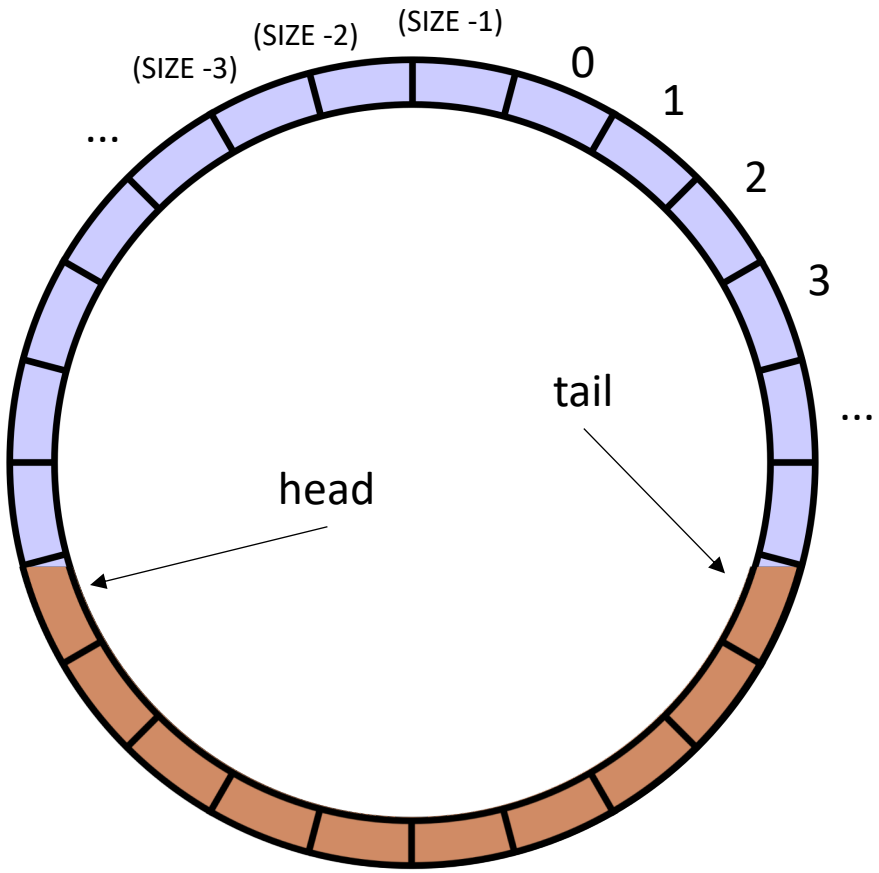
CSE113: Parallel Programming

Feb, 14, 2024

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



Announcements

- Midterm is over!
  - We plan on grading next Friday
  - If there is time, we can go over in class.
    - Won’t be until next Wednesday at the earliest (we need to get all the lectures required for HW 3)
    - That lecture will not be recorded and the slides will not be uploaded. Please attend in person if you want this review.
- We can go over midterm in my office hours, most useful after getting the grade back

Announcements

- HW 3 is scheduled to be released today by midnight.
- 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

Announcements

- HW 1 grades are out
- If you have questions, please make a piazza post to all instructors
- Happy to discuss in office hours too, but probably better on Piazza
- If you got a 0, that is probably because we couldn’t link your account. Post an instructor private post on piazza
- You have 1 week (from now) to raise any issues related to grading

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;
```

what if you have multiple objects?

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();
}
```

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

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;
```

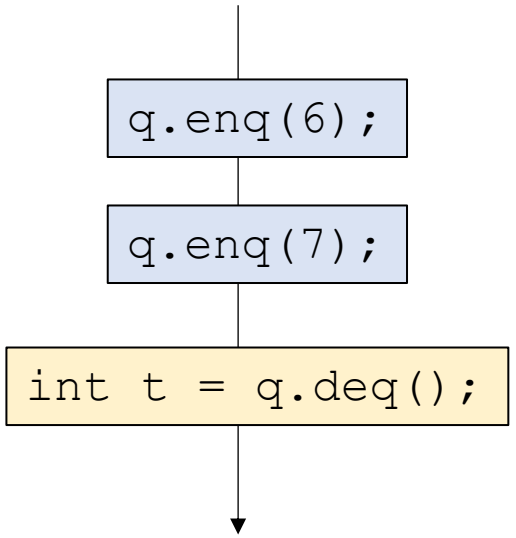
Thread 0:

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

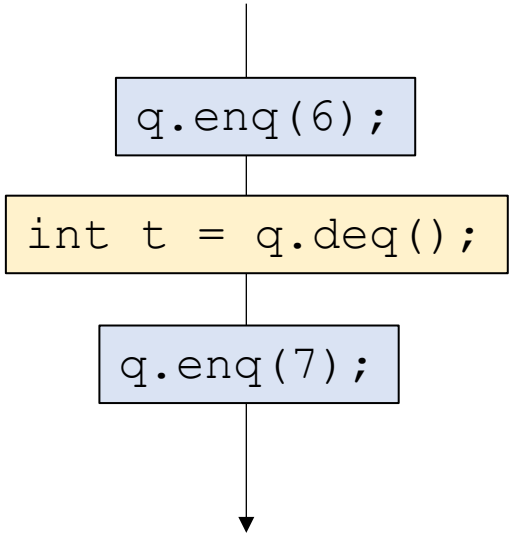
Thread 1:

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

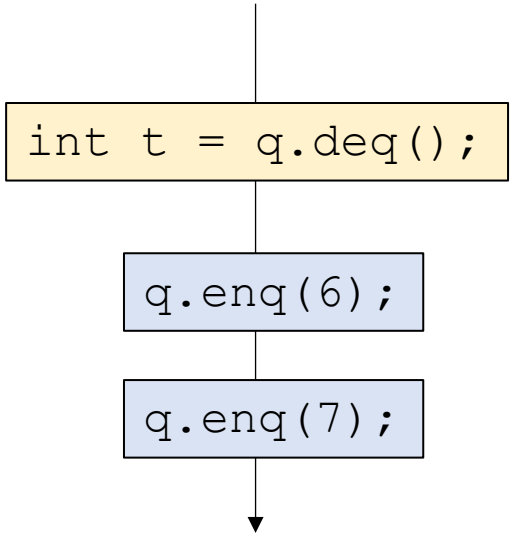
Construct a sequential timeline of API calls
Any sequence is valid:



t is 6



t is 6



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

Didn’t get to this one, sorry!

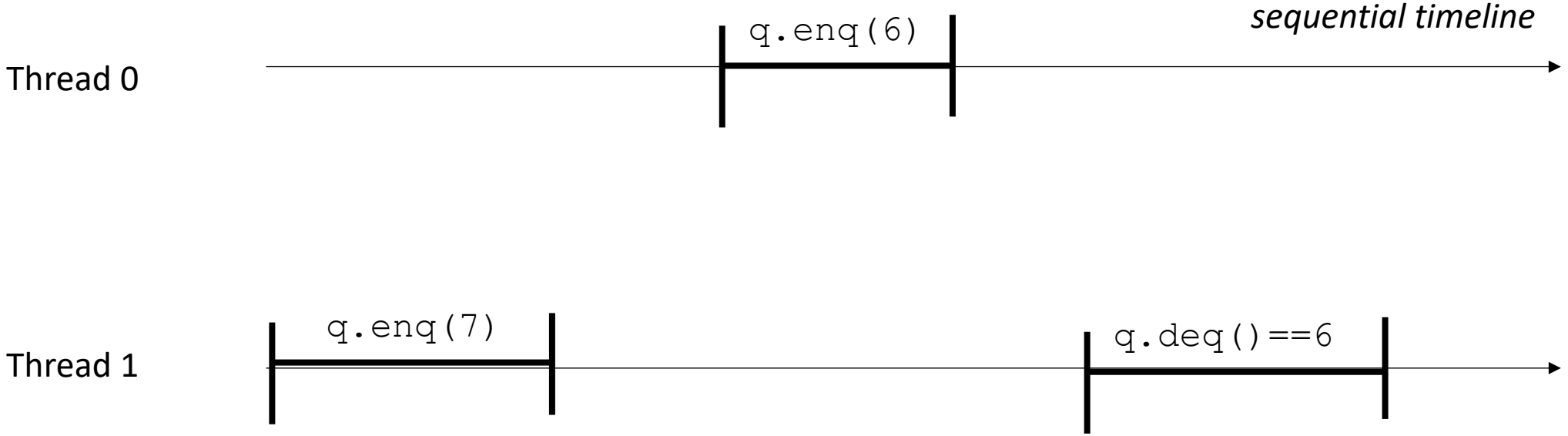
Review

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

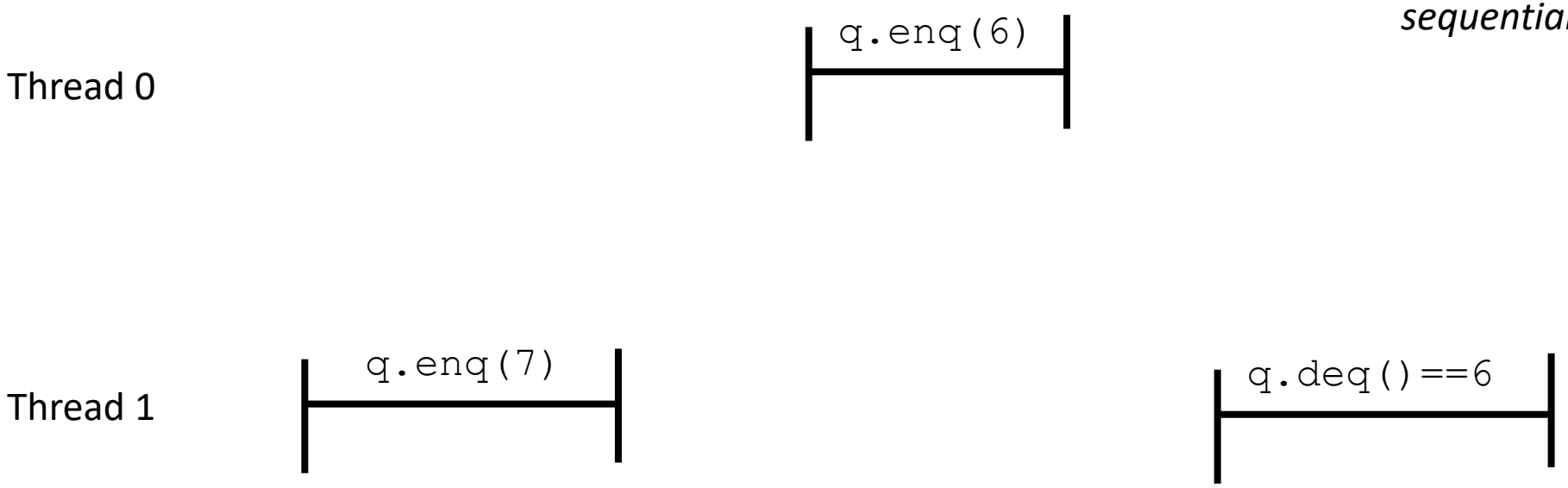


Sequential consistency and real time

- Add in real time:

This execution is allowed in sequential consistency!

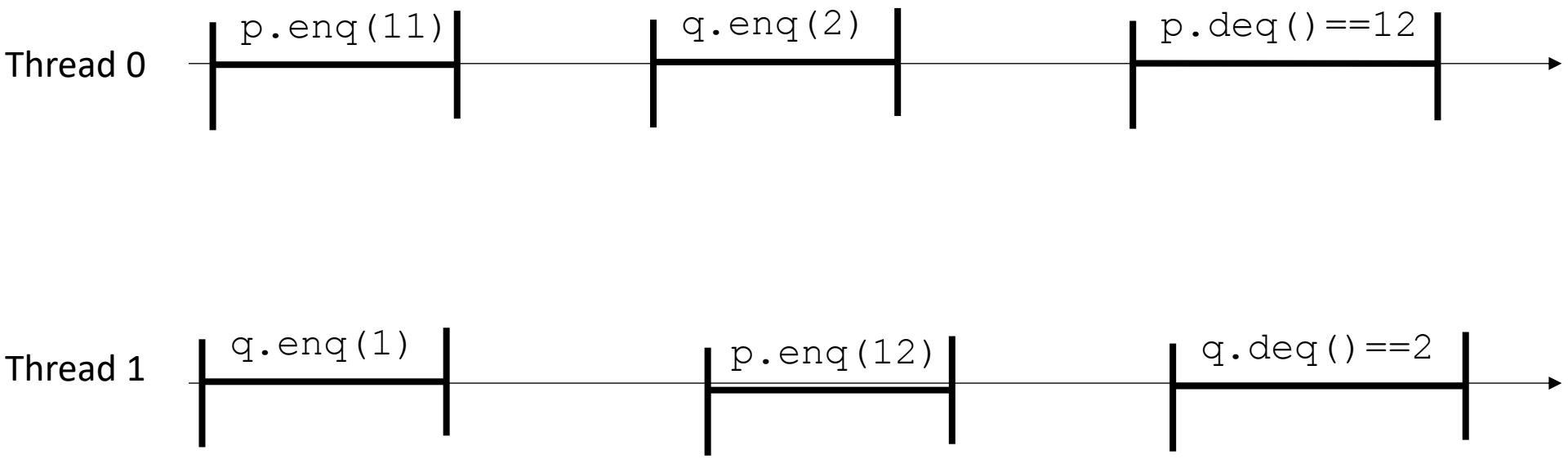
SC doesn't care about real time, only if it can construct its virtual sequential timeline



Sequential consistency and real time

- Add in real time:

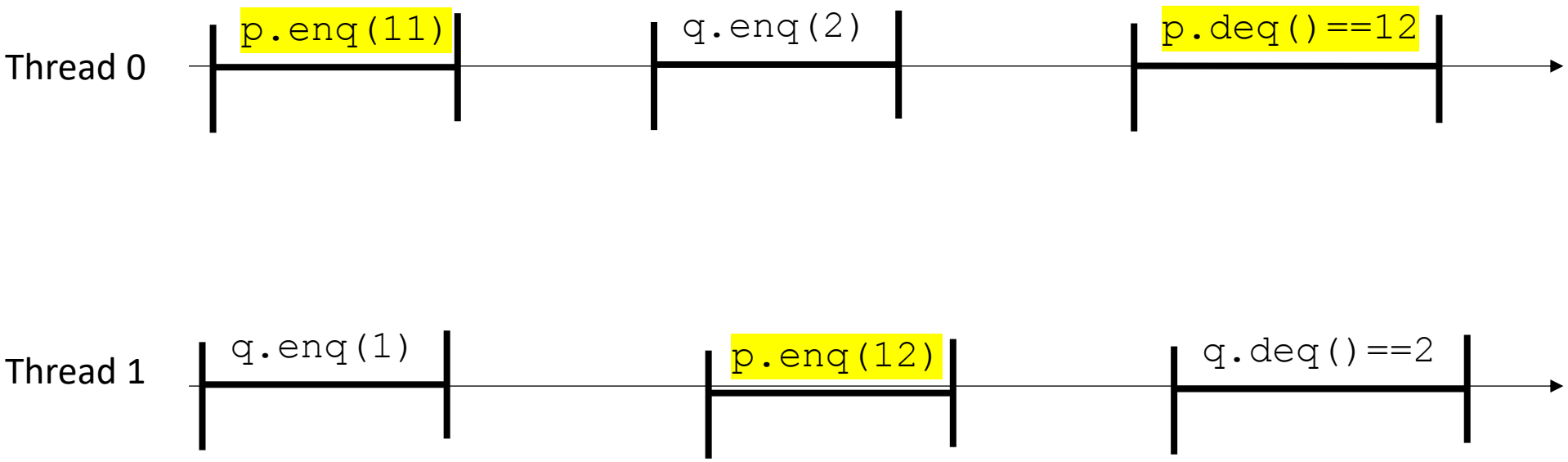
2 objects now: p and q



Sequential consistency and real time

- Add in real time:

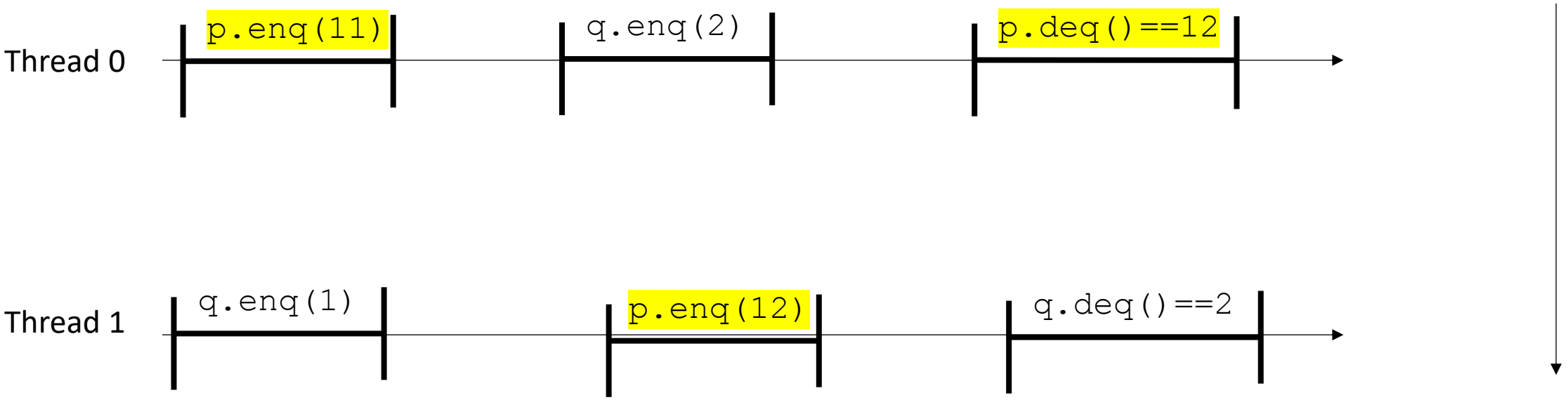
2 objects now: p and q
Consider each object in isolation



Sequential consistency and real time

- Add in real time:

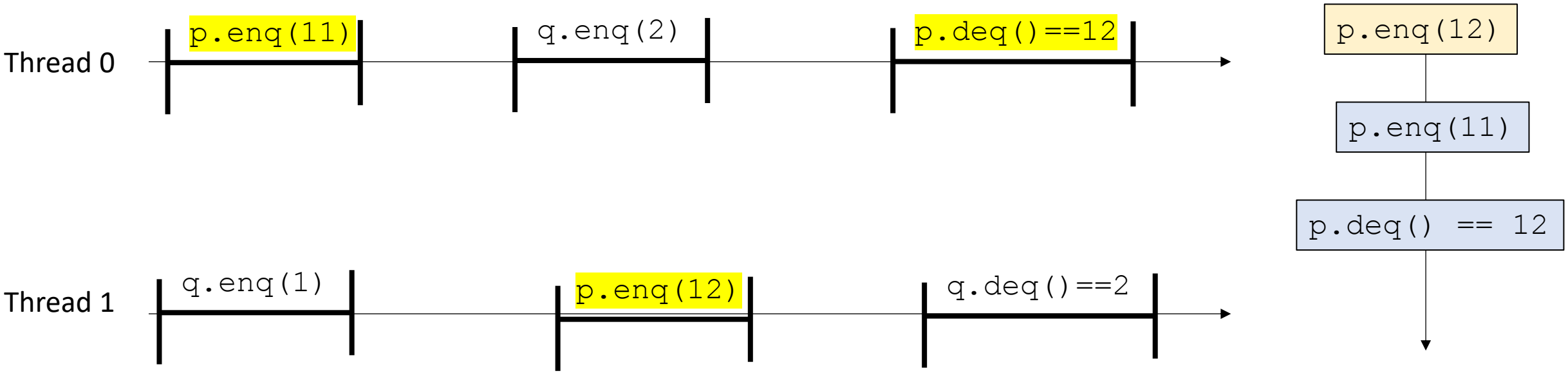
2 objects now: p and q
Consider each object in isolation



Sequential consistency and real time

- Add in real time:

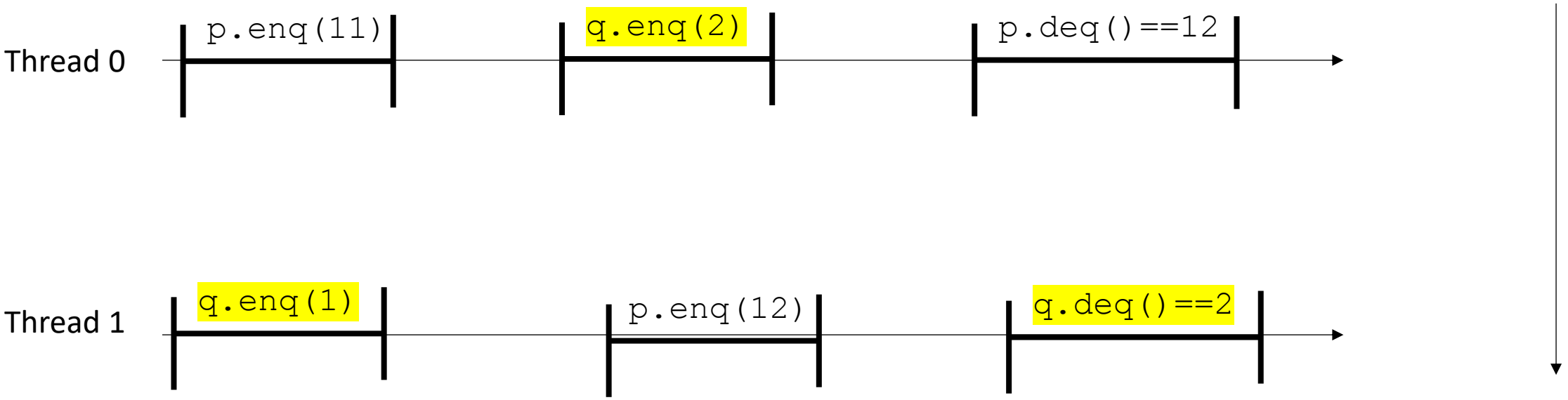
2 objects now: p and q
Consider each object in isolation



Sequential consistency and real time

- Add in real time:

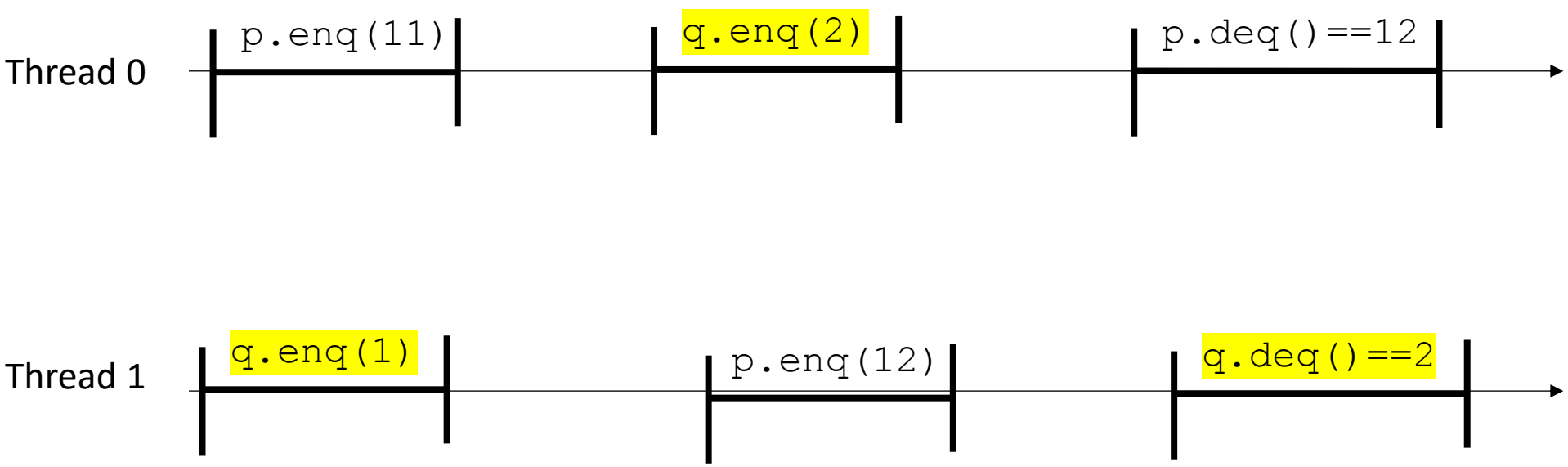
2 objects now: p and q
Consider each object in isolation

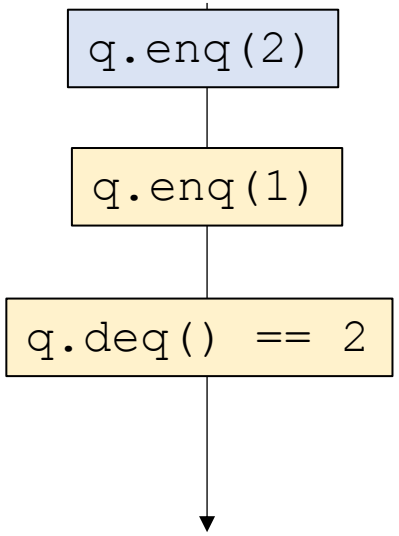


Sequential consistency and real time

- Add in real time:

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

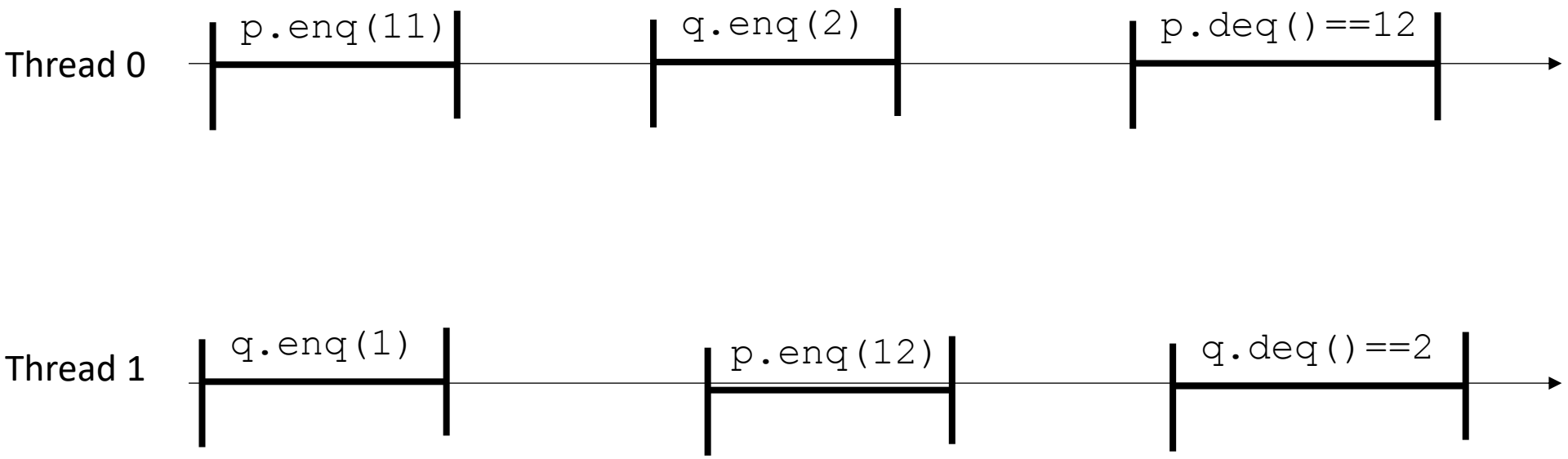




Sequential consistency and real time

- Add in real time:

Now consider them all together

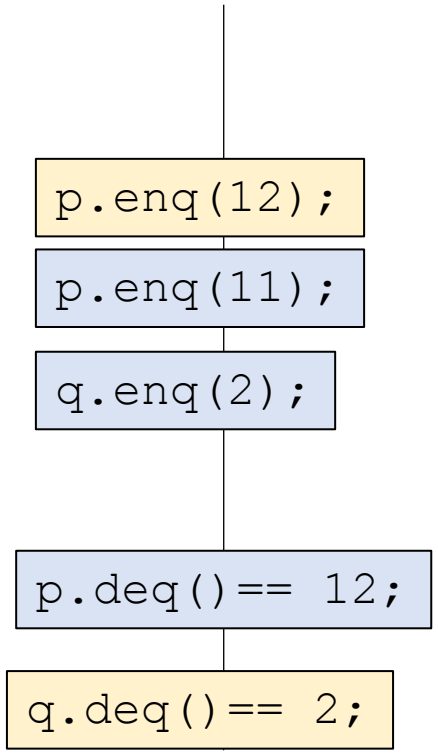


Global variable:

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

Thread 0:

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




```
q.enq(1);
```

Thread 1:

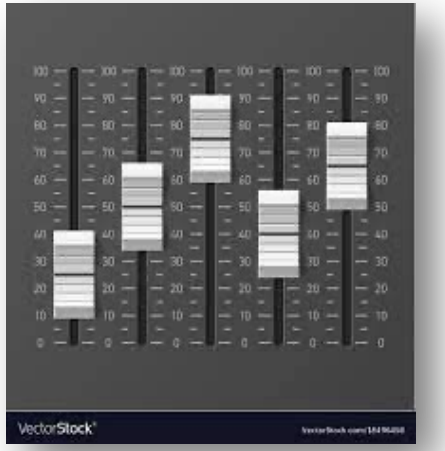
```
q.enq(1)
p.enq(12)
q.deq()==2
```

On to new stuff!

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

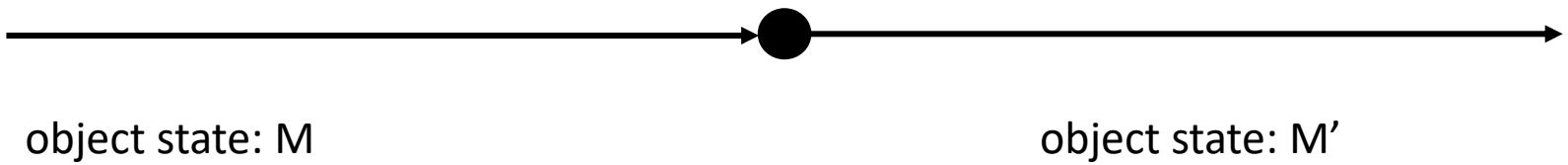
each operation has a linearizability point

- 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

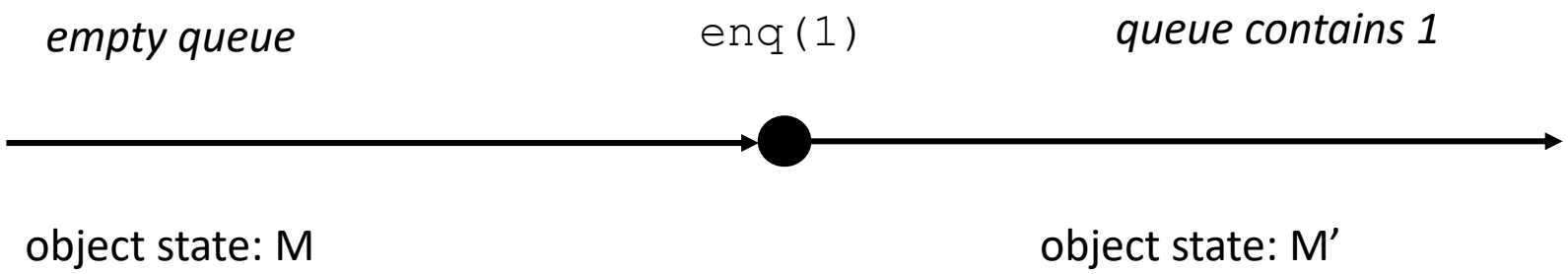
- does not overlap with other with other linearizability points
- indivisible computation (critical section, atomic RMW, atomic load, atomic store)
- object update (or read) occurs exactly at this point



Linearizability

each operation has a linearizability point

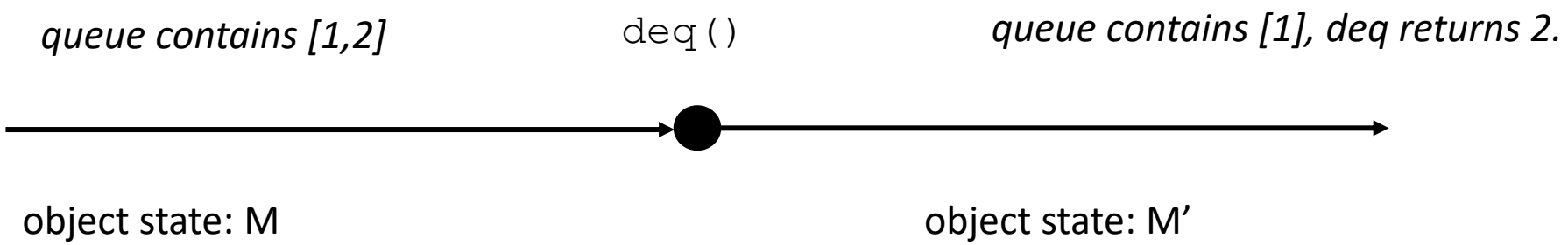
- does not overlap with other with other linearizability points
- indivisible computation (critical section, atomic RMW, atomic load, atomic store)
- object update (or read) occurs exactly at this point



Linearizability

each operation has a linearizability point

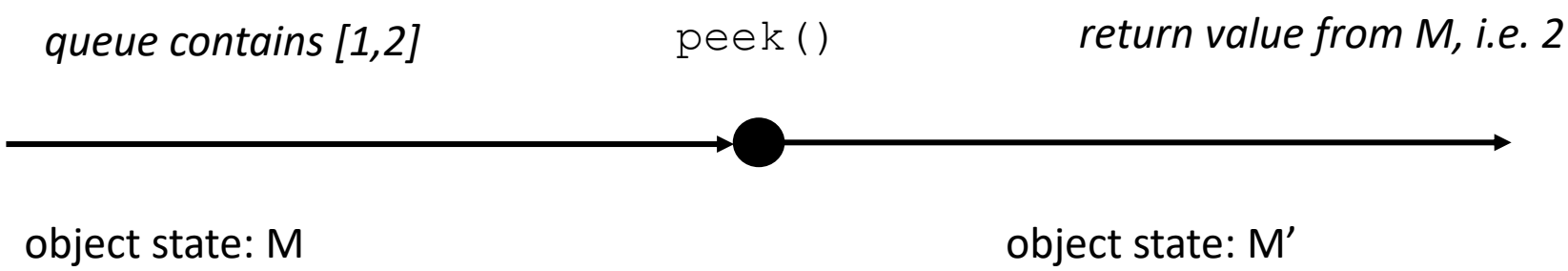
- does not overlap with other with other linearizability points
- indivisible computation (critical section, atomic RMW, atomic load, atomic store)
- object update (or read) occurs exactly at this point



Linearizability

each operation has a linearizability point

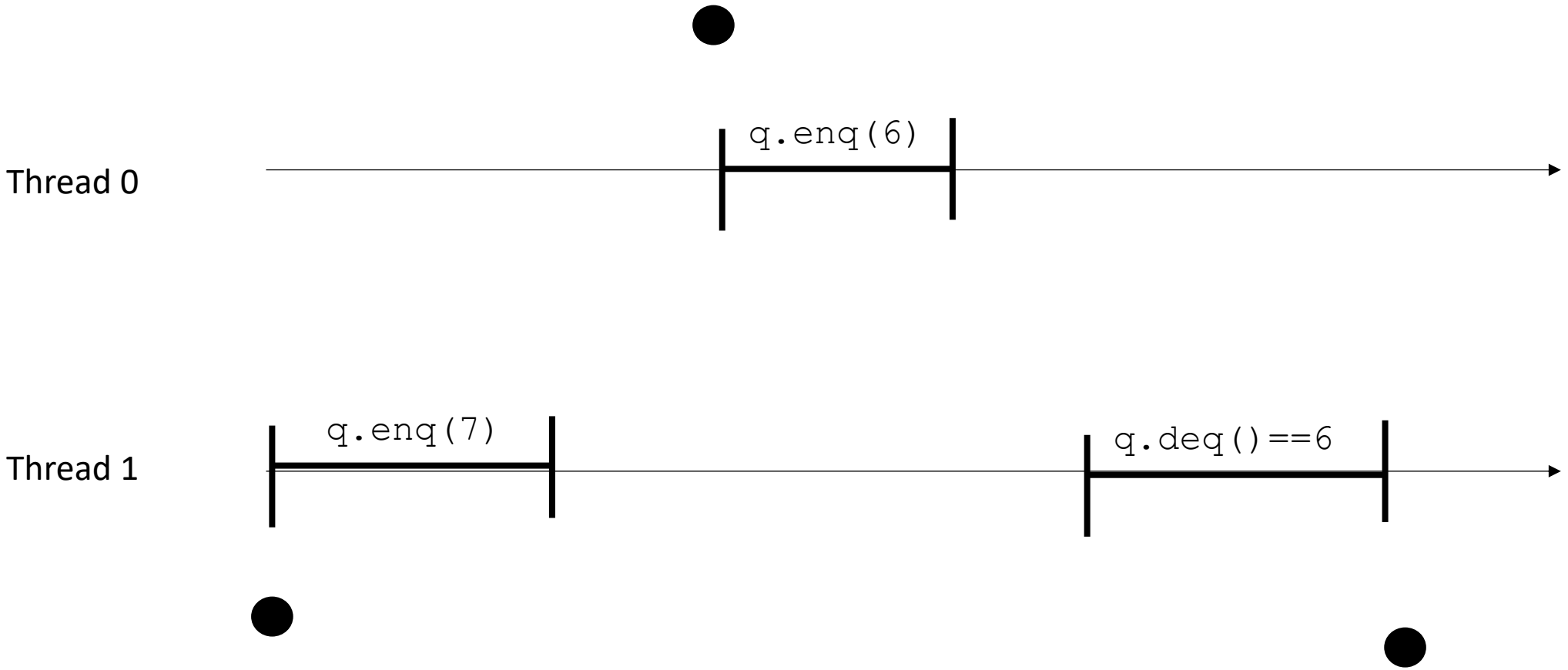
- does not overlap with other with other linearizability points
- indivisible computation (critical section, atomic RMW, atomic load, atomic store)
- object update (or read) occurs exactly at this point



Linearizability

each command gets a linearization point.

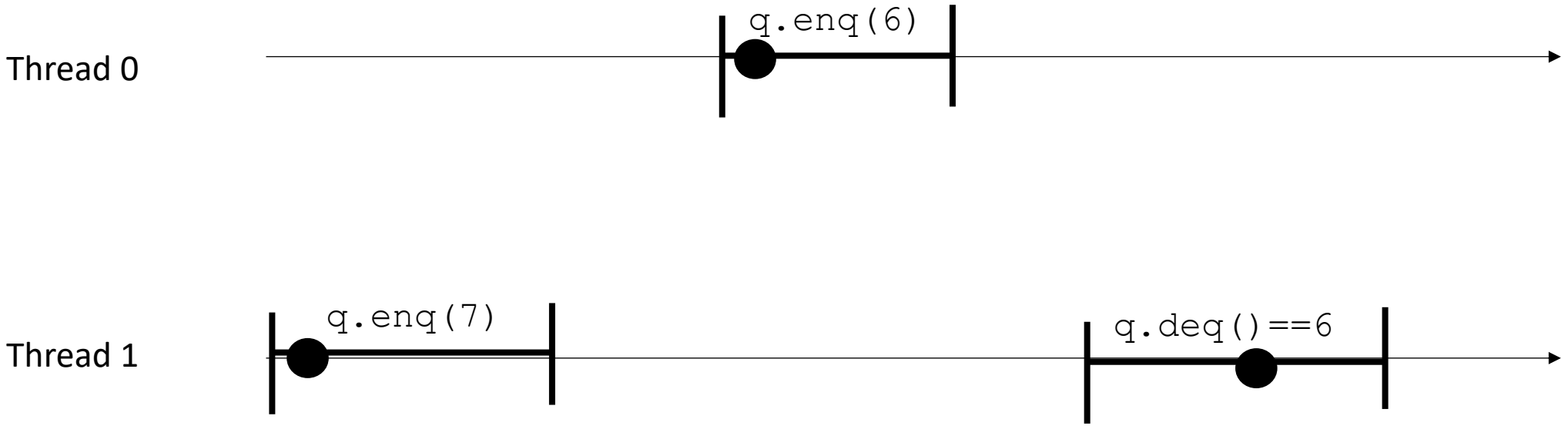
You can place the point any where between its innovation and response!



Linearizability

each command gets a linearization point.

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

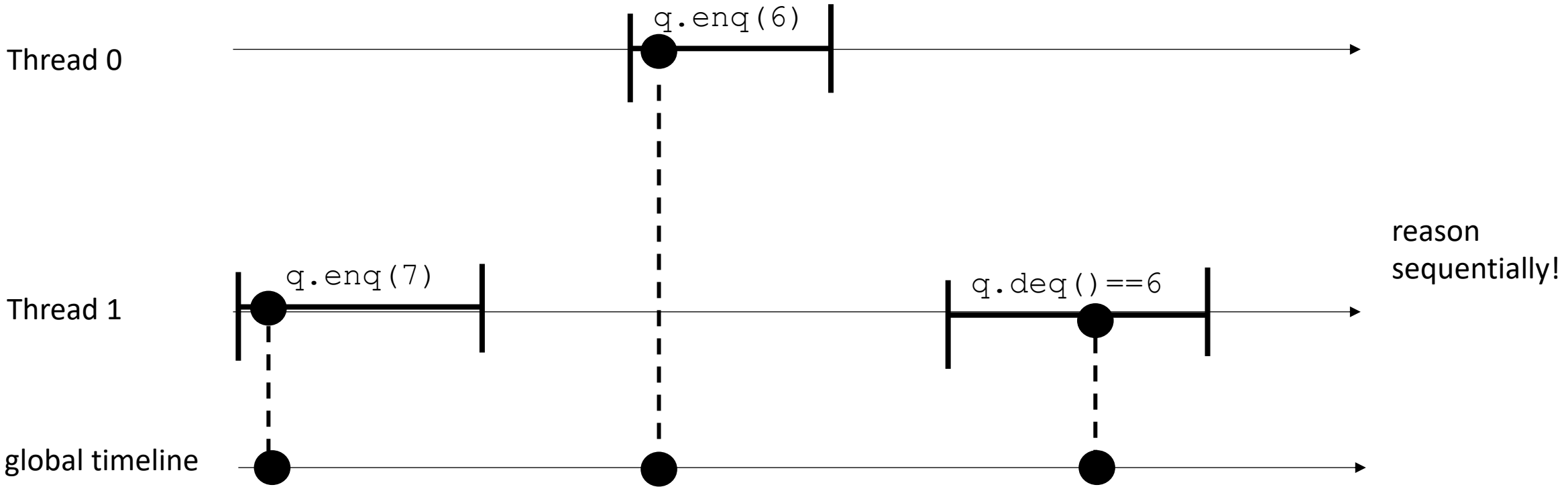


Linearizability

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



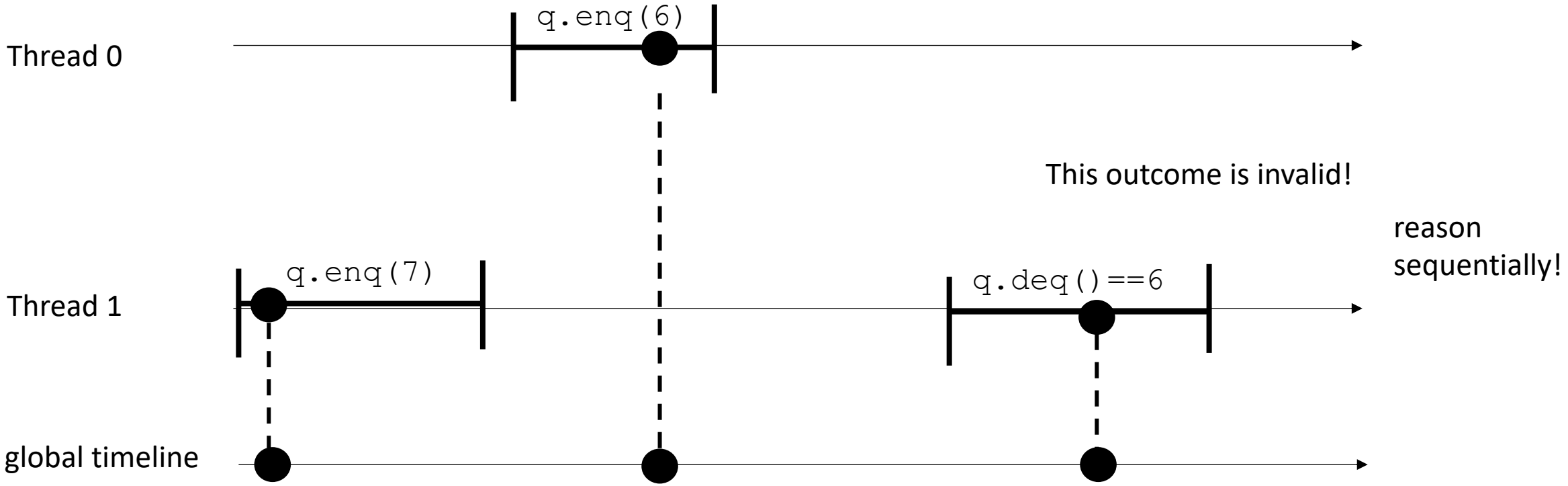
reason sequentially!

Linearizability

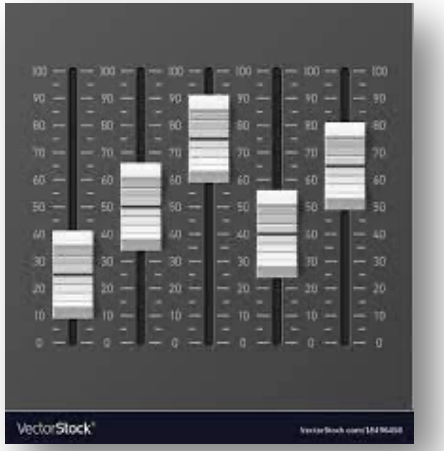
each command gets a linearization point.

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

Project the linearization points to a global timeline



Linearizability

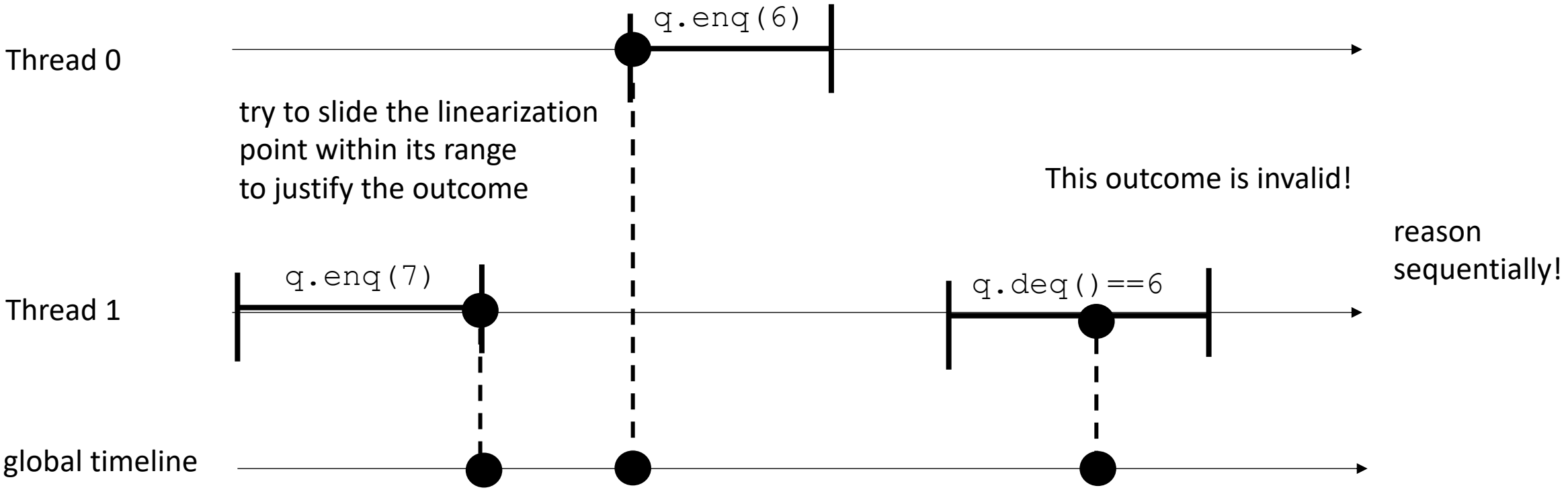


each command gets a linearization point.

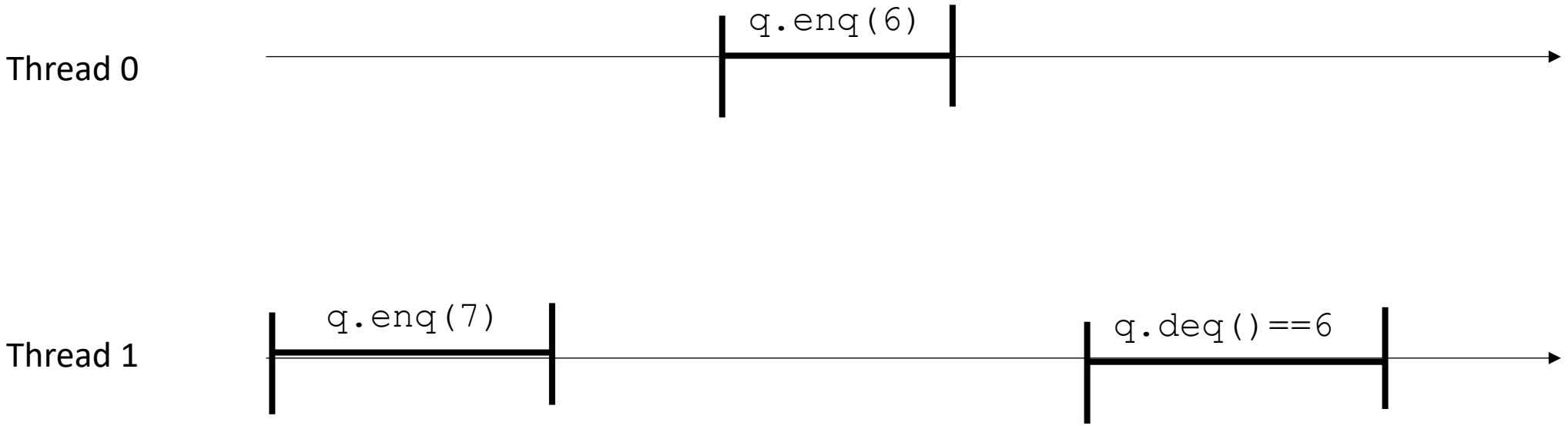
You can place the point any where between its innovation and response!

Project the linearization points to a global timeline

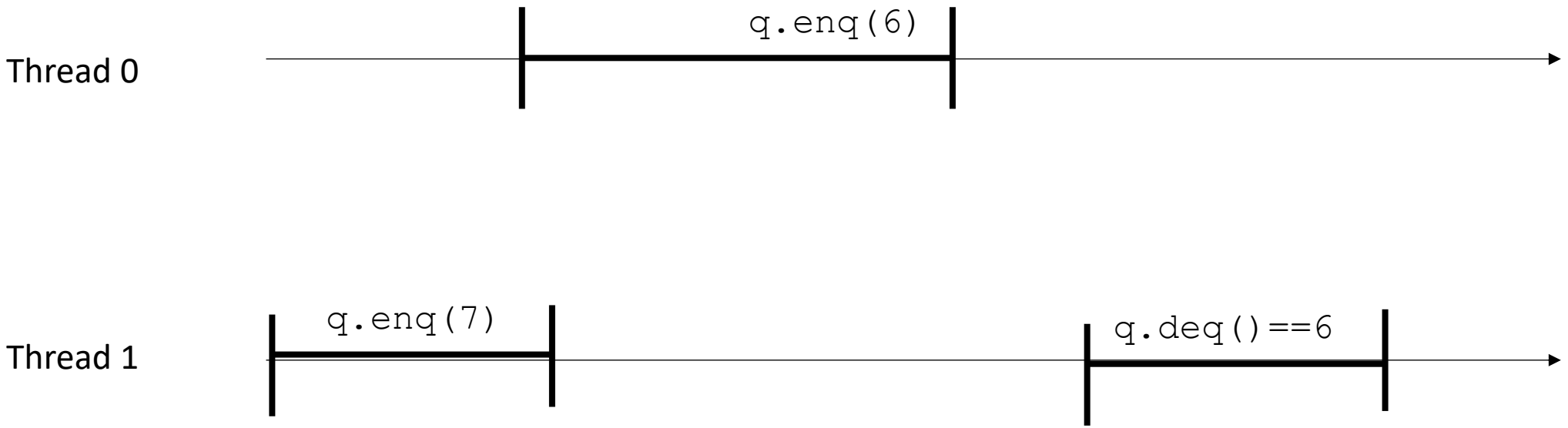
slider game!



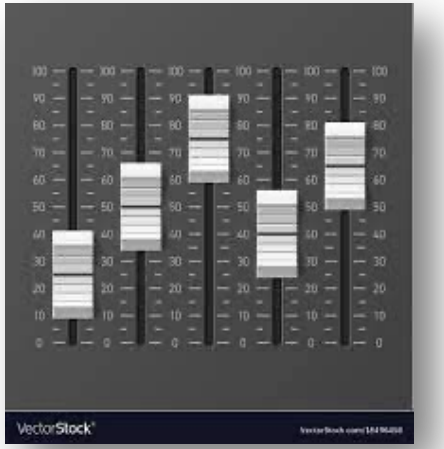
Linearizability

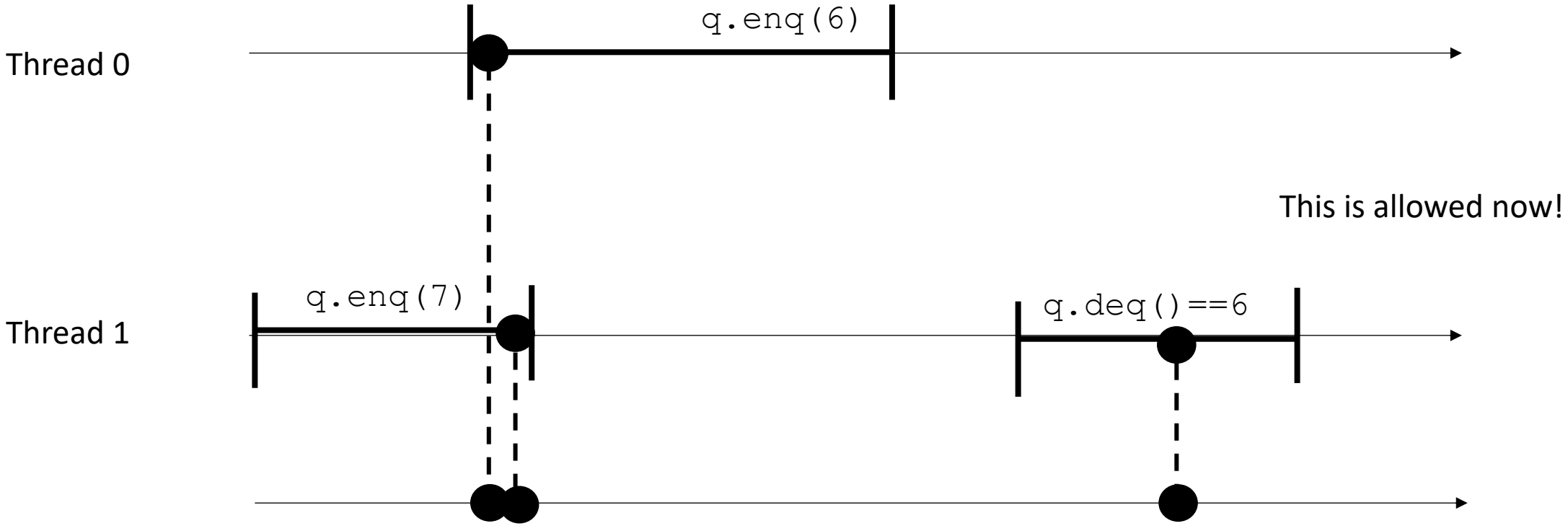


Linearizability

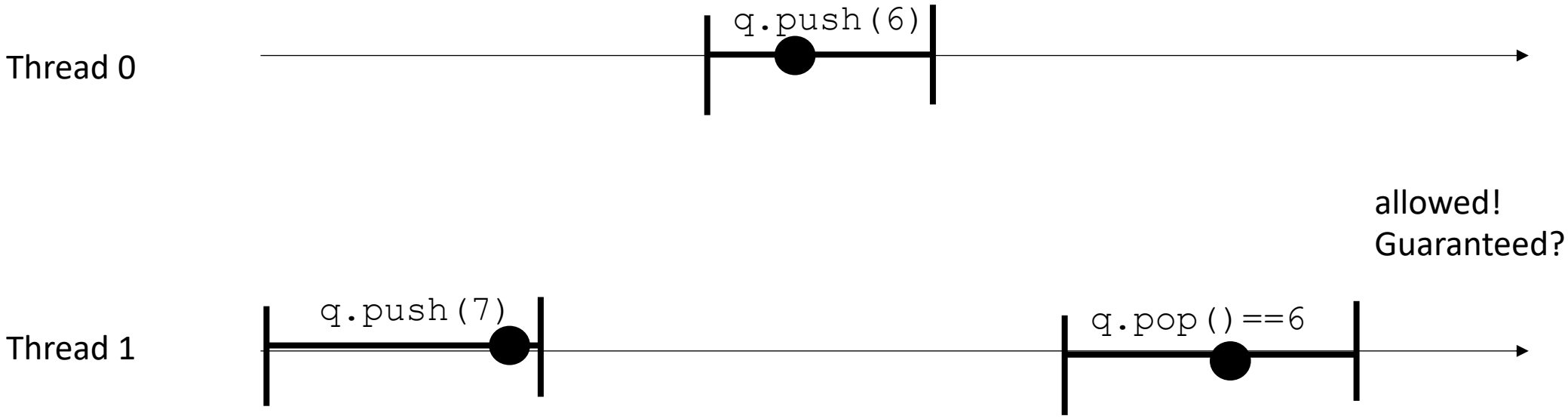


Linearizability

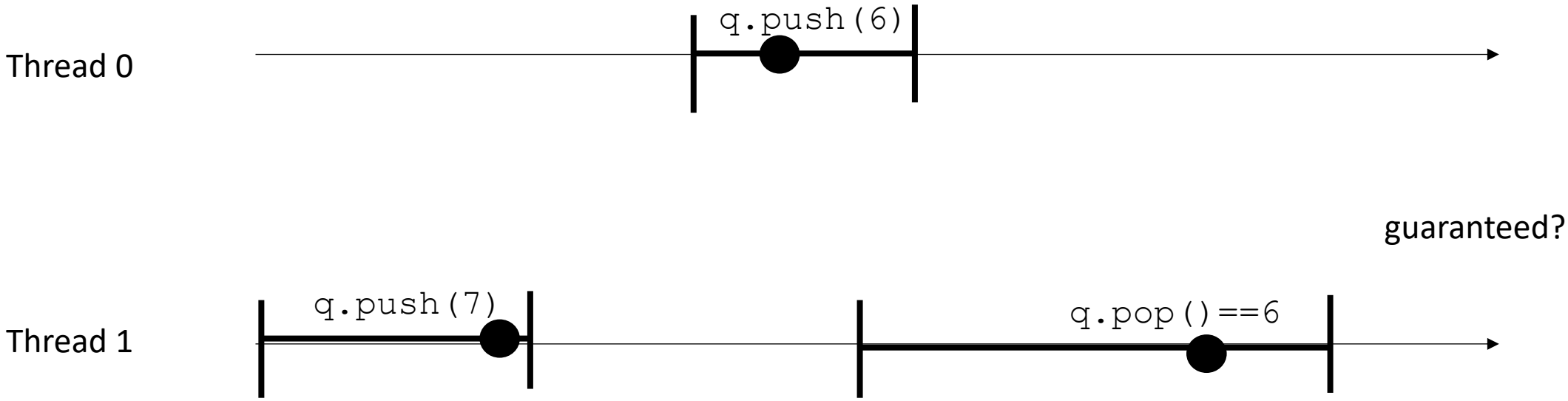




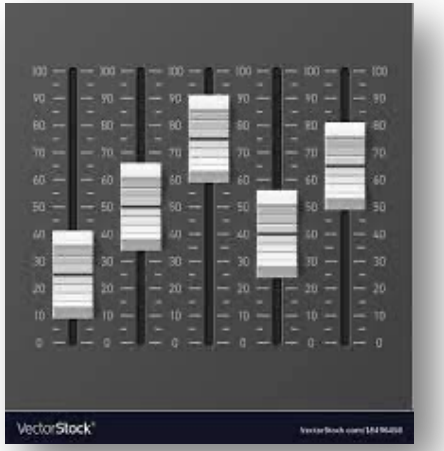
Linearizability

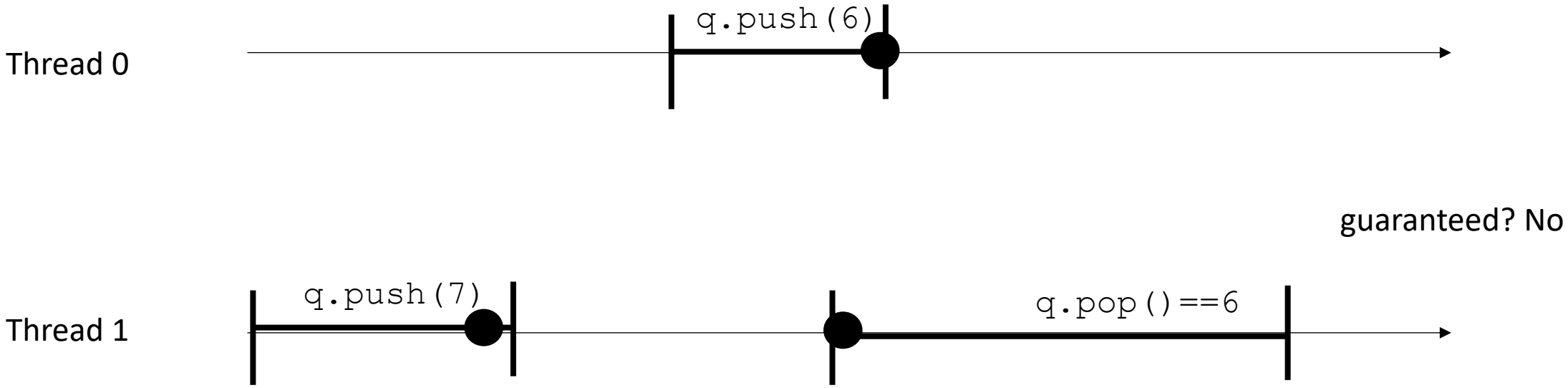


Linearizability



Linearizability





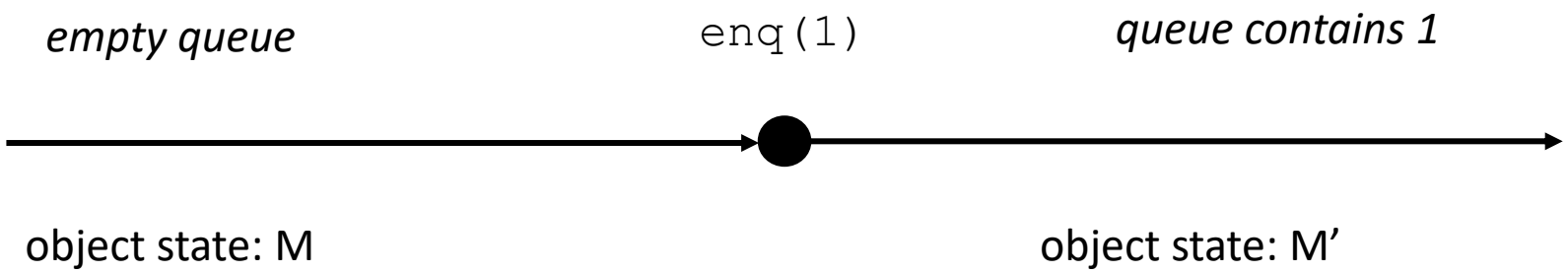
guaranteed? No

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.

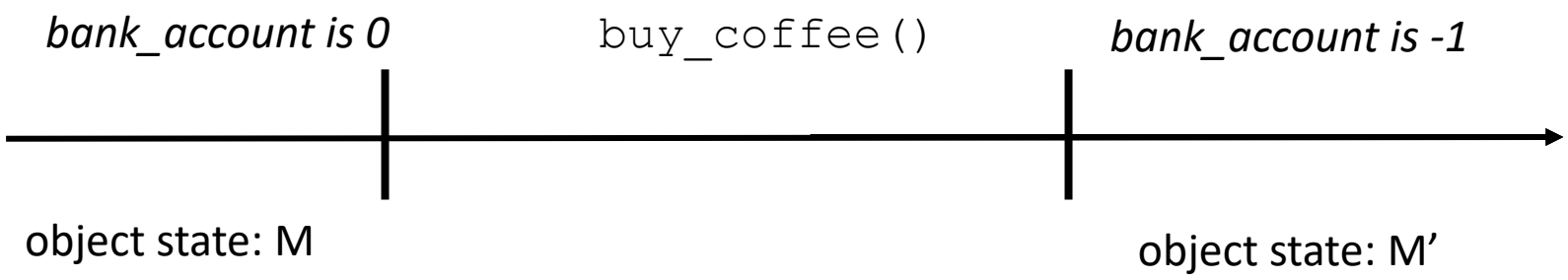
Linearizability

- 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.

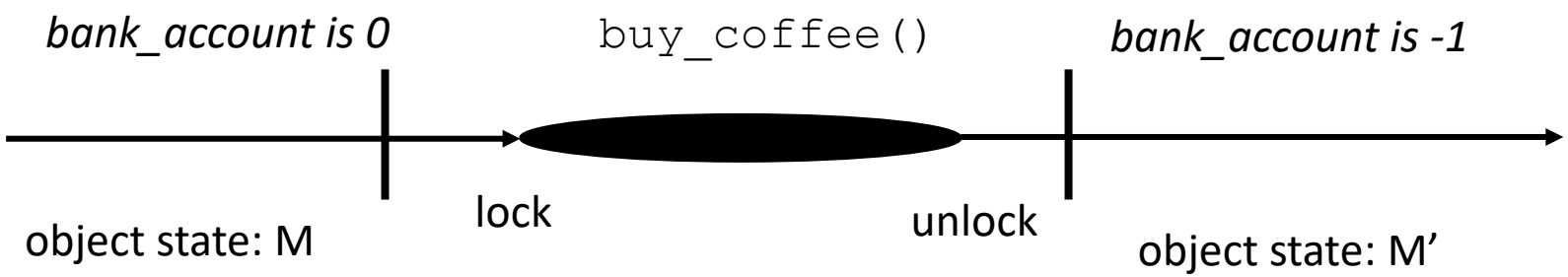



```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    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

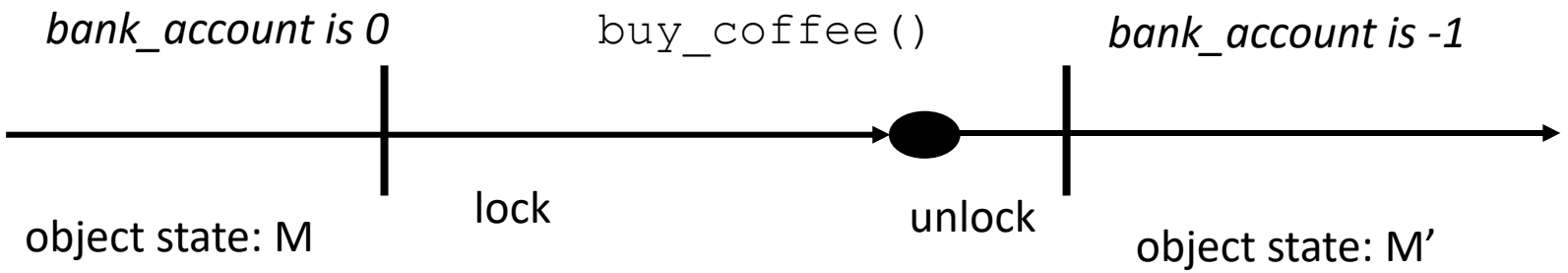



```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    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.

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

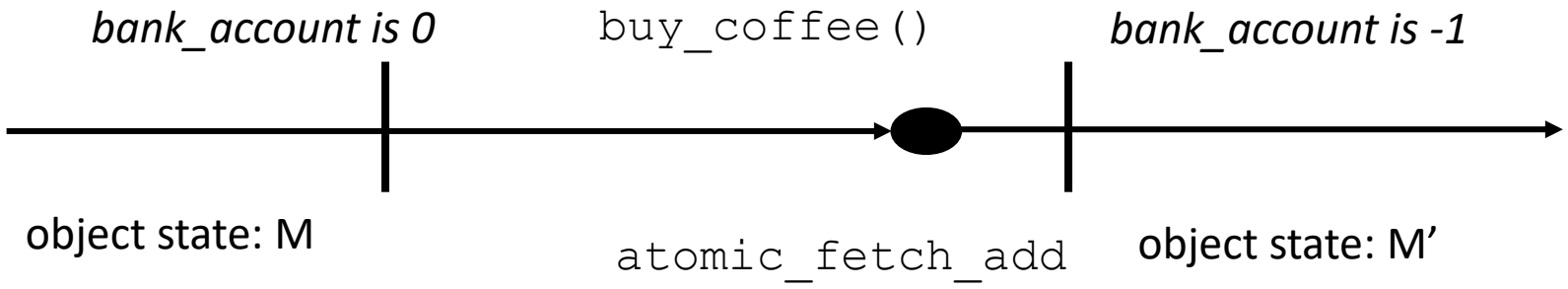



```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

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

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

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

Linearizability

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

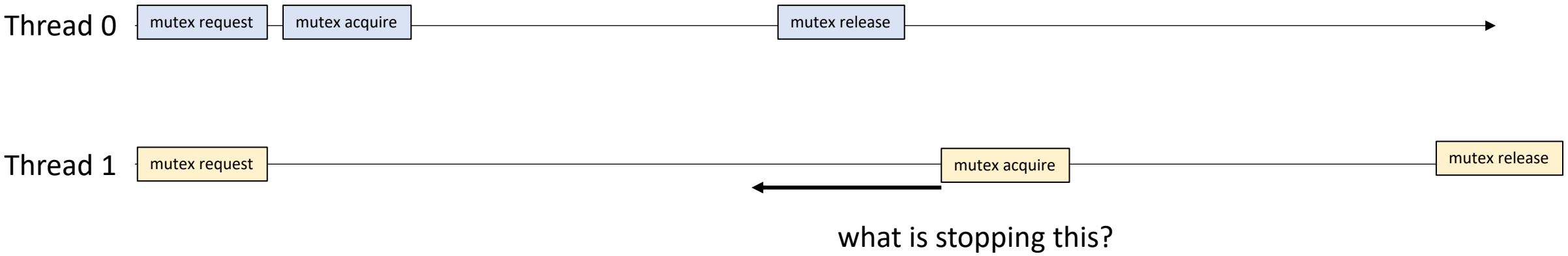
```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    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

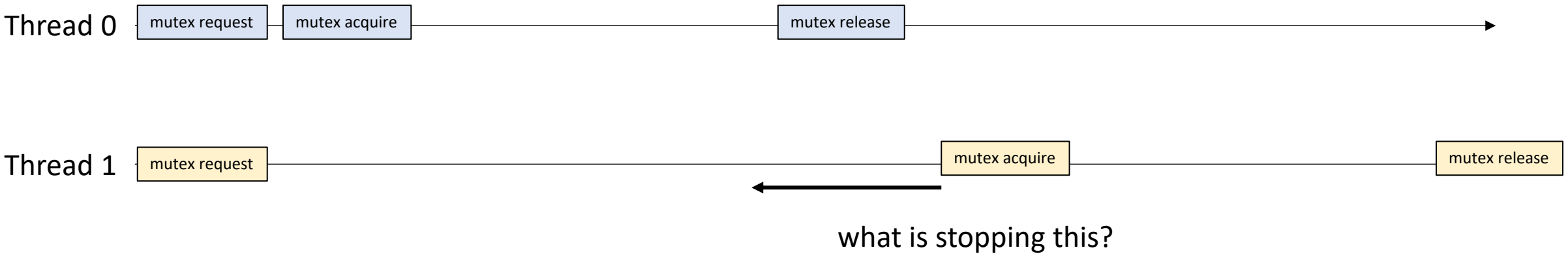


Progress properties

- 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

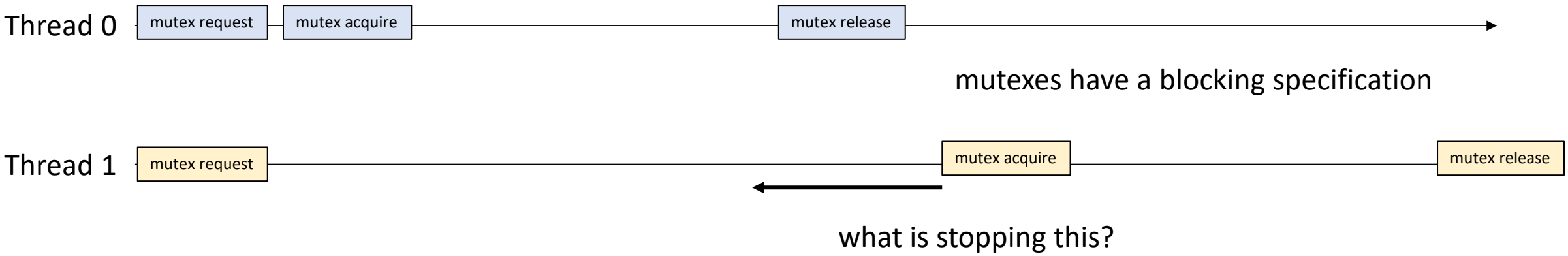


Progress properties

- 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

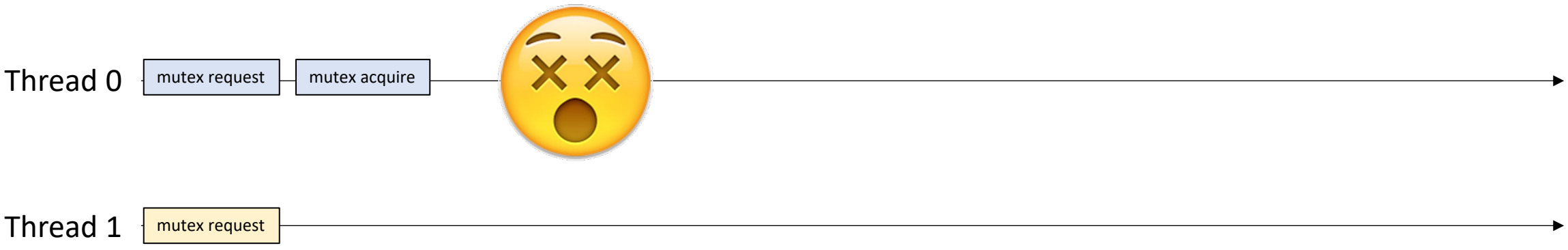


Progress properties

- 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

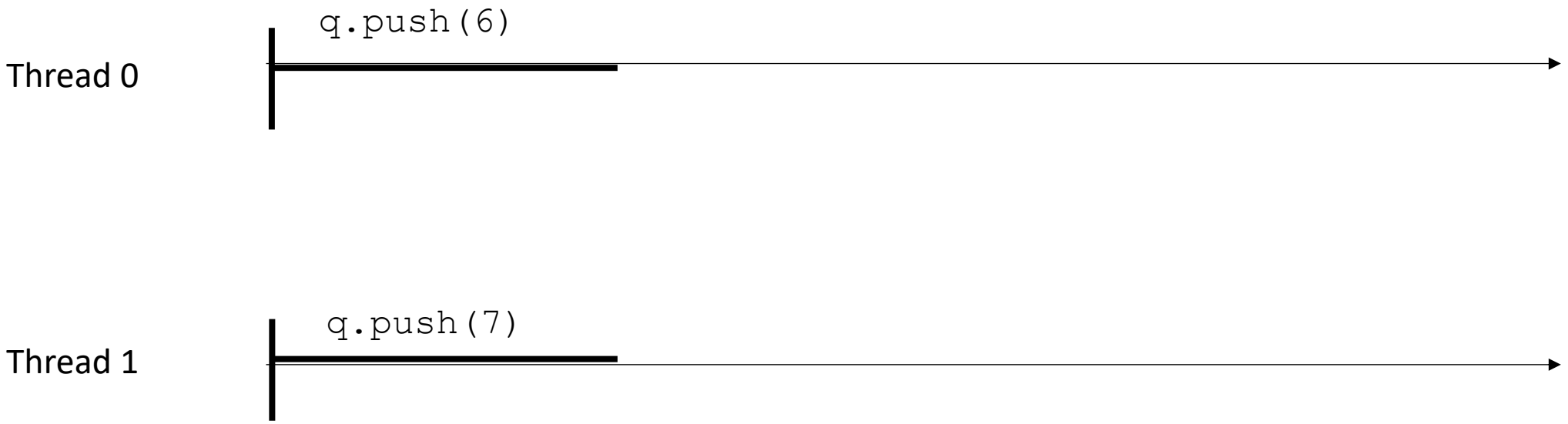
Recall the mutex



What now?!

Linearizability

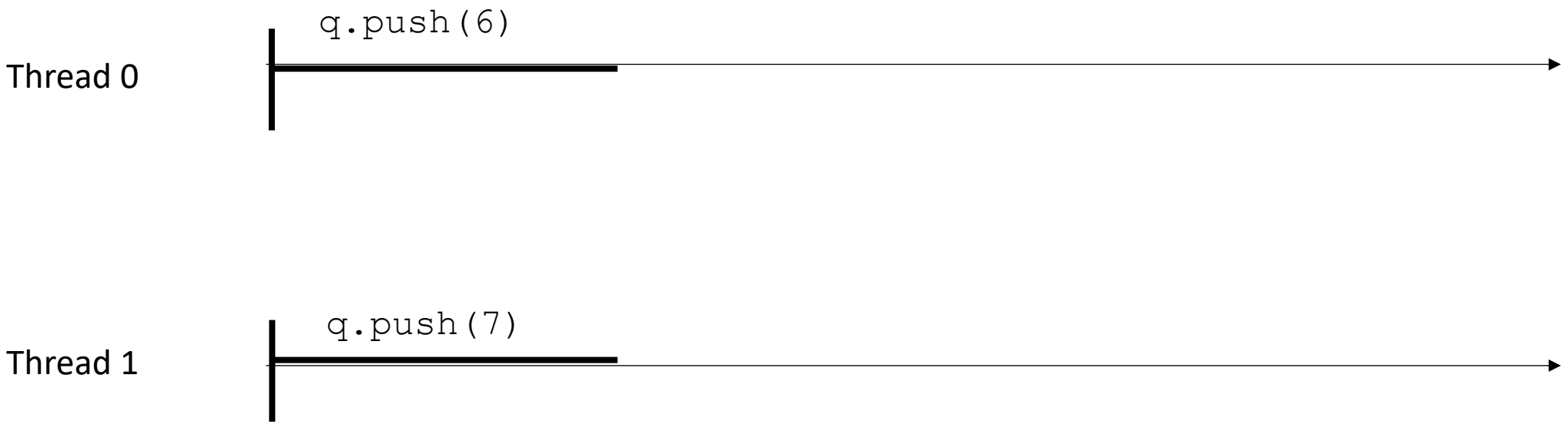
Two unfinished commands.



Linearizability

Two unfinished commands.

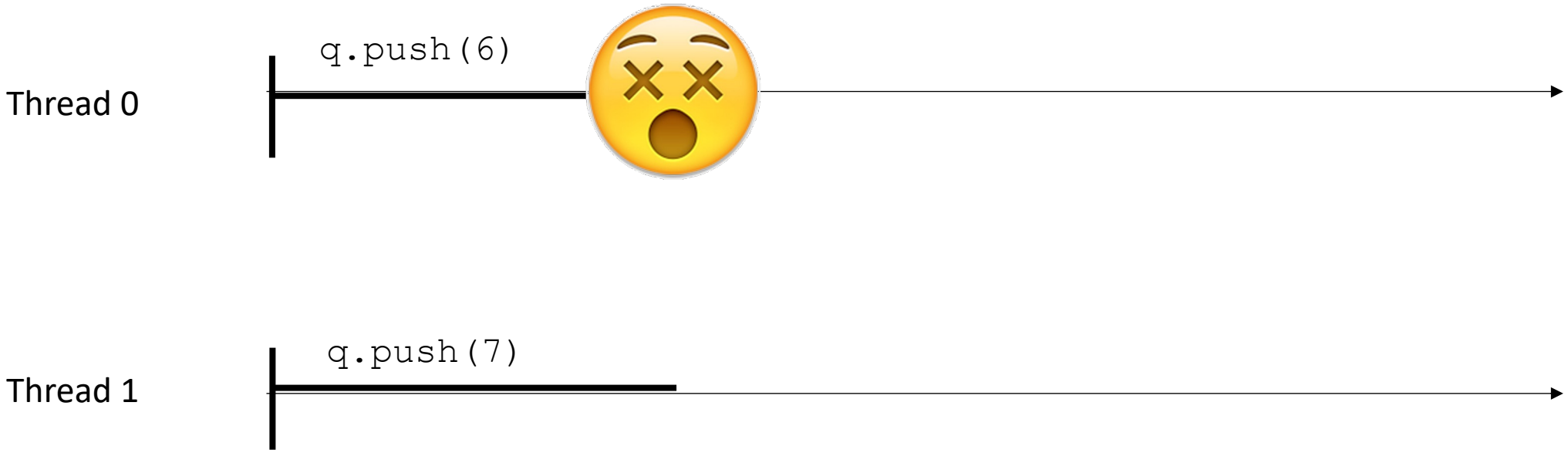
Linearizability does not dictate that one needs to wait for another



Linearizability

Two unfinished commands.

Linearizability does not dictate that one needs to wait for another

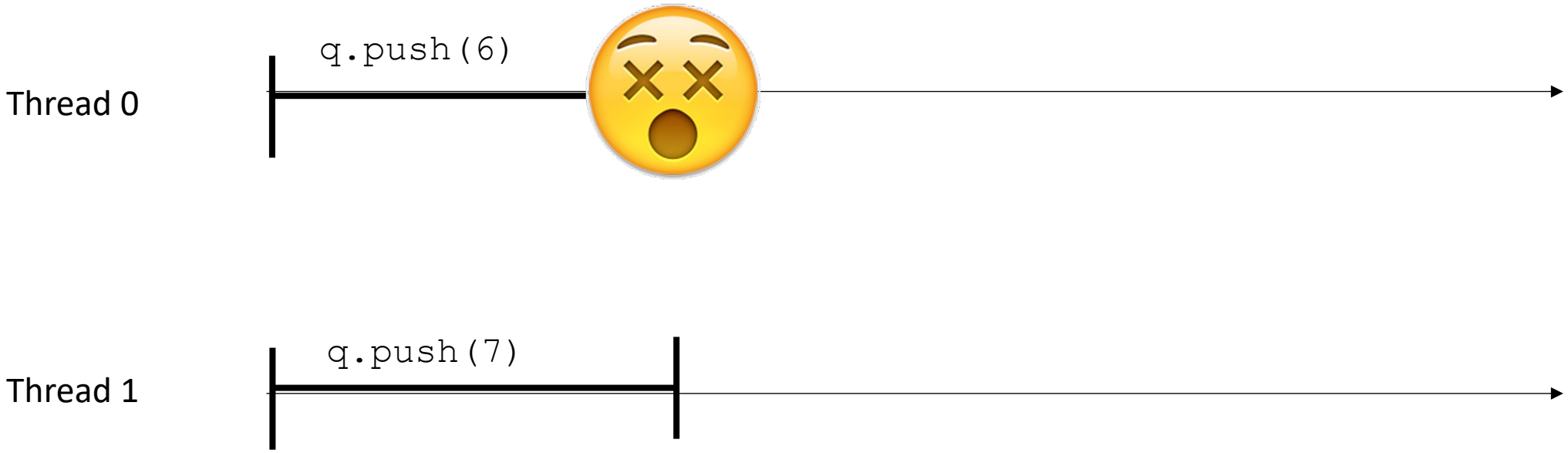


for mutexes, the specification required that the system hang.

Linearizability

Two unfinished commands.

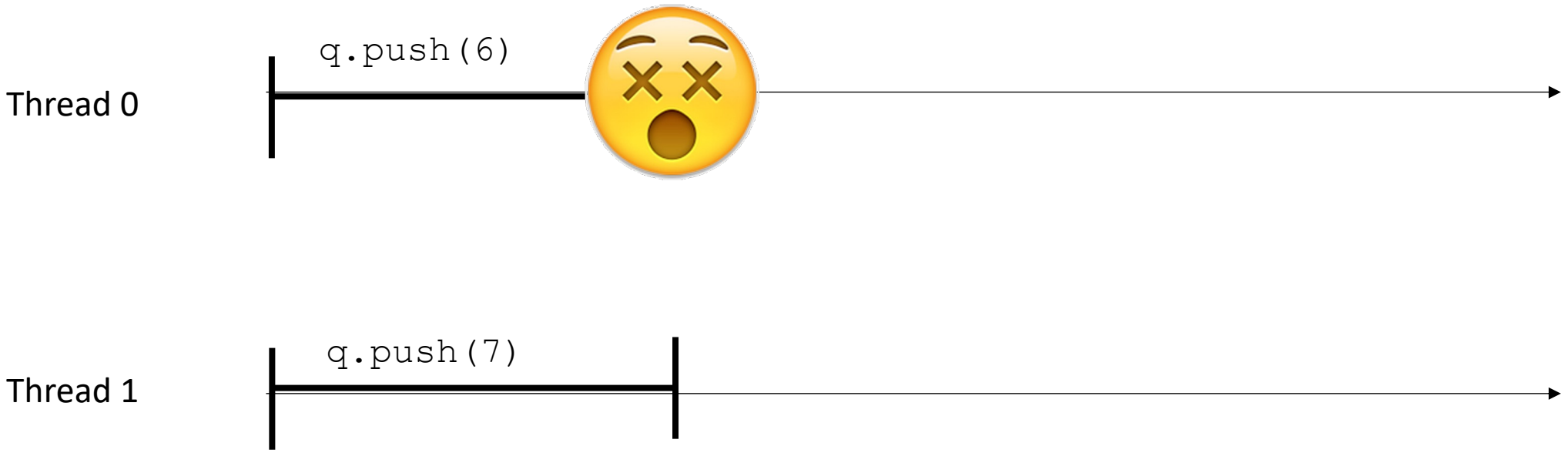
Linearizability does not dictate that one needs to wait for another



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

Linearizability

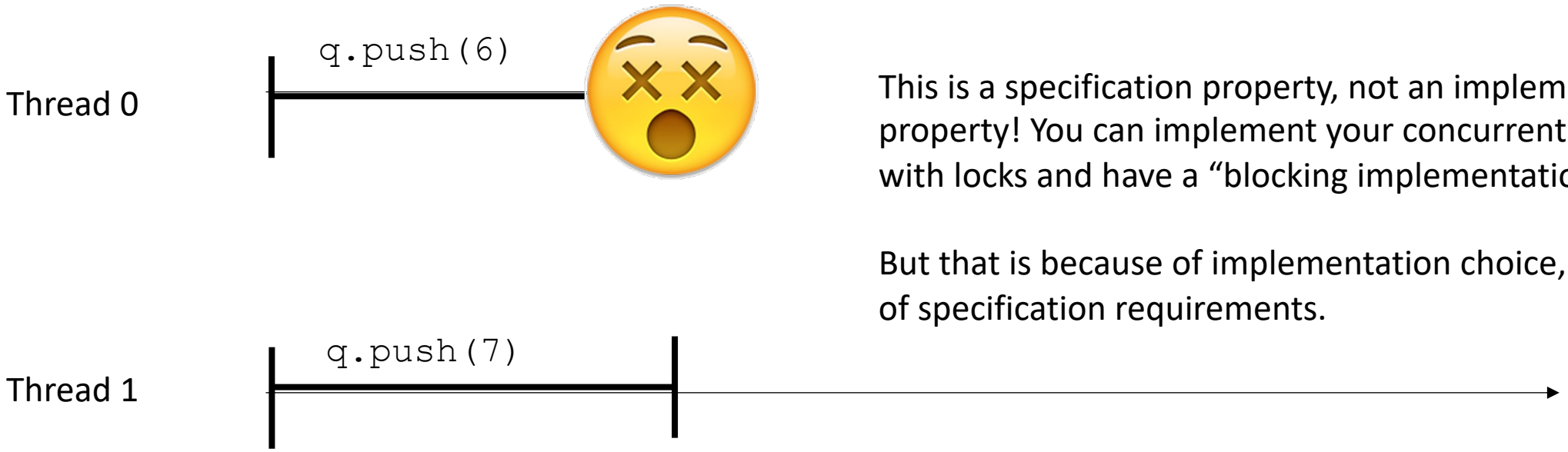
Non-blocking specification:
Every thread is allowed to continue executing
REGARDLESS of the behavior of other threads



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



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 object:
- Lock-free object:
- Blocking specification:
- Non-blocking specification:
- (non-)blocking implementation:

Terminology overview

- Sequential consistency:
- Linearizability:
- Linearizability point:

Starting simple

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!

Input/Output Queues

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

- Why would we want a thing?

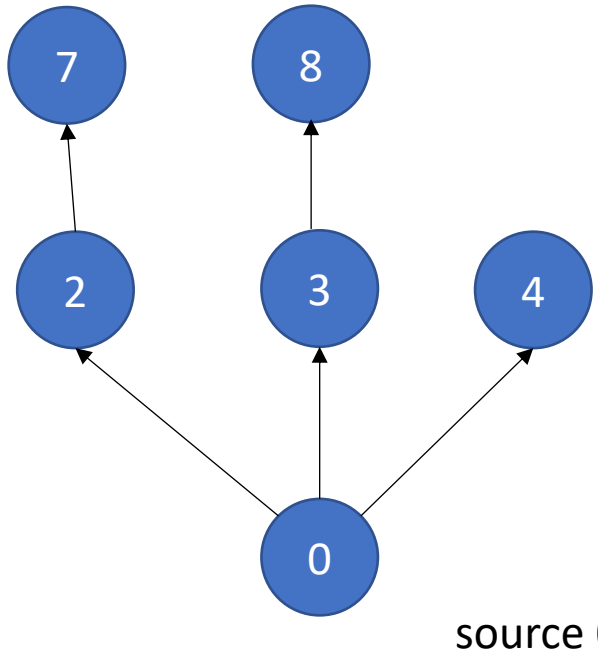
- Computation done in phases:
  - First phase prepares the queue (by writing into it)
  - All threads join
  - Second phase reads values from the queue.

Input/Output Queues

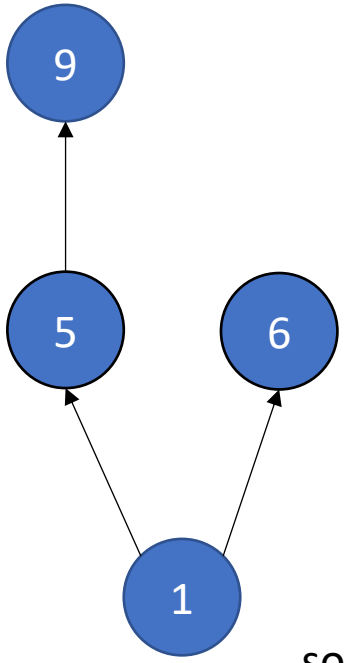
- Example: Information flow in graph applications:

Input/Output Queues

- Example: Information flow in graph applications:



source 0



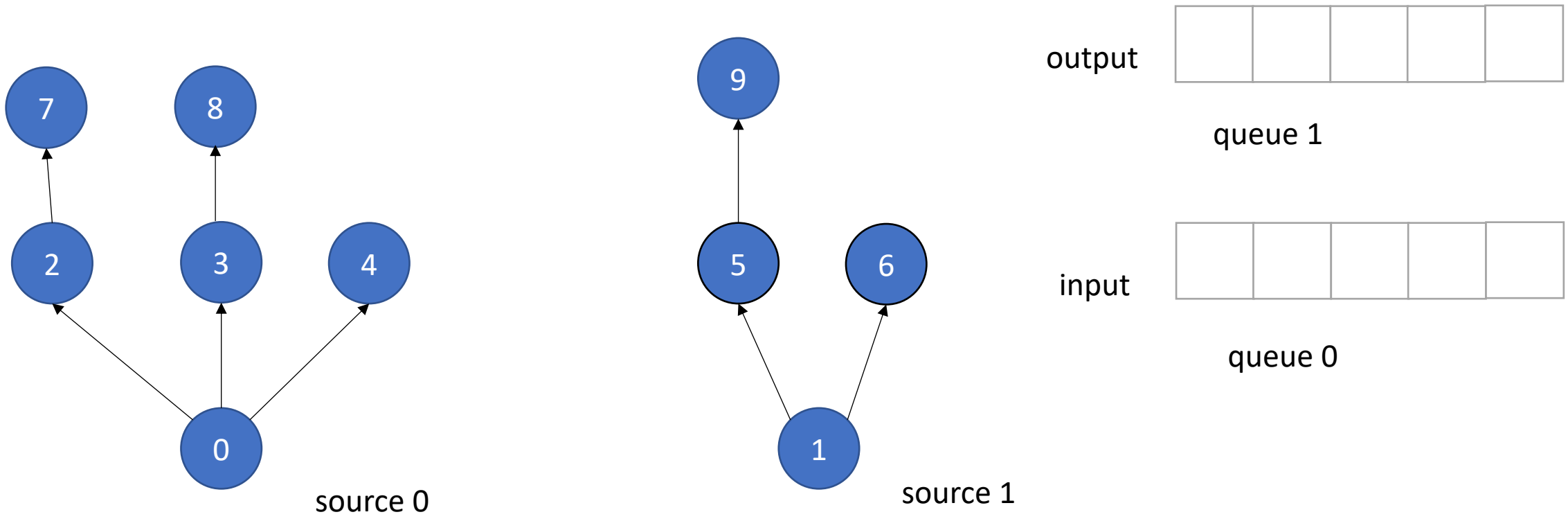
source 1

queue 1

queue 0

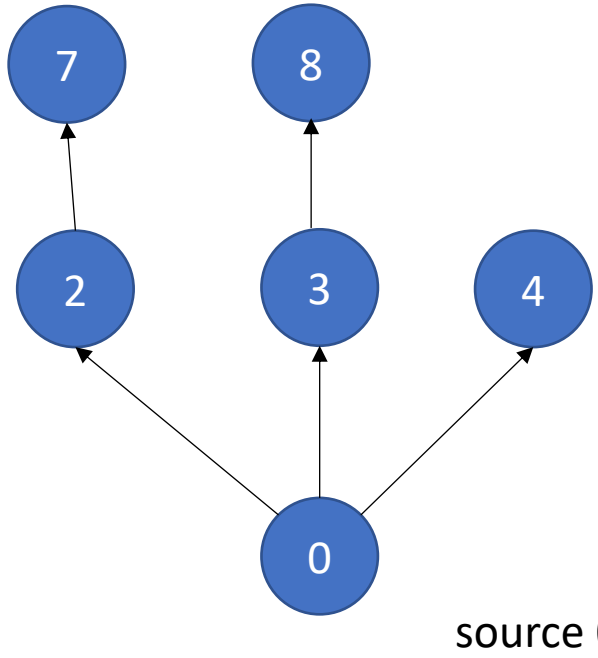
Input/Output Queues

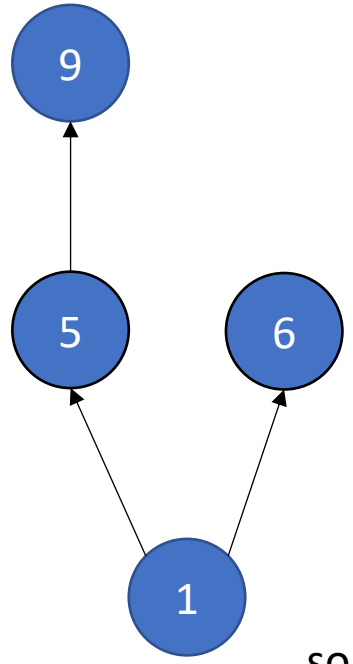
- Example: Information flow in graph applications:

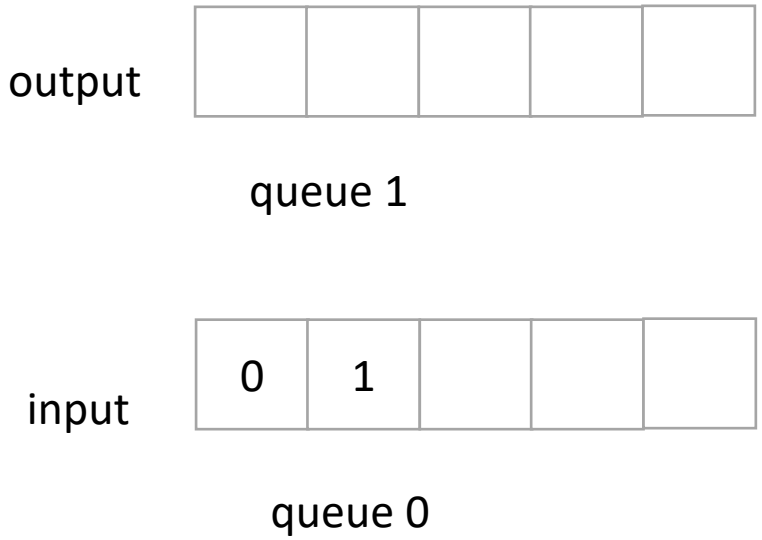


Input/Output Queues

- Example: Information flow in graph applications:

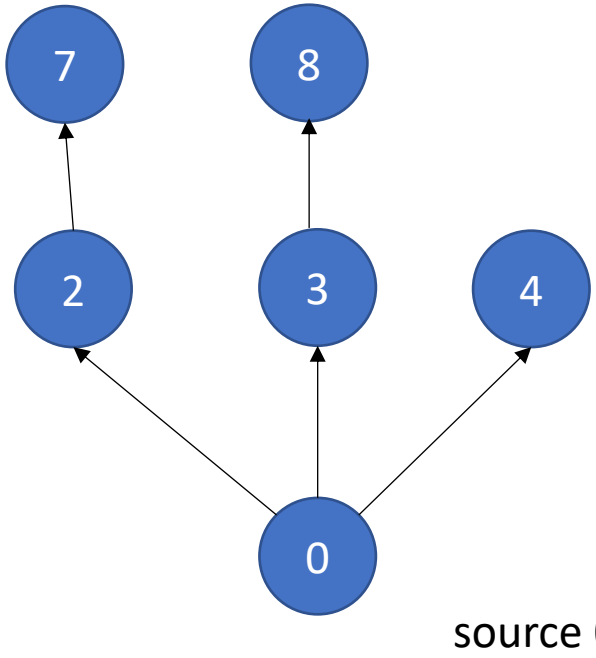


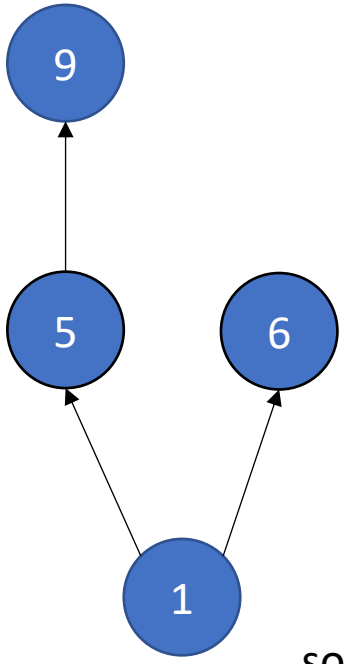


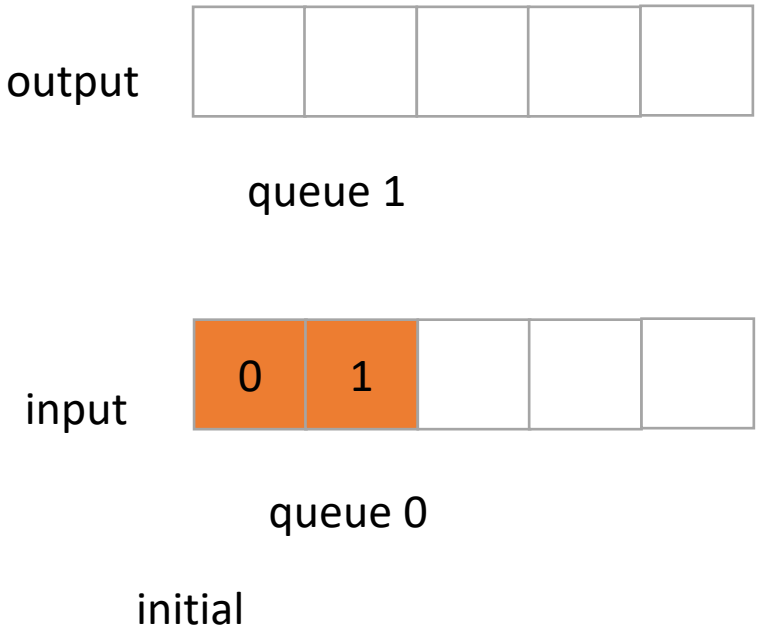


Input/Output Queues

- Example: Information flow in graph applications:

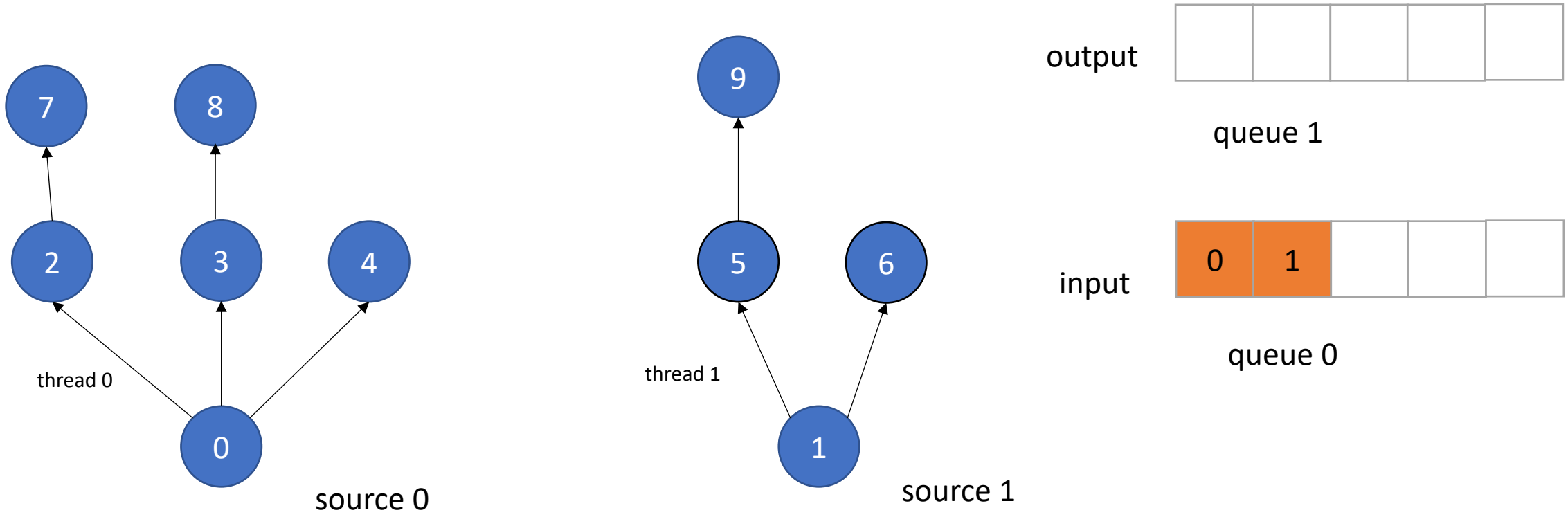






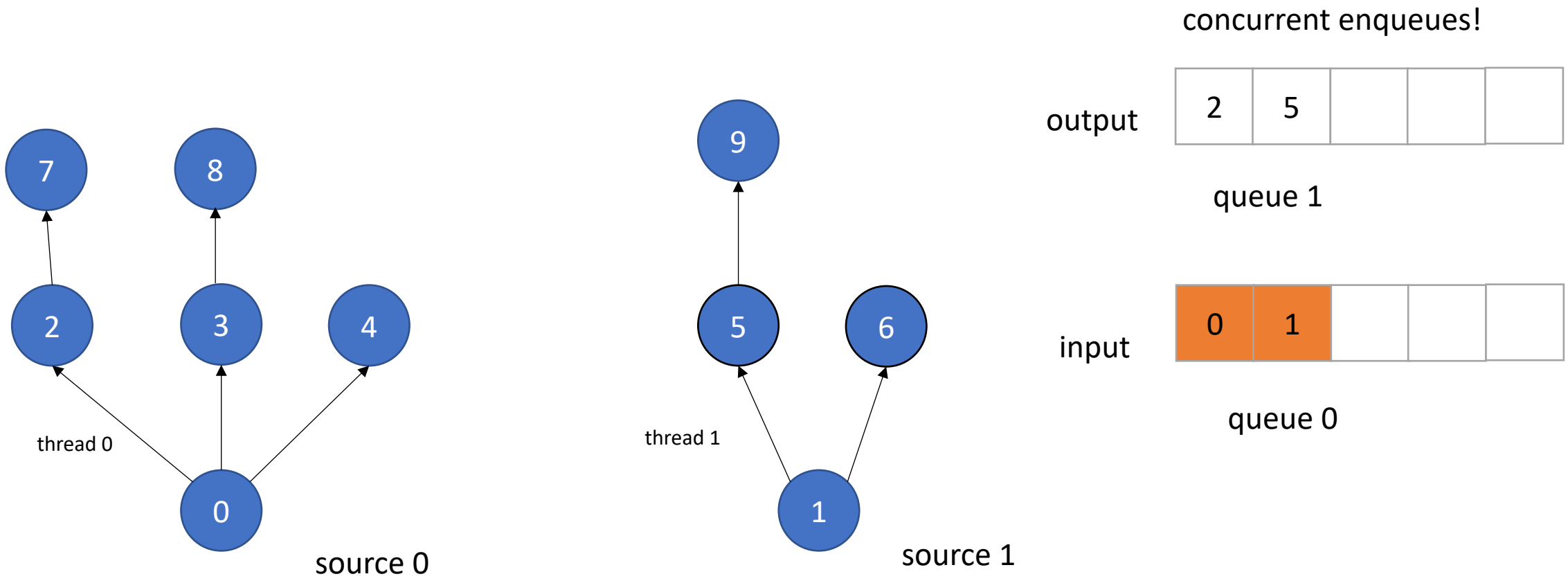
Input/Output Queues

- Example: Information flow in graph applications:



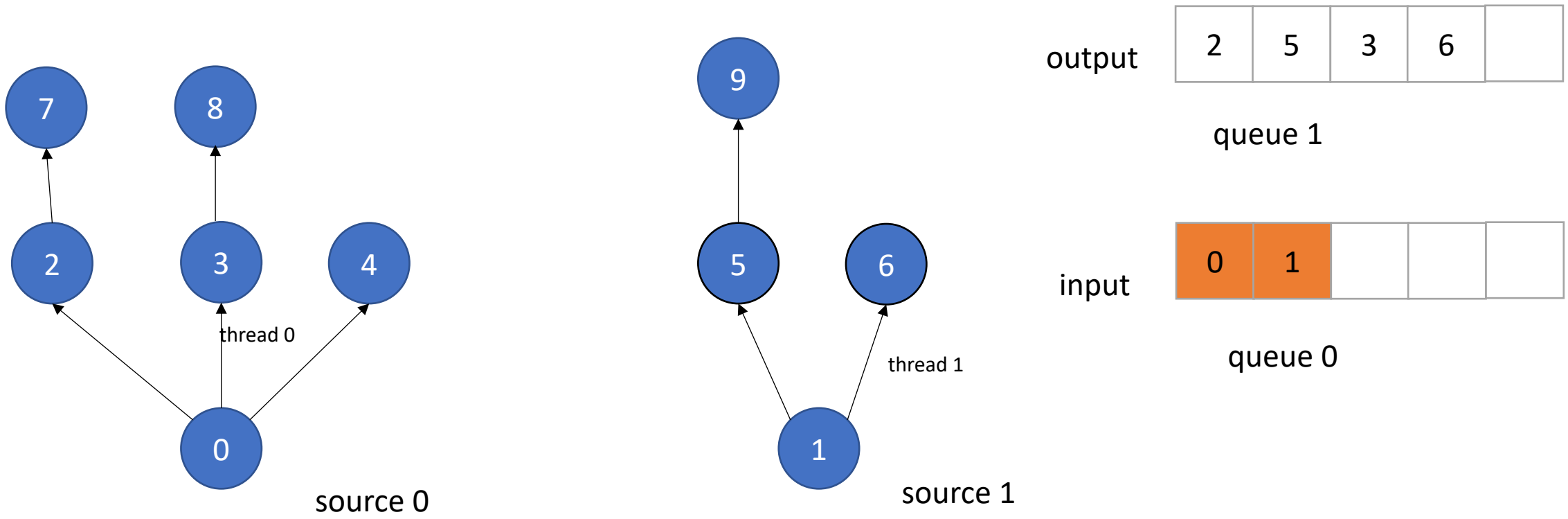
Input/Output Queues

- Example: Information flow in graph applications:



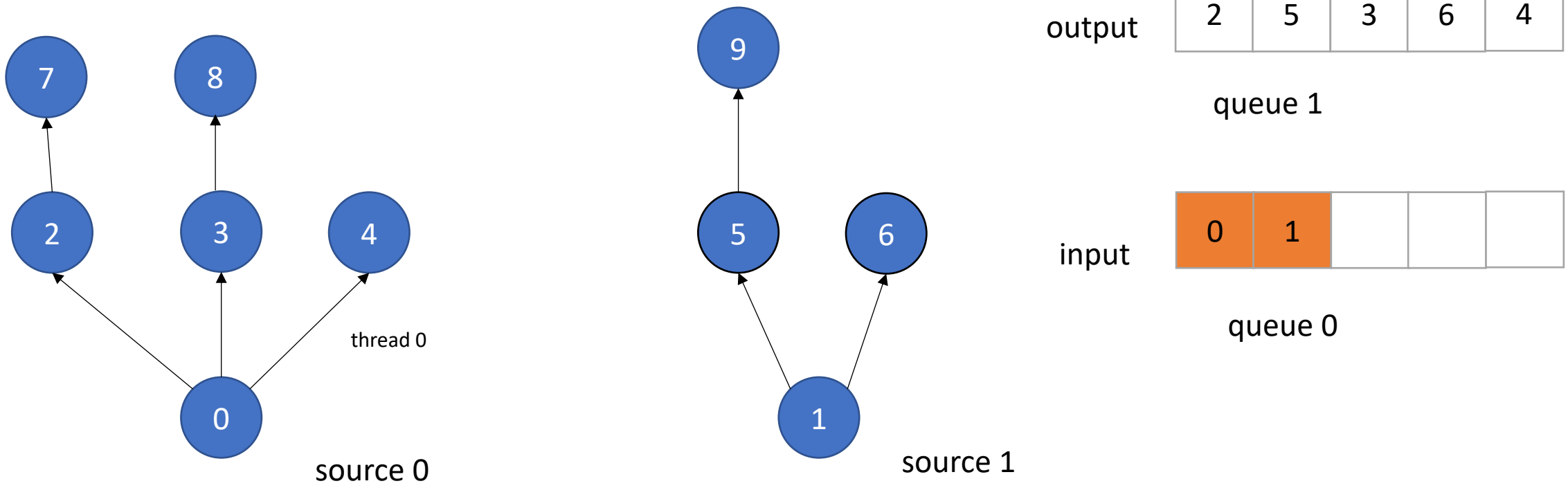
Input/Output Queues

- Example: Information flow in graph applications:



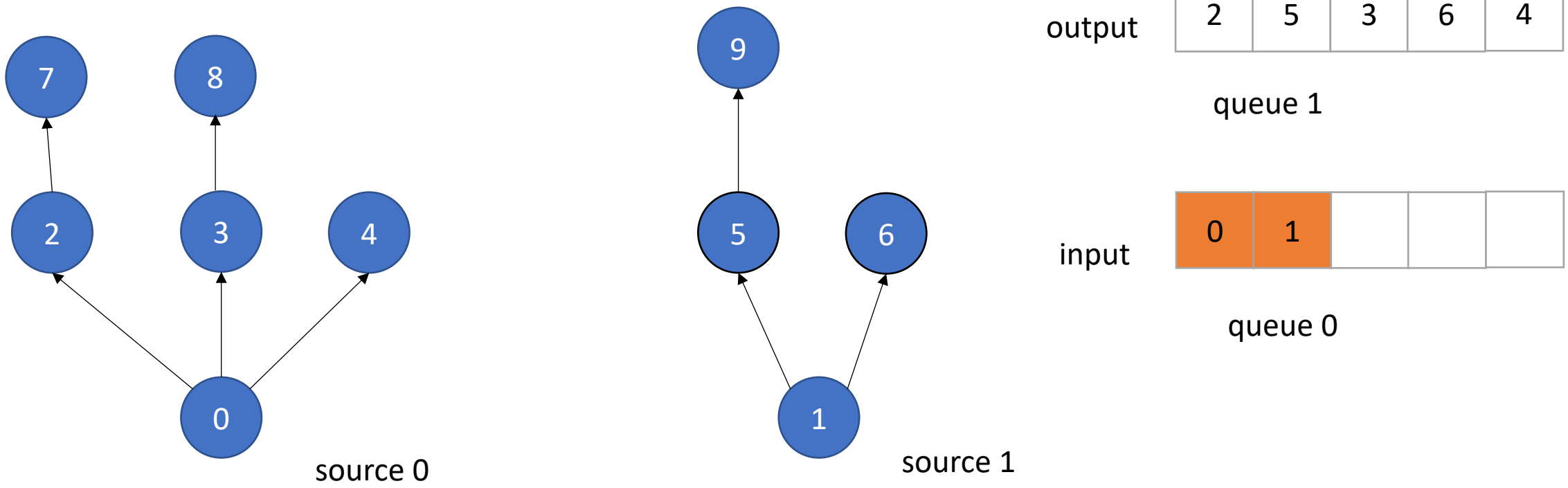
Input/Output Queues

- Example: Information flow in graph applications:



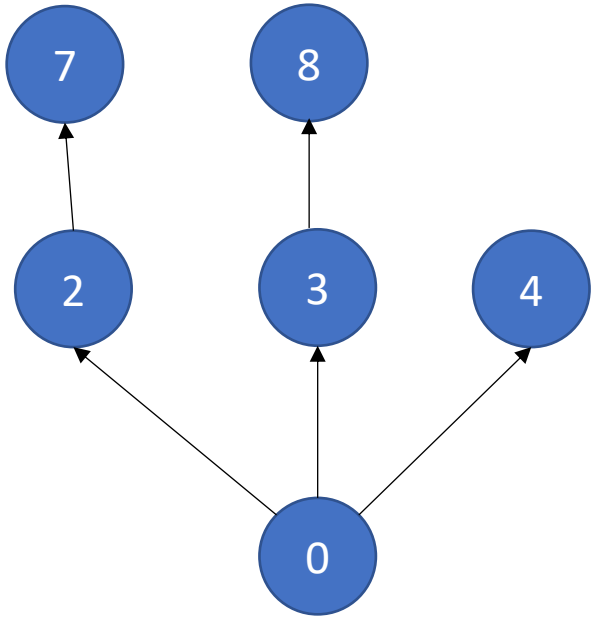
Input/Output Queues

- Example: Information flow in graph applications:

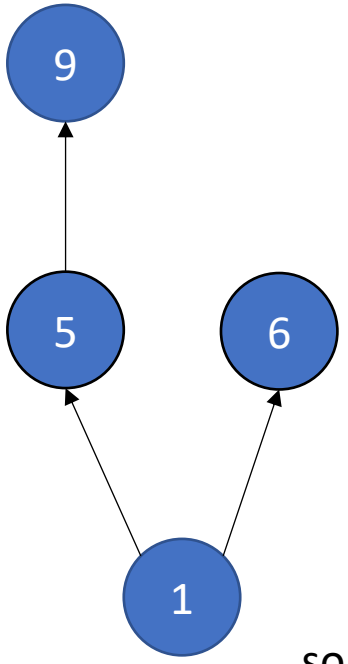


Input/Output Queues

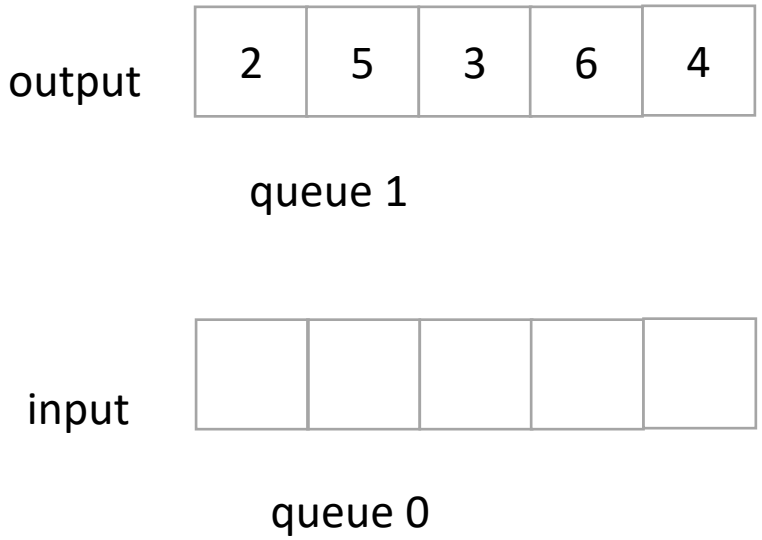
- Example: Information flow in graph applications:



source 0



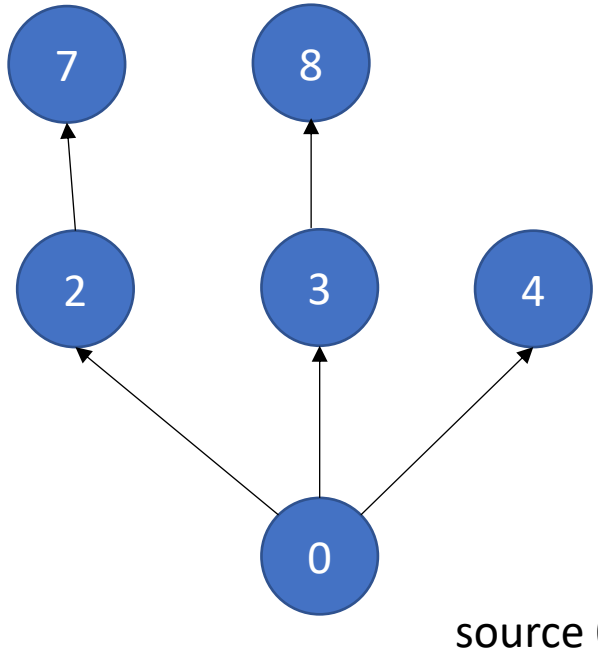
source 1

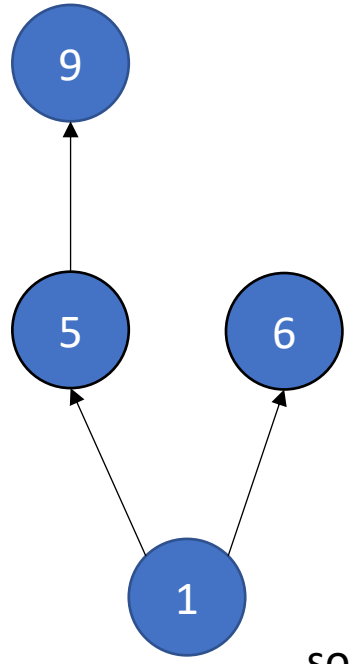


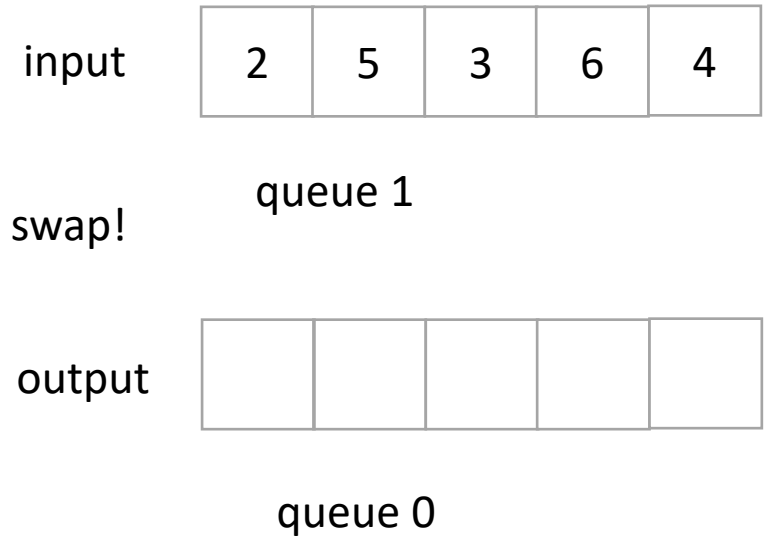
join all threads and clear input

Input/Output Queues

- Example: Information flow in graph applications:

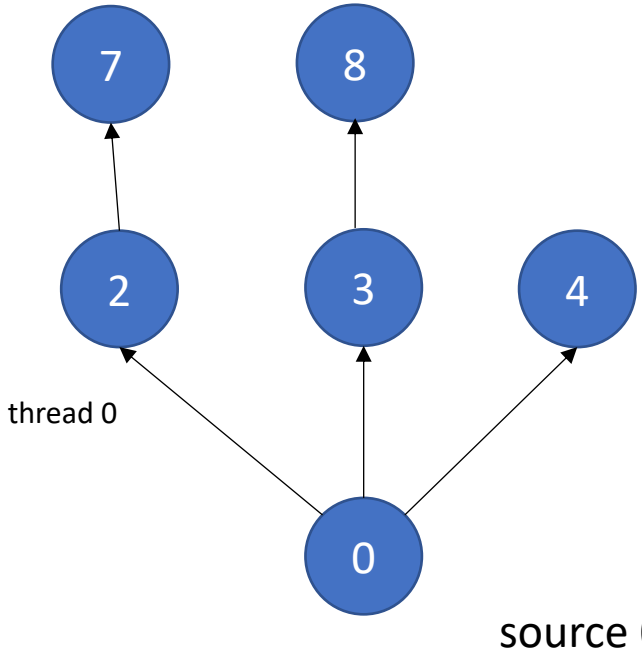


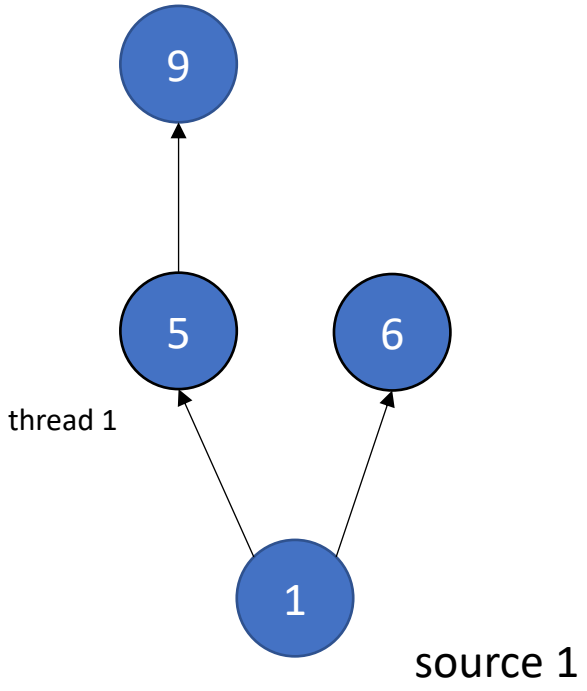


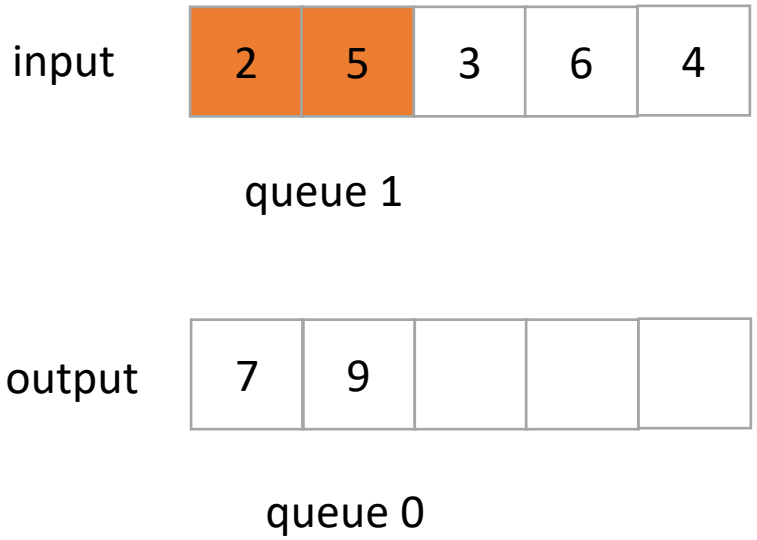


Input/Output Queues

- Example: Information flow in graph applications:

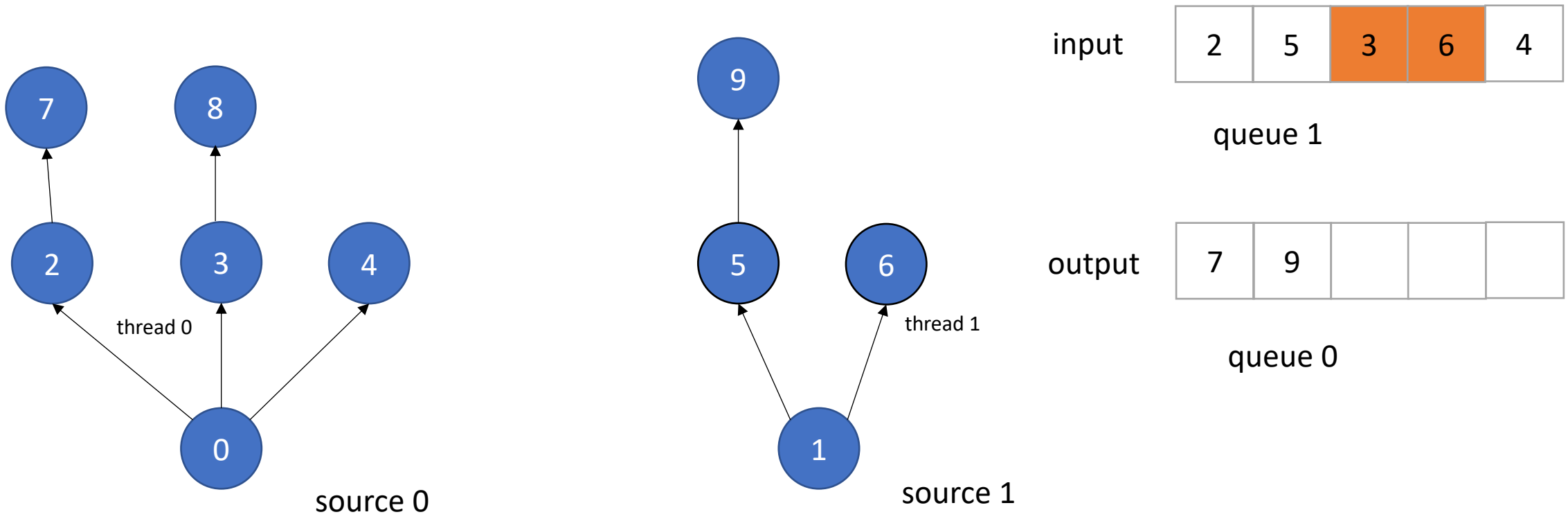






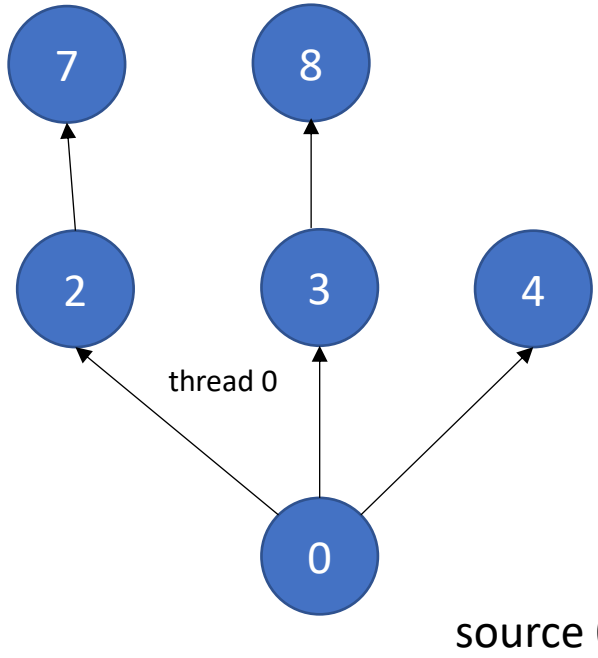
Input/Output Queues

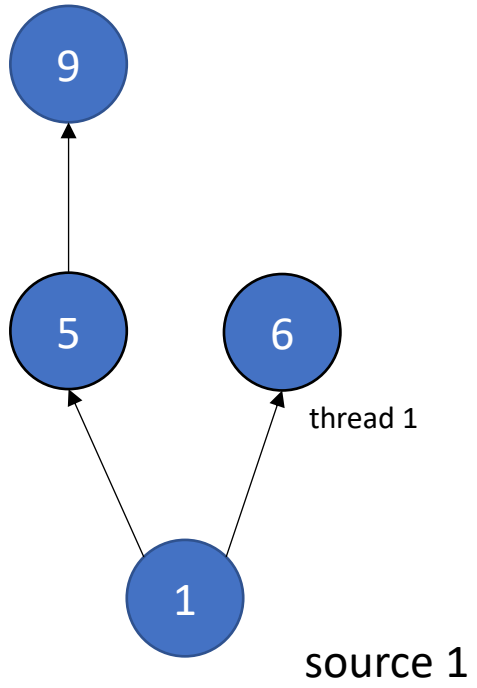
- Example: Information flow in graph applications:

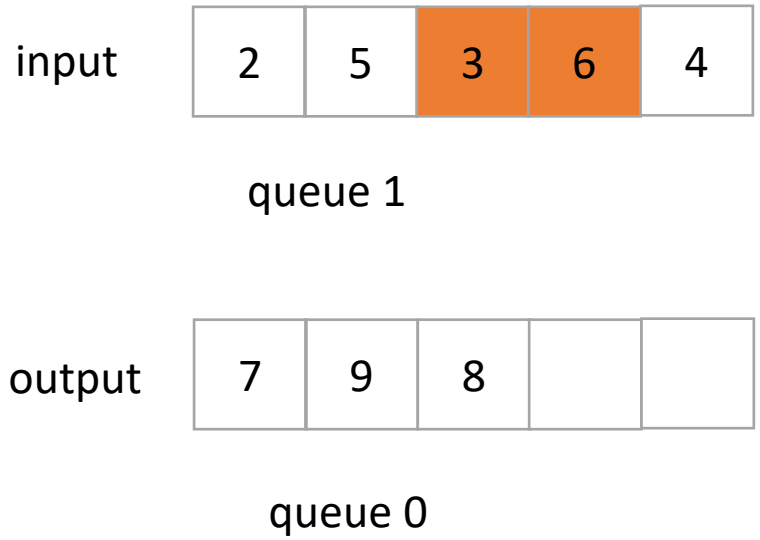


Input/Output Queues

- Example: Information flow in graph applications:

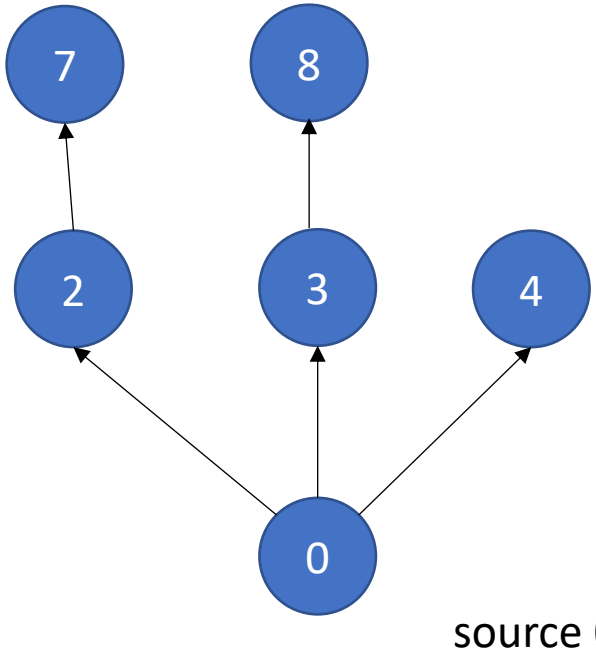


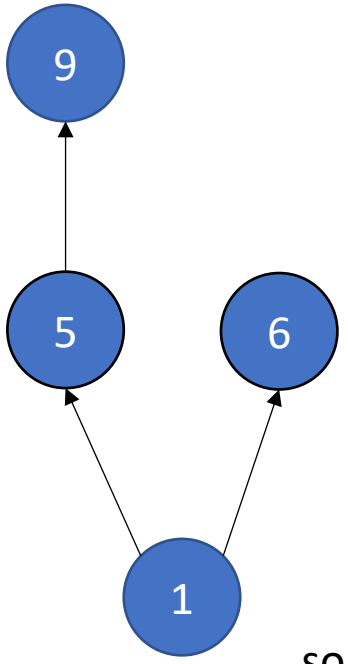


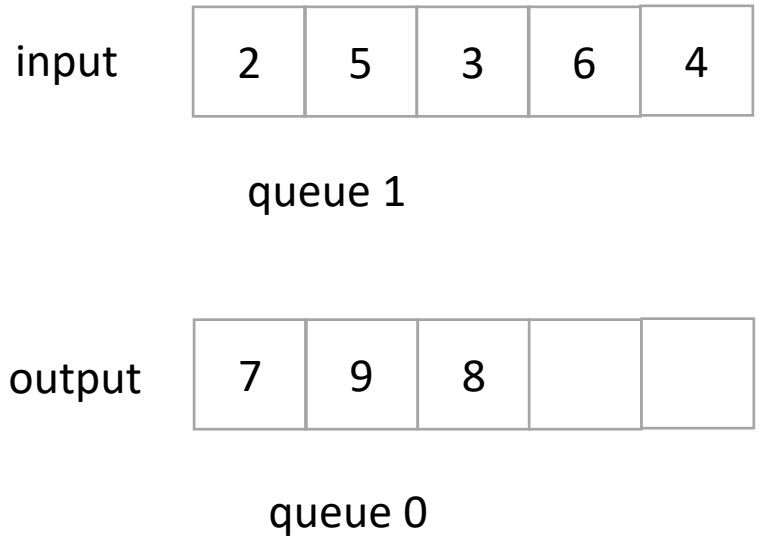


Input/Output Queues

- Example: Information flow in graph applications:

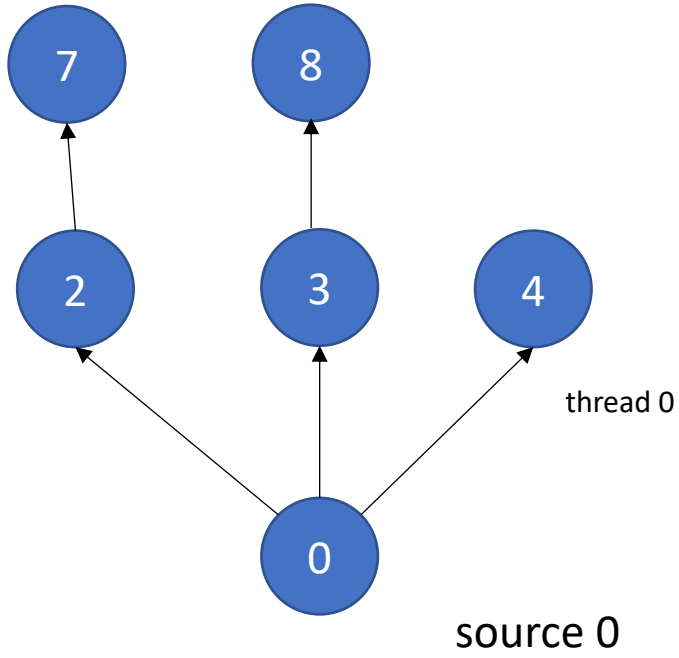


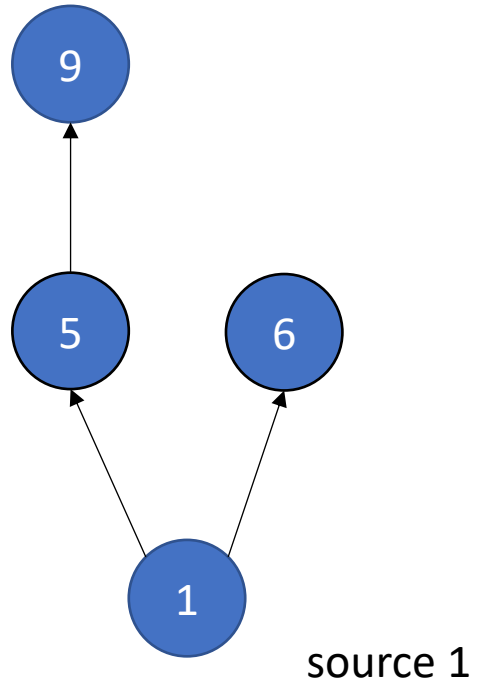


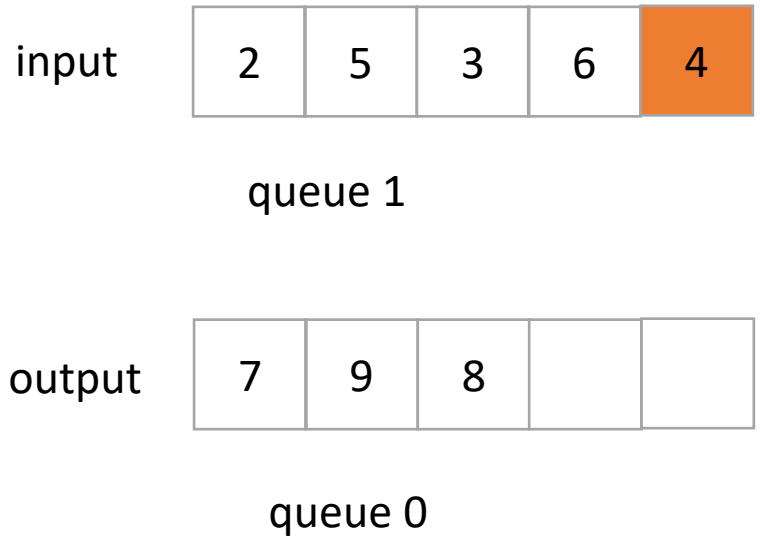


Input/Output Queues

- Example: Information flow in graph applications:

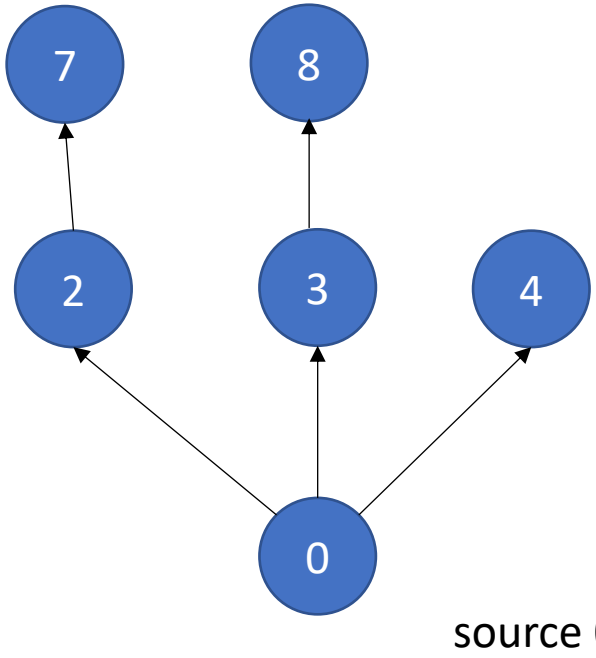


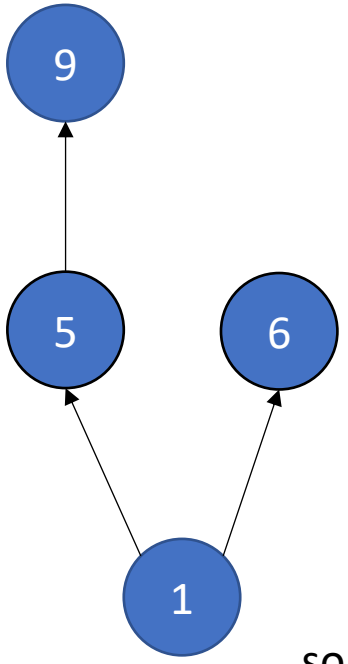


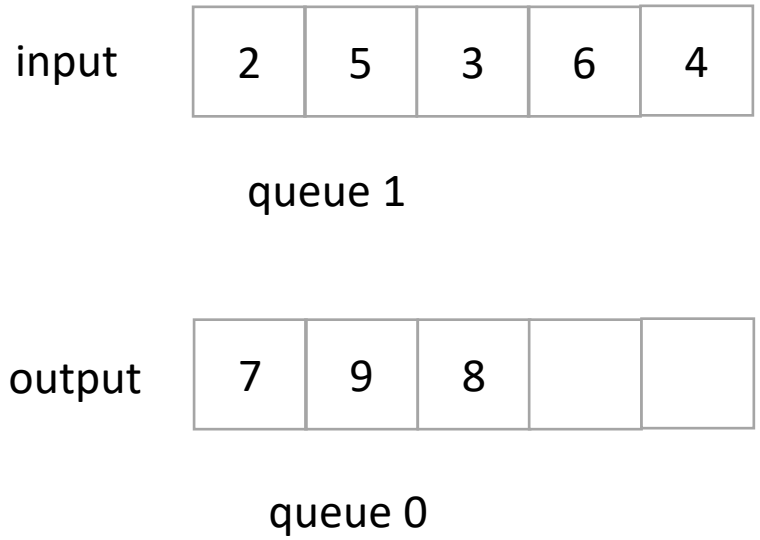


Input/Output Queues

- Example: Information flow in graph applications:

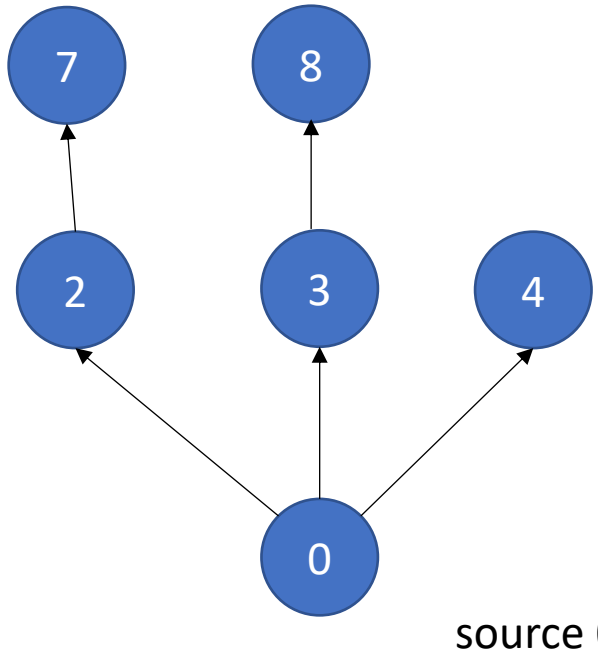


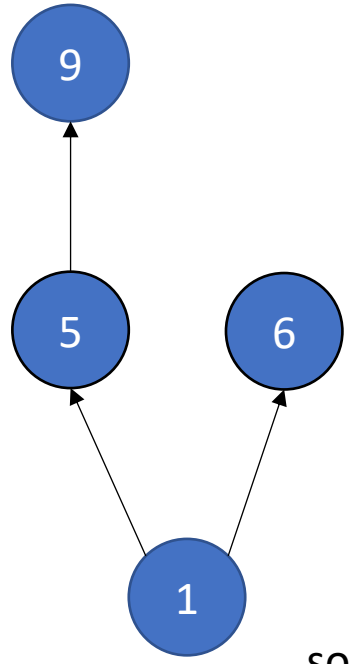




Input/Output Queues

- Example: Information flow in graph applications:





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

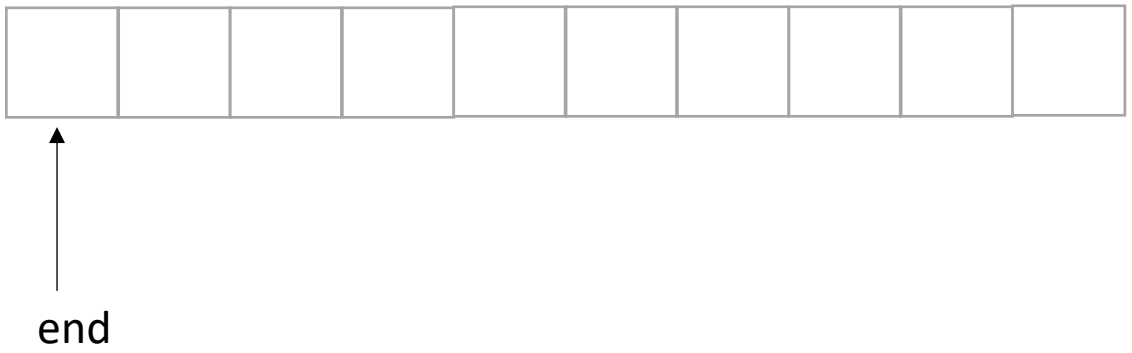
Cons:
- need to reason about overflow!

Note on terminology

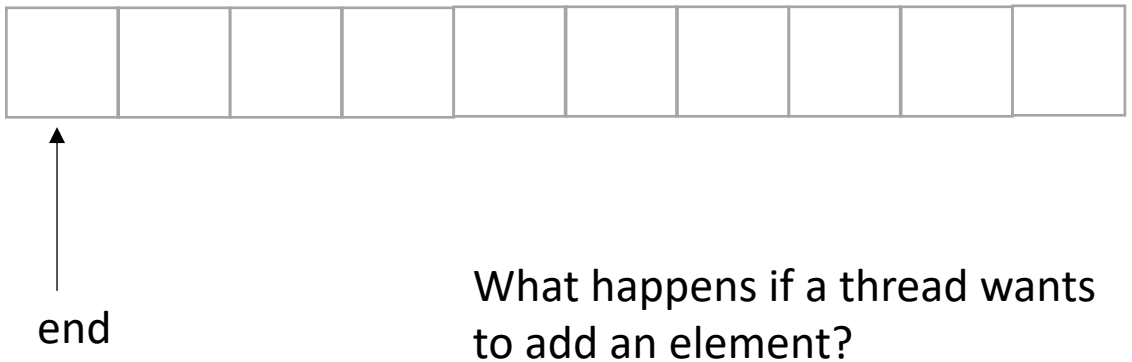
- Head/tail - often used in queue implementations, but switches when we start doing circular buffers.

- Front/end - To avoid confusion, we will use front/end for input/output queues.

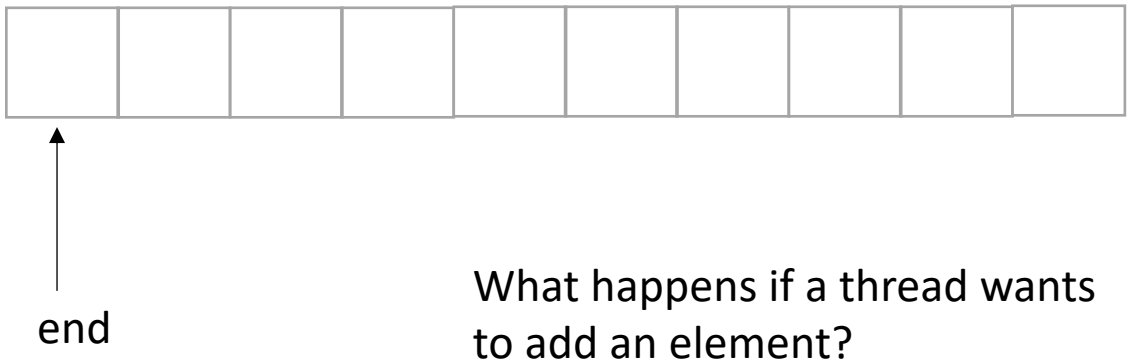
Implementation



Implementation



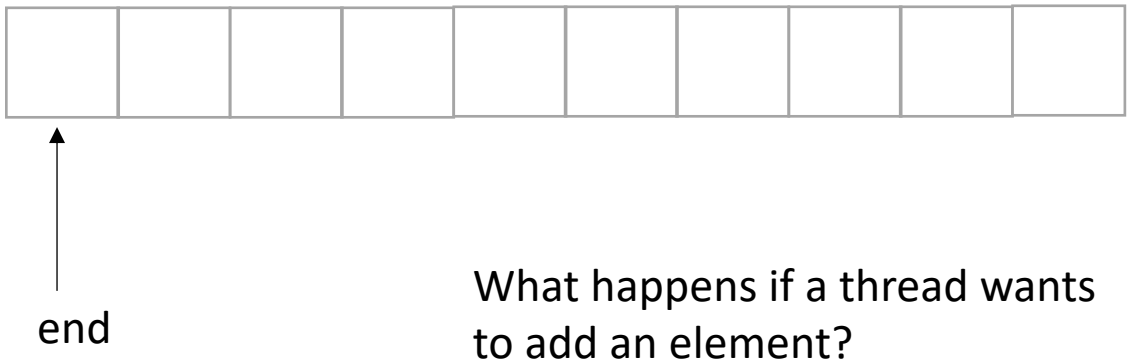
Implementation



What happens if a thread wants to add an element?

Think sequentially:

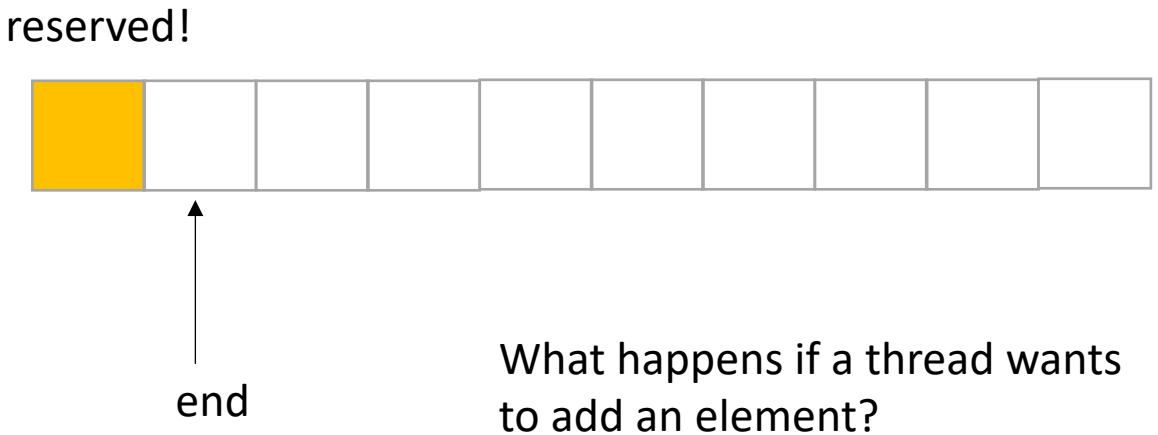
Implementation



What happens if a thread wants to add an element?

Think sequentially:
*reserve a space - increment end

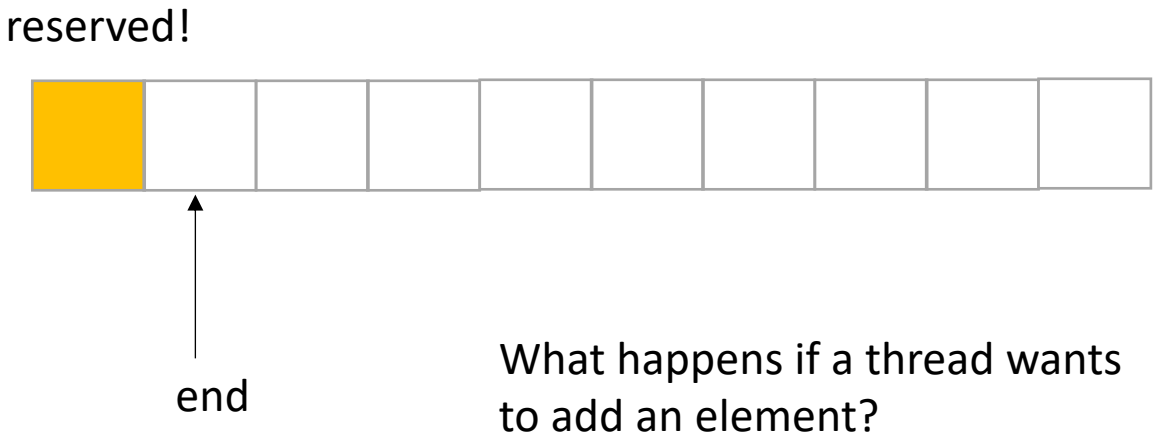
Implementation



What happens if a thread wants to add an element?

Think sequentially:
*reserve a space - increment end

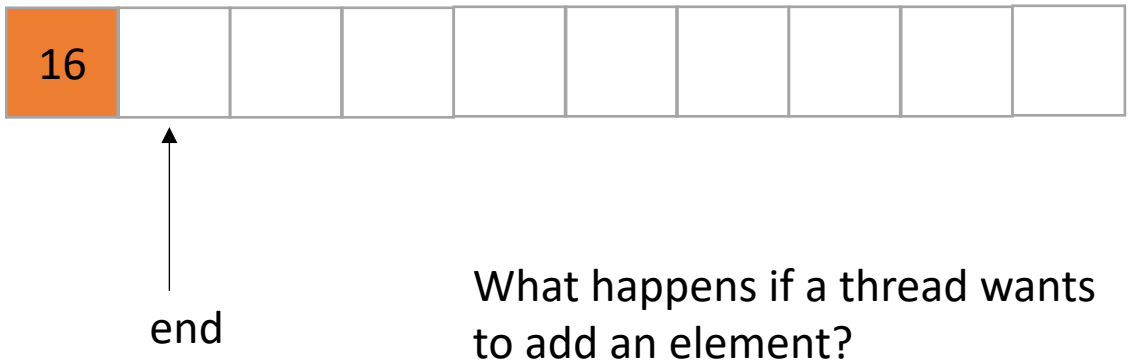
Implementation



What happens if a thread wants to add an element?

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

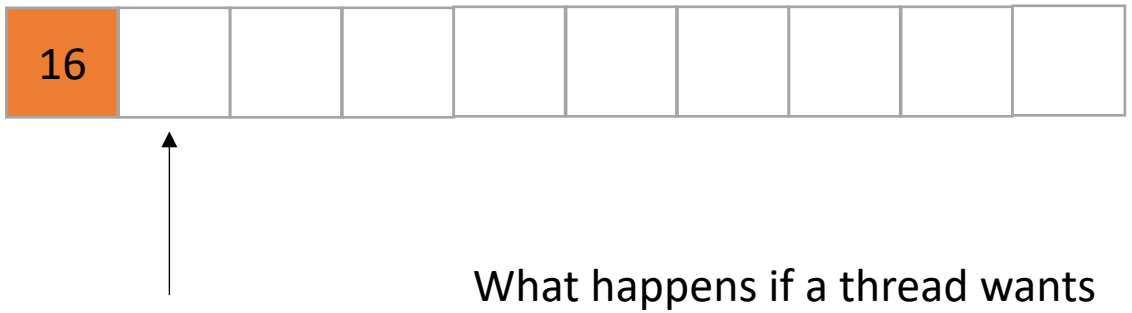
Implementation



What happens if a thread wants to add an element?

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

Implementation

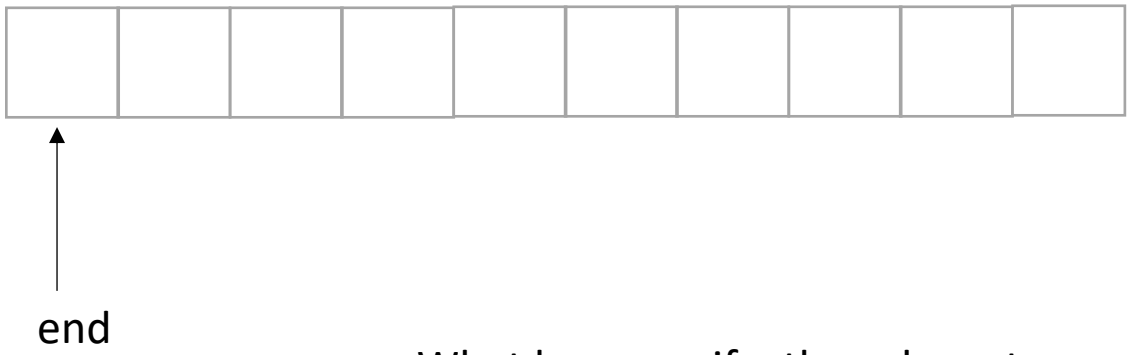


What happens if a thread wants to add an element?

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

done!

Implementation

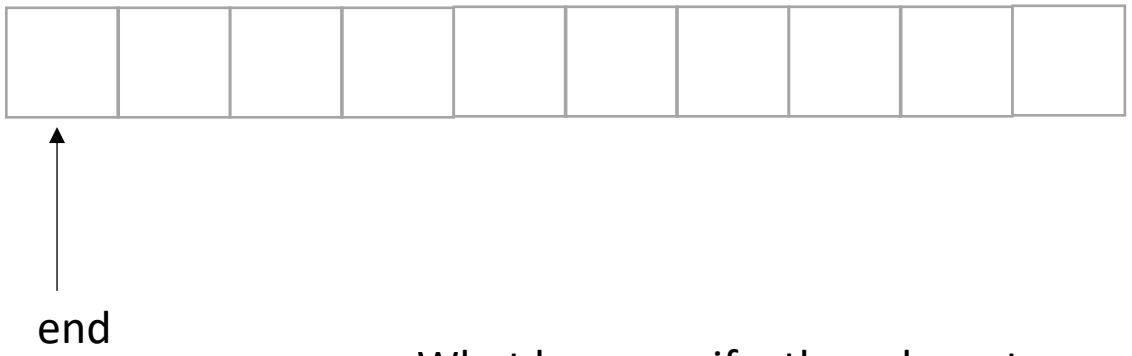


What happens if a thread wants
to add an element?

Think concurrently:

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

Implementation

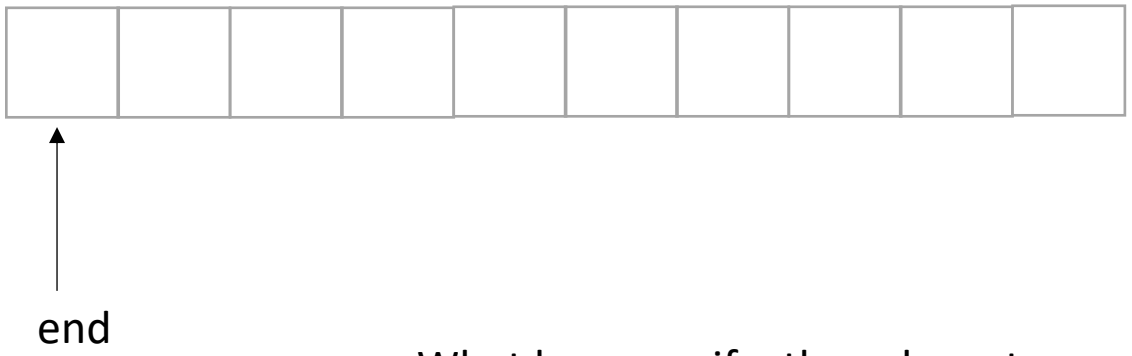


What happens if a thread wants to add an element?

Think concurrently:

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

Implementation



Thread 0:
enq(6);

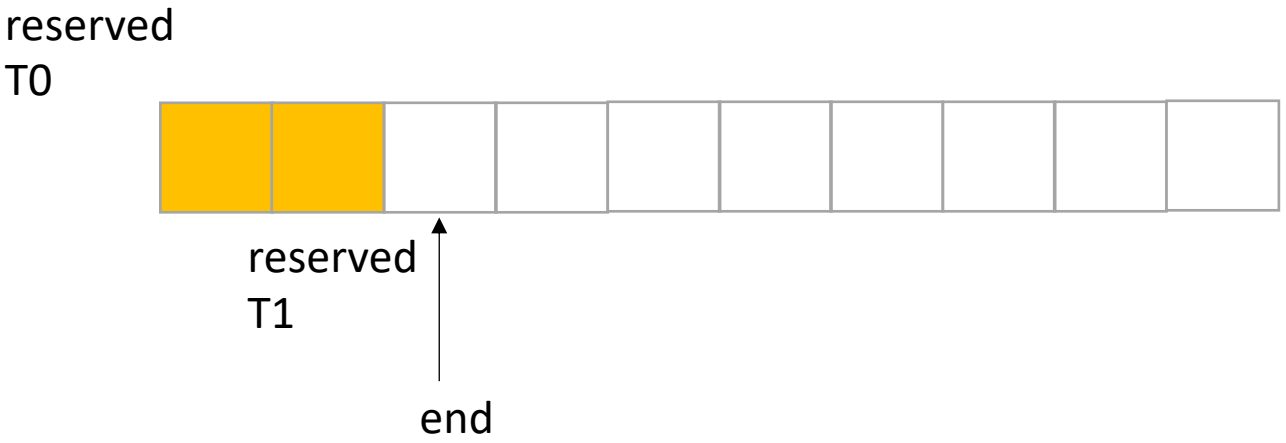
Thread 1:
enq(7);

What happens if a thread wants to add an element?

Think concurrently:

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

Implementation



Thread 0:
enq(6);

Thread 1:
enq(7);

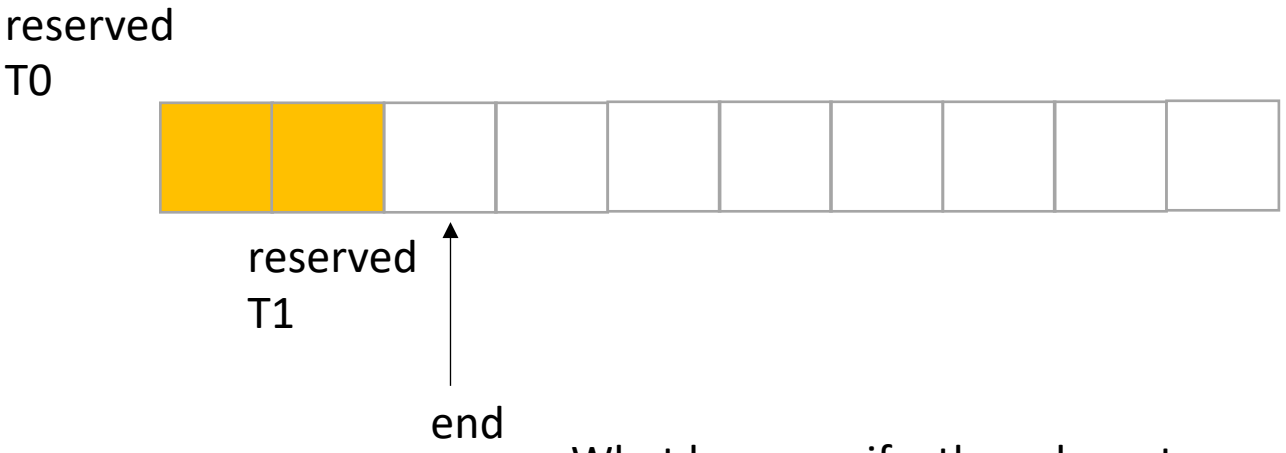
What happens if a thread wants to add an element?

Think concurrently:

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

Implementation

does it matter which order threads add their data?



Thread 0:
enq(6);

Thread 1:
enq(7);

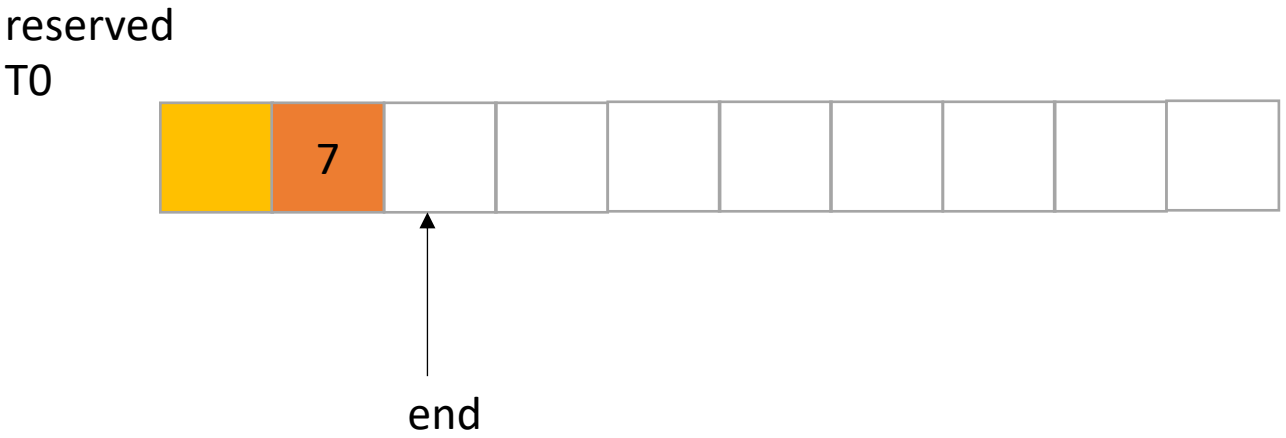
What happens if a thread wants to add an element?

Think concurrently:

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

Implementation

does it matter which order threads add their data?



Thread 0:
enq(6);

Thread 1:
enq(7);

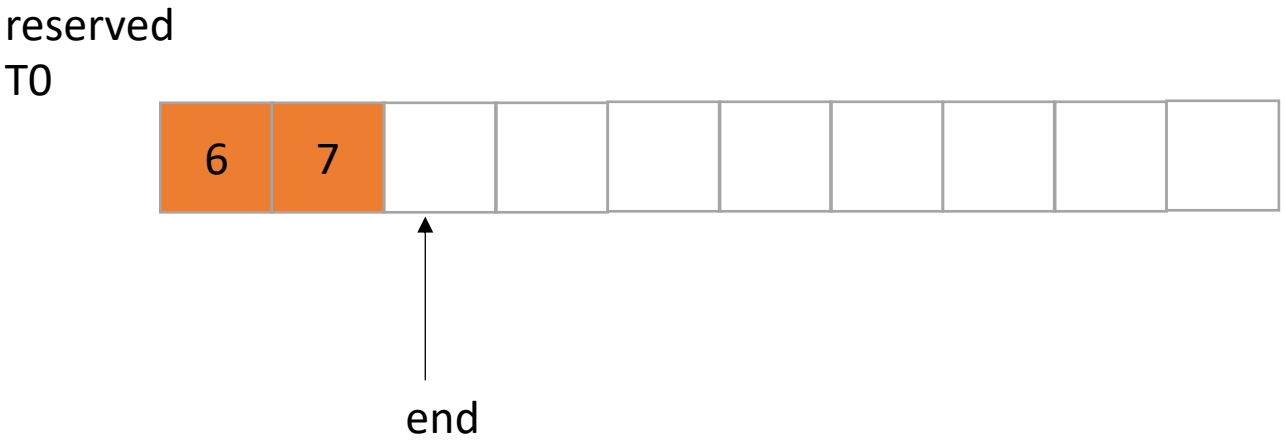
What happens if a thread wants to add an element?

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:

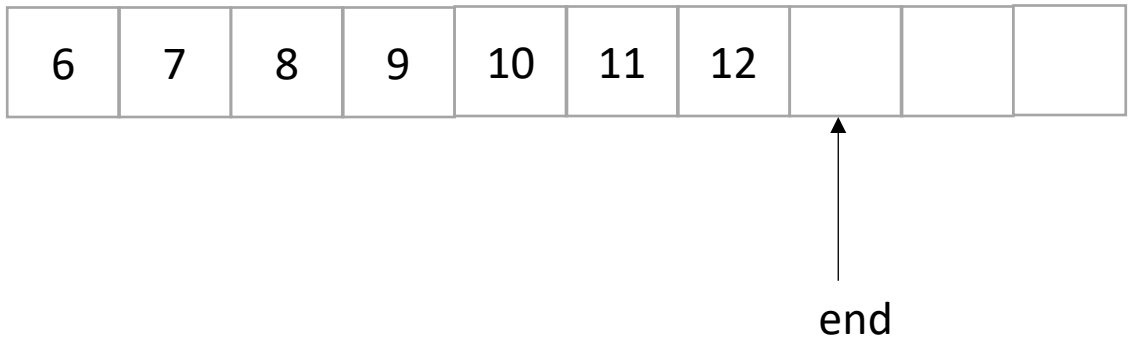
```
reserved_index = atomic_fetch_add(&end, 1);
```

```
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?

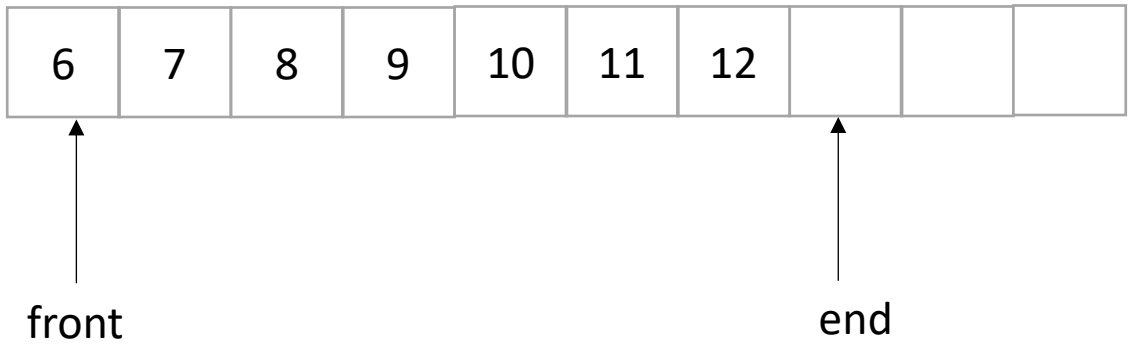
What about Input?

- Now we only do deqs



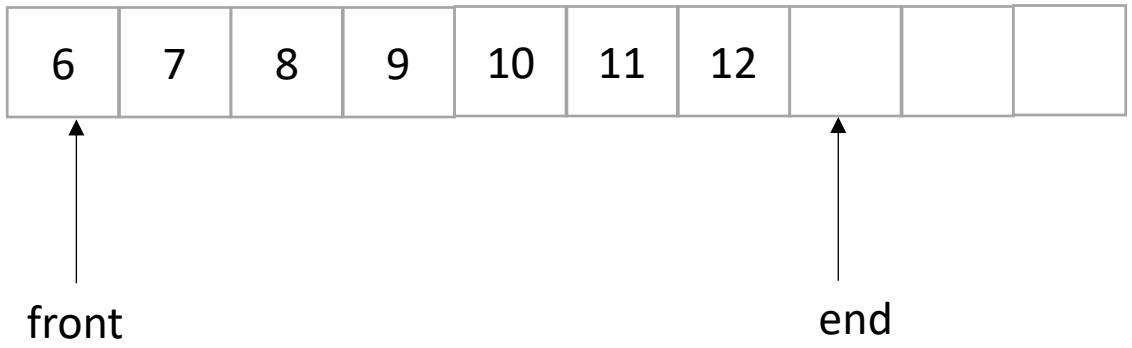
What about Input?

- Now we only do deqs



What about Input?

- Now we only do deqs



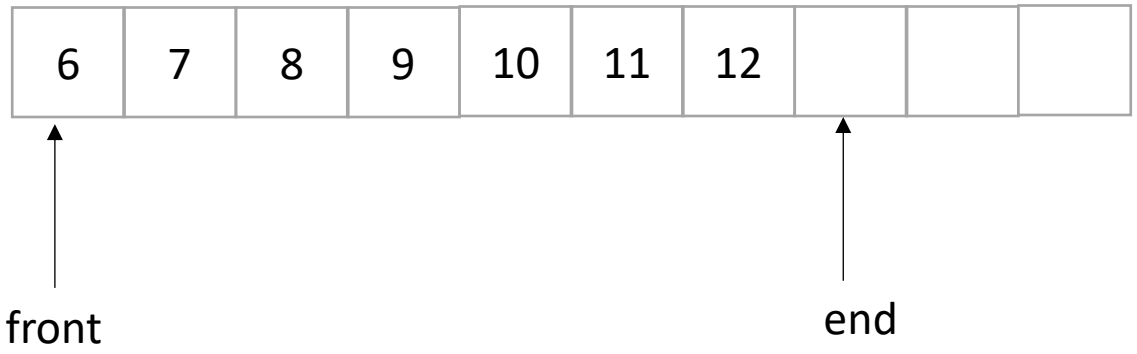
What happens if a thread wants to add an element?

Think concurrently:

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

What about Input?

- Now we only do deqs



Thread 0:
deq();

Thread 1:
deq();

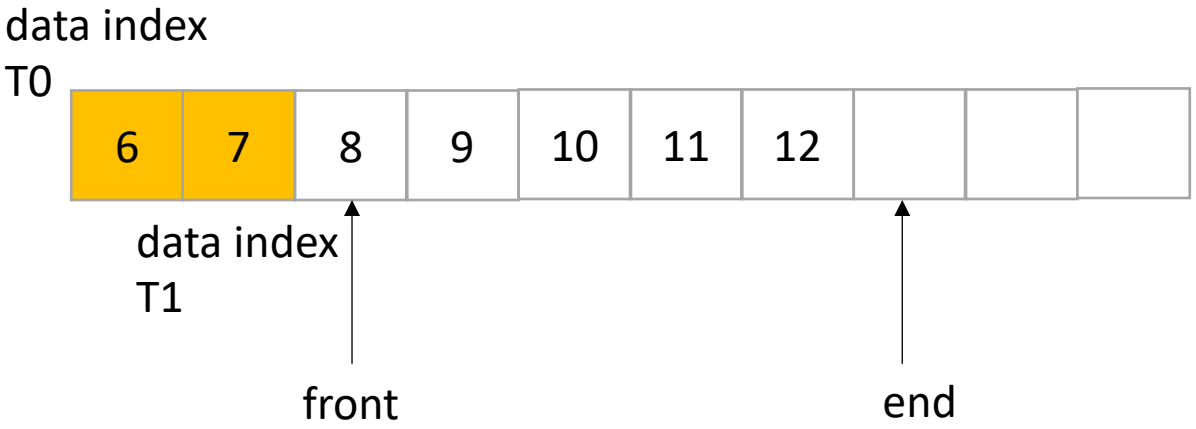
What happens if a thread wants to add an element?

Think concurrently:

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

What about Input?

- Now we only do deqs



Thread 0:
deq();

Thread 1:
deq();

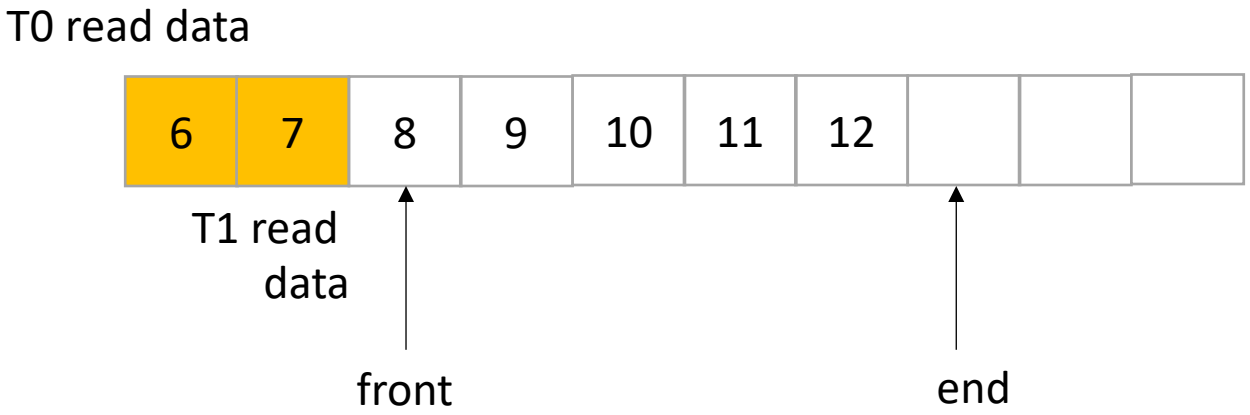
What happens if a thread wants to add an element?

Think concurrently:

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

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 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 ??;
     }
 }
```

```
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 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

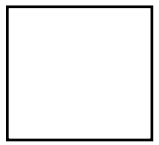
- First implementation:
  - Synchronous
  - Slow
  - Good for debugging

- enq does not return until value is deq’ed

Synchronous Producer Consumer Queues

Producer Thread

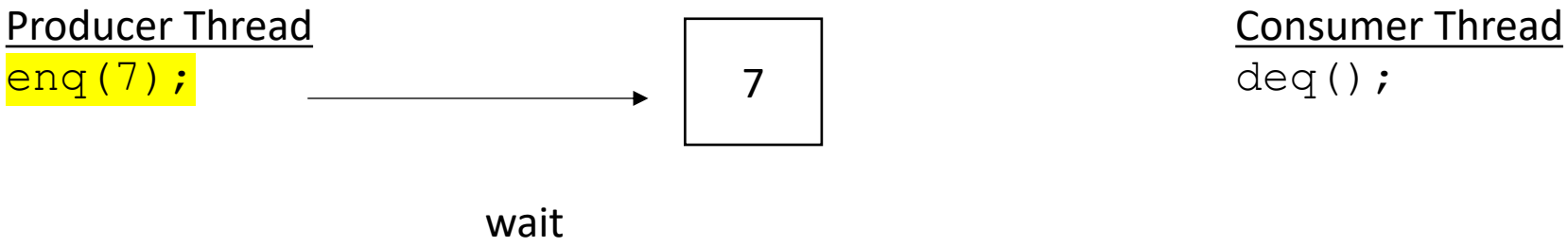
```
enq(7);
```



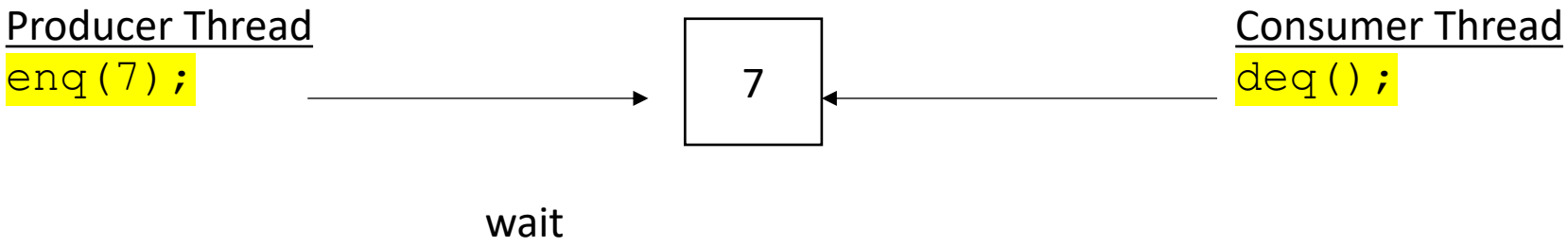
Consumer Thread

```
deq();
```

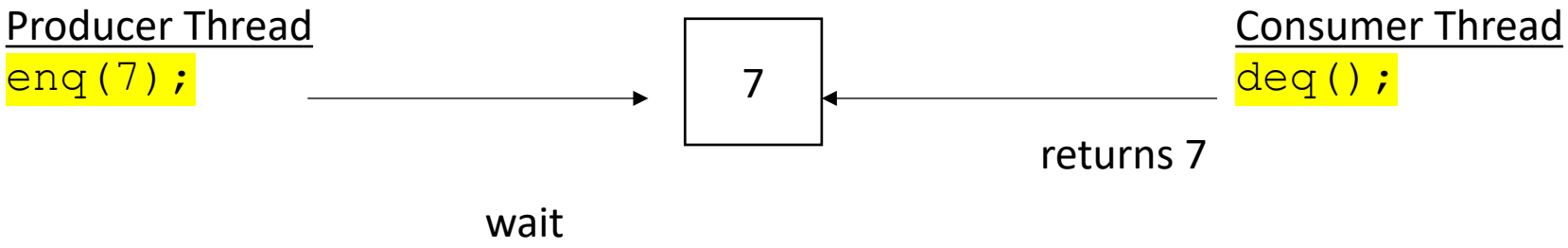
Synchronous Producer Consumer Queues



Synchronous Producer Consumer Queues

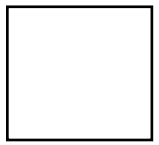


Synchronous Producer Consumer Queues



Synchronous Producer Consumer Queues

Producer Thread
```
enq(7);
```



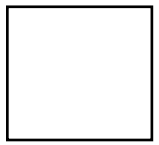
Consumer Thread
```
deq();
```

both can continue

Synchronous Producer Consumer Queues

Producer Thread

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



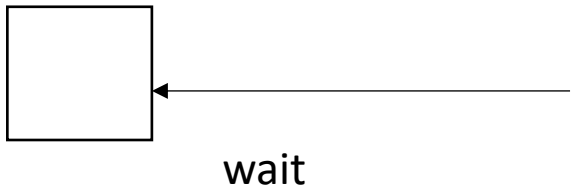
Consumer Thread

```
deq();
```

Synchronous Producer Consumer Queues

Producer Thread

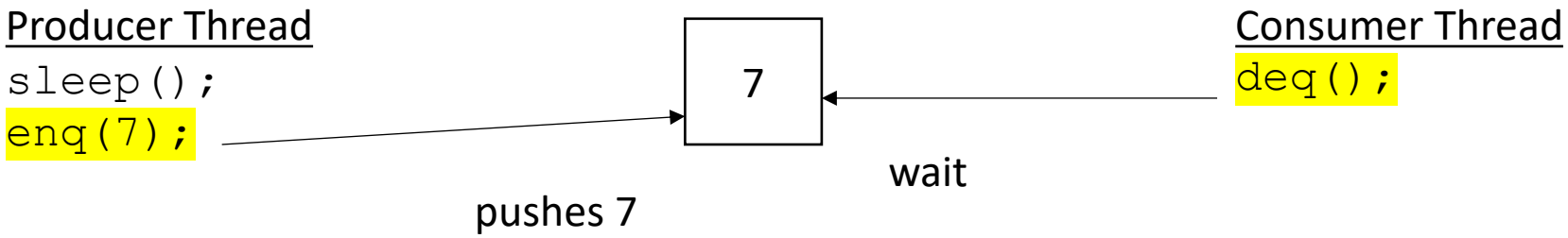
```
sleep();
enq(7);
```



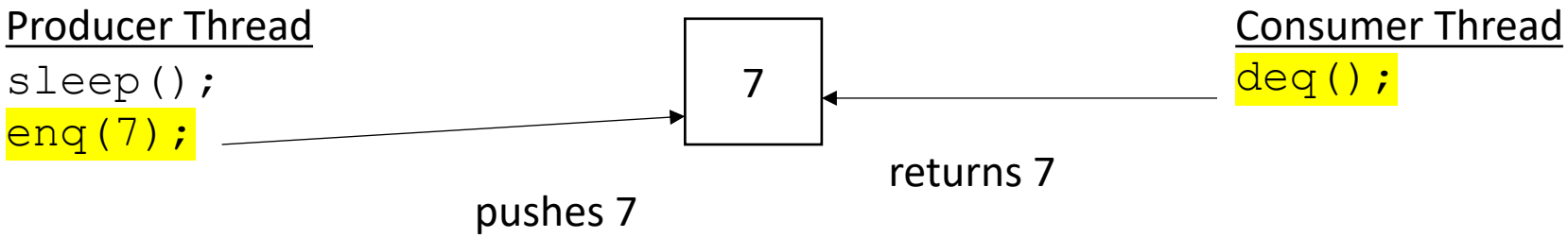
Consumer Thread

```
deq();
```

Synchronous Producer Consumer Queues



Synchronous Producer Consumer Queues

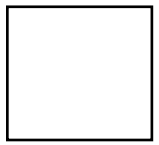


They both can continue

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
```



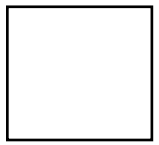
Consumer Thread

```
deq();
```

Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
```



Consumer Thread

```
deq();
```

What is our design doc to implement this?

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);
```

Consumer Thread

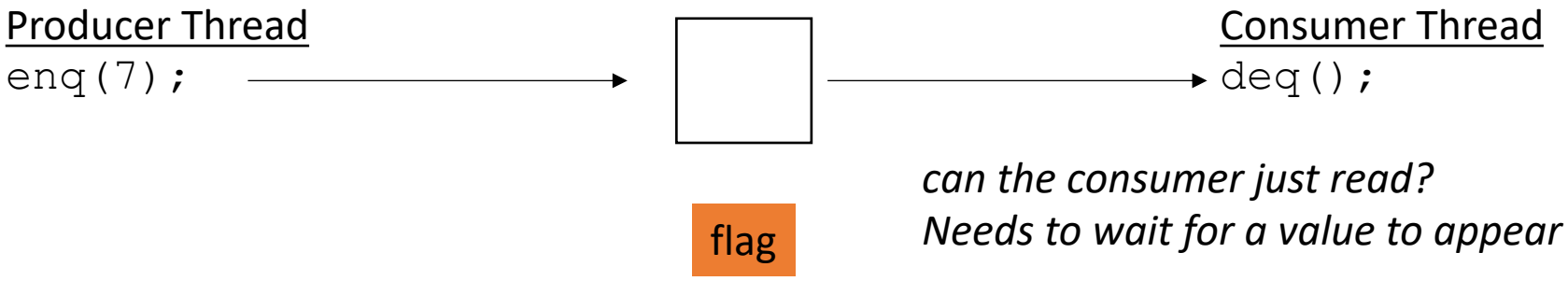
```
deq();
```

flag

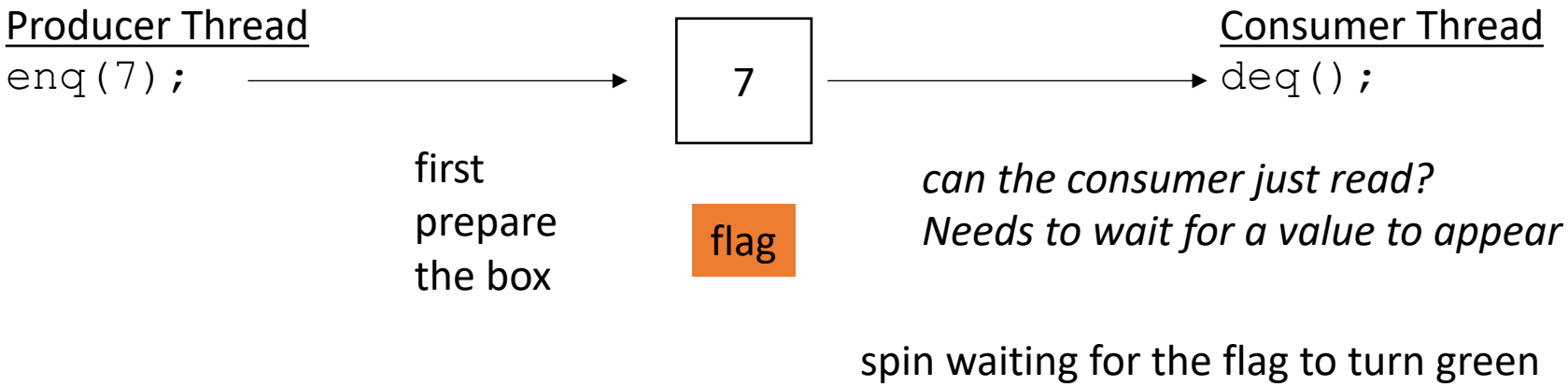
can the consumer just read?
Needs to wait for a value to appear

spin waiting for the flag to turn green

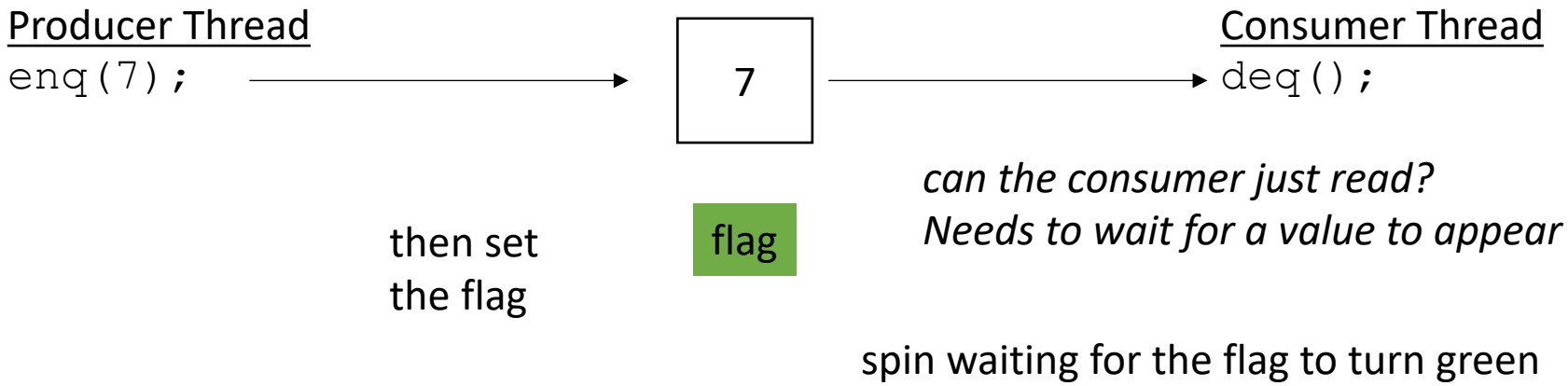
Synchronous Producer Consumer Queues



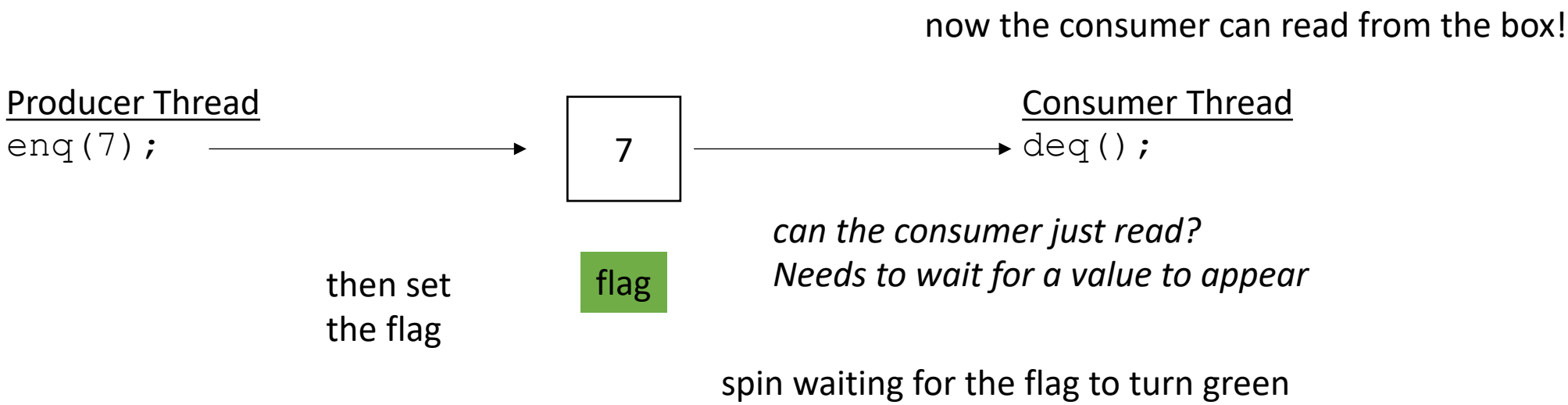
Synchronous Producer Consumer Queues



Synchronous Producer Consumer Queues



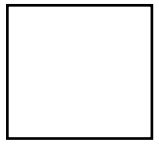
Synchronous Producer Consumer Queues



Synchronous Producer Consumer Queues

Producer Thread

```
enq(7);
```



flag

Consumer Thread

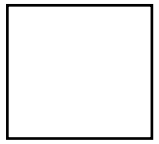
```
deq();
```

```
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

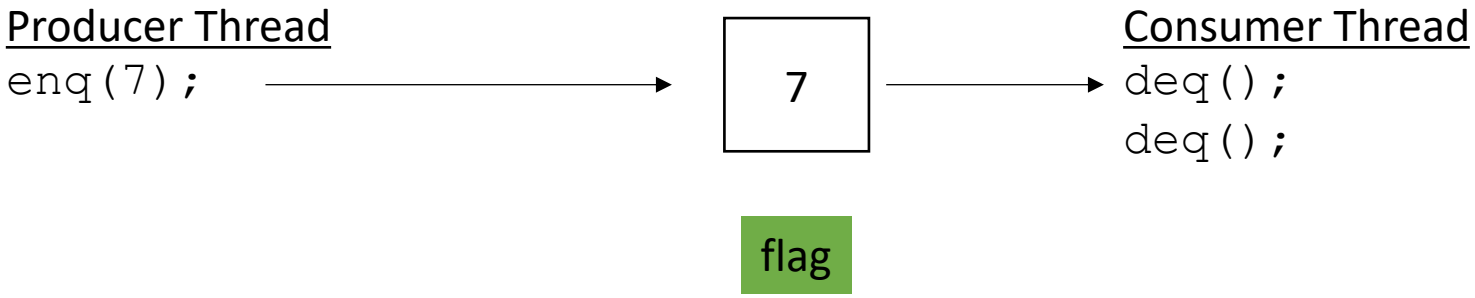
Consumer Thread

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

what happens when there are two deqs?

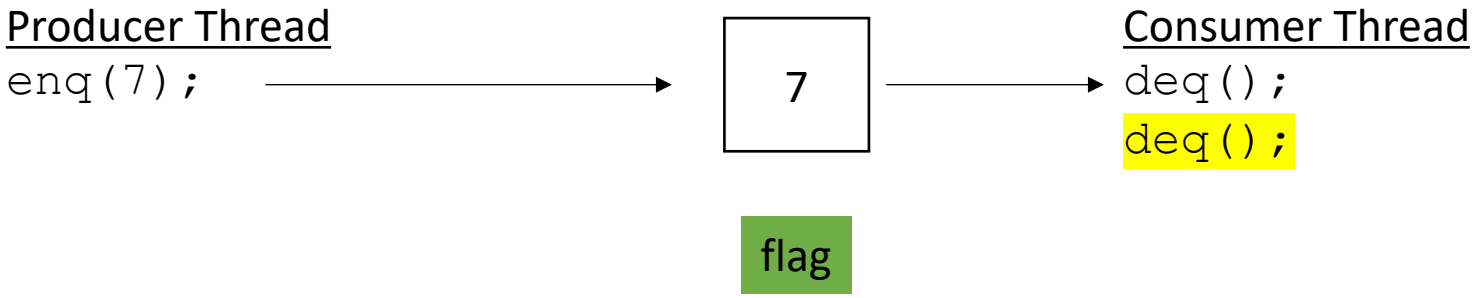
```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  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




```
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

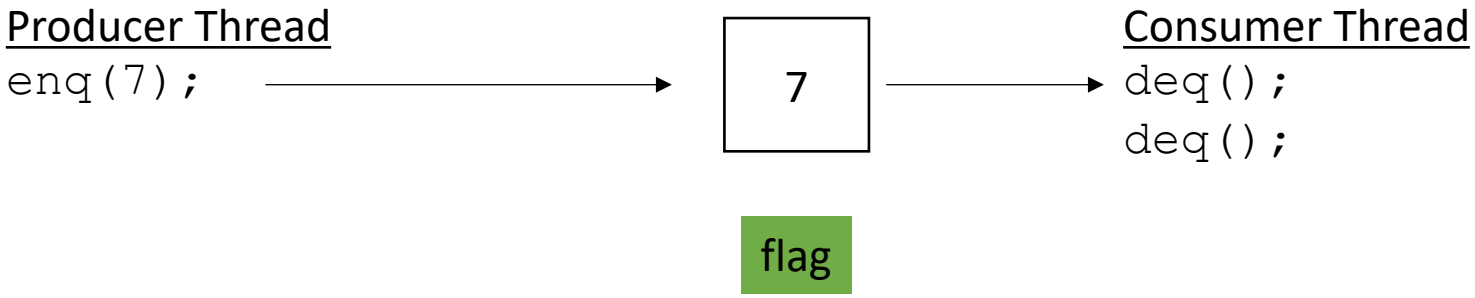



```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  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

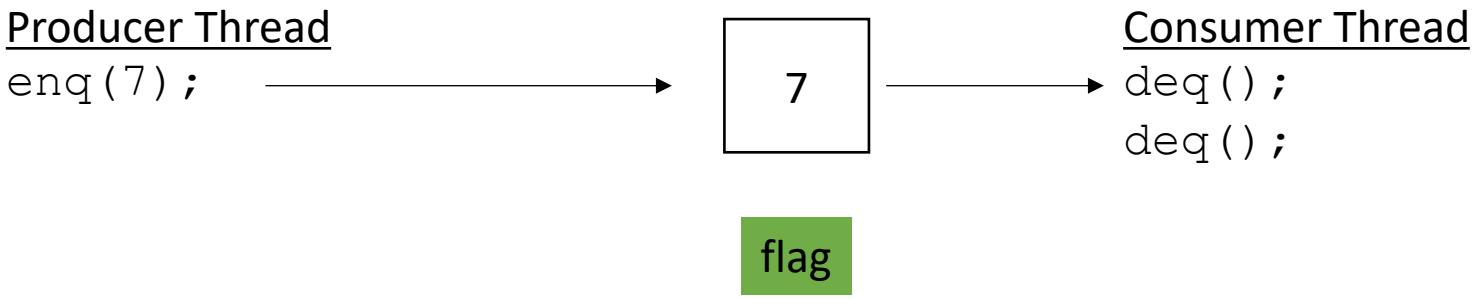



```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  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?

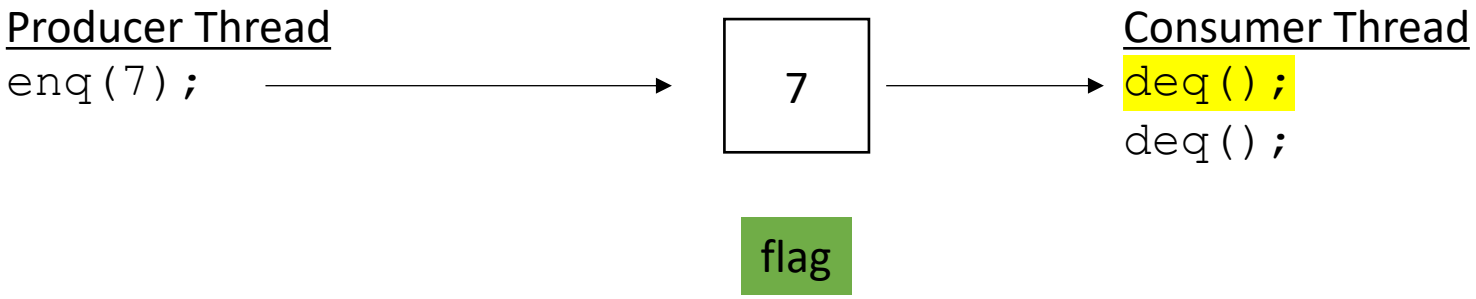
How to fix?

Synchronous Producer Consumer Queues



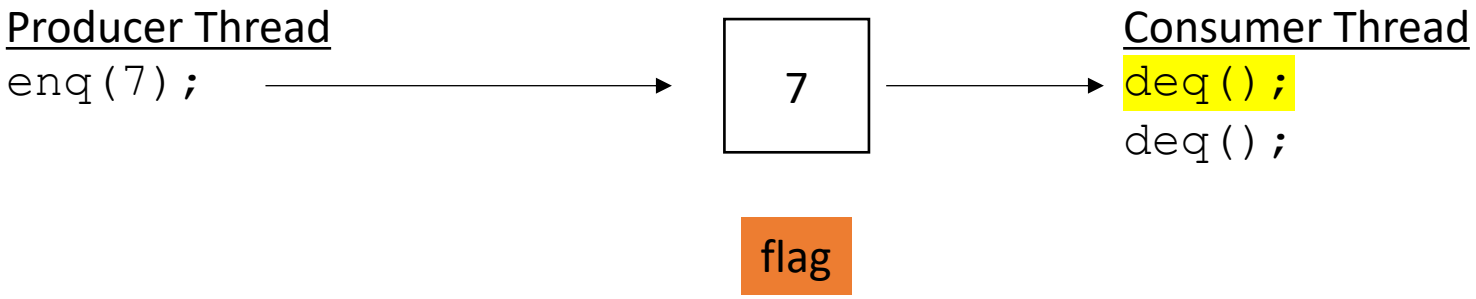

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  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



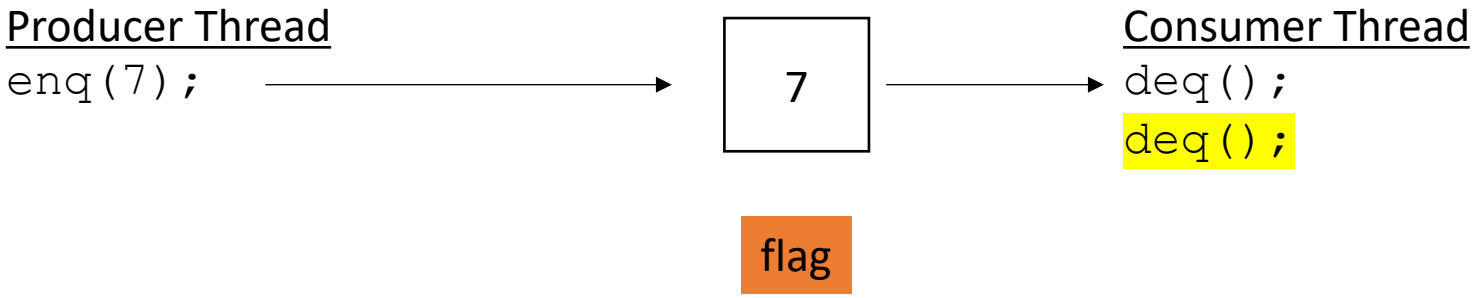

```
class SyncQueue {
  private:
    atomic_int box;
    atomic_bool flag;

  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
    }
  }
```

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
    }
  }
```

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
    }
  }
```

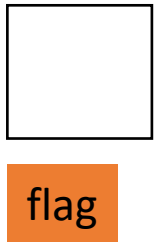
waiting like we are supposed to

Synchronous Producer Consumer Queues

reset (now with extra enq)

Producer Thread

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



Consumer Thread

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

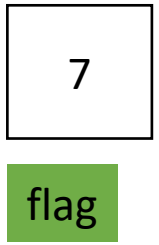
extra enq

```
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);
```



Consumer Thread

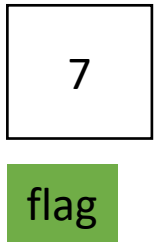
```
deq();
deq();
```

```
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

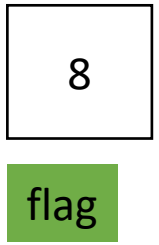
```
deq();
deq();
```

```
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);
```



Consumer Thread

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

7 was dropped!

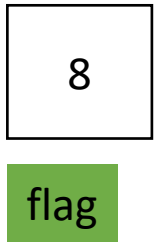
how to fix?

```
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

Producer Thread

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



Consumer Thread

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

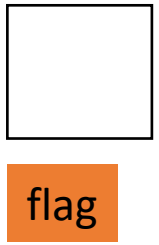
7 was dropped!

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



Producer Thread

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

Consumer Thread

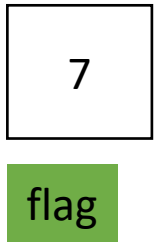
```
deq();
deq();
```

```
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);
```



Consumer Thread

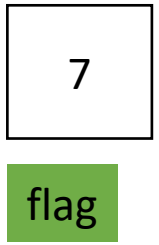
```
deq();
deq();
```

```
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

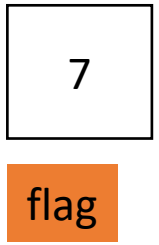
```
deq();
deq();
```

```
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);
```



Consumer Thread

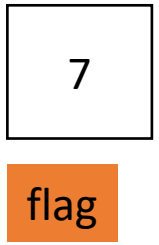
```
deq();
deq();
```

```
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);
```



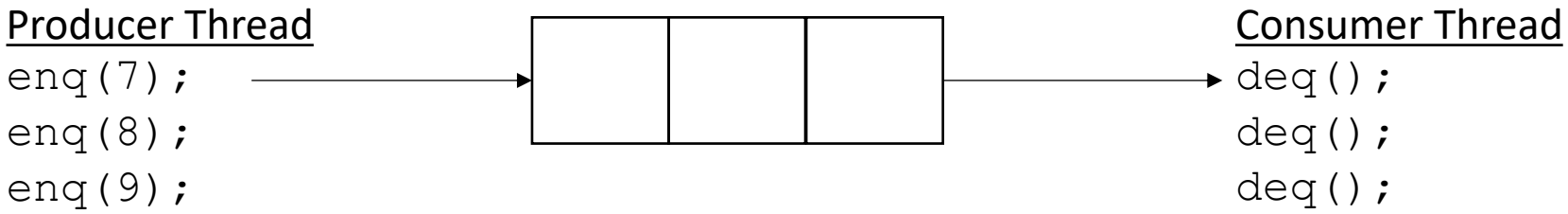
Consumer Thread

```
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
    }
  }
```

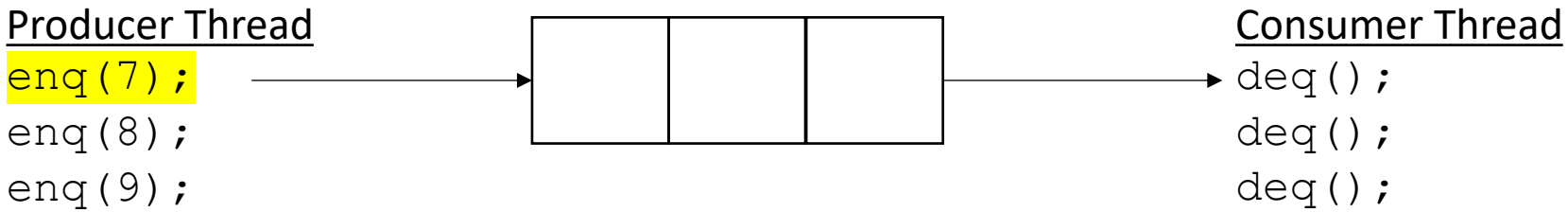
Producer Consumer Queues

- Asynchronous:



Producer Consumer Queues

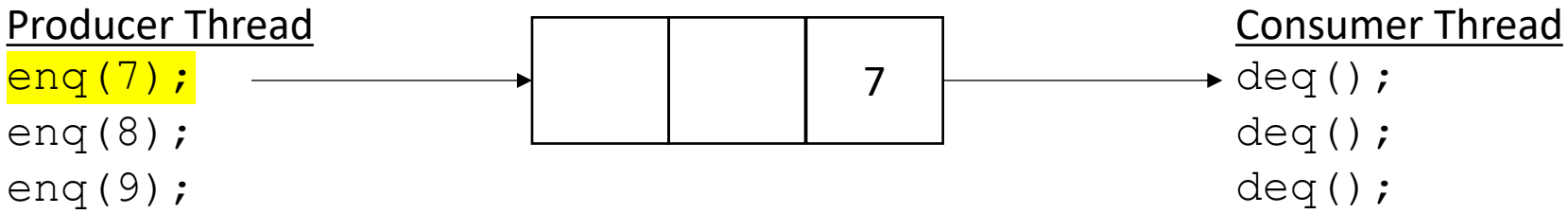
- Asynchronous:



no waiting for producer (while there is room)

Producer Consumer Queues

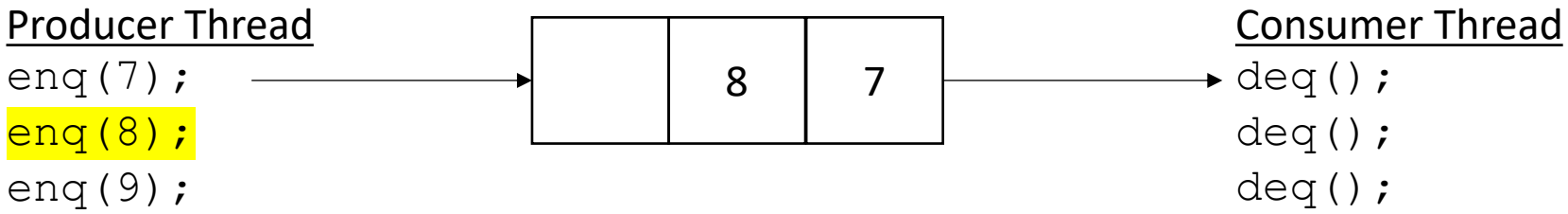
- Asynchronous:



no waiting for producer (while there is room)

Producer Consumer Queues

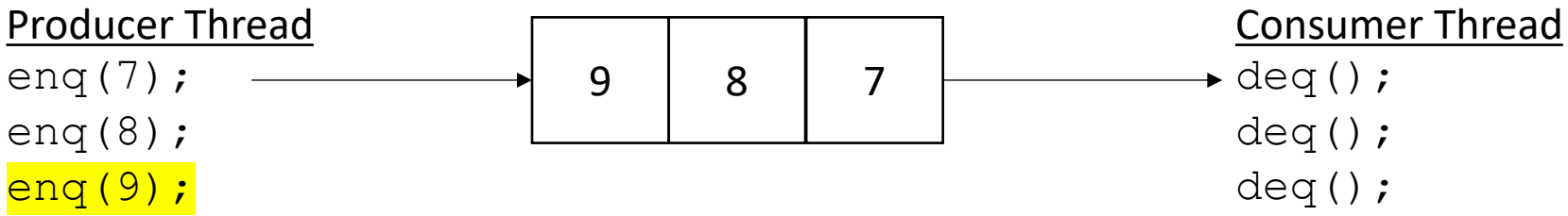
- Asynchronous:



no waiting for producer (while there is room)

Producer Consumer Queues

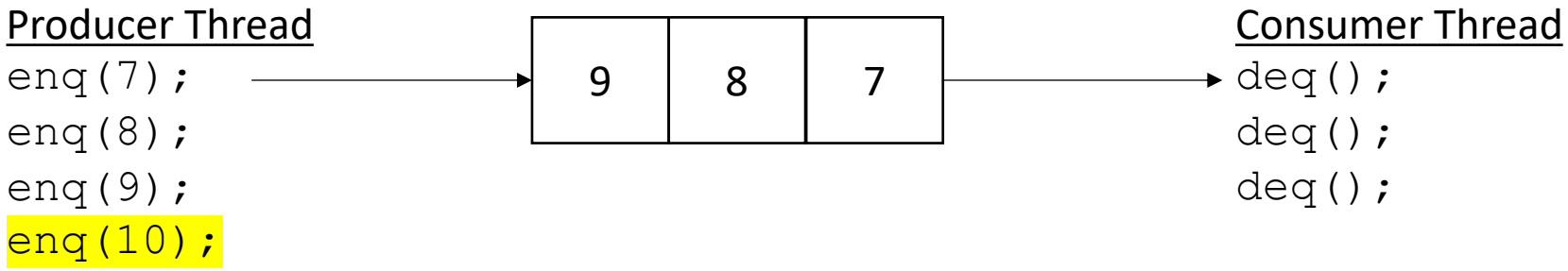
- Asynchronous:



no waiting for producer (while there is room)

Producer Consumer Queues

- Asynchronous:

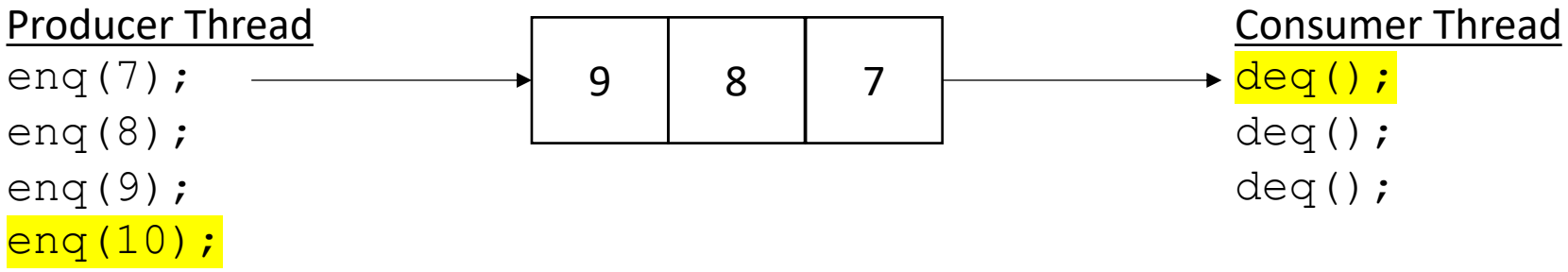


no waiting for producer (while there is room)

when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:



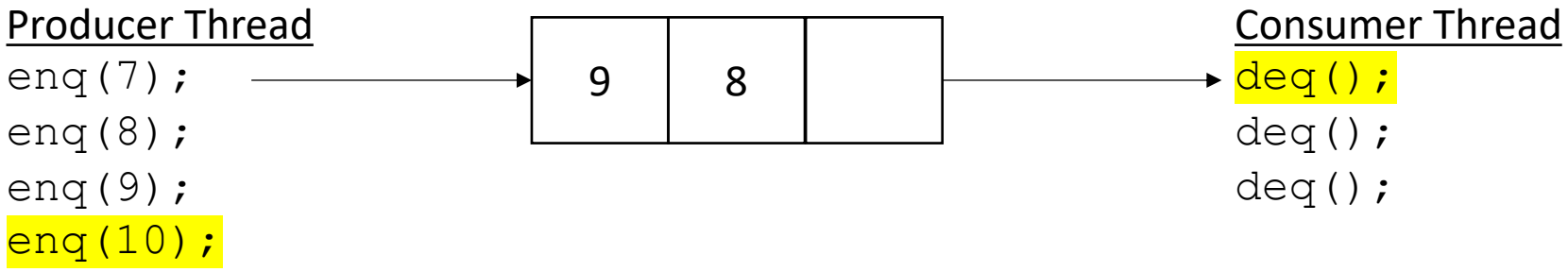
no waiting for producer (while there is room)

returns 7

when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:



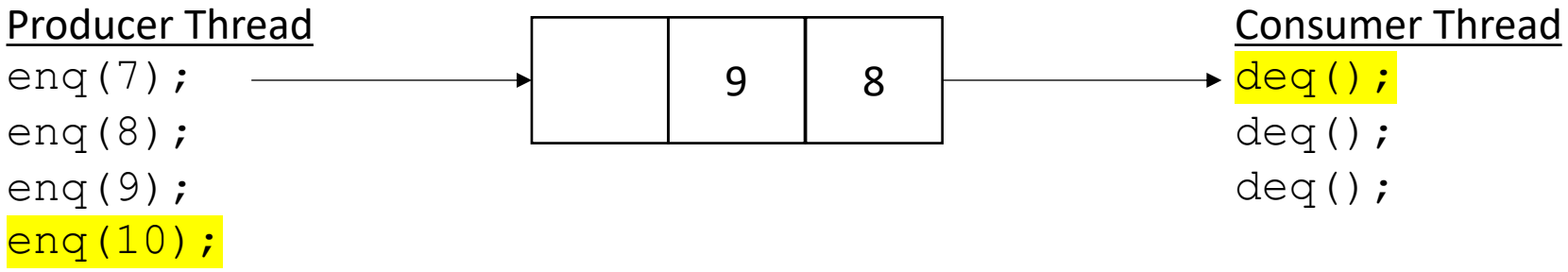
no waiting for producer (while there is room)

returns 7

when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:



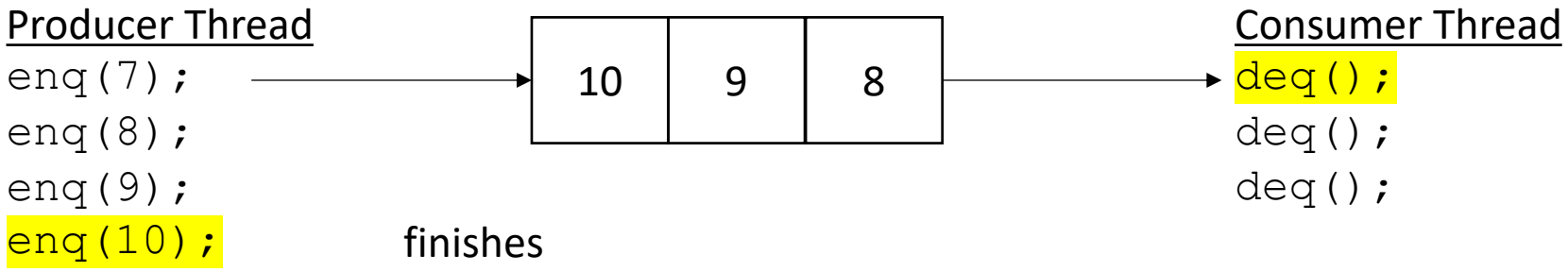
no waiting for producer (while there is room)

returns 7

when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:



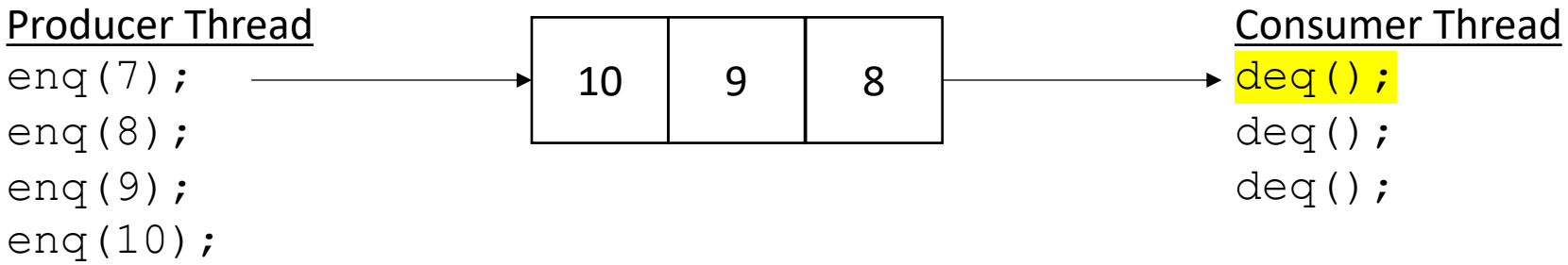
no waiting for producer (while there is room)

returns 7

when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:



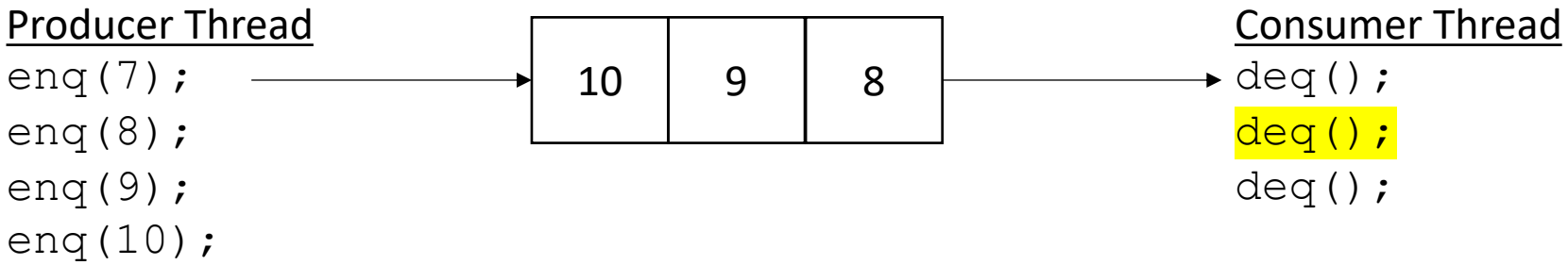
no waiting for producer (while there is room)

returns 7

when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:



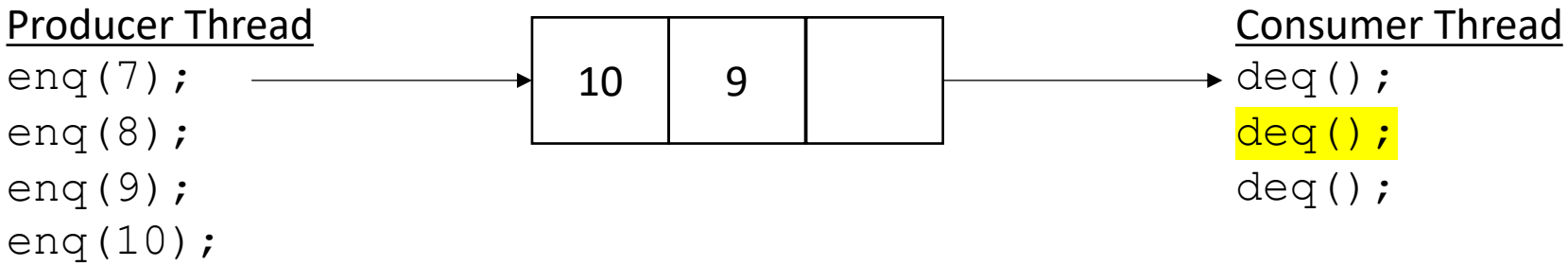
no waiting for producer (while there is room)

returns 8

when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:



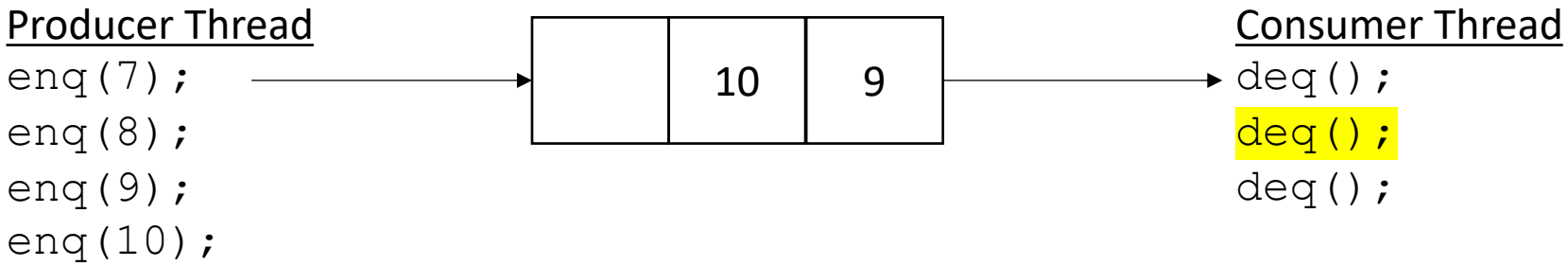
no waiting for producer (while there is room)

returns 8

when there is no room, the queue will wait

Producer Consumer Queues

- Asynchronous:



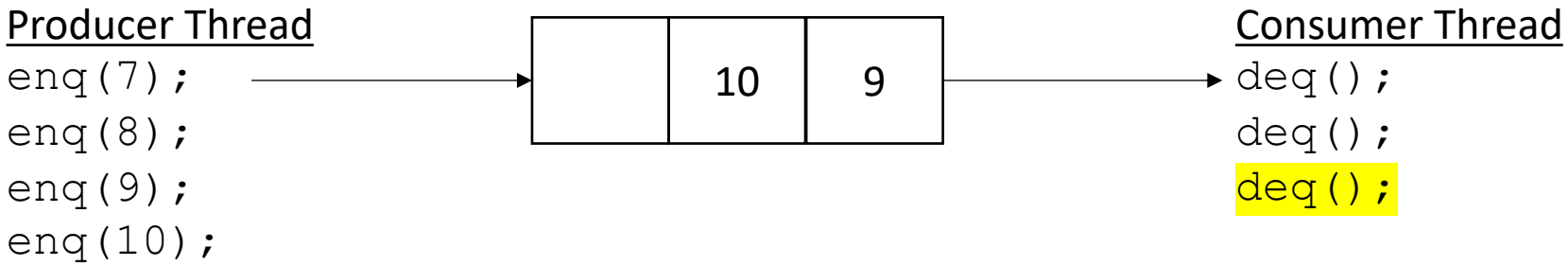
no waiting for producer (while there is room)

returns 8

when there is no room, the queue will wait

Producer Consumer Queues

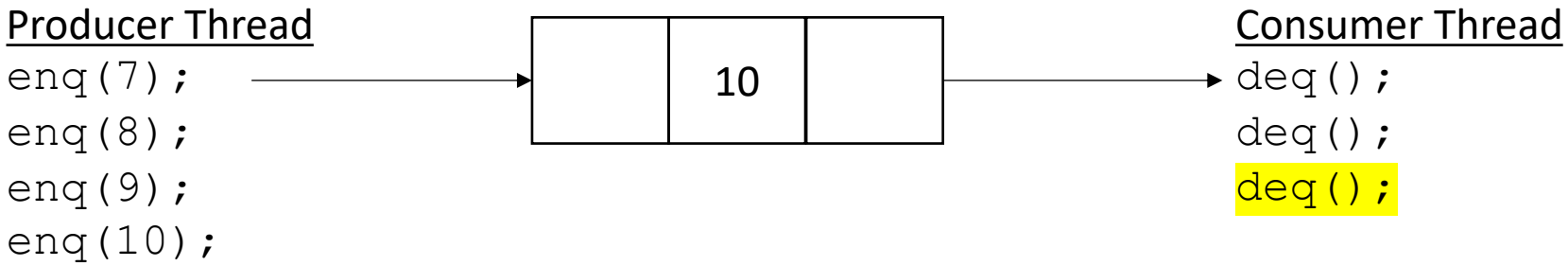
- Asynchronous:



returns 9

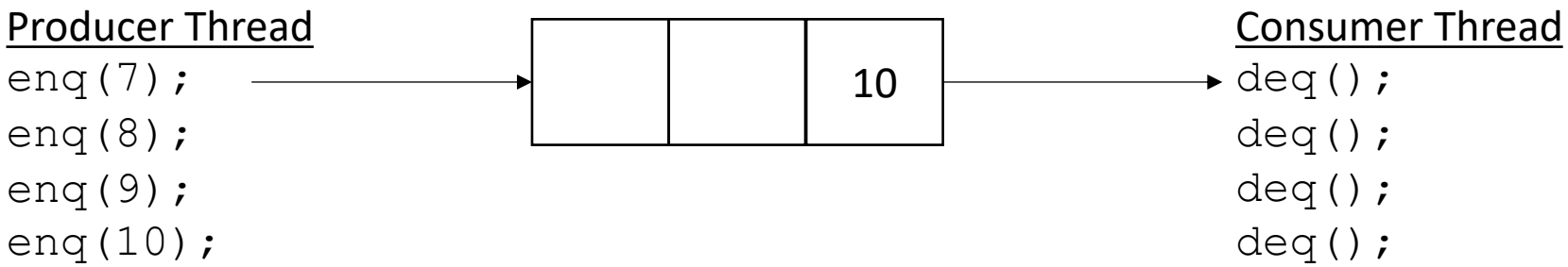
Producer Consumer Queues

- Asynchronous:



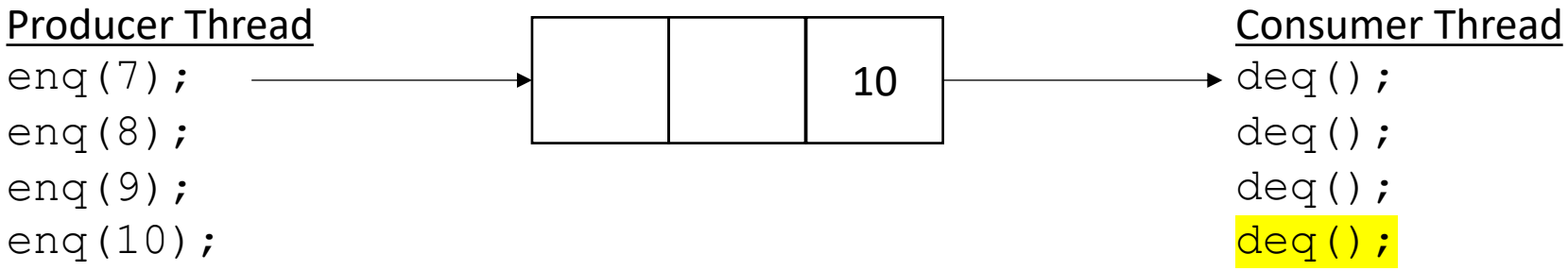
Producer Consumer Queues

- Asynchronous:



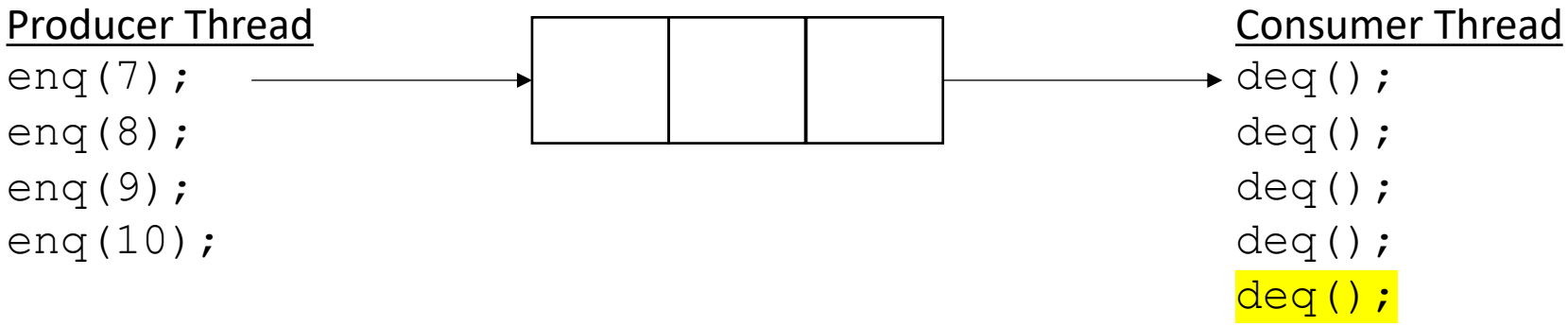
Producer Consumer Queues

- Asynchronous:



Producer Consumer Queues

- Asynchronous:



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

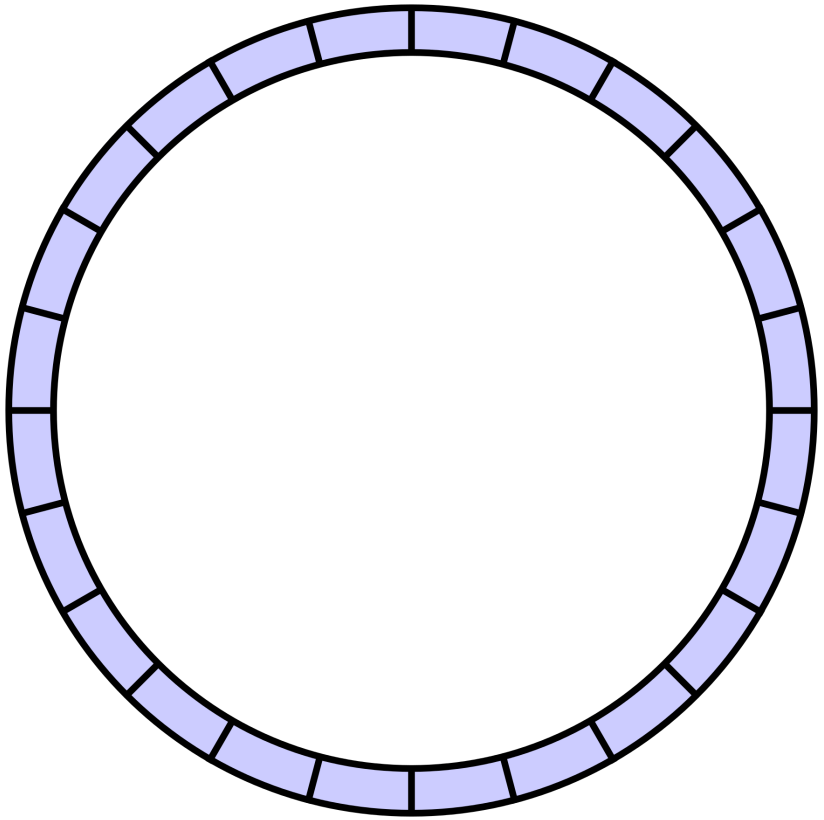
Producer Consumer Queues

- Start with a fixed size array

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

Producer Consumer Queues

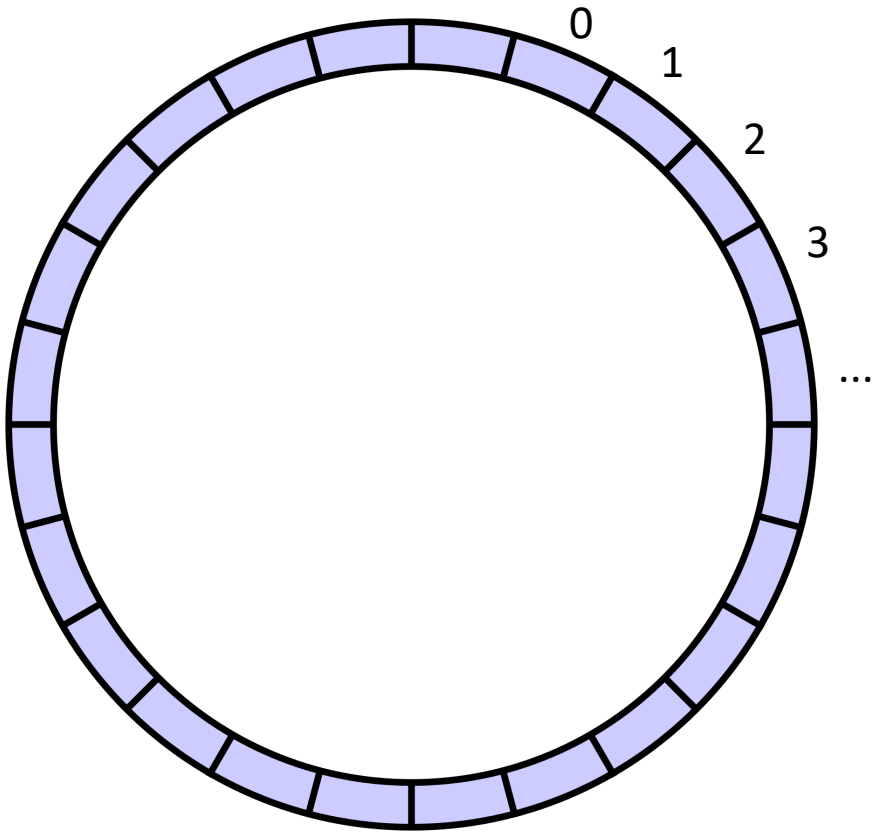
- Start with a fixed size array



conceptually it is a circle

Producer Consumer Queues

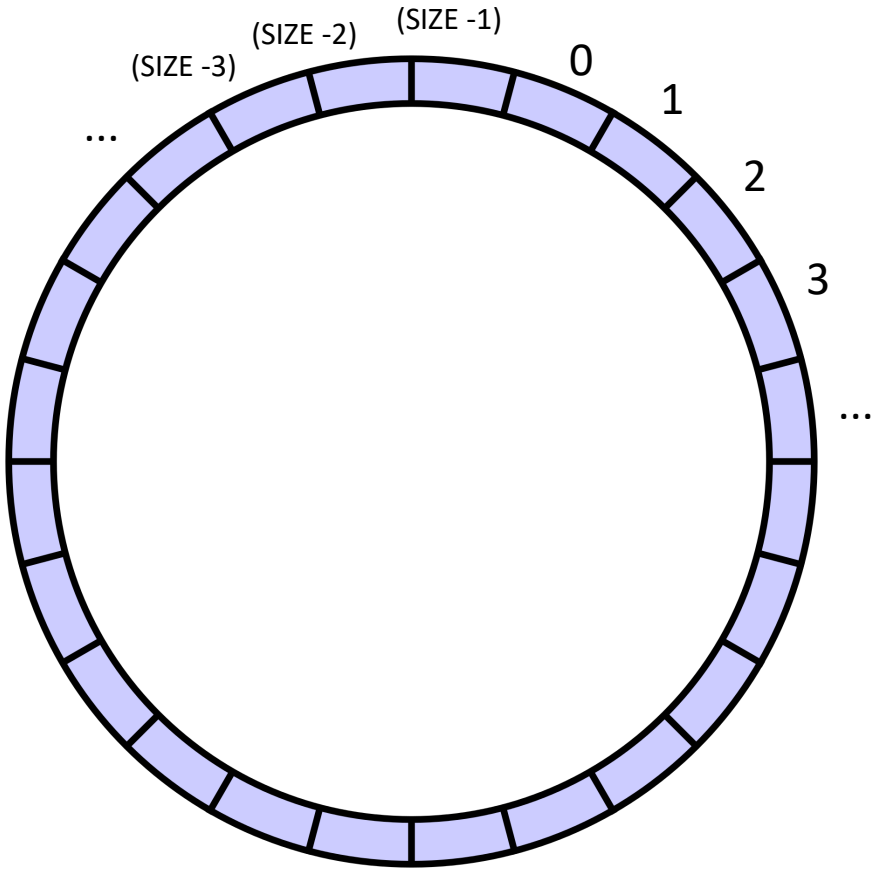
- Start with a fixed size array



conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array

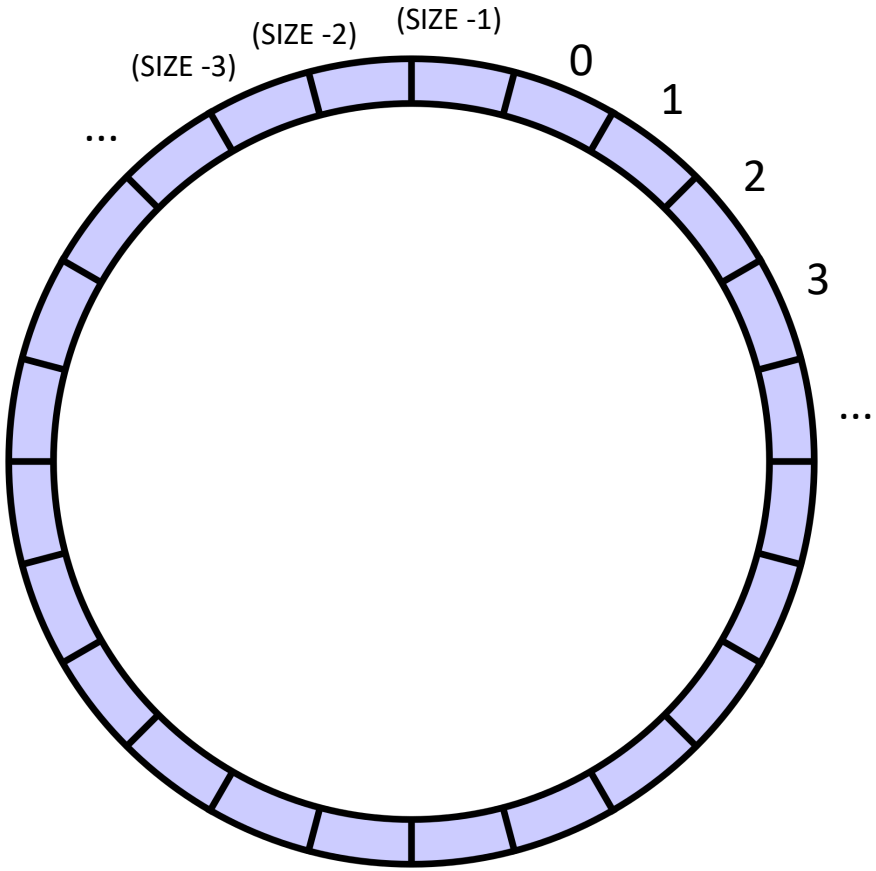


indexes will circulate in order and wrap around

conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array



indexes will circulate in order and wrap around

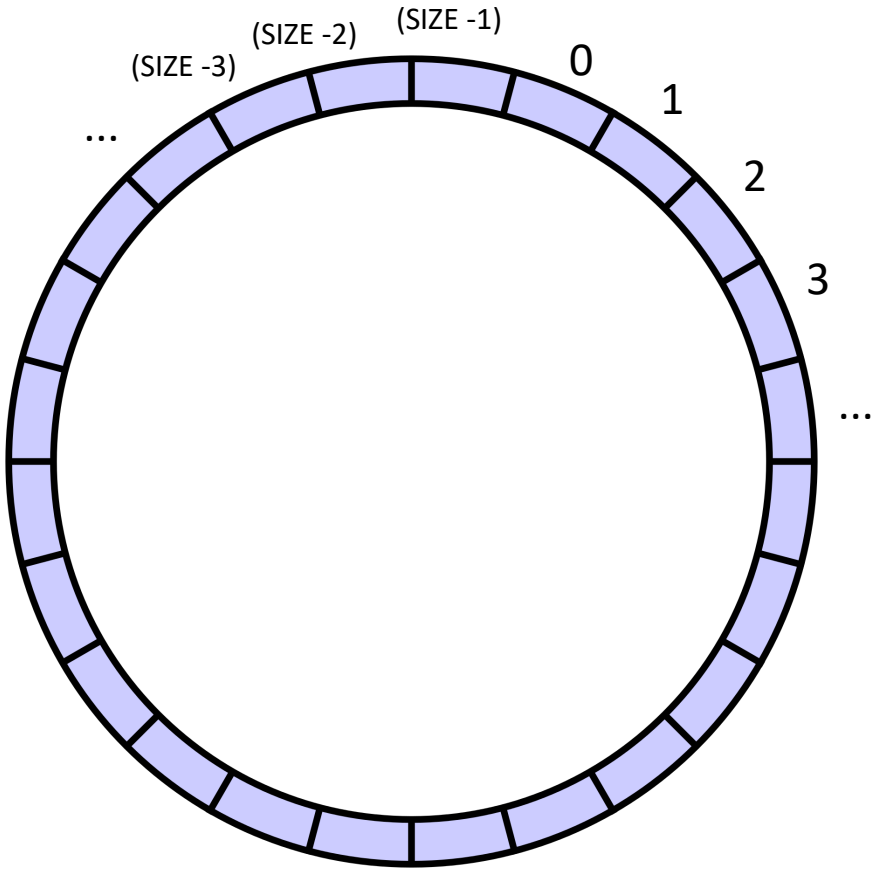
we will assume modular arithmetic:

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

conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array



indexes will circulate in order and wrap around

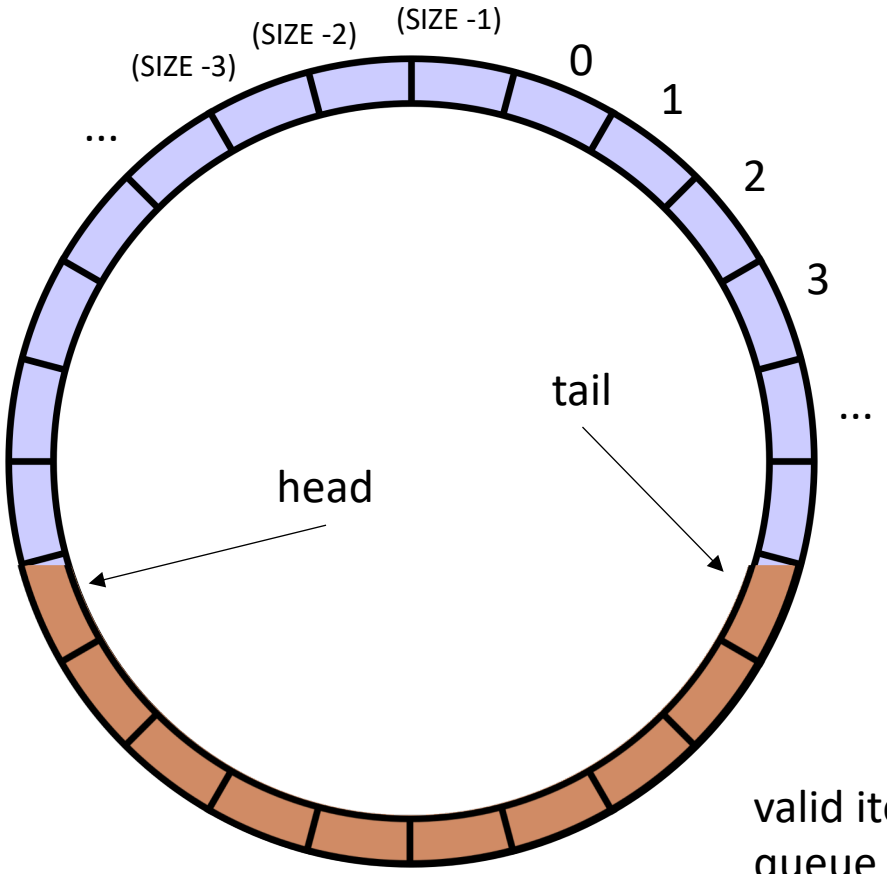
Two variables to keep track of where to deq and enq:

head and tail

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:

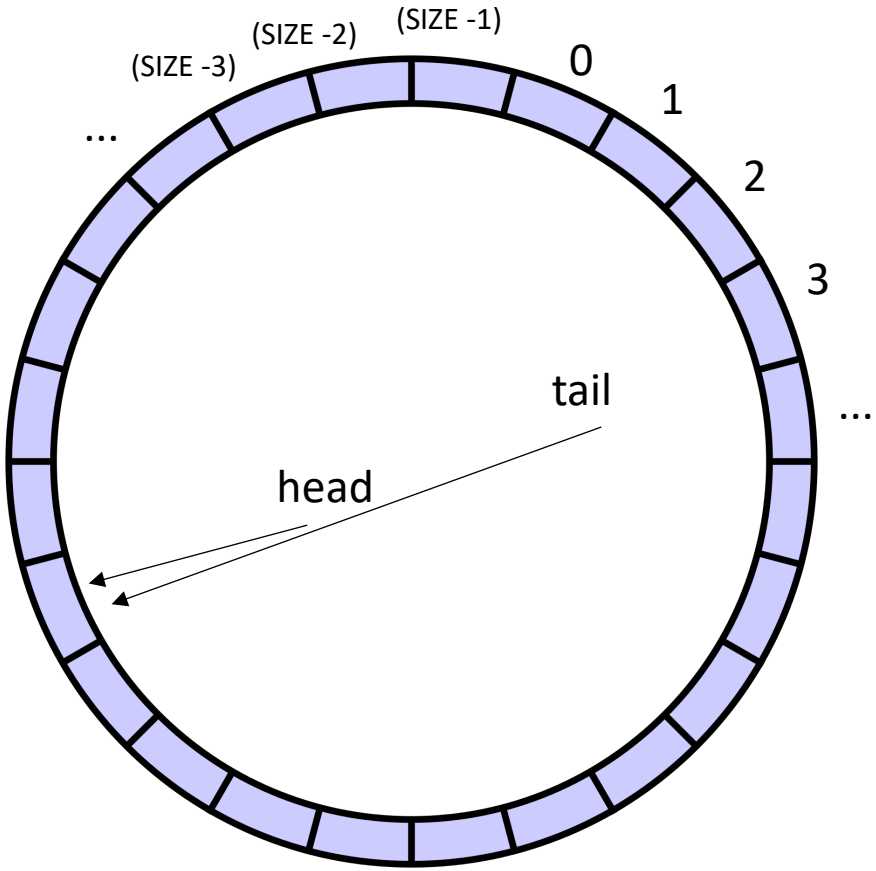
enq to the head, deq from the tail

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:

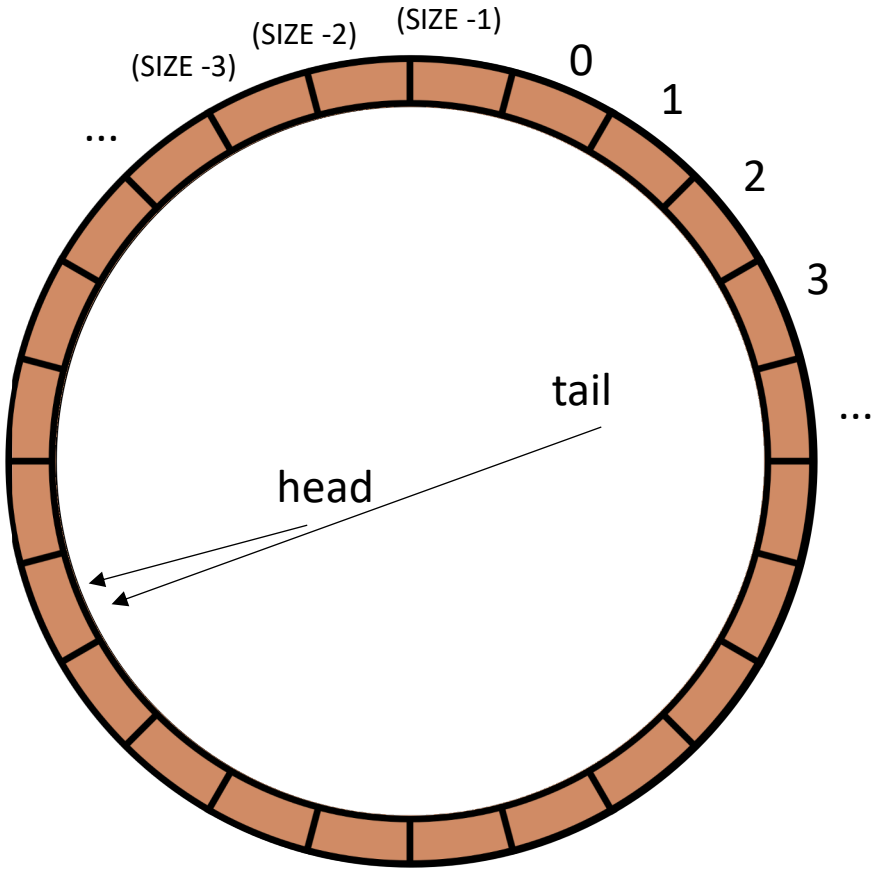
head and tail

Empty queue is when head == tail

conceptually it is a circle

Producer Consumer Queues

- Start with a fixed size array



indexes will circulate in order and wrap around

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

head and tail

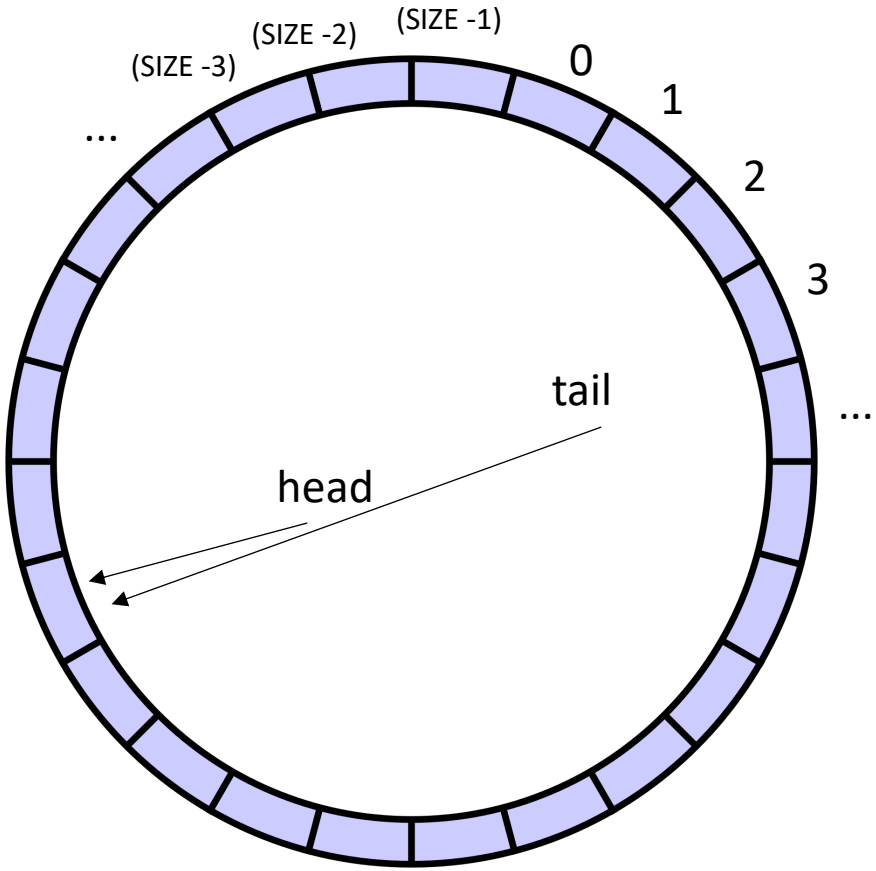
Empty queue is when
head == tail

Full queue is when
head == tail?

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

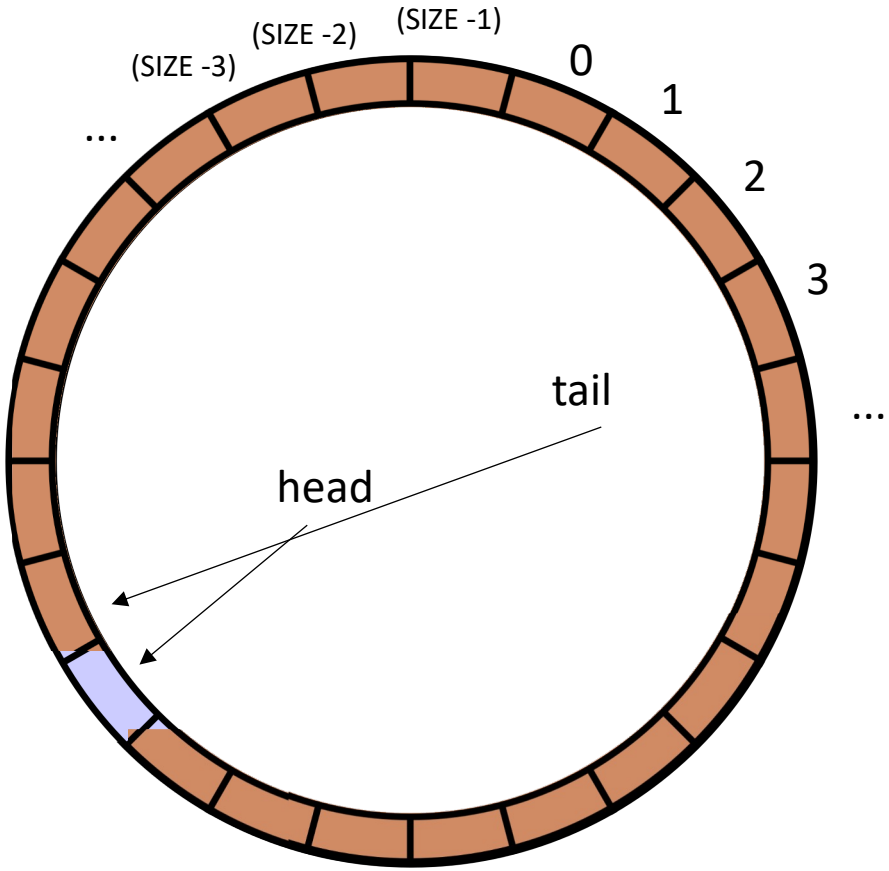
Full queue is when
head == tail?

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:

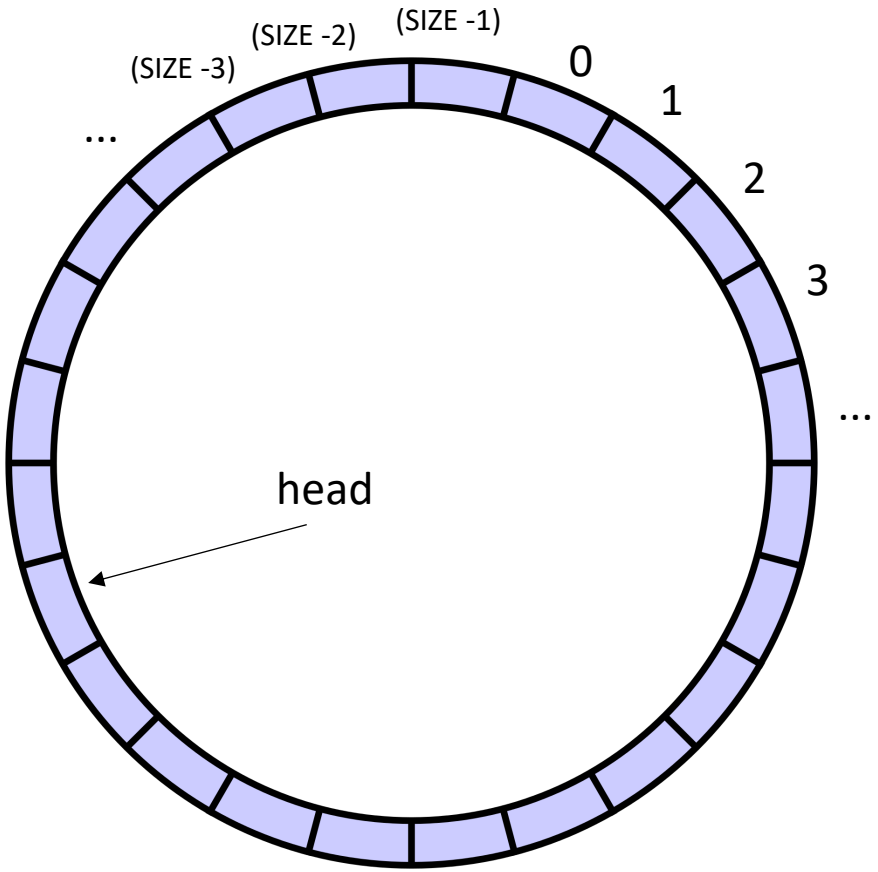
head and tail

Empty queue is when
head == tail

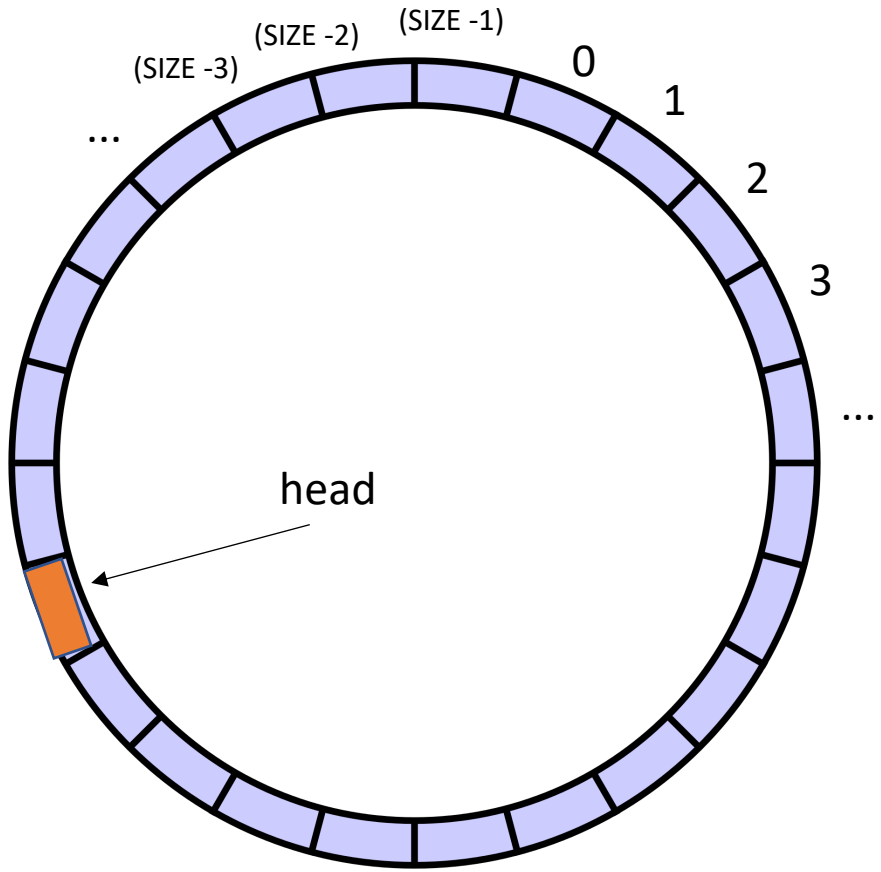
Full queue is when
head + 1 == tail

wasting one location, but its okay...

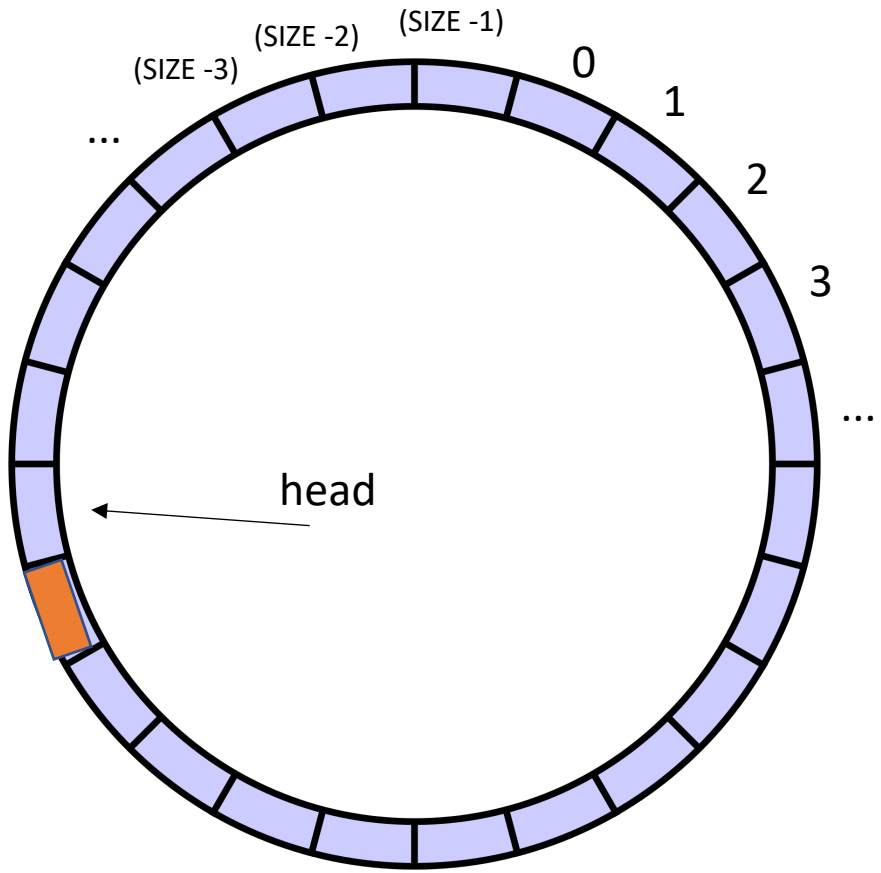
conceptually it is a circle



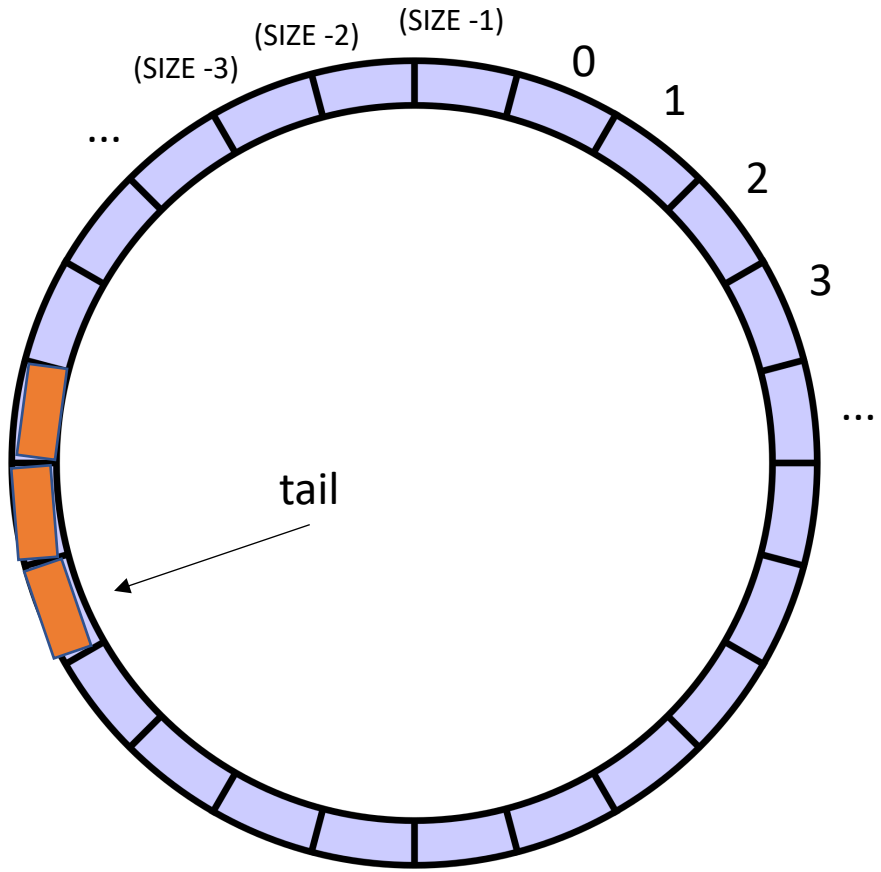

```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

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



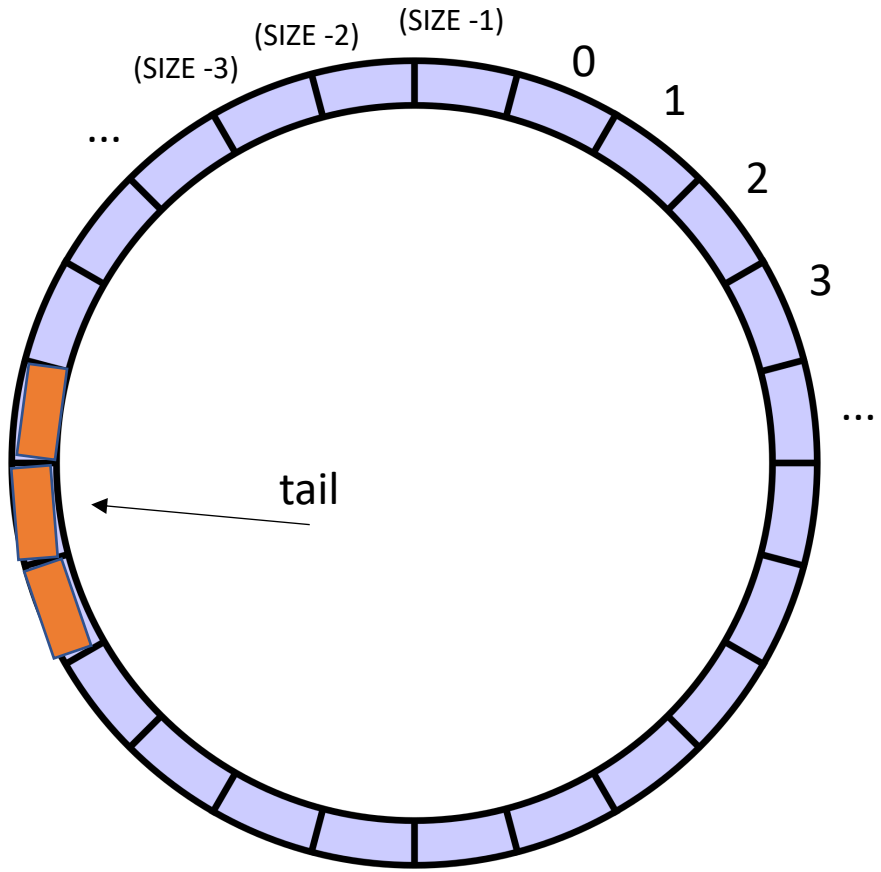

```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  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
    }
}
```

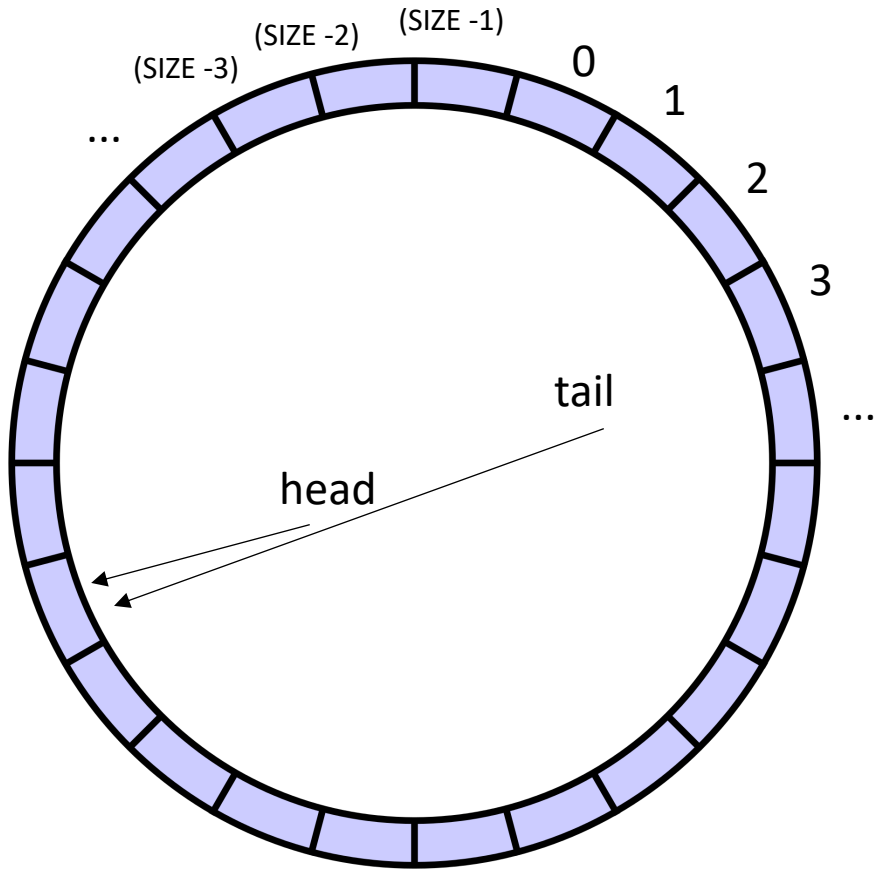



```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

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



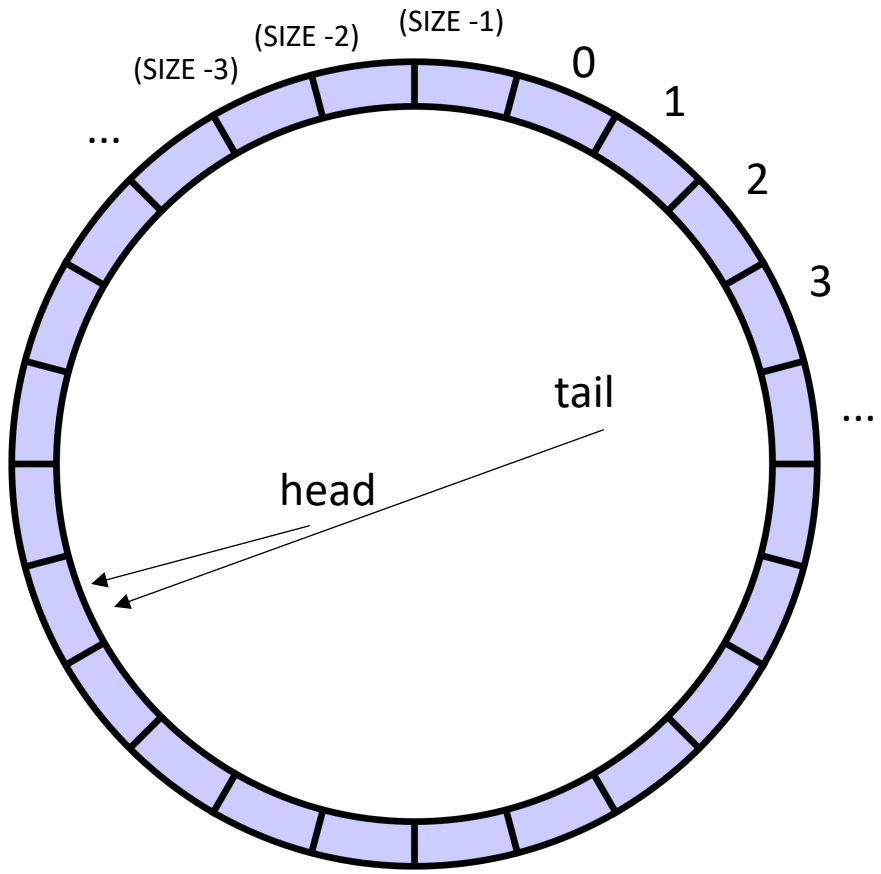

```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  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?

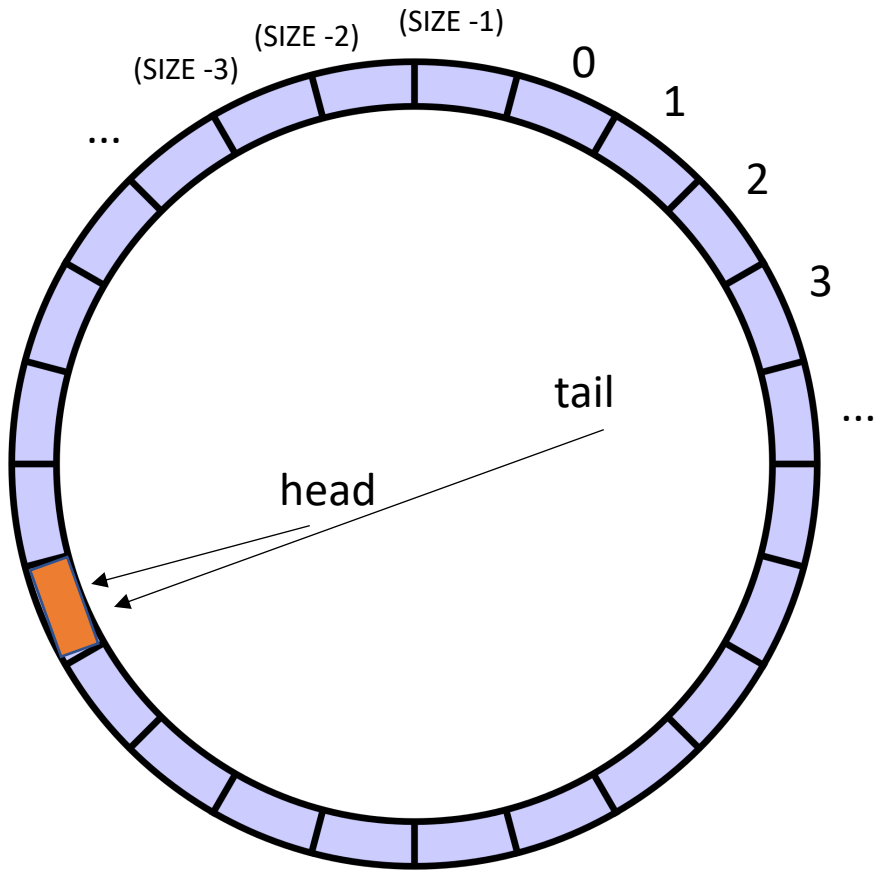



```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

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

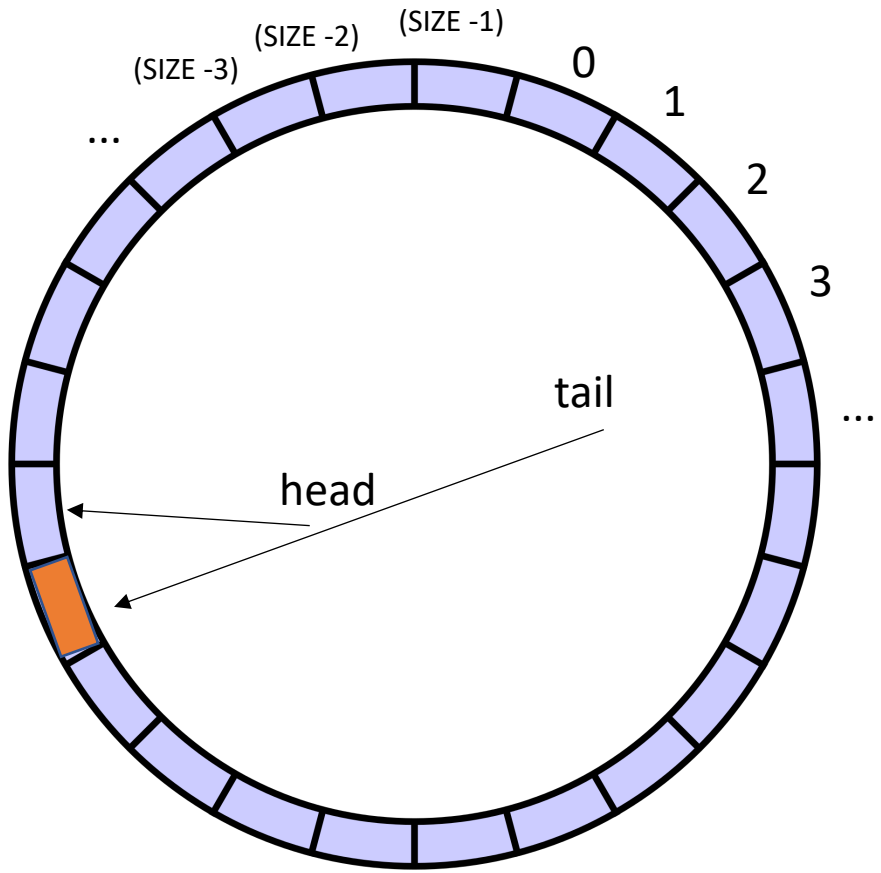
what happens if we try to dequeue here?



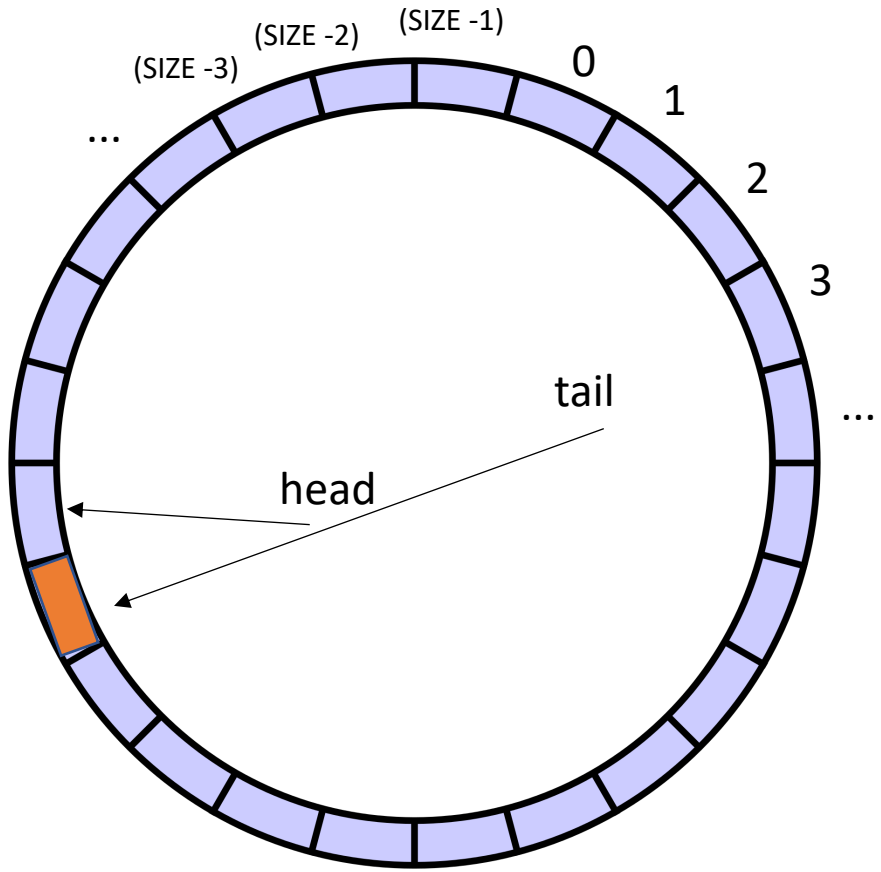

```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

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



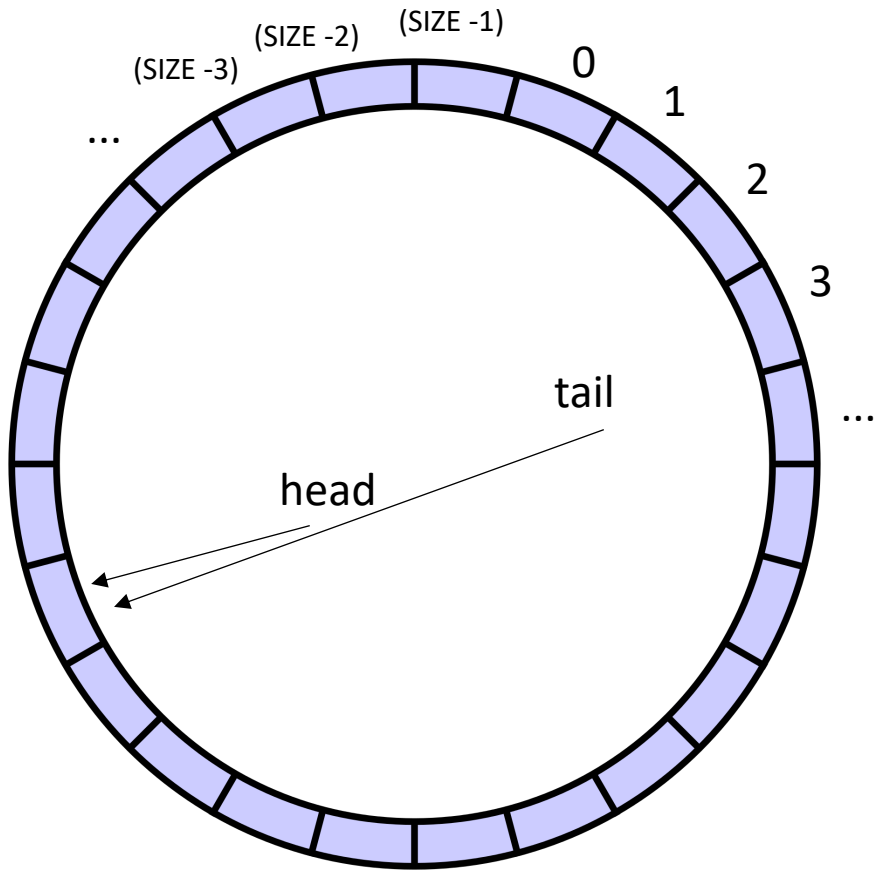

```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

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



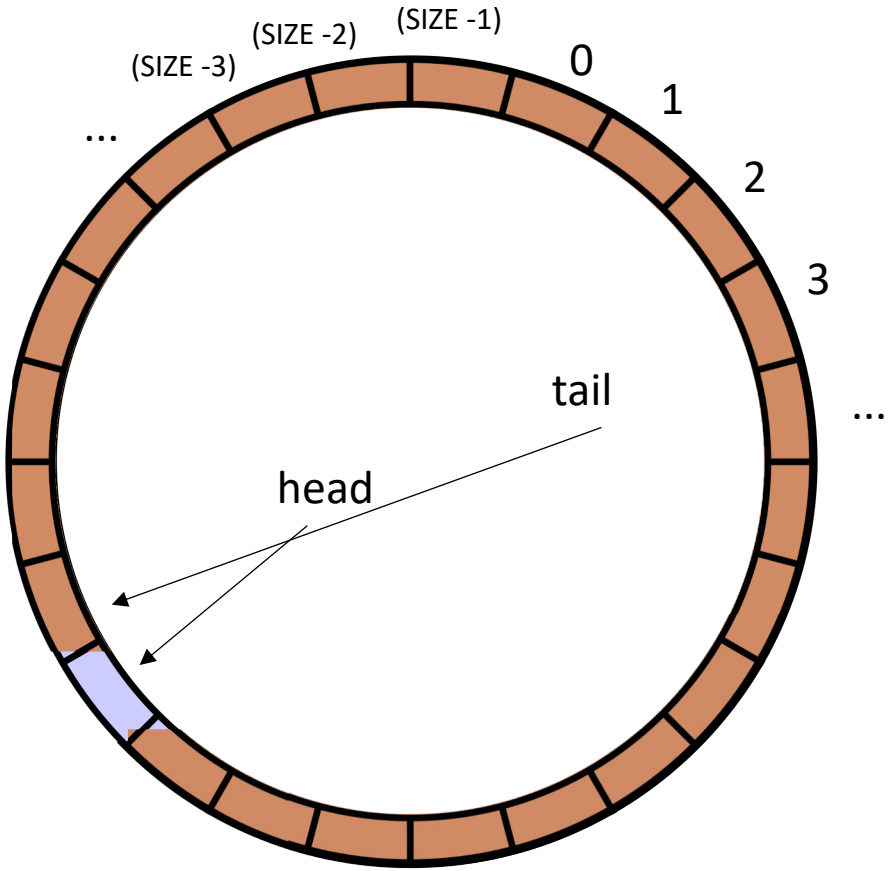

```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  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
    }
}
```



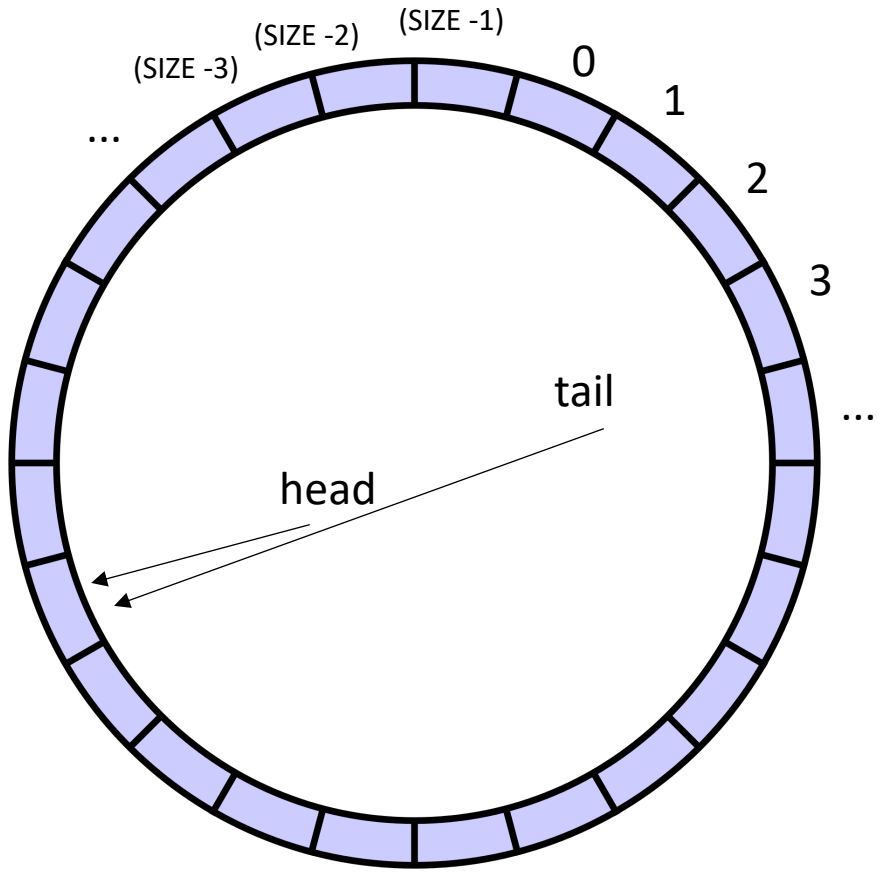

```
class ProdConsQueue {
  private:
    atomic_int head;
    atomic_int tail;
    int buffer[SIZE];

  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
    }
}
```

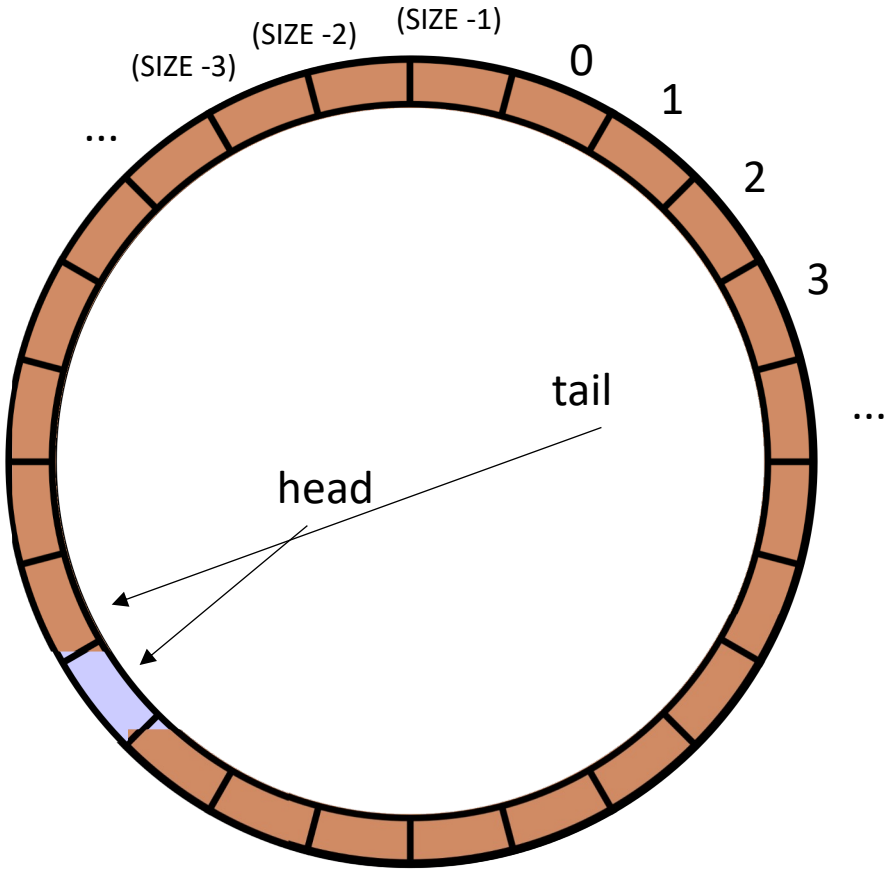
similarly for enqueue

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
    }
}
```

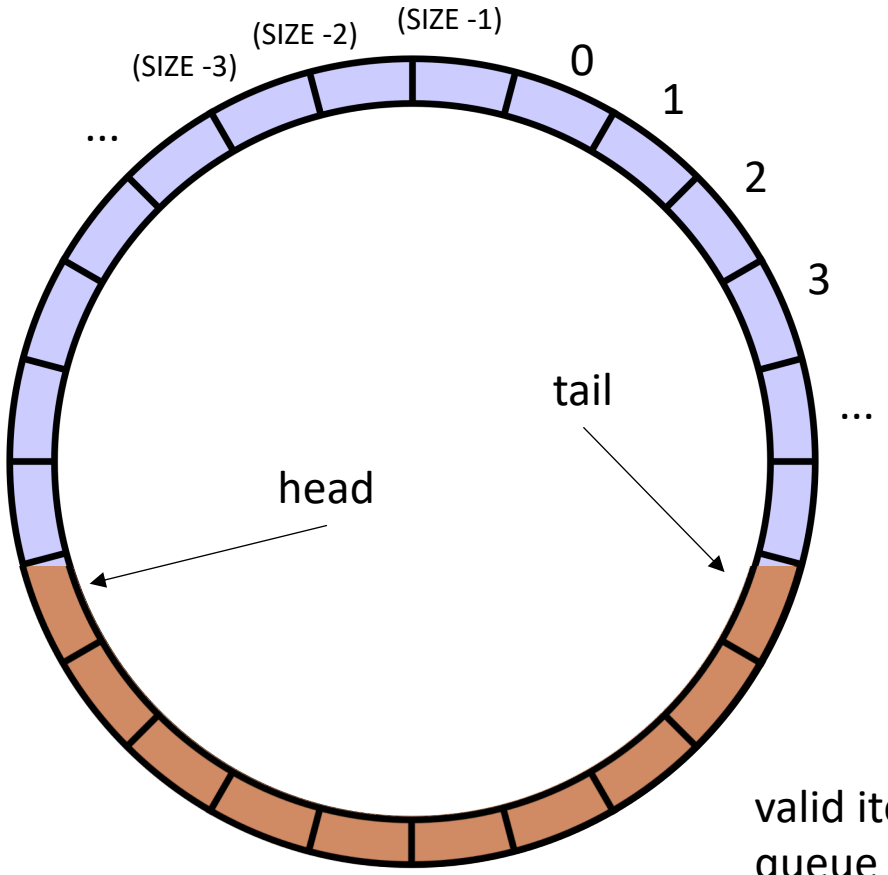
incrementing the head would make it empty!




```
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
    }
}
```

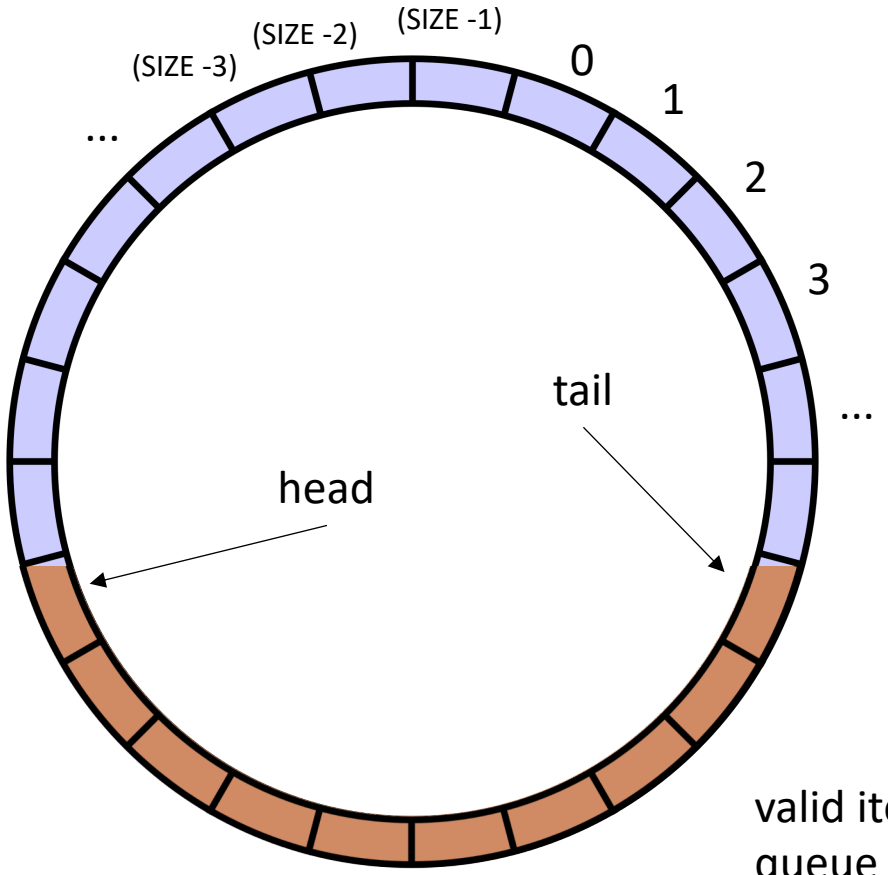
we need to wait for there to be room

Other questions:



valid items in the queue

```
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?

```
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
    }
}
```

Next topic

- Work stealing

adds two arrays

```
for (int i = 0; i < SIZE; i++) {
  a[i] = b[i] + c[i];
}
```

adds elements with neighbors

```
for (int i = 0; i < SIZE; i++) {
  a[i] += a[i+1]
}
```

are they the same if you traverse them backwards?

adds two arrays

```
for (int i = 0; i < SIZE; i++) {
  a[i] = b[i] + c[i];
}
```

```
for (int i = SIZE-1; i >= 0; i--) {
  a[i] = b[i] + c[i];
}
```

adds elements with neighbors

```
for (int i = 0; i < SIZE; i++) {
  a[i] += a[i+1]
}
```

```
for (int i = SIZE-1; i >= 0; i--) {
  a[i] += a[i+1]
}
```

are they the same if you traverse them backwards?

adds two arrays

```
for (int i = 0; i < SIZE; i++) {
  a[i] = b[i] + c[i];
}
```

```
for (int i = SIZE-1; i >= 0; i--) {
  a[i] = b[i] + c[i];
}
```

adds elements with neighbors

```
for (int i = 0; i < SIZE; i++) {
  a[i] += a[i+1]
}
```

```
for (int i = SIZE-1; i >= 0; i--) {
  a[i] += a[i+1]
}
```

No!

adds two arrays

```
for (int i = 0; i < SIZE; i++) {
  a[i] = b[i] + c[i];
}
```

what about a random order?

```
for (pick i randomly) {
  a[i] = b[i] + c[i];
}
```

adds elements with neighbors

```
for (int i = 0; i < SIZE; i++) {
  a[i] += a[i+1]
}
```

```
for (pick i randomly) {
  a[i] += a[i+1]
}
```

adds two arrays

```
for (int i = 0; i < SIZE; i++) {
  a[i] = b[i] + c[i];
}
```

adds elements with neighbors

```
for (int i = 0; i < SIZE; i++) {
  a[i] += a[i+1]
}
```

what about a random order?

```
for (pick i randomly) {
  a[i] = b[i] + c[i];
}
```

```
for (pick i randomly) {
  a[i] += a[i+1]
}
```

No!

```
for (int i = 0; i < SIZE; i++) {
  a[i] = b[i] + c[i];
}
```

These are **DOALL** loops:

- Loop iterations are independent
- You can do them in ANY order and get the same results

```
for (int i = 0; i < SIZE; i++) {
  a[i] = b[i] + c[i];
}
```

These are **DOALL** loops:

- Loop iterations are independent
- You can do them in ANY order and get the same results

- Most importantly: you can do the iterations in parallel!
- Assign each thread a set of indices to compute

DOALL Loops

- Given a nest of For loops, can we make the outer-most loop parallel?
  - Safely
  - Efficiently

DOALL Loops

- We will consider a special type of for loop, common in scientific applications:
  - Operates on N dimensional arrays (only side-effects are array writes)
  - Array bases are disjoint and constant
  - Bounds, indexes are a function of loop variables, input variables and constants
  - Loops Increment by 1

```
for (int i = 0; i < dim1; i++) {
  for (int j = 0; j < dim3; j++) {
    for (int k = 0; k < dim2; k++) {
      a[i][j] += b[i][k] * c[k][j];
    }
  }
}
```

matrix multiplication example

DOALL Loops

- We will consider a special type of for loop, common in scientific applications:
  - Operates on N dimensional arrays (only side-effects are array writes)
  - Array bases are disjoint and constant
  - Bounds, indexes are a function of loop variables, input variables and constants
  - Loops Increment by 1

DOALL Loops

- Given a nest of ***candidate*** *For* loops, determine if we can we make the outer-most loop parallel?
  - Safely
  - efficiently

- Criteria: every iteration of the outer-most loop must be *independent*
  - The loop can execute in any order, and produce the same result

Safety Criteria

- How do we check this?
  - If the property doesn't hold then there exists 2 iterations, such that if they are re-ordered, it causes different outcomes for the loop.
  - **Write-Write conflicts**: two distinct iterations write different values to the same location
  - **Read-Write conflicts**: two distinct iterations where one iteration reads from the location written to by another iteration.

Safety Criteria

- Criteria: every iteration of the outer-most loop must be *independent*
- the loop must produce the same result for any order of the iterations

```
for (i = 0; i < size; i++) {
   a[index(i)] = loop(i);
}
```

Safety Criteria

- Criteria: every iteration of the outer-most loop must be *independent*
- the loop must produce the same result for any order of the iterations

```
for (i = 0; i < size; i++) {
   a[index(i)] = loop(i);
}
```

index calculation based on the loop variable

Safety Criteria

- Criteria: every iteration of the outer-most loop must be *independent*
- the loop must produce the same result for any order of the iterations

```
for (i = 0; i < size; i++) {
   a[index(i)] = loop(i);
}
```

index calculation based on the loop variable
Computation to store in the memory location

Safety Criteria

- Criteria: every iteration of the outer-most loop must be *independent*
- the loop must produce the same result for any order of the iterations

```
for (i = 0; i < size; i++) {
   a[index(i)] = loop(i);
}
```

Write-write conflicts:

for two distinct iteration variables:

i_x != i_y

Check:

index(i_x) != index(i_y)

Safety Criteria

- Criteria: every iteration of the outer-most loop must be *independent*
- the loop must produce the same result for any order of the iterations

```
for (i = 0; i < size; i++) {
   a[index(i)] = loop(i);
}
```

Write-write conflicts:

for two distinct iteration variables:
i_x != i_y
Check:
index(i_x) != index(i_y)

Why?
Because if
index(i_x) == index(i_y)
then:
a[index(i_x)] will equal
either loop(i_x) or loop(i_y)
depending on the order

Safety Criteria

- Criteria: every iteration of the outer-most loop must be *independent*

```
for (i = 0; i < size; i++) {
    a[write_index(i)] = a[read_index(i)] + loop(i);
}
```

Read-write conflicts:

for two distinct iteration variables:

i_x != i_y

Check:

write_index(i_x) != read_index(i_y)

Safety Criteria

- Criteria: every iteration of the outer-most loop must be *independent*

```
for (i = 0; i < size; i++) {
    a[write_index(i)] = a[read_index(i)] + loop(i);
}
```

Read-write conflicts:

for two distinct iteration variables:
i_x != i_y
Check:
write_index(i_x) != read_index(i_y)

Why?

if i_x iteration happens first, then iteration i_y reads an updated value.

if i_y happens first, then it reads the original value

Examples:

```
for (i = 0; i < 128; i++) {
   a[i]= a[i]*2;
}
```

Examples:

```
for (i = 0; i < 128; i++) {
   a[i]= a[i]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i]= a[0]*2;
}
```

Examples:

```
for (i = 0; i < 128; i++) {
   a[i]= a[i]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i]= a[0]*2;
}
```

```
for (i = 1; i < 128; i++) {
   a[i]= a[0]*2;
}
```

Examples:

```
for (i = 0; i < 128; i++) {
   a[i]= a[i]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i]= a[0]*2;
}
```

```
for (i = 1; i < 128; i++) {
   a[i]= a[0]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i%64]= a[i]*2;
}
```

Examples:

```
for (i = 0; i < 128; i++) {
   a[i]= a[i]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i]= a[0]*2;
}
```

```
for (i = 1; i < 128; i++) {
   a[i]= a[0]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i%64]= a[i]*2;
}
```

```
for (i = 0; i < 128; i++) {
   a[i%64]= a[i+64]*2;
}
```

Parallel Schedules

- Consider the following program:

There are 3 arrays: `a, b, c.`

We want to compute

```
for (int i = 0; i < SIZE; i++) {
  c[i] = a[i] + b[i];
}
```

Is this a DOALL loop?

Parallel Schedules

- Consider the following program:

There are 3 arrays: `a, b, c.`

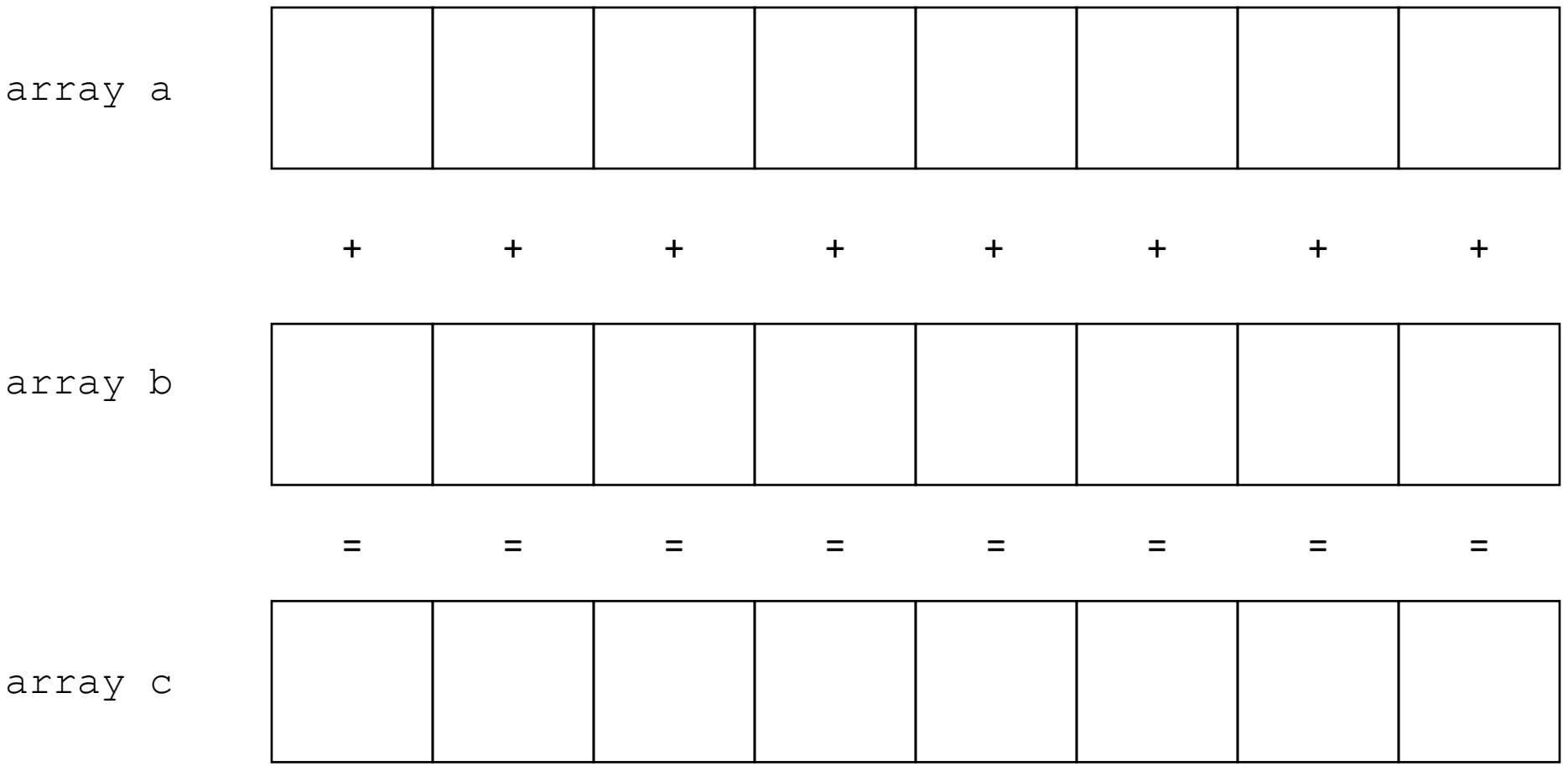
We want to compute

```
for (int i = 0; i < SIZE; i++) {
  c[i] = a[i] + b[i];
}
```

Is this a DOALL loop?

How should we parallelize it?

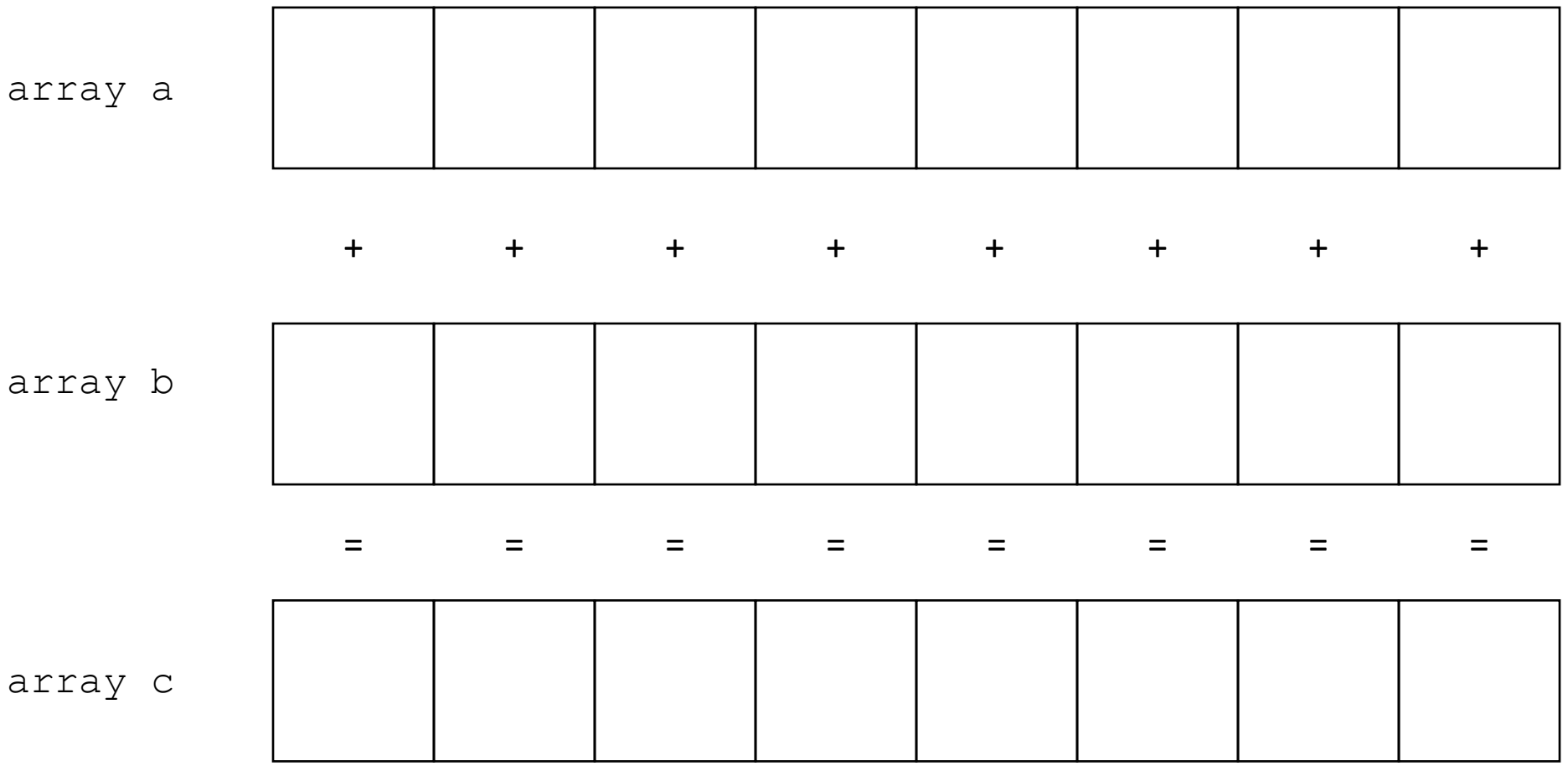
Parallel Schedules



Parallel Schedules

Computation can easily be divided into threads

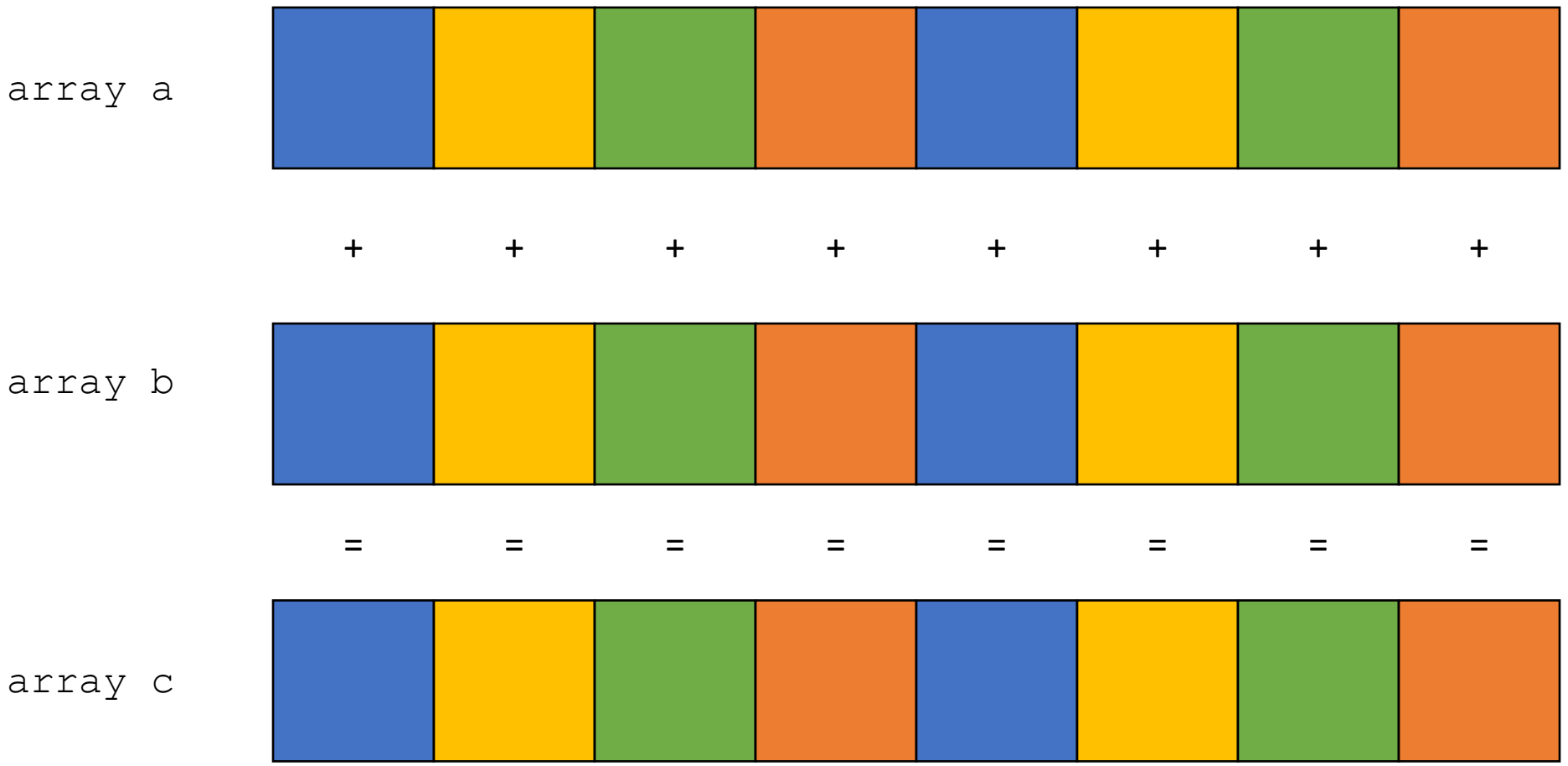
Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



Parallel Schedules

Computation can easily be divided into threads

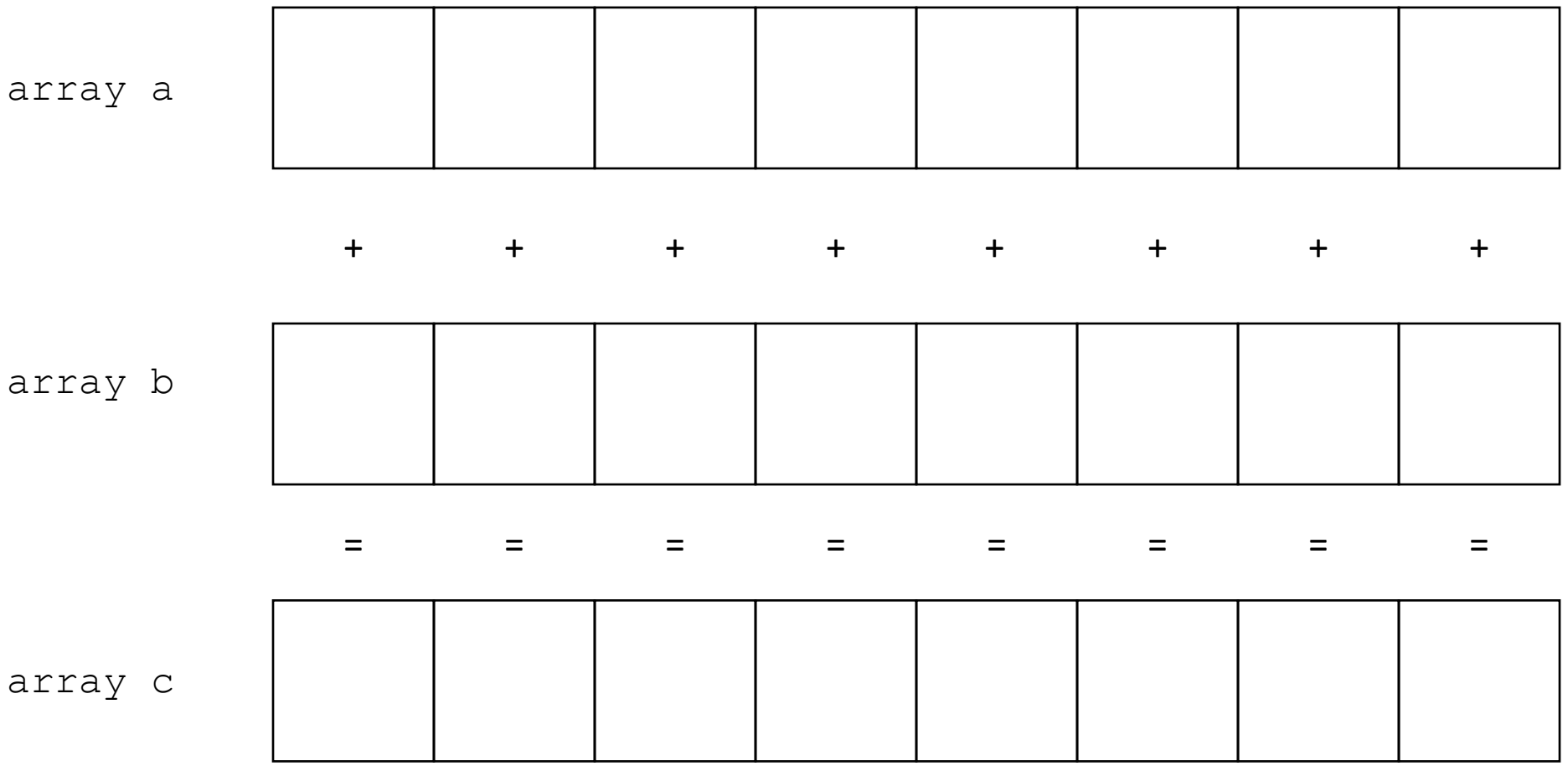
Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



Parallel Schedules

Computation can easily be divided into threads

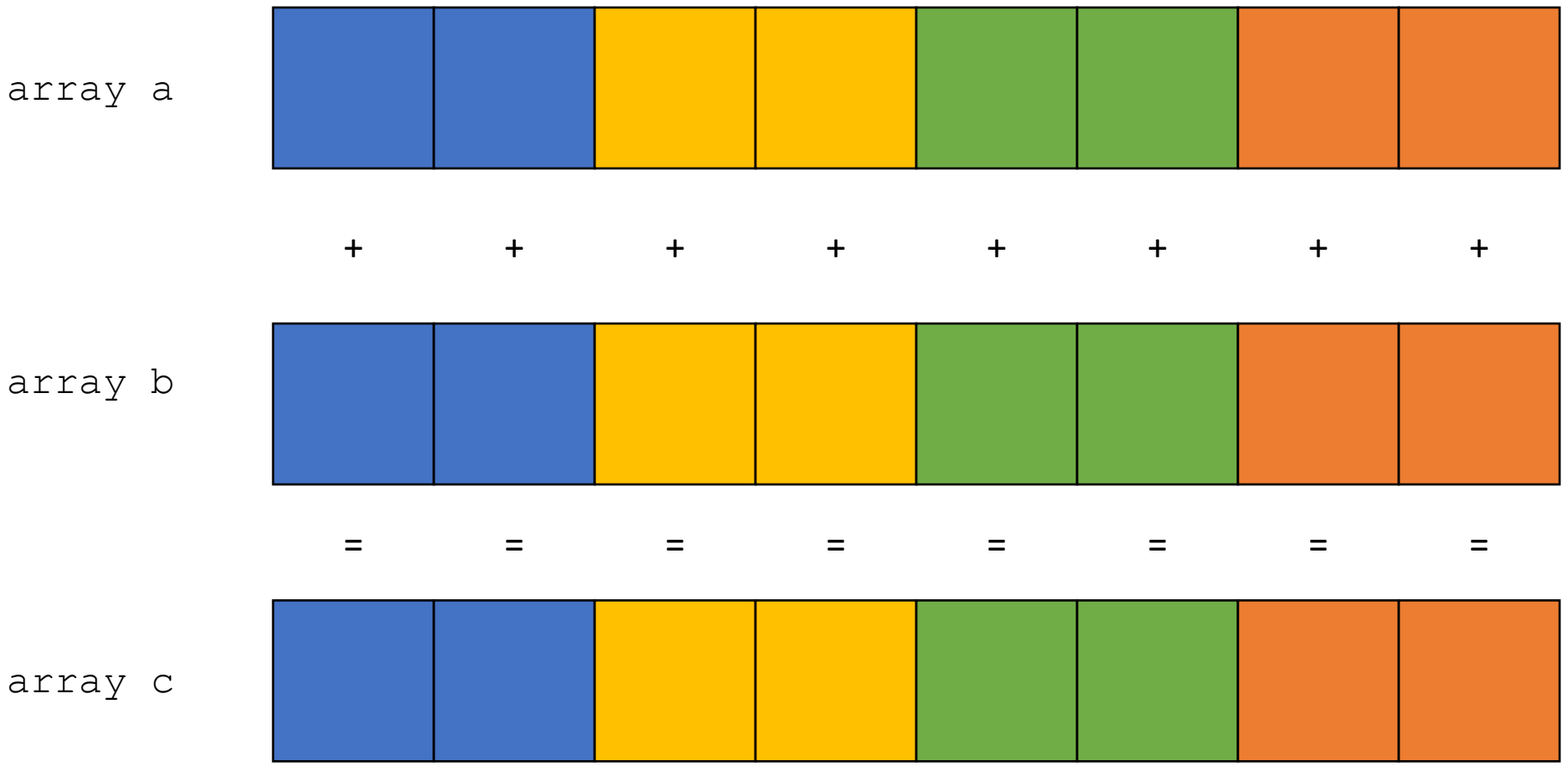
Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



Parallel Schedules

Computation can easily be divided into threads

Thread 0 - Blue
Thread 1 - Yellow
Thread 2 - Green
Thread 3 - Orange



Parallel Schedules

- Which one is more efficient?

Parallel Schedules

- Which one is more efficient?

- These are called Parallel Schedules for DOALL Loops
- We will discuss several of them.

Schedule

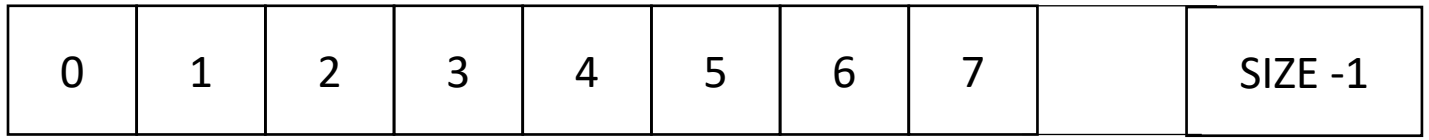
- DOALL Loops

- **Parallel Schedules:**
  - **Static**
  - Global Worklists
  - Local Worklists

Static schedule

- Works well when loop iterations take similar amounts of time

```
void foo() {
...
  for (int x = 0; x < SIZE; x++) {
  // Each iteration takes roughly
  // equal time
  }
...
}
```

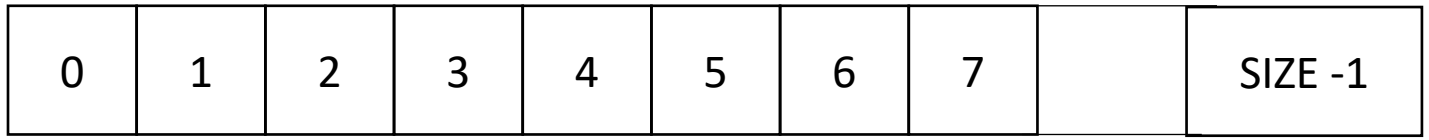


Static schedule

- Works well when loop iterations take similar amounts of time

```
void foo() {
...
  for (int x = 0; x < SIZE; x++) {
  // Each iteration takes roughly
  // equal time
  }
...
}
```

say SIZE / NUM_THREADS = 4

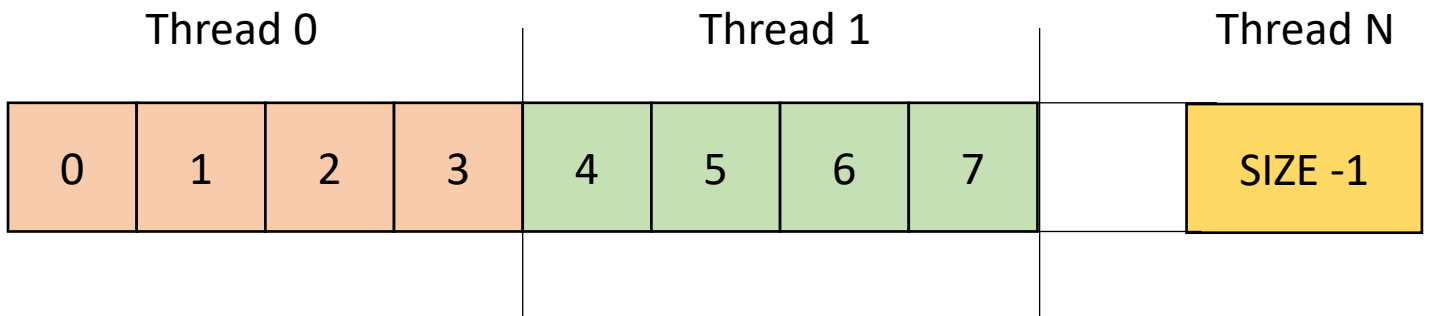


Static schedule

- Works well when loop iterations take similar amounts of time

```
void foo() {
...
  for (int x = 0; x < SIZE; x++) {
  // Each iteration takes roughly
  // equal time
  }
...
}
```

say SIZE / NUM_THREADS = 4



Static schedule

- Works well when loop iterations take similar amounts of time

```
void foo() {
...
  for (int x = 0; x < SIZE; x++) {
  // Each iteration takes roughly
  // equal time
  }
...
}
```

make a new function with the for loop inside. Pass all needed variables as arguments. Take an extra argument for a thread id

Static schedule

- Works well when loop iterations take similar amounts of time

```
void foo() {
...
  for (int x = 0; x < SIZE; x++) {
  // Each iteration takes roughly
  // equal time
  }
...
}
```

```
void parallel_loop(..., int tid, int num_threads)
{

  for (int x = 0; x < SIZE; x++) {
    // work based on x
  }
}
```

make a new function with the for loop inside. Pass all needed variables as arguments. Take an extra argument for a thread id

Static schedule

- Works well when loop iterations take similar amounts of time

```
void foo() {
...
  for (int x = 0; x < SIZE; x++) {
  // Each iteration takes roughly
  // equal time
  }
...
}
```

```
void parallel_loop(..., int tid, int num_threads)
{

  int chunk_size = SIZE / NUM_THREADS;
  for (int x = 0; x < SIZE; x++) {
    // work based on x
  }
}
```

determine chunk size in new function

Static schedule

- Works well when loop iterations take similar amounts of time

```
void foo() {
...
  for (int x = 0; x < SIZE; x++) {
  // Each iteration takes roughly
  // equal time
  }
...
}
```

```
void parallel_loop(..., int tid, int num_threads)
{

  int chunk_size = SIZE / NUM_THREADS;
  int start = chunk_size * tid;
  int end = start + chunk_size;
  for (int x = start; x < end; x++) {
    // work based on x
  }
}
```

Set new loop bounds

Static schedule

- Works well when loop iterations take similar amounts of time

```
void foo() {
...
  for (int t = 0; t < NUM_THREADS; t++) {
    spawn(parallel_loop(..., t, NUM_THREADS))
  }
  join();
...
}
```

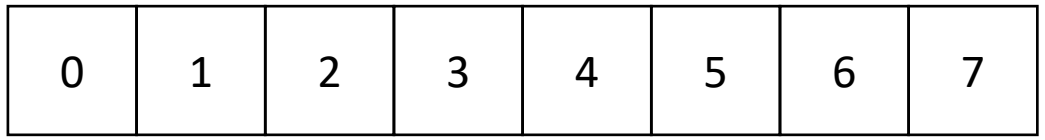
```
void parallel_loop(..., int tid, int num_threads)
{

  int chunk_size = SIZE / NUM_THREADS;
  int start = chunk_size * tid;
  int end = start + chunk_size;
  for (int x = start; x < end; x++) {
    // work based on x
  }
}
```

You will need to adapt the thread spawn, join to C++

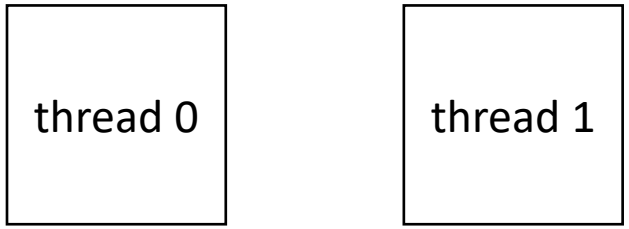
Spawn threads

Static schedule

- Example, 2 threads/cores, array of size 8



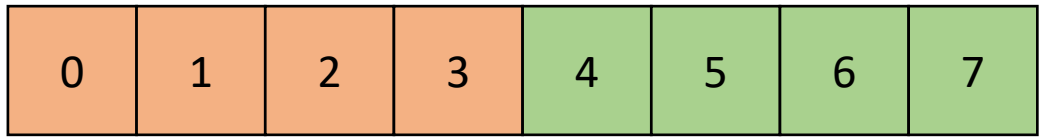

```
chunk_size = ?

0: start = ?    1: start = ?
0: end = ?      1: end = ?
```




```
void parallel_loop(..., int tid, int num_threads)
{

  int chunk_size = SIZE / NUM_THREADS;
  int start = chunk_size * tid;
  int end = start + chunk_size;
  for (int x = start; x < end; x++) {
    // work based on x
  }
}
```

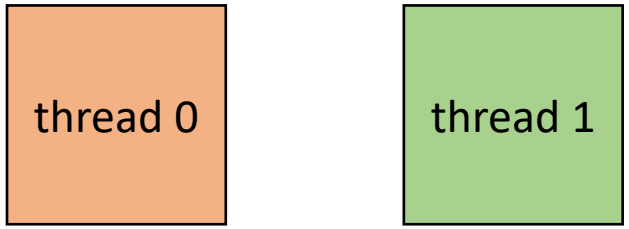
Static schedule

- Example, 2 threads/cores, array of size 8




```
chunk_size = 4
```

```
0: start = 0    1: start = 4
0: end = 4      1: end = 8
```




```
void parallel_loop(..., int tid, int num_threads)
{

  int chunk_size = SIZE / NUM_THREADS;
  int start = chunk_size * tid;
  int end = start + chunk_size;
  for (int x = start; x < end; x++) {
    // work based on x
  }
}
```