# Preshing on Programming

- Twitter
- RSS

Navigate...

- Blog
- Archives
- About

Jun 12, 2012

## An Introduction to Lock-Free Programming

Lock-free programming is a challenge, not just because of the complexity of the task itself, but because of how difficult it can be to penetrate the subject in the first place.

I was fortunate in that my first introduction to lock-free (also known as lockless) programming was Bruce Dawson's excellent and comprehensive white paper, Lockless Programming Considerations. And like many, I've had the occasion to put Bruce's advice into practice developing and debugging lock-free code on platforms such as the Xbox 360.

Since then, a lot of good material has been written, ranging from abstract theory and proofs of correctness to practical examples and hardware details. I'll leave a list of references in the footnotes. At times, the information in one source may appear orthogonal to other sources: For instance, some material assumes sequential consistency, and thus sidesteps the memory ordering issues which typically plague lock-free C/C++ code. The new C++11 atomic library standard throws another wrench into the works, challenging the way many of us express lock-free algorithms.

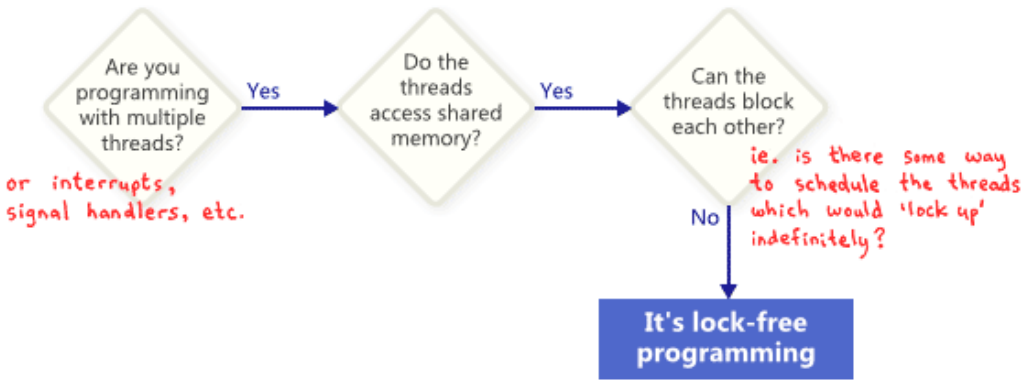
In this post, I'd like to re-introduce lock-free programming, first by defining it, then by distilling most of the information down to a few key concepts. I'll show how those concepts relate to one another using flowcharts, then we'll dip our toes into the details a little bit. At a minimum, any programmer who dives into lock-free programming should already understand how to write correct multithreaded code using mutexes, and other high-level synchronization objects such as semaphores and events.

## What Is It?

People often describe lock-free programming as programming without mutexes,

which are also referred to as locks. That's true, but it's only part of the story. The generally accepted definition, based on academic literature, is a bit more broad. At its essence, lock-free is a property used to describe some code, without saying too much about how that code was actually written.

Basically, if some part of your program satisfies the following conditions, then that part can rightfully be considered lock-free. Conversely, if a given part of your code doesn't satisfy these conditions, then that part is not lock-free.

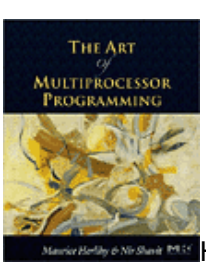



In this sense, the *lock* in lock-free does not refer directly to mutexes, but rather to the possibility of "locking up" the entire application in some way, whether it's deadlock, livelock – or even due to hypothetical thread scheduling decisions made by your worst enemy. That last point sounds funny, but it's key. Shared mutexes are ruled out trivially, because as soon as one thread obtains the mutex, your worst enemy could simply never schedule that thread again. Of course, real operating systems don't work that way – we're merely defining terms.

Here's a simple example of an operation which contains no mutexes, but is still not lock-free. Initially, X = 0. As an exercise for the reader, consider how two threads could be scheduled in a way such that neither thread exits the loop.

```
while (X == 0)
{
    X = 1 - X;
}
```

Nobody expects a large application to be entirely lock-free. Typically, we identify a specific set of lock-free operations out of the whole codebase. For example, in a lock-free queue, there might be a handful of lock-free operations such as `push`, `pop`, perhaps `isEmpty`, and so on.




Herlihy & Shavit, authors of The Art of Multiprocessor Programming, tend to express such operations as class methods, and offer the following succinct definition of lock-free (see slide 150): "In an infinite execution, infinitely often some method call finishes." In other words, as long as

the program is able to keep *calling* those lock-free operations, the number of *completed* calls keeps increasing, no matter what. It is algorithmically impossible for the system to lock up during those operations.

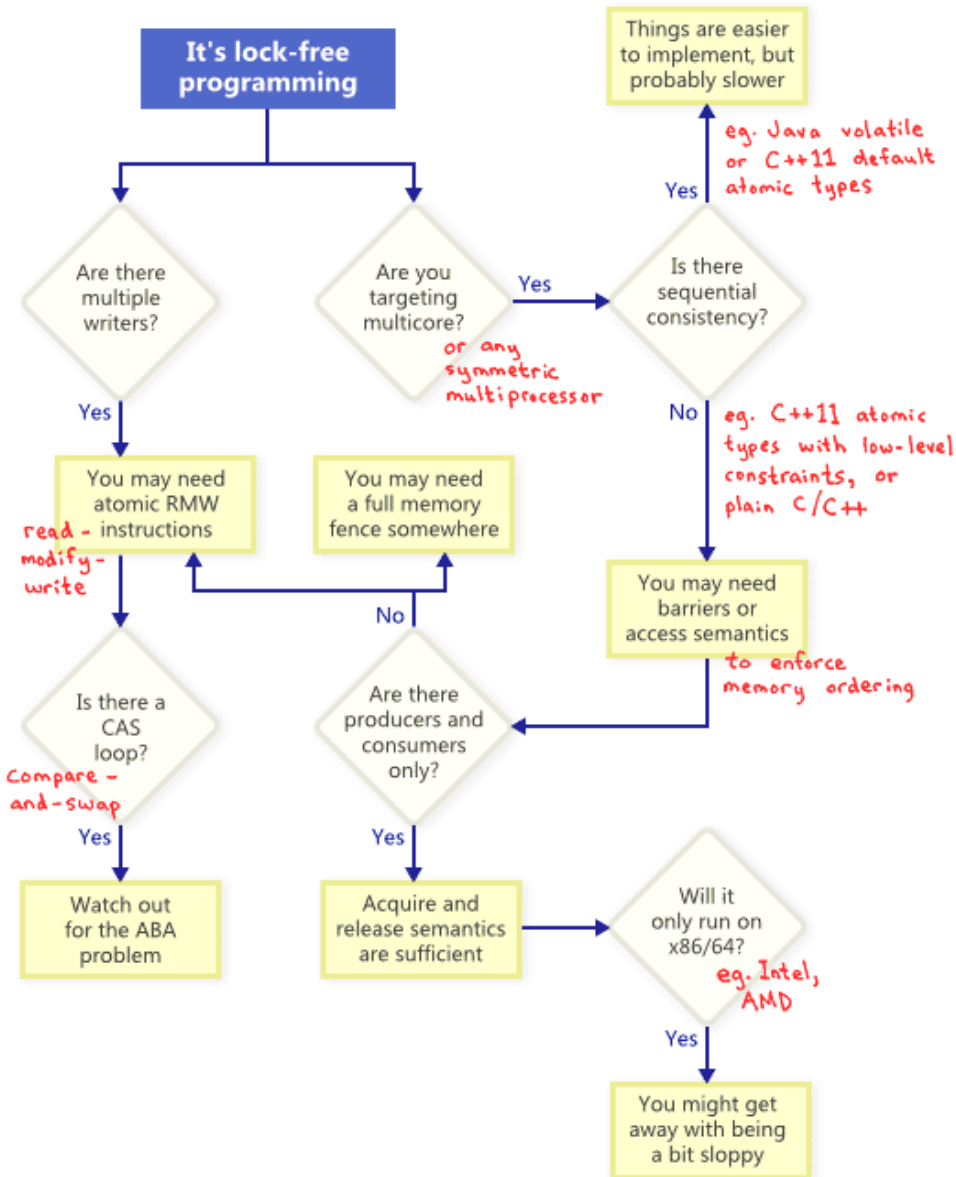
One important consequence of lock-free programming is that if you suspend a single thread, it will never prevent other threads from making progress, as a group, through their own lock-free operations. This hints at the value of lock-free programming when writing interrupt handlers and real-time systems, where certain tasks must complete within a certain time limit, no matter what state the rest of the program is in.

A final precision: Operations that are *designed* to block do not disqualify the algorithm. For example, a queue's pop operation may intentionally block when the queue is empty. The remaining codepaths can still be considered lock-free.

## Lock-Free Programming Techniques

It turns out that when you attempt to satisfy the non-blocking condition of lock-free programming, a whole family of techniques fall out: atomic operations, memory barriers, avoiding the ABA problem, to name a few. This is where things quickly become diabolical.

So how do these techniques relate to one another? To illustrate, I've put together the following flowchart. I'll elaborate on each one below.




## Atomic Read-Modify-Write Operations

Atomic operations are ones which manipulate memory in a way that appears indivisible: No thread can observe the operation half-complete. On modern processors, lots of operations are already atomic. For example, aligned reads and writes of simple types are usually atomic.


Read-modify-write (RMW) operations go a step further, allowing you to perform more complex transactions atomically. They're especially useful when a lock-free algorithm must support multiple writers, because when multiple threads attempt an RMW on the same address, they'll effectively line up in a row and execute those operations one-at-a-time. I've

already touched upon RMW operations in this blog, such as when implementing a lightweight mutex, a recursive mutex and a lightweight logging system.

Examples of RMW operations include `_InterlockedIncrement` on Win32, `OSAtomicAdd32` on iOS, and `std::atomic<int>::fetch_add` in C++11. Be aware that the C++11 atomic standard does not guarantee that the implementation will be lock-free on every platform, so it's best to know the capabilities of your platform and toolchain. You can call `std::atomic<>::is_lock_free` to make sure.

Different CPU families support RMW in different ways. Processors such as PowerPC and ARM expose load-link/store-conditional instructions, which effectively allow you to implement your own RMW primitive at a low level, though this is not often done. The common RMW operations are usually sufficient.

As illustrated by the flowchart, atomic RMWs are a necessary part of lock-free programming even on single-processor systems. Without atomicity, a thread could be interrupted halfway through the transaction, possibly leading to an inconsistent state.

## Compare-And-Swap Loops

Perhaps the most often-discussed RMW operation is compare-and-swap (CAS). On Win32, CAS is provided via a family of intrinsics such as `_InterlockedCompareExchange`. Often, programmers perform compare-and-swap in a loop to repeatedly attempt a transaction. This pattern typically involves copying a shared variable to a local variable, performing some speculative work, and attempting to publish the changes using CAS:

```
void LockFreeQueue::push(Node* newHead)
{
    for (;;)
    {
        // Copy a shared variable (m_Head) to a local.
        Node* oldHead = m_Head;

        // Do some speculative work, not yet visible to other threads.
        newHead->next = oldHead;

        // Next, attempt to publish our changes to the shared variable.
        // If the shared variable hasn't changed, the CAS succeeds and we return.
        // Otherwise, repeat.
        if (_InterlockedCompareExchange(&m_Head, newHead, oldHead) == oldHead)
            return;
    }
}
```

Such loops still qualify as lock-free, because if the test fails for one thread, it means it must have succeeded for another – though some architectures offer a weaker variant of CAS where that's not necessarily true. Whenever implementing a CAS loop, special care must be taken to avoid the ABA problem.

## Sequential Consistency

Sequential consistency means that all threads agree on the order in which memory operations occurred, and that order is consistent with the order of operations in the program source code. Under sequential consistency, it's

impossible to experience memory reordering shenanigans like the one I demonstrated in a previous post.

A simple (but obviously impractical) way to achieve sequential consistency is to disable compiler optimizations and force all your threads to run on a single processor. A processor never sees its own memory effects out of order, even when threads are pre-empted and scheduled at arbitrary times.

Some programming languages offer sequentially consistency even for optimized code running in a multiprocessor environment. In C++11, you can declare all shared variables as C++11 atomic types with default memory ordering constraints. In Java, you can mark all shared variables as `volatile`. Here's the example from my previous post, rewritten in C++11 style:

```
std::atomic<int> X(0), Y(0);
int r1, r2;

void thread1()
{
    X.store(1);
    r1 = Y.load();
}

void thread2()
{
    Y.store(1);
    r2 = X.load();
}
```

Because the C++11 atomic types guarantee sequential consistency, the outcome r1 = r2 = 0 is impossible. To achieve this, the compiler outputs additional instructions behind the scenes – typically memory fences and/or RMW operations. Those additional instructions may make the implementation less efficient compared to one where the programmer has dealt with memory ordering directly.

## Memory Ordering

As the flowchart suggests, any time you do lock-free programming for multicore (or any symmetric multiprocessor), and your environment does not guarantee sequential consistency, you must consider how to prevent memory reordering.

On today's architectures, the tools to enforce correct memory ordering generally fall into three categories, which prevent both compiler reordering and processor reordering:

- A lightweight sync or fence instruction, which I'll talk about in future posts;
- A full memory fence instruction, which I've demonstrated previously;
- Memory operations which provide acquire or release semantics.

Acquire semantics prevent memory reordering of operations which follow it in program order, and release semantics prevent memory reordering of operations preceding it. These semantics are particularly suitable in cases when there's a producer/consumer relationship, where one thread publishes some information and the other reads it. I'll also talk about this more in a future post.

### Different Processors Have Different Memory Models

Different CPU families have different habits when it comes to memory reordering. The rules are documented by each CPU vendor and followed strictly by the hardware. For instance, PowerPC and ARM processors can change the order of memory stores relative to the instructions themselves, but normally, the x86/64 family of processors from Intel and AMD do not. We say the former processors have a more relaxed memory model.

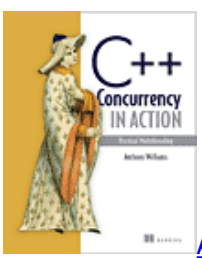
There's a temptation to abstract away such platform-specific details, especially with C++11 offering us a standard way to write portable lock-free code. But currently, I think most lock-free programmers have at least some appreciation of platform differences. If there's one key difference to remember, it's that at the x86/64 instruction level, every load from memory comes with acquire semantics, and every store to memory provides release semantics – at least for non-SSE instructions and non-write-combined memory. As a result, it's been common in the past to write lock-free code which works on x86/64, but fails on other processors.

If you're interested in the hardware details of how and why processors perform memory reordering, I'd recommend Appendix C of Is Parallel Programming Hard. In any case, keep in mind that memory reordering can also occur due to compiler reordering of instructions.

In this post, I haven't said much about the practical side of lock-free programming, such as: When do we do it? How much do we really need? I also haven't mentioned the importance of validating your lock-free algorithms. Nonetheless, I hope for some readers, this introduction has provided a basic familiarity with lock-free concepts, so you can proceed into the additional reading without feeling too bewildered. As usual, if you spot any inaccuracies, let me know in the comments.

*[This article was featured in Issue #29 of Hacker Monthly.]*

## Additional References




- Anthony Williams' blog and his book, C++ Concurrency in Action
- Dmitriy V'jukov's website and various forum discussions
- Bartosz Milewski's blog
- Charles Bloom's Low-Level Threading series on his blog
- Doug Lea's JSR-133 Cookbook
- Howells and McKenney's memory-barriers.txt document
- Hans Boehm's collection of links about the C++11 memory model

- Herb Sutter's Effective Concurrency series

29 242
Like Tweet G+1

« Lightweight In-Memory Logging Memory Ordering at Compile Time »

## Comments (26)

 Logged in as bgn9000 Dashboard | Edit profile | Logout

 marris · *188 weeks ago*

Fantastic article!

The flow charts are very useful. I'm going to save copies.

And the text explanations are top-notch!

Reply

 pikachu · *188 weeks ago*

Ever heard of CCommunicating Sequential Processes or Go programming language?

Reply

 extralongpants · *188 weeks ago*

Thank you for taking the time to write this article. I found it informative and well written. The flow charts were a good idea. :)

Reply

 Wix · *188 weeks ago*

Excellent post! It's like an chapter of well written book. I can't wait to see future articles and in the mean time I'm going to read all the past ones.

Reply **2 replies** · *active 100 weeks ago*

 Jeff Preshing · *188 weeks ago*

Thanks you guys. Glad people liked this one. I found it tough to write.

Reply

 Mairaaj · *100 weeks ago*

Most of the times Well written things are tough to write. Jazak Allah (may God reward you best)

Reply

 Ian · *188 weeks ago*

A lock-free queue should never have a member called isEmpty(). It's a misleading name, since that condition can be fleeting and not relied on. A better name is wasEmpty(), which is a gentle reminder of this fact.

Reply

 William La Forge · *188 weeks ago*

After working on the JActor project, this article feels so wrong-headed. Once you have ultra-high performance actors, it becomes clear that the problem all along was the focus on sharing state across threads--which you say is a prerequisite for lock-free programming. We just didn't have a workable alternative.

Reply **4 replies** · *active 122 weeks ago*

 Jeff Preshing · *188 weeks ago*

Your JActor project uses `java.util.concurrent.ConcurrentLinkedQueue`. Thus, it too incorporates lock-free programming.

Reply

 William La Forge · *188 weeks ago*

You are quite correct. There is also one semaphore in the implementation of the thread pool, but then it is important to block idle threads, eh? So applying your own reasoning and definitions, there is no such thing as a lock-free program.

The real attention should be directed to the application logic. Once you have really fast actors to build on, in my experience, sharing state between threads is not an appropriate focus. Yeah, there are times when it is appropriate, but those are the exceptions.

Reply

 Jeff Preshing · *188 weeks ago*

I wholeheartedly agree that sharing state between threads should not be encouraged! The same advice is given in Bruce Dawson's paper and elsewhere. Thanks for your comments.

Reply

 William La Forge · *168 weeks ago*

I've actually come of late to a better understanding of a significant difference between JActor actors-- http://jactorconsulting.com/product/jactor/ --and Scala actors, and that has to do with the elimination of intermediate or process state.

In JActor you define a callback (usually an anonymous class) which is invoked on the requesting actor's "thread" when a response message is received. Member variables in this callback are the equivalent to method variables as they are not accessible when processing other incoming requests.

Contrast this to Scala/Akka actors which use 1-way messages and which must use shared state for intermediate data and consequently must delay processing of other requests least this intermediate data be overwritten.

With JActor then, actors do not have state that is shared across multiple requests except when that state is updated atomically. And I think that is key.

Reply

 Greg Benison · *188 weeks ago*

Nicely written overview of lock-free programming, especially in that it gives a sense of how tricky it really is (especially if cross-platform compatibility is needed.)

Given how many pitfalls there are with lock-free approaches, would you recommend implementing something with plain old mutexes first, and then benchmarking against that? Lock-free data structures seem like a great avenue for premature optimization.

Reply **1 reply** · *active 122 weeks ago*

 Jeff Preshing · *188 weeks ago*

Yea for sure. I even gave respect to locks (mutexes) in a previous post.

Where I work, nobody has gone overboard with lock-free programming. We've got some base lock-free systems in place. Aside from that, it usually happens as you suggest. When some part of the game is too slow, we profile it. Once in a while, a locking section shows up as a bottleneck. We first try to reduce data sharing, but when that's not possible, we might apply a lock-free technique. The latter choice usually requires careful testing.

Reply

 Mica · *168 weeks ago*

Hi Jeff,

Thx for these useful information.

Are you courageous enough to go a step further and teach us how to program "wait-free" algorithms?

No one seems to know how to ;-)

Regards
Mica

Reply

 adrien · *167 weeks ago*

Thank you for this great article.

I've read several articles on DrDobbs and stuff like that and the doc of c++11, but this one is like a great very clear overview of the all thing well written, with very nice drawings.

Keep up the great work

Reply

 code2live · *165 weeks ago*

Thanks for the very useful article/links.

I had a question regarding the flowchart - "Are there producers and consumers only? Yes -> Acquire and release semantics are sufficient" - Where can I find more information about this - especially involving multiple producers/consumers? Maybe its already there in a particular section of the listed links? Would be great if you have more information

Reply

 drewlu · *163 weeks ago*

Wonderful article especially for beginners!

Reply

 Ali · *130 weeks ago*

Hi...Thanks for this useful article.I am doing FYP on lock free synchronization.I have studied a lot of articles and papers but I could not understand the basic concept that how a single resource cab be utilized by two processors simultaneously..No one have given pictorial /graphical examples.
I need information about Lock free Link Lists.What is the flow of making them lockless...
Kindly help me..

Reply **1 reply** · *active 122 weeks ago*

 Bill La Forge · *130 weeks ago*

Any change should be made to a (deep?) copy of the data structure and, when updating the reference, if it was changed since the copy was made (see java atomic reference or just use a CAS instruction), just repeat the process.

This is a heavy handed approach, but will work for any type of structure. But there are many structures (java's concurrent linked queue for example) that likely work more efficiently, i.e. without having to copy the entire structure. Alternately, you can use an immutable structure which returns a different reference with each update as the immutable structure will cleverly avoid having to copy the entire structure.

Reply

 Rohan · *97 weeks ago*

I had a question. Intel allows reordering of loads with earlier stores, ie: StoreLoad can be reordered to LoadStore. However for that very reordering there is no fence - ie no StoreLoad Barrier. One must use a full barrier - MFENCE. Did I get that right?

Reply **1 reply** · *active 97 weeks ago*

 Jeff Preshing · *97 weeks ago*

You got that right, though "mfence" is not the only instruction which can act as a full barrier. Any locked instruction, such as "xchg", "lock or", and others, will also act as a full memory barrier – provided you don't use SSE instructions or write-combined memory in the neighboring operations you wish to affect. Microsoft C++ generates "xchg" when you use the MemoryBarrier intrinsic, at least in Visual Studio 2008. Mintomic implements mint_thread_fence_seq_cst using "lock or".

Reply

 Rohan · *97 weeks ago*

Thanks for the reply! As I read further into the Intel Manuals I realized that "StoreLoads may be reordered for different mem locations" but "StoreLoads are NOT reordered for same mem location". So in-fact one doesn't need a StoreLoad Barrier at all on x86 if they point to same memory location, and I really cant think of an example where a StoreLoad barrier would be needed if they point to different memory locations. At that point there is no option other than a full barrier.

Reply

Rollen · *92 weeks ago*

Do the parameters of _InterlockedCompareExchange be misplaced?

I mean, should the following piece of code
if (_InterlockedCompareExchange(&m_Head, newHead, oldHead) == oldHead)
return;

be modified as

if (_InterlockedCompareExchange(&m_Head, oldHead, newHead) == oldHead)
return;

Thanks.

Reply

@avushakov · *77 weeks ago*

Hans Boehm's collection of links about the C++11 memory model moved to
http://hboehm.info/c++mm/

Reply

Jaseem Abid · *5 weeks ago*

Very well written. Thank you. Will bookmark and come back when needed.

Reply

## Post a new comment

Enter text right here!

Posting as bgn9000 (Logout)

Subscribe to Replies

Submit Comment

## Recent Posts

- You Can Do Any Kind of Atomic Read-Modify-Write Operation
- Safe Bitfields in C++
- Semaphores are Surprisingly Versatile
- C++ Has Become More Pythonic
- Fixing GCC's Implementation of memory_order_consume
- How to Build a GCC Cross-Compiler
- How to Install the Latest GCC on Windows
- My Multicore Talk at CppCon 2014
- The Purpose of memory_order_consume in C++11
- What Is a Bitcoin, Really?
- Bitcoin Address Generator in Obfuscated Python
- Acquire and Release Fences Don't Work the Way You'd Expect
- Double-Checked Locking is Fixed In C++11
- Acquire and Release Fences
- The Synchronizes-With Relation
- The Happens-Before Relation
- Atomic vs. Non-Atomic Operations
- The World's Simplest Lock-Free Hash Table
- A Lock-Free... Linear Search?
- Introducing Mintomic: A Small, Portable Lock-Free API

### Tip Jar

If you like this blog, leave a tip!


Copyright © 2015 Jeff Preshing - Powered by Octopress