SemaphoreSlim Class

Reference

Definition

Namespace: System.Threading

Assembly: System.Threading.dll

Represents a lightweight alternative to Semaphore that limits the number of threads that can access a resource or pool of resources concurrently.

C#

```
public class SemaphoreSlim : IDisposable
```

Inheritance Object → SemaphoreSlim

Implements IDisposable

Examples

The following example creates a semaphore with a maximum count of three threads and an initial count of zero threads. The example starts five tasks, all of which block waiting for the semaphore. The main thread calls the Release(Int32) overload to increase the semaphore count to its maximum, which allows three tasks to enter the semaphore. Each time the semaphore is released, the previous semaphore count is displayed. Console messages track semaphore use. The simulated work interval is increased slightly for each thread to make the output easier to read.

C#

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
    private static SemaphoreSlim semaphore;

    // A padding interval to make the output more orderly.
    private static int padding;

    public static void Main()
    {
        // Create the semaphore.
        semaphore = new SemaphoreSlim(0, 3);
        Console.WriteLine("{0} tasks can enter the semaphore.",
                          semaphore.CurrentCount);
        Task[] tasks = new Task[5];

        // Create and start five numbered tasks.
        for (int i = 0; i <= 4; i++)
        {
            tasks[i] = Task.Run(() =>
            {
                // Each task begins by requesting the semaphore.
                Console.WriteLine("Task {0} begins and waits for the sema-
phore.",
                                  Task.CurrentId);

                int semaphoreCount;
                semaphore.Wait();
                try
                {
                    Interlocked.Add(ref padding, 100);

                    Console.WriteLine("Task {0} enters the semaphore.",
Task.CurrentId);

                    // The task just sleeps for 1+ seconds.
                    Thread.Sleep(1000 + padding);
                }
                finally {
                    semaphoreCount = semaphore.Release();
                }
                Console.WriteLine("Task {0} releases the semaphore; previous
count: {1}.",
                                  Task.CurrentId, semaphoreCount);
            });
        }

        // Wait for half a second, to allow all the tasks to start and block.
        Thread.Sleep(500);

        // Restore the semaphore count to its maximum value.
        Console.Write("Main thread calls Release(3) --> ");
        semaphore.Release(3);
        Console.WriteLine("{0} tasks can enter the semaphore.",
                          semaphore.CurrentCount);
        // Main thread waits for the tasks to complete.

        Task.WaitAll(tasks);

        Console.WriteLine("Main thread exits.");
    }
}
// The example displays output like the following:
```

```
//       0 tasks can enter the semaphore.
//       Task 1 begins and waits for the semaphore.
//       Task 5 begins and waits for the semaphore.
//       Task 2 begins and waits for the semaphore.
//       Task 4 begins and waits for the semaphore.
//       Task 3 begins and waits for the semaphore.
//       Main thread calls Release(3) --> 3 tasks can enter the semaphore.
//       Task 4 enters the semaphore.
//       Task 1 enters the semaphore.
//       Task 3 enters the semaphore.
//       Task 4 releases the semaphore; previous count: 0.
//       Task 2 enters the semaphore.
//       Task 1 releases the semaphore; previous count: 0.
//       Task 3 releases the semaphore; previous count: 0.
//       Task 5 enters the semaphore.
//       Task 2 releases the semaphore; previous count: 1.
//       Task 5 releases the semaphore; previous count: 2.
//       Main thread exits.
```

Remarks

Semaphores are of two types: local semaphores and named system semaphores. Local semaphores are local to an application, system semaphores are visible throughout the operating system and are suitable for inter-process synchronization. The SemaphoreSlim is a lightweight alternative to the Semaphore class that doesn't use Windows kernel semaphores. Unlike the Semaphore class, the SemaphoreSlim class doesn't support named system semaphores. You can use it as a local semaphore only. The SemaphoreSlim class is the recommended semaphore for synchronization within a single app.

A lightweight semaphore controls access to a pool of resources that is local to your application. When you instantiate a semaphore, you can specify the maximum number of threads that can enter the semaphore concurrently. You also specify the initial number of threads that can enter the semaphore concurrently. This defines the semaphore's count.

The count is decremented each time a thread enters the semaphore, and incremented each time a thread releases the semaphore. To enter the semaphore, a thread calls one of the Wait or WaitAsync overloads. To release the semaphore, it calls one of the Release overloads. When the count reaches zero, subsequent calls to one of the `Wait` methods block until other threads release the semaphore. If multiple threads are blocked, there is no guaranteed order, such as FIFO or LIFO, that controls when threads enter the semaphore.

The basic structure for code that uses a semaphore to protect resources is:

VB

```vb
' Enter semaphore by calling one of the Wait or WaitAsync methods.
SemaphoreSlim.Wait()
'
' Execute code protected by the semaphore.
'
SemaphoreSlim.Release()
```

When all threads have released the semaphore, the count is at the maximum value specified when the semaphore was created. The semaphore's count is available from the CurrentCount property.

> ⓘ **Important**
>
> The **SemaphoreSlim** class doesn't enforce thread or task identity on calls to the **Wait**, **WaitAsync**, and **Release** methods. In addition, if the **SemaphoreSlim(Int32)** constructor is used to instantiate the **SemaphoreSlim** object, the **CurrentCount** property can increase beyond the value set by the constructor. It is the programmer's responsibility to ensure that calls to **Wait** or **WaitAsync** methods are appropriately paired with calls to **Release** methods.

Constructors

SemaphoreSlim(Int32)	Initializes a new instance of the SemaphoreSlim class, specifying the initial number of requests that can be granted concurrently.
SemaphoreSlim(Int32, Int32)	Initializes a new instance of the SemaphoreSlim class, specifying the initial and maximum number of requests that can be granted concurrently.

Properties

AvailableWaitHandle	Returns a WaitHandle that can be used to wait on the semaphore.
CurrentCount	Gets the number of remaining threads that can enter the SemaphoreSlim object.

Methods

Dispose()	Releases all resources used by the current instance of the SemaphoreSlim class.
Dispose(Boolean)	Releases the unmanaged resources used by the SemaphoreSlim, and optionally releases the managed resources.
Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)
GetHashCode()	Serves as the default hash function. (Inherited from Object)
GetType()	Gets the Type of the current instance. (Inherited from Object)
MemberwiseClone()	Creates a shallow copy of the current Object. (Inherited from Object)
Release()	Releases the SemaphoreSlim object once.
Release(Int32)	Releases the SemaphoreSlim object a specified number of times.
ToString()	Returns a string that represents the current object. (Inherited from Object)
Wait()	Blocks the current thread until it can enter the SemaphoreSlim.
Wait(CancellationToken)	Blocks the current thread until it can enter the SemaphoreSlim, while observing a CancellationToken.
Wait(Int32)	Blocks the current thread until it can enter the SemaphoreSlim, using a 32-bit signed integer that specifies the timeout.
Wait(Int32, CancellationToken)	Blocks the current thread until it can enter the SemaphoreSlim, using a 32-bit signed integer that specifies the timeout, while observing a CancellationToken.
Wait(TimeSpan)	Blocks the current thread until it can enter the SemaphoreSlim, using a TimeSpan to specify the timeout.
Wait(TimeSpan, CancellationToken)	Blocks the current thread until it can enter the SemaphoreSlim, using a TimeSpan that specifies the timeout, while observing a CancellationToken.
WaitAsync()	Asynchronously waits to enter the SemaphoreSlim.
WaitAsync(CancellationToken)	Asynchronously waits to enter the SemaphoreSlim, while observing

	a CancellationToken.
WaitAsync(Int32)	Asynchronously waits to enter the SemaphoreSlim, using a 32-bit signed integer to measure the time interval.
WaitAsync(Int32, CancellationToken)	Asynchronously waits to enter the SemaphoreSlim, using a 32-bit signed integer to measure the time interval, while observing a CancellationToken.
WaitAsync(TimeSpan)	Asynchronously waits to enter the SemaphoreSlim, using a TimeSpan to measure the time interval.
WaitAsync(TimeSpan, CancellationToken)	Asynchronously waits to enter the SemaphoreSlim, using a TimeSpan to measure the time interval, while observing a CancellationToken.

Applies to

Product	Versions
.NET	Core 1.0, Core 1.1, Core 2.0, Core 2.1, Core 2.2, Core 3.0, Core 3.1, 5, 6, 7, 8
.NET Framework	4.0, 4.5, 4.5.1, 4.5.2, 4.6, 4.6.1, 4.6.2, 4.7, 4.7.1, 4.7.2, 4.8, 4.8.1
.NET Standard	1.0, 1.1, 1.2, 1.3, 1.4, 1.6, 2.0, 2.1
UWP	10.0
Xamarin.iOS	10.8
Xamarin.Mac	3.0

Thread Safety

All public and protected members of SemaphoreSlim are thread-safe and may be used concurrently from multiple threads, with the exception of Dispose(), which must be used only when all other operations on the SemaphoreSlim have completed.

See also

- Semaphore and SemaphoreSlim