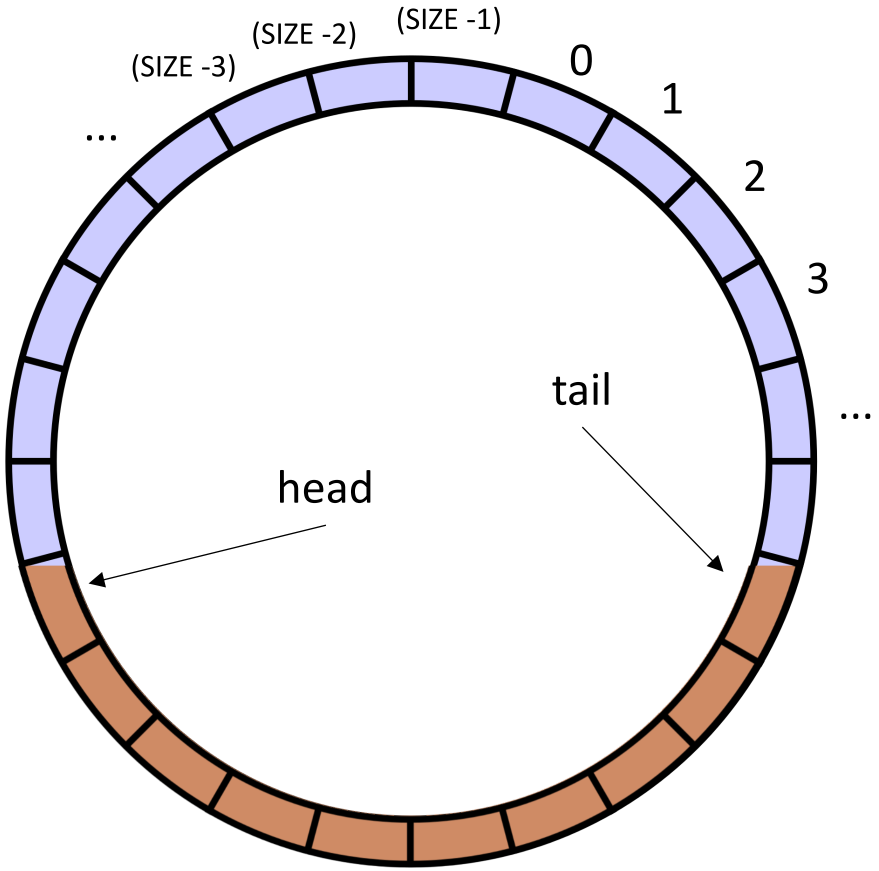
CSE113: Parallel Programming

- **Topics**:
  - Linearizability
  - Input/output queues
  - Producer/consumer queues



Announcements

- Midterm
    - Grades already announced or will be announced soon
    - Our TAs will go over the questions and answers on Thursday
- HW 1
    - Grades will be announced by Thursday

Announcements

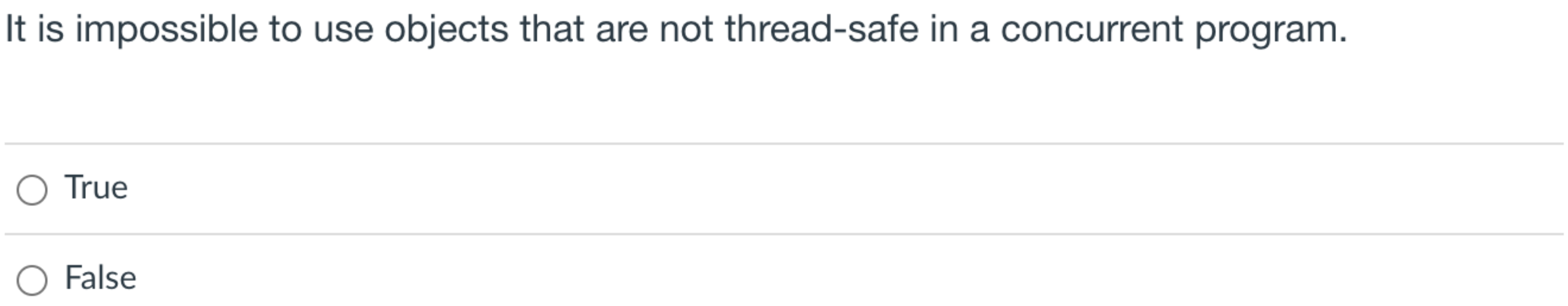
- HW 3 released.

- You will have what you need for Part 1 by the end of today’s lecture

- Due in 10 days, with 3 free late days

Previous quiz + Review

Previous quiz + Review



It is impossible to use objects that are not thread-safe in a concurrent program.

○ True

○ False

global variables:

```
bank_account tylers_account;
mutex m;
```

Tyler's coffee addiction:

```
for (int i = 0; i < HOURS; i++) {
   m.lock();
   tylers_account.buy_coffee();
   m.unlock();
}
```

Tyler's employer

```
for (int j = 0; j < HOURS; j++) {
   m.lock();
   tylers_account.get_paid();
   m.unlock();
}
```

what if you have multiple objects?

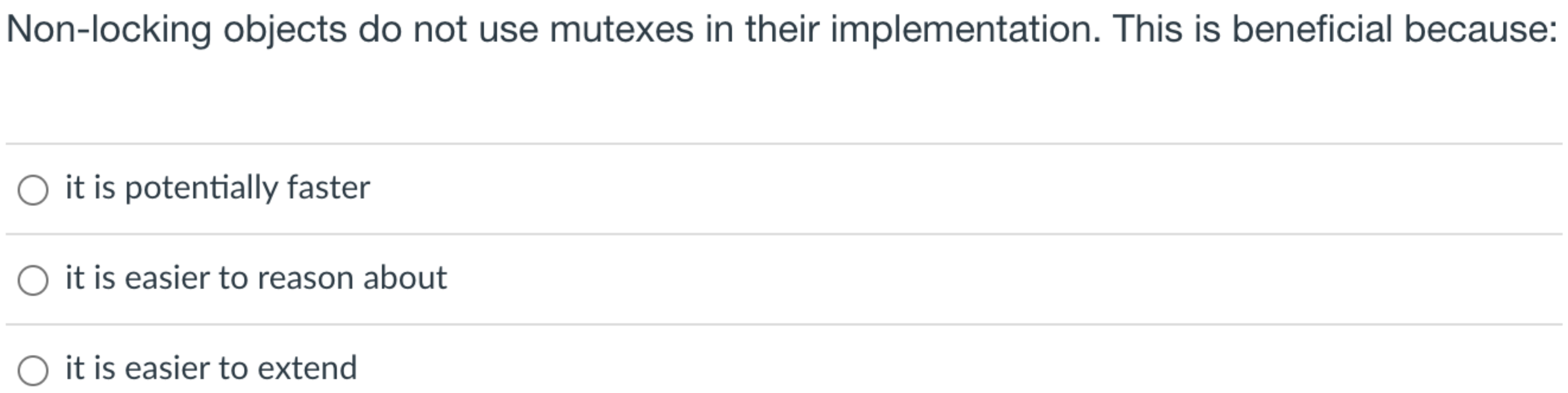
First solution:
The client (user of the object) can use locks.

We might decide to wrap my bank account in an object

```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    void buy_coffee() {
      balance -= 1;
    }

    void get_paid() {
      balance += 1;
    }

  private:
    int balance;
};
```

The object is not "thread safe"

Previous quiz + Review



Non-locking objects do not use mutexes in their implementation. This is beneficial because:

○ it is potentially faster

○ it is easier to reason about

○ it is easier to extend

Bank account example

global variables:

```
bank_account tylers_account;
```

Tyler’s coffee addiction:

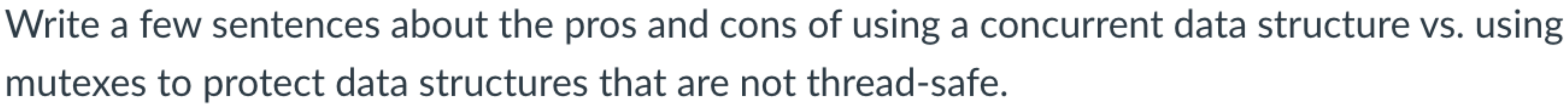
```
for (int i = 0; i < HOURS; i++) {
   tylers_account.buy_coffee();
}
```

Tyler’s employer

```
for (int j = 0; j < HOURS; j++) {
   tylers_account.get_paid();
}
```

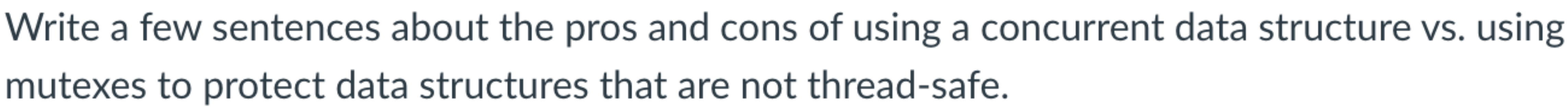
```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    void buy_coffee() {
      atomic_fetch_add(&balance, -1);
    }

    void get_paid() {
      atomic_fetch_add(&balance, 1);
    }

  private:
    atomic_int balance;
};
```

Previous quiz + Review



Write a few sentences about the pros and cons of using a concurrent data structure vs. using mutexes to protect data structures that are not thread-safe.

Previous quiz + Review



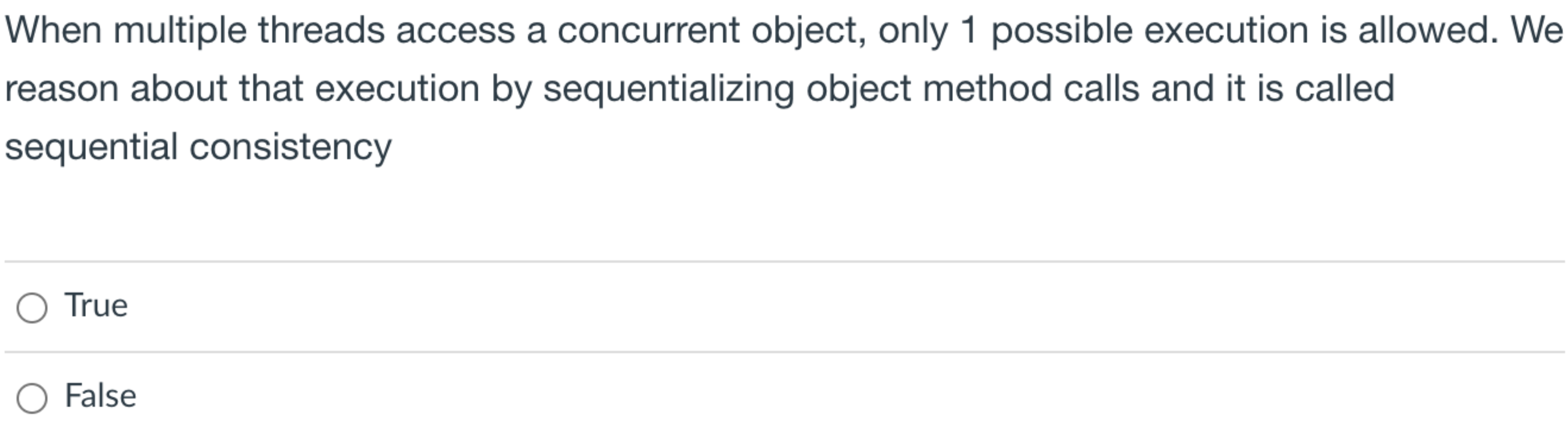
Write a few sentences about the pros and cons of using a concurrent data structure vs. using mutexes to protect data structures that are not thread-safe.

Pros: Easier to use
Cons: Compsability

Previous quiz + Review

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

Previous quiz + Review



When multiple threads access a concurrent object, only 1 possible execution is allowed. We reason about that execution by sequentializing object method calls and it is called sequential consistency

○ True

○ False

Global variable:

```
CQueue<int> q;
```

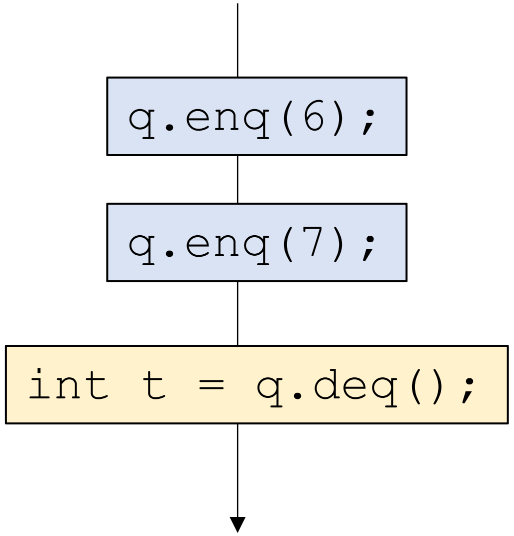
Thread 0:

```
q.enq(6);
q.enq(7);
```

Thread 1:

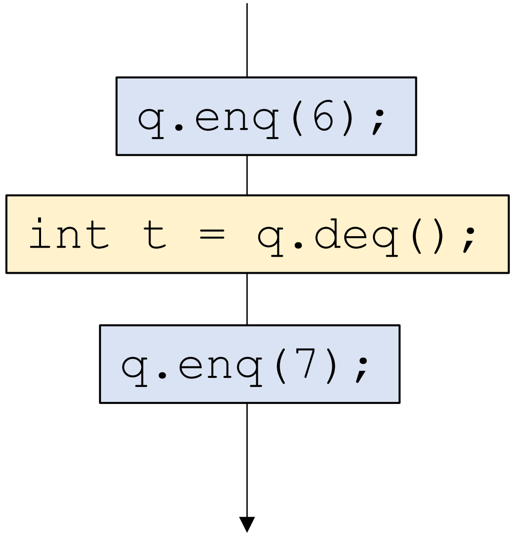
```
int t = q.deq();
```

Construct a sequential timeline of API calls
Any sequence is valid:



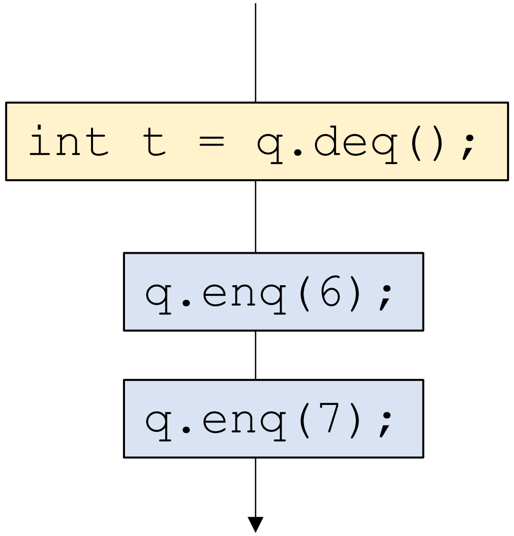
```
q.enq(6);
q.enq(7);
int t = q.deq();
```

t is 6



```
q.enq(6);
int t = q.deq();
q.enq(7);
```

t is 6

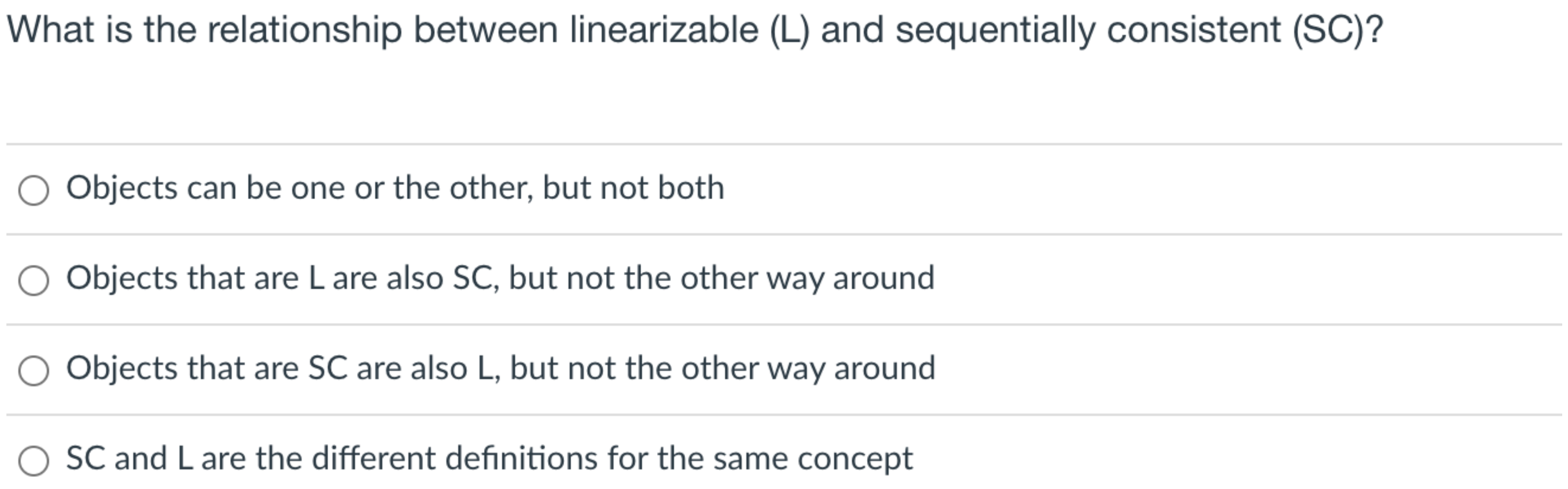


```
int t = q.deq();
q.enq(6);
q.enq(7);
```

t is None

Can t ever be 7?

Previous quiz + Review



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

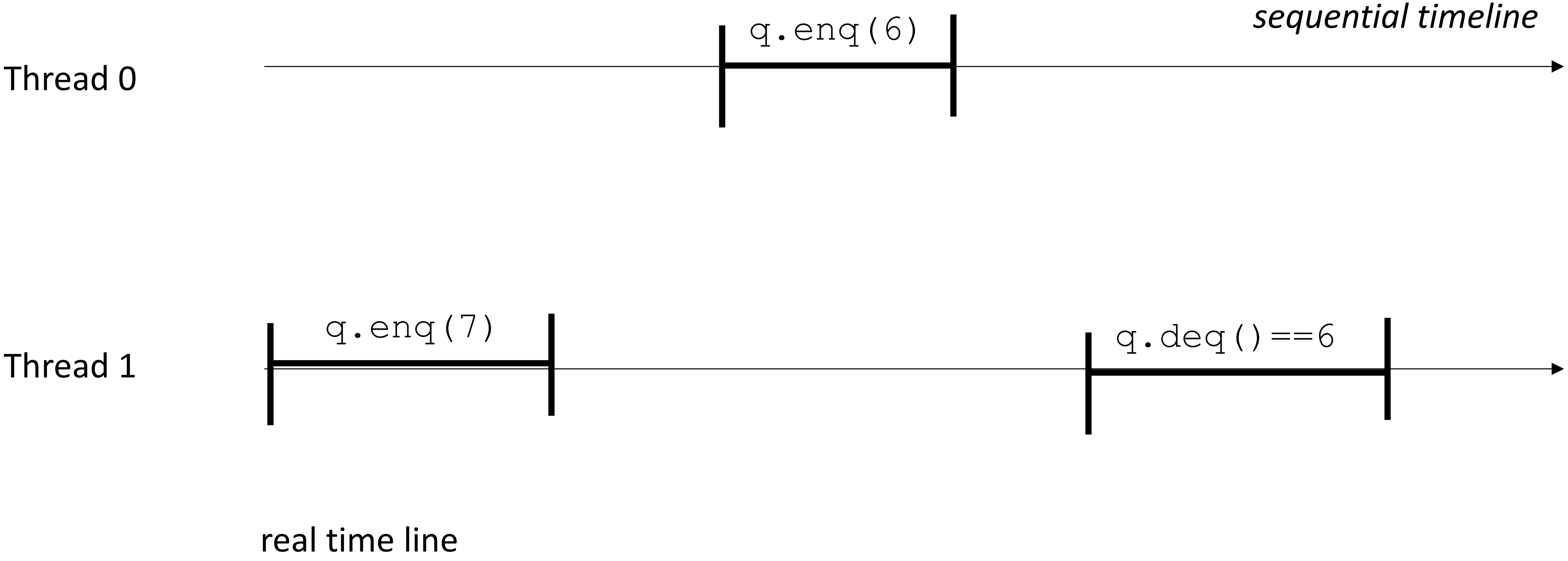
Review

Sequential consistency and real time

- Add in real time:

This execution is allowed in sequential consistency!

SC doesn't care about real time, only if it can construct its virtual sequential timeline

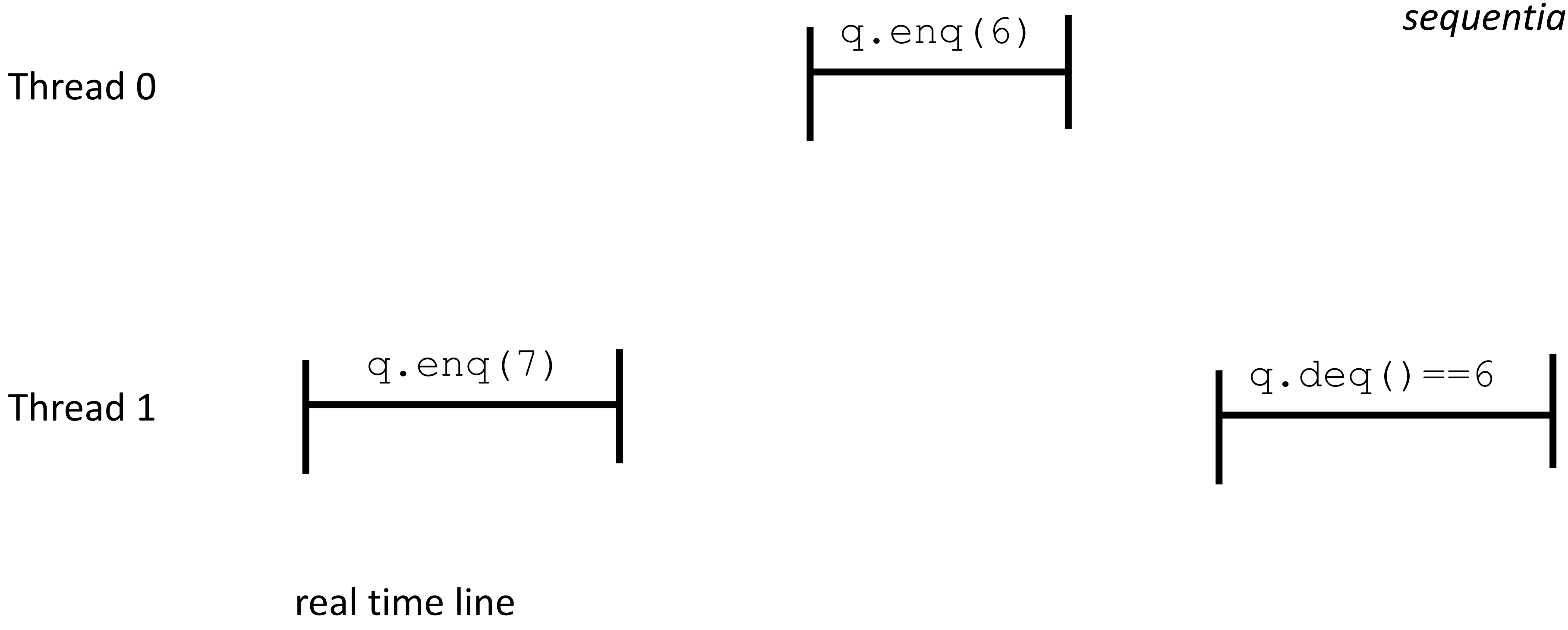


Sequential consistency and real time

- Add in real time:

This execution is allowed in sequential consistency!

SC doesn't care about real time, only if it can construct its virtual sequential timeline



Thread 0

`q.enq(6)`

Thread 1

`q.enq(7)`

`q.deq()==6`

real time line

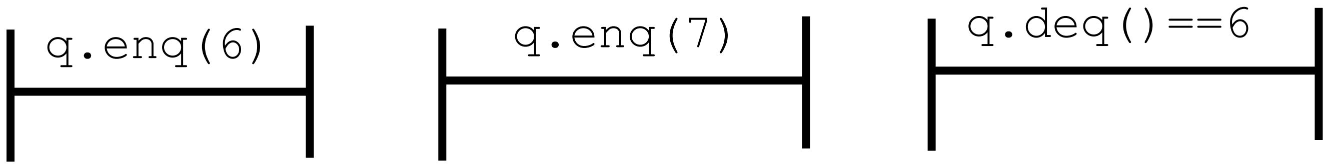
Sequential consistency and real time

- Add in real time:

This execution is allowed in sequential consistency!

SC doesn't care about real time, only if it can construct its virtual sequential timeline

Thread 0

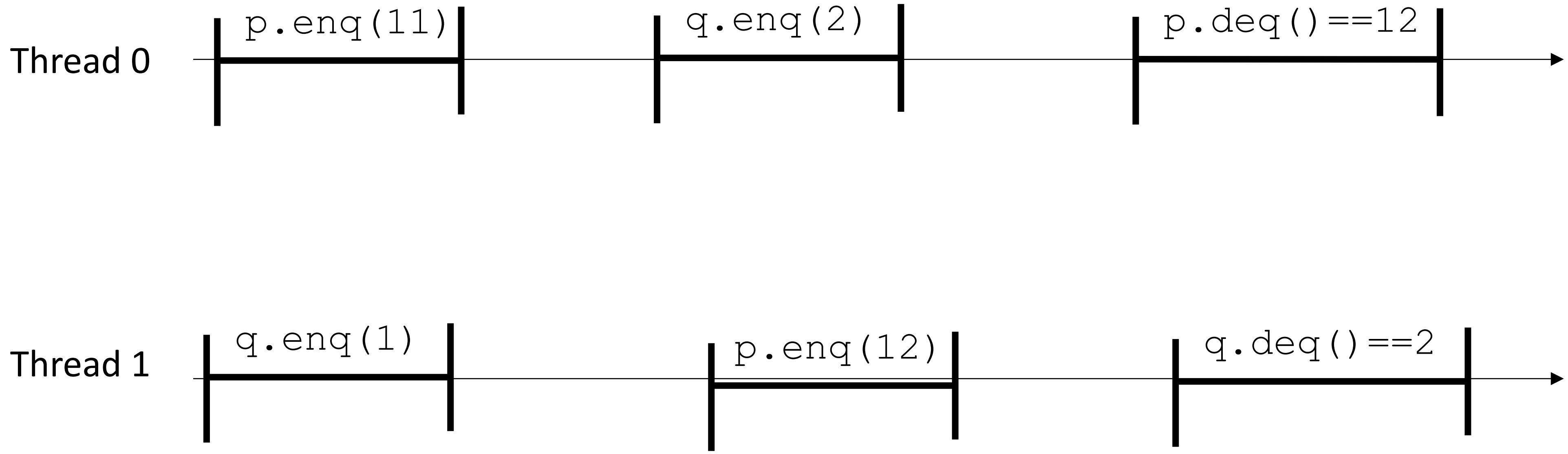


Thread 1

Sequential consistency and real time

- Add in real time:

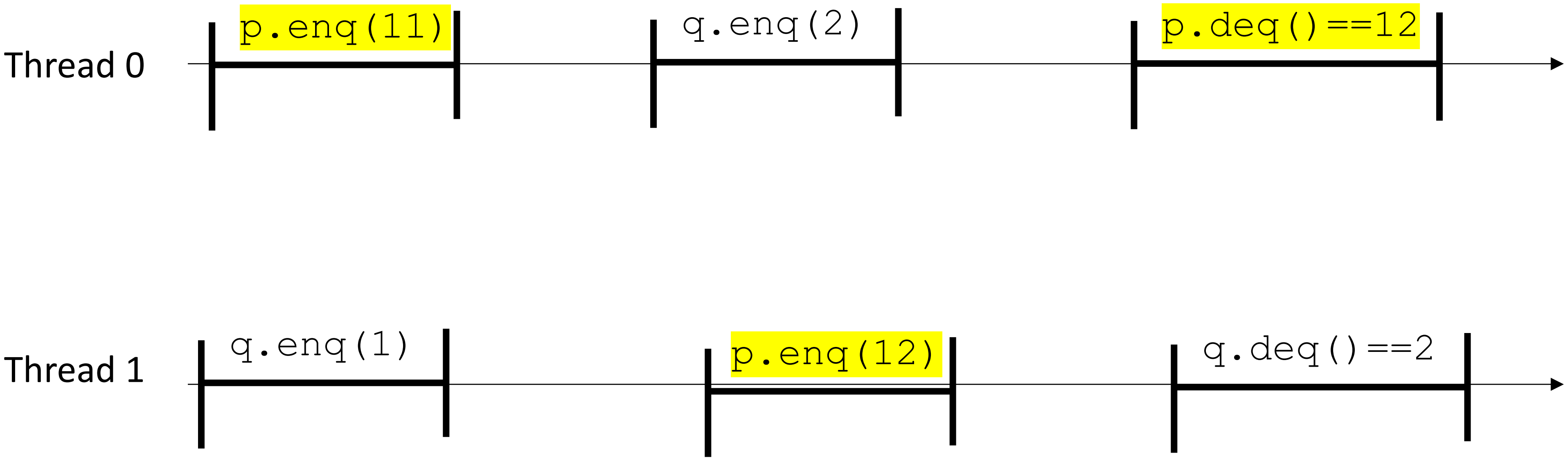
2 objects now: p and q



Sequential consistency and real time

- Add in real time:

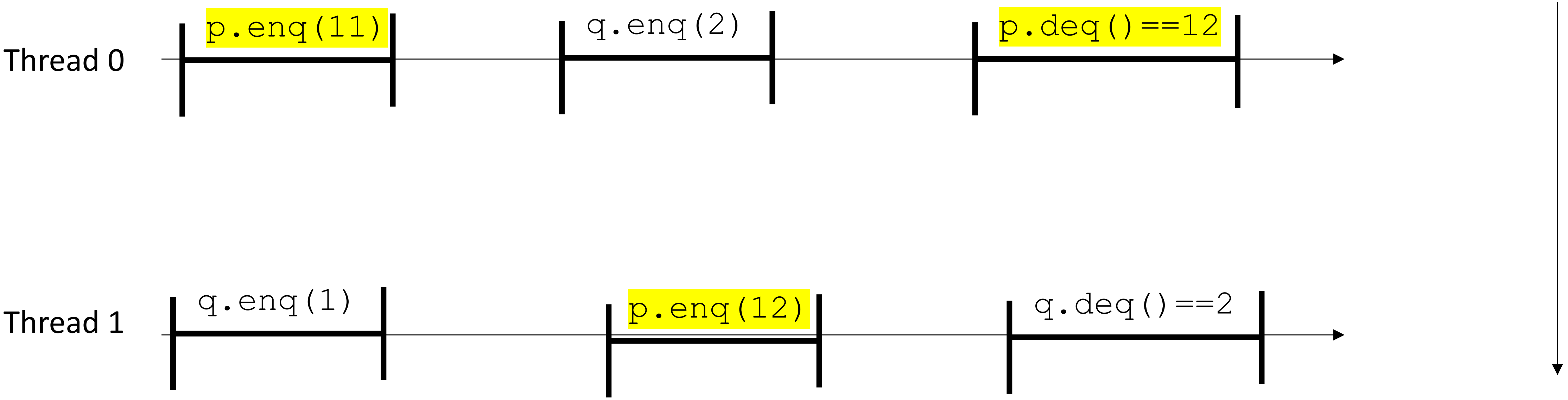
2 objects now: p and q
Consider each object in isolation



Sequential consistency and real time

- Add in real time:

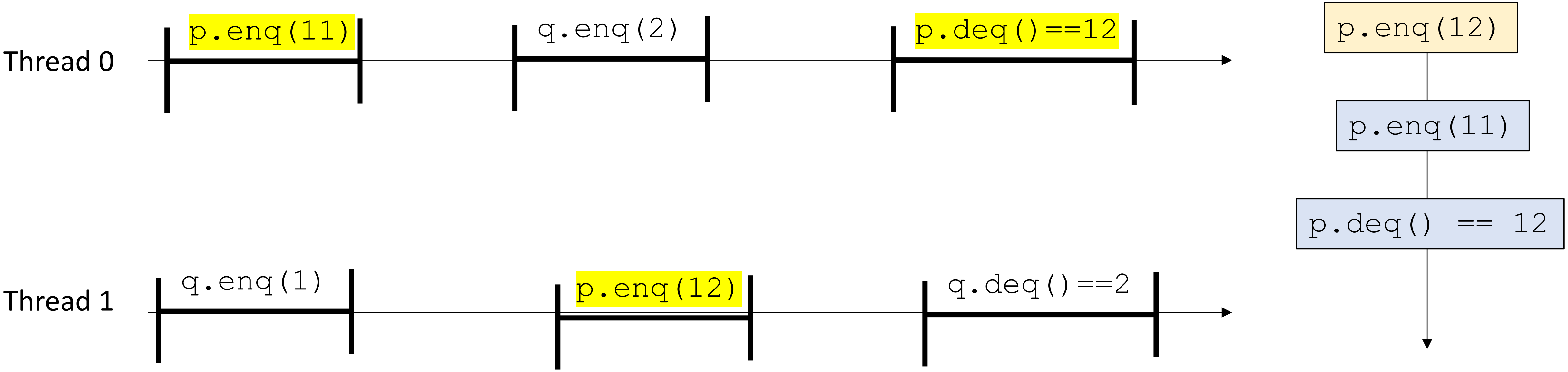
2 objects now: p and q
Consider each object in isolation



Sequential consistency and real time

- Add in real time:

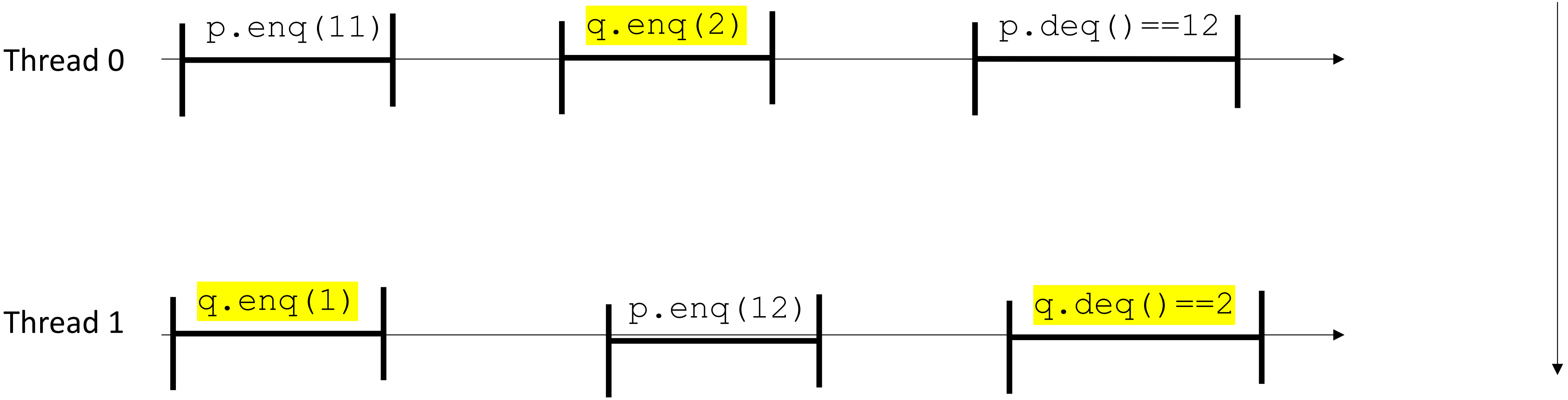
2 objects now: p and q
Consider each object in isolation



Sequential consistency and real time

- Add in real time:

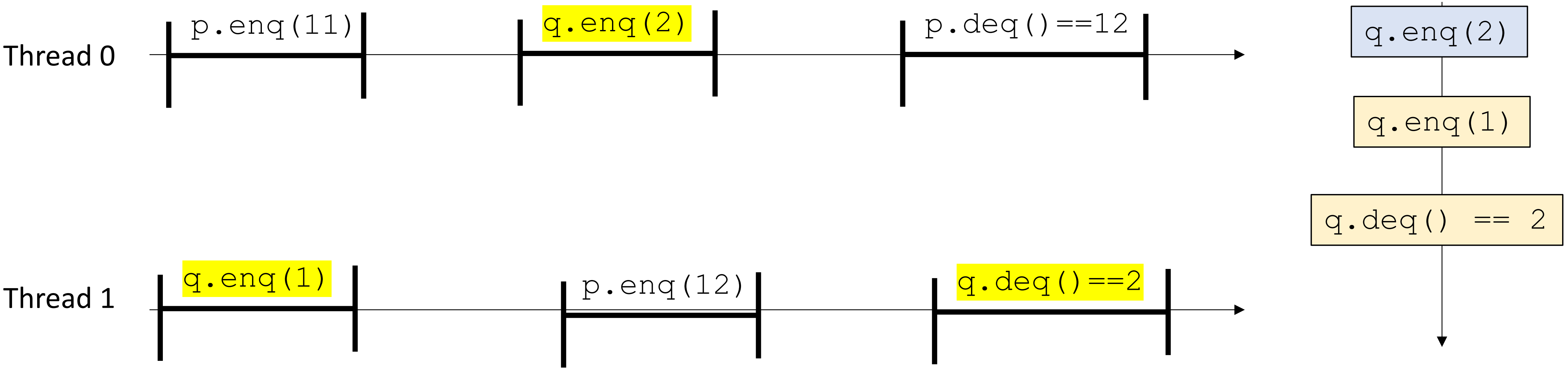
2 objects now: p and q
Consider each object in isolation



Sequential consistency and real time

- Add in real time:

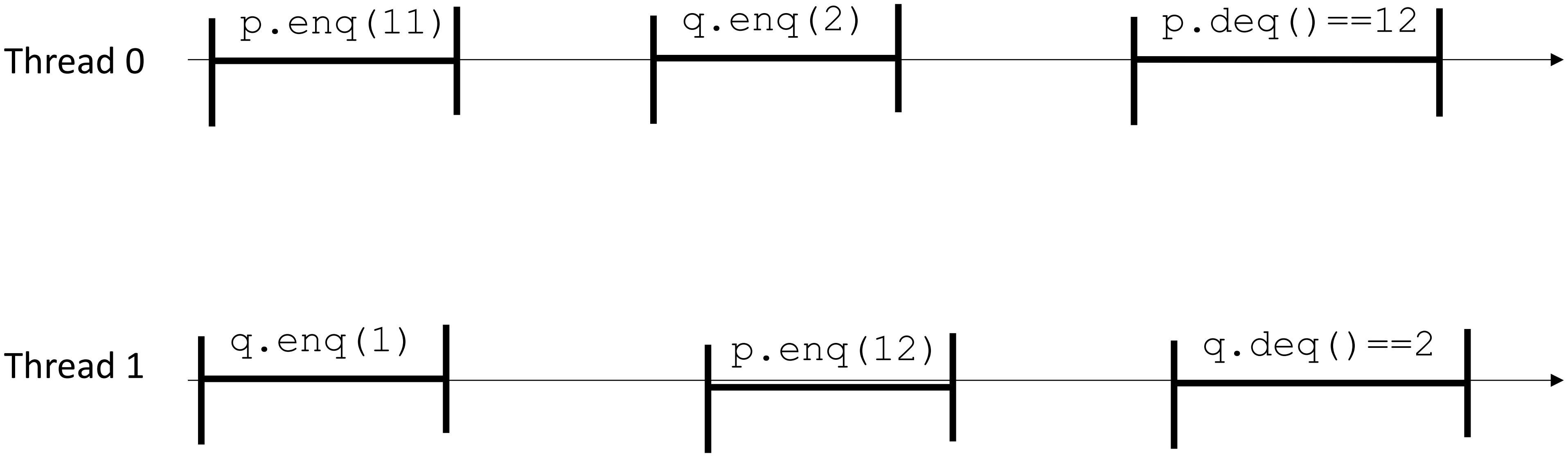
2 objects now: p and q
Consider each object in isolation



Sequential consistency and real time

- Add in real time:

Now consider them all together



Global variable:

```
CQueue<int> p,q;
```

Thread 0:

```
p.enq(11)
q.enq(2)
p.deq()==12
```

Order 1

```
p.enq(12)

p.enq(11)

p.deq()  == 12
```

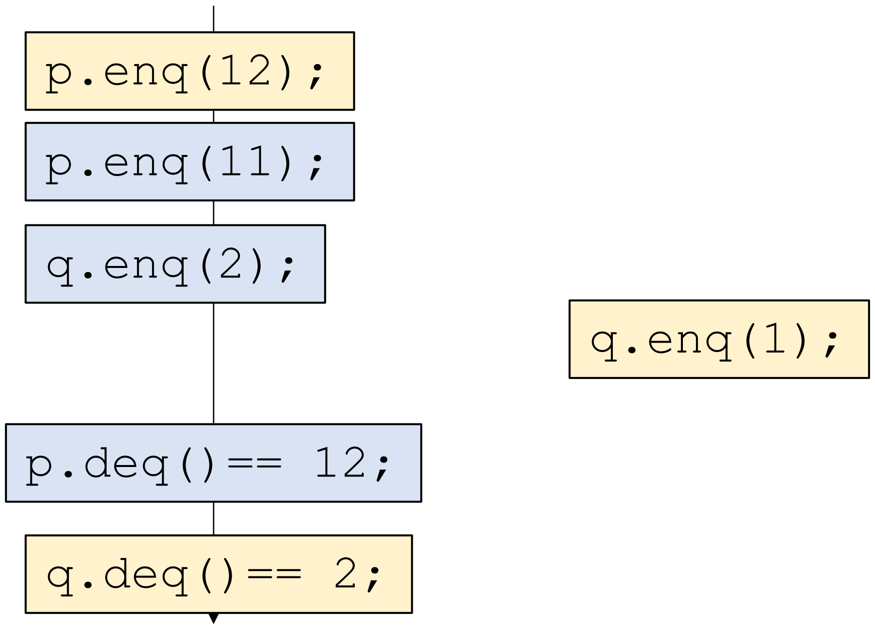
Order 2

```
q.enq(2)

q.enq(1)

q.deq()  == 2
```

Thread 1:

```
q.enq(1)
p.enq(12)
q.deq()==2
```

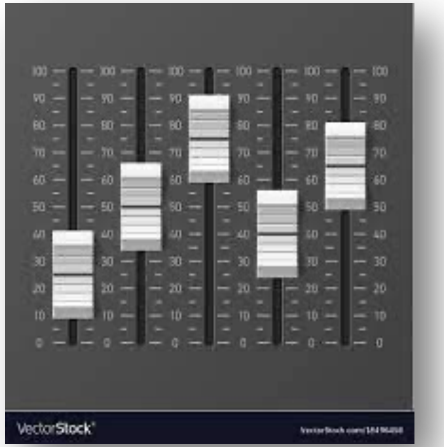
Combine



```
p.enq(12);
p.enq(11);
q.enq(2);
q.enq(1);
p.deq()== 12;
q.deq()== 2;
```

Linearizability

- Linearizability
  - Defined in term of real-time histories
  - We want to ask if an execution is allowed under linearizability
- Slightly different game:
  - Sequential consistency is a game about stacking lego bricks
  - Linearizability is about sliders



Linearizability

Each operation has a linearizability point

- does not overlap with other with other linearizability points
- indivisible computation (critical section, atomic RMW, atomic load, atomic store)
- object update (or read) occurs exactly at this point

●

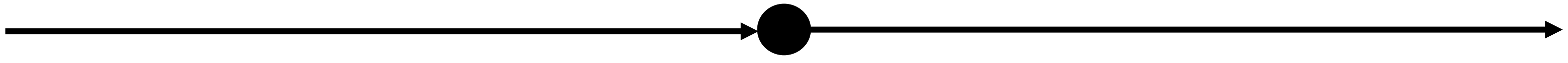
Linearizability

each operation has a linearizability point

- does not overlap with other with other linearizability points

- indivisible computation (critical section, atomic RMW, atomic load, atomic store)

- object update (or read) occurs exactly at this point



object state: M

object state: M’

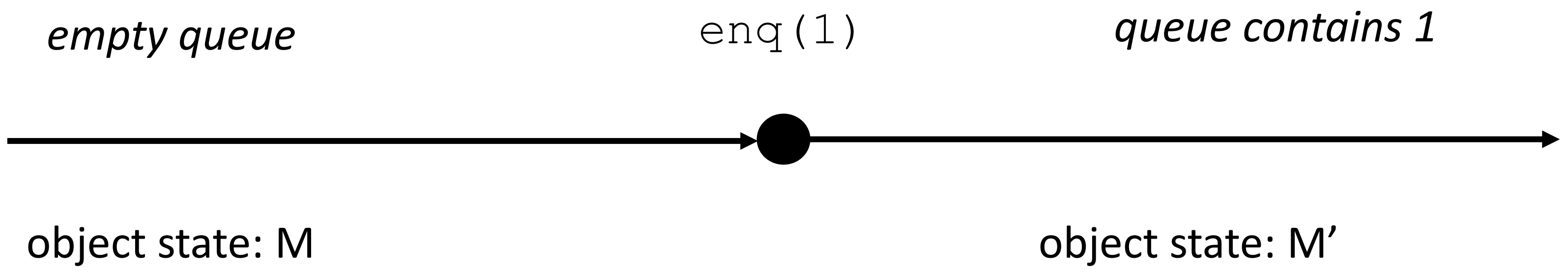
Linearizability

each operation has a linearizability point

- does not overlap with other with other linearizability points

- indivisible computation (critical section, atomic RMW, atomic load, atomic store)

- object update (or read) occurs exactly at this point



empty queue

`enq(1)`

queue contains 1

object state: M

object state: M’

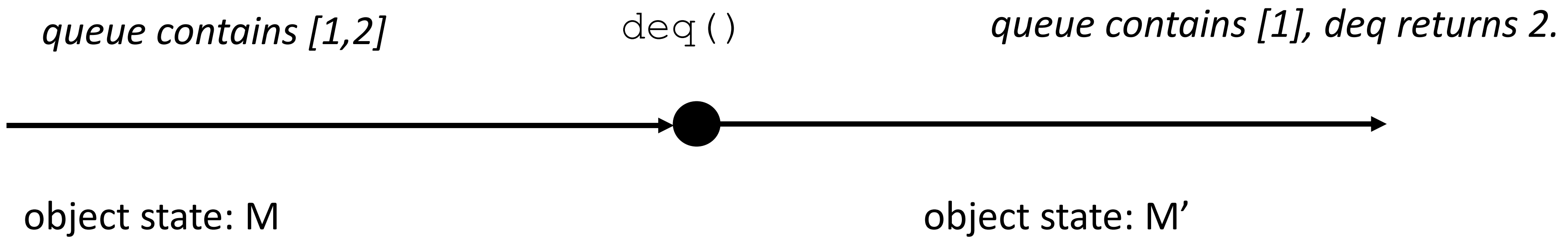
Linearizability

each operation has a linearizability point

- does not overlap with other with other linearizability points

- indivisible computation (critical section, atomic RMW, atomic load, atomic store)

- object update (or read) occurs exactly at this point



queue contains [1,2]

`deq()`

queue contains [1], deq returns 2.

object state: M

object state: M’

Linearizability

each operation has a linearizability point

- does not overlap with other with other linearizability points

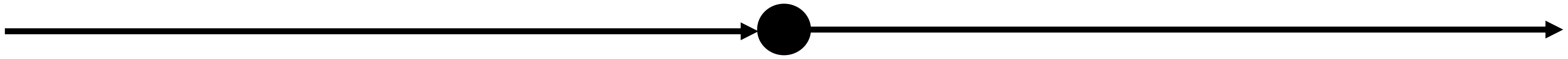
- indivisible computation (critical section, atomic RMW, atomic load, atomic store)

- object update (or read) occurs exactly at this point

queue contains [1,2]

`peek()`

return value from M, i.e. 2



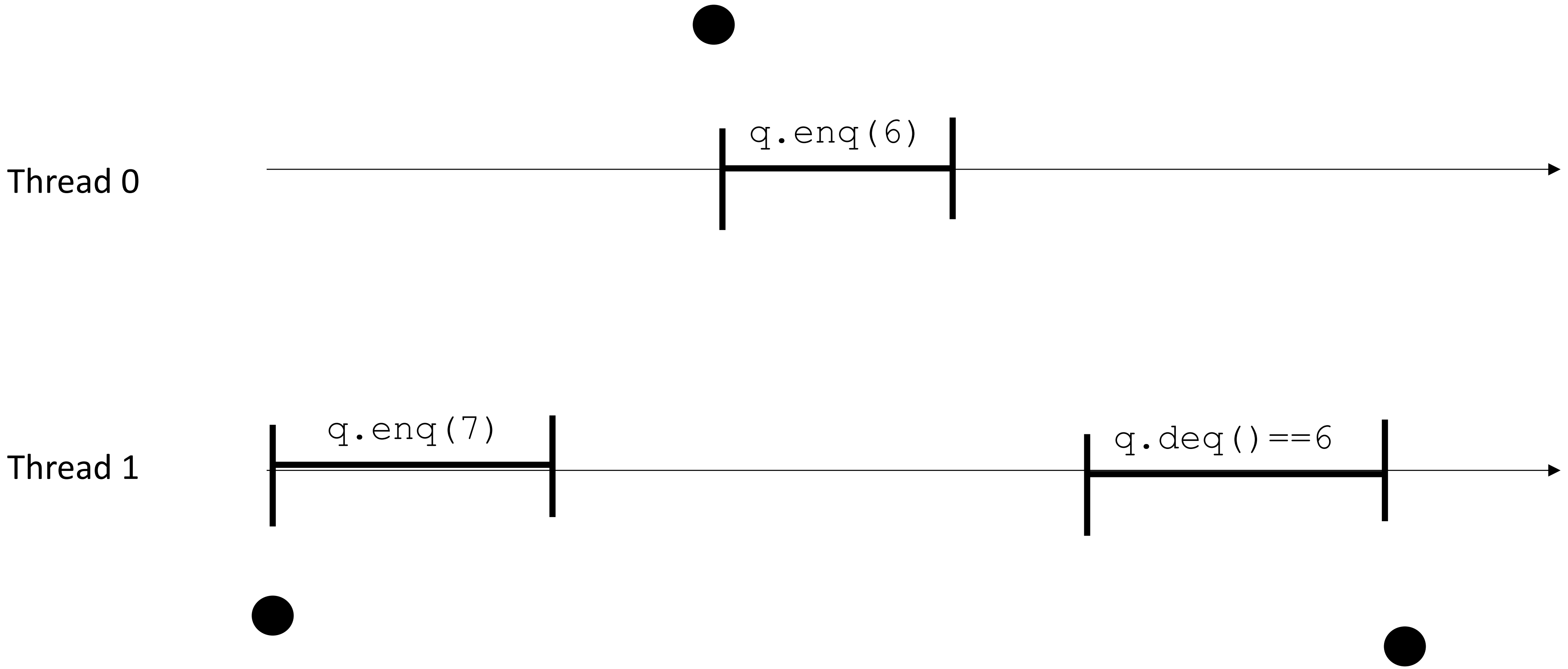
object state: M

object state: M’

Linearizability

each command gets a linearization point.

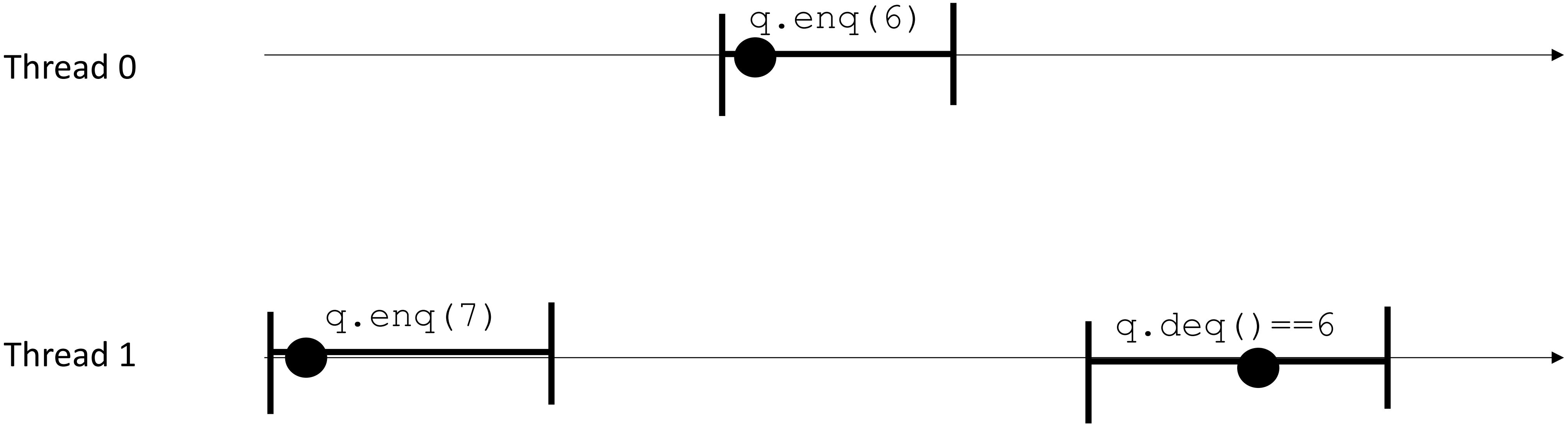
You can place the point any where between its innovation and response!



Linearizability

each command gets a linearization point.

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

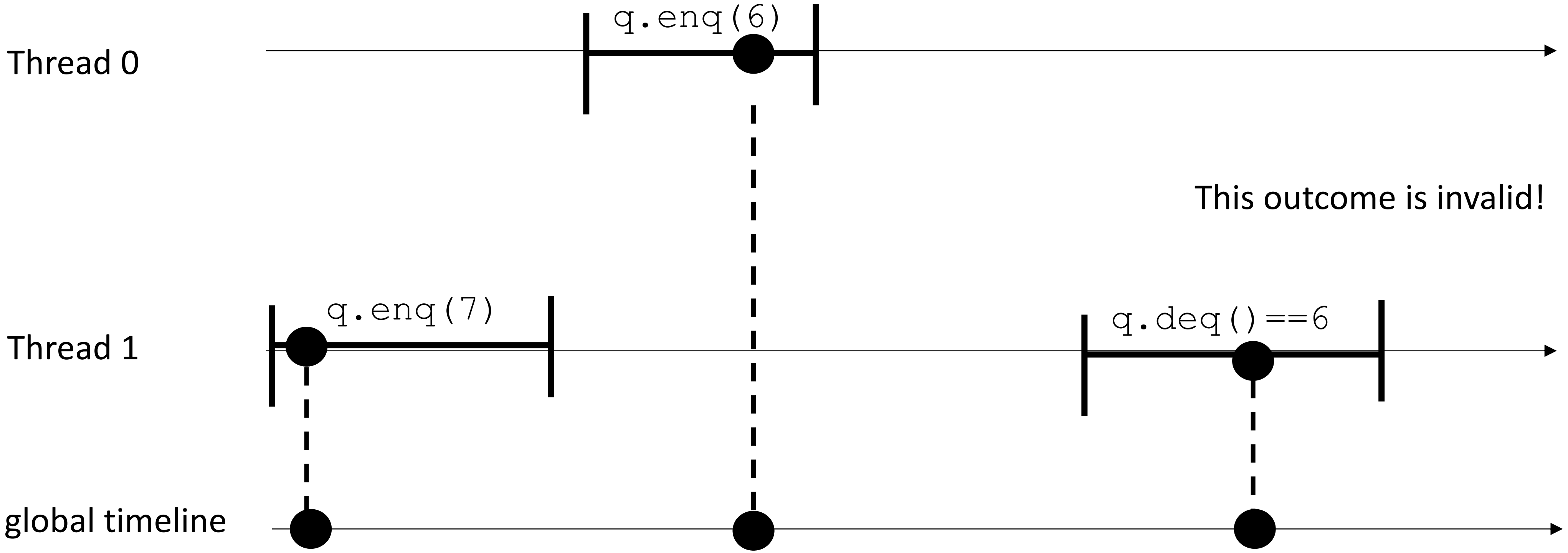


Linearizability

each command gets a linearization point.

You can place the point any where between its innovation and response (so long as they don't overlap)!

Project the linearization points to a global timeline



This outcome is invalid!

reason sequentially!

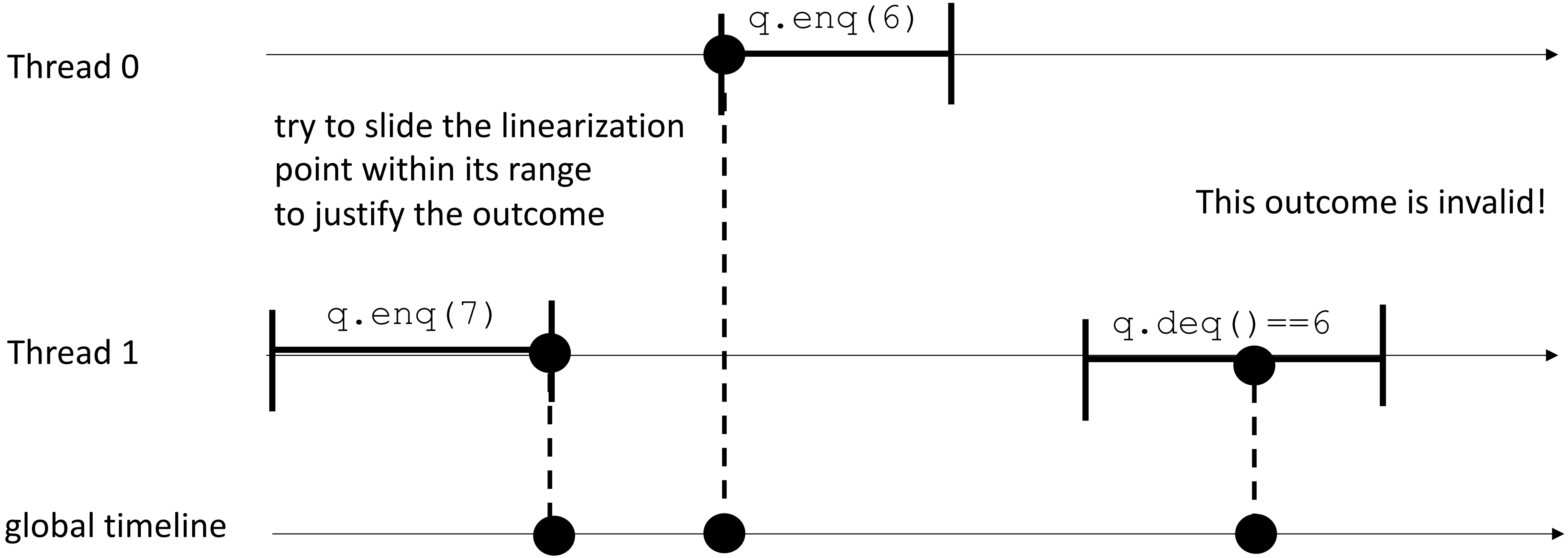
Linearizability

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!

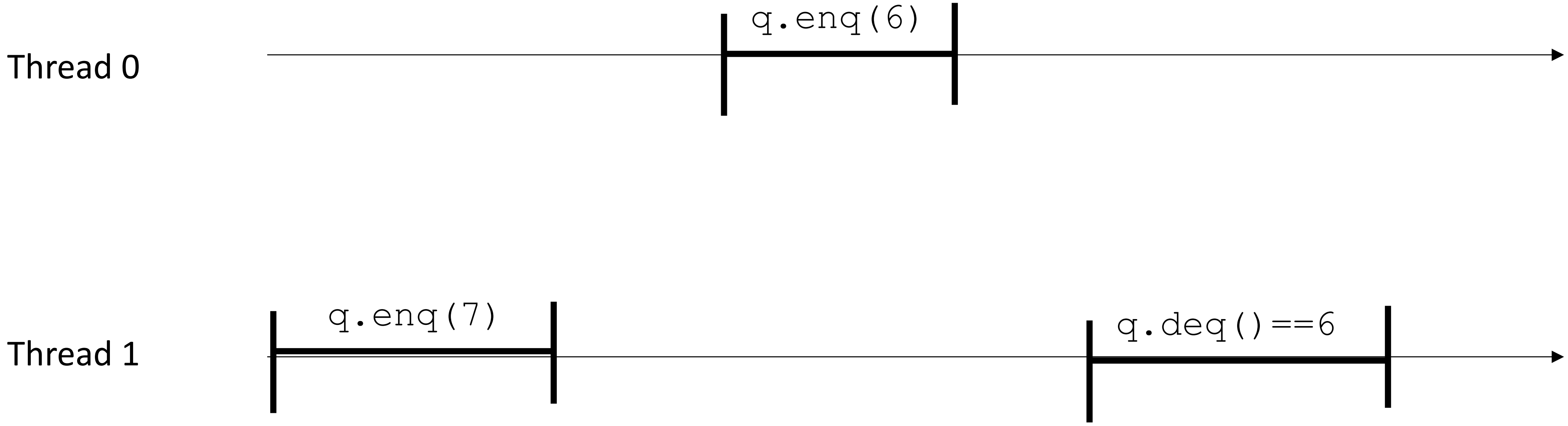


try to slide the linearization point within its range to justify the outcome

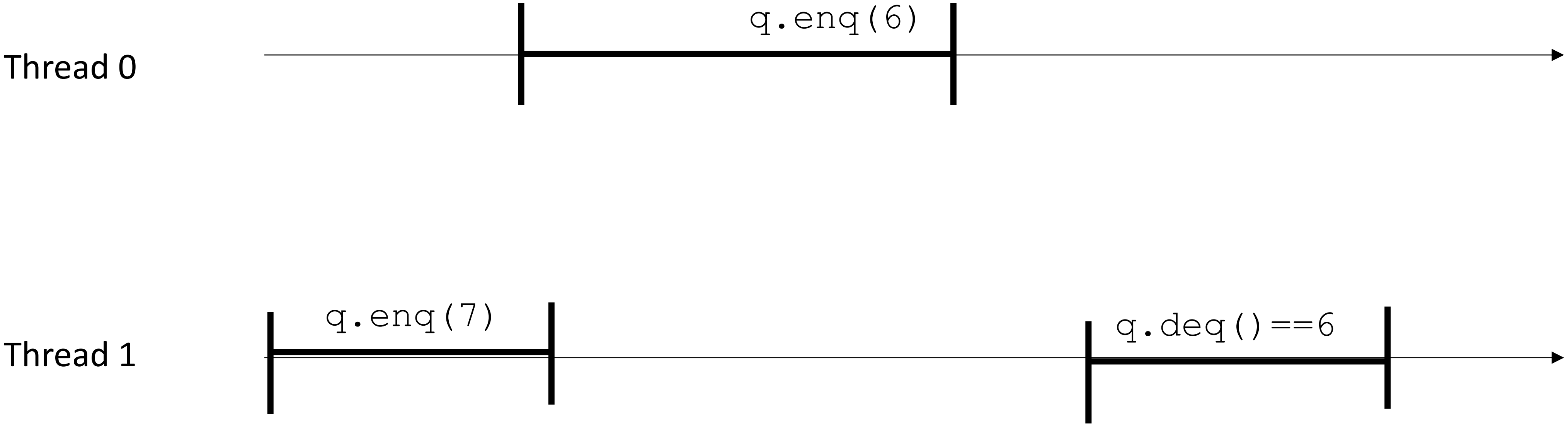
This outcome is invalid!

reason sequentially!

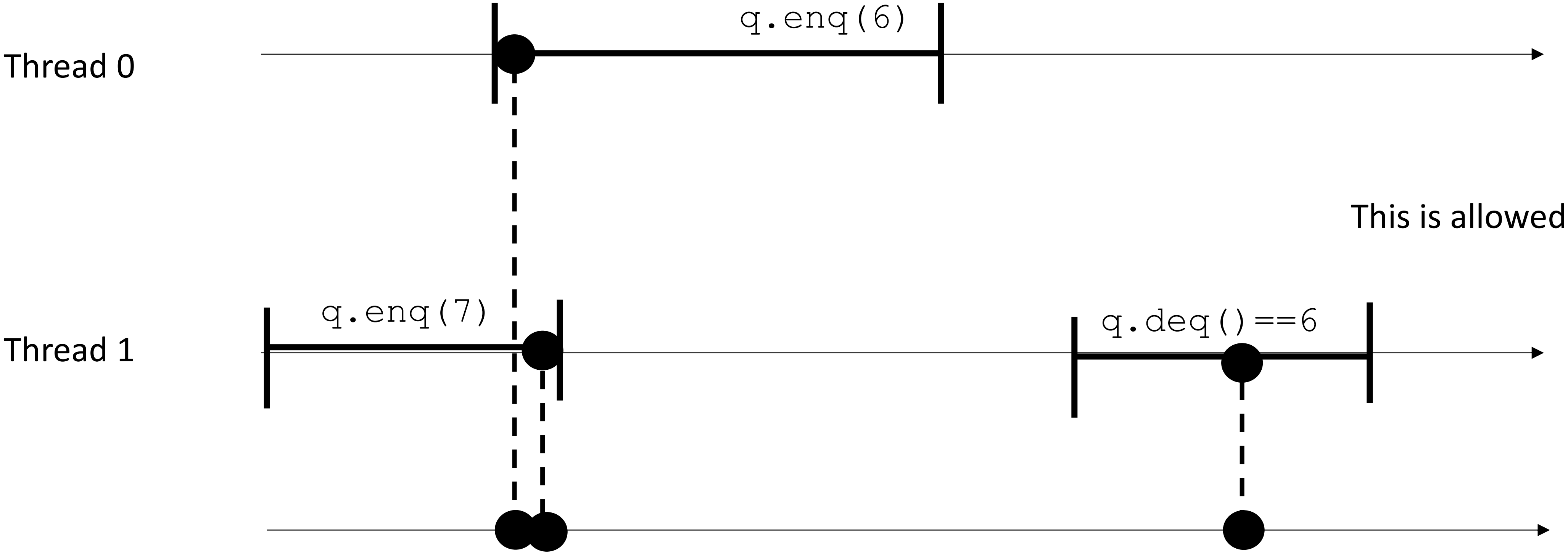
Linearizability



Linearizability



Linearizability



Thread 0

q.enq(6)

Thread 1

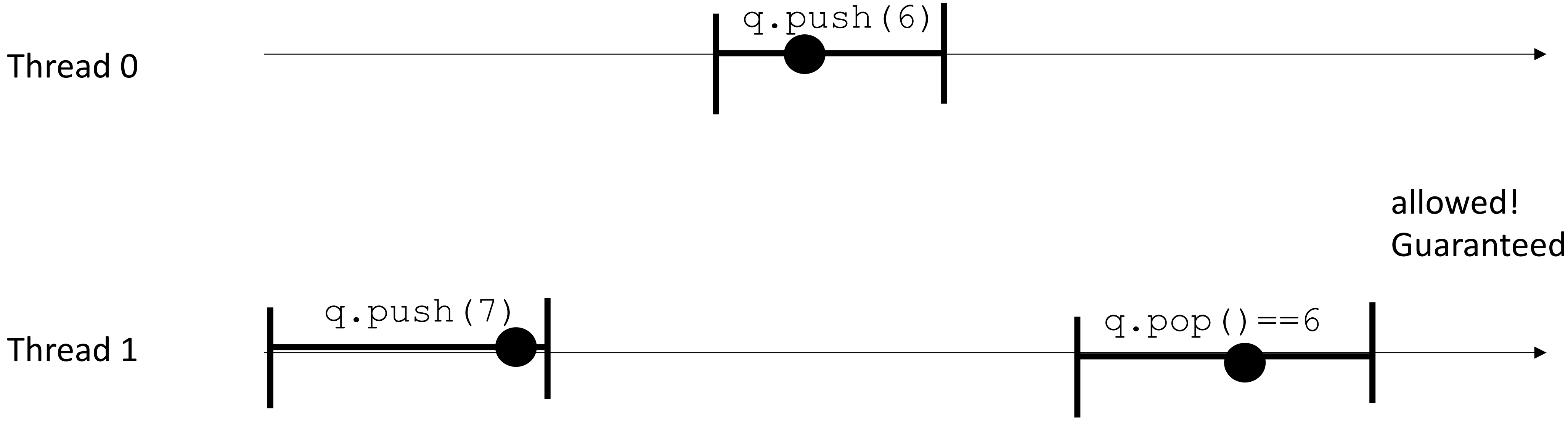
q.enq(7)

q.deq()==6

This is allowed now!

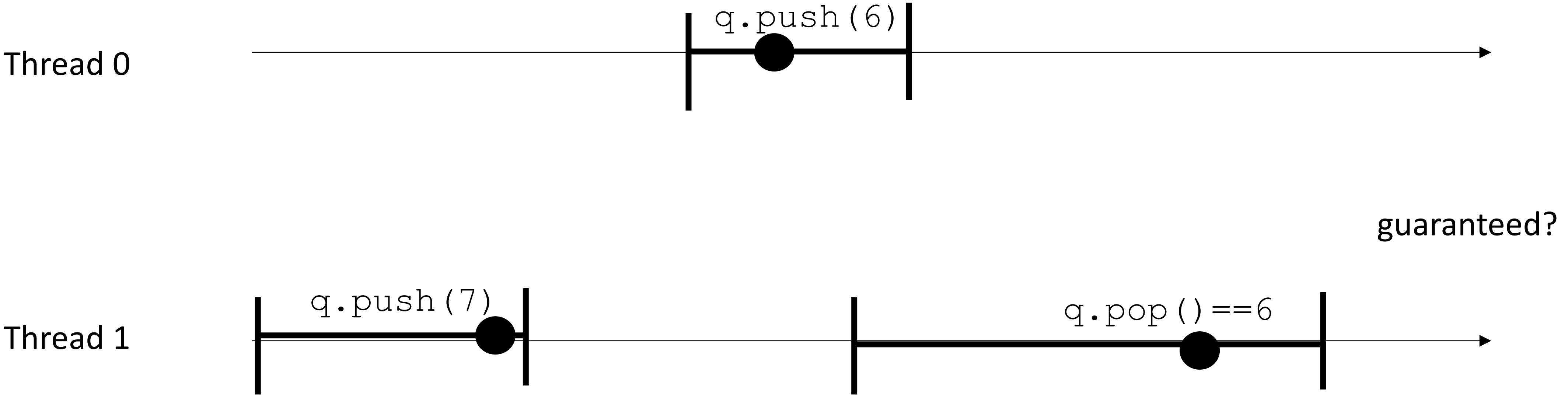
Linearizability

Stack

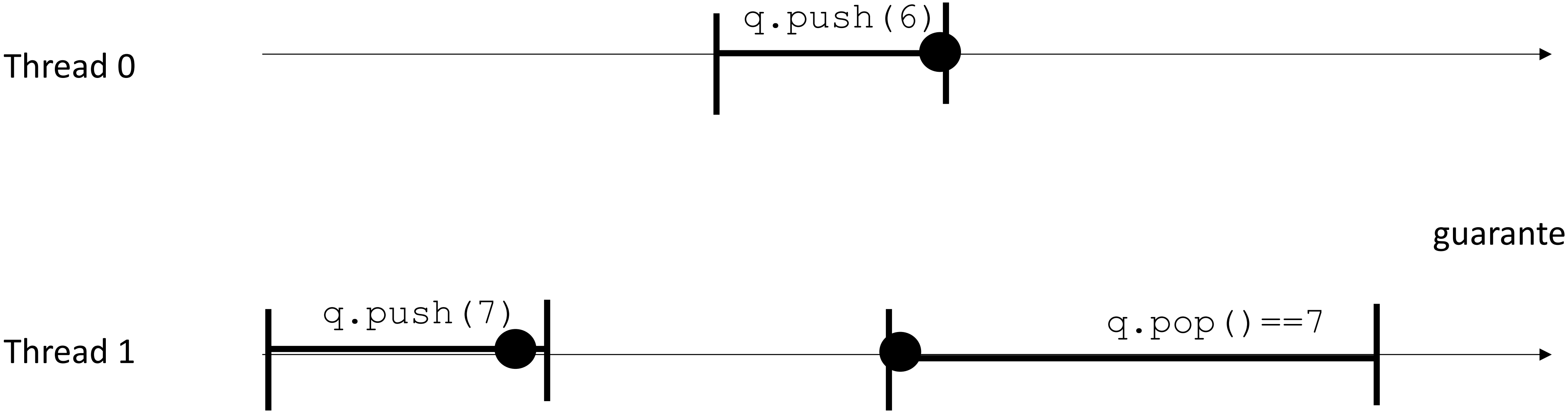


allowed!
Guaranteed? Yes

Linearizability



Linearizability



Thread 0

q.push(6)

Thread 1

q.push(7)

q.pop()==7

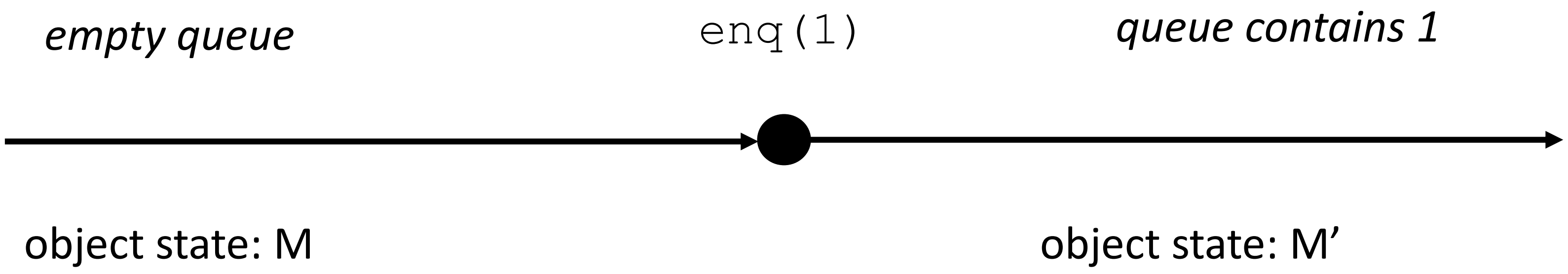
guaranteed? No

Linearizability

- We spent a bunch of time on SC... did we waste our time?
    - No!
    - Linearizability is strictly stronger than SC. Every linearizable execution is SC, but not the other way around.
    - If a behavior is disallowed under SC, it is also disallowed under linearizability.

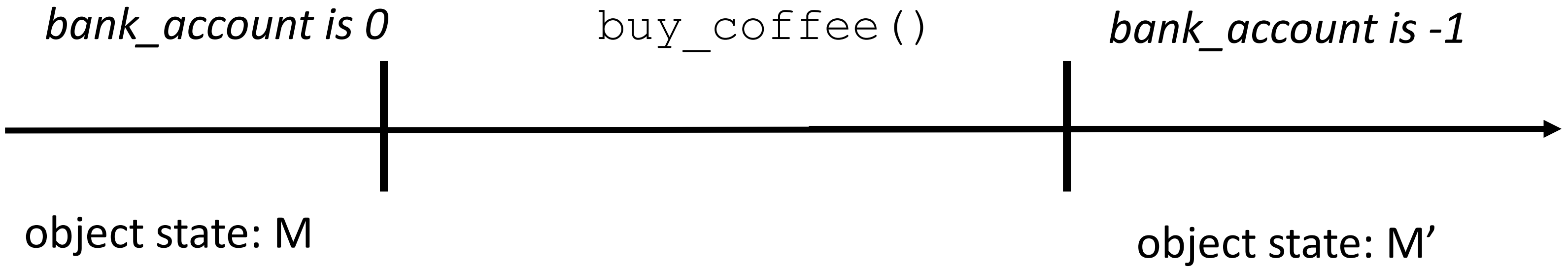
Linearizability

- How do we write our programs to be linearizable?
  - Identify the linearizability point
  - One indivisible region (e.g. an atomic store, atomic load, atomic RMW, or critical section) where the method call takes effect. Modeled as a point.



Linearizability

- Locked data structures are linearizable.

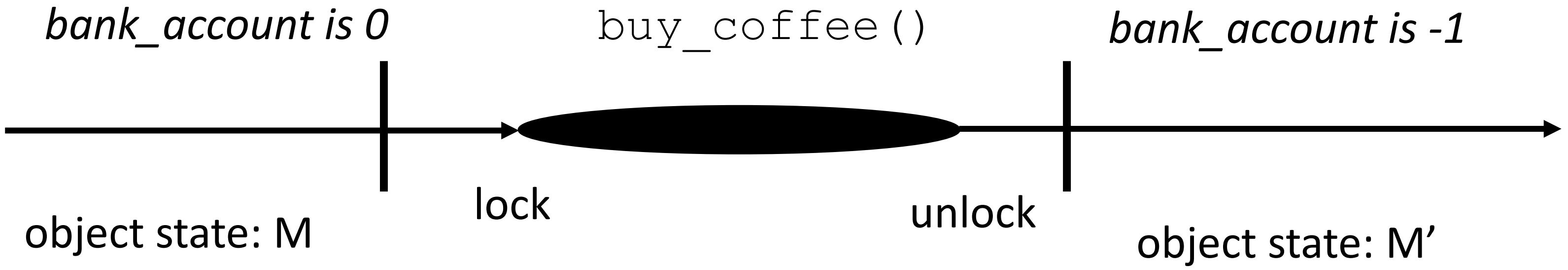


```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    void buy_coffee() {
      m.lock();
      balance -= 1;
      m.unlock();
    }

    void get_paid() {
      m.lock();
      balance += 1;
      m.unlock();
    }

  private:
    int balance;
    mutex m;
};
```

Linearizability

- Locked data structures are linearizable.

typically modeled as the point the lock is acquired or released



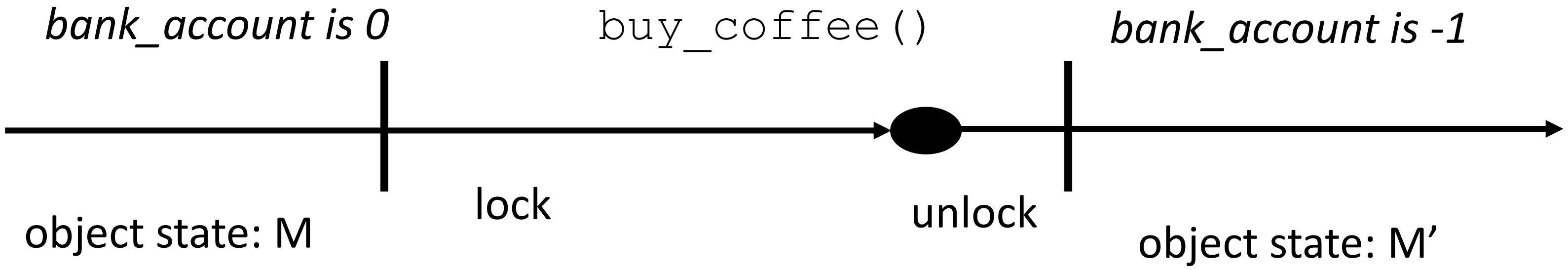
```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    void buy_coffee() {
      m.lock();
      balance -= 1;
      m.unlock();
    }

    void get_paid() {
      m.lock();
      balance += 1;
      m.unlock();
    }

  private:
    int balance;
    mutex m;
};
```

Linearizability

- Locked data structures are linearizable.

```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    void buy_coffee() {
      m.lock();
      balance -= 1;
      m.unlock();
    }

    void get_paid() {
      m.lock();
      balance += 1;
      m.unlock();
    }

  private:
    int balance;
    mutex m;
};
```

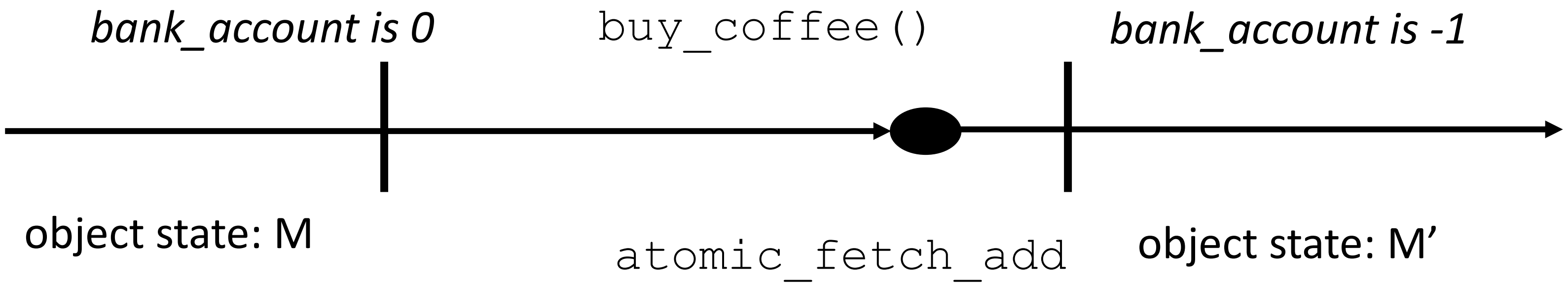
typically modeled as the point the lock is acquired or released lets say released.



Linearizability

- Our lock-free bank account is linearizable:
    - The atomic operation is the linearizable point

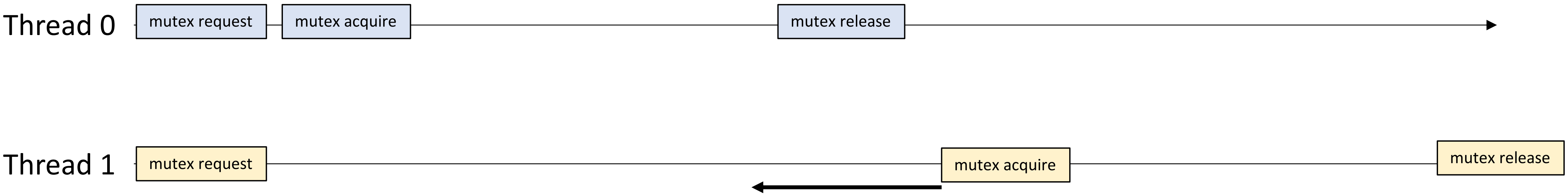
```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    void buy_coffee() {
      atomic_fetch_add(&balance, -1);
    }

    void get_paid() {
      atomic_fetch_add(&balance, 1);
    }

  private:
    atomic_int balance;
};
```



Progress properties

- Going back to specifications:

Recall the mutex



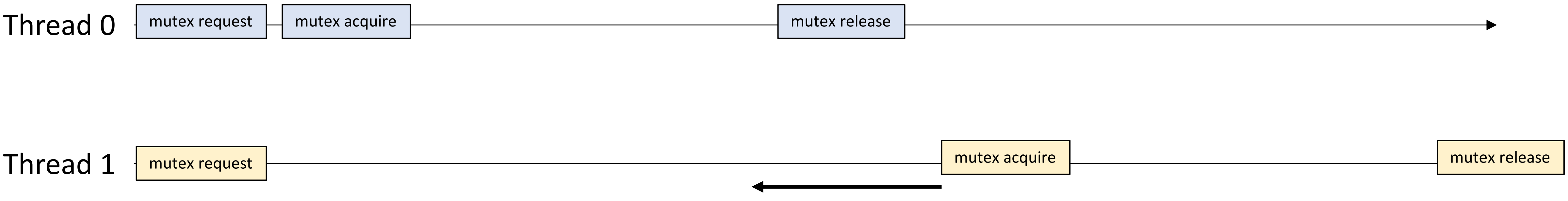
what is stopping this?

Progress properties

- Going back to specifications:

Thread 0 is stopping Thread 1 from making progress.
If delays in one thread can cause delays in other threads, we say that it is blocking

Recall the mutex



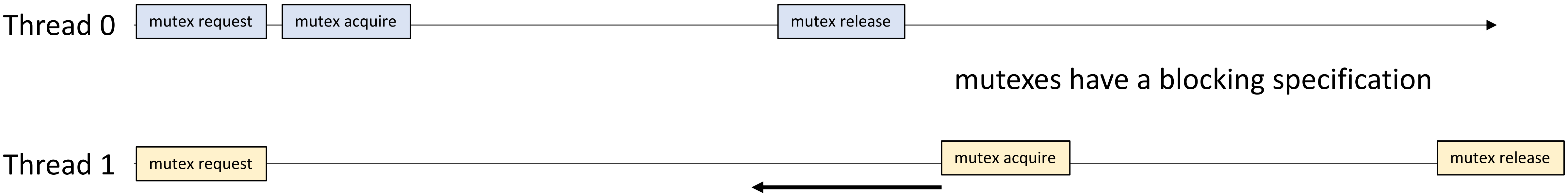
what is stopping this?

Progress properties

- Going back to specifications:

Recall the mutex

Thread 0 is stopping Thread 1 from making progress.
If delays in one thread can cause delays in other threads, we say that it is blocking



mutexes have a blocking specification

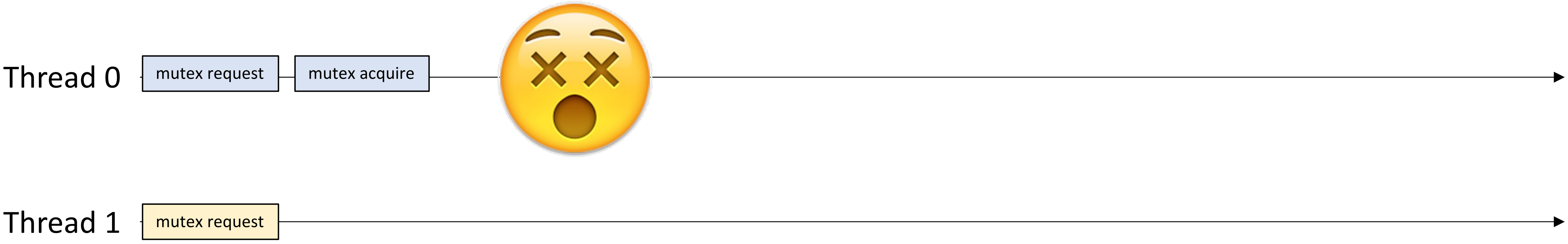
what is stopping this?

Progress properties

- Going back to specifications:

Thread 0 is stopping Thread 1 from making progress.
If delays in one thread can cause delays in other threads, we say that it is blocking

Recall the mutex



Thread 0

mutex request

mutex acquire

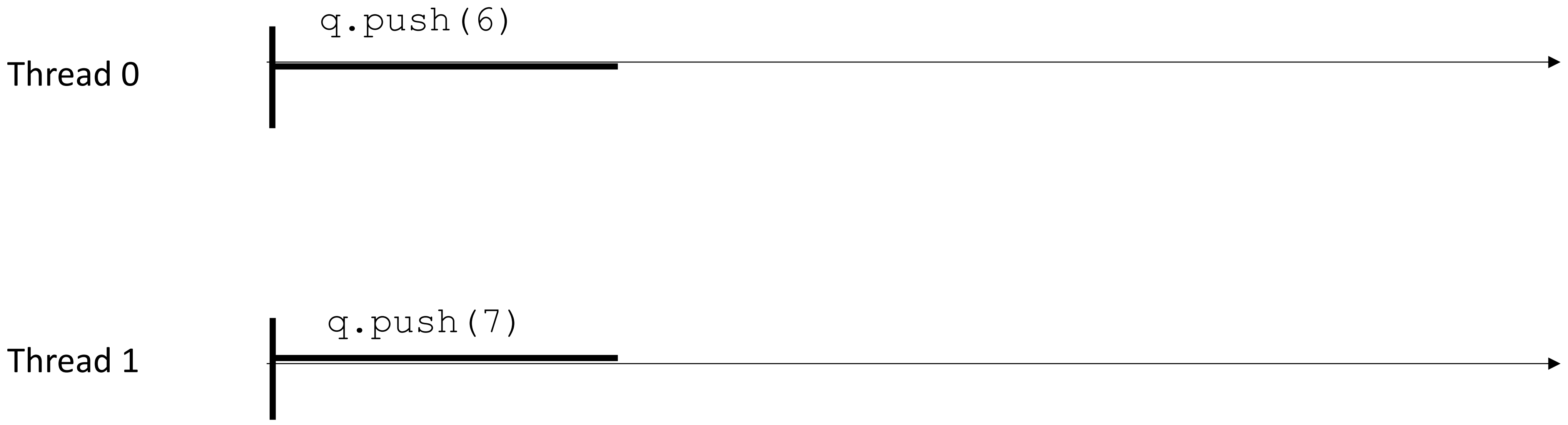
Thread 1

mutex request

What now?!

Linearizability

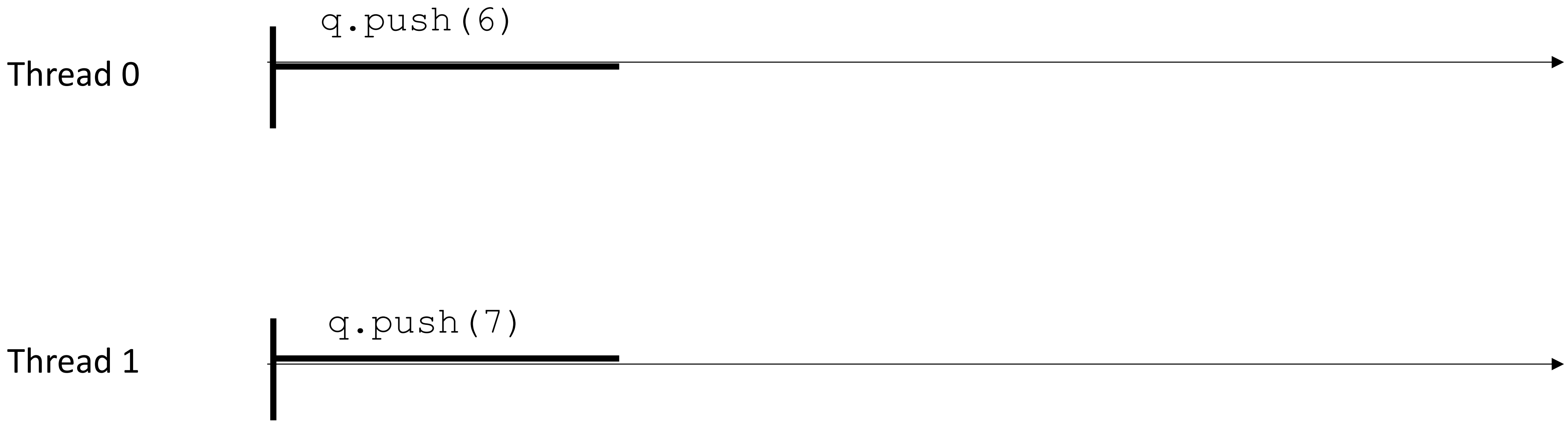
Two unfinished commands.



Linearizability

Two unfinished commands.

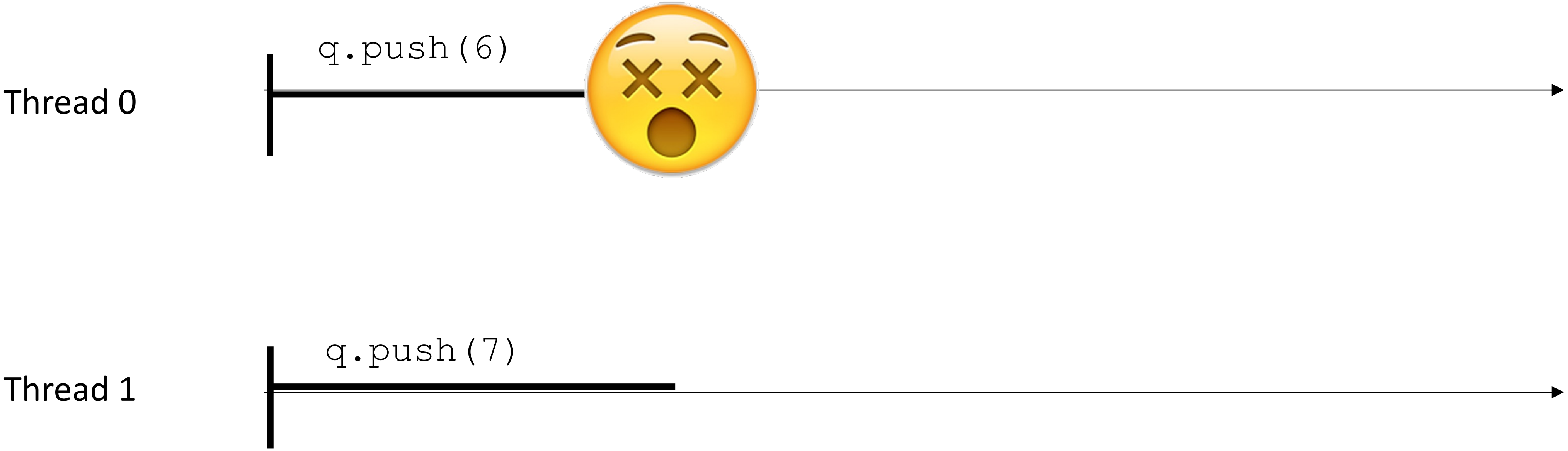
Linearizability does not dictate that one needs to wait for another



Linearizability

Two unfinished commands.

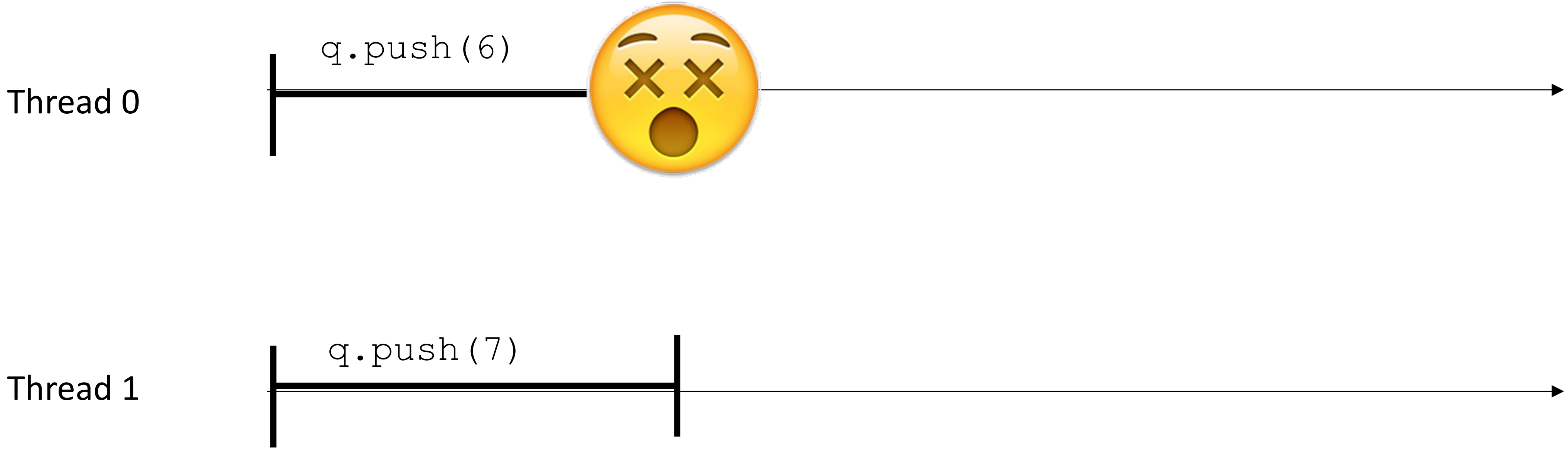
Linearizability does not dictate that one needs to wait for another



for mutexes, the specification required that the system hang.
no such specification here.

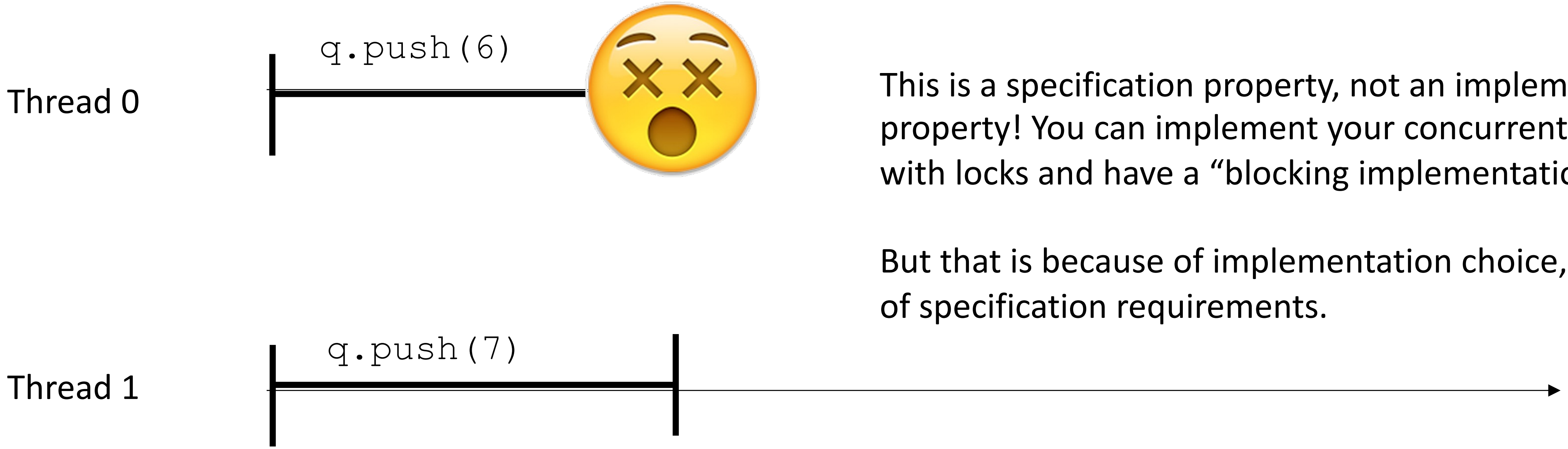
Linearizability

Non-blocking specification:
Every thread is allowed to continue executing
REGARDLESS of the behavior of other threads



Linearizability

Non-blocking specification:
Every thread is allowed to continue executing
REGARDLESS of the behavior of other threads



This is a specification property, not an implementation property! You can implement your concurrent objects with locks and have a “blocking implementation”.

But that is because of implementation choice, not because of specification requirements.

Terminology overview

- Thread-safe implementation:
- Lock-free implementation:
- (Non-)blocking specification:
- (non-)blocking implementation:

Terminology overview

- Sequential consistency:

- Linearizability:

- Linearizability point:

Concurrent Queues

- List of items, accessed in a first-in first-out (FIFO) way
- *duplicates allowed*
- Methods
  - **`enq(x)`** put **`x`** in the list at the end
  - **`deq()`** remove the item at the front of the queue and return it.
  - **`size()`** returns how many items are in the queue

Concurrent Queues

- General implementation given in Chapter 10 of the book.
- Similar types of reasoning as the linked list
    - Lots of reasoning about node insertion, node deletion
    - Using atomic RMWs (CAS) in clever ways
- We will think about specialized queues
    - Implementations can be simplified!

Three Variants of Concurrent Queues

- Input/Output Queues

  Multiple threads enqueue, and multiple threads dequeue but not both

- Producer/Consumer Queues

  1 thread enqueues, 1 thread dequeues

  Two variants:

    Synchronous

    Asynchronous

Input/Output Queues

- Queue in which multiple threads read (deq), or write (enq), but not both.

- Why would we want a thing?

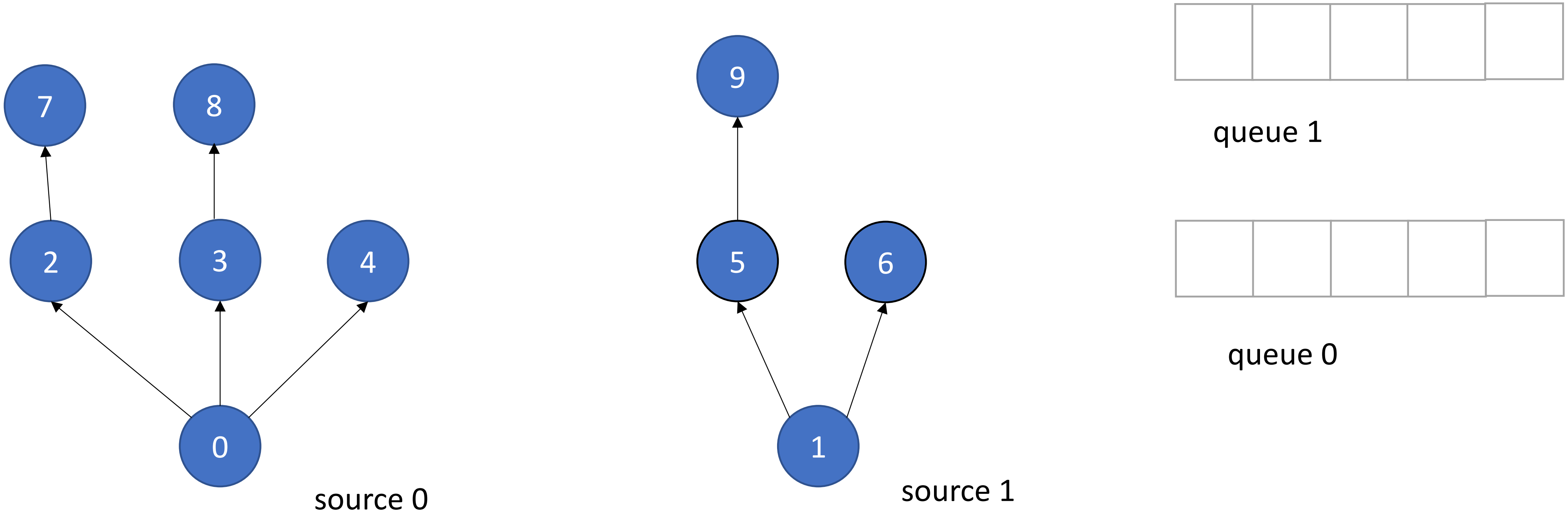
- Computation done in phases:
  - First phase prepares the queue (by writing into it)
  - All threads join
  - Second phase reads values from the queue.

Input/Output Queues

- Example: Information flow in graph applications:

Input/Output Queues

- Example: Information flow in graph applications:



source 0

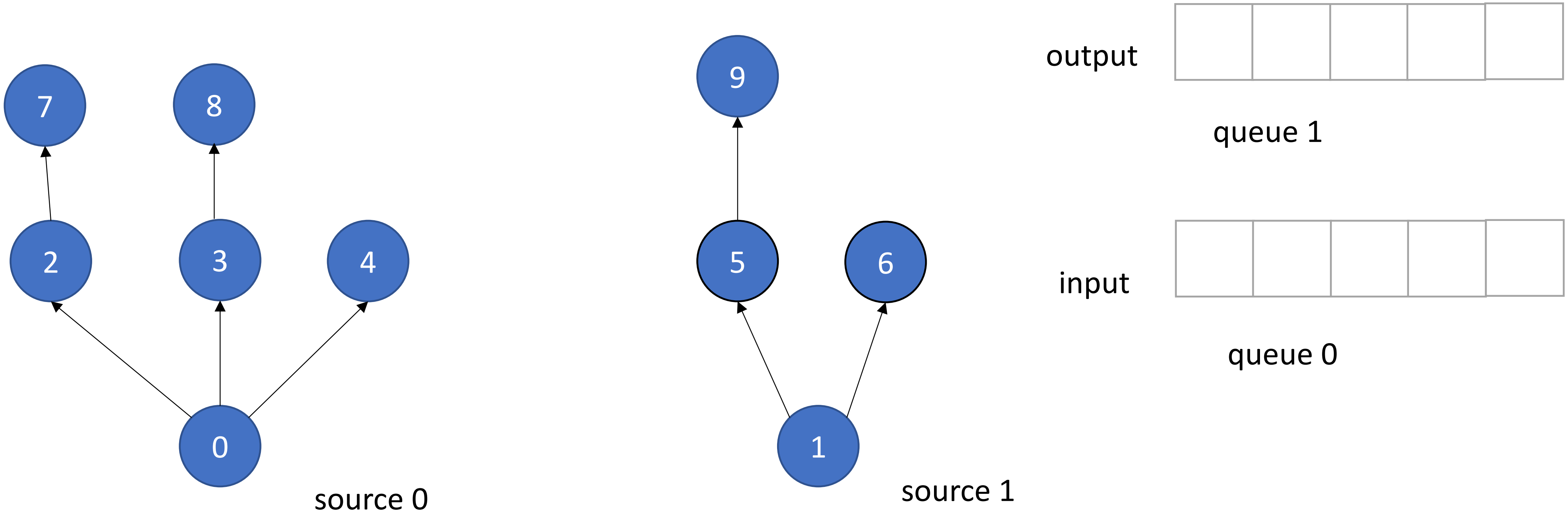
source 1

queue 1

queue 0

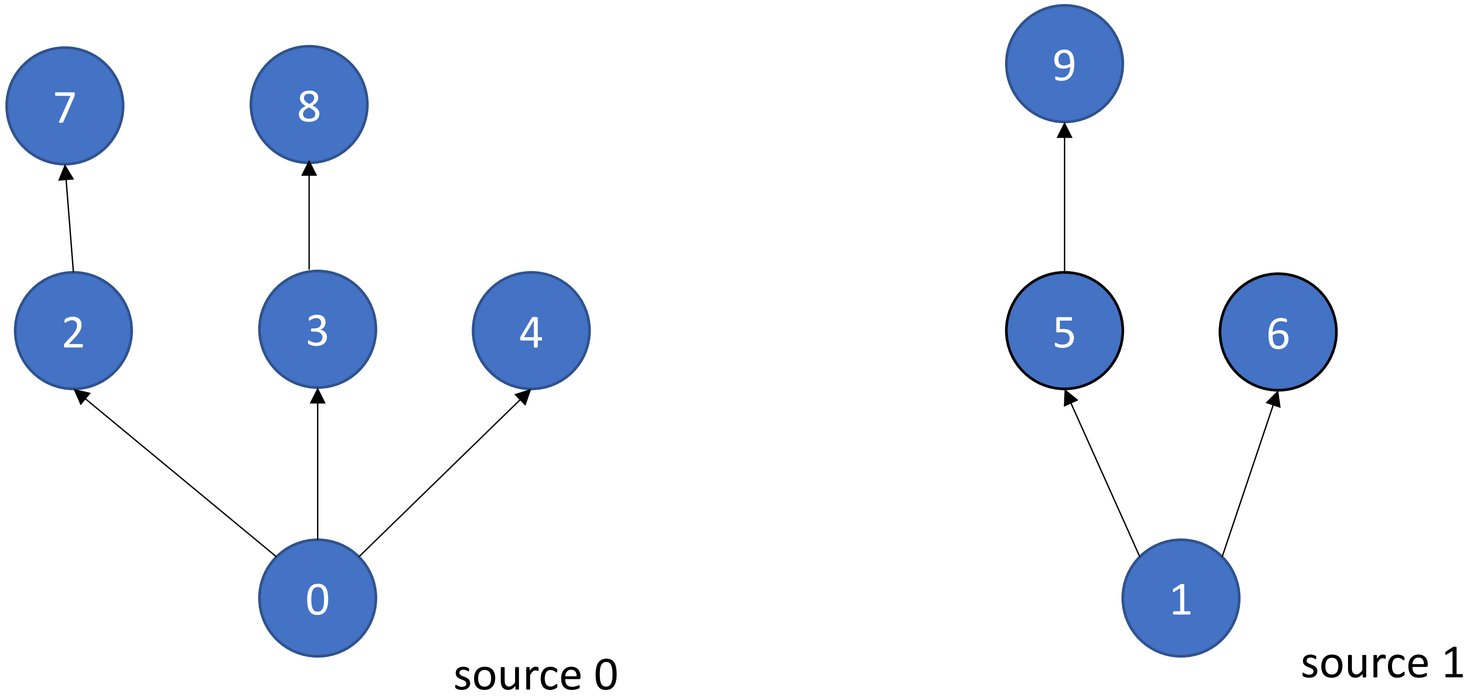
Input/Output Queues

- Example: Information flow in graph applications:



Input/Output Queues

- Example: Information flow in graph applications:



output

queue 1

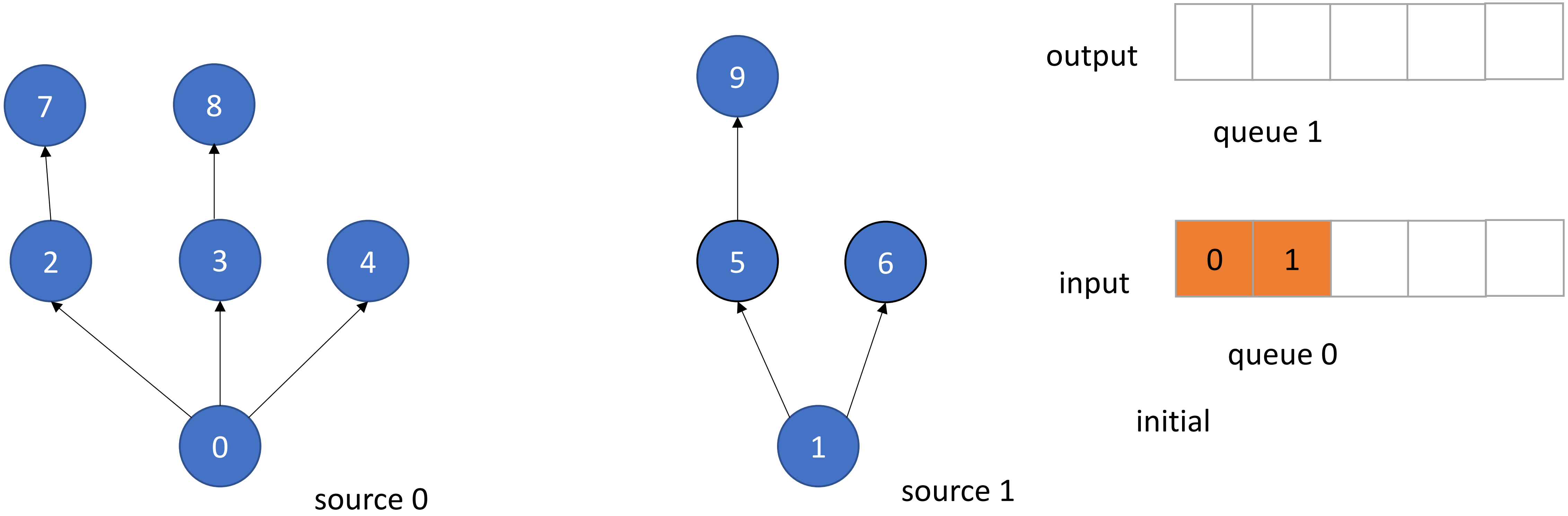
input

0	1			

queue 0

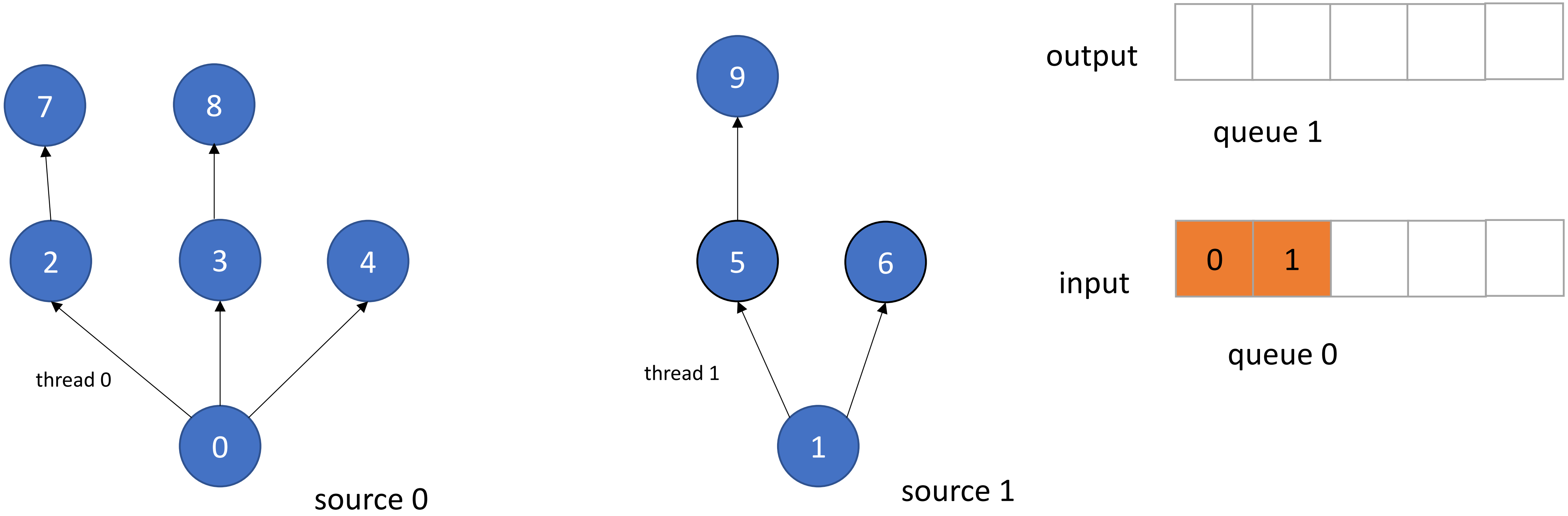
Input/Output Queues

- Example: Information flow in graph applications:



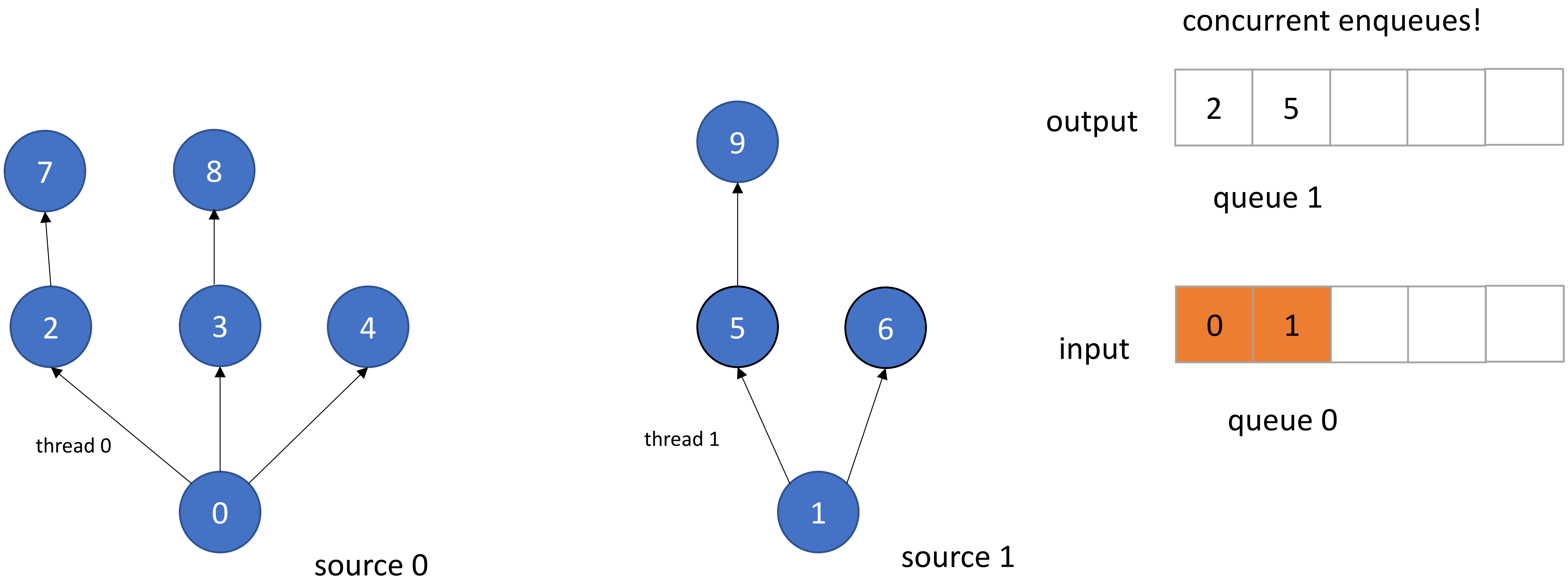
Input/Output Queues

- Example: Information flow in graph applications:



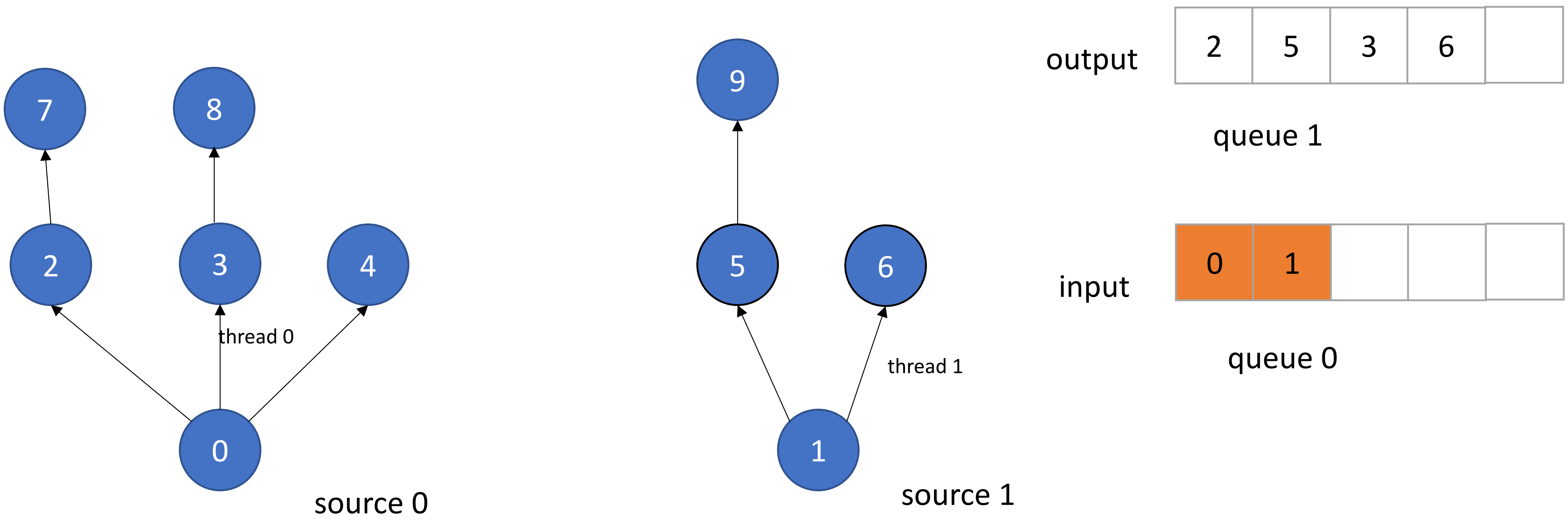
Input/Output Queues

- Example: Information flow in graph applications:



Input/Output Queues

- Example: Information flow in graph applications:



output					
	2	5	3	6	

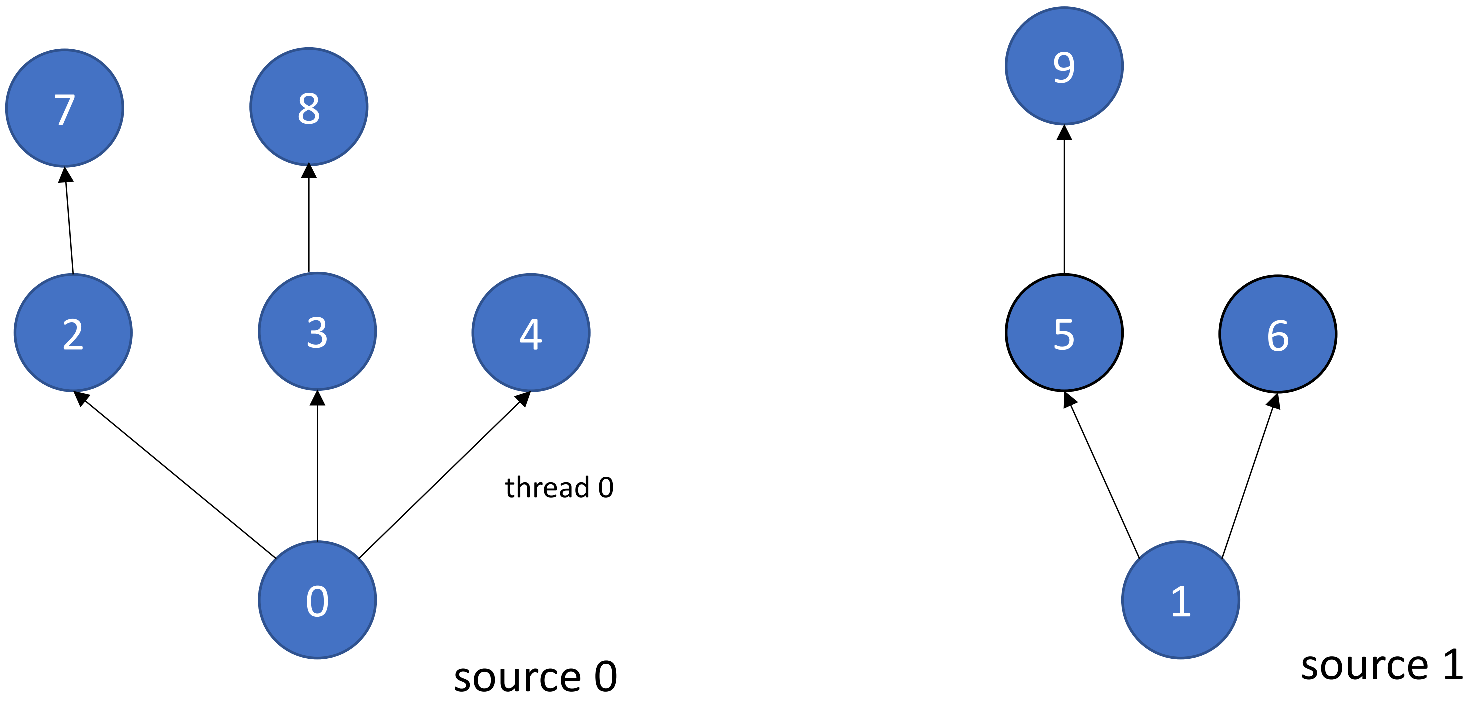
queue 1

input					
	0	1			

queue 0

Input/Output Queues

- Example: Information flow in graph applications:



output

2	5	3	6	4

queue 1

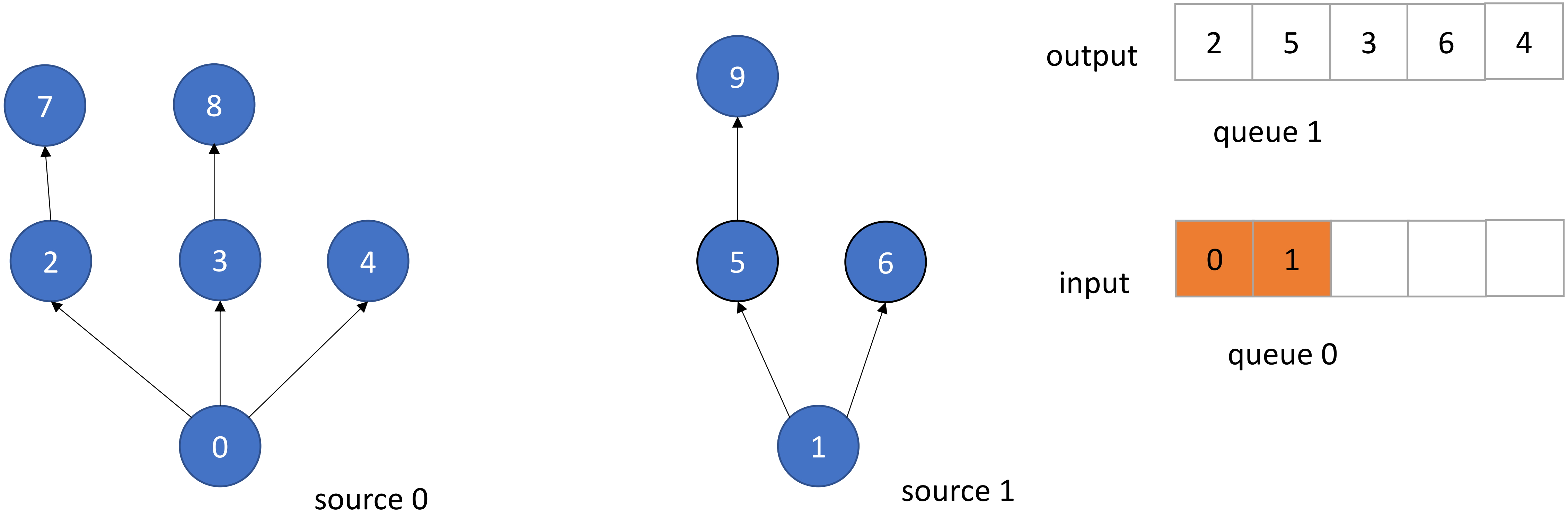
input

0	1			

queue 0

Input/Output Queues

- Example: Information flow in graph applications:



output

2	5	3	6	4

queue 1

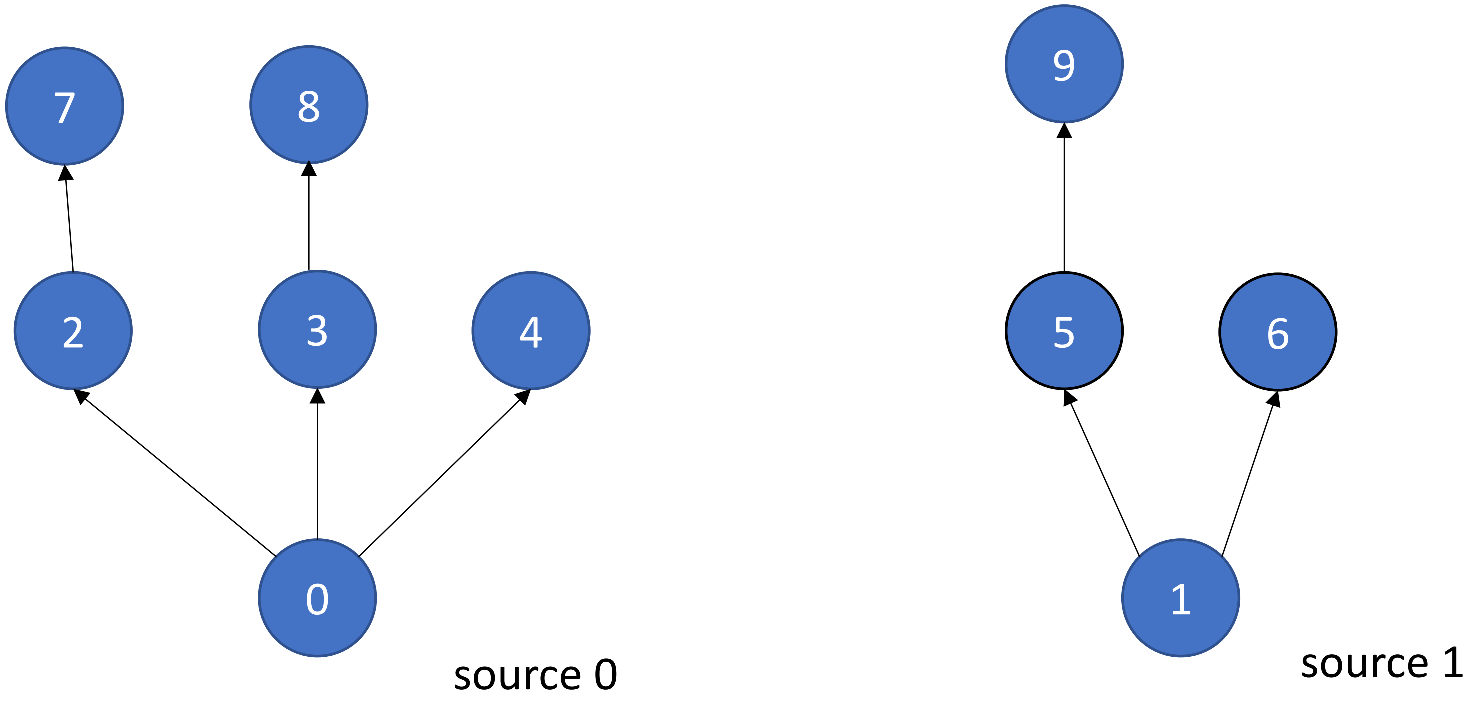
input

0	1			

queue 0

Input/Output Queues

- Example: Information flow in graph applications:



output

2	5	3	6	4

queue 1

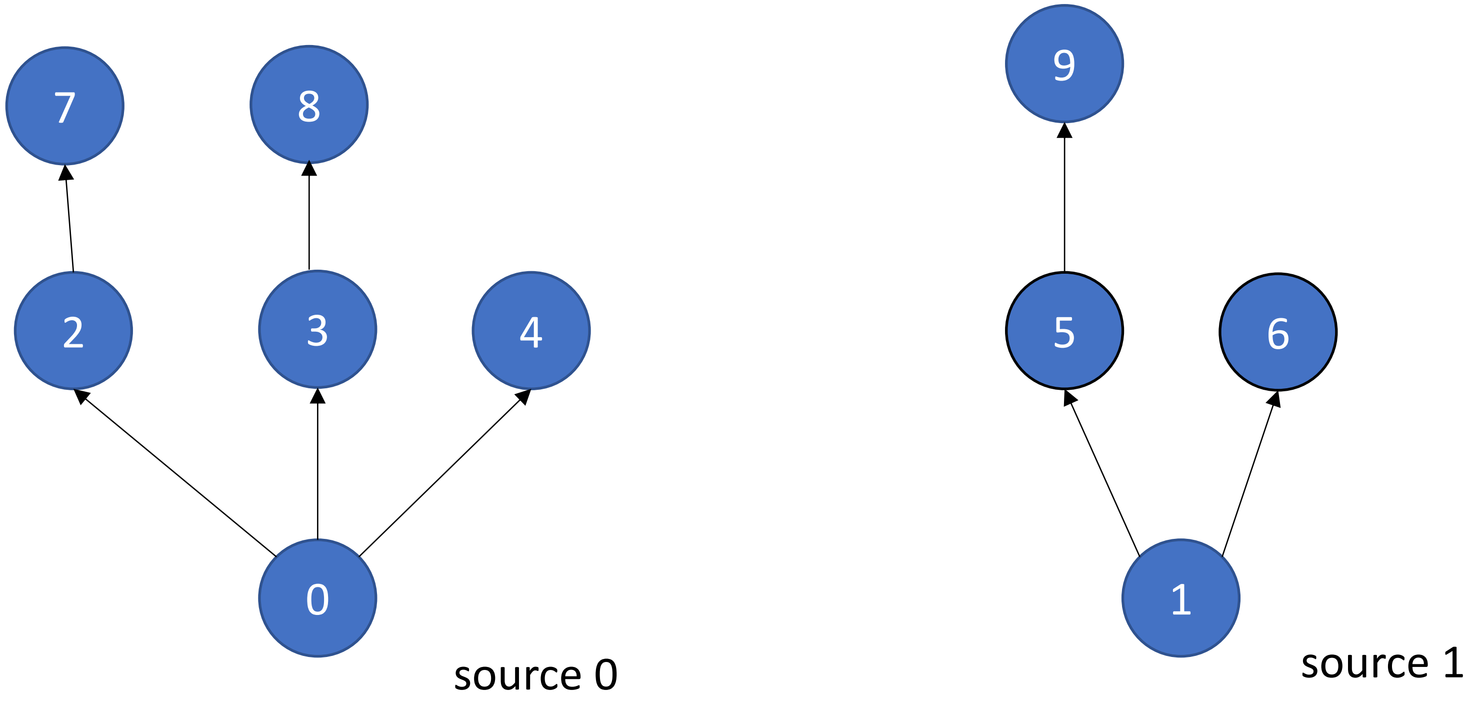
input

queue 0

join all threads and clear input

Input/Output Queues

- Example: Information flow in graph applications:



input	2	5	3	6	4

queue 1

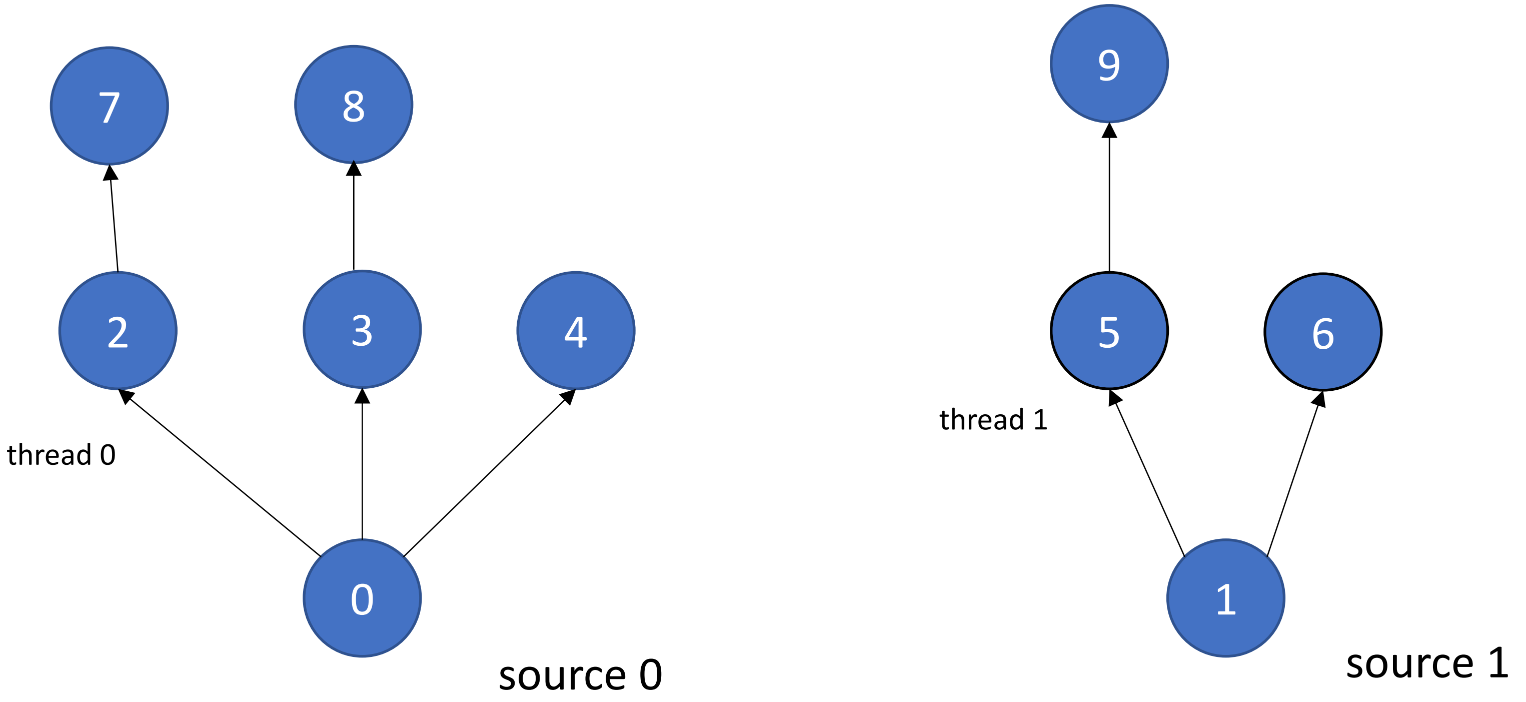
swap!

output					

queue 0

Input/Output Queues

- Example: Information flow in graph applications:



input	2	5	3	6	4

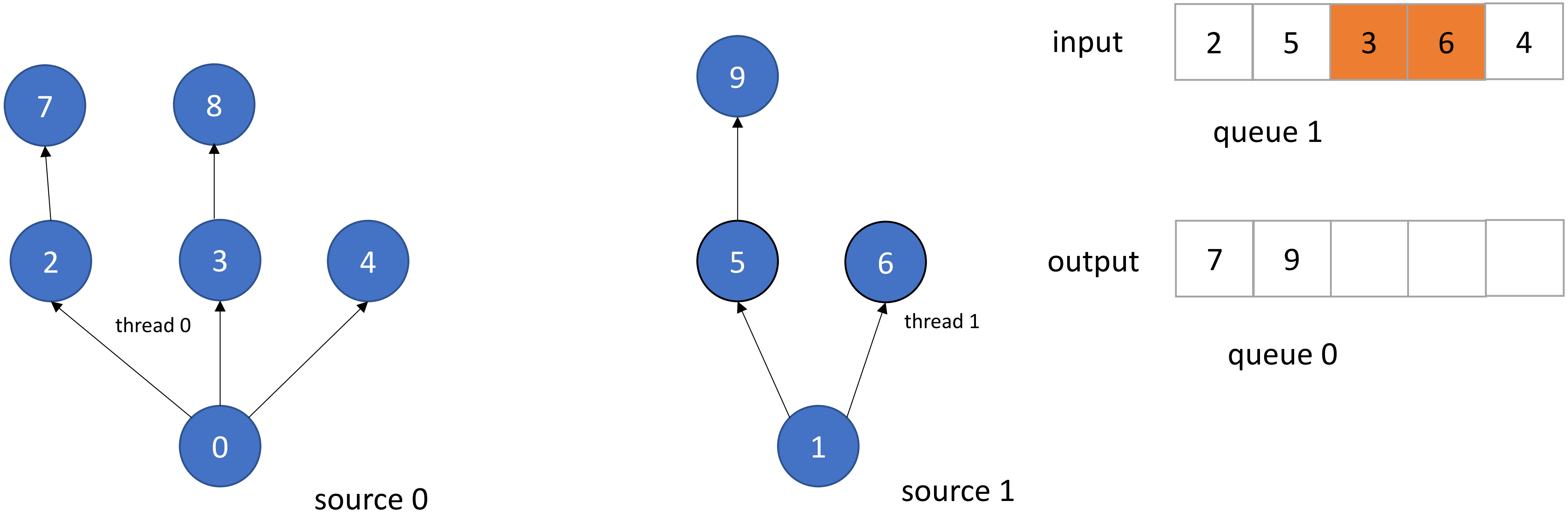
queue 1

output	7	9			

queue 0

Input/Output Queues

- Example: Information flow in graph applications:



input

2	5	3	6	4

queue 1

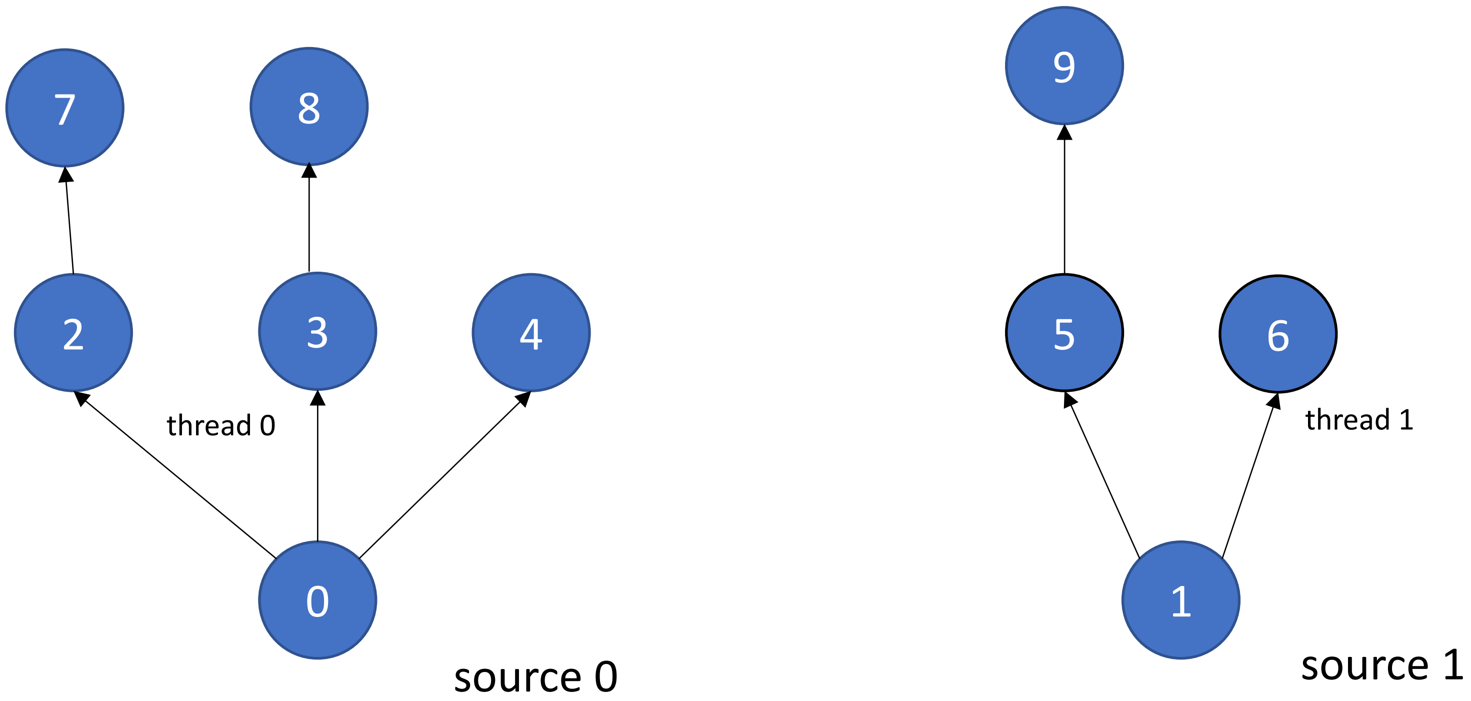
output

7	9			

queue 0

Input/Output Queues

- Example: Information flow in graph applications:



input	2	5	3	6	4

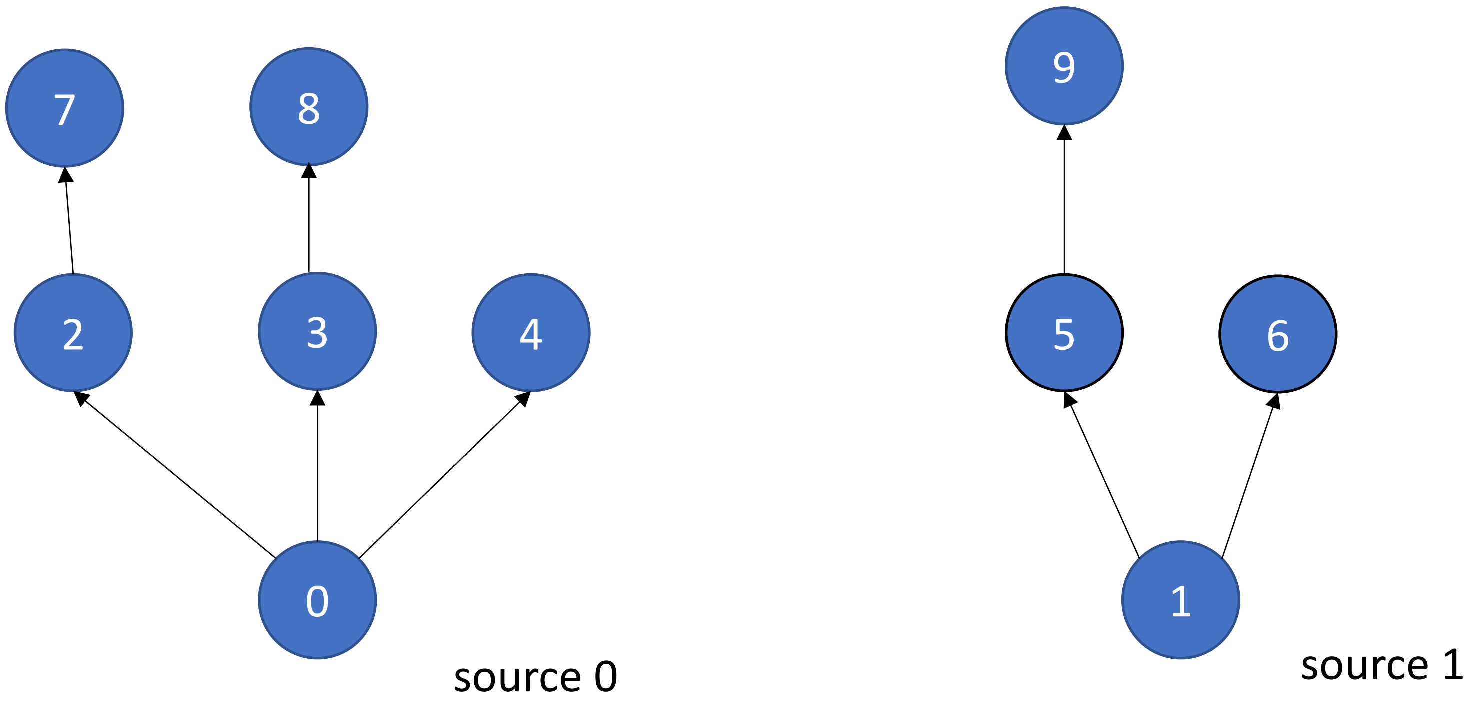
queue 1

output	7	9	8		

queue 0

Input/Output Queues

- Example: Information flow in graph applications:



input	2	5	3	6	4

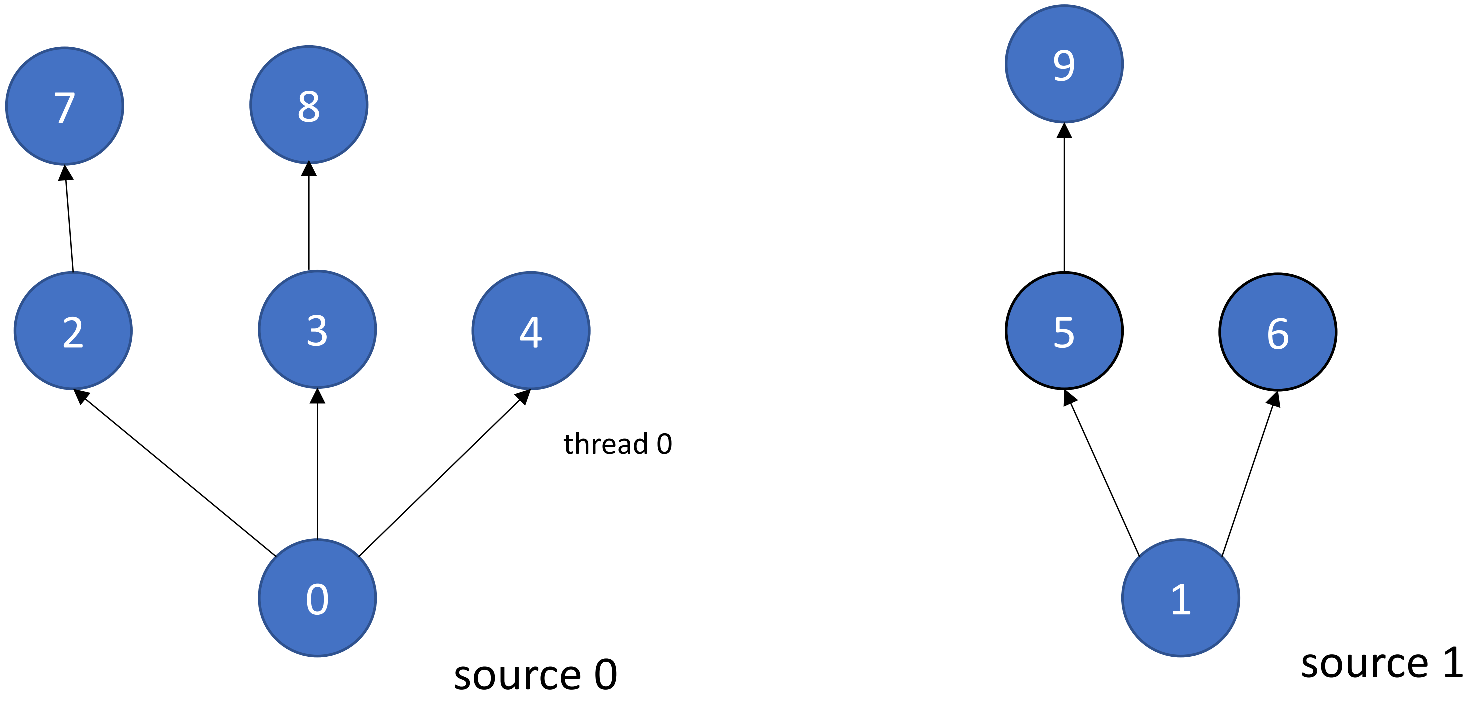
queue 1

output	7	9	8		

queue 0

Input/Output Queues

- Example: Information flow in graph applications:



input	2	5	3	6	4

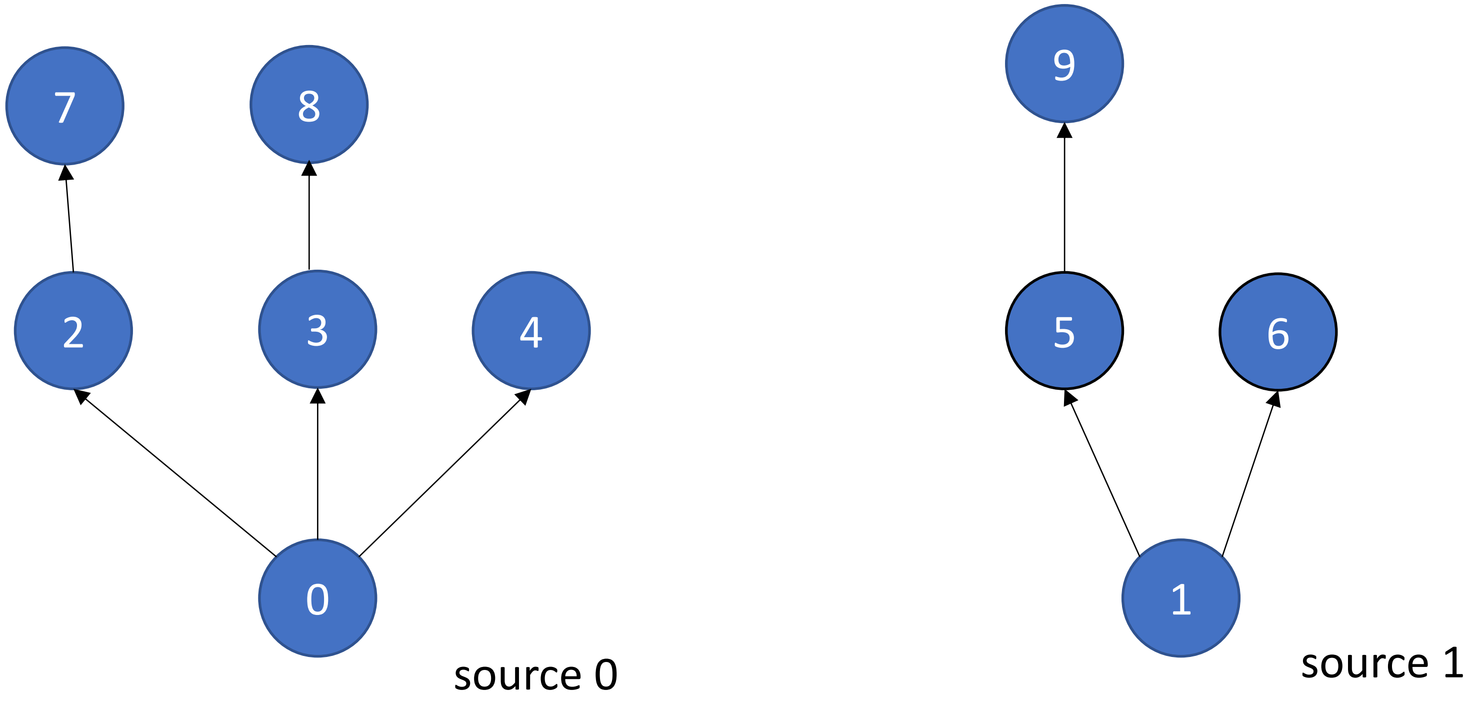
queue 1

output	7	9	8		

queue 0

Input/Output Queues

- Example: Information flow in graph applications:



input	2	5	3	6	4

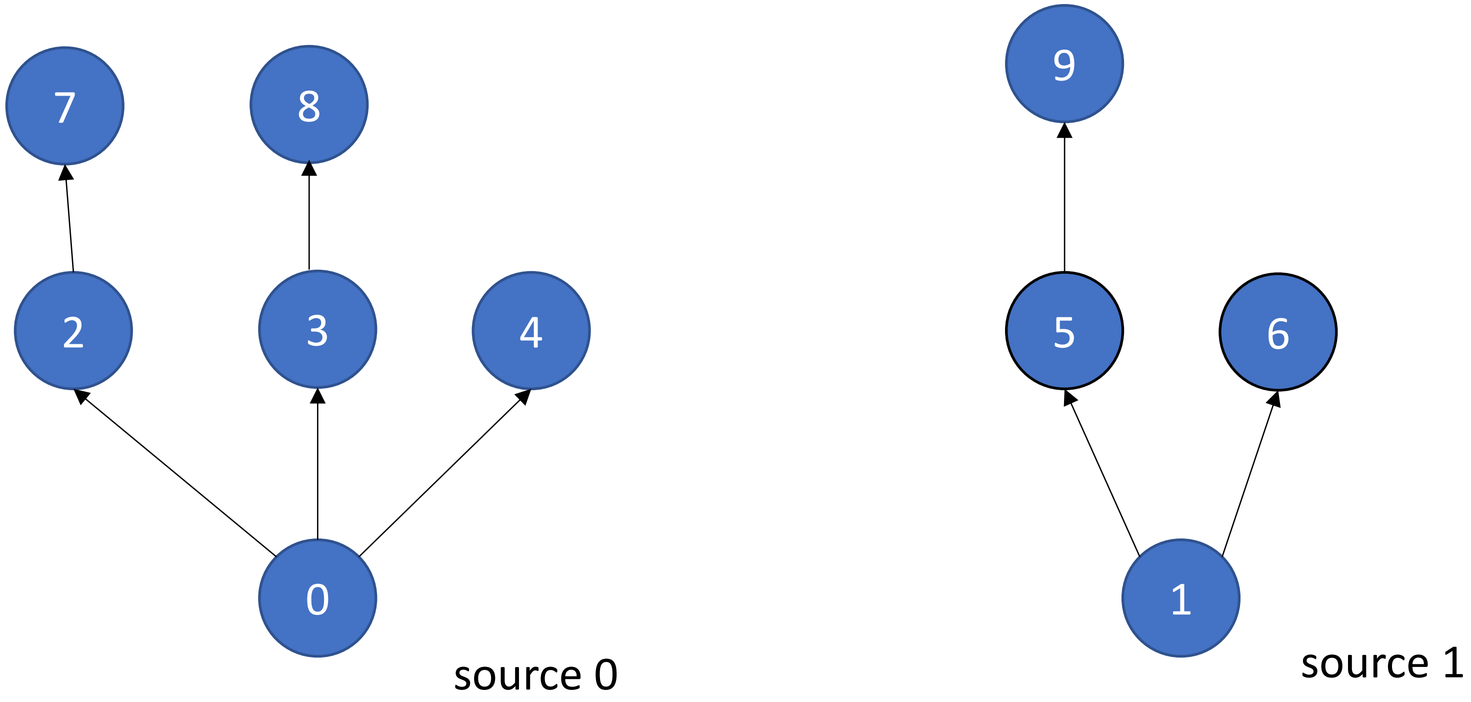
queue 1

output	7	9	8		

queue 0

Input/Output Queues

- Example: Information flow in graph applications:



and so on...

output

queue 1

input

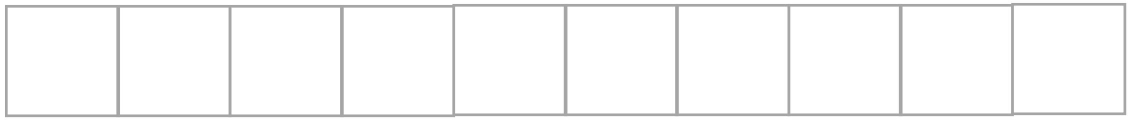
7	9	8		

queue 0

Implementation

Implementation

Allocate a contiguous array

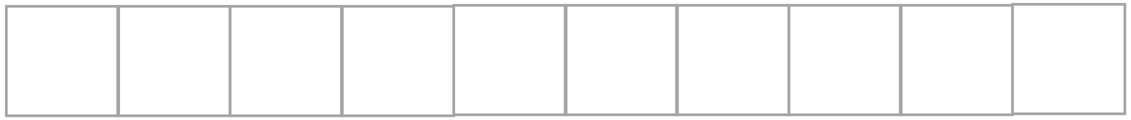


Pros:
?

Cons:
?

Implementation

Allocate a contiguous array



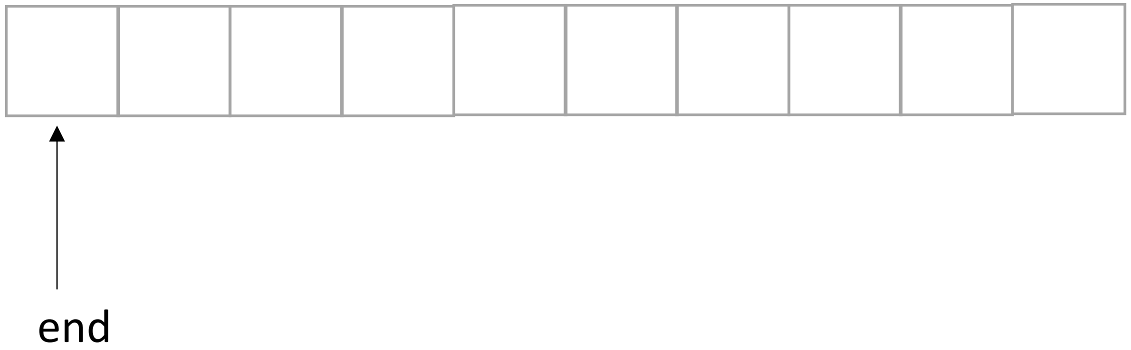
Pros:
+ fast!
+ we can use indexes instead of addresses

Cons:
- need to reason about overflow!

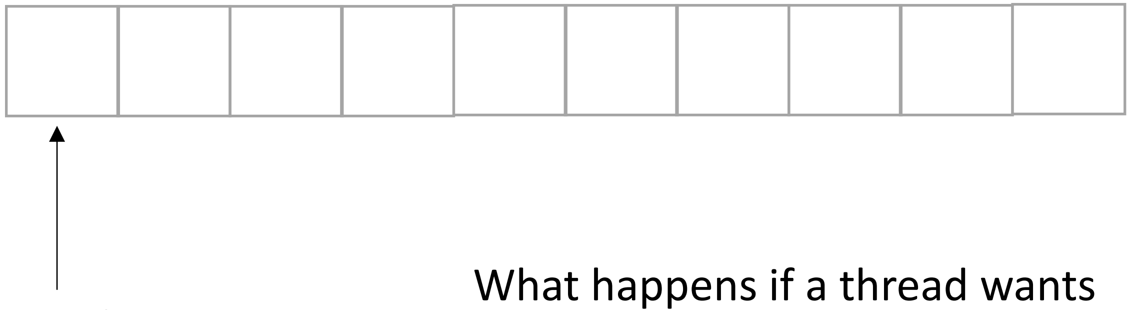
Note on terminology

- Head/tail - often used in queue implementations, but switches when we start doing circular buffers.
- Front/end - To avoid confusion, we will use front/end for input/output queues.

Implementation

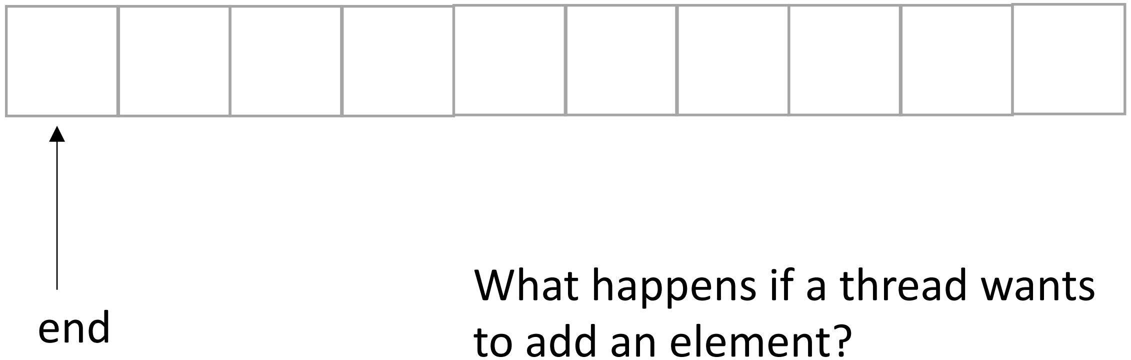


Implementation



What happens if a thread wants to add an element?

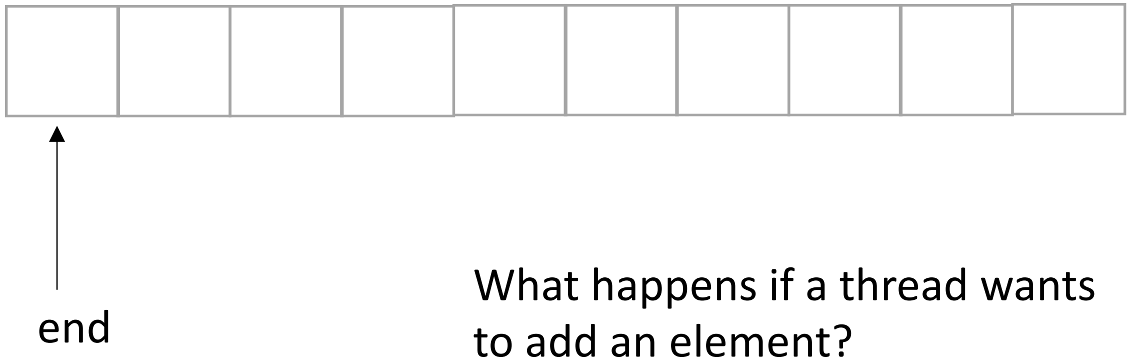
Implementation



What happens if a thread wants to add an element?

Think sequentially:

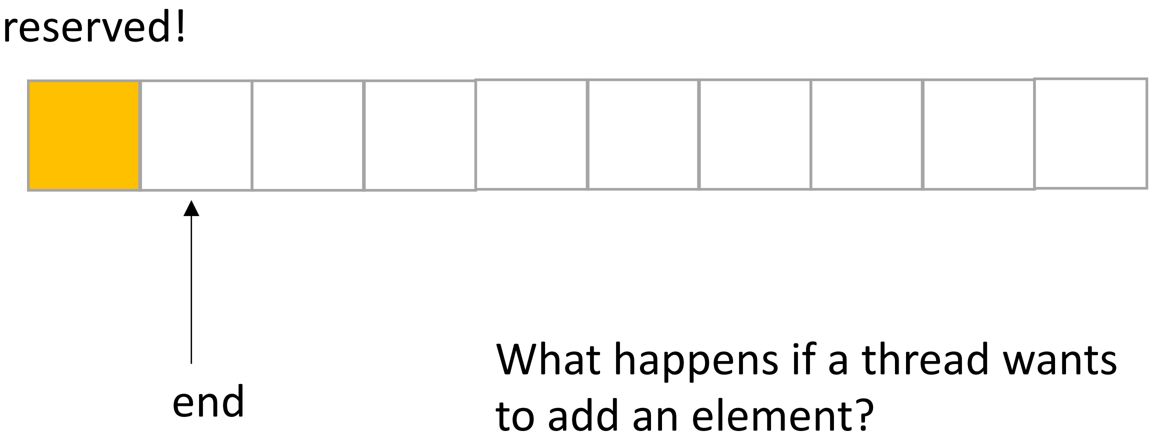
Implementation



What happens if a thread wants to add an element?

Think sequentially:
*reserve a space - increment end

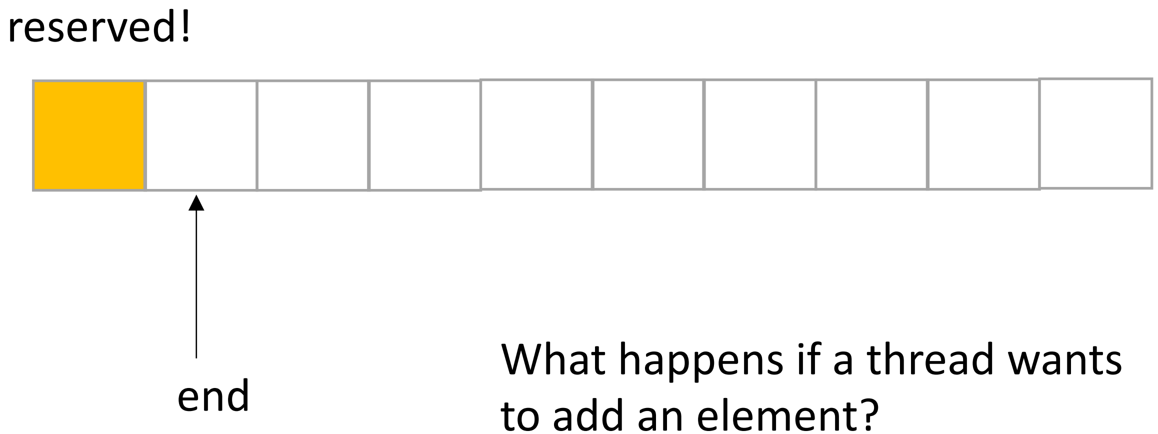
Implementation



What happens if a thread wants to add an element?

Think sequentially:
*reserve a space - increment end

Implementation



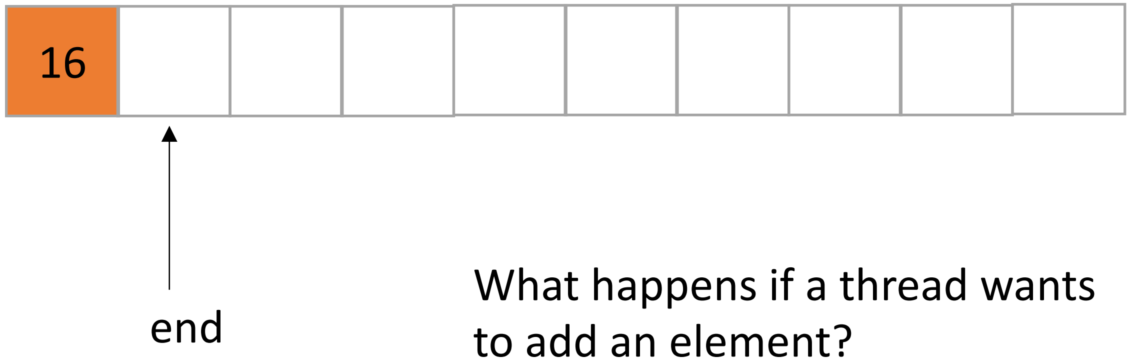
What happens if a thread wants to add an element?

Think sequentially:

* reserve a space - increment end

* add the element

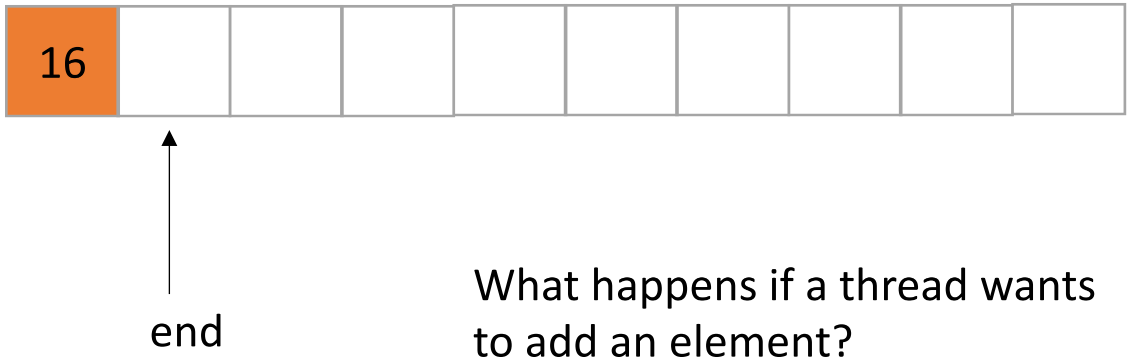
Implementation



What happens if a thread wants to add an element?

Think sequentially:
* reserve a space - increment end
* add the element

Implementation

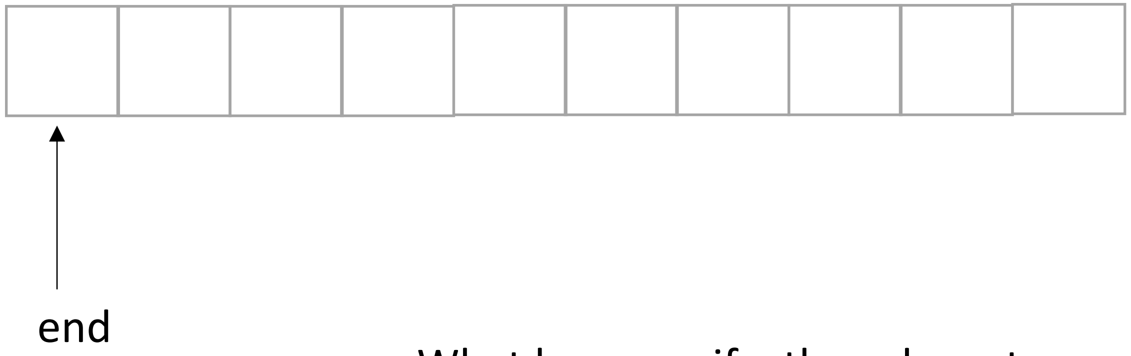


What happens if a thread wants to add an element?

Think sequentially:
* reserve a space - increment end
* add the element

done!

Implementation

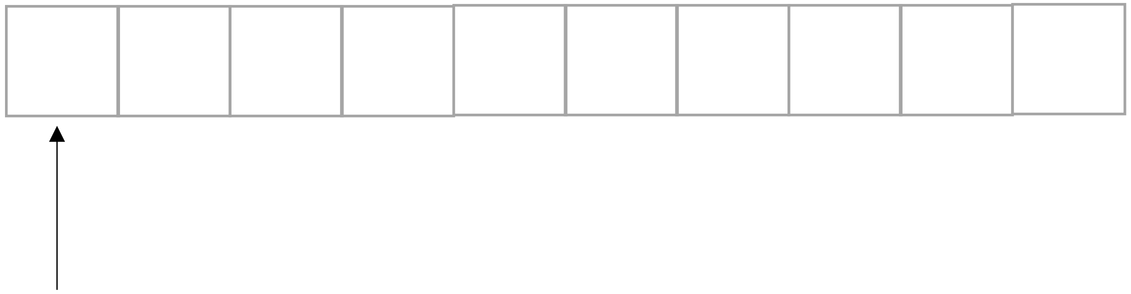


What happens if a thread wants to add an element?

Think concurrently:

Two threads cannot reserve the same space!
We've seen this before

Implementation



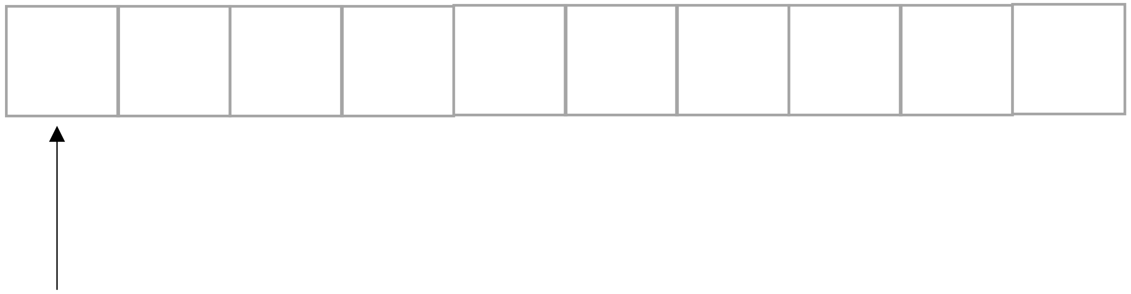
end

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);

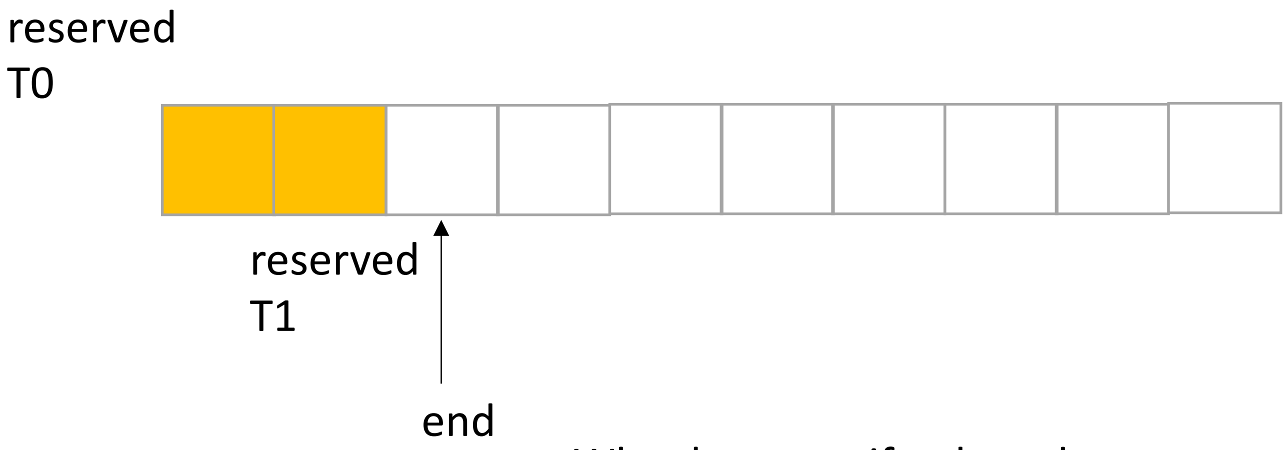
<u>Thread 1:</u>
enq(7);

What happens if a thread wants to add an element?

Think concurrently:

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

Implementation



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

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

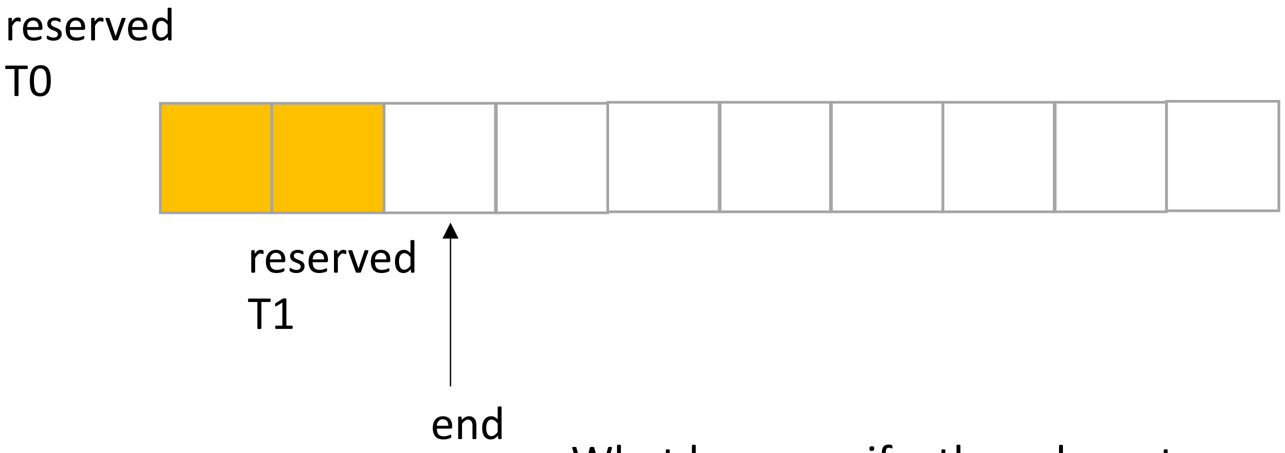
What happens if a thread wants to add an element?

Think concurrently:

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

Implementation

does it matter which order threads add their data?



What happens if a thread wants to add an element?

Think concurrently:

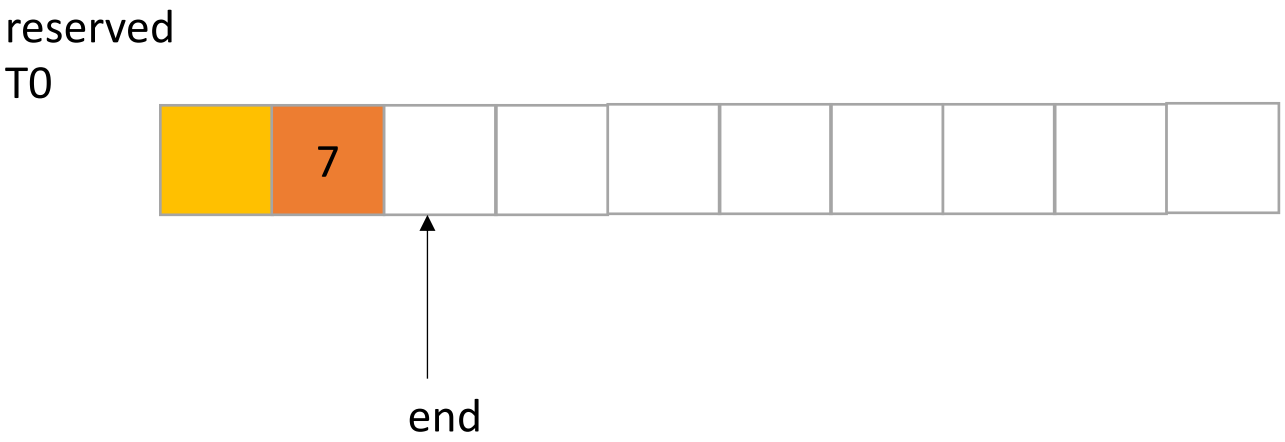
```
reserved_index = atomic_fetch_add(&end, 1);
```

<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);

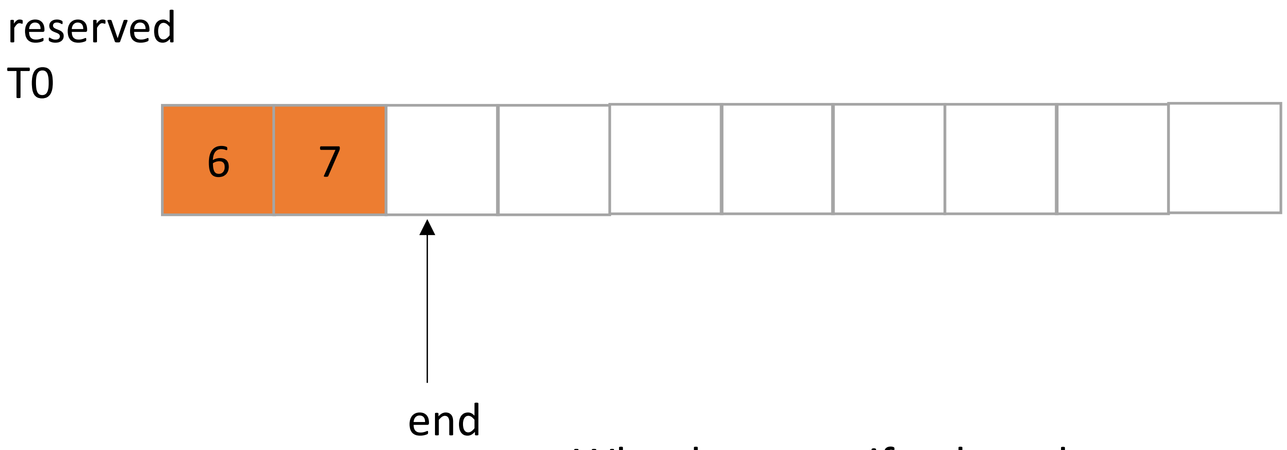
What happens if a thread wants to add an element?

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!



Thread 0:
enq(6);

Thread 1:
enq(7);

What happens if a thread wants to add an element?

Think concurrently:

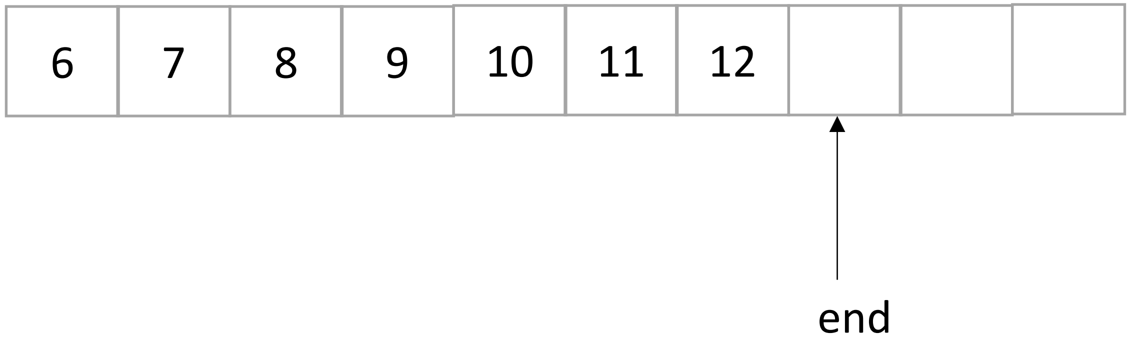
```
reserved_index = atomic_fetch_add(&end, 1);
```

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

  public:
    InputOutputQueue() {
        end = 0;
     }

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

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

How to protect against overflows?

What about Input?

- Now we only do deqs

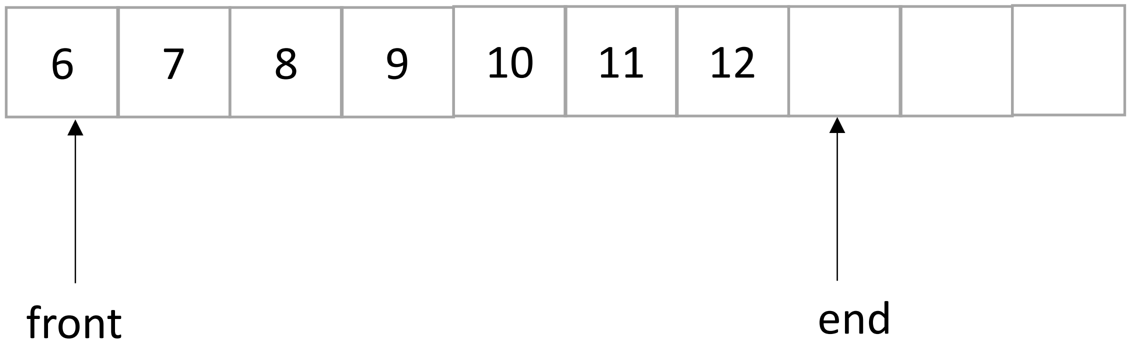


6	7	8	9	10	11	12			

end

What about Input?

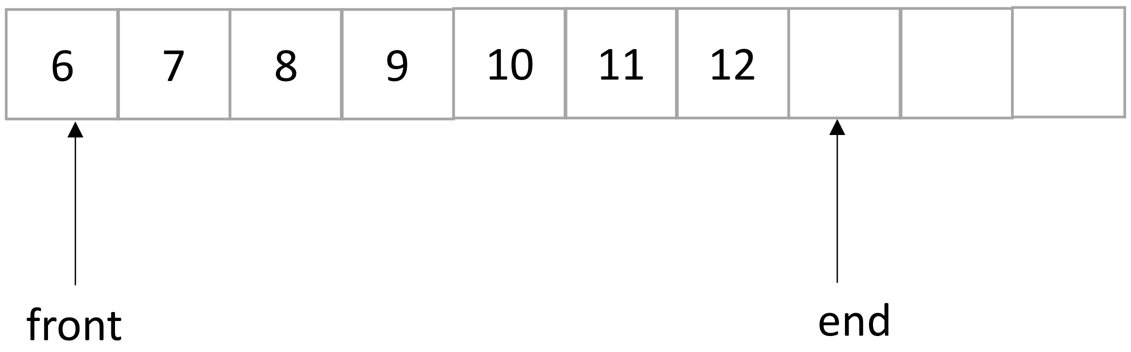
- Now we only do deqs



6	7	8	9	10	11	12			

What about Input?

- Now we only do deqs



6 7 8 9 10 11 12

front

end

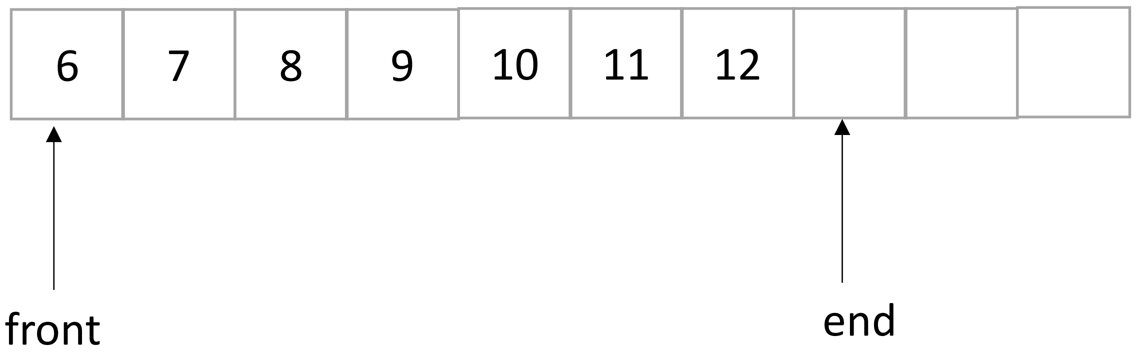
What happens if a thread wants
to dequeue an element?

Think concurrently:

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

What about Input?

- Now we only do deqs



6	7	8	9	10	11	12			

front

end

Thread 0:
deq();

Thread 1:
deq();

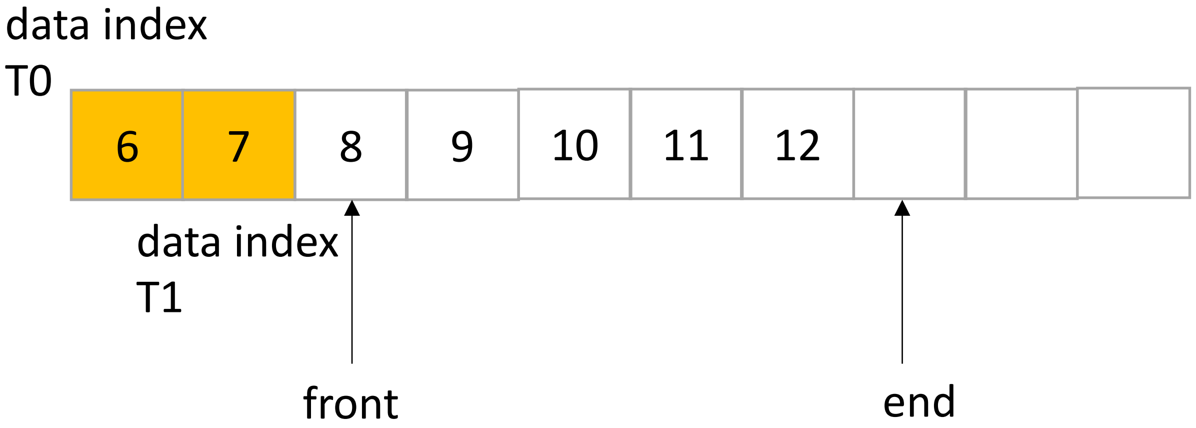
What happens if a thread wants to dequeue an element?

Think concurrently:

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

What about Input?

- Now we only do deqs



data index
T0

6	7	8	9	10	11	12			

data index
T1

front

end

<u>Thread 0:</u>
deq();

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

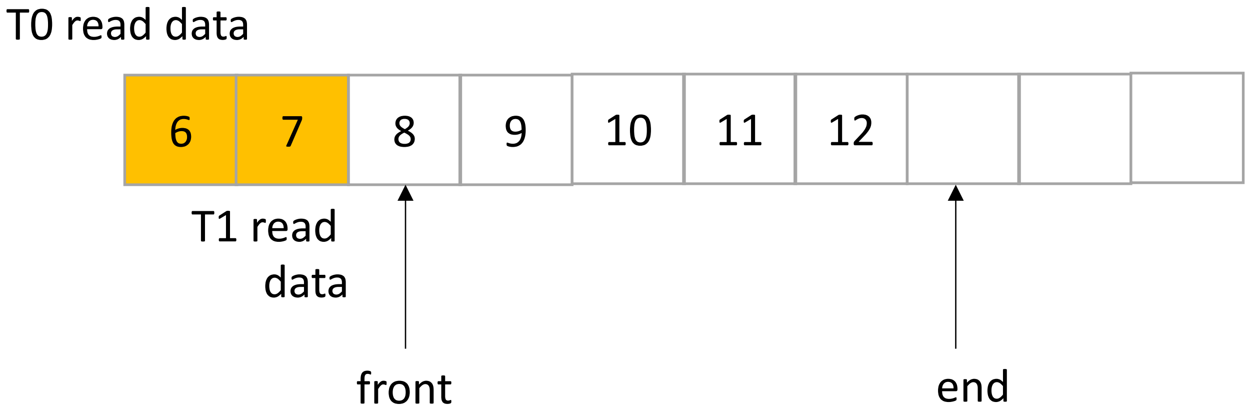
What happens if a thread wants to dequeue an element?

Think concurrently:

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

What about Input?

- Now we only do deqs



Thread 0:
deq(); // reads 6

Thread 1:
deq(); // reads 7

What happens if a thread wants to dequeue 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 ??;
     }
 }
```

```
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 ??;
     }
 }
```

How about size?

```
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();
     }
 }
```

how about size?

how do we reset?

```
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();
     }
 }
```

how about size?

how do we reset?
Reset front and end

```
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();
     }
 }
```

how about size?

how do we reset?
Reset front and end

does the list need
to be atomic?

Producer Consumer Queues

- 1 thread enqueues, 1 thread dequeues
    - enq’er cannot deq
    - deq’er cannot enq

- Example: printf:
    - your program enqueues 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

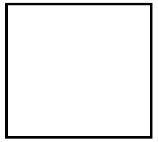
- First implementation:
    - Synchronous
    - Slow
    - Good for debugging

- enq does not return until value is deq’ed

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
```



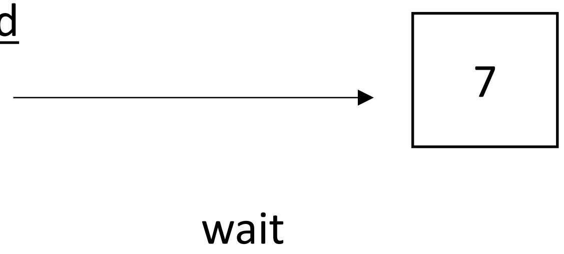
Consumer Thread

```
deq();
```

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
```



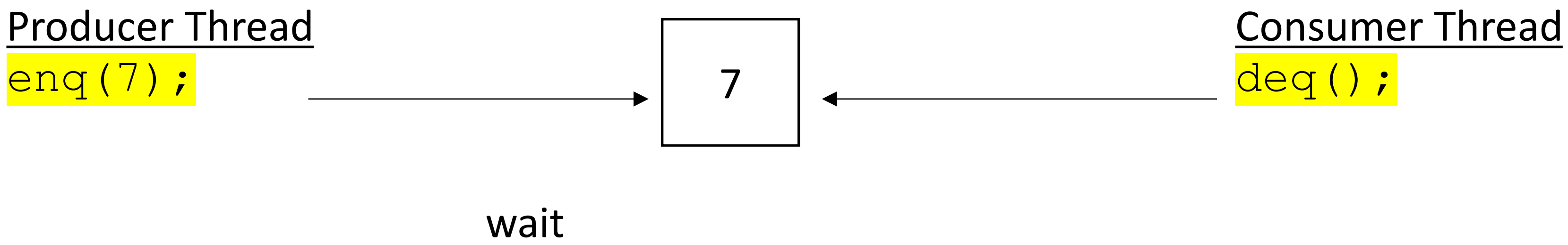
7

wait

Consumer Thread

```
deq();
```

Synchronous Producer Consumer Queues



Producer Thread

`enq(7);`

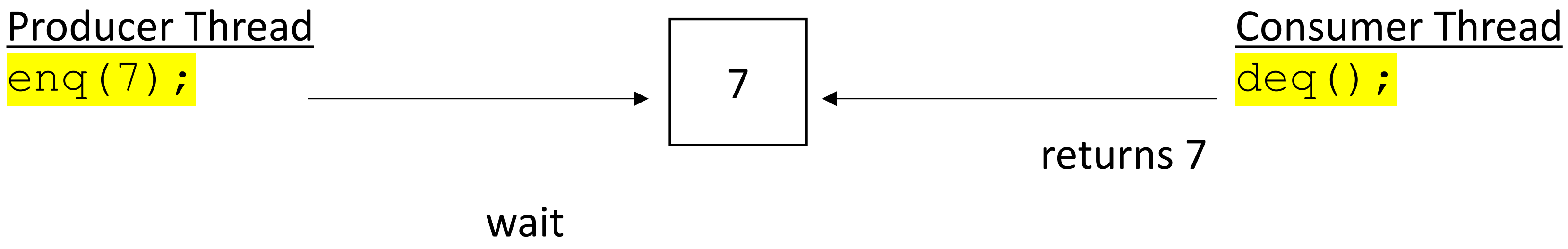
7

Consumer Thread

`deq();`

wait

Synchronous Producer Consumer Queues



Producer Thread

`enq(7);`

7

Consumer Thread

`deq();`

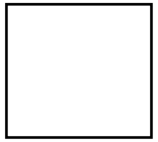
returns 7

wait

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
```



Consumer Thread

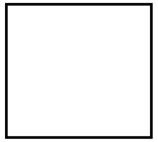
```
deq();
```

both can continue

Synchronous Producer Consumer Queues

Producer Thread

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



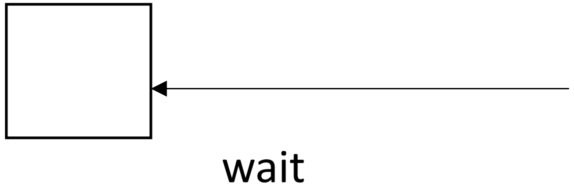
Consumer Thread

```
deq();
```

Synchronous Producer Consumer Queues

Producer Thread

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

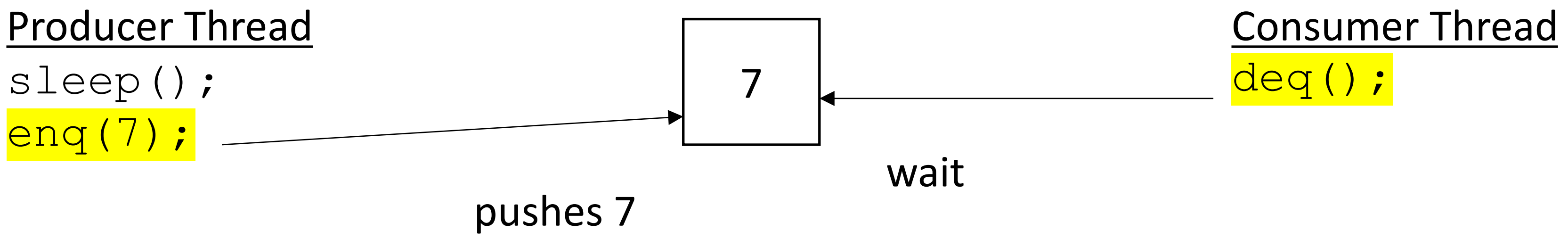


wait

Consumer Thread

```
deq();
```

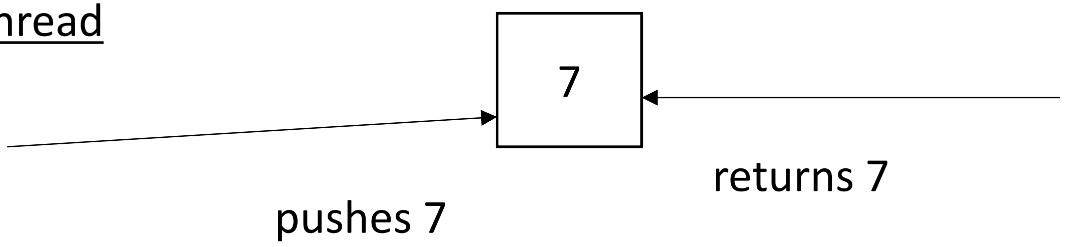
Synchronous Producer Consumer Queues



Synchronous Producer Consumer Queues

Producer Thread

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



pushes 7

returns 7

Consumer Thread

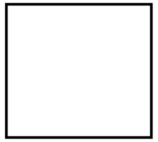
```
deq();
```

They both can continue

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
```



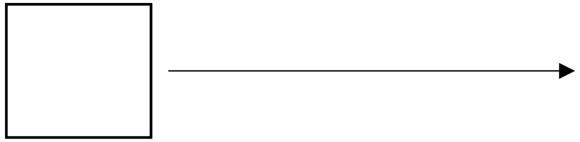
Consumer Thread

```
deq();
```

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
```



Consumer Thread

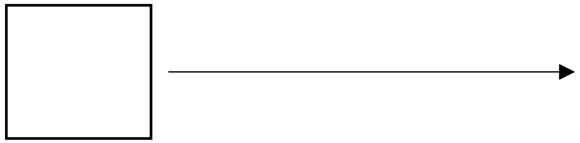
```
deq();
```

Can the consumer just read?

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
```



Consumer Thread

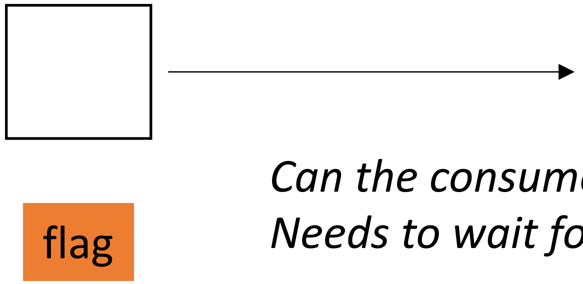
```
deq();
```

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

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
```



flag

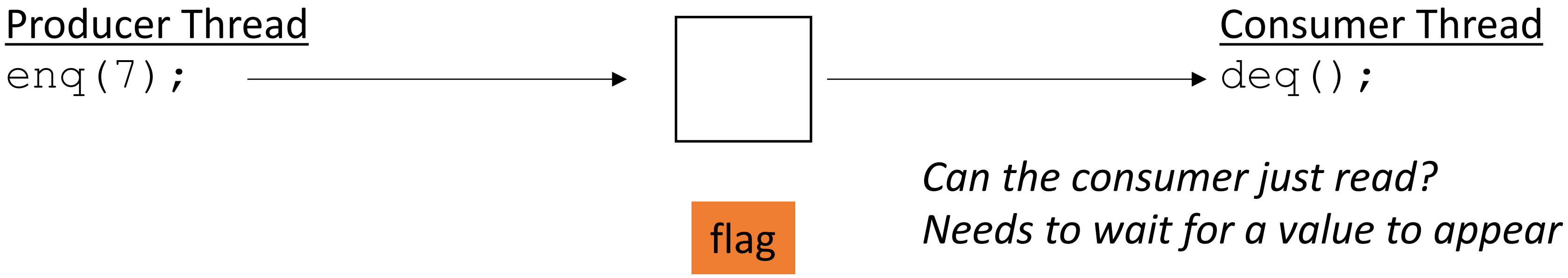
Consumer Thread

```
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



<u>Producer Thread</u>

`enq(7);`

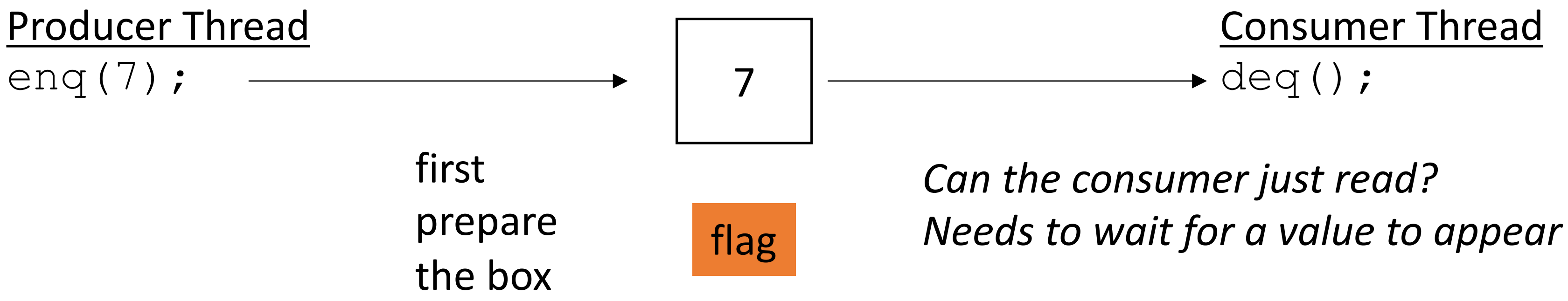
<u>Consumer Thread</u>

`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

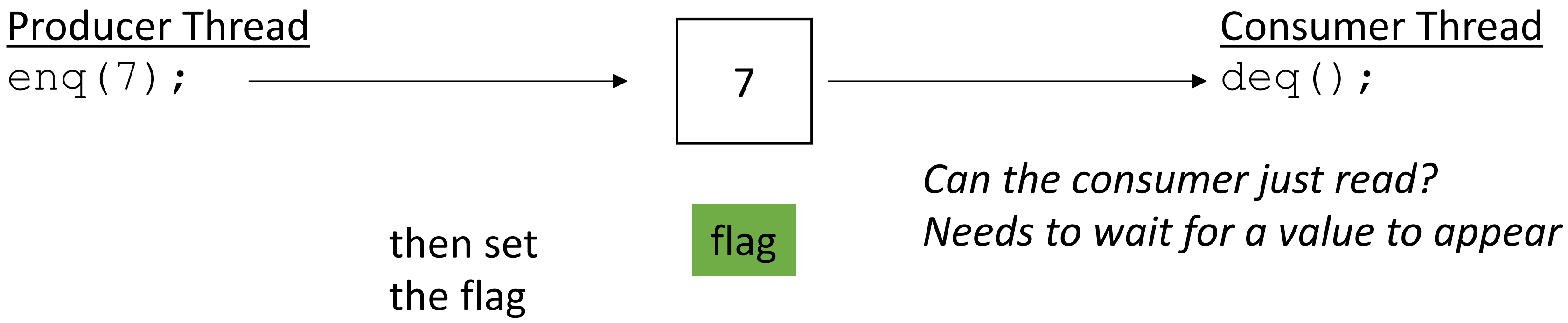
`deq();`

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

then set
the flag

🟩 flag

Consumer Thread

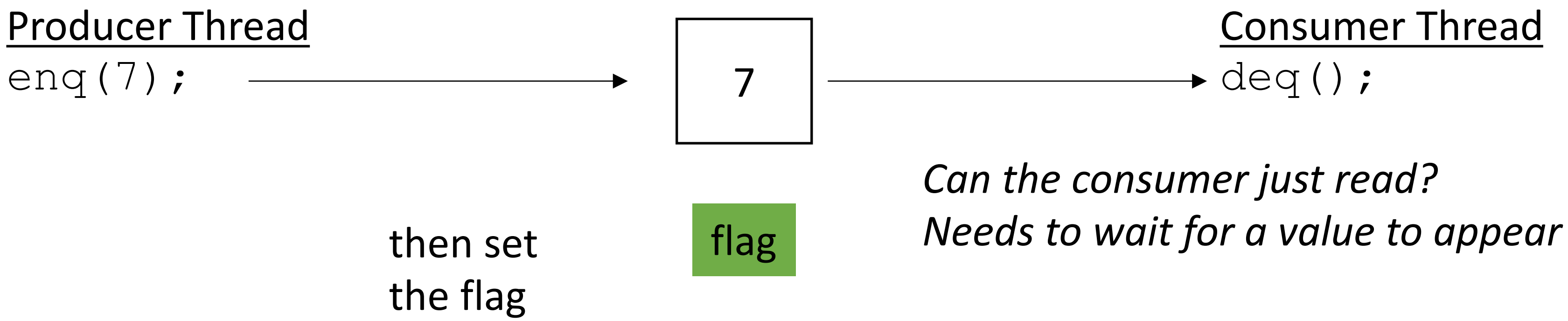
`deq();`

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

now the consumer can read from the box!



<u>Producer Thread</u>

`enq(7);`

7

<u>Consumer Thread</u>

`deq();`

then set
the flag

flag

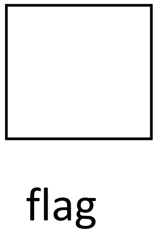
Can the consumer just read?
Needs to wait for a value to appear

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

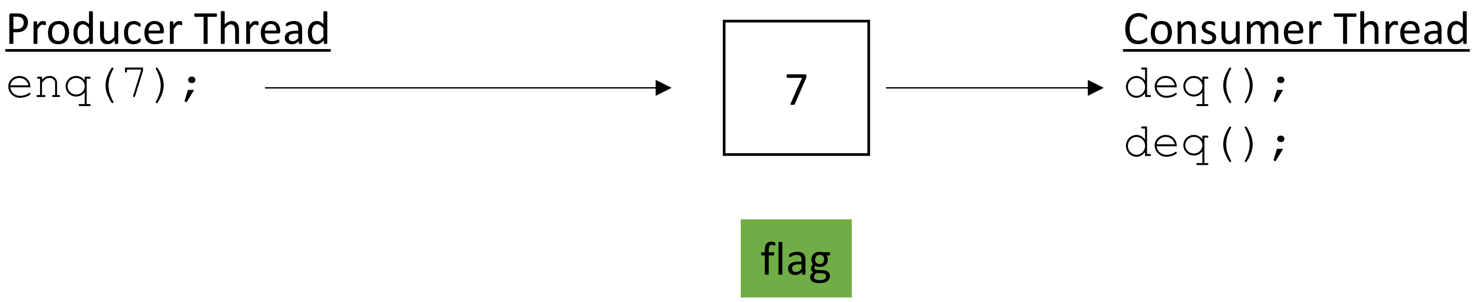
Consumer Thread

```
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);`

7

flag

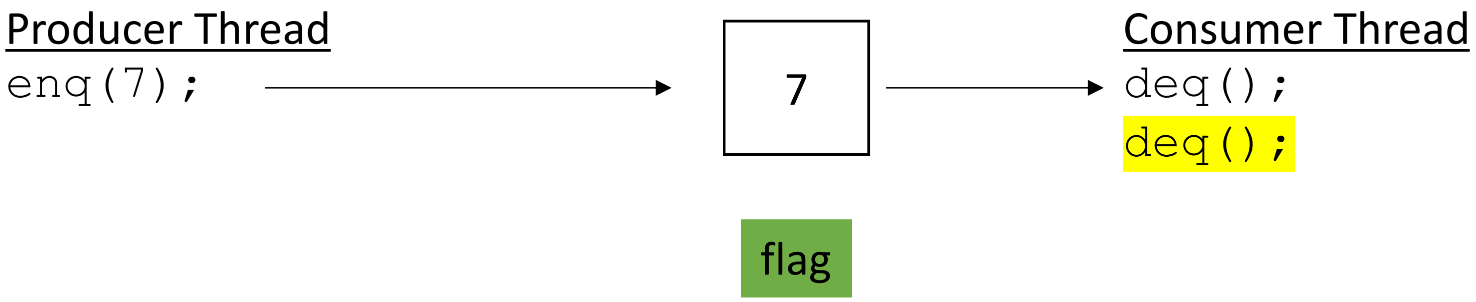
Consumer Thread

`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
    }
  }
```

Synchronous Producer Consumer Queues

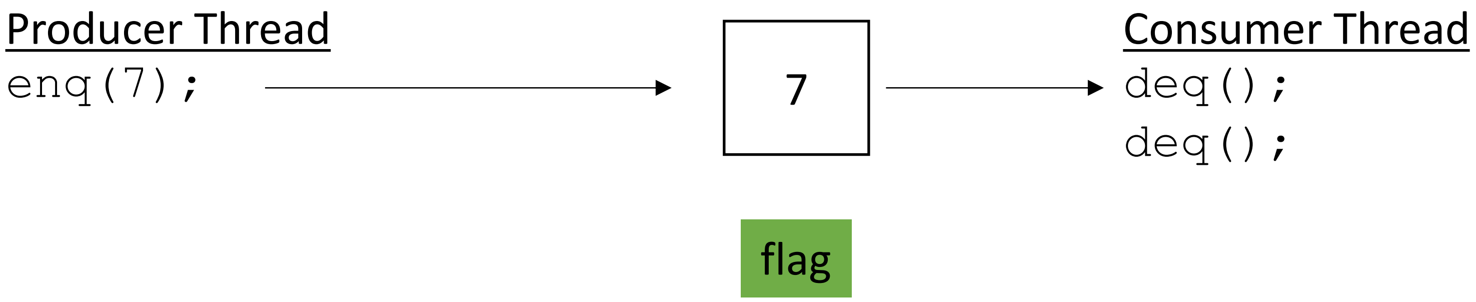


```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  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

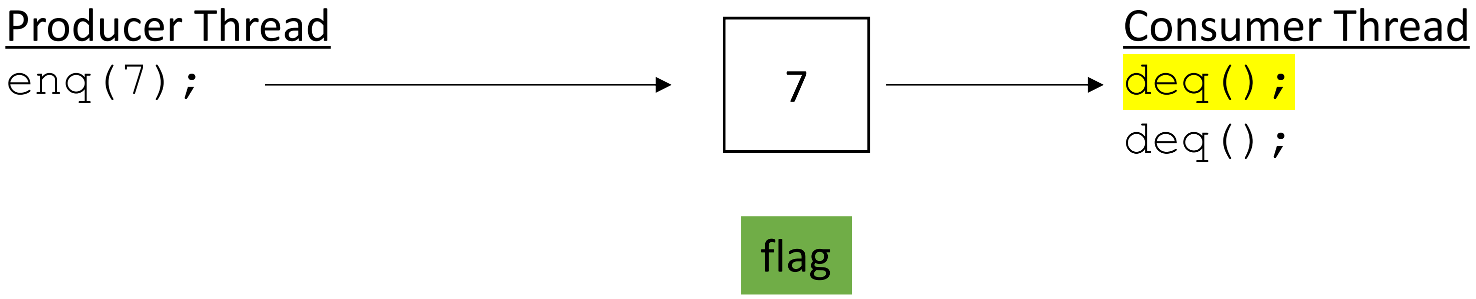


```
enq(7);
```

```
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

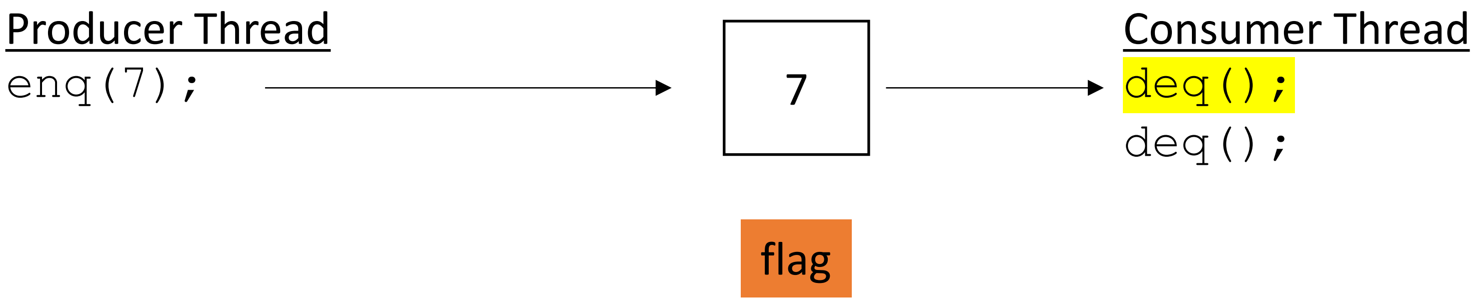
Consumer Thread

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

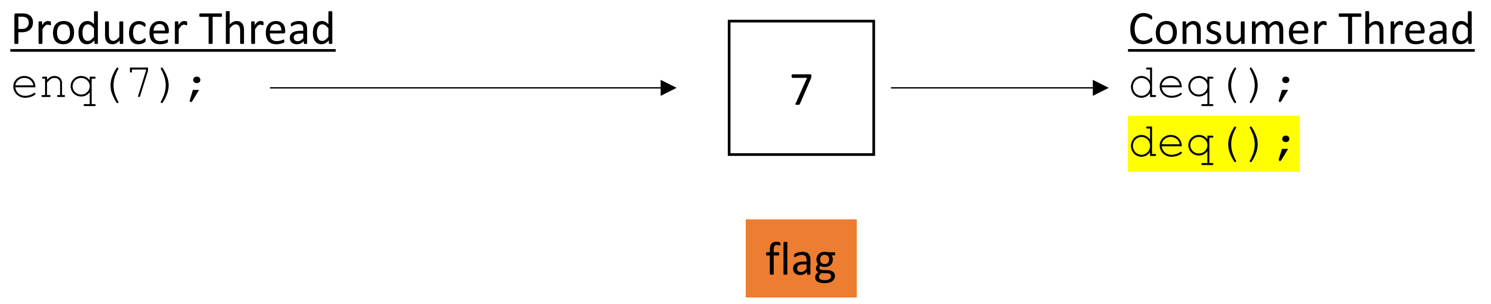
Consumer Thread

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



```
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
    }
  }
```

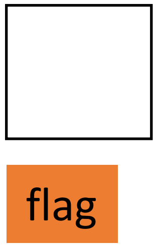
waiting like we are supposed to

Synchronous Producer Consumer Queues

reset (now with extra enq)

<u>Producer Thread</u>

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



<u>Consumer Thread</u>

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

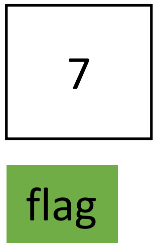
extra enq

```
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);
```



Consumer Thread

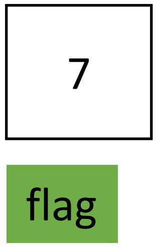
```
deq();
deq();
```

```
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);
```



Consumer Thread

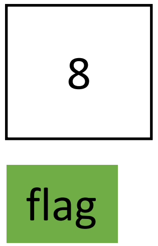
```
deq();
deq();
```

```
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);
```



8

flag

Consumer Thread

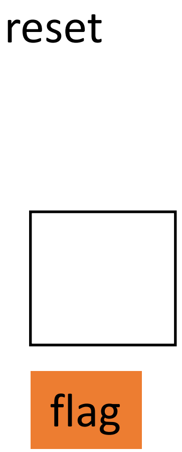
```
deq();
deq();
```

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
    }
    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>

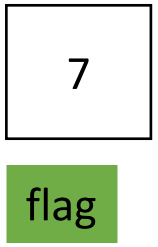
```
deq();
deq();
```

```
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
    }
  }
```

Synchronous Producer Consumer Queues

Producer Thread

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



7

flag

Consumer Thread

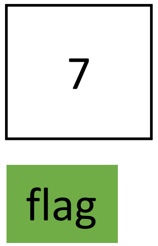
```
deq();
deq();
```

```
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

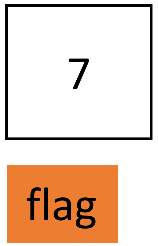
```
deq();
deq();
```

```
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

Producer Thread

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



7

flag

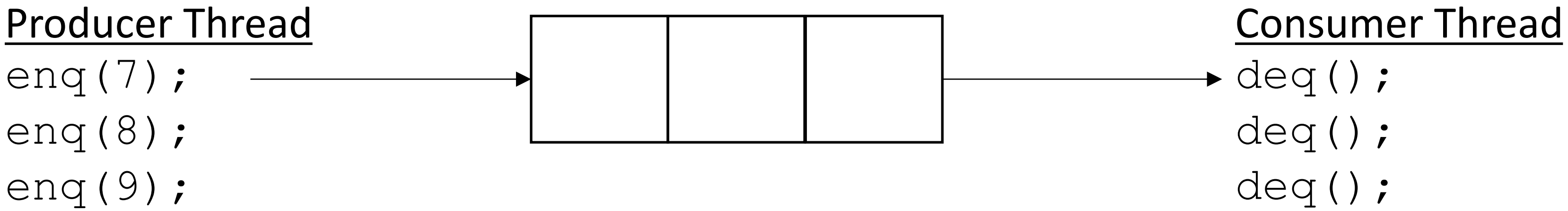
Consumer Thread

```
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
    }
  }
```

Producer Consumer Queues

- Asynchronous:



Producer Thread

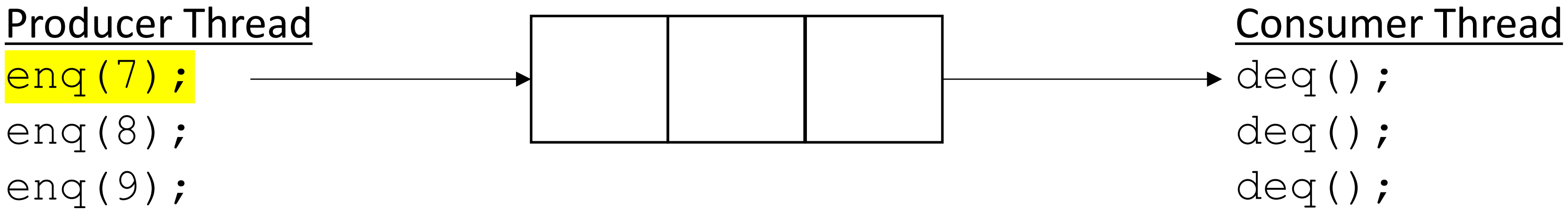
```
enq(7);
enq(8);
enq(9);
```

Consumer Thread

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

Producer Consumer Queues

- Asynchronous:



Producer Thread

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

Consumer Thread

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

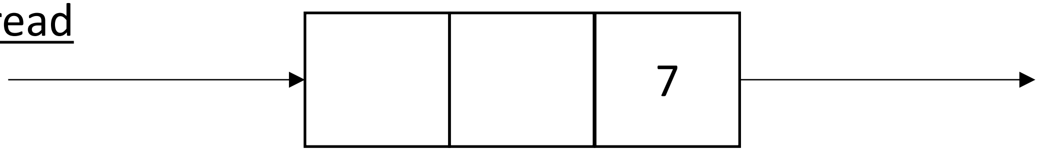
no waiting for producer (while there is room)

Producer Consumer Queues

- Asynchronous:

Producer Thread

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



Consumer Thread

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

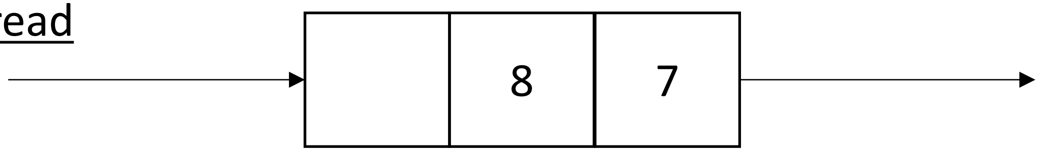
no waiting for producer (while there is room)

Producer Consumer Queues

- Asynchronous:

Producer Thread

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



Consumer Thread

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

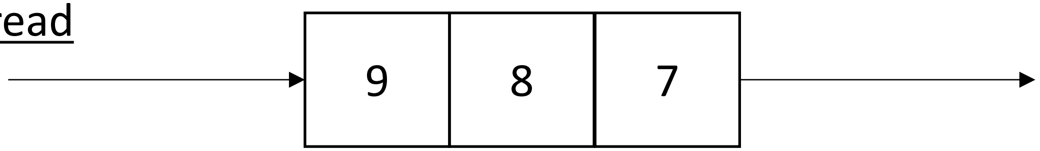
no waiting for producer (while there is room)

Producer Consumer Queues

- Asynchronous:

Producer Thread

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



Consumer Thread

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

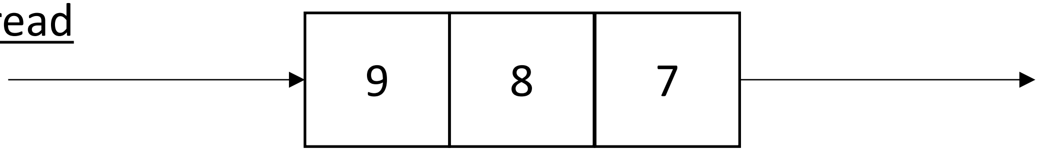
no waiting for producer (while there is room)

Producer Consumer Queues

- Asynchronous:

Producer Thread

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



9

8

7

Consumer Thread

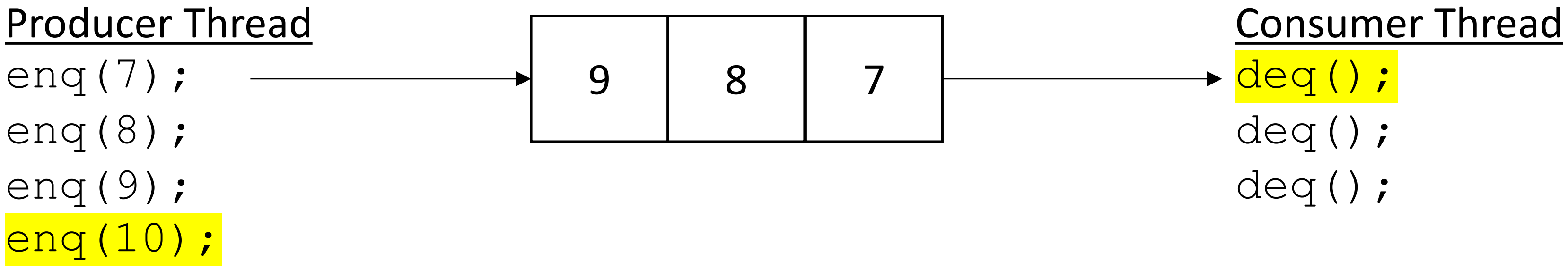
```
deq();
deq();
deq();
```

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

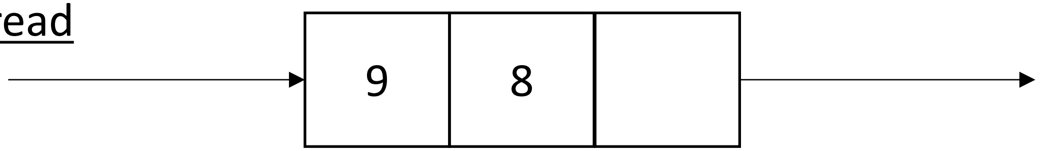
when there is no room, the queue will wait

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

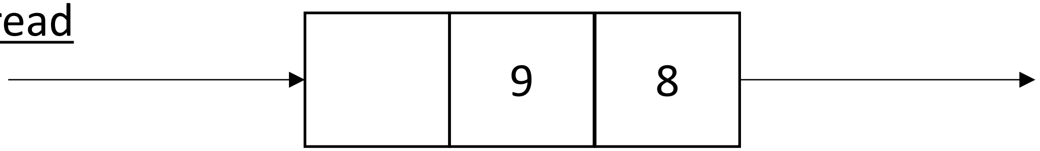
when there is no room, the queue will wait

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

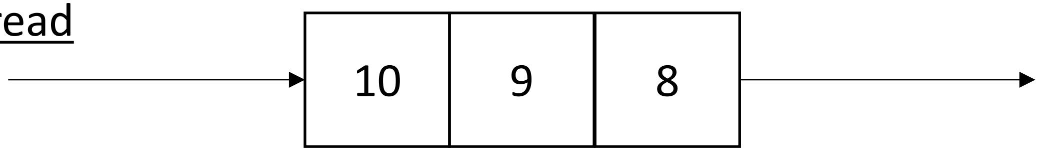
when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:

Producer Thread

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



Consumer Thread

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

finishes

no waiting for producer (while there is room)

returns 7

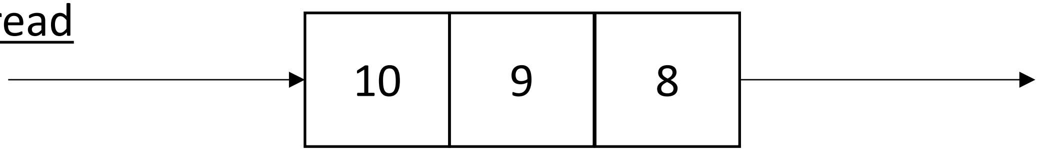
when there is no room, the queue will wait

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

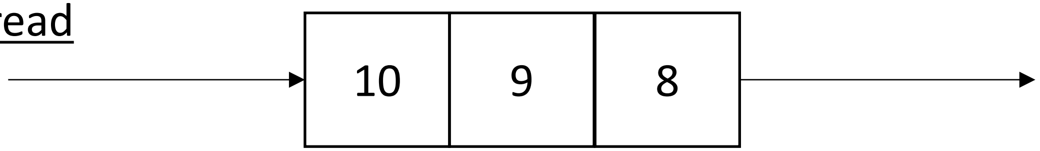
when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:

Producer Thread

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



10 9 8

Consumer Thread

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

no waiting for producer (while there is room)

returns 8

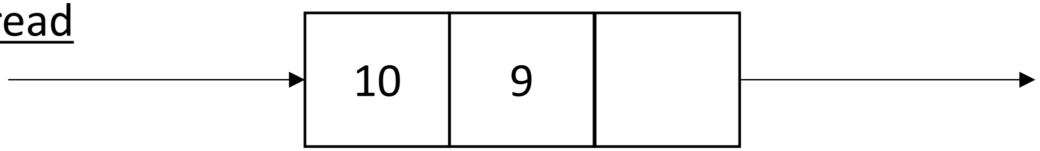
when there is no room, the queue will wait

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

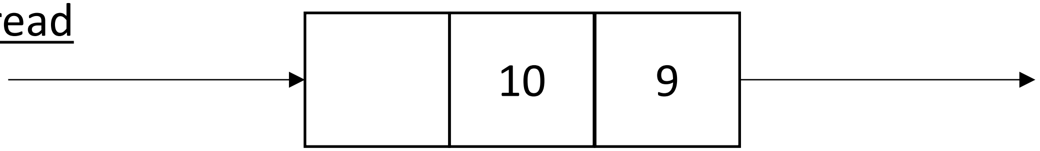
when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:

Producer Thread

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



Consumer Thread

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

no waiting for producer (while there is room)

returns 8

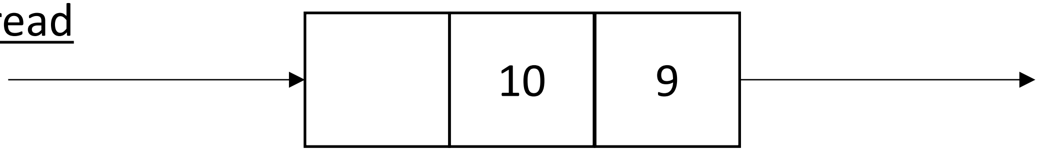
when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:

Producer Thread

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



Consumer Thread

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

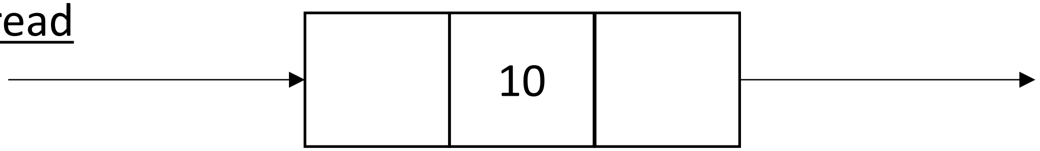
returns 9

Producer Consumer Queues

- Asynchronous:

<u>Producer Thread</u>

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

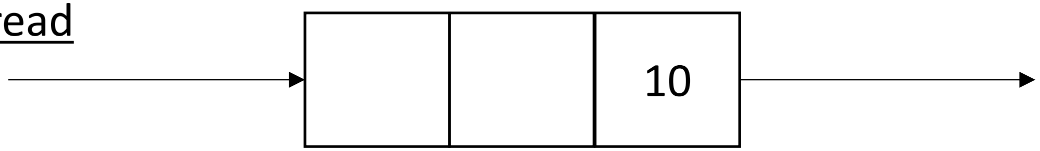


<u>Consumer Thread</u>

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

Producer Consumer Queues

- Asynchronous:



Producer Thread

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

Consumer Thread

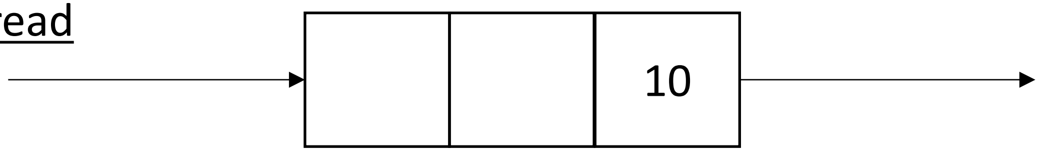
```
deq();
deq();
deq();
deq();
```

Producer Consumer Queues

- Asynchronous:

<u>Producer Thread</u>

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



<u>Consumer Thread</u>

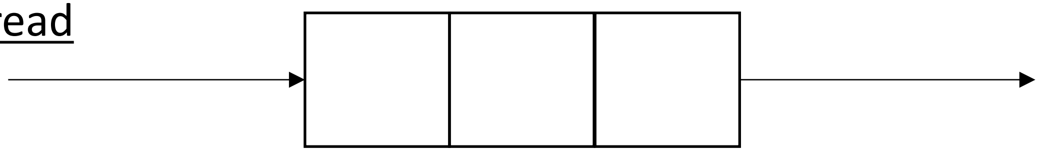
```
deq();
deq();
deq();
deq();
```

Producer Consumer Queues

- Asynchronous:

Producer Thread

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



Consumer Thread

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

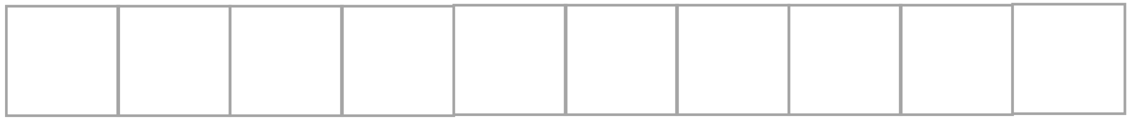
blocks when there is nothing in the queue

Producer Consumer Queues

- How do we implement it?

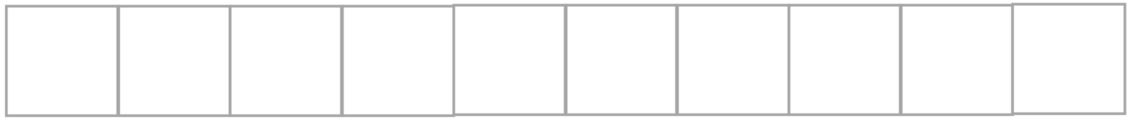
Producer Consumer Queues

- Start with a fixed size array



Producer Consumer Queues

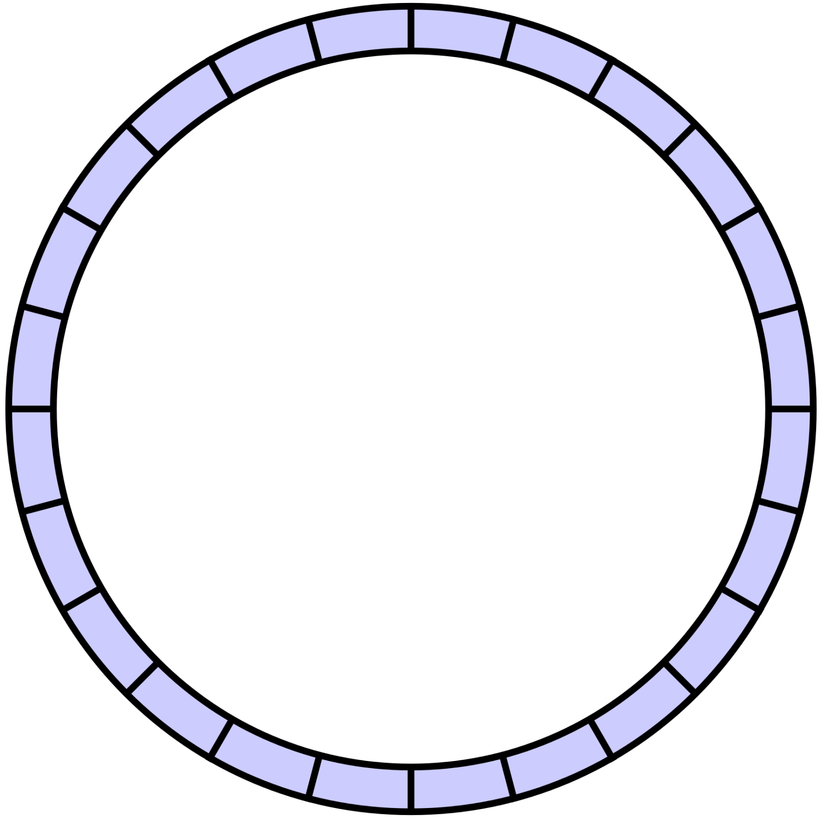
- Start with a fixed size array



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

Producer Consumer Queues

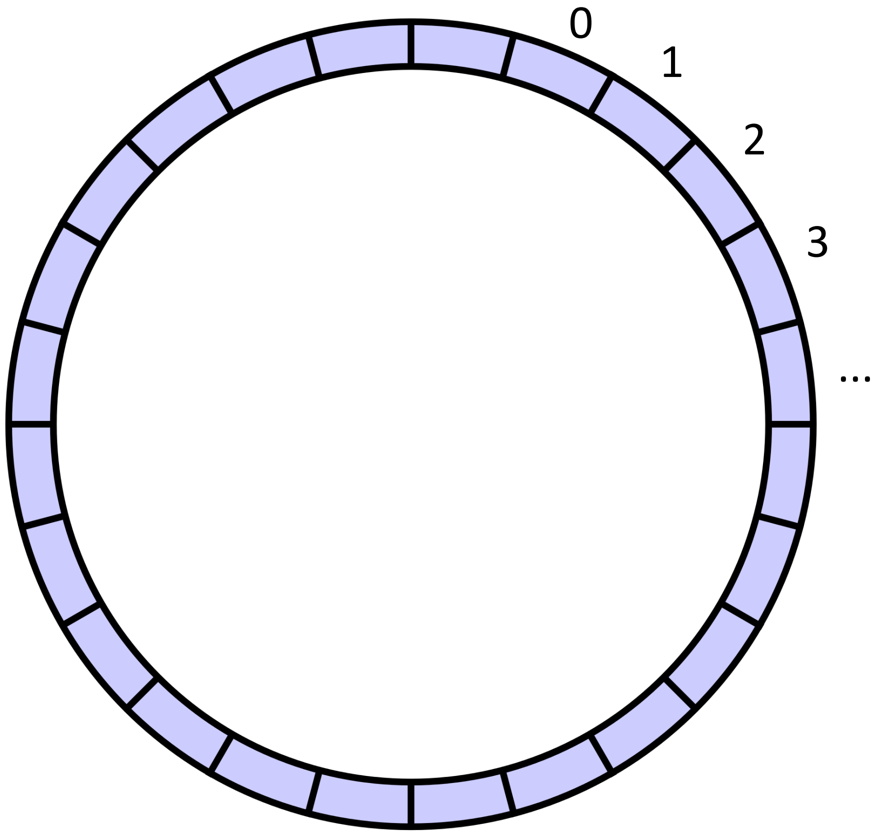
- Start with a fixed size array



conceptually it is a circle

Producer Consumer Queues

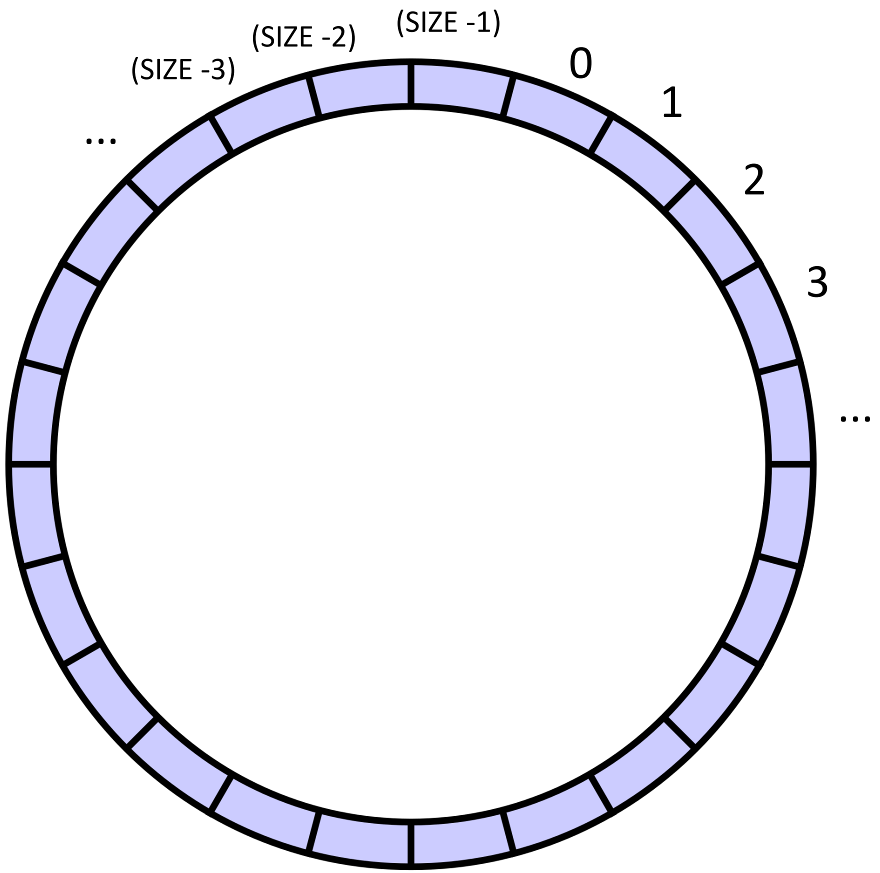
- Start with a fixed size array



conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array



indexes will circulate in order and wrap around

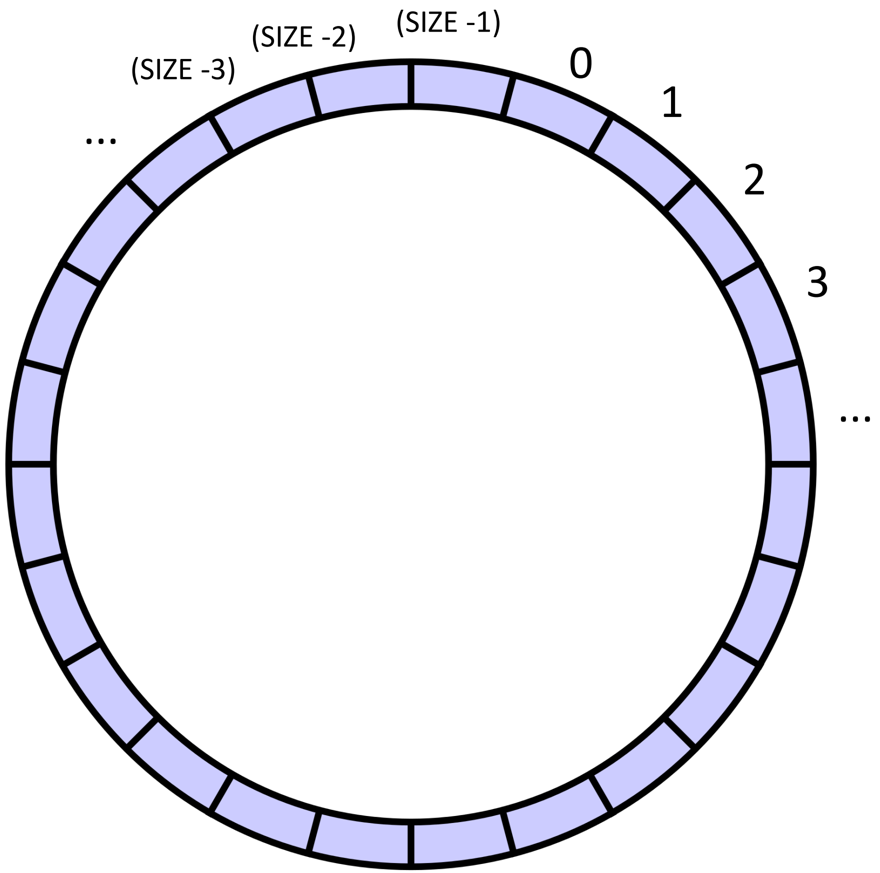
conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array

we will assume modular arithmetic:

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



indexes will circulate in order and wrap around

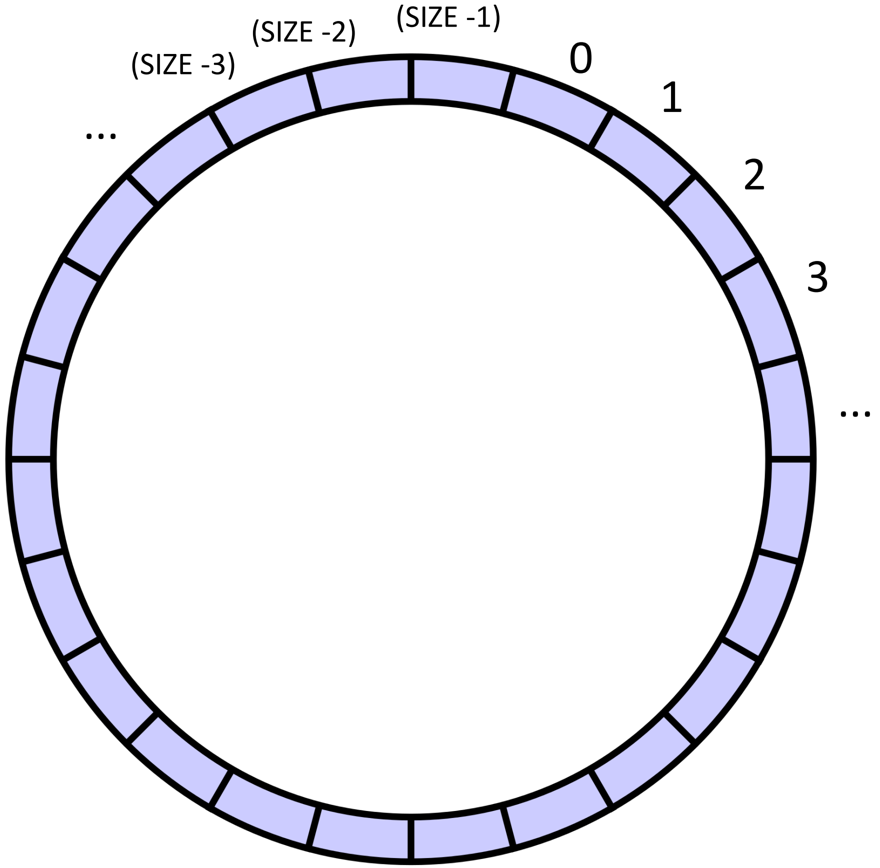
conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array

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

head and tail



indexes will circulate in order and wrap around

conceptually it is a circle

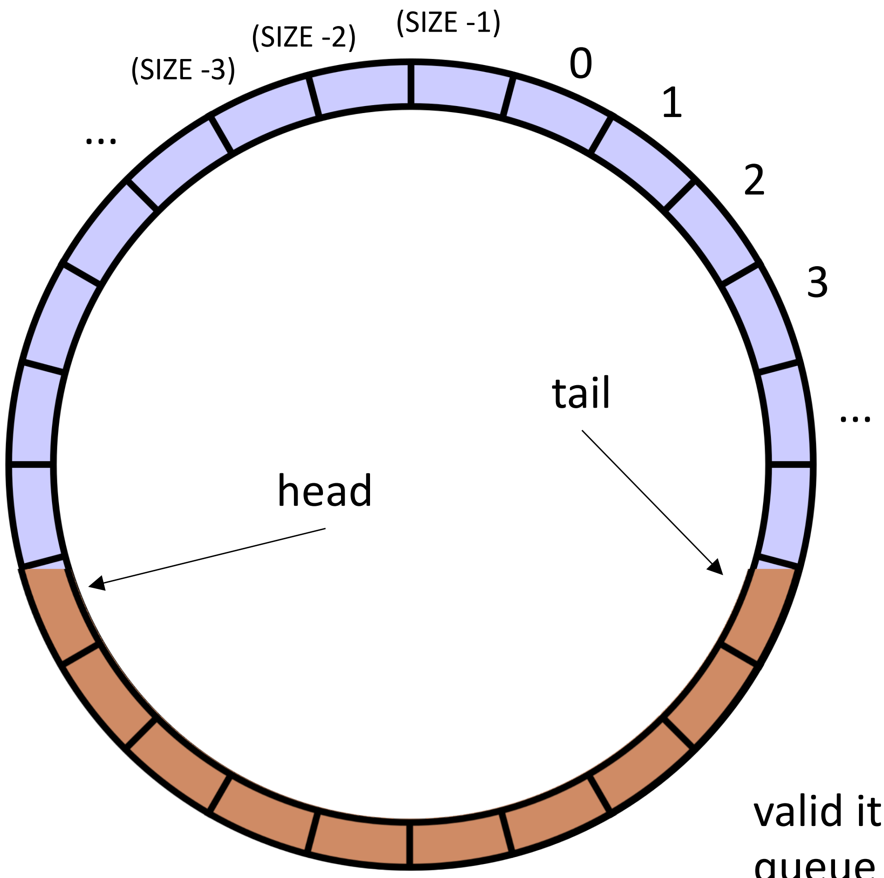
Producer Consumer Queues

- 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

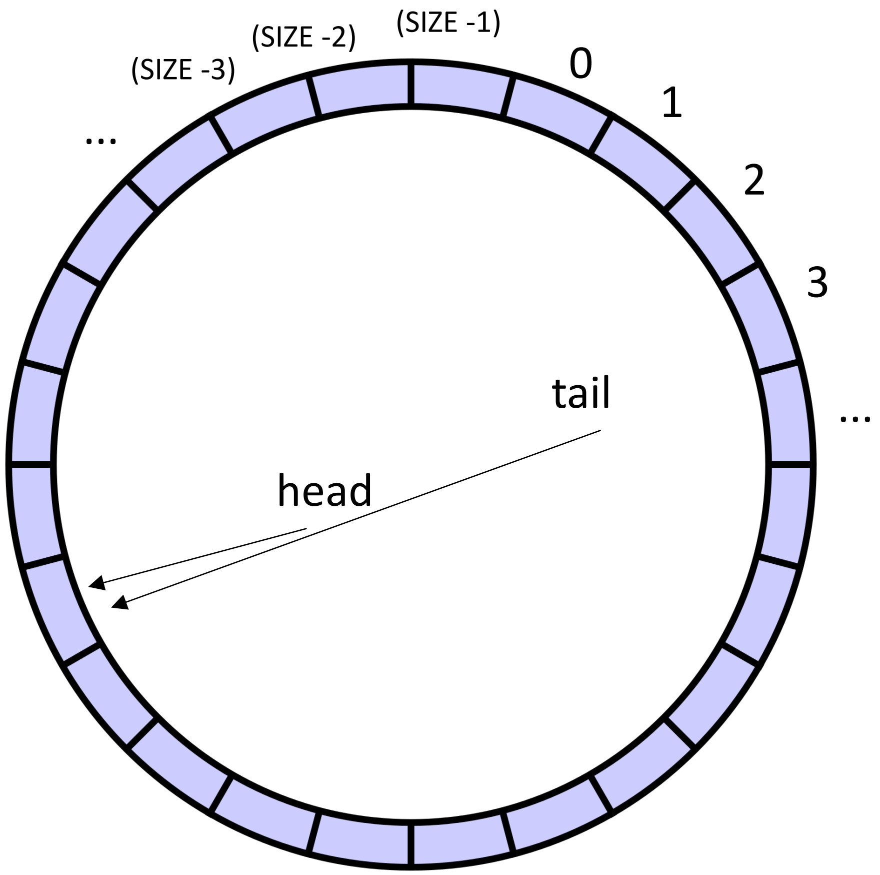
Producer Consumer Queues

- Start with a fixed size array

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

head and tail

Empty queue is when head == tail



indexes will circulate in order and wrap around

conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array

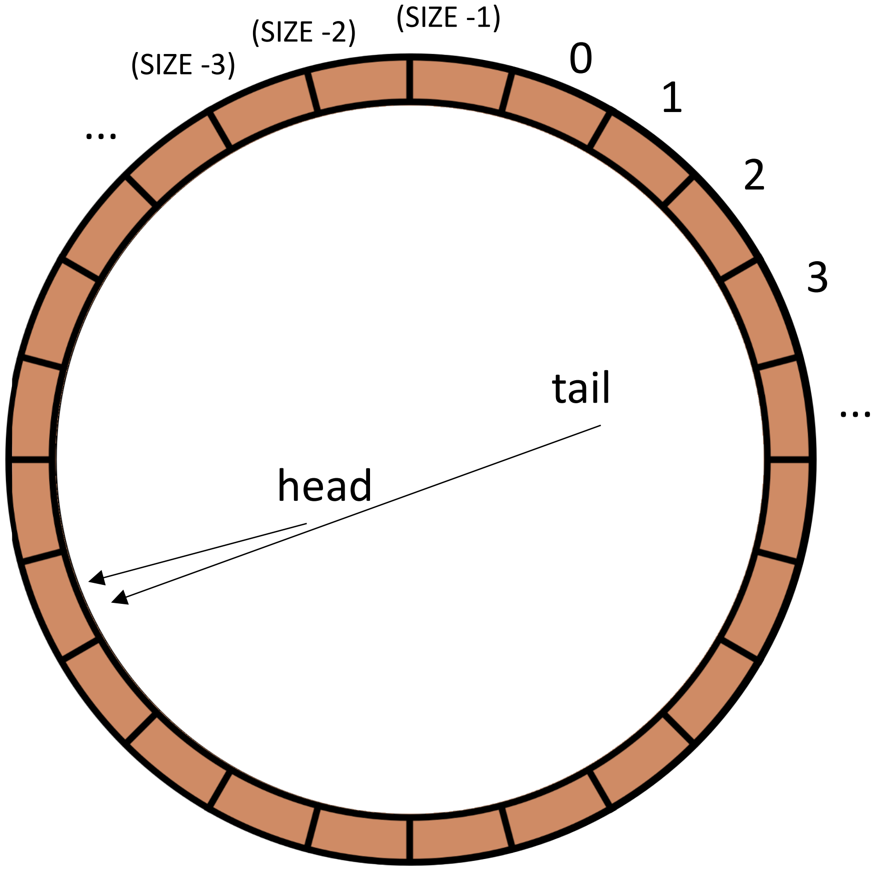
Two variables to keep track of where to deq and enq:

head and tail

Empty queue is when head == tail

Full queue is when head == tail?

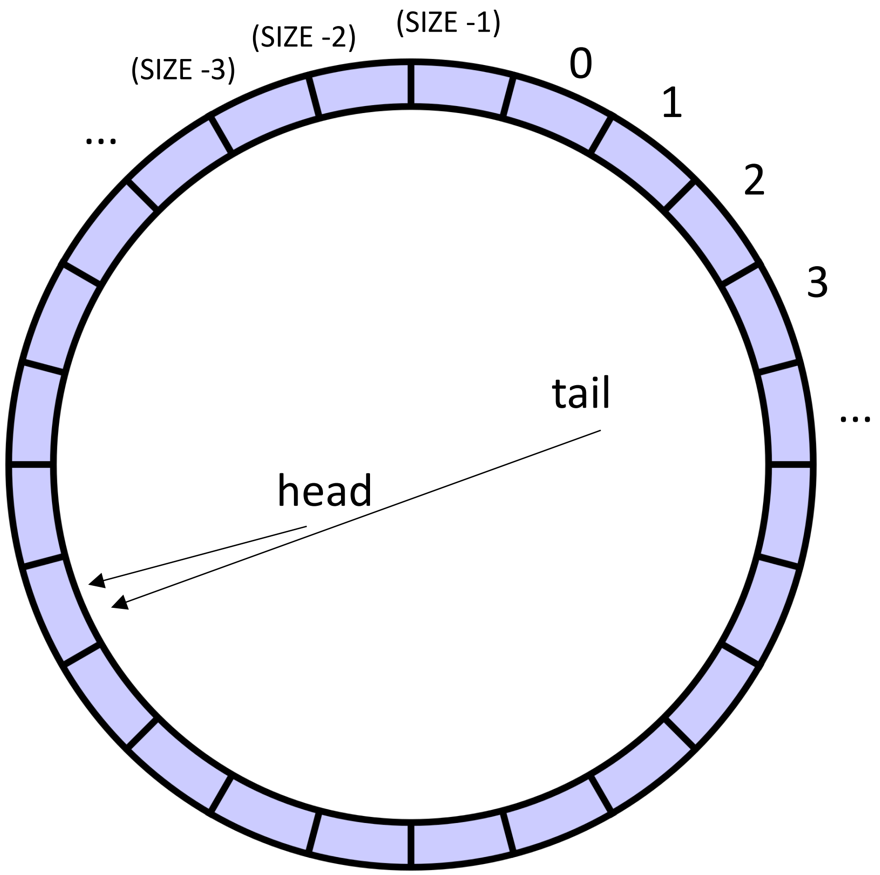
conceptually it is a circle



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

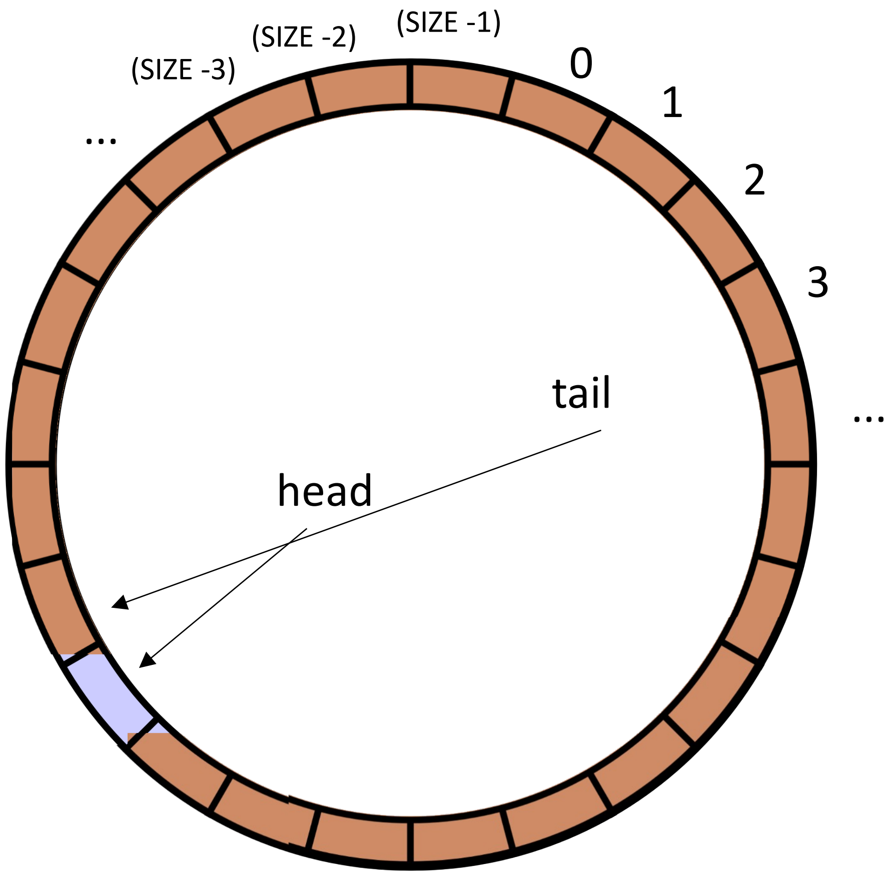
Full queue is when head == tail?

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:

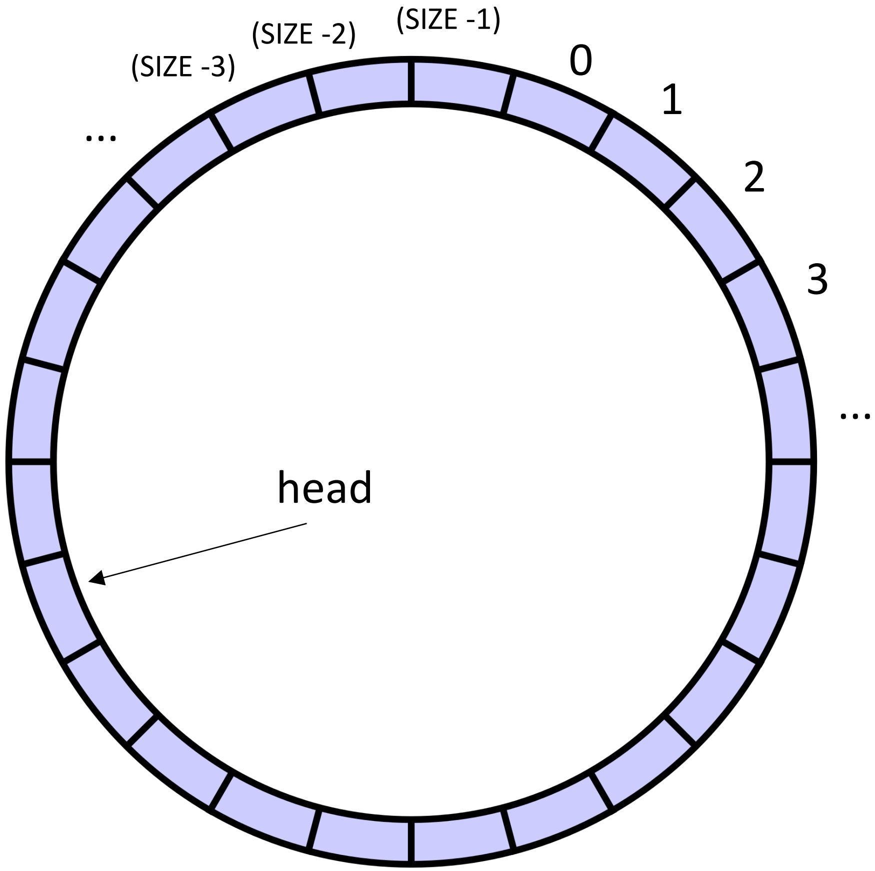
head and tail

Empty queue is when head == tail

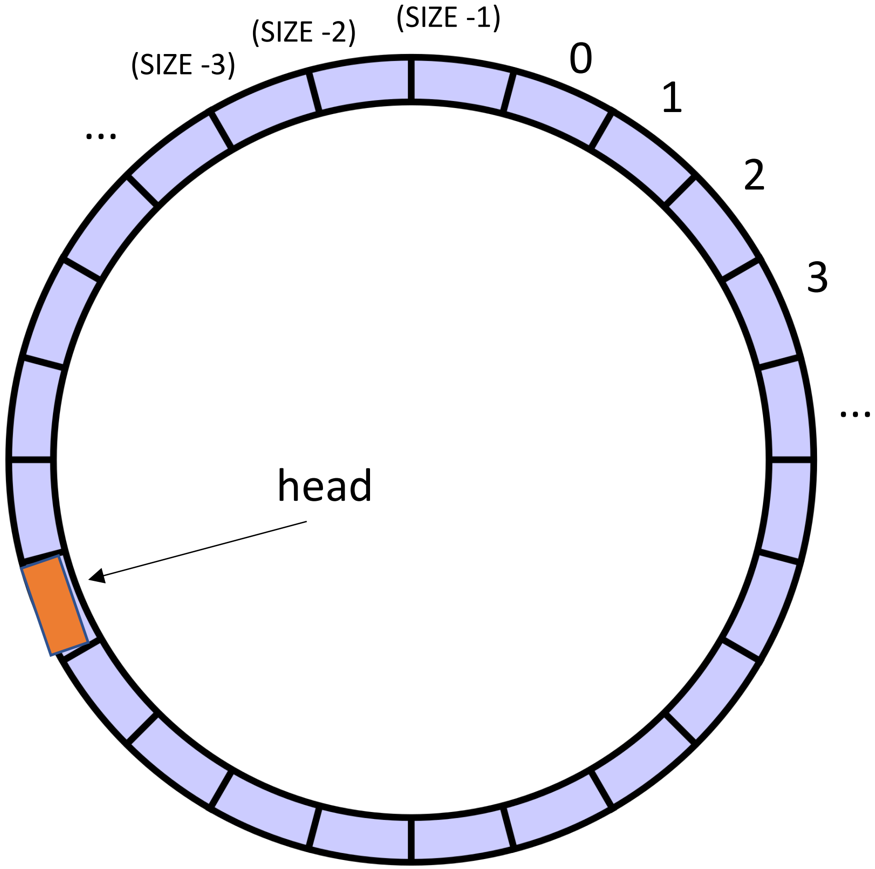
Full queue is when head + 1 == tail

wasting one location, but its okay...

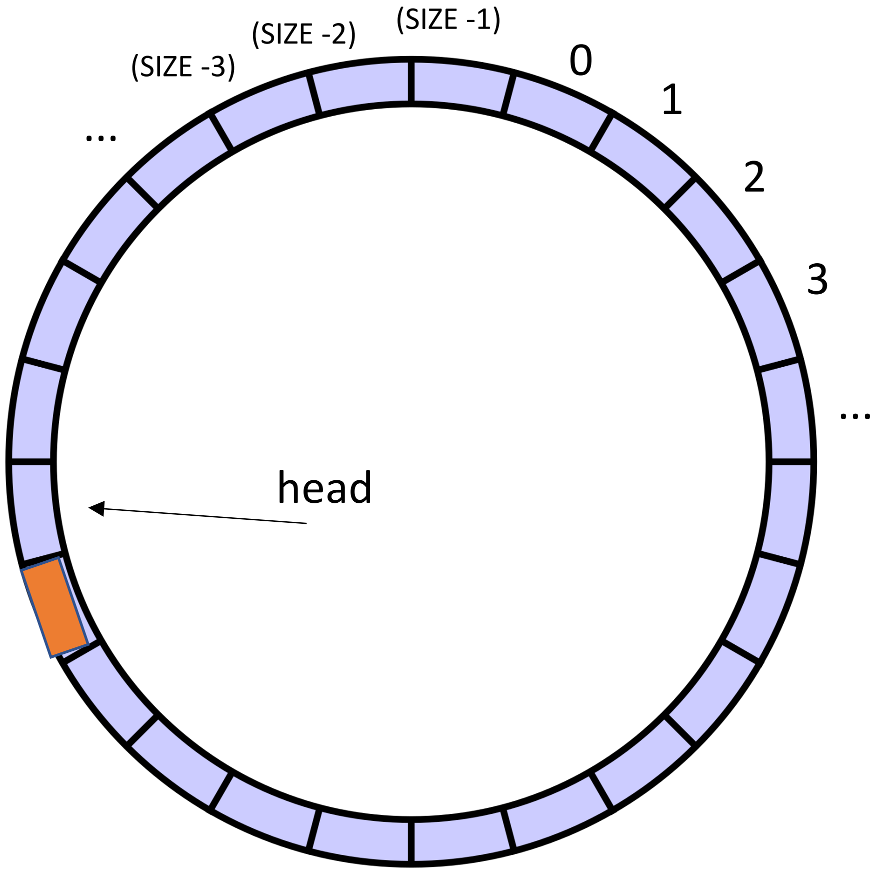
conceptually it is a circle



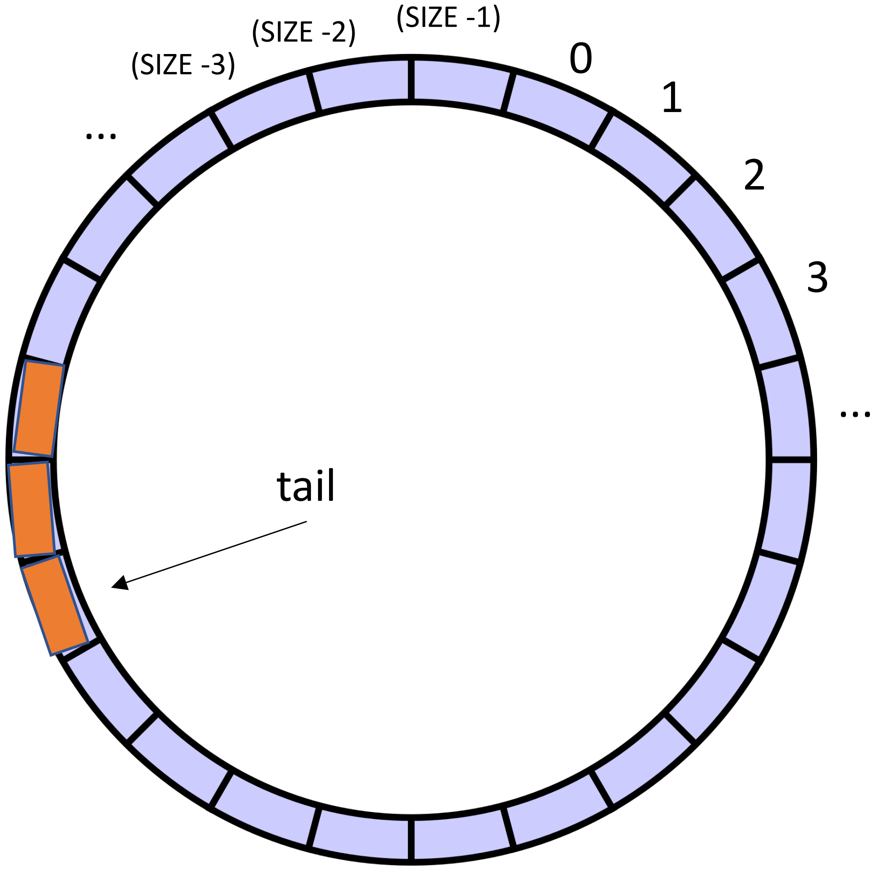
```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

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



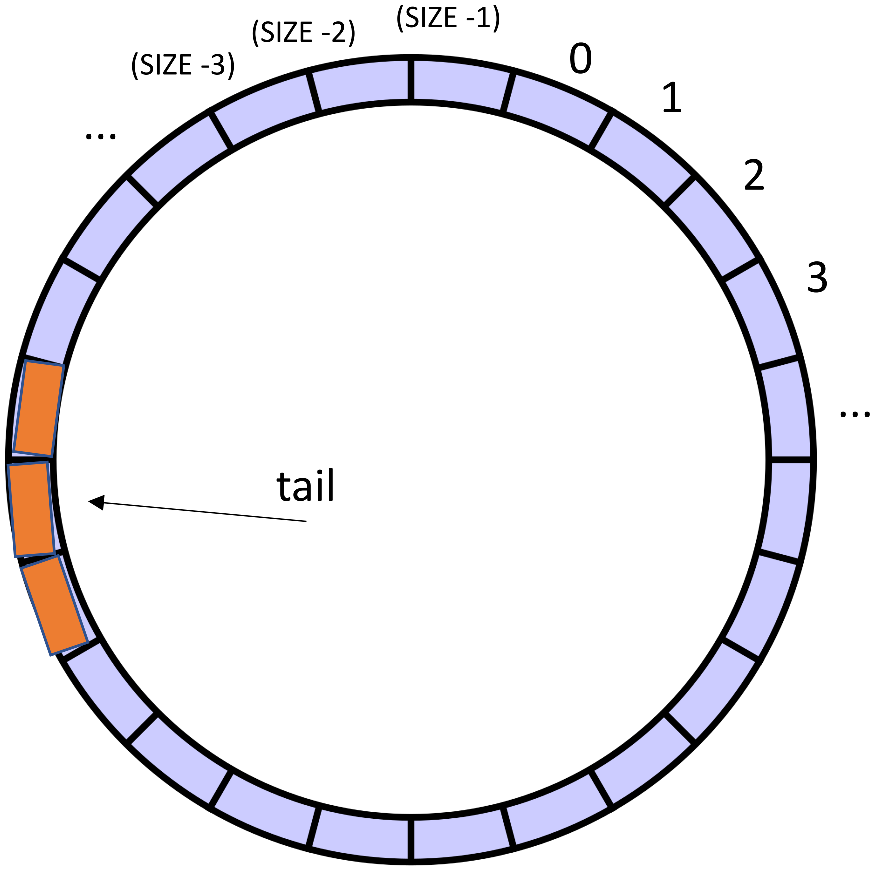
```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  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
    }
}
```

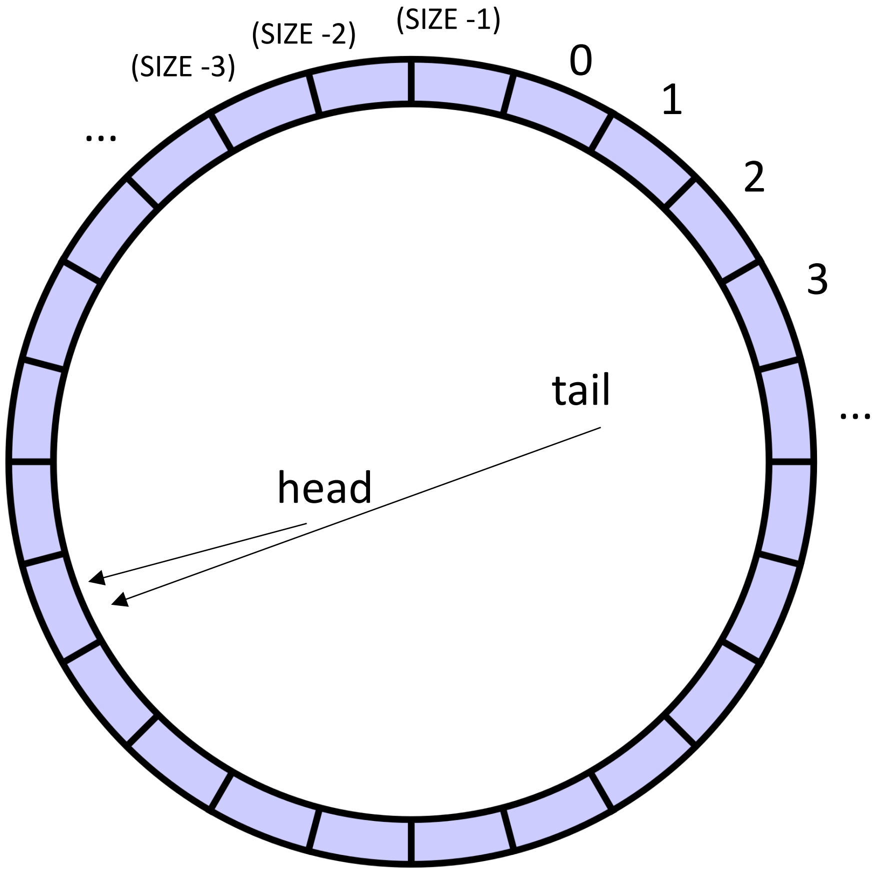


```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

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



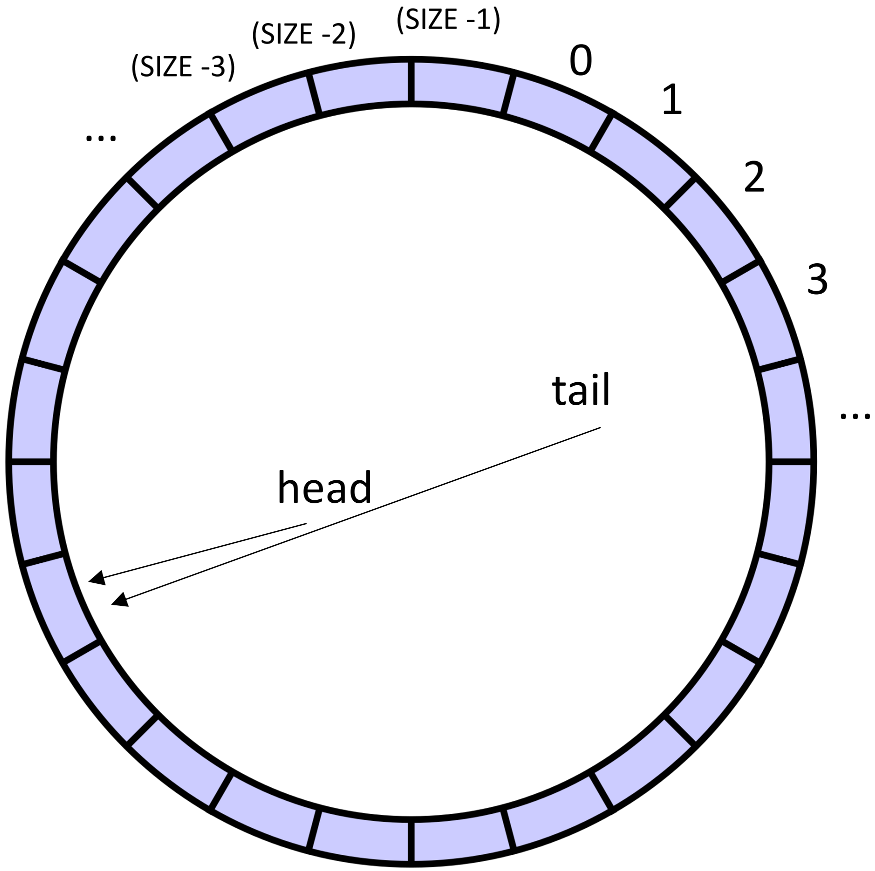
```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  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?

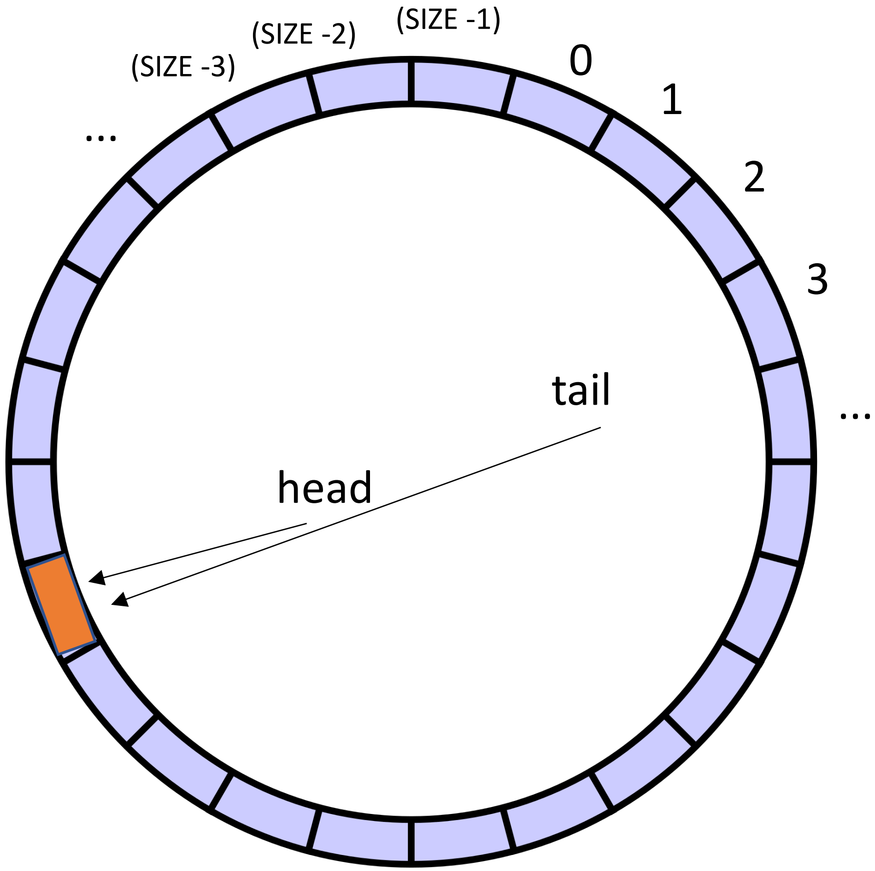


```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

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

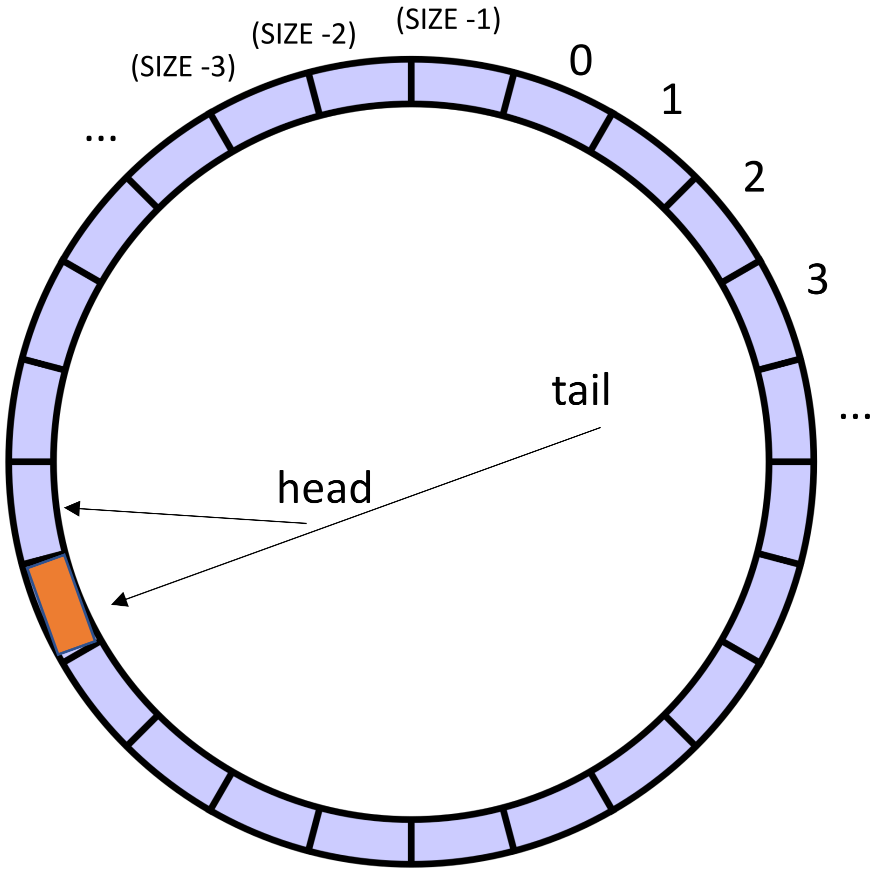
what happens if we try to dequeue here?



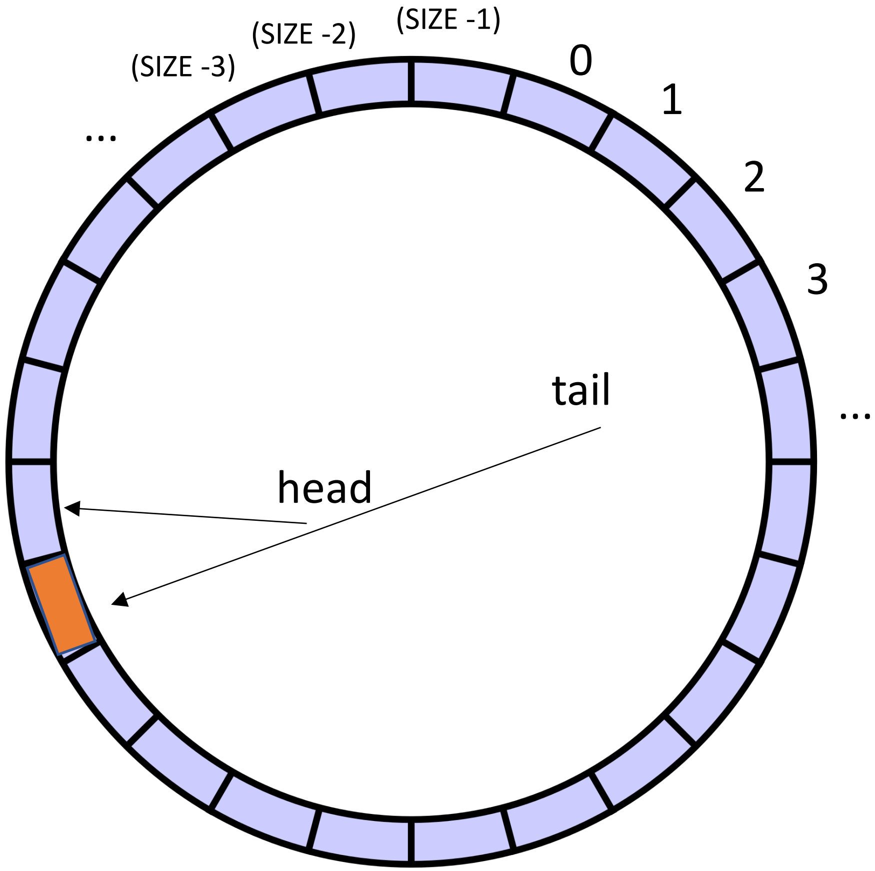
```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

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



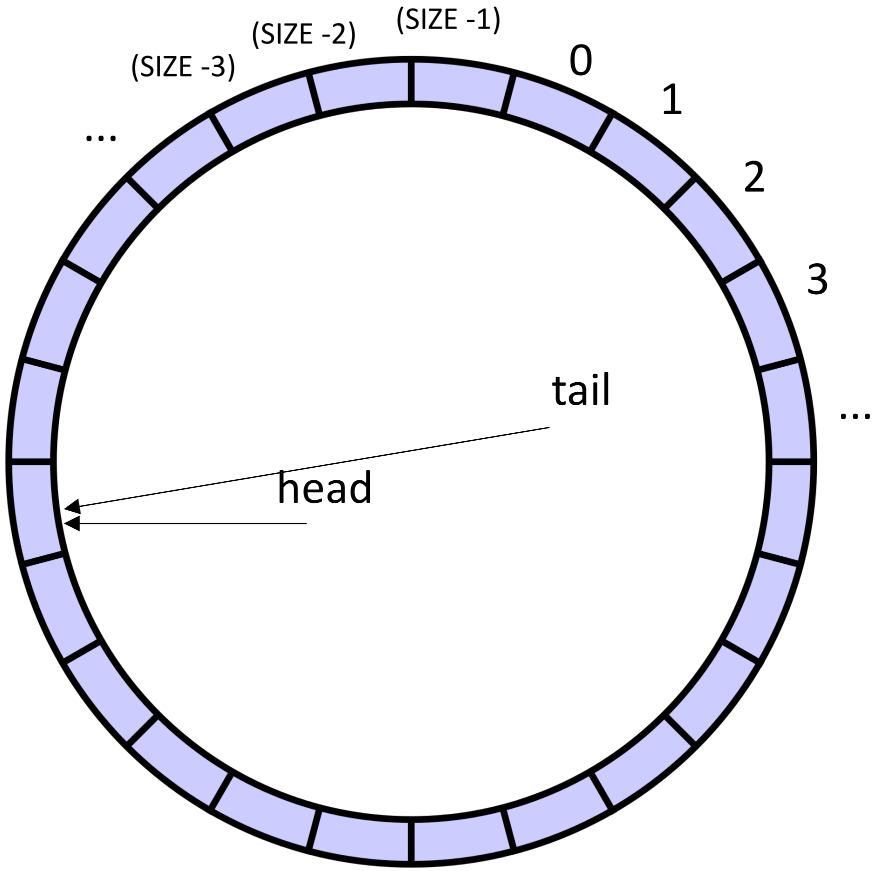
```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

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



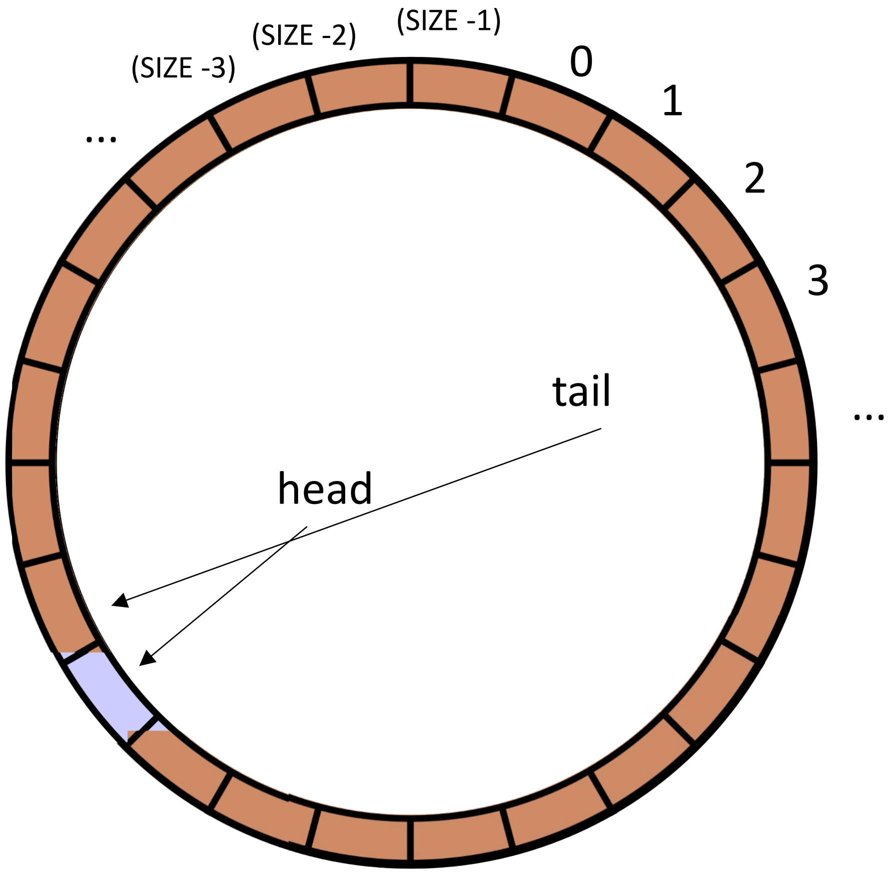
```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  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
    }
}
```



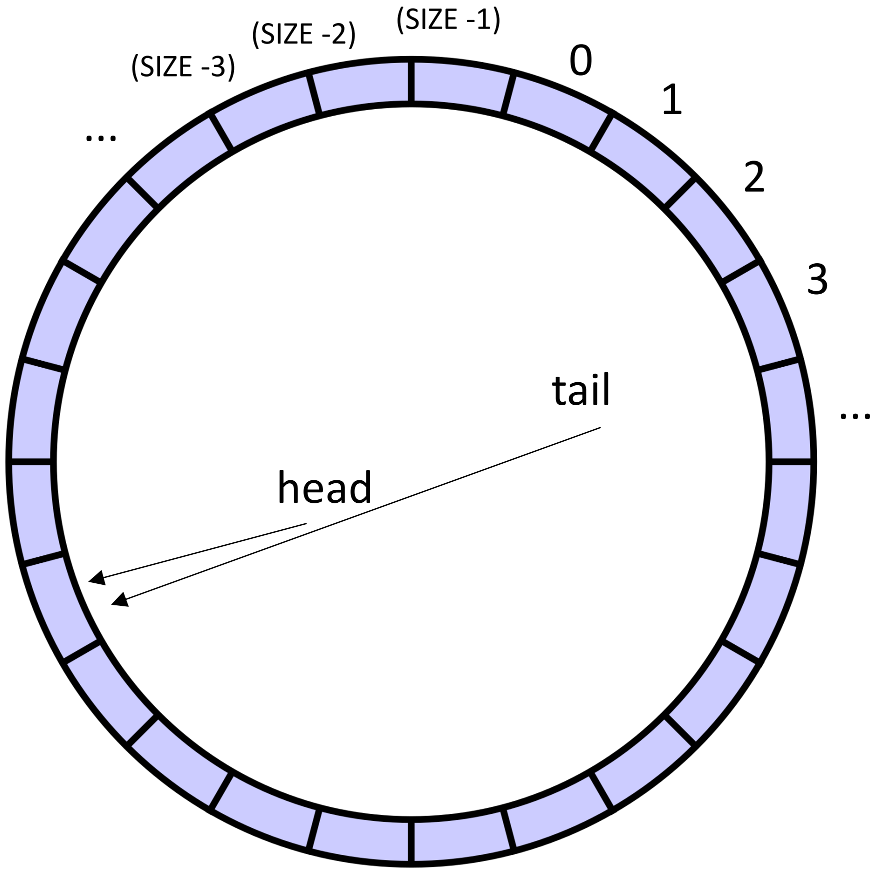
```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  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
    }
}
```

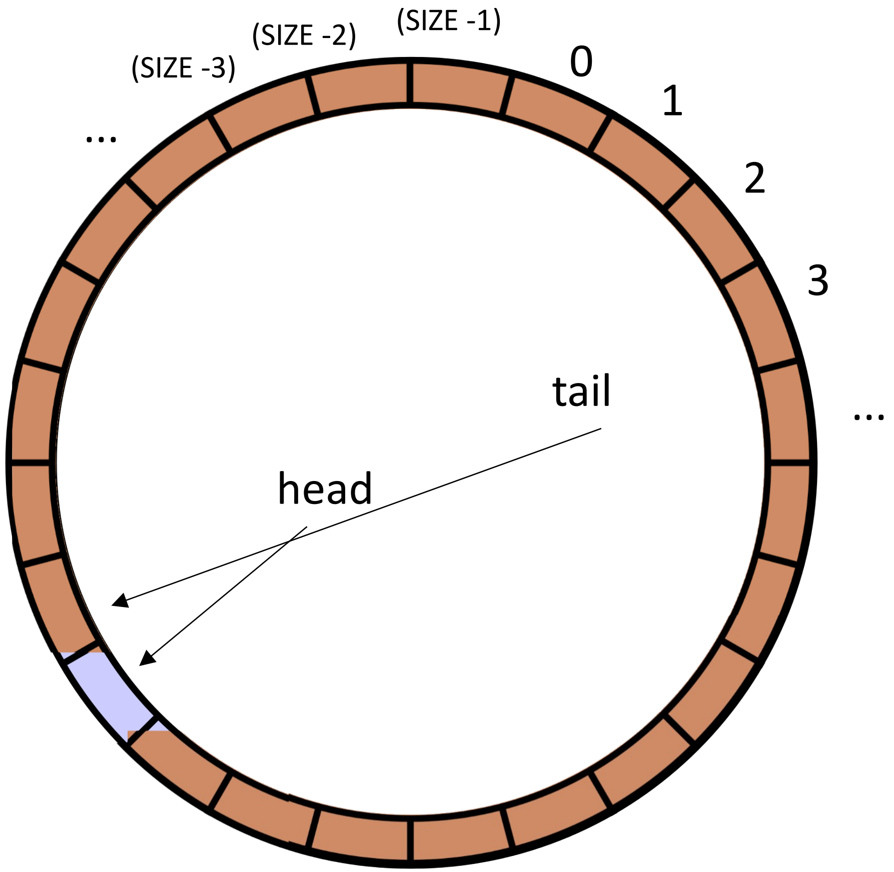
similarly for enqueue

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
    }
}
```

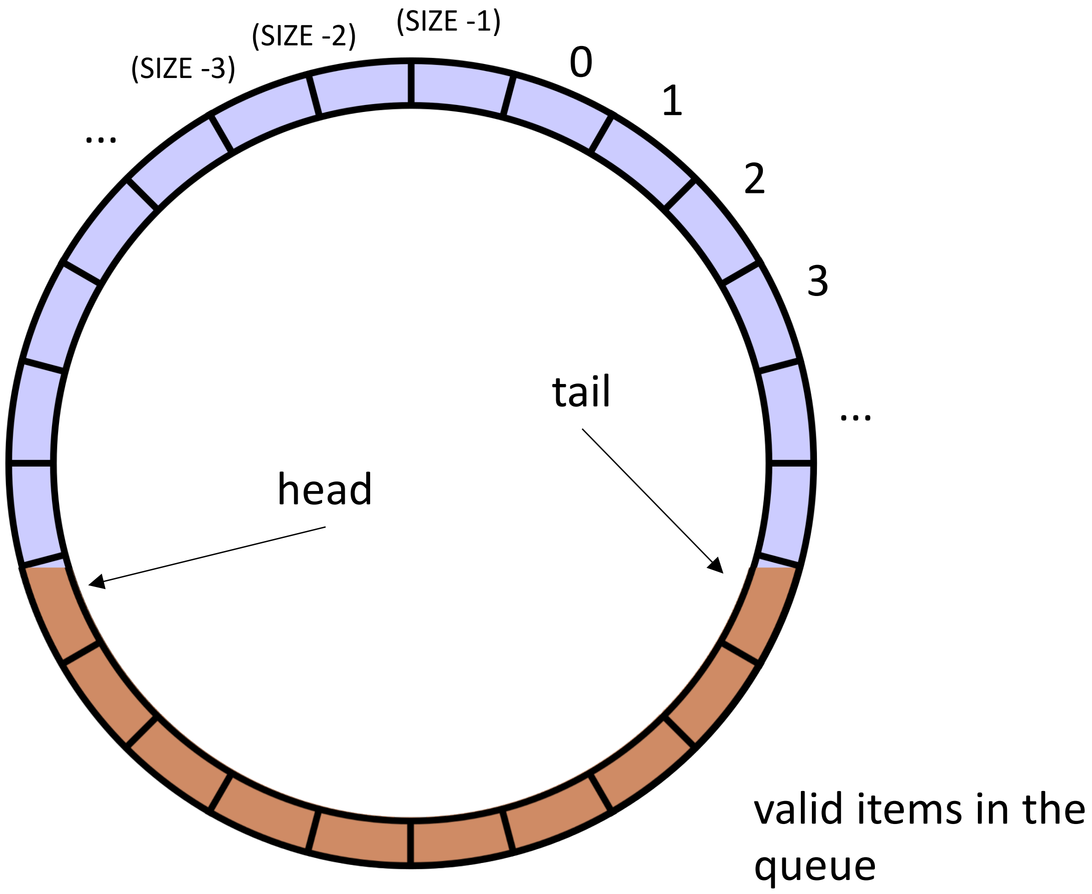
incrementing the head would make it empty!



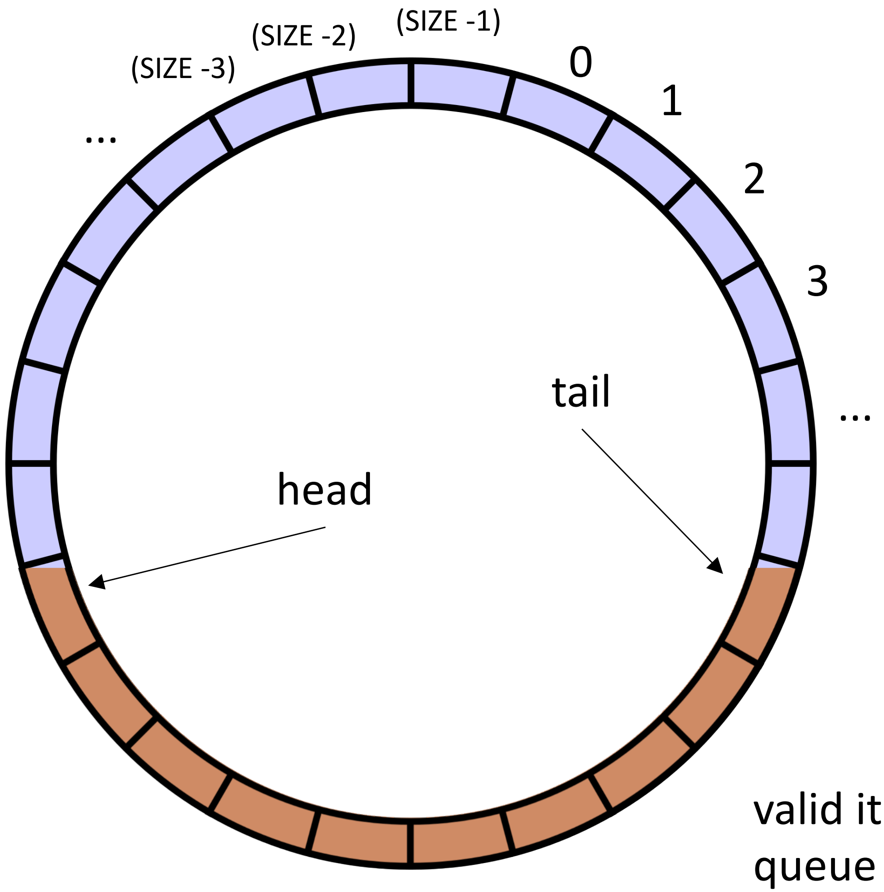
```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

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



valid items in the queue

Other questions:

Do these need to be atomic RMWs?
Remember 1 thread enqueues and 1 thread dequeues

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

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