Threading Objects and Features

.NET Framework (current version)

The .NET Framework provides a number of objects that help you create and manage multithreaded applications. Managed threads are represented by the Thread class. The ThreadPool class provides easy creation and management of multithreaded background tasks. The BackgroundWorker class does the same for tasks that interact with the user interface. The Timer class executes background tasks at timed intervals.

In addition, there are a number of classes that synchronize activities of threads, including the Semaphore and EventWaitHandle classes introduced in the .NET Framework version 2.0. The features of these classes are compared in Overview of Synchronization Primitives.

In This Section

The Managed Thread Pool
: Explains the **ThreadPool** class, which enables you to request a thread to execute a task without having to do any thread management yourself.

Timers
: Explains how to use a **Timer** to specify a delegate to be called at a specified time.

Monitors
: Explains how to use the **Monitor** class to synchronize access to a member or to build your own thread management types.

Wait Handles
: Describes the WaitHandle class, the abstract base class for event wait handles, mutexes, and semaphores, which enables waiting for multiple synchronization events.

EventWaitHandle, AutoResetEvent, CountdownEvent, ManualResetEvent
: Describes managed event wait handles, which are used to synchronize thread activities by signaling and waiting for signals.

Mutexes
: Explains how to use a Mutex to synchronize access to an object or to build your own synchronization mechanisms.

Interlocked Operations
: Explains how to use the Interlocked class to increment or decrement a value and store the value in a single atomic operation.

Reader-Writer Locks
: Defines a lock that implements single-writer/multiple-reader semantics.

Semaphore and SemaphoreSlim
: Describes Semaphore objects and explains how to use them to control access to limited resources.

Overview of Synchronization Primitives
: Compares the features of the .NET Framework classes provided for locking and synchronizing managed threads.

Barrier (.NET Framework)
: Describes Barrier objects that implement the barrier pattern for coordination of threads in phased operations.

SpinLock
: Describes SpinLock, a lightweight alternative to the Monitor class for certain low-level scenarios.

SpinWait
: Describes SpinWait, a low level synchronization primitive that performs busy spinning prior to initiating a kernel-based wait.

Reference

Thread
: Provides reference documentation for the **Thread** class, which represents a managed thread, whether it came from unmanaged code or was created in a managed application.

BackgroundWorker
: Enables background tasks that interact with the user interface, communicating via events raised on the user-interface thread.

Related Sections

Asynchronous File I/O
: Describes how I/O asynchronous completion ports use the thread pool to require processing only when an input/output operation completes.

Task Parallel Library (TPL)
: Describes the recommended approach for multithreaded programming in the .NET Framework 4 and later.

The Managed Thread Pool

.NET Framework (current version)

The ThreadPool class provides your application with a pool of worker threads that are managed by the system, allowing you to concentrate on application tasks rather than thread management. If you have short tasks that require background processing, the managed thread pool is an easy way to take advantage of multiple threads. For example, beginning with the .NET Framework 4 you can create Task and Task(Of TResult) objects, which perform asynchronous tasks on thread pool threads.

Note

Starting with the .NET Framework 2.0 Service Pack 1, the throughput of the thread pool is significantly improved in three key areas that were identified as bottlenecks in previous releases of the .NET Framework: queuing tasks, dispatching thread pool threads, and dispatching I/O completion threads. To use this functionality, your application should target the .NET Framework 3.5 or later.

For background tasks that interact with the user interface, the .NET Framework version 2.0 also provides the BackgroundWorker class, which communicates using events raised on the user interface thread.

The .NET Framework uses thread pool threads for many purposes, including asynchronous I/O completion, timer callbacks, registered wait operations, asynchronous method calls using delegates, and System.Net socket connections.

When Not to Use Thread Pool Threads

There are several scenarios in which it is appropriate to create and manage your own threads instead of using thread pool threads:

- You require a foreground thread.
- You require a thread to have a particular priority.
- You have tasks that cause the thread to block for long periods of time. The thread pool has a maximum number of threads, so a large number of blocked thread pool threads might prevent tasks from starting.
- You need to place threads into a single-threaded apartment. All ThreadPool threads are in the multithreaded apartment.
- You need to have a stable identity associated with the thread, or to dedicate a thread to a task.

Thread Pool Characteristics

Thread pool threads are background threads. See Foreground and Background Threads. Each thread uses the default

stack size, runs at the default priority, and is in the multithreaded apartment.

There is only one thread pool per process.

Exceptions in Thread Pool Threads

Unhandled exceptions on thread pool threads terminate the process. There are three exceptions to this rule:

- A ThreadAbortException is thrown in a thread pool thread, because Abort was called.
- An AppDomainUnloadedException is thrown in a thread pool thread, because the application domain is being unloaded.
- The common language runtime or a host process terminates the thread.

For more information, see Exceptions in Managed Threads.

Note

In the .NET Framework versions 1.0 and 1.1, the common language runtime silently traps unhandled exceptions in thread pool threads. This might corrupt application state and eventually cause applications to hang, which might be very difficult to debug.

Maximum Number of Thread Pool Threads

The number of operations that can be queued to the thread pool is limited only by available memory; however, the thread pool limits the number of threads that can be active in the process simultaneously. Beginning with the .NET Framework 4, the default size of the thread pool for a process depends on several factors, such as the size of the virtual address space. A process can call the GetMaxThreads method to determine the number of threads.

You can control the maximum number of threads by using the GetMaxThreads and SetMaxThreads methods.

Note

In the .NET Framework versions 1.0 and 1.1, the size of the thread pool cannot be set from managed code. Code that hosts the common language runtime can set the size using **CorSetMaxThreads**, defined in mscoree.h.

Thread Pool Minimums

The thread pool provides new worker threads or I/O completion threads on demand until it reaches a specified minimum for each category. You can use the GetMinThreads method to obtain these minimum values.

> **Note**
>
> When demand is low, the actual number of thread pool threads can fall below the minimum values.

When a minimum is reached, the thread pool can create additional threads or wait until some tasks complete. Beginning with the .NET Framework 4, the thread pool creates and destroys worker threads in order to optimize throughput, which is defined as the number of tasks that complete per unit of time. Too few threads might not make optimal use of available resources, whereas too many threads could increase resource contention.

> **Caution**
>
> You can use the SetMinThreads method to increase the minimum number of idle threads. However, unnecessarily increasing these values can cause performance problems. If too many tasks start at the same time, all of them might appear to be slow. In most cases the thread pool will perform better with its own algorithm for allocating threads.

Skipping Security Checks

The thread pool also provides the ThreadPool.UnsafeQueueUserWorkItem and ThreadPool.UnsafeRegisterWaitForSingleObject methods. Use these methods only when you are certain that the caller's stack is irrelevant to any security checks performed during the execution of the queued task. QueueUserWorkItem and RegisterWaitForSingleObject both capture the caller's stack, which is merged into the stack of the thread pool thread when the thread begins to execute a task. If a security check is required, the entire stack must be checked. Although the check provides safety, it also has a performance cost.

Using the Thread Pool

Beginning with the .NET Framework 4, the easiest way to use the thread pool is to use the Task Parallel Library (TPL). By default, parallel library types like Task and Task(Of TResult) use thread pool threads to run tasks. You can also use the thread pool by calling ThreadPool.QueueUserWorkItem from managed code (or **CorQueueUserWorkItem** from unmanaged code) and passing a WaitCallback delegate representing the method that performs the task. Another way to use the thread pool is to queue work items that are related to a wait operation by using the ThreadPool.RegisterWaitForSingleObject method and passing a WaitHandle that, when signaled or when timed out, calls the method represented by the WaitOrTimerCallback delegate. Thread pool threads are used to invoke callback methods.

ThreadPool Examples

The code examples in this section demonstrate the thread pool by using the Task class, the ThreadPool.QueueUserWorkItem method, and the ThreadPool.RegisterWaitForSingleObject method.

- Executing Asynchronous Tasks with the Task Parallel Library

- Executing Code Asynchronously with QueueUserWorkItem
- Supplying Task Data for QueueUserWorkItem
- Using RegisterWaitForSingleObject

Executing Asynchronous Tasks with the Task Parallel Library

The following example shows how to create and use a Task object by calling the TaskFactory.StartNew method. For an example that uses the Task(Of TResult) class to return a value from an asynchronous task, see How to: Return a Value from a Task.

VB

```
Imports System.Threading
Imports System.Threading.Tasks

Module Example
    Public Sub Main()
        Dim action As Action(Of Object) =
              Sub(obj As Object)
                 Console.WriteLine("Task={0}, obj={1}, Thread={2}",
                 Task.CurrentId, obj,
                 Thread.CurrentThread.ManagedThreadId)
              End Sub

        ' Construct an unstarted task
        Dim t1 As New Task(action, "alpha")

        ' Construct a started task
        Dim t2 As Task = Task.Factory.StartNew(action, "beta")
        ' Block the main thread to demonstate that t2 is executing
        t2.Wait()

        ' Launch t1
        t1.Start()
        Console.WriteLine("t1 has been launched. (Main Thread={0})",
                          Thread.CurrentThread.ManagedThreadId)
        ' Wait for the task to finish.
        t1.Wait()

        ' Construct a started task using Task.Run.
        Dim taskData As String = "delta"
        Dim t3 As Task = Task.Run(Sub()
                                     Console.WriteLine("Task={0}, obj={1}, Thread=
{2}",

                                     Task.CurrentId, taskData,
                                     Thread.CurrentThread.ManagedThreadId)
                                  End Sub)
        ' Wait for the task to finish.
        t3.Wait()

        ' Construct an unstarted task
```

```
        Dim t4 As New Task(action, "gamma")
        ' Run it synchronously
        t4.RunSynchronously()
        ' Although the task was run synchronously, it is a good practice
        ' to wait for it in the event exceptions were thrown by the task.
        t4.Wait()
    End Sub
End Module
' The example displays output like the following:
'       Task=1, obj=beta, Thread=3
'       t1 has been launched. (Main Thread=1)
'       Task=2, obj=alpha, Thread=3
'       Task=3, obj=delta, Thread=3
'       Task=4, obj=gamma, Thread=1
```

Executing Code Asynchronously with QueueUserWorkItem

The following example queues a very simple task, represented by the `ThreadProc` method, using the QueueUserWorkItem method.

VB

```
Imports System
Imports System.Threading

Public Class Example
    Public Shared Sub Main()
        ' Queue the task.
        ThreadPool.QueueUserWorkItem(New WaitCallback(AddressOf ThreadProc))

        Console.WriteLine("Main thread does some work, then sleeps.")
        ' If you comment out the Sleep, the main thread exits before
        ' the thread pool task runs.  The thread pool uses background
        ' threads, which do not keep the application running.  (This
        ' is a simple example of a race condition.)
        Thread.Sleep(1000)

        Console.WriteLine("Main thread exits.")
    End Sub

    ' This thread procedure performs the task.
    Shared Sub ThreadProc(stateInfo As Object)
        ' No state object was passed to QueueUserWorkItem, so
        ' stateInfo is null.
        Console.WriteLine("Hello from the thread pool.")
    End Sub
End Class
```

Supplying Task Data for QueueUserWorkItem

The following code example uses the QueueUserWorkItem method to queue a task and supply the data for the task.

VB

```
Imports System
Imports System.Threading

' TaskInfo holds state information for a task that will be
' executed by a ThreadPool thread.
Public class TaskInfo
    ' State information for the task.  These members
    ' can be implemented as read-only properties, read/write
    ' properties with validation, and so on, as required.
    Public Boilerplate As String
    Public Value As Integer

    ' Public constructor provides an easy way to supply all
    ' the information needed for the task.
    Public Sub New(text As String, number As Integer)
        Boilerplate = text
        Value = number
    End Sub
End Class

Public Class Example
    Public Shared Sub Main()
        ' Create an object containing the information needed
        ' for the task.
        Dim ti As New TaskInfo("This report displays the number {0}.", 42)

        ' Queue the task and data.
        If ThreadPool.QueueUserWorkItem(New WaitCallback(AddressOf ThreadProc), ti)
Then
            Console.WriteLine("Main thread does some work, then sleeps.")

            ' If you comment out the Sleep, the main thread exits before
            ' the ThreadPool task has a chance to run.  ThreadPool uses
            ' background threads, which do not keep the application
            ' running.  (This is a simple example of a race condition.)
            Thread.Sleep(1000)

            Console.WriteLine("Main thread exits.")
        Else
            Console.WriteLine("Unable to queue ThreadPool request.")
        End If
    End Sub

    ' The thread procedure performs the independent task, in this case
    ' formatting and printing a very simple report.
    '
    Shared Sub ThreadProc(stateInfo As Object)
```

```
        Dim ti As TaskInfo = CType(stateInfo, TaskInfo)
        Console.WriteLine(ti.Boilerplate, ti.Value)
    End Sub
End Class
```

Using RegisterWaitForSingleObject

The following example demonstrates several threading features.

- Queuing a task for execution by ThreadPool threads, with the RegisterWaitForSingleObject method.
- Signaling a task to execute, with AutoResetEvent. See EventWaitHandle, AutoResetEvent, CountdownEvent, ManualResetEvent.
- Handling both time-outs and signals with a WaitOrTimerCallback delegate.
- Canceling a queued task with RegisteredWaitHandle.

VB

```
Imports System
Imports System.Threading

' TaskInfo contains data that will be passed to the callback
' method.
Public Class TaskInfo
    public Handle As RegisteredWaitHandle = Nothing
    public OtherInfo As String = "default"
End Class

Public Class Example
    Public Shared Sub Main()
        ' The main thread uses AutoResetEvent to signal the
        ' registered wait handle, which executes the callback
        ' method.
        Dim ev As New AutoResetEvent(false)

        Dim ti As New TaskInfo()
        ti.OtherInfo = "First task"
        ' The TaskInfo for the task includes the registered wait
        ' handle returned by RegisterWaitForSingleObject.  This
        ' allows the wait to be terminated when the object has
        ' been signaled once (see WaitProc).
        ti.Handle = ThreadPool.RegisterWaitForSingleObject( _
            ev, _
            New WaitOrTimerCallback(AddressOf WaitProc), _
            ti, _
            1000, _
            false _
        )
```

```
        ' The main thread waits about three seconds, to demonstrate
        ' the time-outs on the queued task, and then signals.
        Thread.Sleep(3100)
        Console.WriteLine("Main thread signals.")
        ev.Set()

        ' The main thread sleeps, which should give the callback
        ' method time to execute.  If you comment out this line, the
        ' program usually ends before the ThreadPool thread can execute.
        Thread.Sleep(1000)
        ' If you start a thread yourself, you can wait for it to end
        ' by calling Thread.Join.  This option is not available with
        ' thread pool threads.
    End Sub

    ' The callback method executes when the registered wait times out,
    ' or when the WaitHandle (in this case AutoResetEvent) is signaled.
    ' WaitProc unregisters the WaitHandle the first time the event is
    ' signaled.
    Public Shared Sub WaitProc(state As Object, timedOut As Boolean)
        ' The state object must be cast to the correct type, because the
        ' signature of the WaitOrTimerCallback delegate specifies type
        ' Object.
        Dim ti As TaskInfo = CType(state, TaskInfo)

        Dim cause As String = "TIMED OUT"
        If Not timedOut Then
            cause = "SIGNALED"
            ' If the callback method executes because the WaitHandle is
            ' signaled, stop future execution of the callback method
            ' by unregistering the WaitHandle.
            If Not ti.Handle Is Nothing Then
                ti.Handle.Unregister(Nothing)
            End If
        End If

        Console.WriteLine("WaitProc( {0} ) executes on thread {1}; cause = {2}.", _
            ti.OtherInfo, _
            Thread.CurrentThread.GetHashCode().ToString(), _
            cause _
        )
    End Sub
End Class
```

See Also

ThreadPool
Task

Task(Of TResult)
Task Parallel Library (TPL)
Task Parallel Library (TPL)
How to: Return a Value from a Task
Threading Objects and Features
Threads and Threading
Asynchronous File I/O
Timers

Timers

.NET Framework (current version)

Timers are lightweight objects that enable you to specify a delegate to be called at a specified time. A thread in the thread pool performs the wait operation.

Using the System.Threading.Timer class is straightforward. You create a **Timer**, passing a TimerCallback delegate to the callback method, an object representing state that will be passed to the callback, an initial raise time, and a time representing the period between callback invocations. To cancel a pending timer, call the **Timer.Dispose** function.

Note

There are two other timer classes. The System.Windows.Forms.Timer class is a control that works with visual designers and is meant to be used in user interface contexts; it raises events on the user interface thread. The System.Timers.Timer class derives from Component, so it can be used with visual designers; it also raises events, but it raises them on a ThreadPool thread. The System.Threading.Timer class makes callbacks on a ThreadPool thread and does not use the event model at all. It also provides a state object to the callback method, which the other timers do not. It is extremely lightweight.

The following code example starts a timer that starts after one second (1000 milliseconds) and ticks every second until you press the **Enter** key. The variable containing the reference to the timer is a class-level field, to ensure that the timer is not subject to garbage collection while it is still running. For more information on aggressive garbage collection, see KeepAlive.

VB

```
Imports System
Imports System.Threading

Public Class Example
    Private Shared ticker As Timer

    Public Shared Sub TimerMethod(state As Object)
        Console.Write(".")
    End Sub

    Public Shared Sub Main()
        ticker = New Timer(AddressOf TimerMethod, Nothing, 1000, 1000)

        Console.WriteLine("Press the Enter key to end the program.")
        Console.ReadLine()
    End Sub
End Class
```

See Also

Timer

Threading Objects and Features

Monitor Class

.NET Framework (current version)

Provides a mechanism that synchronizes access to objects.

Namespace: System.Threading
Assembly: mscorlib (in mscorlib.dll)

Inheritance Hierarchy

System.Object
  System.Threading.Monitor

Syntax

VB

```
<ComVisibleAttribute(True)>
<HostProtectionAttribute(SecurityAction.LinkDemand, Synchronization := True,
        ExternalThreading := True)>
Public NotInheritable Class Monitor
```

Methods

	Name	Description
S	Enter(Object)	Acquires an exclusive lock on the specified object.
S	Enter(Object, Boolean)	Acquires an exclusive lock on the specified object, and atomically sets a value that indicates whether the lock was taken.
S	Exit(Object)	Releases an exclusive lock on the specified object.
S	IsEntered(Object)	Determines whether the current thread holds the lock on the specified object.
S	Pulse(Object)	Notifies a thread in the waiting queue of a change in the locked object's state.
S	PulseAll(Object)	Notifies all waiting threads of a change in the object's state.

S	TryEnter(Object)	Attempts to acquire an exclusive lock on the specified object.
S	TryEnter(Object, Boolean)	Attempts to acquire an exclusive lock on the specified object, and atomically sets a value that indicates whether the lock was taken.
S	TryEnter(Object, Int32)	Attempts, for the specified number of milliseconds, to acquire an exclusive lock on the specified object.
S	TryEnter(Object, Int32, Boolean)	Attempts, for the specified number of milliseconds, to acquire an exclusive lock on the specified object, and atomically sets a value that indicates whether the lock was taken.
S	TryEnter(Object, TimeSpan)	Attempts, for the specified amount of time, to acquire an exclusive lock on the specified object.
S	TryEnter(Object, TimeSpan, Boolean)	Attempts, for the specified amount of time, to acquire an exclusive lock on the specified object, and atomically sets a value that indicates whether the lock was taken.
S	Wait(Object)	Releases the lock on an object and blocks the current thread until it reacquires the lock.
S	Wait(Object, Int32)	Releases the lock on an object and blocks the current thread until it reacquires the lock. If the specified time-out interval elapses, the thread enters the ready queue.
S	Wait(Object, Int32, Boolean)	Releases the lock on an object and blocks the current thread until it reacquires the lock. If the specified time-out interval elapses, the thread enters the ready queue. This method also specifies whether the synchronization domain for the context (if in a synchronized context) is exited before the wait and reacquired afterward.
S	Wait(Object, TimeSpan)	Releases the lock on an object and blocks the current thread until it reacquires the lock. If the specified time-out interval elapses, the thread enters the ready queue.
S	Wait(Object, TimeSpan, Boolean)	Releases the lock on an object and blocks the current thread until it reacquires the lock. If the specified time-out interval elapses, the thread enters the ready queue. Optionally exits the synchronization domain for the synchronized context before the wait and reacquires the domain afterward.

Remarks

The Monitor class allows you to synchronize access to a region of code by taking and releasing a lock on a particular object by calling the Monitor.Enter, Monitor.TryEnter, and Monitor.Exit methods. Object locks provide the ability to restrict access to a block of code, commonly called a critical section. While a thread owns the lock for an object, no other thread can acquire that lock. You can also use the Monitor class to ensure that no other thread is allowed to access a

section of application code being executed by the lock owner, unless the other thread is executing the code using a different locked object.

In this article:

The Monitor class: An overview

Monitor has the following features:

- It is associated with an object on demand.
- It is unbound, which means it can be called directly from any context.
- An instance of the Monitor class cannot be created; the methods of the Monitor class are all static. Each method is passed the synchronized object that controls access to the critical section.

Note

Use the Monitor class to lock objects other than strings (that is, reference types other than String), not value types. For details, see the overloads of the Enter method and The lock object section later in this article.

The following table describes the actions that can be taken by threads that access synchronized objects:

Action	Description
Enter, TryEnter	Acquires a lock for an object. This action also marks the beginning of a critical section. No other thread can enter the critical section unless it is executing the instructions in the critical section using a different locked object.
Wait	Releases the lock on an object in order to permit other threads to lock and access the object. The calling thread waits while another thread accesses the object. Pulse signals are used to notify waiting threads about changes to an object's state.
Pulse (signal), PulseAll	Sends a signal to one or more waiting threads. The signal notifies a waiting thread that the state of the locked object has changed, and the owner of the lock is ready to release the lock. The waiting thread is placed in the object's ready queue so that it might eventually receive the lock for the object. Once the thread has the lock, it can check the new state of the object to see if the required state has been reached.
Exit	Releases the lock on an object. This action also marks the end of a critical section protected by the locked object.

Beginning with the .NET Framework 4, there are two sets of overloads for the Enter and TryEnter methods. One set of overloads has a **ref** (in C#) or **ByRef** (in Visual Basic) Boolean parameter that is atomically set to **true** if the lock is acquired, even if an exception is thrown when acquiring the lock. Use these overloads if it is critical to release the lock in all cases, even when the resources the lock is protecting might not be in a consistent state.

The lock object

The Monitor class consists of **static** (in C#) or **Shared** (in Visual Basic) methods that operate on an object that controls access to the critical section. The following information is maintained for each synchronized object:

- A reference to the thread that currently holds the lock.
- A reference to a ready queue, which contains the threads that are ready to obtain the lock.
- A reference to a waiting queue, which contains the threads that are waiting for notification of a change in the state of the locked object.

Monitor locks objects (that is, reference types), not value types. While you can pass a value type to Enter and Exit, it is boxed separately for each call. Since each call creates a separate object, Enter never blocks, and the code it is supposedly protecting is not really synchronized. In addition, the object passed to Exit is different from the object passed to Enter, so Monitor throws SynchronizationLockException exception with the message "Object synchronization method was called from an unsynchronized block of code."

The following example illustrates this problem. It launches ten tasks, each of which just sleeps for 250 milliseconds. Each task then updates a counter variable, `nTasks`, which is intended to count the number of tasks that actually launched and executed. Because `nTasks` is a global variable that can be updated by multiple tasks simultaneously, a monitor is used to protect it from simultaneous modification by multiple tasks. However, as the output from the example shows, each of the tasks throws a SynchronizationLockException exception.

VB

```
Imports System.Collections.Generic
Imports System.Threading
Imports System.Threading.Tasks

Module Example
   Public Sub Main()
      Dim nTasks As Integer = 0
      Dim tasks As New List(Of Task)()

      Try
         For ctr As Integer = 0 To 9
            tasks.Add(Task.Run( Sub()
                                   ' Instead of doing some work, just sleep.
                                   Thread.Sleep(250)
                                   ' Increment the number of tasks.
                                   Monitor.Enter(nTasks)
                                   Try
                                      nTasks += 1
                                   Finally
                                      Monitor.Exit(nTasks)
```

```
                                    End Try
                                End Sub))
            Next
            Task.WaitAll(tasks.ToArray())
            Console.WriteLine("{0} tasks started and executed.", nTasks)
         Catch e As AggregateException
            Dim msg AS String = String.Empty
            For Each ie In e.InnerExceptions
               Console.WriteLine("{0}", ie.GetType().Name)
               If Not msg.Contains(ie.Message) Then
                  msg += ie.Message + Environment.NewLine
               End If
            Next
            Console.WriteLine(vbCrLf + "Exception Message(s):")
            Console.WriteLine(msg)
         End Try
      End Sub
   End Module
   ' The example displays the following output:
   '    SynchronizationLockException
   '    SynchronizationLockException
   '    SynchronizationLockException
   '    SynchronizationLockException
   '    SynchronizationLockException
   '    SynchronizationLockException
   '    SynchronizationLockException
   '    SynchronizationLockException
   '    SynchronizationLockException
   '    SynchronizationLockException
   '
   '    Exception Message(s):
   '    Object synchronization method was called from an unsynchronized block of code.
```

Each task throws a SynchronizationLockException exception because the `nTasks` variable is boxed before the call to the Monitor.Enter method in each task. In other words, each method call is passed a separate variable that is independent of the others. `nTasks` is boxed again in the call to the Monitor.Exit method. Once again, this creates ten new boxed variables, which are independent of each other, `nTasks`, and the ten boxed variables created in the call to the Monitor.Enter method. The exception is thrown, then, because our code is attempting to release a lock on a newly created variable that was not previously locked.

Although you can box a value type variable before calling Enter and Exit, as shown in the following example, and pass the same boxed object to both methods, there is no advantage to doing this. Changes to the unboxed variable are not reflected in the boxed copy, and there is no way to change the value of the boxed copy.

VB

```
Imports System.Collections.Generic
Imports System.Threading
Imports System.Threading.Tasks

Module Example
   Public Sub Main()
      Dim nTasks As Integer = 0
```

```
        Dim o As Object = nTasks
        Dim tasks As New List(Of Task)()

        Try
           For ctr As Integer = 0 To 9
              tasks.Add(Task.Run( Sub()
                                     ' Instead of doing some work, just sleep.
                                     Thread.Sleep(250)
                                     ' Increment the number of tasks.
                                     Monitor.Enter(o)
                                     Try
                                        nTasks += 1
                                     Finally
                                        Monitor.Exit(o)
                                     End Try
                                  End Sub))
           Next
           Task.WaitAll(tasks.ToArray())
           Console.WriteLine("{0} tasks started and executed.", nTasks)
        Catch e As AggregateException
           Dim msg AS String = String.Empty
           For Each ie In e.InnerExceptions
              Console.WriteLine("{0}", ie.GetType().Name)
              If Not msg.Contains(ie.Message) Then
                 msg += ie.Message + Environment.NewLine
              End If
           Next
           Console.WriteLine(vbCrLf + "Exception Message(s):")
           Console.WriteLine(msg)
        End Try
     End Sub
  End Module
  ' The example displays the following output:
  '       10 tasks started and executed.
```

When selecting an object on which to synchronize, you should lock only on private or internal objects. Locking on external objects might result in deadlocks, because unrelated code could choose the same objects to lock on for different purposes.

Note that you can synchronize on an object in multiple application domains if the object used for the lock derives from MarshalByRefObject.

The critical section

Use the Enter and Exit methods to mark the beginning and end of a critical section.

> **Note**
>
> The functionality provided by the Enter and Exit methods is identical to that provided by the lock statement in C# and the SyncLock statement in Visual Basic, except that the language constructs wrap the Monitor.Enter(Object,

Boolean) method overload and the Monitor.Exit method in a **try**...**finally** block to ensure that the monitor is released.

If the critical section is a set of contiguous instructions, then the lock acquired by the Enter method guarantees that only a single thread can execute the enclosed code with the locked object. In this case, we recommend that you place that code in a **try** block and place the call to the Exit method in a **finally** block. This ensures that the lock is released even if an exception occurs. The following code fragment illustrates this pattern.

VB

```
' Define the lock object.
Dim obj As New Object()

' Define the critical section.
Monitor.Enter(obj)
Try
   ' Code to execute one thread at a time.

' catch blocks go here.
Finally
   Monitor.Exit(obj)
End Try
```

This facility is typically used to synchronize access to a static or instance method of a class.

If a critical section spans an entire method, the locking facility can be achieved by placing the System.Runtime.CompilerServices.MethodImplAttribute on the method, and specifying the Synchronized value in the constructor of System.Runtime.CompilerServices.MethodImplAttribute. When you use this attribute, the Enter and Exit method calls are not needed. The following code fragment illustrates this pattern:

VB

```
<MethodImplAttribute(MethodImplOptions.Synchronized)>
Sub MethodToLock()
   ' Method implementation.
End Sub
```

Note that the attribute causes the current thread to hold the lock until the method returns; if the lock can be released sooner, use the Monitor class, the C# lock statement, or the Visual Basic SyncLock statement inside of the method instead of the attribute.

While it is possible for the Enter and Exit statements that lock and release a given object to cross member or class boundaries or both, this practice is not recommended.

Pulse, PulseAll, and Wait

Once a thread owns the lock and has entered the critical section that the lock protects, it can call the Monitor.Wait, Monitor.Pulse, and Monitor.PulseAll methods.

Wait releases the lock if it is held, allows a waiting thread or threads to obtain the lock and enter the critical section,

and waits to be notified by a call to the Monitor.Pulse or Monitor.PulseAll method. When Wait is notified, it returns and obtains the lock again.

Both Pulse and PulseAll signal for the next thread in the wait queue to proceed.

Monitors and wait handles

It is important to note the distinction between the use of the Monitor class and WaitHandle objects.

- The Monitor class is purely managed, fully portable, and might be more efficient in terms of operating-system resource requirements.
- WaitHandle objects represent operating-system waitable objects, are useful for synchronizing between managed and unmanaged code, and expose some advanced operating-system features like the ability to wait on many objects at once.

Examples

The following example uses the Monitor class to synchronize access to a single instance of a random number generator represented by the Random class. The example creates ten tasks, each of which executes asynchronously on a thread pool thread. Each task generates 10,000 random numbers, calculates their average, and updates two procedure-level variables that maintain a running total of the number of random numbers generated and their sum. After all tasks have executed, these two values are then used to calculate the overall mean.

VB

```
Imports System.Collections.Generic
Imports System.Threading
Imports System.Threading.Tasks

Module Example
   Public Sub Main()
      Dim tasks As New List(Of Task)()
      Dim rnd As New Random()
      Dim total As Long = 0
      Dim n As Integer = 0

      For taskCtr As Integer = 0 To 9
         tasks.Add(Task.Run( Sub()
                                Dim values(9999) As Integer
                                Dim taskTotal As Integer = 0
                                Dim taskN As Integer = 0
                                Dim ctr As Integer = 0
                                Monitor.Enter(rnd)
                                   ' Generate 10,000 random integers.
                                   For ctr = 0 To 9999
                                      values(ctr) = rnd.Next(0, 1001)
                                   Next
```

```
                                Monitor.Exit(rnd)
                                taskN = ctr
                                For Each value in values
                                   taskTotal += value
                                Next

                                Console.WriteLine("Mean for task {0,2}: {1:N2}
(N={2:N0})",
                                                  Task.CurrentId, taskTotal/taskN,
                                                  taskN)
                                Interlocked.Add(n, taskN)
                                Interlocked.Add(total, taskTotal)
                             End Sub ))
      Next

      Try
         Task.WaitAll(tasks.ToArray())
         Console.WriteLine()
         Console.WriteLine("Mean for all tasks: {0:N2} (N={1:N0})",
                           (total * 1.0)/n, n)
      Catch e As AggregateException
         For Each ie In e.InnerExceptions
            Console.WriteLine("{0}: {1}", ie.GetType().Name, ie.Message)
         Next
      End Try
   End Sub
End Module
' The example displays output like the following:
'       Mean for task  1: 499.04 (N=10,000)
'       Mean for task  2: 500.42 (N=10,000)
'       Mean for task  3: 499.65 (N=10,000)
'       Mean for task  8: 502.59 (N=10,000)
'       Mean for task  5: 502.75 (N=10,000)
'       Mean for task  4: 494.88 (N=10,000)
'       Mean for task  7: 499.22 (N=10,000)
'       Mean for task 10: 496.45 (N=10,000)
'       Mean for task  6: 499.75 (N=10,000)
'       Mean for task  9: 502.79 (N=10,000)
'
'       Mean for all tasks: 499.75 (N=100,000)
```

Because they can be accessed from any task running on a thread pool thread, access to the variables `total` and `n` must also be synchronized. The Interlocked.Add method is used for this purpose.

The following example demonstrates the combined use of the Monitor class (implemented with the **lock** or **SyncLock** language construct), the Interlocked class, and the AutoResetEvent class. It defines two **internal** (in C#) or **Friend** (in Visual Basic) classes, `SyncResource` and `UnSyncResource`, that respectively provide synchronized and unsynchronized access to a resource. To ensure that the example illustrates the difference between the synchronized and unsynchronized access (which could be the case if each method call completes rapidly), the method includes a random delay: for threads whose Thread.ManagedThreadId property is even, the method calls Thread.Sleep to introduce a delay of 2,000 milliseconds. Note that, because the `SyncResource` class is not public, none of the client code takes a lock on the synchronized resource; the internal class itself takes the lock. This prevents malicious code from taking a lock on a public

object.

VB

```
Imports System.Threading

Friend Class SyncResource
    ' Use a monitor to enforce synchronization.
    Public Sub Access()
        SyncLock Me
            Console.WriteLine("Staring synchronized resource access on thread #{0}",
                              Thread.CurrentThread.ManagedThreadId)
            If Thread.CurrentThread.ManagedThreadId Mod 2 = 0 Then
                Thread.Sleep(2000)
            End If
            Thread.Sleep(200)
            Console.WriteLine("Stopping synchronized resource access on thread #{0}",
                              Thread.CurrentThread.ManagedThreadId)
        End SyncLock
    End Sub
End Class

Friend Class UnSyncResource
    ' Do not enforce synchronization.
    Public Sub Access()
        Console.WriteLine("Starting unsynchronized esource access on Thread #{0}",
                          Thread.CurrentThread.ManagedThreadId)
        If Thread.CurrentThread.ManagedThreadId Mod 2 = 0 Then
            Thread.Sleep(2000)
        End If
        Thread.Sleep(200)
        Console.WriteLine("Stopping unsynchronized resource access on thread #{0}",
                          Thread.CurrentThread.ManagedThreadId)
    End Sub
End Class

Public Module App
    Private numOps As Integer
    Private opsAreDone As New AutoResetEvent(False)
    Private SyncRes As New SyncResource()
    Private UnSyncRes As New UnSyncResource()

    Public Sub Main()
        ' Set the number of synchronized calls.
        numOps = 5
        For ctr As Integer = 0 To 4
            ThreadPool.QueueUserWorkItem(New WaitCallback(AddressOf SyncUpdateResource))
        Next
        ' Wait until this WaitHandle is signaled.
        opsAreDone.WaitOne()
        Console.WriteLine(vbTab + vbNewLine + "All synchronized operations have
completed.")
        Console.WriteLine()
```

```
        numOps = 5
        ' Reset the count for unsynchronized calls.
        For ctr As Integer = 0 To 4
            ThreadPool.QueueUserWorkItem(New WaitCallback(AddressOf
UnSyncUpdateResource))
        Next

        ' Wait until this WaitHandle is signaled.
        opsAreDone.WaitOne()
        Console.WriteLine(vbTab + vbNewLine + "All unsynchronized thread operations
have completed.")
    End Sub

    Sub SyncUpdateResource()
        ' Call the internal synchronized method.
        SyncRes.Access()

        ' Ensure that only one thread can decrement the counter at a time.
        If Interlocked.Decrement(numOps) = 0 Then
            ' Announce to Main that in fact all thread calls are done.
            opsAreDone.Set()
        End If
    End Sub

    Sub UnSyncUpdateResource()
        ' Call the unsynchronized method.
        UnSyncRes.Access()

        ' Ensure that only one thread can decrement the counter at a time.
        If Interlocked.Decrement(numOps) = 0 Then
            ' Announce to Main that in fact all thread calls are done.
            opsAreDone.Set()
        End If
    End Sub
End Module
' The example displays output like the following:
'    Staring synchronized resource access on thread #6
'    Stopping synchronized resource access on thread #6
'    Staring synchronized resource access on thread #7
'    Stopping synchronized resource access on thread #7
'    Staring synchronized resource access on thread #3
'    Stopping synchronized resource access on thread #3
'    Staring synchronized resource access on thread #4
'    Stopping synchronized resource access on thread #4
'    Staring synchronized resource access on thread #5
'    Stopping synchronized resource access on thread #5
'
'    All synchronized operations have completed.
'
'    Starting unsynchronized esource access on Thread #7
'    Starting unsynchronized esource access on Thread #9
'    Starting unsynchronized esource access on Thread #10
'    Starting unsynchronized esource access on Thread #6
'    Starting unsynchronized esource access on Thread #3
```

```
'    Stopping unsynchronized resource access on thread #7
'    Stopping unsynchronized resource access on thread #9
'    Stopping unsynchronized resource access on thread #3
'    Stopping unsynchronized resource access on thread #10
'    Stopping unsynchronized resource access on thread #6
'
'    All unsynchronized thread operations have completed.
```

The example defines a variable, `numOps`, that defines the number of threads that will attempt to access the resource. The application thread calls the ThreadPool.QueueUserWorkItem(WaitCallback) method for synchronized and unsynchronized access five times each. The ThreadPool.QueueUserWorkItem(WaitCallback) method has a single parameter, a delegate that accepts no parameters and returns no value. For synchronized access, it invokes the `SyncUpdateResource` method; for unsynchronized access, it invokes the `UnSyncUpdateResource` method. After each set of method calls, the application thread calls the AutoResetEvent.WaitOne method so that it blocks until the AutoResetEvent instance is signaled.

Each call to the `SyncUpdateResource` method calls the internal `SyncResource.Access` method and then calls the Interlocked.Decrement method to decrement the `numOps` counter. The Interlocked.Decrement method Is used to decrement the counter, because otherwise you cannot be certain that a second thread will access the value before a first thread's decremented value has been stored in the variable. When the last synchronized worker thread decrements the counter to zero, indicating that all synchronized threads have completed accessing the resource, the `SyncUpdateResource` method calls the EventWaitHandle.Set method, which signals the main thread to continue execution.

Each call to the `UnSyncUpdateResource` method calls the internal `UnSyncResource.Access` method and then calls the Interlocked.Decrement method to decrement the `numOps` counter. Once again, the Interlocked.Decrement method Is used to decrement the counter to ensure that a second thread does not access the value before a first thread's decremented value has been assigned to the variable. When the last unsynchronized worker thread decrements the counter to zero, indicating that no more unsynchronized threads need to access the resource, the `UnSyncUpdateResource` method calls the EventWaitHandle.Set method, which signals the main thread to continue execution.

As the output from the example shows, synchronized access ensures that the calling thread exits the protected resource before another thread can access it; each thread waits on its predecessor. On the other hand, without the lock, the `UnSyncResource.Access` method is called in the order in which threads reach it.

Version Information

Universal Windows Platform
Available since 8
.NET Framework
Available since 1.1
Portable Class Library
Supported in: portable .NET platforms
Silverlight
Available since 2.0
Windows Phone Silverlight
Available since 7.0
Windows Phone
Available since 8.1

Thread Safety

This type is thread safe.

See Also

Thread
System.Threading Namespace
Managed Threading
Threading Objects and Features

Return to top

WaitHandle Class

.NET Framework (current version)

Encapsulates operating system–specific objects that wait for exclusive access to shared resources.

Namespace: System.Threading
Assembly: mscorlib (in mscorlib.dll)

Inheritance Hierarchy

System.Object
  System.MarshalByRefObject
    System.Threading.WaitHandle
      System.Threading.EventWaitHandle
      System.Threading.Mutex
      System.Threading.Semaphore

Syntax

VB

```
<ComVisibleAttribute(True)>
Public MustInherit Class WaitHandle
        Inherits MarshalByRefObject
        Implements IDisposable
```

Constructors

	Name	Description
	WaitHandle()	Initializes a new instance of the WaitHandle class.

Properties

	Name	Description

	Handle	**Obsolete.**Gets or sets the native operating system handle.
	SafeWaitHandle	Gets or sets the native operating system handle.

Methods

	Name	Description
	Close()	Releases all resources held by the current WaitHandle.
	CreateObjRef(Type)	Creates an object that contains all the relevant information required to generate a proxy used to communicate with a remote object.(Inherited from MarshalByRefObject.)
	Dispose()	Releases all resources used by the current instance of the WaitHandle class.
	Dispose(Boolean)	When overridden in a derived class, releases the unmanaged resources used by the WaitHandle, and optionally releases the managed resources.
	Equals(Object)	Determines whether the specified object is equal to the current object.(Inherited from Object.)
	Finalize()	Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection.(Inherited from Object.)
	GetHashCode()	Serves as the default hash function. (Inherited from Object.)
	GetLifetimeService()	Retrieves the current lifetime service object that controls the lifetime policy for this instance.(Inherited from MarshalByRefObject.)
	GetType()	Gets the Type of the current instance.(Inherited from Object.)
	InitializeLifetimeService()	Obtains a lifetime service object to control the lifetime policy for this instance.(Inherited from MarshalByRefObject.)
	MemberwiseClone()	Creates a shallow copy of the current Object.(Inherited from Object.)
	MemberwiseClone(Boolean)	Creates a shallow copy of the current MarshalByRefObject object.(Inherited from MarshalByRefObject.)

S	SignalAndWait(WaitHandle, WaitHandle)	Signals one WaitHandle and waits on another.
S	SignalAndWait(WaitHandle, WaitHandle, Int32, Boolean)	Signals one WaitHandle and waits on another, specifying a time-out interval as a 32-bit signed integer and specifying whether to exit the synchronization domain for the context before entering the wait.
S	SignalAndWait(WaitHandle, WaitHandle, TimeSpan, Boolean)	Signals one WaitHandle and waits on another, specifying the time-out interval as a TimeSpan and specifying whether to exit the synchronization domain for the context before entering the wait.
	ToString()	Returns a string that represents the current object.(Inherited from Object.)
S	WaitAll(WaitHandle())	Waits for all the elements in the specified array to receive a signal.
S	WaitAll(WaitHandle(), Int32)	Waits for all the elements in the specified array to receive a signal, using an Int32 value to specify the time interval.
S	WaitAll(WaitHandle(), Int32, Boolean)	Waits for all the elements in the specified array to receive a signal, using an Int32 value to specify the time interval and specifying whether to exit the synchronization domain before the wait.
S	WaitAll(WaitHandle(), TimeSpan)	Waits for all the elements in the specified array to receive a signal, using a TimeSpan value to specify the time interval.
S	WaitAll(WaitHandle(), TimeSpan, Boolean)	Waits for all the elements in the specified array to receive a signal, using a TimeSpan value to specify the time interval, and specifying whether to exit the synchronization domain before the wait.
S	WaitAny(WaitHandle())	Waits for any of the elements in the specified array to receive a signal.
S	WaitAny(WaitHandle(), Int32)	Waits for any of the elements in the specified array to receive a signal, using a 32-bit signed integer to specify the time interval.
S	WaitAny(WaitHandle(), Int32, Boolean)	Waits for any of the elements in the specified array to receive a signal, using a 32-bit signed integer to specify the time interval, and specifying whether to exit the synchronization domain before the wait.
S	WaitAny(WaitHandle(), TimeSpan)	Waits for any of the elements in the specified array to receive a signal, using a TimeSpan to specify the time interval.

	WaitAny(WaitHandle(), TimeSpan, Boolean)	Waits for any of the elements in the specified array to receive a signal, using a TimeSpan to specify the time interval and specifying whether to exit the synchronization domain before the wait.
	WaitOne()	Blocks the current thread until the current WaitHandle receives a signal.
	WaitOne(Int32)	Blocks the current thread until the current WaitHandle receives a signal, using a 32-bit signed integer to specify the time interval in milliseconds.
	WaitOne(Int32, Boolean)	Blocks the current thread until the current WaitHandle receives a signal, using a 32-bit signed integer to specify the time interval and specifying whether to exit the synchronization domain before the wait.
	WaitOne(TimeSpan)	Blocks the current thread until the current instance receives a signal, using a TimeSpan to specify the time interval.
	WaitOne(TimeSpan, Boolean)	Blocks the current thread until the current instance receives a signal, using a TimeSpan to specify the time interval and specifying whether to exit the synchronization domain before the wait.

Fields

	Name	Description
	InvalidHandle	Represents an invalid native operating system handle. This field is read-only.
	WaitTimeout	Indicates that a WaitAny operation timed out before any of the wait handles were signaled. This field is constant.

Extension Methods

	Name	Description
	GetSafeWaitHandle()	Gets the safe handle for a native operating system wait handle. (Defined by WaitHandleExtensions.)

	SetSafeWaitHandle(SafeWaitHandle)	Sets a safe handle for a native operating system wait handle. (Defined by WaitHandleExtensions.)

Remarks

The WaitHandle class encapsulates Win32 synchronization handles, and is used to represent all synchronization objects in the runtime that allow multiple wait operations. For a comparison of wait handles with other synchronization objects, see Overview of Synchronization Primitives.

The WaitHandle class itself is abstract. Classes derived from WaitHandle define a signaling mechanism to indicate taking or releasing access to a shared resource, but they use the inherited WaitHandle methods to block while waiting for access to shared resources. The classes derived from WaitHandle include:

- The Mutex class. See Mutexes.
- The EventWaitHandle class and its derived classes, AutoResetEvent and ManualResetEvent. See EventWaitHandle, AutoResetEvent, CountdownEvent, ManualResetEvent.
- The Semaphore class. See Semaphore and SemaphoreSlim.

Threads can block on an individual wait handle by calling the instance method WaitOne, which is inherited by classes derived from WaitHandle.

The derived classes of WaitHandle differ in their thread affinity. Event wait handles (EventWaitHandle, AutoResetEvent, and ManualResetEvent) and semaphores do not have thread affinity; any thread can signal an event wait handle or semaphore. Mutexes, on the other hand, do have thread affinity; the thread that owns a mutex must release it, and an exception is thrown if a thread calls the ReleaseMutex method on a mutex that it does not own.

Because the WaitHandle class derives from MarshalByRefObject, these classes can be used to synchronize the activities of threads across application domain boundaries.

In addition to its derived classes, the WaitHandle class has a number of static methods that block a thread until one or more synchronization objects receive a signal.. These include:

- SignalAndWait, which allows a thread to signal one wait handle and immediately wait on another.
- WaitAll, which allows a thread to wait until all the wait handles in an array receive a signal.
- WaitAny, which allows a thread to wait until any one of a specified set of wait handles has been signaled .

The overloads of these methods provide timeout intervals for abandoning the wait, and the opportunity to exit a synchronization context before entering the wait, allowing other threads to use the synchronization context.

Important

This type implements the IDisposable interface. When you have finished using the type or a type derived from it, you should dispose of it either directly or indirectly. To dispose of the type directly, call its Close method in a **try/catch** block. To dispose of it indirectly, use a language construct such as **using** (in C#) or **Using** (in Visual Basic). For more information, see the "Using an Object that Implements IDisposable" section in the IDisposable interface topic.

WaitHandle implements the Dispose pattern. See Dispose Pattern. When you derive from WaitHandle, use the SafeWaitHandle property to store your native handle operating system handle. You do not need to override the protected Dispose method unless you use additional unmanaged resources.

Examples

The following code example shows how two threads can do background tasks while the Main thread waits for the tasks to complete using the static WaitAny and WaitAll methods of the WaitHandle class.

VB

```
Imports System
Imports System.Threading

NotInheritable Public Class App
    ' Define an array with two AutoResetEvent WaitHandles.
    Private Shared waitHandles() As WaitHandle = _
        {New AutoResetEvent(False), New AutoResetEvent(False)}

    ' Define a random number generator for testing.
    Private Shared r As New Random()

    <MTAThreadAttribute> _
    Public Shared Sub Main()
        ' Queue two tasks on two different threads;
        ' wait until all tasks are completed.
        Dim dt As DateTime = DateTime.Now
        Console.WriteLine("Main thread is waiting for BOTH tasks to complete.")
        ThreadPool.QueueUserWorkItem(AddressOf DoTask, waitHandles(0))
        ThreadPool.QueueUserWorkItem(AddressOf DoTask, waitHandles(1))
        WaitHandle.WaitAll(waitHandles)
        ' The time shown below should match the longest task.
        Console.WriteLine("Both tasks are completed (time waited={0})", _
            (DateTime.Now - dt).TotalMilliseconds)

        ' Queue up two tasks on two different threads;
        ' wait until any tasks are completed.
        dt = DateTime.Now
        Console.WriteLine()
        Console.WriteLine("The main thread is waiting for either task to complete.")
        ThreadPool.QueueUserWorkItem(AddressOf DoTask, waitHandles(0))
        ThreadPool.QueueUserWorkItem(AddressOf DoTask, waitHandles(1))
        Dim index As Integer = WaitHandle.WaitAny(waitHandles)
        ' The time shown below should match the shortest task.
        Console.WriteLine("Task {0} finished first (time waited={1}).", _
```

```
            index + 1,(DateTime.Now - dt).TotalMilliseconds)

    End Sub 'Main

    Shared Sub DoTask(ByVal state As [Object])
        Dim are As AutoResetEvent = CType(state, AutoResetEvent)
        Dim time As Integer = 1000 * r.Next(2, 10)
        Console.WriteLine("Performing a task for {0} milliseconds.", time)
        Thread.Sleep(time)
        are.Set()

    End Sub 'DoTask
End Class 'App

' This code produces output similar to the following:
'
'  Main thread is waiting for BOTH tasks to complete.
'  Performing a task for 7000 milliseconds.
'  Performing a task for 4000 milliseconds.
'  Both tasks are completed (time waited=7064.8052)
'
'  The main thread is waiting for either task to complete.
'  Performing a task for 2000 milliseconds.
'  Performing a task for 2000 milliseconds.
'  Task 1 finished first (time waited=2000.6528).
```

Version Information

Universal Windows Platform
Available since 8
.NET Framework
Available since 1.1
Portable Class Library
Supported in: portable .NET platforms
Silverlight
Available since 2.0
Windows Phone Silverlight
Available since 7.0
Windows Phone
Available since 8.1

Thread Safety

This type is thread safe.

See Also

System.Threading Namespace

Managed Threading
Threading Objects and Features
Mutexes
EventWaitHandle, AutoResetEvent, CountdownEvent, ManualResetEvent
Semaphore and SemaphoreSlim

Return to top

EventWaitHandle, AutoResetEvent, CountdownEvent, ManualResetEvent

.NET Framework (current version)

Event wait handles allow threads to synchronize activities by signaling each other and by waiting on each other's signals. These synchronization events are based on Win32 wait handles and can be divided into two types: those that reset automatically when signaled and those that are reset manually.

Event wait handles are useful in many of the same synchronization scenarios as the Monitor class. Event wait handles are often easier to use than the Monitor.Wait and Monitor.Pulse methods, and they offer more control over signaling. Named event wait handles can also be used to synchronize activities across application domains and processes, whereas monitors are local to an application domain.

In This Section

EventWaitHandle
: The EventWaitHandle class can represent either automatic or manual reset events and either local events or named system events.

AutoResetEvent
: The AutoResetEvent class derives from EventWaitHandle and represents a local event that resets automatically.

ManualResetEvent and ManualResetEventSlim
: The ManualResetEvent class derives from EventWaitHandle and represents a local event that must be reset manually. The ManualResetEventSlim class is a lightweight, faster version that can be used for events within the same process.

CountdownEvent
: The CountdownEvent class provides a simplified way to implement fork/join parallelism patterns in code that uses wait handles.

Related Sections

Wait Handles
: The WaitHandle class is the base class for the EventWaitHandle, Semaphore, and Mutex classes. It contains static methods such as SignalAndWait and WaitAll that are useful when working with all types of wait handles.

See Also

- EventWaitHandle
- WaitHandle
- AutoResetEvent
- ManualResetEvent
- Threading Objects and Features

Managed Threading Basics

EventWaitHandle

.NET Framework (current version)

The EventWaitHandle class allows threads to communicate with each other by signaling and by waiting for signals. Event wait handles (also referred to simply as events) are wait handles that can be signaled in order to release one or more waiting threads. After it is signaled, an event wait handle is reset either manually or automatically. The EventWaitHandle class can represent either a local event wait handle (local event) or a named system event wait handle (named event or system event, visible to all processes).

Note

Event wait handles are not events in the sense usually meant by that word in the .NET Framework. There are no delegates or event handlers involved. The word "event" is used to describe them because they have traditionally been referred to as operating-system events, and because the act of signaling the wait handle indicates to waiting threads that an event has occurred.

Both local and named event wait handles use system synchronization objects, which are protected by SafeWaitHandle wrappers to ensure that the resources are released. You can use the IDisposable.Dispose method to free the resources immediately when you have finished using the object,

Event Wait Handles That Reset Automatically

You create an automatic reset event by specifying EventResetMode.AutoReset when you create the EventWaitHandle object. As its name implies, this synchronization event resets automatically when signaled, after releasing a single waiting thread. Signal the event by calling its Set method.

Automatic reset events are usually used to provide exclusive access to a resource for a single thread at a time. A thread requests the resource by calling the WaitOne method. If no other thread is holding the wait handle, the method returns **true** and the calling thread has control of the resource.

Important

As with all synchronization mechanisms, you must ensure that all code paths wait on the appropriate wait handle before accessing a protected resource. Thread synchronization is cooperative.

If an automatic reset event is signaled when no threads are waiting, it remains signaled until a thread attempts to wait on it. The event releases the thread and immediately resets, blocking subsequent threads.

Event Wait Handles That Reset Manually

You create a manual reset event by specifying EventResetMode.ManualReset when you create the EventWaitHandle object. As its name implies, this synchronization event must be reset manually after it has been signaled. Until it is reset, by calling its Reset method, threads that wait on the event handle proceed immediately without blocking.

A manual reset event acts like the gate of a corral. When the event is not signaled, threads that wait on it block, like horses in a corral. When the event is signaled, by calling its Set method, all waiting threads are free to proceed. The event remains signaled until its Reset method is called. This makes the manual reset event an ideal way to hold up threads that need to wait until one thread finishes a task.

Like horses leaving a corral, it takes time for the released threads to be scheduled by the operating system and to resume execution. If the Reset method is called before all the threads have resumed execution, the remaining threads once again block. Which threads resume and which threads block depends on random factors like the load on the system, the number of threads waiting for the scheduler, and so on. This is not a problem if the thread that signals the event ends after signaling, which is the most common usage pattern. If you want the thread that signaled the event to begin a new task after all the waiting threads have resumed, you must block it until all the waiting threads have resumed. Otherwise, you have a race condition, and the behavior of your code is unpredictable.

Features Common to Automatic and Manual Events

Typically, one or more threads block on an EventWaitHandle until an unblocked thread calls the Set method, which releases one of the waiting threads (in the case of automatic reset events) or all of them (in the case of manual reset events). A thread can signal an EventWaitHandle and then block on it, as an atomic operation, by calling the static WaitHandle.SignalAndWait method.

EventWaitHandle objects can be used with the static WaitHandle.WaitAll and WaitHandle.WaitAny methods. Because the EventWaitHandle and Mutex classes both derive from WaitHandle, you can use both classes with these methods.

Named Events

The Windows operating system allows event wait handles to have names. A named event is system wide. That is, once the named event is created, it is visible to all threads in all processes. Thus, named events can be used to synchronize the activities of processes as well as threads.

You can create an EventWaitHandle object that represents a named system event by using one of the constructors that specifies an event name.

Note

Because named events are system wide, it is possible to have multiple EventWaitHandle objects that represent the same named event. Each time you call a constructor, or the OpenExisting method, a new EventWaitHandle object is created. Specifying the same name repeatedly creates multiple objects that represent the same named event.

Caution is advised in using named events. Because they are system wide, another process that uses the same name can block your threads unexpectedly. Malicious code executing on the same computer could use this as the basis of a denial-of-service attack.

Use access control security to protect an EventWaitHandle object that represents a named event, preferably by using a constructor that specifies an EventWaitHandleSecurity object. You can also apply access control security using the

SetAccessControl method, but this leaves a window of vulnerability between the time the event wait handle is created and the time it is protected. Protecting events with access control security helps prevent malicious attacks, but it does not solve the problem of unintentional name collisions.

Note

Unlike the EventWaitHandle class, the derived classes AutoResetEvent and ManualResetEvent can represent only local wait handles. They cannot represent named system events.

See Also

EventWaitHandle
WaitHandle
AutoResetEvent
ManualResetEvent
EventWaitHandle, AutoResetEvent, CountdownEvent, ManualResetEvent

AutoResetEvent

.NET Framework (current version)

The AutoResetEvent class represents a local wait handle event that resets automatically when signaled, after releasing a single waiting thread. This class represents a special case of its base class, EventWaitHandle. See the EventWaitHandle conceptual documentation for the use and features of automatic reset events.

An AutoResetEvent object is automatically reset to non-signaled by the system after a single waiting thread has been released. If no threads are waiting, the event object's state remains signaled. AutoResetEvent corresponds to a Win32 **CreateEvent** call, specifying **false** for the *bManualReset* argument.

For an example that uses AutoResetEvent, see Monitor.

See Also

ManualResetEvent
Monitor
EventWaitHandle, AutoResetEvent, CountdownEvent, ManualResetEvent
Managed Threading
Threading Objects and Features
Wait Handles

ManualResetEvent and ManualResetEventSlim

.NET Framework (current version)

The System.Threading.ManualResetEvent class represents a local wait handle event that must be reset manually after it is signaled. This class represents a special case of its base class, System.Threading.EventWaitHandle. See the EventWaitHandle conceptual documentation for the use and features of manual reset events.

A ManualResetEvent object remains signaled until its EventWaitHandle.Reset method is called. Any number of waiting threads, or threads that wait on the event after it has been signaled, can be released while the object's state is signaled. ManualResetEvent corresponds to a Win32 **CreateEvent** call, specifying **true** for the *bManualReset* argument.

In the .NET Framework 4, you can use the System.Threading.ManualResetEventSlim class for better performance when wait times are expected to be very short, and when the event does not cross a process boundary. ManualResetEventSlim uses busy spinning for a short time while it waits for the event to become signaled. When wait times are short, spinning can be much less expensive than waiting by using wait handles. However, if the event does not become signaled within a certain period of time, ManualResetEventSlim resorts to a regular event handle wait.

See Also

- Managed Threading
- Threading Objects and Features
- Wait Handles
- AutoResetEvent
- SpinWait
- Semaphore and SemaphoreSlim

CountdownEvent

.NET Framework (current version)

System.Threading.CountdownEvent is a synchronization primitive that unblocks its waiting threads after it has been signaled a certain number of times. CountdownEvent is designed for scenarios in which you would otherwise have to use a ManualResetEvent or ManualResetEventSlim and manually decrement a variable before signaling the event. For example, in a fork/join scenario, you can just create a CountdownEvent that has a signal count of 5, and then start five work items on the thread pool and have each work item call Signal when it completes. Each call to Signal decrements the signal count by 1. On the main thread, the call to Wait will block until the signal count is zero.

Note

For code that does not have to interact with legacy .NET Framework synchronization APIs, consider using System.Threading.Tasks.Task objects or the Invoke method for an even easier approach to expressing fork-join parallelism.

CountdownEvent has these additional features:

- The wait operation can be canceled by using cancellation tokens.
- Its signal count can be incremented after the instance is created.
- Instances can be reused after Wait has returned by calling the Reset method.
- Instances expose a WaitHandle for integration with other .NET Framework synchronization APIs such as WaitAll.

Basic Usage

The following example demonstrates how to use a CountdownEvent with ThreadPool work items.

VB

```
Dim source As IEnumerable(Of Data) = GetData()
Dim e = New CountdownEvent(1)

' Fork work:
For Each element As Data In source
    ' Dynamically increment signal count.
    e.AddCount()

    ThreadPool.QueueUserWorkItem(Sub(state)
                                     Try
                                         ProcessData(state)
                                     Finally
```

```
                                        e.Signal()
                                    End Try
                                End Sub,
                                 element)
Next
' Decrement the signal count by the one we added
' in the constructor.
e.Signal()

' The first element could also be run on this thread.
' ProcessData(New Data(0))

' Join with work:
e.Wait()
```

CountdownEvent With Cancellation

The following example shows how to cancel the wait operation on CountdownEvent by using a cancellation token. The basic pattern follows the model for unified cancellation, which is introduced in .NET Framework 4. For more information, see Cancellation in Managed Threads.

VB

```
Option Strict On
Option Explicit On

Imports System.Collections
Imports System.Collections.Generic
Imports System.Linq
Imports System.Threading
Imports System.Threading.Tasks

Module CancelEventWait

    Class Data
        Public Num As Integer
        Public Sub New(ByVal i As Integer)
            Num = i
        End Sub
        Public Sub New()

        End Sub
    End Class

    Class DataWithToken
        Public Token As CancellationToken
        Public _data As Data
        Public Sub New(ByVal d As Data, ByVal ct As CancellationToken)
            Me._data = d
            Me.Token = ct
```

```
        End Sub
    End Class

    Class Program
        Shared Function GetData() As IEnumerable(Of Data)
            Dim nums = New List(Of Data)
            For i As Integer = 1 To 5
                nums.Add(New Data(i))
            Next
            Return nums
        End Function

        Shared Sub ProcessData(ByVal obj As Object)
            Dim dataItem As DataWithToken = CType(obj, DataWithToken)
            If dataItem.Token.IsCancellationRequested = True Then
                Console.WriteLine("Canceled before starting {0}", dataItem._data.Num)
                Exit Sub
            End If

            ' Increase this value to slow down the program.
            For i As Integer = 0 To 10000

                If dataItem.Token.IsCancellationRequested = True Then
                    Console.WriteLine("Cancelling while executing {0}",
dataItem._data.Num)
                    Exit Sub
                End If
                Thread.SpinWait(100000)
            Next
            Console.WriteLine("Processed {0}", dataItem._data.Num)


        End Sub

        Shared Sub Main()
            DoEventWithCancel()
            Console.WriteLine("Press the enter key to exit.")
            Console.ReadLine()
        End Sub

        Shared Sub DoEventWithCancel()
            Dim source As IEnumerable(Of Data) = GetData()
            Dim cts As CancellationTokenSource = New CancellationTokenSource()

            ' Enable cancellation request from a simple UI thread.
            Task.Factory.StartNew(Sub()
                                      If Console.ReadKey().KeyChar = "c"c Then
                                          cts.Cancel()
                                      End If
                                  End Sub)

            ' Must have a count of at least 1 or else it is signaled.
            Dim e As CountdownEvent = New CountdownEvent(1)
```

```
            For Each element As Data In source
                Dim item As DataWithToken = New DataWithToken(element, cts.Token)

                ' Dynamically increment signal count.
                e.AddCount()

                ThreadPool.QueueUserWorkItem(Sub(state)
                                                 ProcessData(state)
                                                 If cts.Token.IsCancellationRequested =
False Then
                                                     e.Signal()
                                                 End If
                                             End Sub,
                                             item)
            Next
            ' Decrement the signal count by the one we added
            ' in the constructor.
            e.Signal()
            ' The first element could be run on this thread.
            ' ProcessData(source(0))

            ' Join with work or catch cancellation exception
            Try
                e.Wait(cts.Token)
            Catch ex As OperationCanceledException
                If ex.CancellationToken = cts.Token Then
                    Console.WriteLine("User canceled.")
                Else : Throw ' we don't know who canceled us.

                End If
            Finally
                e.Dispose()
                cts.Dispose()
            End Try
        End Sub
    End Class
End Module
```

Note that the wait operation does not cancel the threads that are signaling it. Typically, cancellation is applied to a logical operation, and that can include waiting on the event as well as all the work items that the wait is synchronizing. In this example, each work item is passed a copy of the same cancellation token so that it can respond to the cancellation request.

See Also

EventWaitHandle, AutoResetEvent, CountdownEvent, ManualResetEvent

Mutexes

.NET Framework (current version)

You can use a Mutex object to provide exclusive access to a resource. The Mutex class uses more system resources than the Monitor class, but it can be marshaled across application domain boundaries, it can be used with multiple waits, and it can be used to synchronize threads in different processes. For a comparison of managed synchronization mechanisms, see Overview of Synchronization Primitives.

For code examples, see the reference documentation for the Mutex constructors.

Using Mutexes

A thread calls the WaitOne method of a mutex to request ownership. The call blocks until the mutex is available, or until the optional timeout interval elapses. The state of a mutex is signaled if no thread owns it.

A thread releases a mutex by calling its ReleaseMutex method. Mutexes have thread affinity; that is, the mutex can be released only by the thread that owns it. If a thread releases a mutex it does not own, an ApplicationException is thrown in the thread.

Because the Mutex class derives from WaitHandle, you can also call the static WaitAll or WaitAny methods of WaitHandle to request ownership of a Mutex in combination with other wait handles.

If a thread owns a Mutex, that thread can specify the same Mutex in repeated wait-request calls without blocking its execution; however, it must release the Mutex as many times to release ownership.

Abandoned Mutexes

If a thread terminates without releasing a Mutex, the mutex is said to be abandoned. This often indicates a serious programming error because the resource the mutex is protecting might be left in an inconsistent state. In the .NET Framework version 2.0, an AbandonedMutexException is thrown in the next thread that acquires the mutex.

> **Note**
>
> In the .NET Framework versions 1.0 and 1.1, an abandoned Mutex is set to the signaled state and the next waiting thread gets ownership. If no thread is waiting, the Mutex remains in a signaled state. No exception is thrown.

In the case of a system-wide mutex, an abandoned mutex might indicate that an application has been terminated abruptly (for example, by using Windows Task Manager).

Local and System Mutexes

Mutexes are of two types: local mutexes and named system mutexes. If you create a Mutex object using a constructor that accepts a name, it is associated with an operating-system object of that name. Named system mutexes are visible throughout the operating system and can be used to synchronize the activities of processes. You can create multiple Mutex objects that represent the same named system mutex, and you can use the OpenExisting method to open an existing named system mutex.

A local mutex exists only within your process. It can be used by any thread in your process that has a reference to the local Mutex object. Each Mutex object is a separate local mutex.

Access Control Security for System Mutexes

The .NET Framework version 2.0 provides the ability to query and set Windows access control security for named system objects. Protecting system mutexes from the moment of creation is recommended because system objects are global and therefore can be locked by code other than your own.

For information on access control security for mutexes, see the MutexSecurity and MutexAccessRule classes, the MutexRights enumeration, the GetAccessControl, SetAccessControl, and OpenExisting methods of the Mutex class, and the Mutex(Boolean, String, Boolean, MutexSecurity) constructor.

See Also

Mutex
Mutex
MutexSecurity
MutexAccessRule
Managed Threading
Threading Objects and Features
Monitors
Threads and Threading

Interlocked Operations

.NET Framework (current version)

The Interlocked class provides methods that synchronize access to a variable that is shared by multiple threads. The threads of different processes can use this mechanism if the variable is in shared memory. Interlocked operations are atomic — that is, the entire operation is a unit that cannot be interrupted by another interlocked operation on the same variable. This is important in operating systems with preemptive multithreading, where a thread can be suspended after loading a value from a memory address, but before having the chance to alter it and store it.

The Interlocked class provides the following operations:

- In the .NET Framework version 2.0, the Add method adds an integer value to a variable and returns the new value of the variable.
- In the .NET Framework version 2.0, the Read method reads a 64-bit integer value as an atomic operation. This is useful on 32-bit operating systems, where reading a 64-bit integer is not ordinarily an atomic operation.
- The Increment and Decrement methods increment or decrement a variable and return the resulting value.
- The Exchange method performs an atomic exchange of the value in a specified variable, returning that value and replacing it with a new value. In the .NET Framework version 2.0, a generic overload of this method can be used to perform this exchange on a variable of any reference type. See Exchange(Of T)(T, T).
- The CompareExchange method also exchanges two values, but contingent on the result of a comparison. In the .NET Framework version 2.0, a generic overload of this method can be used to perform this exchange on a variable of any reference type. See CompareExchange(Of T)(T, T, T).

On modern processors, the methods of the Interlocked class can often be implemented by a single instruction. Thus, they provide very high-performance synchronization and can be used to build higher-level synchronization mechanisms, like spin locks.

For an example that uses the Monitor and Interlocked classes in combination, see Monitors.

CompareExchange Example

The CompareExchange method can be used to protect computations that are more complicated than simple increment and decrement. The following example demonstrates a thread-safe method that adds to a running total stored as a floating point number. (For integers, the Add method is a simpler solution.) For complete code examples, see the overloads of CompareExchange that take single-precision and double-precision floating-point arguments (CompareExchange(Single, Single, Single) and CompareExchange(Double, Double, Double)).

VB

```
Imports System
Imports System.Threading

Public Class ThreadSafe
```

```
    ' totalValue contains a running total that can be updated
    ' by multiple threads. It must be protected from unsynchronized
    ' access.
    Private totalValue As Double = 0.0

    ' The Total property returns the running total.
    Public ReadOnly Property Total As Double
        Get
            Return totalValue
        End Get
    End Property

    ' AddToTotal safely adds a value to the running total.
    Public Function AddToTotal(addend As Double) As Double
        Dim initialValue, computedValue As Double
        Do
            ' Save the current running total in a local variable.
            initialValue = totalValue

            ' Add the new value to the running total.
            computedValue = initialValue + addend

            ' CompareExchange compares totalValue to initialValue. If
            ' they are not equal, then another thread has updated the
            ' running total since this loop started. CompareExchange
            ' does not update totalValue. CompareExchange returns the
            ' contents of totalValue, which do not equal initialValue,
            ' so the loop executes again.
        Loop While initialValue <> Interlocked.CompareExchange( _
            totalValue, computedValue, initialValue)
        ' If no other thread updated the running total, then
        ' totalValue and initialValue are equal when CompareExchange
        ' compares them, and computedValue is stored in totalValue.
        ' CompareExchange returns the value that was in totalValue
        ' before the update, which is equal to initialValue, so the
        ' loop ends.

        ' The function returns computedValue, not totalValue, because
        ' totalValue could be changed by another thread between
        ' the time the loop ends and the function returns.
        Return computedValue
    End Function
End Class
```

Untyped Overloads of Exchange and CompareExchange

The Exchange and CompareExchange methods have overloads that take arguments of type Object. The first argument of each of these overloads is **ref Object** (**ByRef ... As Object** in Visual Basic), and type safety requires the variable passed to this argument to be typed strictly as Object; you cannot simply cast the first argument to type Object when calling these methods.

Note

In the .NET Framework version 2.0, use the generic overloads of the Exchange and CompareExchange methods to exchange strongly typed variables.

The following code example shows a property of type `ClassA` that can be set only once, as it might be implemented in the .NET Framework version 1.0 or 1.1.

VB

```
Public Class ClassB
    ' The private field that stores the value for the
    ' ClassA property is intialized to null. It is set
    ' once, from any of several threads. The field must
    ' be of type Object, so that CompareExchange can be
    ' used to assign the value. If the field is used
    ' within the body of class Test, it must be cast to
    ' type ClassA.
    Private classAValue As Object = Nothing
    ' This property can be set once to an instance of
    ' ClassA. Attempts to set it again cause an
    ' exception to be thrown.
    Public Property ClassA() As ClassA
        Get
            Return CType(classAValue, ClassA)
        End Get

        Set
            ' CompareExchange compares the value in classAValue
            ' to null. The new value assigned to the ClassA
            ' property, which is in the special variable 'value',
            ' is placed in classAValue only if classAValue is
            ' equal to null.
            If Not (Nothing Is Interlocked.CompareExchange(classAValue, _
                CType(value, [Object]), Nothing)) Then
                ' CompareExchange returns the original value of
                ' classAValue; if it is not null, then a value
                ' was already assigned, and CompareExchange did not
                ' replace the original value. Throw an exception to
                ' indicate that an error occurred.
                Throw New ApplicationException("ClassA was already set.")
            End If
        End Set
    End Property
End Class
```

See Also

Interlocked

Monitor
Managed Threading
Threading Objects and Features

Reader-Writer Locks

.NET Framework (current version)

The ReaderWriterLockSlim class enables multiple threads to read a resource concurrently, but requires a thread to wait for an exclusive lock in order to write to the resource.

You might use a ReaderWriterLockSlim in your application to provide cooperative synchronization among threads that access a shared resource. Locks are taken on the ReaderWriterLockSlim itself.

As with any thread synchronization mechanism, you must ensure that no threads bypass the locking that is provided by ReaderWriterLockSlim. One way to ensure this is to design a class that encapsulates the shared resource. This class would provide members that access the private shared resource and that use a private ReaderWriterLockSlim for synchronization. For an example, see the code example for the ReaderWriterLockSlim class. ReaderWriterLockSlim is efficient enough to be used to synchronize individual objects.

Structure your application to minimize the duration of read and write operations. Long write operations affect throughput directly because the write lock is exclusive. Long read operations block waiting writers, and if at least one thread is waiting for write access, threads that request read access will be blocked as well.

Note

The .NET Framework has two reader-writer locks, ReaderWriterLockSlim and ReaderWriterLock. ReaderWriterLockSlim is recommended for all new development. ReaderWriterLockSlim is similar to ReaderWriterLock, but it has simplified rules for recursion and for upgrading and downgrading lock state. ReaderWriterLockSlim avoids many cases of potential deadlock. In addition, the performance of ReaderWriterLockSlim is significantly better than ReaderWriterLock.

See Also

ReaderWriterLockSlim
ReaderWriterLock
Managed Threading
Threading Objects and Features

Semaphore and SemaphoreSlim

.NET Framework (current version)

The System.Threading.Semaphore class represents a named (systemwide) or local semaphore. It is a thin wrapper around the Win32 semaphore object. Win32 semaphores are counting semaphores, which can be used to control access to a pool of resources.

The SemaphoreSlim class represents a lightweight, fast semaphore that can be used for waiting within a single process when wait times are expected to be very short. SemaphoreSlim relies as much as possible on synchronization primitives provided by the common language runtime (CLR). However, it also provides lazily initialized, kernel-based wait handles as necessary to support waiting on multiple semaphores. SemaphoreSlim also supports the use of cancellation tokens, but it does not support named semaphores or the use of a wait handle for synchronization.

Managing a Limited Resource

Threads enter the semaphore by calling the WaitOne method, which is inherited from the WaitHandle class, in the case of a System.Threading.Semaphore object, or the SemaphoreSlim.Wait or SemaphoreSlim.WaitAsync method, in the case of a SemaphoreSlim object.. When the call returns, the count on the semaphore is decremented. When a thread requests entry and the count is zero, the thread blocks. As threads release the semaphore by calling the Semaphore.Release or SemaphoreSlim.Release method, blocked threads are allowed to enter. There is no guaranteed order, such as first-in, first-out (FIFO) or last-in, first-out (LIFO), for blocked threads to enter the semaphore.

A thread can enter the semaphore multiple times by calling the System.Threading.Semaphore object's WaitOne method or the SemaphoreSlim object's Wait method repeatedly. To release the semaphore, the thread can either call the Semaphore.Release() or SemaphoreSlim.Release() method overload the same number of times, or call the Semaphore.Release(Int32) or SemaphoreSlim.Release(Int32) method overload and specify the number of entries to be released.

Semaphores and Thread Identity

The two semaphore types do not enforce thread identity on calls to the WaitOne, Wait, Release, and SemaphoreSlim.Release methods. For example, a common usage scenario for semaphores involves a producer thread and a consumer thread, with one thread always incrementing the semaphore count and the other always decrementing it.

It is the programmer's responsibility to ensure that a thread does not release the semaphore too many times. For example, suppose a semaphore has a maximum count of two, and that thread A and thread B both enter the semaphore. If a programming error in thread B causes it to call **Release** twice, both calls succeed. The count on the semaphore is full, and when thread A eventually calls **Release**, , a SemaphoreFullException is thrown.

Named Semaphores

The Windows operating system allows semaphores to have names. A named semaphore is system wide. That is, once the

named semaphore is created, it is visible to all threads in all processes. Thus, named semaphore can be used to synchronize the activities of processes as well as threads.

You can create a Semaphore object that represents a named system semaphore by using one of the constructors that specifies a name.

Note

Because named semaphores are system wide, it is possible to have multiple Semaphore objects that represent the same named semaphore. Each time you call a constructor or the Semaphore.OpenExisting method, a new Semaphore object is created. Specifying the same name repeatedly creates multiple objects that represent the same named semaphore.

Be careful when you use named semaphores. Because they are system wide, another process that uses the same name can enter your semaphore unexpectedly. Malicious code executing on the same computer could use this as the basis of a denial-of-service attack.

Use access control security to protect a Semaphore object that represents a named semaphore, preferably by using a constructor that specifies a System.Security.AccessControl.SemaphoreSecurity object. You can also apply access control security using the Semaphore.SetAccessControl method, but this leaves a window of vulnerability between the time the semaphore is created and the time it is protected. Protecting semaphores with access control security helps prevent malicious attacks, but does not solve the problem of unintentional name collisions.

See Also

Semaphore
SemaphoreSlim
Threading Objects and Features

Overview of Synchronization Primitives

.NET Framework (current version)

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

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 (.NET Framework).

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 (.NET Framework)
- SpinWait
- SpinLock

Barrier (.NET Framework)

.NET Framework (current version)

A *barrier* is a user-defined synchronization primitive that enables multiple threads (known as *participants*) to work concurrently on an algorithm in phases. Each participant executes until it reaches the barrier point in the code. The barrier represents the end of one phase of work. When a participant reaches the barrier, it blocks until all participants have reached the same barrier. After all participants have reached the barrier, you can optionally invoke a post-phase action. This post-phase action can be used to perform actions by a single thread while all other threads are still blocked. After the action has been executed, the participants are all unblocked.

The following code snippet shows a basic barrier pattern.

VB

```
' Create the Barrier object, and supply a post-phase delegate
' to be invoked at the end of each phase.
Dim barrier = New Barrier(2, Sub(bar)
                                 ' Examine results from all threads, determine
                                 ' whether to continue, create inputs for next phase,
etc.
                                 If (someCondition) Then
                                     success = True
                                 End If
                             End Sub)



' Define the work that each thread will perform. (Threads do not
' have to all execute the same method.)
Sub CrunchNumbers(ByVal partitionNum As Integer)

    ' Up to System.Int64.MaxValue phases are supported. We assume
    ' in this code that the problem will be solved before that.
    While (success = False)

        ' Begin phase:
        ' Process data here on each thread, and optionally
        ' store results, for example:
        results(partitionNum) = ProcessData(myData(partitionNum))

        ' End phase:
        ' After all threads arrive,post-phase delegate
        ' is invoked, then threads are unblocked. Overloads
        ' accept a timeout value and/or CancellationToken.
        barrier.SignalAndWait()
    End While
End Sub
```

```
' Perform n tasks to run in in parallel. For simplicity
' all threads execute the same method in this example.
Shared Sub Main()

    Dim app = New BarrierDemo()
    Dim t1 = New Thread(Sub() app.CrunchNumbers(0))
    Dim t2 = New Thread(Sub() app.CrunchNumbers(1))
    t1.Start()
    t2.Start()
End Sub
```

For a complete example, see How to: Synchronize Concurrent Operations with a Barrier.

Adding and Removing Participants

When you create a Barrier, specify the number of participants. You can also add or remove participants dynamically at any time. For example, if one participant solves its part of the problem, you can store the result, stop execution on that thread, and call RemoveParticipant to decrement the number of participants in the barrier. When you add a participant by calling AddParticipant, the return value specifies the current phase number, which may be useful in order to initialize the work of the new participant.

Broken Barriers

Deadlocks can occur if one participant fails to reach the barrier. To avoid these deadlocks, use the overloads of the SignalAndWait method to specify a time-out period and a cancellation token. These overloads return a Boolean value that every participant can check before it continues to the next phase.

Post-Phase Exceptions

If the post-phase delegate throws an exception, it is wrapped in a BarrierPostPhaseException object which is then propagated to all participants.

Barrier Versus ContinueWhenAll

Barriers are especially useful when the threads are performing multiple phases in loops. If your code requires only one or two phases of work, consider whether to use System.Threading.Tasks.Task objects with any kind of implicit join, including:

- ContinueWhenAll
- Invoke
- ForEach
- For

For more information, see Chaining Tasks by Using Continuation Tasks.

See Also

Threading Objects and Features
How to: Synchronize Concurrent Operations with a Barrier

How to: Synchronize Concurrent Operations with a Barrier

.NET Framework (current version)

The following example shows how to synchronize concurrent tasks with a Barrier.

Example

The purpose of the following program is to count how many iterations (or phases) are required for two threads to each find their half of the solution on the same phase by using a randomizing algorithm to reshuffle the words. After each thread has shuffled its words, the barrier post-phase operation compares the two results to see if the complete sentence has been rendered in correct word order.

VB

```
Imports System
Imports System.Collections.Generic
Imports System.Linq
Imports System.Text
Imports System.Threading
Imports System.Threading.Tasks


    Class Program
        Shared words1() = New String() {"brown", "jumped", "the", "fox", "quick"}
        Shared words2() = New String() {"dog", "lazy", "the", "over"}
        Shared solution = "the quick brown fox jumped over the lazy dog."

        Shared success = False
        Shared barrier = New Barrier(2, Sub(b)
                                            Dim sb = New StringBuilder()
                                            For i As Integer = 0 To words1.Length - 1
                                                sb.Append(words1(i))
                                                sb.Append(" ")
                                            Next
                                            For i As Integer = 0 To words2.Length - 1

                                                sb.Append(words2(i))

                                                If (i < words2.Length - 1) Then
                                                    sb.Append(" ")
                                                End If
                                            Next
                                            sb.Append(".")

System.Diagnostics.Trace.WriteLine(sb.ToString())
```

```
                                                Console.CursorLeft = 0
                                                Console.Write("Current phase: {0}",
barrier.CurrentPhaseNumber)
                                                If (String.CompareOrdinal(solution,
sb.ToString()) = 0) Then
                                                    success = True
                                                    Console.WriteLine()
                                                    Console.WriteLine("The solution was found
in {0} attempts", barrier.CurrentPhaseNumber)
                                                End If
                                            End Sub)

        Shared Sub Main()
            Dim t1 = New Thread(Sub() Solve(words1))
            Dim t2 = New Thread(Sub() Solve(words2))
            t1.Start()
            t2.Start()

            ' Keep the console window open.
            Console.ReadLine()
        End Sub

        ' Use Knuth-Fisher-Yates shuffle to randomly reorder each array.
        ' For simplicity, we require that both wordArrays be solved in the same phase.
        ' Success of right or left side only is not stored and does not count.
        Shared Sub Solve(ByVal wordArray As String())
            While success = False
                Dim rand = New Random()
                For i As Integer = 0 To wordArray.Length - 1
                    Dim swapIndex As Integer = rand.Next(i + 1)
                    Dim temp As String = wordArray(i)
                    wordArray(i) = wordArray(swapIndex)
                    wordArray(swapIndex) = temp
                Next

                ' We need to stop here to examine results
                ' of all thread activity. This is done in the post-phase
                ' delegate that is defined in the Barrier constructor.
                barrier.SignalAndWait()
            End While
        End Sub
    End Class
```

A Barrier is an object that prevents individual tasks in a parallel operation from continuing until all tasks reach the barrier. It is useful when a parallel operation occurs in phases, and each phase requires synchronization between tasks. In this example, there are two phases to the operation. In the first phase, each task fills its section of the buffer with data. When each task finishes filling its section, the task signals the barrier that it is ready to continue, and then waits. When all tasks have signaled the barrier, they are unblocked and the second phase starts. The barrier is necessary because the second phase requires that each task have access to all the data that has been generated to this point. Without the barrier, the first tasks to complete might try to read from buffers that have not been filled in yet by other tasks. You can synchronize any number of phases in this manner.

See Also

Data Structures for Parallel Programming

SpinLock

.NET Framework (current version)

The SpinLock structure is a low-level, mutual-exclusion synchronization primitive that spins while it waits to acquire a lock. On multicore computers, when wait times are expected to be short and when contention is minimal, SpinLock can perform better than other kinds of locks. However, we recommend that you use SpinLock only when you determine by profiling that the System.Threading.Monitor method or the Interlocked methods are significantly slowing the performance of your program.

SpinLock may yield the time slice of the thread even if it has not yet acquired the lock. It does this to avoid thread-priority inversion, and to enable the garbage collector to make progress. When you use a SpinLock, ensure that no thread can hold the lock for more than a very brief time span, and that no thread can block while it holds the lock.

Because SpinLock is a value type, you must explicitly pass it by reference if you intend the two copies to refer to the same lock.

For more information about how to use this type, see System.Threading.SpinLock. For an example, see How to: Use SpinLock for Low-Level Synchronization.

SpinLock supports a *thread-tracking* mode that you can use during the development phase to help track the thread that is holding the lock at a specific time. Thread-tracking mode is very useful for debugging, but we recommend that you turn it off in the release version of your program because it may slow performance. For more information, see How to: Enable Thread-Tracking Mode in SpinLock.

See Also

Threading Objects and Features

How to: Use SpinLock for Low-Level Synchronization

.NET Framework (current version)

The following example demonstrates how to use a SpinLock.

Example

In this example, the critical section performs a minimal amount of work, which makes it a good candidate for a SpinLock. Increasing the work a small amount increases the performance of the SpinLock compared to a standard lock. However, there is a point at which a SpinLock becomes more expensive than a standard lock. You can use the concurrency profiling functionality in the profiling tools to see which type of lock provides better performance in your program. For more information, see Concurrency Visualizer.

VB

```
Imports System
Imports System.Threading
Imports System.Threading.Tasks
Class SpinLockDemo2

    Const N As Integer = 100000
    Shared _queue = New Queue(Of Data)()
    Shared _lock = New Object()
    Shared _spinlock = New SpinLock()

    Class Data
        Public Name As String
        Public Number As Double
    End Class
    Shared Sub Main()

        ' First use a standard lock for comparison purposes.
        UseLock()
        _queue.Clear()
        UseSpinLock()

        Console.WriteLine("Press a key")
        Console.ReadKey()

    End Sub

    Private Shared Sub UpdateWithSpinLock(ByVal d As Data, ByVal i As Integer)

        Dim lockTaken As Boolean = False
        Try
            _spinlock.Enter(lockTaken)
```

```
            _queue.Enqueue(d)
        Finally

            If lockTaken Then
                _spinlock.Exit(False)
            End If
        End Try
    End Sub

    Private Shared Sub UseSpinLock()


        Dim sw = Stopwatch.StartNew()

        Parallel.Invoke(
               Sub()
                   For i As Integer = 0 To N - 1
                       UpdateWithSpinLock(New Data() With {.Name = i.ToString(), .Number
= i}, i)
                   Next
               End Sub,
                Sub()
                    For i As Integer = 0 To N - 1
                        UpdateWithSpinLock(New Data() With {.Name = i.ToString(), .Number
= i}, i)
                    Next
                End Sub
            )
        sw.Stop()
        Console.WriteLine("elapsed ms with spinlock: {0}", sw.ElapsedMilliseconds)
    End Sub

    Shared Sub UpdateWithLock(ByVal d As Data, ByVal i As Integer)

        SyncLock (_lock)
            _queue.Enqueue(d)
        End SyncLock
    End Sub

    Private Shared Sub UseLock()

        Dim sw = Stopwatch.StartNew()

        Parallel.Invoke(
                Sub()
                    For i As Integer = 0 To N - 1
                        UpdateWithLock(New Data() With {.Name = i.ToString(), .Number =
i}, i)
                    Next
                End Sub,
               Sub()
                   For i As Integer = 0 To N - 1
                       UpdateWithLock(New Data() With {.Name = i.ToString(), .Number =
i}, i)
```

```
                    Next
                End Sub
                 )
            sw.Stop()
            Console.WriteLine("elapsed ms with lock: {0}", sw.ElapsedMilliseconds)
        End Sub
    End Class
```

SpinLock might be useful when a lock on a shared resource is not going to be held for very long. In such cases, on multi-core computers it can be efficient for the blocked thread to spin for a few cycles until the lock is released. By spinning, the thread does not become blocked, which is a CPU-intensive process. SpinLock will stop spinning under certain conditions to prevent starvation of logical processors or priority inversion on systems with Hyper-Threading.

This example uses the System.Collections.Generic.Queue(Of T) class, which requires user synchronization for multi-threaded access. In applications that target the .NET Framework version 4, another option is to use the System.Collections.Concurrent.ConcurrentQueue(Of T), which does not require any user locks.

Note the use of **false** (**False** in Visual Basic) in the call to Exit. This provides the best performance. Specify **true** (**True**)on IA64 architectures to use the memory fence, which flushes the write buffers to ensure that the lock is now available for other threads to exit.

See Also

Threading Objects and Features

How to: Enable Thread-Tracking Mode in SpinLock

.NET Framework (current version)

System.Threading.SpinLock is a low-level mutual exclusion lock that you can use for scenarios that have very short wait times. SpinLock is not re-entrant. After a thread enters the lock, it must exit the lock correctly before it can enter again. Typically, any attempt to re-enter the lock would cause deadlock, and deadlocks can be very difficult to debug. As an aid to development, System.Threading.SpinLock supports a thread-tracking mode that causes an exception to be thrown when a thread attempts to re-enter a lock that it already holds. This lets you more easily locate the point at which the lock was not exited correctly. You can turn on thread-tracking mode by using the SpinLock constructor that takes a Boolean input parameter, and passing in an argument of **true**. After you complete the development and testing phases, turn off thread-tracking mode for better performance.

Example

The following example demonstrates thread-tracking mode. The lines that correctly exit the lock are commented out to simulate a coding error that causes one of the following results:

- An exception is thrown if the SpinLock was created by using an argument of **true** (**True** in Visual Basic).
- Deadlock if the SpinLock was created by using an argument of **false** (**False** in Visual Basic).

VB

```
Imports System.Text
Imports System.Threading
Imports System.Threading.Tasks
Module Module1

    Public Class SpinTest

        ' True means "enable thread tracking." This will cause an
        ' exception to be thrown when the first thread attempts to reenter the lock.
        ' Specify False to cause deadlock due to coding error below.
        Private Shared _spinLock = New SpinLock(True)

        Public Shared Sub Main()

            Parallel.Invoke(
                Sub() DoWork(),
                Sub() DoWork(),
                Sub() DoWork(),
                Sub() DoWork()
                )

            Console.WriteLine("Press any key.")
```

```
            Console.ReadKey()
        End Sub

        Public Shared Sub DoWork()

            Dim sb = New StringBuilder()

            For i As Integer = 1 To 9999

                Dim lockTaken As Boolean = False

                Try
                    _spinLock.Enter(lockTaken)

                    ' do work here protected by the lock
                    Thread.SpinWait(50000)
                    sb.Append(Thread.CurrentThread.ManagedThreadId)
                    sb.Append(" Entered-")

                Catch ex As LockRecursionException
                    Console.WriteLine("Thread {0} attempted to reenter the lock",
                                      Thread.CurrentThread.ManagedThreadId)
                    Throw

                Finally

                    ' INTENTIONAL CODING ERROR TO DEMONSTRATE THREAD TRACKING!
                    ' UNCOMMENT THE LINES FOR CORRECT SPINLOCK BEHAVIOR
                    ' Commenting out these lines causes the same thread
                    ' to attempt to reenter the lock. If the SpinLock was
                    ' created with thread tracking enabled, the exception
                    ' is thrown. Otherwise, if the SpinLock was created with a
                    ' parameter of false, and these lines are left commented, the
spinlock deadlocks.
                    If (lockTaken) Then

                        '  _spinLock.Exit()
                        '  sb.Append("Exited ")
                    End If
                End Try

                ' Output for diagnostic display.
                If (i Mod 4 <> 0) Then
                    Console.Write(sb.ToString())
                Else
                    Console.WriteLine(sb.ToString())
                End If
                sb.Clear()
            Next
        End Sub
    End Class
End Module
```

See Also

SpinLock

SpinWait

.NET Framework (current version)

System.Threading.SpinWait is a lightweight synchronization type that you can use in low-level scenarios to avoid the expensive context switches and kernel transitions that are required for kernel events. On multicore computers, when a resource is not expected to be held for long periods of time, it can be more efficient for a waiting thread to spin in user mode for a few dozen or a few hundred cycles, and then retry to acquire the resource. If the resource is available after spinning, then you have saved several thousand cycles. If the resource is still not available, then you have spent only a few cycles and can still enter a kernel-based wait. This spinning-then-waiting combination is sometimes referred to as a *two-phase wait operation*.

SpinWait is designed to be used in conjunction with the .NET Framework types that wrap kernel events such as ManualResetEvent. SpinWait can also be used by itself for basic spinning functionality in just one program.

SpinWait is more than just an empty loop. It is carefully implemented to provide correct spinning behavior for the general case, and will itself initiate context switches if it spins long enough (roughly the length of time required for a kernel transition). For example, on single-core computers, SpinWait yields the time slice of the thread immediately because spinning blocks forward progress on all threads. SpinWait also yields even on multi-core machines to prevent the waiting thread from blocking higher-priority threads or the garbage collector. Therefore, if you are using a SpinWait in a two-phase wait operation, we recommend that you invoke the kernel wait before the SpinWait itself initiates a context switch. SpinWait provides the NextSpinWillYield property, which you can check before every call to SpinOnce. When the property returns **true**, initiate your own Wait operation. For an example, see How to: Use SpinWait to Implement a Two-Phase Wait Operation.

If you are not performing a two-phase wait operation but are just spinning until some condition is true, you can enable SpinWait to perform its context switches so that it is a good citizen in the Windows operating system environment. The following basic example shows a SpinWait in a lock-free stack. If you require a high-performance, thread-safe stack, consider using System.Collections.Concurrent.ConcurrentStack(Of T).

VB

```
Imports System.Threading
Module SpinWaitDemo


    Public Class LockFreeStack(Of T)
        Private m_head As Node

        Private Class Node
            Public [Next] As Node
            Public Value As T
        End Class

        Public Sub Push(ByVal item As T)
            Dim spin As New SpinWait()
            Dim head As Node, node As New Node With {.Value = item}

            While True
                Thread.MemoryBarrier()
```

```
                head = m_head
                node.Next = head
                If Interlocked.CompareExchange(m_head, node, head) Is head Then Exit While
                spin.SpinOnce()
            End While
        End Sub

        Public Function TryPop(ByRef result As T) As Boolean
            result = CType(Nothing, T)
            Dim spin As New SpinWait()

            Dim head As Node
            While True
                Thread.MemoryBarrier()
                head = m_head
                If head Is Nothing Then Return False
                If Interlocked.CompareExchange(m_head, head.Next, head) Is head Then
                    result = head.Value
                    Return True
                End If
                spin.SpinOnce()
            End While
        End Function
    End Class


End Module
```

See Also

SpinWait
Threading Objects and Features

How to: Use SpinWait to Implement a Two-Phase Wait Operation

.NET Framework (current version)

The following example shows how to use a System.Threading.SpinWait object to implement a two-phase wait operation. In the first phase, the synchronization object, a **Latch**, spins for a few cycles while it checks whether the lock has become available. In the second phase, if the lock becomes available, then the **Wait** method returns without using the System.Threading.ManualResetEvent to perform its wait; otherwise, **Wait** performs the wait.

Example

This example shows a very basic implementation of a Latch synchronization primitive. You can use this data structure when wait times are expected to be very short. This example is for demonstration purposes only. If you require latch-type functionality in your program, consider using System.Threading.ManualResetEventSlim.

VB

```
#Const LOGGING = 1

Imports System.Diagnostics
Imports System.Threading
Imports System.Threading.Tasks

Class Latch
   Private latchLock As New Object()
   ' 0 = unset, 1 = set.
   Private m_state As Integer = 0
   Private totalKernelWaits As Integer = 0

   ' Block threads waiting for ManualResetEvent.
   Private m_ev = New ManualResetEvent(False)

#If LOGGING Then
   ' For fast logging with minimal impact on latch behavior.
   ' Spin counts greater than 20 might be encountered depending on machine config.
   Dim spinCountLog(19) As Long

   Public Sub DisplayLog()
      For i As Integer = 0 To spinCountLog.Length - 1
         Console.WriteLine("Wait succeeded with spin count of {0} on {1:N0} attempts",
                           i, spinCountLog(i))
      Next
      Console.WriteLine("Wait used the kernel event on {0:N0} attempts.",
                        totalKernelWaits)
      Console.WriteLine("Logging complete")
   End Sub
#End If
```

```
    Public Sub SetLatch()
        SyncLock(latchLock)
            m_state = 1
            m_ev.Set()
        End SyncLock
    End Sub

    Public Sub Wait()
        Trace.WriteLine("Wait timeout infinite")
        Wait(Timeout.Infinite)
    End Sub

    Public Function Wait(ByVal timeout As Integer) As Boolean
        ' Allocated on the stack.
        Dim spinner = New SpinWait()
        Dim watch As Stopwatch

        While (m_state = 0)
            ' Lazily allocate and start stopwatch to track timeout.
            watch = Stopwatch.StartNew()

            ' Spin only until the SpinWait is ready
            ' to initiate its own context switch.
            If Not spinner.NextSpinWillYield Then
                spinner.SpinOnce()

                ' Rather than let SpinWait do a context switch now,
                '  we initiate the kernel Wait operation, because
                ' we plan on doing this anyway.
            Else
                Interlocked.Increment(totalKernelWaits)
                ' Account for elapsed time.
                Dim realTimeout As Long = timeout - watch.ElapsedMilliseconds

                ' Do the wait.
                If realTimeout <= 0 OrElse Not m_ev.WaitOne(realTimeout) Then
                    Trace.WriteLine("wait timed out.")
                    Return False
                End If
            End If
        End While

#If LOGGING Then
        Interlocked.Increment(spinCountLog(spinner.Count))
#End If
        ' Take the latch.
        Interlocked.Exchange(m_state, 0)

        Return True
    End Function
End Class

Class Program
```

```
    Shared latch = New Latch()
    Shared count As Integer = 2
    Shared cts = New CancellationTokenSource()
    Shared lockObj As New Object()

    Shared Sub TestMethod()
       While (Not cts.IsCancellationRequested)
          ' Obtain the latch.
          If (latch.Wait(50)) Then
             ' Do the work. Here we vary the workload a slight amount
             ' to help cause varying spin counts in latch.
             Dim d As Double = 0
             If (count Mod 2 <> 0) Then
                d = Math.Sqrt(count)
             End If

             SyncLock(lockObj)
                If count = Int32.MaxValue Then count = 0
                count += 1
             End SyncLock

             ' Release the latch.
             latch.SetLatch()
          End If
       End While
    End Sub

    Shared Sub Main()
       ' Demonstrate latch with a simple scenario:
       ' two threads updating a shared integer and
       ' accessing a shared StringBuilder. Both operations
       ' are relatively fast, which enables the latch to
       ' demonstrate successful waits by spinning only.
       latch.SetLatch()

       ' UI thread. Press 'c' to cancel the loop.
       Task.Factory.StartNew(Sub()
                                Console.WriteLine("Press 'c' to cancel.")
                                If (Console.ReadKey(True).KeyChar = "c"c) Then
                                   cts.Cancel()
                                End If
                             End Sub)
       Parallel.Invoke(
              Sub() TestMethod(),
              Sub() TestMethod(),
              Sub() TestMethod()
              )

#If LOGGING Then
          latch.DisplayLog()
#End If
           If cts IsNot Nothing Then cts.Dispose()
    End Sub
End Class
```

The latch uses the SpinWait object to spin in place only until the next call to **SpinOnce** causes the SpinWait to yield the time slice of the thread. At that point, the latch causes its own context switch by calling WaitOne on the ManualResetEvent and passing in the remainder of the time-out value.

The logging output shows how often the Latch was able to increase performance by acquiring the lock without using the ManualResetEvent.

See Also

SpinWait
Threading Objects and Features