This Blog Has Moved!

February 04, 2021

Right, so yes, five years ago I moved to github pages, and never bothered to redirect any of these pages there. Now I've moved on from there, and... Finally I am using my real domain, trishagee.com . My blog is now at trishagee.com/blog . See you there!

Post a Comment

Dissecting the Disruptor: Demystifying Memory Barriers

August 07, 2011

My recent slow-down in posting is because I've been trying to write a post explaining memory barriers and their applicability in the Disruptor. The problem is, no matter how much I read and no matter how many times I ask the ever-patient Martin and Mike questions trying to clarify some point, I just don't intuitively grasp the subject. I guess I don't have the deep background knowledge required to fully understand.

So, rather than make an idiot of myself trying to explain something I don't really get, I'm going to try and cover, at an abstract / massive-simplification level, what I do understand in the area. Martin has written a post going into memory barriers in some detail, so hopefully I can get away with skimming the subject.

Disclaimer: any errors in the explanation are completely my own, and no reflection on the implementation of the Disruptor or on the LMAX guys who actually do know about this stuff.

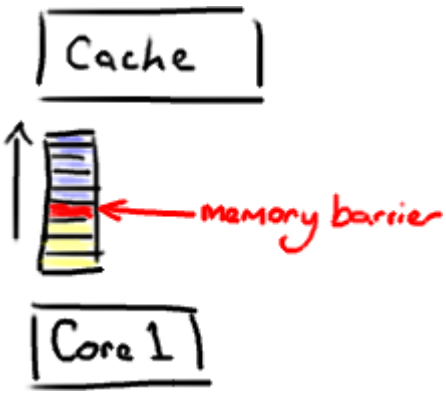
What's the point?
My main aim in this series of blog posts is to explain how the Disruptor works and, to a slightly lesser extent, why. In theory I should be able to provide a bridge between the code and the

technical paper by talking about it from the point of view of a developer who might want to use it.

The paper mentioned memory barriers, and I wanted to understand what they were, and how they apply.

What's a Memory Barrier?
It's a CPU instruction. Yes, once again, we're thinking about CPU-level stuff in order to get the performance we need (Martin's famous Mechanical Sympathy). Basically it's an instruction to a) ensure the order in which certain operations are executed and b) influence visibility of some data (which might be the result of executing some instruction).

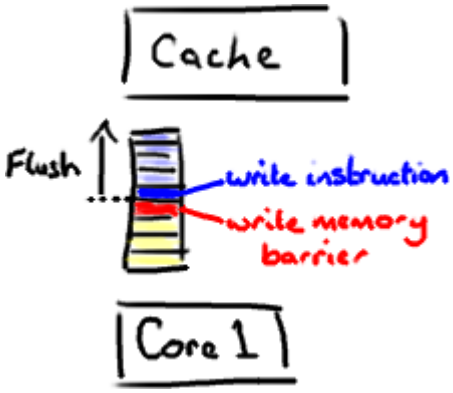
Compilers and CPUs can re-order instructions, provided the end result is the same, to try and optimise performance. Inserting a memory barrier tells the CPU and the compiler that what happened before that command needs to stay before that command, and what happens after needs to stay after. All similarities to a trip to Vegas are entirely in your own mind.



The other thing a memory barrier does is force an update of the various CPU caches - for example, a write barrier will flush all the data that was written before the barrier out to cache, therefore any other thread that tries to read that data will get the most up-to-date version regardless of which core or which socket it might be executing by.

What's this got to do with Java?
Now I know what you're thinking - this isn't assembler. It's Java.

The magic incantation here is the word `volatile` (something I felt was never clearly explained in the Java certification). If your field is `volatile`, the Java Memory Model inserts a write barrier instruction after you write to it, and a read barrier instruction before you read from it.

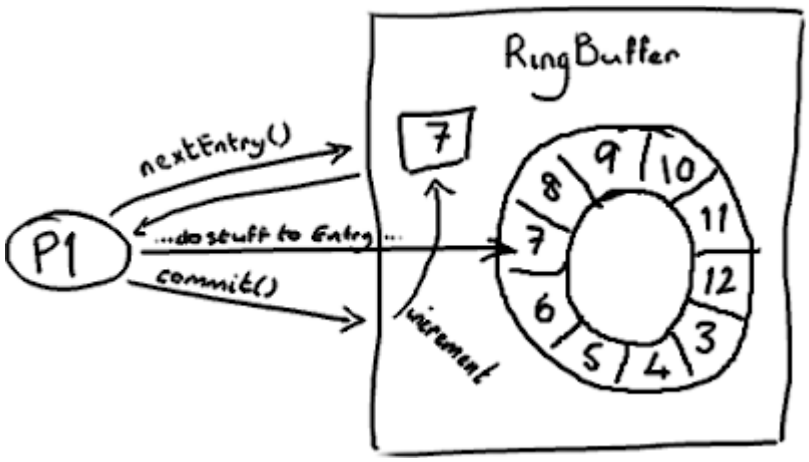


This means if you write to a volatile field, you know that:

1. Any thread accessing that field after the point at which you wrote to it will get the updated value
2. Anything you did before you wrote that field is guaranteed to have happened and any updated data values will also be visible, because the memory barrier flushed all earlier writes to the cache.

Example please!
So glad you asked. It's about time I started drawing doughnuts again.

The RingBuffer `cursor` is one of these magic volatile thingies, and it's one of the reasons we can get away with implementing the Disruptor without locking.



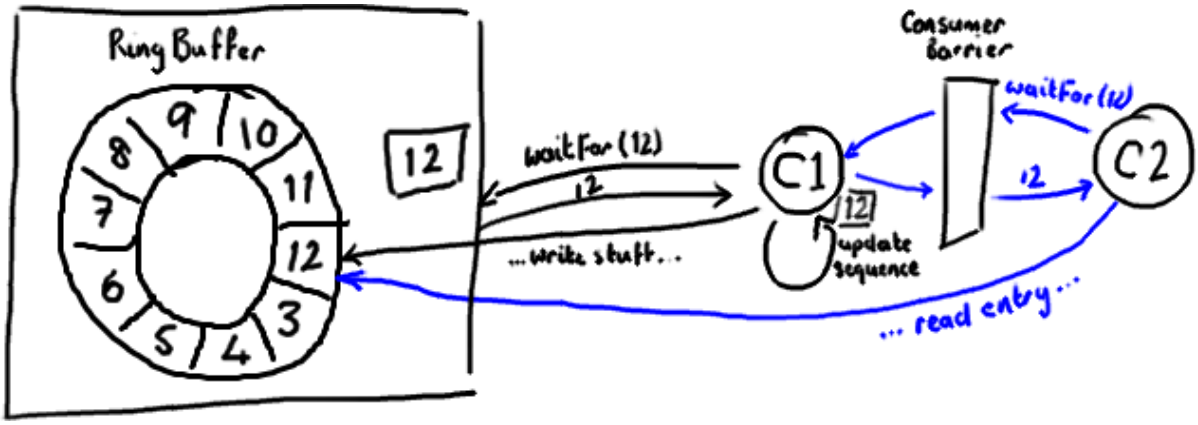
The Producer will obtain the next Entry (or batch of them) and do whatever it needs to do to the entries, updating them with whatever values it wants to place in there. As you know, at the

end of all the changes the producer calls the commit method on the ring buffer, which updates the sequence number. This write of the volatile field (`cursor`) creates a memory barrier which ultimately brings all the caches up to date (or at least invalidates them accordingly).

At this point, the consumers can get the updated sequence number (8), and because the memory barrier also guarantees the ordering of the instructions that happened before then, the consumers can be confident that all changes the producer did to to the Entry at position 7 are also available.

...and on the Consumer side?

The sequence number on the Consumer is volatile, and read by a number of external objects - other downstream consumers might be tracking this consumer, and the ProducerBarrier/RingBuffer (depending on whether you're looking at older or newer code) tracks it to make sure the the ring doesn't wrap.



So, if your downstream consumer (C2) sees that an earlier consumer (C1) reaches number 12, when C2 reads entries up to 12 from the ring buffer it will get all updates C1 made to the entries before it updated its sequence number.

Basically everything that happens after C2 gets the updated sequence number (shown in blue above) must occur after everything C1 did to the ring buffer before updating its sequence number (shown in black).

Impact on performance

Memory barriers, being another CPU-level instruction, don't have the same cost as locks - the kernel isn't interfering and arbitrating between multiple threads. But nothing comes for free. Memory barriers do have a cost - the compiler/CPU cannot re-order instructions, which could potentially lead to not using the CPU as efficiently as possible, and refreshing the caches obviously has a performance impact. So don't think that using volatile instead of locking will get you away scot free.

You'll notice that the Disruptor implementation tries to read from and write to the sequence number as infrequently as possible. Every read or write of a `volatile` field is a relatively costly operation. However, recognising this also plays in quite nicely with batching behaviour -

if you know you shouldn't read from or write to the sequences too frequently, it makes sense to grab a whole batch of Entries and process them before updating the sequence number, both on the Producer and Consumer side. Here's an example from BatchConsumer:

```
    long nextSequence = sequence + 1;
    while (running)
    {
        try
        {
            final long availableSequence =
consumerBarrier.waitFor(nextSequence);
            while (nextSequence <= availableSequence)
            {
                entry = consumerBarrier.getEntry(nextSequence);
                handler.onAvailable(entry);
                nextSequence++;
            }
            handler.onEndOfBatch();
            sequence = entry.getSequence();
        }
        ...
        catch (final Exception ex)
        {
            exceptionHandler.handle(ex, entry);
            sequence = entry.getSequence();
            nextSequence = entry.getSequence() + 1;
        }
    }
```

(You'll note this is the "old" code and naming conventions, because this is inline with my previous blog posts, I thought it was slightly less confusing than switching straight to the new conventions).

In the code above, we use a local variable to increment during our loop over the entries the consumer is processing. This means we read from and write to the volatile sequence field (shown in bold) as infrequently as we can get away with

(shown in bold) as infrequently as we can get away with.

In Summary

Memory barriers are CPU instructions that allow you to make certain assumptions about when data will be visible to other processes. In Java, you implement them with the `volatile`

keyword. Using volatile means you don't necessarily have to add locks willy nilly, and will give you performance improvements over using them. However you need to think a little more carefully about your design, in particular how frequently you use volatile fields, and how frequently you read and write them.

PS Given that the New World Order in the Disruptor uses totally different naming conventions now to everything I've blogged about so far, I guess the next post is mapping the old world to the new one.

concurrency disruptor disruptor-docs java lmax mechanical sympathy

---

Matt Fowles *10 August 2011 at 00:44*

If you have multiple things blocking on a slow consumer, doesn't forcing the sequence update to wait till the end of the batch prevent them from running in parallel on earlier parts of the batch?

REPLY

Trisha *10 August 2011 at 09:47*

Yes, if a slow consumer is doing stuff to entries 10-30 (for example), then consumers that are dependent upon this slow one will only be able to process up to number 10.

If you have a much slower consumer which other things are dependent on, at some point everything's going to be waiting for it anyway - any set of dependencies is only going to be as fast as the slowest thing, no matter what structure you use to organise them. The disruptor is designed to smooth out bursts of activity, in which case at some point the slow consumer will catch up during a period of low activity.

If a consumer is consistently slowing the rest of the system down, it's either a sign you need to address the performance of that consumer, or you can parallelise it, for example having two, one to process odd numbers and one to process even numbers.

REPLY

Michael Bloomfield *11 January 2012 at 17:34*

I'm impressed with the Disruptor design and concept. The LMAX has worked hard to think through how to use the hardware (memory, hard drives, etc.)

I'm wondering if C/C++/C# would have given you guys better control of memory management than Java, especially with low level instructions?

REPLY

Trisha *11 January 2012 at 17:45*

I could write a whole article on why we chose Java over C++!

The short version is that yes, C/C++ might have given us greater control. But modern Java compilers are very efficient, and not worrying about a lot of the low level details is actually an advantage - with Java, we can at least pick out the stuff we want to care about and let the compiler take care of everything else.

Another advantage is the sheer quantity of good quality Java devs in London - it makes hiring a lot easier, and with a good dev you can teach them the specifics of performance for your system even if they're not high performance gurus.

Yet another advantage is (I'm told, I'm not a C/C++ developer but I heard Martin talk about this) that getting the code fast, correct, and readable in C/C++ was going to take longer than in Java. Sure we could get a higher performance system but it would take longer than the time it took to write in Java, and this is fast enough for us for now.

REPLY

Michael Bloomfield *12 January 2012 at 00:32*

I've read all the blogs, technical papers, articles, etc. on LMAX. One part that I haven't wrapped my head around is this:

LMAX is a retail exchange where you have millions of users buying/sellings derivatives. The Disruptor is a Single Writer/Multiple Reader design.

How do you collect 1 million buy/sell orders (from millions of inputs/users) and organize them to write to Disruptor in orderly fashion? I presume in timestamp order, too?

Are you using a round robin approach to giving each input the opportunity to write to the Disruptor?

REPLY

Trisha *12 January 2012 at 17:36*

I'm not sure I'm allowed to talk about that if I'm honest! Obviously we've open sourced various parts of the system like the Disruptor, Freud

(http://code.google.com/p/freud/) and JMicrobench, but talking about how we really make the most of them is probably where we start straying into the territory of our intellectual property.

I will say, however, that we have a number of different channels into the exchange - we have FIX gateways for liquidity providers, FIX gateways for retail users, the Web UI and the API/protocol. So we don't have a single source of millions of orders to marshal into the Disruptor, we have a number of sources of orders, each gateway also using the Disruptor to marshal into the exchange in a predictable fashion.

REPLY

Michael Bloomfield *12 January 2012 at 20:00*

Ah, intellectual property... Do tell :-) "in a predictable fashion" is the key word.

When does a message get stamped with the "exchange timestamp"? Does the input source Disruptor handle timestamping or does the exchange Disruptor handle the timestamping? When I think about the algorithm of multiple input sources converging into one exchange Disruptor, it seems more likely that the "exchange timestamp" occurs in the exchange Disruptor.

REPLY

Trisha *13 January 2012 at 09:48*

I think I'll avoid giving away any more secrets ;-)

REPLY

Michael Bloomfield *13 January 2012 at 17:45*

Smile, no worries, it's fun to think about these things.

So, for 2.0, Martin managed to improve the Disruptor from 6 million mps to 25 million mps. That's insane.

REPLY

stuart cullinan *21 March 2012 at 17:15*

Apologies for the noob question but I'm battling to understand what happens to the sequence number when it reaches it's maximum size? I get the idea of the ring wrapping and how you determine the position in the array however the sequence number is finite, what am I missing?

Trisha *2 April 2012 at 14:15*

It's a fair point, the sequence number is finite, so what happens when we reach the maximum?

The sequence number is a Long, so the maximum value this can be is 9223372036854775807.

If you process a million messages a second, it will take you

9223372036854775807 / 1 000 000 = 9223372036854 seconds to reach this value.

Which is
9223372036854 / 60 = 153722867280 minutes

Which is
153722867280 / 60 / 24 = 106751991 days

Which is approx
106751991 / 365 = 292,471 years

So, yes, you will run out of sequence numbers at some point. But if you're processing a million messages a second it's still going to take a looong time before you wrap the buffer. It's like the y2k problem, but I think global warming is a more pressing matter.

REPLY

Unknown *19 July 2015 at 08:55*

Sorry for the very stupid question to come ... I haven't actually went through the disruptor source code, but why can't it simply reset the sequence number once one cycle is finished ?

Trisha *19 July 2015 at 09:44*

a) there's no need, you'll never run out of numbers
b) define finished? when you've written to all the slots? What if there's still a reader waiting to read from slot one? How can you tell the difference between sequence number one the first time around and sequence number one the second time around?

The nice thing about the disruptor is that if you journal all the entries and their sequence numbers every time they've been written (see Martin Fowler's article for the overall LMAX architecture) you get a clear sequence of events that happened throughout the system, from zero when you started up to some insanely high number when it shut down. This is a really nice feature that can be used for debugging events that went through the system.

REPLY

Apurva Singh *19 October 2015 at 21:24*

Hi Trisha, thx for your effort and mostly I have got it. One q regarding choice of 2 dimensional array.. so the array has reference to a byte array which stores Entry.. but this ref.. is it volatile (performance hit again.. but will work)? I am assuming volatile ref to Entry will make sure all bytes in the volatile ref are correctly 'visible'. How about having just a single array of size = nr_of_elements * size_of_one_element?

Trisha *21 October 2015 at 11:21*

LMAX started to use the Disruptor in a different way after I wrote this post, I believe they no longer store the entries as a byte array. The Disruptor has moved on a lot since I wrote this four years ago, the best place to get info about how it works is now the Google group: https://groups.google.com/forum/#!forum/lmax-disruptor

There's a lot of information there already, and they're a friendly bunch if you want to ask specific questions about how things work.

REPLY

Comments have been disabled since this blog is no longer active.

Popular posts from this blog

Dissecting the Disruptor: What's so special about a ring buffer?

June 22, 2011

Recently we open sourced the LMAX Disruptor , the key to what makes our

exchange so fast. Why did we open source it? Well, we've realised that conventional wisdom around high performance programming ...

READ MORE

Dissecting the Disruptor: Writing to the ring buffer

July 04, 2011

This is the missing piece in the end-to-end view of the Disruptor. Brace yourselves, it's quite long. But I decided to keep it in a single blog so you could have the context in one place. The important areas are: r ...

READ MORE

Dissecting the Disruptor: Why it's so fast (part one) - Locks Are Bad

July 16, 2011

Martin Fowler has written a really good article describing not only the Disruptor , but also how it fits into the architecture at LMAX . This gives some of the context that has been missing so far, but the mos ...

READ MORE

TRISHA

VISIT PROFILE

Archive