Preshing on Programming

Jul 02, 2013

The Happens-Before Relation

Happens-before is a modern computer science term which is instrumental in describing the software memory models behind C++11, Java, Go and even LLVM.

You'll find a definition of the *happens-before* relation in the specifications of each of the above languages. Conveniently, the definitions given in those specifications are basically the same, though each specification has a different way of saying it. Roughly speaking, the common definition can be stated as follows:

> Let A and B represent operations performed by a multithreaded process. If A ***happens-before*** B, then the memory effects of A effectively become visible to the thread performing B before B is performed.

When you consider the various ways in which memory reordering can complicate lock-free programming, the guarantee that A *happens-before* B is a desirable one. There are several ways to obtain this guarantee, differing slightly from one programming language to next – though obviously, all languages must rely on the same mechanisms at the processor level.

No matter which programming language you use, they all have one thing in common: If operations A and B are performed by the same thread, and A's statement comes before B's statement in program order, then A *happens-before* B. This is basically a formalization of the "cardinal rule of memory ordering" I mentioned in an earlier post.

```
int A, B;
```

```
void foo()
{
    // This store to A ...
    A = 5;

    // ... effectively becomes visible before the following loads. Duh!
    B = A * A;
}
```

That's not the only way to achieve a *happens-before* relation. The C++11 standard states that, among other methods, you also can achieve it between operations in different threads using acquire and release semantics. I'll talk about that more in the next post about *synchronizes-with*.

I'm pretty sure that the name of this relation may lead to confusion for some. It's worth clearing up right away: The *happens-before* relation, under the definition given above, is not the same as A actually happening before B! In particular,

1. A *happens-before* B does not imply A happening before B.
2. A happening before B does not imply A *happens-before* B.

These statements might appear paradoxical, but they're not. I'll try to explain them in the following sections. Remember, *happens-before* is a formal relation between operations, defined by a family of language specifications; it exists independently of the concept of time. This is different from what we usually mean when we say that "A happens before B"; referring the order, in time, of real-world events. Throughout this post, I've been careful to always hyphenate the former term *happens-before*, in order to distinguish it from the latter.

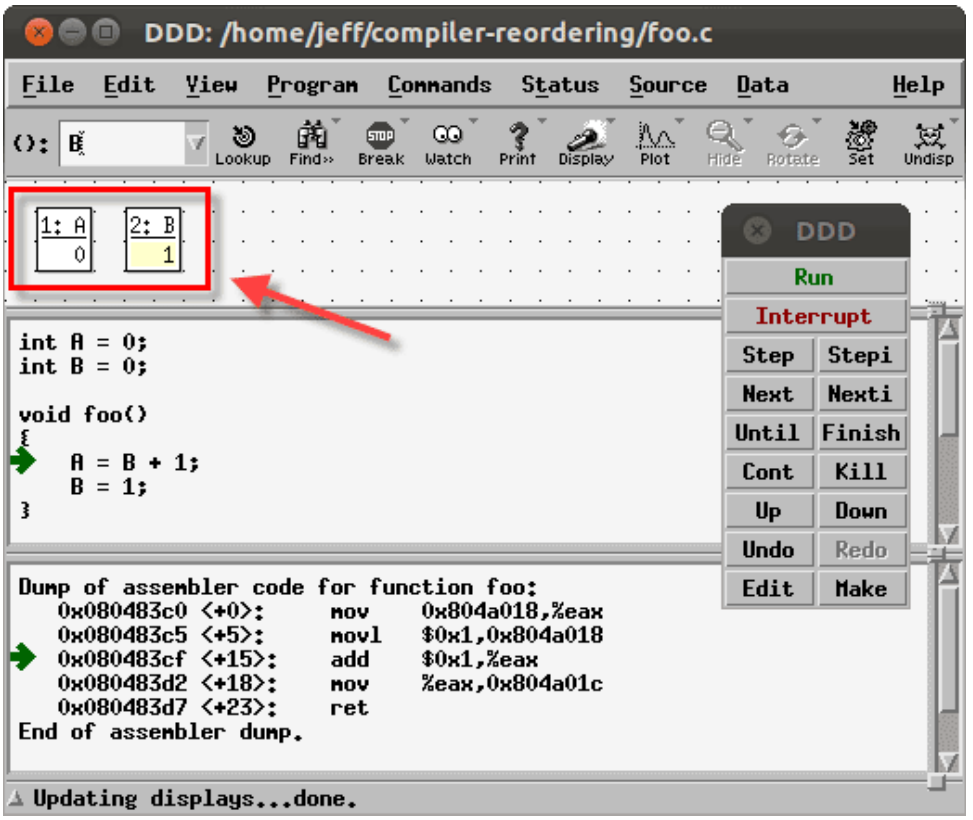
Happens-Before Does Not Imply Happening Before

Here's an example of operations having a *happens-before* relationship without actually happening in that order. The following code performs (1) a store to `A`, followed by (2) a store to `B`. According to the rule of program ordering, (1) *happens-before* (2).

```
int A = 0;
int B = 0;

void foo()
{
    A = B + 1;              // (1)
    B = 1;                  // (2)
}
```

However, if we compile this code with -O2 using GCC, the compiler performs some instruction reordering. As a result, when we step through the resulting code at the disassembly level in the debugger, we clearly see that after the second machine instruction, the store to `B` has completed, but the store to `A` has not. In other words, (1) doesn't actually happen before (2)!



Has the *happens-before* relation been violated? Let's see. According to the definition, the memory effects of (1) must effectively be visible before (2) is performed. In other words, the store to `A` must have a chance to influence the store to `B`.

In this case, though, the store to `A` doesn't actually influence the store to `B`. (2) still behaves the same as it would have even if the effects of (1) *had been* visible, which is *effectively* the same as (1)'s effects being visible. Therefore, this doesn't count as a violation of the *happens-before* rule. I'll admit, this explanation is a bit dicey, but I'm fairly confident it's consistent with the meaning of *happen-before* in all those language specifications.

Happening Before Does Not Imply *Happens-Before*

Here's an example of operations which clearly happen in a specific order without constituting a *happens-before* relationship. In the following code, imagine that one thread calls `publishMessage`, while another thread calls `consumeMessage`. Since we're manipulating shared variables concurrently, let's keep it simple and assume that plain loads and stores of `int` are atomic. Because of program ordering, there is a *happens-before* relation between (1) and (2), and another *happens-before* relation between (3) and (4).

```
int isReady = 0;
int answer = 0;

void publishMessage()
{
    answer = 42;                    // (1)
    isReady = 1;                    // (2)
}
```

```
void consumeMessage()
{
    if (isReady)                        // (3) <-- Let's suppose this line reads 1
        printf("%d\n", answer);         // (4)
}
```

Furthermore, let's suppose that at runtime, the line marked (3) ends up reading 1, the value that was stored at line (2) in the other thread. In this case, we know that (2) must have happened before (3). But that doesn't mean there is a *happens-before* relationship between (2) and (3)!

The *happens-before* relationship only exists where the language standards say it exists. And since these are plain loads and stores, the C++11 standard has no rule which introduces a *happens-before* relation between (2) and (3), even when (3) reads the value written by (2).

Furthermore, because there is no *happens-before* relation between (2) and (3), there is no *happens-before* relation between (1) and (4), either. Therefore, the memory interactions of (1) and (4) can be reordered, either due to compiler instruction reordering or memory reordering on the processor itself, such that (4) ends up printing "0", even though (3) reads 1.

This post hasn't really shown anything new. We already knew that memory interactions can be reordered when executing lock-free code. We've simply examined a term used in C++11, Java, Go and LLVM to formally specify the cases when memory reordering can be observed in those languages. Even Mintomic, a library I published several weeks ago, relies on the guarantees of *happens-before*, since it mimics the behavior of specific C++11 atomic library functions.

I believe the ambiguity that exists between the *happens-before* relation and the actual order of operations is part of what makes low-level lock-free programming so tricky. If nothing else, this post should have demonstrated that *happens-before* is a useful guarantee which doesn't come cheaply between threads. I'll expand on that further in the next post.

« Atomic vs. Non-Atomic Operations The Synchronizes-With Relation »

Comments (5)

 Andrea · *134 weeks ago*

That's great! Absolutely the best and most clarifying explanation I've found in the last 2 years (i.e. since I've started struggling with these concepts while making my mind up around the c++11 memory model).
I now have this mental model "A happens-before B if the execution behaves as-if all the memory effects of A are visible to the thread executing B _before_ executing it" which has

cleaned a lot of confusion I had before.
Hope to see next posts about synchronization soon!

Andrea.

Reply

 Amer Alhalabi · *131 weeks ago*

Thank you very much Jeff!
I owe you big for sharing your knowledge and posting those outstanding articles about lock-free programming.
You're the best!

Reply

 ecb · *121 weeks ago*

Is happens-before not occasionally not transitive? Isn't this the reason for memory_order_consume?

Reply **2 replies** · *active 88 weeks ago*

 Stefan · *100 weeks ago*

At least in C++11, strictly speaking, happens-before is not transitive. According to the standard, §1.10:12 (I am referring to the N3337 draft), an evaluation A happens before an evaluation B if A is sequenced before B, or A inter-thread happens before B.

For example, assume that operation A is dependency-ordered before B (see §1.10:11 for a definition; this is where consume operations come into play). In particular this means that A inter-thread happens before B. Further assume that B is sequenced before C.

Then A happens before B, B happens before C, but A is not required to happen before C by the standard.

This shows that happens-before is not transitive in C++11.

Reply

 Jeff Preshing · *88 weeks ago*

Hi Stefan,

You are totally right. In the current specifications of other languages, *happens-before* is transitive; Java says so explicitly, and in Go and LLVM, it's defined as a partial ordering, which implies transitivity. But in C++11, strictly speaking, it is not always transitive.

It comes pretty close: If we ignore consume operations and the *dependency-ordered-before* relation in C++11, the remaining forms of *happens-before* end up being transitive.

But it was a mistake for me to say that it was *always* transitive in C++11. I've removed that statement from the post. Thanks for the precision!

Reply

Post a new comment

Enter text right here!

Posting as bgn9000 (Logout)

Subscribe to Replies

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!