Locks Aren't Slow; Lock Contention Is

Locks (also known as **mutexes**) have a history of being misjudged. Back in 1986, in a Usenet discussion on multithreading, Matthew Dillon wrote, "Most people have the misconception that locks are slow." 25 years later, this misconception still seems to pop up once in a while.

It's true that locking is slow on some platforms, or when the lock is highly contended. And when you're developing a multithreaded application, it's very common to find a huge performance bottleneck caused by a single lock. But that doesn't mean all locks are slow. As I'll show in this post, sometimes a locking strategy achieves excellent performance.

Perhaps the most easily-overlooked source of this misconception: Not all programmers may be aware of the difference between a lightweight mutex and a "kernel mutex". I'll talk about that in my next post, Always Use a Lightweight Mutex. For now, let's just say that if you're programming in C/C++ on Windows, the Critical Section object is the one you want.

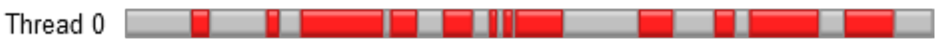
Other times, the conclusion that locks are slow is supported by a benchmark. For example, this post measures the performance of a lock under heavy conditions: each thread must hold the lock to do any work (high contention), and the lock is held for an extremely short interval of time (high frequency). It's a good read, but in a real application, you generally want to avoid using locks in that way. To put things in context, I've devised a benchmark which includes both best-case and worst-case usage scenarios for locks.

Locks may be frowned upon for other reasons. There's a whole other family of techniques out there known as lock-free (or lockless) programming. Lock-free programming is extremely challenging, but delivers huge performance gains in a lot of real-world scenarios. I know programmers who spent days, even weeks fine-tuning a lock-free algorithm, subjecting it to a battery of tests, only to discover hidden timing bugs several months later. The combination of danger and reward can be very enticing to a certain kind of programmer – and this includes me, as you'll see in future posts! With lock-free techniques beckoning us to use them, locks can begin to feel boring, slow and busted.

But don't disregard locks yet. One good example of a place where locks perform admirably, in real software, is when protecting the memory allocator. Doug Lea's Malloc is a popular memory allocator in video game development, but it's single threaded, so we need to protect it using a lock. During gameplay, it's not uncommon to see multiple threads hammering the memory allocator, say around 15000 times per second. While loading, this figure can climb to 100000 times per second or more. It's not a big problem, though. As you'll see, locks handle the workload like a champ.

Lock Contention Benchmark

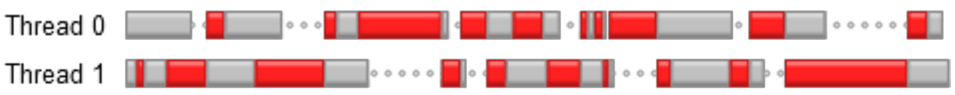
In this test, we spawn a thread which generates random numbers, using a custom Mersenne Twisterimplementation. Every once in a while, it acquires and releases a lock. The lengths of time between acquiring and releasing the lock are random, but they tend towards average values which we decide ahead of time. For example, suppose we want to acquire the lock 15000 times per second, and keep it held 50% of the time. Here's what part of the timeline would look like. Red means the lock is held, grey means it's released:



This is essentially a Poisson process. If we know the average amount of time to generate a single random number – **6.349 ns** on a 2.66 GHz quad-core Xeon – we can measure time in *work units*, rather than seconds. We can then use the technique described in my previous post, How to Generate Random Timings for a Poisson Process, to decide how many work units to perform between acquiring and releasing the lock. Here's the implementation in C++. I've left out a few details, but if you like, you can download the complete source code here.

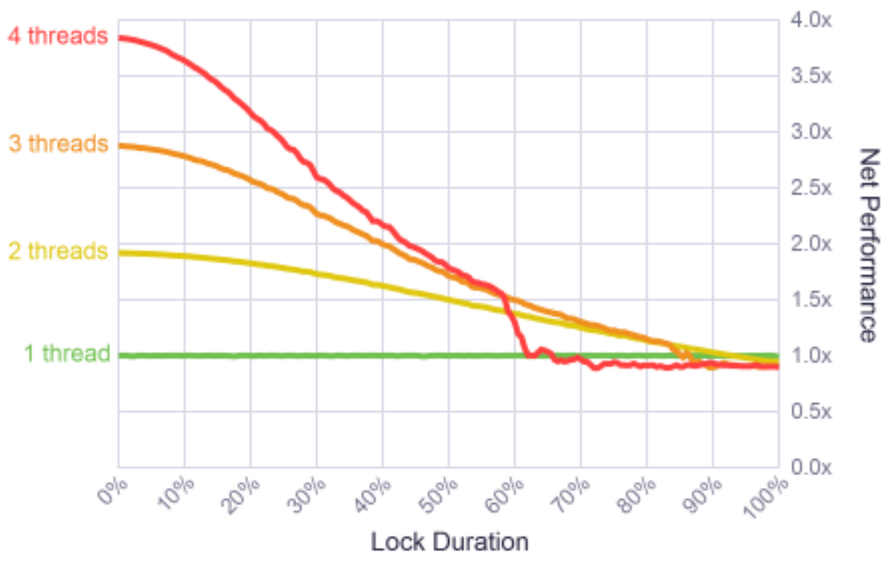
```
QueryPerformanceCounter(&start);
for (;;)
{
    // Do some work without holding the lock
    workunits = (int) (random.poissonInterval(averageUnlockedCount) + 0.5f);
    for (int i = 1; i < workunits; i++)
        random.integer();         // Do one work unit
    workDone += workunits;

    QueryPerformanceCounter(&end);
    elapsedTime = (end.QuadPart - start.QuadPart) * ooFreq;
    if (elapsedTime >= timeLimit)
        break;

    // Do some work while holding the lock
    EnterCriticalSection(&criticalSection);
    workunits = (int) (random.poissonInterval(averageLockedCount) + 0.5f);
    for (int i = 1; i < workunits; i++)
        random.integer();         // Do one work unit
    workDone += workunits;
    LeaveCriticalSection(&criticalSection);

    QueryPerformanceCounter(&end);
    elapsedTime = (end.QuadPart - start.QuadPart) * ooFreq;
    if (elapsedTime >= timeLimit)
        break;
}
```

Now suppose we launch two such threads, each running on a different core. Each thread will hold the lock during 50% *of the time when it can perform work*, but if one thread tries to acquire the lock while the other thread is holding it, it will be forced to wait. This is known as **lock contention**.



In my opinion, this is a pretty good simulation of the way a lock might be used in a real application. When we run the above scenario, we find that each thread spends roughly 25% of its time waiting, and 75% of its time doing actual work. Together, both threads achieve a net performance of **1.5x** compared to the single-threaded case.

I ran several variations of the test on a 2.66 GHz quad-core Xeon, from 1 thread, 2 threads, all the way up to 4 threads, each running on its own core. I also varied the duration of the lock, from the trivial case where the the lock is never held, all the way up to the maximum where each thread must hold the lock for 100% of its workload. In all cases, the lock frequency remained constant – threads acquired the lock 15000 times for each second of work performed.



The results were interesting. For short lock durations, up to say 10%, the system achieved very high parallelism. Not perfect parallelism, but close. Locks are fast!

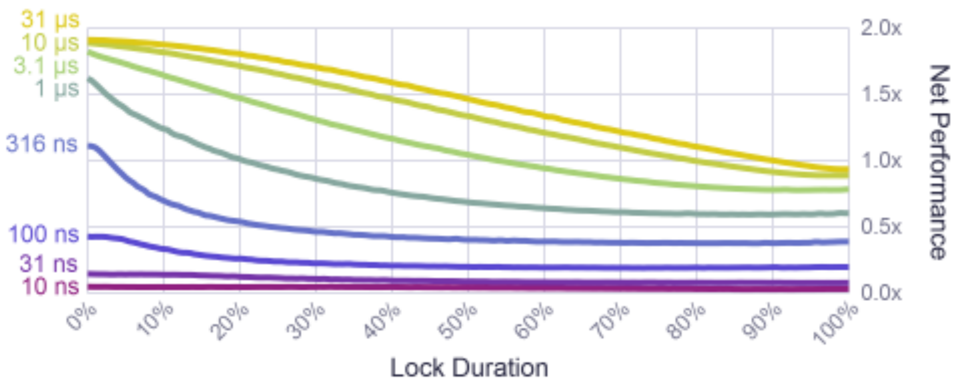
To put the results in perspective, I analyzed the memory allocator lock in a multithreaded game engine using this profiler. During gameplay, with 15000 locks per second coming from 3 threads, the lock duration was in the neighborhood of just **2%**. That's well within the comfort zone on the left side of the diagram.

These results also show that once the lock duration passes 90%, there's no point using multiple threads anymore. A single thread performs better. Most surprising is the way the performance of 4 threads drops off a cliff around the 60% mark! This looked like an anomaly, so I re-ran the tests several additional times, even trying a different testing order. The same behavior happened consistently. My best hypothesis is that the experiment hits some kind of snag in the Windows scheduler, but I didn't investigate further.

Lock Frequency Benchmark

Even a lightweight mutex has overhead. As my next post shows, a pair of lock/unlock operations on a Windows Critical Section takes about **23.5 ns** on the CPU used in these tests. Therefore, 15000 locks per second is low enough that lock overhead does not significantly impact the results. But what happens as we turn up the dial on lock frequency?

The algorithm offers very fine control over the amount of work performed between one lock and the next, so I performed a new batch of tests using smaller amounts: from a very fine-grained 10 ns between locks, all the way up to 31 μs, which corresponds to roughly 32000 acquires per second. Each test used exactly two threads:



As you might expect, for very high lock frequencies, the overhead of the lock itself begins to dwarf the actual work being done. Several benchmarks you'll find online, including the one linked earlier, fall into the bottom-right corner of this chart. At such frequencies, you're talking about some seriously short lock times – on the scale of a few CPU instructions. The good news is that, when the work between locks is that simple, a lock-free implementation is more likely to be feasible.

At the same time, the results show that locking up to 320000 times per second (3.1 μs between successive locks) is not unreasonable. In game development, the memory allocator may flirt with this frequency during load times. You can still achieve more than 1.5x parallelism if the lock duration is short.

We’ve now seen a wide spectrum of lock performance: cases where it performs great, and cases where the application slows to a crawl. I’ve argued that the lock around the memory allocator in a game engine will often achieve excellent performance. Given this example from the real world, it cannot be said that *all*locks are slow. Admittedly, it’s very easy to abuse locks, but one shouldn’t live in too much fear – any resulting bottlenecks will show up during careful profiling. When you consider how reliable locks are, and the relative ease of understanding them (compared to lock-free techniques), locks are actually pretty awesome sometimes.

The goal of this post was to give locks a little respect where deserved – corrections are welcome. I also realize that locks are used in a wide variety of industries and applications, and it may not always be so easy to strike a good balance in lock performance. If you’ve found that to be the case in your own experience, I would love to hear from you in the comments.