The Basics of Lockfree Programming - Notes
Tony Van Eerd
BoostCon 2010

Forewarned is Forearmed

[Slide 1] ``Premature optimization is the root of all evil." -- Knuth/Hoare/Dijkstra [Knuth put it to paper, but claims he got it from Hoare who claims it may have come from Dijkstra.]

[Slide 2] "What 99 percent of programmers need to know is not how to build components but how to use them." -- Alexander Stepanov [via Foster T. Brereton @ Adobe]

So why are you here? You shouldn't need to learn about lock-free programming - leave it to a few specialists and just use their components (via boost maybe?). And 'going lock-free' is definitely an optimization – one that is almost guaranteed to make you code more error-prone (and with harder to find errors), less readable, less maintainable. **By a large factor**. ie lock-free is evil.

Furthermore, I'm an inexperienced speaker. The other talk is probably better. Go.

No? OK more quotes.

[Slide 3]
"Lock-free programming is hard."
...
"Lock-free programming is not that hard."
...
"Lock-free programming is hard."

This is the typical progression of quotes for anyone learning lock-free programming.

And lastly, please note that lock-free programming is not just some techniques that you leave in your 'tool chest' of programming techniques (like, say, the visitor pattern) that you just pull out on the odd occasion when the need arises. Why not? Because it is hard. If you only practise it infrequently, the odds of making an error are increased. You need to dedicate sufficient time to it, and often, each application of lock-free techniques comes with its own unique problems. (ie it is often not a 'pattern' at all, to be re-applied the same way each time.)

[Slide for added emphasis] "Use Locks!" - me, etc

And that concludes the talk. :-)

...So why am I even going on about lock-free programming? Because everyone keeps asking about it. (see comp.programming.threads, boost lists, stackoverflow.com, etc). Typically the questions are full of misconceptions...
... and so are the answers.
So I think we need to start clearing the minefields.

I Am Not An Expert.

I am not on the bleeding edge of lock-free programming. But I've done my fair share over the years (stacks, queues, allocators, etc). I'd say I can at least see the edge from where I stand, I just don't want to stand too close to it.

Not being an expert is an advantage for you. I'm hoping from my vantage-point, I can give you a view of lock-free programming that both makes sense to the novice, and offers a conceptual model that will allow you to make the right decisions when programming. A model without misconceptions. A model that (hopefully) won't steer you wrong.

Are We There Yet? Are We There Now? What About Now?

What's wrong with this code?:

```
if (b != 0)
{
   x = a / b;
}
else ...
```

If doing a code-review, seeing 'a / b' might make you check to make sure the programmer is handling divide by zero, and it appears that she is (leaving arguments about error handling aside...). But what if b is a global (or otherwise shared) variable? And what if another thread is modifying 'b' while this thread is using it? In some (unfortunately probably rare, thus harder to find) instances, b will become 0 right after checking for it, and we will get a divide by 0 error. Even though the code attempted to avoid it. <So this isn't the point I'm trying to make, but it is worth noting that threading makes code review harder. Well, so do global variables. But threading + globals/sharing really causes trouble. And threading without at least some sharing isn't much use (ie without sharing you would never get a result back from the threads!).>
So this looks like threading 101 (ie as basic as it gets), and it is -- what's the point? The point is that the above code is very normal – you check something, and then expect that condition, that state, to hold true while you do what you need to do. That very simple assumption that your variables stay 'stable' after checking them – ie inside the if-statement - that assumption must be thrown away in lock-free programming. It may seem obvious in the example, once we mention that 'b' is being changed in another thread, but it really needs to be stressed strongly – the idea that things don't change inside your if-block is very ingrained via day to day programming, and it is hard to 'unlearn' it. You *will* forget and not notice that you are making this assumption.

Consider another case:

```
if (b != 0)
{
   x = 10;
}
```

Here 'b' may still be changing, and thus the code is still probably, in general, unstable (ie gives different results depending on 'whims' of another thread and the scheduler), but at least if 'b' changes while inside the if-statement, we aren't using it. This is a subtle but important difference. Let's use this to our

advantage:

```
temp = b;
if (temp != 0)
{
   x = a / temp;
}
```

Here the value of x is still dependent on the value of b at a particular point in time – a 'snapshot' of the state of b (and thus may be at the whim of another thread), but at least we won't get a divide by 0.

```
if (b != 0)
{
   x = a / b;
}
```

```
temp = b;
if (temp != 0)
{
   x = a / temp;
}
```

As simple as it may seem, assuming that something does not change from one line of code to the next is a big source of bugs with lock-free programming. As a way to keep this in mind, think about a long family trip with kids in the back seat. The kids ask "Are we there yet?" The parents answer "No". Then they right away ask again "Are we there *Now*?", - "No" - "How about Now?" - "NO!" ... and you thinking they are crazy (or at least annoying), but you should encourage them – they may grow up to be good lock-free programmers. For, in lock-free programming, if you just got a answer a second ago (or a line of code ago), it doesn't mean you won't get a different answer the next time you ask or next line of code. In some sense, this experience can be summed up as:

"Are we there yet?"
"No. Were Here!"

ie right after getting one answer, the state changes. Try that with your kids next time.

A last example. Here's part of some code (that I wrote 15+ years ago) that is being called from a device driver:

```
void Callback()
{
   static inside = false;  // re-entrancy debugging

   if (!inside)
   {
      inside = true;

      // non re-entrant
      do_stuff();
   }
   else
   {
      printf("oh my gosh, it's re-entrant!!!");
   }
}
```

[A quick background - this was code for working with a beta video driver (with little documentation) and things weren't quite working as expected. After much consternation, I began to suspect that the driver was calling me back *while already inside the callback function* – ie it was re-entrant (via low-level interrupts – although the docs didn't mention that I was actually being called at interrupt time!). So I added this debugging code to detect if that was the case...]

So what's wrong with that code? (Besides the big-picture fact that a device driver callback might be taking so long that it is getting re-interrupted before finishing – a definite no-no for interrupt level driver coding.)

This code appeared to show that the callback was re-entrant, but sometimes, even inside the if-block, things would screw up – ie even in the 'non re-entrant' case. Why?

To anyone familiar to threading, or interrupt level coding, it is probably obvious, but to me it was a revelation at the time (the young naive programmer that I was :-).

The problem is that there is a single bad point of re-entry. A single point in the code where if the execution is interrupted and the function is called and re-entered again, we are in a bad state:

```
if (!inside)
{
   // ***bad re-entry right here***
   // ie after the if, but before the set.
   // if execution is interrupted and the function re-entered
   // after reading 'inside' but before writing it
   // then we are inside, but inside == false :-(

   inside = true;
   ...
```

The lesson: *Read between the lines.* Between any two lines of code (or within a single line of code possibly) other code - lots of other code - might get a chance to run.

What we really need is a way to do the !inside *check* and the inside = true *set* at the same time without chance of interruption in between. And so my second revelation came in the solution to this problem: I consulted with the resident 'low-level' programmer, and he introduced me to the asm XCHG "exchange" instruction (Intel x86), which sets a variable and retrieves its old value **in one atomic step** ie no instructions can modify the variable in between the read and the write; they happen 'at the same time' or *atomically*. No 'between the lines' possible. (By the way, it is called "exchange" (instead of "set_and_retrieve_old" because it exchanges (swaps) the variable's value with the value of a register, atomically, letting you then read the now old value from that register. And because "set_and_retrieve_old" is cumbersome to say...) So now I could write (with a suitable C function or macro to hide the asm):

```
if (XCHG(&inside, 1) == 0) // atomically set 'inside' to 1
{                          // while returning its old value
   ...
   // now we know for sure that we weren't already here!
}
else
{
   // reentrant!
}
```

So we come in, stake a flag in the ground marking our presence while simultaneously seeing if anyone else was already there. You might note that if 'inside' was already == 1, then we are just redundantly setting it to 1 again. But of course, that's not a problem in this case. The significant part is when 'inside' *was* 0 and just became 1, which we can check via the return value of XCHG being 0, and thus recognize when we are the first ones in.

Without the single-step XCHG instruction I would not have been able to close the re-entrancy hole. (Well, maybe I could have shut off interrupts, but that's another story. And of course XCHG didn't actually fix the real problem of why I was being re-called so frequently, but at least it confirmed my suspicions and allowed me to move forward with a real solution...)

Anyhow, maybe that's just a long way to say that although I've only been doing lock-free programming for over 5 years (and general threading for 10+), I can say I first used XCHG 15+ years ago.

And why is that significant? Because as we will see, instructions like XCHG (and its brethren) are the bread and butter of lock-free programming...

Meet your best friend, CAS

There is one particular "sibling" of XCHG that you will get to know and love: CAS. CAS stands for Compare And Swap. (Again, the swap is with a register, so the concept gets a bit lost at the C/C++ level.) Also known as Test And Set, or Compare And Set, but let's just stick to calling it CAS. CAS doesn't just blindly do the exchange/swap, instead it only swaps *if* the variable is currently a specific value. In code, it would look something like this:

```
bool CAS(word *target, word oldvalue, word newvalue)
{
   if (*target == oldvalue) // if the target is oldvalue
   {
      *target = newvalue;   // then set it to new value
      return true;  // success!
   }
   return false;  // failure :-(
}
```

But of course the important part is that CAS does all that atomically. Otherwise it wouldn't be your lock-free friend.
Now, it is important to note that there are many variations of CAS:

- different names (InterlockedCompareExchange on Windows, OSCompareAndSwap on Mac, etc)
- different param passing - target could be a reference instead of a pointer
- different param orders (target, old, new) vs (target, new, old) etc – this is particularly frustrating!
- different return values – bool success vs returning previous value of target (like exchange)

Luckily this will all be standardized in the next C++.

For this discussion, we will stick to calling it CAS, and having it return a bool. (Although returning the previous value is often more useful, returning a bool typically makes code easier to understand. If it wasn't for the purpose of exposition, I'd prefer one that returned the previous value.)

A word about 'word'. In general, CAS comes in different sizes depending on the CPU, but typically it will be the same size of an int and/or pointer. Sometimes CAS can work on memory twice as wide as the default int. This is *extremely* handy, as we will see later. But for now, let's just gloss over the size of the CAS, with the word 'word'.

Before getting too far ahead – you might ask, how does CAS do this? In particular, how does it do it *atomically*? Well, the simple (non-)answer is that it is built into the CPU. In other words, it might as well be magic. If your CPU doesn't offer a CAS instruction (of some form), then you won't get far. (Well, without getting too far onto a tangent, your CPU could instead offer the 'hand-wavingly' (but not exactly) equivalent *link-load/store-conditional (LL/SC)* instructions, but let's just wave our hands for now, and hide those as implementation details under CAS.)

What can we do with CAS? Why, we can increment a number across multiple threads without losing count. Because counting (with threads) is harder than it seems! First consider the non-CAS case:

```
int counter = 0;
void main()
{
   start_threads(thread1, thread2);
}

// Thread 1
void thread1()
{
   int i = 100;
   while(i--)
      counter++;
}
```

```
// Thread 2
void thread1()
{
   int i = 100;
   while(i--)
      counter++;
}
```

(yes, there could be some refactoring between thread1() and thread2(), but I wanted them to be shown in parallel. :-)

We have 2 threads, each incrementing the same variable 100 times. 2 * 100 = 200. What are the chances that the program ends with counter == 200?

I'd say not likely. '++' on ints is not typically a single atomic operation. It is effectively

```
counter = counter + 1;
```

or likely even more precisely:

```
register temp = readmemory(&counter);
temp = temp + 1;
writememory(&counter, temp);
```

<Remember how 'read between the lines' can even happen in the middle of a single line!?>

ie the '++' operation is a Read, a (temp) Modify, then a Write operation. This is important enough to be abbreviated as RMW.

The problem, again, is that you cannot make assumptions about what happens to variables after you just read them ("*Are we there NOW?"*). In particular, after reading counter, and finding its value to be, say, 10, you would calculate temp to be 11. Meanwhile thread2 may have changed counter to 11, or 22 or 100, or 5 actually, as we shall see. So once we calculate 11, and rewrite counter, we have lost any work performed by thread2. And/or screwed up any work thread2 is currently attempting, in the same way that it may have interfered with our work. It is easy to see that some of the increments might get 'lost' by the overwriting of the other thread.

But would believe that 'count' could sometimes not just lose a step, but actually go down instead of up? (http://groups.google.com/group/comp.programming.threads/browse_frm/thread/62e5cced42865212).

```
void thread1()
{
   int i = 100;
   while(i--)
   {
      prevCounter = counter;
      counter++;
      if (prevCounter > counter)
         printf("impossible!?");
   }
}
```

```
void thread2()
{
   int i = 100;
   while(i--)
   {
      prevCounter = counter;
      counter++;
      if (prevCounter > counter)
         printf("impossible!?");
   }
}
```

Can this do the impossible, or at least print it?

In each thread 'counter' is only ever incremented, always increasing. How could it go down instead of up?

[This would be a good time for an audience poll, to see if this is easy/obvious, or surprising, and thus gauge in general where the audience is 'at' w.r.t. threading.]

Explanation: basically, thread2 reads, then stalls while thread1 does lots of increments, then thread2 jumps back in and sets the counter to a very outdated (we're talking 1970's bell-bottoms) value:

```
...thread1,thread2 do some incrementing...
thread1 reads counter, finds it is 8
     thread2 reads counter, finds it is 8
thread1 adds then writes counter (now 9)
thread1 reads, adds, writes counter (10)
thread1 reads, adds, writes counter (11)
     thread2 adds 1 to it's old value (8) of counter (8 -> 9)
     thread2 writes counter (9)
thread1 reads counter (9) even though it last wrote 11.
```

Result:
thread1 previously saw counter as 11 now sees it as 9. Yet counter was only ever incremented. Yikes!

CAS to the rescue: Increment a value, without having the other thread destroy your work

```
word atomic_increment(word & target)
{
   word oldval;
   word newval;
   do
   {
      oldval = target;  // READ
      newval = oldval + 1;  // MODIFY
   }
   while ( !CAS(&target, oldval, newval) ); // WRITE or retry

   return newval;
}
```

Note that this is just incrementing by 1 not 100. ie ++ just got a bit more complicated. It has a while loop, even though the end result is a single increment.
This technique is known as a *CAS loop*. We read the target, then basically say, "if target is still what it just was, set it to something new". In this case, a new value of 1 more than what it just was. The important part is the retry – if target changes right after we read it (because another thread modified it), we loop around, and read it again, computing the new desired value based on the re-read, and try the CAS again. (You can see now why a CAS that returns the old value is useful – it allows you to avoid an extra re-read when retying the loop. Remember, all this lockfree madness is for the sake of speed!)

It is important to convince yourself that we never 'lose' an increment in this case. If target gets changed behind our back, we don't just set it to an out of date value. We instead abort the attempt and try again. If you imagine that the target is counting votes, you are just making sure your vote is counted. You come to the voting booth (one of many set up in a room), and you try to make your vote. You see the current value is 12, so you know that since your vote counts (and you can only vote once!) that the new vote count will be 13. But when you try to write 13 (using CAS) you find out you can't because the vote count is now already 15 (wow, must be other people voting in other booths at the same time!). But remember, your vote counts – don't give up, if the current count is 15, set it to 16. Lather, rinse, repeat if necessary. Sooner or later (hopefully) your CAS will succeed (ie no one else tries to vote at the exact same time as you), and your vote will be counted, and off you go.

So our original “counting with threads” now looks like this:

```
// Thread 1
void thread1()
{
   int i = 100;
   while(i--)
      atomic_inc(&counter);
}
```

```
// Thread 2
void thread2()
{
   int i = 100;
   while(i--)
      atomic_inc(&counter);
}
```

(note that *both* threads must call atomic_inc(), not just one of them!) and in the end counter == 200. Always.

Exercise for the reader: Similarly implement atomic_decrement, atomic_add, atomic_multiply, atomic_plus5mod17, etc.

```
do
{
   oldval = target;  // READ
   newval = (oldval + 5) % 17;  // MODIFY
}
while ( !CAS(&target, oldval, newval) ); // WRITE or retry
```

Vitally, note that all these functions can be intermixed at the same time across threads – one may be atomic_incrementing while another is atomic_decrementing, etc. Of course this only works if ALL threads modify the target by the same underlying method – CAS. As long as they are ALL using CAS underneath, they will each see any 'last minute' modifications from other threads, and handle it appropriately.
That is, all voting booths must be operating the same way for your vote to be counted!

Wait, I thought polling was bad! (pun intended)

At first blush, this CAS loop may look very similar to polling for a value in a loop, and I was told that that was bad. For example, you have probably seen code like this before, that continually 'polls' for the value of a variable until it changes:

What's wrong with this code:

```
while (!done)
   yield();

// or worse
while (!done)
   ;
```

(Answer: lots.)

Typically, in a code-review say, I'd put a big red flag on this code. It is just burning CPU time without

getting anything useful done, while it waits for some other thread (presumably) to get something 'done'.

How is a CAS loop any different? (Besides being a do-while instead of a while-statement.) If the CAS fails, it loops around, doing nothing, just like polling... [Audience?...]

What's the difference?

```
// SPIN/POLL                    // CAS LOOP

// see if done;                // see if we can update target;
// if not, try again           // if not, try again

while (!done)                  do
   yield();                    {
                                  old = target;
// or worse (?)                   new = old + 1;
while (!done)                  }
   ;                           while (!CAS(&target, old, new));
```

The difference is that the CAS loop *knows* that if it failed, at least *some other thread* must have managed to get something done (ie modified the target). Whereas the 'while (!done)' polling may be waiting for a thread that is currently paused – ie doesn't even currently have a time slice in the scheduler. Ah the scheduler. This is a significant difference. Significant enough to be the basis of the definition of lock-free programming:

> An algorithm is *lock-free* if at all times at least one thread is guaranteed to be making progress.

A CAS loop is lock-free because either the current thread modifies the target and leaves the loop (ie makes progress) or some other thread snuck in, modified the target and left the loop (made progress).

What about polling? Isn't the other thread (the 'doing' thread - one that will set 'done') making progress? While that may be the intent, it may not be the case. It may very well be that the scheduler has paused the 'doing' thread. Meaning it isn't making progress. But neither is the polling thread getting anywhere! Polling is NOT lock-free because a paused (ie by the scheduler) thread can pause the polling thread(s), such that no thread is making progress. (Yes, the scheduler could pause both threads in the lock-free case, but there's nothing we can do about that.)
So a CAS-loop might at first feel like polling/spinning (it did to me), but it is fundamentally different.

Before moving on, let's spin for a second on spinning:

```
while (!done)
   ;

// or
while (!done)
   Sleep(0);  // or maybe Sleep(1) or yield() etc
```

Besides burning CPU time, code like this might actually be an infinite loop. What if this thread and the doing thread are actually *sharing* the same CPU? Quite possibly, the system is attempting to time-slice between this 'waiting' thread (doing nothing worthwhile) and the 'doing' thread. So the doing thread is, for example, taking twice as long, because it keeps giving up CPU time to the waiting thread. Now, what if the waiting thread is *at a higher priority* than the doing thread? This may mean it gets more time to wait ,leaving less time for 'doing', - or worse it takes all the time, leaving *no* time for 'doing' anything, and thus we get an infinite loop! This is a severe case of *priority inversion*.

> *Priority inversion* occurs whenever a high-priority thread must wait for a low-priority thread.

(Note that based on the definition, this could also happen using proper locks (ie a high-priority thread waiting for a lock held by a low priority thread), not just spinning, but the consequences might be less severe.)
And a note about Sleep(0) for Windows programmers out there: Read the docs – Sleep(0) only gives up its time-slice to equal or higher-priority threads. If the doing thread was lower priority, we would be into infinite loop territory. So in most cases, even Sleep(1) would be better (as it does give up to threads of all priorities). It would still be stealing CPU time, but eventually the other thread would get 'done'.

Is spinning always bad? No. In fact many lock implementations (ie EnterCriticalSection under Windows for example) actually spin on the 'is_the_lock_available' flag for a bit before going into the kernel to do a full wait. Small amounts of spinning can be OK – it depends on the situation. In fact:

What does this code do?:

```
void spin_lock(word & lock)
{
   while (!CAS(&lock, 0, 1))  // or XCHG(&lock, 1)
      ;

   // what's the value of lock here, after the loop?
   // what was the value of lock when XCHG/CAS last 'saw' it?
}

void spin_unlock(word & lock)
{
   XCHG(&lock, 0);
}
```

This is what is known as a spin-lock. This is sometimes worthwhile when you know that a full lock (which typically requires going into the kernel) would be too 'heavy'. (ie in a case where by the time you have switched into the kernel, the lock is probably free already). But you need to be aware of the potential of priority inversion.

And now, back to CAS-loops, and atomic_increment().

```
do
{
   oldval = target;  // re-READ
   newval = oldval + 1;  // MODIFY
}
while (!CAS(&target, oldval, newval)); // WRITE
```

Now, a CAS-loop is still, well -looping- (at least potentially), and so possibly at least one thread is NOT making progress. In fact, in theory at least, if other threads keep interfering one thread could be stuck here forever, and *never* make progress! Wouldn't it be nice if we could avoid any even potential looping? Sometimes we can. Not often, but enough for it to have its own terminology:

> An algorithm is *wait-free* if ALL threads are guaranteed to be making progress.

A CAS loop is lock-free, but not wait-free, because the chance of failed attempts and retries mean that a thread may not be making progress. ie we cannot *guarantee* that all threads are always making progress. Only an algorithm without retry-loops can be wait-free.

Our version of atomic_increment() uses a CAS-loop (lock-free). But atomic_increment (or the more general atomic_add) is so common and useful, that it is often its own CPU instruction (often called fetch_and_add or similar). fetch_and_add doesn't fail/loop/retry. It just does. (more magic). Which means, by the way, that it is wait-free – no chance of failure - thus always making progress. This has the somewhat unfortunate and debatable outcome that an algorithm (on paper) that relies on atomic_increment/add/decrement/etc, may or may not be wait-free, depending on the hardware on which it will run. :-(

Since this discussion is entitled “Lock-free...” and not “Wait-free...” we will stick mostly to lock-free algorithms, and somewhat assume atomic_increment et al is built on CAS. If it happens to be intrinsic, then that's just a bonus. Regardless of how atomic_increment is implemented, we will have plenty of other more specific cases where there is no CPU intrinsic, and a CAS loop will be required.

Let's Get *Started*

What's wrong with this code?
(From an actual commercial software product sold to 100K+ people. Variable and function names changed to protect the guilty.)

```
void someFunc()
{
   static MyCriticalSection cs();

   scope_lock lock(cs);

      do_important_stuff();

   // cs unlocked in scope_lock destructor
 }
```

If you need a hint as to what is wrong:
this was under Windows...
With MyCriticalSection implemented via the OS's CRITICAL_SECTION structure...
Which needs to be first initted with InitializeCriticalSection(&critical_section)...
Which happens in the constructor of MyCriticalSection...
Which happens... um, when, exactly?

Since cs is a function level static, its constructor is called *the first time control flow enters the function*, according to the standard (paraphrased). But what about when two different threads enter the function for 'the first time' at the 'same time'? What does the standard say about that? Well, currently, nothing. Not until the next standard (C++0x) will there be any talk about threads. So at best, it is implementation defined. And the implementation in use (for our example) was doing nothing special for function-level statics with respect to threads. So what does a non-thread-aware version of function-level-statics look like?:

Consider a seemingly simple function foo:

```
void foo()
{
    static MyClass mine;

    do_stuff(mine);
}
```

the compiler needs to ensure that 'mine' is constructed only once – the *first time* the function foo is entered. It typically does this, as you might expect, using a global flag – called, say, 'done_yet':

What a function level static really looks like:

```
void foo()
{
    // POD, no constructor to worry about.
    // Set to 0 at compile/link/load
    static bool done_yet = false;

    // can't call constructor, so just reserve memory
    // again POD, no constructor
    static storage_for<MyClass> __mine_storage;
    static MyClass & mine = reinterpret_cast<MyClass &>
                                            (&__mine_storage);

    if (!done_yet)
    {
          done_yet = true; // set here (A)...
          new (&mine) MyClass(); // call constructor!
          done_yet = true; // or 'equivalently' set here (B)
    }

    do_stuff(mine);
}
```

Just to show what the compiler is doing, we've split up MyClass into its memory-storage and its 'classness', to show in C++ roughly what the compiler needs to do in assembler. *Exercise for the reader:* Implement storage_for<T>. *Hints:* boost::aligned_storage, boost::alignment_of.
And a note about those static PODs. As we mentioned the standard doesn't really say when constructors are called for function-local statics. In fact, even statically-initted structs or the zero-init of statics can not be assumed to happen before entering the function. But in practice ALL systems do the zero-init of statics (function-local or not) basically intrinsically at load time – the 'zeros' and/or static values (ie static foo = { 0,1,false, 17};) are compiled into the executable image, and thus are 'there' when the program is loaded into memory. So technically these static locals are not guaranteed safe, but are as safe as you can get. P.S. I think C++0x will clarify this issue.

So, the compiler's use of 'done_yet' should look scarily similar to our 'inside' (re-entrancy problem) and our very first use of XCHG. And it is. If there are threads to worry about, another thread could come in at just the right time (after checking, but before setting), and we would end up with 2 threads within the 'if (!`done_yet`)' block, and we could call mine's constructor twice (which may or may not be a disaster, but in general, assume it is!). And/or (in particular if `done_yet` is set to true at A) the call to do_stuff(mine) could actually be working with a 'mine' that has yet to be constructed at all (or is anywhere moving along in the process of being constructed in the other thread). BOOM! The 'mine' exploded! (again, pun intended)

So, in general, IF threads are involved, function level statics are unsafe. Note that an easy solution is often to just pull the static outside the function, which forces the static to be constructed at program startup when there should only be one thread running (or, for dynamically loaded libraries, at least at library load time, which is carefully guarded against threading on *most* OSes). ie

```
static MyClass mine; // static outside of function == safer!
void foo()
{
   do_stuff(mine);
}
```

So, back to our original real actual code from a shipping product. Do you think threads were involved in that case? Well, we had a big clue – the function local static *was a critical section object*. ie it is only useful when threads are involved, so someone was at least concerned enough that threads *might* be involved. (I personally found this irony quite amusing!) As yet a further wrench in the system, the function was actually a template function

```
template <T> someFunc()...
```

so that a `static MyCriticalSection cs;` existed *for each* type T used. This was intentional, and necessary for the system. However this made it impossible to move the function-level static outside the function! (At least for our compiler.) What to do?

[Slide] “Use Locks!”

Oops. Just a reminder. Alas, this begs the question. It is a lock (in this case MyCriticalSection) that we need to put a lock around...

CAS to the rescue(?)
Ensure the function-local static is only constructed once

```
void someFunc()
{
   static bool done_yet = false;
   static storage_for<MyCriticalSection> cs_storage;
   static MyCriticalSection & cs =
         reinterpret_cast<MyCriticalSection &>(&storage);

   if (XCHG(done_yet, true) == false) // was false, now true
   {
      new (&cs) cs();  // now call constructor, ONCE!
   }

   scope_lock lock(cs);  // we can now use the cs (?)
   do_important_stuff();
 }
```

So now the constructor only happens once. *What's wrong with this code?*

Well, we closed the door on the if-statement before a second thread could come in, so we won't get double-construction, but the second thread isn't waiting for construction to complete. It is off doing important_stuff with cs at an unknown stage of construction. So far, this isn't much better than what the compiler was doing. We need to wait for construction to complete:

```
void someFunc()
{
   static bool started = false;
   static bool done_yet = false;
   static storage_for<MyCriticalSection> cs_storage;
   static MyCriticalSection & cs =
         reinterpret_cast<MyCriticalSection &>(&storage);

   if (XCHG(started, true) == false) // was false, now true
   {
      new (&cs) MyCriticalSection(); // call constructor, ONCE!
      done_yet = true;
   }
   else // wait
   {
      while (!done_yet)
         Sleep(0);
   }

   scope_lock lock(cs);
   do_important_stuff();
 }
```

Excellent. So now we wait until the cs is properly constructed, and then we use it to protect the important_stuff.

Hey wait a second, but now we are polling again! There has got to be a better way...

[Slide] "Use ~~Locks~~ Boost!"

```
// dangerous function-object, not to be used lightly
// calls constructor on target
// whenever operator() is called!
// This would be nicer with C++0x lambdas
template <typename T>
struct inplace_construct_er
{
   T & target;
   initter(T & t) : target(t) {}
   void operator()()
   {
      new (&t) T();
   }
}

void someFunc()
{
   static boost::once_flag onceFlag = BOOST_ONCE_FLAG_INIT;
   static storage_for<MyCriticalSection> cs_storage;
   static MyCriticalSection & cs =
         reinterpret_cast<MyCriticalSection &>(&storage);

   inplace_construct_er<MyCriticalSection> initter(cs);

   boost::call_once(onceFlag, initter); // call ctor ONCE!

   scope_lock lock(cs);
   do_important_stuff();
 }
```

boost::call_once(once_flag, function_obj) calls function_obj() *once* even in the presence of threads. The once_flag is where we track whether the function_obj has been called once yet, so it needs to be static. The function_obj just needs to be any callable object (function pointer, object with operator(), result of a bind(), etc), and does not need to be static. The once_flag needs to be initted with BOOST_ONCE_FLAG_INIT which is implementation defined (and different per OS, etc), but is always a compile-time constant expression (typically something like { 0, 0, 0 } for a once_flag that is a small POD struct). So no constructors to worry about for the once_flag.

Let's recap so far:

Avoid lock-free when you can.
- Use locks.
- Use boost.
- Etc. (other libraries or proven code)

See Stepanov quote from first slide.
We don't write our own STL if we don't have to...

Avoid lock-free. We don't typically write our own linked-lists anymore, we use STL. And linked-lists are easy to write! But STL is easier to use, and more importantly, safer. Similarly, use libraries that work (ie boost!) instead of doing it yourself.

But, let's just say we didn't have boost::call_once (or pthread_once, or an OS with Mutexes that are default initted...). Or in fact, what if *we* were implementing call_once...?

Everything I need to know about lock-free I learned from call_once()...

When I first came across this local-static CriticalSection problem, boost::threads was unavailable. I basically needed to write my own call_once. So, how *does* one implement call_once?

Here is the general form, similar to our specific attempt to init the critical section:

```
void call_once(once_flag &flag, Function someFunction)
{
   if (!flag.done_yet)
   {
      if (XCHG(flag.inside_function, true) == 0)
      {
         someFunction();
         flag.done_yet = true;
      }
      else
         wait_for(flag.done_yet);
   }
}
```

We need to somehow wait for done_yet to become true. Well, we could always poll, right?

```
while (!flag.done_yet)
   Sleep(0);
```

This is in fact what pthreads-win32 was doing circa 2005. (pthreads-win32 is an open-source implementation of the POSIX threading API standard (give or take) for Windows.) I came across this code around the same time as my CriticalSection problem (probably while looking for a solution!). I pointed out the priority inversion problem of Sleep(0), and eventually, after lots of attempts and discussions and learning all around, pthreads-win32 didn't just stick in a Sleep(1). Let's see some of the problems encountered...

Since this was on Windows, instead of polling, we can wait on an event:

```
void call_once(once_flag &flag, Function someFunction)
{
   if (!flag.done_yet)
   {
      if (XCHG(flag.started, true) == 0)
      {
         someFunction();
         flag.done_yet = true;
         // notify waiting threads that we are done:
         SetEvent(flag.hEvent);
      }
      else
         WaitForSingleObject(flag.hEvent, INFINITE);
   }
}
```

But of course, we need to first call CreateEvent to, well, create the event:

```
void call_once(once_flag &flag, Function someFunction)
{
   if (!flag.done_yet)
   {
      if (XCHG(flag.started, true) == 0)
      {
         flag.hEvent = CreateEvent(0, true, true, 0);
         someFunction();
         flag.done_yet = true;
         // notify waiting threads that we are done:
         SetEvent(flag.hEvent);
      }
      else
         WaitForSingleObject(flag.hEvent, INFINITE);
   }
}
```

But of course, as we have learned, we need to wait for this CreateEvent call to finish before we can use the event to, well, wait. So use call_once to create the event. Oh, wait, we are writing call_once. Are we ever going to get anywhere? Or is it “but but but turtles” all the way down?

Luckily, we've narrowed the problem enough to make progress. We can use the Tuttle "We're all in it together" technique (that's a Brazil movie reference). Note that even though the first thread enters does the real work (calls someFunction), and the second thread just waits, we don't actually care which thread creates the hEvent. The threads can work together – one can do the work of the other – as long as they can do it without mixing each other up. This works when

Two (or more) threads need to do something and:
1. any thread can do the work (ie every thread has all necessary information, etc)
2. the work can be made *visible* to all threads using a single CAS

And to see, by example, what this means:

```
void AtomicCreateEvent(HANDLE & hEvent)
{
   if (!hEvent) // anyone created it yet?
   {
      // Create an event - stored LOCALLY for now
      HANDLE temp_event = CreateEvent(0, true, true, 0);

      // race other threads for setting the shared event handle:
      // must use CAS!
      // *atomically* only set the handle if it is currently 0:
      bool winner = CAS(hEvent, 0, temp_event);

      if (winner) // wasn't 0 - we lost the race :-(
         CloseHandle(temp_event); // our event thus isn't used
      // but either way, the good news is that
      // there is now an event available for all threads to use
   }
}
```

A few things to note:
- it is possible that more than one thread attempts to construct an event at the same time -> this is inefficient (but safe). The losing thread throws away its work.
- Most likely, only one thread ever creates an event (ie the 'window of opportunity' is small), - depending on degree of contention (ie number of threads, how often this is called) of course
- ironically, CreateEvent might contain locks (internally in its implementation), such that the 2 (or more) racing threads get serialized for at least part of the CreateEvent call
- And lastly: * At least one thread makes progress * ie the function is lock-free

This is a very common technique of lock-free programming.

Harry Tuttle "We're all in it together" technique:

1. multiple threads attempt to do the same work
2. each thread independently does the **work locally**,
3. attempt to make work **globally visible** using a single atomic operation (typically **CAS**)
4. one thread wins, losers throw away temporary work

The is really two techniques in one. The inner technique is actually the **most** common technique, really the **only** technique of lock-free programming. Everything else relies upon it.

[Slide for emphasis] Work Locally. Publish Atomically.

Or for the bumper sticker: (ie similar to the familiar eco-friendly "think global, act local"):

[Slide] Act Local. CAS Global.

Go back and look at atomic_increment, and see that it in fact does the same thing – work locally (increment into local variable) *publish* the result using a single atomic operation.

Back to our code. Now every thread ensures that the hEvent is ready before it is used:

```
void call_once(once_flag &flag, Function someFunction)
{
   if (!flag.done_yet)
   {
      if (XCHG(flag.started, true) == 0)
      {
         someFunction();
         flag.done_yet = true;
         AtomicCreateEvent(flag.hEvent);
         SetEvent(flag.hEvent); // wake up waiting threads!
      }
      else
      {
         AtomicCreateEvent(flag.hEvent);
         WaitForSingleObject(flag.hEvent, INFINITE);
      }
   }
}
```

And now we have a 'working' call_once.

What's wrong with that code?

1. flag.hEvent is never deleted (or maybe not until flag's destructor, and since flag is static, this means not until the program ends)
2. flag.hEvent is created *even if no threads are waiting* (ie OK, but inefficient, particularly considering that in most real-world cases, there are probably not any waiting threads)
3. What if someFunction() throws?!
4. Didn't you just say something about CAS Global?...

For now, let's leave 1. and 2. and 3. to the ever famous "exercise for the reader" category. In particular *Write an implementation of* `call_once` *that*

- leaks no resources
- doesn't create resources unnecessarily
- handles exceptions (act as if call_once wasn't called by this thread at all)

This is the exercise pthreads-win32 eventually took on and succeeded in. Boost::call_once (Windows version) was recently changed such that it now succeeds on all accounts except one - it leaves the event until the program ends (which was a conscious decision to pick clarity/maintainability over "perfection"). Honestly, that exercise alone can make you a lock-free expert. The resulting code is likely more complicated than many are comfortable with (and makes polling with Sleep(1) – which is inefficient but *simple* – look like a serious consideration). So add 'doesn't poll' to your list of requirements, else Sleep(1) would be a good solution.

[Please see me at the end of the week with your versions of call_once!]

So, if you are familiar with lock-free or call_once or initialization problems or ...(Singletons! Did someone say Singletons?) then you are probably itching to point out that we still have big problems with our code. For example, did we "CAS Global" everything?

What about the ever important 'done_yet'? In our example, we use XCHG on flag.started. But what about done_yet? Why don't we need a CAS/XCHG to set done_yet? The obvious answer would be because we aren't 'racing' with another thread to set the flag – the other threads are waiting. Or possibly checking 'if (!flag.done_yet)' also without a CAS. But they are just reading, not racing to set the flag. Is that a problem? Hmmm....

DCLP – The Double-Checked Locking Pattern

You may/probably have heard of the double-checked locking pattern, and about its difficulties, but some background just to be clear:

Singletons / Singleton Pattern

A singleton is a type of object where only one instance of the object is allowed. For example, in a GUI system you might want to enforce that only one Mouse object can be created, because (typically at least) a GUI only has a single physical mouse input device. And typically, a singleton is a single *shared* resource, again like the mouse, so singletons are typically global.

```
static Mouse theMouse; // global, one and only mouse

main(...)
{
}
```

[Slide]Globals are Bad.
[Slide]Singletons are Bad.
[Slide]Use Locks. (of course)
[Slide]MACROS are EVIL.
[Slide]I'm Overgeneralizing.

<rant>
We aren't really here to discuss they're pros and cons, but I just wanted to let you know my opinion – which, at a minimum, is that the singleton pattern is 'overused'. So think carefully before using one. A bit of advice – if you think something might be a singleton, code it as a normal non-single object – push the 'singleton-ness' outside the class and onto the user of the class. ie even if you have to supply the helper functions (like a GetTheOneAndOnlyMouse() function) separate this from the class itself. You might think the singleton-ness is inherent to the class and thus should be part of the class (and yes, sometimes it is), but then you might later be surprised to find that actually, some GUIs do have more than one mouse...
Looking at it the other way – I see too many singletons that are 'single' only because the *use case* requires just one, not because the object *inherently* must be single. For example, we only use one sun, but there are quite a number of them in the universe. So it is not a singleton by nature, just by use. On the other hand, there is only one universe – oh wait, that depends on which physicist you ask...
</rant>

And of course, we want to limit this to a single mouse so what happens with:

```
static Mouse theMouse; // "there can be only one" (Highlander)
static Mouse theOtherMouse; // ??!!
```

You will need to do some C++ wrapping to make sure that theMouse and theOtherMouse actually refer to the same one-and-only mouse internally. That is, they really are just wrapping 2 separate references to the same singleton object. Etc.

LazySingeton

Now, once you have a Singleton, it needs to be initialized at some point (once!). This could be at the start of the program:

```
static Mouse theMouse; // global, so constructed at startup

main(...)
{
}
```

But sometimes the startup costs of the singleton are high, and the singleton isn't widely used, so you might not want to initialize the singleton until it is really needed. (Imagine a program that had, say, 1000 singletons, all initializing at startup. Slllloooooowww. Again, names omitted to protect the guilty.)

```
static Mouse theMouse; // global, but empty/light constructor
void init_mouse(); // this does the real (heavy) initialization

void foo()
{
   if (!mouse_initted)
      init_mouse();

   // use mouse...
   theMouse.setPosition(x, y);
}

void bar()
{
   if (!mouse_initted)
      init_mouse();

   // use mouse...
   theMouse.etc();
}
```

Now, this is C++ and so we also would wrap this up, typically with something like 'get_instance()':

```
Mouse & Mouse::get_instance()  // static function of class Mouse
{
    if (!mouse_initted)
      init_mouse();

   return theRealMouse;  // return reference to the real mouse
}

// usage:
void bar()
{
   Mouse & theMouse = Mouse::get_instance(); // inits if req.

   // use mouse...
   theMouse.etc();
}
```

Or internally, every Mouse call could first check an init flag before doing the real work. Either way, the result is that if no one ever needs the mouse, no one ever calls it, and the mouse is never wastefully initted. (Or, more typically, at least it wasn't initted during program startup – lots of global inits make startup slooooow. Init-on-first-use spreads the init times out over the life of the program, typically making each small init imperceptible, while noticeably speeding up program start time.)

And you can think up C++ ways of preventing duplicate mice, or how to keep the underlying Mouse

object 'private' while still allowing useful access. Etc. Or look up Singletons in Scott Meyers' "Effective C++" book, or Andrei Alexandrescu's "Loki" library, etc. For threading, the thing to focus on is how lazy singleton initialization is handled.

Speaking of Meyers and Alexandrescu... "C++ and the Perils of Double-Checked Locking" Meyers, Scott and Alexandrescu, Andrei, September 2004.
(http://www.aristeia.com/Papers/DDJ_Jul_Aug_2004_revised.pdf)

This is the article that hit home the subtle problems of multi-threaded programs for many people. And it is well described and explained, so I'm not sure why I still see so many questions about DCLP. And not just questions because they haven't read the article, but because they *have*. If they hadn't read it, they typically wouldn't even know there was a problem to begin with. Yet most of the questions I see are of the form "I just read Meyers/Alexandrescu on DCLP. Here's my simple solution; why didn't they fix it this way?...". (The usual answer is because your solution *isn't*, and you need to re-read the article.)
So I'm not sure how better to explain it than they did. I think the real issue is just plain disbelief. And the cure for that is reiteration - ie someone saying "No really, it's true" and explaining it again. So go read the article while I wait....

[Slide] "No really, it's true."

And now let's go over it again so it sinks in :-)

DCLP: What's wrong with this code?

```
static ExpensiveSingleton expensive; // POD, with expensive init
static Mutex mutex;
static bool done_yet = false;

// initting is expensive, so only do so if necessary
void expensive_init()
{
   // locking/unlocking mutex is also expensive***,
   // so check flag first before bothering with mutex
   if (!done_yet)
   {
      // instead of CAS or wait, just use a mutex:
      scope_lock lock(mutex);
      if (!done_yet) // double-check - for race to the mutex
      {
         // expensive init:
         expensive.important_ptr = new Important_stuff();
         expensive.moredata = etc();
         done_yet = true;
      }
      //unlock in scope_lock destructor
   }
}
// ***not really
```

I've stripped away the singleton-class parts, leaving just a potential solution to lazy initialization. A “thread-safe” solution.

Well, we see our friend 'done_yet' from previous examples, has reappeared. ie we again have the same pattern - check if (!done_yet), and do something if necessary. More importantly: if it IS already done, then, well don't bother doing anything. In particular, if it is already done *by some other thread perhaps*.

Short code explanation just to be clear on what the programmer is attempting to do:

[Slide] Use locks!

Yep, in this example the programmer knows that they need a mutex to safely do the real work. Furthermore, they realize they need to check 'done_yet' inside the mutex, *otherwise* another thread could slip inside the if-block and both threads would see 'done_yet' as false. (Maybe they just learned that from our above discussions!)
But if the task is a one-time-only task, ie initialization of a Singleton, then they know once it is done, it need never be done again, so they'd like to avoid the mutex completely if there is nothing to do. ie optimize the common case of done_yet == true.

[Slide] Premature Optimizations...

So they check `done_yet` once outside the mutex – hoping to avoid the mutex altogether, then only if `false`, they check a second time (the *double* check) *inside* the mutex in case another thread snuck in there in the last second. Thus the name – the Double Check Locking Pattern.

(Oh and by the way, locking a mutex is typically fast if it is not already locked – ie it is a simple CAS on most systems, not a kernel call – so not as fast as reading a global, but not too scary either. So the whole idea of DCLP – ie of avoiding the lock, is often a case of *premature optimization*.)

So now, *what's wrong with that code?* Meyers and Alexandrescu did call it the *Perils* of DLCP, there must be something wrong with the pattern. What is it?

Answer: *Memory Barriers*. You need some.

That's the short answer. The long answer is, well, long:

To get to the answer, let's start with what the problem really is, as it is not obvious, and in fact very unintuitive, if you haven't seen it before.

Imagine that ThreadA is clearly first on the scene, gets the mutex and does the initialization:

ThreadA:
calls expensive_init() with done_yet == false, successfully calls do_real_init(), which lets say looks something like this:

```
expensive.important_ptr = new important_stuff();
expensive.moredata = etc();
```

and then after do_real_init():

```
done_yet = true;
unlock();
```

...and it is on its way.

ThreadB now comes in:

```
void some_function_that_uses_expensive()
{
   // imagine done_yet == true via ThreadA,
   // so this returns quickly:
   expensive_init();

   x = expensive.important_ptr->x;   // BOOM??!?
}
```

And ThreadB crashes on the NULL pointer expensive.important_ptr.

What? Wait – ThreadA set important_ptr, then set done_yet, important_ptr can't be NULL. It can't.

Can it?

Yes, in fact it can. In the new world of multiprocessor computing, things are not always what they seem. In particular, things don't always happen in the order you were expecting. The instructions may get reordered by the CPU.

Oh I get it – the compiler reorders the instructions when optimizing the code. Sure, I've heard of that.

No, not the compiler, the CPU. In particular, the CPU in conjunction with the memory system. (Oh, the compiler could as well, but that is usually easier to handle. The trickier case is the CPU. Even if you avoid the compiler and code directly in assembler, the CPU might decide to execute the instructions in a different order.)

For our particular case, what can happen is that the CPU running ThreadB might decide to read important_ptr first, *then* read done_yet. This may be interleaved with what ThreadA is doing, such that...

Our Plan (typical execution):

```
ThreadA
-------
// first thread inside expensive_init()
// initializes expensive
expensive.important_ptr = new...;
// then sets flag
done_yet = true;
                ThreadB:
                -------
                // inside expensive_init():
                read done_yet; // true! - exit expensive_init()
                // outside expensive_init():
                r1 = read expensive.important_ptr;
                use r1->x;
```

CPU's Answer: reorder instructions:

```
ThreadA
-------
done_yet = true;
expensive.important_ptr = new...;
                ThreadB
                -------
                r1 = read expensive.important_ptr;
                read done_yet;
                use r1->x;
```

Now interleave the Threads:

```
ThreadA         ThreadB
-------         -------
                r1 = read expensive.important_ptr; // == NULL!!
done_yet = true;
important_ptr = new...;   // now not null
                read done_yet; // true - exit expensive_init();
                use r1->x;  // CRASH. r1 = NULL
```

(Note that this is just one of a number of possible problematic interleavings + reorderings)

So why would the CPU do something like this – I expect my instructions to happen in the order I wrote them. Well, I suppose I'm OK with the *compiler* reordering for optimizations *as long as it doesn't change how the program works*. But now the *CPU* is reordering AND breaking things?

Actually the CPU and the compiler optimize under the same basic rule:

[Slide] *CPU and compiler optimization/reordering rule:*

> Do any optimizations you'd like, as long as it doesn't change how the program works.

[Slide] Oh wait, *The Fine Print:*

CPU and compiler optimization/reordering rule:

> Do any optimizations you'd like, as long as it doesn't change how the program works.*

> *....assuming a single-threaded program.

And, as always, the details are in the fine-print. Now, the compiler has an excuse – C++ (currently, until C++0x arrives) doesn't have threads. Even though there are millions of lines of multi-threaded C++ code in use today, the language itself says nothing about threads and, for the most part, the compiler optimizes in with total disregard for threads. So for the particular case of reordering read instructions, well read p, then q, or q first then p makes no difference for a single threaded program, so the compiler is free to reorder.

But, again, forget about the compiler. What about the CPU? Surely the CPU running your program *knows* if there are other CPUs present or not. Why would a CPU reorder instructions that might cause your logically correct code to fail?
Because *most* code is single threaded. By a large margin. And *not* reordering would be *orders of magnitude* slower.

What's going on?

(Imagine) How CPUs Work

Caveat: I am in not a hardware engineer, but here is a general outline of how CPUs work, or at least how we can imagine they work, which is hopefully consistent with how they really work, and thus gives us proper insight into what is going on when our code is run, and allows us to program so that what happens is what we need and expect to happen.

> *That was Then:*
>
> speed of RAM == speed of CPU
>
> *This is Now:*
>
> speed of RAM <<< speed of CPU

Once upon a time, the speed of RAM == speed of CPU. But 'Moore's Law' has been applied to CPUs and CPUs are now 100s of times faster than RAM. A *similar* 'law' has happened for RAM – but it has been the size of RAM that's been repeatedly doubling, not its speed. Actually, you *can*, in fact, get really fast RAM, it is just *really really* expensive. This has led to the typical structure of modern

computers where you have a (relatively) small amount of expensive fast RAM, called the cache(s), and then lots and lots of slower RAM or 'main memory'.

Aha! It's the cache's fault. My variables are 'stale' and being read from the cache, when really the right value was in main memory all along!

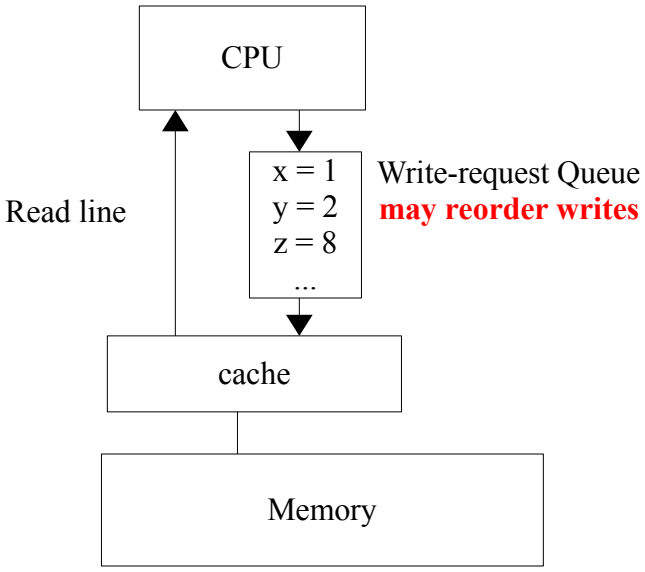
Actually, no. Let's get that misconception out of the way right now. The vast majority of modern hardware architectures implement *cache-coherency* – the cache(s) and main memory are always kept 'in sync'. For example, if a CPU writes to address 0x1700abcd, that address will be marked 'invalid' in all other CPU caches *before* the write is allowed to happen. This and many other operations and communications between the CPUs ensures cache coherency. For more information, look up the MESI protocol (Modified/Exclusive/Shared/Invalid – the 4 typical states of a cache line). There are variations and derivatives of the protocol, but that basic MESI protocol should be enough to understand the basics.

So even with fast caches, memory operations are slower than internal processor operations (registers, etc). If the processor waits for each memory operation to complete, 10s or 100s or cycles can go by with no instructions running. What is the processor to do?

Well, basically, *not wait*. CPU designers decided that memory was too slow, and the processor should just keep going, instead of waiting for memory operations to complete.

How does this work?

Write Request Queue Diagram:



For writes, it is pretty easy. The processor adds the write ("write 123 to location 0x1700abcd please") to a write queue, then continues to the next instruction. The next instruction probably doesn't need to know the read completed (and if it needs the value, well it already has it!), so it can just move on to the next instruction. This allows the memory system time to go through the write queue, and process the write-requests 'at its leisure' so to speak. And here's the thing: it doesn't need to process them in order,

it can reorder the requests if it is faster (because a cache line is busy or another is available or any other reason). Again, for a single processor system, the reorderings would be unobservable.

So:

```
x = 1;
y = 2;
```

can effectively become

```
y = 2;
x = 1;
```

The writes have been reordered.

What about reads?

Well, with reads, the next instruction, or at least a 'soon' instruction probably needs to know the result of the read (else why did we read in the first place), so the processor can't just skip past a read without waiting.
UNLESS the processor was smart enough to look-ahead, see that a read will be necessary soon, start the read early, and hope that the read is completed before it is actually needed.

So:

```
x = 1;  // why not start reading z here?
y = 2;
a = z;
```

If the processor is 'at' line 1, it can look ahead and see that it will later need to read z, so it starts the read 'now' and hopes for it to be done by the time it gets to 'a = z' on line 3.

And processors *are* that 'smart' and do look ahead. Furthermore, they look right past forks in the road:

```
if (x != 0)
   a = z;
```

A processor may look right past the if-statement, and pre-read z, even though it may end up not using z at all. But instead of waiting to find out what x is (or will be at that point), the processor knows that if it does need z, it should get it early and avoid the stall of waiting for the potential read. This is called *speculative execution*, and *branch prediction*. (Speculative execution is working ahead without knowing whether the branch will be taken, branch prediction is the method of deciding whether to speculatively execute or not. ie the processor only pre-executes a branch that it has determined (by previous history for example) it will probably have to take anyhow. Or at least likely enough to make the speculative execution worthwhile.)

In general, we can think of the reads in the same form as the writes – that the read-requests are added to a queue, and again, the requests may be reordered while being processed by the memory system.

So, for registers r1, r2:

```
r1 = x;
r2 = y;
```

can become reordered into

```
r2 = y;
r1 = x;
```

And of course, both reads and writes can be happening, and they can all be reordered:

```
r1 = x;
z = 7;
r2 = y;
w = 9;
```

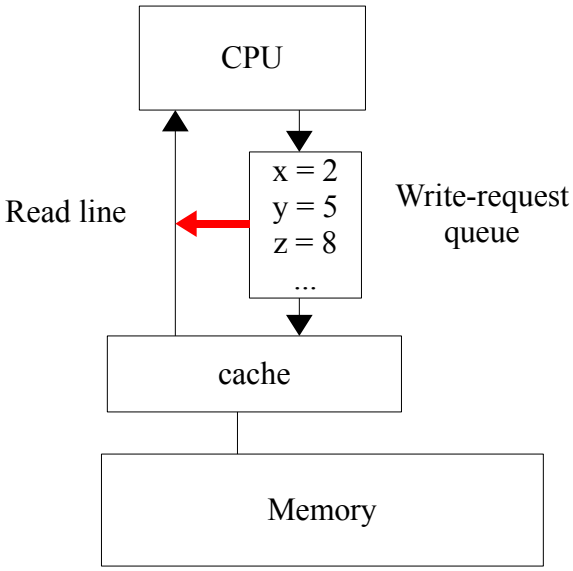
the above can happen in any order.

Note, of course, that something like x = y, requiring a read and a write, won't be reordered:

```
r1 = y;  // read into register
x = r1;  // write result to memory
```

The above *can not* be reordered, as it would change the behaviour of a single-processor system.

And one more tidbit regarding the interplay of reads and writes: a read from memory may first check the write-queue to see if a corresponding write is already pending. ie:



eg:

```
x = r1;  // write x
...
r1 = y;  // register reused, so can't get value of x from r1
...
a = x;   // need to re-read x
```

If the write to x is still pending (not yet committed to memory), the processor knows what the result will be (by checking the write-queue), and uses that instead of reading from cache/memory. (Basically saying 'why read the value when I haven't even managed to write it yet. When I *do* write it, well, then I know what it will be, won't I?!)

I mention that tidbit for a few reasons. One in particular, is that varying processor designs (x86 vs Itanium vs PowerPC vs Alpha vs etc etc) do different amount of reordering. Some won't reorder writes before reads, etc. Some will. But often, even if they say they don't do a particular reordering, they *do* sneak-peek into the request queues, which can have similar results as reordering.

So that's how it works from the single-processor point of view. All these reorderings are valid (maintain program correctness) *if there is only one processor*. Now let's add more CPUs and see what happens, remembering that each CPU will, for the most part, still act as if it is alone, doing all the above reorderings.

Well, back to our DCLP 'done_yet' example, once again. Recall that Thread2 saw done_yet == true, but remarkably, important_ptr was NULL. It should now be clear what happened:

Code as programmed:

ThreadA	ThreadB
`A1) important_ptr = new...;`	`B1) if (done_yet)`
`A2) done_yet = true;`	`B2)  read important_ptr`

Instructions reordered:

ThreadA	ThreadB
`A2) done_yet = true;`	`B2)  read important_ptr`
`A1) important_ptr = new...;`	`B1) if (done_yet)`

And we see important_ptr being read before it is set. BOOM!

And note that either processor may have reordered its instructions, or both. Even if ThreadA ran 'correctly', just the reordering on B would be fatal. If B ran correctly, but A was reordered, it could still be fatal if B happened to run between A1 and A2: ie A1,B1,B2,A2. Actually, looking at the interleaving of the two threads, we can see that there are 4! = 24 different ways it could run, and of those 6 fail. It's surprising that it ever works!

(Why *does* it typically work? In fact, almost never fail? At least 3 reasons:

1. Interleaving is rare, particularly for singletons: there usually isn't that much room for

contention – not many instructions, and not many threads trying to access the Singleton at the same time (for the first time);

2. most Singletons, even lazy Singletons, get initialized early in the program, before any threads are created;

3. Intel x86 CPUs don't do much reordering. In fact, on x86 DCLP *won't* fail. But on other processors it will (and on other Intels – ie Itanium, and who knows what in the future). (Oh, but don't 'fret', there *are* other algorithms that fail even with the limited amount of reordering on x86, this just isn't one of them.)

So you may not have seen DCLP fail, but be assured that it does.)

So what can we do about this? And not just DCLP, but lots of code that makes poor assumptions about threads. How does anything work with threads? Are we doomed? No. In fact, you should realize that

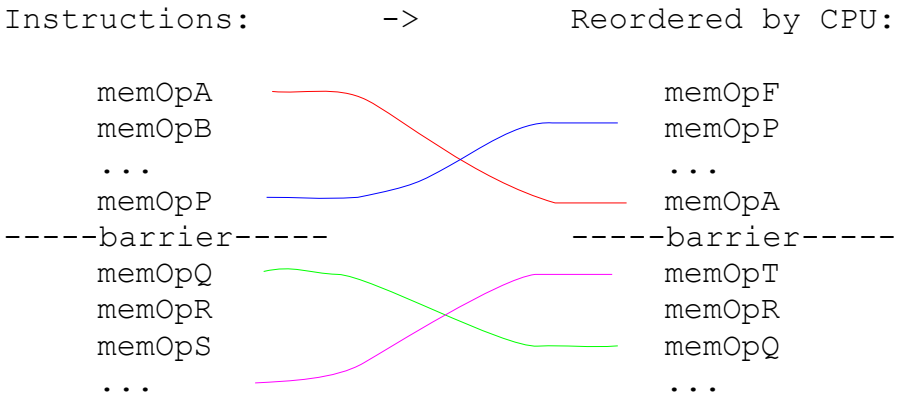
[Slide] Use Locks

using locks *does* work. There are no reordering problems if locks are used (correctly). So locks must be doing some magic to prevent instruction reordering. What's the magic?

Memory Barriers

Although CPUs have been designed to optimize 'locally', they *do* know that they aren't alone – that there are other processors in the machine. So there are special instructions that you can use to limit the reordering of memory operations. These special instructions are called *memory barriers* because they place barriers between memory instructions – barriers that the CPU will not reorder across. By giving developers a way to control instruction reordering *when needed,* the CPU if free to reorder/optimize assumed-to-be single-threaded code (ie most code), while still enabling correct multi-threaded code. The only problem (for C/C++) is that the memory barriers are not part of the language (yet!). Which is one reason why we have different threading libraries on every system – you need different assembly instructions, or OS-specific functions for each platform to use the barriers. (Again, until C++0x arrives).

General Concept of a Memory Barrier:



Memory Operations memOpA,B...P can be reordered amongst themselves, but not <u>after</u> the barrier. They will all be processed before operations Q,R,S... Similarly, Q,R,S... may be reordered amongst themselves, but not <u>before</u> the barrier.

A barrier that prevents any memory operations from moving past it (in either direction) is called a *full barrier*. But we often do not want to be quite so restrictive. As it turns out, there are many useful times (as we will see) where we are only concerned about certain types of memory operations, in certain combinations. So (most) CPU's implement *partial barriers*. We can figure out what these partial barriers are by considering the possible combinations we might wish to confine. There are basically 2 memory operations that we are concerned with – reading memory (a.k.a *load*) and writing memory (a.k.a. s*tore*).

[footnote](There are actually many more memory operations than just load and store, or at least 'specializations' of load and store – ie for memory I/O or for writing past the cache, special memory regions, etc, - and they all get there own barriers, but unless you are writing an OS kernel (see ....txt) you only need to be concerned with read and write, so let's keep it simple...)

So, given 2 arbitrary memory operations that we want to prevent from being reordered, with a memory barrier placed between them; each operation is either a load or a store:

```
     memOpA        // a load (read) or a store (write)
-----barrier-----
     memOpB        // a load (read) or a store (write)
```

=> This implies four possible combinations of what the instructions might be:

loadOp	loadOp	storeOp	storeOp
loadOp	storeOp	loadOp	storeOp

=> And thus at most Four Partial Memory Barriers:

Load|Load - prevents reordering of preceding loads with following loads
Load|Store - prevents reordering of preceding loads with following stores
Store|Store - prevents reordering of preceding stores with following stores
Store|Load - prevents reordering of preceding stores with following loads

(note that they are normally written <u>without</u> the separator | - ie LoadLoad instead of Load|Load, but the separator helps depict the nature of the barrier)

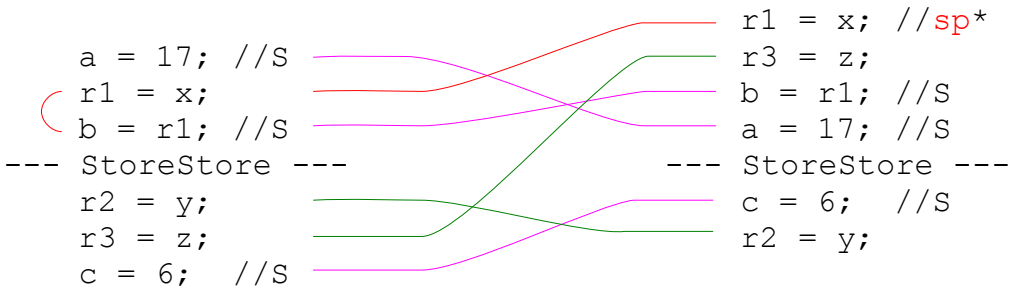
So each barrier describes which type of memory operation must "stay on its own side" (again sounds like kids in the backseat on a car ride!) ***<u>and</u>*** which side it needs to stay on.

It should be mentioned that these are sometimes called Sparc-style barriers, as that is where the terminology first became common.

Let's look at some examples (where r1, r2, r3 are registers (that we load into - ie *not* memory writes), all other variables live in main memory/cache).

In each case, we are given the same initial set of instructions (on the left),
but different barriers result in different reorderings (on the right).
We won't show all possible reorderings (how many would that be!?) but instead one possible reordering, typically favouring 'extreme' reorders whenever possible, for exposition:

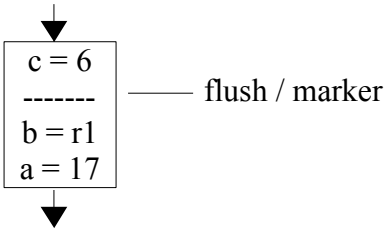
StoreStore
Stores can't cross; Loads are free to move



*"a and b must be **written** before c is **written**"*

- *Operations on a single side still free to reorder*
- *Loads can move freely*
- *Stores can't cross in either direction*
- **sp - single-processor correctness constraint: As r1 = x is a load, it could normally move down, but since b = r1 is a store and needs to stay above the barrier, r1 = x also needs to stay above -- both above the barrier and above b = r1.*

The StoreStore is typically called a *Write Barrier.*



Thinking about the CPU diagram and the write-request-queue you can imagine that the write barrier is implemented by flushing the write-queue (after a and b are added, but before c is added). The writes to a and b can be reordered in the queue, but not after the flush – ie not with the write to c or any writes after that. Similarly the write to c can be reordered with later writes added to the queue, but not with a or b because they have been flushed. (You can also, equivalently imagine just a *marker* or, well, barrier in the queue to mark where the flush would be, and just not reorder across the barrier. Point is, the barrier is typically implemented by how the write-request-queue is handled.)

Uses

enqueuing data

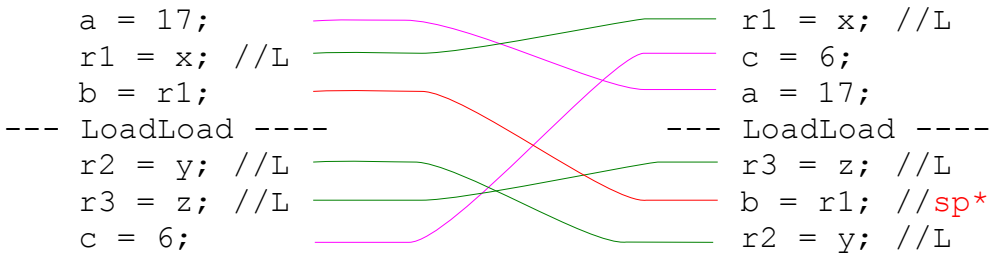
```
data.a = 17;
data.b = x;
#StoreStore  // ensure data is written before queuing
add_to_queue(data); // <- does a write internally
```

initializing singletons

```
temp->a = 17;
temp->b = x;
// ensure data is written before being exposed
#StoreStore
singleton.instance = temp;
```

In both cases, we prep our data locally first, then, *with a barrier,* expose our data to other CPUs.

LoadLoad
Loads can't cross; Stores are free to move



*"x must be **read** before y and z are **read**"*

- *Operations still free to reorder if they stay on one side of the barrier*
- *Stores can move across the barrier in either direction*
- *Loads can't cross at all*
- **sp - single processor constraint -- r1 = x must come before b = r1 to ensure single-threaded program correctness, and thus these 2 instructions are never reordered amongst themselves (also, technically, because no processor currently does 'speculative writes' – ie x must really be known before committing the write to b – we don't speculate!). Being a store, b = r1 can still move past the LoadLoad barrier, however, so it ends up fairly low in our example.*

The LoadLoad is also known as a *Read Barrier.*
Similar to the write barrier, you can imagine that the read barrier is implemented by how the CPU limits reorders inside the read-request-queue. Either by flushing after read(x) is queued and before read(y) and read(z) are queued, or by adding a marker that is in effect equivalent to the flush.
Keep speculative execution in mind when looking at these Uses:

Reading from a queue:

```
if (queue.data_ready) // or wait until ready, etc
#LoadLoad
{
     dequeue_data();  // read data!
}
```

ie a read barrier ensures that the data isn't pre-read before checking the data_ready flag.

Singletons

```
if (singleton != null)
#LoadLoad
{
     read_from_singleton();
}
```

ie Make sure that anything written by singleton's constructor is not pre-read early.
(Note that if you also plan to *write* to the singleton, then LoadLoad is insufficient. For now, let's just worry about reading...)

Note how the read barrier is used as a **mirror** to the write barrier. Both at the higher level of the examples (enqueue vs dequeue, init singleton vs use singleton) and how they are implemented – the order of the instructions. ie

The write does

data -> barrier -> ready_flag, // write, barrier, then *publish*

whereas the read does

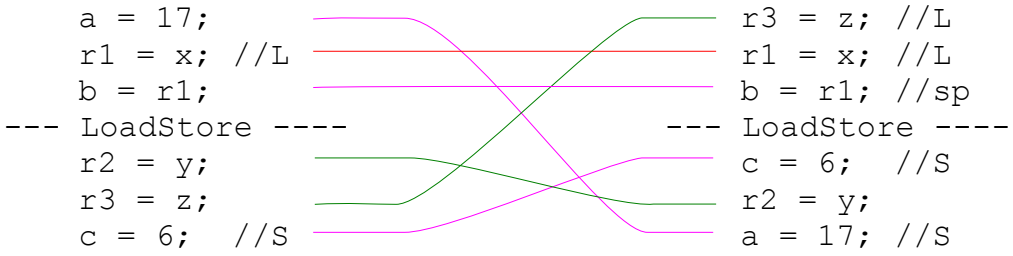
ready_flag -> barrier -> data. // *subscribe*, barrier, then read

It is important to convince yourself that any code that only barriers one side of the 'mirroring' is most likely broken code.

LoadStore

Loads can't move down; Stores can't move up

(each trends to its own side)



*"x must be **read** before c is **written**"*

- *Operations on a single side still free to reorder*
- *Loads can't move down, but **can** move **up***
- *Stores can't move up, but **can** move down*
- **sp* single-processor correctness constraint: r1 = x must come before b = r1 to ensure program correctness*

LoadStore requires a bit more cooperation between the read-request queue and the write queue – the *write* queue needs to wait until the *read* has completed. (Which tends to mean that this is a slower barrier than a read barrier or a write barrier.)

Uses:

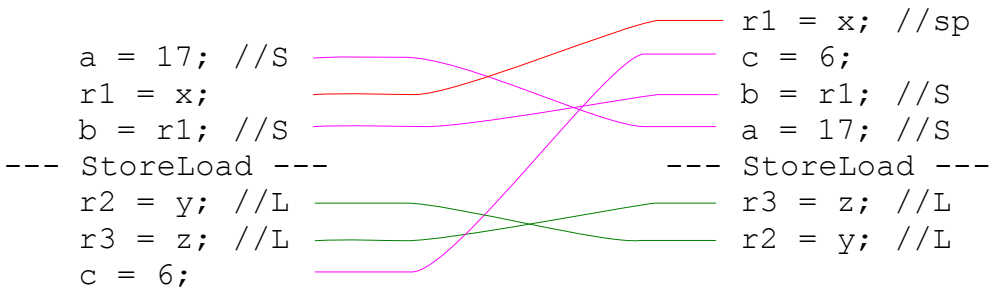
Make sure something is 'ready' before being written:

```
if (destination_ready)     // load
#LoadStore
{
     fill_destination();   // store
}
```

Or, make sure you finish reading before saying you are finished!

```
temp = important.data;
#LoadStore
important.in_use = false; // now free to recycle, etc
```

StoreLoad
Stores can't move down; Loads can't move up
(Stores tend upward; Loads tend downward)



*"a and b must be **written** before y and z are **read**"*

- *Operations on a single side still free to reorder*
- *Loads can move up*
- *Stores can move down*
- **sp* single-processor correctness constraint: again r1 = x is fairly stuck!*

Again, at the implementation level, this requires that one queue wait for the completion of events on the other queue. StoreLoad is, in fact, typically the slowest of the barriers. Also, as it turns out, it is typically the least useful – which might also be *why* it is slow – CPU designers made optimizing it a lower priority than the other barriers.

Uses:

None, really. It is basically useless. How often do you need to 'write this before reading that'? How does the write affect what will be read? Well, maybe the write signals to other threads to "back off" because you are about to read! ie here's something that seems worthwhile:

```
// flag data as 'yours'
// others will need to stay away:
important.lock = true;
#StoreLoad
temp = important.data;
// then release, etc...
```

Well, that actually looks really useful, what's wrong?

1. Well, first of all, it is quite rude – you didn't first *read* `lock` to see if someone else already locked it, so if each thread does that, then we have problems;

2. so you will need to combine it with other barriers; (yes you can combine them! Just wait...)
3. it is typically the slowest barrier;
4. other combinations work better/faster...

Note that many of these examples are starting to look like *locks*. Particularly if we combine the examples:

LoadStore: if (ready) read...
LoadLoad: if (ready) write...

so:

```
if (foo.ready)
#LoadStore
#LoadLoad
{
    // free to both read and write foo
   int x = foo.x;
   foo.a  = 10;
   etc(foo);
}
```

But that's still not quite a lock, because we typically don't just read the lock, we also need to set it. Of course, we will use CAS for that:

```
if (CAS(foo.lock, 0, 1)) // was 0 now 1 means it's ours
#LoadStore
#LoadLoad
{
    // free to both read and write foo
   int x = foo.x;
   foo.a  = 10;
   etc(foo);
}
else
   do_something_else();
```

(Note that this isn't a traditional lock that actually waits for the lock to be available – instead it is more like a try_lock. To do a traditional lock, we would need to wait somehow (ie interact with the scheduler/kernel) and/or spin-wait.)

The interesting trick here is that the CAS is both a load *and* a store, but we only need to order it as one or the other – since it is an atomic operation on its own, the load and the store happen at the same time, and thus it is enough to order it with only Load-first barriers. Similarly, we could have used just Store-first barriers – StoreStore+StoreLoad, but since those are typically slower, we instead favour LoadLoad+LoadStore.

But wait, we are being rude – we forgot to release the lock for the next person:

```
if (CAS(foo.lock, 0, 1)) // was 0 now 1 means it's ours
#LoadLoad
#LoadStore
{
    // free to both read and write foo
   int x = foo.x;
   foo.a  = 10;
   etc(foo);

#LoadStore // make sure all loads are done
#StoreStore // make sure all stores are done
foo.lock = false;
}
```

Notice that the slowest barrier – StoreLoad – wasn't needed at all.

A Different Kind of Company, a Different Kind of Barrier...

As was mentioned briefly when they were first introduced, Load/Store style barriers are often associated with SPARC (and other RISC) processors. Intel, on the other hand, decided to handle memory reordering a bit differently. On Intel, the barriers are not independent instructions, but instead are modifiers on memory reads and writes. Including on the combined read/writes of XCHG and CAS. Each memory operation can be given *acquire semantics* or *release semantics*, resulting in functions like InterlockedCompareExchangeAcquire and InterlockedCompareExchangeRelease.

What do acquire and release mean? From MSDN (http://msdn.microsoft.com/en-us/library/ms684122(VS.85).aspx):

"Acquire memory semantics specify that the memory operation being performed by the current thread will be visible before any other memory operations are attempted. Release memory semantics specify that the memory operation being performed by the current thread will be visible after all other memory operations have been completed."

To explain what that means: First of all, 'visible' is a fancy way of saying 'completed' – it is just a bit more precise. The main concern is whether the result of the operation is visible to every other CPU in the system. Whether a write, for example, has actually been written to main memory is not as important as whether all CPUs will see it as written when they go to read it. For example, if the write has been propagated to all CPU caches, then we don't care whether it has actually been written to main memory yet, as it is already *visible* to all CPUs. Or even if the write was only in one CPU's cache, as long as all other CPUs know that's where the most up to date value is (and that their caches and main memory are currently invalid) then it would still be visible to the other processors – if they need it, they know where to go. Etc. Saying 'visible' allows for flexibility as to how it is actually implemented at the hardware level.

Another way to look at it, which is something important to keep in mind with multi-threaded programming - you can't actually tear open the case of your computer and look at the memory values directly. You can only 'see' them via CPUs, and thus your are always getting a CPU's view of things,

not the 'real' view. So all we care about is how instructions are *visible* to each of the CPUs in the system.

Consider

```
if (CAS_acquire(foo.lock, 0, 1)
{
  int x = foo.x;
  foo.a = 10;
write_release(lock, 0);
}
```

So the acquire ensures that we have read (and written) lock *before* we start messing with the rest of foo. And the write_release(lock, 0) would mean that ALL CPUs, if they chose to read it, would see the lock as 0. And, importantly, that lock' is set *after* instructions on 'x' and 'a' have been completed. Which is exactly what you want for a lock – that you don't start before the data is ready, and that you don't say you are done until you really are done. (And note that those are the typical ways of combining reads and writes with acquires and releases – read_acquire, and write_release. Things like read_release are not typically useful, and possibly not even supported on some platforms.)

Now what should, at this point, be clear, although it seems impossible to find on the internet, is that it is easy to relate Load/Store barriers to Acquire/Release barriers. So, for the first time before a live audience, here's how they relate:

Acquire/Release semantics	As Load/Store barriers
bool locked = Read_WithAcquire(lock);	bool locked = lock; #LoadLoad #LoadStore

Write_WithRelease(lock, false)	#LoadStore
	#StoreStore
	lock = false

It should be noted, however, that Acquire/Release are actually slightly more 'relaxed' than the LoadStore versions. To see that, let's look at an example, in the next section...

An Invasion from Above and Below!

"Real" locks (boost::mutex, POSIX mutex, Windows CriticalSection, etc), like our make-shift examples, may also use Acquire on lock, and Release on unlock (if available, although on some systems a full barrier may be all that is available). Now, consider this typical (if simplified) example of mutex usage:

```
// some code using unshared data,
// thus it can be outside/before the mutex
int x = unshared.x;
unshared.x = 10;
// need to read/write some shared data
// so protect it with a mutex
mutex.lock();
   do_stuff(shared_data);
mutex.unlock();
// some code using unshared data
// thus outside (after) the mutex
int y = unshared.y;
unshared.y = 20;
```

Question: was unshared.x written before unshared.y was read? Was unshared.x written before unshared.y was written?

Given what you now know about memory barriers and reordering, is it possible that y was read before x was written? How much reordering, with respect to unshared, is possible?

Answer: Do you care?

I think that is the important answer (er, question?) - x and y are dealing with unshared data, their order should only be important to this thread, no other (else you'd put them inside the mutex, right!?), so whatever single-processor reordering the CPU (and/or compiler) want to do, it should be OK with you. In other words, you don't care if unshared gets reordered. But just to be clear, here's the actual answer:

We can translate the example into Load and Store barriers:

```
int x = LOAD(unshared.x);
STORE(unshared.x, 10);
CAS_or_wait(mutex.lock, 0, 1);
#LoadLoad
#LoadStore
   LOAD_and_or_STORE(shared_data);
#loadStore
#StoreStore
mutex.lock = 0;
int y = LOAD(unshared.y);
STORE(unshared.y, 20);
```

And then reorder (as much as possible!):

```
CAS_or_wait(mutex.lock, 0, 1);
int x = LOAD(unshared.x);
#LoadLoad
int y = LOAD(unshared.y);
#LoadStore
   LOAD_and_or_STORE(shared_data);
#loadStore
STORE(unshared.x, 10);
#StoreStore
STORE(unshared.y, 20);
mutex.lock = 0;
```

And now untranslate:

```
mutex.lock();
   int x = LOAD(unshared.x);
   int y = LOAD(unshared.y);
   LOAD_and_or_STORE(shared_data);
   STORE(unshared.x, 10);
   STORE(unshared.y, 20);
mutex.lock = 0;
```

We see that instructions outside the lock can move inside the lock, both from above and from below. Of course, instructions *inside* the lock need to stay inside because of how we set up the barriers, and this is what makes locks work as we expect them to, but instructions from outside can move in, and mix with those inner instructions. In this case we set that the write of unshared.x could move down, and pass the read of unshared.y moving up, thus reordering the 2 operations. And that is the most that could happen with the Load/Store version – at that point, each of the instructions were pushed up against the lock barriers, and could go no further. But what about acquire/release instead of Load/Store barriers? Let's look again at the original:

```
int x = unshared.x;
unshared.x = 10;
mutex.lock();   // ACQUIRE
   do_stuff(shared_data);
mutex.unlock();  // RELEASE
int y = unshared.y;
unshared.y = 20;
```

All Acquire says is that the target memory operation will complete before any operations that come *after* it. It doesn't say *anything* about operations *before* it. The difference, w.r.t. LoadLoad+LoadStore, is that Acquire knows exactly which load instruction it is concerned with, whereas LoadLoad+LoadStore doesn't know which load is essential, and thus prevents ALL loads from moving past it.

And of course Release has the mirror opposite case – nothing before the target operation can happen after it, but anything can move before it. Whereas the LoadStore+StoreStore variant needs to keep all stores on the after-side of the barrier. So with acquire/release we can reorder further:

```
mutex.lock();
   int y = unshared.y;
   unshared.y = 20;
   do_stuff(shared_data);
   int x = unshared.x;
   unshared.x = 10;
mutex.unlock();
```

And in this case we see maximal reordering, with all the outer instructions invading the lock-section from both above and below, and intermixing with the lock-sensitve instructions. And, as we asked earlier, yes, unshared.y can be (both read and) written before unshared.x is (read and) written.

So it may be surprising to some, but instructions can move inside areas protected by locks. But again, stick to the original and best answer – Do you care? Typically, if your code isn't making poor assumptions, it shouldn't matter if the unprotected instructions move into the protected section.

And that's about it for reordering and memory barriers. Just a couple of leftover points and/or, for the sake of completion.

1. Just to be clear, besides the specific Load/Store or acquire/release barriers, there is also a *full barrier*. This is a barrier that doesn't let any instructions reorder past it. Ie LoadLoad+StoreStore = Full, and Acquire+Release = Full. On many processors, this is the only barrier available, and thus the barrier used even if a looser barrier would suffice.

2. Barriers are paired. Carefully ordering your memory writes is useless if the next thread doesn't carefully order its reads. As much as we'd like to think that "well we got it written to memory OK" that all that matters, it doesn't mean that the other thread didn't read "done_yet" or "data_ready" too early and saw half-finished data. Unless the reads are barriered along with the writes.

3. You can't open the box and look at the memory. You can only see what the CPUs see. It is very much like Einsteins relativity – everything is relative to the observer (CPU) and *when* things happen (like reads or writes) can no longer be seen as questions with absolute answers, but instead with different answers for each observer.

4. Again, to be clear, many processors do only limited reordering. For example, most Intel chips don't do any Load reordering at all, so a LoadLoad barrier is never needed. Which means that lock-free code is often very processor specific, or at least processor specific via #defines/etc under the hood. Luckily C++0x (to be discussed later) handles much of this nicely for us.

5. CAS / XCHG typically includes barriers. Of course each OS does this differently, with different options and different names. Again, C++0x will clear this up. CAS with barriers explains why I said the rule was:

   > Act Local; CAS Global.

   The actual rule should be more like

   > Act Local; Barrier Global.

But it just doesn't have the same ring to it. Since CAS typically involves barriers, and is often the only way to access barriers (via OS functions at the C/C++ level without dipping to the ASM level), I tend to think of all barrier read/write instructions as being under the CAS 'umbrella'.

6. There are often other types of memory instructions, and thus other types of barriers. For example, writing to device i/o could be different than regular memory, and thus require special instructions and thus special barriers. And write-back vs write-through, instruction vs data caches, etc, etc, etc. Thankfully we usually only need to worry about regular memory. For some of those other concerns, see www.kernel.org/doc/Documentation/memory-barriers.txt , by David Howells and Paul E. McKenney.

7. The *data dependency barrier*. This is also explained in Linux's memory-barriers.txt, but it is scary enough that it has made the Alpha processor (possibly the only processor where it is required) a somewhat legendary CPU in the world of lock-free programming, and thus is worth mentioning here:

Consider the simple operation or dereferencing a pointer:

```
int r1 = *p;
```

Obviously, this statement actually requires 2 reads:

- read p (more precisely: read the memory location where p 'is')
- read *p (interpret the result of the first read as a memory location, and then read there)

In essence, *The Alpha processor can **reorder** these 2 reads.*

Obviously this is “impossible” - how could it 'know' where the second read was to be without reading the first! Actually there are a number of possibilities – for example, “address prediction” similar to branch prediction – ie I've seen this 'p' before, I can guess its value without waiting for the 'real' read to arrive, once it arrives, I can check if I was right. Or by looking into the write-request queue – oh I'm still waiting to write p, so I'll ignore whether another processor also wrote to it, and I'll just get the value from my request queue. Or, once reading p and seeing its value, deciding that it had just read that memory a second ago, so no need to read it again. Etc. It's actually a bit more complicated – it deals with how the Alpha has 2 separate caches for *each* processor (one for odd lines, one for even!), update-queues (update this cache with a value from another processor's cache), and a very loose definition of cache coherency. I'll leave exploring that one as a 'exercise for the reader'. For most of us, we can ignore this peculiarity, at least until some other processor comes along that has reordering constraints as loose as the Alpha.

And lastly, we never completed our DCLP fix. Recall:

```
void expensive_init()
{
   if (!done_yet)
   {
      scope_lock lock(mutex);
      if (!done_yet) // double-check - for race to the mutex
      {
         expensive.important_ptr = new Important_stuff();
         expensive.moredata = etc();
         done_yet = true;
      }
      //unlock in scope_lock destructor
   }
}
```

Hopefully by now you realize that the proper fix would be something like:

```
void expensive_init()
{
   if (!read_acquire(done_yet))
   {
      scope_lock lock(mutex);
      if (!done_yet) // double-check - for race to the mutex
      {
            expensive.important_ptr = new Important_stuff();
            expensive.moredata = etc();
         #StoreStore
         done_yet = true;
      }
      //unlock in scope_lock destructor
   }
}
```

Convince yourself that read_acquire() is the minimum needed for the read. Just a read barrier (LoadLoad) is not enough as we don't know (in general) if the code that eventually uses the singleton will just read, or read and write, so an Acquire or LoadLoad+LoadStore is necessary. However, once the init is finished, we only need a write barrier – we weren't reading any shared data (the singleton isn't shared at all yet), only writing data. Only the writes need to be done before the last write of done_yet, which is the one that 'publishes' to the other threads that writes are completed ('visible') and the singleton is ready for use.

Last question: *Why do we need the #StoreStore at all,when the scope_lock will issue a release barrier when it unlocks the mutex?*

Answer: Since done_yet is read outside the lock, we need the singleton's members written before done_yet is written, otherwise done_yet could be written early, and another thread could read it as true, and thus start using the singleton before it was completely initialized.

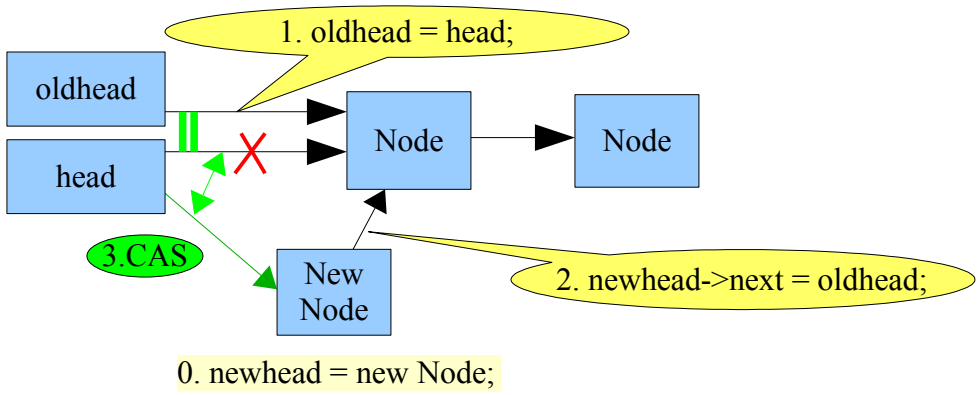
CAS'ing pointers?

Are we done yet with done_yet? Yes. Let's finally move on to lock-free data structures. First a quick note: up to now, we've only CAS'ed ints. But typically, the default word size for CAS is the same as the size of a pointer (if not bigger). So we can also CAS pointers – it is all the same to the CPU – just a bunch of bits. CAS'ing pointers is what opens up the possibility of creating various lock-free data structures – ie stacks, queues, lists, etc.
(Unless otherwise specified, the rest of the examples will assume that CAS() means CAS with the right memory barriers built in (whichever might be necessary for the given situation), so we won't even talk to worry about memory barriers for a while :-)

The Lock-free stack

The 'easiest' lock-free structure is probably the stack. It is also one of the most useful, often lurking underneath other more complicated lock-free structures, so let's start there:

Push a Value onto a Lock-free Stack




```
Void push(Val val)
{
   Node * newhead = new Node(val);    // 0.
   do
   {
      Node * oldhead = stack.head;        // 1.
      // newhead will become head, head will become next;
      // it is important to set newhead->next FIRST before
      // making it the actual new head, because as soon as
      // it is 'published' as such (via the CAS) then any thread
      // can see it, so it better already be in the proper state
      newhead->next = oldhead;    // 2.
   }
   while (!CAS(&stack.head, oldhead, newhead));  // 3.
}
```

Note that steps 0,1,2 are the 'Act Local' part. (Although we do read from stack.head, which is shared, so maybe that should have an acquire barrier.**) Step 3 is the 'CAS Global" or 'publish' part, where we

make our new local and ready-to-go Node a member of the shared stack.

Take a good look, and get comfortable with the lock-free stack push, because that's about it – that's about all you can do with lock free data structures. Or at least all you can do easily. Everything else, even pop(), suddenly becomes much more complicated...

What's wrong with this code?

```
Val pop()
{
   Node * newhead;
   do
   {
      Node * oldhead = stack.head;
      if (!oldhead)
         throw StackEmpty(); // or return special Val, etc

      // oldhead is != null, so we can read from it. Right?
      newhead = oldhead->next;
   }
   while (!CAS(&head, oldhead, newhead));

   Val val = oldhead->val;
   delete oldhead;
   return val;
}
```

Pop has a problem (if not a few). Note the if (!`oldhead`) – and remember *'Are we there yet?'* - any time we have an 'if' statement, we can't assume that the state of things checked by the 'if' will last as long as we need them. In this case it is a bit more subtle. We read 'head' into '`oldhead`' and then only deal with '`oldhead`', so if head changes, we are still OK, right? Using a local `oldhead` is the pattern we've been using elsewhere! Well, in this case, consider - why are we checking for NULL? - because we are about to read the memory that `oldhead` points to (via `oldhead`->next), and we don't want to walk off the memory peer. So yes, the value of `oldhead` won't change, but what about the memory it points to? If another thread calls pop at the same time, head may have been popped, and *deleted*! So `oldhead`->next may be reading deleted memory. Is that bad? It sounds bad.
... but it may or may not be. If the memory has been completely given back to the system, the read may cause a memory fault. If the memory is still 'around' but just marked deleted, then the read will succeed, the value of `oldhead`->next (and thus newhead) will just be garbage. Having newhead be garbage sounds bad (ie if head then gets set to garbage...!), but note that we can only get garbage when concurrent pops are happening – and in that case, head will have been modified by the other threads, CAS will see this modification (ie head != oldhead), fail, and we will retry the loop, hopefully getting a new, non-garbage `oldhead`->next on the new attempt. Thus CAS saves the day again!
...Or does it? What about that bad memory read? Well, we can actually hand-wave that away in a few ways: one is to not delete popped nodes (ie save them for reuse, in, say, a another lock-free stack!), or defer the deletion to a later, safer time (this deferred deletion is a whole field of research on its own). We could also, on most OSes, set up a handler to catch the bad read (which sounds heavy-handed, but the bad reads should be rare, as the memory was just recently used). Lastly, *we may BE the memory allocator* and thus we know when the memory is still around or not. (ie lock-free stacks are often part

of lock-free or 'mostly' lock-free memory allocators).
So, with that bit of hand-waving, CAS *does* save the day. Right? Right?

ABA – our friend CAS has issues...

There is yet another, even subtler issue with pop. What if, after reading head into `oldhead`, head gets popped by another thread, and then *gets pushed back on by a push*!? Is it possible for the same head to be pushed back on? At first blush it doesn't appear so – we allocate a new Node for head inside of push, so it is always a new memory address, right? No, of course not. The memory allocator tends to reuse previously freed memory locations (and/or as mentioned earlier we saved nodes for reuse ourselves to avoid garbage reads). So, yes, we might get a 'new' head that is the same as the old head. And why is this a problem? Well, 2 things: if head is the same as before – ie the same as oldhead – then the CAS will succeed. But, even though head is the same as before, we don't know if head->next is the same as before – anything may have happened since we read 'head' into `oldhead` a few lines ago (always imagine an eternity happens between each line of code). The entire stack may be different, *except* that head happens to be the same as `oldhead`.
So the CAS succeeds, and we just set head to newhead == `oldhead`->next == maybe anything.
BOOM.

This, in general, is called the *ABA problem*. It occurs when a value is being CAS'ed in order to detect external modifications, but the detection fails because the value was A, became B, and then somehow went back to A. In this case, the CAS, looking for "is the value still A" misses that it was temporarily B, and thus gets a false-positive, thinking everything is OK when it really isn't. :-(Does this really happen (ie frequently enough to be concerned, or is it just theoretical)? Yes it really happens. Frequently? Typically at the worst frequency you can imagine: at a frequency where you, as a coder, won't see it happen doing limited testing on your code, but it will happen rarely (and seemingly randomly) for your full time testers, and *definitely* to your users! Thus at one of the worst frequencies available to programmers. :-(

This sounds pretty drastic. It is. We are still just on a simple stack, and already foiled. Remember "Lock-free programming is hard..."? The ABA problem often gets in the way of otherwise lovely plans. One often used solution is DWCAS...

DWCAS to the rescue.

Remember that I mentioned that CAS typically works on int-sized and/or pointer-sized targets? Well, there is a somewhat common variant that works on doublely-wide targets, ie the size of 2 pointers side-by-side. Side-by-side is important. There is a mostly theoretical DCAS (double CAS) or CAS2 that works on 2 targets NOT side by side, but since most CPUs don't implement it, it is typically not used, except in papers requesting it be more broadly implemented :-).
[By the way, the abbreviation DWCAS is not as common as CAS, but I'm pushing for it as a way to disambiguate from the non-side-by-side DCAS.]

So what do we do with the double-wide CAS? [Ask audience for suggestions?]

```
[___counter___|___pointer___]
```

For every pointer we need, the pointer is stored on one side of the double-wide target, and a counter is stored on the other. Every time we mark the pointer for reuse, or every time we allocate, we increment

the counter. Thus if/when ABA happens, what we will see is more like (An)B(Am) or succinctly: ABA' (that's an “A prime” for the second A). Ie the second A will have a different counter than the first A, and the DWCAS will now see the difference and correctly detect that the value has been modified 'behind its back'. (Well, technically, there is still a 1 in 2^32 (for 32 bit wide values) that you get back the exact same A and with the same counter – but that is typically good enough odds for most programmers. Particularly when considering not only did the counter have to land on the exact same value, but the CAS'ing thread had to stall long enough (inside the tight CAS loop) to allow other threads to overflow the counter and bring it back to its initial value. (Of course, a 4 GHz machine could in a sense roll-over a 32 bit int (ie run through 4G of values) in about a second if that was all it was doing (assuming increment was a single cycle instruction). In other words, in the realm of seconds – *if that was all that it was doing*, which, luckily, is typically not the case. But it is something to think about.)

So now our pop() is:

```
Val pop()
{
   NodePtr newhead;
   do
   {
      NodePtr oldhead = stack.head;
      if (!oldhead.pointer)
         throw StackEmpty(); // or return special Val, etc

      // oldhead is != null, so we can read from it. Right?
      newhead = oldhead->next;
   }
   while (!DWCAS(&head, oldhead, newhead));

   Val val = oldhead->val;
   delete oldhead;
   return val;
}
```

(Which is basically the same as before, because, like all good C++ programmers, I've wrapped the details of the {counter+pointer} into a class, overridden operator->(), etc. I left the DWCAS in there (instead of calling a NodePtr member function) since that is what the example was meant to show.)

Speaking of DWCAS and {counter+pointer}, it might occur to some that a reference-counter pointer (ie boost::shared_ptr or instrusive_ptr) might make use of DWCAS. That is definitely possible. For now, it is another 'exercise for the reader'.

Conclusions

[Slide] “Use locks”

We can't prevent programmers from being curious. So some programmers will go out and investigate lockfree programming no matter how much discouragement is given. Instead of just ignoring this I thought it made sense to provide

- scary information
- correct information

(Because scary is correct – it is not an exaggeration!)
And if we do not provide correct information, then the adventurous is left with incorrect, or worse, only *mostly* correct information, making all our lives more difficult. So 'use with caution' correct information appears to be the right approach. I call this 'FCD' – Fear, Certainty, and Doubt.

P.S. One last scare tactic:

[Slide] I'm not sure everything I said is correct.

As I said, I'm not an expert. Mostly I'm just passing along the information I've gathered from other experts. So I may have made some mistakes. Lock-free is that way. It requires peer-review. This doc is a bit of a stake in the ground that can be used as a reference, and that I will correct as mistakes are found. (So wait for the .1 release – isn't that always the case?...)

Oh, and the experts make mistakes as well. I've seen it, you've seen it, they've seen it.

Happy Coding!

Addendum - Lock Calculus – An Alternative Definition of Lock-free Programming

When using locks, the pattern (obviously) is obtain the lock, hold the lock while doing some work, then release the lock. Let's call the length of time, or amount of code/instructions executed while holding the lock the *lock-length*.
Now, as anyone familiar with threaded programming knows, we try to minimize the amount of work done while holding the lock – ie minimize the lock-length – by only doing work that actually needs the lock while inside the lock. Anything that can be done outside the lock is done outside. But this often means convoluting the code in order to move steps (instructions, function calls) outside the lock, which can be error-prone. So we often leave unnecessary steps inside the lock-window, to avoid convolution, and to keep the code more clear and maintainable. The downside is that firstly, the code is less concurrent than it could be, and secondly, one of those steps - function calls in particular - may (now or in the future as the code changes over time) grab another lock – which can quickly lead to hidden chances of deadlocks.
So more code inside the lock increases the chance of deadlocks, which is a very common type of

threading bug. But the other common threading bug is finding code that *should be* done inside a lock, but isn't. In fact, this is often how the lock gets added to begin with – when someone realizes that 2 pieces of code were accessing an object at the same time, and making the object inconsistent (if not crashing altogether). So we often find ourselves adding code to lock-windows, increasing the lock-length, to make the code *correct*, and eliminate logic errors.

Thus we are left with conflicting tensions, pulling at us in different directions, to increase or decrease the lock-length of the locks in our programs.

Now let's look at the big picture of this, and do some "lock calculus". Let's look at the limits of these tensions, the limits of the lock length – ie on one hand, what happens when we allow the lock-length to increase to a maximum, and on the other, what happens when we attempt to shrink the lock-length to a minimum?

lim P = sequential lock→∞ + no Deadlocks! + no livelocks + minimizes logic errors − poor use of resources	*lim* P = lock-free lock→0 + no Deadlocks! − chance of livelocks − maximizes logic errors + good use of resources

(Where P is your program, and 'inifinity' is really the maximum lock-length - complete length of the program.)

I would posit that rate of change for most of these characteristics is each a monotonic function – ie the chance of a logic error steadily increases as the lock-length goes from 'inifinity' down to 0. But what about deadlocks? They seem to go from 0 to 0? Without any actual measurements, I would imagine that the deadlock/locklength curve looks something like the Gaussian curve – approaching zero on either side, but with a big lump in the middle – ie medium size lock-lengths (for some definition of medium) tend to have the highest likelihood of deadlocks. Unfortunately, this may also be the typical lock-length found in most programs. (Alternatively, maybe the curve is lop-sided towards longer lock-lengths, rising as lock-length increases, but with a sudden downward denouement as the lock-length subsumes the entire program.)

1 == 0. What happens as lock-length approaches 0? Well, *at* 0 would mean no instructions inside the lock – ie that the lock wasn't necessary at all. Which probably means that either only one thread was involved, or that there were multiple threads, but they were not sharing any data. These are not interesting cases for us. But as we *approach* 0 we get to the interesting minimum lock-length of 1. What does lock-length == 1 mean? It means you have shrunk your lock window down to one instruction – not one line of code, but one CPU instruction. Now, if it is not just any instruction, but one of a small set of special instructions – ie memory barriers and the CAS family of instructions. Once your lock-length is down to a single special instruction, well, then you no longer need the lock at all. Thus in this case, lock-length == 1 becomes lock-length == 0. This is, obviously, as the name implies, "lock-free" programming. But in a deeper sense, it is my definition of lock-free programming – the limit of programming as lock-length approaches 0.