Performance Characteristics of New Synchronization Primitives in the .NET Framework 4

Emad Omara
Parallel Computing Platform
Microsoft Corporation

Introduction

The .NET Framework 4 introduces a set of new synchronization primitives designed primarily for one of three reasons: to simplify the coding of synchronization between threads, to improve performance when synchronizing between threads, or both. These new primitives include **ManualResetEventSlim**, **SemaphoreSlim**, **SpinLock**, **CountdownEvent** and **Barrier**. Among these primitives, the first three types address similar scenarios to those addressed by the existing types ManualResetEvent, Semaphore, and Monitor, while the remaining two address scenarios not well covered by earlier Framework releases. All of these new types are defined in the System.Threading namespace.

This document provides discussion and analysis of the performance of these new types, including an explanation for the methodology used in the analysis. The tests we employed in our analysis were run on a specific set of machines, the configurations for which are described in the appendix. Our performance analysis was based primarily on statistics around test completion time, and we expect completion times may diverge when tests are run on varying hardware. For this reason, the test results provided are to be used purely as a starting point for your own performance tuning efforts.

All of the performance tests use floating point operations to simulate real-world workloads, as in the following DoDummyWork method[1]:

```
[MethodImpl(MethodImplOptions.NoInlining)]
public static double DoDummyWork(long iterations)
{
    double a = 1000, b = 1000;
    double c = 0;
    for (long i = 0; i < iterations; i++)
    {
        c += a * b;
    }
    return c;
}
```

[1] We added the MethodImplOptions.NoInlining attribute to this method to prevent it from being inlined. If allowed to be inlined, the JIT compiler is then able to apply unanticipated optimiations, which nullify the test.

In each performance test, the new .NET 4 type is compared against a baseline type. If a similar type exists from previous versions of .NET Framework, we use that similar type for comparison (for example, we compare ManualResetEventSlim to ManualResetEvent, SemaphoreSlim to Semaphore, and SpinLock to Monitor). If there is no such pre-existing type, as is the case with CountdownEvent and Barrier, we use a minimal, custom implementation of the type's core functionality as baseline.

The two major parameters in the tests are the number of threads the tests are run with and the total cost of the workload being simulated. We ran each test under at least two parameter sets, one with varying thread count (N) while maintaining a fixed total workload (W) in order to measure the scalability (where each thread performs $\frac{W}{N}$ floating point operations), and the other with varying total workload while maintaining a fixed thread count in order to measure the performance of those workloads and the speedup of our types against the baseline implementations.

ManualResetEventSlim

ManualResetEventSlim is a lightweight synchronization event that provides similar intra-process synchronization capabilities to that of ManualResetEvent. Whereas ManualResetEvent is a very thin wrapper for an underlying kernel event object, ManualResetEventSlim maintains user-mode state which mirrors the configuration of such an underlying event. This enables ManualResetEventSlim to satisfy certain operations without having to make kernel transitions, and in some cases, without even having to allocate the underlying kernel object. This avoidance of both kernel transitions and kernel allocations leads to the "Slim" nomenclature.

ManualResetEvent invokes kernel functionality and blocks directly when its WaitOne method is called, whereas ManualResetEventSlim's Wait functionality first spins in user-mode and checks the user-mode event state in case it receives a signal (e.g. from a call to Set). If a signal was not received during the spinning phase, ManualResetEventSlim will fall back to a true kernel-based wait just as does ManualResetEvent. The spinning phase, however, is very important for scenarios where the waiting time is expected to be short. ManualResetEventSlim also provides some programming model niceties, such as a parameterless constructor and an inexpensive property **IsSet** which enables cheaply reading whether the event is set or not; with ManualResetEvent, this task is only possible by calling WaitOne with a timeout of zero.

Performance Test

ManualResetEventSlim is useful for increasing the scalability of scenarios in which the time between wait and set operations is expected to be short. For other scenarios with longer wait times, ManualResetEventSlim eventually delegates to an underlying ManualResetEvent, and thus adds very little overhead and does not affect scalability significantly. To aid in this performance test, we built a very simple MyCountdownEvent type using either ManualResetEventSlim and ManualResetEvent (note that for simplicity, we've kept the implementation to a minimum, including omitting all error checking).

```
    public class MyCountdownEvent
    {
          public int Count { get { return m_count; } }
          private int m_count;
#if SLIM
          private ManualResetEventSlim m_mre;
#else
        private ManualResetEvent m_mre;
#endif

        public SimpleCountdownEvent(int count)
        {
#if SLIM
            m_mre = new ManualResetEventSlim();
#else
            m_mre = new ManualResetEvent(false);
#endif
            m_count = count;
            if (count == 0) m_mre.Set();
        }

        public void Signal()
        {
            if (Interlocked.Decrement(ref m_count) == 0)
            {
                m_mre.Set();
            }
        }

        public void Wait()
        {
#if SLIM
            m_mre.Wait();
#else
            m_mre.WaitOne();
#endif
        }
    }
```

As is noted in the experiment details section at the end of this document, to reduce noise we had to ensure that the elapsed time of test iteration was more than 100 milliseconds. Hence, each thread performs signal and wait operations on an array of MyCountdownEvent objects instead of a single one, as shown below.

```
for (int i = 0; i < X; i++)
{
    DoDummyWork(totalWork / N);
    cde[i].Signal();
    cde[i].Wait();
}
```

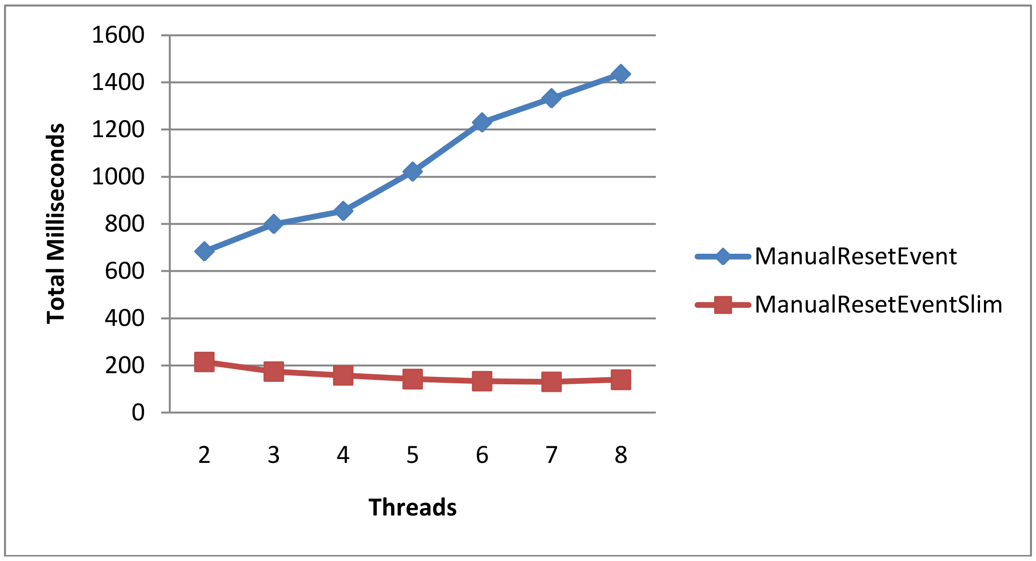


Figure 1: Varying thread count operating on ManualResetEvent / ManualResetEventSlim with a fixed total workload

Figure 1, which was generated using a workload of 4000 floating point operations for each iteration, shows the scalability of ManualResetEventSlim compared against that of ManualResetEvent. ManualResetEventSlim scales well as we increase the number of threads up to 7 in this test. As the number of threads increases, the local workload gets smaller, and as a result synchronization happens more frequently. There is a tipping point where the synchronization time becomes dominant of the overall execution time and scalability is lost, and in this test, that is when number of threads reaches 8. In comparison, Figure 1 also shows that ManualResetEvent does not scale at all, because the number of costly kernel transitions increases as the number of threads increases.

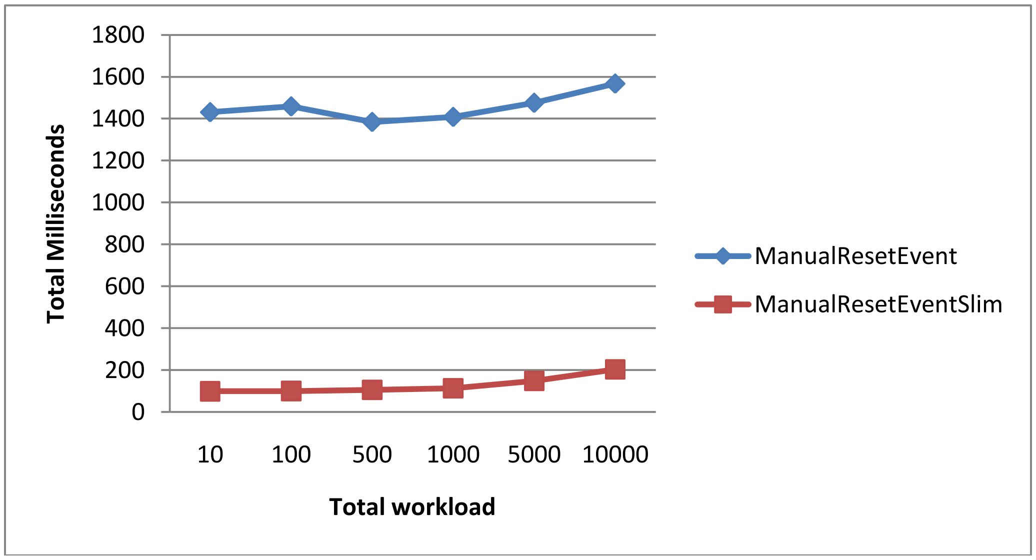


Figure 2: Eight threads operating on ManualResetEvent / ManualResetEventSlim with a varying total workload

Figure 2 shows the speedup of ManualResetEventSlim over ManualResetEvent for varying total workloads and a fixed number of threads (8). As is evident from the graph, ManualResetEventSlim is significantly faster in this test.

- ***ManualResetEventSlim is very fast for small and medium task sizes, and its performance decreases as the task size increase.***

Other Considerations

By default, ManualResetEventSlim's Wait method spins 10 iterations before yielding and blocking. This can be configured in the constructor by specifying for how many iterations you want to spin. Fine tuning the default spin iterations should be done only after studying the performance characteristics of your scenario.

Accessing the ManualResetEventSlim's **WaitHandle** property will force initialization of the underlying WaitHandle object, thereby increasing the memory footprint/overhead ofManualResetEventSlim. This, however, can be a necessary act if you need to synchronize across multiple ManualResetEventSlim instances, such as with the WaitHandle.WaitAll function.

SemaphoreSlim

SemaphoreSlim is a type similar to System.Threading.Semaphore, and it can be used to limit the number of concurrent accesses to a shared resource. As with ManualResetEvent, Semaphore will always transition into the kernel; in contrast, as with ManualResetEventSlim, SemaphoreSlim has been designed to stay in user-mode as much as possible, typically resulting in better performance and scalability for intra-process thread communication (unlike Semaphore, SemaphoreSlim cannot be used for inter-process coordination). SemaphoreSlim does not block directly when its Wait method is called while the semaphore's count is zero. Instead, it spins briefly first, checking the count along the way. If the count is still zero after the spinning phase, it then blocks. SemaphoreSlim also provides a **CurrentCount** property which gives access to that count value, typically used only for debugging purposes.

Performance Test

SemaphoreSlim is designed to increase scalability of scenarios that share resources between multiple threads within a process. The performance test we used to compare SemaphoreSlim against Semaphore first initializes the semaphore with an initial count of X and then spawns Y threads. Each of these threads waits on the semaphore, then performs $\frac{Z}{Y}$ work (where Z is the total work to be done), and finally releases the semaphore. This test was implemented using both SemaphoreSlim and Semaphore.

As shown below, to magnify the elapsed time within test iteration, the action was executed in a loop (where loopCount = 10000).

```
for (int i = 0; i < loopCount; i++)
{
    sem.Wait();
    DoDummyWork(totalWork / numberOfThreads);
    sem.Release();
}
```

As we did with the ManualResetEventSlim experiment, we want to understand the scalability of SemaphoreSlim as thread count increases.

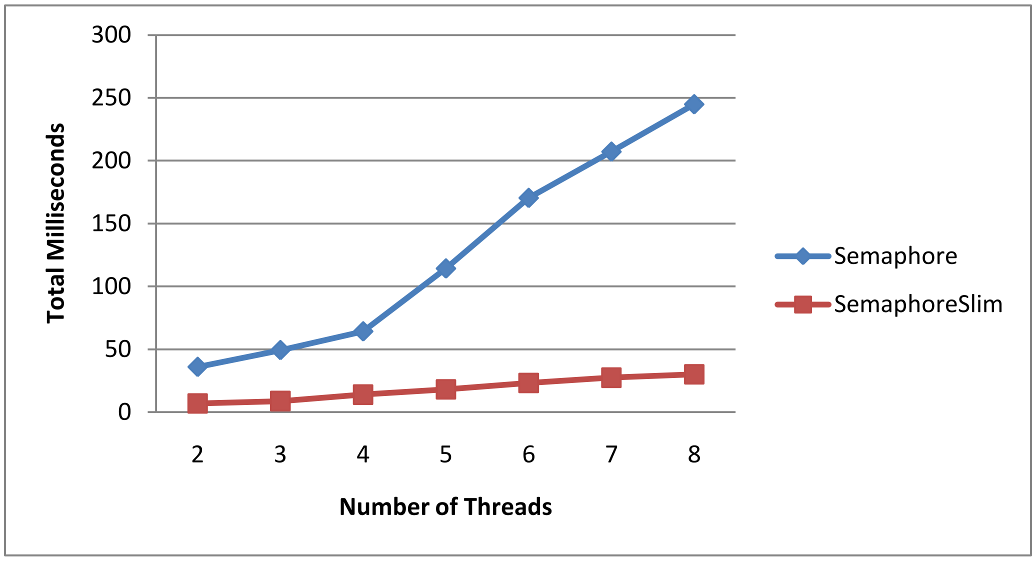


Figure 3: Varying thread count operating on a Semaphore and a SemaphoreSlim with a fixed total workload

Figure 3 represents the performance test for a fixed workload of 100 floating point operations per iteration and an initial semaphore count of 4. As shown, the elapsed time for both SemaphoreSlim and Semaphore goes up as the number of threads increases. This is expected, since the semaphore’s initial count is constant, so increasing the number of threads will force threads to block. However, SemaphoreSlim consistently runs faster. Also as the number of threads increases, SemaphoreSlim does not regress as much as Semaphore.

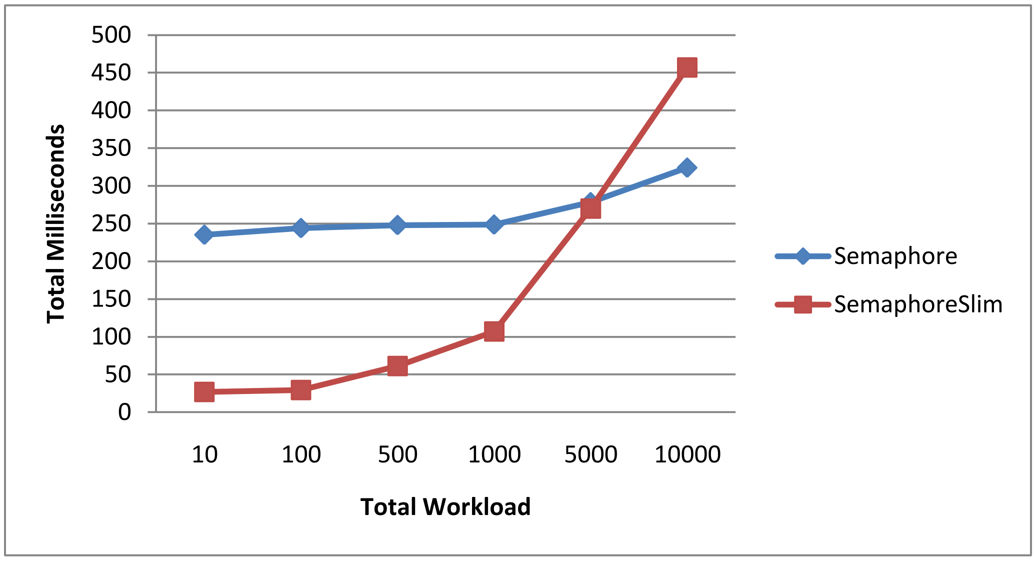


Figure 4: Eight threads operating on Semaphore / SemaphoreSlim with a varying total workload

Figure 4 graphs a fixed number of threads (8) with an initial semaphore count of 4 on varying sized workloads. This yields a very interesting picture, as it shows that SemaphoreSlim is much faster for small task sizes, with the gap between SemaphoreSlim and Semaphore decreasing as the workload increases. This is due to the overhead of SemaphoreSlim's extra spinning, since all Wait calls will spin prior to blocking if the SemaphoreSlim is still not signaled.

> ➢ ***SemaphoreSlim is very fast for small task sizes and its performance decreases as the task size increase.***

As one more test, we vary the initial semaphore count while keeping the thread count constant at 8 and the workload constant at 100. The results are shown in Figure 5. For both Semaphore and SemaphoreSlim, the total time decreases as the initial semaphore count increases, due to the resulting decrease in the number of blocked threads.

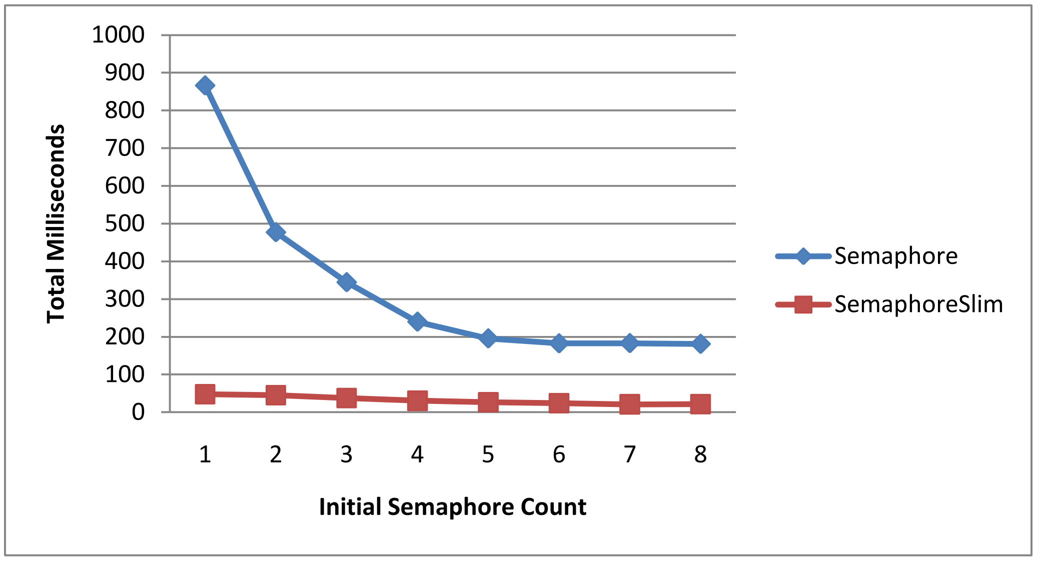


Figure 5: Varying initial semaphore counts

Other Considerations

As with ManualResetEventSlim, accessing SemaphoreSlim's AvailableWaitHandle property forces initialization of an underlying kernel object. This adds overhead to subsequent Wait and Release invocations, which now need to maintain the state of that WaitHandle.

Note that the name of this property, AvailableWaitHandle, differs from the name of the WaitHandle property on ManualResetEventSlim. This is due to a functional difference in the returned WaitHandle. Waiting on a ManualResetEvent or a ManualResetEventSlim does not change the signaled status of the event. However, waiting on a semaphore may mutate the semaphore, as it can decrease the semaphore's internal count. As such, the AvailableWaitHandle property returns a WaitHandle that merely indicates a high likelihood that there is count available for consumption in the SemaphoreSlim; to actually obtain access to the associated shared resource, however, one must wait on the SemaphoreSlim directly.

This all means that certain synchronization operations possible with Semaphore are not possible with SemaphoreSlim. For example, with Semaphore it is possible to use WaitHandle.WaitAll to atomically acquire resources from multiple semaphores; that functionality does not exist with SemaphoreSlim.

Barrier

System.Threading.Barrier is designed to coordinate participating threads in a way that blocks those threads until all have reached the barrier. Barriers are often used in search algorithms, distributed operations, and anywhere a fork/join or merge is necessary. Barrier is very much like a reusable CountdownEvent.

Common use cases involve multiple threads working on a partitioned pool of data, and blocking until all threads can move forward. A great example is any time a large data set can only be processed sequentially, but the data inside of an element may be processed in parallel. One example is processing an image, where line *n+1* cannot be processed before line *n* is processed, but columns within line *n* can be processed in parallel. Another example is batch file processing where order of lines must be strictly preserved but the contents of each line may be processed in parallel.

Performance Test

For comparative purposes, we built a simple barrier implementation using System.Threading.Monitor class:

```
public class SimpleBarrier
{
    public SimpleBarrier(int participantCount)
    {
        m_participantCount = participantCount;
    }

    static object m_lock = new object();
    int m_participantCount { get; set; }
    int m_curCount = 0;
    public void SignalAndWait()
    {
        lock (m_lock)
        {
            m_curCount++;

            if (m_curCount == m_participantCount)
            {
                m_curCount = 0;
                Monitor.PulseAll(m_lock);
            }
            else
            {
                Monitor.Wait(m_lock);
            }
        }
    }
```

The performance test spawns N threads; each thread iterates through X phases, and in each phase it does some workload and then calls SignalAndWait.

```
for (int j = 0; j < phaseCount; j++)
{
     DoDummyWork(workPerThread);
     barrier.SignalAndWait();
}
```

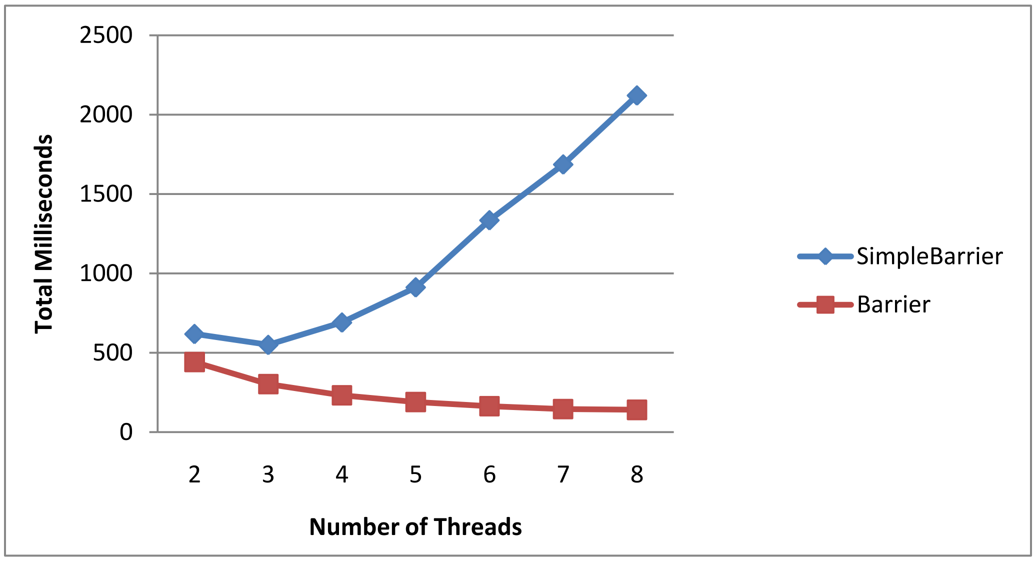
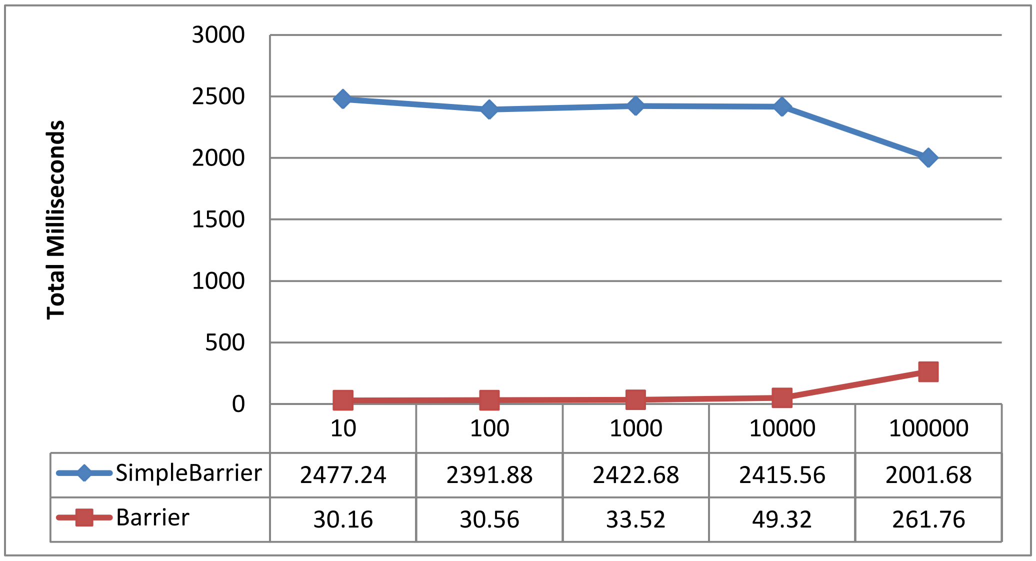


Figure 6: Varying thread count with Barrier and SimpleBarrier tests

Figure 8 shows that Barrier scales very well while SimpleBarrier doesn't scale at all. The SimpleBarrier suffers from a lot of kernel transitions in acquiring the lock to update the count and blocking until the phase is done. The new Barrier, on the other hand, uses Interlocked operations instead of locks to update the count, and uses ManualResetEventSlim for the blocking part (which spins a little before actually blocking).



	10	100	1000	10000	100000
SimpleBarrier	2477.24	2391.88	2422.68	2415.56	2001.68
Barrier	30.16	30.56	33.52	49.32	261.76

Figure 7: Fixed thread count and varying workload with Barrier and SimpleBarrier

In Figure 9, which shows the results from eight threads on varying workloads, we see that Barrier in .NET 4 is significantly faster than the simple version.

➢ ***Barrier is recommended to be used over a simple, lock-based approach whenever possible.***

SpinLock

A spin lock is a mutual exclusion lock primitive where a thread trying to acquire the lock waits in a loop ("spins") repeatedly trying to acquire the lock until it becomes available. As the thread remains active and doesn't block, the use of such a lock is a kind of busy waiting, consuming CPU resources without performing real work. However, spin locks can be more efficient than other kinds of locks on multi-processor machines if threads are only likely to be blocked for a very short period of time. This is due to potentially avoiding unnecessary context switches and associated kernel transitions, which might otherwise be more costly than spinning.

Writing a correct spin lock algorithm, however, requires extreme care. Improperly written ones can starve other logical processors (e.g. in systems with simultaneous multithreading, such as Intel's Hyper-threading), lead to priority inversion if the occasional yield is not issued correctly (the most popular approach, Thread.Sleep(0), isn't correct), and can make it difficult to perform a GC or break into a debugger. The new System.Threading.SpinLock type in .NET 4 provides reusable and correct spinning behavior.

SpinLock should be preferred over Monitor or kernel-based locks only in cases where the time the thread has to wait to acquire the lock and to execute the work protected by the lock is smaller than the amount of time to do the context switches and kernel transitions that would be necessary with other locks. In particular, only use SpinLock after you've tried another locking solution (e.g. Monitor) and found through performance testing that a spin lock would yield better results.

Performance Test

We compare the performance of SpinLock with Monitor. In this scenario, two or more threads attempt to acquire a SpinLock and release it after a very short period of time. N threads were spawned, each thread iterated X times, and in each iteration it acquires the lock, performs a constant amount of protected work, releases the lock, and finally does a variable amount of unprotected work.

```
for (int i = 0; i < loopCount; i++)
 {
     bool lockTaken = false;
     lock.Enter(o, ref lockTaken);
     DoDummyWork(100);
     lock.Exit(o);
     DoDummyWork(unprotectedWorkload / numberOfThreads);
 }
```

Figure 10 represents the scalability test of SpinLock versus Monitor using an `unprotectedWorkload` value of 4000 (floating point operations). The performance gap between SpinLock and Monitor is very tiny, due to the fact that Monitor also does some spinning internally before actually blocking.

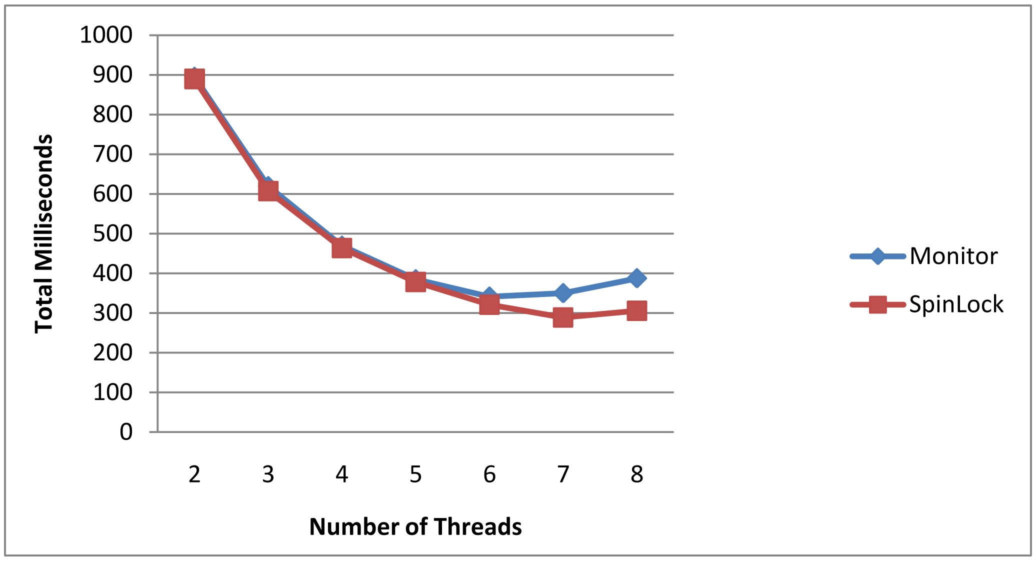


Figure 8 varying thread count with SpinLock and Monitor tests

Figure 11 represents the performance of SpinLock versus Monitor with a fixed number of threads (8) and varying workloads. The SpinLock is a little bit faster, but it is not a significant difference.

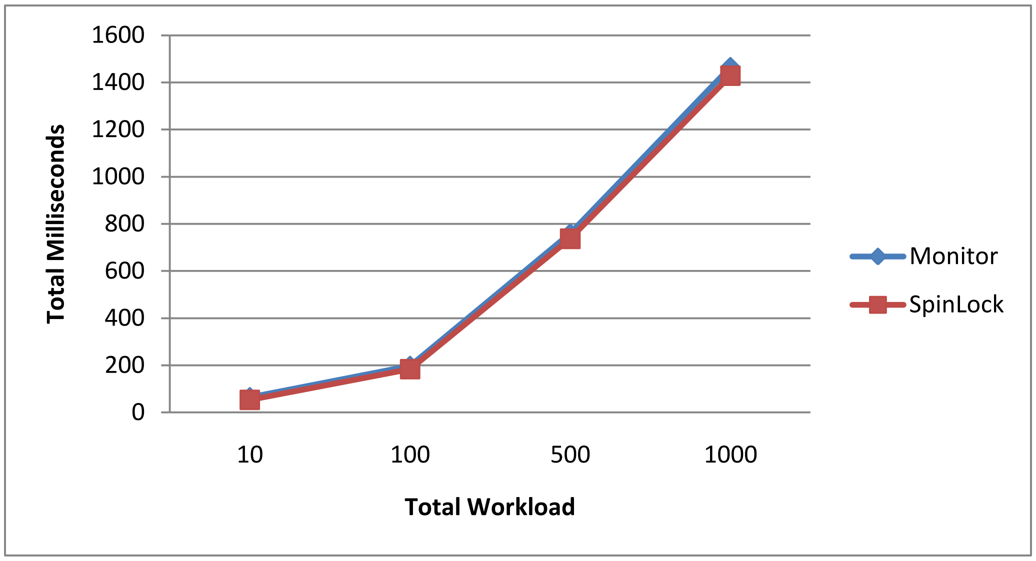


Figure 9 Elapsed time of SpinLock and Monitor tests on 8 threads using different workloads.

In general, SpinLock doesn't provide a significant performance gain over Monitor; as we explained before, Monitor also spins briefly before the actually blocking. However, Spinlock provides a very lightweight locking primitive memory wise, as it is only a 4-byte structure that doesn't require any heap allocation (which Monitor does require). As such, in scenarios where many locks need to be used (such as a lock-per-node in a tree structure or a list where each element is protected by a separate lock), SpinLock may result in less memory being allocated than would Monitor.

Other Considerations

SpinLock keeps track of the number of waiting threads, and if the number of waiting threads exceeded the number of physical cores on the machines, the extra threads will yield directly and will not spin at all. This policy conserves machine resources and achieves better performance than the simple SpinLock where each new thread spins if the lock is not available.

By default, SpinLock is enabled with a thread-tracking mode, which exists primarily for debugging purposes to enable the developer to determine which thread currently owns the lock. This, however, adds significant overhead. All of the above tests for SpinLock were performed with the thread tracking mode disabled, which can be done by passing a Boolean value of false to the SpinLock's constructor.

CountdownEvent

CountdownEvent is a synchronization primitive that unblocks its waiting threads after the event 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 maintain a count separate from the event. For example, in a fork/join scenario where you launch five asynchronous operations and don't want to make forward progress until all of them complete, you can create a CountdownEvent initialized with a count of 5, start the five operations, and have each of them invoke the CountdownEvent's Signal method when the operation completes. Each call to Signal decrements the signal count by one. On the main thread, a call to Wait will block until the signal count reaches zero.

Most of the new synchronizations primitives in .NET 4 were introduced for performance reasons rather than to improve upon the existing primitives' usability. For example, ManualResetEventSlim covers many of the scenarios targeted by ManualResetEvent scenarios, performing better in most of them. However, CountdownEvent was added to encapsulate a common pattern and to improve the programming model. As a result, it is not worth comparing CountdownEvent's performance against a simple implementation, as CountdownEvent is itself a relatively simple implementation and a performance analysis of it would yield uninteresting results when compared to a custom implementation based on the Interlocked class.

Other Considerations

CountdownEvent is a lock-free type that uses single variable to maintain the count. This variable is a single point of contention that could affect the performance when the task size is very small and the number of threads is large enough.

Details of the Experimental Framework

All tests were run on identical 8-core machines with the following hardware configurations:

- Intel ® Xeon® CPU E5345 @2.   3GHz, 2 sockets x 4-core.
- 8GB RAM
- .NET Framework 4 Beta 2
- All tests were run on both Windows Vista Ultimate 32-bit and Windows Vista Ultimate 64-bit

Each test was executed multiple times in an environment that reduced unnecessary external noise, and we computed the mean elapsed time as the standard measure. Our results had standard deviations of less than 10% of the mean. We used several approaches to reduce noise in order to obtain stable results:

- We stopped all unnecessary background services and applications and turned off network access and peripherals where possible.
- We selected test parameters such that the each individual test iteration took at least 100 milliseconds. This reduced most noise.
- To reduce the impact of garbage collection, the tests kept memory use to a minimum by using simple types and small data volumes, and we forced garbage collections between runs.
- All timing measurements excluded the time required to initialize the application, warm up the .NET Framework ThreadPool, and perform other house-keeping operations.

Due to the nature of our tests, they do not represent entire applications. Thus, the data we presented should be considered only indicative of the performance of similar units of a program on similar hardware, and not entire systems or applications.

Finally, we note that experiments run on 32-bit and 64-bit platforms may show significant variance in both speedup and scalability. There are many factors that can influence this variance and, in some cases, the differences favor one architecture over the other. If maximum performance is crucial and an application can run on either a 32-bit or a 64-bit platform, then specific performance measurements are required to select between the two alternatives. We structured our experiments and presentation of results to be agnostic to the specific choice of platform.