



# [Preshing on Programming](#)

- [Twitter](#)
- [RSS](#)

Navigate...

- [Blog](#)
- [Archives](#)
- [About](#)

Sep 22, 2013

## Acquire and Release Fences



Acquire and release fences, in my opinion, are rather misunderstood on the web right now. That's too bad, because the C++11 Standards Committee did a great job specifying the meaning of these memory fences. They enable robust algorithms which scale well across multiple cores, and map nicely onto today's most common CPU architectures.

First things first: Acquire and release fences are considered *low-level* lock-free operations. If you stick with higher-level, [sequentially consistent](#) atomic types, such as `volatile` variables in Java 5+, or default atomics in C++11, you don't need acquire and release fences. The tradeoff is that sequentially consistent types are slightly less scalable or performant for some algorithms.

On the other hand, if you've developed for multicore devices in the days before C++11, you might feel an affinity for acquire and release fences. Perhaps, like me, you remember struggling with the placement of some `lwsync` intrinsics while synchronizing threads on Xbox 360. What's cool is that once you understand acquire and release fences, you actually see what we were trying to accomplish using those platform-specific fences all along.

Acquire and release fences, as you might imagine, are standalone memory fences, which means that they aren't coupled with any particular memory operation. So, how do they work?

An **acquire fence** prevents the memory reordering of any **read** which precedes it in program order with any **read or write** which follows it in program order.

A **release fence** prevents the memory reordering of any **read or write** which precedes it in program order with any **write** which follows it in program order.

In other words, in terms of the barrier types [explained here](#), an acquire fence serves as both a #LoadLoad + #LoadStore barrier, while a release fence functions as both a #LoadStore + #StoreStore barrier. That's all they purport to do.



When programming in C++11, you can invoke them using the following functions:

```
#include <atomic>
std::atomic_thread_fence(std::memory_order_acquire);
std::atomic_thread_fence(std::memory_order_release);
```

In C11, they take this form:

```
#include <stdatomic.h>
atomic_thread_fence(memory_order_acquire);
atomic_thread_fence(memory_order_release);
```

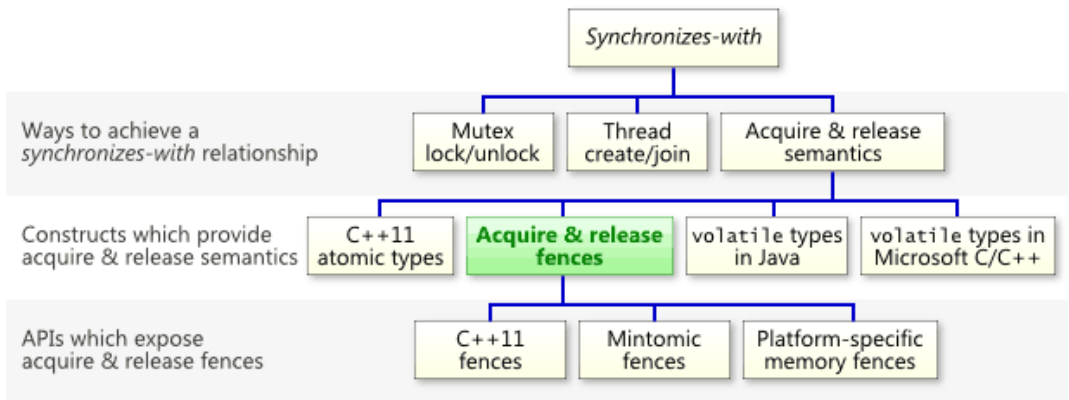
And using [Mintomic](#), a small, portable lock-free API:

```
#include <mintomic/mintomic.h>
mint_thread_fence_acquire();
mint_thread_fence_release();
```

On the SPARC-V9 architecture, an acquire fence can be implemented using the `membar #LoadLoad | #LoadStore` instruction, and an a release fence can be implemented as `membar #LoadStore | #StoreStore`. On other CPU architectures, the people who implement the above libraries must translate these operations into the next best thing – some CPU instruction which provides *at least* the required barrier types, and possibly more. On PowerPC, the next best thing is `lwsync`. On ARMv7, the next best thing is `dmb`. On Itanium, the next best thing is `mf`. And on x86/64, [no CPU instruction](#) is needed at all. As you might expect, acquire and release fences restrict reordering of neighboring operations [at compile time](#) as well.

## They Can Establish *Synchronizes-With* Relationships

The most important thing to know about acquire and release fences is that they can establish a [synchronizes-with](#) relationship, which means that they prohibit memory reordering in a way that allows you to pass information reliably between threads. Keep in mind that, as the following chart illustrates, acquire and release fences are just one of many constructs which can establish a *synchronizes-with* relationship.



As I've [shown before](#), a relaxed [atomic](#) load immediately followed by an acquire fence will convert that load into a read-acquire. Similarly, a relaxed atomic store immediately preceded by a release fence will convert that store into a write-release. For example, if `g_guard` has type `std::atomic<int>`, then this line

```
g_guard.store(1, std::memory_order_release);
```

can be safely replaced with the following.

```
std::atomic_thread_fence(std::memory_order_release);
g_guard.store(1, std::memory_order_relaxed);
```

One precision: In the latter form, it is no longer the store which *synchronizes-with* anything. It is the fence itself. To see what I mean, let's walk through a detailed example.

## A Walkthrough Using Acquire and Release Fences

We'll take the example from my [previous post](#) and modify it to use C++11's standalone acquire and release fences. Here's the `SendTestMessage` function. The atomic write is now `relaxed`, and a release fence has been placed immediately before it.

```
void SendTestMessage(void* param)
{
    // Copy to shared memory using non-atomic stores.
    g_payload.tick = clock();
    g_payload.str = "TestMessage";
    g_payload.param = param;

    // Release fence.
    std::atomic_thread_fence(std::memory_order_release);

    // Perform an atomic write to indicate that the message is ready.
    g_guard.store(1, std::memory_order_relaxed);
}
```

Here's the `TryReceiveMessage` function. The atomic read has been relaxed, and an acquire fence has been placed slightly after it. In this case, the fence does not occur *immediately* after the read; we first check whether `ready != 0`, since that's the only case where the fence is really needed.

```
bool TryReceiveMessage(Message& result)
{
    // Perform an atomic read to check whether the message is ready.
    int ready = g_guard.load(std::memory_order_relaxed);

    if (ready != 0)
    {
        // Acquire fence.
        std::atomic_thread_fence(std::memory_order_acquire);
    }
}
```

```

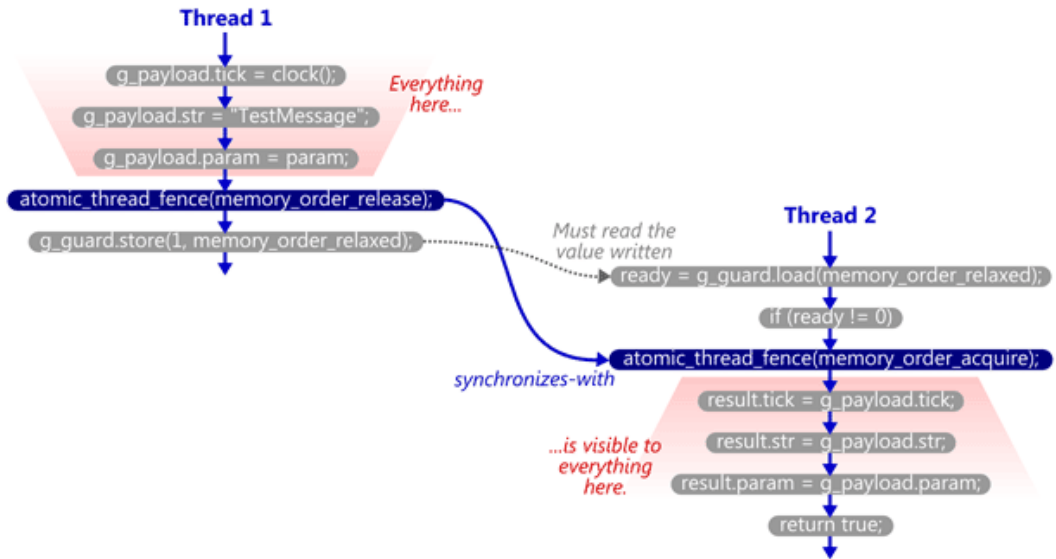
// Yes. Copy from shared memory using non-atomic loads.
result.tick = g_payload.tick;
result.str = g_msg_str;
result.param = g_payload.param;

return true;
}

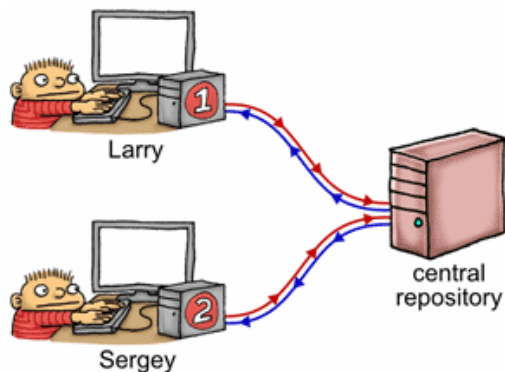
// No.
return false;
}

```

Now, if `TryReceiveMessage` happens to see the write which `SendTestMessage` performed on `g_guard`, then it will issue the acquire fence, and the *synchronizes-with* relationship is complete. Again, strictly speaking, it's the fences which *synchronizes-with* each other.



Let's back up for a moment and consider this example in terms of the [source control analogy](#) I made a while back. Imagine shared memory as a central repository, with each thread having its own private copy of that repository. As each thread manipulates its private copy, modifications are constantly "leaking" to and from the central repository at unpredictable times. Acquire and release fences are used to enforce ordering among those leaks.



If we imagine Thread 1 as a programmer named Larry, and Thread 2 as a programmer named Sergey, what happens is the following:

1. Larry performs a bunch of non-atomic stores to his private copy of `g_payload`.
2. Larry issues a release fence. That means that all his previous memory operations will be committed to the repository – whenever that happens –

*before* any store he performs next.

3. Larry stores 1 to his private copy of `g_guard`.
4. At some random moment thereafter, Larry's private copy of `g_guard` leaks to the central repository, entirely on its own. Remember, once this happens, we're guaranteed that Larry's changes to `g_payload` are in the central repository, too.
5. At some random moment thereafter, the updated value of `g_guard` leaks from the central repository to Sergey's private copy, entirely on its own.
6. Sergey checks his private copy of `g_guard` and sees 1.
7. Seeing that, Sergey issues an acquire fence. All the contents of Sergey's private copy become *at least* as new as his previous load. This completes the *synchronizes-with* relationship.
8. Sergey performs a bunch of non-atomic loads from his private copy of `g_payload`. At this point, he is guaranteed to see the values that were written by Larry.

Note that the guard variable must “leak” from Larry's private workspace over to Sergey's all by itself. When you think about it, acquire and release fences are just a way to piggyback additional data on top of such leaks.

## The C++11 Standard's Got Our Back

The C++11 standard explicitly states that this example will work on any compliant implementation of the library and language. The promise is made in §29.8.2 of [working draft N3337](#):

A release fence A **synchronizes with** an acquire fence B if there exist atomic operations X and Y, both operating on some atomic object M, such that A is sequenced before X, X modifies M, Y is sequenced before B, and Y reads the value written by X or a value written by any side effect in the hypothetical release sequence X would head if it were a release operation.

That's a lot of letters. Let's break it down. In the above example:

- *Release fence A* is the release fence issued in `SendTestMessage`.
- *Atomic operation X* is the relaxed atomic store performed in `SendTestMessage`.
- *Atomic object M* is the guard variable, `g_guard`.
- *Atomic operation Y* is the relaxed atomic load performed in `TryReceiveMessage`.
- *Acquire fence B* is the acquire fence issued in `TryReceiveMessage`.

And finally, *if* the relaxed atomic load reads the value written by the relaxed atomic store, the C++11 standard says that the fences *synchronize-with* each other, just as I've shown.

I like C++11's approach to portable memory fences. Other people have attempted to design portable memory fence APIs in the past, but in my opinion, few of them hit the sweet spot for lock-free programming like C++11, as far as standalone fences go. And while acquire and release fences may not translate directly into native CPU instructions, they're close enough that you can still squeeze out as much performance as is currently possible from the vast majority of multicore devices. That's why Mintomic, an open source library I released earlier this year, offers [acquire and release fences](#) – along with a consume and full memory fence – as its only memory ordering operations. [Here's](#) the example from this post, rewritten using Mintomic.

In an upcoming post, I'll highlight a couple of misconceptions about acquire & release fences which are currently floating around the web, and discuss some performance concerns. I'll also talk a little bit more about their relationship with read-acquire and write-release operations, including some consequences of that relationship which tend to trip people up.

 2

 28


 Like

 Tweet


 G+1

[« The Synchronizes-With Relation Double-Checked Locking is Fixed In C++11 »](#)

## Comments (19)

 Logged in as [bgn9000](#)

[Dashboard](#) | [Edit profile](#) | [Logout](#)



 tobi · 121 weeks ago

About the code of the relaxed example:

```
std::atomic_thread_fence(std::memory_order_release);
g_guard.store(1, std::memory_order_relaxed);
```

Is it guaranteed that the relaxed `g_guard` store cannot move upwards to before the release fence? In my understanding release constrains stuff from moving below it, not above it. And a relaxed store should be able to move pretty freely around.


Reply  [1 reply](#) · active 121 weeks ago

  [Jeff Preshing](#) · 121 weeks ago

A release fence does not constrain stuff from moving below itself. [That's a misconception.](#)


A release fence constrains stuff from moving below *subsequent writes*.

Reply

 [brucedawson](#) · 121 weeks ago

Excellent article as always. I really like that C++ 11 is bringing sanity to the implementation of fences. The number of caveats and exceptions previously required when describing memory ordering (especially around volatile and InterlockedXXX on Xbox 360) has always made me uncomfortable. It is great to finally be getting rid of the accidental complexity around this. This still leaves the not-inconsiderable essential complexity of lockless programming, but that's okay.

Reply

 [Nawaz](#) · 119 weeks ago

Great article, as usual.

BTW, when describing `TryReceiveMessage()`, there is a typo in this statement:

"The atomic read has been relaxed, and a release fence has been placed slightly after it."

which should be this:

"The atomic read has been relaxed, and AN ACQUIRE fence has been placed slightly after it."

Hope you fix that. :-)

Reply

[3 replies](#) · active 118 weeks ago



[Jeff Preshing](#) · 119 weeks ago

Fixed now. Thanks, eagle-eyed reader!

Reply



[Nawaz](#) · 119 weeks ago

I finished reading few more parts of the article, and found few more typos (serious ones):

6. Larry checks his private copy of `g_guard` and sees 1.
7. Seeing that, Larry issues an acquire fence. All the contents of Larry's private copy become at least as new as his previous load. This completes the synchronizes-with relationship.
8. Larry performs a bunch of non-atomic loads from his private copy of `g_payload`. At this point, he is guaranteed to see the values that were written by Larry.

As per my understanding these lines should talk about Sergey (not Larry), so it should be:

6. Sergey checks his private copy of `g_guard` and sees 1.
7. Seeing that, Sergey issues an acquire fence. All the contents of Larry's private copy become at least as new as his previous load. This completes the synchronizes-with relationship.
8. Sergey performs a bunch of non-atomic loads from his private copy of `g_payload`. At this point, he is guaranteed to see the values that were written by Larry.

Let me know if I got it wrong. If I'm right, please fix these too. :-)

Reply



[Jeff Preshing](#) · 118 weeks ago

Yeah, those were pretty serious. Glad somebody was paying attention. There was even one more Larry that should have been a Sergey. Fixed now! I hope you've subscribed so that you'll proofread my future posts. :)

Reply



[Petr Machata](#) · 118 weeks ago

It appears that ARMv8 (AArch64) has a bunch of operations with exactly the release/acquire semantics.

Reply

[1 reply](#) · active 118 weeks ago



[Jeff Preshing](#) · 118 weeks ago

To be clear, ARMv8 adds **read**-acquire and **write**-release instructions. Those are different from standalone acquire and release **fences**, as described here.

Reply



[alex](#) · 117 weeks ago

thatnks for great articles

there is a small omission in one picture (synchronizes with) - if(ready!=0) is placed before acquiring

if i got i right

Reply

[4 replies](#) · active 117 weeks ago



· 117 weeks ago

[Jeff Preshing](#) 

Not sure which error you are referring to. Can you be more specific?

Reply



alex · 117 weeks ago

sorry  
was unattentive to what actually this check relates

Reply



alex · 117 weeks ago

i was looking at this piece of code

```
bool TryReceiveMessage(Message& result)
{
    int ready = g_guard.load(std::memory_order_relaxed);
    if (ready != 0)
    {
        std::atomic_thread_fence(std::memory_order_acquire);
```

and it seems to me, that acquire should happen before check

i guess i am wrong :)

Reply



[Jeff Preshing](#) · 117 weeks ago

You could put the acquire fence before the check, as long as it remains after the load from g\_guard, as follows:

```
bool TryReceiveMessage(Message& result)
{
    int ready = g_guard.load(std::memory_order_relaxed);
    std::atomic_thread_fence(std::memory_order_acquire);
    if (ready != 0)
    {
        result.tick = g_payload.tick;
        ...
    }
}
```

That's still correct. However, now you're paying the cost of the fence even when g\_guard is still 0. That's why I prefer to put it after the check.

The important thing is that the fence takes place at some point after g\_guard is loaded, but before g\_payload is touched.

Reply



[Joe](#) · 72 weeks ago

Great article, I'm currently looking in to implementing lock-free producer/consumer queues for inter-thread communication, and I think I'm gradually getting my head around what the memory orders mean.

I have one question, in the example code does g\_guard have to be an atomic type? It's my understanding that the main thing is that I don't want the compiler/cpu to reorder the writes of the payload, with the write of the guard, and need the guard value to be written last. So for that to happen, isn't it enough to just put a release fence before setting the guard value?



Reply

[3 replies](#) · active 44 weeks ago



[Jeff Preshing](#) · 44 weeks ago

Technically, for portable code, yes, `g_guard` has to have an atomic type. Otherwise, you risk [torn reads and torn writes](#).

Having said that, if `g_guard` is just 32 bits, then plain (non-atomic) loads and stores are already atomic on basically all modern processors. So if you cheat, you'll pretty much always get away with it.

Reply



[Joe](#) · 44 weeks ago

But am I right in thinking that a torn read/write could still occur with a 32 bit guard variable which crosses a cache line boundary?

Though, if the guard variable is only a boolean anyway, do torn reads/writes actually matter?

Don't worry, I'll use `std::atomic` anyway, I'm just curious =)

Reply



[Jeff Preshing](#) · 44 weeks ago

Right, except when it crosses a cache line boundary. I showed it in the post I linked. I should have written "if `g_guard` is naturally-aligned 32 bits".

As for a bool, good question. I'll just defer to the standard, which says you should use `std::atomic<>` any time there is concurrent access with at least one write.

Reply



[Jens](#) · 37 weeks ago

Thank you for a very informative article!

Just one question. What makes me nervous about those fences and the relaxed atomic write is this:

> At some random moment thereafter, Larry's private copy of `g_guard` leaks to the central repository, entirely on its own

So theoretically it could take minutes, hours or longer, before the data eventually propagates to the central repository and finally to the private copy of Sergey. Without an upper bound on the time it takes, fences are not of much use, I think.

Mutexes on the contrary guarantee, that all data propagate to the other thread in a timely manner (as soon as the other thread acquires the mutex).

I was always under the impression, that C++11 atomic operations guarantee a timely visibility of the changed value to other threads, even if a relaxed memory model is used. Is this not the case? Do atomic operations not propagate any faster than non-atomic "normal" stores?

Reply

## Post a new comment

Enter text right here!

Posting as [bgn9000](#) ([Logout](#))

Subscribe to

[Submit Comment](#)

## Recent Posts

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

## Tip Jar

If you like this blog, [leave a tip!](#)



Copyright © 2015 Jeff Preshing - Powered by [Octopress](#)