Preshing on Programming

- Twitter
- RSS

Navigate…

- Blog
- Archives
- About
- Contact

Sep 30, 2013

Double-Checked Locking is Fixed In C++11

The **double-checked locking pattern** (DCLP) is a bit of a notorious case study in lock-free programming. Up until 2004, there was no safe way to implement it in Java. Before C++11, there was no safe way to implement it in portable C++.

The pattern gained attention for the shortcomings it exposed in those languages, and people began to write about it. In 2000, a group of high-profile Java developers got together and signed a declaration entitled "Double-Checked Locking Is Broken". In 2004, Scott Meyers and Andrei Alexandrescu published an article entitled "C++ and the Perils of Double-Checked Locking". Both papers are great primers on what DCLP is, and why, at the time, those languages were inadequate for implementing it.

All of that's in the past. Java now has a revised memory model, with new semantics for the `volatile` keyword, which makes it possible to implement DCLP safely. Likewise, C++11 has a shiny new memory model and atomic library that enable a wide variety of portable DCLP implementations. C++11, in turn, inspired Mintomic, a small library I released earlier this year which makes it possible to implement DCLP on some older C/C++ compilers as well.

In this post, I'll focus on the C++ implementations of DCLP.

What Is Double-Checked Locking?

Suppose you have a class that implements the well-known Singleton pattern, and you want to make it thread-safe. The obvious approach is to ensure mutual exclusivity by adding a lock. That way, if two threads call `Singleton::getInstance` simultaneously, only one of them will create the singleton.

```
Singleton* Singleton::getInstance() {
    Lock lock;      // scope-based lock, released automatically when the function returns
    if (m_instance == NULL) {
        m_instance = new Singleton;
    }
    return m_instance;
}
```

It’s a totally valid approach, but once the singleton is created, there isn’t really any need for the lock anymore. Locks aren’t necessarily slow, but they don’t scale well under heavy contention.

The double-checked locking pattern avoids this lock when the singleton already exists. However, it’s not so simple, as the Meyers-Alexandrescu paper shows. In that paper, the authors describe several flawed attempts to implement DCLP in C++, dissecting each attempt to explain why it’s unsafe. Finally, on page 12, they show an implementation that *is* safe, but that depends on unspecified, platform-specific memory barriers.

```
Singleton* Singleton::getInstance() {
    Singleton* tmp = m_instance;
    ...                     // insert memory barrier
    if (tmp == NULL) {
        Lock lock;
        tmp = m_instance;
        if (tmp == NULL) {
            tmp = new Singleton;
            ...             // insert memory barrier
            m_instance = tmp;
        }
    }
    return tmp;
}
```

Here, we see where the double-checked locking pattern gets its name: We only take a lock when the singleton pointer `m_instance` is `NULL`, which serializes the first group of threads that happen to see that value. Once inside the lock, `m_instance` is checked a second time, so that only the first thread will create the singleton.

This is very close to a working implementation. It’s just missing some kind of memory barrier on the highlighted lines. At the time when the authors wrote the paper, there was no portable C/C++ function that could fill in the blanks. Now, with C++11, there is.

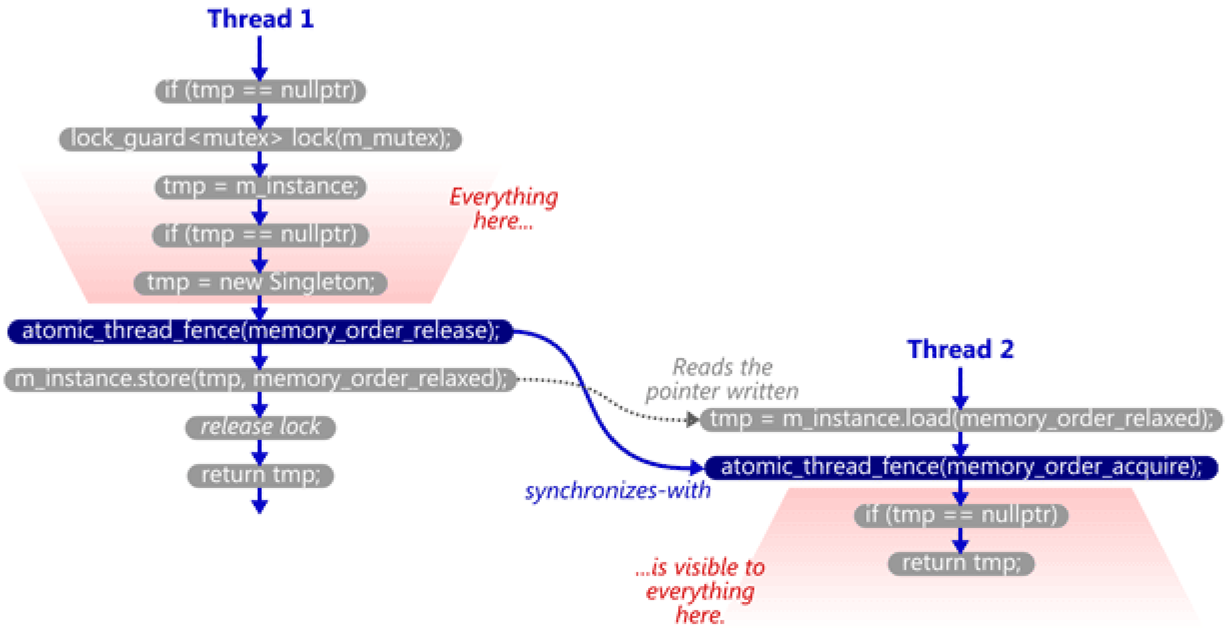
Using C++11 Acquire and Release Fences

You can safely complete the above implementation using acquire and release fences, a subject I explained at length in my previous post. However, to make this code truly portable, you must also wrap `m_instance` in a C++11 atomic type and manipulate it using relaxed atomic operations. Here’s the resulting code, with the acquire and release fences highlighted.

```
std::atomic<Singleton*> Singleton::m_instance;
std::mutex Singleton::m_mutex;

Singleton* Singleton::getInstance() {
    Singleton* tmp = m_instance.load(std::memory_order_relaxed);
    std::atomic_thread_fence(std::memory_order_acquire);
    if (tmp == nullptr) {
        std::lock_guard<std::mutex> lock(m_mutex);
        tmp = m_instance.load(std::memory_order_relaxed);
        if (tmp == nullptr) {
            tmp = new Singleton;
            std::atomic_thread_fence(std::memory_order_release);
            m_instance.store(tmp, std::memory_order_relaxed);
        }
    }
    return tmp;
}
```

This works reliably, even on multicore systems, because the memory fences establish a *synchronizes-with* relationship between the thread that creates the singleton and any subsequent thread that skips the lock. `Singleton::m_instance` acts as the guard variable, and the contents of the singleton itself are the payload.



That's what all those flawed DCLP implementations were missing: Without any *synchronizes-with* relationship, there was no guarantee that all the writes performed by the first thread – in particular, those performed in the `Singleton` constructor – were visible to the second thread, even if the `m_instance` pointer itself was visible! The lock held by the first thread didn't help, either, since the second thread doesn't acquire any lock, and can therefore run concurrently.

If you're looking for a deeper understanding of how and why these fences make DCLP work reliably, there's some background information in my previous post as well as in earlier posts on this blog.

Using Mintomic Fences

Mintomic is a small C library that provides a subset of functionality from C++11's atomic library, including acquire and release fences, and which works on older compilers. Mintomic relies on the assumptions of the C++11 memory model – specifically, the absence of out-of-thin-air stores – which is technically not guaranteed by older compilers, but it's the best we can do without C++11. Keep in mind that these are the circumstances in which we've written multithreaded C++ code for years. Out-of-thin-air stores have proven unpopular over time, and good compilers tend not to do it.

Here's a DCLP implementation using Mintomic's acquire and release fences. It's basically equivalent to the previous example using C++11's acquire and release fences.

```
mint_atomicPtr_t Singleton::m_instance = { 0 };
mint_mutex_t Singleton::m_mutex;

Singleton* Singleton::getInstance() {
    Singleton* tmp = (Singleton*) mint_load_ptr_relaxed(&m_instance);
    mint_thread_fence_acquire();
    if (tmp == NULL) {
        mint_mutex_lock(&m_mutex);
        tmp = (Singleton*) mint_load_ptr_relaxed(&m_instance);
        if (tmp == NULL) {
            tmp = new Singleton;
            mint_thread_fence_release();
            mint_store_ptr_relaxed(&m_instance, tmp);
        }
        mint_mutex_unlock(&m_mutex);
    }
```

```
    return tmp;
}
```

To implement acquire and release fences, Mintomic tries to generate the most efficient machine code possible on every platform it supports. For example, here's the resulting machine code on Xbox 360, which is based on PowerPC. On this platform, an inline `lwsync` is the [leanest instruction](#) that can serve as both an acquire and release fence.

```
Singleton::getInstance:
82010098  mflr r12
8201009C  bl __savegprlr + 003ch (8201386ch)
820100A0  stwu r1,-70h(r1)
$M65928:
820100A4  lis r30,-32252        ; 8204h
820100A8  lwz r3,137Ch(r30)
820100AC  lwsync
820100B0  cmplwi cr6,r3,0
820100B4  bne cr6,Singleton::getInstance + 0060h (820100f8h)
820100B8  lis r11,-32252        ; 8204h
820100BC  addi r29,r11,4960     ; 1360h
820100C0  mr r3,r29
820100C4  bl RtlEnterCriticalSection (82030420h)
820100C8  lwz r11,137Ch(r30)
820100CC  cmplwi cr6,r11,0
820100D0  mr r31,r11
820100D4  bne cr6,Singleton::getInstance + 0054h (820100ech)
820100D8  li r3,1
820100DC  bl operator new (82013778h)
820100E0  mr r31,r3
820100E4  lwsync
820100E8  stw r3,137Ch(r30)
820100EC  mr r3,r29
820100F0  bl RtlLeaveCriticalSection (82030430h)
820100F4  mr r3,r31
820100F8  addi r1,r1,112        ; 70h
820100FC  b __restgprlr + 003ch (820138bch)
```

The previous C++11-based example could (and ideally, would) generate the exact same machine code for PowerPC when optimizations are enabled. Unfortunately, I don't have access to a C++11-compliant PowerPC compiler to verify this.

Using C++11 Low-Level Ordering Constraints

C++11's acquire and release fences can implement DCLP correctly, and *should* be able to generate optimal machine code on the majority of today's multicore devices (as Mintomic does), but they're not considered very fashionable. The preferred way to achieve the same effect in C++11 is to use atomic operations with low-level ordering constraints. As I've [shown previously](#), a write-release can *synchronize-with* a read-acquire.

```
std::atomic<Singleton*> Singleton::m_instance;
std::mutex Singleton::m_mutex;

Singleton* Singleton::getInstance() {
    Singleton* tmp = m_instance.load(std::memory_order_acquire);
    if (tmp == nullptr) {
        std::lock_guard<std::mutex> lock(m_mutex);
        tmp = m_instance.load(std::memory_order_relaxed);
        if (tmp == nullptr) {
            tmp = new Singleton;
            m_instance.store(tmp, std::memory_order_release);
```

```
        }
    }
    return tmp;
}
```

Technically, this form of lock-free synchronization is less strict than the form using standalone fences; the above operations are only meant to prevent memory reordering around *themselves*, as opposed to standalone fences, which are meant to prevent certain kinds of memory reordering around *neighboring operations*. Nonetheless, on the x86/64, ARMv6/v7, and PowerPC architectures, the best possible machine code is the same for both forms. For example, in an [older post](), I showed how C++11 low-level ordering constraints emit `dmb` instructions on an ARMv7 compiler, which is the same thing you'd expect using standalone fences.

One platform on which the two forms are likely to generate different machine code is Itanium. Itanium can implement C++11's `load(memory_order_acquire)` using a single CPU instruction, `ld.acq`, and `store(tmp, memory_order_release)` using `st.rel`. I'd love to investigate the performance difference of these instructions versus standalone fences, but have no access to an Itanium machine.

Another such platform is the recently introduced ARMv8 architecture. ARMv8 offers `ldar` and `stlr` instructions, which are similar to Itanium's `ld.acq` and `st.rel` instructions, except that they also enforce the heavier [StoreLoad]() ordering between the `stlr` instruction and any subsequent `ldar`. In fact, ARMv8's new instructions are intended to implement C++11's SC atomics, described next.

Using C++11 Sequentially Consistent Atomics

C++11 offers an entirely different way to write lock-free code. (We can consider DCLP "lock-free" in certain codepaths, since not all threads take the lock.) If you omit the optional `std::memory_order` argument on all atomic library functions, the default value is `std::memory_order_seq_cst`, which turns all atomic variables into [sequentially consistent]() (SC) atomics. With SC atomics, the whole algorithm is guaranteed to appear sequentially consistent as long as there are no [data races](). SC atomics are really similar to `volatile` variables in Java 5+.

Here's a DCLP implementation that uses SC atomics. As in all previous examples, the second highlighted line will *synchronize-with* the first once the singleton is created.

```
std::atomic<Singleton*> Singleton::m_instance;
std::mutex Singleton::m_mutex;

Singleton* Singleton::getInstance() {
    Singleton* tmp = m_instance.load();
    if (tmp == nullptr) {
        std::lock_guard<std::mutex> lock(m_mutex);
        tmp = m_instance.load();
        if (tmp == nullptr) {
            tmp = new Singleton;
            m_instance.store(tmp);
        }
    }
    return tmp;
}
```

SC atomics are considered easier for programmers to reason about. The tradeoff is that the generated machine code tends to be less efficient than that of the previous examples. For example, here's some x64 machine code for the above code listing, as generated by Clang 3.3 with optimizations enabled:

```
Singleton::getInstance():
    pushq   %rbx
    movq    Singleton::m_instance(%rip), %rbx
    testq   %rbx, %rbx
    jne .LBB0_7
    movl    $__pthread_key_create, %eax
    testq   %rax, %rax
    je  .LBB0_3
    movl    Singleton::m_mutex, %edi
    callq   pthread_mutex_lock
    testl   %eax, %eax
    jne .LBB0_8
.LBB0_3:
    movq    Singleton::m_instance(%rip), %rbx
    testq   %rbx, %rbx
    jne .LBB0_5
    movl    $1, %edi
    callq   operator new(unsigned long)
    movq    %rax, %rbx
    xchgq   %rax, Singleton::m_instance(%rip)
.LBB0_5:
    movl    $__pthread_key_create, %eax
    testq   %rax, %rax
    je  .LBB0_7
    movl    Singleton::m_mutex, %edi
    callq   pthread_mutex_unlock
.LBB0_7:
    movq    %rbx, %rax
    popq    %rbx
    ret
.LBB0_8:
    movl    %eax, %edi
    callq   std::__throw_system_error(int)
```

Because we've used SC atomics, the store to `m_instance` has been implemented using an `xchg` instruction, which acts as a full memory fence on x64. That's heavier instruction than DCLP really needs on x64. A plain `mov` instruction would have done the job. It doesn't matter too much, though, since the `xchg` instruction is only issued once, in the codepath where the singleton is first created.

On the other hand, if you compile SC atomics for PowerPC or ARMv6/v7, you're pretty much guaranteed lousy machine code. For the gory details, see 00:44:25 - 00:49:16 of Herb Sutter's [atomic<> Weapons talk, part 2].

Using C++11 Data-Dependency Ordering

In all of the above examples I've shown here, there's a *synchronizes-with* relationship between the thread that creates the singleton and any subsequent thread that avoids the lock. The guard variable is the singleton pointer, and the payload is the contents of the singleton itself. In this case, the payload is considered a **data dependency** of the guard pointer.

It turns out that when working with data dependencies, a read-acquire operation, which all of the above examples use, is actually overkill! It's sufficient to perform a **consume operation** instead. Consume operations are meant to eliminate one of the `lwsync` instructions on PowerPC, and one of the `dmb` instructions on ARMv7. I'll write more about data dependencies and consume operations in a [future post].

Using a C++11 Static Initializer

Some readers already know the punch line to this post: C++11 doesn't require you to jump through any of the above hoops to get a thread-safe singleton. You can simply [use a static initializer].

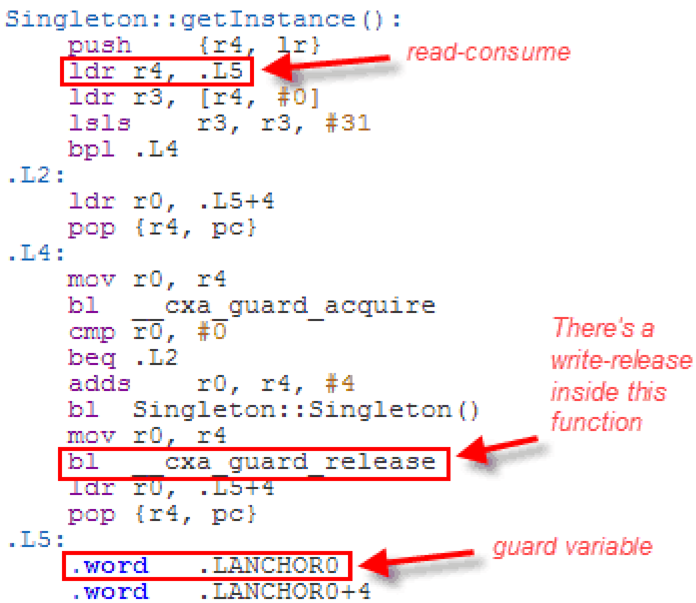
[Update: Beware! As Rober Baker points out in the comments, this example doesn't work in Visual Studio 2012 SP4. It only works in compilers that fully comply with this part of the C++11 standard.]

```
Singleton& Singleton::getInstance() {
    static Singleton instance;
    return instance;
}
```

The C++11 standard's got our back in §6.7.4:

> If control enters the declaration concurrently while the variable is being initialized, the concurrent execution shall wait for completion of the initialization.

It's up to the compiler to fill in the implementation details, and DCLP is the obvious choice. There's no guarantee that the compiler will use DCLP, but it just so happens that some (perhaps most) C++11 compilers do. Here's some machine code generated by GCC 4.6 when compiling for ARM with the `-std=c++0x` option:



Since the `Singleton` is constructed at a fixed address, the compiler has introduced a separate guard variable for synchronization purposes. Note in particular that there's no `dmb` instruction to act as an acquire fence after the initial read of this guard variable. The guard variable is a pointer to the singleton, and therefore the compiler can take advantage of the data dependency to omit the `dmb` instruction. `__cxa_guard_release` performs a write-release on the guard, is therefore *dependency-ordered-before* the read-consume once the guard has been set, making the whole thing resilient against memory reordering, just like all the previous examples.

As you can see, we've come a long way with C++11. Double-checked locking is fixed, and then some!

Personally, I've always thought that if you want to initialize a singleton, best to do it at program startup. But DCLP can certainly help you out of a jam. And as it happens, you can also use DCLP to store arbitrary value types in a lock-free hash table. More about that in a future post as well.

« Acquire and Release Fences Acquire and Release Fences Don't Work the Way You'd Expect »

Comments (49)

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

Umar · *353 weeks ago*

What are your thoughts on using std::call_once instead of DCL?

Reply [5 replies](#) · *active 192 weeks ago*

[Jeff Preshing](#) · *353 weeks ago*

`std::call_once` is a totally viable option since the first (successful) call *synchronizes-with* all subsequent calls (30.4.4.3). And compilers can implement it in a way that closely resembles DCLP. I could add an example to this post, but it's already long enough and besides, I mainly wanted to focus on the implementations which use atomic operations explictly.

Reply

Brian · *301 weeks ago*

I did some benchmarking of various Singleton implementations, and found those using std::call_once() to be quite slow compared to those using static initialization.

Reply

's avat [@rainer_grimm](#) · *224 weeks ago*

I did the same in 6 different variants and get similar results: Have a look here: http://www.grimm-jaud.de/index.php/blog/threadsic...
Of course, the article is in German, but their is a translate button in the right top corner.

Reply

[Rainer](#) · *192 weeks ago*

Now my post is in English: http://www.modernescpp.com/index.php/thread-safe-...

Reply

Shanker · *206 weeks ago*

You can use a static variable inside the initialization member of the singletons. The initialization is defined to happen on exactly one thread, and no other threads will proceed until that initialization is complete, so the race condition is just over which thread gets to do the initialization rather than anything more problematic.

Reply

Tobias Bruell · *352 weeks ago*

I do not understand the part about "Using C++11 Low-Level Ordering Constraints" completely. Is the code in that section still a 100% valid standard-conforming implementation of the DCLP?

Also, I do not understand what you mean by saying that "this form is less strict". And I find the "around themselves" and "around all neighboring operations" somewhat unclear.

Very interesting read anyway. I think this whole blog should be wrapped up into a book.

Reply [1 reply](#) · *active 352 weeks ago*

[Jeff Preshing](#) · *352 weeks ago*

All the C++11 examples in this post are valid and standard-conforming. I've obviously left out the class definitions (header file), but you can easily infer what those should be. I suppose I could have gone further and created a Github repository with a working project, but I don't think a lot of people would build/run it.

As for the difference between atomic memory operations which prevent reordering only around themselves, versus memory fences which prevent reordering around all neighboring operations, I'll try to clarify that further in the next post!

Reply

Anirban Mitra · *351 weeks ago*

Would this be a valid DCLP

```
Singleton* Singleton::m_instance;
std::mutex Singleton::m_mutex;

Singleton* Singleton::getInstance() {
std::atomic_thread_fence(std::memory_order_acquire);
Singleton* tmp = m_instance;

if (tmp == nullptr) {
std::lock_guard<std::mutex> lock(m_mutex);
tmp = m_instance.load(std::memory_order_relaxed);
if (tmp == nullptr) {
tmp = new Singleton;
m_instance = tmp, std;
std::atomic_thread_fence(std::memory_order_release);
}
}
return tmp;
}
```

if yes, then why is the first one better. Also, how is it that release and acquire synchronizes means the the atomic statements before the acquire and after the release will also synchronize?

Reply **1 reply** · *active 351 weeks ago*

Jeff Preshing · *351 weeks ago*

That's not a valid DCLP implementation, because:
1. Your fences are out of place. As a result, there is no *synchronizes-with* relationship anywhere. One loose way of understanding it: A write-release *synchronizes-with* a read-acquire if it sees the write, but an acquire fence can only promote a *preceding* read to a read-acquire, and a release fence can only promote a *following* write to a write-release.
2. Singleton::m_instance ought to be wrapped in a std::atomic<>, like the examples in this post, to ensure it's always manipulated atomically regardless which platform you're compiling for.

To answer your second question, we don't tend to say that the statements before the acquire and after the release *synchronize-with* each other, but they do *happen-before* each other, because *happens-before* is transitive. Review my previous three posts for more background on that.

Reply

alexxys · *347 weeks ago*

One more great chapter in your great online book!

I found one minor typo:
Here http://preshing.com/images/two-cones-dclp.png in the line
m_instance.store(1, std::memory_order_relaxed);
you have to use tmp instead of 1.

Jeff, thank you very much for your great blog!

Reply **1 reply** · *active 345 weeks ago*

Jeff Preshing · *345 weeks ago*

I've corrected the diagram. Thanks!

Reply

Rob Baker · *345 weeks ago*

Sadly, the really neat, simple example of 'Using a C++11 Static Initializer' doesn't work in VS2012 SP4, despite my yearning for it to do so. A fraught debugging session tells me that we're waiting for Microsoft to play catchup with the C++11 standard.

Reply **1 reply** · *active 344 weeks ago*

Jeff Preshing · *344 weeks ago*

Thanks for the heads up! I updated the post to mention it.

Reply

Igipit · *342 weeks ago*

May be vs2012 Doesn't work with the static singleton, but I used it massively since 2003 with out problems on vs2003, vs2005, vs2008, vs2010 and Now with vs2013. I dont remember where i saw it the first time, may be in Herb Sutter book.

Reply **1 reply** · *active 337 weeks ago*

Rob Baker · *337 weeks ago*

Did you try it with multiple threads all hitting it at the same time? It's the concurrent access that screwed up for me, though in a single-threaded environment it functioned well enough - as you'd expect.

Reply

Ankur · *321 weeks ago*

Thanks for this informative article.

However I have a doubt. In the example for "Using C++11 Acquire and Release Fences"

what if 1st thread is executing :-
tmp = new Singleton;

and as main problem of DCLP, it gets suspended by CPU due to which tmp point to memory which has not been constructed. This can happen since tmp is NOT atomic (only m_instance is atomic.)

wont this give rise to same DCLP problem as 2nd thread access memory which has not been constructed yet.

How std::atomic_thread_fence(std::memory_order_acquire) solve this issue as we have not yet called std::atomic_thread_fence(std::memory_order_release) in 1st thread. Does 2nd thread keeps waiting till release operation is complete. Then it looks like its mutex locking.

Reply **5 replies** · *active 316 weeks ago*

Jeff Preshing · 321 weeks ago

tmp is a local variable, not a shared variable. Each thread has its own tmp. So it's harmless if the first thread gets suspended at the point you describe. m_instance, the shared variable, simply remains null. That's the only thing the other threads are checking.

If a second thread comes along while the first thread is still suspended on that line, then yes, the second thread will block on std::lock_guard<std::mutex> lock(m_mutex). But the 2nd thread doesn't wait for the 1st thread to complete the release operation. (Acquire/release operations are non-blocking). It waits for the 1st thread to release the mutex by exiting the scope containing the lock_guard.

Reply

Ankur · 320 weeks ago

Thanks Jeff. Thats very useful. I did not notice the local variable part.

Just to clarify. In above scenario. 1st thread is suspended at tmp = new Singleton; while 2nd thread gets stuck at Lock acquisition.

Now thread 1 resumes and m_instance points Singelton object. After release, its available to all other threads as final value.

Now, thread 2 contiune , loads m_instance. It sees thats its not empty. So, it access the singleton object. Great.

But where did Acquire becomes useful for us. Is it beacuse if we dont use acquire fence then "Singleton* tmp = m_instance.load(std::memory_order_relaxed);" and "tmp = m_instance.load(std::memory_order_relaxed);
" might be redorderd ?

Thanks for your help.

Reply

Jeff Preshing · 320 weeks ago

Well, the lock (m_mutex) makes sure that only one thread has write access to m_instance at a time. Looks like you get that part.

The read-acquire is performed by a thread without taking the lock -- it doesn't wait. Therefore, it can run concurrently with another thread that is modifying m_instance. We need there to be a synchronizes-with relationship between those two threads. That's why one performs a write-release, and the other performs a read-acquire.

Reply

Ankur · 320 weeks ago

I have read so many tutorials and articles about acquire & release. Can you guide me exactly how compiler does that "synchronize".

Thanks

Reply

Jeff Preshing · 320 weeks ago

Try This Is Why They Call It a Weakly-Ordered CPU.

Reply

Alexei Kruglikov · *312 weeks ago*

Thank you for article... one question.

Release memory fence (std::atomic_thread_fence(std::memory_order_release);) in 1st C++ 11 implementation ("Using C++11 Acquire and Release Fences") seems excessive to me. For temp is local thread stack variable not in contention from other threads and CPU guarantees m_instance.store() would execute after it for that (same) thread.

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

Jeff Preshing · *311 weeks ago*

Without the release fence, the write to m_instance could become visible to other threads before the contents of the Singleton are visible.

Reply

's avat

Alexei Kruglikov · *311 weeks ago*

I do not see how that is possible, could you elaborate...

These line are lock protected and executed by no more than 1 thread at any time:
tmp = new Singleton;
std::atomic_thread_fence(std::memory_order_release);
m_instance.store(tmp, std::memory_order_relaxed);

For single thread you would not need the fence, hence by the time CPU executes atomic store, singleton should be fully constructed and temp point to it.

Reply

Jeff Preshing · *310 weeks ago*

The lock is only there to prevent creating multiple copies of the singleton. The lock does not help in any way between the thread that creates the singleton and other threads that just *use* the singleton after it's created. Note that once m_instance (the pointer itself) is initialized and visible to other threads, those threads don't even take the lock anymore! In other words, there is concurrent access to m_instance (the pointer itself) between the writer thread and reader threads. That's why we need to concern ourselves with memory ordering.

Also note that "tmp = new Singleton" is a statement that breaks into several steps. It calls the constructor of Singleton, which can (and usually does) write to shared memory: for example, by initializing Singleton::x to 42. On a multiprocessor system, we want to make sure that when m_instance (the pointer itself) becomes visible to other threads, Singleton::x is already visible to those threads, too! That's we need acquire and release semantics.

Even on a uniprocessor system, there is a risk of memory reordering between m_instance and Singleton::x, due to compiler transformations (optimizations).

You should probably read the Meyers-Alexandrescu paper to gain a complete appreciation for the DCLP problem.

Reply

MciprianM · *234 weeks ago*

Microsoft says that the technique described in section "Using a C++11 Static Initializer" works starting with 2015(look for magic statics in: https://msdn.microsoft.com/en-us/library/hh567368.... They link http://www.open-std.org/jtc1/sc22/wg21/docs/paper... which says: "If control enters the declaration concurrently while the object is being initialized, the concurrent execution waits for completion of the initialization."

Reply

AnandK · *219 weeks ago*

Hi,

Thank you so much for an interesting post on DCLP and C++11.
Are there any benefits to swapping out the getInstance function itself to a version without memory barriers once the object has been fully constructed?

Singleton* (*getInstance) = &createInstanceDCLP;

Singleton* getInstanceLightWeight() { return g_Instance; }

Singleton* createInstanceDCLP()
{
// construct object fully via DCLP
// *then*
getInstance = &getInstanceLightWeight; // switch to a light-weight getter
}

Reply [**1 reply**](#) · *active 219 weeks ago*

[Jeff Preshing](#) · *219 weeks ago*

By substituting the direct function call with an indirect one, you've added a pointer lookup that the original didn't have.

Reply

Luis A · *209 weeks ago*

Nice work on the low level. Invalid solution though.

First of all, this singleton implementation requires of the manual coding of the two members (pointer and mutex) whenever a programmer wants to use the pattern. This is cumbersome and error prone, and introduces the following scenario:

Class A uses singleton B in constructor.
Class A is allocated globally, Constructor is called in initialization. Therefore might want to access B singleton instance, which is not yet allocated.
The mutex of B singleton is not yet initialized and program will crash.

Reply

Zou · *208 weeks ago*

Hi,
Thanks for your wonderful article about DCLP. I have some questions as flow:

in your code of the section 'Using C++11 Low-Level Ordering Constraints', m_mutex is a static member variable with constructor, and it will be initialized at run-time rather than compile-time. as i known, c++ does not define the order of initialization for dynamically initialized objects [1], so when we call Singleton::getInstance, we cannot ensure that m_mutex has been initialized.

my question was origin from a discussion about 'multithread safe singleton in c++' on stackoverflow:
http://stackoverflow.com/questions/11711920/how-t...

Wheezil gave a advice in his reply:
'To get around that, you have to use, not a mutex, but a pointer-to-mutex, which is guaranteed to be zero-initialized before CRT initialization starts. Then you would have to use std::atomic::compare_exchange_strong to create and use the mutex'.

But I don't really known how to get it. In my opinion, maybe we can do some small modifications to your code like these:

```
std::atomic<Singleton*> Singleton::m_instance;

Singleton* Singleton::getInstance() {
static std::mutex my_mutex; // now, mutex has became a local static variable. it will get the benefit of C++11
Singleton* tmp = m_instance.load(std::memory_order_acquire);
if (tmp == nullptr) {
std::lock_guard<std::mutex> lock(my_mutex);
tmp = m_instance.load(std::memory_order_relaxed);
if (tmp == nullptr) {
tmp = new Singleton;
m_instance.store(tmp, std::memory_order_release);
}
}
return tmp;
}
```

thanks

Reply [1 reply](#) · *active 207 weeks ago*

Zou · *207 weeks ago*

Hi, Jeff.
Apology for my carelessness! Maybe the problem about initialization of static std::mutex is nonexistent here. When I saw the declaration of std::mutex, I found that it has a constexpr constructor. On the website:
http://en.cppreference.com/w/cpp/thread/mutex/mut...
It wrote:
"Because the default constructor is constexpr, static mutexes are initialized as part of static non-local initialization, before any dynamic non-local initialization begins. This makes it safe to lock a mutex in a constructor of any static object."
So, I think your example code has no problem. Thanks

Reply

Renjie Liu · *207 weeks ago*

Hi:
I think there exists some problems in your example of "Using C++11 Acquire and Release Fences" .
1. "std::atomic_thread_fence(std::memory_order_acquire);" should be placed before "Singleton* tmp = m_instance.load(std::memory_order_relaxed);" and "m_instance.store(tmp, std::memory_order_relaxed);" should be place after " std::atomic_thread_fence(std::memory_order_release);"
2. you do not need to use atomic variables since sequence before is also a happen before rule.

Reply [1 reply](#) · *active 207 weeks ago*

[Jeff Preshing](#) · *207 weeks ago*

That is incorrect. See http://preshing.com/20131125/acquire-and-release-...

Reply

vijay yande · *207 weeks ago*

Making sure that it is initialized forehead at the start of program will eliminate need to single and double locking. But somehow we need to make it sure that at the start of program it is called only once.

Reply

youchao · *204 weeks ago*

Hi, Jeff

Thank you for the excellent articles which helped me to see something I never thought before.

I have a question about thread safe DCLP pattern : In your article, you mentioned for DCLP pattern,
"The guard variable is the singleton pointer, and the payload is the contents of the singleton itself. In this case, the payload is considered a data dependency of the guard pointer."

which means it is correct to use consume instead of acquire in DCLP, something like the following:

```
T& Singleton()
{
T* tmp = instance.load(std::memory_order_consume);
if (nullptr == tmp)
{
std::unique_lock lock(m);
tmp = instance.load(std::memory_order_relaxed);
if (nullptr == tmp)
{
instance.store(new X(), std::memory_order_release);
}
}
return *instance;
}
```

My question is :

From the look of language statement itself, it looks like only the guard (singleton pointer) carries-a-dependency; not the content of the singleton (the class member of singleton). Why can't I think it's only the pointer is synchronized not the content in this case if consume is used instead of acquire.

Reply [1 reply](#) · *active 204 weeks ago*

> [Jeff Preshing](#) · *204 weeks ago*
>
> The caller will access some member variables of the return value. When it does, the load-consume in your sample code will carry-a-dependency to those member variable accesses.
>
> Reply

[Arctic](#) · *183 weeks ago*

Getting rid of the mutex (though this is not longer DCLP):

```
std::atomic<Singleton*> Singleton::m_instance;

Singleton* Singleton::getInstance() {
Singleton* current_instance;
Singleton* new_instance;

current_instance = m_instance.load(std::memory_order_relaxed);
if(current_instance) return current_instance;

new_instance = new Singleton;

if( m_instance.compare_exchange_strong(current_instance,new_instance,std::memory_order_relaxed))
```

return new_instance:

delete new_instance;

return m_instance.load(std::memory_order_relaxed);
}

Reply

Jeff Buxton · *177 weeks ago*

Is the static initializer technique usable in Visual Studio 2013 ?

Reply **1 reply** · *active 175 weeks ago*

Anu · *175 weeks ago*

Static initializer technique is not usable in VS2013

Reply

Anu · *175 weeks ago*

Thanks Jeff for the nice article.

I found one issue while implementing the singleton in a static library. While unloading one of the dlls which uses the singleton class, the static mutex used in the singleton class is unloaded automatically, which leads to access violation while accessing the singleton class later.

Reply

gnasher729 · *156 weeks ago*

The "singleton as a static object" has one severe disadvantage: It can lead to crashes when your application exits. When the app exits, the main thread calls the destructors for all static objects in some order, so the singleton will get destructed. At the same time, other threads could still be using the singleton! People rely on singletons to exist as long as their code runs, and with static objects there is a small gap at the end.

In many cases this is not a problem, because a crash one millisecond before the app finishes exiting would be unnoticed. But in some cases it can be fatal.

Reply **1 reply** · *active 156 weeks ago*

Jeff Preshing · *156 weeks ago*

If you want a clean shutdown, join/terminate all your threads first.

Reply

Pete Chapman · *150 weeks ago*

Could you please explain why you chose to use the relaxed memory ordering plus a fence, rather than the default s.c. atomic memory order and no fence. My initial reaction is that you have over-complicated the solution. Thanks!

Reply

Taras · *121 weeks ago*

Thanks for this very useful article!

Can you please clarify why do we need std::memory_order_release
at line
m_instance.store(tmp, std::memory_order_release);
in the example Using C++11 Low-Level Ordering Constraints?

Won't std::memory_order_relaxed be enough, if the next thing we do is unlock the mutex, which should have release semantics as well?

Reply **3 replies** · *active 119 weeks ago*

Jeff Preshing · *121 weeks ago*

Read the Meyers/Alexandrescu paper linked at the top.

Reply

Taras · *120 weeks ago*

Hello Jeff!
Thanks for you reply.

I just read the article, but still am not 100% sure what the answer to my question is.
But now I have one assumption.

If the store to the pointer was relaxed, the compiler could reorder instructions in such a way that the pointer could be modified before the Singleton object was actually constructed on that memory location. So, some other thread could have a possibility to read a non-null pointer to Singleton which has not been constructed yet.
The issue Meyers/Alexandrescu addressed at page 4 of their article.

Can you please confirm or refute whether my assumption is correct?

Reply

Jeff Preshing · *119 weeks ago*

Yes, that's why memory_order_relaxed is not enough.

Reply

AndrewC · *102 weeks ago*

In your final C++11 example using the SC atomics, is there any particular reason to be using a temporary variable instead of just testing an atomic directly. For example

```
std::atomic<bool> init;
.....

if(!init){
std::lock_guard<std::mutex> lock(m_mutex);
if(!init){
... etc...
init=true;
}
}
```

Reply

Recent Posts

- I/O in Plywood
- A New Cross-Platform Open Source C++ Framework
- A Flexible Reflection System in C++: Part 2
- A Flexible Reflection System in C++: Part 1
- How to Write Your Own C++ Game Engine
- Can Reordering of Release/Acquire Operations Introduce Deadlock?
- 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
- 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

Patreon

Plywood is a versatile cross-platform open source C++ framework.

Consider supporting its ongoing development on Patreon!