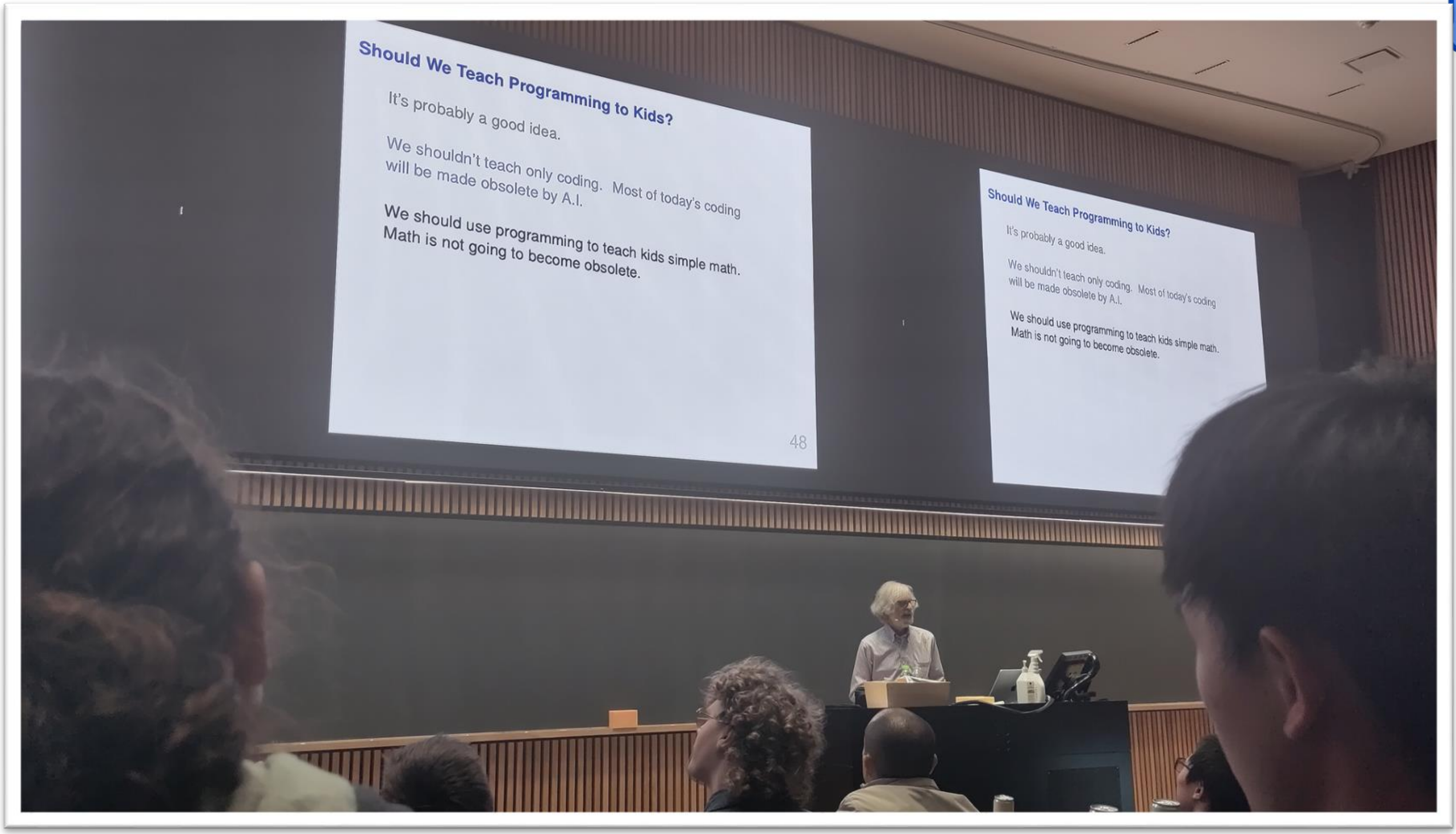
Practical Concurrent and Parallel Programming VI

Linearizability

Raúl Pardo

Based on slides of PCPP 2019

Leslie Lamport at DTU – Oct 2023



Previously on PCPP…

- Compare-And-Swap (CAS)
  - Lock-free atomic integer
  - Lock-free number range
  - Atomic libraries
  - CAS based lock implementation
- Lock-free stack
- ABA problem
- Lock-free queue

Agenda

- Revisit
  - Progress conditions
  - Lock-free queue
- Linearizability
  - Sequential consistency
  - Definition of Linearizability
  - Linearizable concurrent objects
- Work-stealing queues
- BONUS: Sequential consistency and the Java memory model

Progress conditions (revisited)

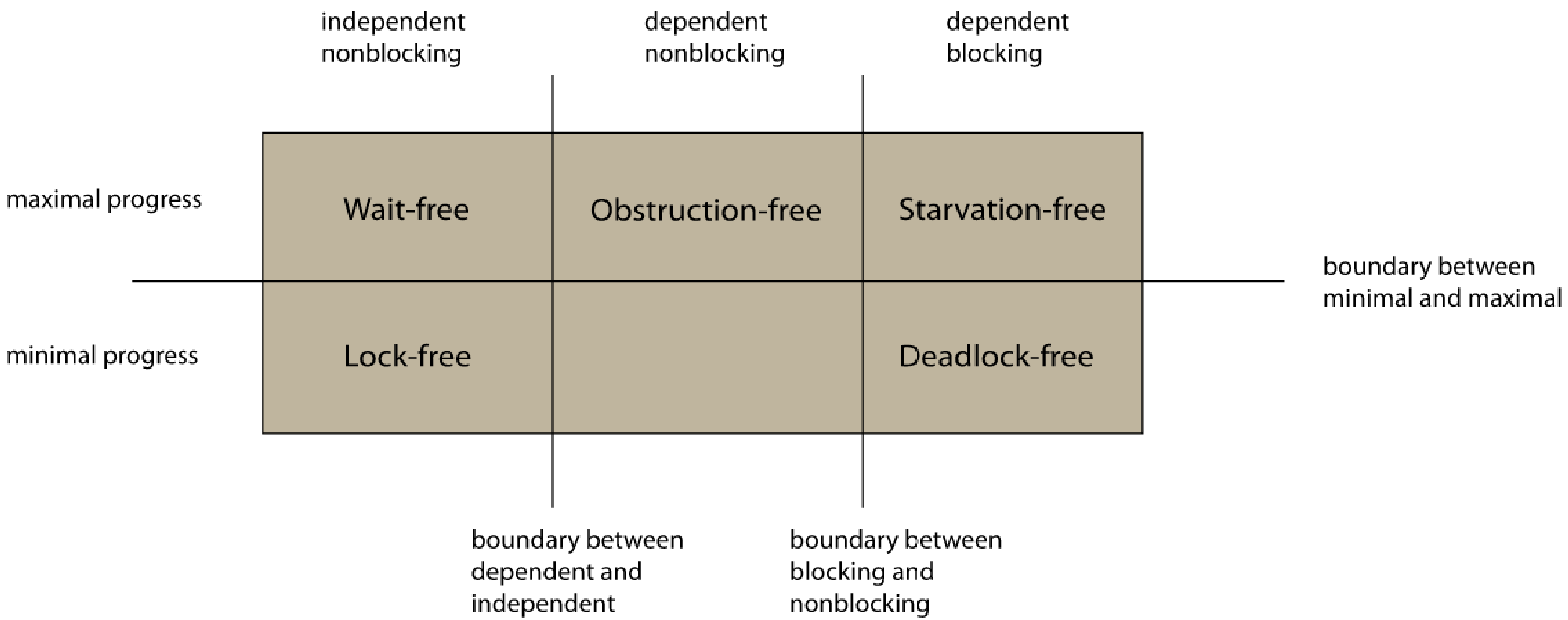


FIGURE 3.10

Progress conditions and their properties.

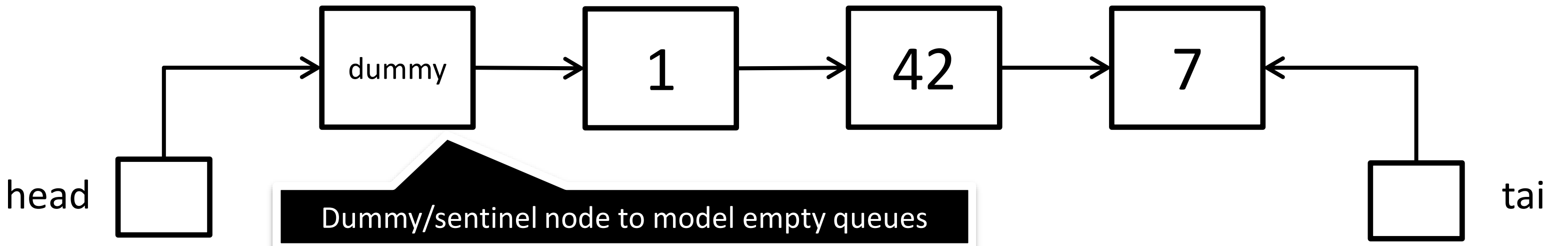
Herlihy, page 68

Lock-free data structures: Queue

Sequential Queue Specification

- A queue is a data structure following a FIFO *(first-in-first-out)* policy
  - enqueue() – Adds an element to the tail of the queue.
  - dequeue() – Removes the element at the head of the queue if the queue is not empty. Otherwise, it returns null.

- It is typically implemented as a linked list



Lock-free queue

- Michael-Scott lock free queue, introduced in 1996 (see optional readings)
- Implemented in ConcurrentLinkedQueue in java.concurrent.* by Doug Lea et. al. (see here)
  - The version on the right is not the JDK implementation

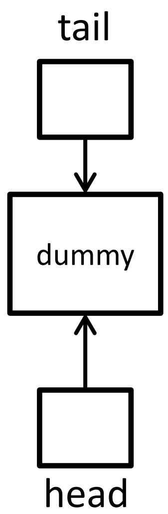
```
private static class Node<T> {
  final T item;
  final AtomicReference<Node<T>> next;

  public Node(T item, Node<T> next) {
    this.item = item;
    this.next = new AtomicReference<Node<T>>(next);
  }
}
```

```
class MSQueue<T> implements UnboundedQueue<T> {
    private final AtomicReference<Node<T>> head, tail;

    public MSQueue() {
         Node<T> dummy = new Node<T>(null, null);
         head = new AtomicReference<Node<T>>(dummy);
         tail = new AtomicReference<Node<T>>(dummy);
    }

    public void enqueue(T item) {
         Node<T> node = new Node<T>(item, null);
         while (true) {
             Node<T> last = tail.get(), next = last.next.get();
             if (last == tail.get()) {
                  if (next == null)  {
                      // In quiescent state, try inserting new node
                      if (last.next.compareAndSet(next, node)) {
                           // Insertion succeeded, try advancing tail
                           tail.compareAndSet(last, node);
                           return;
                      }
                  } else
                      // Queue in intermediate state, advance tail
                      tail.compareAndSet(last, next);
             }
         }
    }

    public T dequeue() {
         while (true) {
             Node<T> first = head.get(), last = tail.get(), next = first.next.get();
             if (first == head.get()) {
                  if (first == last) {
                      if (next == null)
                           return null;
                      else
                           tail.compareAndSet(last, next);
                  } else {
                      T result = next.item;
                      if (head.compareAndSet(first, next))
                           return result;
                  }
             }
         }
    }
…
```

See **`TestMSQueue.java`**

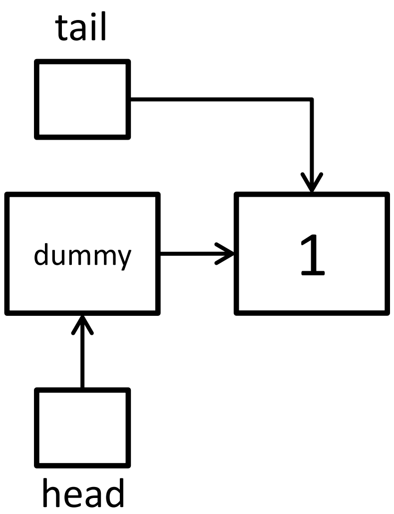
Lock-free queue | initialization


```
class MSQueue<T> implements UnboundedQueue<T> {
    private final AtomicReference<Node<T>> head, tail;

    public MSQueue() {
      Node<T> dummy = new Node<T>(null, null);
      head = new AtomicReference<Node<T>>(dummy);
      tail = new AtomicReference<Node<T>>(dummy);
    }
…
}
```



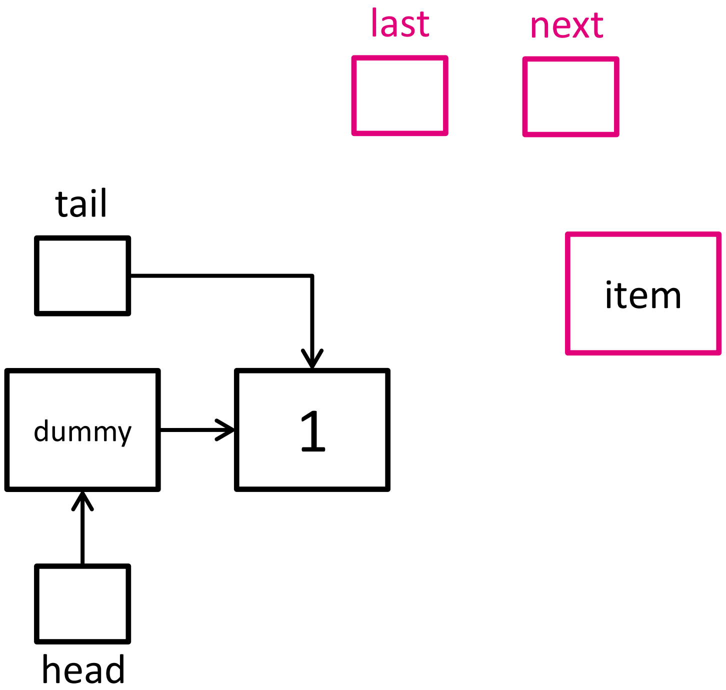
Lock-free queue | enqueue


```
class MSQueue<T> implements UnboundedQueue<T> {
…
 public void enqueue(T item) {
      Node<T> node = new Node<T>(item, null);
      while (true) {
           Node<T> last = tail.get();
           Node<T> next = last.next.get();
           if (last == tail.get()) {
             if (next == null)   {
                 // In quiescent state, try inserting new node
                 if (last.next.compareAndSet(next, node)) {
                   // Insertion succeeded, try advancing tail
                   tail.compareAndSet(last, node);
                   return;
                 }
             } else
                 // Queue in intermediate state, advance tail
                 tail.compareAndSet(last, next);
           }
      }
    }
…
}
```



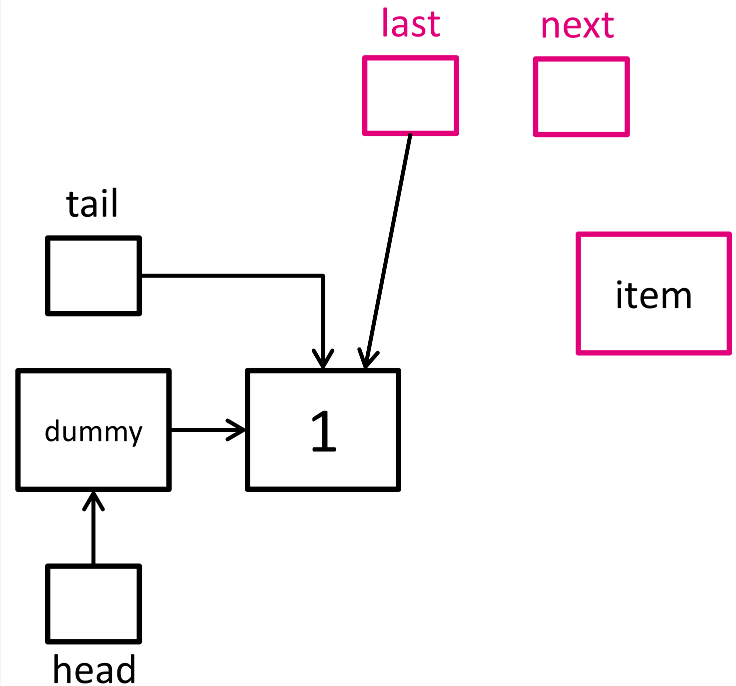
Lock-free queue | enqueue


```
class MSQueue<T> implements UnboundedQueue<T> {
…
 public void enqueue(T item) {
      Node<T> node = new Node<T>(item, null);
      while (true) {
          Node<T> last = tail.get();
          Node<T> next = last.next.get();
          if (last == tail.get()) {
            if (next == null)   {
                // In quiescent state, try inserting new node
                if (last.next.compareAndSet(next, node)) {
                  // Insertion succeeded, try advancing tail
                  tail.compareAndSet(last, node);
                  return;
                }
            } else
                // Queue in intermediate state, advance tail
                tail.compareAndSet(last, next);
          }
      }
    }
…
}
```



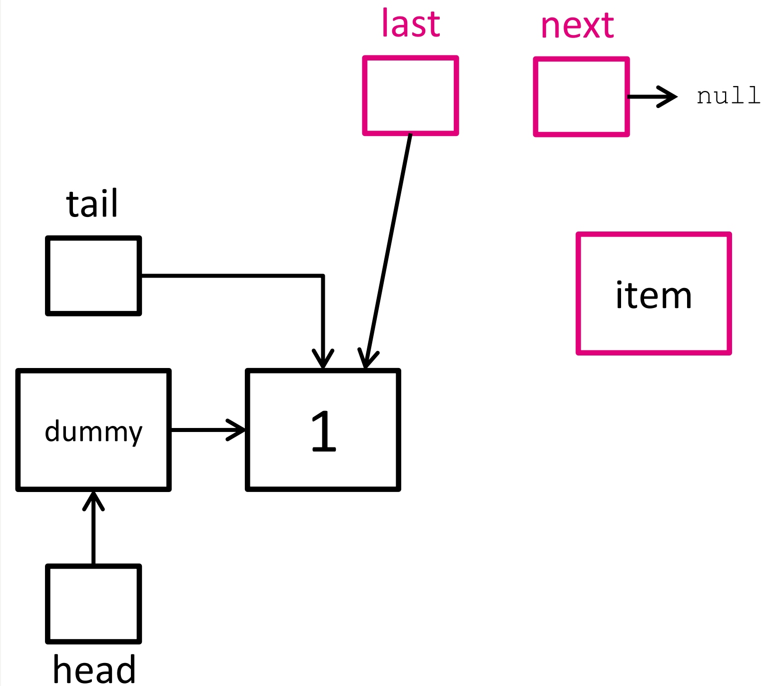
Lock-free queue | enqueue


```
class MSQueue<T> implements UnboundedQueue<T> {
…
 public void enqueue(T item) {
      Node<T> node = new Node<T>(item, null);
      while (true) {
          Node<T> last = tail.get();
          Node<T> next = last.next.get();
          if (last == tail.get()) {
            if (next == null)  {
                // In quiescent state, try inserting new node
                if (last.next.compareAndSet(next, node)) {
                  // Insertion succeeded, try advancing tail
                  tail.compareAndSet(last, node);
                  return;
                }
            } else
                // Queue in intermediate state, advance tail
                tail.compareAndSet(last, next);
          }
      }
    }
…
}
```



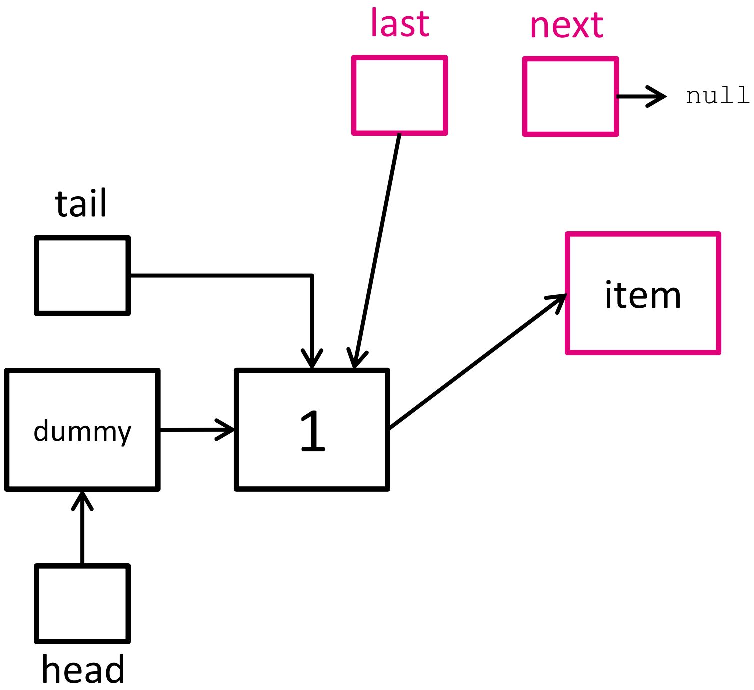
Lock-free queue | enqueue


```
class MSQueue<T> implements UnboundedQueue<T> {
…
 public void enqueue(T item) {
      Node<T> node = new Node<T>(item, null);
      while (true) {
           Node<T> last = tail.get();
           Node<T> next = last.next.get();
           if (last == tail.get()) {
             if (next == null)  {
                 // In quiescent state, try inserting new node
                 if (last.next.compareAndSet(next, node)) {
                   // Insertion succeeded, try advancing tail
                   tail.compareAndSet(last, node);
                   return;
                 }
             } else
                 // Queue in intermediate state, advance tail
                 tail.compareAndSet(last, next);
           }
      }
    }
…
}
```



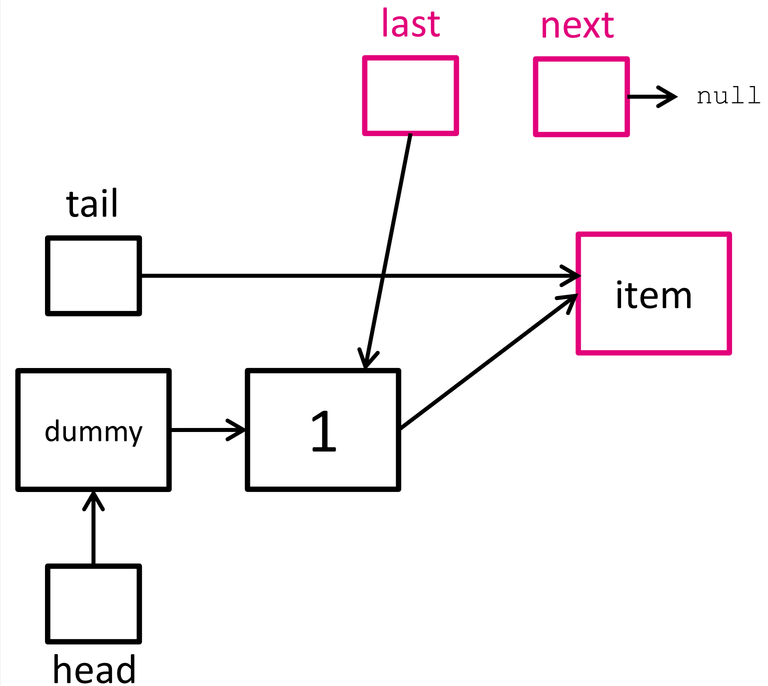
Lock-free queue | enqueue


```
class MSQueue<T> implements UnboundedQueue<T> {
…
 public void enqueue(T item) {
      Node<T> node = new Node<T>(item, null);
      while (true) {
           Node<T> last = tail.get();
           Node<T> next = last.next.get();
           if (last == tail.get()) {
             if (next == null)   {
                 // In quiescent state, try inserting new node
                 if (last.next.compareAndSet(next, node)) {
                   // Insertion succeeded, try advancing tail
                   tail.compareAndSet(last, node);
                   return;
                 }
             } else
                 // Queue in intermediate state, advance tail
                 tail.compareAndSet(last, next);
           }
      }
    }
…
}
```



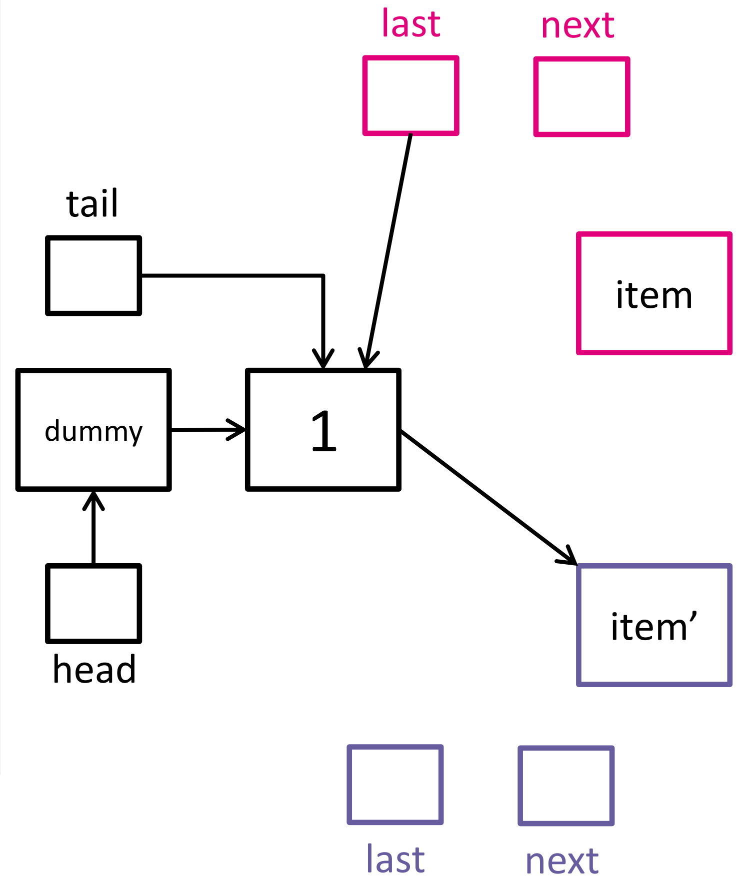
Lock-free queue | enqueue


```
class MSQueue<T> implements UnboundedQueue<T> {
…
 public void enqueue(T item) {
      Node<T> node = new Node<T>(item, null);
      while (true) {
           Node<T> last = tail.get();
           Node<T> next = last.next.get();
           if (last == tail.get()) {
             if (next == null)   {
                 // In quiescent state, try inserting new node
                 if (last.next.compareAndSet(next, node)) {
                   // Insertion succeeded, try advancing tail
                   tail.compareAndSet(last, node);
                   return;
                 }
             } else
                 // Queue in intermediate state, advance tail
                 tail.compareAndSet(last, next);
           }
      }
    }
…
}
```



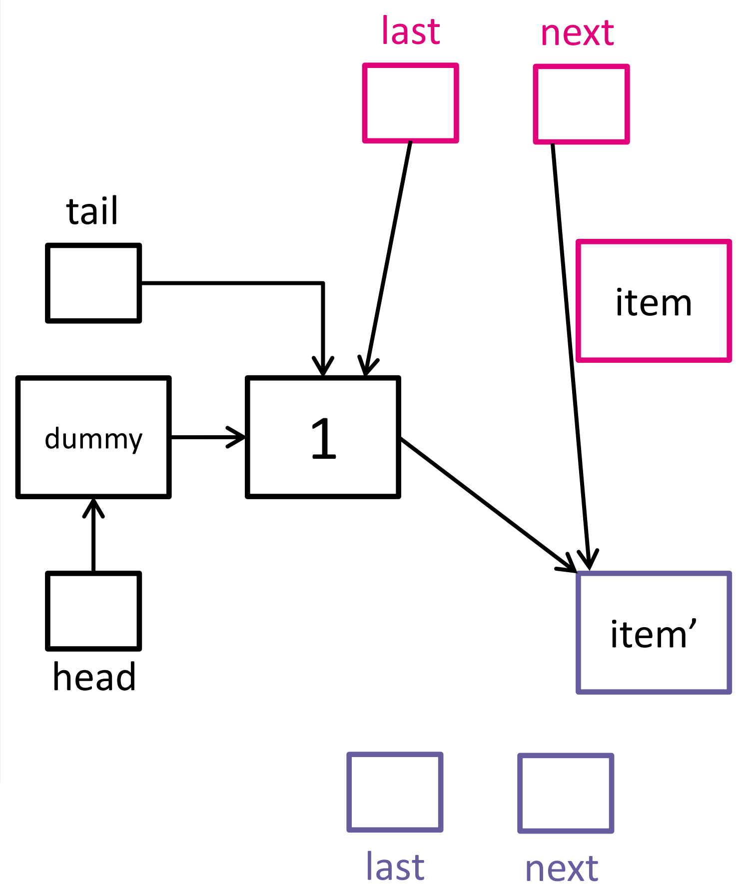
Lock-free queue | enqueue


```
class MSQueue<T> implements UnboundedQueue<T> {
…
 public void enqueue(T item) {
      Node<T> node = new Node<T>(item, null);
      while (true) {
           Node<T> last = tail.get();
           Node<T> next = last.next.get();
           if (last == tail.get()) {
             if (next == null)   {
                  // In quiescent state, try inserting new node
                  if (last.next.compareAndSet(next, node)) {
                    // Insertion succeeded, try advancing tail
                    tail.compareAndSet(last, node);
                    return;
                  }
             } else
                  // Queue in intermediate state, advance tail
                  tail.compareAndSet(last, next);
           }
      }
    }
…
}
```



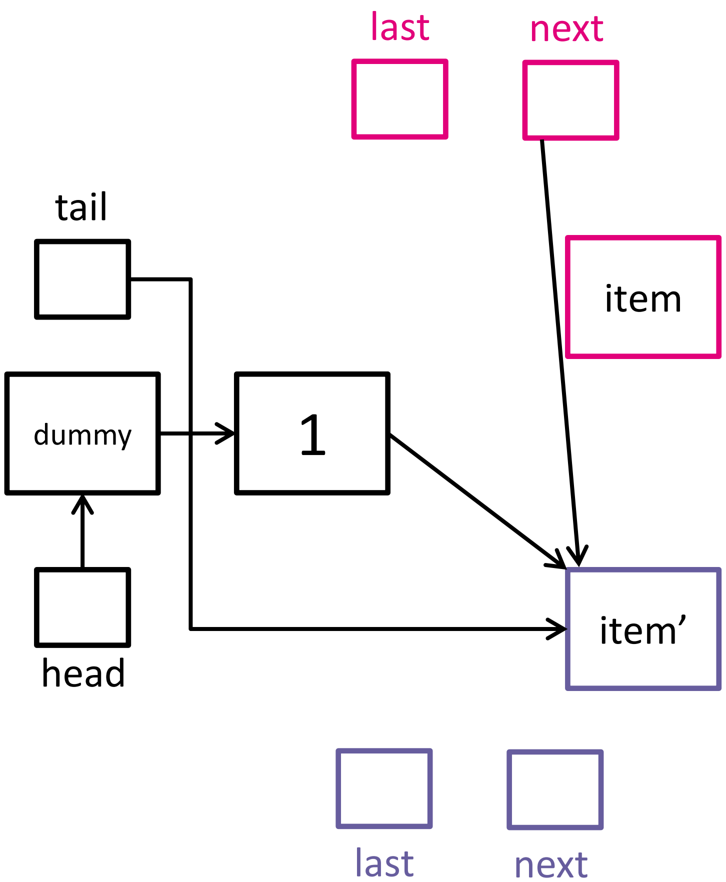
Lock-free queue | enqueue


```
class MSQueue<T> implements UnboundedQueue<T> {
…
 public void enqueue(T item) {
      Node<T> node = new Node<T>(item, null);
      while (true) {
          Node<T> last = tail.get();
          Node<T> next = last.next.get();
          if (last == tail.get()) {
            if (next == null)   {
                // In quiescent state, try inserting new node
                if (last.next.compareAndSet(next, node)) {
                  // Insertion succeeded, try advancing tail
                  tail.compareAndSet(last, node);
                  return;
                }
            } else
                // Queue in intermediate state, advance tail
                tail.compareAndSet(last, next);
          }
      }
    }
…
}
```



Lock-free queue | enqueue

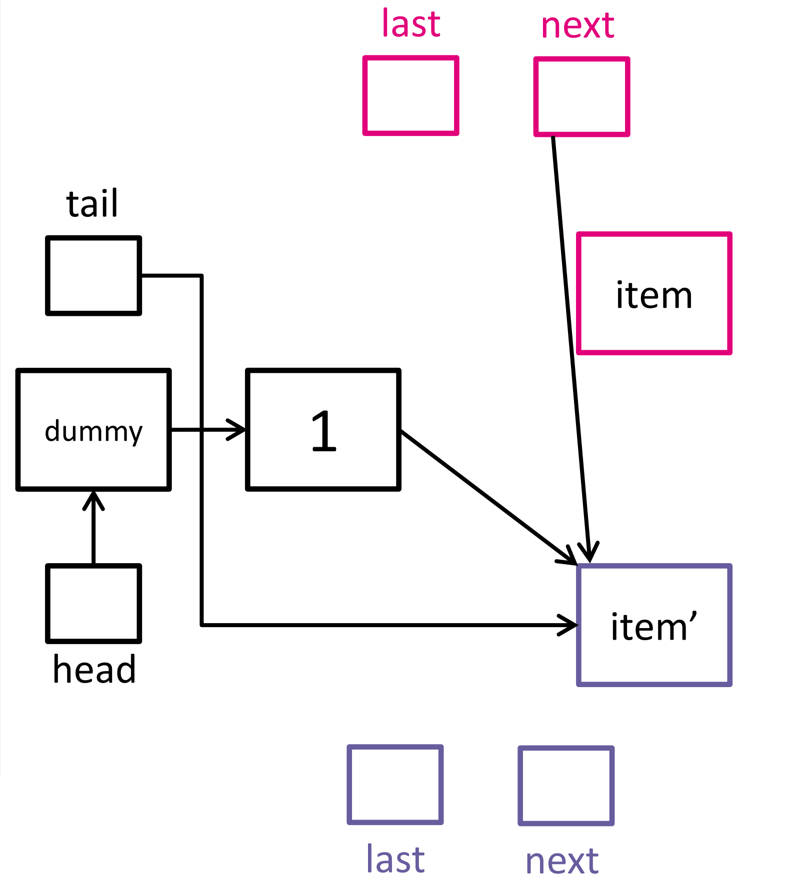

```
class MSQueue<T> implements UnboundedQueue<T> {
…
 public void enqueue(T item) {
      Node<T> node = new Node<T>(item, null);
      while (true) {
          Node<T> last = tail.get();
          Node<T> next = last.next.get();
          if (last == tail.get()) {
            if (next == null)   {
                // In quiescent state, try inserting new node
                if (last.next.compareAndSet(next, node)) {
                  // Insertion succeeded, try advancing tail
                  tail.compareAndSet(last, node);
                  return;
                }
            } else
                // Queue in intermediate state, advance tail
                tail.compareAndSet(last, next);
          }
      }
    }
…
}
```



Lock-free queue | enqueue


```
class MSQueue<T> implements UnboundedQueue<T> {
…
 public void enqueue(T item) {
      Node<T> node = new Node<T>(item, null);
      while (true) {
          Node<T> last = tail.get();
          Node<T> next = last.next.get();
          if (last == tail.get()) {
            if (next == null)   {
                // In quiescent state, try inserting new node
                if (last.next.compareAndSet(next, node)) {
                  // Insertion succeeded, try advancing tail
                  tail.compareAndSet(last, node);
                  return;
                }
            } else
                // Queue in intermediate state, advance tail
                tail.compareAndSet(last, next);
          }
      }
    }
…
}
```

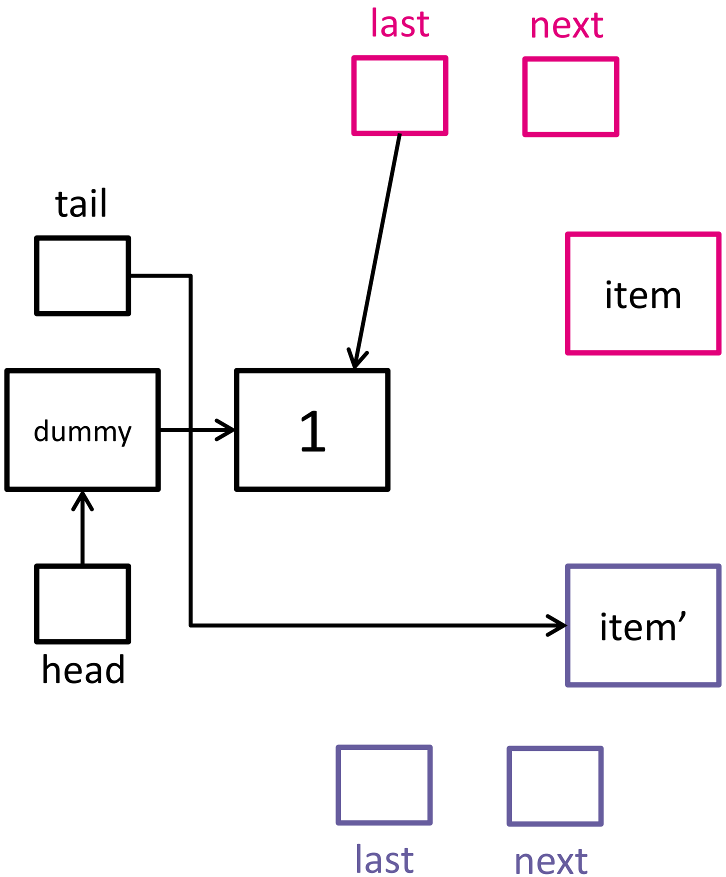
In case another thread is enqueuing, and didn't update the tail, the current thread helps by advancing the tail



Lock-free queue | enqueue

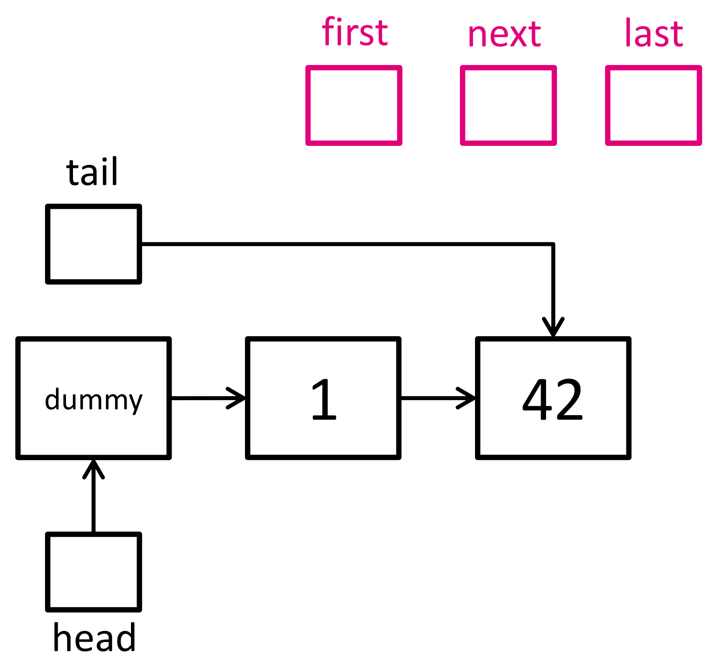

```
class MSQueue<T> implements UnboundedQueue<T> {
…
 public void enqueue(T item) {
      Node<T> node = new Node<T>(item, null);
      while (true) {
          Node<T> last = tail.get();
          Node<T> next = last.next.get();
          if (last == tail.get()) {
            if (next == null)  {
                // In quiescent state, try inserting new node
                if (last.next.compareAndSet(next, node)) {
                  // Insertion succeeded, try advancing tail
                  tail.compareAndSet(last, node);
                  return;
                }
            } else
                // Queue in intermediate state, advance tail
                tail.compareAndSet(last, next);
          }
      }
    }
…
}
```

If the tail has been changed, then the thread restarts right away



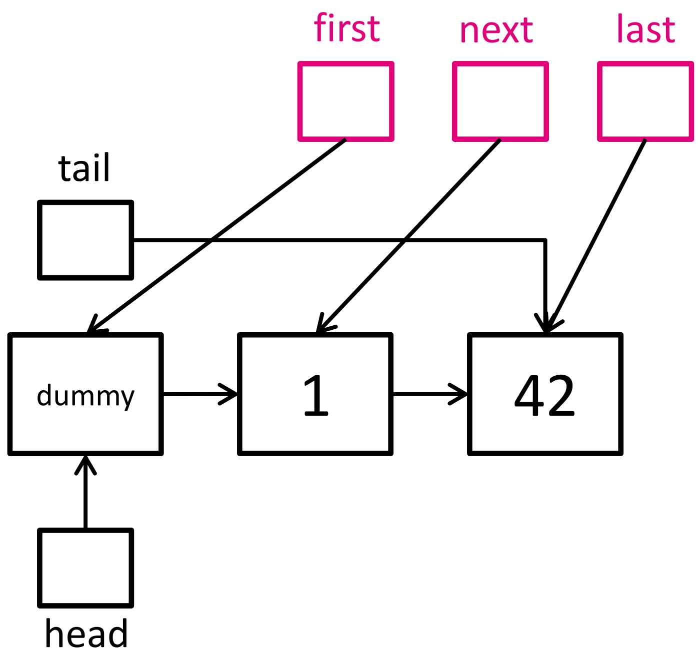
Lock-free queue | dequeue

```
class MSQueue<T> implements UnboundedQueue<T> {
…

    public T dequeue() {
      while (true) {
          Node<T> first = head.get();
          Node<T> last = tail.get();
         Node<T> next = first.next.get();
          if (first == head.get()) {
           if (first == last) {
               if (next == null)
                return null;
               else
                tail.compareAndSet(last, next);
           } else {
               T result = next.item;
               if (head.compareAndSet(first, next))
                return result;
           }
          }
      }
    }
…
}
```



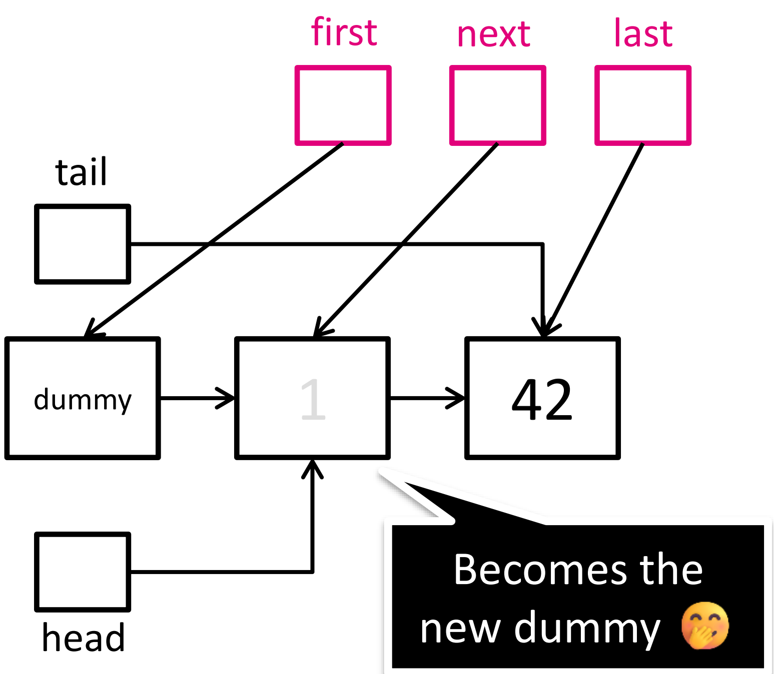
Lock-free queue | dequeue


```
class MSQueue<T> implements UnboundedQueue<T> {
…

    public T dequeue() {
      while (true) {
          Node<T> first = head.get();
          Node<T> last = tail.get();
         Node<T> next = first.next.get();
          if (first == head.get()) {
           if (first == last) {
              if (next == null)
                return null;
              else
                tail.compareAndSet(last, next);
           } else {
              T result = next.item;
              if (head.compareAndSet(first, next))
                return result;
           }
          }
      }
    }
…
}
```



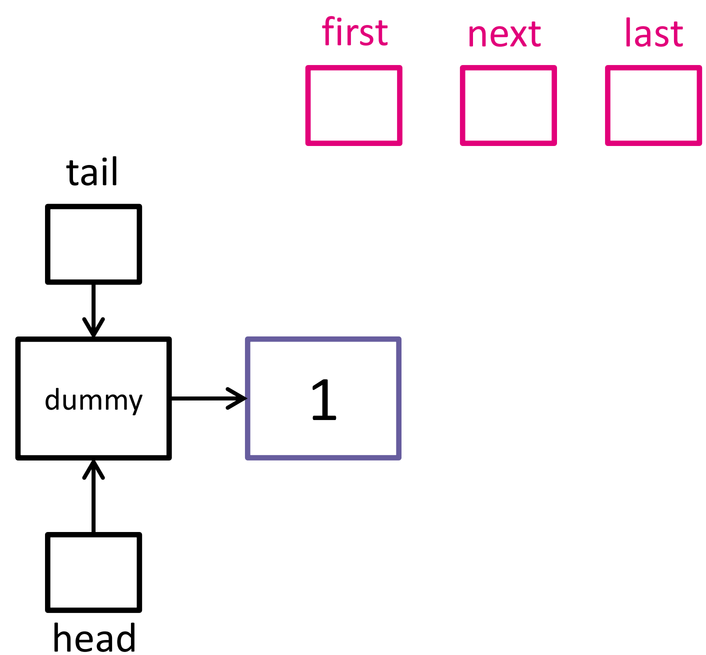
Lock-free queue | dequeue


```
class MSQueue<T> implements UnboundedQueue<T> {
…

    public T dequeue() {
      while (true) {
          Node<T> first = head.get();
          Node<T> last = tail.get();
          Node<T> next = first.next.get();
          if (first == head.get()) {
            if (first == last) {
                if (next == null)
                  return null;
                else
                  tail.compareAndSet(last, next);
            } else {
                T result = next.item;
                if (head.compareAndSet(first, next))
                  return result;
            }
          }
      }
    }
…
}
```



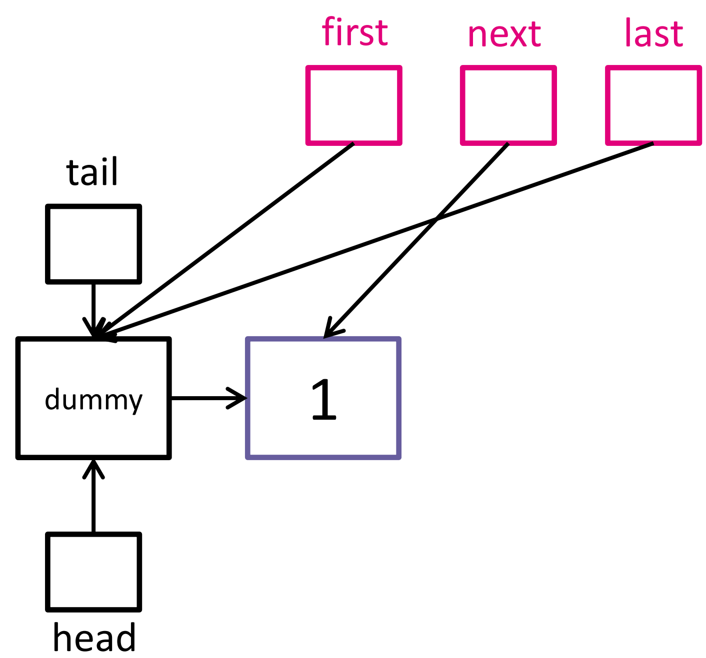
Lock-free queue | dequeue

```
class MSQueue<T> implements UnboundedQueue<T> {
…

    public T dequeue() {
      while (true) {
          Node<T> first = head.get();
          Node<T> last = tail.get();
         Node<T> next = first.next.get();
          if (first == head.get()) {
            if (first == last) {
                if (next == null)
                  return null;
                else
                  tail.compareAndSet(last, next);
            } else {
                T result = next.item;
                if (head.compareAndSet(first, next))
                  return result;
            }
          }
      }
    }
…
}
```



Lock-free queue | dequeue

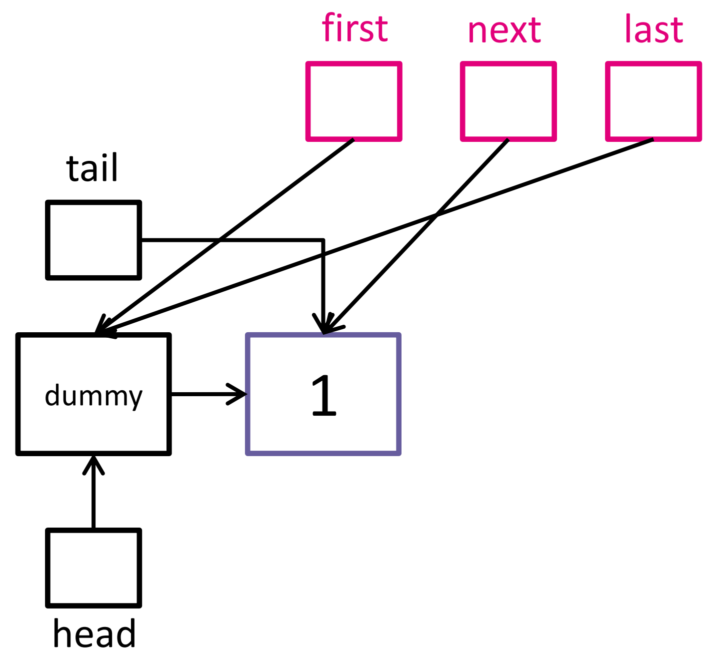

```
class MSQueue<T> implements UnboundedQueue<T> {
…

    public T dequeue() {
      while (true) {
          Node<T> first = head.get();
          Node<T> last = tail.get();
         Node<T> next = first.next.get();
          if (first == head.get()) {
           if (first == last) {
               if (next == null)
                return null;
               else
                tail.compareAndSet(last, next);
           } else {
               T result = next.item;
               if (head.compareAndSet(first, next))
                return result;
           }
          }
      }
    }
…
}
```



Lock-free queue | dequeue

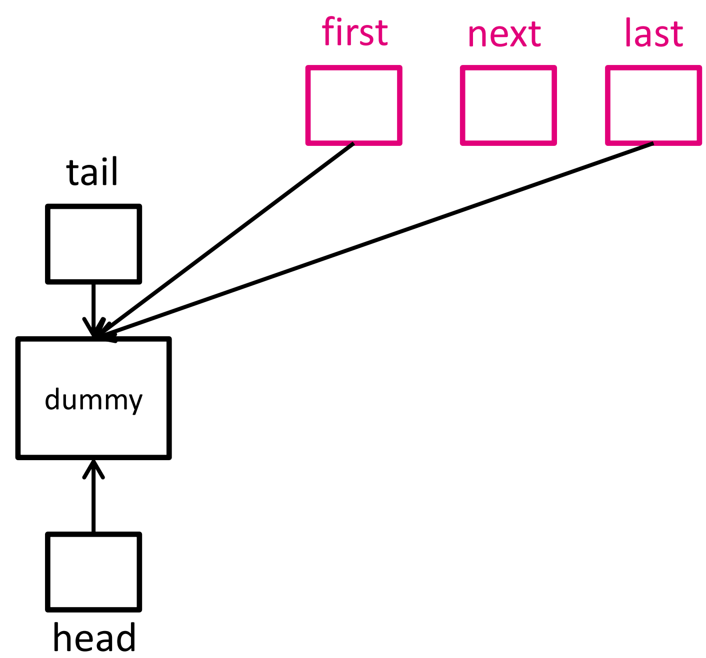
```
class MSQueue<T> implements UnboundedQueue<T> {
…

    public T dequeue() {
      while (true) {
          Node<T> first = head.get();
          Node<T> last = tail.get();
          Node<T> next = first.next.get();
          if (first == head.get()) {
            if (first == last) {
                if (next == null)
                  return null;
                else
                  tail.compareAndSet(last, next);
            } else {
                T result = next.item;
                if (head.compareAndSet(first, next))
                  return result;
            }
          }
      }
    }
…
}
```

If the next field of the head is not null (because another thread push in the meantime), then the calling thread helps advancing the tail and tries to dequeue again.



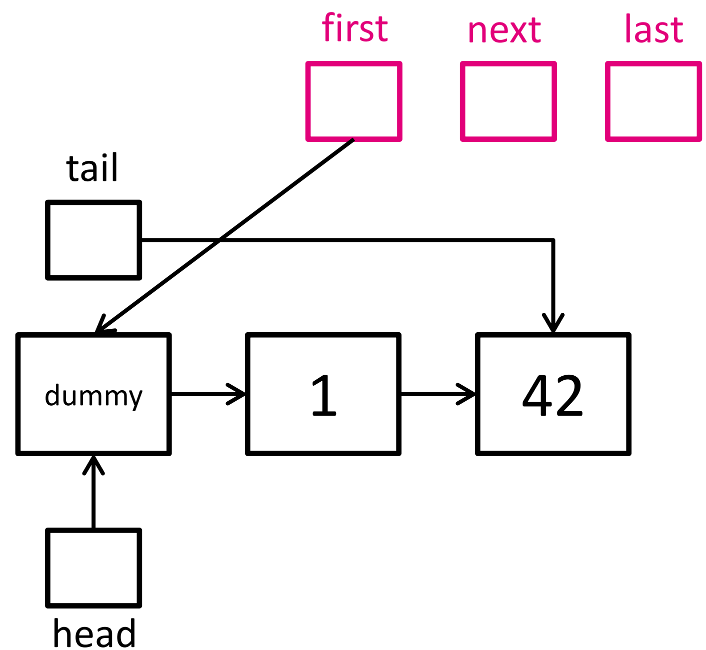
Lock-free queue | dequeue

```
class MSQueue<T> implements UnboundedQueue<T> {
…

    public T dequeue() {
      while (true) {
          Node<T> first = head.get();
          Node<T> last = tail.get();
         Node<T> next = first.next.get();
          if (first == head.get()) {
            if (first == last) {
                if (next == null)
                  return null;
                else
                  tail.compareAndSet(last, next);
            } else {
                T result = next.item;
                if (head.compareAndSet(first, next))
                  return result;
            }
          }
      }
    }
…
}
```



Lock-free queue | dequeue


```
class MSQueue<T> implements UnboundedQueue<T> {
…

    public T dequeue() {
      while (true) {
          Node<T> first = head.get();
          Node<T> last = tail.get();
         Node<T> next = first.next.get();
          if (first == head.get()) {
           if (first == last) {
               if (next == null)
                return null;
               else
                tail.compareAndSet(last, next);
           } else {
               T result = next.item;
               if (head.compareAndSet(first, next))
                return result;
           }
          }
      }
    }
…
}
```



What about correctness?

- We have seen several implementations of lock-free data structures today and last week

- However, we have not seen any techniques to reason about their correctness…
  - How do we even write the *concurrent* specification of the lock-free queue we have seen?

Linearizability

Linearizability – Intro

- Defining correctness of (non-blocking) concurrent objects is tricky
  - Recall correctness is defined by a specification

- ***The motivation behind linearizability is to use specifications of sequential objects as a basis for the correctness of concurrent objects***
  - Specifications of sequential objects are typically expressed as pre- and post-conditions for method calls

Wait Free Queue


```
class WaitFreeQueue<T> {
  int head = 0, tail = 0;
  T[] items;

  public WaitFreeQueue(int capacity) {
    items = (T[]) new Object[capacity]
  }

  public void enq(T x) throws Exception {
    if (tail - head == items.length)
      thrown new Exception();
    items[tail % items.length] = x;
    tail++;
  }

  public void deq() throws Exception {
    if (tail - head == 0)
      thrown new Exception();
    T x  = items[head % items.length];
    head++;
    return x;
  }
}
```

Herlihy p. 52

- Informally, we would like concurrent operations to happen as if they were executed sequentially

- The specification of a concurrent queue is the sequential behaviour of queues

Wait Free Queue


```
class WaitFreeQueue<T> {
  int head = 0, tail = 0;
  T[] items;

  public WaitFreeQueue(int capacity) {
    items = (T[]) new Object[capacity]
  }

  public void enq(T x) throws Exception {
    if (tail - head == items.length)
      thrown new Exception();
    items[tail % items.length] = x;
    tail++;
  }

  public void deq() throws Exception {
    if (tail - head == 0)
      thrown new Exception();
    T x  = items[head % items.length];
    head++;
    return x;
  }
}
```

This is pseudo-code for illustration purposes. Assume all variables have volatile semantics and each program line is atomic.

Are there any problems with this implementation?

Herlihy p. 52

Sequential consistency

- Recall that, *<u>in sequential programs</u>*, the compiler and CPU are allowed to re-order instructions as long as they produce a valid result (according to the specification of the object)

```
q.enq(x)
q.deq(x)
p.enq(y)
p.deq(y)
```

- When executed sequentially, these are possible orderings for execution:

--- q.enq(x) - p.enq(y) - q.deq(x) - p.deq(y) --->

--- p.enq(y) - q.enq(x) - p.deq(y) - q.deq(x) --->

--- q.enq(x) - p.deq(y) - q.deq(x) - p.enq(y) --->

...

Sequential consistency

- Recall that, *in sequential programs*, the compiler and CPU are allowed to re-order instructions as long as they produce a valid result (according to the specification of the object)

```
q.enq(x)
q.deq(x)
p.enq(y)
p.deq(y)
```

- When executed sequentially, these are possible orderings for execution:

--- q.enq(x) - p.enq(y) - q.deq(x) - p.deq(y) --->

--- p.enq(y) - q.enq(x) - p.deq(y) - q.deq(x) --->

--- q.enq(x) - p.deq(y) - q.deq(x) - p.enq(y) --->

...

This syntax means: The pink thread enqueues x on queue q successfully

This syntax means: The pink thread dequeues x from queue q successfully

Sequential consistency

- Recall that, *in sequential programs*, the compiler and CPU are allowed to re-order instructions as long as they produce a valid result (according to the specification of the object)

```
q.enq(x)
q.deq(x)
p.enq(y)
p.deq(y)
```

- When executed sequentially, these are possible orderings for execution:

`q.enq(x)` - `p.enq(y)` - `q.deq(x)` - `p.deq(y)`

`p.enq(y)` - `q.enq(x)` - `p.deq(y)` - `q.deq(x)`

`q.enq(x)` - `p.deq(y)` - `q.deq(x)` - `p.enq(y)`

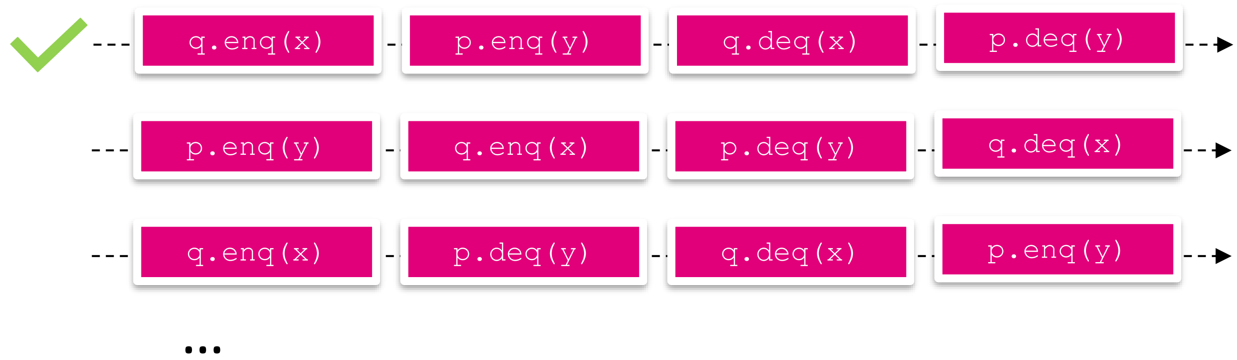
…

Sequential consistency

- Recall that, <u>*in sequential programs*</u>, the compiler and CPU are allowed to re-order instructions as long as they produce a valid result (according to the specification of the object)

```
q.enq(x)
q.deq(x)
p.enq(y)
p.deq(y)
```

- When executed sequentially, these are possible orderings for execution:



Sequential consistency

- Recall that, *<u>in sequential programs</u>*, the compiler and CPU are allowed to re-order instructions as long as they produce a valid result (according to the specification of the object)

```
q.enq(x)
q.deq(x)
p.enq(y)
p.deq(y)
```

- When executed sequentially, these are possible orderings for execution:

q.enq(x) → p.enq(y) → q.deq(x) → p.deq(y)

p.enq(y) → q.enq(x) → p.deq(y) → q.deq(x)

q.enq(x) → p.deq(y) → q.deq(x) → p.enq(y)

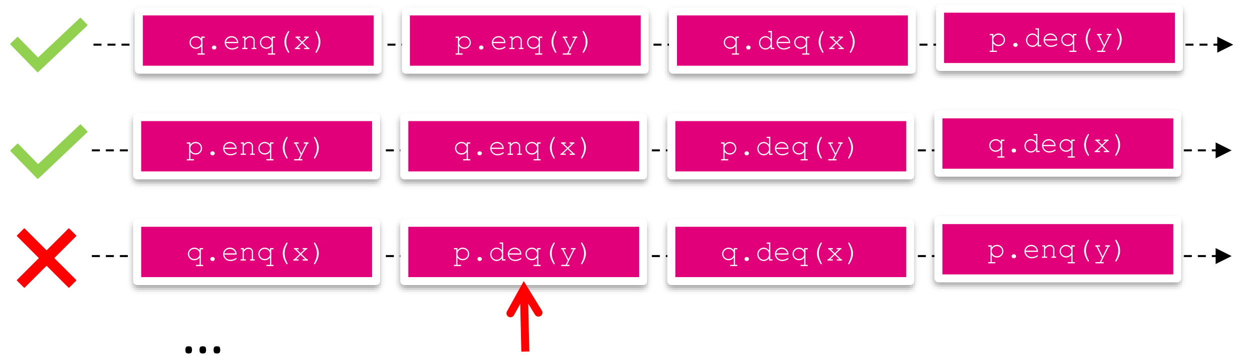
…

Sequential consistency

- Recall that, *in sequential programs*, the compiler and CPU are allowed to re-order instructions as long as they produce a valid result (according to the specification of the object)

```
q.enq(x)
q.deq(x)
p.enq(y)
p.deq(y)
```

- When executed sequentially, these are possible orderings for execution:



Sequential consistency

- For concurrent executions, we must define the conditions asserting that every thread is behaving consistently w.r.t. a sequential execution

- For executions of concurrent objects, an execution is ***sequential consistency*** iff
  1. Method calls appear to happen in a one-at-a-time, sequential order
  2. Method calls should appear to take effect in program order

Sequential consistency

- For concurrent executions, we must define the conditions asserting that every thread is behaving consistently w.r.t. a sequential execution

- For executions of concurrent objects, an execution is *<u>sequential consistency</u>* iff
  1. Method calls appear to happen in a one-at-a-time, sequential order

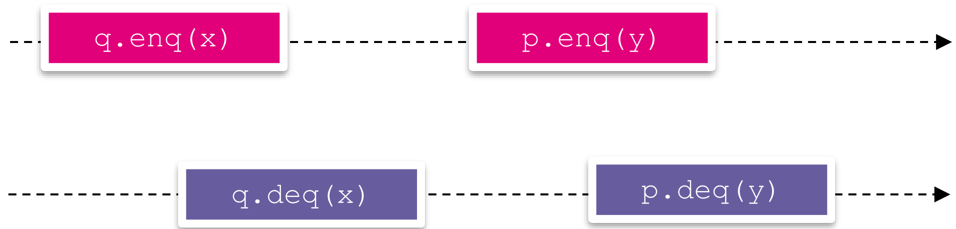
     Requires memory commands to be executed in order
  2. Method calls should appear to take effect in program order

Sequential consistency

- For concurrent executions, we must define the conditions asserting that every thread is behaving consistently w.r.t. a sequential execution

- For executions of concurrent objects, an execution is *sequential consistency* iff
  1. Method calls appear to happen in a one-at-a-time, sequential order
     Requires memory commands to be executed in order
  2. Method calls should appear to take effect in program order
     Requires sequential program order for each thread

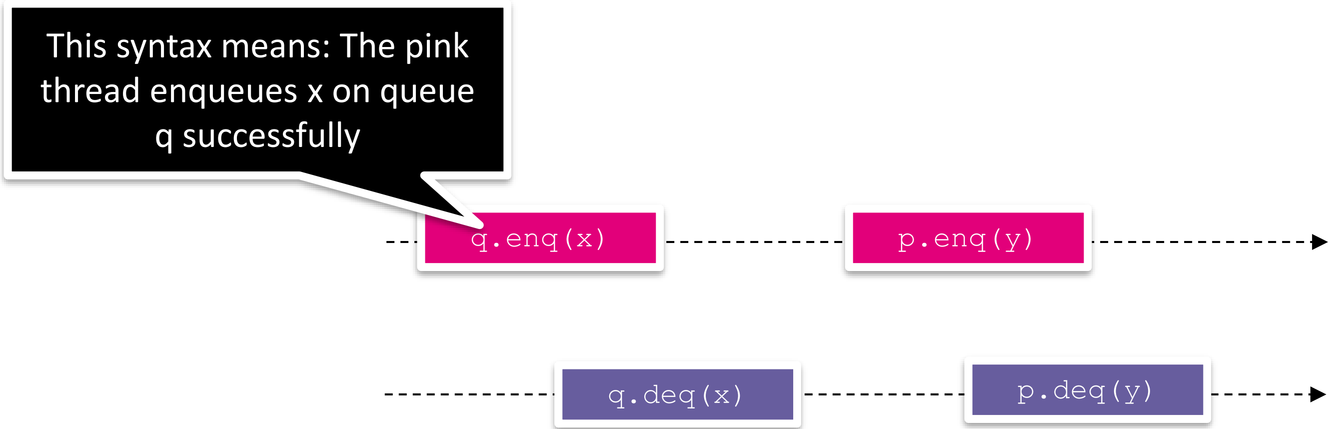
Sequential consistency



A concurrent execution is sequentially consistent iff there exists *a* re-ordering of operations producing a sequential execution where:

1. Operations happen one-at-a-time
2. Program order is preserved (for each thread)
3. The execution satisfies the specification of the object

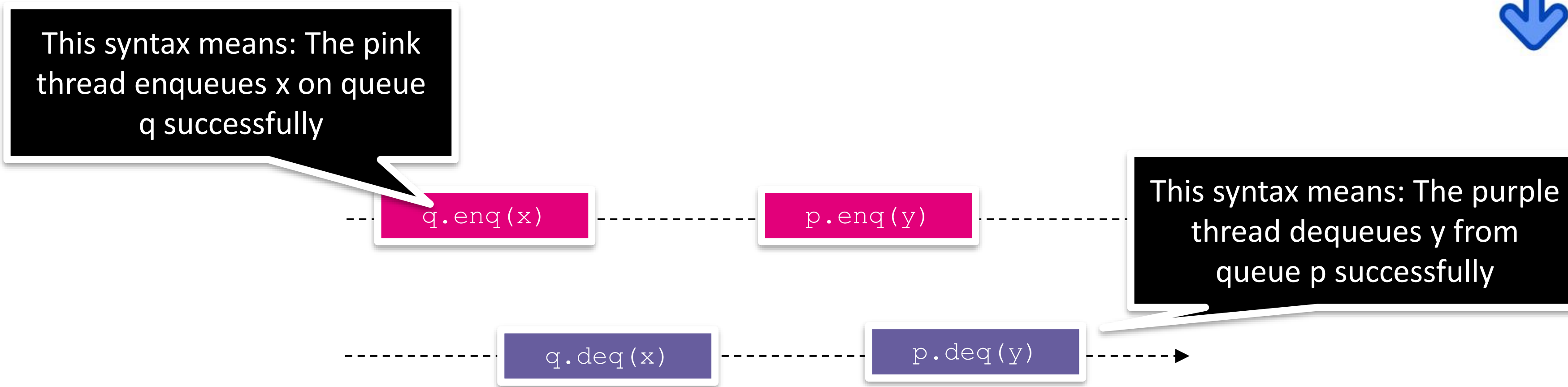
Sequential consistency



A concurrent execution is sequentially consistent iff there exists *a* re-ordering of operations producing a sequential execution where:

1. Operations happen one-at-a-time
2. Program order is preserved (for each thread)
3. The execution satisfies the specification of the object

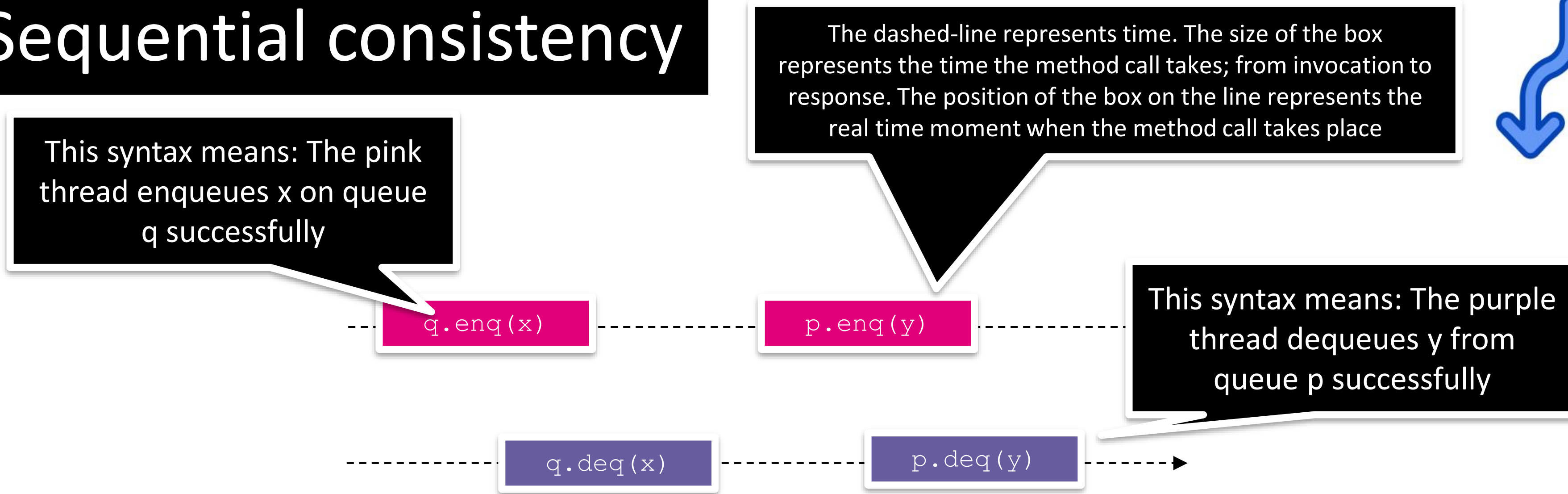
Sequential consistency



A concurrent execution is sequentially consistent iff there exists *a* re-ordering of operations producing a sequential execution where:

1. Operations happen one-at-a-time
2. Program order is preserved (for each thread)
3. The execution satisfies the specification of the object

Sequential consistency



A concurrent execution is sequentially consistent iff there exists *a* re-ordering of operations producing a sequential execution where:

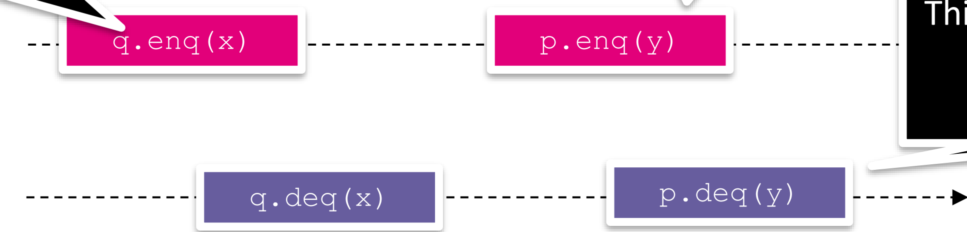
1. Operations happen one-at-a-time
2. Program order is preserved (for each thread)
3. The execution satisfies the specification of the object

Sequential consistency

The dashed-line represents time. The size of the box represents the time the method call takes; from invocation to response. The position of the box on the line represents the real time moment when the method call takes place

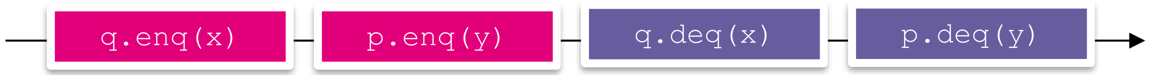
This syntax means: The pink thread enqueues x on queue q successfully

This syntax means: The purple thread dequeues y from queue p successfully



A concurrent execution is sequentially consistent iff there exists *a* re-ordering of operations producing a sequential execution where:

1. Operations happen one-at-a-time
2. Program order is preserved (for each thread)
3. The execution satisfies the specification of the object

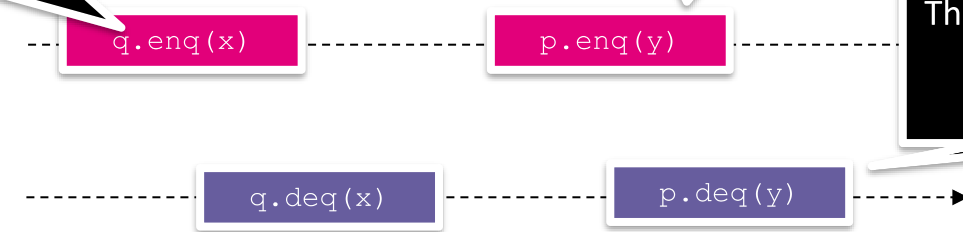


Sequential consistency

This syntax means: The pink thread enqueues x on queue q successfully

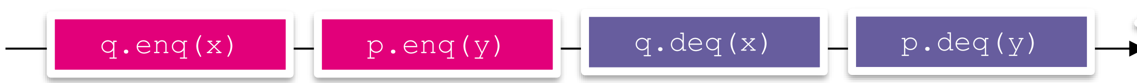
The dashed-line represents time. The size of the box represents the time the method call takes; from invocation to response. The position of the box on the line represents the real time moment when the method call takes place

This syntax means: The purple thread dequeues y from queue p successfully



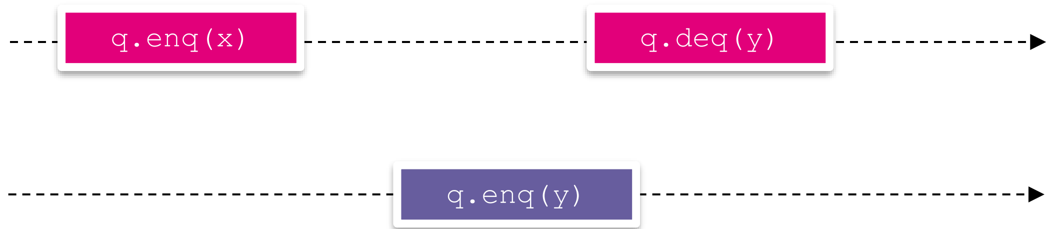
A concurrent execution is sequentially consistent iff there exists *a* re-ordering of operations producing a sequential execution where:

1. Operations happen one-at-a-time
2. Program order is preserved (for each thread)
3. The execution satisfies the specification of the object



These are instantaneous executions, and the line only represents execution order (not real time)

Sequential consistency



Is this concurrent execution sequentially consistent?
(Recall we are working with a FIFO queue)

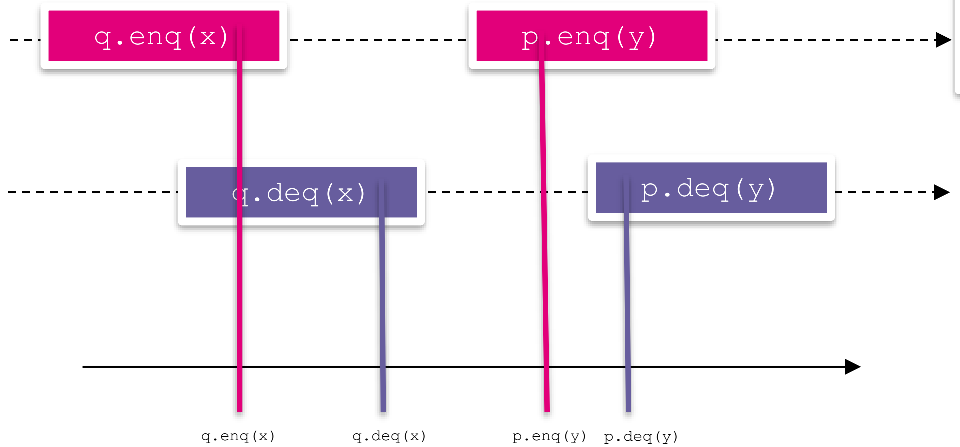
Linearizability

- Linearizability extends sequential consistency by requiring that the real time order of the execution is preserved

- Linearizability extends sequential consistency with the following condition:
    1. Each method call should appear to take effect instantaneously at some moment between its invocation and response

Linearizability

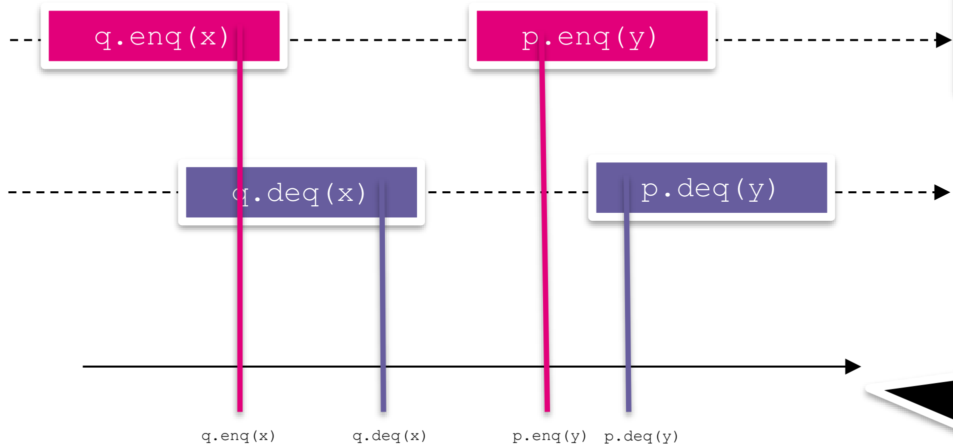
- ***To show that a concurrent execution is linearizable, we must define linearization points within each method call, which map to sequential execution that satisfy the specification of the object***
  - A *linearization point* defines the instant in the execution when the method call "takes effect"



Example linearization of this execution

Linearizability

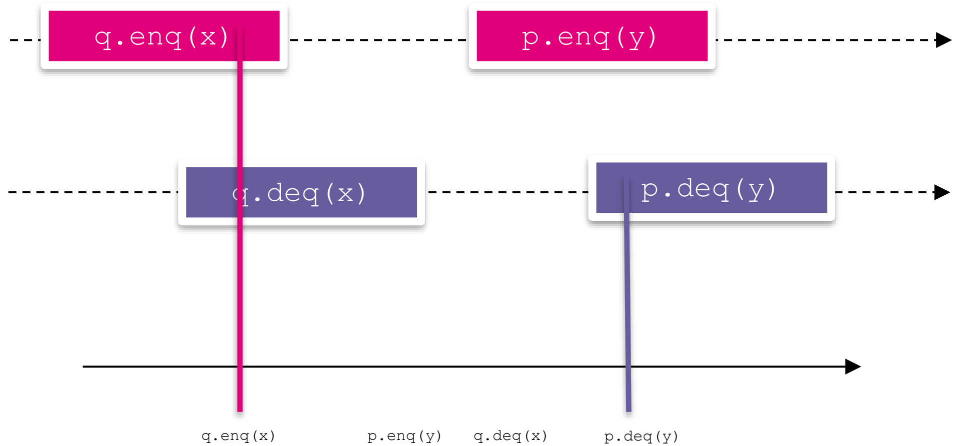
- ***To show that a concurrent execution is linearizable, we must define linearization points within each method call, which map to sequential execution that satisfy the specification of the object***
  - A *linearization point* defines the instant in the execution when the method call "takes effect"



Example linearization of this execution

Same as before, these are instantaneous executions, and the line only represents execution order. Just different syntax for convenience.

- To show that a concurrent execution is linearizable, we must define linearization points within each method call, which map to sequential execution that satisfy the specification of the object
  - A *linearization point* defines the instant in the execution when the method call “takes effect”

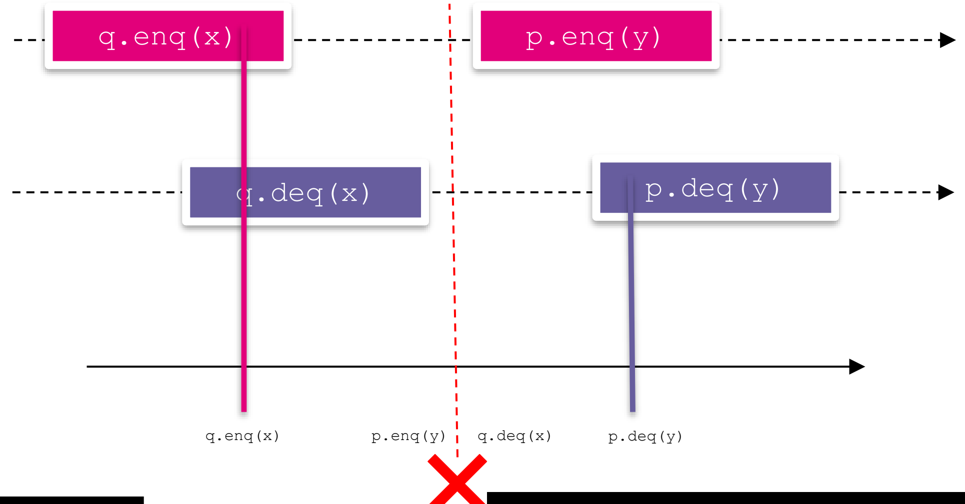


The sequentially consistent order from before now is not a valid linearization!

The reason is that it does not satisfy the real time ordering

Linearizability

- To show that a concurrent execution is linearizable, we must define linearization points within each method call, which map to sequential execution that satisfy the specification of the object
  - A *linearization point* defines the instant in the execution when the method call “takes effect”

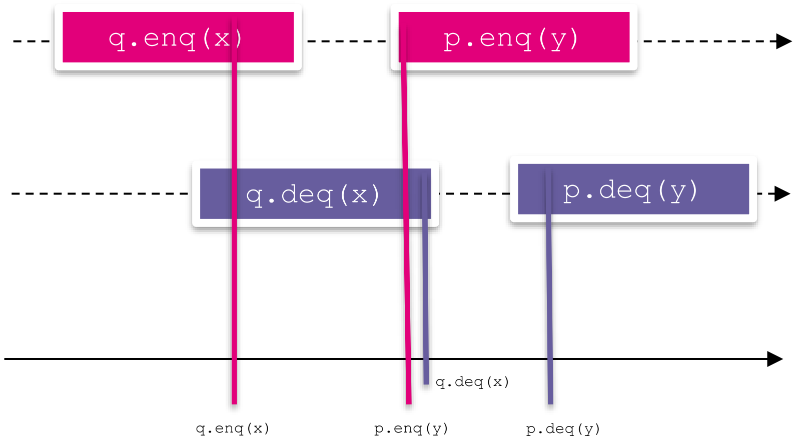


The sequentially consistent order from before now is not a valid linearization!

The reason is that it does not satisfy the real time ordering

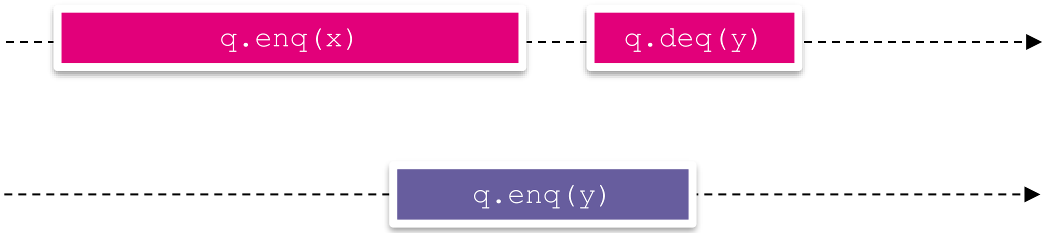
Linearizability

- To show that a concurrent execution is linearizable, we must define linearization points within each method call, which map to sequential execution that satisfy the specification of the object
  - A *linearization point* defines the instant in the execution when the method call "takes effect"



If q.deq(x) and p.enq(y) overlap, then the sequentially consistent order is a valid linearization

Sequential consistency



Is this concurrent execution linearizable?
(Recall we are working with a FIFO queue)

Linearizable Concurrent Objects

- Until now, linearizability is presented as a property of *executions*, not concurrent objects

- A concurrent object is linearizable iff
  - All executions are linearizable, and
  - All linearizations satisfy the sequential specification of the object

Proving this is hard

- To show that an object is linearizable first we must select its linearization points in the source code of the object class
  - Very often linearization points correspond to CAS operations

Linearizable Concurrent Objects


```
class WaitFreeQueue<T> {
  int head = 0, tail = 0;
  T[] items;

  public WaitFreeQueue(int capacity) {
    items = (T[]) new Object[capacity]
  }

  public void enq(T x) throws Exception {
    if (tail - head == items.length) // E1
      thrown new Exception(); // E2
    items[tail % items.length] = x; // E3
    tail++; // E4
  }

  public void deq() throws Exception {
    if (tail - head == 0)  // D1
      thrown new Exception(); // D2
    T x  = items[head % items.length]; // D3
    head++; // D4
    return x;
  }
}
```

Recall to assume all variables have volatile semantics and each program line is atomic.

Linearization points for enqueue

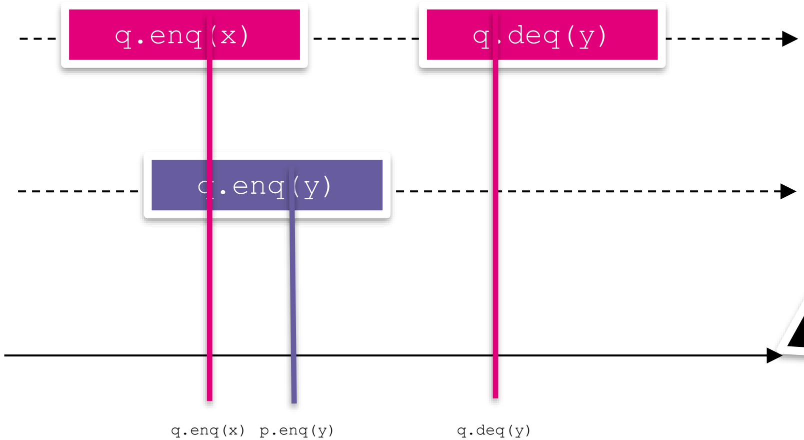
- E4 is the point when an enqueue is completed if the queue is not full
- E1 is the point when an enqueue is completed if the queue is full

Linearization points in dequeue

- D4 is the point when a dequeue is completed if the queue is not empty
- D1 is the point when a dequeue is completed when queue is empty

Linearizable Concurrent Objects

- Finding linearizable points in the object class is not enough to show correctness
  - Consider this interleaving: E1,E2,E3,E1,E2,E3,E4,E4
  - It corresponds to a linearization of this kind:



This does not satisfy the sequential spec of a FIFO queue!

Thus, we can conclude that WaitFreeQueue is *not* linearizable

Linearizability of MS Queue

Consult this slide for exercise 6.2.1


```
class MSQueue<T> implements UnboundedQueue<T> {
…
 public void enqueue(T item) {
    Node<T> node = new Node<T>(item, null);
    while (true) {
      Node<T> last = tail.get();
      Node<T> next = last.next.get();
      if (last == tail.get()) { // E7
        if (next == null)  { // E8
          // In quiescent state, try inserting new node
          if (last.next.compareAndSet(next, node)) { // E9
            // Insertion succeeded, try advancing tail
            tail.compareAndSet(last, node);
            return;
          }
        } else
        // Queue in intermediate state, advance tail
        tail.compareAndSet(last, next);
      }
    }
 }
…
}
```

- Enqueue has one linearization point:
  - E9 – if successfully executed, the element has been enqueued
- Correctness (informal but systematic, tries to cover all branches):
  - If two threads execute enqueue concurrently before tail and next are updated, then only one of them succeeds in executing E9 (and possible update the tail). The other fails and repeats the enqueuing
  - If a thread executes enqueue after another thread updated the tail, then E7 fails and it repeats the enqueue
  - If a thread executes enqueue after another thread updated next, then E8 fails, the thread tries to advance the tail, and it restarts the enqueue

Linearizability of MS Queue

Consult this slide for exercise 6.2.1


```
class MSQueue<T> implements UnboundedQueue<T> {
…

  public T dequeue() {
    while (true) {
      Node<T> first = head.get();
       Node<T> last = tail.get();
      Node<T> next = first.next.get(); // D3
       if (first == head.get()) { // D5
         if (first == last) { // D6
           if (next == null) // D7
             return null;
           else
             tail.compareAndSet(last, next);
         } else {
           T result = next.item;
           if (head.compareAndSet(first, next)) // D13
             return result;
         }
       }
    }
  }
…
}
```

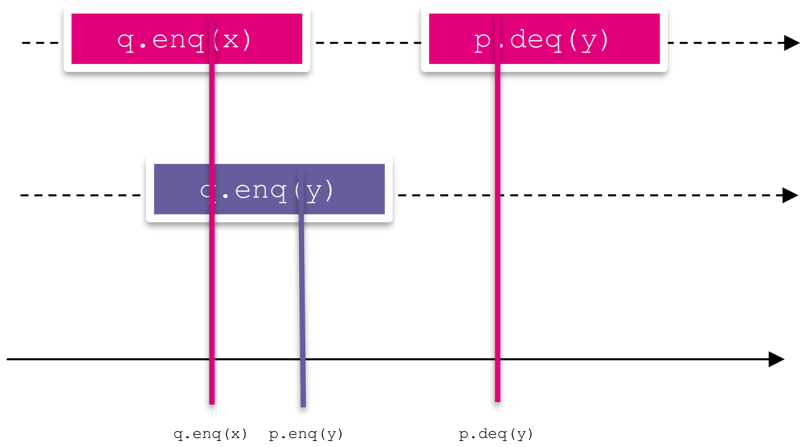
- Dequeue has two linearization points
  - D3 - if the queue is empty. After its execution, the evaluation of D7 is determined and whether the method will return null.
  - D13 - if successfully executed, the element has been dequeued
- Correctness (informal but systematic, tries to cover all branches):
  - If two threads execute dequeue concurrently before the head is updated (D5 succeeds for both) and the queue is not empty (D6 fails), then D13 succeeds for only one of them. The other restarts the dequeue
  - If a thread executes dequeue after another thread updated the head, then D5 fails and it restarts the dequeue
  - If a thread execute dequeue while another thread executed enqueue (E9) and before the enqueuing thread updates the tail, then D7 fails, the dequeuing thread tries to update the tail and restarts the dequeue

Linearizability of MS Queue

- The correctness arguments in the previous slides increase our confidence that all linearizations satisfy the FIFO queue spec

- This linearization is impossible in the MS Queue
- If pink executes E9 before purple, then either:
  - Purple restart the enqueue due to CAS failure when updating next
  - Purple restart the enqueue because the tail was updated by pink
  - Updates the tail due to pink not having done so yet

- Note that in no case y is enqueued before x (or replaced)



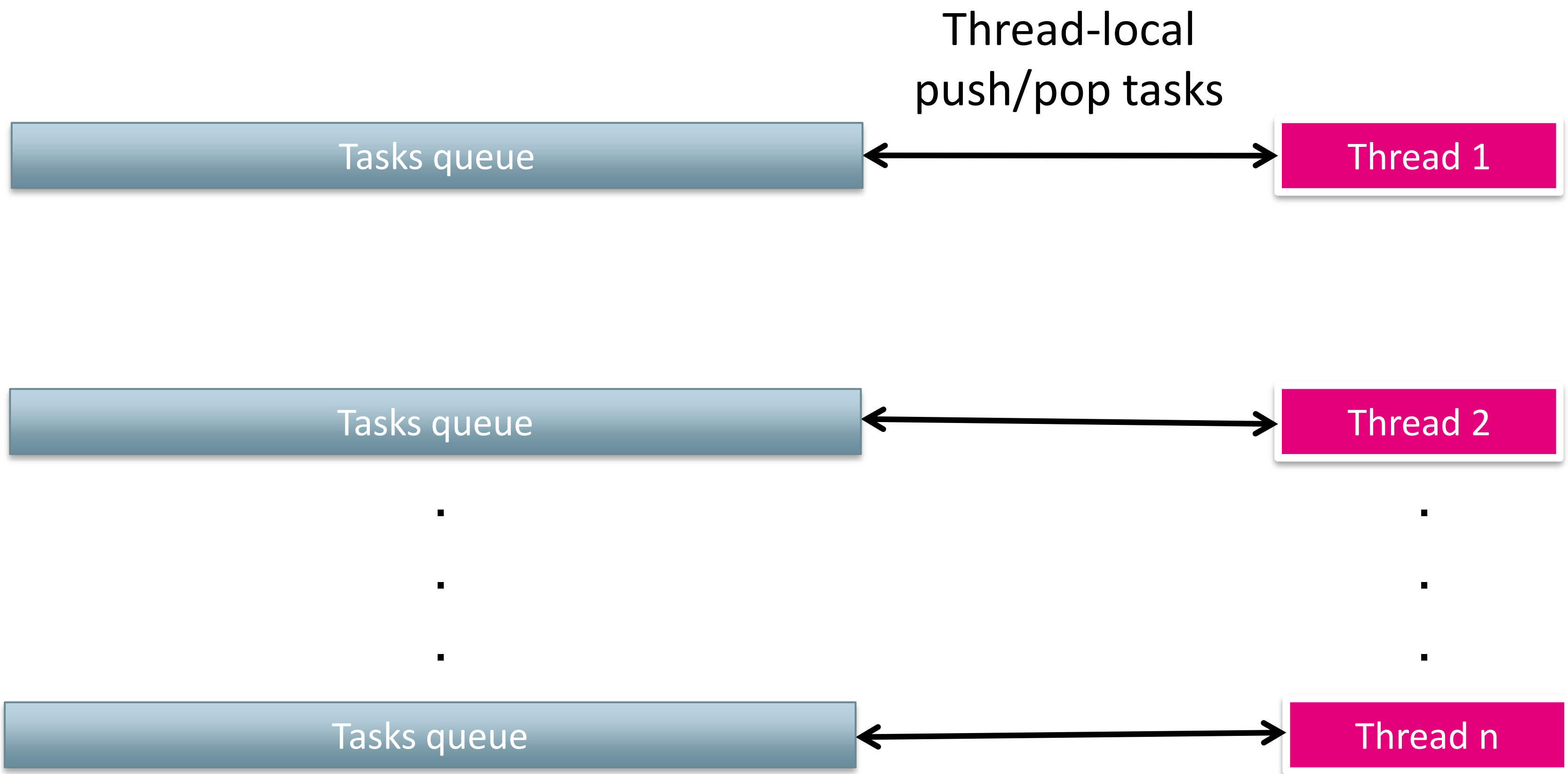
IT UNIVERSITY OF COPENHAGEN

Advantages of linearizability

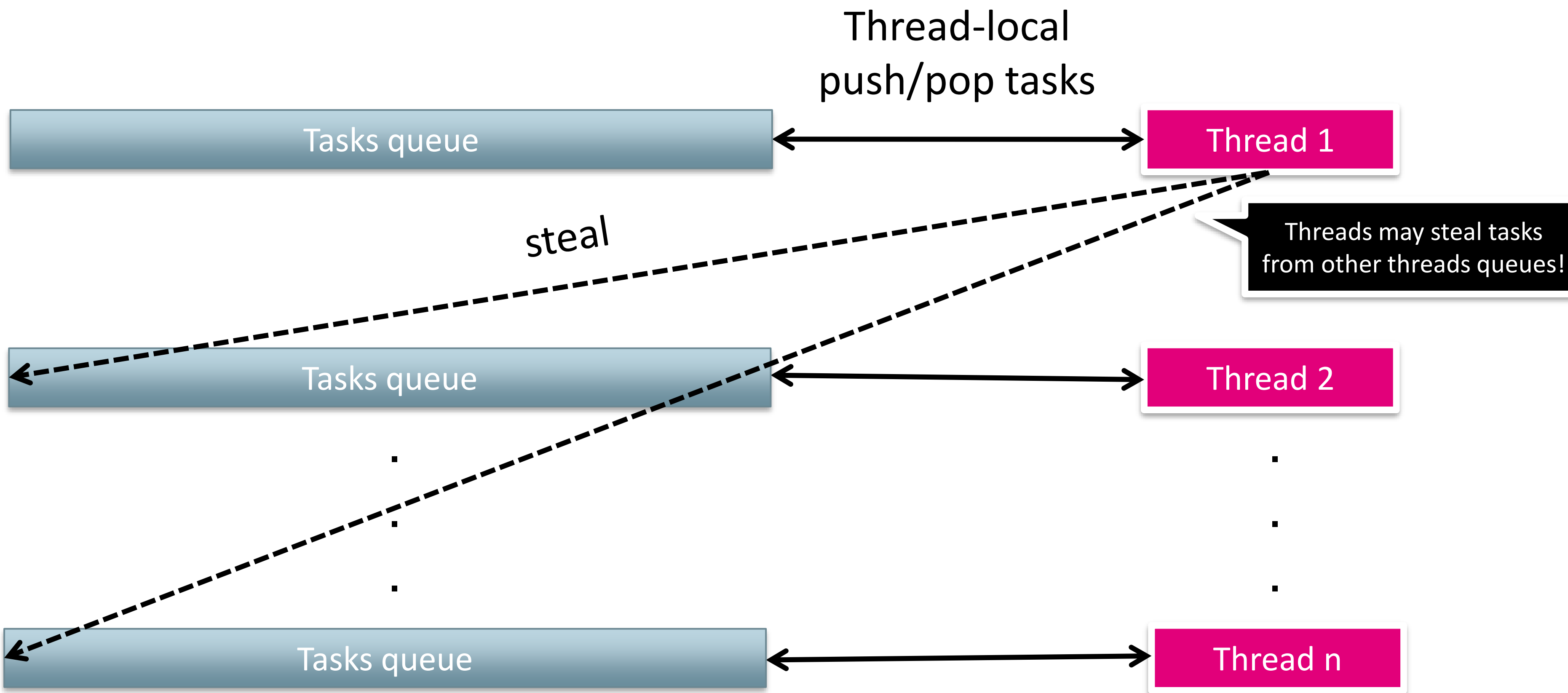
- It re-uses sequential specifications
  - Specifying the sequential behaviour of an object is rather intuitive
- **<u>It is a compositional property</u>**
  - If two objects are linearizable, any concurrent execution involving the two objects remains linearizable
  - Correctness proofs can be split into single objects that later can be safely composed
- It is a non-blocking property
  - It can be used to reason about objects that do not use locks

Work-stealing queues

Work-stealing task queue (intutition)

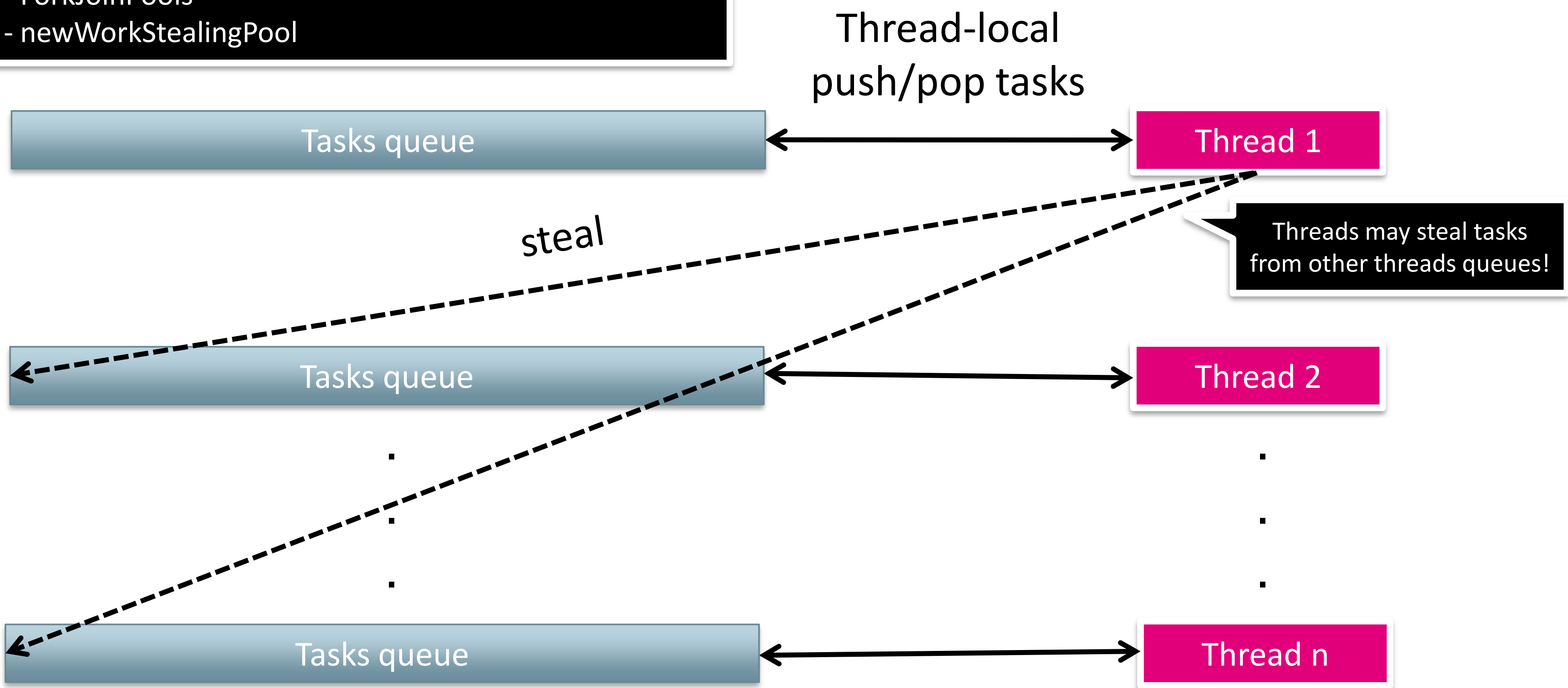


Work-stealing task queue (intutition)



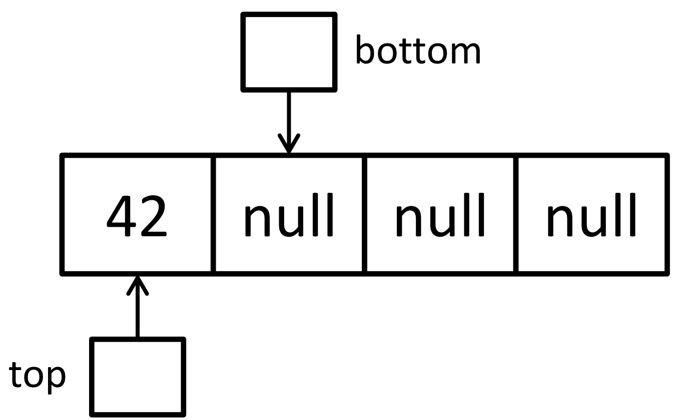
Work-stealing task queue (intutition)

It is used in the implementation thread pools (week 9):
- ForkJoinPools
- newWorkStealingPool



Work-stealing queues

- A work-stealing queue has the following methods
  - Push – adds an element at the bottom of the queue (thread-local)
  - Pop – removes an element from the bottom of the queue (thread-local)
  - Steal – removes an element from the top of the queue (concurrent)

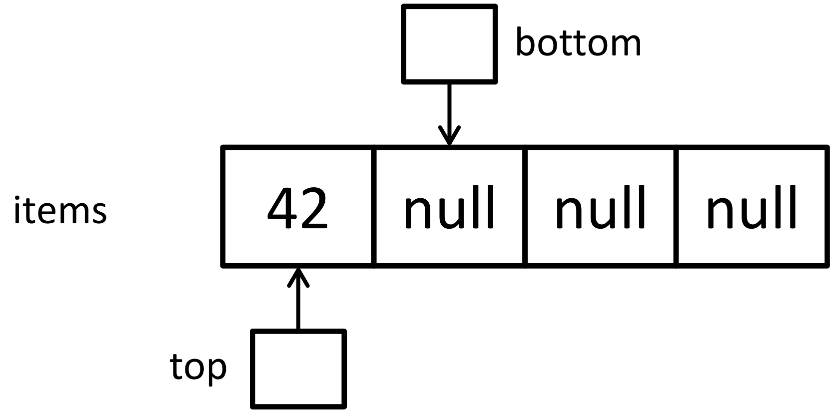



```
interface Deque<T> {
  void push(T item); // at bottom
  T pop();           // from bottom
  T steal();         // from top
}
```

We consider a simplified implementation with a fix size array

Chase-Lev work-stealing queue - state


```
class ChaseLevDeque<T> implements Deque<T> {
  private volatile long bottom = 0;
  private final AtomicLong top = new AtomicLong();
  private final T[] items;
…
}
```

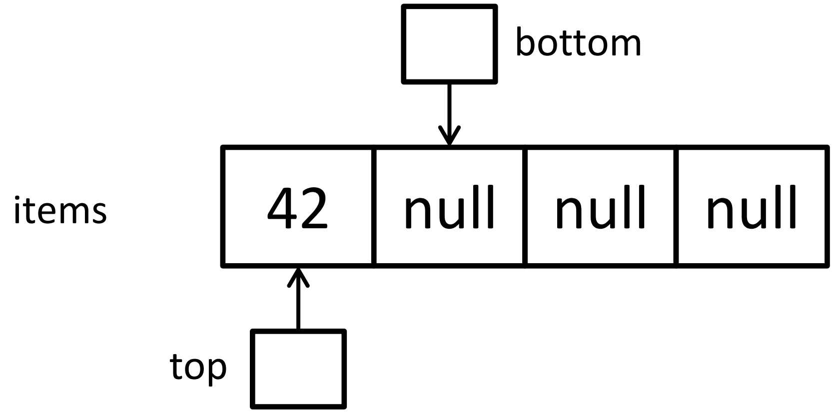


- The variable bottom is thread-local
  - Only the thread assigned to the queue can write it (other threads may read it)
- Any thread can read/write the variable top
  - We need an atomic variable to prevent data races
- For simplicity, we consider a fix-size array to store the elements of the queue
  - The array is used as a circular buffer

Chase-Lev work-stealing queue - state


```
class ChaseLevDeque<T> implements Deque<T> {
  private volatile long bottom = 0;
  private final AtomicLong top = new AtomicLong();
  private final T[] items;
…
}
```

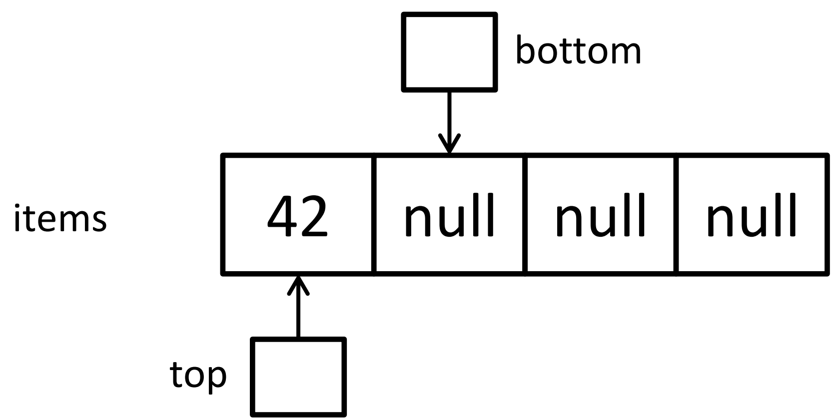
Why is volatile enough for bottom?



- The variable bottom is thread-local
  - Only the thread assigned to the queue can write it (other threads may read it)
- Any thread can read/write the variable top
  - We need an atomic variable to prevent data races
- For simplicity, we consider a fix-size array to store the elements of the queue
  - The array is used as a circular buffer

Chase-Lev work-stealing queue - push

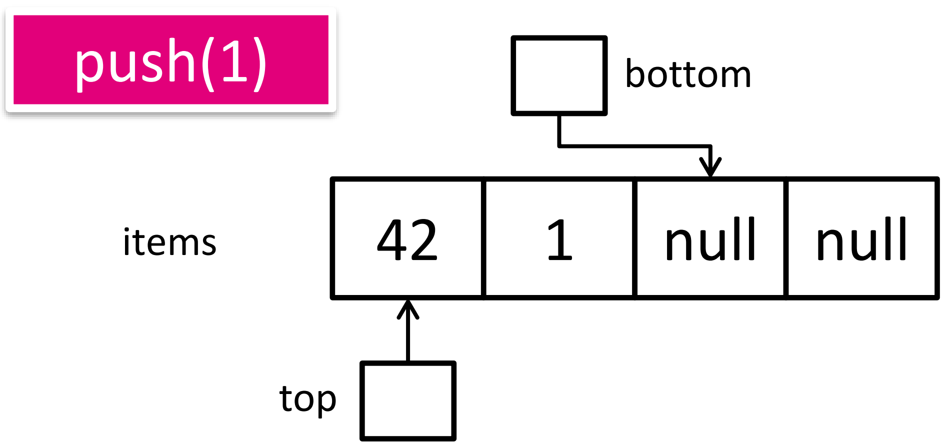

```
public void push(T item) { // at bottom
  final long b = bottom, t = top.get(), size = b - t;
  if (size == items.length)
    throw new RuntimeException("queue overflow");
  items[index(b, items.length)] = item;
  bottom = b+1;
}
```



- Thread-safe because it is assumed to be thread-local
  - Always the same thread executes this method
  - The thread only writes in bottom

Chase-Lev work-stealing queue - push


```
public void push(T item) { // at bottom
  final long b = bottom, t = top.get(), size = b - t;
  if (size == items.length)
    throw new RuntimeException("queue overflow");
  items[index(b, items.length)] = item;
  bottom = b+1;
}
```

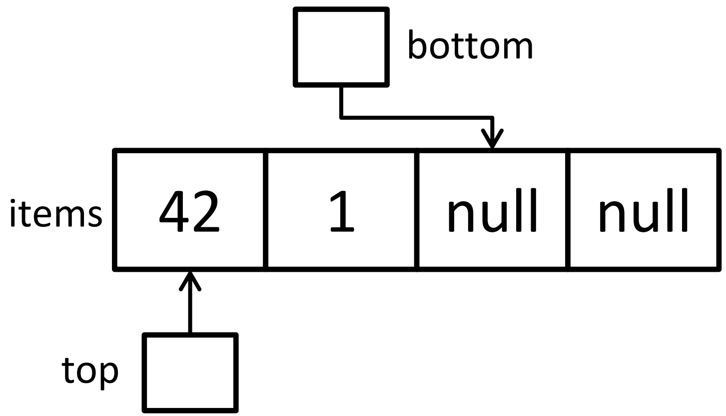


- Thread-safe because it is assumed to be thread-local
  - Always the same thread executes this method
  - The thread only writes in bottom

Chase-Lev work-stealing queue - steal


```
public T steal() { // from top
  final long t = top.get();
  final long b = bottom;
  final long size = b - t;
  if (size <= 0)
    return null;
  else {
    T result = items[index(t, items.length)];
    if (top.compareAndSet(t, t+1))
      return result;
    else
      return null;
  }
}
```

- It is executed by multiple threads
- Only reads bottom
- Performs a CAS on top to steal the top element
  - Only if not empty

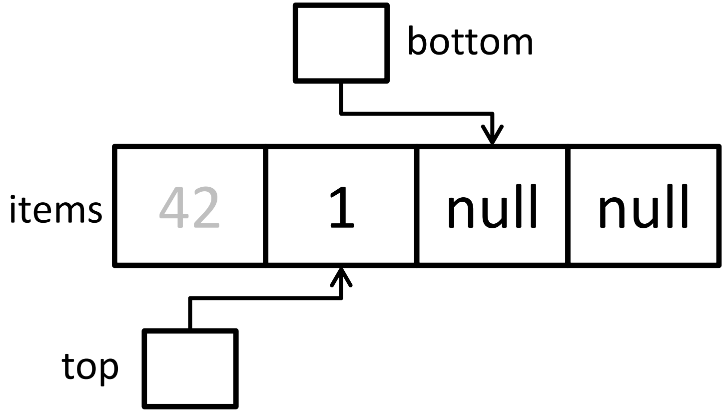


Chase-Lev work-stealing queue - steal


```
public T steal() { // from top
  final long t = top.get();
  final long b = bottom;
  final long size = b - t;
  if (size <= 0)
    return null;
  else {
    T result = items[index(t, items.length)];
    if (top.compareAndSet(t, t+1))
      return result;
    else
      return null;
  }
}
```

steal() -> 42

- It is executed by multiple threads
- Only reads bottom
- Performs a CAS on top to steal the top element
  - Only if not empty

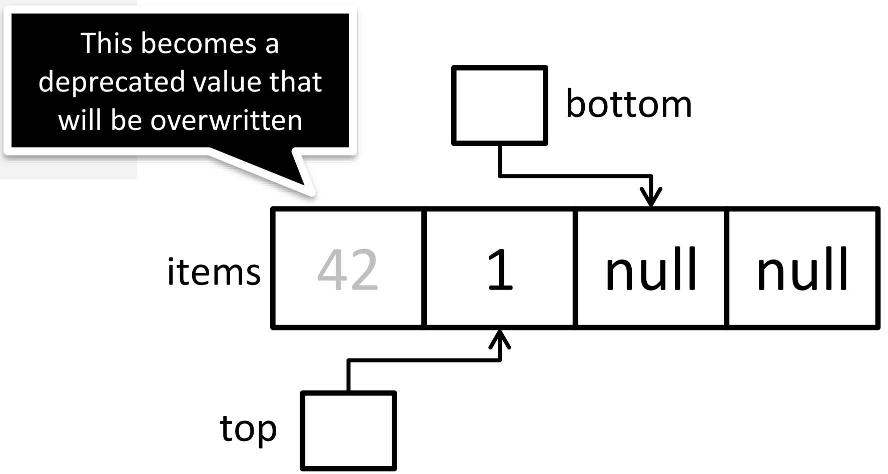


Chase-Lev work-stealing queue - steal


```
public T steal() { // from top
  final long t = top.get();
  final long b = bottom;
  final long size = b - t;
  if (size <= 0)
    return null;
  else {
    T result = items[index(t, items.length)];
    if (top.compareAndSet(t, t+1))
      return result;
    else
      return null;
  }
}
```

steal() -> 42

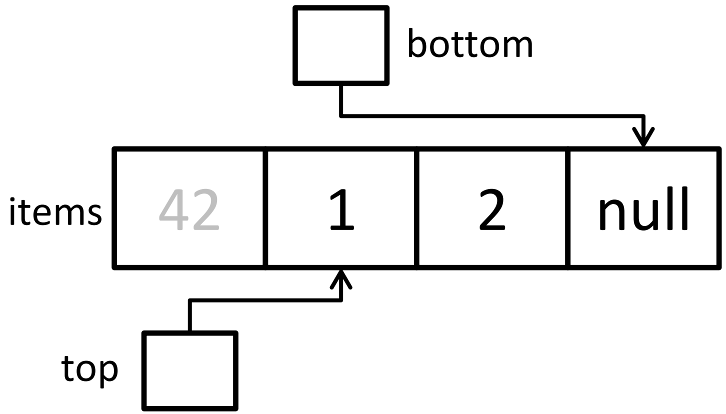
- It is executed by multiple threads
- Only reads bottom
- Performs a CAS on top to steal the top element
  - Only if not empty



Chase-Lev work-stealing queue - pop


```
public T pop() { // from bottom
  final long b = bottom - 1;
  bottom = b;
  final long t = top.get(),
  final long afterSize = b - t;
  if (afterSize < 0) {
    bottom = t;
    return null;
  } else {
    T result = items[index(b, items.length)];
    if (afterSize > 0)
      return result;
    else {
      if (!top.compareAndSet(t, t+1))
          result = null;
      bottom = t+1;
      return result;
    }
  }
}
```

- Thread-local but more subtle than push
- It updates bottom (thread-local) and possibly top (concurrent)

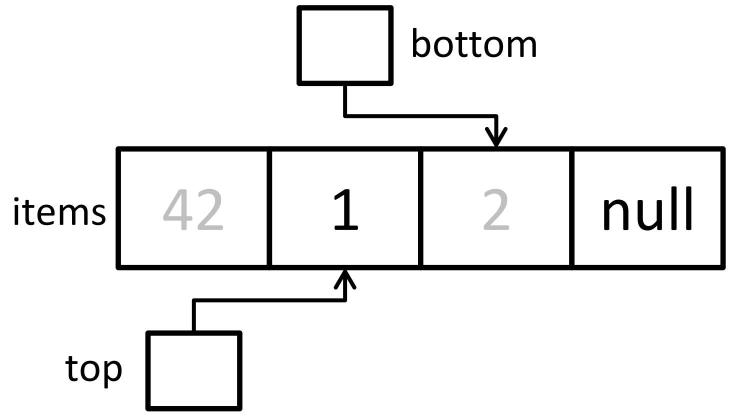


Chase-Lev work-stealing queue - pop


```
public T pop() { // from bottom
  final long b = bottom - 1;
  bottom = b;
  final long t = top.get(),
  final long afterSize = b - t;
  if (afterSize < 0) {
    bottom = t;
    return null;
  } else {
    T result = items[index(b, items.length)];
    if (afterSize > 0)
      return result;
    else {
      if (!top.compareAndSet(t, t+1))
          result = null;
      bottom = t+1;
      return result;
    }
  }
}
```

pop() -> 2

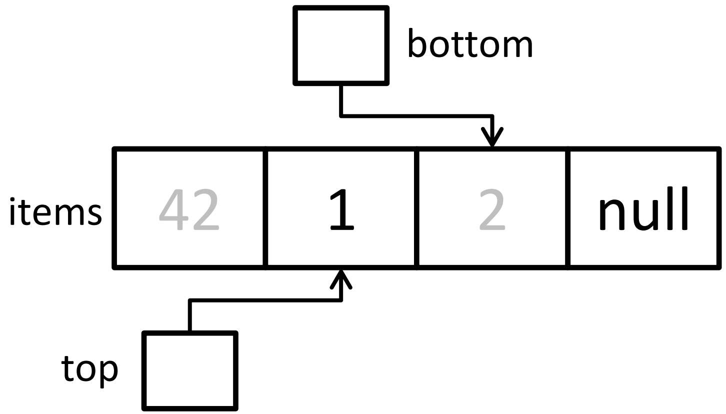
- When only the assign thread executes, then we simply update bottom and return the element



Chase-Lev work-stealing queue - pop


```
public T pop() { // from bottom
  final long b = bottom - 1;
  bottom = b;
  final long t = top.get(),
  final long afterSize = b - t;
  if (afterSize < 0) {
    bottom = t;
    return null;
  } else {
    T result = items[index(b, items.length)];
    if (afterSize > 0)
      return result;
    else {
      if (!top.compareAndSet(t, t+1))
          result = null;
      bottom = t+1;
      return result;
    }
  }
}
```

- What if we had pop() and steal() concurrently?

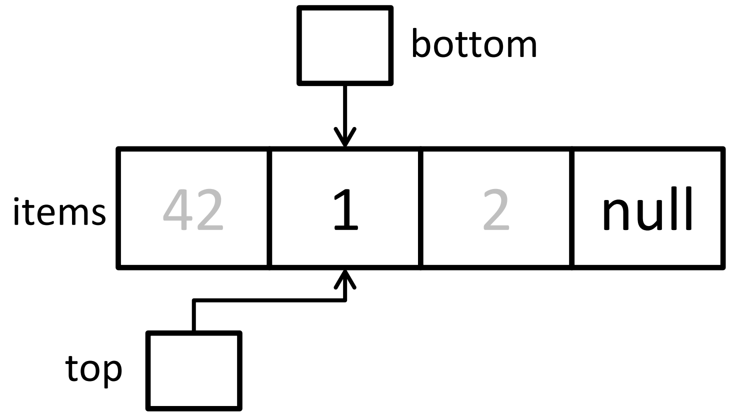


Chase-Lev work-stealing queue - pop


```
public T pop() { // from bottom
  final long b = bottom - 1;
  bottom = b;
  final long t = top.get(),
  final long afterSize = b - t;
  if (afterSize < 0) {
    bottom = t;
    return null;
  } else {
    T result = items[index(b, items.length)];
    if (afterSize > 0)
      return result;
    else {
      if (!top.compareAndSet(t, t+1))
          result = null;
      bottom = t+1;
      return result;
    }
  }
}
```

pop() -> ?

- What if we had pop() and steal() concurrently?



Chase-Lev ue - pop

```
public T steal() { // from top
  final long t = top.get();
  final long b = bottom;
  final long size = b - t;
  if (size <= 0)
    return null;
  else {
    T result = items[index(t, items.length)];
    if (top.compareAndSet(t, t+1))
      return result;
    else
      return null;
  }
}
```

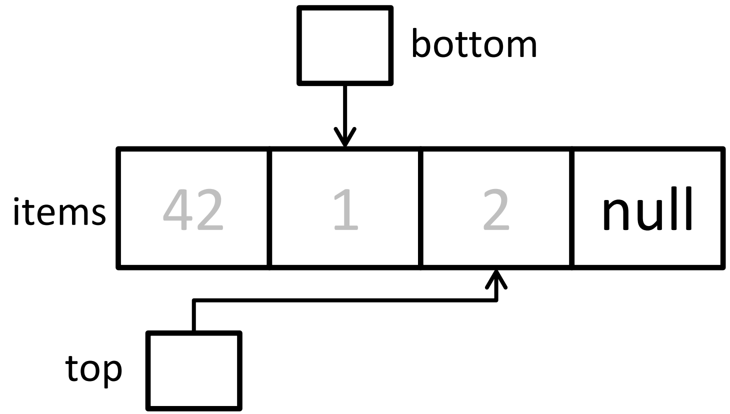
```
public T pop() { /
  final long b = b
  bottom = b;
  final long t = t
  final long afterSize = b - t;
  if (afterSize < 0) {
    bottom = t;
    return null;
  } else {
    T result = items[index(b, items.length)];
    if (afterSize > 0)
      return result;
    else {
      if (!top.compareAndSet(t, t+1))
          result = null;
      bottom = t+1;
      return result;
    }
  }
}
```

pop() -> ?

steal() -> ?

What if we had pop() and steal() concurrently?

- Whatever thread succeeds in the CAS operation gets the element



Chase-Lev ue - pop


```
public T steal() { // from top
  final long t = top.get();
  final long b = bottom;
  final long size = b - t;
  if (size <= 0)
    return null;
  else {
    T result = items[index(t, items.length)];
    if (top.compareAndSet(t, t+1))
      return result;
    else
      return null;
  }
}
```

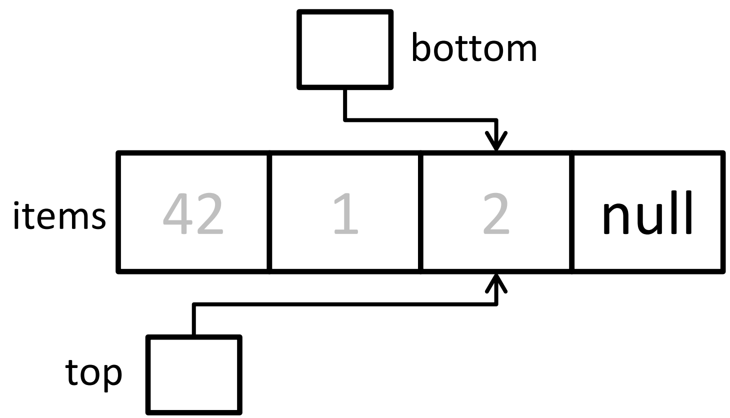
```
public T pop() { /
  final long b = b
  bottom = b;
  final long t = t
  final long afterSize = b - t;
  if (afterSize < 0) {
    bottom = t;
    return null;
  } else {
    T result = items[index(b, items.length)];
    if (afterSize > 0)
      return result;
    else {
      if (!top.compareAndSet(t, t+1))
          result = null;
      bottom = t+1;
      return result;
    }
  }
}
```

pop() -> ?

steal() -> ?

What if we had pop() and steal() concurrently?

- Whatever thread succeeds in the CAS operation gets the element
- Afterwards, pop always fixes the bottom variable



Sequential consistency and the Java memory model

Sequential consistency in Java

JLS Sequential Consistency definition:

- *"Within a sequentially consistent execution, there is*
  - *a total order over all individual actions (such as reads and writes) which is consistent with the order of the program, and*
  - *each individual action is atomic and is immediately visible to every thread."*

Ensuring sequential consistency in a runtime environment such as the JVM is very hard and often counterproductive

- JLS: *"If we were to use sequential consistency as our memory model, many of the compiler and processor optimizations that we have discussed would be illegal."*

Sequential consistency in Java

- Software engineers like sequential consistency!
  - Sequentially consistent executions are the expected behaviour of concurrent programs, as they imply no visibility issues

- Happens-before to the rescue!
  - Lemma: *"Correctly synchronized programs exhibit only sequentially consistent behavior"* [The Java Memory Model paper]

Correctly synchronized programs in Java

- A program is correctly synchronized iff all executions are free from data races
- An execution is any sequence of program operations that obeys program order and synchronization order
- The synchronization order is a total order between synchronization operations (lock/unlock/read volatile/write volatile) that is consistent with program order and where lock and unlock operations are correctly nested

Correctly synchronized programs in Java


```
lock l = new Lock();
for (i = 1; i < 3; i++) {

  new Thread(() -> {
       l.lock()                 // (1)
       int temp = counter;      // (2)
       counter = temp + 1;      // (3)
       l.unlock()               // (4)
    }).start();
}
```

Is this program correctly synchronized?
To answer this question, we must show data race freedom for all executions.

Correctly synchronized programs in Java


```
lock l = new Lock();
for (i = 1; i < 3; i++) {

  new Thread(() -> {
        l.lock()                  // (1)
        int temp = counter;       // (2)
        counter = temp + 1;       // (3)
        l.unlock()                // (4)
    }).start();
}
```

<u>Is this program correctly synchronized?</u>
To answer this question, we must show data race freedom for all executions.

Formally, we must show that

$$t_1(2) \rightarrow t_2(3)$$
$$t_1(3) \rightarrow t_2(2)$$

or

$$t_2(2) \rightarrow t_1(3)$$
$$t_2(3) \rightarrow t_1(2)$$

Correctly synchronized programs in Java

- Only two possible synchronization orders:

$$t_1(1), t_1(4), t_2(1), t_2(4)$$
$$t_2(1), t_2(4), t_1(1), t_1(4)$$

- The following program order happens-before pairs must be enforced in all executions, for $x \in \{1,2\}$

$$t_x(1) \rightarrow t_x(2) \text{ and } t_x(2) \rightarrow t_x(3) \text{ and } t_x(3) \rightarrow t_x(4)$$

- Thus, only two executions can occur

$$t_1(1), t_1(2), t_1(3), t_1(4), t_2(1), t_2(2), t_2(3), t_2(4)$$
$$t_2(1), t_2(2), t_2(3), t_2(4), t_1(1), t_1(2), t_1(3), t_1(4)$$

```
lock l = new Lock();
for (i = 1; i < 3; i++) {

  new Thread(() -> {
       l.lock()                 // (1)
       int temp = counter;      // (2)
       counter = temp + 1;      // (3)
       l.unlock()               // (4)
    }).start();
}
```

Is this program correctly synchronized?
To answer this question, we must show data race freedom for all executions.

Formally, we must show that

$$t_1(2) \rightarrow t_2(3)$$
$$t_1(3) \rightarrow t_2(2)$$

or

$$t_2(2) \rightarrow t_1(3)$$
$$t_2(3) \rightarrow t_1(2)$$

Correctly synchronized programs in Java

- Only two possible synchronization orders:

$$t_1(1), t_1(4), t_2(1), t_2(4)$$
$$t_2(1), t_2(4), t_1(1), t_1(4)$$

Only lock/unlock synchronization operations, and two possible ways to correctly nest them.

```
lock l = new Lock();
for (i = 1; i < 3; i++) {

  new Thread(() -> {
      l.lock()                    // (1)
      int temp = counter;         // (2)
      counter = temp + 1;         // (3)
      l.unlock()                  // (4)
    }).start();
}
```

- The following program order happens-before pairs must be enforced in all executions, for $x \in \{1,2\}$

$$t_x(1) \rightarrow t_x(2) \;\text{ and }\; t_x(2) \rightarrow t_x(3) \;\text{ and }\; t_x(3) \rightarrow t_x(4)$$

Is this program correctly synchronized?
To answer this question, we must show data race freedom for all executions.

- Thus, only two executions can occur

$$t_1(1), t_1(2), t_1(3), t_1(4), t_2(1), t_2(2), t_2(3), t_2(4)$$
$$t_2(1), t_2(2), t_2(3), t_2(4), t_1(1), t_1(2), t_1(3), t_1(4)$$

Formally, we must show that

$$t_1(2) \rightarrow t_2(3)$$
$$t_1(3) \rightarrow t_2(2)$$

or

$$t_2(2) \rightarrow t_1(3)$$
$$t_2(3) \rightarrow t_1(2)$$

Correctly synchronized programs in Java

- Only two possible synchronization orders:

$$t_1(1), t_1(4), t_2(1), t_2(4)$$
$$t_2(1), t_2(4), t_1(1), t_1(4)$$

Only lock/unlock synchronization operations, and two possible ways to correctly nest them.

```
lock l = new Lock();
for (i = 1; i < 3; i++) {

  new Thread(() -> {
       l.lock()                  // (1)
       int temp = counter;       // (2)
       counter = temp + 1;       // (3)
       l.unlock()                // (4)
    }).start();
}
```

- The following program order happens-before pairs must be enforced in all executions, for $x \in \{1,2\}$

$$t_x(1) \rightarrow t_x(2) \text{ and } t_x(2) \rightarrow t_x(3) \text{ and } t_x(3) \rightarrow t_x(4)$$

- Thus, only two executions can occur

By program order we have the following happens-before pairs in the executions

$$t_1(1) \rightarrow t_1(2) \rightarrow t_1(3) \rightarrow t_1\ (4), t_2(1) \rightarrow t_2(2) \rightarrow t_2(3) \rightarrow t_2(4)$$
$$t_2(1) \rightarrow t_2(2) \rightarrow t_2(3) \rightarrow t_2(4), t_1(1) \rightarrow t_1(2) \rightarrow t_1(3) \rightarrow t_1(4)$$

Formally, we must show that

$$t_1(2) \rightarrow t_2(3)$$
$$t_1(3) \rightarrow t_2(2)$$

or

$$t_2(2) \rightarrow t_1(3)$$
$$t_2(3) \rightarrow t_1(2)$$

Correctly synchronized programs in Java

- Only two possible synchronization orders:

$$t_1(1), t_1(4), t_2(1), t_2(4)$$
$$t_2(1), t_2(4), t_1(1), t_1(4)$$

Only lock/unlock synchronization operations, and two possible ways to correctly nest them.

```
lock l = new Lock();
for (i = 1; i < 3; i++) {

  new Thread(() -> {
       l.lock()                // (1)
       int temp = counter;     // (2)
       counter = temp + 1;     // (3)
       l.unlock()              // (4)
    }).start();
}
```

- The following program order happens-before pairs must be enforced in all executions, for $x \in \{1,2\}$

$$t_x(1) \rightarrow t_x(2) \text{ and } t_x(2) \rightarrow t_x(3) \text{ and } t_x(3) \rightarrow t_x(4)$$

- Thus, only two executions can occur

By program order we have the following happens-before pairs in the executions

By the lock rule we have the following happens-before pairs in the executions

$$t_1(1) \rightarrow t_1(2) \rightarrow t_1(3) \rightarrow t_1(4) \rightarrow t_2(1) \rightarrow t_2(2) \rightarrow t_2(3) \rightarrow t_2(4)$$
$$t_2(1) \rightarrow t_2(2) \rightarrow t_2(3) \rightarrow t_2(4) \rightarrow t_1(1) \rightarrow t_1(2) \rightarrow t_1(3) \rightarrow t_1(4)$$

Formally, we must show that

$$t_1(2) \rightarrow t_2(3)$$
$$t_1(3) \rightarrow t_2(2)$$

or

$$t_2(2) \rightarrow t_1(3)$$
$$t_2(3) \rightarrow t_1(2)$$

Correctly synchronized programs in Java

- Only two possible synchronization orders:

$$t_1(1), t_1(4), t_2(1), t_2(4)$$
$$t_2(1), t_2(4), t_1(1), t_1(4)$$

Only lock/unlock synchronization operations, and two possible ways to correctly nest them.

```
lock l = new Lock();
for (i = 1; i < 3; i++) {

  new Thread(() -> {
      l.lock()                 // (1)
      int temp = counter;      // (2)
      counter = temp + 1;      // (3)
      l.unlock()               // (4)
    }).start();
}
```

- The following program order happens-before pairs must be enforced in all executions, for $x \in \{1,2\}$

$$t_x(1) \rightarrow t_x(2) \text{ and } t_x(2) \rightarrow t_x(3) \text{ and } t_x(3) \rightarrow t_x(4)$$

- Thus, only two executions can occur

By program order we have the following happens-before pairs in the executions

By the lock rule we have the following happens-before pairs in the executions

By the transitivity we have the following happens-before pairs in the executions

$$t_1(1) \rightarrow t_1(2) \rightarrow t_1(3) \rightarrow t_1(4) \rightarrow t_2(1) \rightarrow t_2(2) \rightarrow t_2(3) \rightarrow t_2(4)$$
$$t_2(1) \rightarrow t_2(2) \rightarrow t_2(3) \rightarrow t_2(4) \rightarrow t_1(1) \rightarrow t_1(2) \rightarrow t_1(3) \rightarrow t_1(4)$$

Formally, we must show that

$$t_1(2) \rightarrow t_2(3)$$
$$t_1(3) \rightarrow t_2(2)$$

or

$$t_2(2) \rightarrow t_1(3)$$
$$t_2(3) \rightarrow t_1(2)$$

Correctly synchronized programs in Java

- Only two possible synchronization orders:

$$t_1(1), t_1(4), t_2(1), t_2(4)$$
$$t_2(1), t_2(4), t_1(1), t_1(4)$$

Only lock/unlock synchronization operations, and two possible ways to correctly nest them.

```
lock l = new Lock();
for (i = 1; i < 3; i++) {

  new Thread(() -> {
      l.lock()                 // (1)
      int temp = counter;      // (2)
      counter = temp + 1;      // (3)
      l.unlock()               // (4)
    }).start();
}
```

- The following program order happens-before pairs must be enforced in all executions, for $x \in \{1,2\}$

$$t_x(1) \rightarrow t_x(2) \text{ and } t_x(2) \rightarrow t_x(3) \text{ and } t_x(3) \rightarrow t_x(4)$$

- Thus, only two executions can occur

By program order we have the following happens-before pairs in the executions

By the lock rule we have the following happens-before pairs in the executions

By the transitivity we have the following happens-before pairs in the executions

$$t_1(1) \rightarrow t_1(2) \rightarrow t_1(3) \rightarrow t_1\ (4) \rightarrow t_2(1) \rightarrow t_2(2) \rightarrow t_2(3) \rightarrow t_2(4)$$
$$t_2(1) \rightarrow t_2(2) \rightarrow t_2(3) \rightarrow t_2(4) \rightarrow t_1(1) \rightarrow t_1(2) \rightarrow t_1(3) \rightarrow t_1(4)$$

Formally, we must show that

$t_1(2) \rightarrow t_2(3)$ ☑

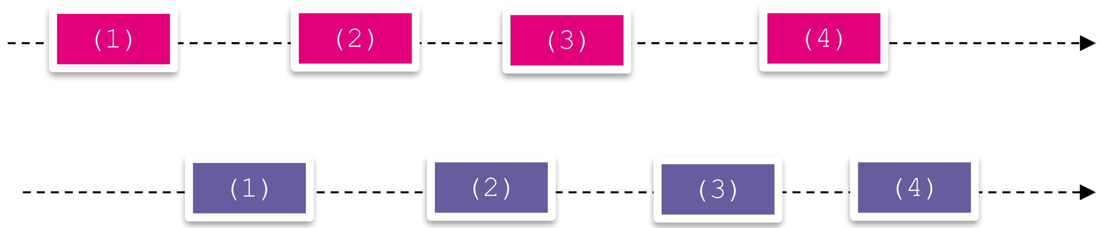
$t_1(3) \rightarrow t_2(2)$ ☑

or

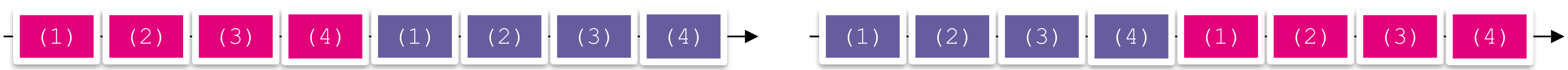
$t_2(2) \rightarrow t_1(3)$ ☑

$t_2(3) \rightarrow t_1(2)$ ☑

Sequential consistency in Java

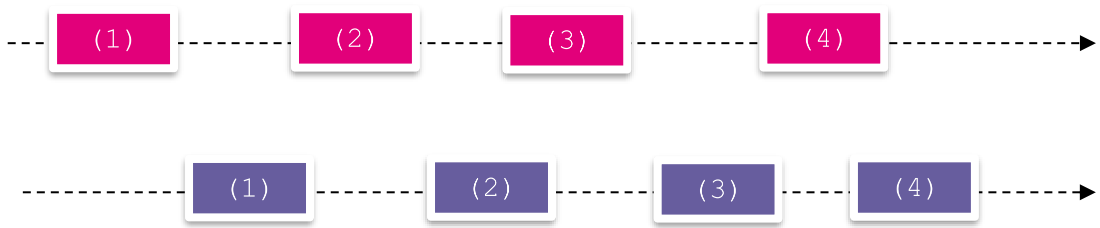


As consequence of having a correctly synchronized program, the JVM is designed to only produce executions that exhibit any of the two sequential operation orderings below. Most importantly, note that both are ***sequentially consistent***
(we get sequential consistency---for free---if the program is correctly synchronized)

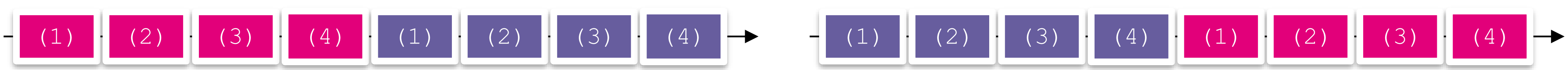


Sequential consistency in Java

BTW, we have also shown mutual exclusion!



As consequence of having a correctly synchronized program, the JVM is designed to only produce executions that exhibit any of the two sequential operation orderings below.
Most importantly, note that both are ***sequentially consistent***
(we get sequential consistency---for free---if the program is correctly synchronized)



Guest lecture next week!

Agenda

- Revisit
  - Progress conditions
  - Lock-free queue
- Linearizability
  - Sequential consistency
  - Definition of Linearizability
  - Linearizable concurrent objects
- Work-stealing queues
- BONUS: Sequential consistency and the Java memory model