SpinLock Struct

Reference

Definition

Namespace: System.Threading

Assembly: System.Threading.dll

Provides a mutual exclusion lock primitive where a thread trying to acquire the lock waits in a loop repeatedly checking until the lock becomes available.

C#

```
public struct SpinLock
```

Inheritance Object → ValueType → SpinLock

Examples

The following example shows how to use a SpinLock:

C#

```
using System;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

class SpinLockDemo
{

    // Demonstrates:
    //      Default SpinLock construction ()
    //      SpinLock.Enter(ref bool)
    //      SpinLock.Exit()
    static void SpinLockSample1()
    {
        SpinLock sl = new SpinLock();

        StringBuilder sb = new StringBuilder();
```

```
        // Action taken by each parallel job.
        // Append to the StringBuilder 10000 times, protecting
        // access to sb with a SpinLock.
        Action action = () =>
        {
            bool gotLock = false;
            for (int i = 0; i < 10000; i++)
            {
                gotLock = false;
                try
                {
                    sl.Enter(ref gotLock);
                    sb.Append((i % 10).ToString());
                }
                finally
                {
                    // Only give up the lock if you actually acquired it
                    if (gotLock) sl.Exit();
                }
            }
        };

        // Invoke 3 concurrent instances of the action above
        Parallel.Invoke(action, action, action);

        // Check/Show the results
        Console.WriteLine("sb.Length = {0} (should be 30000)", sb.Length);
        Console.WriteLine("number of occurrences of '5' in sb: {0} (should be
3000)",
            sb.ToString().Where(c => (c == '5')).Count());
    }

    // Demonstrates:
    //      Default SpinLock constructor (tracking thread owner)
    //      SpinLock.Enter(ref bool)
    //      SpinLock.Exit() throwing exception
    //      SpinLock.IsHeld
    //      SpinLock.IsHeldByCurrentThread
    //      SpinLock.IsThreadOwnerTrackingEnabled
    static void SpinLockSample2()
    {
        // Instantiate a SpinLock
        SpinLock sl = new SpinLock();

        // These MRESs help to sequence the two jobs below
        ManualResetEventSlim mre1 = new ManualResetEventSlim(false);
        ManualResetEventSlim mre2 = new ManualResetEventSlim(false);
        bool lockTaken = false;

        Task taskA = Task.Factory.StartNew(() =>
```

```
        {
            try
            {
                sl.Enter(ref lockTaken);
                Console.WriteLine("Task A: entered SpinLock");
                mre1.Set(); // Signal Task B to commence with its logic

                // Wait for Task B to complete its logic
                // (Normally, you would not want to perform such a potentially
                // heavyweight operation while holding a SpinLock, but we do
it
                // here to more effectively show off SpinLock properties in
                // taskB.)
                mre2.Wait();
            }
            finally
            {
                if (lockTaken) sl.Exit();
            }
        });

        Task taskB = Task.Factory.StartNew(() =>
        {
            mre1.Wait(); // wait for Task A to signal me
            Console.WriteLine("Task B: sl.IsHeld = {0} (should be true)",
sl.IsHeld);
            Console.WriteLine("Task B: sl.IsHeldByCurrentThread = {0} (should
be false)", sl.IsHeldByCurrentThread);
            Console.WriteLine("Task B: sl.IsThreadOwnerTrackingEnabled = {0}
(should be true)", sl.IsThreadOwnerTrackingEnabled);

            try
            {
                sl.Exit();
                Console.WriteLine("Task B: Released sl, should not have been
able to!");
            }
            catch (Exception e)
            {
                Console.WriteLine("Task B: sl.Exit resulted in exception, as
expected: {0}", e.Message);
            }

            mre2.Set(); // Signal Task A to exit the SpinLock
        });

        // Wait for task completion and clean up
        Task.WaitAll(taskA, taskB);
        mre1.Dispose();
        mre2.Dispose();
    }
```

```
    // Demonstrates:
    //      SpinLock constructor(false) -- thread ownership not tracked
    static void SpinLockSample3()
    {
        // Create SpinLock that does not track ownership/threadIDs
        SpinLock sl = new SpinLock(false);

        // Used to synchronize with the Task below
        ManualResetEventSlim mres = new ManualResetEventSlim(false);

        // We will verify that the Task below runs on a separate thread
        Console.WriteLine("main thread id = {0}",
Thread.CurrentThread.ManagedThreadId);

        // Now enter the SpinLock.  Ordinarily, you would not want to spend so
        // much time holding a SpinLock, but we do it here for the purpose of
        // demonstrating that a non-ownership-tracking SpinLock can be exited
        // by a different thread than that which was used to enter it.
        bool lockTaken = false;
        sl.Enter(ref lockTaken);

        // Create a separate Task from which to Exit() the SpinLock
        Task worker = Task.Factory.StartNew(() =>
        {
            Console.WriteLine("worker task thread id = {0} (should be differ-
ent than main thread id)",
                Thread.CurrentThread.ManagedThreadId);

            // Now exit the SpinLock
            try
            {
                sl.Exit();
                Console.WriteLine("worker task: successfully exited SpinLock,
as expected");
            }
            catch (Exception e)
            {
                Console.WriteLine("worker task: unexpected failure in exiting
SpinLock: {0}", e.Message);
            }

            // Notify main thread to continue
            mres.Set();
        });

        // Do this instead of worker.Wait(), because worker.Wait() could in-
line the worker Task,
        // causing it to be run on the same thread.  The purpose of this exam-
ple is to show that
        // a different thread can exit the SpinLock created (without thread
```

```
tracking) on your thread.
        mres.Wait();

        // now Wait() on worker and clean up
        worker.Wait();
        mres.Dispose();
    }
}
```

Remarks

For an example of how to use a Spin Lock, see How to: Use SpinLock for Low-Level Synchronization.

Spin locks can be used for leaf-level locks where the object allocation implied by using a Monitor, in size or due to garbage collection pressure, is overly expensive. A spin lock can be useful to avoid blocking; however, if you expect a significant amount of blocking, you should probably not use spin locks due to excessive spinning. Spinning can be beneficial when locks are fine-grained and large in number (for example, a lock per node in a linked list) and also when lock hold-times are always extremely short. In general, while holding a spin lock, one should avoid any of these actions:

- blocking,
- calling anything that itself may block,
- holding more than one spin lock at once,
- making dynamically dispatched calls (interface and virtuals),
- making statically dispatched calls into any code one doesn't own, or
- allocating memory.

SpinLock should only be used after you have been determined that doing so will improve an application's performance. It is also important to note that SpinLock is a value type, for performance reasons. For this reason, you must be very careful not to accidentally copy a SpinLock instance, as the two instances (the original and the copy) would then be completely independent of one another, which would likely lead to erroneous behavior of the application. If a SpinLock instance must be passed around, it should be passed by reference rather than by value.

Do not store SpinLock instances in readonly fields.

Constructors

SpinLock(Boolean)	Initializes a new instance of the SpinLock structure with the option to track thread IDs to improve debugging.

Properties

IsHeld	Gets whether the lock is currently held by any thread.
IsHeldByCurrentThread	Gets whether the lock is held by the current thread.
IsThreadOwnerTrackingEnabled	Gets whether thread ownership tracking is enabled for this instance.

Methods

Enter(Boolean)	Acquires the lock in a reliable manner, such that even if an exception occurs within the method call, `lockTaken` can be examined reliably to determine whether the lock was acquired.
Exit()	Releases the lock.
Exit(Boolean)	Releases the lock.
TryEnter(Boolean)	Attempts to acquire the lock in a reliable manner, such that even if an exception occurs within the method call, `lockTaken` can be examined reliably to determine whether the lock was acquired.
TryEnter(Int32, Boolean)	Attempts to acquire the lock in a reliable manner, such that even if an exception occurs within the method call, `lockTaken` can be examined reliably to determine whether the lock was acquired.
TryEnter(TimeSpan, Boolean)	Attempts to acquire the lock in a reliable manner, such that even if an exception occurs within the method call, `lockTaken` can be examined reliably to determine whether the lock was acquired.

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 members of SpinLock are thread-safe and may be used from multiple threads concurrently.

See also

- SpinLock
- How to: Use SpinLock for low-level synchronization
- How to: Enable Thread-Tracking Mode in SpinLock