Preshing on Programming

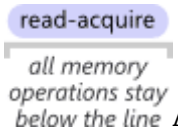
Sep 13, 2012

Acquire and Release Semantics

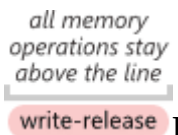
Generally speaking, in lock-free programming, there are two ways in which threads can manipulate shared memory: They can compete with each other for a resource, or they can pass information co-operatively from one thread to another. Acquire and release semantics are crucial for the latter: reliable passing of information between threads. In fact, I would venture to guess that incorrect or missing acquire and release semantics is the #1 type of lock-free programming error.

In this post, I'll demonstrate various ways to achieve acquire and release semantics in C++. I'll touch upon the C++11 atomic library standard in an introductory way, so you don't already need to know it. And to be clear from the start, the information here pertains to lock-free programming *without* sequential consistency. We're dealing directly with memory ordering in a multicore or multiprocessor environment.

Unfortunately, the terms *acquire and release semantics* appear to be in even worse shape than the term *lock-free*, in that the more you scour the web, the more seemingly contradictory definitions you'll find. Bruce Dawson offers a couple of good definitions (credited to Herb Sutter) about halfway through this white paper. I'd like to offer a couple of definitions of my own, staying close to the principles behind C++11 atomics:

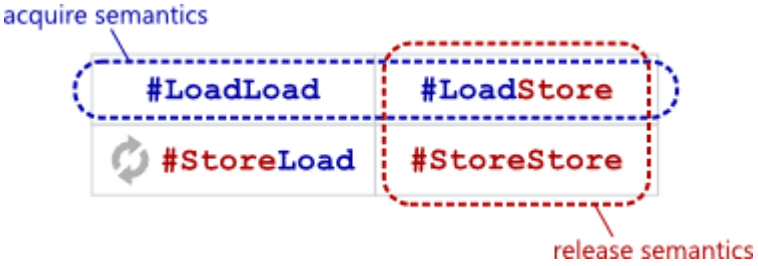


Acquire semantics is a property that can only apply to operations that **read** from shared memory, whether they are read-modify-write operations or plain loads. The operation is then considered a **read-acquire**. Acquire semantics prevent memory reordering of the read-acquire with any read or write operation that **follows** it in program order.



Release semantics is a property that can only apply to operations that **write** to shared memory, whether they are read-modify-write operations or plain stores. The operation is then considered a **write-release**. Release semantics prevent memory reordering of the write-release with any read or write operation that **precedes** it in program order.

Once you digest the above definitions, it's not hard to see that acquire and release semantics can be achieved using simple combinations of the memory barrier types I described at length in my previous post. The barriers must (somehow) be placed *after* the read-acquire operation, but *before* the write-release. *[Update: Please note that these barriers are technically more strict than what's required for acquire and release semantics on a single memory operation, but they do achieve the desired effect.]*



What's cool is that neither acquire nor release semantics requires the use of a `#StoreLoad` barrier, which is often a more expensive memory barrier type. For example, on PowerPC, the `lwsync` (short for "lightweight sync") instruction acts as all three `#LoadLoad`, `#LoadStore` and `#StoreStore` barriers at the same time, yet is less expensive than the `sync` instruction, which includes a `#StoreLoad` barrier.

With Explicit Platform-Specific Fence Instructions

One way to obtain the desired memory barriers is by issuing explicit fence instructions. Let's start with a simple example. Suppose we're coding for PowerPC, and `__lwsync()` is a compiler intrinsic function that emits the `lwsync` instruction. Since `lwsync` provides so many barrier types, we can use it in the following code to establish either acquire or release semantics as needed. In Thread 1, the store to `Ready` turns into a write-release, and in Thread 2, the load from `Ready` becomes a read-acquire.

```
// Shared global variables
int A = 0;
int Ready = 0;
```

Thread 1

```
A = 42;
__lwsync();
Ready = 1;
```

#LoadStore + #StoreStore keeps all memory operations above the line

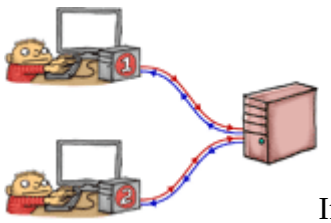
This becomes the write-release

Thread 2

```
int r1 = Ready;
__lwsync();
int r2 = A;
```

This becomes the read-acquire

#LoadLoad + #LoadStore keeps all memory operations below the line



If we let both threads run and find that `r1 == 1`, that serves as confirmation that the value of `A` assigned in Thread 1 was passed successfully to Thread 2. As such, we are guaranteed that `r2 == 42`. In my previous post, I already gave a lengthy analogy for `#LoadLoad` and `#StoreStore` to illustrate how this works, so I won't rehash that explanation here.

In formal terms, we say that the store to `Ready` *synchronized-with* the load. I've written a separate post about *synchronizes-with* here. For now, suffice to say that for this technique to work in general, the acquire and release semantics must apply to the same variable – in this case, `Ready` – and both the load and store must be atomic operations. Here, `Ready` is a simple aligned `int`, so the operations are already atomic on PowerPC.

With Fences in Portable C++11

The above example is compiler- and processor-specific. One approach for supporting multiple platforms is to convert the code to C++11. All C++11 identifiers exist in the `std` namespace, so to keep the following examples brief, let's assume the statement `using namespace std;` was placed somewhere earlier in the code.

C++11's atomic library standard defines a portable function `atomic_thread_fence()` that takes a single argument to specify the type of fence. There are several possible values for this argument, but the values we're most interested in here are `memory_order_acquire` and `memory_order_release`. We'll use this function in place of `__lwsync()`.

There's one more change to make before this example is complete. On PowerPC, we knew that both operations on `Ready` were atomic, but we can't make that assumption about every platform. To ensure atomicity on all platforms, we'll change the type of `Ready` from `int` to `atomic<int>`. I know, it's kind of a silly change, considering that aligned loads and stores of `int` are already atomic on every modern CPU that exists today. I'll write more about this in the post on *synchronizes-with*, but for now, let's do it for the warm fuzzy feeling of 100% correctness in theory. No changes to `A` are necessary.

```
// Shared global variables
int A = 0;
atomic<int> Ready(0);
```

Thread 1

```
A = 42;
atomic_thread_fence(memory_order_release);
Ready.store(1, memory_order_relaxed);
```

Thread 2

```
int r1 = Ready.load(memory_order_relaxed);
atomic_thread_fence(memory_order_acquire);
int r2 = A;
```

The `memory_order_relaxed` arguments above mean "ensure these operations are atomic, but don't impose any ordering constraints/memory barriers that aren't already there."

Once again, both of the above `atomic_thread_fence()` calls can be (and hopefully are) implemented as `lwsync` on PowerPC. Similarly, they could both emit a `dmb` instruction on ARM, which I believe is at least as effective as PowerPC's `lwsync`. On x86/64, both `atomic_thread_fence()` calls can simply be implemented as compiler barriers, since *usually*, every load on x86/64 already implies acquire semantics and every store implies release semantics. This is why x86/64 is often said to be strongly ordered.

Without Fences in Portable C++11

In C++11, it's possible to achieve acquire and release semantics on `Ready` without issuing explicit fence instructions. You just need to specify memory ordering constraints directly on the operations on `Ready`:

Thread 1

```
A = 42;
Ready.store(1, memory_order_release);
```

Thread 2

```
int r1 = Ready.load(memory_order_acquire);
int r2 = A;
```

Think of it as rolling each fence instruction into the operations on `Ready` themselves. *[Update: Please note that this form is not exactly the same as the version using standalone fences; technically, it's less strict.]* The compiler will emit any instructions necessary to obtain the required barrier effects. In particular, on Itanium, each operation can be easily implemented as a single instruction: `ld.acq` and `st.rel`. Just as before, `r1 == 1` indicates a *synchronizes-with* relationship, serving as confirmation that `r2 == 42`.

This is actually the preferred way to express acquire and release semantics in C++11. In fact, the `atomic_thread_fence()` function used in the previous example was added relatively late in the creation of the standard.

Acquire and Release While Locking

As you can see, none of the examples in this post took advantage of the `#LoadStore` barriers provided by acquire and release semantics. Really, only the `#LoadLoad` and `#StoreStore` parts were necessary. That's just because in this post, I chose a simple example to let us focus on API and syntax.

One case in which the `#LoadStore` part becomes essential is when using acquire and release semantics to implement a (mutex) lock. In fact, this is where the names come from: acquiring a lock implies acquire semantics, while releasing a lock implies release semantics! All the memory operations in between are contained inside a nice little barrier sandwich, preventing any undesireable memory reordering across the boundaries.

```
pthread_mutex_lock(&mutex);
    all memory
    operations stay
    between the lines
pthread_mutex_unlock(&mutex);
```

Here, acquire and release semantics ensure that all modifications made while holding the lock will propagate fully to the next thread that obtains the lock. Every implementation of a lock, even one you roll on your own, should provide these guarantees. Again, it's all about passing information reliably between threads, especially in a multicore or multiprocessor environment.

In a followup post, I'll show a working demonstration of C++11 code, running on real hardware, which can be plainly observed to break if acquire and release semantics are not used.

« Memory Barriers Are Like Source Control Operations Weak vs. Strong Memory Models »

Comments (50)

Commenting Disabled
Further commenting on this page has been disabled by the blog admin.

tobi · *558 weeks ago*

"usually, every load on x86/64 already implies acquire semantics and every store implies release semantics."

Wouldn't this mean that on x86/64 the only possible reordering is a st,ld pair becoming ld,st?

So loads cannot reorder with other loads at all? The same for stores?

Reply **4 replies** · *active 259 weeks ago*

Jeff Preshing · *558 weeks ago*

Yep, that's right... usually! It's documented in Volume 3, Section 8.2.3 of Intel's x86/64 Architecture Specification. I haven't pored through AMD's specification but my understanding from other people on the web is that it's more or less the same. That's why x86/64 is often said to be strongly ordered. As you mention, StoreLoad is usually the only kind of reordering which can occur, so that's the type I demonstrated in an earlier post.

Having said that, the strong ordering guarantees of x86/64 go out the window when you do certain things, which are also documented in the same section of Intel's docs:

Marking memory memory as non-cacheable write-combined (for example using VirtualProtect on Windows or mmap on Linux); something only driver developers normally do.
Using fancy SSE instructions like movntdq or "string" instructions like rep movs. If you use those, you lose StoreStore ordering on the processor and can only get it back using an sfence instruction. To be honest, I often wonder if there's any risk of a compiler using those instructions when optimizing lock-free code. Personally, I haven't seen it happen yet.

Reply

tobi · *558 weeks ago*

Very interesting, thank you.

Reply

Travis Downs · *259 weeks ago*

There is a second type of reordering that isn't covered by the "four possible" reorderings like StoreLoad, and which is allowed on x86: CPUs are allowed to see *their own stores* out of order with respect to the stores from other CPUs, and you can't explain this by simple store-load reordering.

This is explained in the Intel manual with statements like "Any two CPUs *other than those performing the stores* see stores in a consistent order". The underlying hardware reason is store-forwarding: a CPU may consume its own stores from the store buffer, long before those stores have become globally visible, resulting in those stores appearing "earlier" to that CPU than to all the other CPUs.

This is often glossed over by people that say x86 only exhibits "StoreLoad" reordering: in fact it has StoreLoad reordering plus "SLF reordering" (SLF being store-to-load forwarding). SLF doesn't fit cleanly in into that 2x2 matrix of reorderings, it kind of has to be described explicitly.

Reply

Bruce Dawson · *557 weeks ago*

Tobi, when observing that the reordering restrictions on x86/x64 it is always important to remember Jeff's warning that the *compiler* may still rearrange things, so memory barriers are still needed to keep the compiler behaving. In this case the memory barriers needn't map to an instruction -- just a directive to the compiler.

Reply

Bruce Dawson · *557 weeks ago*

Jeff, excellent work as always. I like the diagram showing the "do not cross" lines.

One nitpick: you say "aligned int is already atomic on every modern CPU". I know what you *mean*, but a type cannot be atomic -- only an operation can be. Read from or write to an aligned int is atomic on every modern CPU. Increment, for instance, is not (as you know).

Reply **1 reply** · *active 505 weeks ago*

Jeff Preshing · *557 weeks ago*

Thanks for the precision, Bruce. I revised the text.

Reply

John Bartholomew · *523 weeks ago*

In your release example in "With Fences in Portable C++11":

```
A = 42;
atomic_thread_fence(memory_order_release);
Ready.store(1, memory_order_relaxed);
```

You've stated that a release fence prevents memory operations moving down below it. What prevents the store_relaxed(&ready, 1); operation from moving up above the release fence (and therefore potentially above the A = 42; assignemnt)?

Reply **9 replies** · *active 463 weeks ago*

Jeff Preshing · *523 weeks ago*

That's a good question, and an important one.

To be precise: The release fence doesn't prevent memory operations moving down below itself. I was careful not to state it this way. (And I've since learned it's a very common misconception.) The role of a release fence, as defined by the C++11 standard, is to prevent previous memory operations from moving past **subsequent stores**. I should revise the post to state that more explicitly. (In the current draft, I really only implied it by calling it a #LoadStore + #StoreStore fence and linking to a previous post which defines those terms.)

Section 29.8.2 of working draft N3337 of the C++11 standard guarantees that if r1 = 1 in this example, then the two fences synchronize-with each other, and therefore r2 must equal 42. If the relaxed store was allowed to move up above A = 42, it would contradict the C++11 standard.

Reply

John Bartholomew · *523 weeks ago*

Ok, that makes more sense. Thanks for the clarification.

Reply

Jeff Preshing · *523 weeks ago*

Out of curiosity, did you ask this question because of 1:10:35 - 1:11:01 in Herb Sutter's atomic Weapons talk, part 1?

Reply

John Bartholomew · *523 weeks ago*

Indeed I did. Could be a source of confusion for others, too.

Reply

Jeff Preshing · *523 weeks ago*

Yeah, that's a problem.

I sent him an e-mail two weeks ago to clarify this part of the talk, but didn't hear back. Maybe I used an old address. I'm thinking I should do a dedicated post on it.

Reply

ilimpo · *474 weeks ago*

Have you seen the part 2 as well?
On page 54 of the slides, it says that the exchange operation can't be "relaxed".
Is this correct? From my understanding, the object creation part won't be moved upward since it depends on the outcome of the IF statement purely.
Btw, the slides link is available here http://channel9.msdn.com/Shows/Going+Deep/Cpp-and...

Reply

ilimpo · *463 weeks ago*

Hi Jeff,
Would you be able to confirm on my understanding above?

Thanks.

Reply

Herb Sutter · *508 weeks ago*

Thanks to Dlip Ranganathan for bringing this thread to my attention today... I didn't see it and I don't remember getting email.

Yes, this is a bug in my presentation (the words more than the actual slide). The example is fine but I should fix the description of "if this was a release fence." In particular:

- starting at 1:10:30, I was incorrect to say that a release fence has a correctness problem because it allows stores to float up (it does not, as noted the rule is in 29.8.2; thanks!) – what I should have said was that it's a still a performance pessimization because the fence is not associated with THAT intended store, but since we don't know which following store it has to pessimistically apply to ALL ensuing ordinary stores until the next applicable special memory operation synchronization point – it pushes them all down and often doesn't need to

- starting at 1:13:00, my ensuing discussion of the pessimizations are correct, and gives a related example caused by a similar effect of a full fence on ensuing stores – note that the 1:13:00 example is about a full barrier, but I believe it also applies to the corrected version of the release fence case mentioned above if the fence in Thread 1 was a release (and analogously acq/rel for the fences in Thread 2)

Thanks!

Reply

Jeff Preshing · *508 weeks ago*

Thanks for acknowledging the error, Herb. I'm relieved to know that we're on the same page about that. After watching the video, I had to triple-check the standard to feel sure!

I also see your point about the possibility of compiler pessimization. Thanks for emphasizing that. My personal interest in lock-free programming is to eke the most performance out of multicore devices, so I care about that quite a bit. Currently, my gut feeling is that it's still possible to implement data structures that are relatively free from such pessimizations -- I'll be sharing examples on this blog, and will definitely be on the lookout for such pessimizations.

Reply

John Bartholomew · *523 weeks ago*

I think I have misunderstood something. If you have:

A = 42;
#StoreStore
atomic_store_relaxed(&ready, 1);

Then that seems to be a different guarantee than:

A = 42;
release_fence(); // memory operations can move up but not down past this fence
atomic_store_relaxed(&ready, 1);

The first seems to provide the necessary guarantee (that the result becomes visible before the ready flag becomes visible), while the second seems to provide practically no guarantees at all.

Perhaps I am misunderstanding "keeps all memory operations above the line"?

Reply **3 replies** · *active 255 weeks ago*

 Jeff Preshing · *523 weeks ago*

Hi John, I've replied to your previous comment above. Hope it helps.

In the fence examples, the store occurs immediately after the fence, so it's possible to describe the ordering guarantees by drawing a single line above the store. (If say, the example had some loads between the fence and the store, it wouldn't be so simple as drawing a single line -- a release fence only guarantees that memory operations before itself won't be reordered with the next store.)

Reply

 Henry · *255 weeks ago*

Hi Jeff,

In the example, your article mentions, "On x86/64, both atomic_thread_fence() calls can simply be implemented as compiler barriers". However, your reply states, "a release fence only guarantees that memory operations before itself won't be reordered with the next store". Without a hardware fence or a special instruction, it seems the CPU can still reorder at runtime. No?

And according to Herb's post (http://preshing.com/20120913/acquire-and-release-semantics/#IDComment721195803) -- "since we don’t know which following store it has to pessimistically apply to ALL ensuing ordinary stores until the next applicable special memory operation synchronization point – it pushes them all down and often doesn’t need to". It seems to suggest that it does more than just a compiler barrier?

Or actually, you guys are still talking about the instruction reordering at compiler time not runtime?

Thanks!

Reply

 preshing · *255 weeks ago*

Normally, the x86/64 doesn't reorder reads and doesn't reorder writes at the hardware level (among other things). That's why a compiler barrier is sufficient to implement those examples on x86/64. See Weak vs. Strong Memory Models.

Reply

 Kjell · *516 weeks ago*

Thanks for a really interesting blog posts about barriers and lock-free programming.

It seems like one can do a lot without using a full memory barrier on x86 since it has such a strong memory model. Is it sometimes good for performance reasons to issue a full memory barrier even if it is not necessary for correctness? The scenarios I'm thinking about is illustrated in the following C code for x86:

```
volatile int x = 1;

void store1(){
x = 0;
//Other threads can see the previous store now or later
}

void store2(){
x = 0;
FULL_MEMORY_BARRIER();
//Other threads can see the previous store
}

//Will store1() or store2() make the store visible to other threads faster than the other?

void wait1(){
do{
}while(x)
}

void wait2(){
do{
FULL_MEMORY_BARRIER();
}while(x)
}

//Will one of wait1() and wait2() notice a store to x faster than the other?
```

Reply

Herb Sutter · *508 weeks ago*

Let me dispute the claim that there's no need for #StoreLoad. :)

I realize you added the disclaimer "without sequential consistency (SC)" which does remove the need for #StoreLoad, but that's a subtle and potentially misleading statement in that I'm not sure people will necessarily understand what that implies. For example, it means certain classes of lock-free algorithms will not work -- they will compile and appear to work, but fail intermittently. The class is "IRIW" or "independent reads of independent writes" examples like Dekker's and Peterson's algorithms.

In particular, SC-DRF (SC as long as the program is free of data races) is the Java and default C++ standard model, and the weakest model even expert lock-free developers can deal with directly (yes, a few still dispute that, but they have not been persuasive).

What this article describes as acquire/release is what I called "pure acquire/release" in my talk. [1] It's what ld.acq and st.rel implement for Itanium, for example. And yes, it does not require #StoreLoad, but neither does it support "your code does what it looks like from reading the source," a.k.a. SC(-DRF), and lock-free code like Dekker's and others will be silently broken.

My opinion, which some still dispute but more are increasingly agreeing with at least in the mainstream with moderate core counts, is that #StoreLoad is not optional for usability by programmers in general, and hardware that is inefficient by requiring expensive fences on loads will gradually disappear or not scale.

Notably, as I mention in my talk, ARM v8 adds explicit "SC load acquire" and "SC store release" instructions. This is no accident; it's

exactly what hardware should be optimizing for, because it's now what mainstream software is specified for and written against.

[1] http://channel9.msdn.com/Shows/Going+Deep/Cpp-and...

Reply **3 replies** · *active 263 weeks ago*

Jeff Preshing · *506 weeks ago*

Hi Herb!

I'm excited that you've joined the discussion on my blog. And I understand that you are a big proponent of the SC-DRF programming model, which in my view is a somewhat higher-level model of lock-free programming.

Having said that, I feel that it's a little misleading to say that "certain classes of lock-free algorithms will not work" (without sequential consistency). Which class of algorithms? Only the incorrect ones, really! The correct ones will always work. Therefore, the trick is to write correct code. Part of the challenge is the shortage of clear information about low-level lock-free programming. This blog tries to help with that a bit.

You make an interesting point that there's a difference between "SC acquire" and "pure acquire". I reviewed this part of your atomic Weapons talk, and gave it a lot of thought. For anyone following along, this point is made between 42:30 - 45:22 in part 1. The main drawback you mention of "pure" acquire/release is that, if a spinlock release were reordered against a subsequent spinlock acquire, it could introduce deadlock (or rather, livelock). This would obviously be a terrible thing, but it really strikes me as a specific issue with spinlocks.

Nonetheless, to prevent such livelock, I don't think a #StoreLoad barrier is necessary at the processor level. A compiler barrier would be sufficient; we just need to prevent the compiler from delaying the write-release past the entire read-acquire loop which follows.

Reply

James · *263 weeks ago*

Hi Jeff and Herb,

Thanks for the excellent posts and discussions!

I am still trying to understand acquire/release semantics in C++, especially how it works in building a lock-free linked list. Suppose we have two threads (A and B) trying to add new nodes to a shared linked list. Am I right that "pure release" (LoadStore and StoreStore) and "pure acquire" (LoadLoad and LoadStore) are not sufficient to guarantee correctness without a StoreLoad in the following scenario?

Let's say both threads are running on two different cores. Firstly thread A inserts a new node successfully using head.compare_exchange_weak(new_node.ptr->next,new_node, std::memory_order_release, std::memory_order_relaxed), then thread B tries to add its own new node by doing a head.load(std::memory_order_acquire) followed by head.compare_exchange_weak(new_node.ptr->next,new_node, std::memory_order_release, std::memory_order_relaxed). Without StoreLoad in between, thread B may still see the old head and then add a new node to point to it, thus corrupt the data structure. Am I correct here or missed anything? Thanks!

Reply

preshing · *263 weeks ago*

I don't think your code will corrupt the data structure. Note that compare_exchange_weak() is a read-modify-write operation. It always "sees" the latest head. See §32.4.11 in the standard: "Atomic read-modify-write operations shall always read the last value (in the modification order) written before the write associated with the read-modify-write operation."

Acquire and release have nothing to do with that point! Acquire and release would only be used, in your example, to ensure that the contents of memory *pointed to* by head are made visible across threads, which is a different matter.

To complete your example (so that thread B actually inserts successfully), note that thread B should check the return value of compare_exchange_weak(), because the head may have been changed (by another thread) since the preceding load(). If the compare_exchange_week() failed, you just need to update new_node.ptr->next, then attempt the compare_exchange_weak() again (with a different "expected" argument). Repeat until it succeeds.

Reply

 vineelkumarreddy · *495 weeks ago*

When software tools do you use to prepare the illustrations? They look really good. And thanks for great articles

Reply **1 reply** · *active 495 weeks ago*

 preshing · *495 weeks ago*

Mostly Inkscape.

Reply

 Ravi · *475 weeks ago*

I am a beginner in concurrency. I didn't understand why is it guaranteed that r2 will always end up with 42. If Thread 2 executes completely before Thread 1, wouldn't r2 == 0?

Reply

 Ramesh · *463 weeks ago*

Hi,

One thing that was not clear to me is the scope of the Acquire / Release orderings. For example the call to Ready.store(release) in thread T1 ensures that other Writes to Other Variables e.g. variable A is visible in thread T2 following the Ready.load(acquire) operation.

If the call to Acquire / Release were to occur in a method with 4 different scopes then do the Writes in scopes prior to the Ready.store(release) also become visible post Ready.load(acquire).

Reply **1 reply** · *active 437 weeks ago*

 preshing · *463 weeks ago*

Yes. Everything the thread did before Ready.store(release) becomes visible.

Reply

 Scott Peterson · *437 weeks ago*

Hi!
I was wondering if there are any valid cases where an acquire or release barrier appears without a matching pair.
I've asked this question on stackoverflow (http://stackoverflow.com/q/27792476/4414075). It would be really helpful if you could provide some insight, either here or on SO.

Thanks

Reply **1 reply** · *active 434 weeks ago*

preshing · *437 weeks ago*

When using C++11 atomics, there are no such valid cases. A release operation/fence must always be combined with an acquire or consume operation/fence, because the standard says so.

If you're programming the low-level operations yourself, which is unlikely, then there is some possibility that you can omit barrier instructions depending on your exact model of processor. Check section 4.7 of this paper for some examples which distinguish between several POWER and ARM models. Personally, I don't see the value in going that route. The C++11 memory model hits the sweet spot, in my opinion.

Reply

Jens · *420 weeks ago*

Hi,

I am wondering in which cases standalone fences would be useful. In the chapter "Without Fences in Portable C++11" you say that "This is actually the preferred way to express acquire and release semantics in C++11". Why are standalone fences useful then? Could you provide an example for such a use case which couldn't be solved with atomic operations alone?

Thanks and best regards!

Reply

hashb · *387 weeks ago*

Great article. I hope you will write a book about c++11 concurrency

Reply

Tamas · *369 weeks ago*

Hello Jeff!

Is it true that if a load to a NON-atomic location M happens-before a store to M, than the load cannot observe the effect of the store? I cannot surely prove this from the C++11/14 standard.

I know that if a store happens-before a load of the same non-atomic object (and there are no more stores to the object), than the store will be a visible-side-effect when the value of the load is calculated. But is it true if the second operation is a store? I am confused because the second store does not use the original value of the object, so maybe it can be done before the first store.

If "B" synchronizes with "C" than is it possible that "out" will be 24 (D gets reordered before A)? Your illustrations suggest that acquire makes all reads and writes "stay below itself", but from the standard I only get that it makes reads "stay below itself". Please help me prove it from the standard.
I see that A happens-before D, so they cannot introduce a data-race.

```
int data;
atomic<bool> hasData(false);

// assume hasData == true, && data == 42

thread1:
int out = data;//A
hasData.store(false, std::memory_order_release); //B
```

thread2:
while(hasData.load(std::memory_order_acquire)); //C
data = 24;//D
hasData.store(true, std::memory_order_release);

Thanks, and sorry for the long comment.

Reply **1 reply** · *active 369 weeks ago*

preshing · *369 weeks ago*

Take a look at §1.10.15 in N4296: "...operations on ordinary objects are not visibly reordered."

Reply

Igor · *361 weeks ago*

Hello Jeff!

You wrote: "To ensure atomicity on all platforms, we'll change the type of Ready from int to atomic<int>. ... No changes to A are necessary."

However, the "A=42;" and "int r2=A;" statements can be executed at the same time by two threads, and so r2 may end up equal to neither 42 nor 0 due to the data race (or something worse can happen because of undefined behavior). So A should be atomic<int> as well according to your statement in the [Atomic vs. Non-Atomic Operations] article: "Any time two threads operate on a shared variable concurrently, and one of those operations performs a write, both threads must use atomic operations.".

Is this really a mistake in the article, or am I misunderstanding something?
Thanks.

Reply **1 reply** · *active 361 weeks ago*

preshing · *361 weeks ago*

Perhaps I tried to simplify the example too much here, but my point was that *if* we end up with r1 == 1, then we are guaranteed that r2 == 42. If r1 == 1, there couldn't have been a data race.

If r1 != 1, then you are right that there was a possibility of a data race, which the standard says is undefined behavior.

Reply

Andrey · *356 weeks ago*

The example appears a little bit strange to me. It doesn't ensure any synchronization between threads at all.

Reply

Sharath Gururaj · *327 weeks ago*

Hi Jeff, thank you for the excellent article.

1. you say: "Acquire semantics is a property which can only apply to operations which read from shared memory..."
So even though an acquire-semantic operation (#LoadLoad + #LoadStore) occurs at the beginning of a critical section, a store operation above the critical section can still float down freely anywhere inside the critical section. Would this not cause any problems?

2. In Herb Sutter's atomic weapons talk part 1 at 0:36:46, the diagram shows z="everything" floating up inside the critical section, above y="universe"
How is this possible if mutex.unlock() has release semantics? https://channel9.msdn.com/Shows/Going+Deep/Cpp-an...

3. What is so special about a read operation in "read-acquire" that has to happen at the beginning of the critical section and a write operation in "write-release" that happens at the end of a critical section?
Couldn't we have equally-well defined acquire-semantics as "write-acquire" (#StoreStore+#StoreLoad) and release semantics as read-release ($LoadLoad+$StoreLoad)?
you have mentioned that #StoreLoad is an expensive operation, but is that the only reason for defining it as the current definition?

Reply **1 reply** · *active 327 weeks ago*

preshing · *327 weeks ago*

Hi Sharath,

1. Not if the code is written correctly. (In particular, that code must not have any data races with other threads.) If the code is outside the critical section, then its execution order is not important with respect to other threads. And if that's the case, all that matters is that the compiler maintains the appearance of sequential order in its own thread, as discussed in http://preshing.com/20120625/memory-ordering-at-c...

2. Because a release operation doesn't prevent the reordering of memory operations that *follow* it in program order.

3. This combination of memory barriers is special because it forces all the memory operations inside the critical section to stay inside the critical section. The alternative you proposed would not do that.

Reply

Sharath Gururaj · *327 weeks ago*

Thanks for the reply.

1. got it!

2. But a release operation has a #LoadStore+#StoreStore barrier. In particular, the #StoreStore barrier will prevent stores outside the critical section from floating up above any stores inside the critical section, contrary to Herb Sutters slide.

3. I dont see why my alternative would allow operations inside the critical section to wander outside. A general critical section (in my scheme) would look like

store [var] <- 1 // a guard store
#StoreLoad + #StoreStore // compiled to processor specific instructions
// start of critical section
load operations
store operations
// end of critical section
$LoadLoad+$StoreLoad
load r1 <- [var2] // a guard load

Reply **1 reply** · *active 326 weeks ago*

preshing · *326 weeks ago*

2. You're confusing release operations with release fences. A release fence (such as std::atomic_thread_fence(std::memory_order_release)) would prevent the reordering of y="universe" & z="everything", as you suggest. But a release operation (such as a mutex unlock) does not necessarily prevent that reordering.

A release fence *can* be used to implement a release operation (if, say, you are implementing a mutex yourself), but that doesn't mean *all* release operations are based on release fences. That's what I meant in the post by "...these barriers are technically more strict than what's required for acquire and release semantics on a single memory operation."

See also http://preshing.com/20131125/acquire-and-release-...

3. Sure, you've prevented the operations between those barriers from wandering outside. But your example is not a critical section! If two threads call that code at the same time, they will both store "1" to var, cross the barrier, and then they will both happily execute the "critical" code at the same time. That defeats the purpose of having a critical section in the first place.

If you'd like to study more mutex implementations, see http://preshing.com/20120226/roll-your-own-lightw... and http://preshing.com/20120305/implementing-a-recur.... (Just keep in mind that there isn't much reason to actually implement your own mutex, except as an example. std::mutex is pretty close to optimal.)

Reply

Mike · *311 weeks ago*

Can you please elaborate a bit on acquire/release operations on atomic variables vs acquire/release fences? Let's say, that we implemented simple test-and-set spinlock using an acquire RMW operation in the lock function and release write operation in the unlock and didn't use standalone fences. Now let's consider an example where we have two such locks (lockA and lockB) and two threads acquiring/releasing the locks as follows:

```
// thread 1
lockA.lock();
lockA.unlock();
lockB.lock();
lockB.unlock();

// thread 2
lockB.lock();
lockA.lock();
lockB.unlock();
lockA.unlock();
```

AFAIU, in the first thread lockB.lock() and lockA.unlock() can be reordered (from the point of view of the second thread) since they perform acquire/release operations on different variables and that might lead to a deadlock, but if instead of acquire/release operations we used acquire/release fences we would be safe. Is my understanding correct or did I miss something?

Reply **3 replies** · *active 310 weeks ago*

preshing · *310 weeks ago*

That's a good question. Where did you get the example? It makes me think of the example at 44:35 in Herb Sutter's Atomic Weapons talk.

If lockB.lock() and lockA.unlock() can be reordered, I don't see how acquire/release operations vs. fences would make a difference. What makes you think fences would be safer?

I'm starting to think that the reordering is forbidden, regardless of whether fences are used, by section 32.4.1.12 of the latest C++ working draft: "Implementations should make atomic stores visible to atomic loads within a reasonable amount of time." I'm not 100% sure yet, but I might do a dedicated post on this question to at least explain it better.

Reply

Mike · *310 weeks ago*

I just made it up to illustrate my understanding of the difference between atomic operations on atomic variables and fences.

I thought that since atomic operations, unlike fences, are somewhat bound to memory locations atomic operations on different memory locations can be reordered more freely. Now when I've finally read the standard, I understand that I was wrong and fences work only when there are atomic operations on the same memory location, thus fences, in a way, are bound to memory locations too.

I think that reordering in such cases must be forbidden, but even though I understand why this behaviour is perfectly reasonable, I just fail to see how this behaviour follows from the standard and would love to read an article that would explain how to interpret related parts of the standard.

Reply

preshing · *310 weeks ago*

The post is up: http://preshing.com/20170612/can-reordering-of-re...

Reply

fanghaos · *309 weeks ago*

Thanks for this post!
I am studying the book C++ concurrency in Act, and here is an example I am confused, it's in 5.3.2 ACQUIRE-RELEASE ORDERING Listing 5.7 Acquire-release doesn’t imply a total ordering page 133.

```
#include <atomic>
#include <thread>
#include <assert.h>

std::atomic<bool> x,y;
std::atomic<int> z;
void write_x()
{
x.store(true,std::memory_order_release);
}
void write_y()
{
y.store(true,std::memory_order_release);
}
void read_x_then_y()
{
while(!x.load(std::memory_order_acquire));
if(y.load(std::memory_order_acquire)) // 1
++z;
}
void read_y_then_x()
{
while(!y.load(std::memory_order_acquire));
if(x.load(std::memory_order_acquire)) //2
++z;
}
int main()
{
x=false;
y=false;
z=0;
```

```
std::thread a(write_x);
std::thread b(write_y);
std::thread c(read_x_then_y);
std::thread d(read_y_then_x);
a.join();
b.join();
c.join();
d.join();
assert(z.load()!=0); // 3
}
```

The author said "In this case the assert 3 can fire (just like in the relaxed-ordering case), because it’s possible for both the load of x 2 and the load of y 1 to read false. x and y are written by different threads, so the ordering from the release to the acquire in each case has no effect on the operations in the other threads.
Figure 5.6 shows the happens-before relationships from listing 5.7, along with a possible outcome where the two reading threads each have a different view of the world. This is possible because there’s no happens-before relationship to force an ordering, as described previously."

In my option, the function "read_x_then_y" and "read_y_then_x" use the memory_order_acquire, the subsequent statements after memory_order_acquire won't be reorder to the location that is before the statement which use memory_order_acquire. So, the statement if(y.load(std::memory_order_acquire)) ++z won't be executed before while(!x.load(std::memory_order_acquire));, if(x.load(std::memory_order_acquire)) ++z won't be executed before while(!y.load(std::memory_order_acquire));.
If I understand you correctly, the z will not be 0.
Do you think so? Thanks a lot

Reply

 zhihao · *298 weeks ago*

Thanks very much for this post! Good job!
I am puzzled about "For now, suffice to say that for this technique to work in general, the acquire and release semantics must apply to the same variable – in this case, Ready – and both the load and store must be atomic operations."

What will happen if the the load or store is not atomic operations to Ready ? Have i miss something ?

Reply

- Here's a Standalone Cairo DLL for Windows
- Learn CMake's Scripting Language in 15 Minutes
- How to Build a CMake-Based Project
- Using Quiescent States to Reclaim Memory
- Leapfrog Probing
- A Resizable Concurrent Map
- New Concurrent Hash Maps for C++
- You Can Do Any Kind of Atomic Read-Modify-Write Operation
- Safe Bitfields in C++
- Semaphores are Surprisingly Versatile