SPSC Bounded Queue

Ditch the lock, speed the Queue!

Sarthak Sehgal

C++ Software Engineer

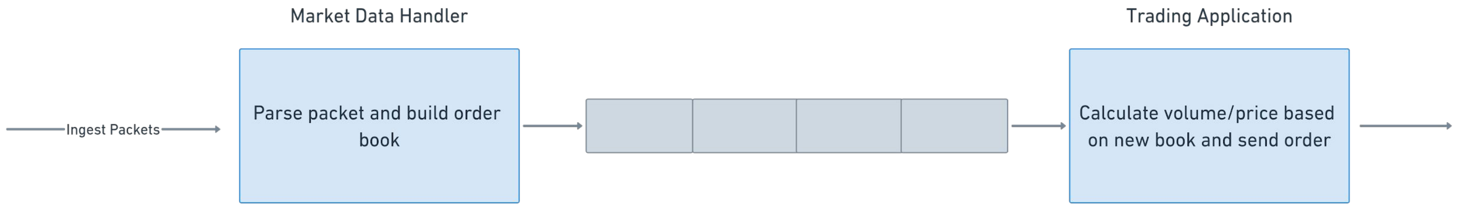
- Working at a high frequency options market making firm
- Interested in finance, low level programming, and C++ under the hood
- sartech.substack.com
- LinkedIn://sarthaksehgal99

Setup

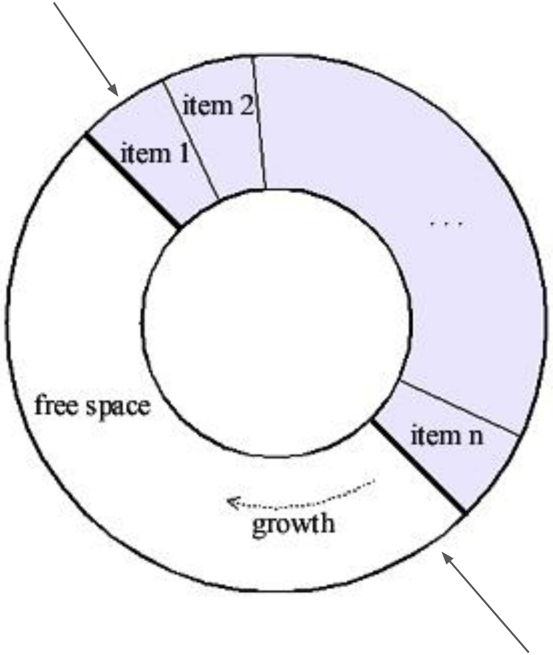
- Exactly one producer and one consumer
- Fixed capacity
- Producer and consumer threads are pinned to separate physical cores and continuously poll for data

Setup

- Exactly one producer and one consumer
- Fixed capacity
- Producer and consumer threads are pinned to separate physical cores and continuously poll for data

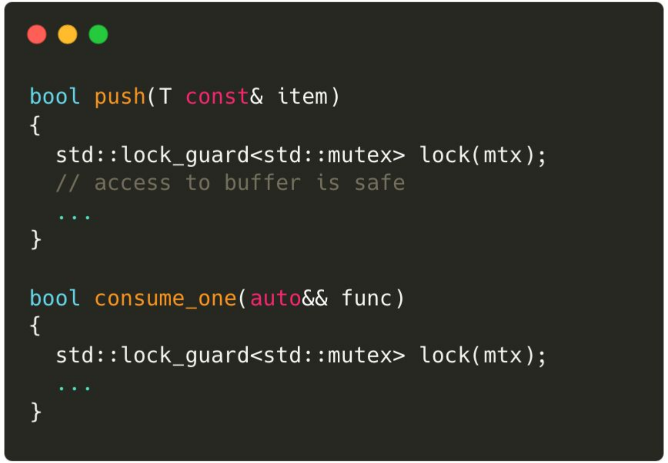

Market Data Handler
Trading Application
Ingest Packets
Parse packet and build order book
Calculate volume/price based on new book and send order

head (read end)



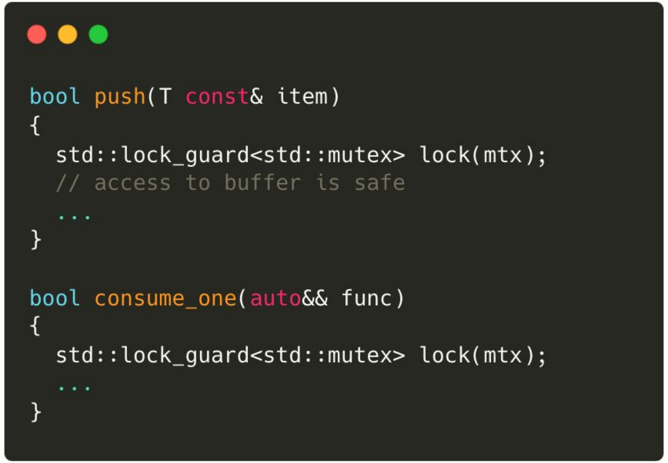
tail (write end)

v1: Good old mutex



```
bool push(T const& item)
{
  std::lock_guard<std::mutex> lock(mtx);
  // access to buffer is safe
  ...
}

bool consume_one(auto&& func)
{
  std::lock_guard<std::mutex> lock(mtx);
  ...
}
```

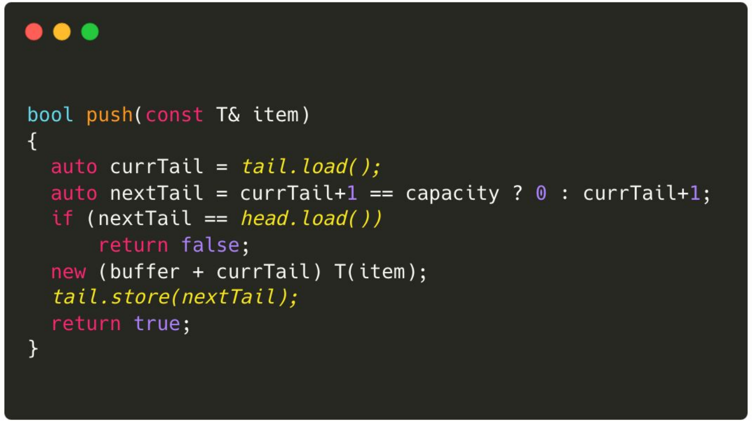
v1: Good old mutex



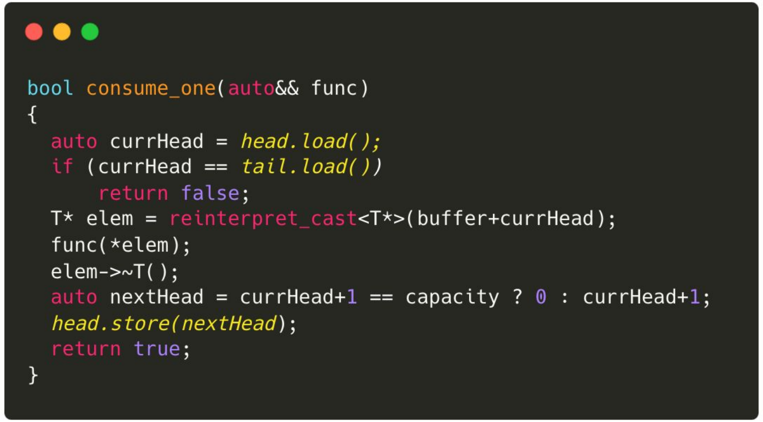
```
bool push(T const& item)
{
  std::lock_guard<std::mutex> lock(mtx);
  // access to buffer is safe
  ...
}

bool consume_one(auto&& func)
{
  std::lock_guard<std::mutex> lock(mtx);
  ...
}
```

blocking, expensive system calls

v2: Using atomics



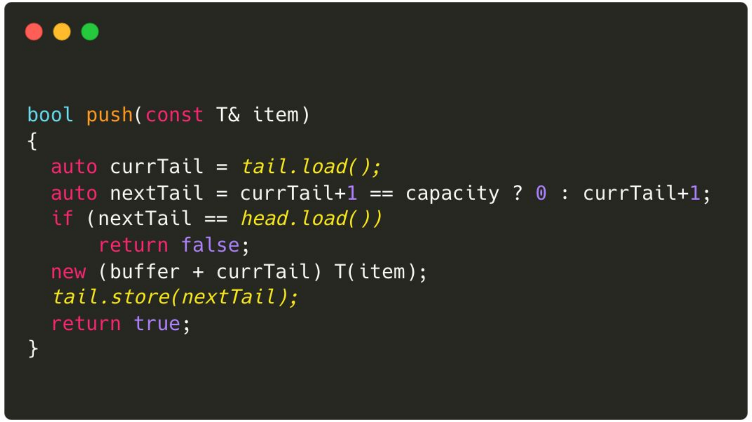
```
bool push(const T& item)
{
  auto currTail = tail.load();
  auto nextTail = currTail+1 == capacity ? 0 : currTail+1;
  if (nextTail == head.load())
      return false;
  new (buffer + currTail) T(item);
  tail.store(nextTail);
  return true;
}
```



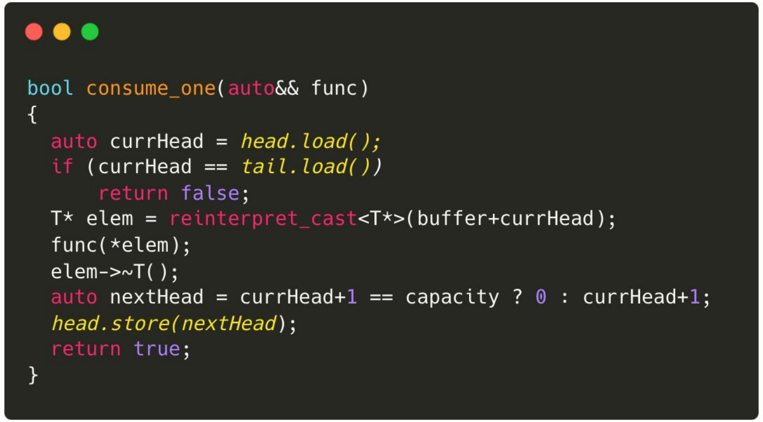
```
bool consume_one(auto&& func)
{
  auto currHead = head.load();
  if (currHead == tail.load())
      return false;
  T* elem = reinterpret_cast<T*>(buffer+currHead);
  func(*elem);
  elem->~T();
  auto nextHead = currHead+1 == capacity ? 0 : currHead+1;
  head.store(nextHead);
  return true;
}
```

atomics are used to *synchronize access* to the shared memory

v2: Using atomics



```
bool push(const T& item)
{
  auto currTail = tail.load();
  auto nextTail = currTail+1 == capacity ? 0 : currTail+1;
  if (nextTail == head.load())
      return false;
  new (buffer + currTail) T(item);
  tail.store(nextTail);
  return true;
}
```

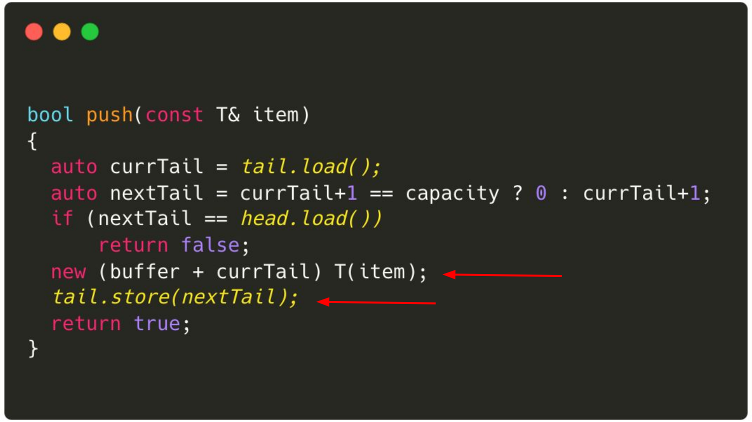


```
bool consume_one(auto&& func)
{
  auto currHead = head.load();
  if (currHead == tail.load())
      return false;
  T* elem = reinterpret_cast<T*>(buffer+currHead);
  func(*elem);
  elem->~T();
  auto nextHead = currHead+1 == capacity ? 0 : currHead+1;
  head.store(nextHead);
  return true;
}
```

atomics are used to *synchronize access* to the shared memory

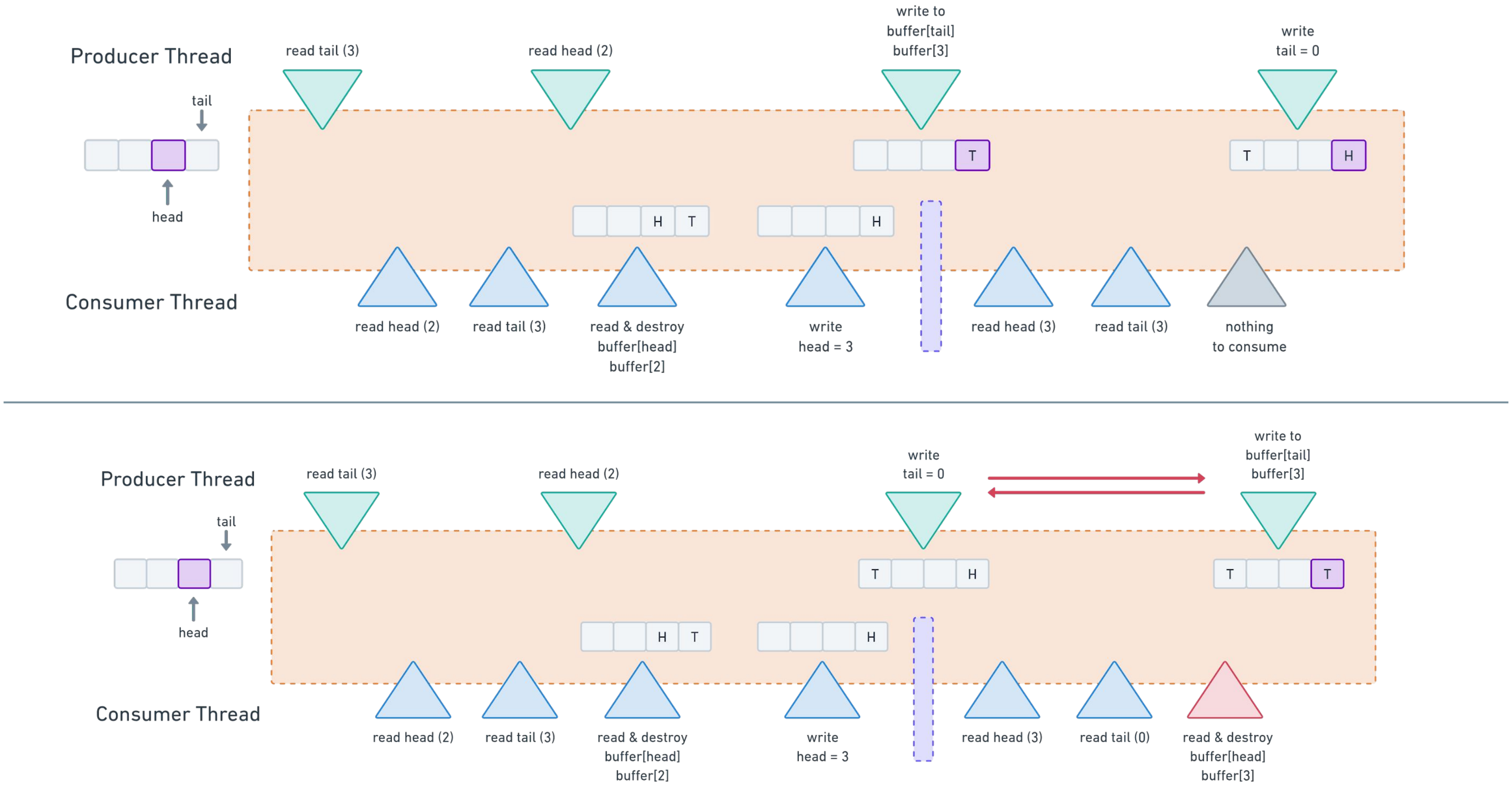
non-blocking ✅ lock-free ✅ strict memory ordering ⚠️

Out of order execution



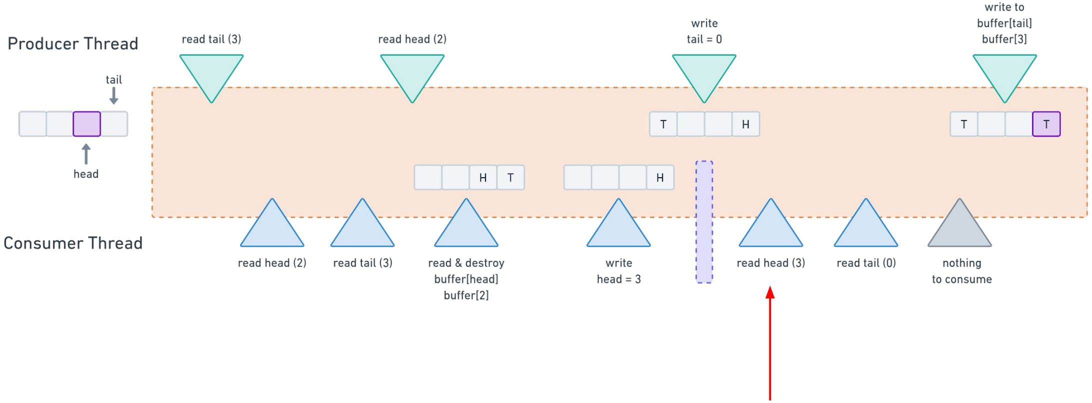
```
bool push(const T& item)
{
  auto currTail = tail.load();
  auto nextTail = currTail+1 == capacity ? 0 : currTail+1;
  if (nextTail == head.load())
      return false;
  new (buffer + currTail) T(item);
  tail.store(nextTail);
  return true;
}
```

Out of order execution

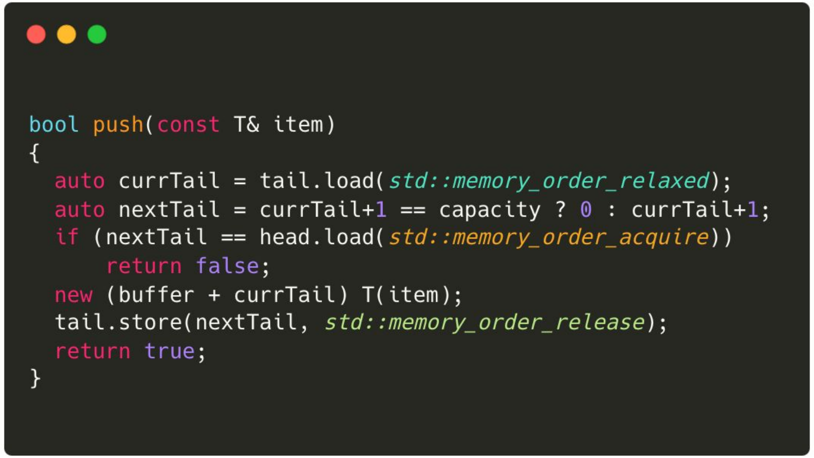


Memory Ordering

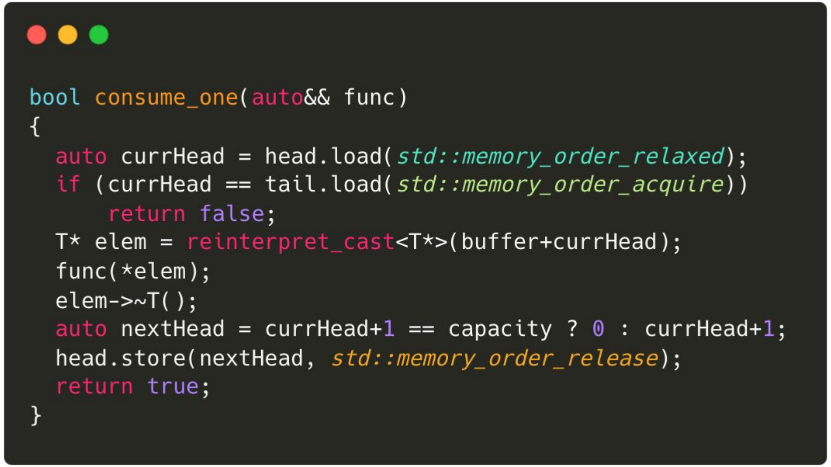
relaxed: only atomicity, no ordering constraints



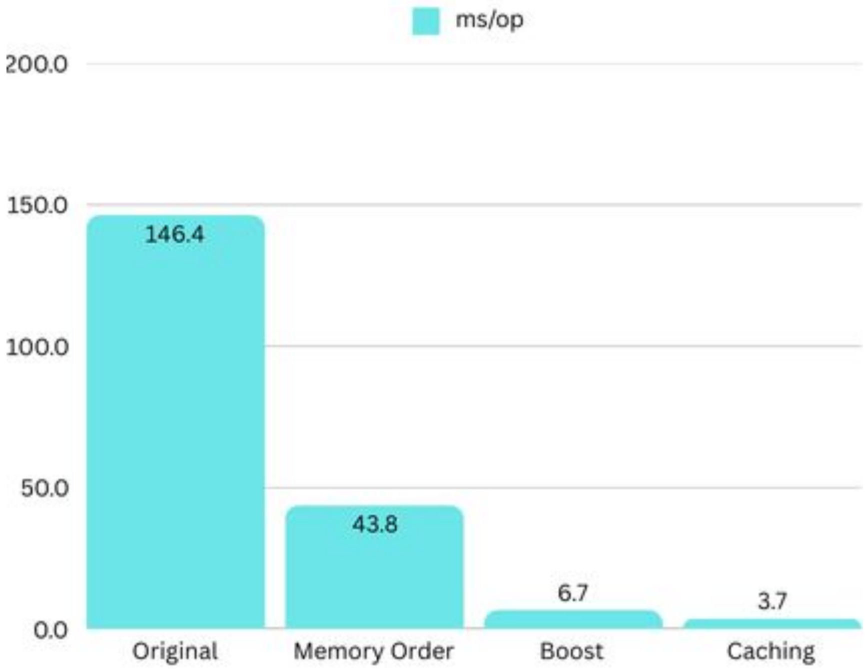
v2.1: Optimized memory ordering



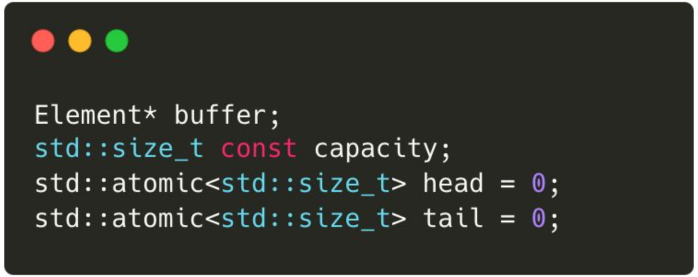
```
bool push(const T& item)
{
  auto currTail = tail.load(std::memory_order_relaxed);
  auto nextTail = currTail+1 == capacity ? 0 : currTail+1;
  if (nextTail == head.load(std::memory_order_acquire))
      return false;
  new (buffer + currTail) T(item);
  tail.store(nextTail, std::memory_order_release);
  return true;
}
```



```
bool consume_one(auto&& func)
{
  auto currHead = head.load(std::memory_order_relaxed);
  if (currHead == tail.load(std::memory_order_acquire))
      return false;
  T* elem = reinterpret_cast<T*>(buffer+currHead);
  func(*elem);
  elem->~T();
  auto nextHead = currHead+1 == capacity ? 0 : currHead+1;
  head.store(nextHead, std::memory_order_release);
  return true;
}
```

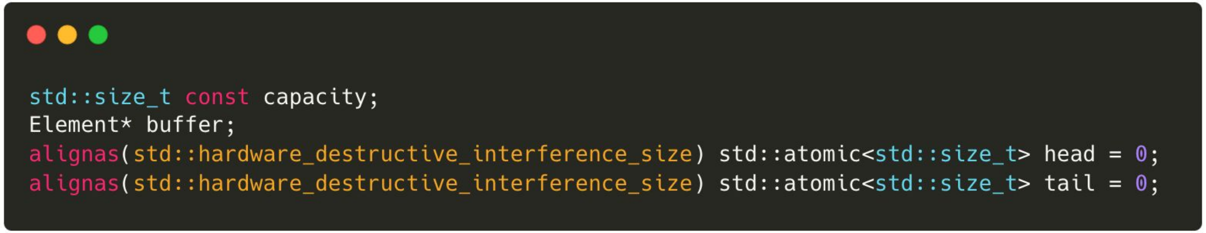

ms/op
200.0
150.0
100.0
50.0
0.0
146.4
43.8
6.7
3.7
Original
Memory Order
Boost
Caching

False Sharing



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

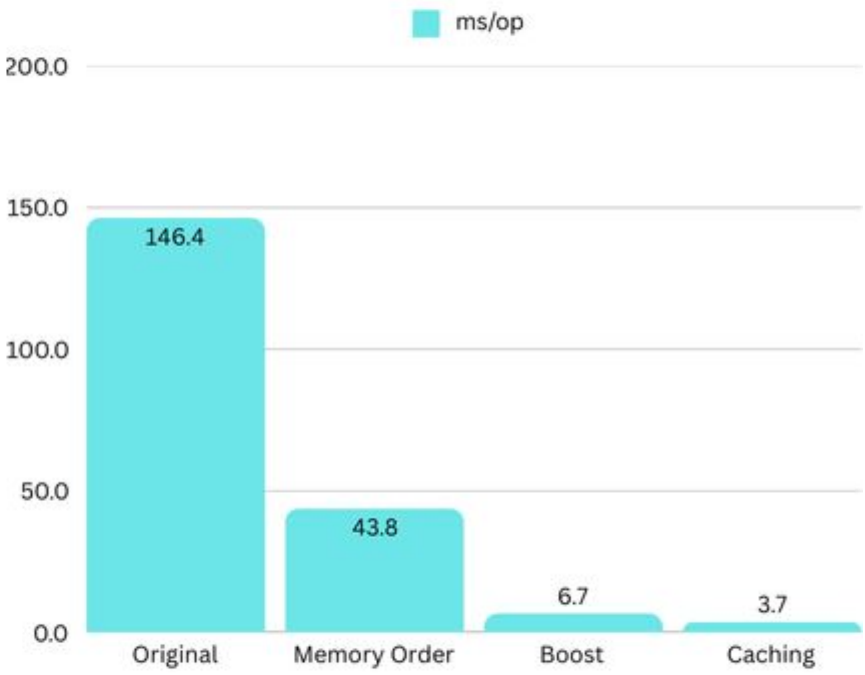
False Sharing



```cpp
std::size_t const capacity;
Element* buffer;
alignas(std::hardware_destructive_interference_size) std::atomic<std::size_t> head = 0;
alignas(std::hardware_destructive_interference_size) std::atomic<std::size_t> tail = 0;
```

Minimum offset between two objects to avoid false sharing

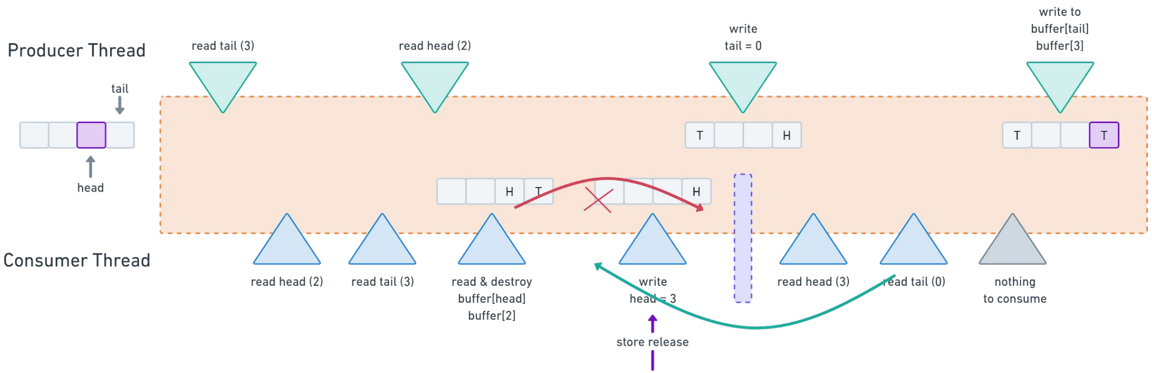
Since C++17



Interested to learn more? Hop over to my poster booth tomorrow

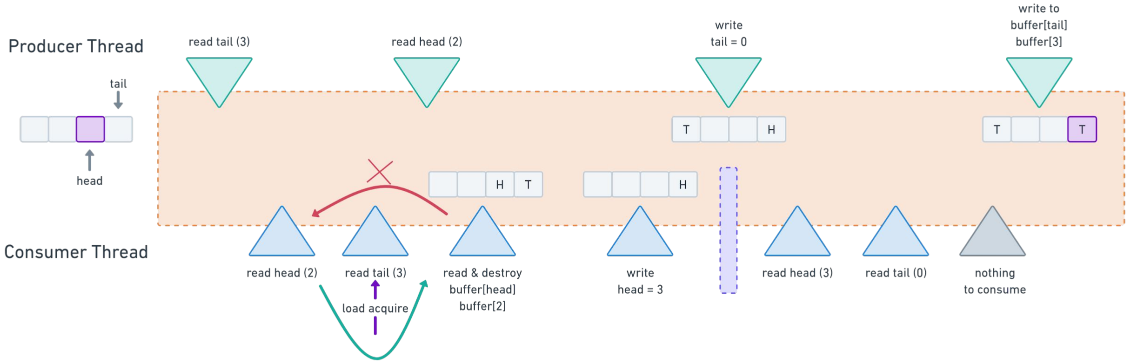
Memory Ordering

release: no reads/writes in current thread can be reordered <u>after</u> the *store*



Memory Ordering

acquire: no reads/writes in current thread can be reordered before the *load*



All writes in other threads that release the same atomic variable are visible in the current thread