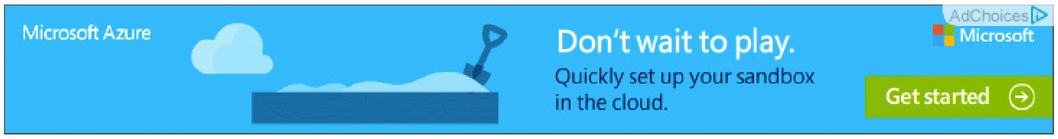
Lock-free multi-threading is for real threading experts



I was reading through an answer that Jon Skeet gave to a question and in it he mentioned this:

> As far as I'm concerned, lock-free multi-threading is for real threading experts, of which I'm not one.

Its not the first time that I have heard this, but I find very few people talking about how you actually do it if you are interested in learning how to write lock-free multi-threading code.

So my question is besides learning all you can about threading, etc where do you start trying to learn to specifically write lock-free multi-threading code and what are some good resources.

Cheers

c# .net multithreading lock-free

asked Mar 27 '10 at 10:50
vdhant
828 2 13 25

I use gcc, linux, and X86/X68 platforms. Lock-free is not nearly as hard as they all make it sound! The gcc atomic builtins have memory barriers on intel, but that doesn't matter in real life. What matters is that the memory is modified atomically. It just shakes out when you design "lock free" data structures that it doesn't matter when another thread sees a change. Single linked lists, skip lists, hash tables, free lists, etc are all pretty easy to do lock free. Lock free is not for everything. It's just another tool that is right for certain situations. – johnnycrash Jun 4 '12 at 21:54

1024cores.net – Mankarse May 29 '13 at 13:31

6 Answers

Current "lock-free" implementations follow the same pattern most of the time:

- *read some state and make a copy of it**
- *modify copy**
- do an interlocked operation
- retry if it fails

*(*optional: depends on the data structure/algorithm)*

The last bit is eerily similar to a spinlock. In fact, it is a basic spinlock. :)
I agree with @nobugz on this: the cost of the interlocked operations used in lock-free multi-threading is dominated by the cache and memory-coherency tasks it must carry out.

What you gain however with a data structure that is "lock-free" is that your "locks" are very fine grained. This decreases the chance that two concurrent threads access the same "lock" (memory location).

The trick most of the time is that you do not have dedicated locks - instead you treat e.g. all elements in an array or all nodes in a linked list as a "spin-lock". You read, modify and try to update if there was no update since your last read. If there was, you retry.
This makes your "locking" (oh, sorry, non-locking :) very fine grained, without introducing additional memory or resource requirements.
Making it more fine-grained decreases the probability of waits. Making it as fine-grained as possible without introducing additional resource requirements sounds great, doesn't it?

Most of the fun however can come from ensuring correct load/store ordering.

Contrary to one's intuitions, CPUs are free to reorder memory reads/writes - they are very smart, by the way: you will have a hard time observing this from a single thread. You will, however run into issues when you start to do multi-threading on multiple cores. Your intuitions will break down: just because an instruction is earlier in your code, it does not mean that it will actually happen earlier. CPUs can process instructions out of order: and they especially like to do this to instructions with memory accesses, to hide main memory latency and make better use of their cache.

Now, it is sure against intuition that a sequence of code does not flow "top-down", instead it runs as if there was no sequence at all - and may be called "devil's playground". I believe it is infeasible to give an exact answer as to what load/store re-orderings will take place. Instead, one always speaks in terms of *mays* and *mights* and *cans* and prepare for the worst. "Oh, the CPU *might* reorder this read to come before that write, so it is best to put a memory barrier right here, on this spot."

Matters are complicated by the fact that even these *mays* and *mights* can differ across CPU architectures. It *might* be the case, for example, that something that is *guaranteed to not happen* in one architecture *might happen* on another.

To get "lock-free" multi-threading right, you have to understand memory models.
Getting the memory model and guarantees correct is not trivial however, as demonstrated by this story, whereby Intel and AMD made some corrections to the documentation of `MFENCE` causing some stir-up among JVM developers. As it turned out, the documentation that developers relied on from the beginning was not so precise in the first place.

Locks in .NET result in an implicit memory barrier, so you are safe using them (most of the time, that is... see for example this Joe Duffy - Brad Abrams - Vance Morrison greatness on lazy initialization, locks, volatiles and memory barriers. :) (Be sure to follow the links on that page.)

As an added bonus, you will get introduced to the .NET memory model on a side quest. :)

There is also an "oldie but goldie" from Vance Morrison: What Every Dev Must Know About Multithreaded Apps.

...and of course, as @Eric mentioned, Joe Duffy is a definitive read on the subject.

A good STM can get as close to fine-grained locking as it gets and will probably provide a performance that is close to or on par with a hand-made implementation. One of them is STM.NET from the DevLabs projects of MS.

If you are not a .NET-only zealot, Doug Lea did some great work in JSR-166.
Cliff Click has an interesting take on hash tables that does not rely on lock-striping - as the Java and .NET concurrent hash tables do - and seem to scale well to 750 CPUs.

If you are not afraid to venture into Linux territory, the following article provides more insight into the internals of current memory architectures and how cache-line sharing can destroy performance: What every programmer should know about memory.

@Ben made many comments about MPI: I sincerely agree that MPI may shine in some areas. An MPI based solution can be easier to reason about, easier to implement and less error-prone than a half-baked locking implementation that tries to be smart. (It is however - subjectively - also true for an STM based solution.) I would also bet that it is light-years easier to correctly write a decent *distributed* application in e.g. Erlang, as many successful examples suggest.

MPI, however has its own costs and its own troubles when it is being run on a *single, multi-core system*. E.g. in Erlang, there are issues to be solved around the synchronization of process scheduling and message queues.
Also, at their core, MPI systems usually implement a kind of cooperative N:M scheduling for "lightweight processes". This for example means that there is an inevitable context switch between lightweight processes. It is true that it is not a "classic context switch" but mostly a user space operation and it can be made fast - however I sincerely doubt that it can be brought under the 20-200 cycles an interlocked operation takes. User-mode context switching is certainly slower even in the the Intel McRT library. N:M scheduling with light-weight processes is not new. LWPs were there in Solaris for a long time. They were abandoned. There were fibers in NT. They are mostly a relic now. There were "activations" in NetBSD. They were abandoned. Linux had its own take on the subject of N:M threading. It seems to be somewhat dead by now.
From time to time, there are new contenders: for example McRT from Intel, or most recently User-Mode Scheduling together with ConCRT from Microsoft.
At the lowest level, they do what an N:M MPI scheduler does. Erlang - or any MPI system -, might benefit greatly on SMP systems by exploiting the new UMS.

I guess the OP's question is not about the merits of and subjective arguments for/against any solution, but if I had to answer that, I guess it depends on the task: for building low level, high performance basic data structures that run on a *single system* with *many cores*, either low-lock/"lock-free" techniques or an STM will yield the best results in terms of performance and would probably beat an MPI solution any time performance-wise, even if the above wrinkles are ironed out e.g. in Erlang.
For building anything moderately more complex that runs on a single system, I would perhaps choose classic coarse-grained locking or if performance is of great concern, an STM.
For building a distributed system, an MPI system would probably make a natural choice.
Note that there are MPI implementations for .NET as well (though they seem to be not as active).

edited Jun 25 '12 at 20:22

answered Mar 27 '10 at 16:50
Andras Vass
8,276 18 37

That's a typical way of converting code that relies on locks to be "lock-free". It's not at all typical of code designed from the ground up to avoid locks, which frequently use some form of producer/consumer queues for message passing. – Ben Voigt Mar 27 '10 at 23:06

@Ben: That's an illusion, if you look deep - these systems use their own internal structures for message passing and resource scheduling. These structures use locking or the above typical "lock-free" techniques. As a good example, Erlang is built from the ground up with MPI in mind. It still uses locking at the lowest level. It turns out that it used one big fat lock for a global process queue. This created scalability problems, however. Now they plan to use multiple locks instead of one: stackoverflow.com/questions/605183/… – Andras Vass Mar 27 '10 at 23:26

@Ben: ...so in the end, someone, somewhere, sometime will have to write a low-lock/"lock-free" queue implementation for Erlang - running into the same problems as above...;) – Andras Vass Mar 27 '10 at 23:38

@andras: But only queues which have more than one writer require a lock (whether software lock, or hardware bus lock during an interlocked operation as you mentioned). Yes, there's some shared state written by multiple producers in any interesting system, but that can be made wait free (producer makes the receiver runnable, doesn't matter what the prior state was, doesn't matter whether someone else makes the receiver runnable as well). But more importantly, shared state is minimized in a system designed for message passing, so you don't pay for the cache sync very often at all. – Ben Voigt Mar 28 '10 at 0:21

I guess what I'm saying is that some of the steps are the same. But it's possible to get rid of the (make a copy) and the (retry) steps with a lock-free wait-free message queue and those are the bulk of the costs of parallelism. – Ben Voigt Mar 28 '10 at 0:24

Joe Duffy's book:

http://www.bluebytesoftware.com/books/winconc/winconc_book_resources.html

He also writes a blog on these topics.

The trick to getting low-lock programs right is to understand at a deep level *precisely* what the rules of the memory model are on your particular combination of hardware, operating system, and runtime environment.

I personally am not anywhere near smart enough to do correct low-lock programming beyond InterlockedIncrement, but if you are, great, go for it. Just make sure that you leave lots of documentation in the code so that people who are not as smart as you don't accidentally break one of your memory model invariants and introduce an impossible-to-find bug.

answered Mar 27 '10 at 15:12
Eric Lippert
325k 82 705 1553

22 So if both Eric Lippert and Jon Skeet think lock free programming is only for people smarter than themselves, then I will humbly run away screaming from the idea immediately. ;-) – dodgy_coder Apr 19 '12 at 7:06

There is no such thing as "lock-free threading" these days. It was an interesting playground for academia and the like, back in the end of the last century when computer hardware was slow and expensive. Dekker's algorithm was always my favorite, modern hardware has put it out to pasture. It doesn't work anymore.

Two developments have ended this: the growing disparity between the speed of RAM and the CPU. And the ability of chip manufacturers to put more than one CPU core on a chip.

The RAM speed problem required the chip designers to put a buffer on the CPU chip. The buffer stores code and data, quickly accessible by the CPU core. And can be read and written from/to RAM at a much slower rate. This buffer is called the CPU cache, most CPUs have at least two of them. The 1st level cache is small and fast, the 2nd is big and slower. As long as the CPU can read data and instructions from the 1st level cache it will run fast. A cache miss is really expensive, it puts the CPU to sleep for as many as 10 cycles if the data is not in the 1st cache, as many as 200 cycles if it isn't in the 2nd cache and it needs to be read from RAM.

Every CPU core has its own cache, they store their own "view" of RAM. When the CPU writes data the write is made to cache which is then, slowly, flushed to RAM. Inevitable, each core will now have a different view of the RAM contents. In other words, one CPU doesn't know what another CPU has written until that RAM write cycle completed *and* the CPU refreshes its own view.

That is dramatically incompatible with threading. You always *really* care what the state of another thread is when you must read data that was written by another thread. To ensure this, you need to explicitly program a so-called memory barrier. It is a low-level CPU primitive that ensures that all CPU caches are in a consistent state and have an up to date view of RAM. All pending writes have to flushed to RAM, the caches then need to be refreshed.

This is available in .NET, the Thread.MemoryBarrier() method implements one. Given that this is 90% of the job that the lock statement does (and 95+% of the execution time), you are simply not ahead by avoiding the tools that .NET gives you and trying to implement your own.

answered Mar 27 '10 at 15:01

Hans Passant
520k 54 582 1123

1 Maybe you could explain why you are considering "lock free threading". What do you hope to gain? If you think you can somehow make it more efficient then I failed to explain what the problem with the cpu cache is all about. – Hans Passant Mar 27 '10 at 15:59

1 @Davy8: composition makes it still hard. If I have two lock-free hash-tables and as a consumer I access both of them, this will not guarantee consistency of state as a whole. The closest you can come today is STMs where you can put the two accesses e.g. in a single `atomic` block. All in all, consuming lock-free structures can be just as tricky in many cases. – Andras Vass Mar 27 '10 at 17:04

1 I may be wrong, but I think you've mis-explained how cache coherency works. Most modern multicore processors have coherent caches, which means that the cache hardware handles making sure that all processes have the same view of RAM contents -- by blocking "read" calls until all corresponding "write" calls have completed. The Thread.MemoryBarrier() documentation (msdn.microsoft.com/en-us/library/…) says nothing about cache behavior at all -- it's simply a directive that prevents the processor from reordering reads and writes. – Brooks Moses Mar 28 '10 at 5:20

5 "There is no such thing as "lock-free threading" these days." Tell that to the Erlang and Haskell programmers. – Juliet Mar 28 '10 at 15:39

3 @HansPassant: "a so-called memory barrier. It is a low-level CPU primitive that ensures that all CPU caches are in a consistent state and have an up to date view of RAM. All pending writes have to flushed to RAM, the caches then need to be refreshed". A memory barrier in this context prevents memory instructions (loads and stores) from being reordered by the compiler or CPU. Nothing to do with the consistency of CPU caches. – Jon Harrop Jul 15 '13 at 10:54

Google for lock free data structures and software transactional memory.

I'll agree with John Skeet on this one; lock-free threading is the devil's playground, and best left up to people who know that they know what they need to know.

edited Mar 27 '10 at 13:35

answered Mar 27 '10 at 10:54

Marcelo Cantos
97.2k 13 186 249

When it comes to multi-threading you have to know exactly what you are doing. I mean explore all the possible scenarios/cases that might occur when you are working in a multithreaded environment. Lock-free multithreading is not a library or a class which we incorporate, its a knowledge/experience that we earn during our journey on threads.

answered Mar 27 '10 at 10:54
bragboy
14.5k 14 58 127

There are numerous libraries that provide lock-free threading semantics. STM is of particular interest, of which there are quite a number of implementations around. – Marcelo Cantos Mar 27 '10 at 10:56

I see both sides of this one. Getting effective performance out of a lock-free library requires deep knowledge of memory models. But a programmer who doesn't have that knowledge can still benefit from the correctness advantages. – Ben Voigt Mar 28 '10 at 16:44

Even though lock-free threading may be difficult in .NET, often you can make significant improvements when using a lock by studying exactly what needs to be locked, and minimizing the locked section... this is also known as minimizing the lock *granularity*.

As an example, just say you need to make a collection thread safe. Don't just blindly throw a lock around a method iterating over the collection if it performs some CPU-intensive task on each item. You *might* only need to put a lock around creating a shallow copy of the collection. Iterating over the copy could then work without a lock. Of course this is highly dependent on the specifics of your code, but I have been able to fix a lock convoy issue with this approach.

edited Apr 20 '12 at 4:14

answered Apr 19 '12 at 7:18
dodgy_coder
3,486 3 19 44