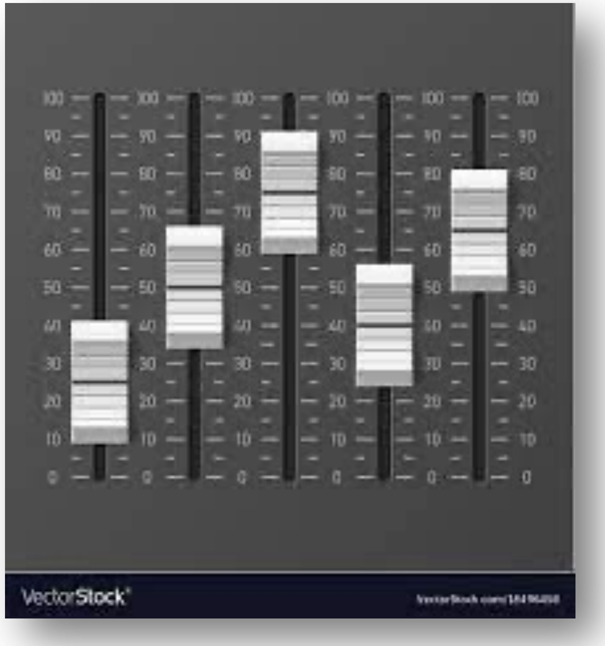
CSE113: Parallel Programming

Feb. 2, 2022

- **Topics**:
  - Input/output queues
  - Producer consumer queues
    - Synchronous
    - Circular buffer



Announcements

- HW1 grades might be delayed until Monday
  - Let us know ASAP if there are issues
- Homework 2 is due today
  - Sanya has office hours
  - We will keep an eye on Piazza and try to ask questions asked before 5 pm
- Homwork 3 will be released today by midnight
  - Due in 2 weeks

Announcements

- Midterm is released on Monday
  - asynchronous, 1 week (no time limit)
  - Open note, open internet (to a reasonable extent: no googling exact questions or asking questions on forums)
  - do not discuss with classmates AT ALL while the test is active
  - **No late tests will be accepted.**

- **Prioritize midterm next week!**

Homework clarifications

- Conditional variables
  - They are **not** allowed in your solution, but they are interesting
  - https://en.cppreference.com/w/cpp/thread/condition_variable

- Part 2: reader/writer
  - You cannot significantly slow down readers in isolation

- Part 3: keeping the structure:
  - you can re-arrange functions, just no changing the high-level implementation

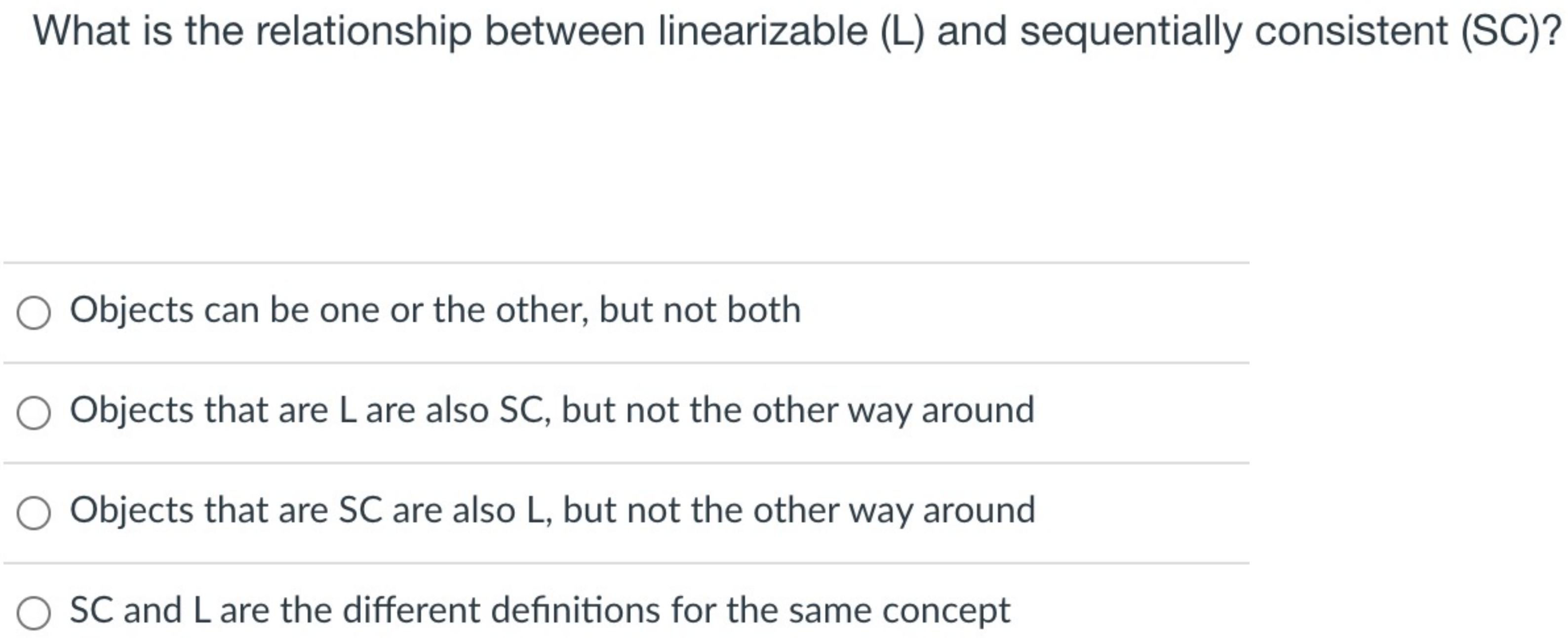
Homework clarifications

- You can share results, but not code

Today’s Quiz

- Due Monday by class. Please do it!

Previous quiz



What is the relationship between linearizable (L) and sequentially consistent (SC)?

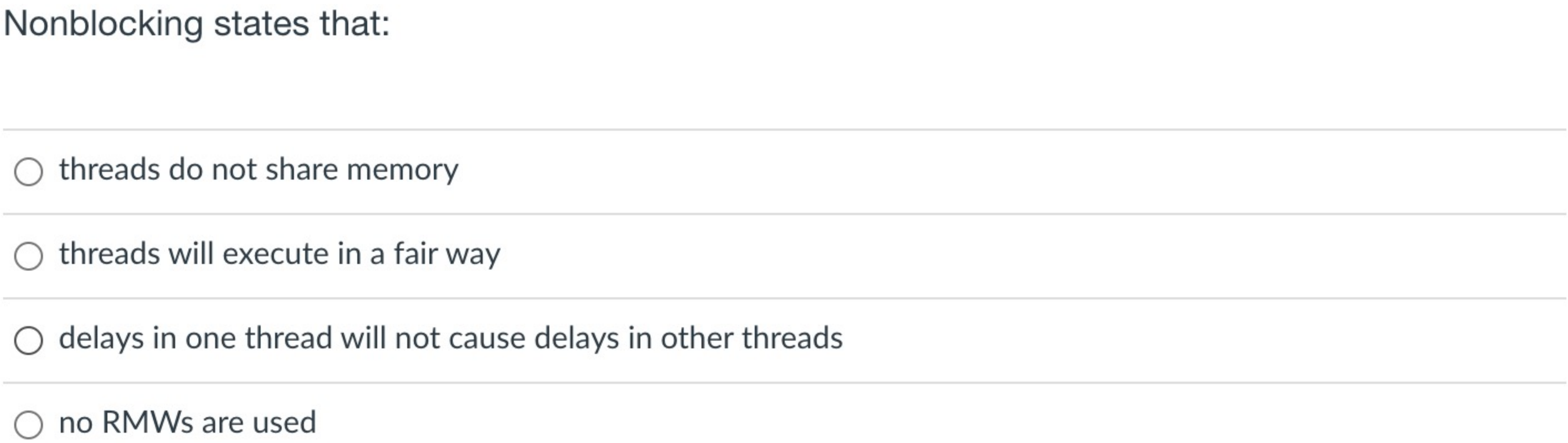
○ Objects can be one or the other, but not both

○ Objects that are L are also SC, but not the other way around

○ Objects that are SC are also L, but not the other way around

○ SC and L are the different definitions for the same concept

Previous quiz



Nonblocking states that:

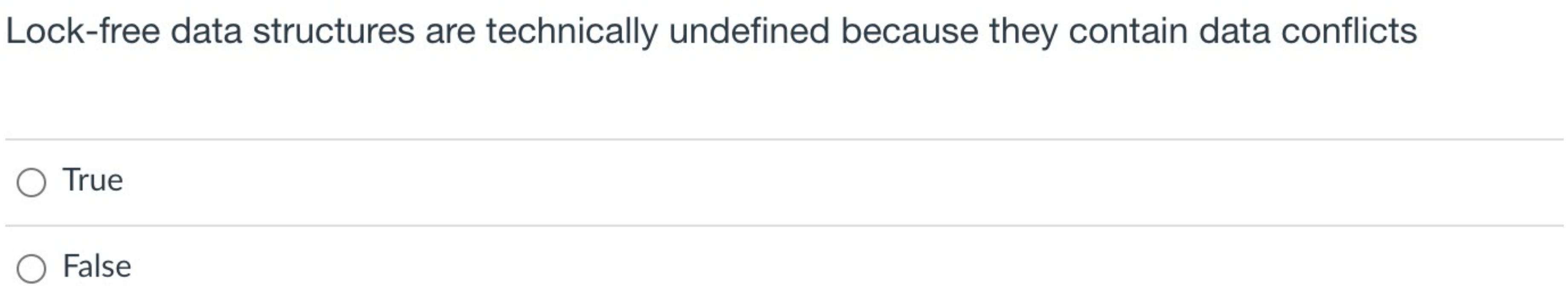
○ threads do not share memory

○ threads will execute in a fair way

○ delays in one thread will not cause delays in other threads

○ no RMWs are used

Previous quiz

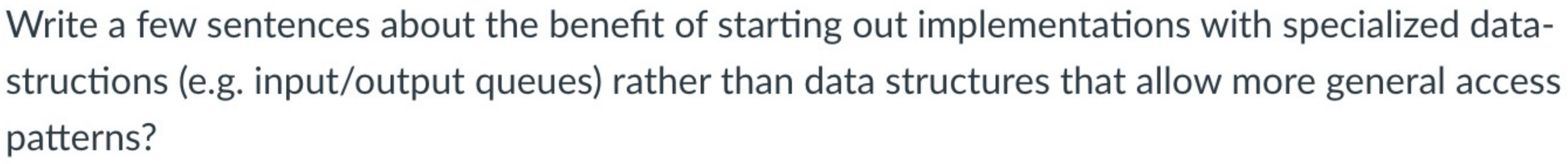


Lock-free data structures are technically undefined because they contain data conflicts

○ True

○ False

Previous quiz



Write a few sentences about the benefit of starting out implementations with specialized data-structions (e.g. input/output queues) rather than data structures that allow more general access patterns?

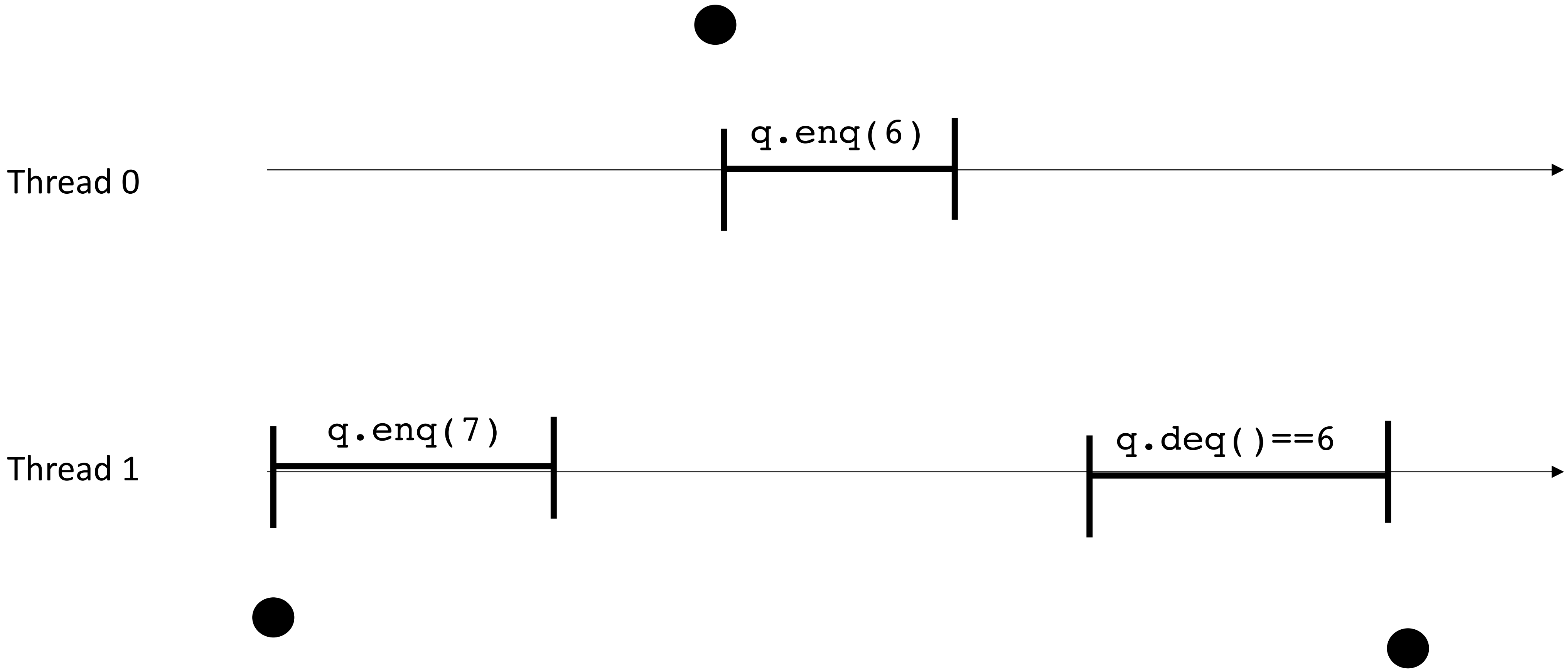
Review

Linearizability

Linearizability

each command gets a linearization point.

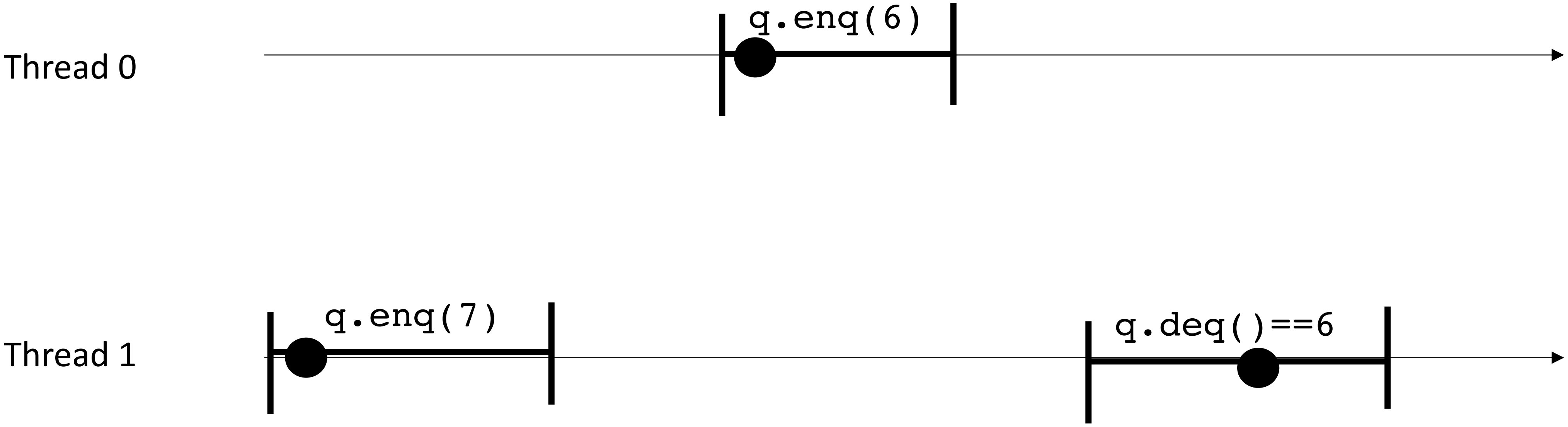
You can place the point any where between its innovation and response!



Linearizability

each command gets a linearization point.

You can place the point any where between its innovation and response!

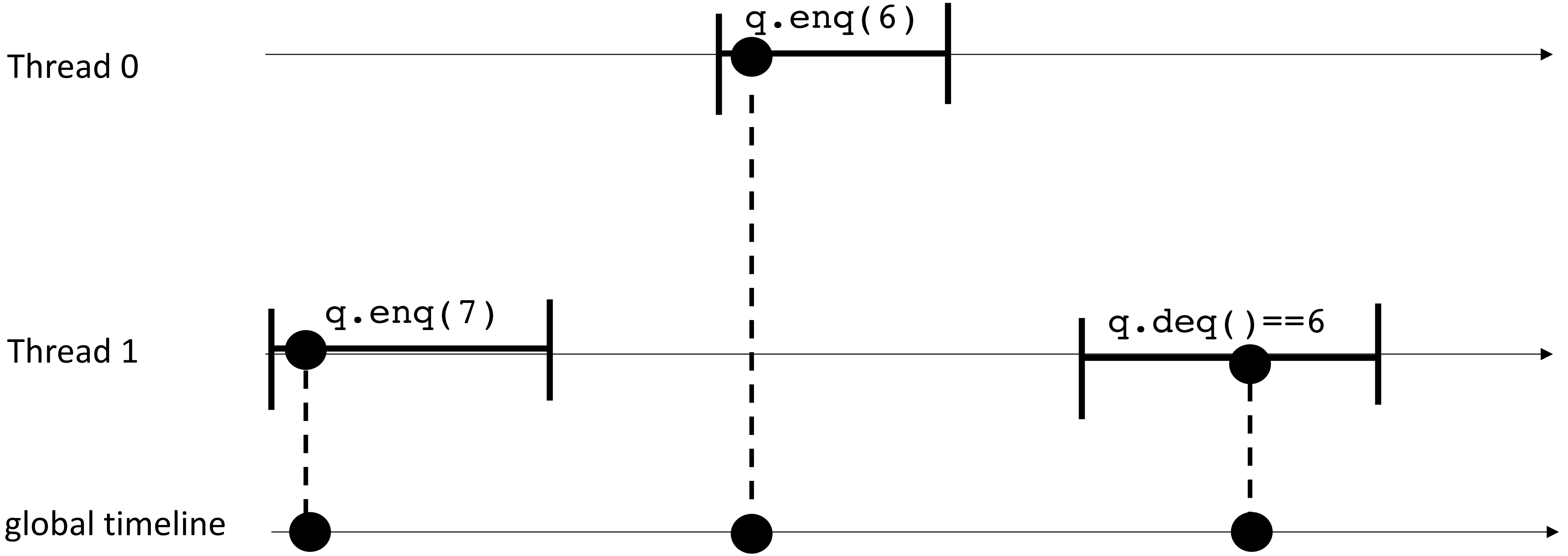


Linearizability

each command gets a linearization point.

You can place the point any where between its innovation and response!

Project the linearization points to a global timeline



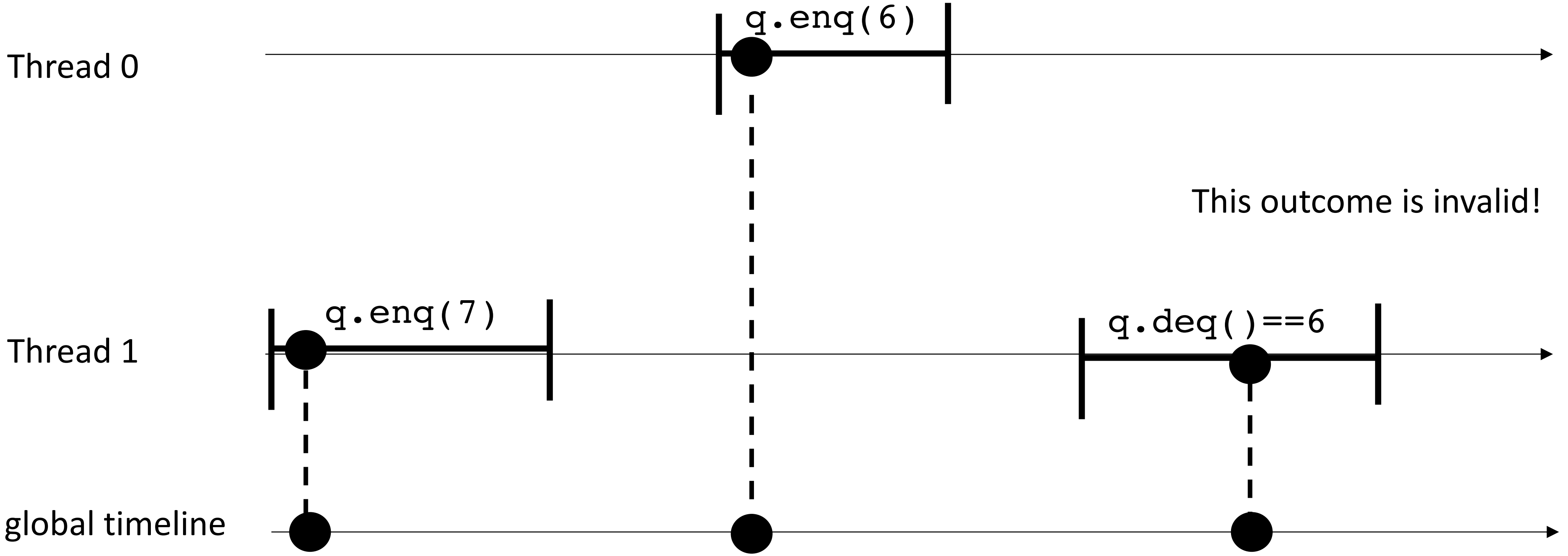
reason sequentially!

Linearizability

each command gets a linearization point.

You can place the point any where between its innovation and response!

Project the linearization points to a global timeline



This outcome is invalid!

reason sequentially!

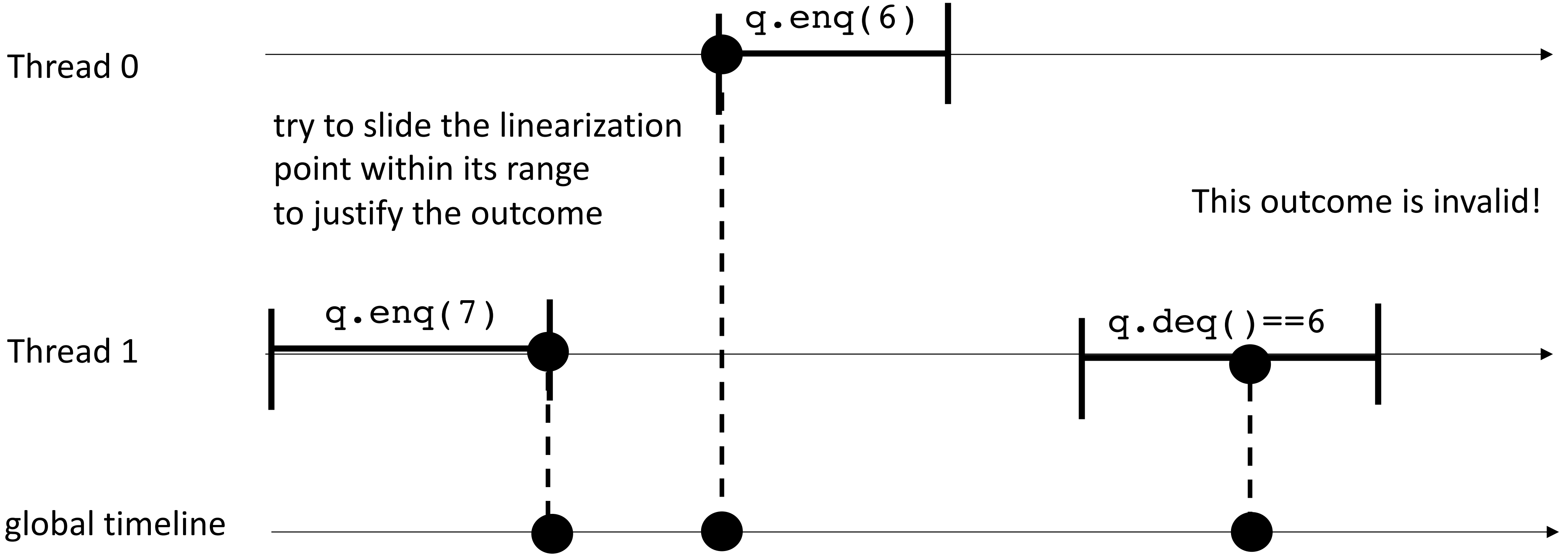
Linearizability

each command gets a linearization point.

You can place the point any where between its innovation and response!

Project the linearization points to a global timeline

slider game!

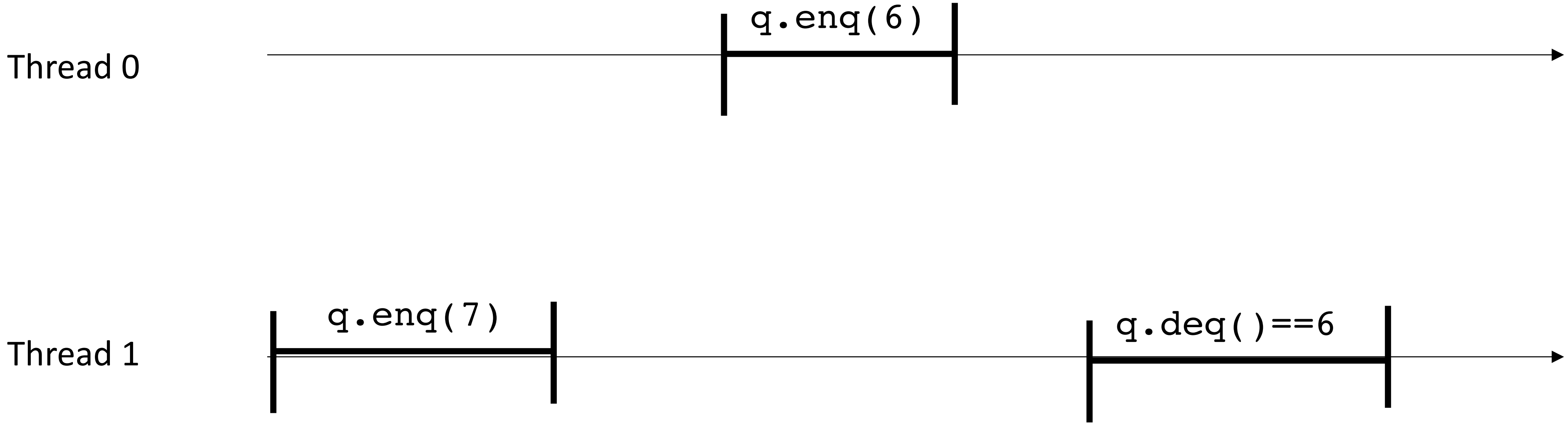


try to slide the linearization point within its range to justify the outcome

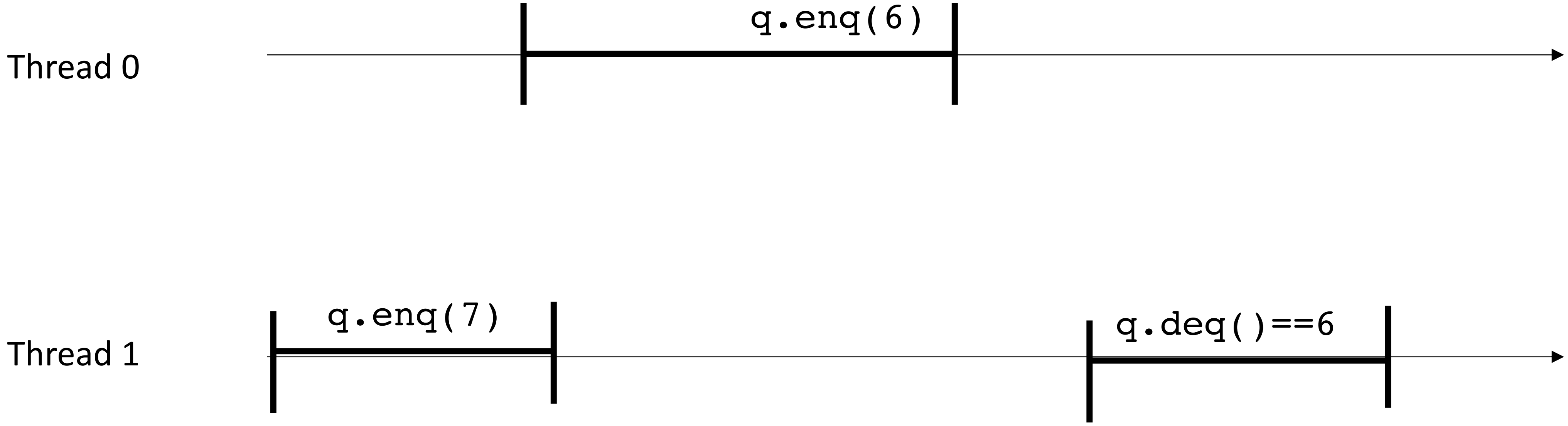
This outcome is invalid!

reason sequentially!

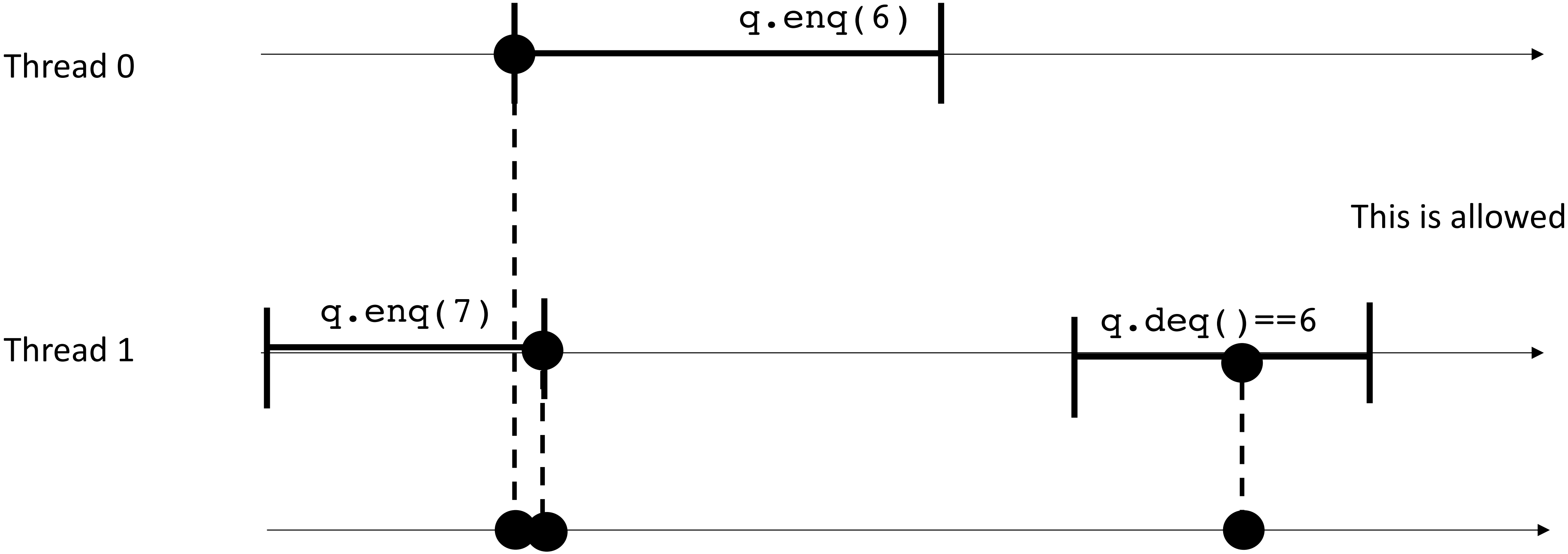
Linearizability



Linearizability



Linearizability



Thread 0

q.enq(6)

Thread 1

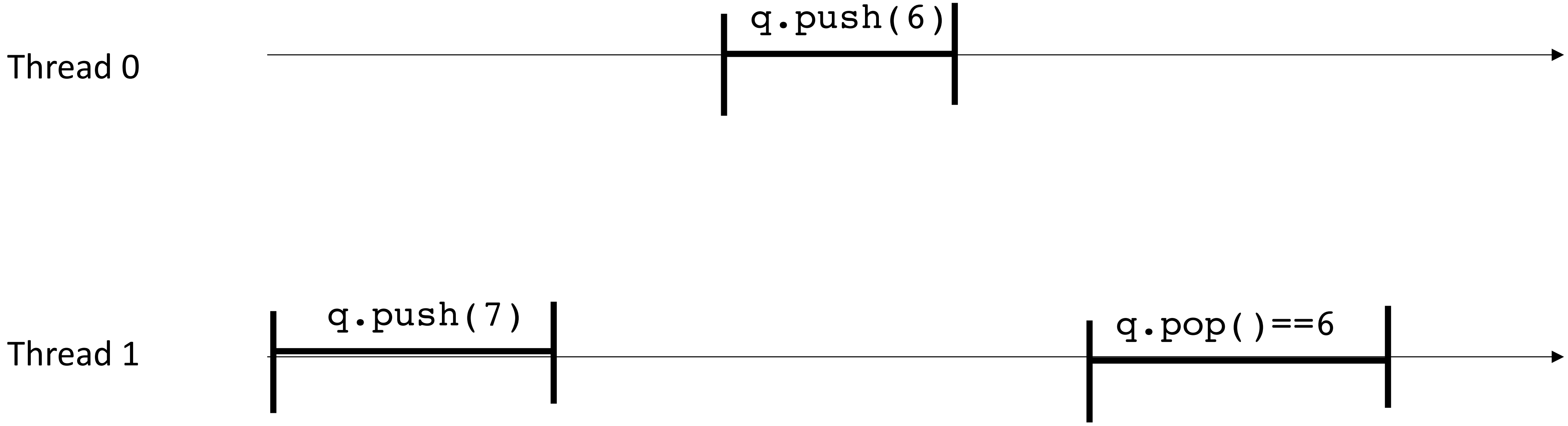
q.enq(7)

q.deq()==6

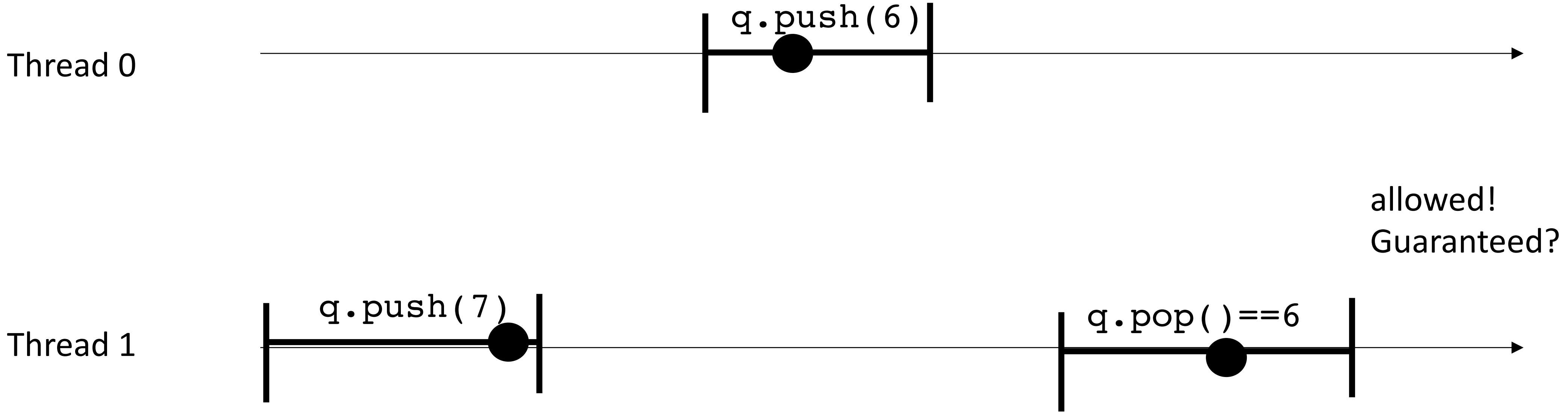
This is allowed now!

Linearizability

How about a stack?

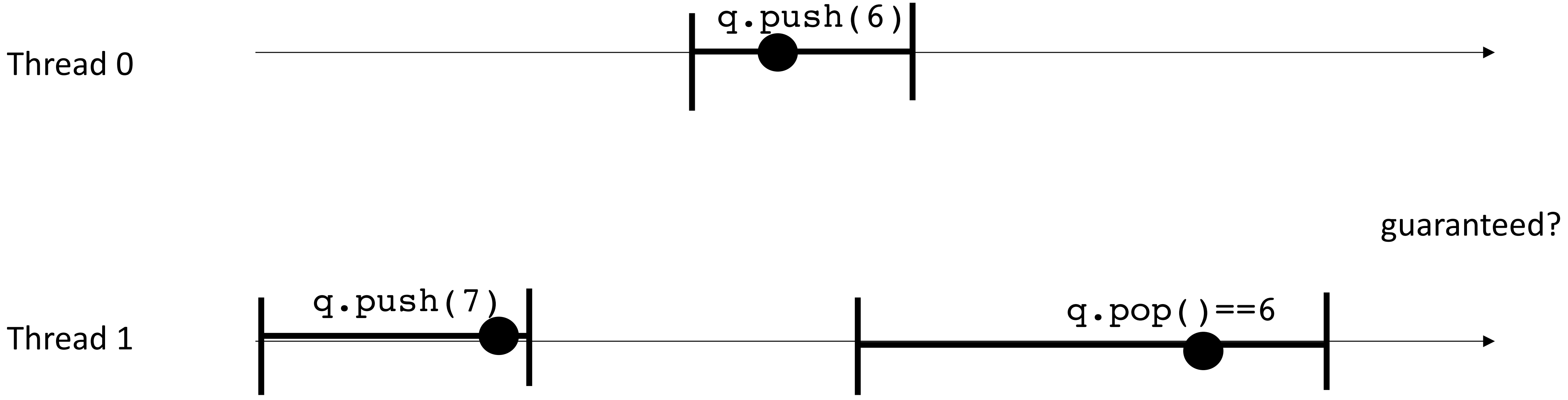


Linearizability

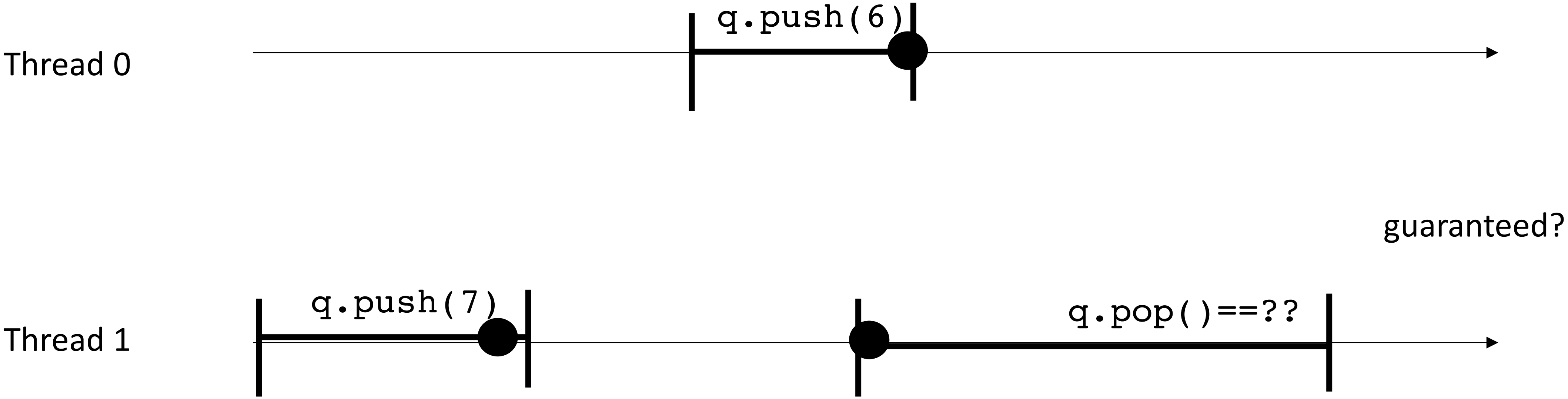


allowed!
Guaranteed?

Linearizability



Linearizability



Thread 0

q.push(6)

guaranteed?

Thread 1

q.push(7)

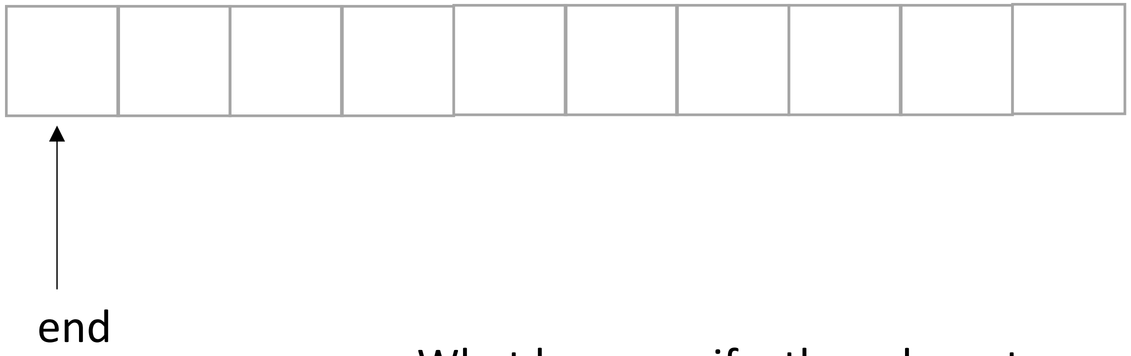
q.pop()==??

Input/Output Queues

Input/Output Queues

- Queue in which multiple threads read (deq), or write (enq), but not both.
- Why would we want a thing?
- Computation done in phases:
  - First phase prepares the queue (by writing into it)
  - All threads join
  - Second phase reads values from the queue.

Implementation

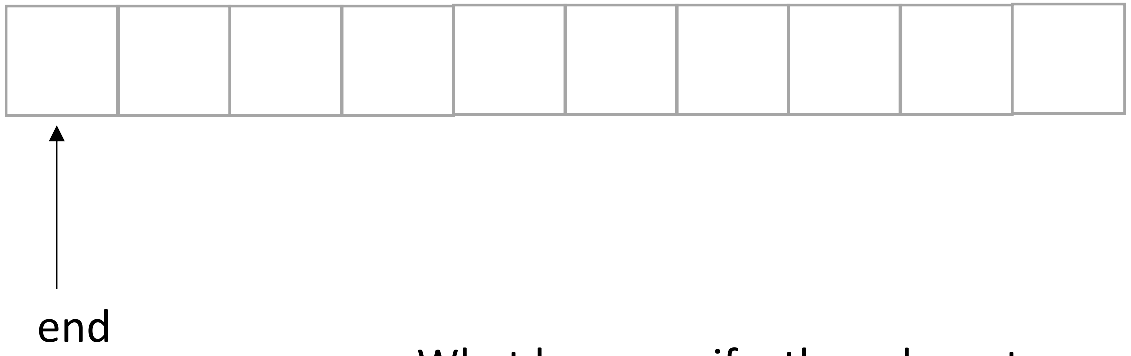


What happens if a thread wants to add an element?

Think concurrently:

```
reserved_index = atomic_fetch_add(&end, 1);
```

Implementation



Thread 0:
enq(6);

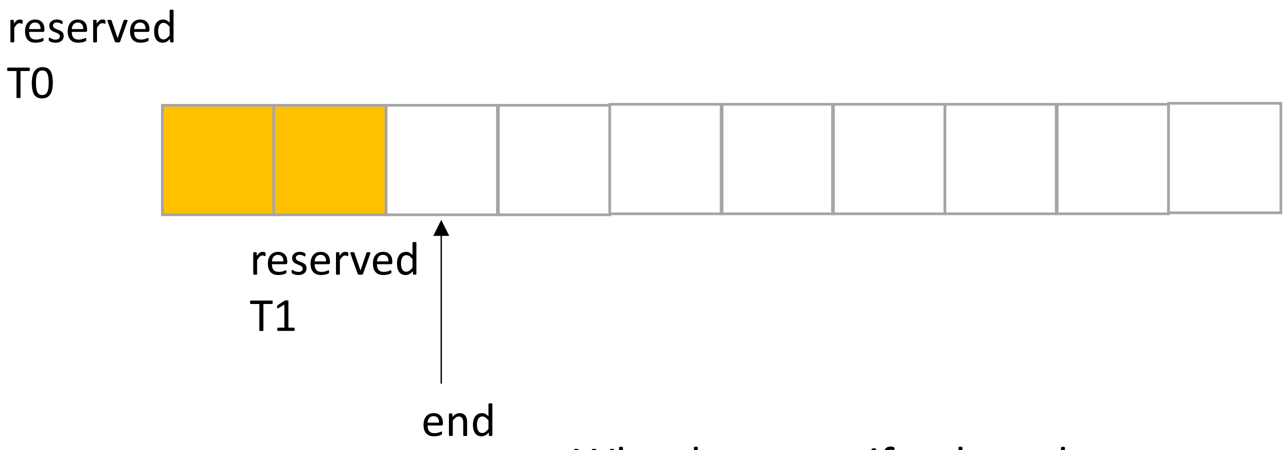
Thread 1:
enq(7);

What happens if a thread wants to add an element?

Think concurrently:

```
reserved_index = atomic_fetch_add(&end, 1);
```

Implementation



<u>Thread 0:</u>
enq(6);

<u>Thread 1:</u>
enq(7);

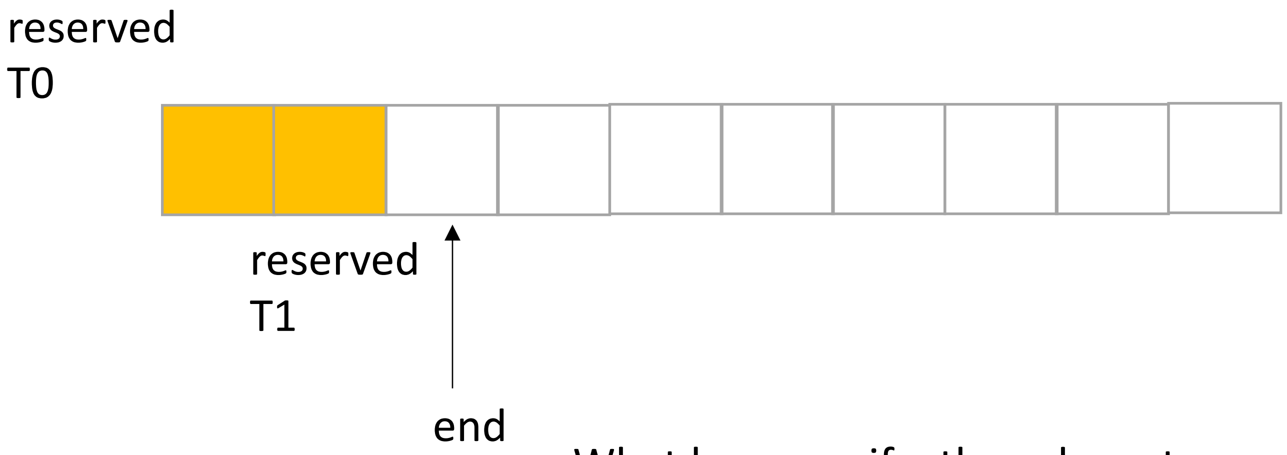
What happens if a thread wants to add an element?

Think concurrently:

```
reserved_index = atomic_fetch_add(&end, 1);
```

Implementation

does it matter which order threads add their data?



reserved
T0

reserved
T1

end

What happens if a thread wants to add an element?

Think concurrently:

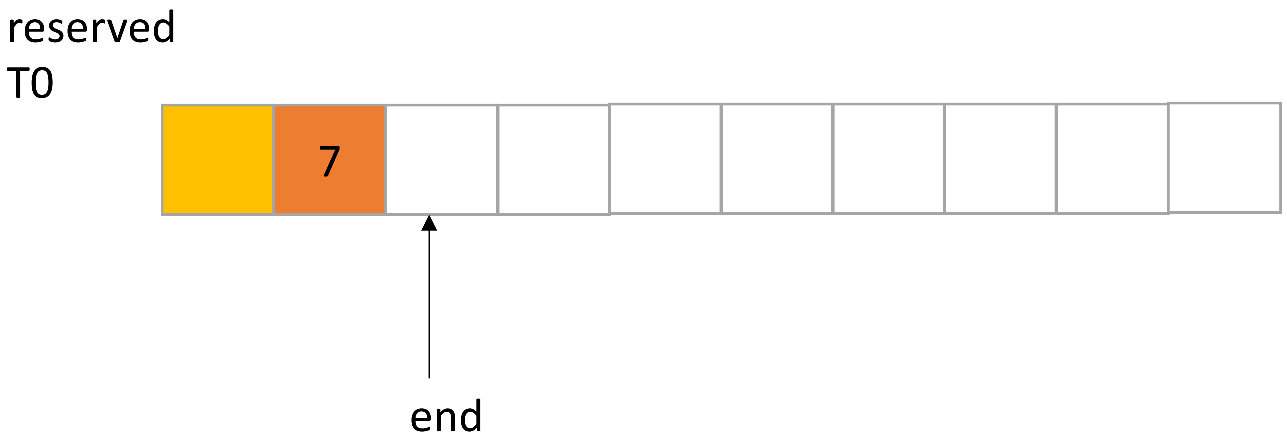
```
reserved_index = atomic_fetch_add(&end, 1);
```

<u>Thread 0:</u>
enq(6);

<u>Thread 1:</u>
enq(7);

Implementation

does it matter which order threads add their data?



Thread 0:
enq(6);

Thread 1:
enq(7);

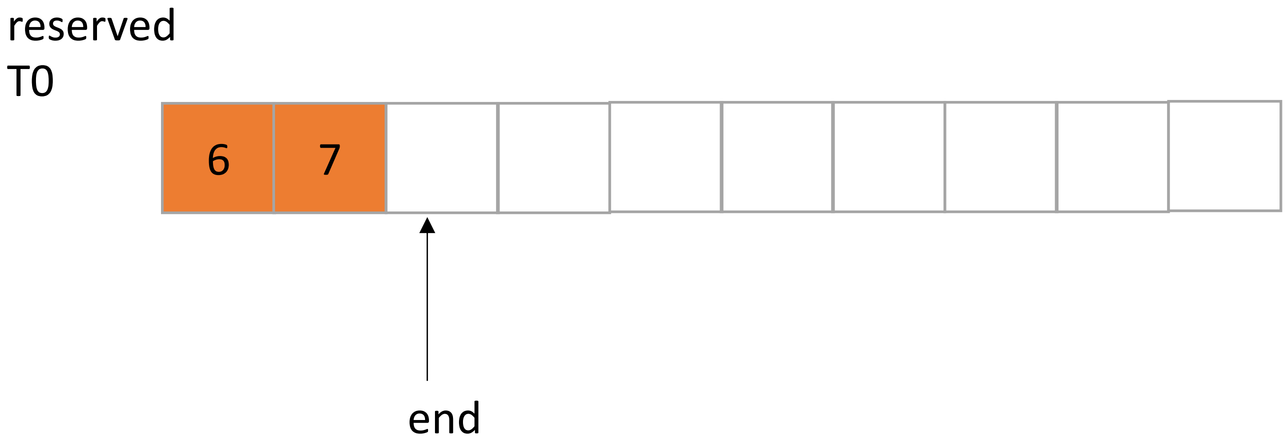
What happens if a thread wants to add an element?

Think concurrently:

```
reserved_index = atomic_fetch_add(&end, 1);
```

Implementation

does it matter which order threads add their data? No! Because there are no deqs!



<u>Thread 0:</u>
enq(6);

<u>Thread 1:</u>
enq(7);

What happens if a thread wants to add an element?

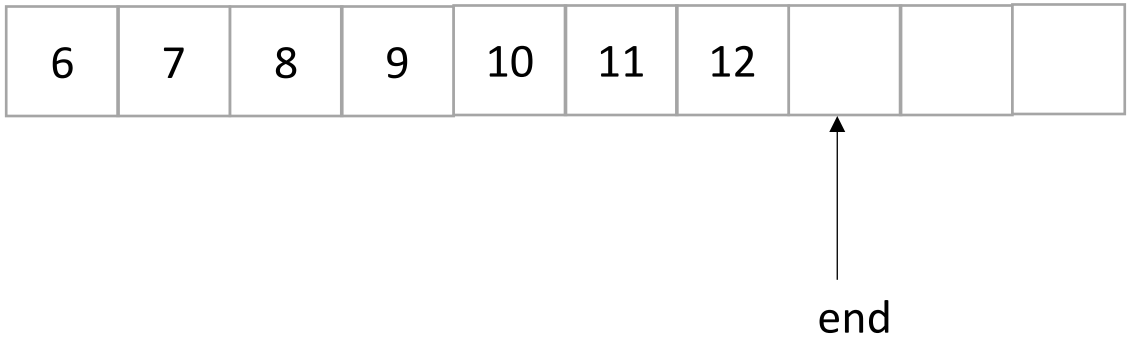
Think concurrently:

```
reserved_index = atomic_fetch_add(&end, 1);
```

Now enqueue

enq

- Now we only do deqs

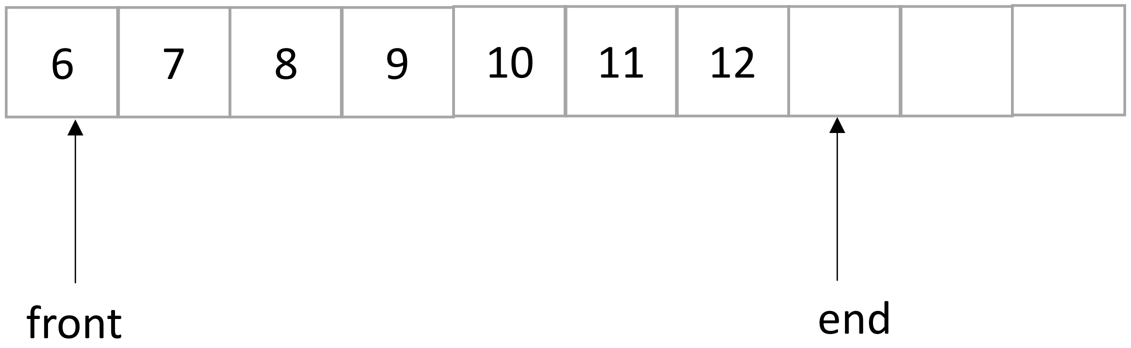


6	7	8	9	10	11	12			

end

enq

- Now we only do deqs



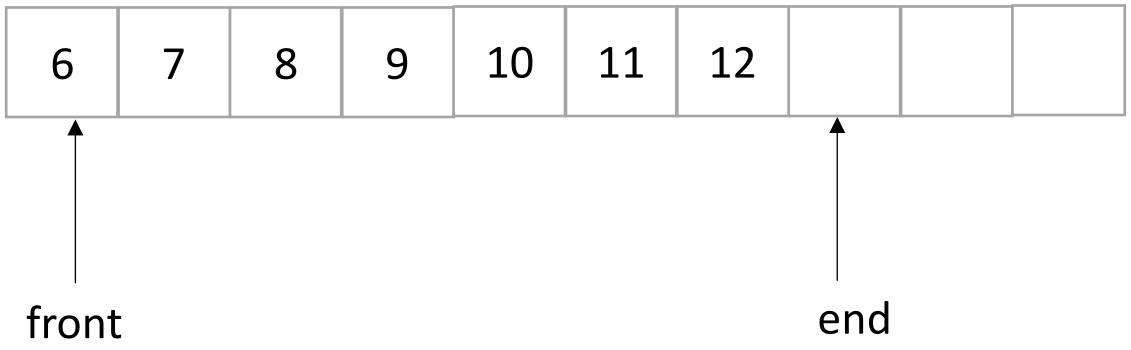
6	7	8	9	10	11	12			

front

end

enq

- Now we only do deqs



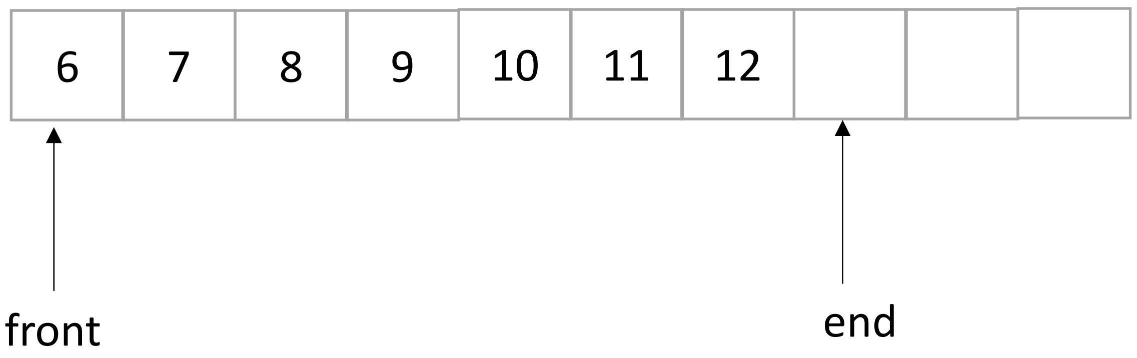
What happens if a thread wants to add an element?

Think concurrently:

```
data_index = atomic_fetch_add(&front, 1);
```

enq

- Now we only do deqs



Thread 0:
deq();

Thread 1:
deq();

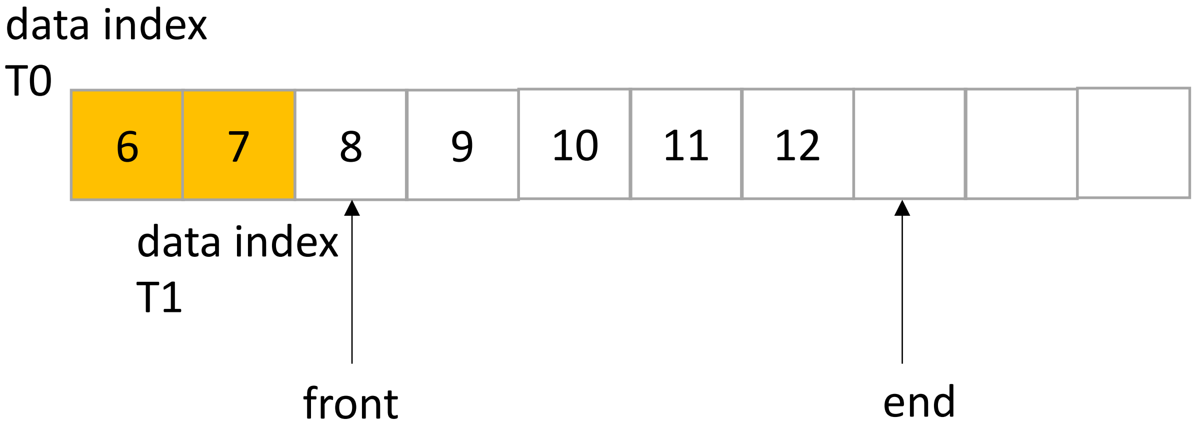
What happens if a thread wants to add an element?

Think concurrently:

```
data_index = atomic_fetch_add(&front, 1);
```

enq

- Now we only do deqs



data index T0

6 7 8 9 10 11 12

data index T1

front

end

Thread 0:
deq();

Thread 1:
deq();

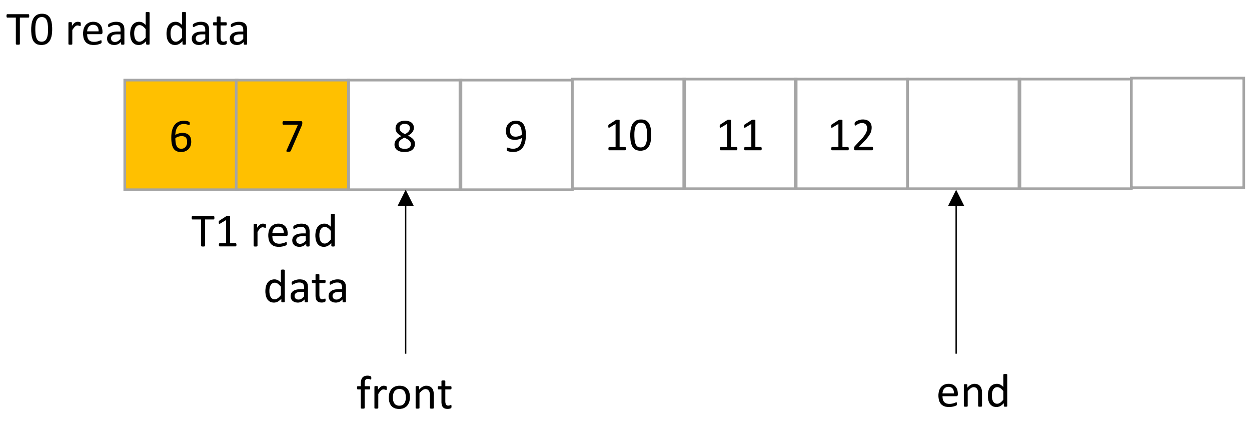
What happens if a thread wants to add an element?

Think concurrently:

```
data_index = atomic_fetch_add(&front, 1);
```

enq

- Now we only do deqs



T0 read data

6	7	8	9	10	11	12			

T1 read data

front

end

Thread 0:

deq(); // reads 6

Thread 1:

deq(); // reads 7

What happens if a thread wants to add an element?

Think concurrently:

```
data_index = atomic_fetch_add(&front, 1);
```

```
class InputOutputQueue {
  private:
    atomic_int front;
    atomic_int end;
    int list[SIZE];

  public:
    InputOutputQueue() {
        front = end = 0;
     }

     void enq(int x) {
        int reserved_index = atomic_fetch_add(&end, 1);
        list[reserved_index] = x;
     }

     void deq() {
       int reserved_index = atomic_fetch_add(&front, 1);
       return list[reserved_index];
     }

     int size() {
        return end.load() - front.load();
     }
 }
```

```
class InputOutputQueue {
  private:
    atomic_int front;
    atomic_int end;
    int list[SIZE];

  public:
    InputOutputQueue() {
        front = end = 0;
     }

     void enq(int x) {
        int reserved_index = atomic_fetch_add(&end, 1);
        list[reserved_index] = x;
     }

     void deq() {
       int reserved_index = atomic_fetch_add(&front, 1);
       return list[reserved_index];
     }

     int size() {
        return end.load() - front.load();
     }
 }
```

Does list need to be atomic?

How to make sure the queue has an element in it before you dequeue?

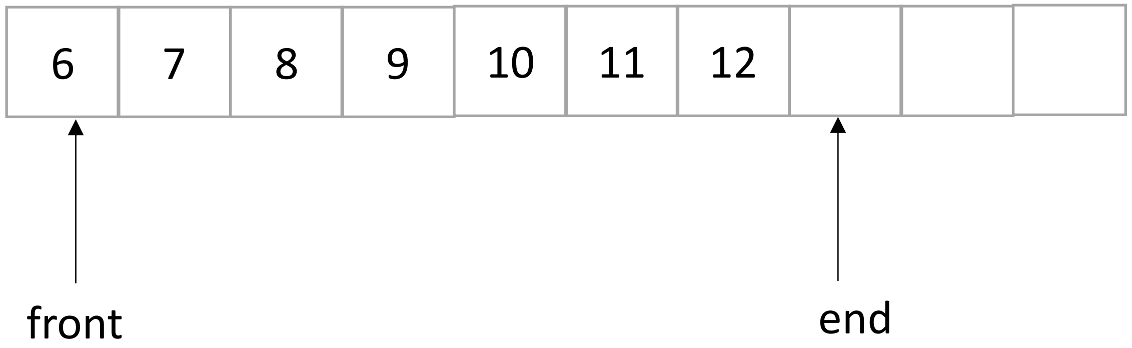
What can go wrong if we deq and enq?

<u>Thread 0:</u>
enq(1);

<u>Thread 1:</u>
deq();

```
void enq(int x) {
        int reserved_index = atomic_fetch_add(&end, 1);
        list[reserved_index] = x;
}
```

```
void deq() {
       int reserved_index = atomic_fetch_add(&front, 1);
       return list[reserved_index];
}
```



What can go wrong if we deq and enq?

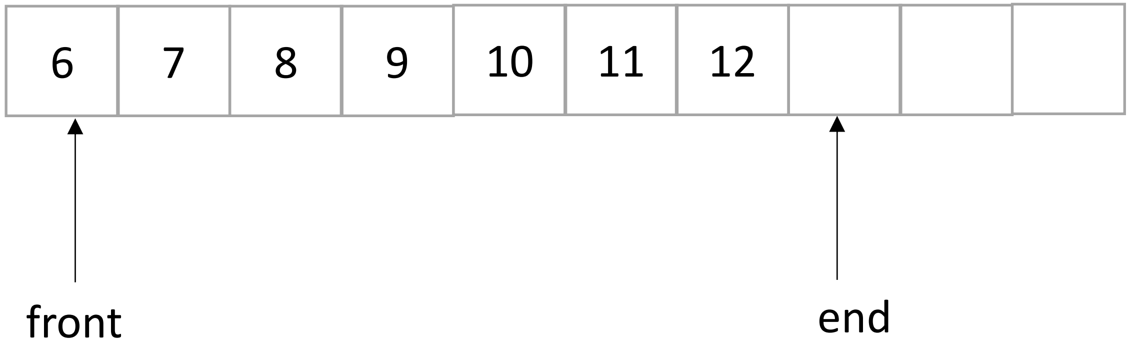
<u>Thread 0:</u>
enq(1);

<u>Thread 1:</u>
deq();

```
void enq(int x) {
        int reserved_index = atomic_fetch_add(&end, 1);
        list[reserved_index] = x;
}
```

```
void deq() {
       int reserved_index = atomic_fetch_add(&front, 1);
       return list[reserved_index];
}
```

Nothing!
Seems to work
if the queue is like
this


6 7 8 9 10 11 12
front
end

but we need to
think about corner cases

What can go wrong if we deq and enq?

Thread 0:
enq(1);

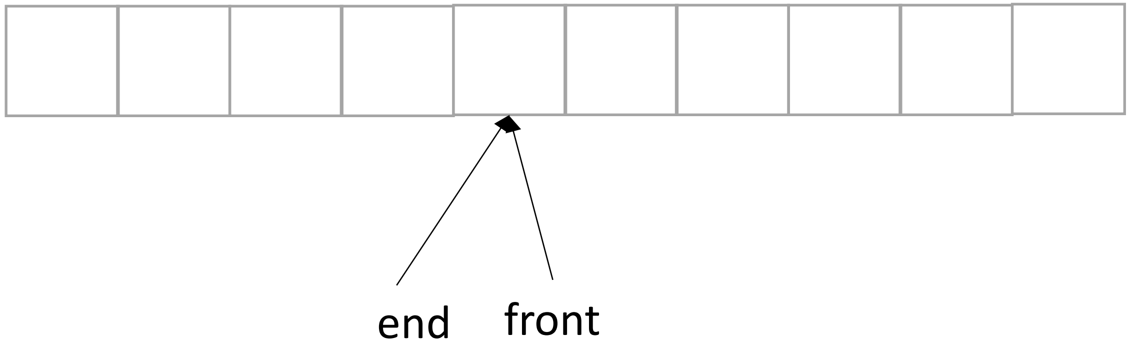
Thread 1:
deq();

```
void enq(int x) {
        int reserved_index = atomic_fetch_add(&end, 1);
        list[reserved_index] = x;
}
```

```
void deq() {
       int reserved_index = atomic_fetch_add(&front, 1);
       return list[reserved_index];
}
```

Nothing!
Seems to work
if the queue is like
this

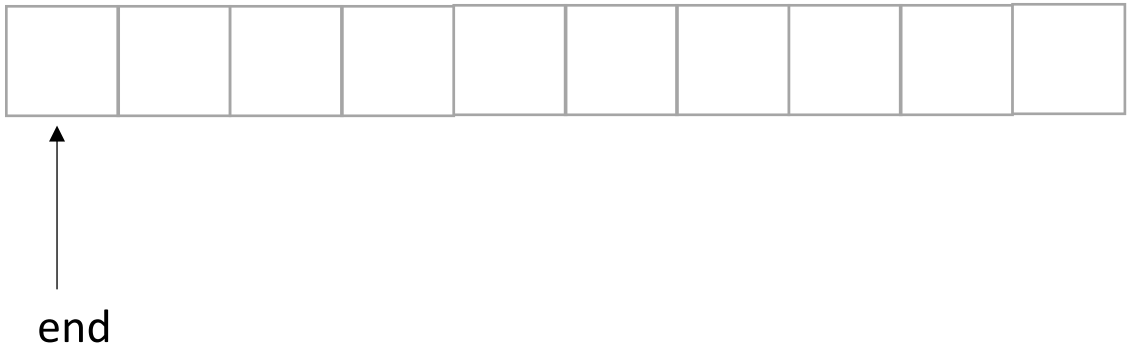
but we need to
think about corner cases



Blocking?

- Does the input/output queue block?

Implementation

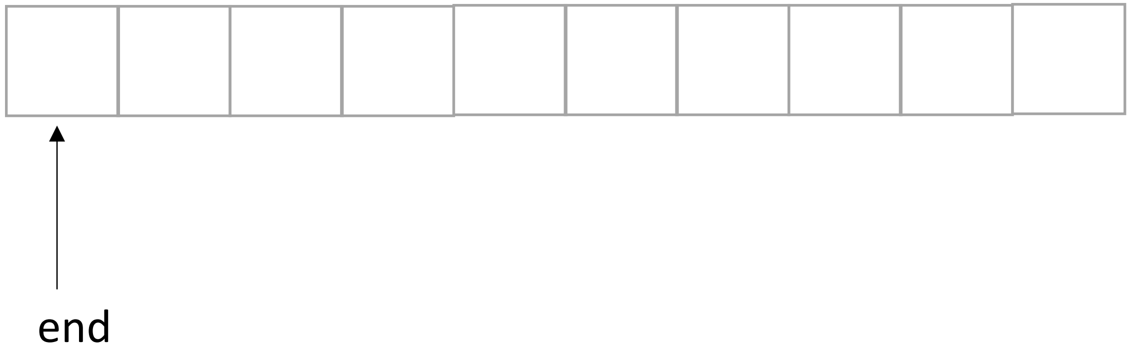


Thread 0:
enq(6);

Thread 1:
enq(7);

```
void enq(int x) {
        int reserved_index = atomic_fetch_add(&end, 1);
        list[reserved_index] = x;
}
```

Implementation

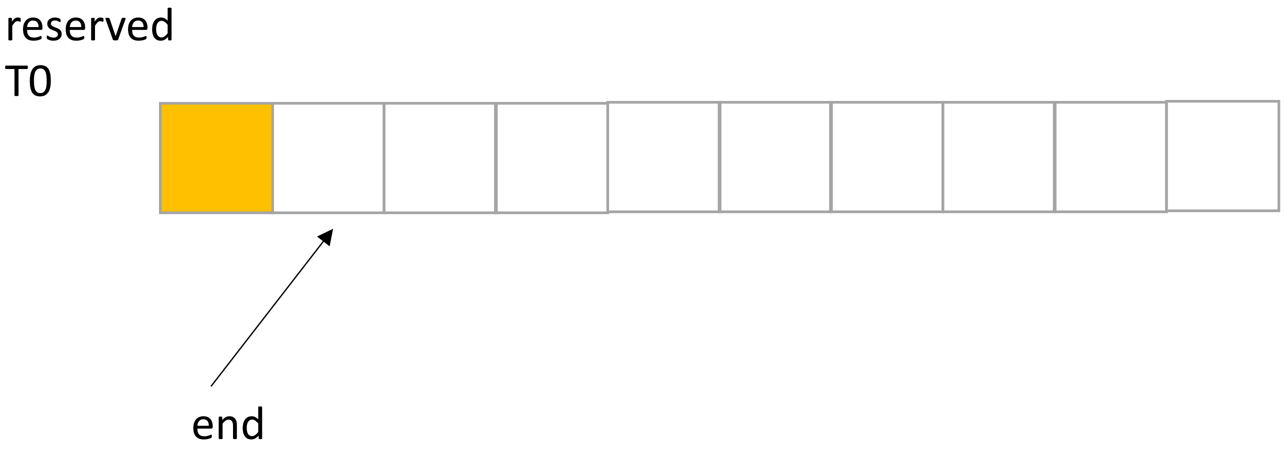


Thread 0:
enq(6);

Thread 1:
enq(7);

```
void enq(int x) {
        int reserved_index = atomic_fetch_add(&end, 1);
        list[reserved_index] = x;
}
```

Implementation

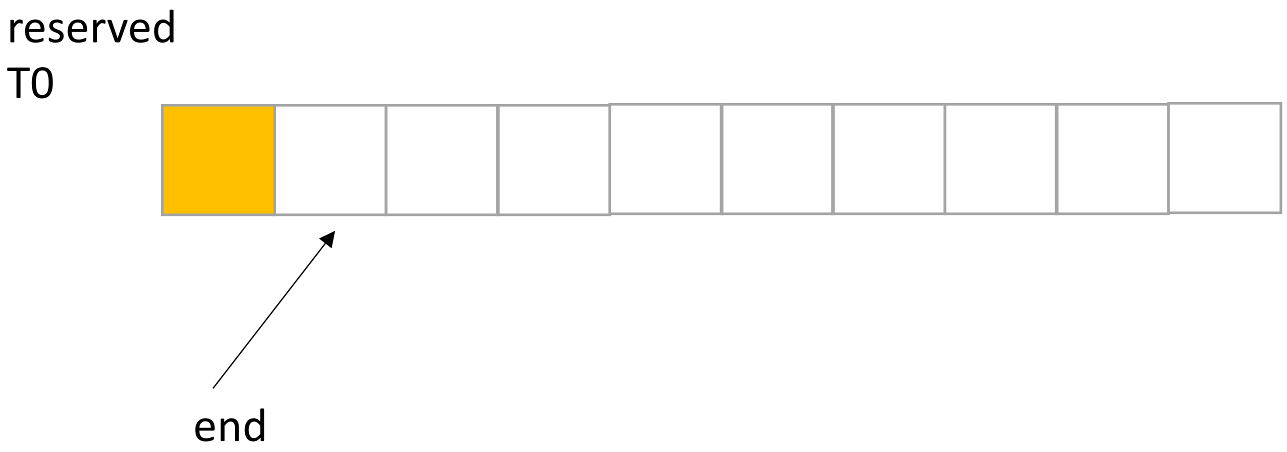


Thread 0:
enq(6);

Thread 1:
enq(7);

```
void enq(int x) {
        int reserved_index = atomic_fetch_add(&end, 1);
        list[reserved_index] = x;
}
```

Implementation



reserved
T0

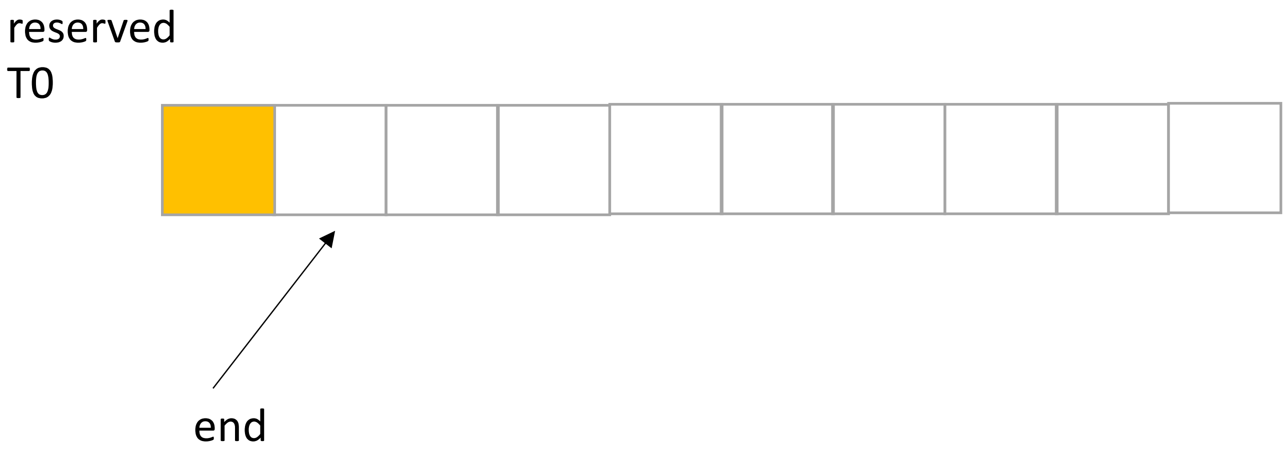
end

Thread 0:
enq(6);

Thread 1:
enq(7);

```
void enq(int x) {
        int reserved_index = atomic_fetch_add(&end, 1);
        list[reserved_index] = x;
}
```

Implementation

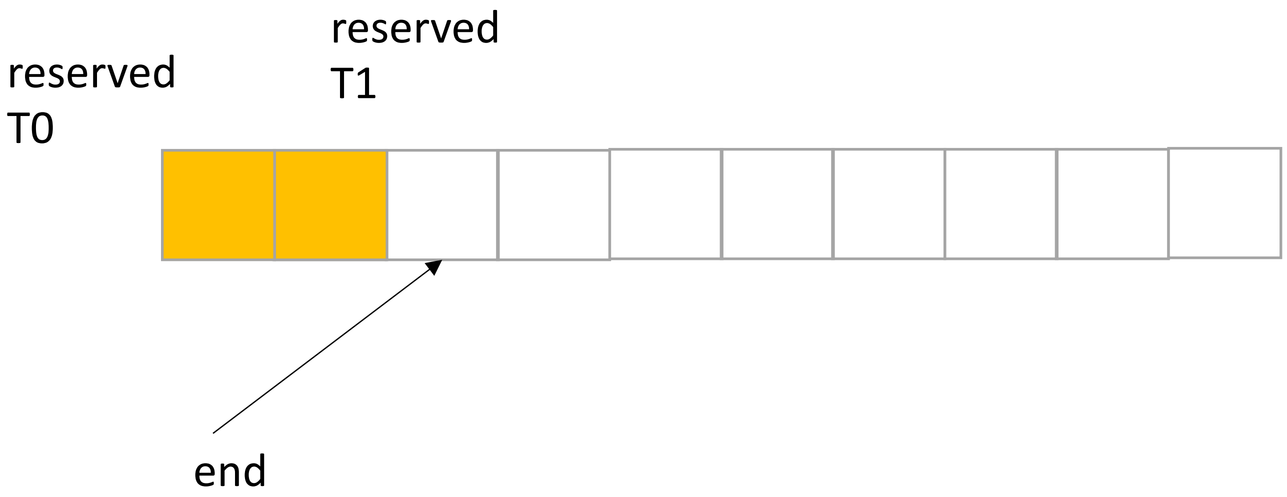


Thread 0:
enq(6);

Thread 1:
enq(7);

```
void enq(int x) {
        int reserved_index = atomic_fetch_add(&end, 1);
        list[reserved_index] = x;
}
```

Implementation

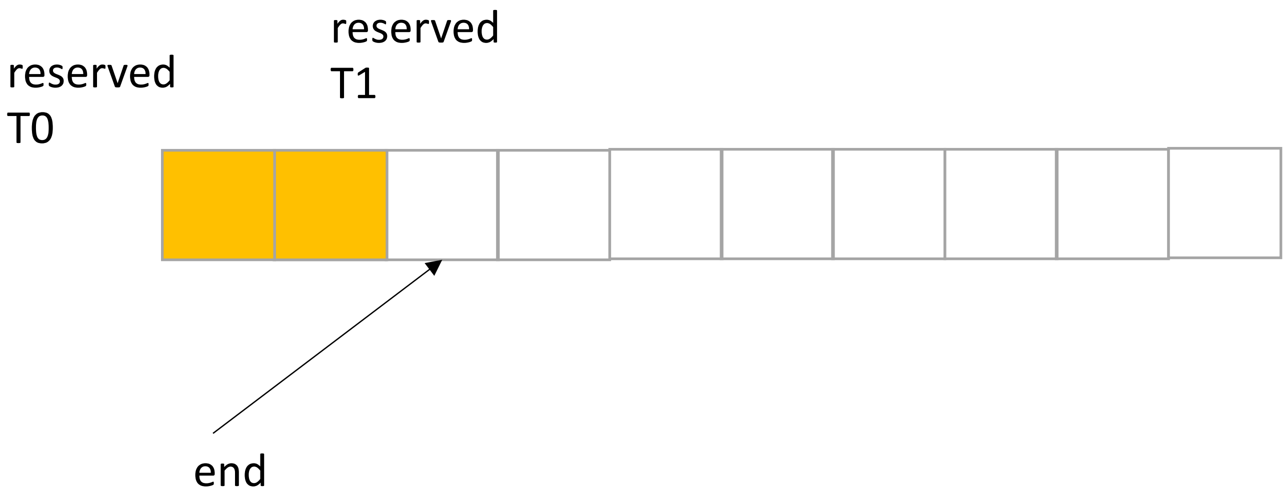


Thread 0:
enq(6);

Thread 1:
enq(7);

```
void enq(int x) {
        int reserved_index = atomic_fetch_add(&end, 1);
        list[reserved_index] = x;
}
```

Implementation



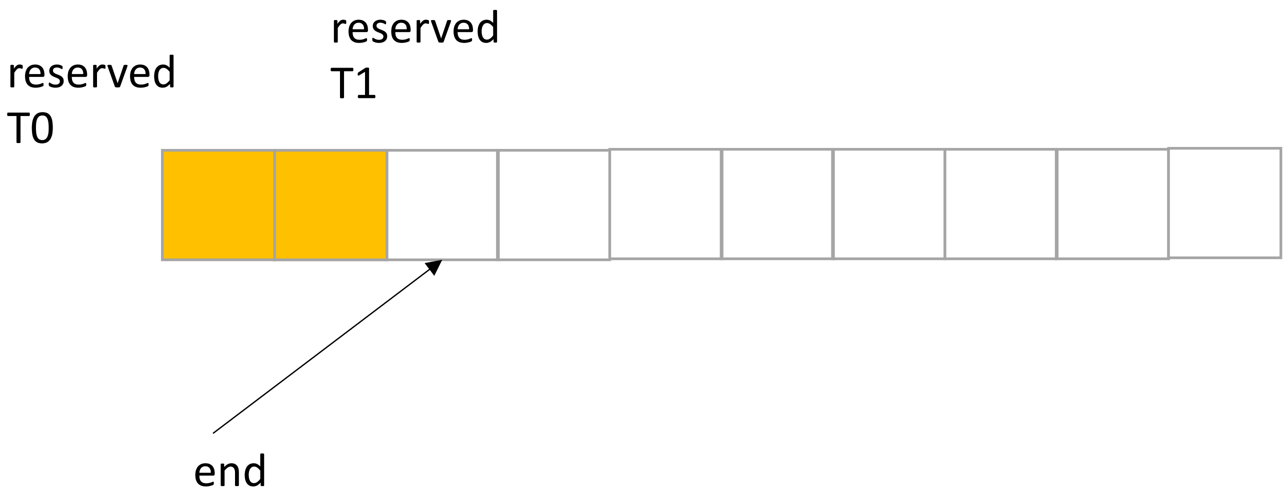
<u>Thread 0:</u>
enq(6);

<u>Thread 1:</u>
enq(7);

```
void enq(int x) {
        int reserved_index = atomic_fetch_add(&end, 1);
        list[reserved_index] = x;
}
```

Both threads need to execute this instruction. What happens if one is delayed?

Implementation



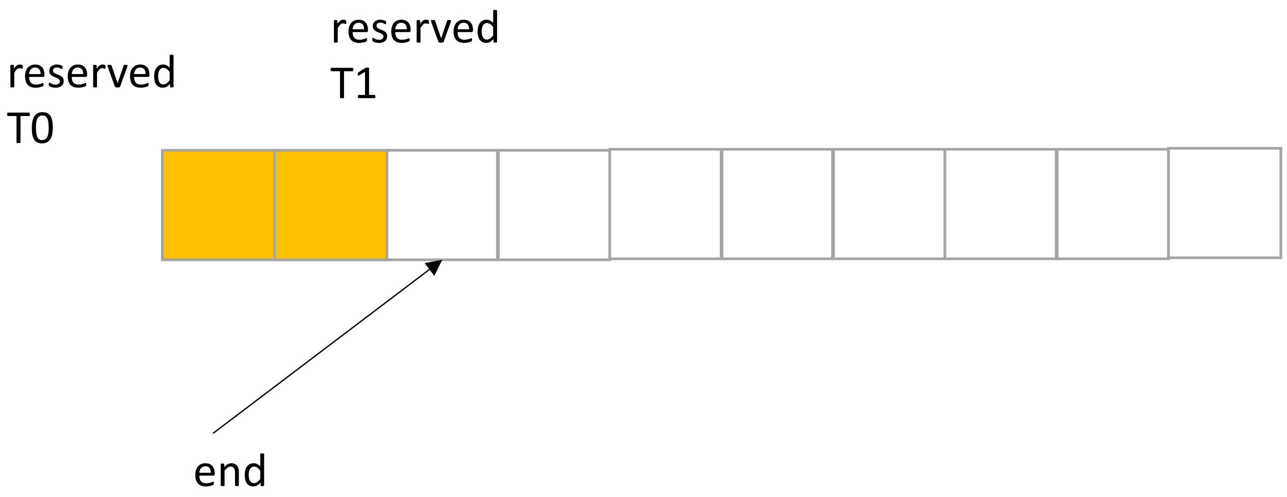
Thread 0:
enq(6);

Thread 1:
enq(7);

```
void enq(int x) {
        int reserved_index = atomic_fetch_add(&end, 1);
        list[reserved_index] = x;
}
```

Both threads need to execute this instruction. What happens if one is delayed?
It doesn't matter! The other thread can still keep going!

Implementation



Thread 0:
enq(6);

Thread 1:
enq(7);

```
void enq(int x) {
        int reserved_index = atomic_fetch_add(&end, 1);
        list[reserved_index] = x;
}
```

Both threads need to execute this instruction. What happens if one is delayed?
It doesn't matter! The other thread can still keep going!

```
class InputOutputQueue {
  private:
    atomic_int front;
    atomic_int end;
    int list[SIZE];

  public:
    InputOutputQueue() {
        front = end = 0;
     }

     void enq(int x) {
        int reserved_index = atomic_fetch_add(&end, 1);
        list[reserved_index] = x;
     }

     int deq() {
       int reserved_index = atomic_fetch_add(&front, 1);
       return list[reserved_index];
     }

     int size() {
        return end.load() - front.load();
     }
 }
```

Could we implement a blocking version of this queue?

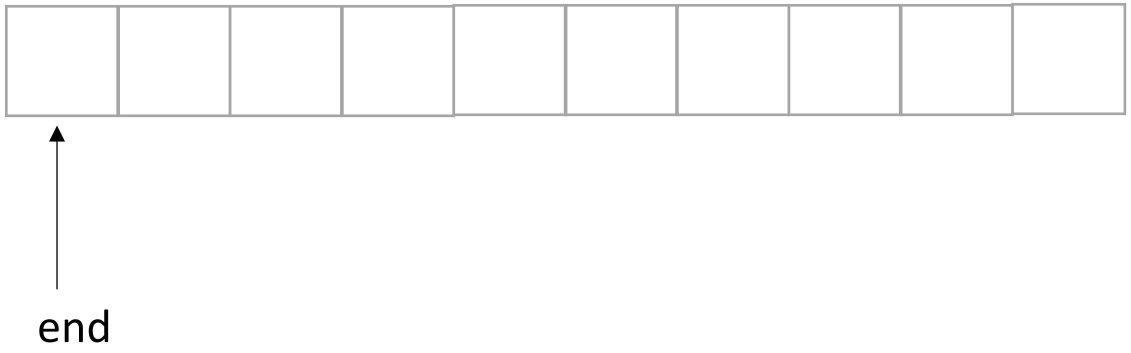
```
class InputOutputQueue {
  private:
    int front;
    int end;
    int list[SIZE];
    mutex m;

  public:
     void enq(int x) {
        m.lock();
        list[end] = x;
        end++;
        m.unlock();
     }

     int deq() {
        m.lock();
        int tmp = list[front];
        front++;
        m.unlock();
        return tmp;
     }
}
```

Could we implement a blocking version of this queue?

Just add a mutex!

What are the pros and cons?

Implementation

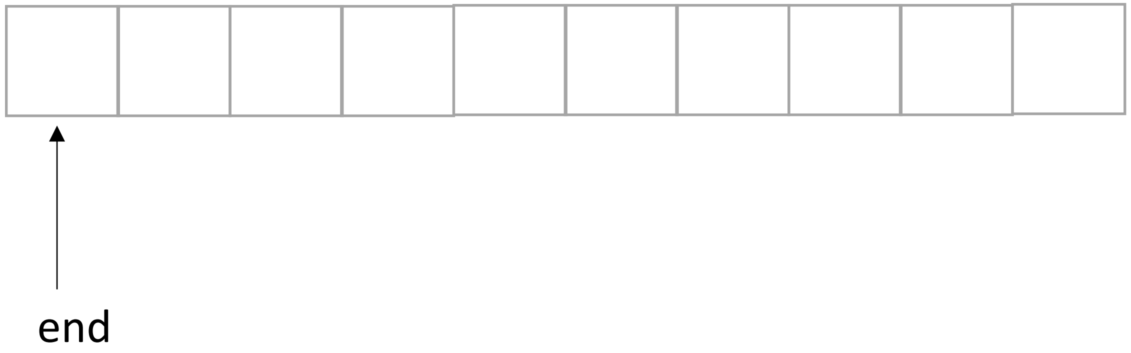


<u>Thread 0:</u>
enq(6);

<u>Thread 1:</u>
enq(7);

```
void enq(int x) {
        m.lock();
        end++;
        list[end] = x;
        m.unlock();
    }
```

Implementation



Thread 0:

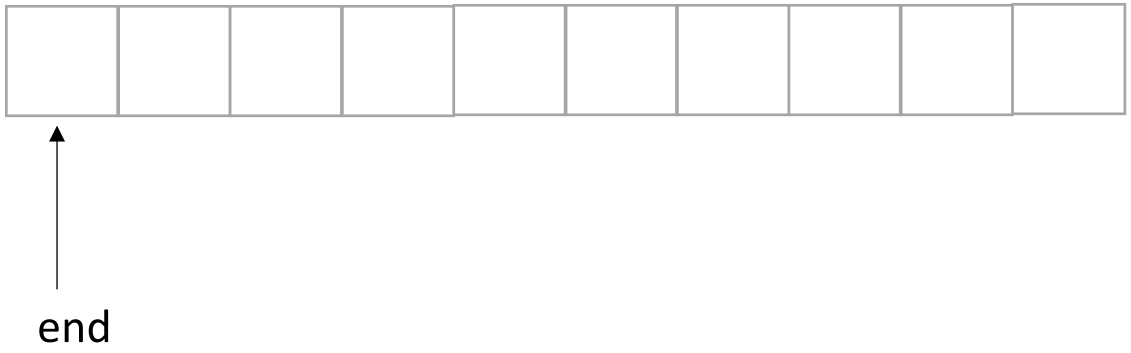
enq(6);

Thread 1:

enq(7);

```
void enq(int x) {
        m.lock();
        end++;
        list[end] = x;
        m.unlock();
    }
```

Implementation

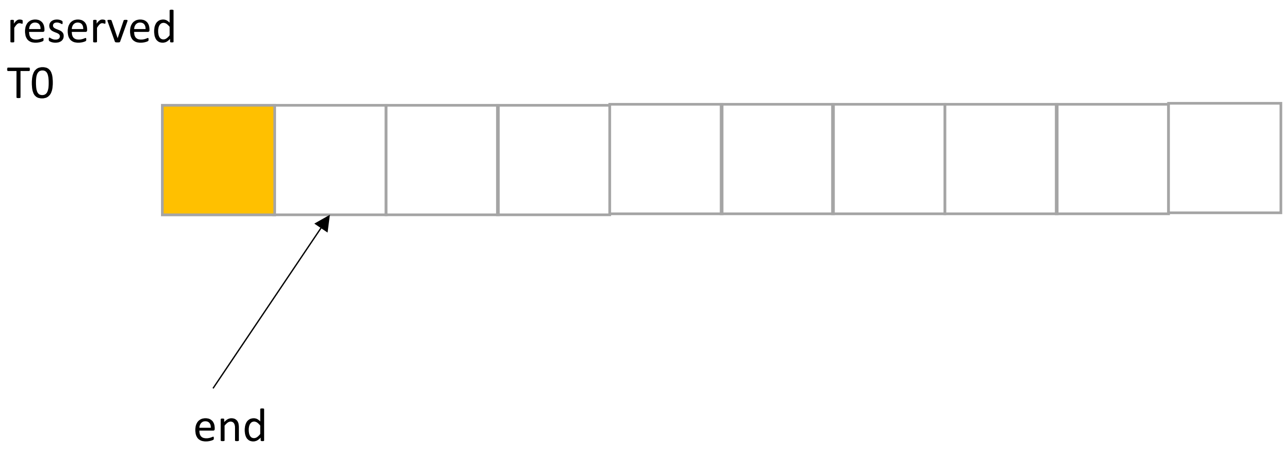


Thread 0:
enq(6);

Thread 1:
enq(7);

```
void enq(int x) {
      m.lock();
      end++;
      list[end] = x;
      m.unlock();
   }
```

Implementation



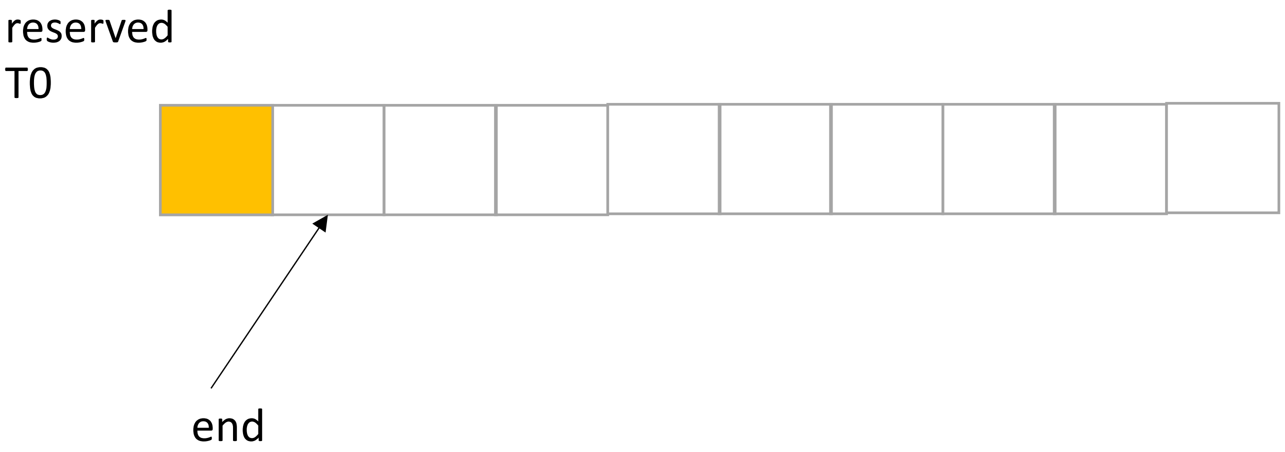
Thread 0:
enq(6);

say this thread is delayed

Thread 1:
enq(7);

```
void enq(int x) {
      m.lock();
      end++;
      list[end] = x;
      m.unlock();
   }
```

Implementation



Thread 0:
enq(6);

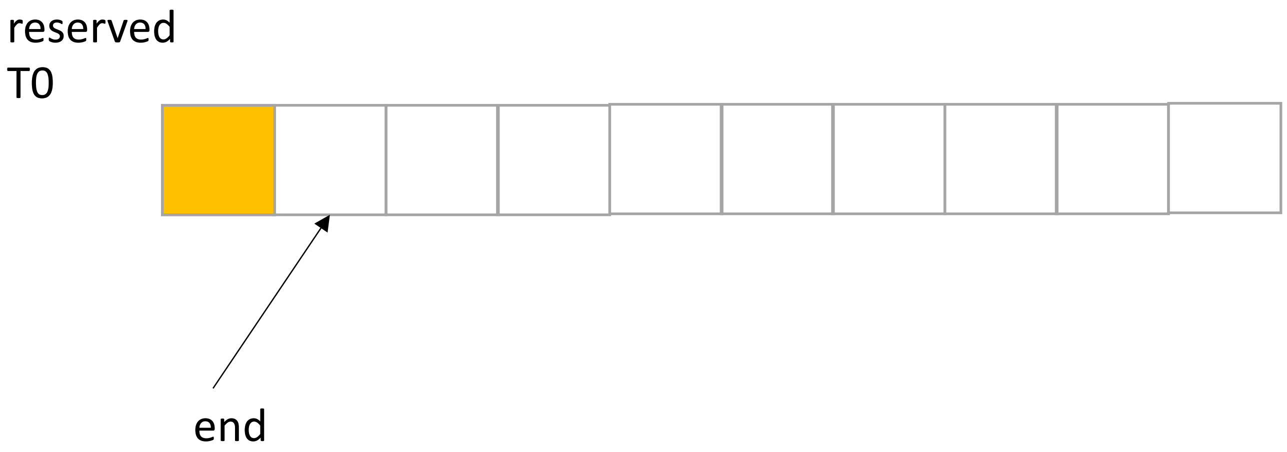
say this thread is delayed now

Thread 1:
enq(7);

Can thread 1 make progress?

```
void enq(int x) {
        m.lock();
        end++;
        list[end] = x;
        m.unlock();
    }
```

Implementation



reserved
T0

end

Thread 0:
enq(6);

say this thread is delayed now

Thread 1:
enq(7);

Can thread 1
make progress?

```
void enq(int x) {
        m.lock();
        end++;
        list[end] = x;
        m.unlock();
    }
```

This implementation is blocking!

On to new material!

Schedule

- Producer Consumer queues
  - **Synchronous**
  - Circular buffer

Producer Consumer Queues

- 1 enq, 1 deq
  - enq’er cannot deq
  - deq’er cannot enq

- Example: printf:
  - your program equeues values to print
  - the terminal process dequeues values and prints them

Synchronous Producer Consumer Queues

- First implementation:
  - Synchronous
  - Slow
  - Good for debugging

Synchronous Producer Consumer Queues

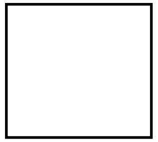
- First implementation:
    - Synchronous
    - Slow
    - Good for debugging

- enq does not return until value is deq’ed

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
```



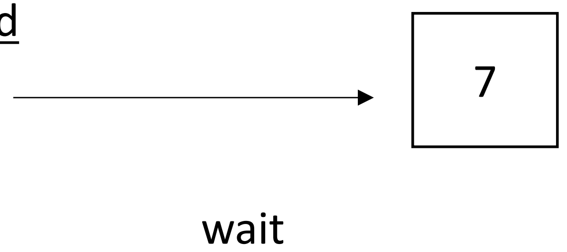
Consumer Thread

```
deq();
```

Synchronous Producer Consumer Queues

Producer Thread

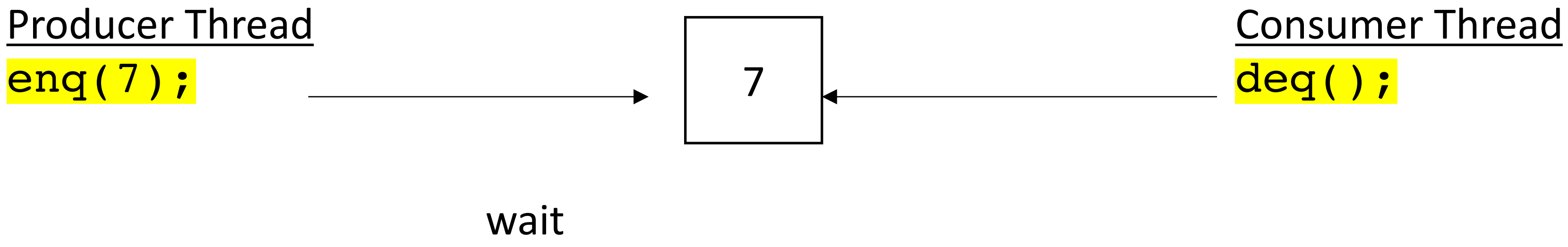
```
enq(7);
```



Consumer Thread

```
deq();
```

Synchronous Producer Consumer Queues



Producer Thread

`enq(7);`

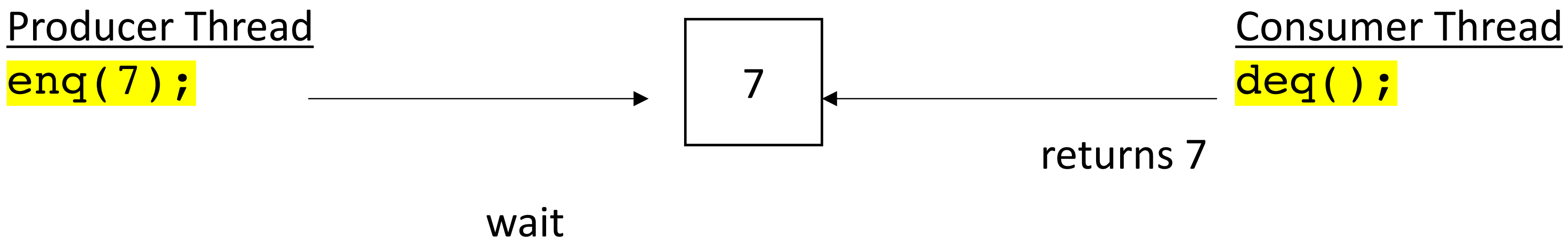
7

Consumer Thread

`deq();`

wait

Synchronous Producer Consumer Queues



Producer Thread

`enq(7);`

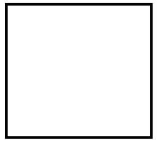
Consumer Thread

`deq();`

Synchronous Producer Consumer Queues

<u>Producer Thread</u>

```
enq(7);
```



<u>Consumer Thread</u>

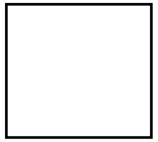
```
deq();
```

both can continue

Synchronous Producer Consumer Queues

Producer Thread

```
sleep();
enq(7);
```



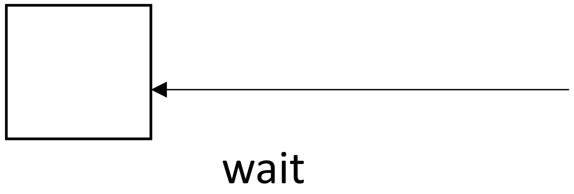
Consumer Thread

```
deq();
```

Synchronous Producer Consumer Queues

Producer Thread

```
sleep();
enq(7);
```



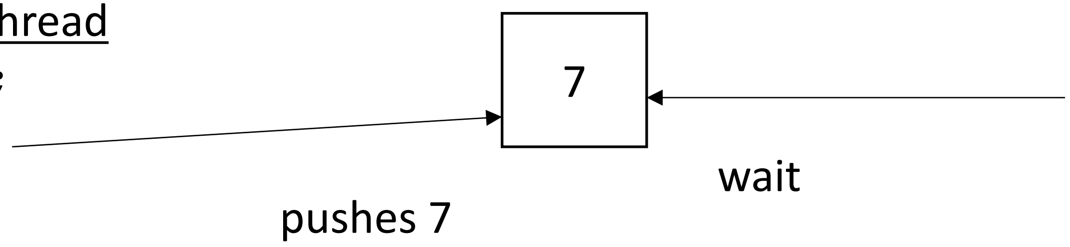
Consumer Thread

```
deq();
```

Synchronous Producer Consumer Queues

Producer Thread

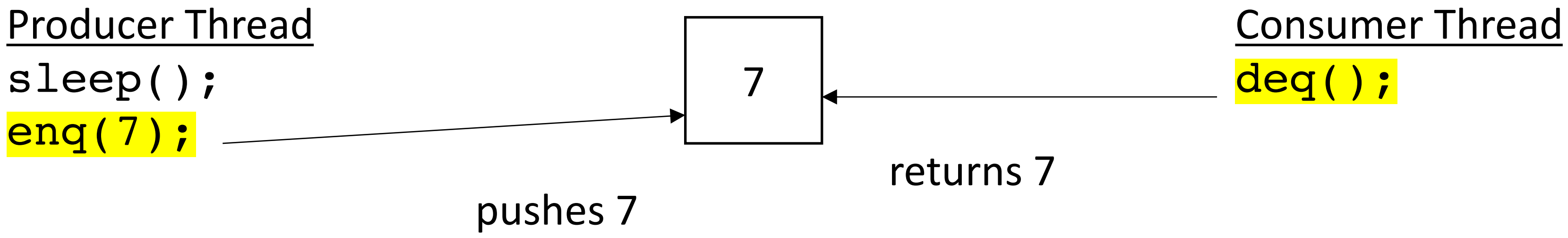
```
sleep();
enq(7);
```



Consumer Thread

```
deq();
```

Synchronous Producer Consumer Queues



Producer Thread

```
sleep();
enq(7);
```

7

Consumer Thread

```
deq();
```

pushes 7

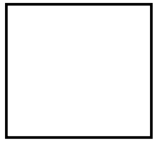
returns 7

They both can continue

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
```



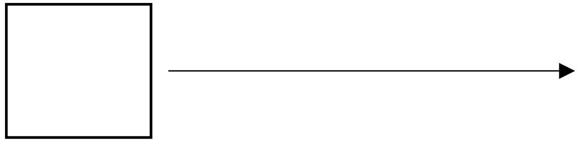
Consumer Thread

```
deq();
```

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
```

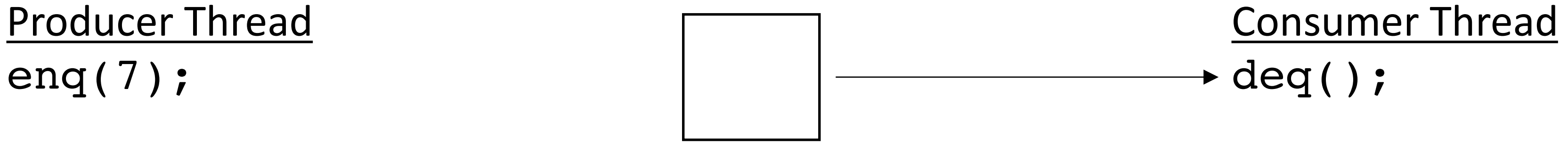


Consumer Thread

```
deq();
```

can the consumer just read?

Synchronous Producer Consumer Queues



Producer Thread

```
enq(7);
```

Consumer Thread

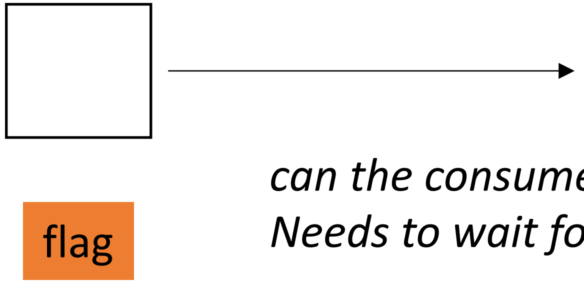
```
deq();
```

can the consumer just read?
Needs to wait for a value to appear

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
```



flag

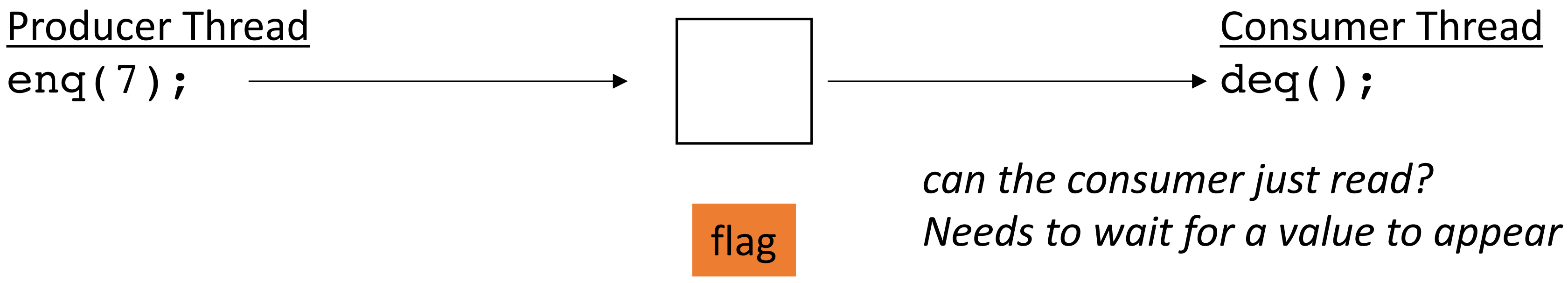
Consumer Thread

```
deq();
```

can the consumer just read?
Needs to wait for a value to appear

spin waiting for the flag to turn green

Synchronous Producer Consumer Queues



Producer Thread

`enq(7);`

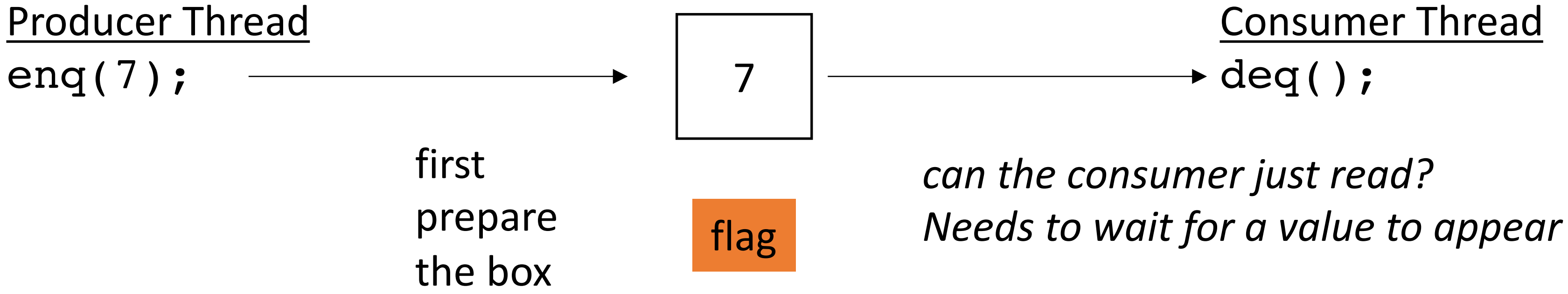
Consumer Thread

`deq();`

can the consumer just read?
Needs to wait for a value to appear

spin waiting for the flag to turn green

Synchronous Producer Consumer Queues

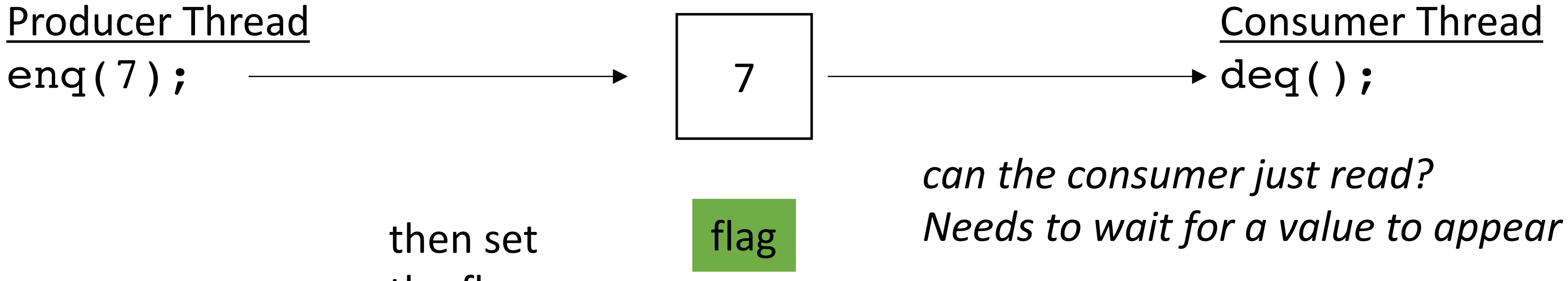


first prepare the box

can the consumer just read?
Needs to wait for a value to appear

spin waiting for the flag to turn green

Synchronous Producer Consumer Queues



Producer Thread

`enq(7);`

7

Consumer Thread

`deq();`

can the consumer just read?
Needs to wait for a value to appear

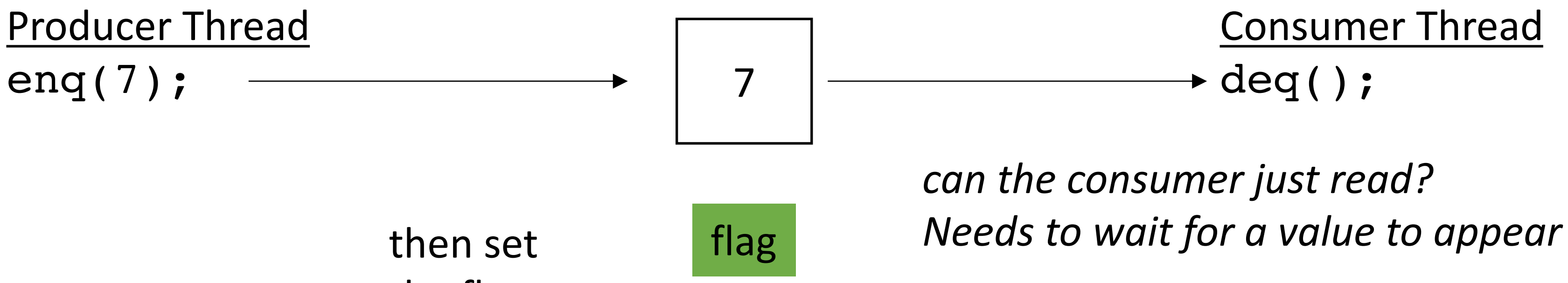
then set
the flag

flag

spin waiting for the flag to turn green

Synchronous Producer Consumer Queues

now the consumer can read from the box!



then set
the flag

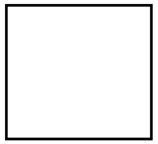
can the consumer just read?
Needs to wait for a value to appear

spin waiting for the flag to turn green

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
```



Consumer Thread

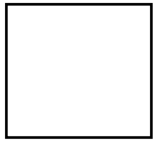
```
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
    }
  }
```

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
```



flag

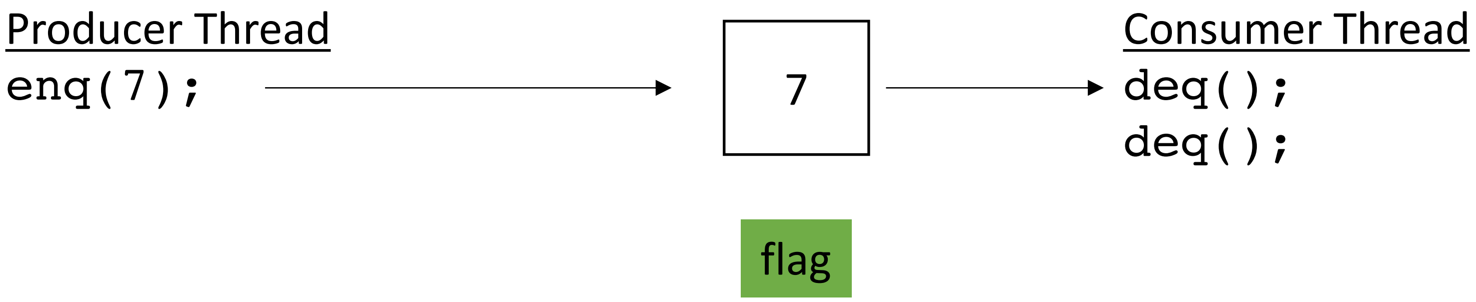
Consumer Thread

```
deq();
deq();
```

what happens when there are two deqs?

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
    }
  }
```

Synchronous Producer Consumer Queues



Producer Thread

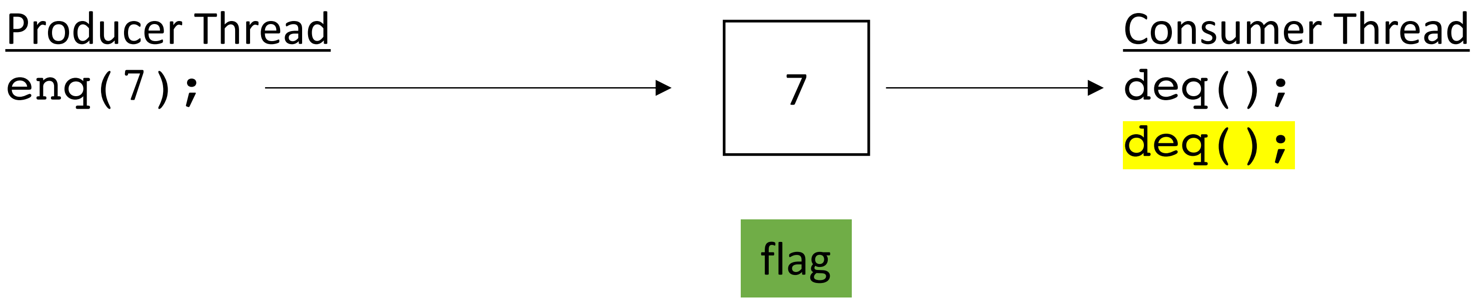
`enq(7);`

Consumer Thread

```
deq();
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
    }
  }
```

Synchronous Producer Consumer Queues



Producer Thread

enq(7);

7

flag

Consumer Thread

deq();

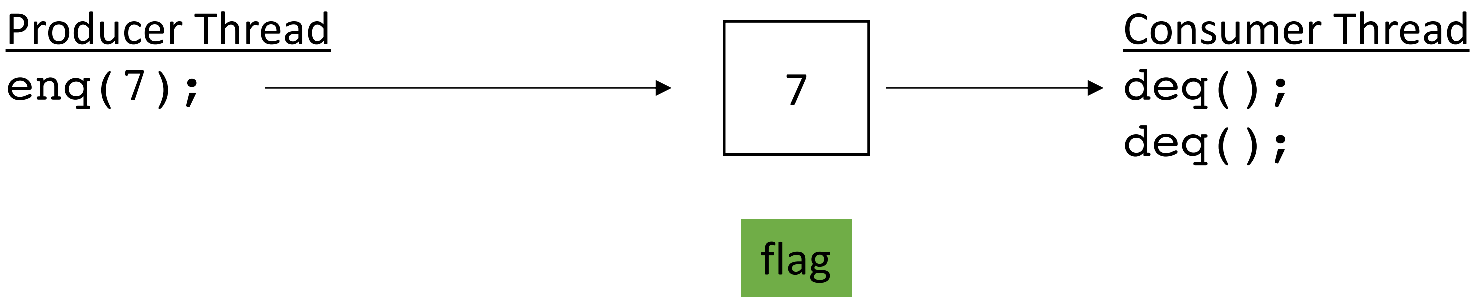
deq();

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
    }
  }
```

what happens in the next deq?

How to fix?

Synchronous Producer Consumer Queues



Producer Thread

`enq(7);`

7

flag

Consumer Thread

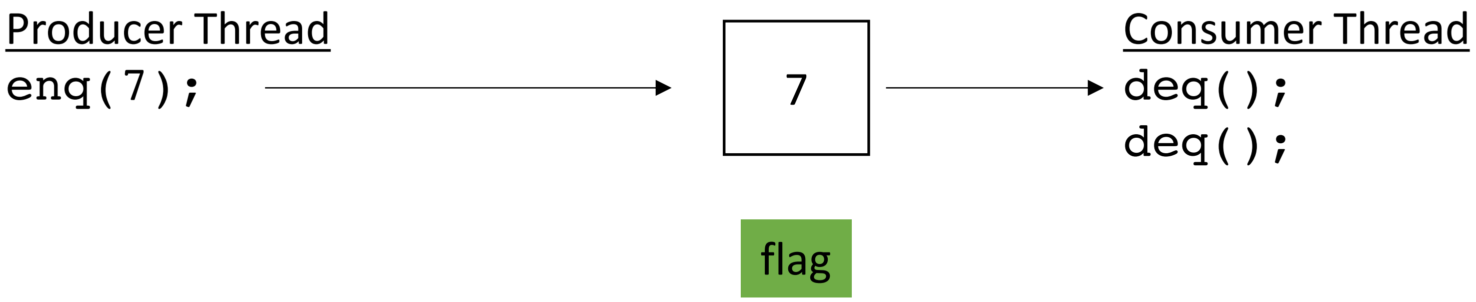
```
deq();
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

what happens in the next deq?

How to fix?

Synchronous Producer Consumer Queues



Producer Thread

`enq(7);`

7

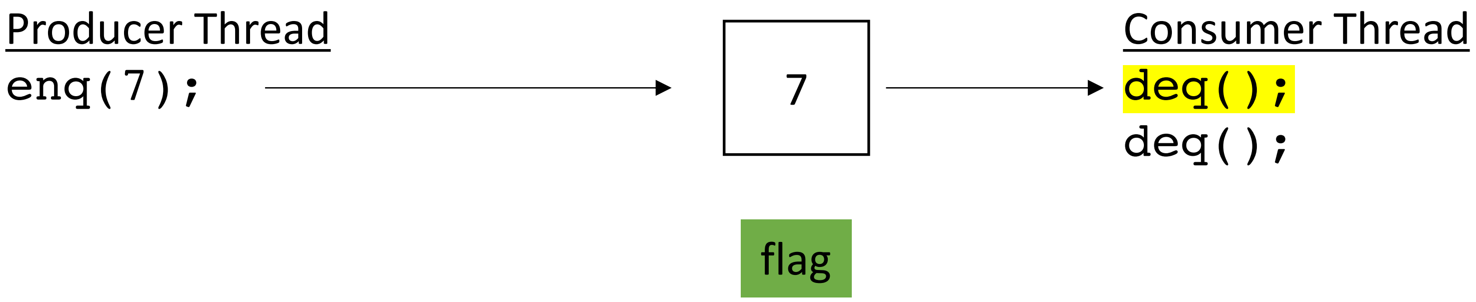
flag

Consumer Thread

`deq();`
`deq();`

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues



Producer Thread

enq(7);

7

flag

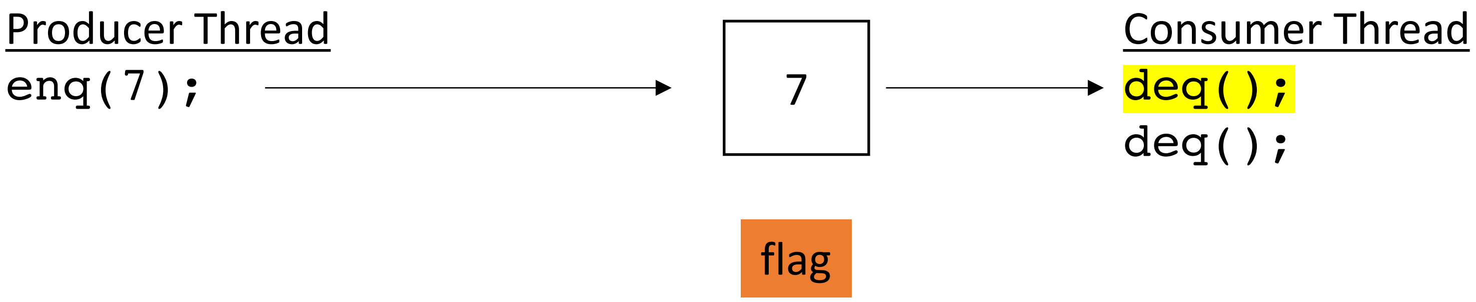
Consumer Thread

deq();

deq();

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues



Producer Thread

enq(7);

7

flag

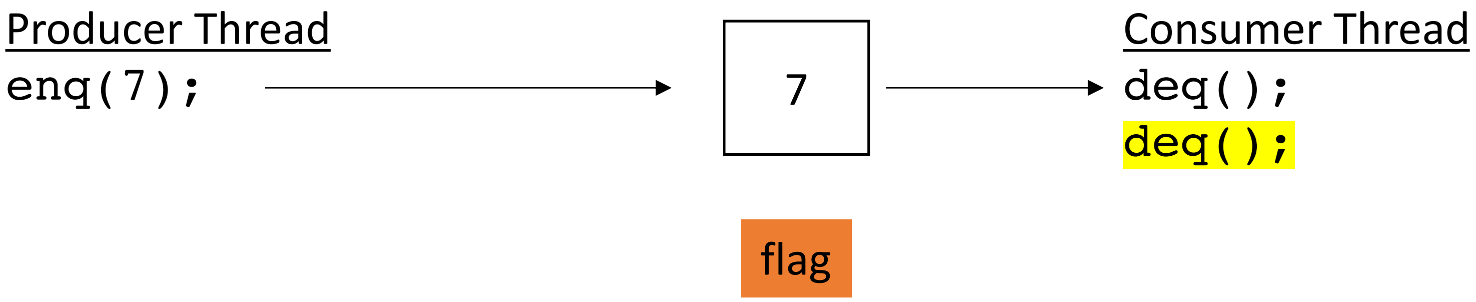
Consumer Thread

deq();

deq();

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x)  {
      // put value in box
      // set flag
    }
    void deq()  {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues



Producer Thread

enq(7);

7

flag

Consumer Thread

deq();

deq();

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

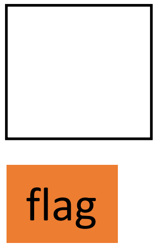
waiting like we are
supposed to

Synchronous Producer Consumer Queues

reset (now with extra enq)

Producer Thread

```
enq(7);
enq(8);
```



Consumer Thread

```
deq();
deq();
```

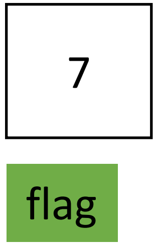
extra enq

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
enq(8);
```



7

flag

Consumer Thread

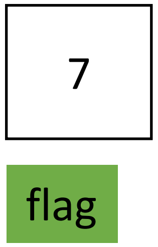
```
deq();
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
enq(8);
```



7

flag

Consumer Thread

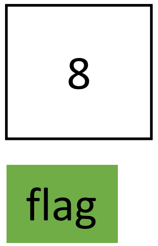
```
deq();
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

<u>Producer Thread</u>

```
enq(7);
enq(8);
```



8

flag

<u>Consumer Thread</u>

```
deq();
deq();
```

7 was dropped!

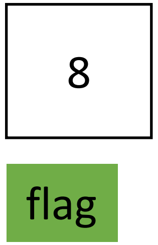
how to fix?

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
enq(8);
```



8

flag

Consumer Thread

```
deq();
deq();
```

7 was dropped!

how to fix?

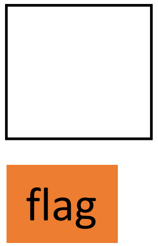
```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
      // wait for flag to be reset
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

reset

<u>Producer Thread</u>

```
enq(7);
enq(8);
```



flag

<u>Consumer Thread</u>

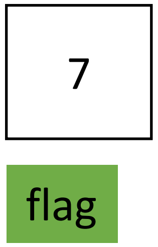
```
deq();
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
      // wait for flag to be reset
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
enq(8);
```



7

flag

Consumer Thread

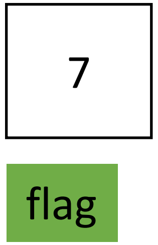
```
deq();
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
      // wait for flag to be reset
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
enq(8);
```



7

flag

Consumer Thread

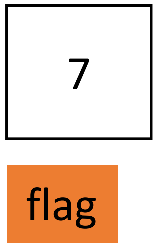
```
deq();
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
      // wait for flag to be reset
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
enq(8);
```



Consumer Thread

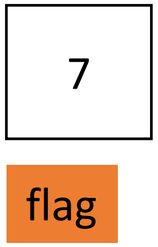
```
deq();
deq();
```

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
      // wait for flag to be reset
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Synchronous Producer Consumer Queues

<u>Producer Thread</u>

```
enq(7);
enq(8);
```



<u>Consumer Thread</u>

```
deq();
deq();
```

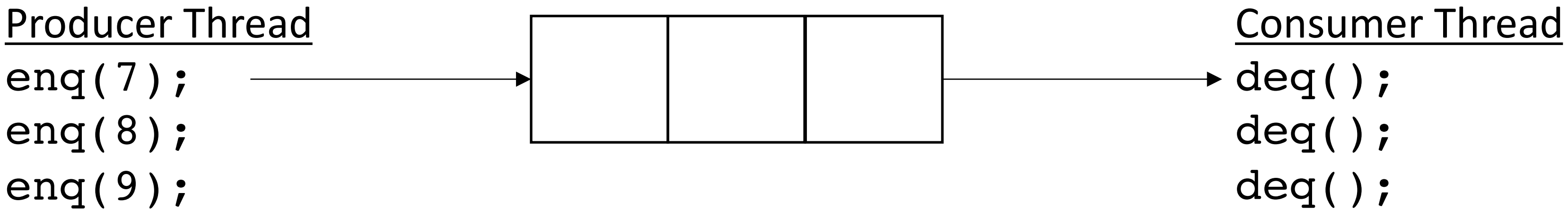
```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  public:
    void enq(int x) {
      // put value in box
      // set flag
      // wait for flag to be reset
    }
    void deq() {
      // wait for flag to be set
      // read from the box
      // reset flag
    }
  }
```

Schedule

- Producer Consumer Queues
  - Synchronous
  - Circular buffer

Producer Consumer Queues

- Asynchronous:



Producer Thread

```
enq(7);
enq(8);
enq(9);
```

Consumer Thread

```
deq();
deq();
deq();
```

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
```

Consumer Thread

```
deq();
deq();
deq();
```

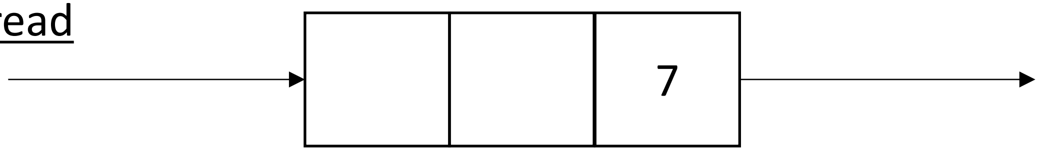
no waiting for producer (while there is room)

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
```



Consumer Thread

```
deq();
deq();
deq();
```

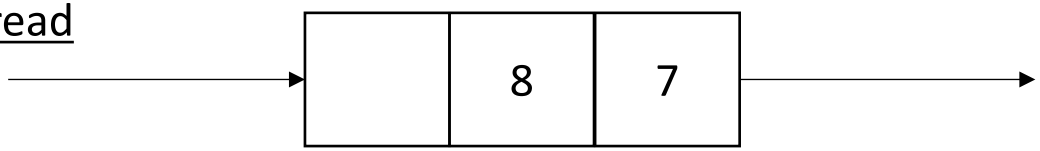
no waiting for producer (while there is room)

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
```



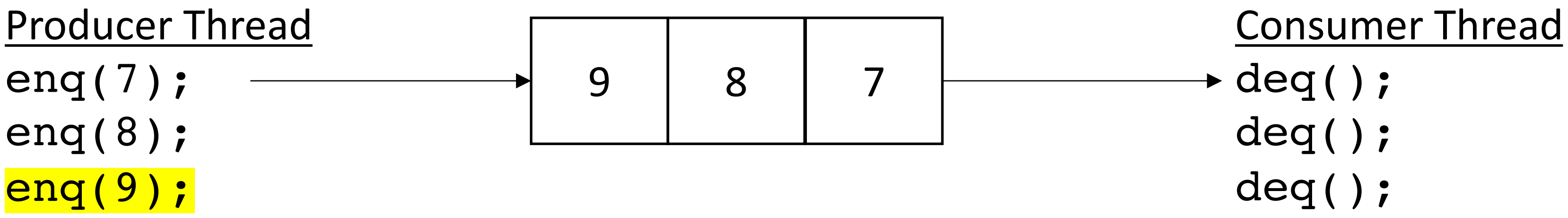
Consumer Thread

```
deq();
deq();
deq();
```

no waiting for producer (while there is room)

Producer Consumer Queues

- Asynchronous:



Producer Thread

```
enq(7);
enq(8);
enq(9);
```

Consumer Thread

```
deq();
deq();
deq();
```

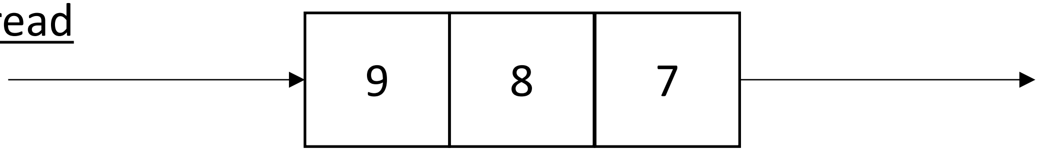
no waiting for producer (while there is room)

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```



Consumer Thread

```
deq();
deq();
deq();
```

no waiting for producer (while there is room)

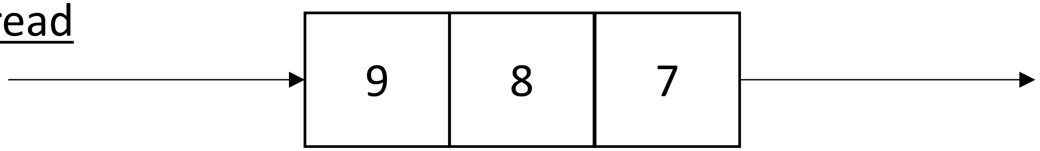
when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```



Consumer Thread

```
deq();
deq();
deq();
```

no waiting for producer (while there is room)

returns 7

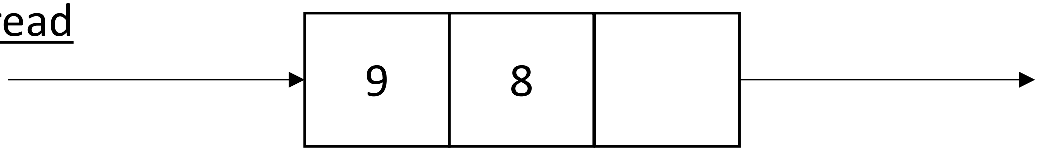
when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```



Consumer Thread

```
deq();
deq();
deq();
```

no waiting for producer (while there is room)

returns 7

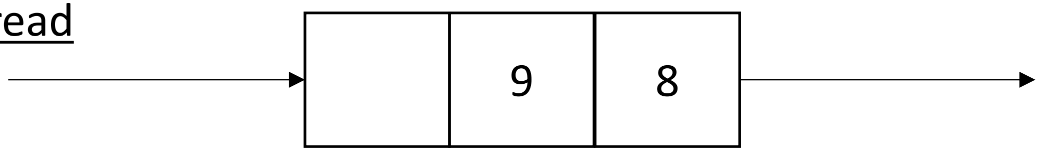
when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```



Consumer Thread

```
deq();
deq();
deq();
```

no waiting for producer (while there is room)

returns 7

when there is no room, the queue will wait

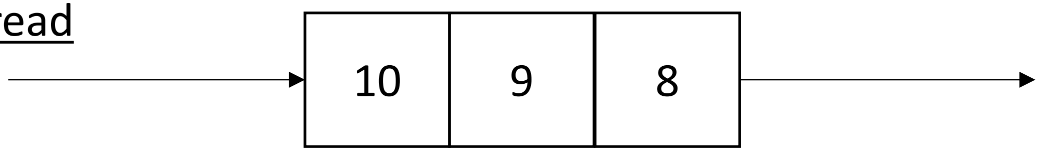
Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```

finishes



Consumer Thread

```
deq();
deq();
deq();
```

no waiting for producer (while there is room)

returns 7

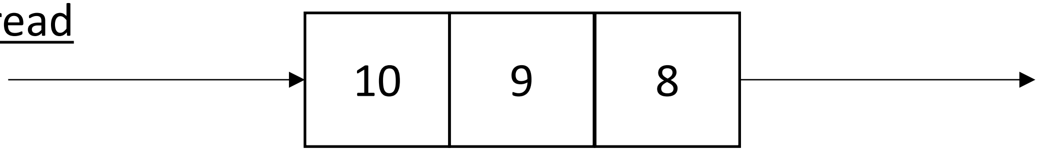
when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```



Consumer Thread

```
deq();
deq();
deq();
```

no waiting for producer (while there is room)

returns 7

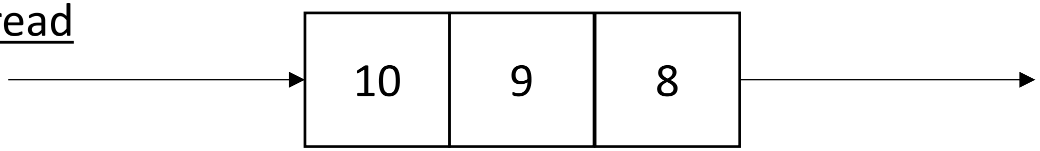
when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:

<u>Producer Thread</u>

```
enq(7);
enq(8);
enq(9);
enq(10);
```



<u>Consumer Thread</u>

```
deq();
deq();
deq();
```

no waiting for producer (while there is room)

returns 8

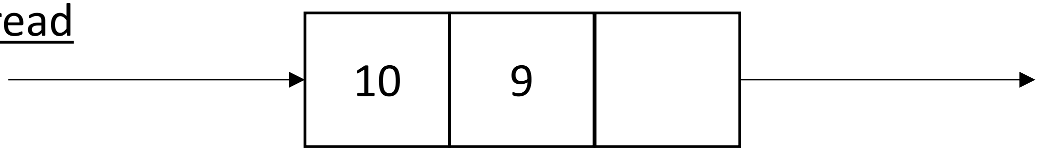
when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```



Consumer Thread

```
deq();
deq();
deq();
```

no waiting for producer (while there is room)

returns 8

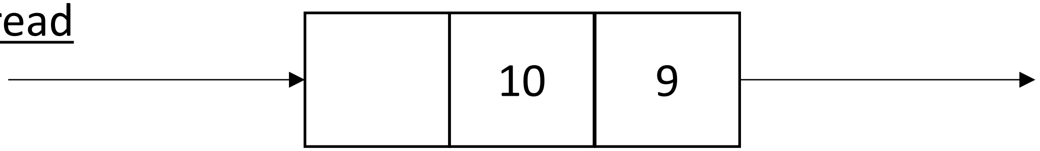
when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```



Consumer Thread

```
deq();
deq();
deq();
```

no waiting for producer (while there is room)

returns 8

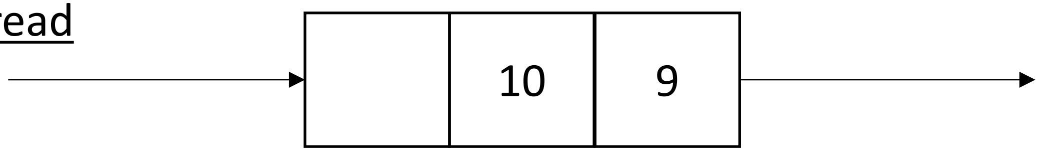
when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```



Consumer Thread

```
deq();
deq();
deq();
```

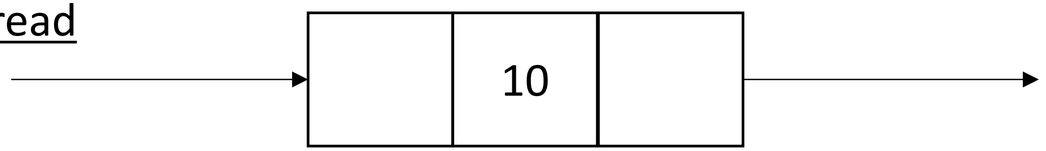
returns 9

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```



Consumer Thread

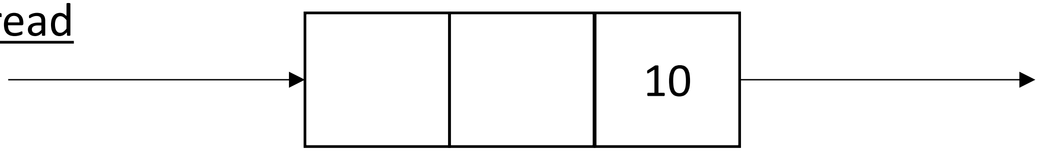
```
deq();
deq();
deq();
```

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```



Consumer Thread

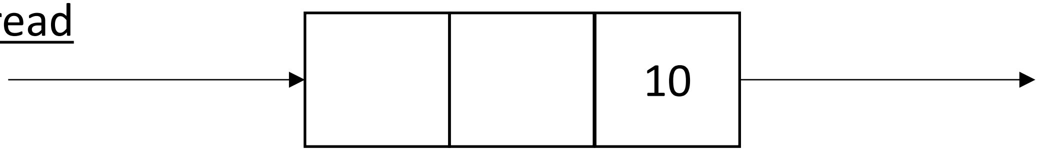
```
deq();
deq();
deq();
deq();
```

Producer Consumer Queues

- Asynchronous:

Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```

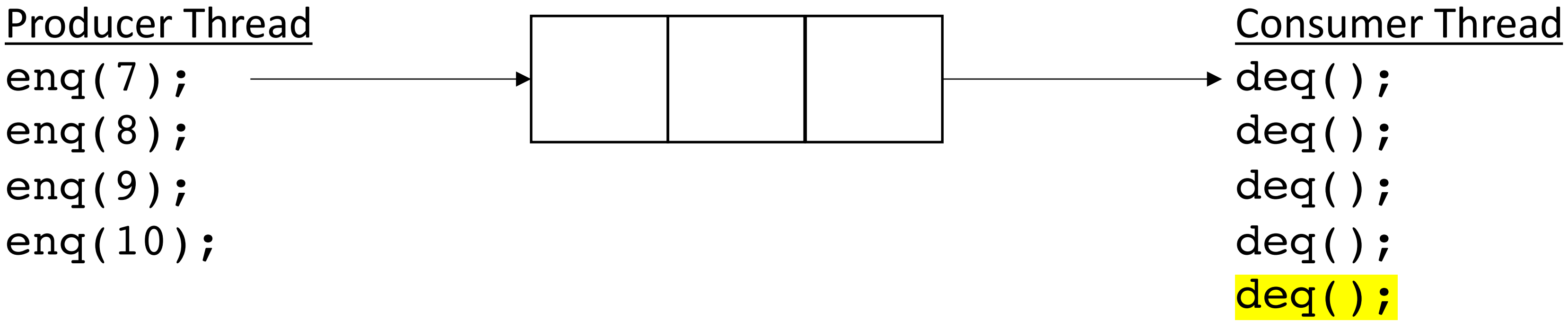


Consumer Thread

```
deq();
deq();
deq();
deq();
```

Producer Consumer Queues

- Asynchronous:



Producer Thread

```
enq(7);
enq(8);
enq(9);
enq(10);
```

Consumer Thread

```
deq();
deq();
deq();
deq();
deq();
```

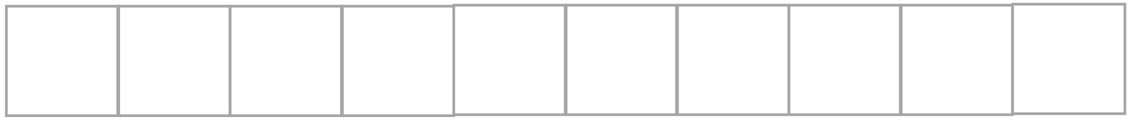
blocks when there is nothing in the queue

Producer Consumer Queues

- How do we implement it?

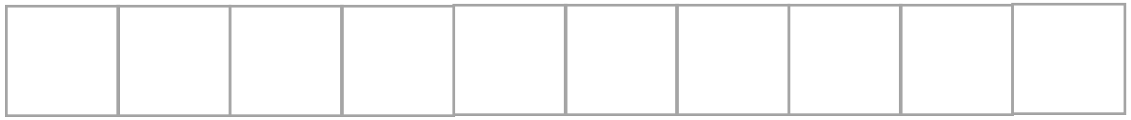
Producer Consumer Queues

- Start with a fixed size array



Producer Consumer Queues

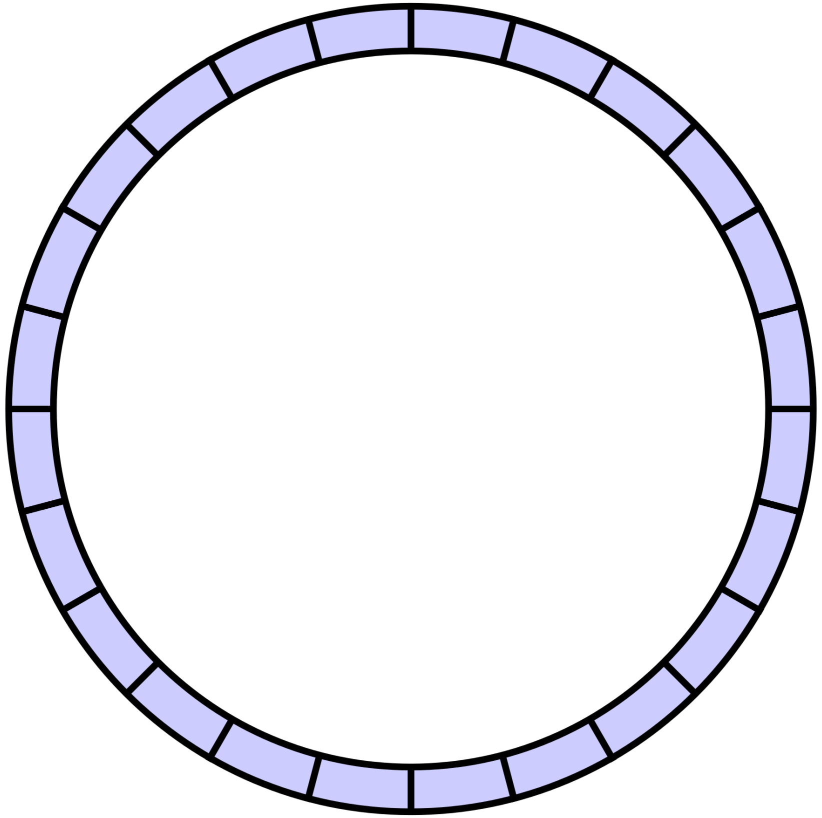
- Start with a fixed size array



We will use what is called a *circular buffer method*

Producer Consumer Queues

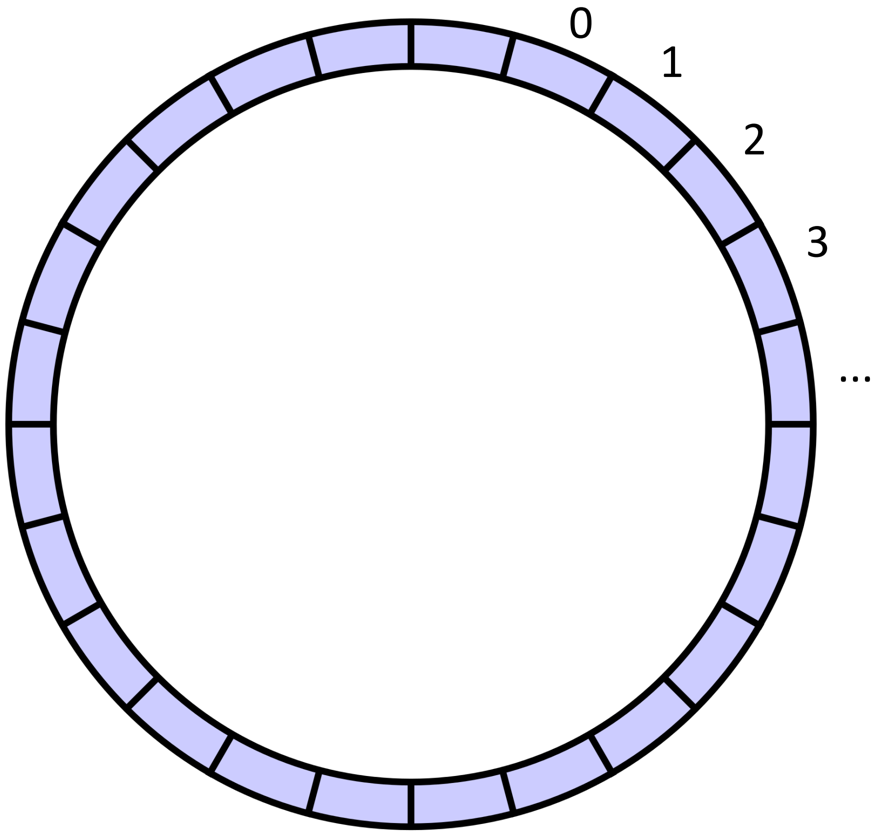
- Start with a fixed size array



conceptually it is a circle

Producer Consumer Queues

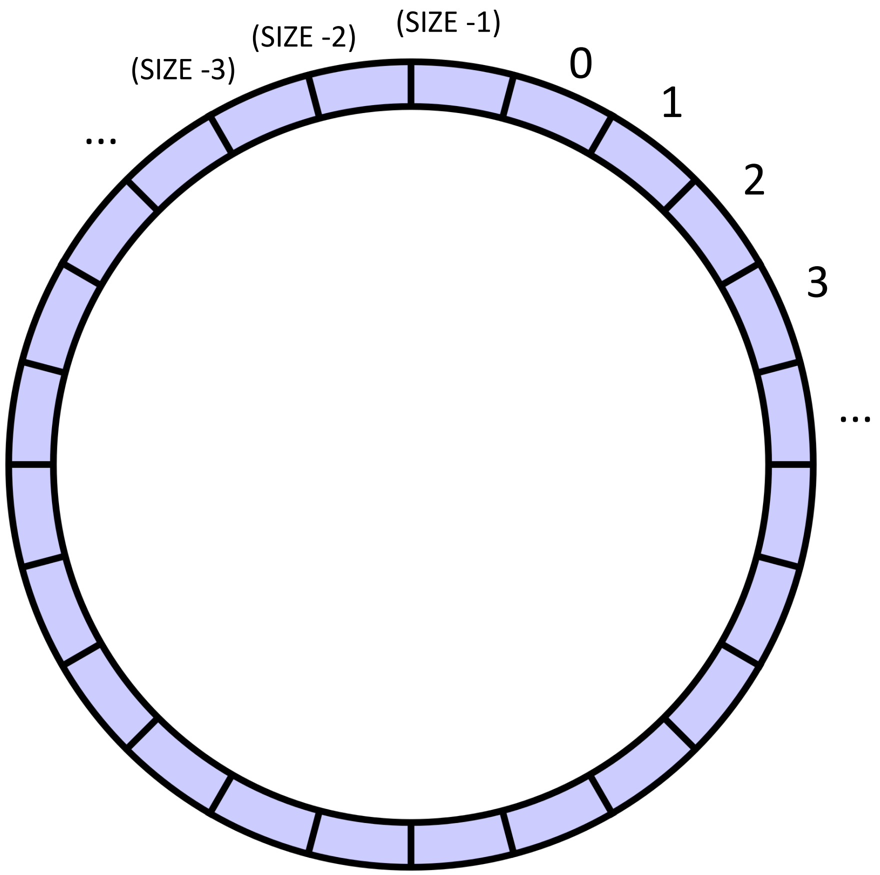
- Start with a fixed size array



conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array

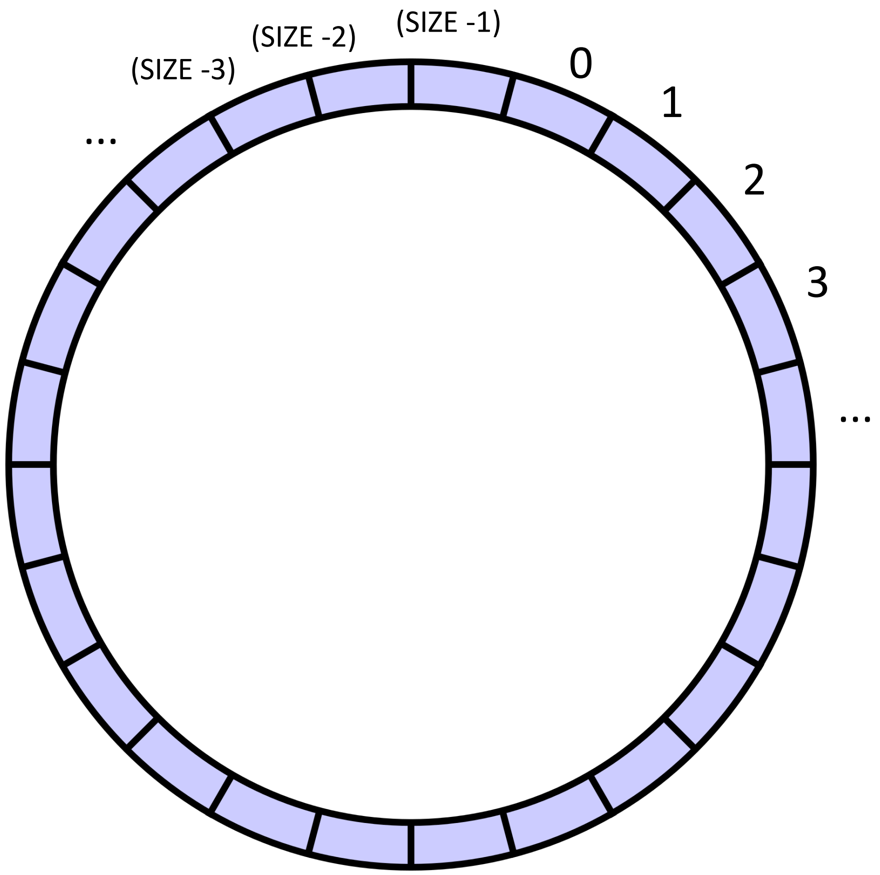


indexes will circulate in order and wrap around

conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array



indexes will circulate in order and wrap around

we will assume modular arithmetic:

if x = (SIZE - 1) then
x + 1 == 0;

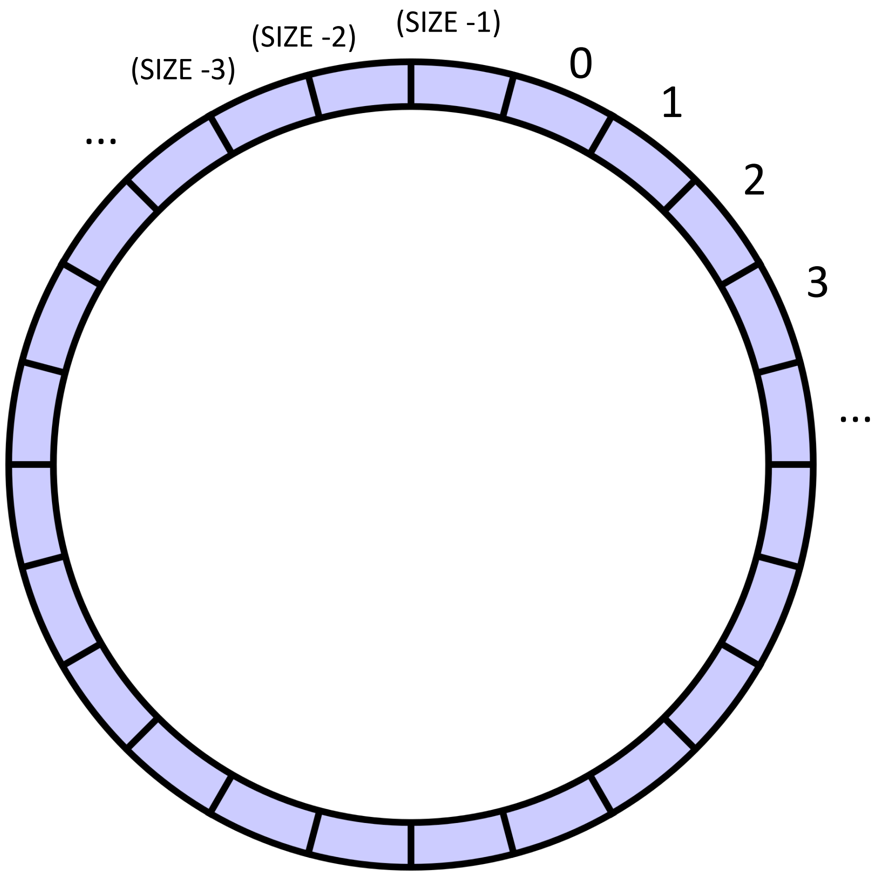
conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array

Two variables to keep track of where to deq and enq:

head and tail



indexes will circulate in order and wrap around

conceptually it is a circle

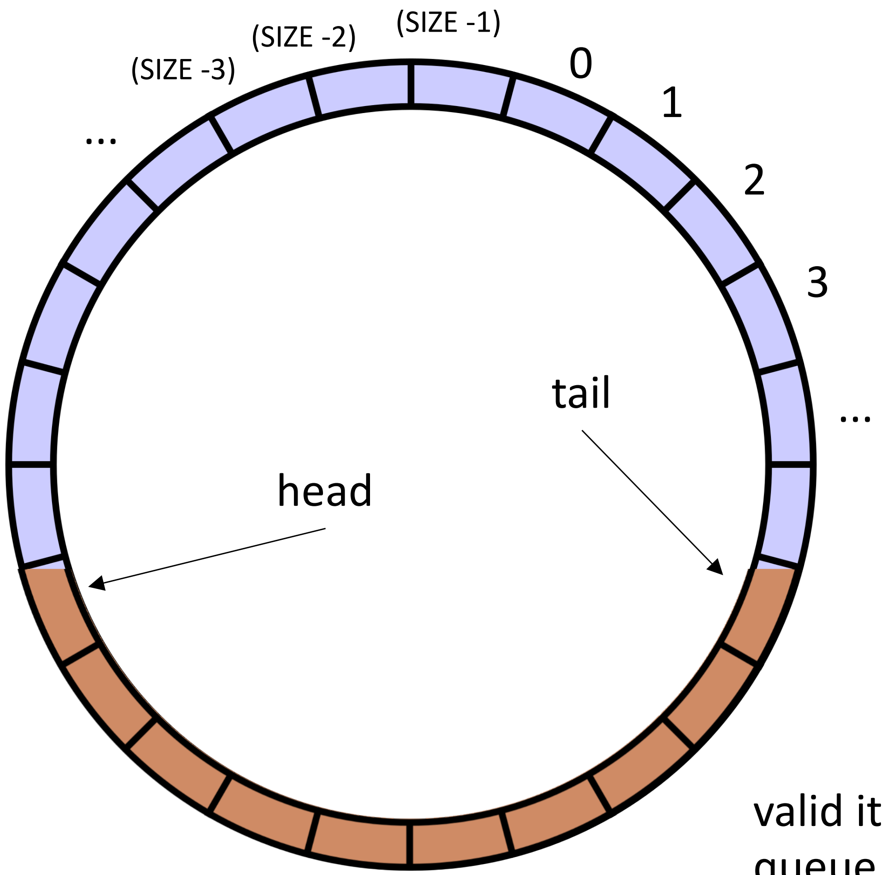
Producer Consumer Queues

- Start with a fixed size array

Two variables to keep track of where to deq and enq:

head and tail:

enq to the head, deq from the tail



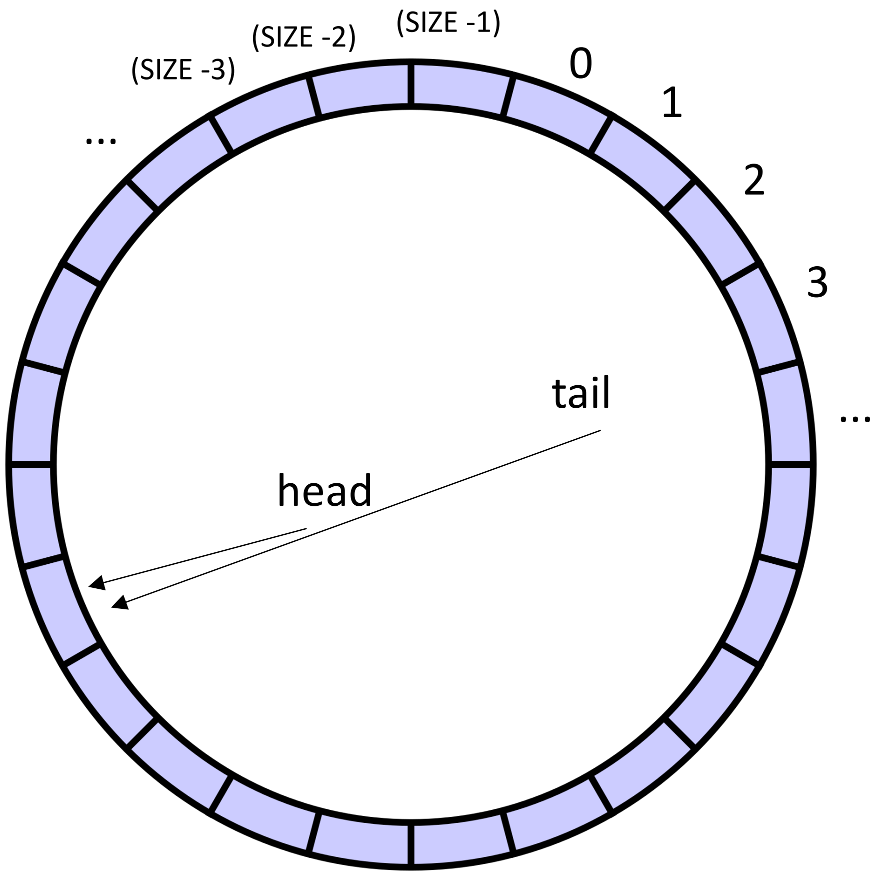
indexes will circulate in order and wrap around

valid items in the queue

conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array



indexes will circulate in order and wrap around

Two variables to keep track of where to deq and enq:

head and tail

Empty queue is when head == tail

conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array

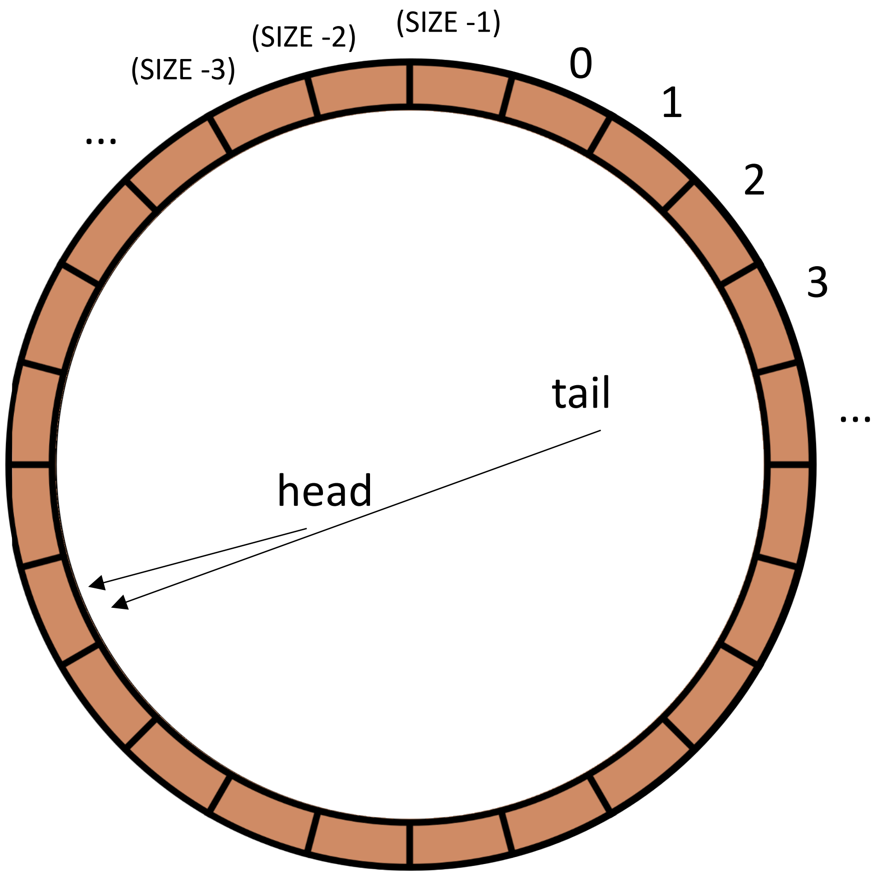
Two variables to keep track of where to deq and enq:

head and tail

Empty queue is when head == tail

Full queue is when head == tail?

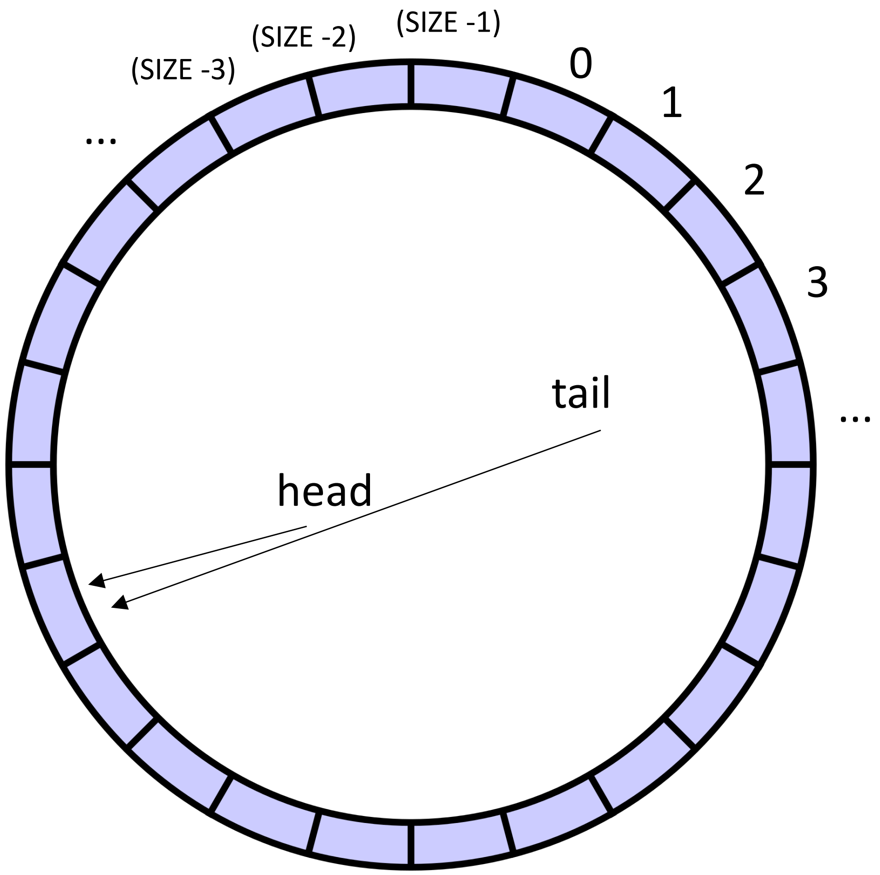
conceptually it is a circle



indexes will circulate in order and wrap around

Producer Consumer Queues

- Start with a fixed size array



indexes will circulate in order and wrap around

Two variables to keep track of where to deq and enq:

head and tail

Empty queue is when head == tail

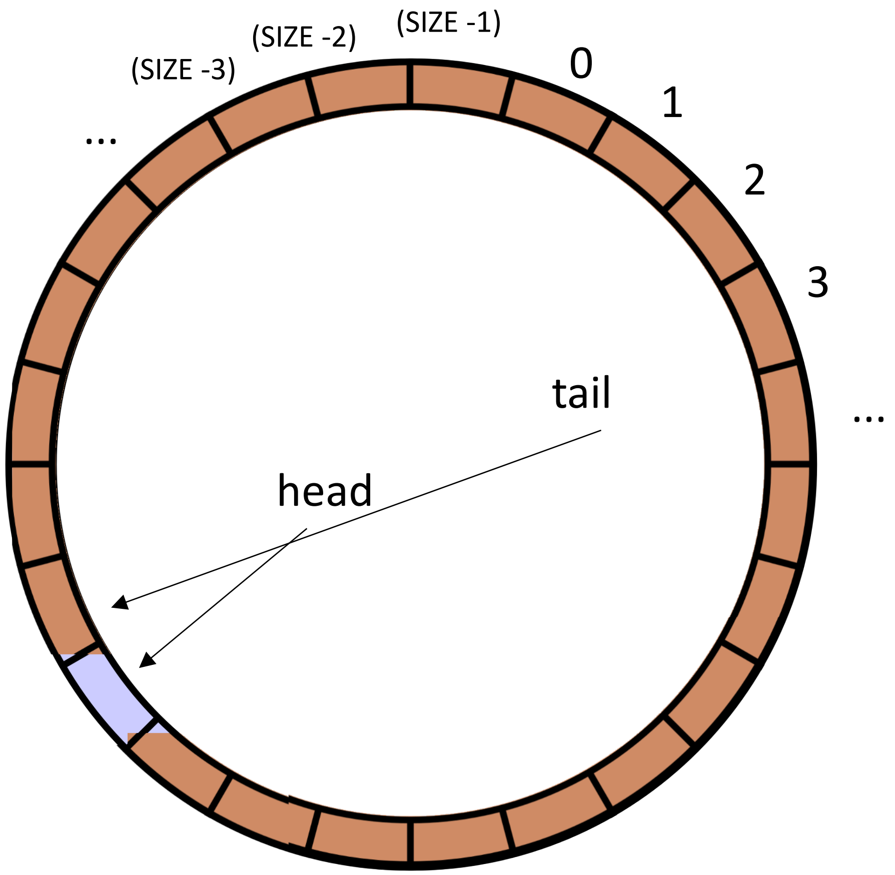
Full queue is when head == tail?

but then how to tell full queue from empty?

conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array



indexes will circulate in order and wrap around

Two variables to keep track of where to deq and enq:

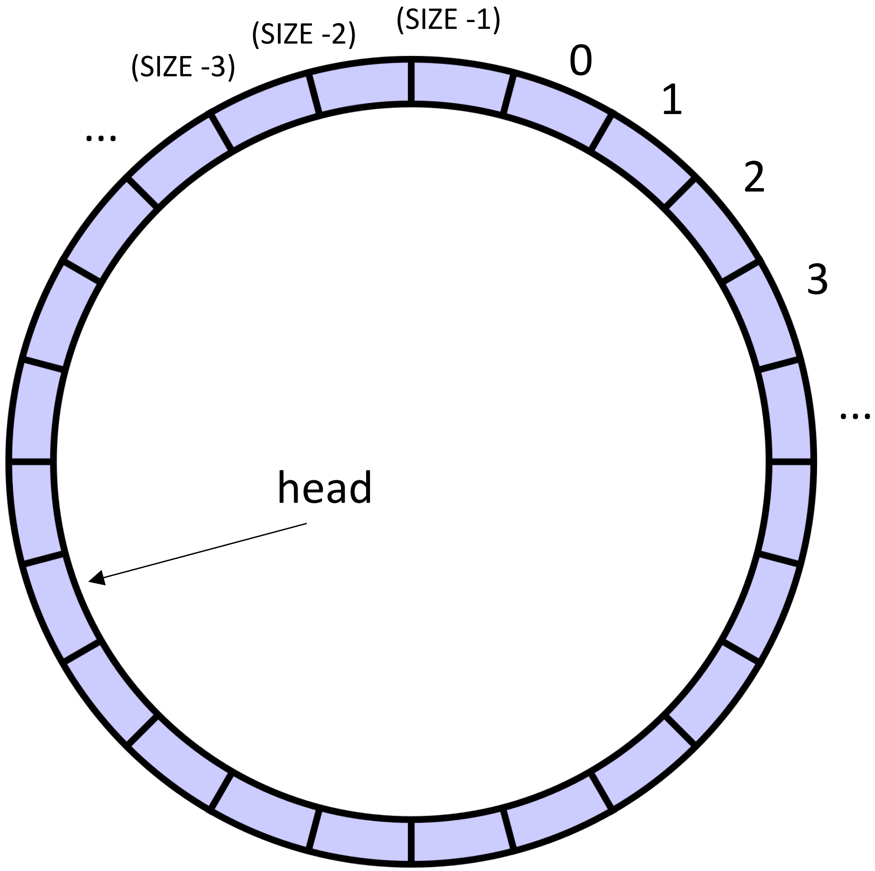
head and tail

Empty queue is when head == tail

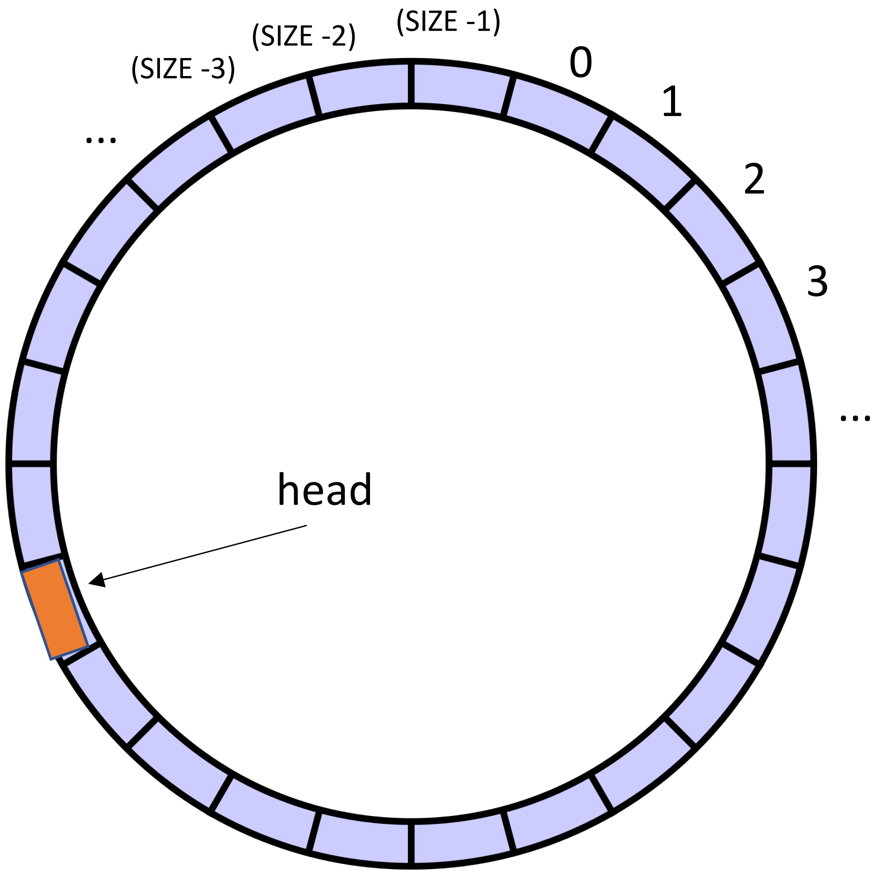
Full queue is when head + 1 == tail

wasting one location, but its okay...

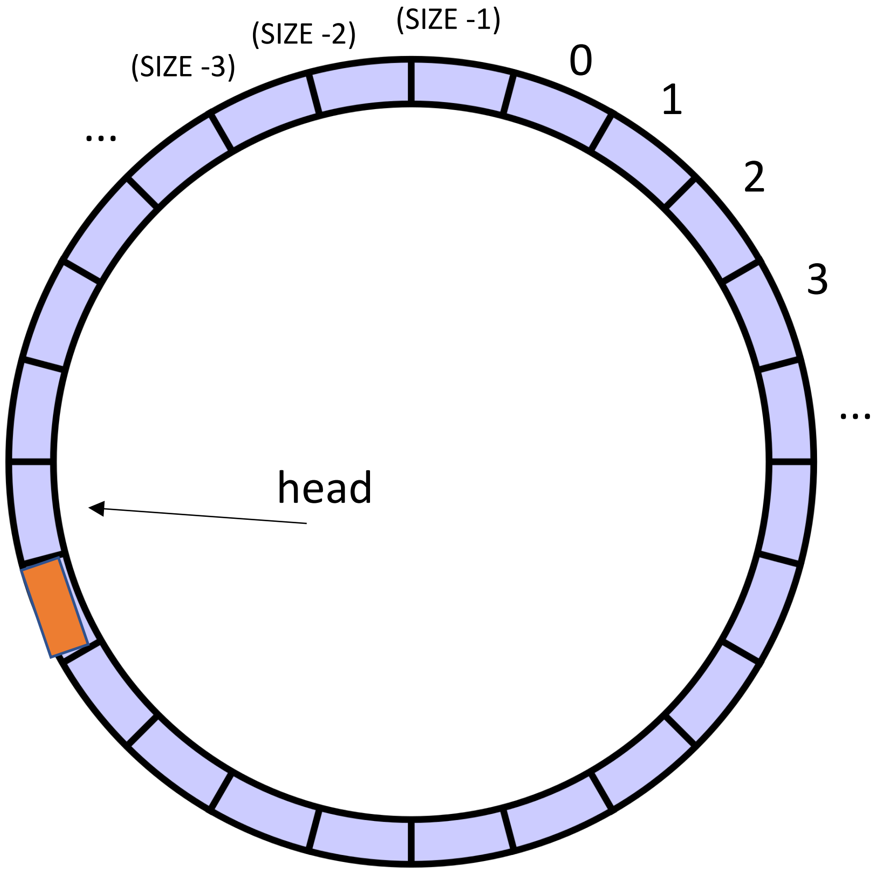
conceptually it is a circle



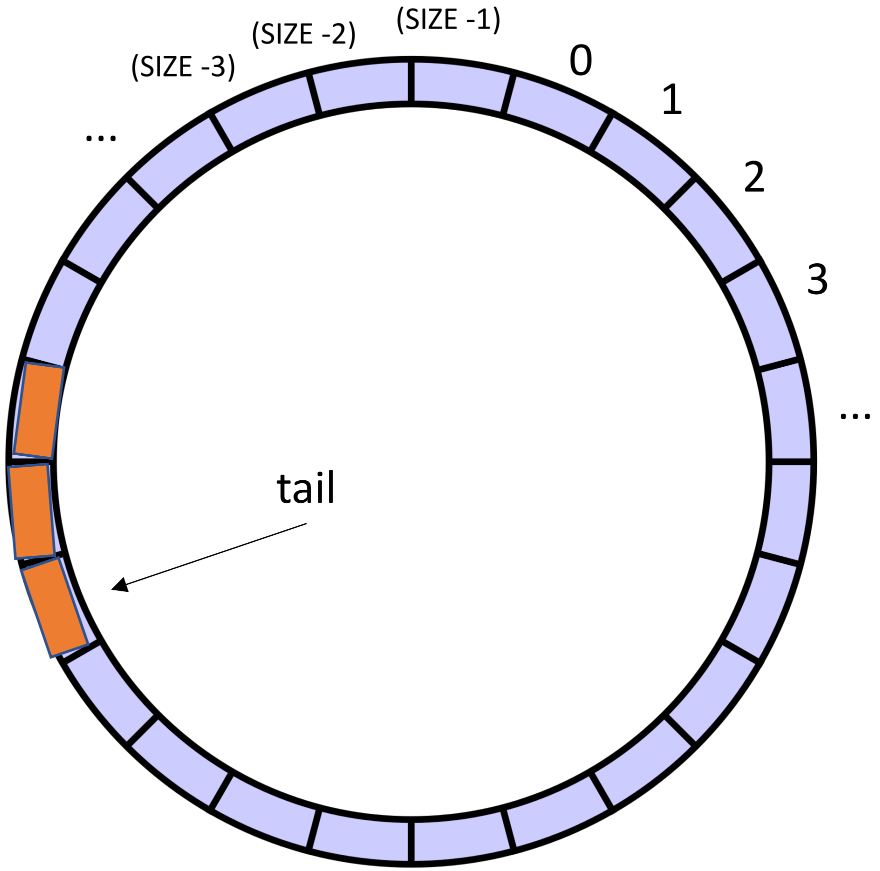
```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
}
```



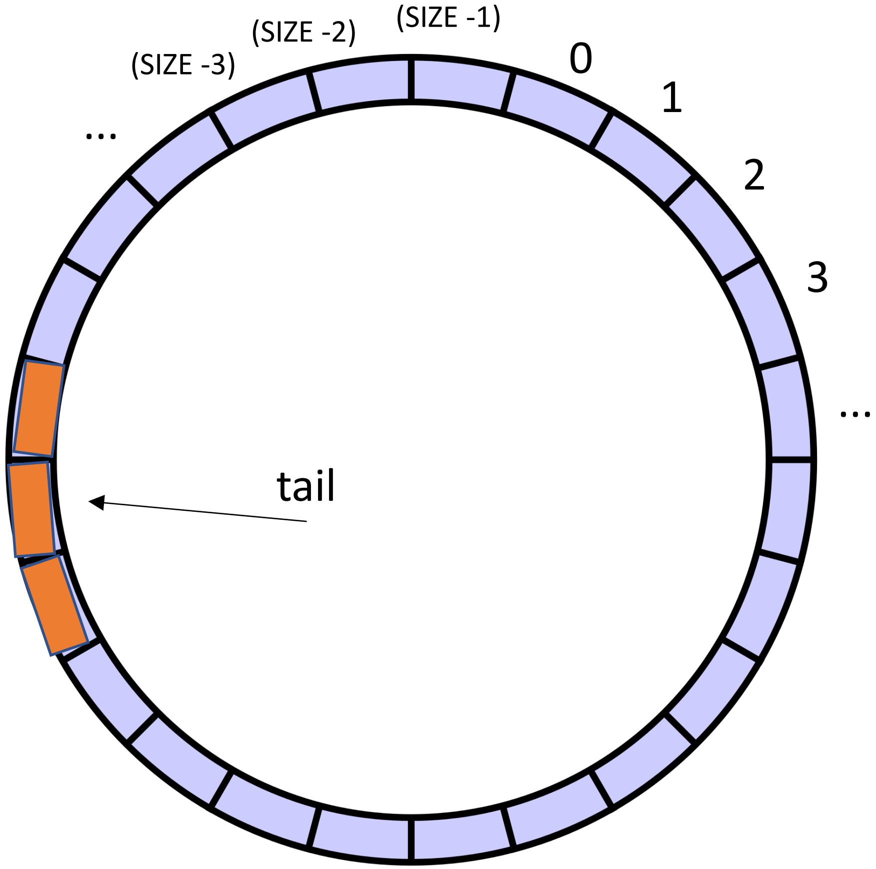
```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
}
```



```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
}
```

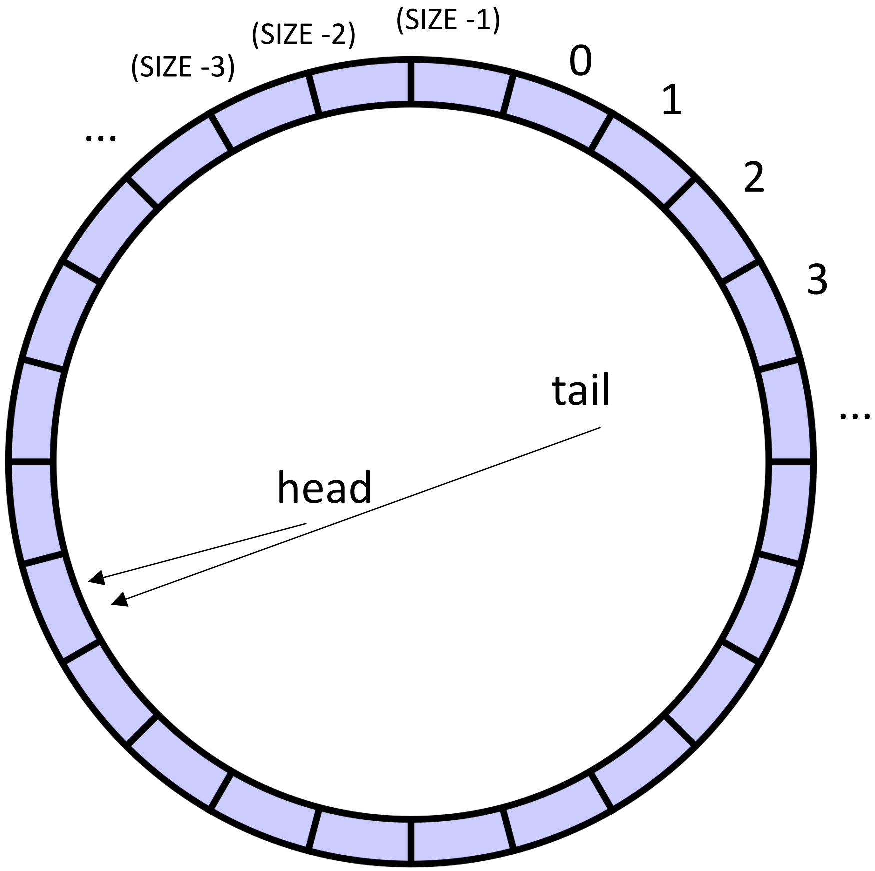


```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // get value at tail
      // increment tail
    }
}
```



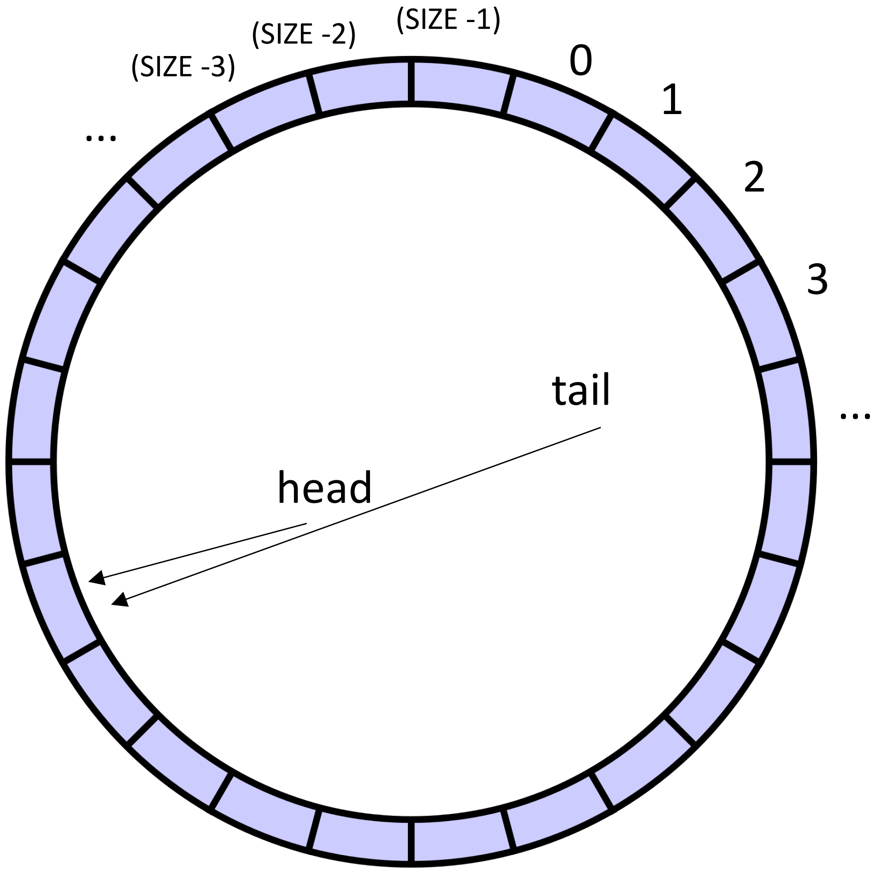
```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // get value at tail
      // increment tail
    }
}
```

This looks like the two threads don't even share head and tail! What is missing?

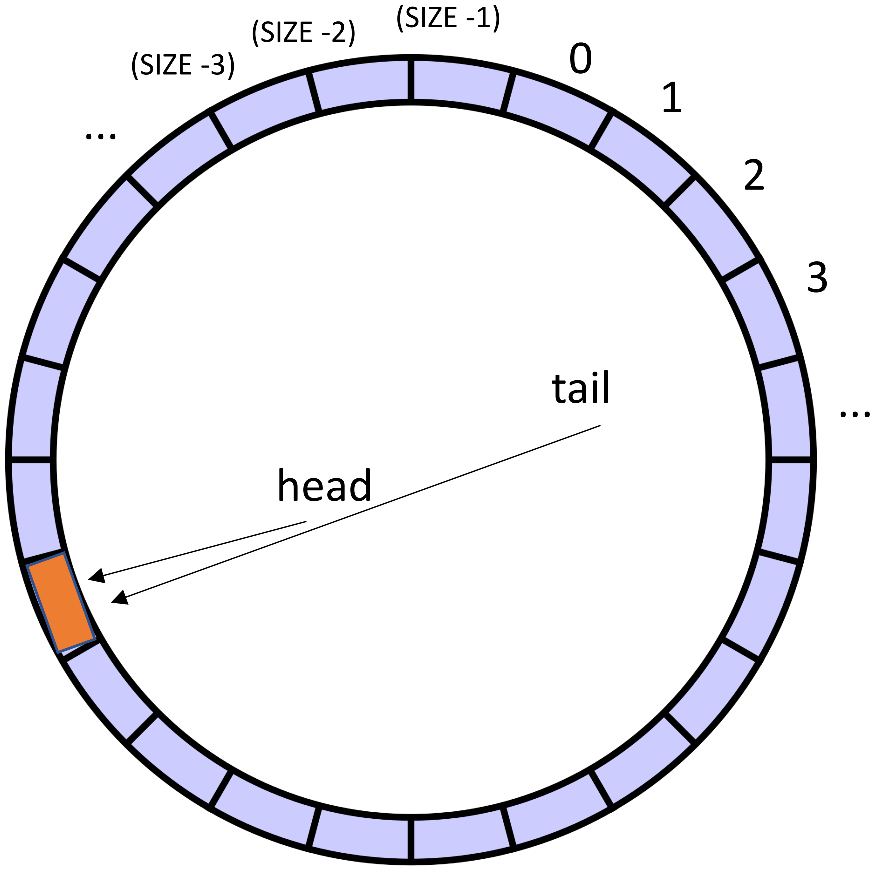


```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // get value at tail
      // increment tail
    }
}
```

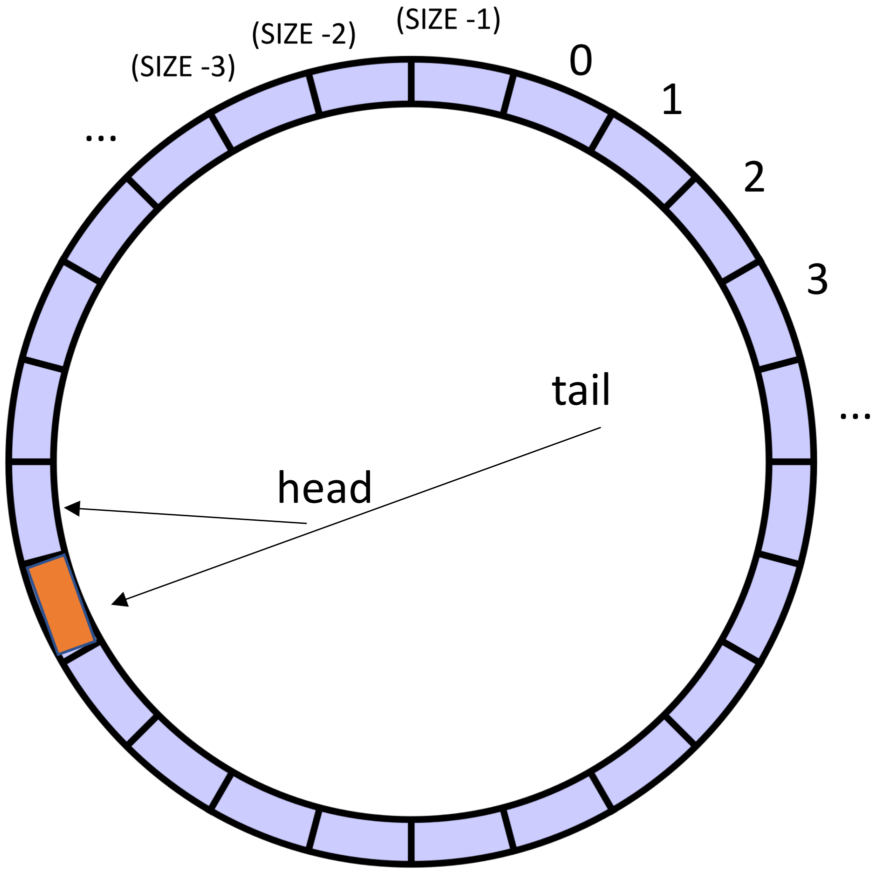
what happens if we try to dequeue here?



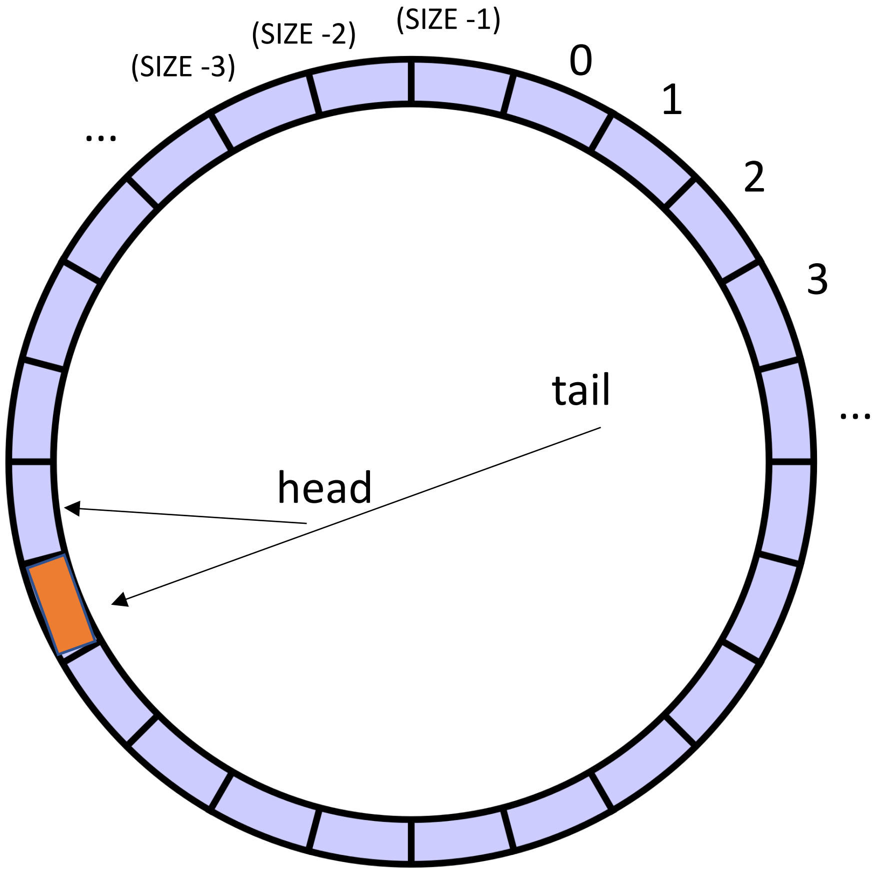
```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```



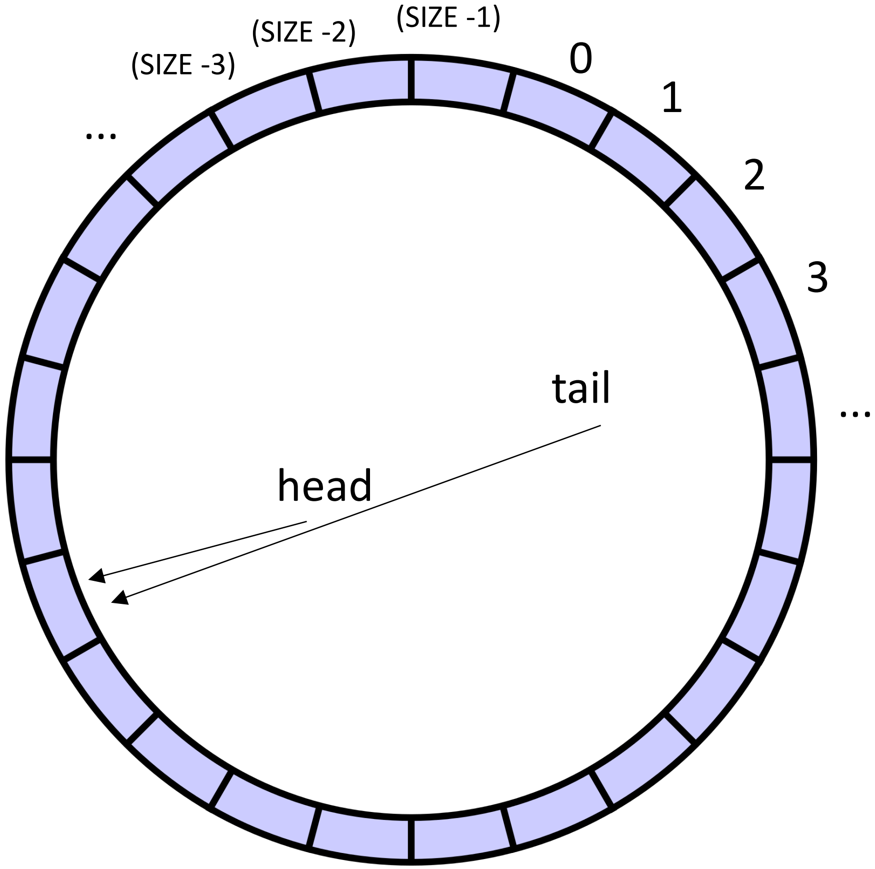
```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```



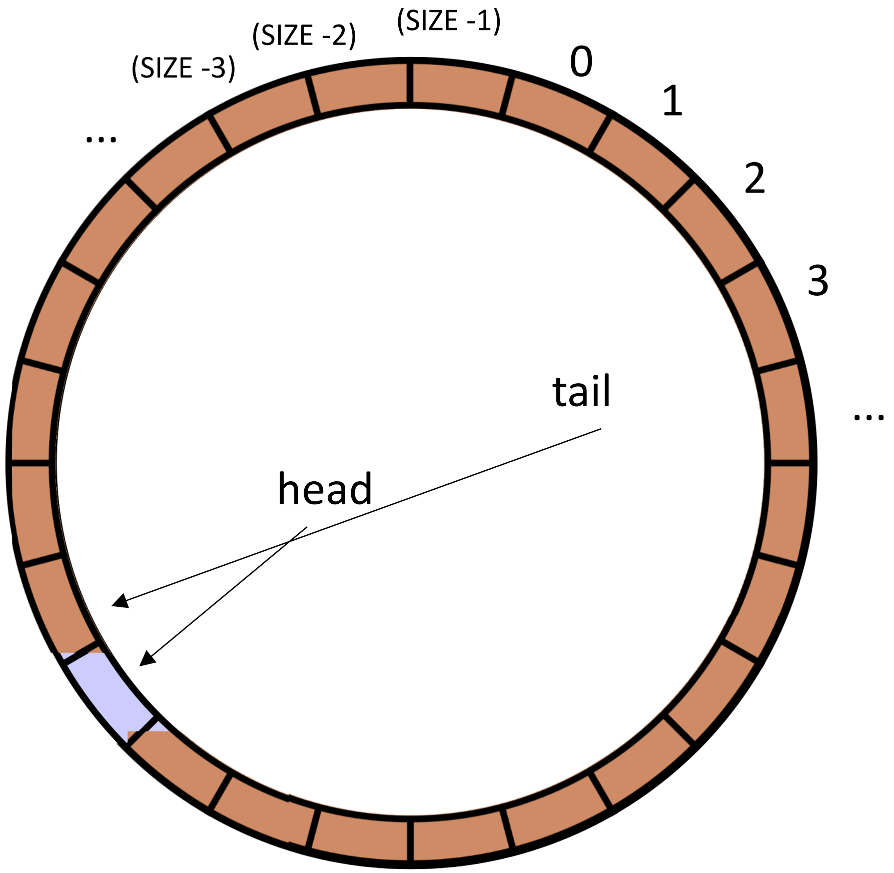
```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```



```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```



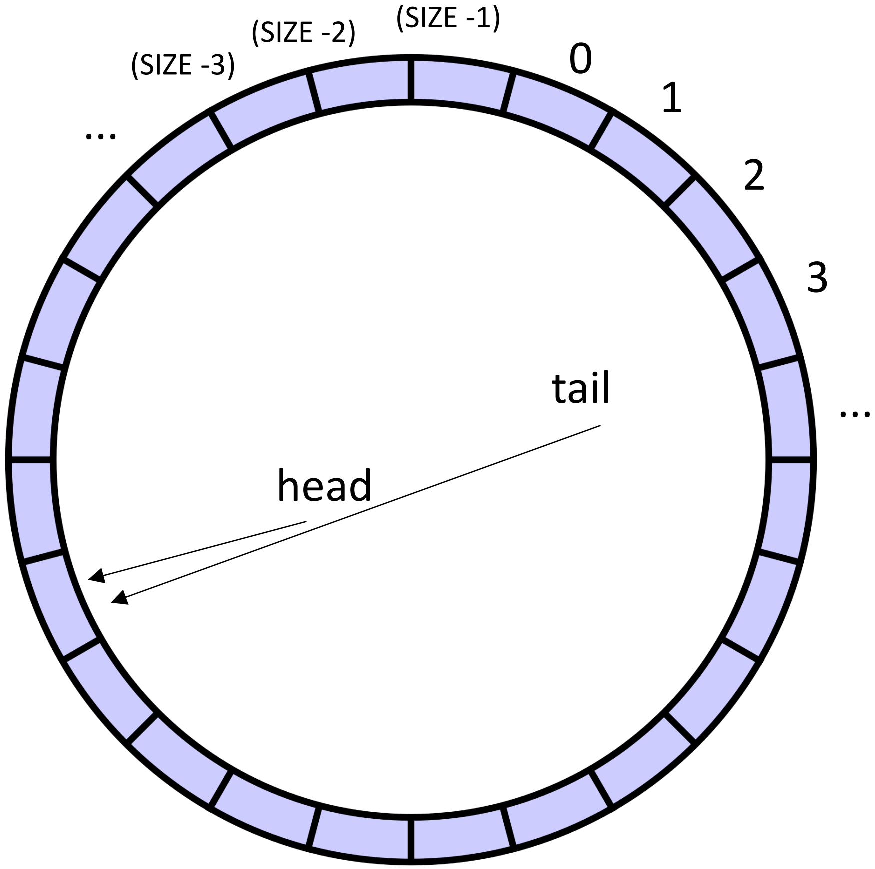
```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```



```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```

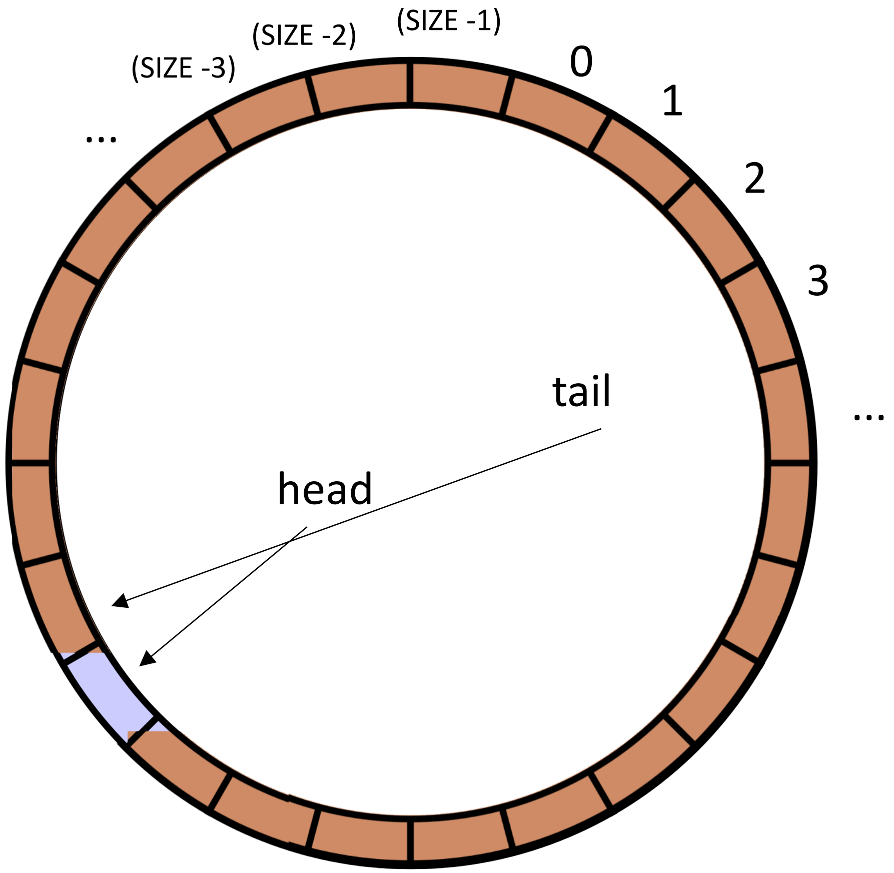
similarly for enqueue

but why can't we enqueue?



```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```

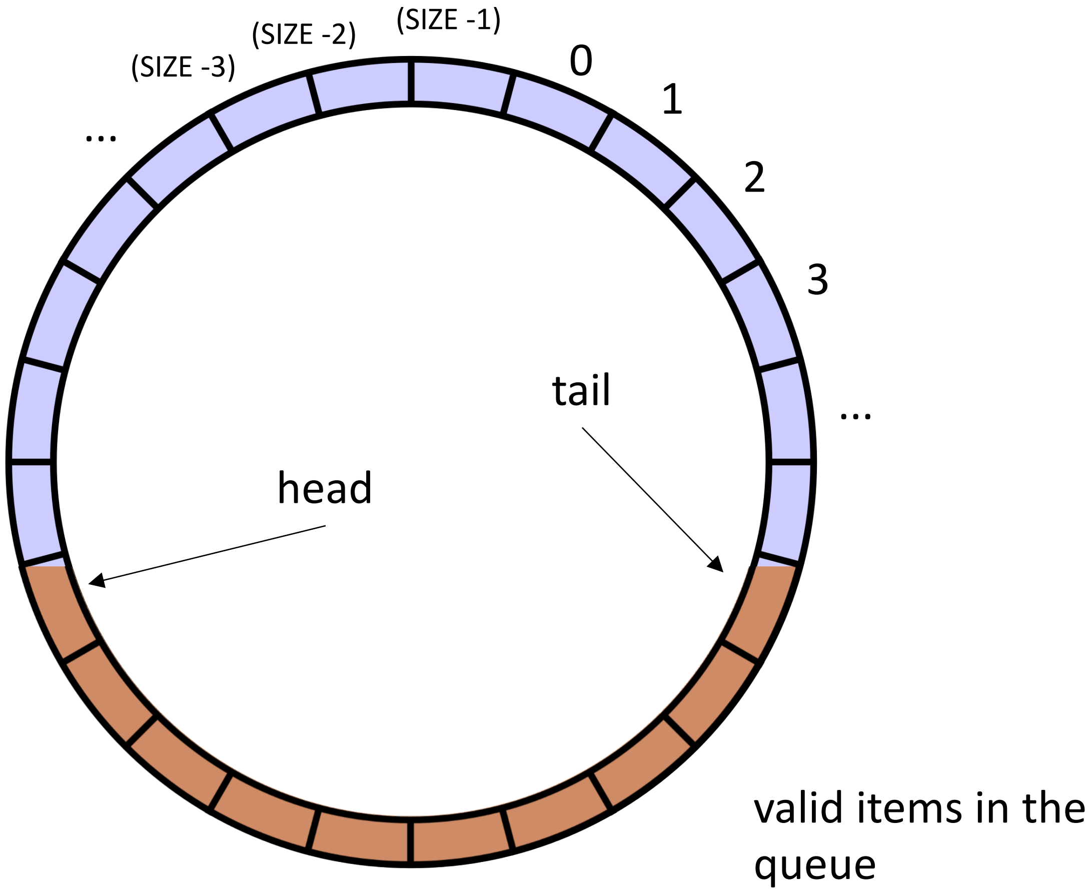
incrementing the head would make it empty!



```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // wait for their to be room
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```

we need to wait for there
to be room

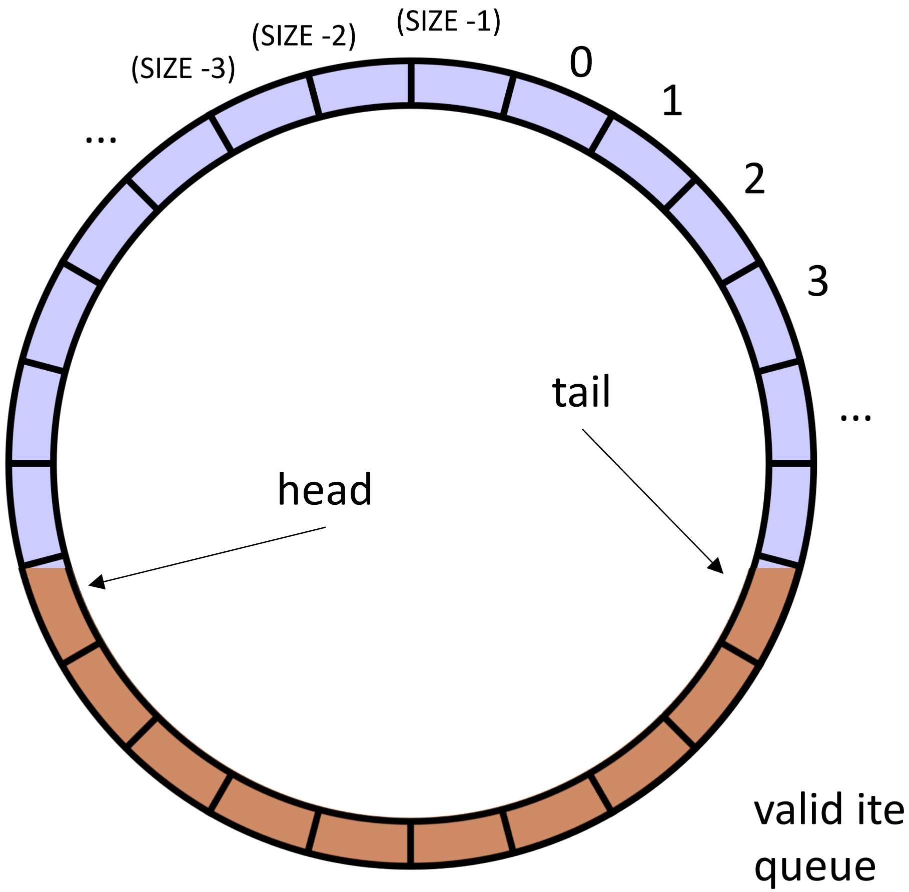
Other questions:



```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // wait for their to be room
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```

Other questions:

Do these need to be atomic RMWs?



valid items in the queue

```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  public:
    void enq(int x) {
      // wait for their to be room
      // store value at head
      // increment head
    }
    int deq() {
      // wait while queue is empty
      // get value at tail
      // increment tail
    }
}
```

Next week

- Work stealing and generalized concurrent objects

- Get HW 2 turned in today!

- HW 3 is out today. You can get started on Part 1

- Prepare for midterm on Monday