Building a lockfree SPSC bounded queue
std::atomic, memory ordering, and false sharing

Abstract

This poster explores optimization techniques for designing an efficient Single Producer Single Consumer (SPSC) bounded queue using atomic operations. It presents benchmarking results comparing various implementations, and `boost::lockfree::spsc_queue` .

Introduction

Efficient inter-thread communication is a crucial aspect of high-performance systems, particularly in low-latency environments such as trading systems. Traditional synchronization mechanisms, such as mutexes often introduce significant overhead and degrade performance under high-frequency workloads. Lock-free data structures offer an alternative by leveraging non-blocking atomic operations. This poster explores key techniques such as memory ordering to minimize synchronization overhead and avoid unnecessary fencing.

Profiling Setup

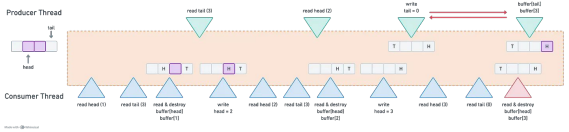
The queue has a fixed buffer size. The performance will be tested with 1 million integers, using a buffer size of 1,000. Producer and consumer threads are pinned to separate physical cores and continuously poll for data.

Lockfree Queue v1

```
template <typename T>
struct SPSCQueue {
    bool push(const T& item)
    {
        std::size_t const currTail = tail.load();
        std::size_t const nextTail = increment(currTail);
        if (nextTail == head.load())
            return false;
        new (buffer + currTail) T(item);
        tail = nextTail;
        return true;
    }
    bool consume_one(auto&& func)
    {
        std::size_t const currHead = head.load();
        if (currHead == tail)
            return false;
        T* elem = reinterpret_cast<T*>(buffer+currHead);
        func(*elem);
        elem->~T();
        head = increment(currHead);
        return true;
    }
    struct Element
    {
        alignas(T) std::byte storage[sizeof(T)];
    };

    std::size_t const capacity;
    std::atomic<std::size_t> head = 0;
    std::atomic<std::size_t> tail = 0;
    Element* buffer;
};
```

Memory Ordering

In modern systems, the CPU may reorder instructions dynamically to maximize efficiency and instructions per cycle (IPC) — a technique known as out-of-order execution. Out of order atomic update and memory access may lead to incorrect behaviour.



Out of order atomic update and memory access leads to incorrect behavior

Memory ordering can be specified in the load and store instructions to relax or enforce synchronization constraints.

Optimization: Using relaxed ordering

Relaxed: Relaxed ordering only ensures the atomicity of the read/write to the atomic variable. There are no synchronization or ordering constraints imposed on other reads or writes. The CPU is allowed to reorder the operations.

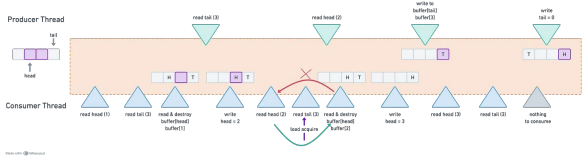
```
bool push(const T& item)
    auto currTail = tail.load(std::memory_order_relaxed);
bool consume_one(auto&& func)
    auto currHead = head.load(std::memory_order_relaxed);
```

Presented by Sarthak Sehgal
sartech.substack.com | github.com/sarthak-sehgal

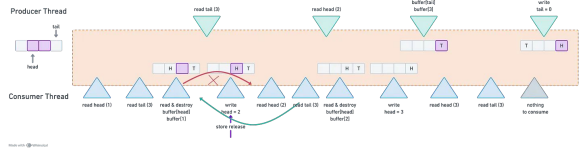
Building a lockfree SPSC bounded queue
std::atomic, memory ordering, and false sharing

std::memory_order

Acquire: A load operation with acquire ordering ensures that no reads or writes in the current thread can be reordered before this load.



Release: A store operation with release ordering ensures that no reads or writes in the current thread can be reordered *after* this store.



Sequentially Consistent: A load operation with this memory order performs an acquire operation, a store performs a release operation. *Additionally, a single total order exists in which all threads observe all modifications in the same order.* This is the default memory order. Sequentially consistent ordering synchronizes amongst different atomic variables which makes it less efficient than release/acquire.

Optimization: Using release-acquire

```
bool push(const T& item)
{
    auto currTail = tail.load(std::memory_order_relaxed);
    auto nextTail = increment(currTail);
    if (nextTail == head.load(std::memory_order_acquire))
        return false;
    ...
    tail.store(nextTail, std::memory_order_release);
}

bool consume_one(auto&& func)
{
    auto currHead = head.load(std::memory_order_relaxed);
    if (currHead == tail.load(std::memory_order_acquire))
        return false;
    ...
    head.store(increment(currHead), std::memory_order_release);
}
```

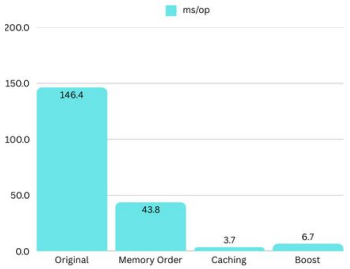
Optimization: Alignment & Caching

```
push:
    if (nextTail == headCache)
    {
        headCache = head.load(std::memory_order_acquire);
        if (nextTail == headCache)
            return false;
    }
pop:
    if (currHead == tailCache)
    {
        tailCache = tail.load(std::memory_order_acquire);
        if (currHead == tailCache)
            return false;
    }

alignas(CACHE_LINE) std::atomic<std::size_t> head = 0;
std::size_t tailCache = 0;

alignas(CACHE_LINE) std::atomic<std::size_t> tail = 0;
std::size_t headCache = 0;
```

False sharing & Caching

False sharing occurs when two or more threads access different variables that are located on the same cache line. In the queue implementation, the head and tail variables will reside in the same cache line. When the producer thread updates the tail, it leads to the cache line being modified and the consumer thread refetching each time it accesses the head or the tail. This can be addressed by aligning the head and tail as *hardware_destructive_interference_size*. The queue can be further optimized by caching the head and tail.

Results



Using proper memory ordering achieves a tenfold performance improvement. Additionally, optimizing cache alignment and caching the head/tail results in a queue that outperforms Boost's SPSC queue.

Complete code can be found on my blog and github. Credits to Erik Rigtorp's implementation for the head and taill index caching

Presented by Sarthak Sehgal
sartech.substack.com | github.com/sarthak-sehgal