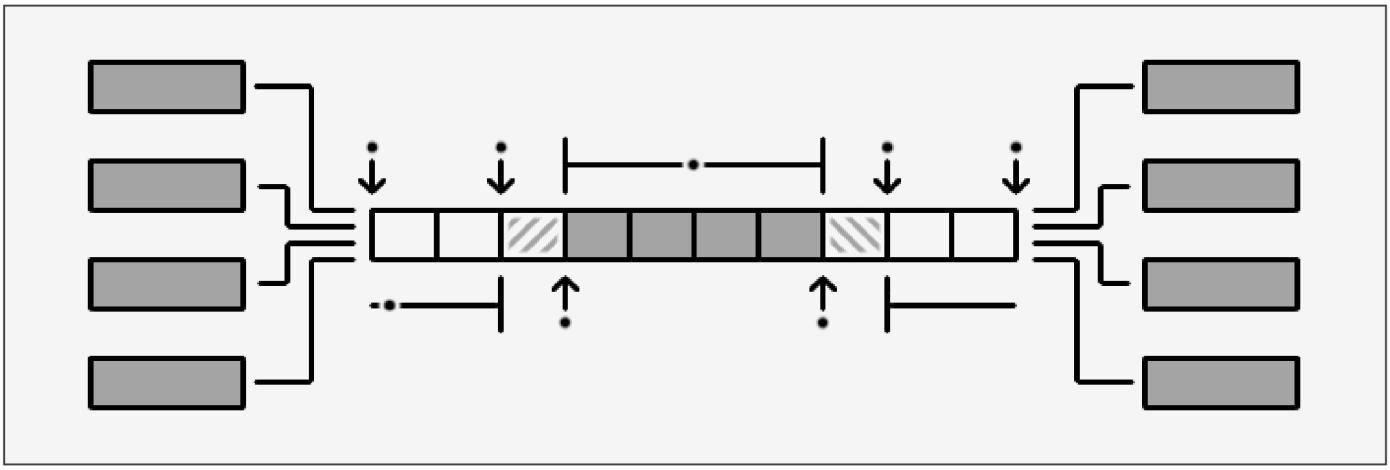
kmdreko's
A SIMPLE BLOG

A Simple Lock-free Ring Buffer

OCTOBER 3, 2019 | C++, UTILITY, MULTITHREADING

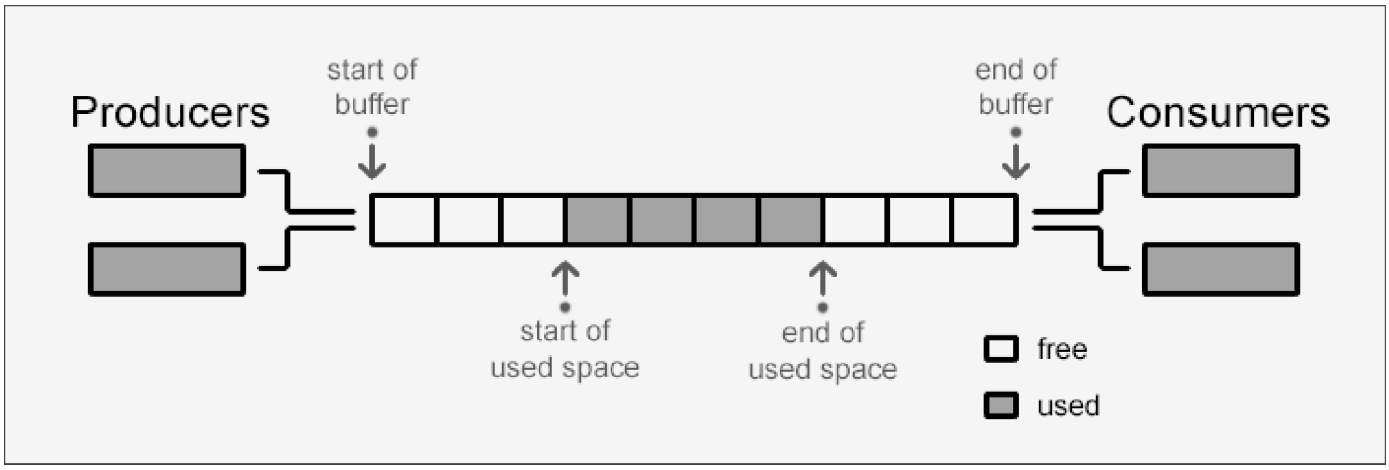
A while back, I wanted to try my hand at writing a lock-free, multi-producer, multi-consumer ring-buffer. My goal was to design a concurrent queue that didn't use any locks, allocations, or thread-count dependent structures. The final code can be found on my **GitHub**.



Basic Design

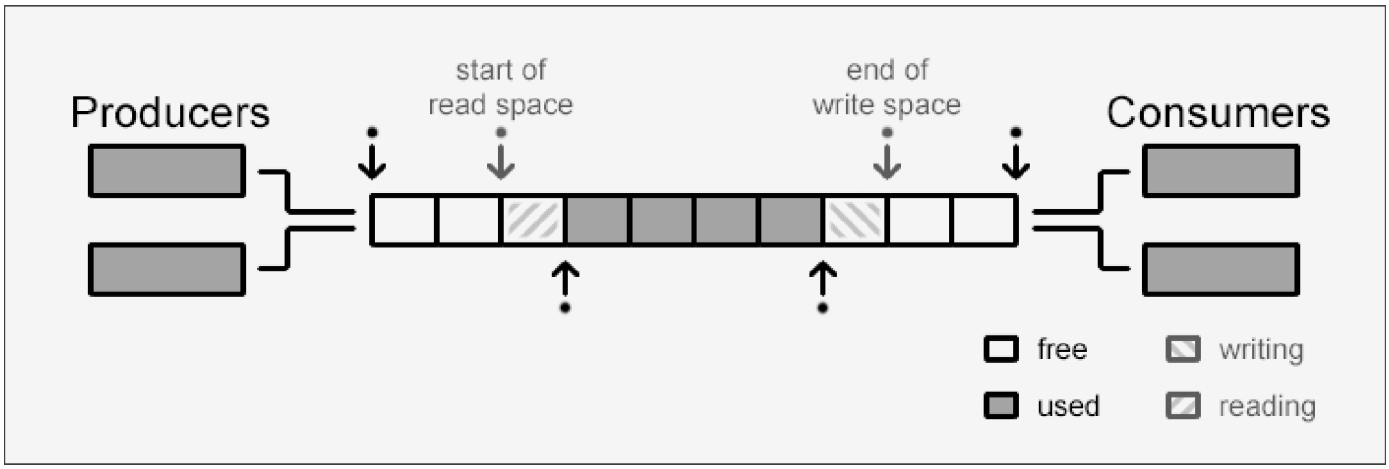
For those that don't know, a ring buffer, also called a circular buffer, is a type of queue with a fixed maximum allowed size that continually reuses the allocated space to store the elements.

When writing a simple non-concurrent ring buffer, it has to have at least four values: two for the start and end of the allocated buffer, and two for the start and end of the used space. It is pretty simple to imagine an enqueue function would simply check there is space, create a new element at the end of the used space, and increment the end marker. A dequeue function would behave similarily with the start of the used space.



Naive Lock-free Design

To account for concurrent operations without using a lock, a bit more bookeeping is required. The solution is to add markers to account for the space where elements are added or removed but aren't yet completed. The enqueue function would load the end marker for the write space, update it, create the element, and update the used space to allow it to be read (all using atomic operations of course).



Unfortunately, there is an issue; the enqueue function cannot update the used space without regard to previous writes. Consider the situation where thread 1 has started a write followed by thread 2 but thread 2 finished before thread 1, it cannot announce that whole space is available when thread 1 might not be finished. The way I decided to work around this was to have operations wait until previous operations are completed by checking against the free and used markers. In classifying algorithms, this implementation is still "lock-free" but it is not "wait-free" since some threads may have to wait on others.

There is actually an even more troubling issue, this particular design suffers from ABA problems.

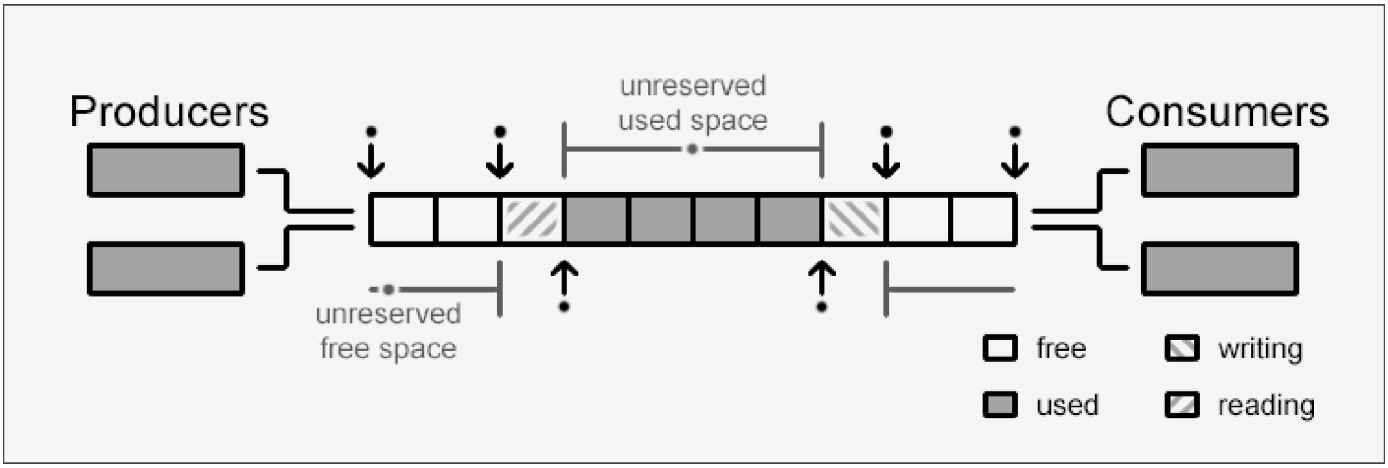
> *The ABA problem occurs during synchronization, when a location is read twice, has the same value for both reads, and "value is the same" is used to indicate "nothing has changed". However, another thread can execute between the two reads and*

> *change the value, do other work, then change the value back, thus fooling the first thread into thinking "nothing has changed" even though the second thread did work that violates that assumption. Source:* ***Wikipedia***

The enqueue and dequeue operations both require loading a marker, checking for space, and updating that marker via compare-and-swap. The compare-and-swap is intended to verify nothing has changed so it assumes it successfully acquired the write space, but that doesn't consider that other operations may have ocurred such that there's no space available anymore. This does require the buffer to wrap around on itself, but it can happen, especially with limited space. Ring buffers are particularly susceptible to ABA problems because they reuse memory by-design.

Final Design

The way I solved this required adding variables to keep track of the used and free space and implemented what I called a "reserve-commit" system. As part of the enqueue function, before "committing" to the write by updating the marker, it would first decrement the free space to "reserve" it. If the compare-and-swap failed, it would "un-reserve" by adding back the free space and try again, if it succeeded, it would continue as normal. The free space would be incremented later at the end of the dequeue function.



It may seem like this only adds more variables that don't address the problem. You'd be right, these don't help avoid ABA scenarios, but it no longer matters. The issue that the ABA problem caused was the potential to not have space to work with, but with this change the space is already "reserved", so there will always be space.

Having these additional variables also removes the ambiguity where a full buffer and an empty buffer end up looking the same.

Performance

A concurrent queue without allocations or locks! It must be very fast, right? Well...

It performs pretty terribly. I wasn't super interested in performance initially, but I did get it running in the **moodycamel::ConcurrentQueue** benchmarks to see how it did. It performed well in the single-producer-single-consumer scenarios, and it was fine in the two-producers-two-consumers scenarios, but any more threads and the performance dropped off dramatically.

The reason for the poor performance is primarily due to the number of atomic operations that are required. They aren't free and this design needs six for each enqueue or dequeue operation. This magnifies the performance impact of any thread contention on the ring buffer. A high-quality concurrent queue would strive to keep the atomic operations to a minimum. So I don't recommend using it in any real application.

Conclusion

Even though what I built didn't end up being useful, I was really pleased with the result. I finally got around to understanding the C++ various memory order behaviours. I strengthened my understanding of lock-free algorithms. I discovered first-hand that ring buffers aren't that great for concurrent queues. And I confirmed the conventional wisdom of "don't roll your own", or if you do, at least test it first.