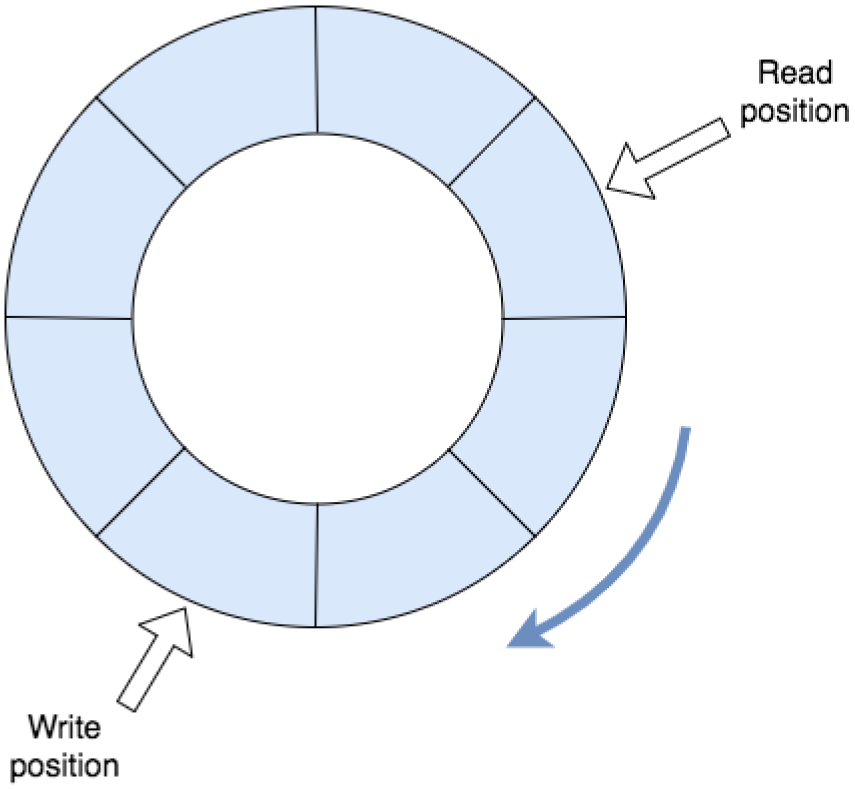
Ring Buffer: A Data Structure Behind Disruptor

In this article, we look at Disruptor, a library for passing messages between threads, and look at how it's data structure enables high performance.

Disruptor is a high-performance library for passing messages between threads, developed and open sourced some years ago by LMAX Exchange company. They created this piece of software to handle an enormous traffic (more than 6 million TPS) in their retail financial trading platform. In 2010, they surprised everyone with how fast their system can be by doing all the business logic on a single thread. Even though a single thread was an important concept in their solution, Disruptor (which is not part of the business logic) works in a multithreaded environment. Disruptor is based on ring buffer, which is definitely not a new concept.

Ring Buffer

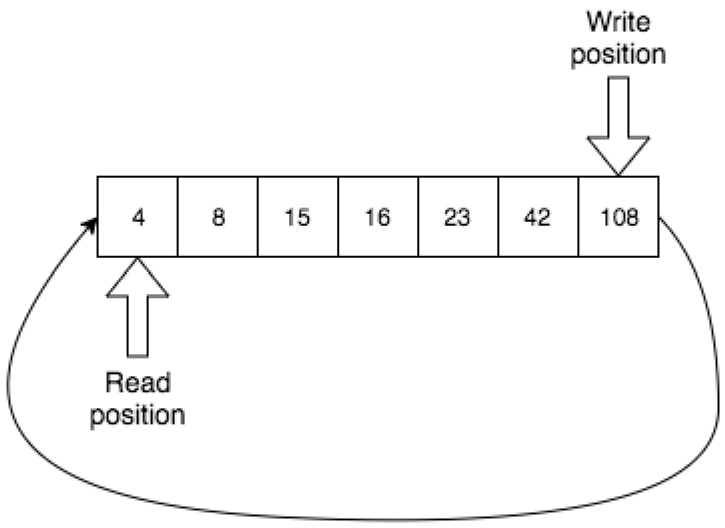
Ring buffer has many names. You might have heard of a circular buffer, circular queue, or cyclic buffer. All of them mean the same thing. It is basically a linear data structure in which the end points to the beginning of the structure. It’s easy to reason about it as a circular array without the end.



As you can imagine, a ring buffer is mostly used as a queue. It has read and write positions which are used by the consumer and the producer, respectively. When the read or write index reaches the end of the underlying array, it is set back to 0. This activity is usually called “wrapping around,” and it requires a bit more explanation.

Wrapping Around

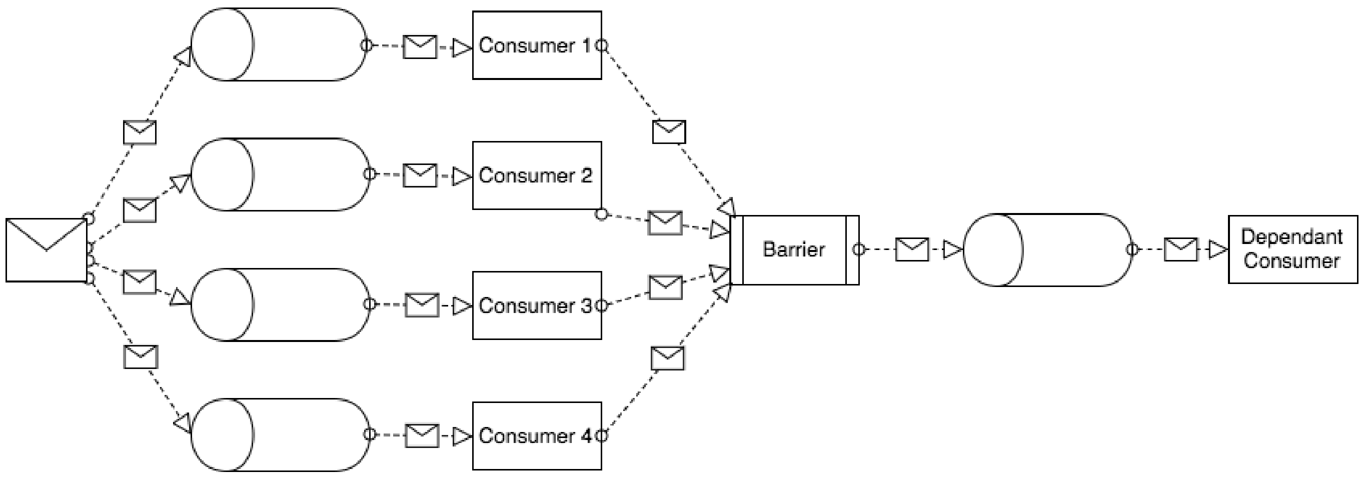
Consider the following situation: we have the write index at the end of the array and the read index at the beginning. Is it safe to wrap around?



Well, it’s easy to imagine a streaming example where stale data is not needed anymore because the fresher and more relevant data keeps coming, but usually, we do care about not yet processed data. If that’s the case, either returning a false boolean or blocking, as traditional bounded queues do, would work. If none of these solutions satisfies us, we can implement a ring buffer which can resize itself (but only when it gets full, not just when a producer reaches the end of the array and it can safely wrap around). Resizing would require moving all the elements to a newly allocated bigger array (if an array is used as an underlying data structure) which is an expensive operation, of course.

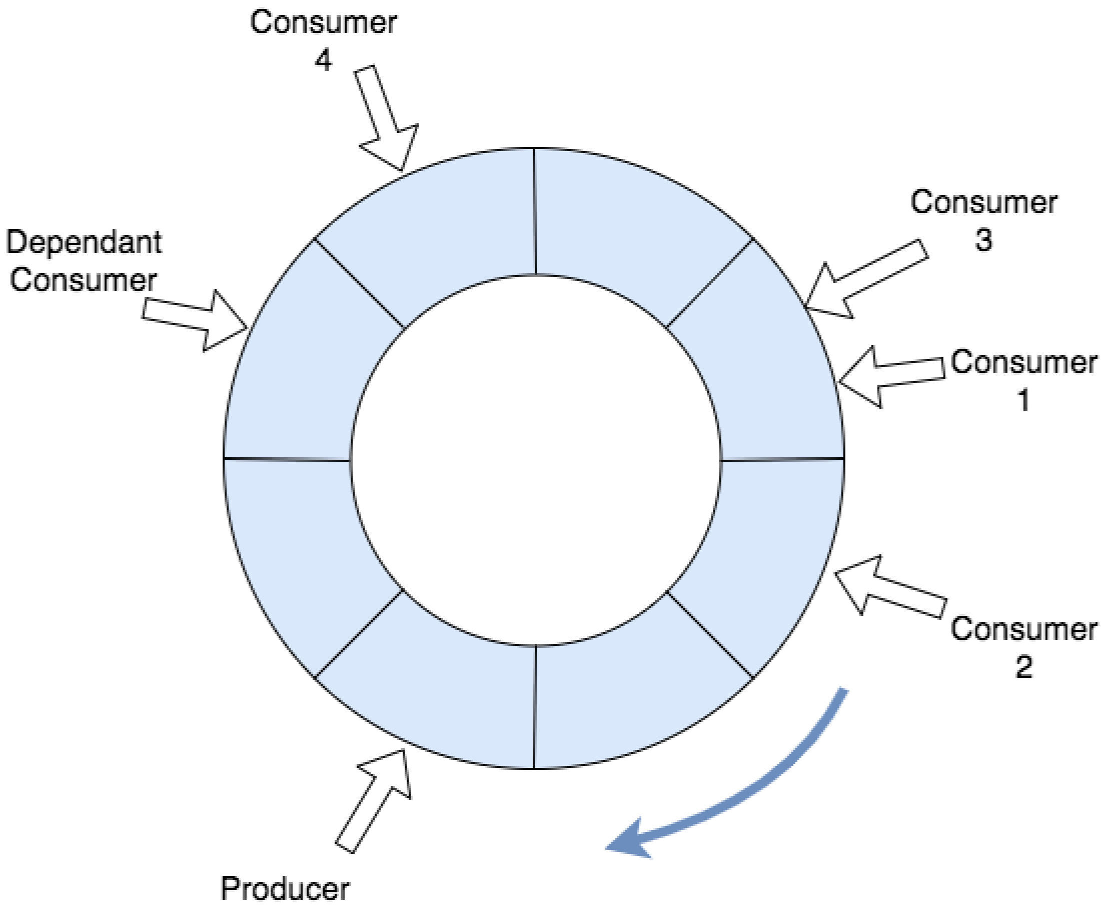
Ring Buffer in Disruptor

What kind of problem does ring buffer solve in Disruptor? Let’s look at the typical use case for Disruptor:



As you can see, we have a single message being sent to the input queues of multiple consumers (1-4) - so it’s a typical multicast (sort of publish and subscribe semantics). Then we have a barrier - we want all consumers to finish processing the message before we move to the next step. When the barrier is passed, the message goes to the input queue of the last consumer. To sum up, we have four independent (and thus potentially parallel) tasks (Consumers 1-4) operating on the same data and one task (Dependant Consumer) which requires all the previous tasks to be finished before it starts working.

And where is the ring buffer in this architecture? Well, a ring buffer is actually this architecture.



Let’s imagine that a message is a single item in our ring buffer. Ring buffer represents our actual queue (or multiple downstream queues if you prefer). Each consumer is a separate thread traversing ring buffer round and round (or consuming messages from the queue if you prefer).

The barrier is implemented in a way that dependant consumer cannot go past any of the consumers which are required to be finished before it starts processing a ring buffer item (a message).

Single Writer Principle

Even though the most performant way of using Disruptor is to have a single producer and multiple consumers, a multi-producers scenario is also possible. In that case, there might be a write contention on the same cursor. Disruptor doesn't use traditional locks to solve this problem - so it’s lock-free (with the exception of BlockingWaitStrategy). Instead, it relies on memory barriers and/or high-performance CAS operations.

What if the dependant consumer requires some data produced by any of the previous consumers? Can a consumer also modify the item in the ring buffer and thus be a writer? It turns out that a very simple rule is used here - each item in a ring buffer consists of a set of fields and each field has, at most, one consumer which is allowed to write to it. This prevents any write-contention between consumers.

Summary

There are many other things which makes Disruptor fast, for example:

- Consuming messages in a batching mode.
- CPU-cache-friendly benefits using data locality in a ring buffer.
- Preallocation of the messages in ring buffer to avoid frequent garbage collection of the items (so they are reused).

As I explained here, even though ring buffer is a relatively simple data structure, it can be easily used to implement way more complex scenarios and Disruptor is just an example.