

C++11 introduced a standardized memory model. What does it mean? And how is it going to affect C++ programming?

Asked 13 years, 6 months ago Modified 3 months ago

Viewed 300k times



2218

C++11 introduced a standardized memory model, but what exactly does that mean? And how is it going to affect C++ programming?



[This article](#) (by **Gavin Clarke** who quotes [Herb Sutter](#)) says that,



The memory model means that C++ code now has a standardized library to call regardless of who made the compiler and on what platform it's running. There's a standard way to control how different threads talk to the processor's memory.

"When you are talking about splitting [code] across different cores that's in the standard, we are talking about the memory model. We are going to optimize it without breaking the following assumptions people are going to make in the code," **Sutter** said.

Well, I can *memorize* this and similar paragraphs available online (as I've had my own memory model since birth :P) and can even post as an answer to questions asked by others, but to be honest, I don't exactly understand this.

C++ programmers used to develop multi-threaded applications even before, so how does it matter if it's POSIX threads, or Windows threads, or C++11 threads? What are the benefits? I want to understand the low-level details.

I also get this feeling that the C++11 memory model is somehow related to C++11 multi-threading support, as I often see these two together. If it is, how exactly? Why should they be related?

I don't know how the internals of multi-threading work, and what memory model means in general.

c++

multithreading

c++11

language-lawyer

memory-model

Share

Improve this question

Follow

edited Jun 9, 2022 at 11:31



Stephen Ostermiller ♦


25.5k ● 16 ● 94 ● 114

asked Jun 11, 2011 at 23:30



Sarfaraz Nawaz

361k ● 120 ● 676 ● 860

-
- 8 @curiousguy: Write a blog then...and propose a fix as well. There is no other way to make your point valid and rationale.
– [Sarfaraz Nawaz](#) Jul 27, 2019 at 19:12
-
- 7 I mistook that site as a place to ask Q and exchange ideas. My bad; it's place for conformity where you can't disagree with Herb Sutter even when he flagrantly contradicts himself about throw spec. – [curiousguy](#) Jul 27, 2019 at 19:46
-
- 17 @curiousguy: C++ is what the Standard says, not what a random guy on the internet says. So yes, there has to be *conformity* with the Standard. C++ is NOT an open philosophy where you can talk about anything which does not *conform* to the Standard. – [Sarfaraz Nawaz](#) Jul 28, 2019 at 19:50 
-
- 10 *"I proved that no C++ program can have well defined behavior."*. Tall claims, without any proof!
– [Sarfaraz Nawaz](#) Jul 29, 2019 at 6:30
-
- 3 No. I've not deleted any question or answer. Anyway, the primitives has certain guarantees, right? If so, then you compose bigger guarantees built on those primitive guarantees. Anyway, do you think it is a problem in C++ (and probably C too) only, or it is a problem in ALL languages?
– [Sarfaraz Nawaz](#) Jul 29, 2019 at 9:02
-

9 Answers

Sorted by:

Highest score (default)





2566



+250



First, you have to learn to think like a Language Lawyer.

The C++ specification does not make reference to any particular compiler, operating system, or CPU. It makes reference to an *abstract machine* that is a generalization of actual systems. In the Language Lawyer world, the job of the programmer is to write code for the abstract machine; the job of the compiler is to actualize that code on a concrete machine. By coding rigidly to the spec, you can be certain that your code will compile and run without modification on any system with a compliant C++ compiler, whether today or 50 years from now.

The abstract machine in the C++98/C++03 specification is fundamentally single-threaded. So it is not possible to write multi-threaded C++ code that is "fully portable" with respect to the spec. The spec does not even say anything about the *atomicity* of memory loads and stores or the *order* in which loads and stores might happen, never mind things like mutexes.

Of course, you can write multi-threaded code in practice for particular concrete systems – like pthreads or Windows. But there is no *standard* way to write multi-threaded code for C++98/C++03.

The abstract machine in C++11 is multi-threaded by design. It also has a well-defined *memory model*; that is, it says what the compiler may and may not do when it comes to accessing memory.

Consider the following example, where a pair of global variables are accessed concurrently by two threads:

```
Global
int x, y;

Thread 1          Thread 2
x = 17;           cout << y << " ";
y = 37;           cout << x << endl;
```

What might Thread 2 output?

Under C++98/C++03, this is not even Undefined Behavior; the question itself is *meaningless* because the standard does not contemplate anything called a "thread".

Under C++11, the result is Undefined Behavior, because loads and stores need not be atomic in general. Which may not seem like much of an improvement... And by itself, it's not.

But with C++11, you can write this:

```
Global
atomic<int> x, y;

Thread 1          Thread 2
x.store(17);       cout << y.load() << " ";
y.store(37);       cout << x.load() << endl;
```

Now things get much more interesting. First of all, the behavior here is *defined*. Thread 2 could now print `0 0` (if it runs before Thread 1), `37 17` (if it runs after Thread

1), or `0 17` (if it runs after Thread 1 assigns to x but before it assigns to y).

What it cannot print is `37 0`, because the default mode for atomic loads/stores in C++11 is to enforce *sequential consistency*. This just means all loads and stores must be "as if" they happened in the order you wrote them within each thread, while operations among threads can be interleaved however the system likes. So the default behavior of atomics provides both *atomicity* and *ordering* for loads and stores.

Now, on a modern CPU, ensuring sequential consistency can be expensive. In particular, the compiler is likely to emit full-blown memory barriers between every access here. But if your algorithm can tolerate out-of-order loads and stores; i.e., if it requires atomicity but not ordering; i.e., if it can tolerate `37 0` as output from this program, then you can write this:

```
Global
atomic<int> x, y;

Thread 1                                Thread 2
x.store(17, memory_order_relaxed);      cout << y.load(memory_order_relaxed);
";
y.store(37, memory_order_relaxed);      cout << x.load(memory_order_relaxed);
endl;
```

The more modern the CPU, the more likely this is to be faster than the previous example.

Finally, if you just need to keep particular loads and stores in order, you can write:

```
Global
atomic<int> x, y;

Thread 1                                Thread 2
x.store(17, memory_order_release);      cout << y.load(memory_order_relaxed);
";
y.store(37, memory_order_release);      cout << x.load(memory_order_relaxed);
endl;
```

This takes us back to the ordered loads and stores – so `37 0` is no longer a possible output – but it does so with minimal overhead. (In this trivial example, the result is the same as full-blown sequential consistency; in a larger program, it would not be.)

Of course, if the only outputs you want to see are `0 0` or `37 17`, you can just wrap a mutex around the original code. But if you have read this far, I bet you already know how that works, and this answer is already longer than I intended :-).

So, bottom line. Mutexes are great, and C++11 standardizes them. But sometimes for performance reasons you want lower-level primitives (e.g., the classic [double-checked locking pattern](#)). The new standard provides high-level gadgets like mutexes and condition variables, and it also provides low-level gadgets like atomic types and the various flavors of memory barrier. So now you can write sophisticated, high-performance concurrent routines entirely within the language specified

by the standard, and you can be certain your code will compile and run unchanged on both today's systems and tomorrow's.

Although to be frank, unless you are an expert and working on some serious low-level code, you should probably stick to mutexes and condition variables. That's what I intend to do.

For more on this stuff, see [this blog post](#).

Share Improve this answer

edited Jun 16, 2021 at 23:31

Follow

answered Jun 12, 2011 at 0:23



Nemo

71.4k ● 10 ● 122 ● 159

45 Nice answer, but this is really begging for some actual examples of the new primitives. Also, I think the memory ordering without primitives is the same as pre-C++0x: there are no guarantees. – [John Ripley](#) Jun 12, 2011 at 0:37

54 @Nawaz: Yes! Memory accesses can get reordered by the compiler or CPU. Think about (e.g.) caches and speculative loads. The order in which system memory gets hit can be nothing like what you coded. The compiler and CPU will ensure such reorderings do not break *single-threaded* code. For multi-threaded code, the "memory model" characterizes the possible re-orderings, and what happens if two threads read/write the same location at the same time, and how you exert control over both. For single-threaded code, the memory model is irrelevant. – [Nemo](#) Jun 12, 2011 at 17:08

- 29 @Nawaz, @Nemo - A minor detail: the new memory model is relevant in single-threaded code insofar as it specifies the undefinedness of certain expressions, such as `i = i++`. The old concept of *sequence points* has been discarded; the new standard specifies the same thing using a *sequenced-before* relation which is just a special case of the more general inter-thread *happens-before* concept. – [JohannesD](#) Jun 13, 2011 at 13:14 ✎
-
- 21 @AJG85: Section 3.6.2 of the draft C++0x spec says, "Variables with static storage duration (3.7.1) or thread storage duration (3.7.2) shall be zero-initialized (8.5) before any other initialization takes place." Since x,y are global in this example, they have static storage duration and therefore will zero-initialized, I believe. – [Nemo](#) Jun 13, 2011 at 20:16
-
- 9 @Bemipefe: No, the compiler is not obliged to translate your code in the same order you wrote it - it is allowed to re-order operations, provided the overall effect is the same. It might do this, for example, because reordering allows it to produce faster (or smaller) code. – [psmeares](#) Nov 21, 2017 at 16:13
-



394



+50



I will just give the analogy with which I understand memory consistency models (or memory models, for short). It is inspired by Leslie Lamport's seminal paper ["Time, Clocks, and the Ordering of Events in a Distributed System"](#). The analogy is apt and has fundamental significance, but may be overkill for many people. However, I hope it provides a mental image (a pictorial representation) that facilitates reasoning about memory consistency models.

Let's view the histories of all memory locations in a space-time diagram in which the horizontal axis

represents the address space (i.e., each memory location is represented by a point on that axis) and the vertical axis represents time (we will see that, in general, there is not a universal notion of time). The history of values held by each memory location is, therefore, represented by a vertical column at that memory address. Each value change is due to one of the threads writing a new value to that location. By a **memory image**, we will mean the aggregate/combination of values of all memory locations observable **at a particular time** by **a particular thread**.

Quoting from ["A Primer on Memory Consistency and Cache Coherence"](#)

The intuitive (and most restrictive) memory model is sequential consistency (SC) in which a multithreaded execution should look like an interleaving of the sequential executions of each constituent thread, as if the threads were time-multiplexed on a single-core processor.

That global memory order can vary from one run of the program to another and may not be known beforehand. The characteristic feature of SC is the set of horizontal slices in the address-space-time diagram representing **planes of simultaneity** (i.e., memory images). On a given plane, all of its events (or memory values) are simultaneous. There is a notion of *Absolute Time*, in which all threads agree on which memory values are simultaneous. In SC, at every time instant, there is only

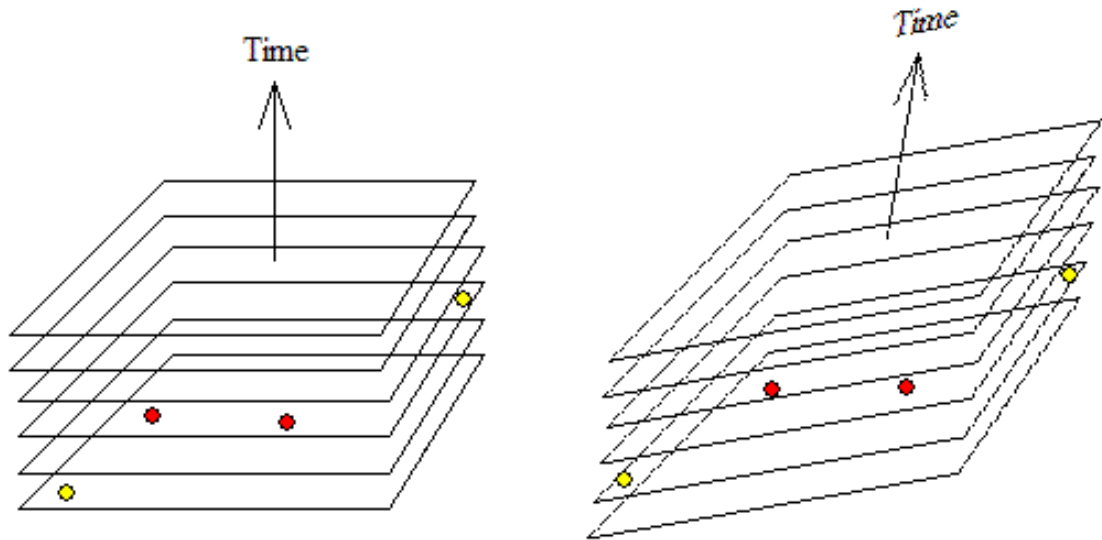
one memory image shared by all threads. That's, at every instant of time, all processors agree on the memory image (i.e., the aggregate content of memory). Not only does this imply that all threads view the same sequence of values for all memory locations, but also that all processors observe the same *combinations of values* of all variables. This is the same as saying all memory operations (on all memory locations) are observed in the same total order by all threads.

In relaxed memory models, each thread will slice up address-space-time in its own way, the only restriction being that slices of each thread shall not cross each other because all threads must agree on the history of every individual memory location (of course, slices of different threads may, and will, cross each other). There is no universal way to slice it up (no privileged foliation of address-space-time). Slices do not have to be planar (or linear). They can be curved and this is what can make a thread read values written by another thread out of the order they were written in. Histories of different memory locations may slide (or get stretched) arbitrarily relative to each other ***when viewed by any particular thread***.

Each thread will have a different sense of which events (or, equivalently, memory values) are simultaneous. The set of events (or memory values) that are simultaneous to one thread are not simultaneous to another. Thus, in a relaxed memory model, all threads still observe the same history (i.e., sequence of values) for each memory location. But they may observe different memory images (i.e., combinations of values of all memory locations).

Even if two different memory locations are written by the same thread in sequence, the two newly written values may be observed in different order by other threads.

[Picture from Wikipedia]



Readers familiar with Einstein's **Special Theory of Relativity** will notice what I am alluding to. Translating Minkowski's words into the memory models realm: address space and time are shadows of address-space-time. In this case, each observer (i.e., thread) will project shadows of events (i.e., memory stores/loads) onto his own world-line (i.e., his time axis) and his own plane of simultaneity (his address-space axis). Threads in the C++11 memory model correspond to **observers** that are moving relative to each other in special relativity.

Sequential consistency corresponds to the **Galilean space-time** (i.e., all observers agree on one absolute order of events and a global sense of simultaneity).

The resemblance between memory models and special relativity stems from the fact that both define a partially-ordered set of events, often called a causal set. Some

events (i.e., memory stores) can affect (but not be affected by) other events. A C++11 thread (or observer in physics) is no more than a chain (i.e., a totally ordered set) of events (e.g., memory loads and stores to possibly different addresses).

In relativity, some order is restored to the seemingly chaotic picture of partially ordered events, since the only temporal ordering that all observers agree on is the ordering among “timelike” events (i.e., those events that are in principle connectible by any particle going slower than the speed of light in a vacuum). Only the timelike related events are invariantly ordered. [Time in Physics, Craig Callender](#).

In C++11 memory model, a similar mechanism (the acquire-release consistency model) is used to establish these ***local causality relations***.

To provide a definition of memory consistency and a motivation for abandoning SC, I will quote from ["A Primer on Memory Consistency and Cache Coherence"](#)

For a shared memory machine, the memory consistency model defines the architecturally visible behavior of its memory system. The correctness criterion for a single processor core partitions behavior between “*one correct result*” and “*many incorrect alternatives*”. This is because the processor’s architecture mandates that the execution of a thread transforms a given

input state into a single well-defined output state, even on an out-of-order core. Shared memory consistency models, however, concern the loads and stores of multiple threads and usually allow *many correct executions* while disallowing many (more) incorrect ones. The possibility of multiple correct executions is due to the ISA allowing multiple threads to execute concurrently, often with many possible legal interleavings of instructions from different threads.

Relaxed or **weak** memory consistency models are motivated by the fact that most memory orderings in strong models are unnecessary. If a thread updates ten data items and then a synchronization flag, programmers usually do not care if the data items are updated in order with respect to each other but only that all data items are updated before the flag is updated (usually implemented using FENCE instructions).

Relaxed models seek to capture this increased ordering flexibility and preserve only the orders that programmers “*require*” to get both higher performance and correctness of SC. For example, in certain architectures, FIFO write buffers are used by each core to hold the results of committed (retired) stores before writing the results to the caches. This optimization enhances performance but violates SC. The write buffer hides the latency of servicing a store miss. Because stores are common, being able to

avoid stalling on most of them is an important benefit. For a single-core processor, a write buffer can be made architecturally invisible by ensuring that a load to address A returns the value of the most recent store to A even if one or more stores to A are in the write buffer. This is typically done by either bypassing the value of the most recent store to A to the load from A, where “most recent” is determined by program order, or by stalling a load of A if a store to A is in the write buffer. When multiple cores are used, each will have its own bypassing write buffer. Without write buffers, the hardware is SC, but with write buffers, it is not, making write buffers architecturally visible in a multicore processor.

Store-store reordering may happen if a core has a non-FIFO write buffer that lets stores depart in a different order than the order in which they entered. This might occur if the first store misses in the cache while the second hits or if the second store can coalesce with an earlier store (i.e., before the first store). Load-load reordering may also happen on dynamically-scheduled cores that execute instructions out of program order. That can behave the same as reordering stores on another core (Can you come up with an example interleaving between two threads?). Reordering an earlier load with a later store (a load-store reordering) can cause many incorrect behaviors, such as loading a value after

releasing the lock that protects it (if the store is the unlock operation). Note that store-load reorderings may also arise due to local bypassing in the commonly implemented FIFO write buffer, even with a core that executes all instructions in program order.

Because cache coherence and memory consistency are sometimes confused, it is instructive to also have this quote:

Unlike consistency, **cache coherence** is neither visible to software nor required. Coherence seeks to make the caches of a shared-memory system as functionally invisible as the caches in a single-core system. Correct coherence ensures that a programmer cannot determine whether and where a system has caches by analyzing the results of loads and stores. This is because correct coherence ensures that the caches never enable new or different **functional** behavior (programmers may still be able to infer likely cache structure using **timing** information). The main purpose of cache coherence protocols is maintaining the single-writer-multiple-readers (SWMR) invariant for every memory location. An important distinction between coherence and consistency is that coherence is specified on a **per-memory location basis**, whereas

consistency is specified with respect to *all* memory locations.

Continuing with our mental picture, the SWMR invariant corresponds to the physical requirement that there be at most one particle located at any one location but there can be an unlimited number of observers of any location.

Share Improve this answer

Follow

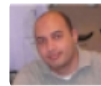
edited Jun 20, 2020 at 9:12



Community Bot

1 • 1

answered Aug 29, 2013 at 20:42



Ahmed Nassar

4,793 • 2 • 21 • 26

61 +1 for the analogy with special relativity, I've been trying to make the same analogy myself. Too often I see programmers investigating threaded code trying to interpret the behavior as operations in different threads occurring interleaved with one another in a specific order, and I have to tell them, nope, with multi-processor systems the notion of simultaneity between different <s>frames of reference</s> threads is now meaningless. Comparing with special relativity is a good way to make them respect the complexity of the problem. – [Pierre Lebeaupin](#) Jun 26, 2014 at 19:42

92 So should you conclude that the Universe is multicore? – [Peter K](#) Apr 28, 2015 at 11:36

6 @PeterK: Exactly :) And here is a very nice visualization of this picture of time by physicist Brian Greene: [youtube.com/watch?v=4BjGWLJNPcA&t=22m12s](https://www.youtube.com/watch?v=4BjGWLJNPcA&t=22m12s) This is "The Illusion of Time [Full Documentary]" at minute 22 and 12 seconds. – [Ahmed Nassar](#) Jul 19, 2015 at 2:17 ✎

4 Is it just me or is he switching from a 1D memory model (horizontal axis) to a 2D memory model (planes of simultaneity). I find this a bit confusing but maybe that is because I am not a native speaker... Still a very interesting read. – [Kami Kaze](#) Jan 12, 2017 at 11:31

12 I lived to see relativity being used as a simplified analogy... – [Szczepan Hołyszewski](#) Nov 1, 2021 at 16:18



145



This is now a multiple-year old question, but being very popular, it's worth mentioning a fantastic resource for learning about the C++11 memory model. I see no point in summing up his talk in order to make this yet another full answer, but given this is the guy who actually wrote the standard, I think it's well worth watching the talk.

Herb Sutter has a three hour long talk about the C++11 memory model titled "atomic<> Weapons", available on the ~~Channel9~~ site YouTube - [part 1](#) and [part 2](#). The talk is pretty technical, and covers the following topics:

1. Optimizations, Races, and the Memory Model
2. Ordering – What: Acquire and Release
3. Ordering – How: Mutexes, Atomics, and/or Fences
4. Other Restrictions on Compilers and Hardware
5. Code Gen & Performance: x86/x64, IA64, POWER, ARM
6. Relaxed Atomics

The talk doesn't elaborate on the API, but rather on the reasoning, background, under the hood and behind the

scenes (did you know relaxed semantics were added to the standard only because POWER and ARM do not support synchronized load efficiently?).

Share Improve this answer

edited Jul 10, 2022 at 13:46

Follow

answered Dec 20, 2013 at 13:22



Eran

22k ● 6 ● 58 ● 91

4 @eran do you guys happen to have the slides? links on the channel 9 talk pages do not work. – [athos](#) Aug 30, 2016 at 2:33

2 @athos I don't have them, sorry. Try contacting channel 9, I don't think the removal was intentional (my guess is that they got the link from Herb Sutter, posted as is, and he later removed the files; but that's just a speculation...). – [Eran](#) Aug 30, 2016 at 6:06

2 Channel 9 went dead, and an available source is YouTube : [part1](#) and [part2](#). The slides can be downloaded [here](#). – [o_oTurtle](#) Jul 8, 2022 at 12:51

Thanks @o_oTurtle! fixed the links in the answer. Feel free to make such fixes yourself whenever you come across them. – [Eran](#) Jul 10, 2022 at 13:48



81

It means that the standard now defines multi-threading, and it defines what happens in the context of multiple threads. Of course, people used varying implementations,



but that's like asking why we should have a `std::string` when we could all be using a home-rolled `string` class.



When you're talking about POSIX threads or Windows threads, then this is a bit of an illusion as actually you're talking about x86 threads, as it's a hardware function to run concurrently. The C++0x memory model makes guarantees, whether you're on x86, or ARM, or [MIPS](#), or anything else you can come up with.

Share Improve this answer

Follow

edited Nov 5, 2017 at 23:06



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Jun 11, 2011 at 23:42



Puppy

147k ● 40 ● 266 ● 477

35 Posix threads are not restricted to x86. Indeed, the first systems they were implemented on were probably not x86 systems. Posix threads are system-independent, and are valid on all Posix platforms. It's also not really true that it's a hardware property because Posix threads can also be implemented through cooperative multitasking. But of course most threading issues only surface on hardware threading implementations (and some even only on multiprocessor/multicore systems). – [celtschk](#) Aug 18, 2013 at 19:56

This answer is really evasive.. – [kingsjester](#) Aug 14, 2022 at 14:36



60



For languages not specifying a memory model, you are writing code for the language *and* the memory model specified by the processor architecture. The processor may choose to re-order memory accesses for performance. So, **if your program has data races** (a data race is when it's possible for multiple cores / hyper-threads to access the same memory concurrently) then your program is not cross platform because of its dependence on the processor memory model. You may refer to the Intel or AMD software manuals to find out how the processors may re-order memory accesses.

Very importantly, locks (and concurrency semantics with locking) are typically implemented in a cross platform way... So if you are using standard locks in a multithreaded program with no data races then you **don't have to worry about cross platform memory models**.

Interestingly, Microsoft compilers for C++ have acquire / release semantics for volatile which is a C++ extension to deal with the lack of a memory model in C++
[http://msdn.microsoft.com/en-us/library/12a04hfd\(v=vs.80\).aspx](http://msdn.microsoft.com/en-us/library/12a04hfd(v=vs.80).aspx). However, given that Windows runs on x86 / x64 only, that's not saying much (Intel and AMD memory models make it easy and efficient to implement acquire / release semantics in a language).

Share Improve this answer

edited Nov 5, 2017 at 23:09

Follow



Peter Mortensen

31.6k ● 22 ● 109 ● 133

answered Jul 26, 2011 at 4:27



ritesh

982 ● 5 ● 5

-
- 3 It is true that, when the answer was written, Windows run on x86/x64 only, but Windows run, at some point in time, on IA64, MIPS, Alpha AXP64, PowerPC and ARM. Today it runs on various versions of ARM, which is quite different memory wise from x86, and nowhere nearly as forgiving.

– [Lorenzo Dematté](#) Dec 6, 2016 at 10:12

That link is somewhat broken (says "*Visual Studio 2005 Retired documentation*"). Care to update it?

– [Peter Mortensen](#) Nov 5, 2017 at 23:09

-
- 4 It was not true even when the answer was written. – [Ben](#) Dec 2, 2017 at 10:14

"to access the same memory concurrently" to access in a **conflicting** way – [curiousguy](#) Jun 13, 2018 at 23:22



32

If you use mutexes to protect all your data, you really shouldn't need to worry. Mutexes have always provided sufficient ordering and visibility guarantees.



Now, if you used atomics, or lock-free algorithms, you need to think about the memory model. The memory model describes precisely when atomics provide ordering and visibility guarantees, and provides portable fences for hand-coded guarantees.



Previously, atomics would be done using compiler intrinsics, or some higher level library. Fences would have

been done using CPU-specific instructions (memory barriers).

Share Improve this answer

answered Jun 11, 2011 at 23:49

Follow



ninjalj

43.6k ● 11 ● 110 ● 150

22 The problem before was that there was not such thing as a mutex (in terms of the C++ standard). So the only guarantees you were provided were by the mutex manufacturer, which was fine as long as you did not port the code (as minor changes to guarantees are hard to spot). Now we are get guarantees provided by the standard which should be portable between platforms. – [Loki Astari](#) Jun 12, 2011 at 0:09

4 @Martin: in any case, one thing is the memory model, and another are the atomics and threading primitives that run on top of that memory model. – [ninjalj](#) Jun 12, 2011 at 0:18

4 Also, my point was mostly that previously there was mostly no memory model at the language level, it happened to be the memory model of the underlying CPU. Now there is a memory model which is part of the core language; OTOH, mutexes and the like could always be done as a library. – [ninjalj](#) Jun 12, 2011 at 0:36

3 It could also be a real problem for the people trying to *write* the mutex library. When the CPU, the memory controller, the kernel, the compiler, and the "C library" are all implemented by different teams, and some of them are in violent disagreement as to how this stuff is supposed to work, well, sometimes the stuff we systems programmers have to do to present a pretty facade to the applications level is not pleasant at all. – [zwol](#) Jun 12, 2011 at 2:02

12 Unfortunately it is not enough to guard your data structures with simple mutexes if there is not a consistent memory

model in your language. There are various compiler optimizations which make sense in a single threaded context but when multiple threads and cpu cores come into play, reordering of memory accesses and other optimizations may yield undefined behavior. For more information see "Threads cannot be implemented as a library" by Hans Boehm:

citeseer.ist.psu.edu/viewdoc/... – exDM69 Jun 13, 2011 at 12:45 



12



Some of the other answers get at the most fundamental aspects of the C++ memory model. In practice, most uses of `std::atomic<>` "just work", at least until the programmer over-optimizes (e.g., by trying to relax too many things).



There is one place where mistakes are still common: *sequence locks*. There is an excellent and easy-to-read discussion of the challenges at <https://www.hpl.hp.com/techreports/2012/HPL-2012-68.pdf>. Sequence locks are appealing because the reader avoids writing to the lock word. The following code is based on Figure 1 of the above technical report, and it highlights the challenges when implementing sequence locks in C++:

```
atomic<uint64_t> seq; // seqlock representation
int data1, data2;    // this data will be protected b

T reader() {
    int r1, r2;
    unsigned seq0, seq1;
    while (true) {
        seq0 = seq;
        r1 = data1; // INCORRECT! Data Race!
```



```

        r2 = data2; // INCORRECT!
        seq1 = seq;

        // if the lock didn't change while I was reading, then
        // the lock wasn't held while I was reading, then
        // reads should be valid
        if (seq0 == seq1 && !(seq0 & 1))
            break;
    }
    use(r1, r2);
}

void writer(int new_data1, int new_data2) {
    unsigned seq0 = seq;
    while (true) {
        if ((!(seq0 & 1)) && seq.compare_exchange_weak(
            seq0, seq0 + 1)) break; // atomically moving the lock from
    }
    data1 = new_data1;
    data2 = new_data2;
    seq = seq0 + 2; // release the lock by increasing
}

```

As unintuitive as it seems at first, `data1` and `data2` need to be `atomic<>`. If they are not atomic, then they could be read (in `reader()`) at the exact same time as they are written (in `writer()`). According to the C++ memory model, this is a race *even if* `reader()` *never actually uses the data*. In addition, if they are not atomic, then the compiler can cache the first read of each value in a register. Obviously you wouldn't want that... you want to re-read in each iteration of the `while` loop in `reader()`.

It is also not sufficient to make them `atomic<>` and access them with `memory_order_relaxed`. The reason for this is that the reads of `seq` (in `reader()`) only have *acquire* semantics. In simple terms, if X and Y are

memory accesses, X precedes Y, X is not an acquire or release, and Y is an acquire, then the compiler can reorder Y before X. If Y was the second read of seq, and X was a read of data, such a reordering would break the lock implementation.

The paper gives a few solutions. The one with the best performance today is probably the one that uses an `atomic_thread_fence` with `memory_order_relaxed` *before* the second read of the seqlock. In the paper, it's Figure 6. I'm not reproducing the code here, because anyone who has read this far really ought to read the paper. It is more precise and complete than this post.

The last issue is that it might be unnatural to make the `data` variables atomic. If you can't in your code, then you need to be very careful, because casting from non-atomic to atomic is only legal for primitive types. C++20 is supposed to add `atomic_ref<>`, which will make this problem easier to resolve.

To summarize: even if you think you understand the C++ memory model, you should be very careful before rolling your own sequence locks.

Share Improve this answer

Follow

edited Jul 10, 2023 at 15:12



Sarfaraz Nawaz

361k ● 120 ● 676 ● 860

answered Dec 20, 2019 at 3:56



Mike Spear

882 ● 9 ● 13



0



With memory model in C++ , programmers have been provided with the abstraction layer of underlying machine. Earlier for C++ (pre C++11) , we need posix threads/boost threads (3rd party libraries) to perform multithreading in C++. But now it is fairly possible in C++.



Share Improve this answer

answered Aug 29 at 13:18



Follow



ravi

19 ● 3



-5



C and C++ used to be defined by an execution trace of a well formed program.

Now they are half defined by an execution trace of a program, and half a posteriori by many orderings on synchronisation objects.



Meaning that these language definitions make no sense at all as no logical method to mix these two approaches. In particular, destruction of a mutex or atomic variable is not well defined.

Share Improve this answer

answered Jul 28, 2019 at 20:09

Follow



curiousguy

8,234 ● 2 ● 43 ● 60

2 I share your fierce desire for improvement of the language design, but I think your answer would be more valuable if it were centered on a simple case, for which you showed clearly and explicitly how that behavior violates specific

language design principles. After that I would strongly recommend you, if you allow me, to give in that answer a very good argumentation for the relevance of each of those points, because they will be contrasted against the relevance of the immense productivity benefits perceived by C++ design – [Matias Haeussler](#) Nov 21, 2019 at 20:33 ✎

- 1 @MatiasHaeussler I think you misread my answer; I'm not objecting to the definition of a particular C++ feature here (I also have many such pointed criticisms but not here). **I'm arguing here that there is no well defined construct in C++ (nor C).** The whole MT semantics are a complete mess, as you don't have sequential semantics anymore. (I believe Java MT is broken but less.) The "simple example" would be almost any MT program. If you disagree, you are welcome to answer my question about [how to prove correctness of MT C++ programs](#). – [curiousguy](#) Nov 21, 2019 at 21:49
-

Interesting, I think I understand more what you mean after reading your question. If I am right you are referring to *the impossibility of developing proofs for C++ MT programs correctness*. In such a case I would say that for me is something of huge importance for the future of computer programming, in particular for the arrival of artificial intelligence. But I would point too that for the great majority of people asking questions in stack overflow that is not something they are even aware of, and even after understanding what you mean and becoming interested – [Matias Haeussler](#) Nov 22, 2019 at 0:17

- 1 "Should questions about the demonstrability of computer programs be posted in stackoverflow or in stackexchange (if in neither, where)?" This one seems to be one for meta stackoverflow, is it not? – [Matias Haeussler](#) Nov 22, 2019 at 0:35
-

- 1 @MatiasHaeussler 1) C and C++ essentially share the "memory model" of atomic variables, mutexes and multithreading. 2) The relevance on this is about the benefits

of having the "memory model". I think the benefit is zero as the model is unsound. – [curiousguy](#) Nov 22, 2019 at 0:56



Highly active question. Earn 10 reputation (not counting the [association bonus](#)) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.