



[Preshing on Programming](#)

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

Navigate...

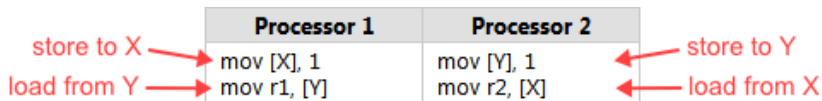
- [Blog](#)
- [Archives](#)
- [About](#)
- [Contact](#)
- [Tip Jar](#)

May 15, 2012

Memory Reordering Caught in the Act

When writing lock-free code in C or C++, one must often take special care to enforce correct memory ordering. Otherwise, surprising things can happen.



Intel lists several such surprises in Volume 3, §8.2.3 of their [x86/64 Architecture Specification](#). Here's one of the simplest examples. Suppose you have two integers *x* and *y* somewhere in memory, both initially 0. Two processors, running in parallel, execute the following machine code:



Don't be thrown off by the use of assembly language in this example. It's really the best way to illustrate CPU ordering. Each processor stores 1 into one of the integer variables, then loads the other integer into a register. (*r1* and *r2* are just placeholder names for actual x86 registers, such as *eax*.)

Now, no matter which processor writes 1 to memory first, it's natural to expect the *other* processor to read that value back, which means we should end up with either *r1* = 1, *r2* = 1, or perhaps both. But according to Intel's specification, that won't necessarily be the case. The specification says it's legal for both *r1* and *r2* to equal 0 at the end of this example – a counterintuitive result, to say the least!

One way to understand this is that Intel x86/64 processors, like most processor families, are allowed to **reorder** the memory interactions of machine instructions according to certain rules, as long it never changes the execution of a single-threaded program. In particular, each processor is allowed to delay the effect of a store past any load from a different location. As a result, it might end up as though the instructions had executed in this order:

Processor 1	Processor 2
 <pre> mov r1, [Y] mov [X], 1 </pre>	 <pre> mov r2, [X] mov [Y], 1 </pre>

Let's Make It Happen

It's all well and good to be told this kind of thing *might* happen, but there's nothing like seeing it with your own eyes. That's why I've written a small sample program to show this type of reordering *actually happening*. You can download the source code [here](#).

The sample comes in both a Win32 version and a POSIX version. It spawns two worker threads which repeat the above transaction indefinitely, while the main thread synchronizes their work and checks each result.

Here's the source code for the first worker thread. `X`, `Y`, `r1` and `r2` are all globals, and POSIX semaphores are used to co-ordinate the beginning and end of each loop.

```

sem_t beginSema1;
sem_t endSema;

int X, Y;
int r1, r2;

void *thread1Func(void *param)
{
    MersenneTwister random(1); // Initialize random number generator
    for (;;) // Loop indefinitely
    {
        sem_wait(&beginSema1); // Wait for signal from main thread
        while (random.integer() % 8 != 0) {} // Add a short, random delay

        // ----- THE TRANSACTION! -----
        X = 1;
        asm volatile("" ::: "memory"); // Prevent compiler reordering
        r1 = Y;

        sem_post(&endSema); // Notify transaction complete
    }
    return NULL; // Never returns
};

```

A short, random delay is added before each transaction in order to stagger the timing of the thread. Remember, there are two worker threads, and we're trying to get their instructions to overlap. The random delay is achieved using the same `MersenneTwister` implementation I've used in previous posts, such as when [measuring lock contention](#) and when [validating that the recursive Benaphore worked](#).

Don't be spooked by the presence of the `asm volatile` line in the above code listing. This is just a directive [telling the GCC compiler not to rearrange the store and the load](#) when generating machine code, just in case it starts to get any funny ideas during optimization. We can verify this by checking the assembly code listing, as seen below. As expected, the store and the load occur in the desired order. The instruction after that writes the resulting register `eax` back to the global variable `r1`.

```

$ gcc -O2 -c -S -masm=intel ordering.cpp
$ cat ordering.s
...
mov     DWORD PTR _X, 1
mov     eax, DWORD PTR _Y
mov     DWORD PTR _r1, eax
...

```

The main thread source code is shown below. It performs all the administrative work.

After initialization, it loops indefinitely, resetting `x` and `y` back to 0 before kicking off the worker threads on each iteration.

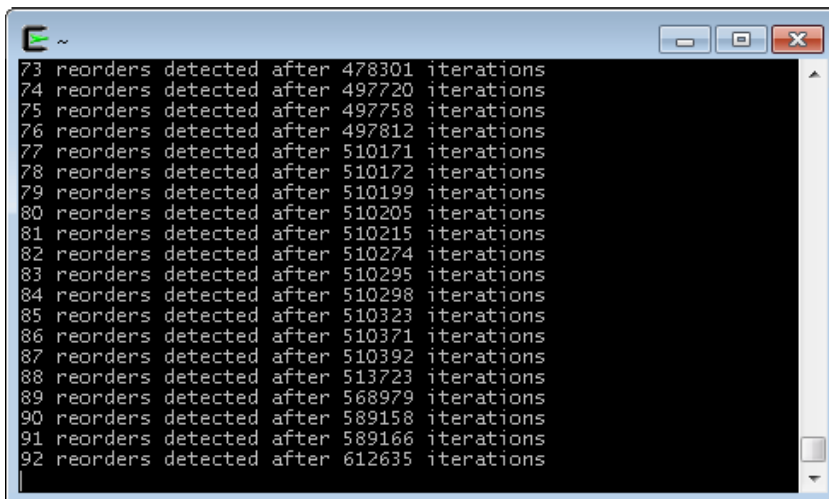
Pay particular attention to the way all writes to shared memory occur before `sem_post`, and all reads from shared memory occur after `sem_wait`. The same rules are followed in the worker threads when communicating with the main thread. Semaphores give us [acquire and release semantics](#) on every platform. That means we are guaranteed that the initial values of `x = 0` and `y = 0` will propagate completely to the worker threads, and that the resulting values of `r1` and `r2` will propagate fully back here. In other words, the semaphores prevent memory reordering issues in the framework, allowing us to focus entirely on the experiment itself!

```
int main()
{
    // Initialize the semaphores
    sem_init(&beginSema1, 0, 0);
    sem_init(&beginSema2, 0, 0);
    sem_init(&endSema, 0, 0);

    // Spawn the threads
    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, thread1Func, NULL);
    pthread_create(&thread2, NULL, thread2Func, NULL);

    // Repeat the experiment ad infinitum
    int detected = 0;
    for (int iterations = 1; ; iterations++)
    {
        // Reset X and Y
        X = 0;
        Y = 0;
        // Signal both threads
        sem_post(&beginSema1);
        sem_post(&beginSema2);
        // Wait for both threads
        sem_wait(&endSema);
        sem_wait(&endSema);
        // Check if there was a simultaneous reorder
        if (r1 == 0 && r2 == 0)
        {
            detected++;
            printf("%d reorders detected after %d iterations\n", detected, iterations);
        }
    }
    return 0; // Never returns
}
```

Finally, the moment of truth. Here's some sample output while running in [Cygwin](#) on an Intel Xeon W3520.



```
73 reorders detected after 478301 iterations
74 reorders detected after 497720 iterations
75 reorders detected after 497758 iterations
76 reorders detected after 497812 iterations
77 reorders detected after 510171 iterations
78 reorders detected after 510172 iterations
79 reorders detected after 510199 iterations
80 reorders detected after 510205 iterations
81 reorders detected after 510215 iterations
82 reorders detected after 510274 iterations
83 reorders detected after 510295 iterations
84 reorders detected after 510298 iterations
85 reorders detected after 510323 iterations
86 reorders detected after 510371 iterations
87 reorders detected after 510392 iterations
88 reorders detected after 513723 iterations
89 reorders detected after 568979 iterations
90 reorders detected after 589158 iterations
91 reorders detected after 589166 iterations
92 reorders detected after 612635 iterations
```

And there you have it! During this run, a memory reordering was detected approximately once every **6600** iterations. When I tested in Ubuntu on a Core 2 Duo

E6300, the occurrences were even more rare. One begins to appreciate how subtle timing bugs can creep undetected into lock-free code.

Now, suppose you wanted to eliminate those reorderings. There are at least two ways to do it. One way is to set thread affinities so that both worker threads run exclusively on the same CPU core. There's no portable way to set affinities with Pthreads, but on Linux it can be accomplished as follows:

```
cpu_set_t cpus;
CPU_ZERO(&cpus);
CPU_SET(0, &cpus);
pthread_setaffinity_np(thread1, sizeof(cpu_set_t), &cpus);
pthread_setaffinity_np(thread2, sizeof(cpu_set_t), &cpus);
```

After this change, the reordering disappears. That's because a single processor never sees its own operations out of order, even when threads are pre-empted and rescheduled at arbitrary times. Of course, by locking both threads to a single core, we've left the other cores unused.

On a related note, I compiled and ran this sample on Playstation 3, and no memory reordering was detected. This suggests (but doesn't confirm) that the [two hardware threads](#) inside the PPU may effectively act as a single processor, with very fine-grained hardware scheduling.

Preventing It With a StoreLoad Barrier

Another way to prevent memory reordering in this sample is to introduce a CPU barrier between the two instructions. Here, we'd like to prevent the effective reordering of a store followed by a load. In common barrier parlance, we need a **StoreLoad** barrier.

On x86/64 processors, there is no specific instruction which acts *only* as a StoreLoad barrier, but there are several instructions which do that and more. The `mfence` instruction is a full memory barrier, which prevents memory reordering of any kind. In GCC, it can be implemented as follows:

```
for (;;)                                // Loop indefinitely
{
    sem_wait(&beginSema1);               // Wait for signal from main thread
    while (random.integer() % 8 != 0) {} // Add a short, random delay

    // ----- THE TRANSACTION! -----
    X = 1;
    asm volatile("mfence" ::: "memory"); // Prevent memory reordering
    r1 = Y;

    sem_post(&endSema);                  // Notify transaction complete
}
```

Again, you can verify its presence by looking at the assembly code listing.

```
...
mov     DWORD PTR _X, 1
mfence
mov     eax, DWORD PTR _Y
mov     DWORD PTR _r1, eax
...
```

With this modification, the memory reordering disappears, and we've still allowed both threads to run on separate CPU cores.

Similar Instructions and Different Platforms

Interestingly, `mfence` isn't the only instruction which acts as a full memory barrier on x86/64. On these processors, any locked instruction, such as `xchg`, also acts as a full memory barrier – provided you don't use SSE instructions or write-combined memory, which this sample doesn't. In fact, the Microsoft C++ compiler generates `xchg` when you use the [MemoryBarrier](#) intrinsic, at least in Visual Studio 2008.

The `mfence` instruction is specific to x86/64. If you want to make the code more portable, you could wrap this intrinsic in a preprocessor macro. The Linux kernel has wrapped it in a macro named `smp_mb`, along with related macros such as `smp_rmb` and `smp_wmb`, and provided [alternate implementations on different architectures](#). For example, on PowerPC, `smp_mb` is implemented as `sync`.

All these different CPU families, each having unique instructions to enforce memory ordering, with each compiler exposing them through different intrinsics, and each cross-platform project implementing its own portability layer... none of this helps simplify lock-free programming! This is partially why the [C++11 atomic library](#) standard was recently introduced. It's an attempt to standardize things, and make it easier to write portable lock-free code.

 12

 63

 Like

 Tweet

 G+


[« How to Remove Camera Shake from iPhone 4S Videos Lightweight In-Memory Logging »](#)

Comments (40)



Logged in as [bgn9000](#)

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





[Matt Diesel](#) · 193 weeks ago

I'm very interested to learn what it is that makes only 1/6600 iterations use this. Surely if there is a benefit to switching the instructions, it should be done so predictably?

Most things like the pipeline act in a predictable fashion, which we can learn, and use.

Reply



 [2 replies](#) · active 123 weeks ago



Drew Amato · 193 weeks ago

I'm not the author (and I'm only pretending to understand all this) but I assume the unpredictability in the results is due to the need for special timing conditions to occur (even when the instruction order is consistently switched): the two calls need to overlap, the two reads need to happen before the two writes.

Reply



[Jeff Preshing](#) · 193 weeks ago

The main reason reordering is detected so rarely in this sample is because the semaphore timing is unpredictable and I introduced randomness to compensate. You could probably detect reordering much more frequently if you use a different synchronization method in the worker threads; for example, busy-waiting on a flag. I chose not to do this in the sample because then, I'd need to ensure correct memory ordering around the flag, and that would draw attention away from the place where I really wanted to demonstrate reordering.

Also, as Bruce Dawson points out below, memory reordering is not necessarily an artifact of CPU

instructions being switched. Memory reordering tends to be more the result of the way CPU caches work: If you are really interested in those details, I'd recommend Appendix C of [Is Parallel Programming Hard](#).

Reply



David Brown · 193 weeks ago

Good article. One minor nit, the gcc Makefile should have the ordering.cpp filename after the colon, not before it. As is, it doesn't rebuild if you edit the source.

Reply [1 reply](#) · active 123 weeks ago



[Jeff Preshing](#) · 193 weeks ago

Ha. Windows guy alert! Fixed now.

Reply



Azeem Jiva · 193 weeks ago

One thing to realize is that mfence is expensive! At least on AMD processors, there are faster ways that get you the same effect as mfence (lock; addl \$0,0(%%esp))

Reply



[Patrick Baggett](#) · 193 weeks ago

Memory ordering on x86 is actually really strong compared to others. You've picked the one example of when x86 `//is//` allowed to reorder memory operations: StoreLoad.

The rest of the cases {StoreStore, LoadLoad, LoadStore} are ordered.

I'm kind of surprised you didn't mention the really common StoreStore case of:

```
value = ptr;
valueAvailable = true;
```

On weakly ordered system, you need a StoreStore barrier or else you can read 'valueAvailable' == true but 'value' == garbage. This is a problem on the Xbox360 and other PowerPC-based systems.

Also, mfence is a really strong memory barrier (it fences all 4 types of relationships). On x86, locked (atomic) operations provide total order and don't allow any other type of operation to be reordered. The "mfence" instruction was introduced with SSE because some of the 16-byte loads/stores (movnt*) don't have the same ordering characteristics -- i.e. they are weakly ordered.

Reply [1 reply](#) · active 123 weeks ago



[Jeff Preshing](#) · 193 weeks ago

All valid points. The fact that this particular reordering can be demonstrated on x86 is exactly why I chose to blog about it. Also, I've updated the post to improve the distinction between mfence and locked instructions like xchg.

Reply



[Bruce Dawson](#) · 193 weeks ago

Nice work -- writing test code to demonstrate this is quite cool.

I'd like to offer one correction and one explanation.

The correction is that reordering of reads and writes is orthogonal to instruction reordering. The Xbox 360 CPU does no instruction reordering, but the underlying reads/writes are reordered. In fact the Intel/AMD out-of-order CPUs do significantly *less* read/write reordering than do the in-order

Xbox 360 CPUs.

In this case the reordering can be seen to inevitable for any multi-core system in a universe where the speed of light is finite. If the two processors write '1' simultaneously then it is unreasonable to expect that they could possibly see each other's write on the next instruction. In fact, it can take dozens of cycles for a write to propagate so that it can be seen by other cores. Any system that prohibited this type of reordering would be unable to run faster than perhaps 100 MHz, maybe even slower, because it would constantly have to wait for signals from the other cores.

Reply  [1 reply](#) · active 123 weeks ago



[Jeff Preshing](#) · 193 weeks ago

Hi Bruce. Indeed, my original wording made it sound like instruction reordering was the sole reason for memory reordering. I've since revised that paragraph. In this post, I don't really want to say too much about the causes of memory reordering -- just the effects.

Reply



Tom Forsyth · 193 weeks ago

Right now you may just be testing the OOO of the internal memory controller of a single core, and it tries quite hard not to go very far out of order because the tracking is expensive.

But if you make sure X and Y are on separate cachelines and set thread affinities so the threads are on physically different cores (as opposed to being hyperthreads on the same core), you may see a significant increase in the number of times this happens. This is because ownership between cores is done on a cacheline granularity - if the two variables are in the same line, it's highly likely that each core will either own both or neither. Forcing them into different cachelines allows split ownership which can last many hundreds of clocks.

Reply  [1 reply](#) · active 123 weeks ago



[Jeff Preshing](#) · 193 weeks ago

I tried it on a Core 2 Duo E6300 in Win32. The rate of occurrence indeed increased from 1 out of 7500 to 1 out of 4700 or so.

Reply



Jules · 193 weeks ago

If you're interested in exploring the weirdnesses of your computer memory model you should check out DIY:

<http://diy.inria.fr/>

It allows you to harness these small tests like the one you described automatically, and to witness if weak executions occur.

Reply



Adam White · 193 weeks ago

I have little to say about the article. I just want to compliment you on your awesome Jumpman avatar. It really brings back thoughts of my childhood spent on a C64.

Reply



Roy · 193 weeks ago

On my Dual Dual-Core Xeon 5150, I get a significantly higher rate of reordering...

163090 reorders detected after 3000077 iterations - 5.4% of the time. That's not rare.

Reply





[Nicolae Vartolomei](#) · 192 weeks ago

On core 2 duo strange things happens

1001 reorders detected after 17590 iterations

12785 reorders detected after 2013606 iterations

Reply



Dmytro · 189 weeks ago

First of all thanks for the article.

I'm really surprised that binding threads to once CPU solves the issue. I would say it's just masking the problem.

Say if instruction reordering takes place($r1=Y, X=1; r2=X, Y=1$) for the both threads what does prevent OS scheduler to preempt $t\#1$ before $\{X = 1;\}$ and run $t\#2$? In the end you'll get both $r1$ and $r2$ equal to 0. To prevent this you should use `_single_` binary semaphore, mutex, whatever.

In you case with binding threads it's likely that linux just executes the code within the "critical" section fast enough that no forced preemption needed as you do it voluntarily with `sem_wait/sem_post`.

Another interesting point for me is scheduling threads on different CPUs(COREs). If linux does this now by default - then it's a BIG problem. There will be L1+ cache ping pong during simultaneous access to non-TLS data.

I know that linux kernel does not make huge difference between threads and processes but task migration is a `_big_` deal. For the processes that did `exec` call - it's ok, since old VM-space will be completely replaced.

The only one reason I can see is that nothing else is executing on your COREs within some threshold and the threads consume most of the CPU time and migration is forced. I won't sleep well with the thought that kernel spawns any newly created thread on the other CORE if the latter is not fully loaded.

Reply

[3 replies](#) · active 123 weeks ago



[Jeff Preshing](#) · 189 weeks ago

I think the first part of your comment can be resolved by recognizing that, at the very least, we must [prevent compiler reordering](#), even for a single-processor execution. Once the machine instructions are in the right order, the processor will never see its own memory effects out of order.

Regarding cache ping-pong, I've never seen anyone identify and optimize this kind of problem away in a real application, but I'm sure it has happened somewhere. Come to think of it, the performance of my [lightweight logging system](#) -- or any lock-free queue, really -- might be limited by exactly this. In the context of a large application, though, I'm not sure it's a big deal.

Reply



Dmytro · 189 weeks ago

Thanks for the reply, Jeff.

I was talking about instruction reordering in CPU pipeline(execution out of order). Now I recognized that I'm likely mistaken here.

If the CPU does instruction reordering then the execution of the scheduled instruction can not be preempted on single CPU. The list of instruction must be completed(or vanished and changes rolled back) before the CPU could be interrupted.

Reply



[Jeff Preshing](#) · 189 weeks ago

OK, yeah. I don't think the CPU can reorder instructions it hasn't fetched yet (advancing the instruction pointer). And once they're fetched, they go all the way down the pipeline.

Reply



bru · 177 weeks ago

Thanks for the article, which inspired me to some fun experiments. I observed that you can get much higher rates of reorderings when you drop the random delay in the thread functions and synchronize them with spin waits instead, i.e., let them wait for each other before performing the STORE/LOAD operations. The critical parts were implemented as follows ('spin1' and 'spin2' are 1 at the begin of each iteration; the second thread is a mirror image of the first one):

```
# in thread 1:
...
# unblock thread 2:
mov dword ptr [spin2], 0
mfence
# wait for thread 2, if necessary:
L1: pause
cmp dword ptr [spin1], 0
je L1
# STORE/LOAD sequence:
mov dword ptr [X], 1
mov eax, dword ptr [Y]
mov dword ptr [r1], eax
...
```

Intel's Optimization Manual recommends to insert PAUSE instructions into spin loops, and here its use actually seems to improve the synchronicity. The same holds for the MFENCE instruction, most probably because without it, the accesses to the 'spin flags' could also be reordered. And last but not least, it seems crucial to have the spin flags in the same cache line. If they are separated, the rate of reorderings drops from over 90% to 0.15% on my machine (Intel Core 2 Duo, 64-bit Linux).

Reply  [4 replies](#) · active 115 weeks ago



bru · 177 weeks ago

When rewriting the code to make it readable, I made a mistake: the conditional branch is 'jne L1', of course.

And, to be clear about it, the spin waits are a less general approach than the random delay. On some systems, they might regularly cause a delay between the STORE/LOAD sequences that leaves no chance for memory reordering. On the other hand, they might be the only way to get it in 90% of the iterations.

Reply



lynxjerm · 137 weeks ago

Hey can I see the source file you used? I used the assembly you wrote but can't seem to reproduce the 90% (or just high for that matter) reorderings.

Reply



bru · 126 weeks ago

I wouldn't mind to share or publish the sources. However, I believe the comments section here is not the place to do that. So if you tell me how, I'll send you the files.

Maybe two hints might help. First, the variables in the assembler part are distributed among cache lines as follows (I hope you don't mind AS syntax):

```
.data
.globl X, Y, r1, r2, spin1, spin2
.align 128
X: .long 0
.align 128
```

```
Y: .long 0
.align 128
r1: .long 0
.align 128
r2: .long 0
.align 128
spin1: .long 1
spin2: .long 1
```

Second, the overall structure is the same as in Jeff Preshing's code. The two threads each wait on a semaphore before they enter the spinlock and then the STORE/LOAD part, and use another semaphore to signal to the main thread that they are done. The main loop is simply as follows (on my machine the test runs for about three seconds when `ITERATIONS == 100000`):

```
int reordered = 0;
...
for (i = 0; i < ITERATIONS; i++) {
    X = (Y = 0);
    r1 = (r2 = 0);
    spin1 = (spin2 = 1);
    sem_post(semaphore_1a);
    sem_post(semaphore_2a);
    sem_wait(semaphore_1b);
    sem_wait(semaphore_2b);
    if (!r1 && !r2) reordered++;
}
```

Reply



lynxjerm · 115 weeks ago

Thanks brue. I just noticed your reply. Maybe PM me on reddit. My name is lynxjerm there too.

Reply



[Java Geek](#) · 172 weeks ago

No one has explained memory reordering better than this article to me before. Amazing stuff.

Reply



mihai · 167 weeks ago

a stupid question maybe.

But where is global int r2 used ?

Reply

[1 reply](#) · active 123 weeks ago



[Jeff Preshing](#) · 167 weeks ago

You can check the [source](#).

Reply



[he dengcheng](#) · 135 weeks ago

Hi, Preshing

Thanks for this great article, inspired me a lot.

But there is one question. In Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1. Section 8.2.1, which says "...Read misses are permitted to go ahead of buffered writes on the system bus when all the buffered writes are cache hits and, therefore, are not directed to the same address being accessed by the read miss. ...", which means

only read cache miss and write cache hits can be reordered. In this article, global variables X, Y, r1, r2 are all on the same cache line, so later read can not be cache miss followed by a write cache hits. Under this situation, why read still reordered with previous write?

Best Wishes!

Reply



Nate · 124 weeks ago

When testing the sample code with `USE_CPU_FENCE=1`, i still get tons of reorders on mac os x. It works fine on Linux (no reorders as expected). Does anyone have any ideas what might be causing this?

Disassembler outs clearly show emitted mfence in both cases:

osx / i686-apple-darwin11-llvm-gcc-4.2: <http://pastebin.com/tjtW3Egb>
linux / Debian gcc 4.7.2-5: <http://pastebin.com/8TRVPMFM>

Reply

2 replies · active 123 weeks ago



Nate · 124 weeks ago

Ah, this is actually just a semaphore issue. The code compiles fine on os x, but anonymous semaphores do not work correctly there. The functions are all stubs. One fix is to use macros such as the following: <http://pastebin.com/WRhQAKWt>

Reply



Jeff Preshing · 124 weeks ago

Good catch. There's a version of this experiment in the [Mintomic](#) test suite. It uses `mint_sem_create`, a wrapper which works even on Apple platforms, to avoid this exact problem.

Reply



shawnh310 · 108 weeks ago

I am sorry if this have already been explained..

In thread 1 we set `X = 1`, in thread 2 we set `Y = 1`. How is it guaranteed that these two threads can see these updates? I understand that between main thread and these 2 threads we use semaphore. Why does it need not to make `X` and `Y` volatile to ensure visibility? Thanks in advance.

Reply

1 reply · active 31 weeks ago



Ian Ni-Lewis · 31 weeks ago

The volatile keyword doesn't precisely ensure visibility. Instead, it ensures that compiler always issues a load instruction to read the variable, rather than keeping the variable in a register. I've seen compilers that also inserted memory barriers around volatiles, but AFAIK that's not at all required by the standard.

In order to absolutely ensure visibility without C++11-style fences, you need to both declare the variables volatile *and* insert a memory barrier.

Reply



Eric · 76 weeks ago

Hi Jeff,

A quick question.

On a weak memory order machine like ARM, two stores may be reordered fi there are no data

dependency between them. Consider the classic case where 'data' is set and then 'flag' is set to inform another thread to read the data. In this case, even there is a single core, there can be a synchronization bug that the second thread (reader) can read 'data' before 'flag' is read, is that right? The single core processor can also have this problem, right?

Reply [1 reply](#) · active 46 weeks ago



[Jeff Preshing](#) · 46 weeks ago

Yes. A single core processor can have this problem too. In general, if a multicore processor can introduce memory reordering, a C/C++ compiler "could" do it too.

Reply



Neo · 46 weeks ago

@Eric

I believe since your code will have an if statement, the compiler/processor can figure out it is a dependent read and insert write-write barrier or not reorder them.

Reply [1 reply](#) · active 46 weeks ago



[Jeff Preshing](#) · 46 weeks ago

That's a reasonable guess, and probably true of specific compilers under specific settings, but not true in general. Even in the following example, a compiler "could" (surprisingly) still reorder the loads of g and v. Specifically, it could introduce a new local variable t, and speculatively load from a known address before loading g. The ISO C++ committee spent a lot of time worrying about such possibilities when defining the C++11 memory model.

```
int* Guard;

void foo()
{
    ...
    int* g = Guard;
    if (g != NULL)
    {
        int v = *g;
        ...
    }
}
```

Reply



[Ian Ni-Lewis](#) · 31 weeks ago

Coming in late on this, but you're mostly correct about the Playstation 3 PPU, at least if the design is anything like the Xbox360 CPU (which I'm pretty sure it is). The two hardware "threads" are merely interleaved instruction streams, with each thread able to issue on alternate clock cycles.

But more to the point, there's not much gain to be had from reordering memory accesses between hardware threads. As Bruce notes, the incentive for load and store reordering is to reduce perceived latency from the cache and memory system. But hardware threads share L1--the only unshared storage is the register file. So any latency experienced by one core would be experienced in exactly the same way by the other core.

Because the two threads share the load/store unit, all loads and stores are serialized anyway. I think, but do not know for sure, that dependent read and load-hit-store detection are shared as well. That would imply that memory ordering is maintained globally across all hardware threads.

Reply



Ramesh · 27 weeks ago

```
int var1 = 0;
int var2 = 0;
```

```
int var3 = 0;

threadOne() {

// update var1
var1 = 0x11231926;

// wait for var2 update
while (var2 == 0);

var3 = 0x11231926;

print("Thread One is Done");
}

threadTwo() {

// wait for first update
while (var1 != 0x11231926);

// update var1
var2 = 0x11231926;

// Wait for second update
while (var3 != 0x11231926);

print("Thread Two is Done");
}
```

If there is no re-ordering (compile and runtime) of code then I expect both threads should progress and complete i.e. updates by either thread should flow out and become available to the other thread. However, I am not really sure i.e. is it possible for threadOne to be stuck in the while loop waiting for update from threadTwo to become available.

Reply

Post a new comment

Enter text right here!

Posting as [bqn9000](#) ([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 © 2016 Jeff Preshing - Powered by [Octopress](#)