Benchmarking Multi-Threaded Data Synchronization

by James B. Higgins
http://OriginalCoder.dev

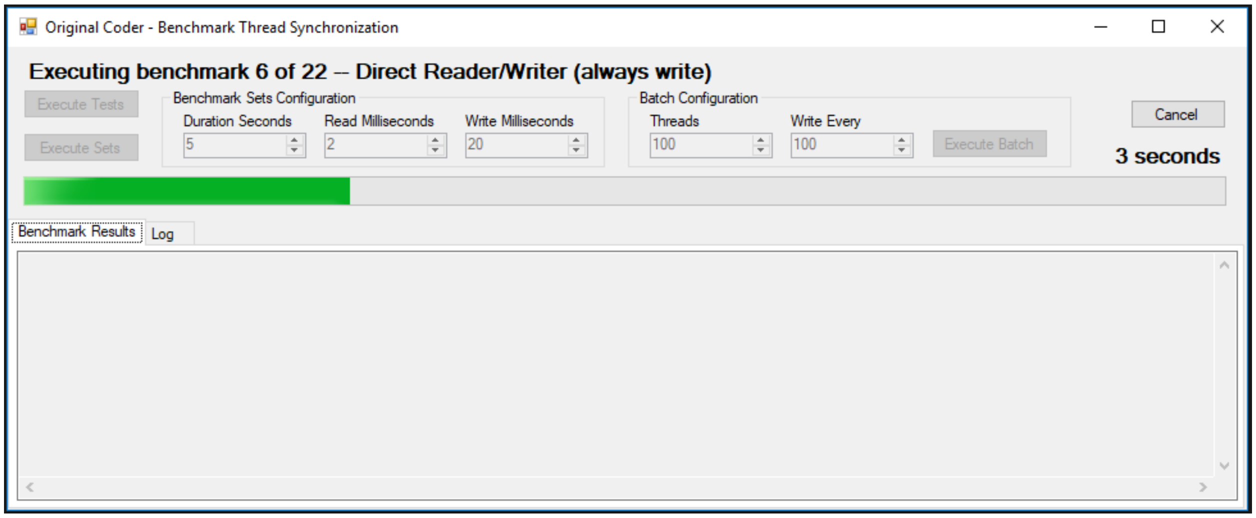
Written on July 22nd, 2019

When programming in C# there are many different methods available to synchronize data in a multi-threaded system. This article covers some benchmarking I performed to help answer the question of which synchronization methods are appropriate in various situations.

Table of Contents

APPLICATION & SOURCE CODE

In addition to the benchmarks themselves just as significant is the source code used to perform them. The benchmark application implements all of the synchronization methods being benchmarked plus also uses quite a lot of multi-threading itself. I figured other developers might find it illustrative or at least interesting to take a look at.



NOTICE: The application code is protected by copyright and provided only for reference purposes. Please do not republish or copy any of the code. The code works fine for my purposes here but various portions of the code (namely the included interfaced locking pattern implementations) have not been seriously tested and I would not trust the code for production use! Other portions, such as the logging and GUI synchronization code, are overly simplistic and only applicable for this particular use. I have extensive libraries which have been tested and used over many years which contain the locking, logging and other code I use for production implementations. None of this code was not taken from my libraries, all of this code was thrown together over a weekend because I felt like playing with some locking benchmarks.

I plan to release portions of my C# libraries via GitHub in the future and have already started some of the refactoring and minor cleanup necessary. That will include the implementation of my Interfaced Thread Locking pattern along with much else. Keep an eye on my blog at http://OriginalCoder.dev for news and future developments.

The benchmarking application is written using WinForms and uses a lot of multi-threading capabilities:

- Creates numerous Thread instances that must be initialized, orchestrated and then cleaned up.
- Performs all benchmarking in a background thread so that the GUI can continue being refreshed.
- Displays ongoing progress of the benchmarking in the GUI.
- Allows the user to cancel benchmarking via the GUI.
- Implements a simple thread-safe logging system.

The following are used from System.Threading:

- Barrier
- Interlocked.Increment
- LockCookie
- ManualResetEvent
- Monitor
- Mutex
- ReaderWriterLock
- ReaderWriterLockSlim
- Semaphore
- SemaphoreSlim
- Thread
- Thread.CurrentThread
- Thread.Sleep

These are used from System.Threading.Tasks:

- Task.Run
- Task<TResult>
- TaskStatus

Quick Multi-threading Overview

First a quick bit of background for anyone reading this who hasn't done multi-threaded programming.

When writing code that will be executed using multiple threads there are a lot of important considerations. This is not an article about how to write multi-threaded code, so I'm not going to cover most of those topics. The topic central to this article is synchronizing access to shared data. When multiple threads are reading and writing shared data great care must be taken to make sure that data does not become corrupt and that the individual threads get accurate data when reading. The more complicated and inter-related the data is the more likely problems would occur without proper synchronization. But, to be clear, all multi-threaded code should always use appropriate synchronization strategies to ensure the shared data never gets corrupted and read operations don't get bad data.

Essentially only a single thread should be allowed to write to shared data at a time and no threads should be allowed to read the shared data while a write is occurring. The easiest way to implement this is to have a locking mechanism that only allows a single thread to hold the lock at a time and then have any thread needing to either read or write the data obtain the lock before doing so. This is the simplest implementation (such as using the "lock" keyword) but can have performance issues if there are many threads and most of the time they only read data because the threads will block each other from reading simultaneously. For this reason, some locking mechanisms (such as ReaderWriterLock) support separate read and write locks which allow any number of threads to obtain a read lock simultaneously, but make write locks exclusive including denying reads when a write lock is held.

A quick example would be code in multiple threads working with the same Dictionary<string, object> structure. If 2 separate threads were to make changes to the dictionary at the same time the internal data structures could get corrupted because the operations would likely step over each other. Only a single thread should be allowed to write to shared data at one time, so an exclusive write lock should be obtained before making any changes. That would ensure that the Dictionary never becomes corrupt, but threads that only read would still have problems if they are not included in synchronization. Consider a simple case where a thread gets the Count of the dictionary which is then used to make a decision before attempting to read specific keys. If the reading thread gets the count then gets suspended (put on hold) while another thread performs a write operation, when the reading thread continues the data in the dictionary has changes so the count is different, but the reading thread will continue doing its work as if the count was the same. This makes the work performed by the reading thread invalid which is why reading also requires synchronization.

One other detail to note is that when implementing multi-threaded code it is important to reduce the amount of time spent inside any lock as much as possible. Because many threads may be competing for access to the locks it is important to reduce contention as much as possible. Never obtain a lock and then perform a bunch of work / calculations inside the lock when that work could have been done in preparation before obtaining the lock. The ideal situation is where all work and calculations can be performed outside of the lock and then only a few quick & simple reads or writes are necessary from inside the lock. That is, of course, not always possible. If for example the total sum for a particular property for all objects contained in a shared list is required, then a read lock would need to be obtained before iterating through the list and computing the total.

Data Synchronization Options

The following list of .NET Framework classes all provide very similar capabilities to synchronize data in a multi-threaded environment. All of these provide a locking mechanism that, once obtained, any number of operations can be performed while the lock is held. Most of these only provide a single locking mechanism (for both reads and writes), but a couple provide for separate read and write locks.

- lock (keyword)
- Monitor
- Mutex
- ReaderWriterLock
- ReaderWriterLockSlim
- Semaphore
- SemaphoreSlim

In addition, there is the "lock" keyword built into C# which is just a syntactic shortcut for using a Monitor. I do benchmark these separately below, but for discussion I'll just talk about Monitor.

Monitor and Mutex both provide the capability of a single exclusive lock. When using either of these both read and write operations obtain the same lock. This means that multiple reading threads will block each other. If there were many threads trying to read they would essentially queue up waiting for the lock and then proceed through single file. Depending on the exact implementation details this could remove any advantages of using multiple threads.

Semaphore and SemaphoreSlim also provide the capability of a single lock, but can potentially allow multiple threads to obtain one simultaneously depending on how they are configured. For this article and the benchmarks I configured the Semaphores so that they only allowed a single thread to obtain a lock at one time which made them operate similarly to Monitor and Mutex. As such the same caveats above about readers blocking each other and queueing up also apply to Semaphores in this context.

Both ReaderWriterLock and ReaderWriterLockSlim provide the same capabilities, but are different implementations. These classes provide separate locks for reading and writing where any number of readers are allowed simultaneously but only a single write lock is allowed (which also blocks any readers when held). Used properly these classes can greatly improve performance when many threads are executing and most operations are read only. Because these classes provide more capabilities they can be implemented and used in different ways, so I've benchmarked each of these classes 3 different ways. First using read locks when reading and then only obtaining a write lock when a write is necessary. Second, always obtaining an upgradable read lock and then upgrading it to a write when necessary. Lastly, I also benchmarked these where all operations obtained a write lock. The last one, only using write locks, is bad practice and makes these classes operate similarly to a Monitor and Mutex (destroying the point of using these instead of one of those). I do not at all recommend ever doing that in code, but I decided to benchmark the write only case just for informational comparison purposes.

Eliminating the use of either Reader/Writer class as write-only, that gives us 9 different implementation possibilities (including the "lock" keyword) for implementing data synchronization. That's 5 different implementations to do write-only locking and 4 different implementations to provide separate read and write locks.

Special case: Interlocked

The Interlocked class is something of a special case when it comes to data synchronization because it performs synchronization very differently than the other options. The above synchronization methods provide a locking mechanism that, once obtained, any number of operations can be performed while the lock is held. But the Interlocked class provides methods for atomic data synchronization. Atomic means that the write operation occurs in a single step and will never conflict with other threads. This only works for a single, simple data type (such as an integer counter) and does not apply to cases where there are multiple data items that are related and must be kept in sync. For cases where Interlocked does apply it is by far the fastest and easiest to use.

Interlocked offers a number of possible operations, for the purpose of these benchmarks I only used Interlocked.Increment because the benchmarking code was written around reading and incrementing a shared integer counter.

My Interfaced Locking Pattern

Implementing locking code correctly can be a bit tricky and it is all too easy to make a mistake that results in a lock never being properly released. If a lock is acquired and does not get released the system will freeze up. For this reason, locking should always be implemented using try/finally blocks. But even then it is quite easy to make a small mistake in the code that could cause the lock not to be released under certain circumstances. Finding bugs of this nature is extremely difficult and time consuming because they tend to occur unpredictably and unless the system happens to freeze up for a developer while the debugger is running figuring out exactly where the problem occurs can be nearly impossible.

For these reasons any pattern that greatly reduces or eliminates the change of programmer error when using locks is highly desirable. The lock keyword built into C# does this because behind the scenes it automatically adds a try/finally block that will always release the lock appropriately. This is great if the using Monitor for locking aligns well with the work being done by the system. But it does not help in cases where Monitor does not perform well (such as when there are many more concurrent reads than writes).

The solution I came up with years ago (before the lock keyword existed) was to create a pattern that uses a locking mechanism exposed via an interface that implements IDisposable to handle releasing of the lock. With this pattern locks are always obtained via a using statement which will always ensure the lock gets released correctly (much like the lock keyword). I've found this pattern works extremely well and since adopting it I can't remember a single case where my code didn't correctly release a lock.

Do note that this pattern may not be 100% perfect though (especially depending on exactly how it is implemented). In environments where a lot of threads get aborted using ThreadAbortException it may be theoretically possible for the exception to get thrown between when the lock is acquired and the using block takes effect. I'm not completely certain about this and would need to carefully analyze the CIL that gets produced by the using statement to figure it out. It would be a rare occurrence in any case and would never be a meaningful issue for systems that don't utilize ThreadAbortException except during shutdown. This is important to be aware of when considering adopting this pattern. In my personal experience the elimination of coding errors for locks has been worth the slight risks in the situations I've used this pattern. It is worth noting that I read something from Microsoft (release notes?) that stated the lock keyword also had this same risk under some circumstances in the past (I believe reading that they eventually fixed it).

The lock keyword coding pattern and my interfaced locking pattern are very similar except that the interfaced pattern requires instantiating a class that implements the interface before it can be used. This can easily be viewed as a benefit though, because it allows the developer to choose the underlying locking mechanism instead of having it chosen for them (lock always uses Monitor).

The lock keyword coding pattern:

```
lock (someObject)
{
    // perform work inside of lock
}
```

My interfaced locking pattern:

```
private readonly _lock = new ThreadLockReadWrite();

using (_lock.ReadLock())
{
    // perform read-only work inside of lock
}

using (_lock.WriteLock())
{
    // perform work inside of lock
}

using (_lock.ReadUpgradableLock())
{
    // perform read work inside of lock
    using (_lock.UpgradeLock())
    {
        // perform write work inside of lock
    }
}
```

USE CASES

Benchmarking multi-threaded data synchronization isn't as simple as running a few benchmarks and picking the fastest. The environment in which the data synchronization will occur has a very large impact on which implementation performs the best. There are a few key factors that help determine this which I've factored into the benchmarks.

Number of threads - How many threads will be running simultaneously and needed access to the shared data is an important consideration. Separate benchmarks for 1, 2, 10 and 100 threads were performed.

Reads vs Writes - How often threads will only need read access compared to how often they will modify the data is a critical factor. Separate benchmarks for writing 100%, 10% and 1% of the time were performed.

Amount of time locks are held - This can be a factor in the decision making, but for these benchmarks I've used consistent delay times to similar work being done inside read & write locks. Since the goal is to keep time inside of locks to the minimal possible and the biggest impact here is how much time is spent in read locks this decision can be simplified. The key factor here is that if threads need to do a lot of work inside read operations then it is important to make sure reads do not bock each other. For these situations either ReaderWriterLock or ReaderWriterLockSlim should always be used with separate locks for reading and writing.

RECOMMENDATIONS BASED ON THE BENCHMARK RESULTS

Interlocked

When the data to be synchronized is simple, not inter-related and lends itself to atomic operations use the Interlocked class. While there are a few cases where the Reader/Writer locks perform equally well there are no cases where anything outperform Interlocked. So use it whenever possible.

Mostly Writes

When most of the operations are writes - Use the lock keyword. When most operations require write access all of the locking mechanisms (except Interlocked, see above) perform about the same. The lock keyword provides a very nice coding pattern that prevents errors caused by locks not getting released.

ReaderWriterLock vs ReaderWriterLockSlim

The API for using the ReaderWriterLockSlim class is easier to work with compared to the ReaderWriterLock class. This is particularly true when dealing with upgrading read locks to write locks. Unfortunately, something is wrong inside the Slim class that can causes its lock upgrading to execute very slowly so I would highly recommend against ever using ReaderWriterLockSlim with upgradable locks. When a lot of threads are involved the ReaderWriterLock (non-slim) class also tends to outperform the Slim variant by a noticeable amount. Given those I'd recommend using ReaderWriterLock at this time instead of ReaderWriterLockSlim.

Read/Write vs Read/Upgrade

WARNING: If using the ReaderWriterLockSlim class DO NOT obtain an upgradable read lock and then upgrade to write locks. The benchmarks show that, for some reason, this can execute VERY SLOWLY and tends to perform closer to mechanisms that only supply a single (write) lock such as Monitor. Please note that, for whatever reason, this does not happen with the ReaderWriterLock implementation only the Slim variant! Sadly the API for the slim variant is much easier to use and implementing read->upgrade with the non-slim class is somewhat of a pain to code. This is another case where my interfaced locking pattern would be very helpful (since it hides the additional complexity required for the non-slim class).

When dealing with a modest number of threads obtaining a read lock than then a write lock when needed appears to run a bit faster than obtaining an upgradable read lock and then upgrading to a write lock when needed. The catch is that with the non-upgradable the read lock must be released before obtaining the write lock which probably requires some additional thought to implement the threaded code correctly. The performance difference doesn't seem to be very much so the reduced complexity of read->upgrade is probably warranted for most circumstances.

Performance for read->upgrade was particularly good for the benchmarks with 100 threads where reads were 99.9% (only 1 out of 1,000 operations was a write). So it may be worth considering using the upgradable locks when dealing with very many threads that only read the vast majority of the time. But as mentioned elsewhere in this document never use the ReaderWriterLockSlim with upgradable locks because it has internal problems.

My Interfaced Locking Pattern

My most interesting take away from these benchmarks is how my interfaced locking pattern performs compared to direct, in-place implementations. I had assumed that since the pattern instantiates objects to handle the IDisposable which are then discarded this additional work would have at least a small but noticeable impact on performance. Surprisingly the interfaced vs direct implementations of the same underlying mechanism tended to perform equally well in the benchmarks. This is because the amount of additional work for the interfaced pattern is small and the amount of simulated work in the threads is much more significant. For cases where locks were held for extremely short durations the ratio would change and the interfaced pattern may perform slower than the direct alternative. But, as noted above, the point of my interfaced locking pattern is to reduce bugs and program freezes so a minor performance cost would be fine in most cases.

Benchmark Results

Benchmarks with a Single Thread

If a single thread will be used all of the locking mechanisms are almost identical in their performance so which one is chosen doesn't matter much. Even Interlock.Increment doesn't execute faster when only a single thread is used. Something to consider: If there is only a single thread working with the data are you sure you need thread synchronization? This might occur if most of the time only a single thread will be running but under certain circumstances / high load additional threads could be started. If that is the case I'd suggest optimizing for when multiple threads are running.

Benchmarks with Two Threads

Mostly writes - When most (or all) operations are writes then Interlocked.Increment is about twice as fast as anything else and all of the alternatives are roughly equal in performance. If possible use Interlocked, otherwise I'd suggest using the lock keyword for these situations.

10x Reads - When reads occur 90% of the time Interlocked.Increment is still fastest, but the Reader/Writer locks aren't far behind. All of the other mechanisms (that only implement a single write lock) are equally slower. For high performance systems the additional complexity of reader/writer locks with separate read & write locks would be recommended. But the simplicity of the lock keyword may outweigh the performance gains in some circumstances.

100x Reads - When reads occur 99% or more of the time the Reader/Writer locks are almost as fast as Interlocked.Increment with all of the other alternatives being much slower. I'd definately recommend using a Reader/Writer with separate read & write locks in all cases under these circumstances.

10 Threads (close to the number of physical threads in my CPU)

Mostly writes - When most (or all) operations are writes then Interlocked.Increment is 10 times as fast as anything else and all of the alternatives are roughly equal in performance. If possible use Interlocked, otherwise I'd suggest using the lock keyword for these situations.

10x Reads - When reads occur 90% of the time Interlocked.Increment is still fastest by a wide margin, so use that if possible. The Reader/Writer locks perform much better (about 3.5 times faster) than the other single (write only) locking mechanisms. I'd recommend using a Reader/Writer lock implementation with separate read & write locks under these circumstances.

100x Reads - When reads occur 99% or more of the time the Reader/Writer locks are almost as fast as Interlocked.Increment with all of the other alternatives being much slower. Separate Reader/Writer locks perform about 8.5 times faster than the single (write only) lock alternatives. I'd definitely recommend using a Reader/Writer with separate read & write locks in all cases under these circumstances.

Benchmarks with Many (100) Threads

Mostly writes - When most (or all) operations are writes then Interlocked.Increment is almost 100 times faster than anything else! When possible definitely use Interlocked. For locking situations that can't be handled with Interlocked I'd suggest using the lock keyword.

10x Reads - When reads occur 90% of the time Interlocked.Increment is still by far fastest (~10 times faster than Reader/Writer locks). When possible definitely use Interlocked. If Interlocked can't be used the Reader/Writer locks perform about 10 times faster than their single (write only) lock alternatives. I'd definitely recommend using a Reader/Writer lock implementation with separate read & write locks under these circumstances.

100x Reads - When reads occur 99% or more of the time the Interlocked.Increment still pulls ahead with this many threads involved. Interlocked.Increment is about 50% faster than using Reader/Writer locks but is far more restrictive. I'd probably use Interlocked if possible. Separate Reader/Writer locks perform about 15 times faster than the single (write only) lock alternatives. I'd definitely recommend using a Reader/Writer with separate read & write locks in all cases under these circumstances.