How can I write a lock free structure?

Asked 16 years, 3 months ago Modified 14 years, 1 month ago

Viewed 12k times

35

In my multithreaded application and I see heavy lock contention in it, preventing good scalability across multiple cores. I have decided to use lock free programming to solve this.

How can I write a lock free structure?

multithreading multicore lock-free

Share

Improve this question

Follow

edited Sep 10, 2010 at 12:52

raven

18.1k ● 17 ● 82 ● 114

asked Sep 18, 2008 at 13:18

Suma

34.3k ● 18 ● 129 ● 199

6 I think you mean a thread-safe lock free structure.
– Paul van Brenk Sep 18, 2008 at 13:19

21 Answers

Sorted by: Highest score (default)

44

Short answer is:

You cannot.

Long answer is:

If you are asking this question, you do not probably know enough to be able to create a lock free structure. Creating lock free structures is extremely hard, and only experts in this field can do it. Instead of writing your own, search for an existing implementation. When you find it, check how widely it is used, how well is it documented, if it is well proven, what are the limitations - even some lock free structure other people published are broken.

If you do not find a lock free structure corresponding to the structure you are currently using, rather adapt the algorithm so that you can use some existing one.

If you still insist on creating your own lock free structure, be sure to:

- start with something very simple
- understand memory model of your target platform (including read/write reordering constraints, what operations are atomic)
- study a lot about problems other people encountered when implementing lock free structures
- do not just guess if it will work, prove it
- heavily test the result

More reading:

[Lock free and wait free algorithms at Wikipedia]

[Herb Sutter: Lock-Free Code: A False Sense of Security]

Share Improve this answer Follow

edited Sep 18, 2008 at 14:24

answered Sep 18, 2008 at 13:36

Suma

34.3k • 18 • 129 • 199

1 Exactly what I wanted to write :) – gabr Sep 18, 2008 at 14:25

12 I am asking them to help other people who might be searching for the answer here. – Suma Sep 18, 2008 at 19:15

1 For a ROBUST sudo code example see the following paper research.ibm.com/people/m/michael/podc-1996.pdf This implements a linked list of elements allowing multiple concurrent accesses without the use of locks. – Howard May Feb 4, 2009 at 11:14

(@HowardMay) A current link to the mentioned M. Michael paper about the topic: cs.rochester.edu/~scott/papers/1996_PODC_queues.pdf – jjmontes Jan 12 at 20:49

Use a library such as Intel's Threading Building Blocks, it contains quite a few lock -free structures and algorithms. I

16

really wouldn't recommend attempting to write lock-free code yourself, it's extremely error prone and hard to get right.

Share Improve this answer

Follow

answered Sep 18, 2008 at 13:25

Niall

5,121 ● 1 ● 22 ● 12

12

Writing thread-safe lock free code is hard; but this article from Herb Sutter will get you started.

Share Improve this answer

Follow

answered Sep 18, 2008 at 13:23

Paul van Brenk

7,540 ● 2 ● 35 ● 39

12

As *sblundy* pointed out, if all objects are immutable, read-only, you don't need to worry about locking, however, this means you may have to copy objects a lot. Copying usually involves malloc and malloc uses locking to synchronize memory allocations across threads, so immutable objects may buy you less than you think (malloc itself scales rather badly and malloc is **slow**; if you do a lot of malloc in a performance critical section, don't expect good performance).

When you only need to update simple variables (e.g. 32 or 64 bit int or pointers), perform simply addition or

subtraction operations on them or just swap the values of two variables, most platforms offer "atomic operations" for that (further GCC offers these as well). *Atomic is not the same as thread-safe*. However, atomic makes sure, that if one thread writes a 64 bit value to a memory location for example and another thread reads from it, the reading one either gets the value before the write operation or after the write operation, but never a *broken* value in-between the write operation (e.g. one where the first 32 bit are already the new, the last 32 bit are still the old value! This can happen if you don't use atomic access on such a variable).

However, if you have a C struct with 3 values, that want to update, even if you update all three with atomic operations, these are three independent operations, thus a reader might see the struct with one value already being update and two not being updated. Here you will need a lock if you must assure, the reader either sees all values in the struct being either the old or the new values.

One way to make locks scale a lot better is using R/W locks. In many cases, updates to data are rather infrequent (write operations), but accessing the data is very frequent (reading the data), think of collections (hashtables, trees). In that case R/W locks will buy you a huge performance gain, as many threads can hold a read-lock at the same time (they won't block each other) and only if one thread wants a write lock, all other threads are blocked for the time the update is performed.

The best way to avoid thread-issues is to not share any data across threads. If every thread deals most of the time with data no other thread has access to, you won't need locking for that data at all (also no atomic operations). So try to share as little data as possible between threads. Then you only need a fast way to move data between threads if you really have to (ITC, Inter Thread Communication). Depending on your operating system, platform and programming language (unfortunately you told us neither of these), various powerful methods for ITC might exist.

And finally, another trick to work with shared data but without any locking is to make sure threads don't access the same parts of the shared data. E.g. if two threads share an array, but one will only ever access even, the other one only odd indexes, you need no locking. Or if both share the same memory block and one only uses the upper half of it, the other one only the lower one, you need no locking. Though it's not said, that this will lead to good performance; especially not on multi-core CPUs. Write operations of one thread to this shared data (running one core) might force the cache to be flushed for another thread (running on another core) and these cache flushes are often the bottle neck for multithread applications running on modern multi-core CPUs.

answered Sep 18, 2008 at 13:42

Mecki

133k ● 34 ● 259 ● 269

"Here you will need a lock if you must assure"... No - you mutate a new copy of the structure instead of doing it in place, and switch which one is active as your atomic operation. – moonshadow Sep 18, 2008 at 14:30

But that means you will have to malloc again, assuming that this is not stack data (which it most likely won't be) and like I said, malloc can be a huge bottle neck. In one of our software, reusing the same memory block each time compared to using malloc each time caused a speed gain of 80%. – Mecki Sep 18, 2008 at 20:10

You could have changed to using a thread optimized malloc instead, one that uses a per-thread arena. – Zan Lynx Nov 11, 2010 at 20:18

10

As my professor (Nir Shavit from "The Art of Multiprocessor Programming") told the class: Please don't. The main reason is testability - you can't test synchronization code. You can run simulations, you can even stress test. But it's rough approximation at best. What you really need is mathematical correctness proof. And very few capable understanding them, let alone writing them. So, as others had said: use existing libraries. Joe Duffy's blog surveys some techniques (section 28). The first one you should try is tree-splitting - break to smaller tasks and combine.

Share Improve this answer

Follow

answered Apr 9, 2009 at 8:47

felixg

974 ● 1 ● 9 ● 10

7

Immutability is one approach to avoid locking. See Eric Lippert's discussion and implementation of things like immutable stacks and queues.

Share Improve this answer

Follow

answered Sep 18, 2008 at 13:24

Jeff Moser

20k ● 6 ● 66 ● 85

6

in re. Suma's answer, Maurice Herlithy shows in The Art of Multiprocessor Programming that actually *anything* can be written without locks (see chapter 6). iirc, This essentially involves splitting tasks into processing node elements (like a function closure), and enqueuing each one. Threads will calculate the state by following all nodes from the latest cached one. Obviously this could, in worst case, result in sequential performance, but it does have important lockless properties, preventing scenarios where threads could get scheduled out for long peroids of time when they are holding locks. Herlithy also achieves theoretical wait-free performance, meaning that one thread will not end up waiting forever to win the atomic enqueue (this is a lot of complicated code).

A multi-threaded queue / stack is surprisingly hard (check the ABA problem). Other things may be very simple. Become accustomed to while(true) { atomicCAS until I swapped it } blocks; they are incredibly powerful. An intuition for what's correct with CAS can help

development, though you should use good testing and maybe more powerful tools (maybe SKETCH, upcoming MIT Kendo, or spin?) to check correctness if you can reduce it to a simple structure.

Please post more about your problem. It's difficult to give a good answer without details.

edit immutibility is nice but it's applicability is limited, if I'm understanding it right. It doesn't really overcome write-after-read hazards; consider two threads executing "mem = NewNode(mem)"; they could both read mem, then both write it; not the correct for a classic increment function. Also, it's probably slow due to heap allocation (which has to be synchronized across threads).

Share Improve this answer Follow

edited Apr 9, 2009 at 8:02

answered Apr 9, 2009 at 7:50

gatoatigrado

16.8k ● 19 ● 84 ● 145

5

Inmutability would have this effect. Changes to the object result in a new object. Lisp works this way under the covers.

Item 13 of Effective Java explains this technique.

answered Sep 18, 2008 at 13:21

Share Improve this answer

Follow

sblundy

61.4k ● 22 ● 123 ● 124

4

Cliff Click has dome some major research on lock free data structures by utilizing finite state machines and also posted a lot of implementations for Java. You can find his papers, slides and implementations at his blog: http://blogs.azulsystems.com/cliff/

Share Improve this answer

Follow

answered Sep 18, 2008 at 13:37

dsp

43 ● 3

2

Use an existing implementation, as this area of work is the realm of domain experts and PhDs (if you want it done right!)

For example there is a library of code here:

http://www.cl.cam.ac.uk/research/srg/netos/lock-free/

Share Improve this answer

Follow

answered Sep 18, 2008 at 15:30

JeeBee

17.5k ● 5 ● 52 ● 60

1

Most lock-free algorithms or structures start with some atomic operation, i.e. a change to some memory location that once begun by a thread will be completed before any

other thread can perform that same operation. Do you have such an operation in your environment?

See here for the canonical paper on this subject.

Also try this wikipedia article article for further ideas and links.

Share Improve this answer Follow

answered Sep 18, 2008 at 13:32

community wiki
Justsalt

This "atomic operation" sounds suspiciously like a lock. What's the difference? – cHao May 1, 2013 at 14:45

1

The basic principle for lock-free synchronisation is this:

- whenever you are reading the structure, you follow the read with a test to see if the structure was mutated since you started the read, and retry until you succeed in reading without something else coming along and mutating while you are doing so;
- whenever you are mutating the structure, you arrange your algorithm and data so that there is a single atomic step which, if taken, causes the entire change to become visible to the other threads, and arrange things so that none of the change is visible unless that step is taken. You use whatever lockfree

atomic mechanism exists on your platform for that step (e.g. compare-and-set, load-linked+store-conditional, etc.). In that step you must then check to see if any other thread has mutated the object since the mutation operation began, commit if it has not and start over if it has.

There are plenty of examples of lock-free structures on the web; without knowing more about what you are implementing and on what platform it is hard to be more specific.

Share Improve this answer

Follow

edited Sep 18, 2008 at 13:35

answered Sep 18, 2008 at 13:28

moonshadow

88.9k ● 7 ● 86 ● 121

1

If you are writing your own lock-free data structures for a multi-core cpu, do not forget about memory barriers! Also, consider looking into Software Transaction Memory techniques.

Share Improve this answer

Follow

answered Sep 18, 2008 at 15:24

jdkoftinoff

2,421 ● 1 ● 18 ● 18

0

Well, it depends on the kind of structure, but you have to make the structure so that it carefully and silently detects and handles possible conflicts.

I doubt you can make one that is 100% lock-free, but again, it depends on what kind of structure you need to build.

You might also need to shard the structure so that multiple threads work on individual items, and then later on synchronize/recombine.

Share Improve this answer

Follow

answered Sep 18, 2008 at 13:22

Lasse V. Karlsen

391k ● 106 ● 646 ● 844

0

As mentioned, it really depends on what type of structure you're talking about. For instance, you can write a limited lock-free queue, but not one that allows random access.

Share Improve this answer

Follow

answered Sep 18, 2008 at 13:24

Oliver Mellet

2,387 ● 12 ● 10

0

Reduce or eliminate shared mutable state.

Share Improve this answer

Follow

answered Sep 18, 2008 at 13:26

Craig Day
2,535 ● 1 ● 24 ● 26

0

In Java, utilize the java.util.concurrent packages in JDK 5+ instead of writing your own. As was mentioned above, this is really a field for experts, and unless you have a spare year or two, rolling your own isn't an option.

Share Improve this answer

Follow

answered Sep 18, 2008 at 14:42

Munger
43 ● 1

0

Can you clarify what you mean by structure?

Right now, I am assuming you mean the overall architecture. You can accomplish it by not sharing memory between processes, and by using an actor model for your processes.

Share Improve this answer

Follow

answered Sep 18, 2008 at 16:06

Dan
512 ● 1 ● 4 ● 12

0

Take a look at my link ConcurrentLinkedHashMap for an example of how to write a lock-free data structure. It is not based on any academic papers and doesn't require

years of research as others imply. It simply takes careful engineering.

My implementation does use a ConcurrentHashMap, which is a lock-per-bucket algorithm, but it does not rely on that implementation detail. It could easily be replaced with Cliff Click's lock-free implementation. I borrowed an idea from Cliff, but used much more explicitly, is to model all CAS operations with a state machine. This greatly simplifies the model, as you'll see that I have psuedo locks via the 'ing states. Another trick is to allow laziness and resolve as needed. You'll see this often with backtracking or letting other threads "help" to cleanup. In my case, I decided to allow dead nodes on the list be evicted when they reach the head, rather than deal with the complexity of removing them from the middle of the list. I may change that, but I didn't entirely trust my backtracking algorithm and wanted to put off a major change like adopting a 3-node locking approach.

The book "The Art of Multiprocessor Programming" is a great primer. Overall, though, I'd recommend avoiding lock-free designs in the application code. Often times it is simply overkill where other, less error prone, techniques are more suitable.

answered Sep 20, 2008 at 2:26

Ben Manes

9,561 ● 3 ● 38 ● 40

At the "concurrentlinkedhashmap" there is an interesting comment written now: Note: A rare race condition was

uncovered by Greg Luck (Ehcache). This algorithm is deprecated. I guess this shows what to expect when developing lock free data on your own. – Suma Sep 23, 2009 at 17:20

That comment has been there for ages. The comment that the project was for personal educational purposes for understanding concurrent algorithms has been there since pretty much the beginning. You try to use lock freedom for your own personal growth and you try to avoid it for production. That's pretty much what I said in my original post. – Ben Manes Sep 26, 2009 at 20:15

0

If you see lock contention, I would first try to use more granular locks on your data structures rather than completely lock-free algorithms.

For example, I currently work on multithreaded application, that has a custom messaging system (list of queues for each threads, the queue contains messages for thread to process) to pass information between threads. There is a global lock on this structure. In my case, I don't need speed so much, so it doesn't really matter. But if this lock would become a problem, it could be replaced by individual locks at each queue, for example. Then adding/removing element to/from the specific queue would didn't affect other queues. There still would be a global lock for adding new queue and such, but it wouldn't be so much contended.

Even a single multi-produces/consumer queue can be written with granular locking on each element, instead of

having a global lock. This may also eliminate contention.

Share Improve this answer Follow

answered Apr 9, 2009 at 8:02

J S

1,107 ● 7 ● 6

0

If you read several implementations and papers regarding the subject, you'll notice there is the following common theme:

1) **Shared state objects are lisp/clojure style inmutable**: that is, all write operations are implemented copying the existing state in a new object, make modifications to the new object and then try to update the shared state (obtained from a aligned pointer that can be updated with the CAS primitive). In other words, you NEVER EVER modify an existing object that might be read by more than the current thread. Inmutability can be optimized using Copy-on-Write semantics for big, complex objects, but thats another tree of nuts

2) **you clearly specify what allowed transitions between current and next state are valid**: Then validating that the algorithm is valid become orders of magnitude easier

3) **Handle discarded references in hazard pointer lists per thread**. After the reference objects are safe, reuse if possible

See another related post of mine where some code implemented with semaphores and mutexes is (partially) reimplemented in a lock-free style: [Mutual exclusion and semaphores]

Share Improve this answer

Follow

edited May 23, 2017 at 12:00

Community Bot

1 • 1

answered Nov 11, 2010 at 18:38

lurscher

26.9k • 30 • 129 • 191