Trisha's Ramblings

My goal? Only to change the world...

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: 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 most frequently asked question is still "What is the Disruptor?".

I'm working up to answering that. I'm currently on question number two: "Why is it so fast?".

These questions do go hand in hand, however, because I can't talk about why it's fast without saying what it does, and I can't talk about what it is without saying why it is that way.

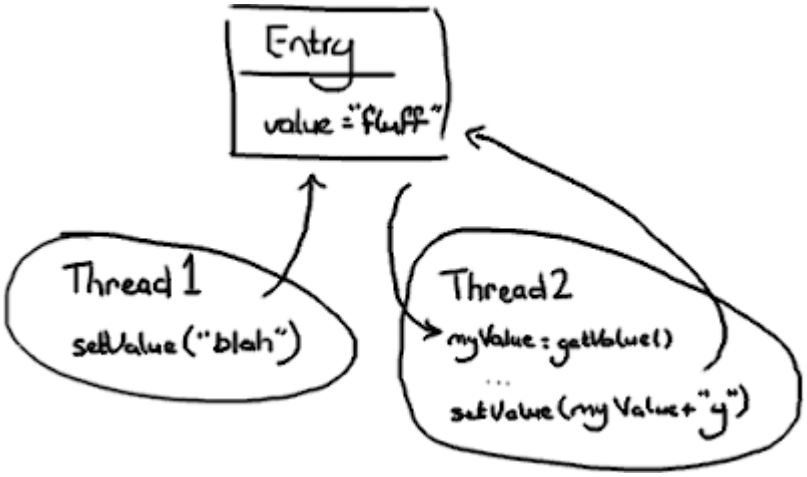
So I'm trapped in a circular dependency. A circular dependency of blogging.

To break the dependency, I'm going to answer question one with the simplest answer, and with any luck I'll come back to it in a later post if it still needs explanation: the Disruptor is a way to pass information between threads.

As a developer, already my alarm bells are going off because the word "thread" was just

mentioned, which means this is about concurrency, and Concurrency Is Hard.

Concurrency 101



Imagine two threads are trying to change the same value.

Case One: Thread 1 gets there first:

1. The value changes to "blah"
2. Then the value changes to "blahy" when Thread 2 gets there.

Case Two: Thread 2 gets there first:

1. The value changes to "fluffy"
2. Then the value changes to "blah" when Thread 1 gets there.

Case Three: Thread 1 interrupts Thread 2:

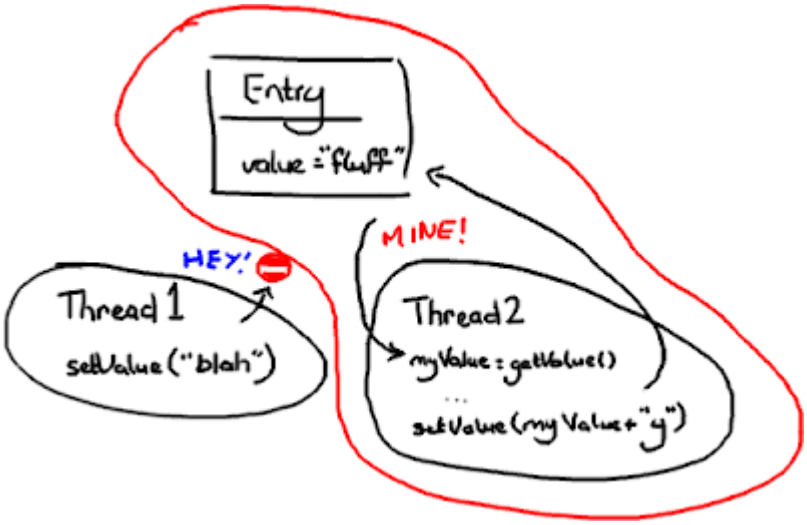
1. Thread 2 gets the value "fluff" and stores it as `myValue`
2. Thread 1 goes in and updates value to "blah"
3. Then Thread 2 wakes up and sets the value to "fluffy".

Case Three is probably the only one which is definitely wrong, unless you think the naive approach to wiki editing is OK (Google Code Wiki, I'm looking at you...). In the other two cases it's all about intentions and predictability. Thread 2 might not care what's in `value`, the intention might be to append "y" to whatever is in there regardless. In this circumstance, cases one and two are both correct.

But if Thread 2 only wanted to change "fluff" to "fluffy", then both cases two and three are incorrect.
Assuming that Thread 2 wants to set the value to "fluffy", there are some different approaches

to solving the problem.

Approach One: Pessimistic locking



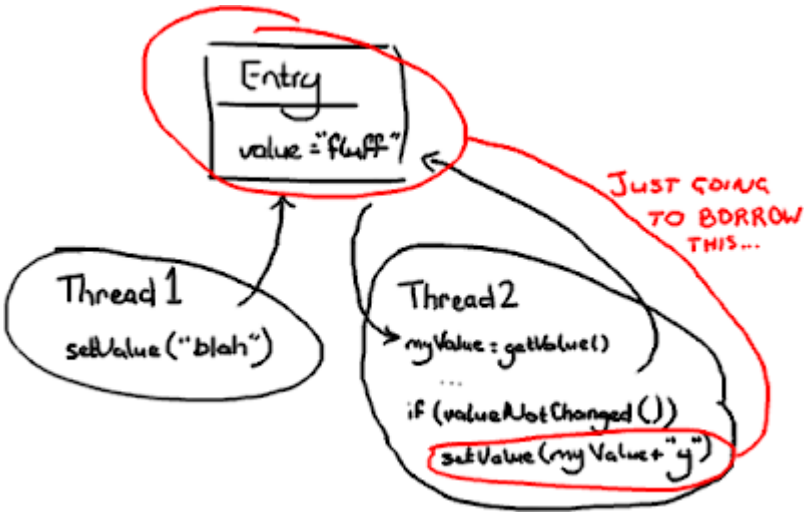
(Does the "No Entry" sign make sense to people who don't drive in Britain?)

The terms pessimistic and optimistic locking seem to be more commonly used when talking about database reads and writes, but the principal applies to getting a lock on an object.

Thread 2 grabs a lock on `Entry` as soon as it knows it needs it and stops anything from setting it. Then it does its thing, sets the value, and lets everything else carry on.

You can imagine this gets quite expensive, with threads hanging around all over the place trying to get hold of objects and being blocked. The more threads you have, the more chance that things are going to grind to a halt.

Approach Two: Optimistic locking

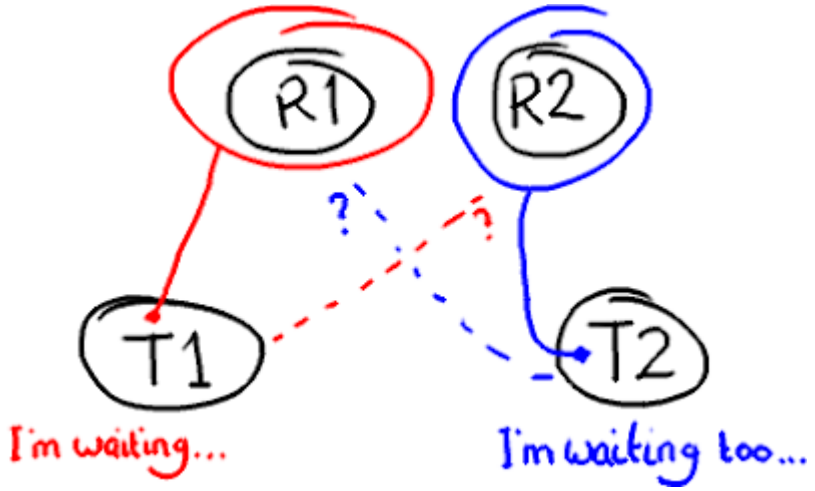


In this case Thread 2 will only lock `Entry` when it needs to write to it. In order to make this work, it needs to check if `Entry` has changed since it first looked at it. If Thread 1 came in and changed the value to "blah" after Thread 2 had read the value, Thread 2 couldn't write "fluffy" to

the `Entry` and trample all over the change from Thread 1. Thread 2 could either re-try (go back, read the value, and append "y" onto the end of the new value), which you would do if Thread 2 didn't care what the value it was changing was; or it could throw an exception or return some sort of failed update flag if it was expecting to change "fluff" to "fluffy". An example of this latter case might be if you have two users trying to update a Wiki page, and you tell the user on the other end of Thread 2 they'll need to load the new changes from Thread 1 and then reapply their changes.

Potential Problem: Deadlock

Locking can lead to all sorts of issues, for example deadlock. Imagine two threads that need access to two resources to do whatever they need to do:



If you've used an over-zealous locking technique, both threads are going to sit there forever waiting for the other one to release its lock on the resource. That's when you reboot ~~Windows~~ your computer.

Definite Problem: Locks are sloooow...

The thing about locks is that they need the operating system to arbitrate the argument. The threads are like siblings squabbling over a toy, and the OS kernel is the parent that decides which one gets it. It's like when you run to your Dad to tell him your sister has nicked the Transformer when you wanted to play with it - he's got bigger things to worry about than you two fighting, and he might finish off loading the dishwasher and putting on the laundry before settling the argument. If you draw attention to yourself with a lock, not only does it take time to get the operating system to arbitrate, the OS might decide the CPU has better things to do

than servicing your thread.

The Disruptor paper talks about an experiment we did. The test calls a function incrementing a 64-bit counter in a loop 500 million times. For a single thread with no locking, the test takes 300ms. If you add a lock (and this is for a single thread, no contention, and no additional complexity other than the lock) the test takes 10,000ms. That's, like, two orders of magnitude slower. Even more astounding, if you add a second thread (which logic suggests should take maybe half the time of the single thread with a lock) it takes 224,000ms. Incrementing a counter 500 million times takes nearly a *thousand* times longer when you split it over two threads instead of running it on one with no lock.

Concurrency Is Hard and Locks Are Bad

I'm just touching the surface of the problem, and obviously I'm using very simple examples. But the point is, if your code is meant to work in a multi-threaded environment, your job as a developer just got a lot more difficult:

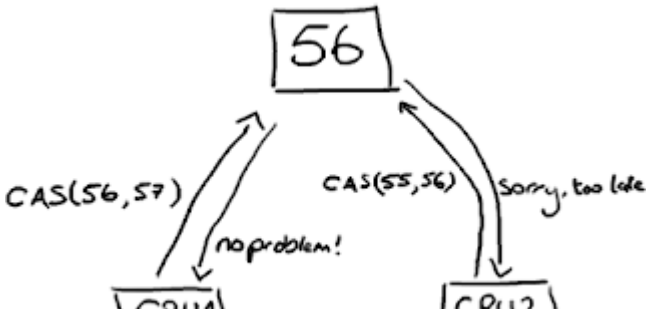
- **Naive code can have unintended consequences.** Case Three above is an example of how things can go horribly wrong if you don't realise you have multiple threads accessing and writing to the same data.
- **Selfish code is going to slow your system down.** Using locks to protect your code from the problem in Case Three can lead to things like deadlock or simply poor performance.

This is why many organisations have some sort of concurrency problems in their interview process (certainly for Java interviews). Unfortunately it's very easy to learn how to answer the questions without really understanding the problem, or possible solutions to it.

How does the Disruptor address these issues?

For a start, it doesn't use locks. At all.

Instead, where we need to make sure that operations are thread-safe (specifically, updating the next available sequence number in the case of multiple producers), we use a CAS (Compare And Swap/Set) operation. This is a CPU-level instruction, and in my mind it works a bit like optimistic locking - the CPU goes to update a value, but if the value it's changing it from is not the one it expects, the operation fails because clearly something else got in there first.



Note this could be two different cores rather than two separate CPUs.

CAS operations are much cheaper than locks because they don't involve the operating system, they go straight to the CPU. But they're not cost-free - in the experiment I mentioned above, where a lock-free thread takes 300ms and a thread with a lock takes 10,000ms, a single thread using CAS takes 5,700ms. So it takes less time than using a lock, but more time than a single thread that doesn't worry about contention at all.

Back to the Disruptor - I talked about the ClaimStrategy when I went over the producers. In the code you'll see two strategies, a `SingleThreadedStrategy` and a `MultiThreadedStrategy`. You could argue, why not just use the multi-threaded one with only a single producer? Surely it can handle that case? And it can. But the multi-threaded one uses an AtomicLong (Java's way of providing CAS operations), and the single-threaded one uses a simple long with no locks and no CAS. This means the single-threaded claim strategy is as fast as possible, given that it knows there is only one producer and therefore no contention on the sequence number.

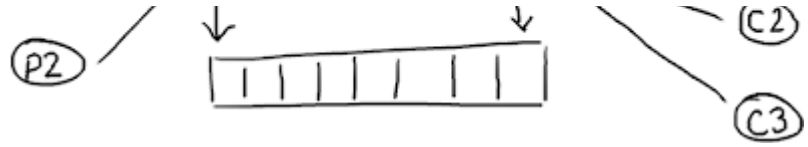
I know what you're thinking: turning one single number into an AtomicLong can't possibly have been the only thing that is the secret to the Disruptor's speed. And of course, it's not - otherwise this wouldn't be called "Why it's so fast (part *one*)".

But this is an important point - there's only one place in the code where multiple threads might be trying to update the same value. Only one place in the whole of this complicated data-structure-slash-framework. And that's the secret. Remember everything has its own sequence number? If you only have one producer then every sequence number in the system is only ever written to by one thread. That means there is no contention. No need for locks. No need even for CAS. The only sequence number that is ever written to by more than one thread is the one on the `ClaimStrategy` if there is more than one producer.

This is also why each variable in the `Entry` can only be written to by one consumer. It ensures there's no write contention, therefore no need for locks or CAS.

Back to why queues aren't up to the job

So you start to see why queues, which may implemented as a ring buffer under the covers, still can't match the performance of the Disruptor. The queue, and the basic ring buffer, only has two pointers - one to the front of the queue and one to the end:

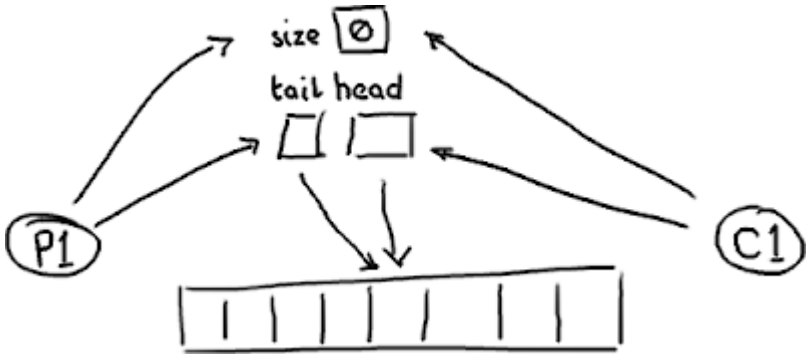


If more than one producer wants to place something on the queue, the tail pointer will be a point of contention as more than one thing wants to write to it. If there's more than one consumer, then the head pointer is contended, because this is not just a read operation but a write, as the pointer is updated when the item is consumed from the queue.

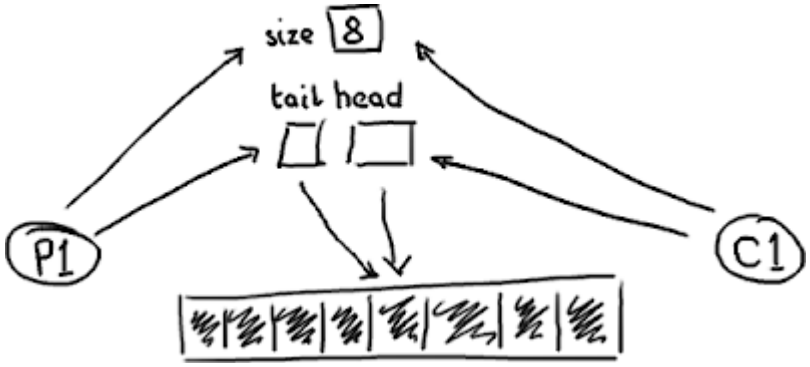
But wait, I hear you cry foul! Because we already knew this, so queues are usually single producer and single consumer (or at least they are in all the queue comparisons in our performance tests).

There's another thing to bear in mind with queues/buffers. The whole point is to provide a place for things to hang out between producers and consumers, to help buffer bursts of messages from one to the other. This means the buffer is usually full (the producer is out-pacing the consumer) or empty (the consumer is out-pacing the producer). It's rare that the producer and consumer will be so evenly-matched that the buffer has items in it but the producers and consumers are keeping pace with each other.

So this is how things really look. An empty queue:



...and a full queue:



The queue needs a size so that it can tell the difference between empty and full. Or, if it doesn't, it might determine that based on the contents of that entry, in which case reading an entry will require a write to erase it or mark it as consumed.

Whichever implementation is chosen, there's quite a bit of contention around the tail, head and size variables, or the entry itself if a consume operation also includes a write to remove it.

On top of this, these three variables are often in the same cache line, leading to false sharing. So, not only do you have to worry about the producer and the consumer both causing a write to the size variable (or the entry), updating the tail pointer could lead to a cache-miss when the head pointer is updated because they're sat in the same place. I'm going to duck out of going into that in detail because this post is quite long enough as it is.

So this is what we mean when we talk about "Teasing Apart the Concerns" or a queue's "conflated concerns". By giving everything its own sequence number and by allowing only one consumer to write to each variable in the `Entry`, the only case the Disruptor needs to manage contention is where more than one producer is writing to the ring buffer.

In summary
The Disruptor a number of advantages over traditional approaches:

1. No contention = no locks = it's very fast.
2. Having everything track its own sequence number allows multiple producers and multiple consumers to use the same data structure.
3. Tracking sequence numbers at each individual place (ring buffer, claim strategy, producers and consumers), plus the magic cache line padding, means no false sharing and no unexpected contention.

EDIT: Note that version 2.0 of the Disruptor uses different names to the ones in this article. Please see my summary of the changes if you are confused about class names.

concurrency disruptor disruptor-docs java lmax

Joe Bowbeer *17 July 2011 at 01:10*

Link to Fowler's article:

http://martinfowler.com/articles/lmax.html