Original Prezi viewable at:
https://prezi.com/jjtdti-h_sil/unbounded-array-based-lock-free-multi-producer-multi-con/

Unbounded, Array-Based, ~Lock-Free, Multi-Producer, Multi-Consumer Concurrent EIEO Queue - *UABLFMPMCCEIEOQ*

Jaedyn Kitt Draper
Epoch Labs
jaedyn.cppcon@jaedyn.co

https://swarm.workshop.perforce.com/users/ShadauxCat/
http://github.com/ShadauxCat

Disclaimer

- This is young code - not ready for production use!
- It's only been tested on x86_64
    - (But it works on x86_64)
- Memory barriers may need tweaking on other platforms (and can probably be relaxed)
- Likely additional room for improvement

Motivation

- ~~Performance! (Duh)~~
  - ~~But lock free isn't always faster~~
- ~~Reduce thread contention.~~
- ~~Avoid deadlocks, livelocks, etc.~~
- Because...
  - Lock-free is hard.
  - Correct lock-free is harder.
  - Correct generic lock-free is even harder.
  - Controversial statement: Lock-free is fun!

So, what are we looking for?

- **Generic queue (templated)**
- **Array-based**
  - **(Mostly) contiguous**
  - **(Mostly) cache-friendly**
  - **Few dynamic allocations**
- **Unbounded (capable of growth)**
  - **Difficult for lock-free/array-based because array can't be reallocated**
- **EIEO (Early In/Early Out)**
  - **FIFO has questionable meaning in threading - order in memory does not guarantee order of retrieval**
  - **Items added to the queue earlier are statistically likely to be retrieved earlier - *New items added to the queue will not be placed ahead of any items already fully in the queue***
  - **In single-threaded environment... FIFO**
  - **Also referred to as "Approximately FIFO"**
  - **(Or because I know you're thinking it... Early In/Early Issued Out.)**
- **Memory safe:**
  - **No leaks**
  - **No corruption**
  - **No dangling**

First Attempt: Linked List of Arrays

- **Forward-linked list with "head" and "tail" objects**
- **Each item in the list is an array with *N* elements in it.**
- **Typical std::deque implementation**
- **~~This worked!~~**
- **...Until I cared about actually releasing memory.**
- **Safely deleting an array that may be referenced by both head and tail across *N* consumers and *M* producers proved difficult.**
  - **Why? Because head and tail might point to the same thing, and if they do and that item is released, *both head AND tail must be modified atomically with respect to each other!***
  - **If one is destroyed and then the other accessed... Dangling pointer!**

New Strategy: Nested Array Structure

- **Nested structure masquerades as contiguous array, though in reality it's only contiguous for a certain active range.**

- **Designed for maximum efficiency in the most common case - efficiency drops very rarely.**

- **In memory, similar to a linked list of arrays - difference is access with increase-only indexes, which allows for better performance.**

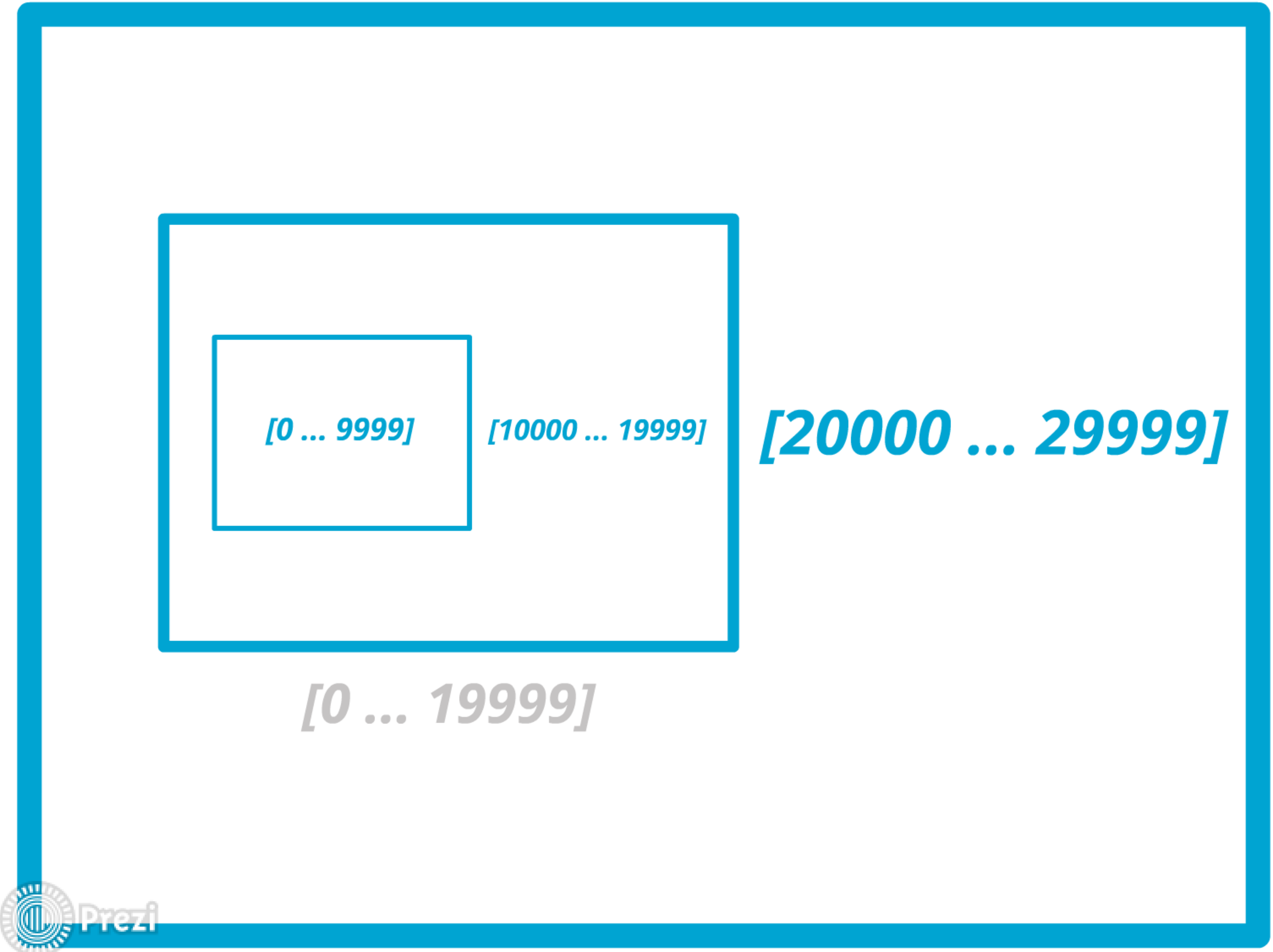
- **Because only one pointer is kept, it's possible to atomically release it when clearing up memory**

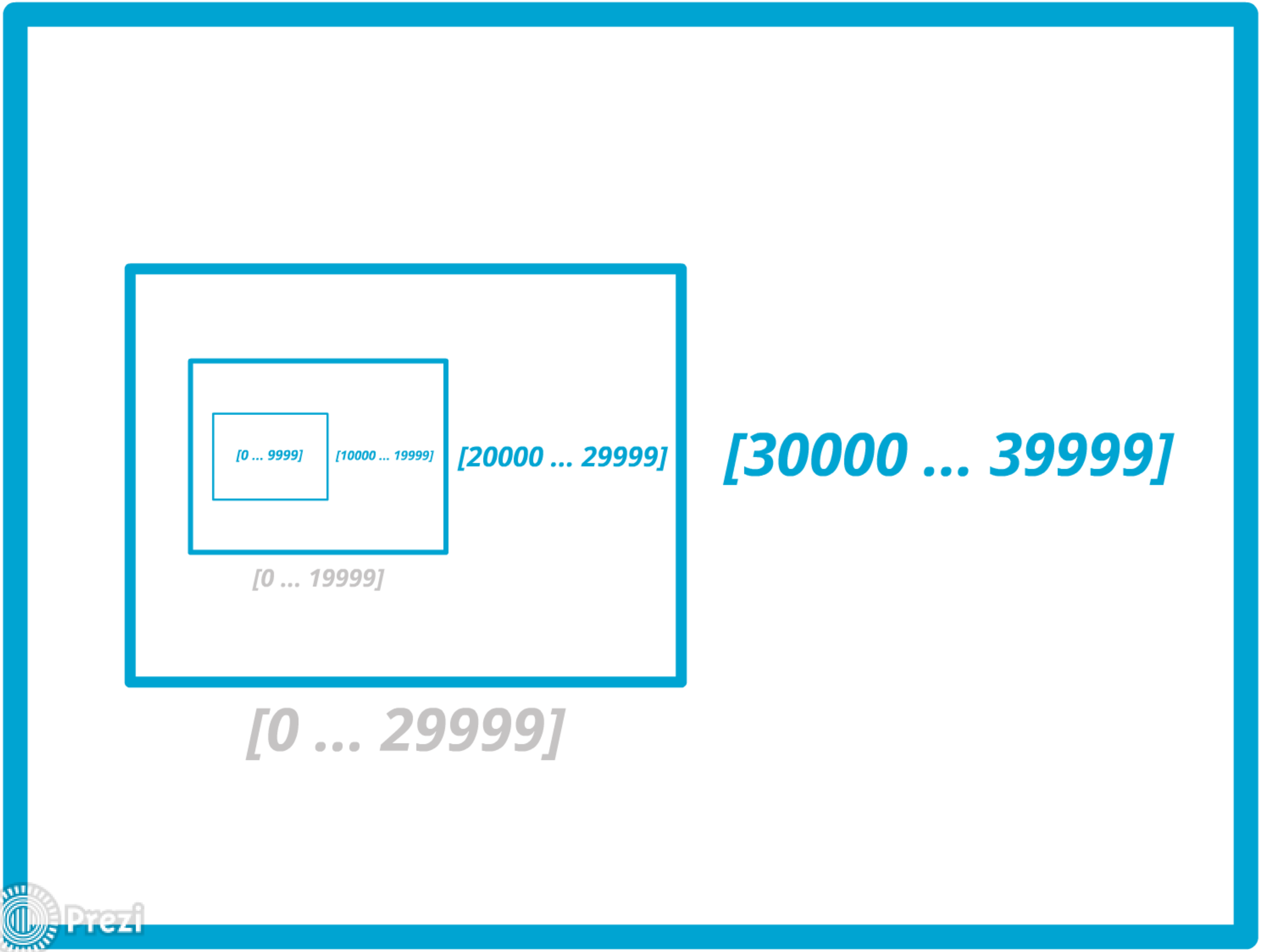
- **Unlike most linked lists, pointer is to tail, where most operations for both producers and consumers will take place. Head is mostly forgotten once a new link is added.**

[0 ... 9999]

[0 ... 9999]

[10000 ... 19999]


[0 ... 9999]
[10000 ... 19999]
[20000 ... 29999]
[0 ... 19999]
Prezi


[0 ... 9999]
[10000 ... 19999]
[20000 ... 29999]
[30000 ... 39999]
[0 ... 19999]
[0 ... 29999]

New Strategy: Nested Array Structure

- **"Black-Hole" array - index increments forever, slots are never reused.**
- **Block keeps a count of how many indexes have been consumed - when none are left, it marks itself for deletion and can be cleaned up by reference counting.**
- **In the vast majority of cases (assuming sufficient block size), all reads and writes will go to the last block of the array and not incur recursion.**

Enqueue Operation

```
void Enqueue(T const& val) {
   RefCounter buffer(m_buffer.load(std::memory_order_acquire));

   // Reserve a position before doing anything.
   // Incrementing before storing is safe because of a boolean in each item that gets set later
   size_t pos = m_writePos.fetch_add(1, std::memory_order_acq_rel);

   // But what if the queue is full...? It's rare, but if it happens, we have to reallocate...
   // And only one thread can perform the allocation! The others will have to wait!

   BufferElement& element = buffer->GetForAdd(pos);

   //Construct the item and then set "state" to READY - order matters.
   new(&element.item) T(val);
   element.state.store(ElementState::READY, std::memory_order_release);
}
```

Enqueue Operation: Reallocation

- Exceptional case operation that occurs almost never
- When it occurs, it must be synchronized
- A simple, fast call can be made to detect the exceptional case
- Only one thread can perform the operation...

Double-Check Lock!

(Technically this breaks lock free...)

(But it's a tradeoff to achieve unbounded)

Enqueue Operation

```
void Enqueue(T const& val) {
   RefCounter buffer(m_buffer.load(std::memory_order_acquire));

   // Reserve a position before doing anything.
   // Incrementing before storing is safe because of a boolean in each item that gets set later
   size_t pos = m_writePos.fetch_add(1, std::memory_order_acq_rel);

   // If the position is greater than the capacity, we need to grow the buffer.
   while(pos >= buffer->Capacity()) {

      bool expected = false;

      if(m_reallocatingBuffer.compare_exchange_strong(expected, true, std::memory_order_acq_rel)) {
         // Won the reallocation lottery!
         // Double check that we actually need it...
         size_t capacity = buffer->Capacity();

         if(pos >= capacity) {
            //Create a new buffer by extending the current one
            Buffer* newBuffer = new Buffer(*buffer);
            m_buffer.store(newBuffer, std::memory_order_release);
            buffer->DecRef();
         }

         //We're done reallocating, clear the lottery flag.
         m_reallocatingBuffer.store(false, std::memory_order_release);
      }
      else {
         // Let another thread move forward so this one's not just looping endlessly.
         sched_yield();
      }

      // Whether we reallocated or not, we need to reacquire the buffer now.
      buffer = m_buffer.load(std::memory_order_acquire);
   }

   // -----

   BufferElement& element = buffer->GetForAdd(pos);

   //Construct the item and then set "state" to READY - order matters.
   new(&element.item) T(val);
   element.state.store(ElementState::READY, std::memory_order_release);
}
```

Dequeue Operation

```
bool Dequeue(T& val) {
   // Loop because we may have to try a few times to get a value.
   // Using a label instead of a for loop to avoid needing an extra boolean to get out of nested loop
   startLoop:
   RefCounter buffer(m_buffer.load(std::memory_order_acquire));

   while(*buffer == nullptr)
      // In the middle of an exchange, try again RIGHT NOW, don't return.
      buffer = m_buffer.load(std::memory_order_acquire);

   // Get the current position to read from. We're not incrementing it yet because it may be empty
   // If that's the case, nothing has been added to the queue, so we return false.
   size_t pos = m_readPos.load(std::memory_order_acquire), writePos = m_writePos.load(std::memory_order_acquire);
   if(pos >= writePos || pos >= buffer->Capacity())
      return false;

   size_t readPos = pos;

   Buffer* bufObtainedFrom;
   BufferElement* element;

   for( ;pos < writePos; ++pos) {
      // Try to get an item for removal. It's possible this will fail due to the buffer we have being released.
      // If this fails we need to go back and reobtain the buffer and try again.
      if(!buffer->GetForRemove(pos, bufObtainedFrom, element))
         //Break out of the inner loop and jump to the top of the outer loop - goto is more efficient than a boolean
         goto startLoop;

      // If this fails, another thread beat us to the value at this position. Try again with the next value.
      ElementState expected = ElementState::READY;
      if(element->state.compare_exchange_strong(expected, ElementState::CONSUMED, std::memory_order_acq_rel))
         break;

      // If readPos has been consumed, we can increment m_readPos past it if another thread hasn't already done so
      // Then we can increment our own readPos value regardless so we keep detecting this case
      if(pos == readPos && expected == ElementState::CONSUMED) {
         size_t attemptIncrement = readPos;
         m_readPos.compare_exchange_strong(attemptIncrement, attemptIncrement + 1, std::memory_order_acq_rel);
         ++readPos;
      }
   }

   if(pos == writePos)
      // Ran out of positions to look at without managing to take ownership of any of them. Nothing we can return.
      return false;

   // Now we've gotten our item. We should be the ONLY ones to have this item.
   // Hold onto the buffer, but tell it we're consuming a block so it can be freed later if necessary.
   RefCounter counter(bufObtainedFrom);
   bufObtainedFrom->ConsumeBlock();

   val = std::move(element->item);
   element->item.~T();
   return true;
}
```

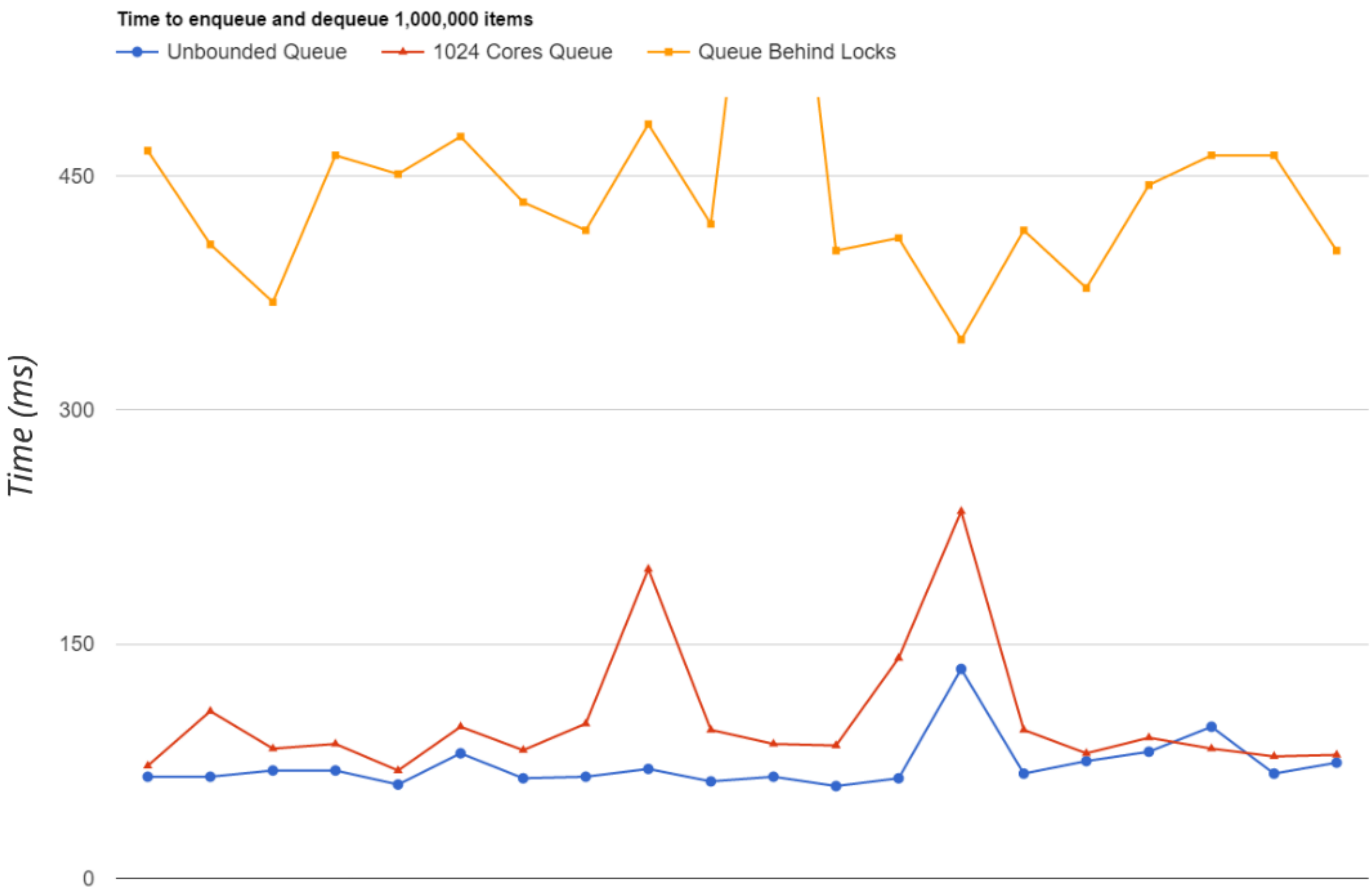
Recap: what are we looking for?

- Generic queue (templated)
- Array-based
  - Mostly contiguous
  - Few dynamic allocations
- Unbounded (capable of growth)
- EIEO (Early In/Early Out)
- Memory safe:
  - No leaks
  - No corruption
  - No dangling
- ***Bonus: 99.9+% Wait-Free Population Oblivious enqueues.***

Results

Linux Mint VM on Microsoft Surface Pro 3
Intel Core i5-4300 @ 1.9 Ghz
2 cores w/ hyperthreading

1,000,000 enqueues/dequeues
10 producer threads
10 consumer threads



Averages: Unbounded - 72.6ms; 1024cores - 102ms; Mutexed - 442.5ms

1024Cores queue available at:
1024cores.net/home/lock-free-algorithms/queues/bounded-mpmc-queue

Questions?

Jaedyn Kitt Draper
Epoch Labs ***(We're Hiring!)***
jaedyn.cppcon@jaedyn.co

Full source code available as a part of libSprawl, available at:

https://swarm.workshop.perforce.com/projects/shadauxcat-libsprawl/files/mainline/collections/ConcurrentQueue.hpp

https://github.com/ShadauxCat/Sprawl/blob/master/collections/ConcurrentQueue.hpp

Original Prezi presentation at:

https://prezi.com/jjtdti-h_sil/unbounded-array-based-lock-free-multi-producer-multi-con/