitAll, WaitAny, or SignalAndWait. How they are released depends on which method was called, and on the kind of wait handles.

For a conceptual overview, see Wait Handles.

Event Wait Handles

Event wait handles include the EventWaitHandle class and its derived classes, AutoResetEvent and ManualResetEvent. Threads are released from an event wait handle when the event wait handle is signaled by calling its Set method or by using the SignalAndWait method.

Event wait handles either reset themselves automatically, like a turnstile that allows only one thread through each time it is signaled, or must be reset manually, like a gate that is closed until signaled and then open until someone closes it. As their names imply, AutoResetEvent and ManualResetEvent represent the former and latter, respectively. System.Threading.ManualResetEventSlim is a lightweight event for synchronization within a single process boundary.

An EventWaitHandle can represent either type of event, and can be either local or global. The derived classes AutoResetEvent and ManualResetEvent are always local.

Event wait handles do not have thread affinity. Any thread can signal an event

wait handle.

For a conceptual overview, see EventWaitHandle, AutoResetEvent, CountdownEvent, ManualResetEvent.

Mutex and Semaphore Classes

Because the Mutex and Semaphore classes derive from WaitHandle, they can be used with the static methods of WaitHandle. For example, a thread can use the WaitAll method to wait until all three of the following are true: an EventWaitHandle is signaled, a Mutex is released, and a Semaphore is released. Similarly, a thread can use the WaitAny method to wait until any one of those conditions is true.

For a Mutex or a Semaphore, being signaled means being released. If either type is used as the first argument of the SignalAndWait method, it is released. In the case of a Mutex, which has thread affinity, an exception is thrown if the calling thread does not own the mutex. As noted previously, semaphores do not have thread affinity.

Barrier

The Barrier class provides a way to cyclically synchronize multiple threads so that they all block at the same point and wait for all other threads to complete. A barrier is useful when one or more threads require the results of another thread before continuing to the next phase of an algorithm. For more information, see Barrier.

Back to top

Lightweight Synchronization Types

Starting with the .NET Framework 4, you can use synchronization primitives that

provide fast performance by avoiding expensive reliance on Win32 kernel objects such as wait handles whenever possible. In general, you should use these types when wait times are short and only when the original synchronization types have been tried and found to be unsatisfactory. The lightweight types cannot be used in scenarios that require cross-process communication.

- System.Threading.SemaphoreSlim is a lightweight version of System.Threading.Semaphore.
- System.Threading.ManualResetEventSlim is a lightweight version of System.Threading.ManualResetEvent.
- System.Threading.CountdownEvent represents an event that becomes signaled when its count is zero.
- System.Threading.Barrier enables multiple threads to synchronize with one another without requiring control by a master thread. A barrier prevents each thread from continuing until all threads have reached a specified point.

Back to top

SpinWait

Starting with the .NET Framework 4, you can use the System.Threading.SpinWait structure when a thread has to wait for an event to be signaled or a condition to be met, but when the actual wait time is expected to be less than the waiting time required by using a wait handle or by otherwise blocking the current thread. By using SpinWait, you can specify a short period of time to spin while waiting, and then yield (for example, by waiting or sleeping) only if the condition was not met in the specified time.

Back to top

Interlocked Operations

Interlocked operations are simple atomic operations performed on a memory location by static methods of the Interlocked class. Those atomic operations include addition, increment and decrement, exchange, conditional exchange depending on a comparison, and read operations for 64-bit values on 32-bit platforms.

> ⓘ **Note**
>
> The guarantee of atomicity is limited to individual operations; when multiple operations must be performed as a unit, a more coarse-grained synchronization mechanism must be used.

Although none of these operations are locks or signals, they can be used to construct locks and signals. Because they are native to the Windows operating system, interlocked operations are extremely fast.

Interlocked operations can be used with volatile memory guarantees to write applications that exhibit powerful non-blocking concurrency. However, they require sophisticated, low-level programming, so for most purposes, simple locks are a better choice.

For a conceptual overview, see Interlocked Operations.

See Also

Synchronizing Data for Multithreading
Monitors
Mutexes

Semaphore and SemaphoreSlim

EventWaitHandle, AutoResetEvent, CountdownEvent, ManualResetEvent

Wait Handles

Interlocked Operations

Reader-Writer Locks

Barrier

SpinWait

SpinLock