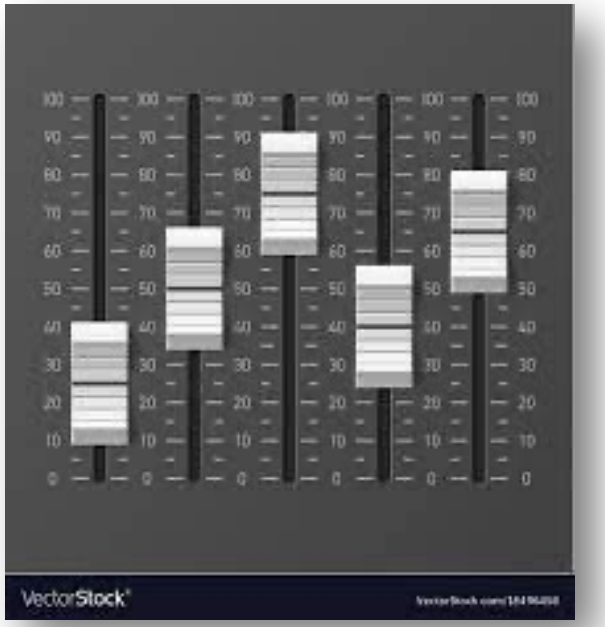
CSE113: Parallel Programming

Feb. 2, 2022

- **Topics**:
  - Concurrent data structure specifications
    - Breaking sequential consistency
    - Linearizability
  - Input/output concurrent queues



Announcements

- Expect HW1 grades by Friday
  - Let us know if there are any issues ASAP
  - Sanya is traveling so there may be some delay

- Homework 2 is due on Friday
  - Please use office hours or piazza if you have questions
  - Remember, nights and weekends have no guarantees of responses. Get started ASAP!
  - I have office hours tomorrow. They are hybrid; please indicate on the sign up sheet if you are in person or remote.
  - Reese has office hours after class today; Tim has office hours on Thursday. Sanya has them on Friday

Announcements

- Homework 3 assigned on Friday by midnight
  - Should have material to get started by end of Friday lecture

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

Homework clarifications

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

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

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

Homework clarifications

- You can share results, but not code

Today’s Quiz

- Due Tomorrow by midnight

Previous quiz

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

- ◯ True
- ◯ False

Previous quiz

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

Previous quiz

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

Previous quiz

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.

Review

Concurrent Data Structures

Shared memory concurrent objects

```
printf("hello world\n");
```

How does it actually work?

```
printf("h");
printf("e");
printf("l");
printf("l");
printf("o");
```

concurrent queue

```
terminal:
$ ./a.out
```

./a.out

terminal display

You can force a flush with: `fflush(stdout)`

Bank account example

global variables:

```
int tylers_account = 0;
```

Tyler's coffee addiction:

```
for (int i = 0; i < HOURS; i++) {
   tylers_account -= 1;
}
```

Tyler's employer

```
for (int j = 0; j < HOURS; j++) {
   tylers_account += 1;
}
```

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() {
      balance -= 1;
    }

    void get_paid() {
      balance += 1;
    }

  private:
    int balance;
};
```

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();
}
```

what happens if we run these concurrently?

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

    void buy_coffee() {
      balance -= 1;
    }

    void get_paid() {
      balance += 1;
    }

  private:
    int balance;
};
```

Non-thread-Safe Data Structures

global variables:

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

Tyler's coffee addiction:

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

Tyler's employer

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

```
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"

Thread-safe Data Structures

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

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

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

Non-locking Concurrent Data Stuctures

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;
};
```

Sequential Consistency

Global variable:

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

Thread 0:

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

Thread 1:

```
int t0 = q.deq();
int t1 = q.deq();
```


Can t0 == 0 and t1 == 6?

Global variable:

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

Thread 0:

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

Thread 1:

```
int t0 = q.deq();
int t1 = q.deq();
```

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

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

```
int t0 = q.deq();
```

```
int t1 = q.deq();
```

Can t0 == 0 and t1 == 6?

Global variable:

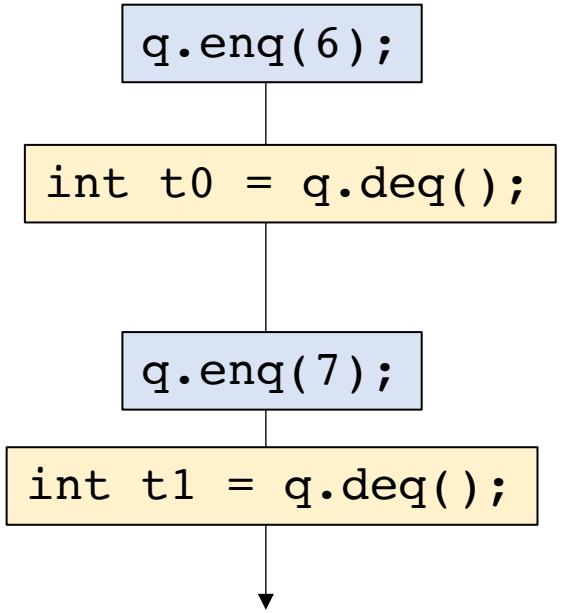
```
CQueue<int> q;
```

Thread 0:

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

Thread 1:

```
int t0 = q.deq();
int t1 = q.deq();
```



Can t0 == 0 and t1 == 6?

Valid execution!

Are there others?

Next example

Global variable:

```
CQueue<int> s;
```

Thread 0:

```
s.enq(7);
int t0 = q.dec();
```

Thread 1:

```
int t1 = q.dec();
```

Is it possible for both t0 and t1 to be 0 at the end?

Global variable:

```
CQueue<int> s;
```

Thread 0:

```
s.enq(7);
int t0 = q.dec();
```

Thread 1:

```
int t1 = q.dec();
```

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

```
int t0 = q.deq();
```

```
int t1 = q.dec();
```

Is it possible for both t0 and t1 to be 0 at the end?

Global variable:

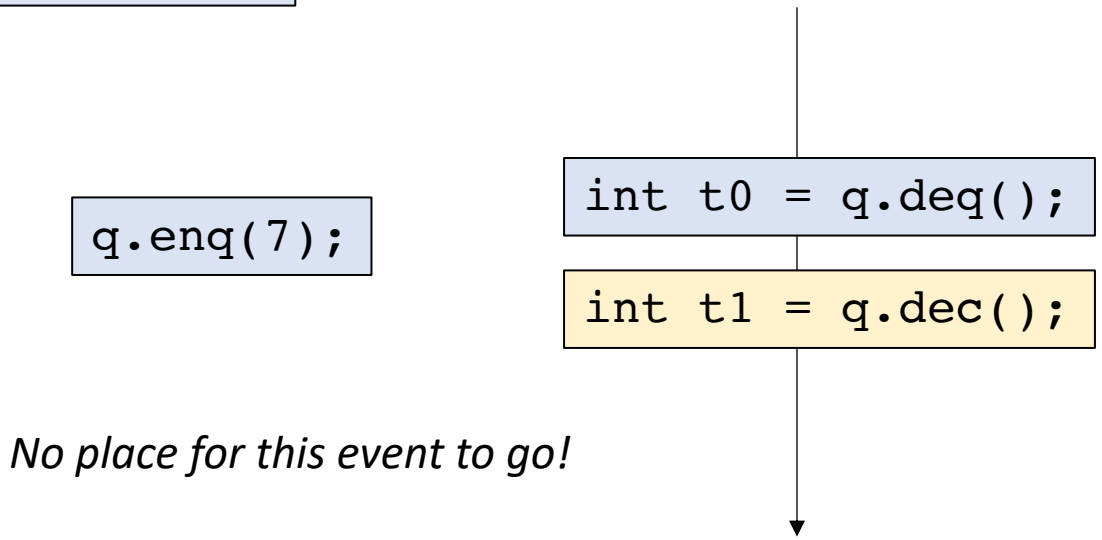
```
CQueue<int> s;
```

Thread 0:

```
s.enq(7);
int t0 = q.dec();
```

Thread 1:

```
int t1 = q.dec();
```



No place for this event to go!

Is it possible for both t0 and t1 to be 0 at the end?

New material!

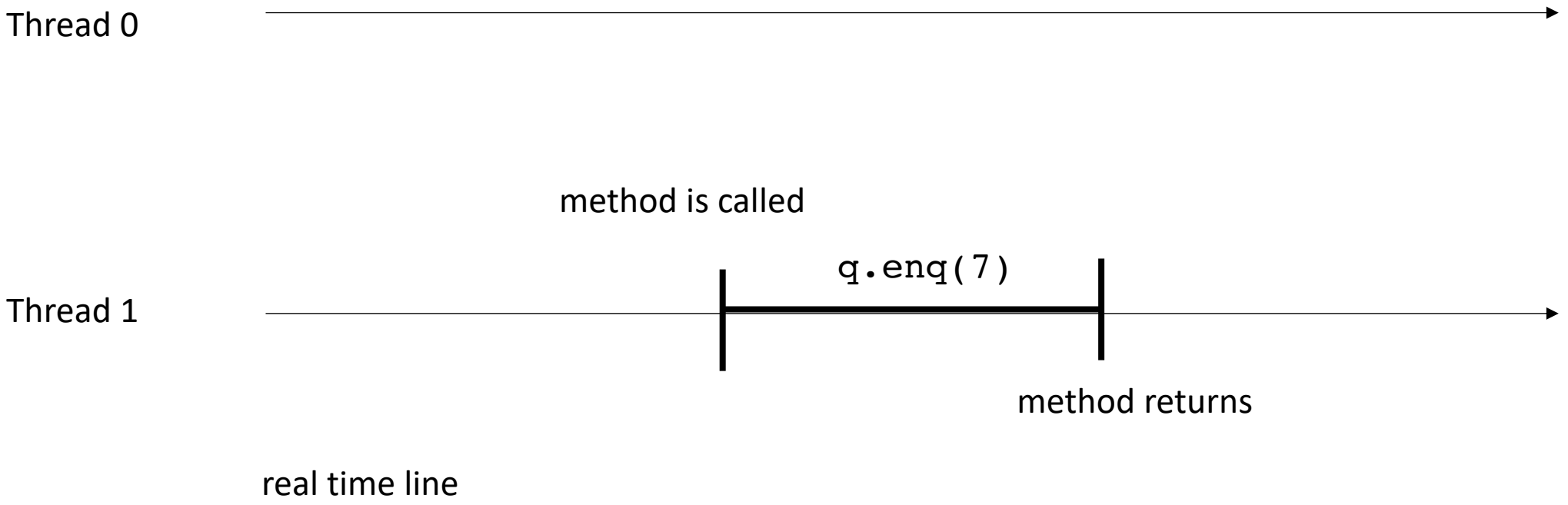
Schedule

- **Problems with sequential consistency**
- Linearizability
- Specialized concurrent queues

Sequential consistency and real time

- Add in real time:

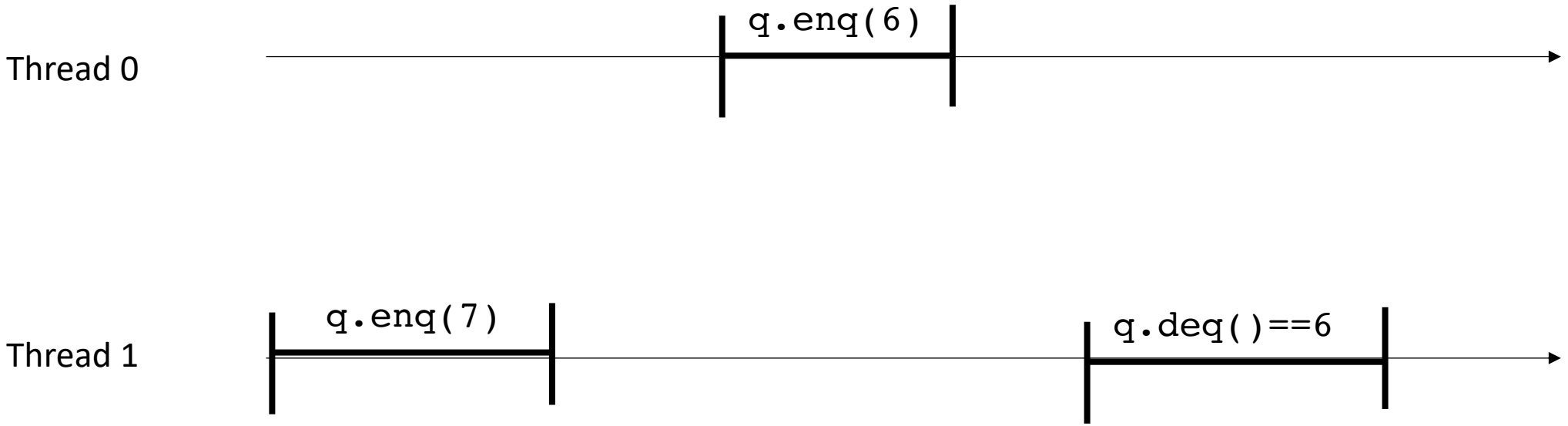
each method as a start, and end time stamp



Sequential consistency and real time

- Add in real time:

This timeline seems strange...

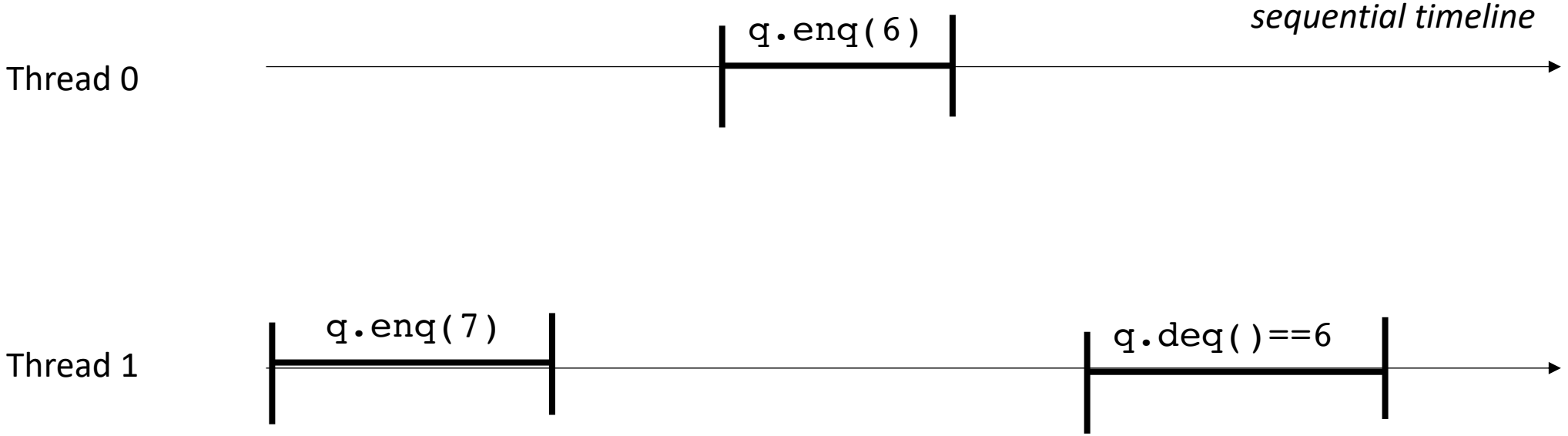


Sequential consistency and real time

- 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

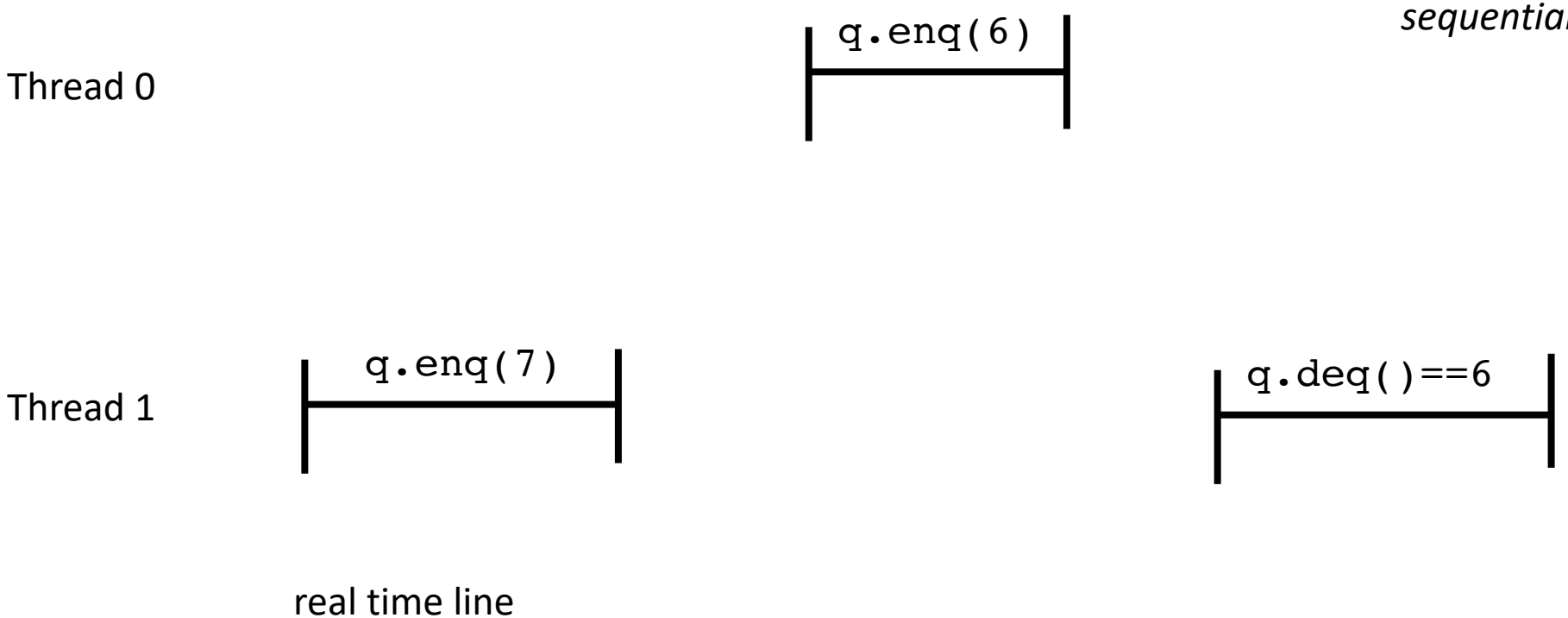


Sequential consistency and real time

- 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

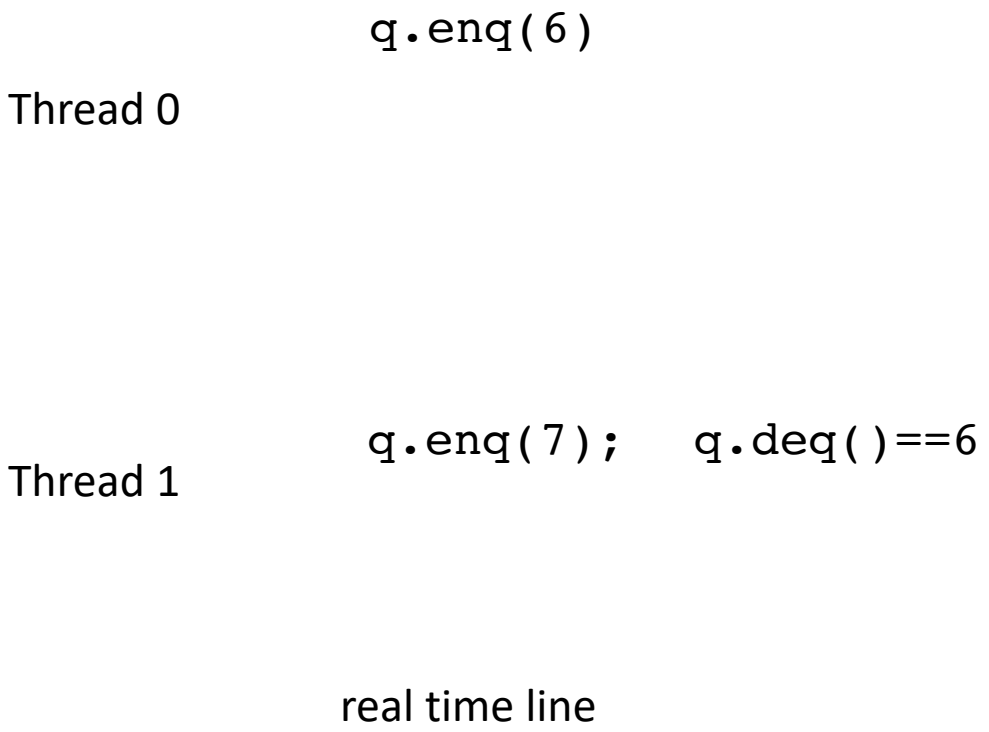


Sequential consistency and real time

- Add in real time:

This execution is allowed in sequential consistency!

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



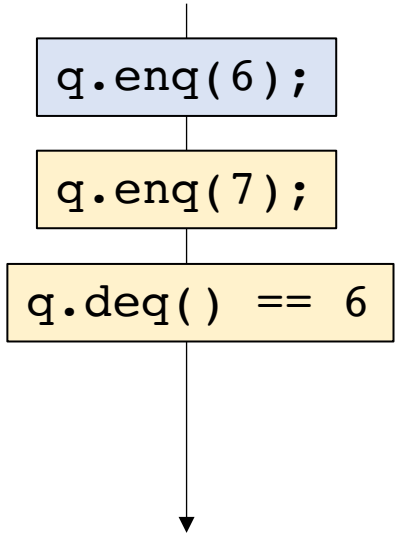
Sequential consistency and real time

- Add in real time:

```
                q.enq(6)
Thread 0

                q.enq(7);   q.deq()==6
Thread 1

real time line
```

This execution is allowed in sequential consistency!

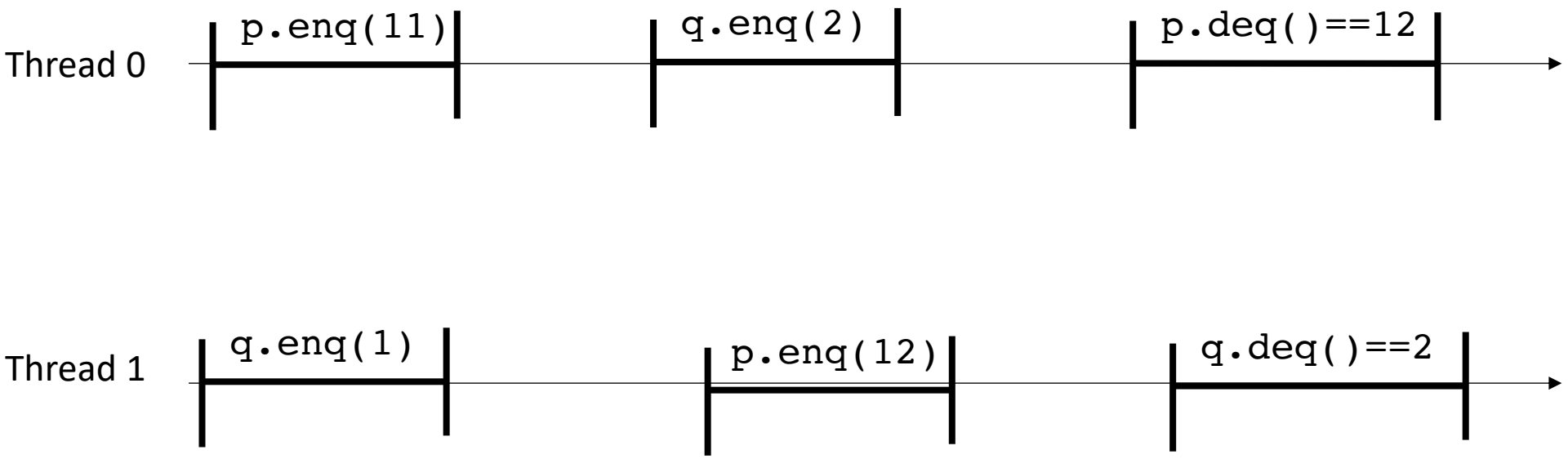
SC doesn't care about real time, only if it can construct its virtual sequential timeline



Sequential consistency and real time

- Add in real time:

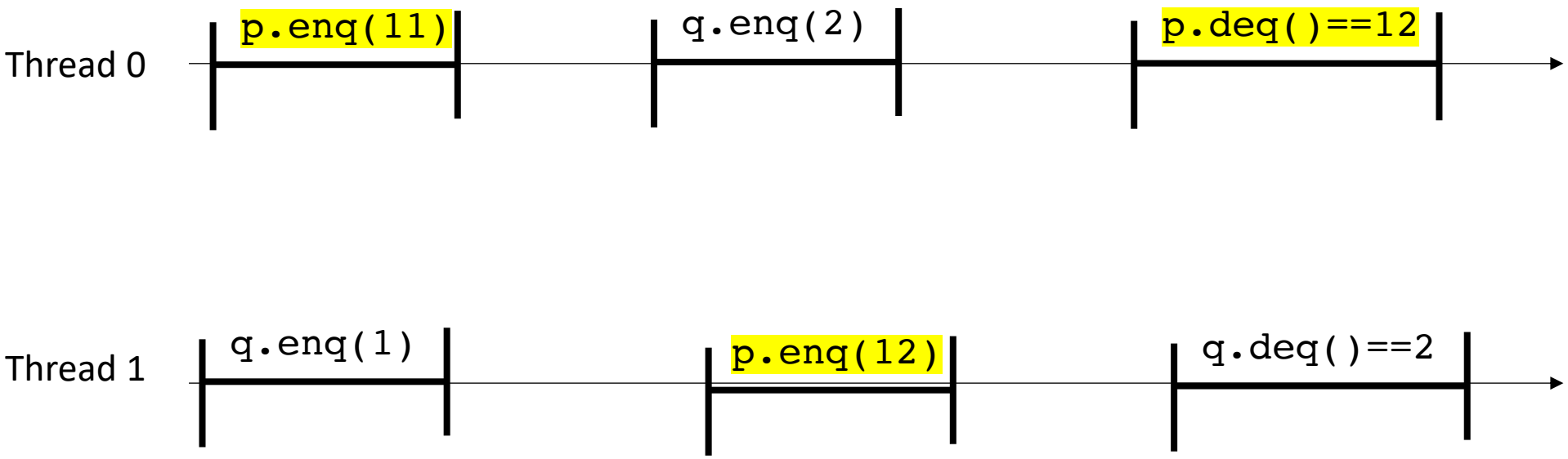
2 objects now: p and q



Sequential consistency and real time

- Add in real time:

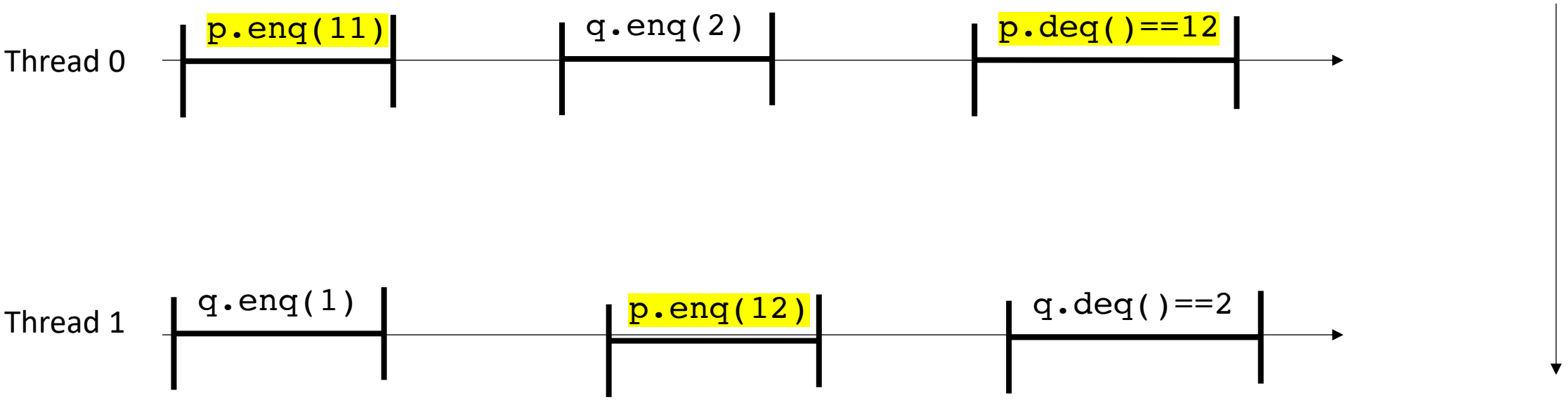
2 objects now: p and q
Consider each object in isolation



Sequential consistency and real time

- Add in real time:

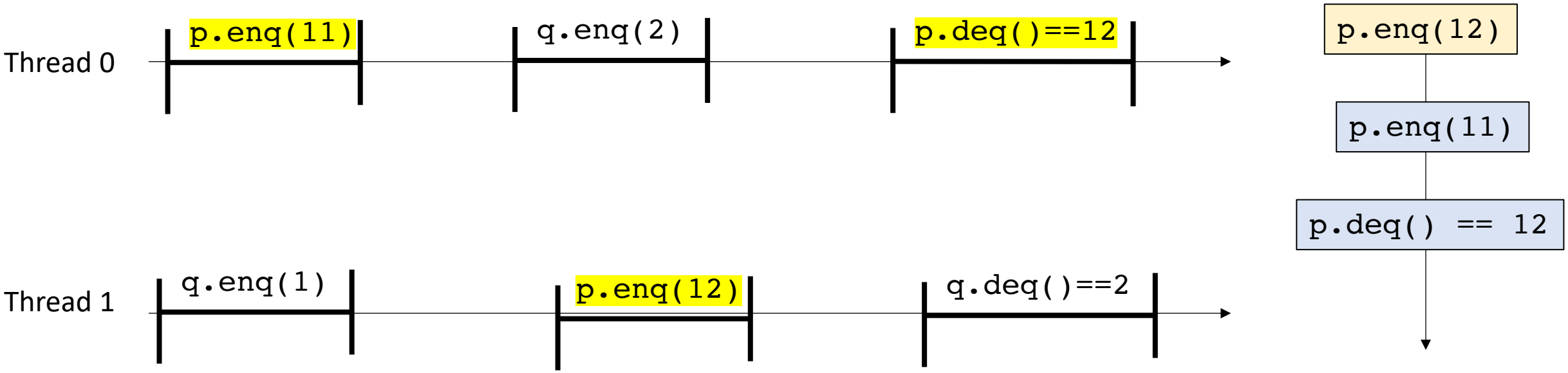
2 objects now: p and q
Consider each object in isolation



Sequential consistency and real time

- Add in real time:

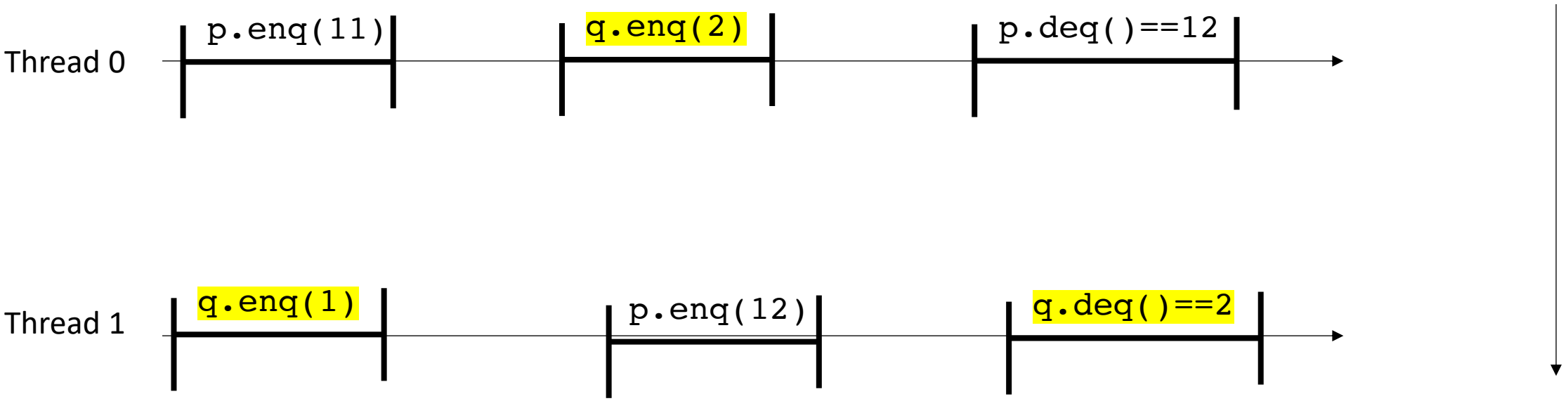
2 objects now: p and q
Consider each object in isolation



Sequential consistency and real time

- Add in real time:

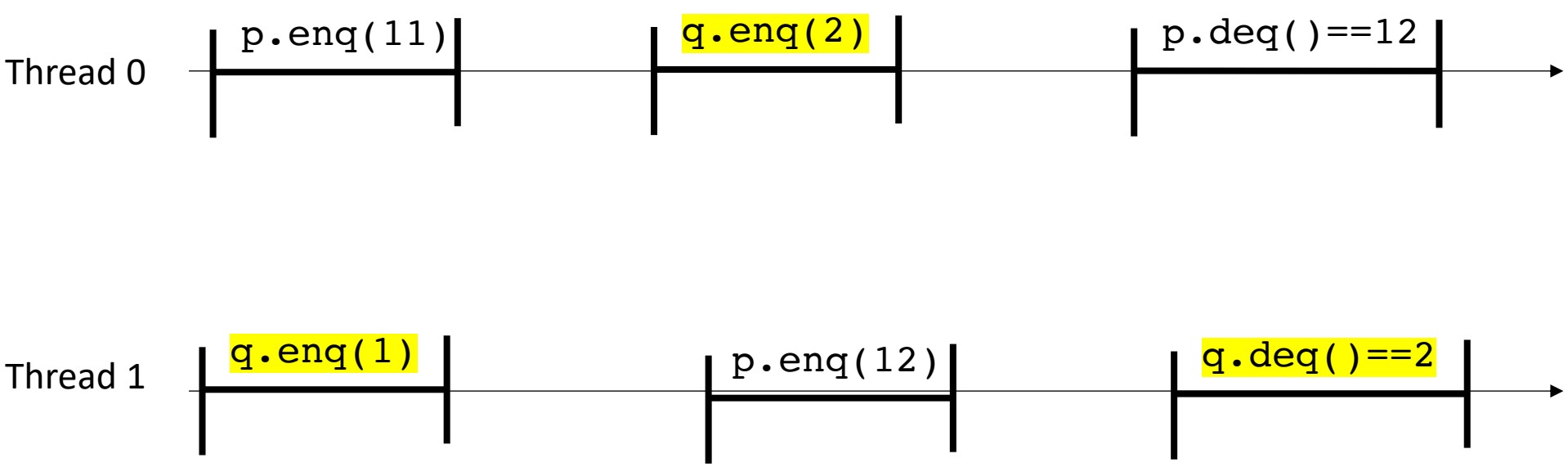
2 objects now: p and q
Consider each object in isolation

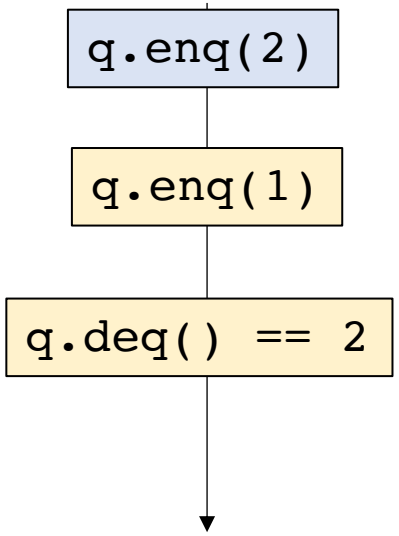


Sequential consistency and real time

- Add in real time:

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

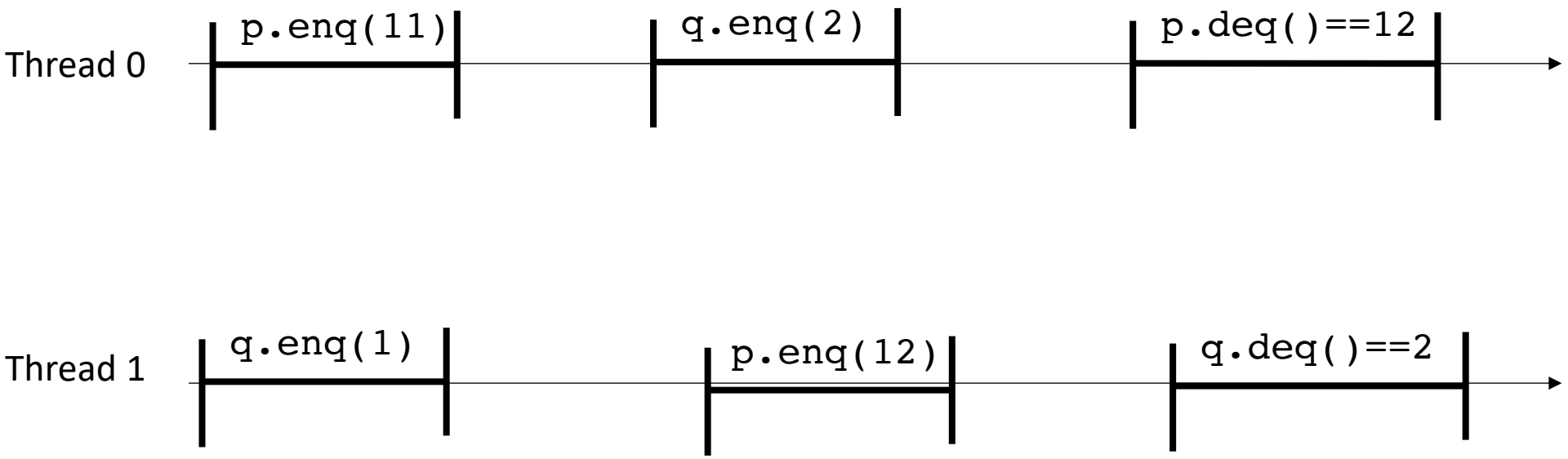




Sequential consistency and real time

- 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
```


Thread 1:

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

Global variable:

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

Thread 0:

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

Thread 1:

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

```
p.enq(11);
```

```
q.enq(2);
```

```
p.deq()== 12;
```

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

```
p.enq(12);
```

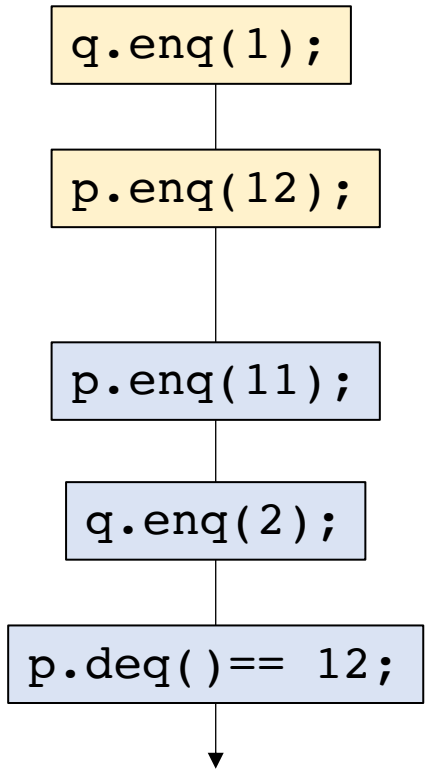
```
q.deq()== 2;
```

Global variable:

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

Thread 0:

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



Thread 1:

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

```
q.deq()== 2;
```

No place for this one to go!

What does this mean?

- Even if objects in isolation are sequentially consistent
- Programs composed of multiple objects might not be!
- We would like to be able to use more than 1 object in our programs!

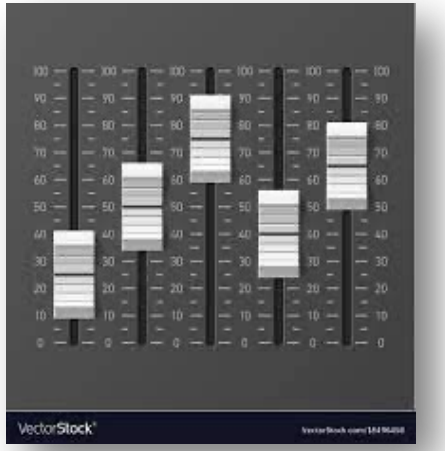
Schedule

- Problems with sequential consistency
- **Linearizability**
- Specialized concurrent queues

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

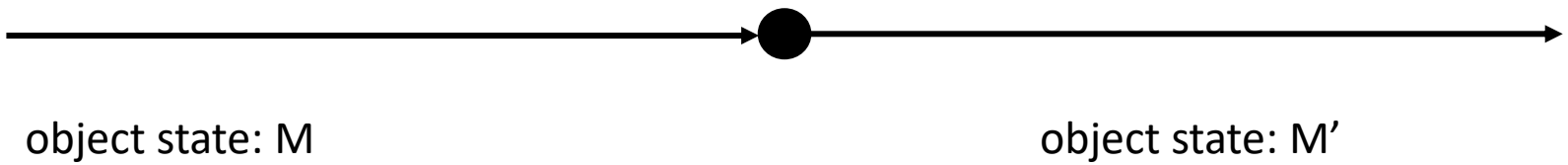
each operation has a linearizability point

- 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

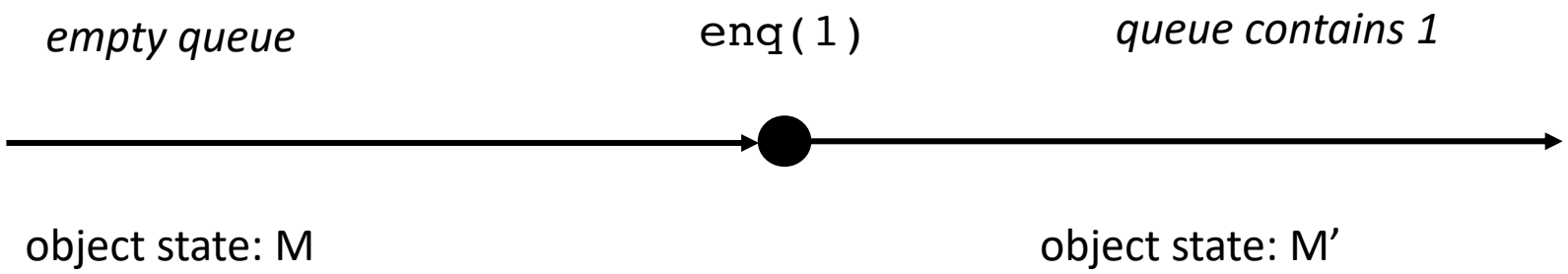
- does not overlap with other with other linearizability points
- indivisible computation (critical section, atomic RMW, atomic load, atomic store)
- object update (or read) occurs exactly at this point



Linearizability

each operation has a linearizability point

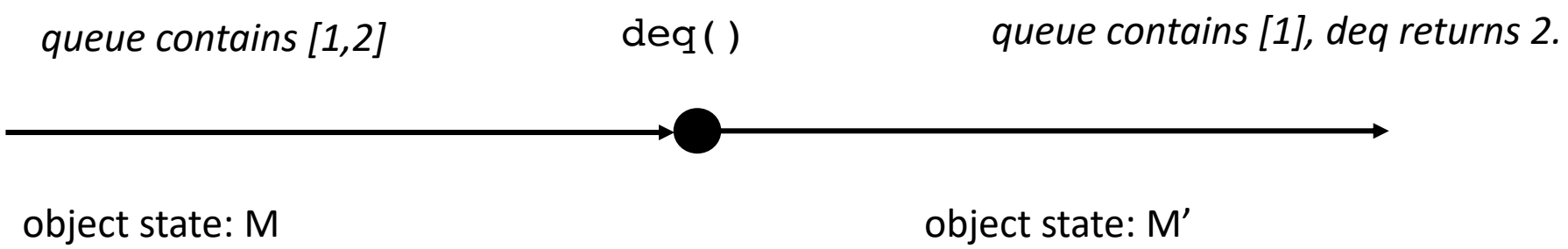
- does not overlap with other with other linearizability points
- indivisible computation (critical section, atomic RMW, atomic load, atomic store)
- object update (or read) occurs exactly at this point



Linearizability

each operation has a linearizability point

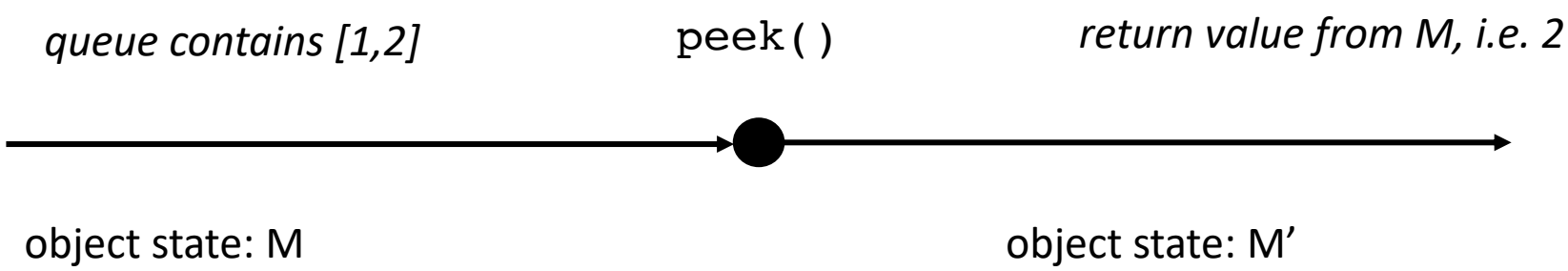
- does not overlap with other with other linearizability points
- indivisible computation (critical section, atomic RMW, atomic load, atomic store)
- object update (or read) occurs exactly at this point



Linearizability

each operation has a linearizability point

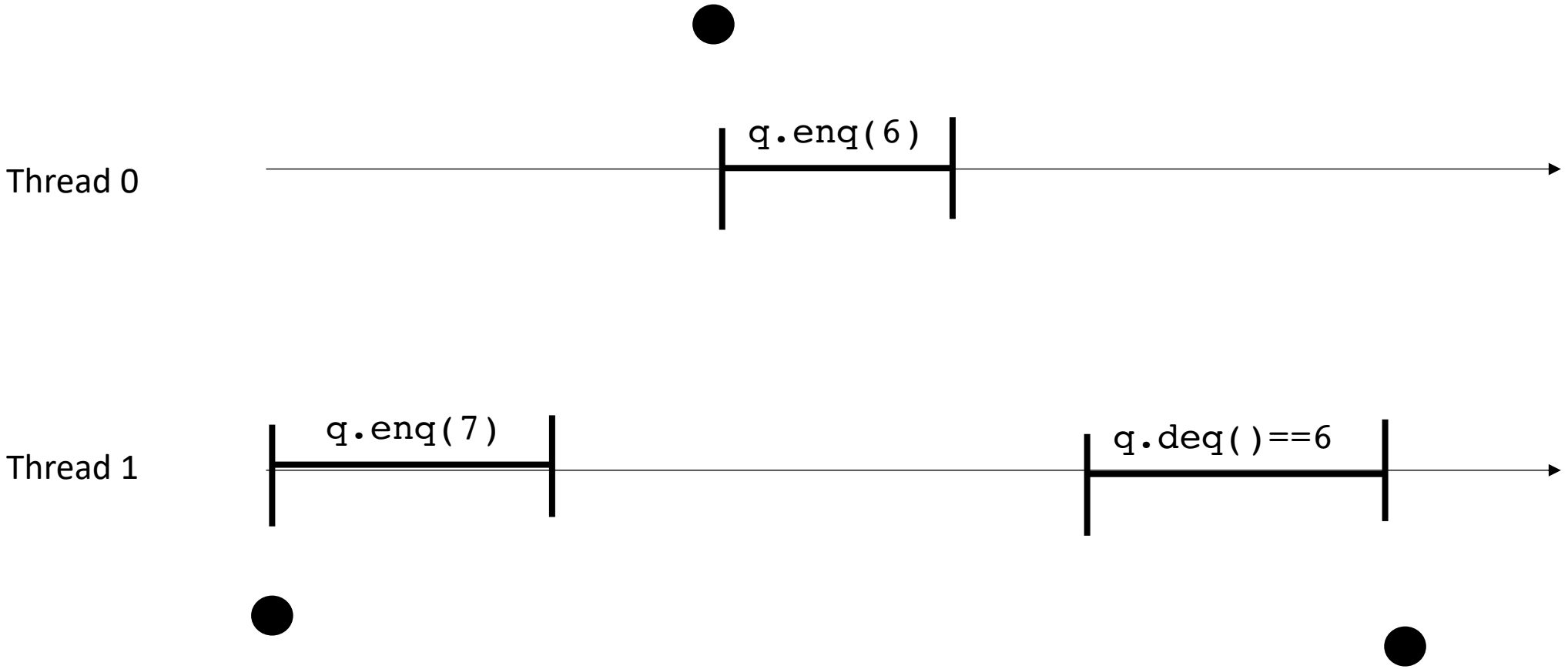
- does not overlap with other with other linearizability points
- indivisible computation (critical section, atomic RMW, atomic load, atomic store)
- object update (or read) occurs exactly at this point



Linearizability

each command gets a linearization point.

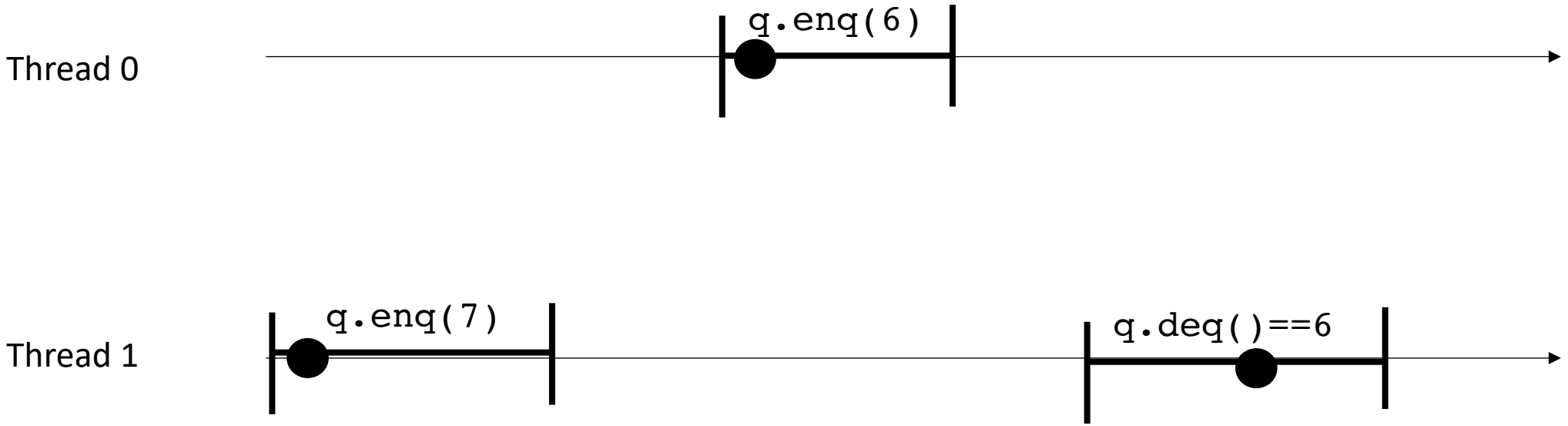
You can place the point any where between its innovation and response!



Linearizability

each command gets a linearization point.

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

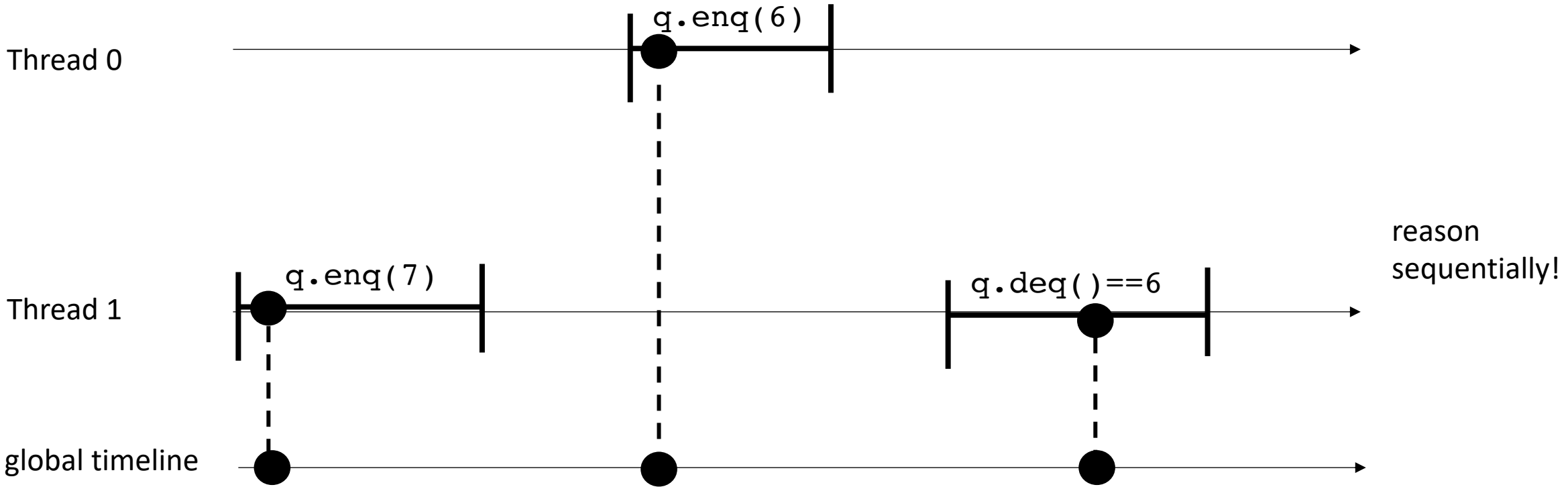


Linearizability

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



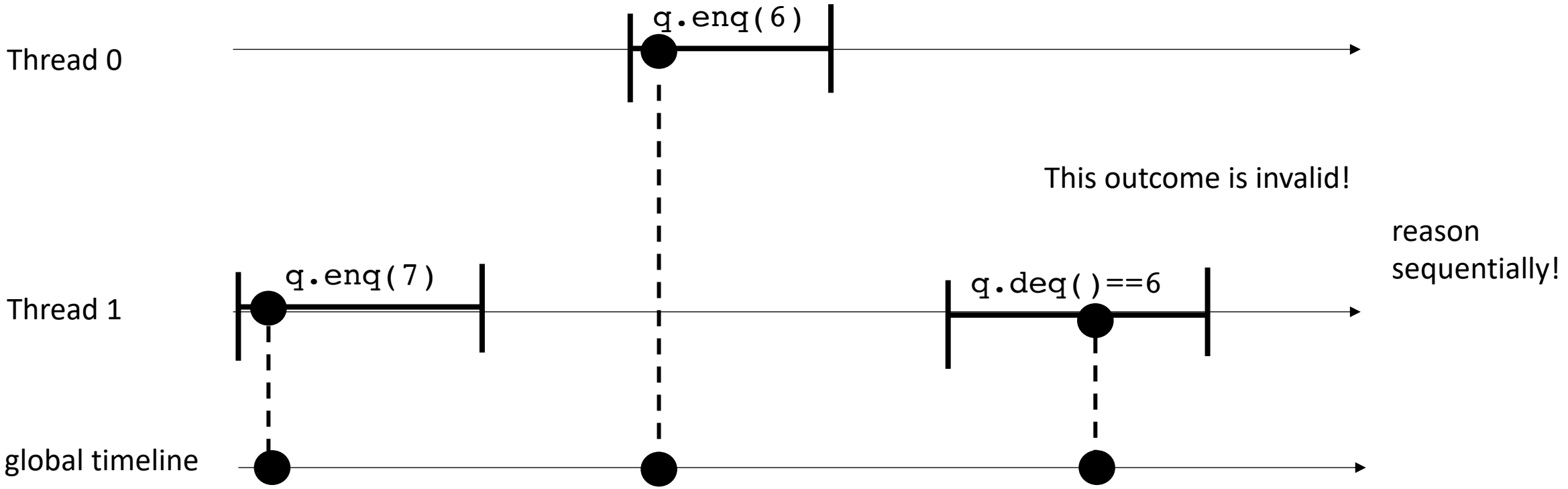
reason sequentially!

Linearizability

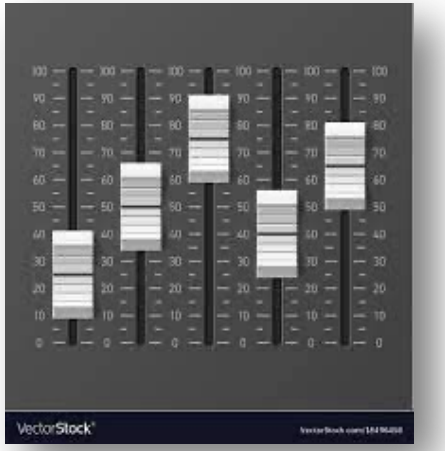
each command gets a linearization point.

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

Project the linearization points to a global timeline



Linearizability

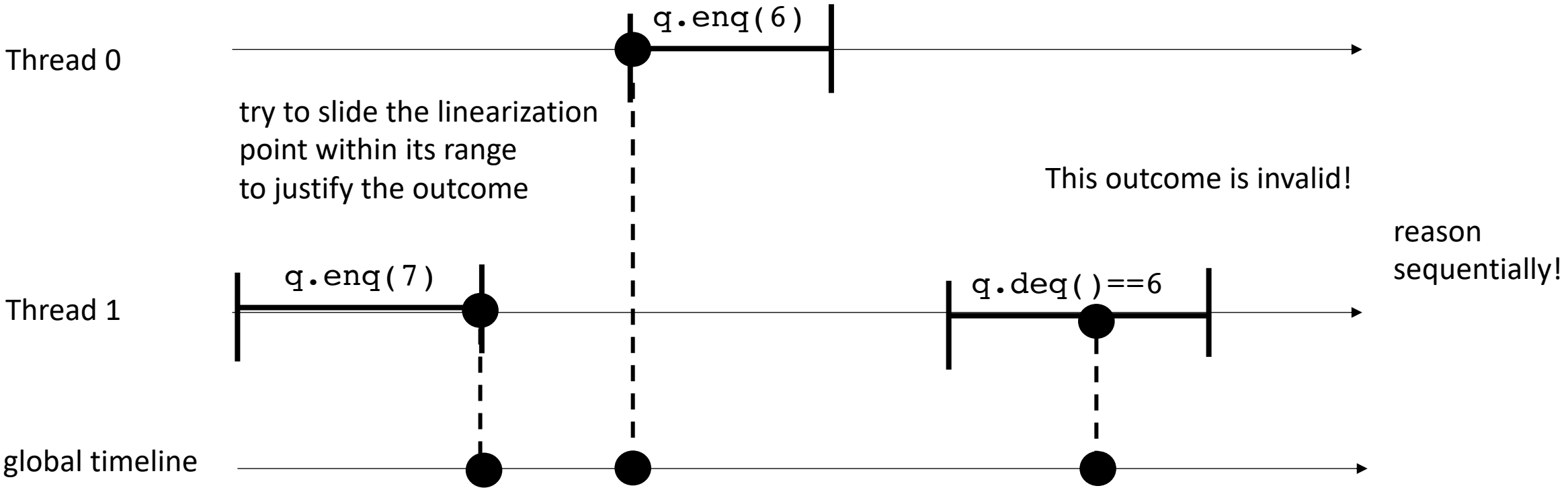


each command gets a linearization point.

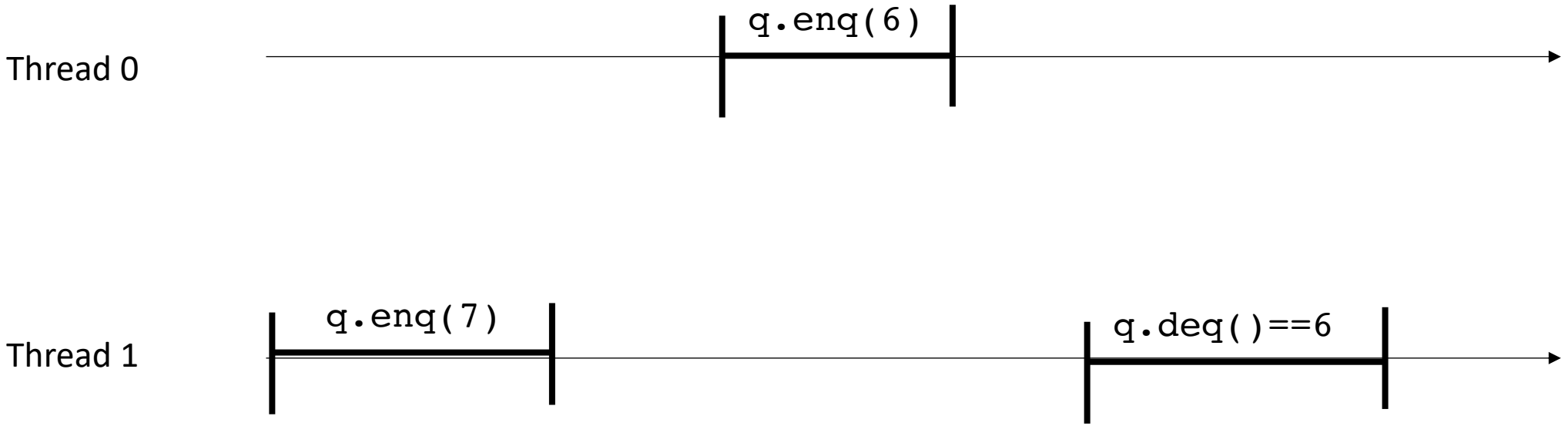
You can place the point any where between its innovation and response!

Project the linearization points to a global timeline

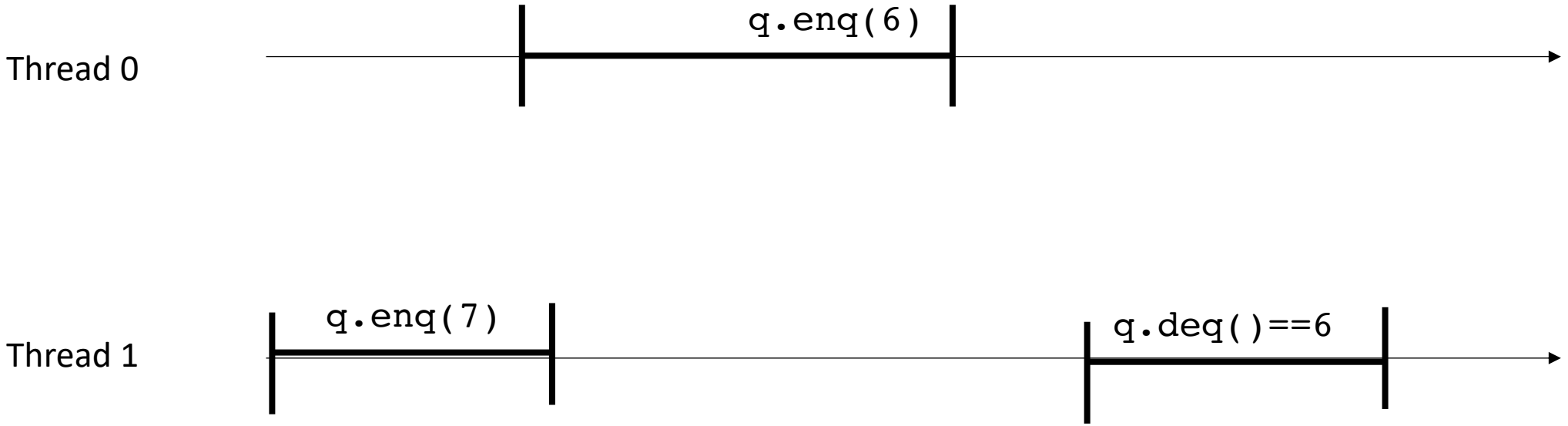
slider game!



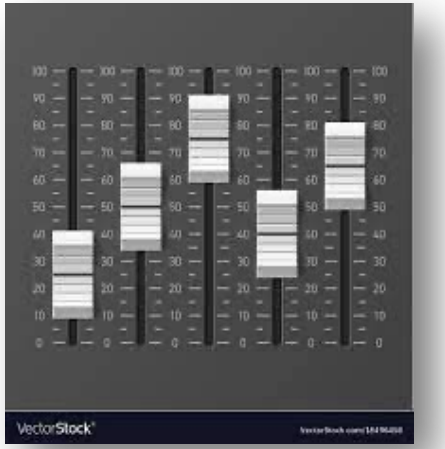
Linearizability

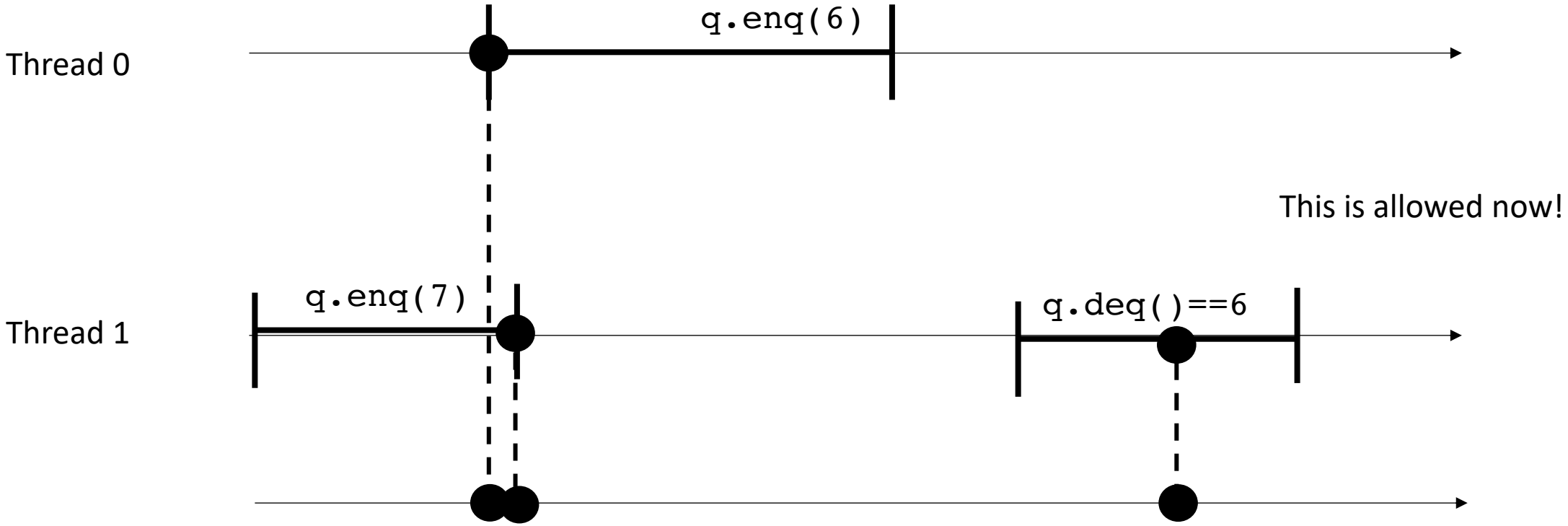


Linearizability

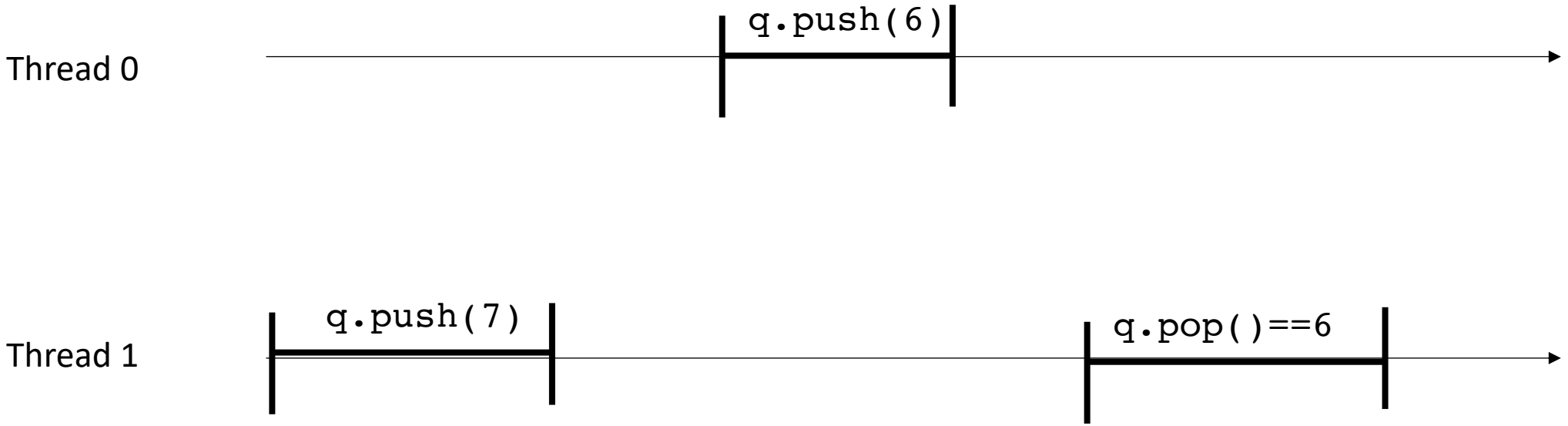


Linearizability

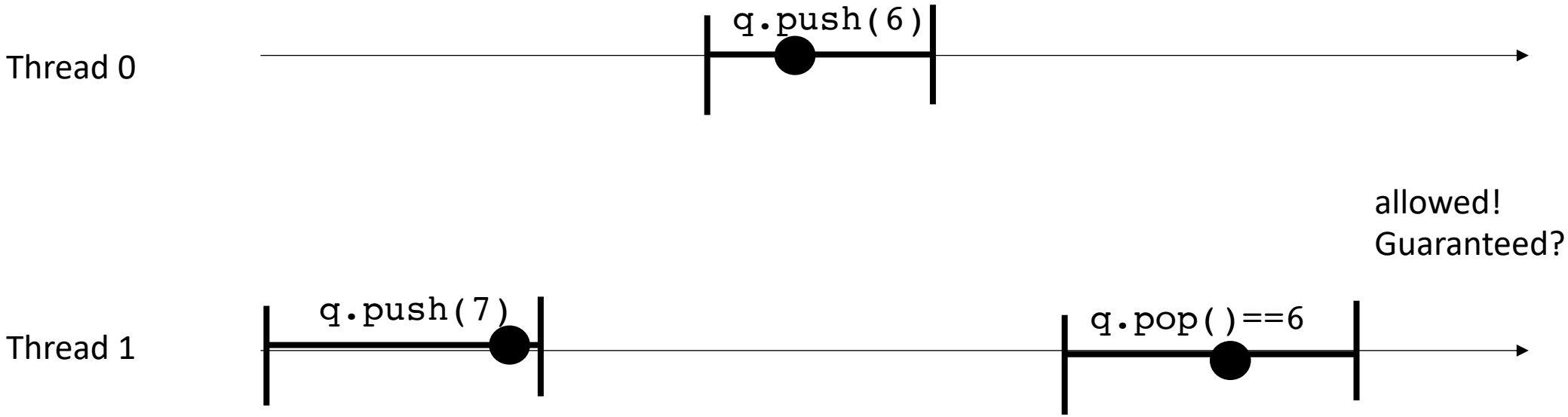




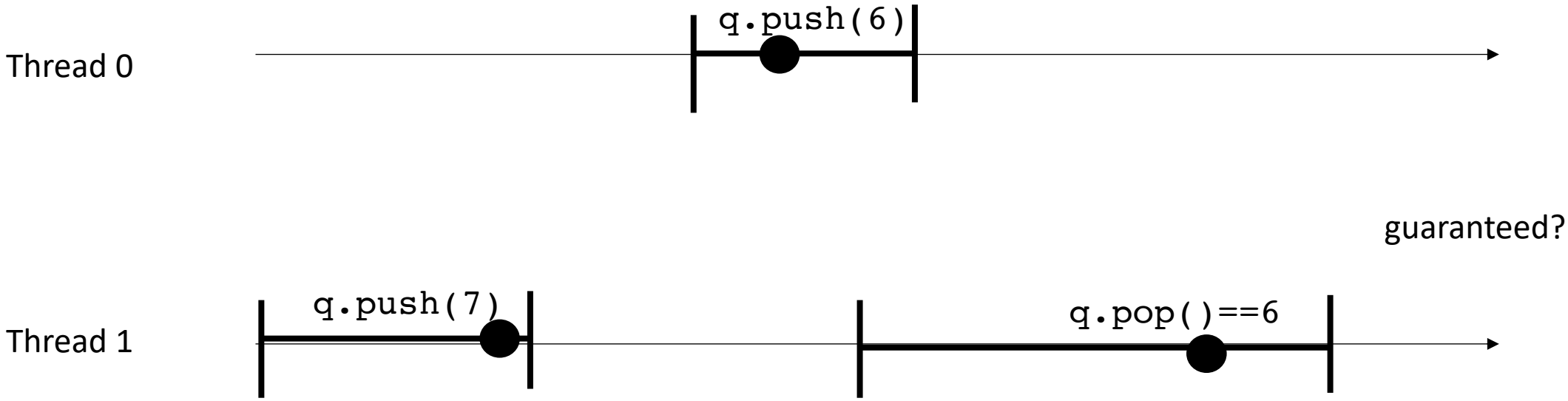
Linearizability



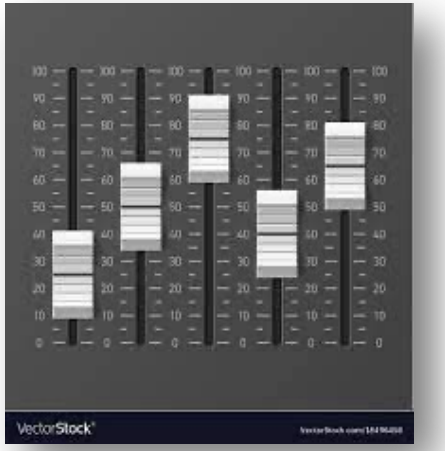
Linearizability

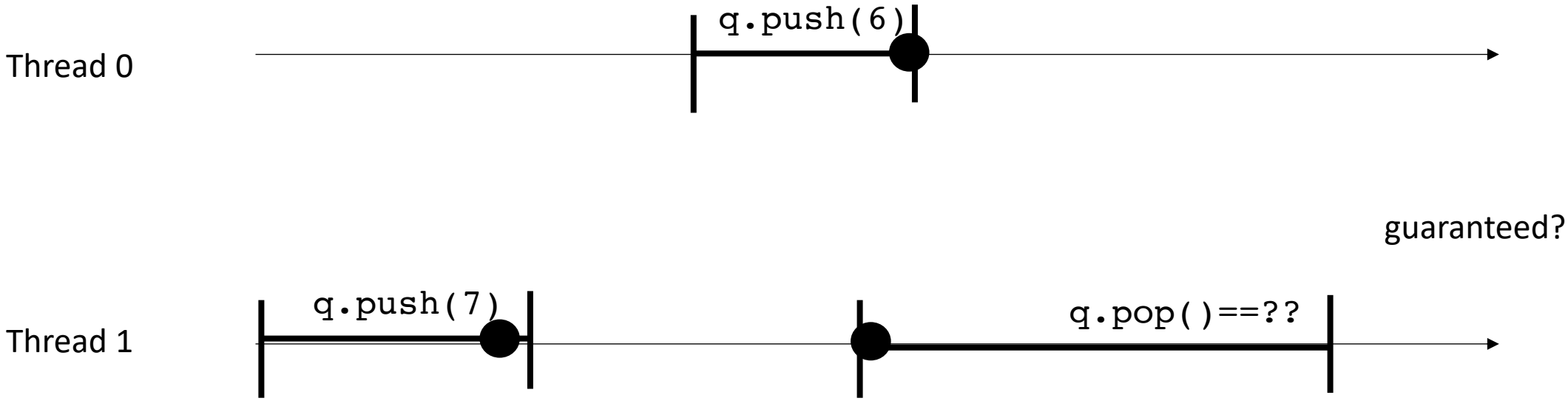


Linearizability



Linearizability



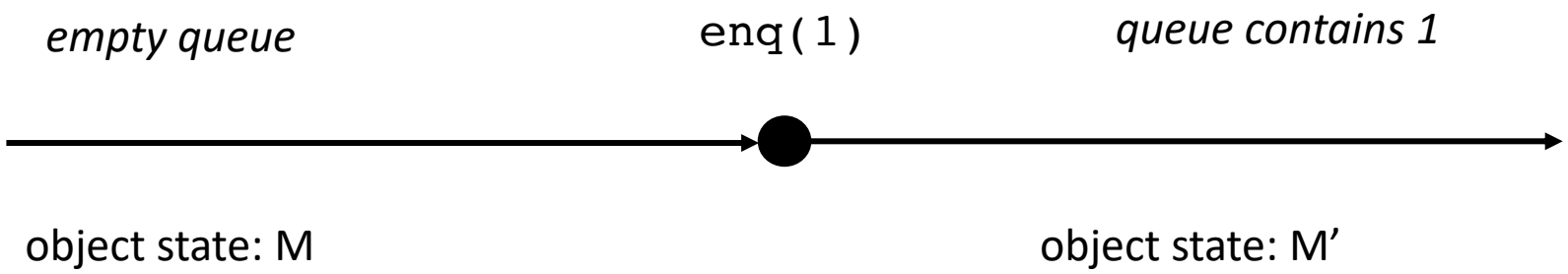


Linearizability

- 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.
- Overall strategy:
  - Write our objects to be linearizable: need to identify linearizable points
  - Reason about our programs using SC: no need for timelines, just need code

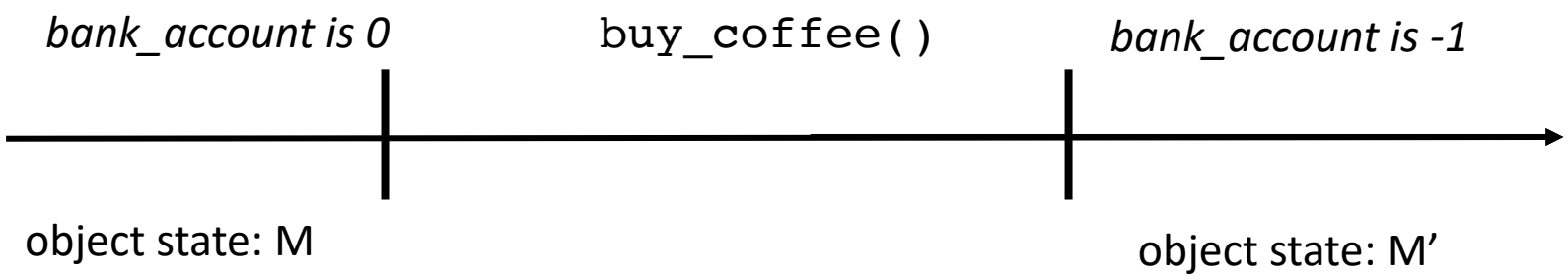
Linearizability

- 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.

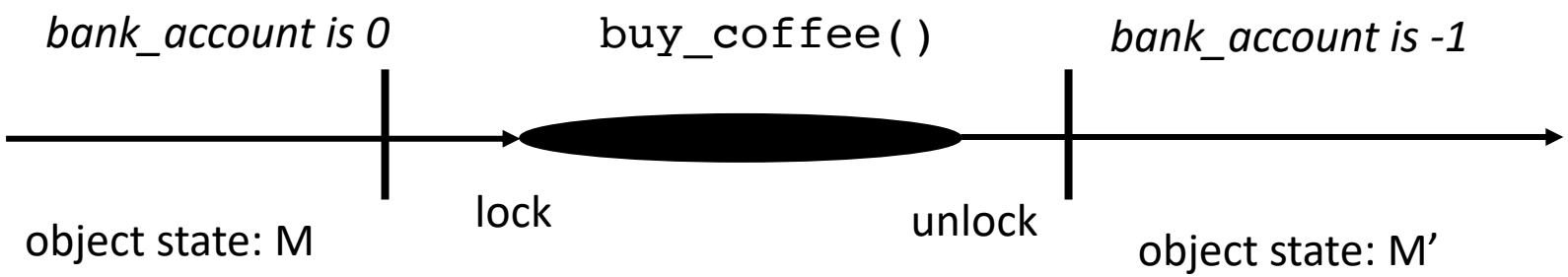



```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    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

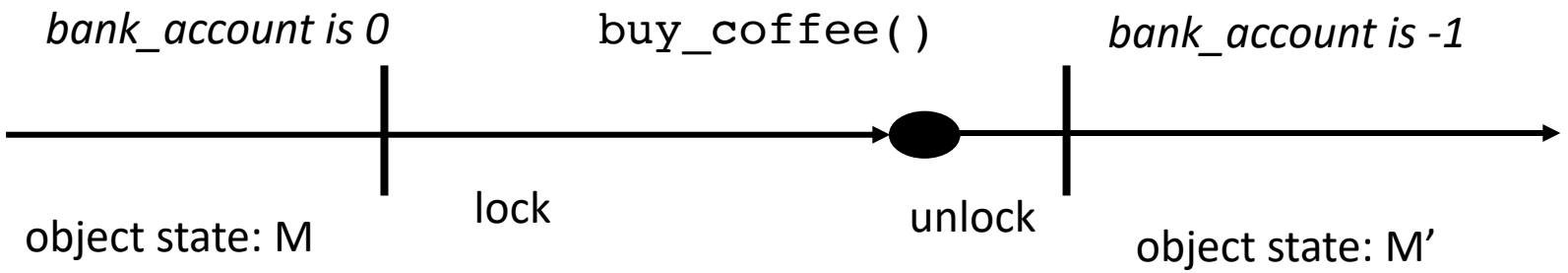



```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    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.

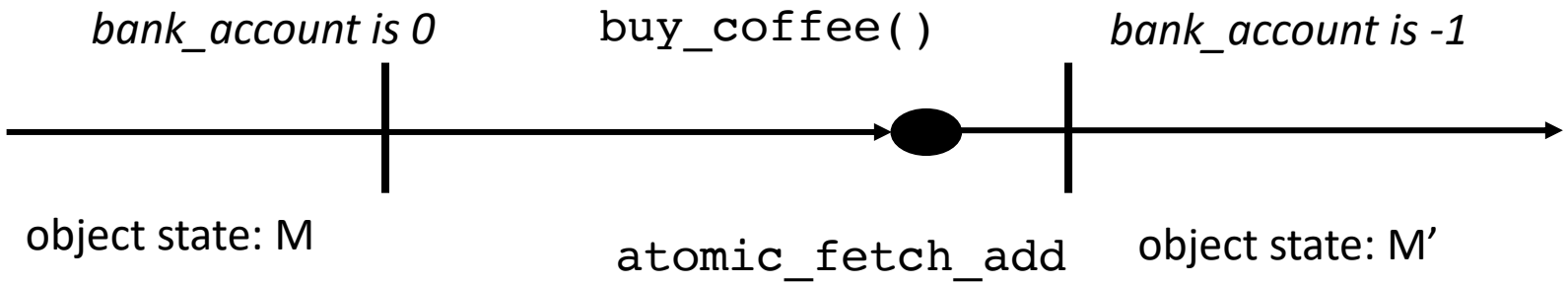



```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    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;
};
```



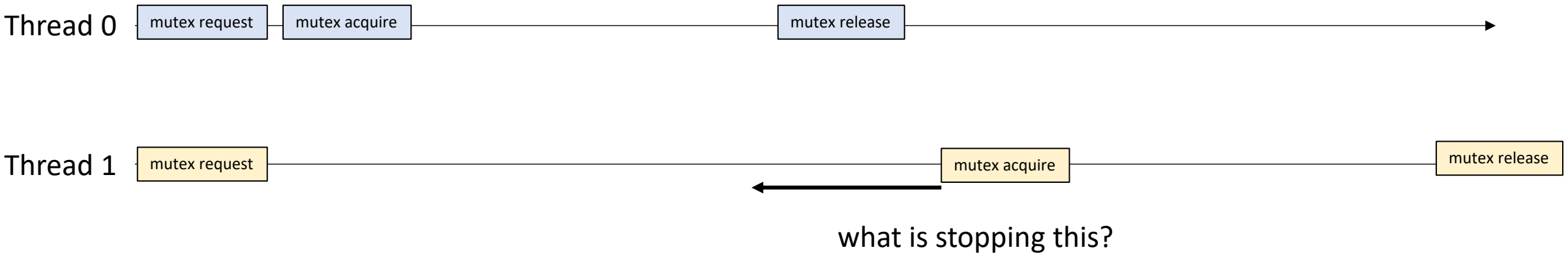
Lecture schedule

- Revisiting sequential consistency
- Linearizablity
- **Progress Properties**
- Implementing a set

Progress properties

- Going back to specifications:

Recall the mutex

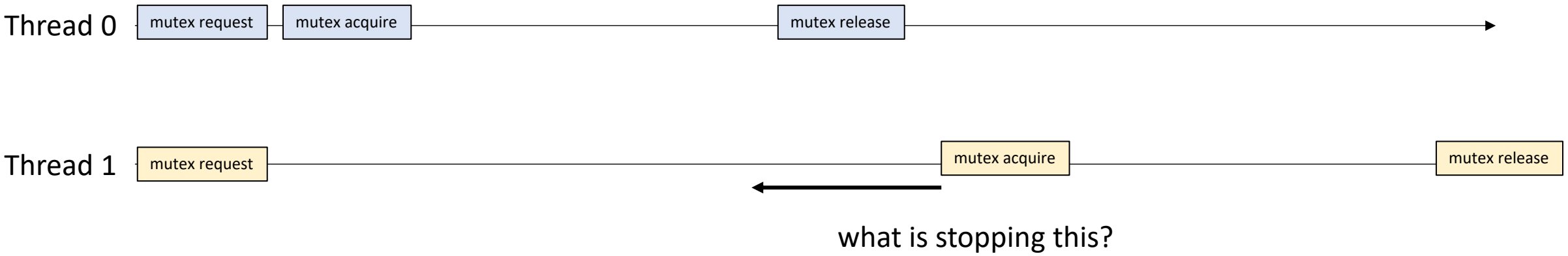


Progress properties

- 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

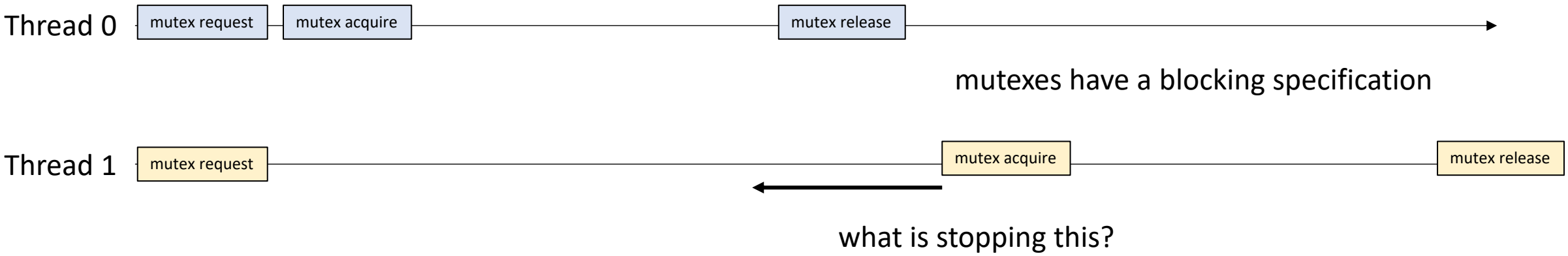


Progress properties

- 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

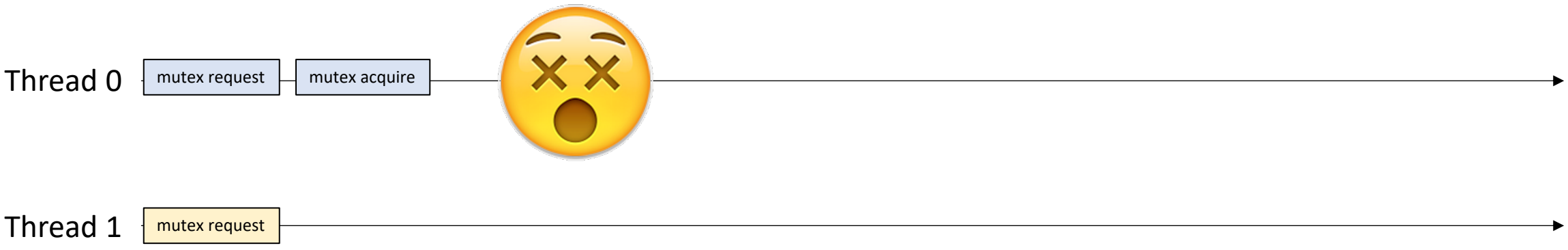


Progress properties

- 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

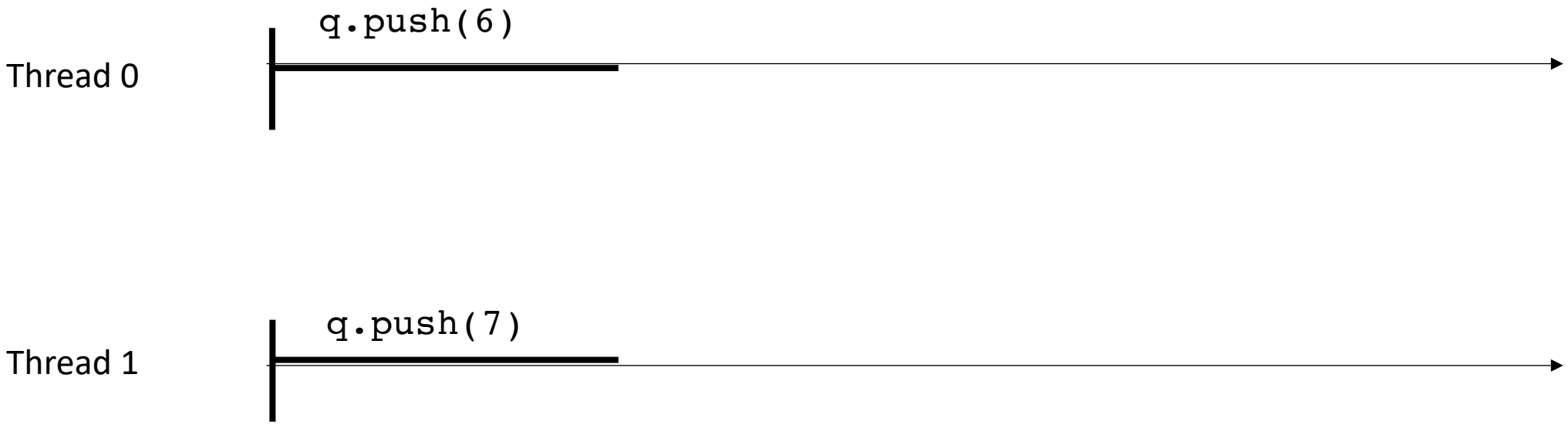
Recall the mutex



What now?!

Linearizability

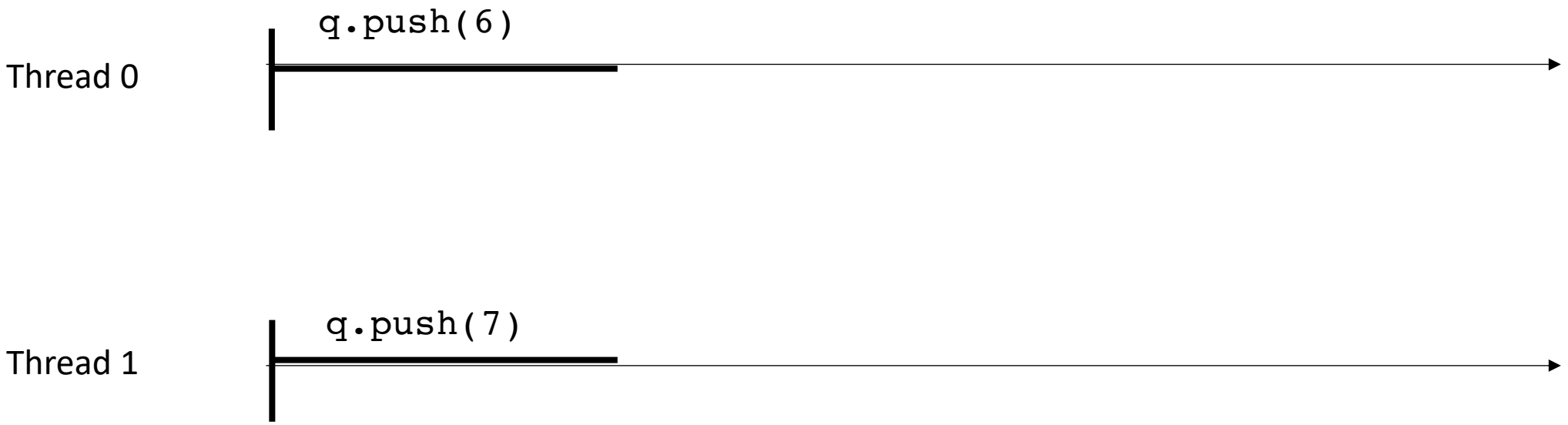
Two unfinished commands.



Linearizability

Two unfinished commands.

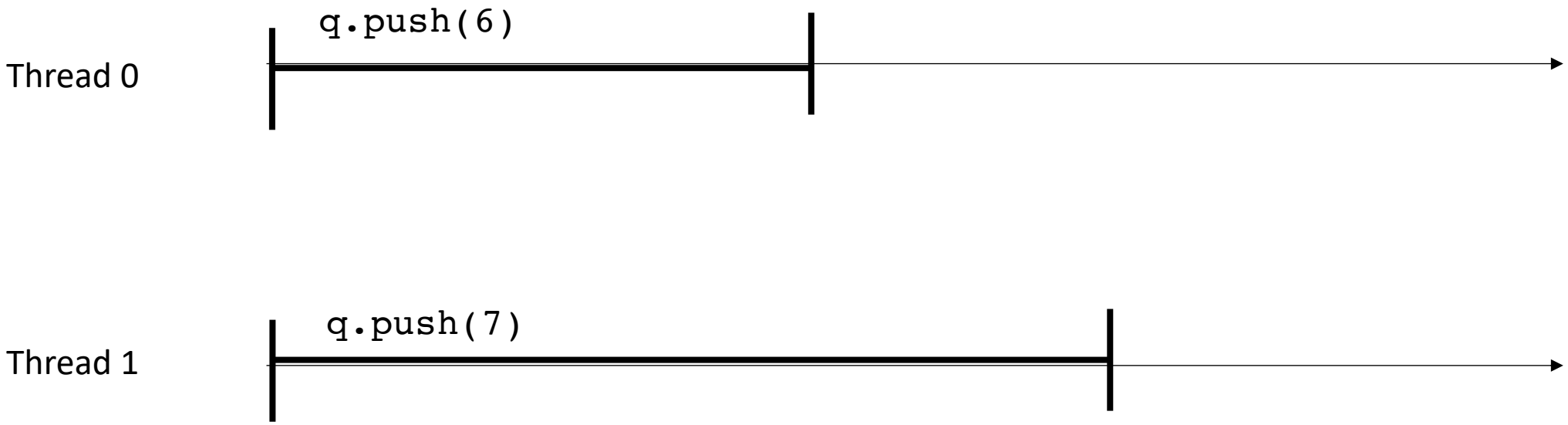
Linearizability does not dictate that one needs to wait for another



Linearizability

Two unfinished commands.

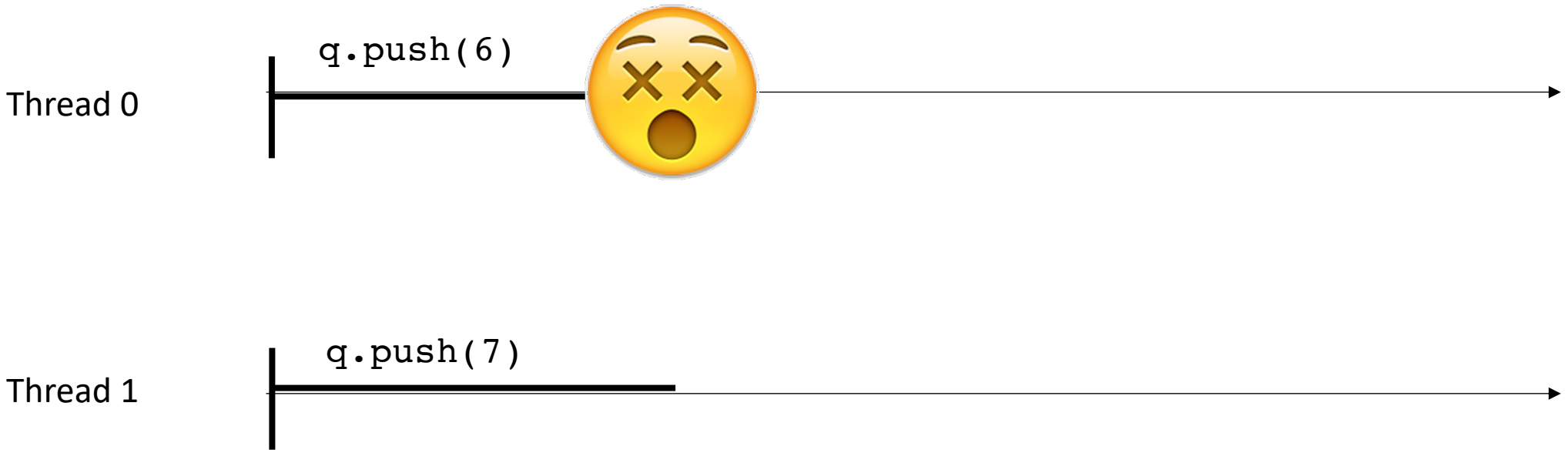
Linearizability does not dictate that one needs to wait for another



Linearizability

Two unfinished commands.

Linearizability does not dictate that one needs to wait for another

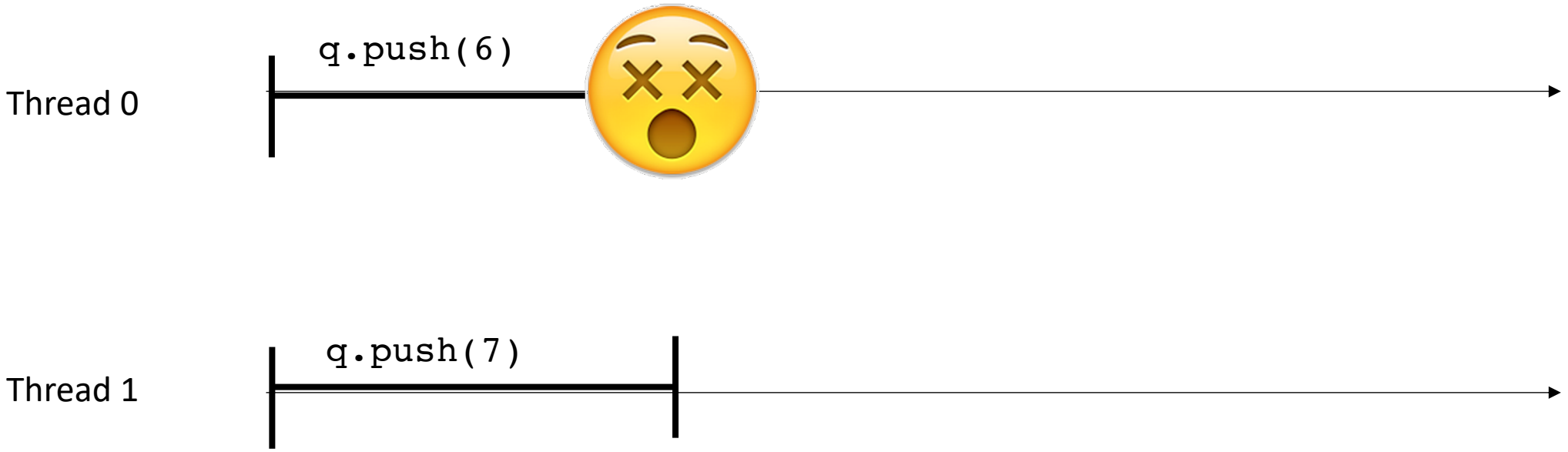


for mutexes, the specification required that the system hang.

Linearizability

Two unfinished commands.

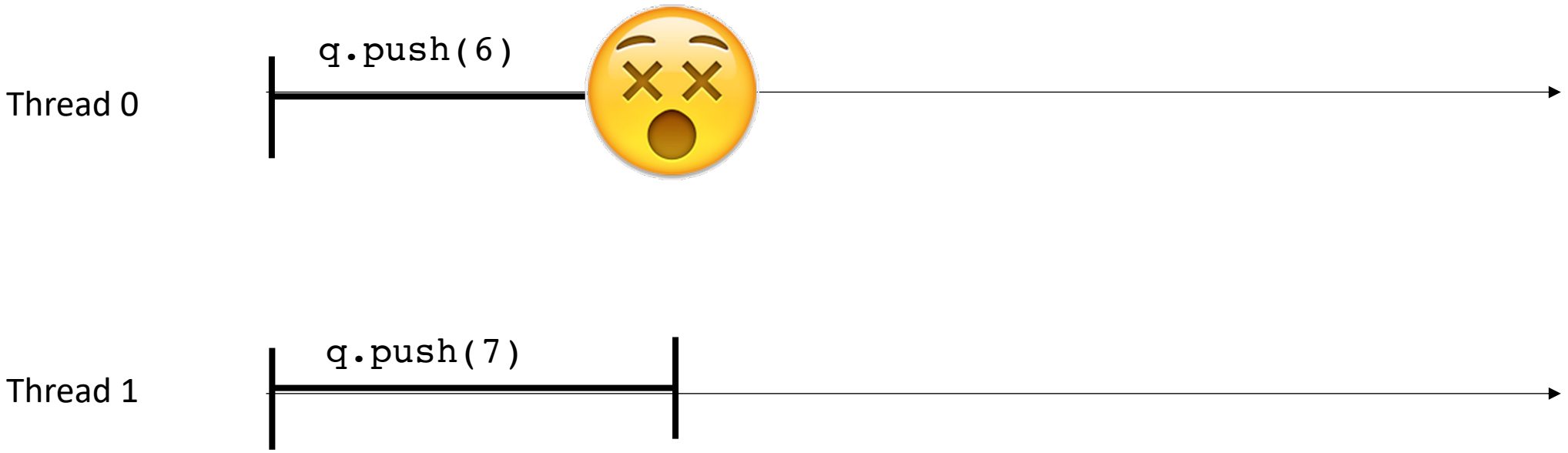
Linearizability does not dictate that one needs to wait for another



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

Linearizability

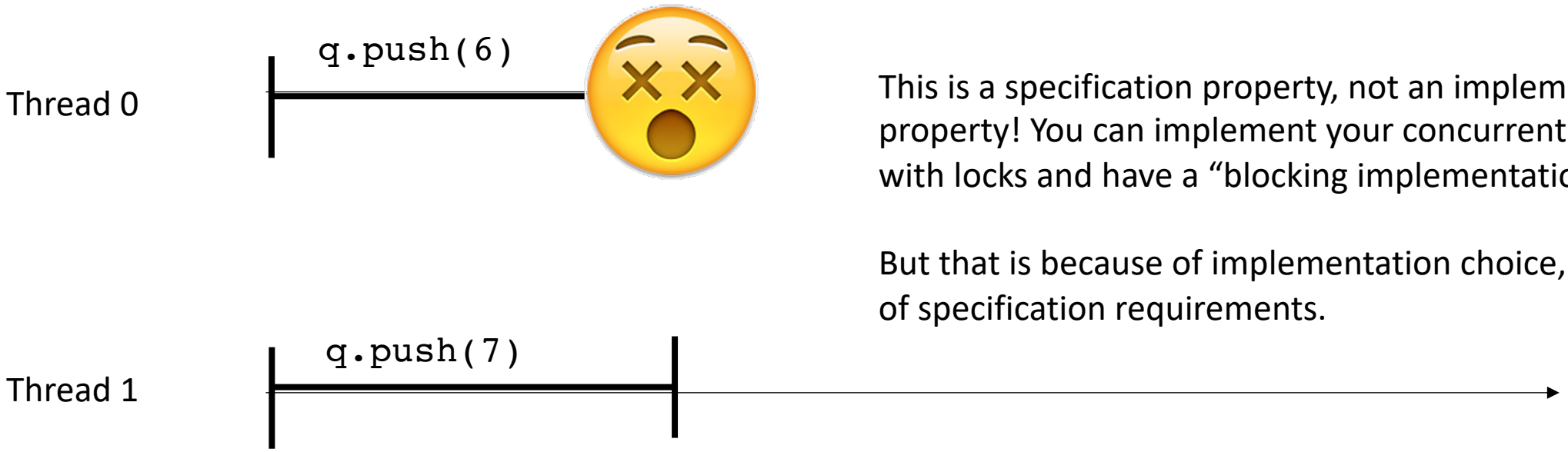
Non-blocking specification:
Every thread is allowed to continue executing
REGARDLESS of the behavior of other threads



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:

Schedule

- Problems with sequential consistency
- Linearizability
- **Specialized concurrent queues**

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?

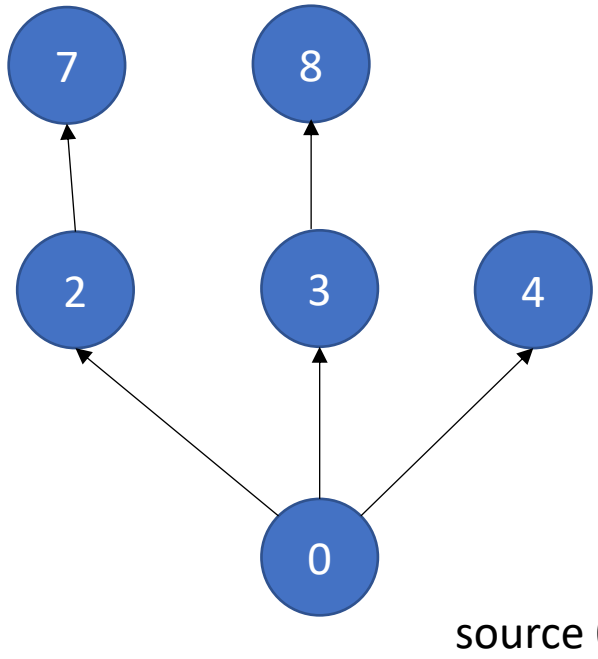
- Computation done in phases:
  - First phase prepares the queue (by writing into it)
  - All threads join
  - Second phase reads values from the queue.

Input/Output Queues

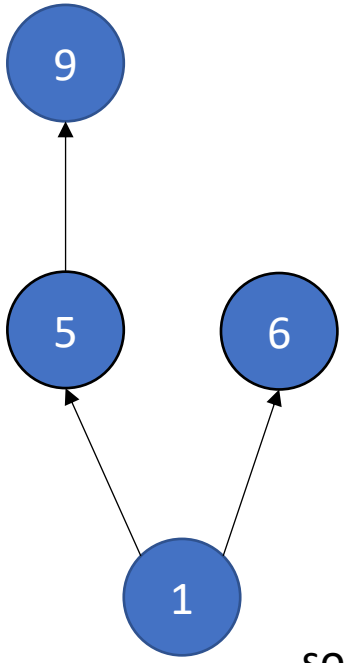
- Example: Information flow in graph applications:

Input/Output Queues

- Example: Information flow in graph applications:



source 0



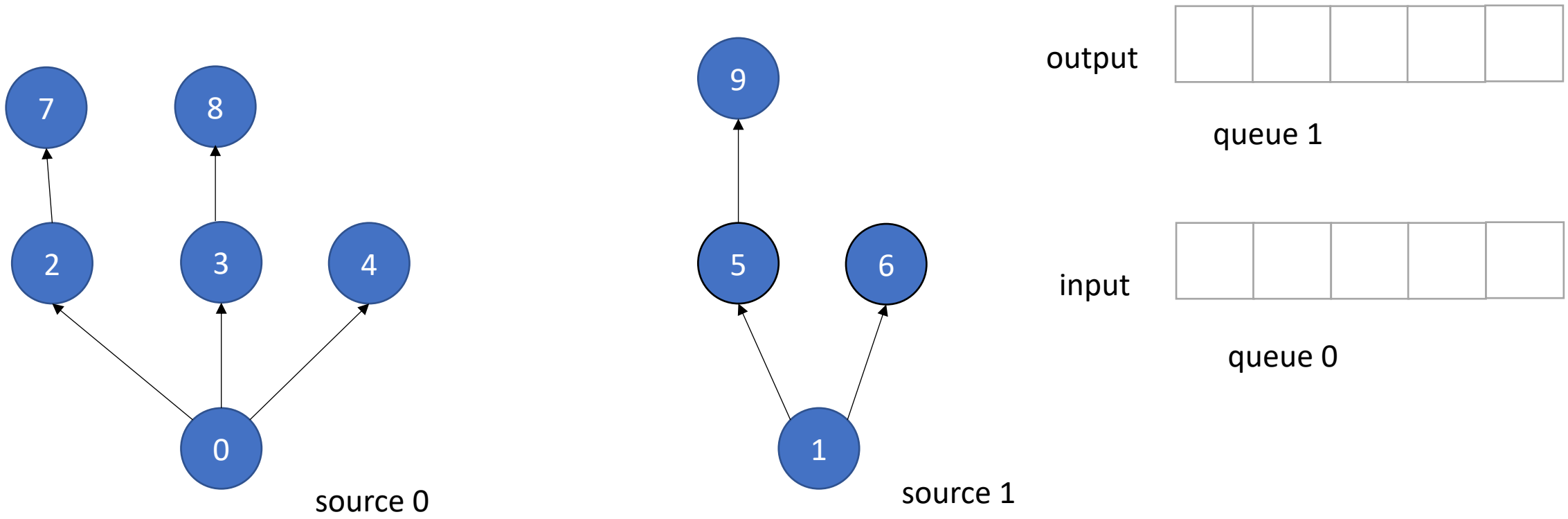
source 1

queue 1

queue 0

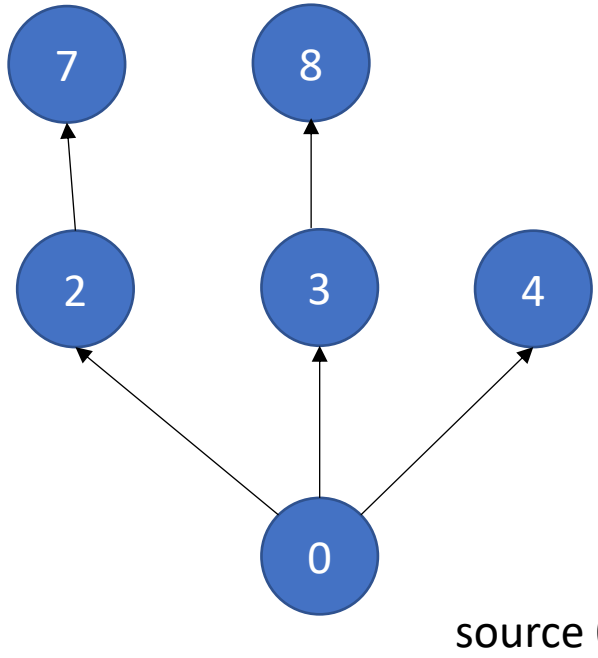
Input/Output Queues

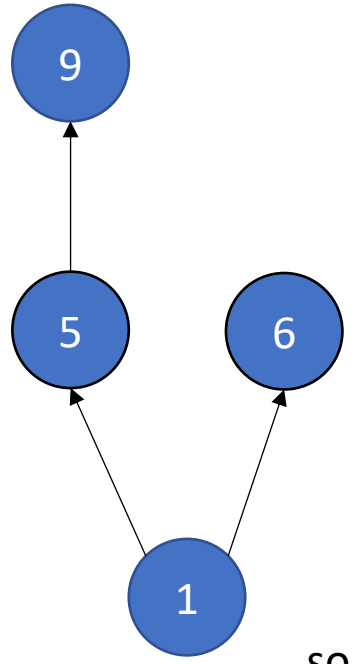
- Example: Information flow in graph applications:

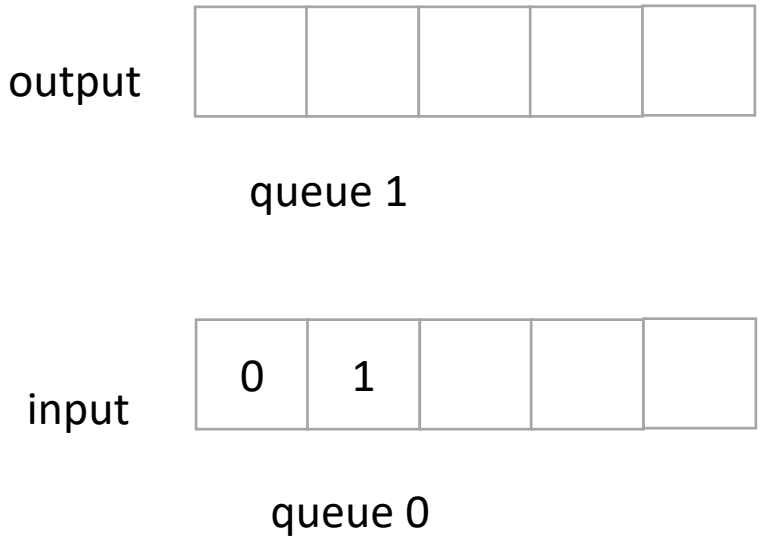


Input/Output Queues

- Example: Information flow in graph applications:

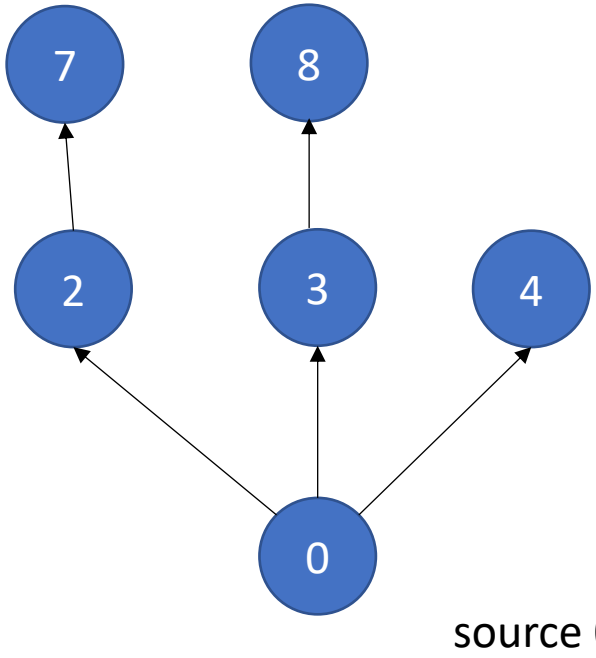


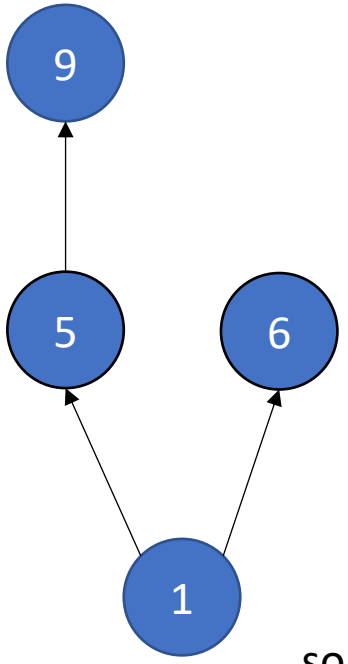


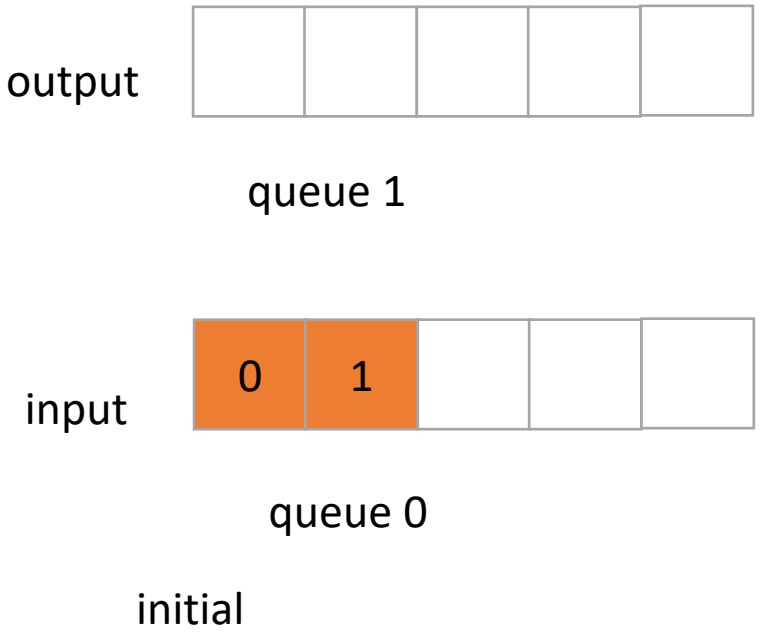


Input/Output Queues

- Example: Information flow in graph applications:

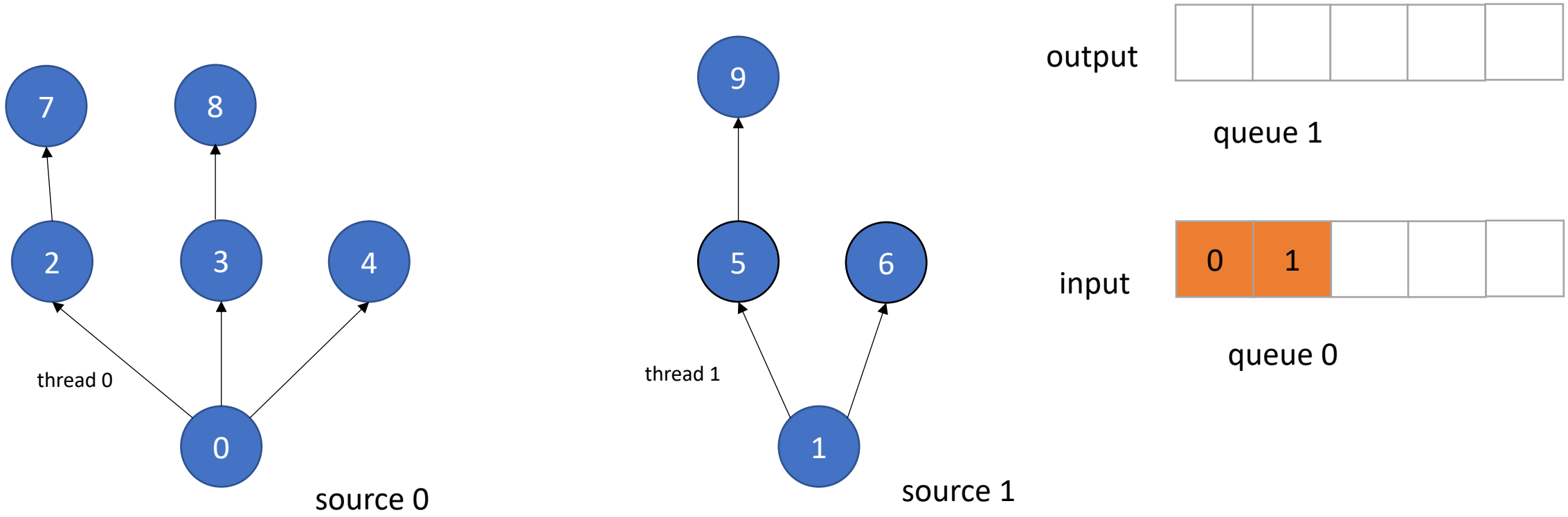






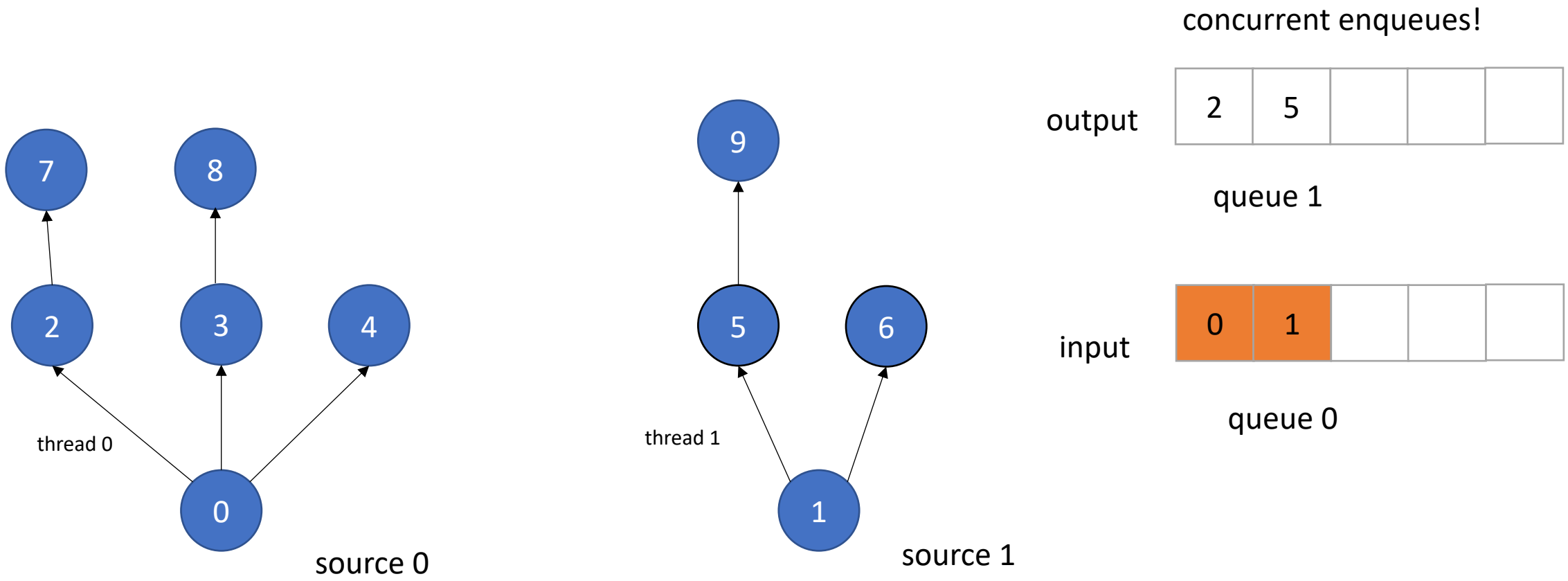
Input/Output Queues

- Example: Information flow in graph applications:



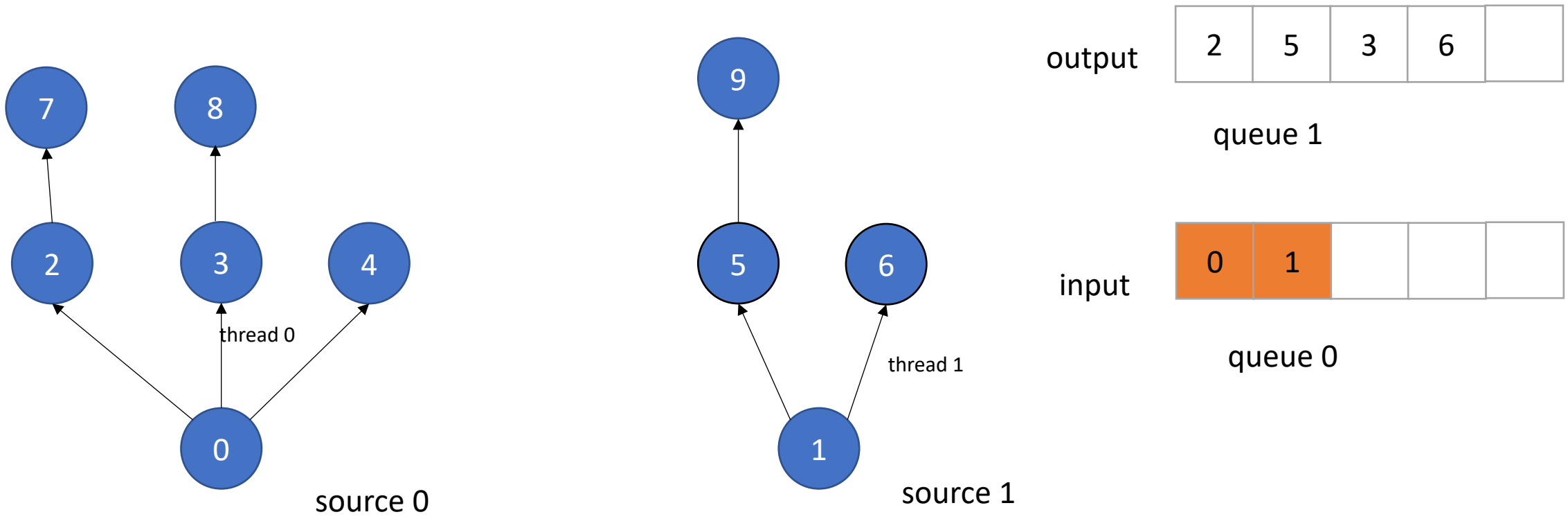
Input/Output Queues

- Example: Information flow in graph applications:



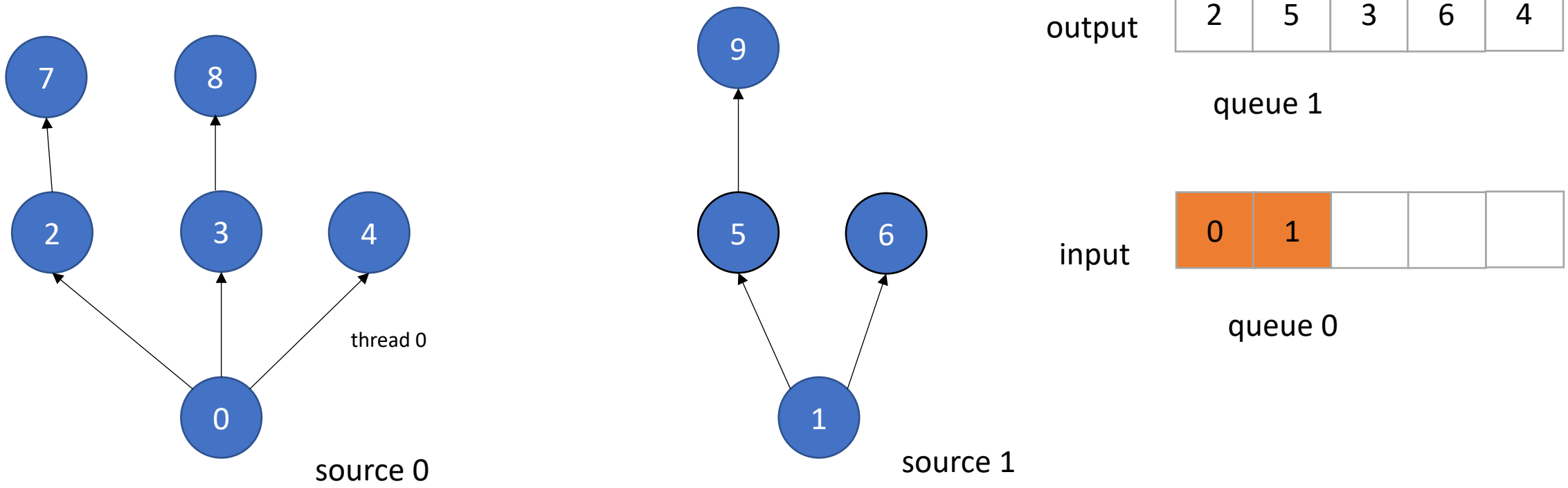
Input/Output Queues

- Example: Information flow in graph applications:



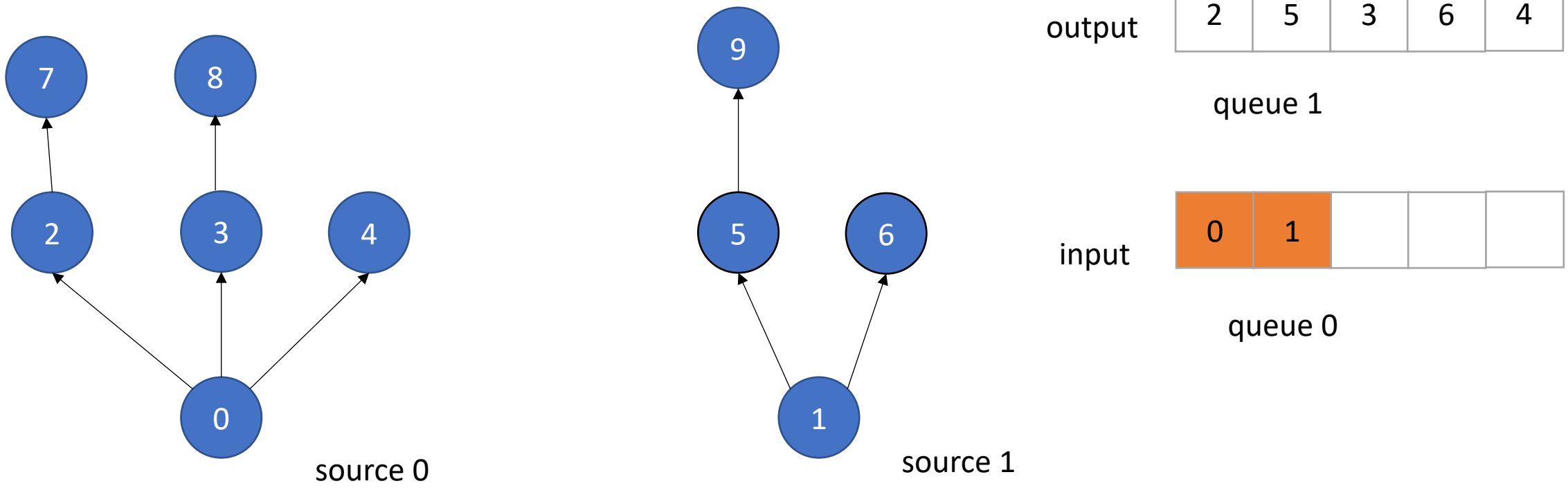
Input/Output Queues

- Example: Information flow in graph applications:



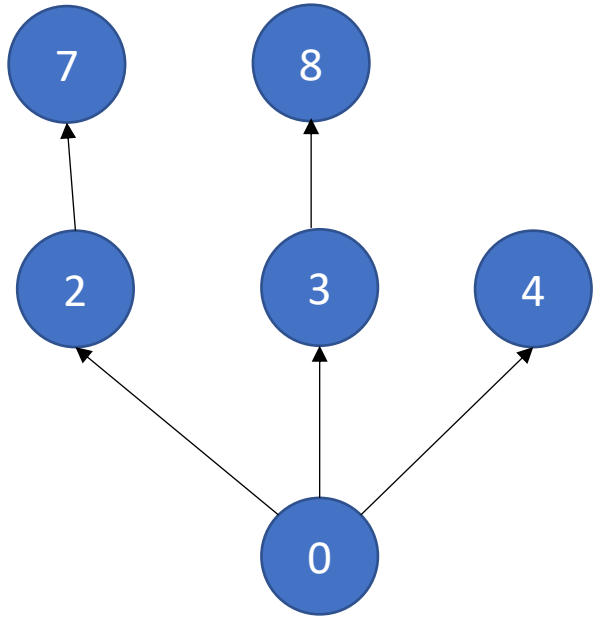
Input/Output Queues

- Example: Information flow in graph applications:

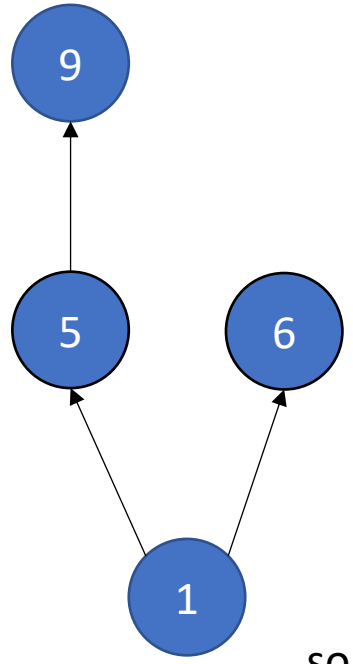


Input/Output Queues

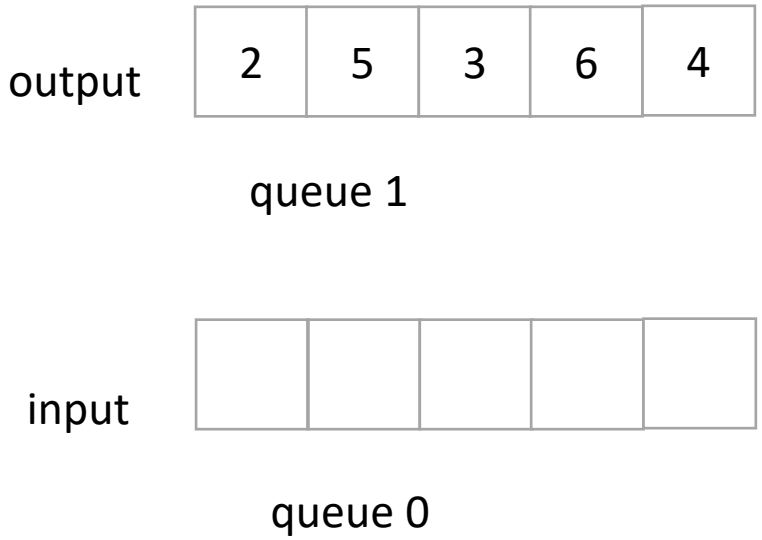
- Example: Information flow in graph applications:



source 0



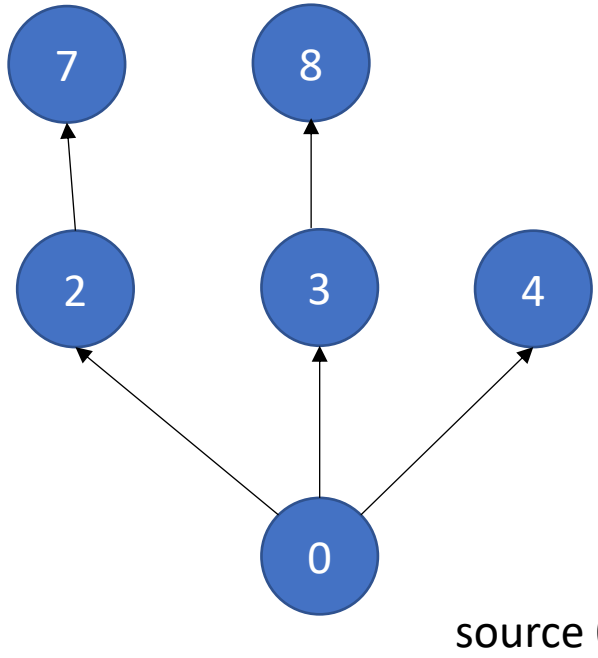
source 1

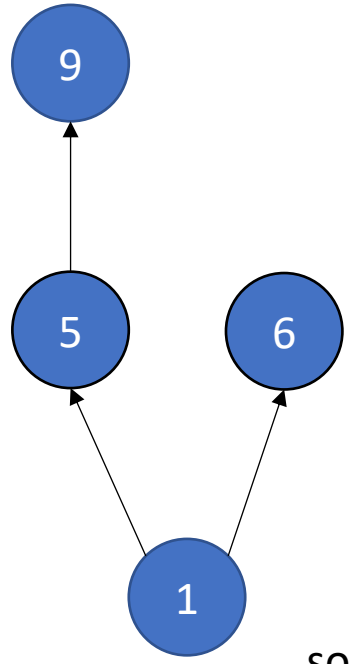


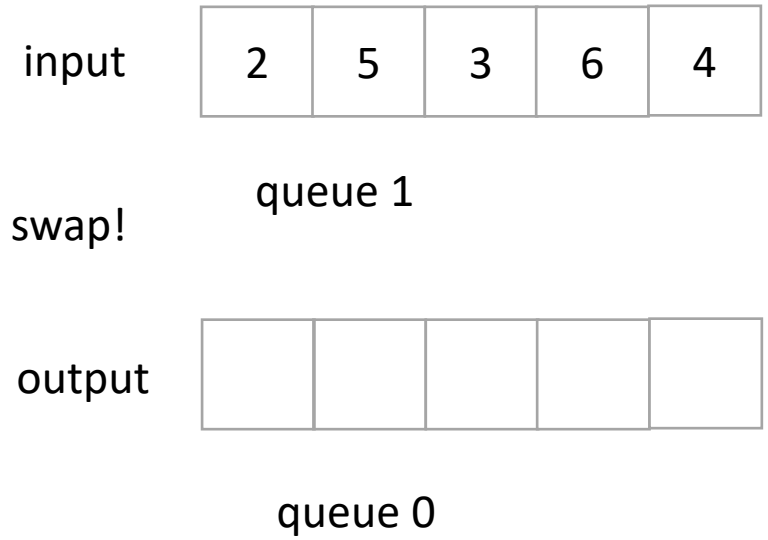
join all threads and clear input

Input/Output Queues

- Example: Information flow in graph applications:

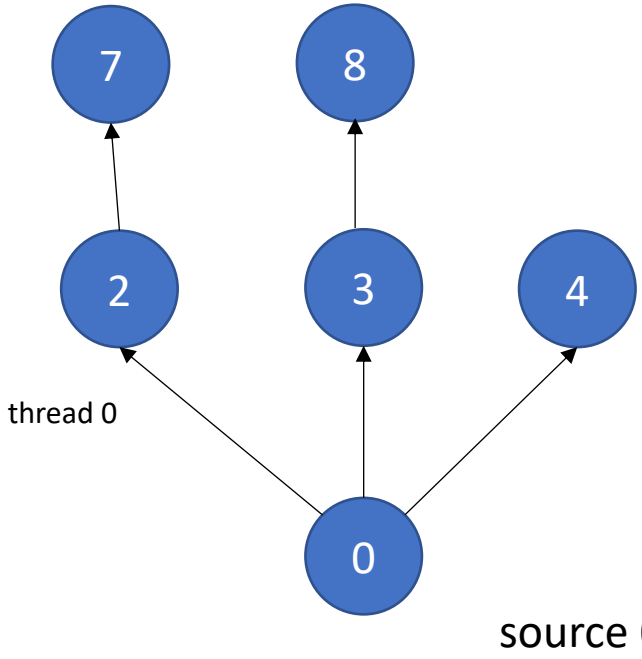


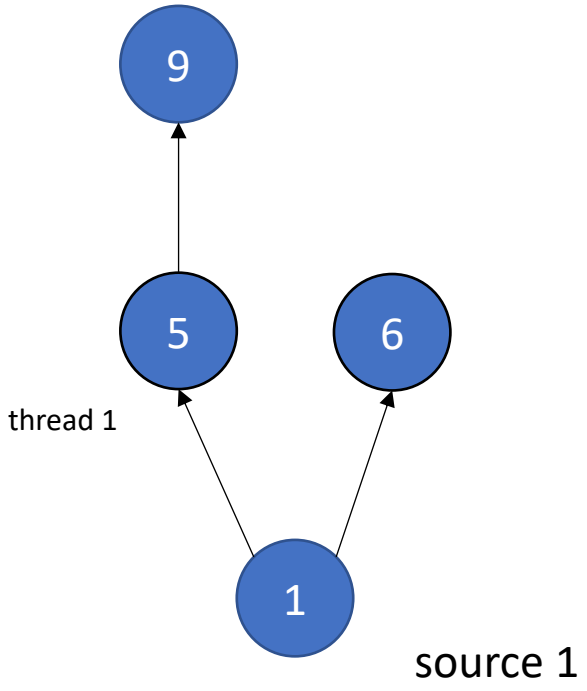


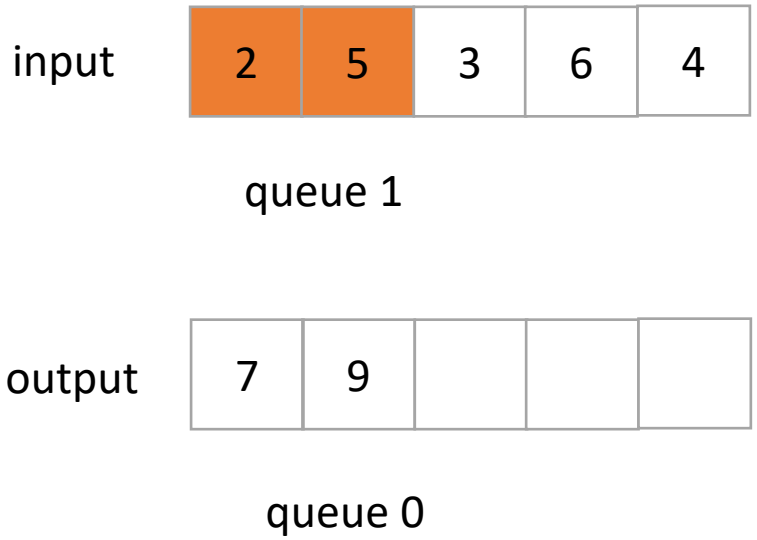


Input/Output Queues

- Example: Information flow in graph applications:

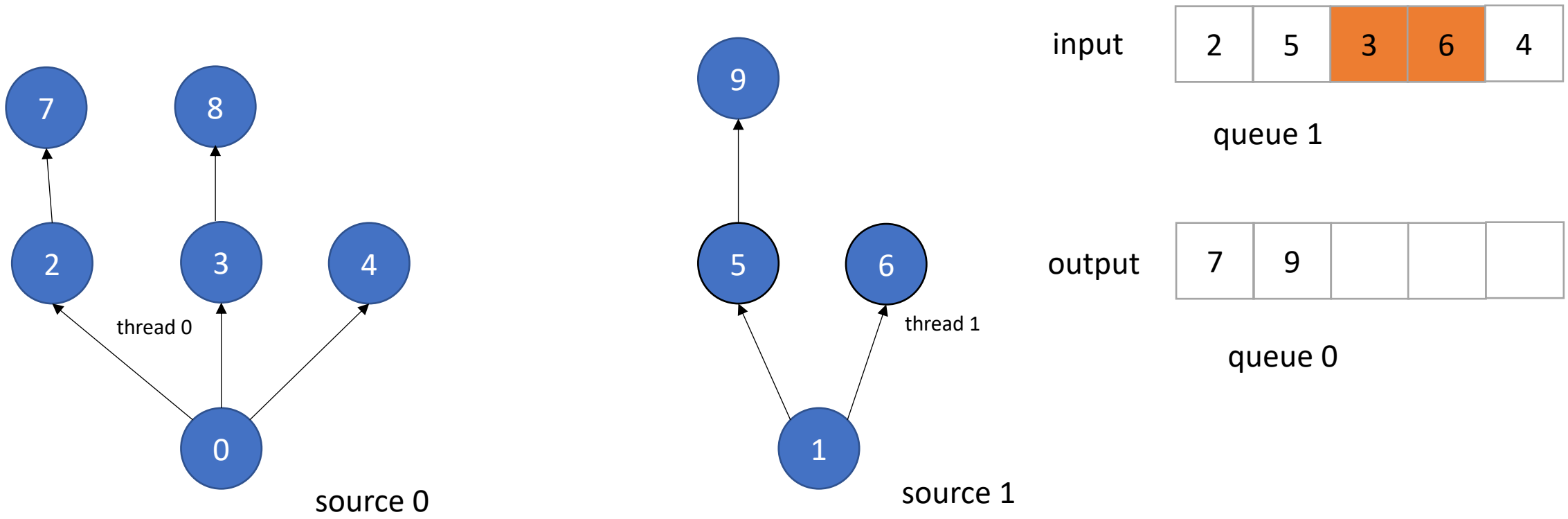






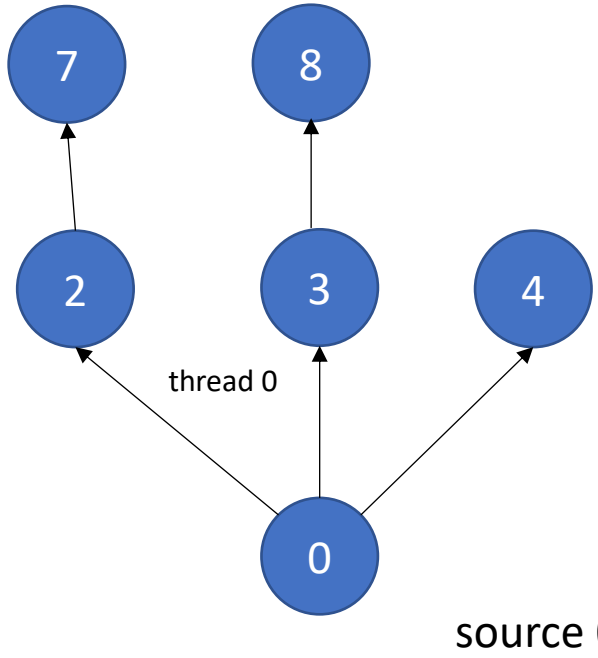
Input/Output Queues

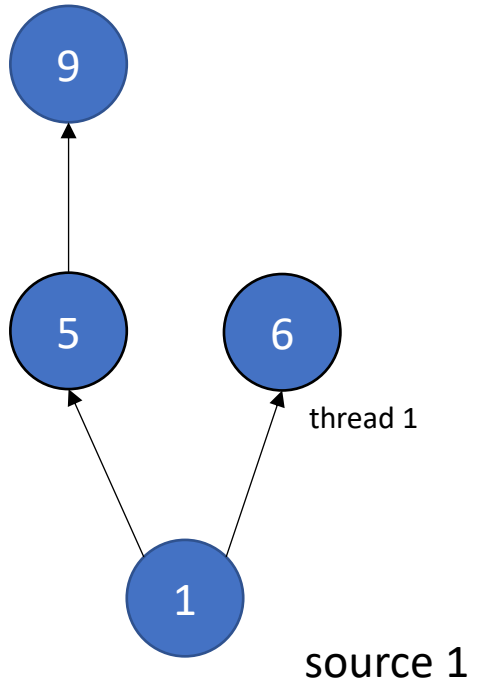
- Example: Information flow in graph applications:

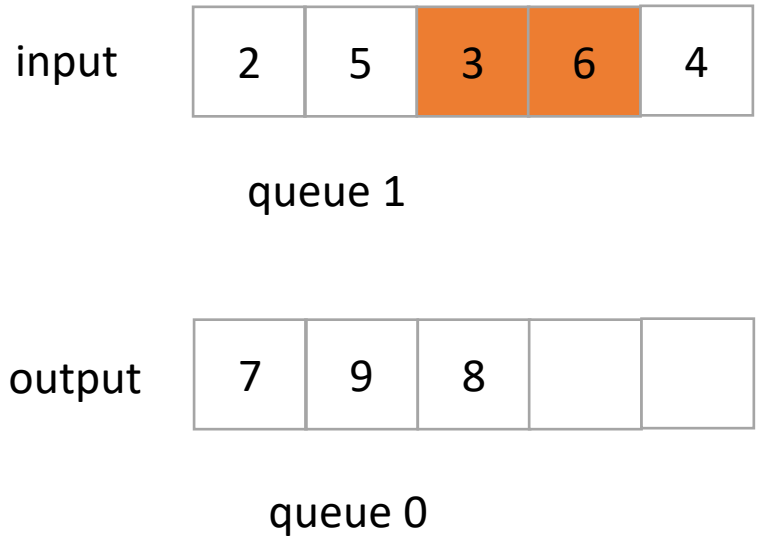


Input/Output Queues

- Example: Information flow in graph applications:

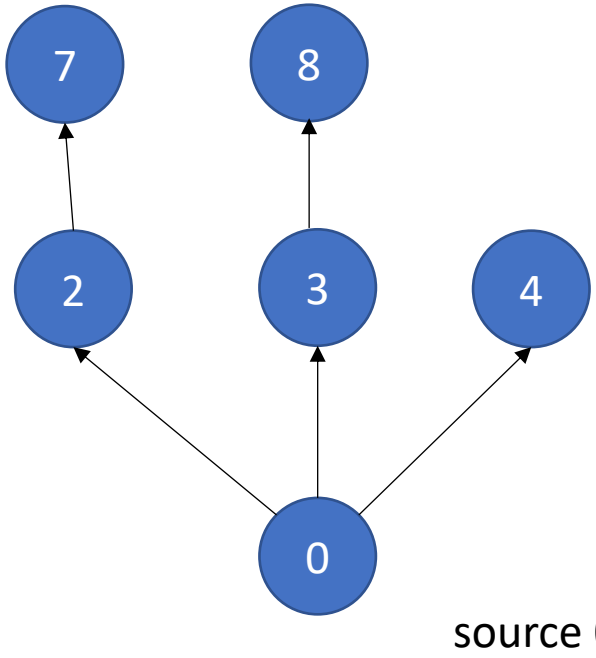


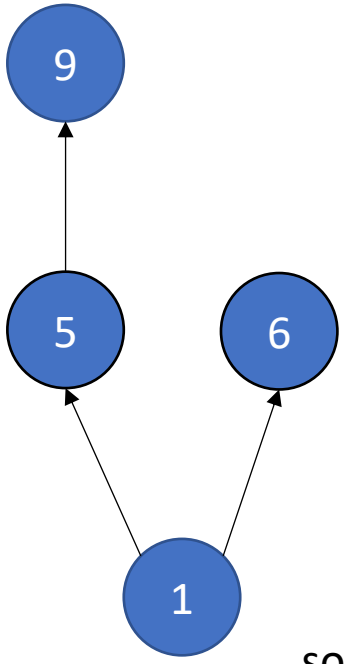


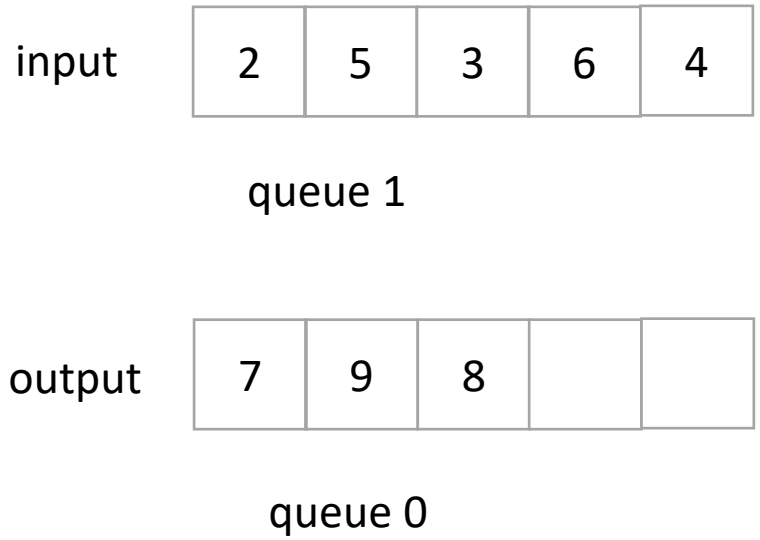


Input/Output Queues

- Example: Information flow in graph applications:

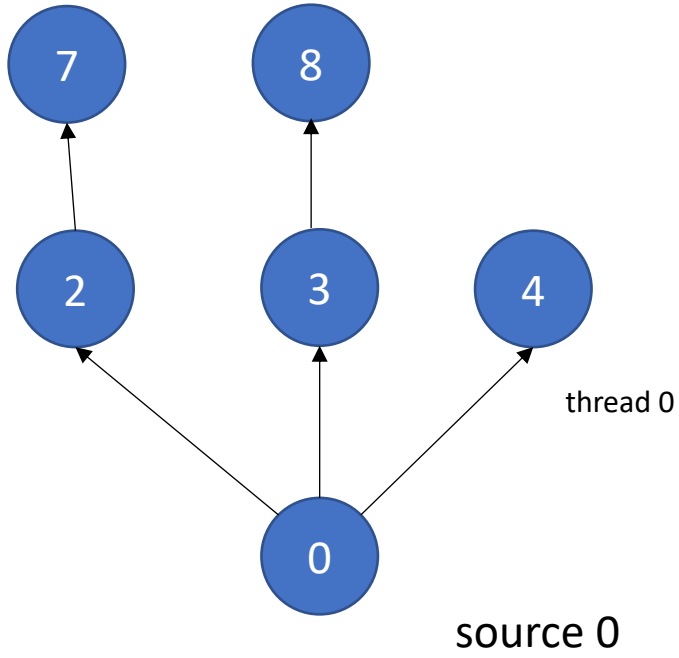


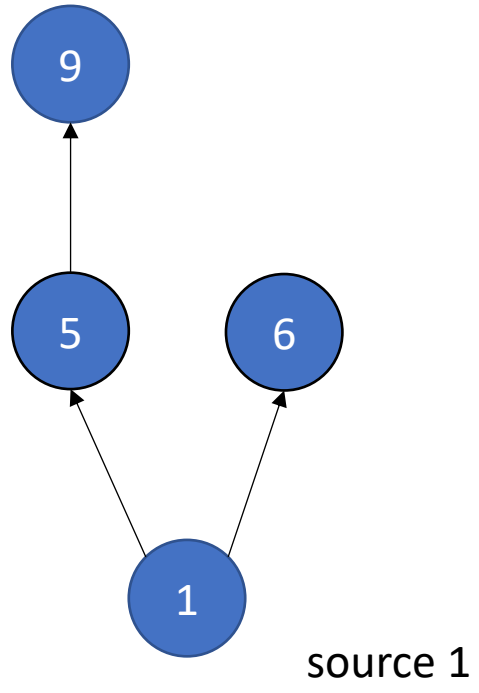


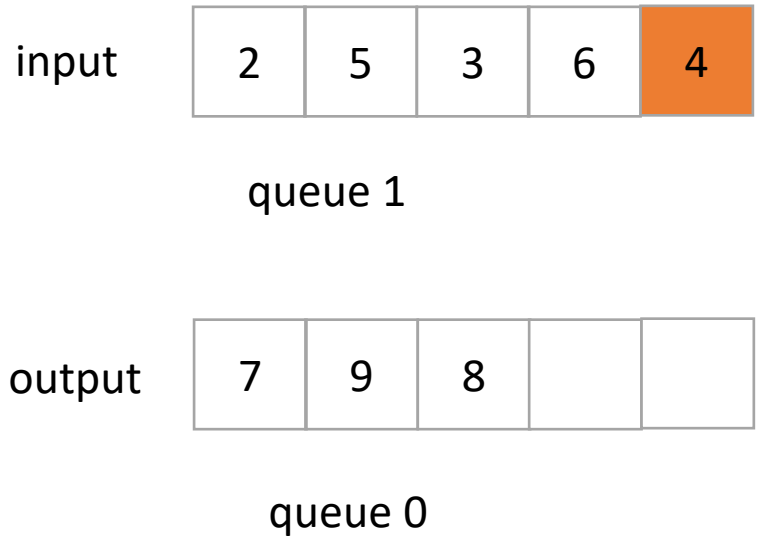


Input/Output Queues

- Example: Information flow in graph applications:

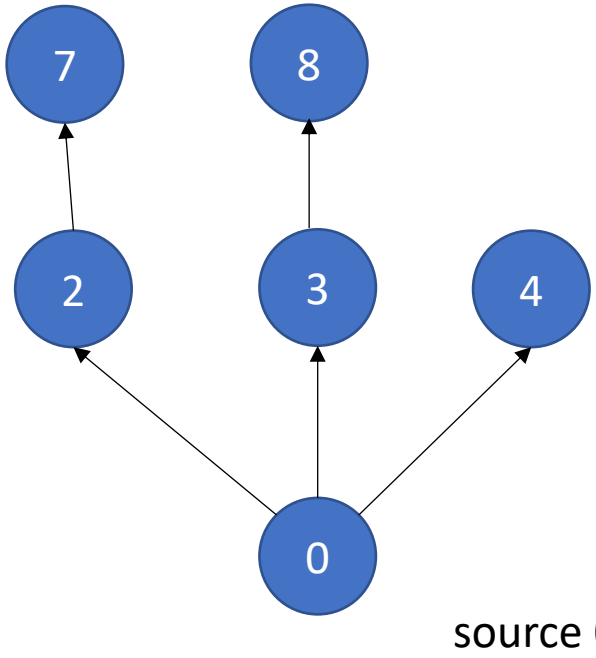


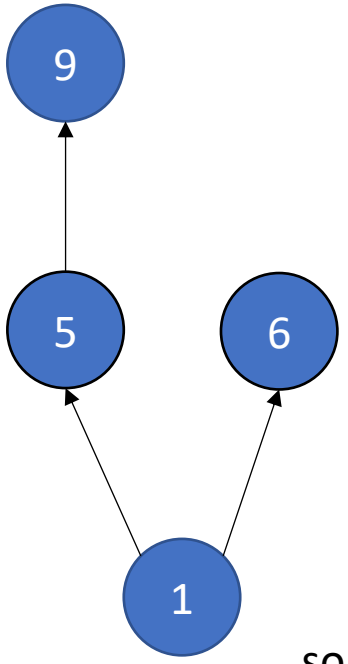


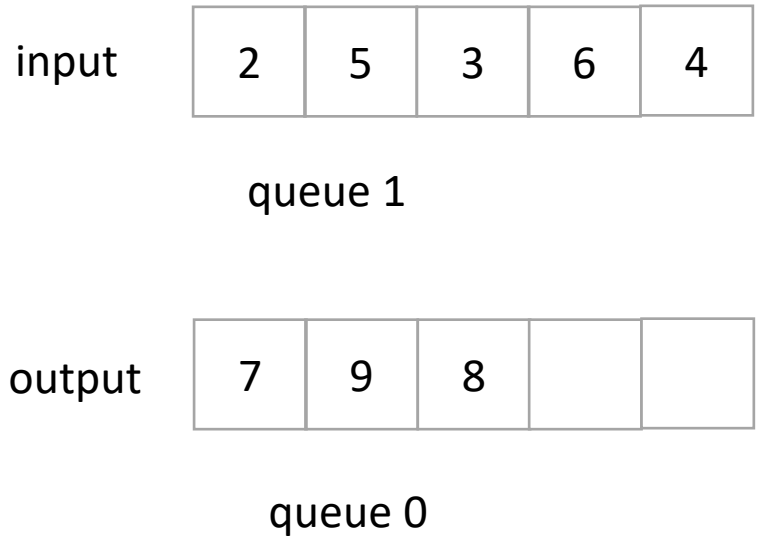


Input/Output Queues

- Example: Information flow in graph applications:

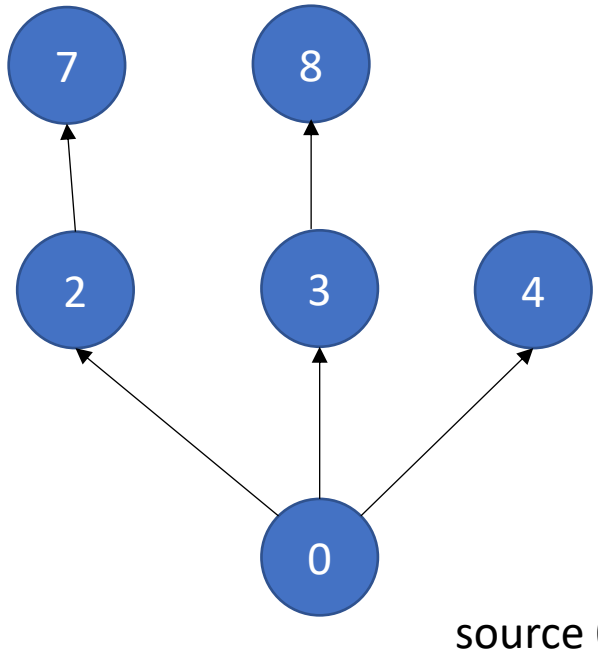


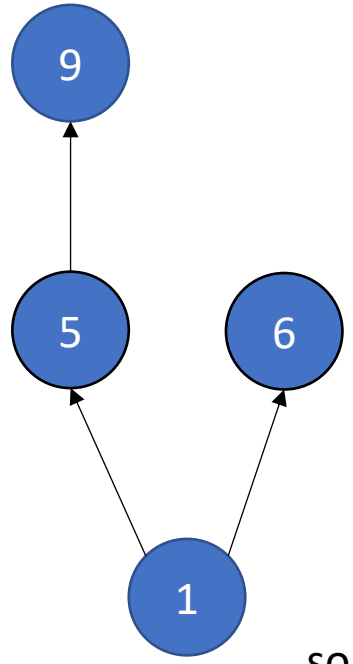




Input/Output Queues

- 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

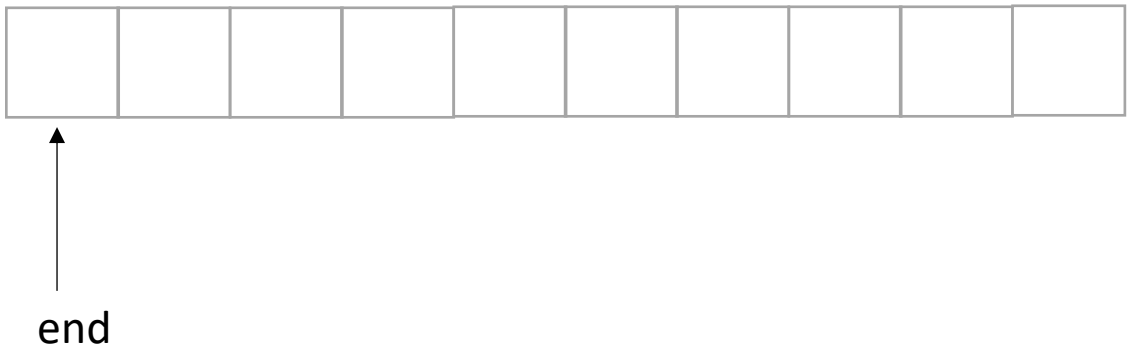
Cons:
- need to reason about overflow!

Note on terminology

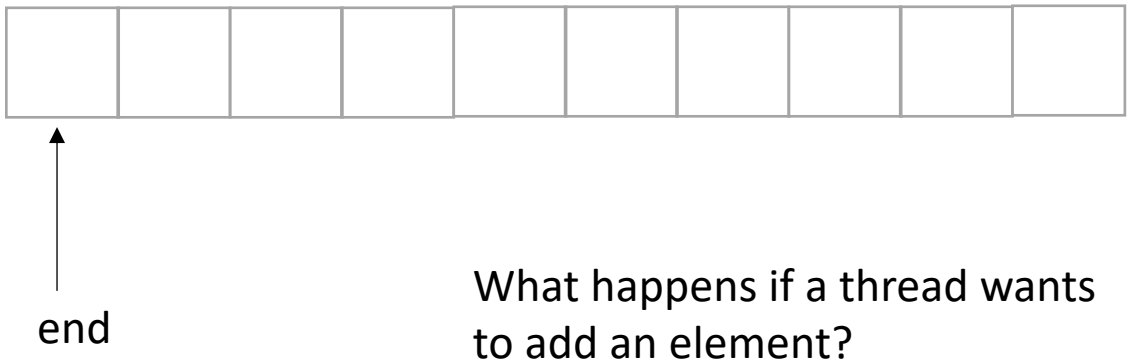
- Head/tail - often used in queue implementations, but switches when we start doing circular buffers.

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

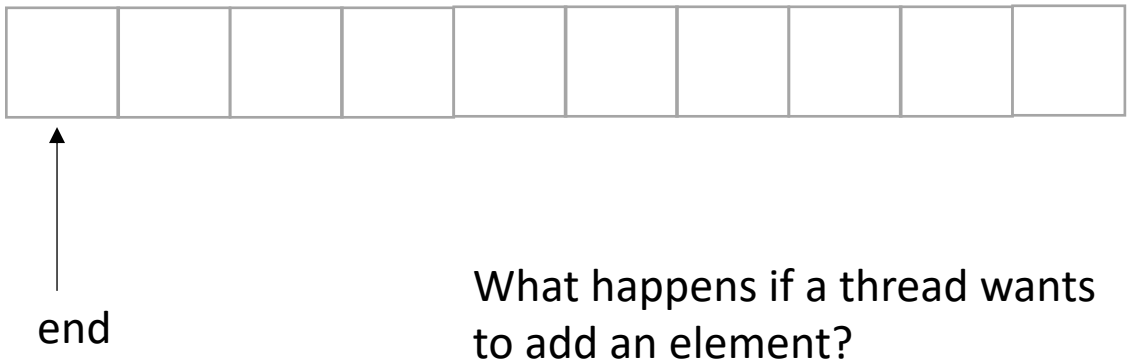
Implementation



Implementation



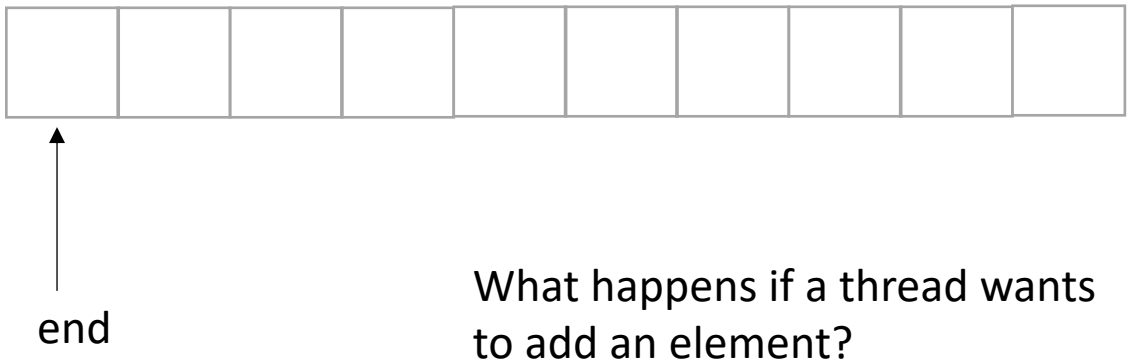
Implementation



What happens if a thread wants to add an element?

Think sequentially:

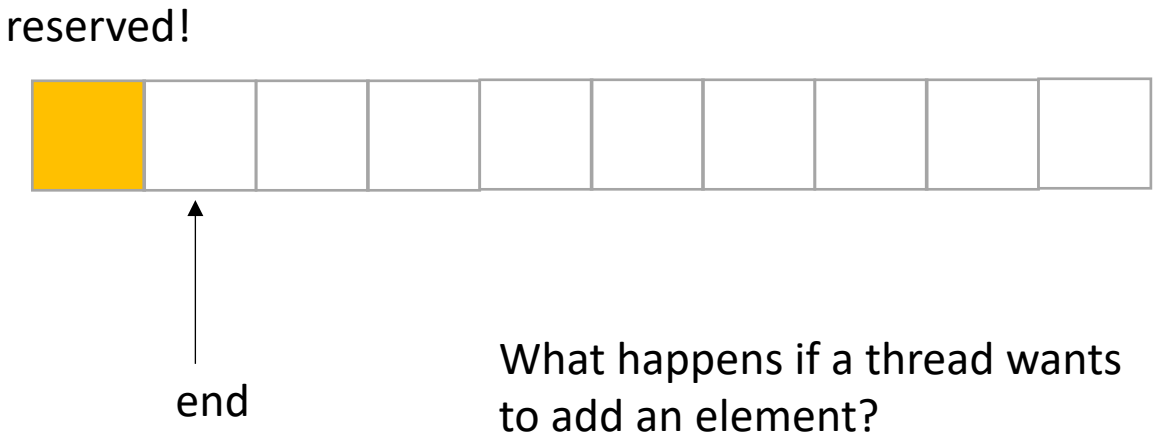
Implementation



What happens if a thread wants to add an element?

Think sequentially:
*reserve a space - increment end

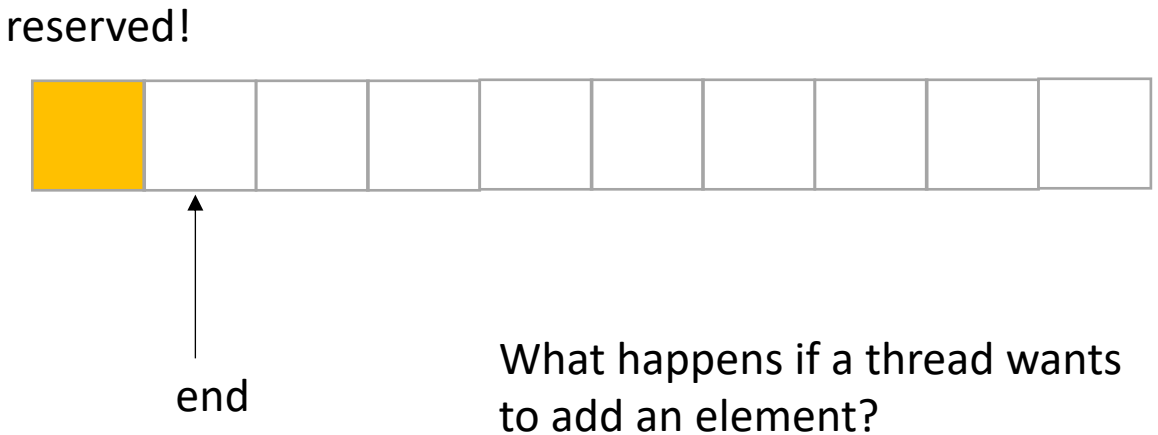
Implementation



What happens if a thread wants to add an element?

Think sequentially:
*reserve a space - increment end

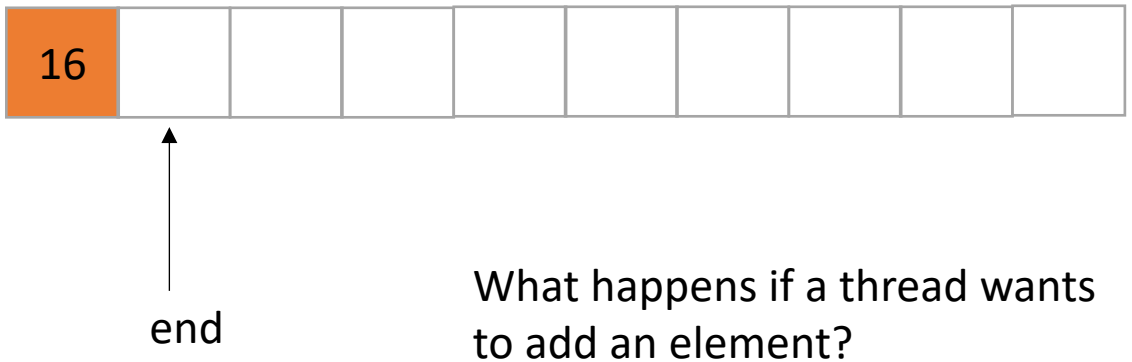
Implementation



What happens if a thread wants to add an element?

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

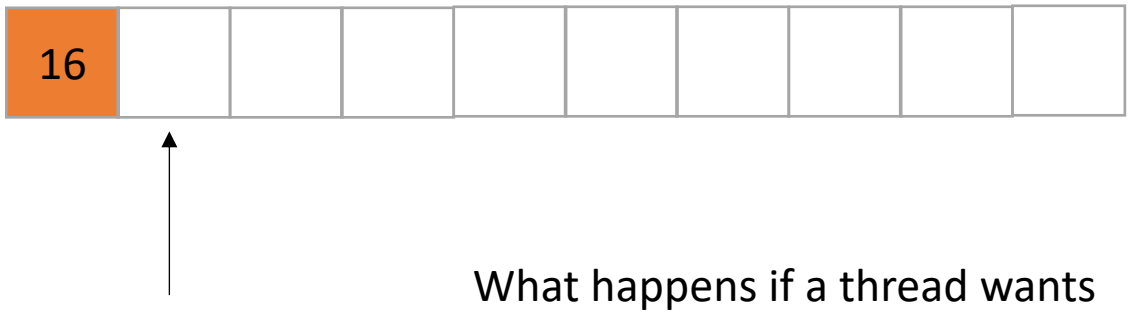
Implementation



What happens if a thread wants to add an element?

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

Implementation

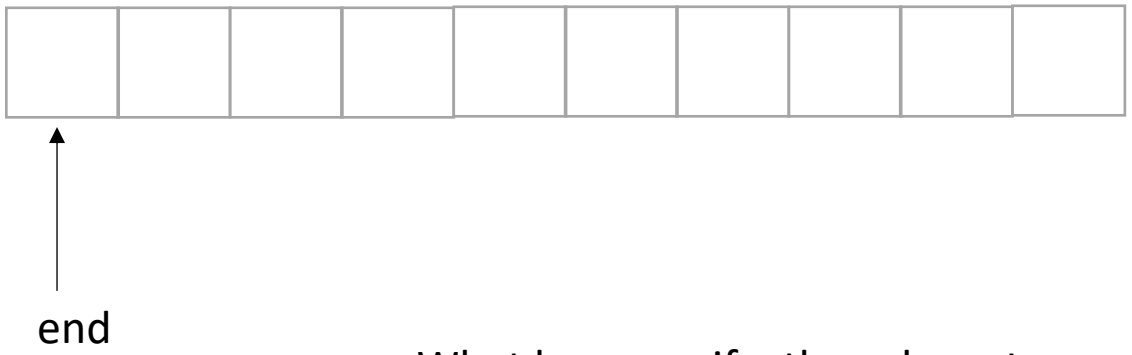


What happens if a thread wants to add an element?

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

done!

Implementation

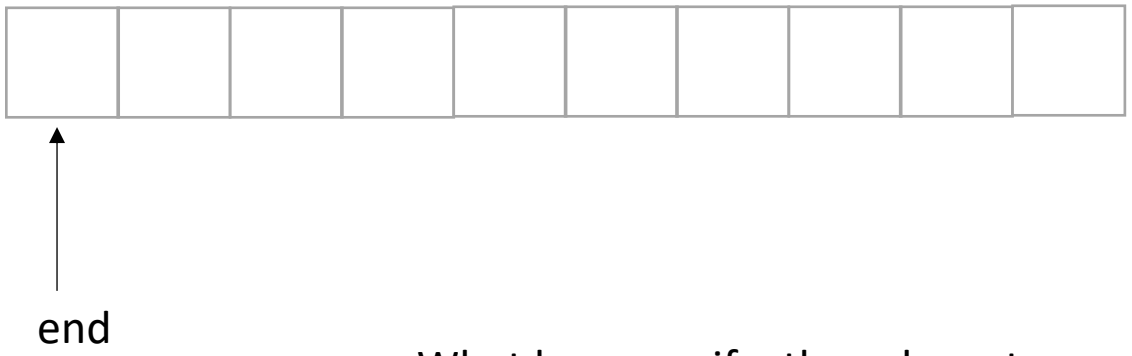


What happens if a thread wants
to add an element?

Think concurrently:

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

Implementation

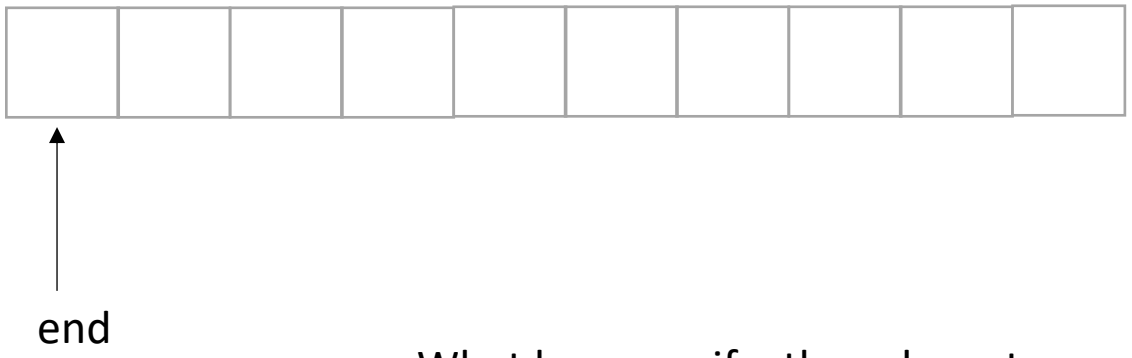


What happens if a thread wants to add an element?

Think concurrently:

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

Implementation



Thread 0:
enq(6);

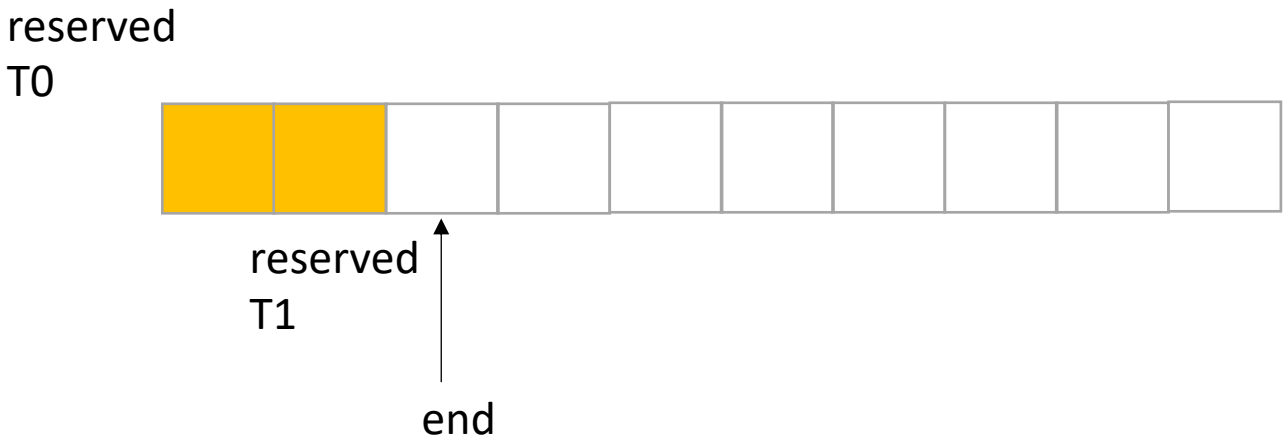
Thread 1:
enq(7);

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);

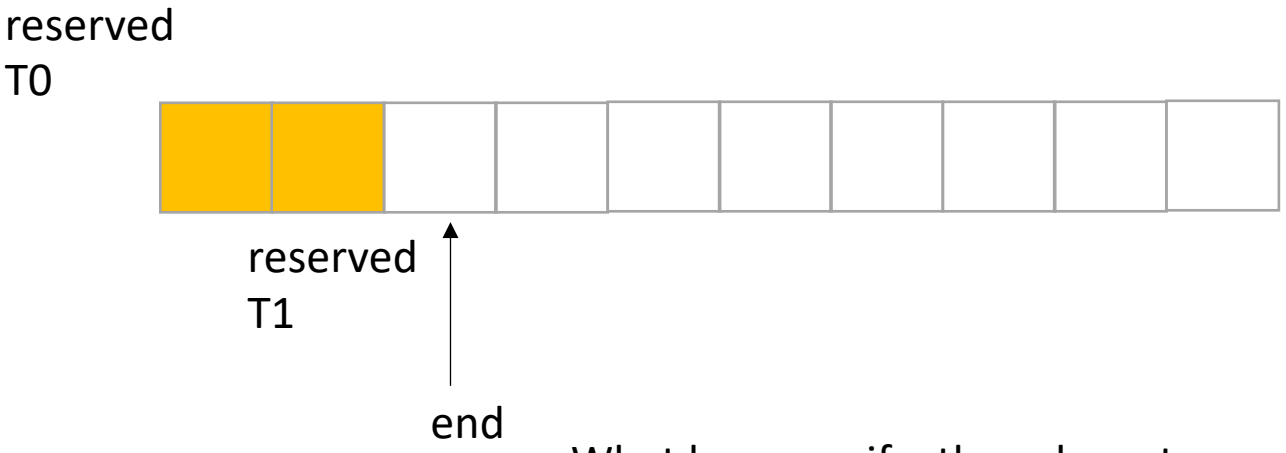
What happens if a thread wants to add an element?

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);

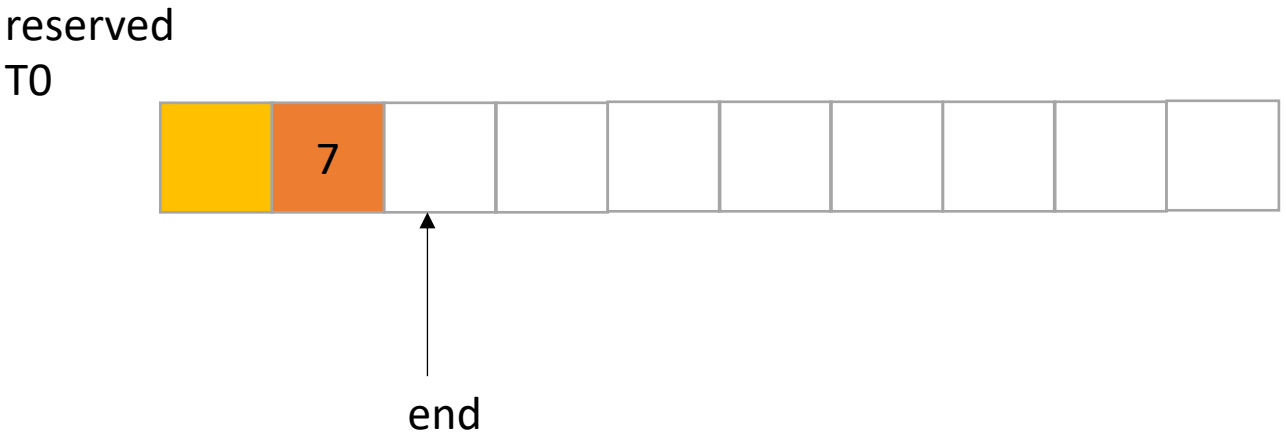
What happens if a thread wants to add an element?

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);

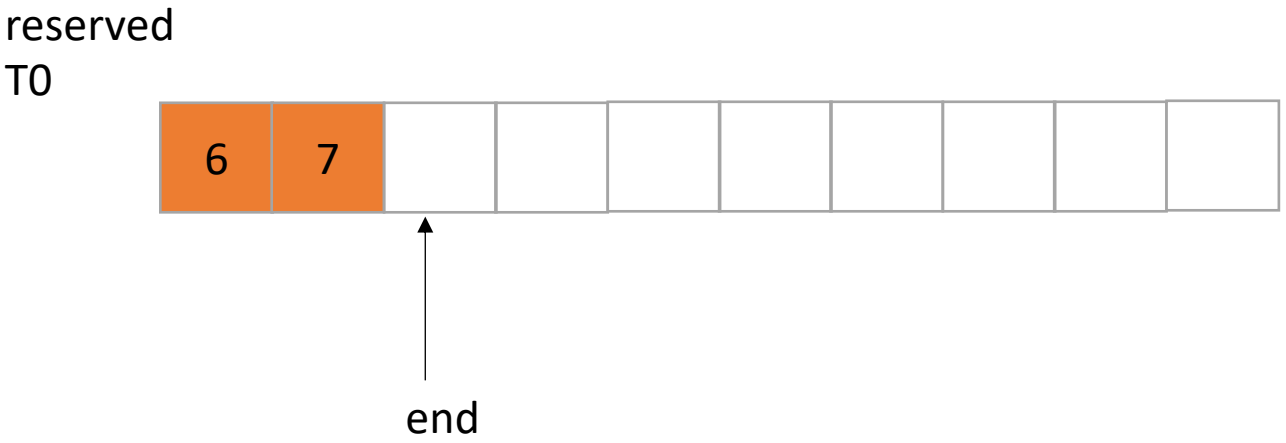
What happens if a thread wants to add an element?

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:

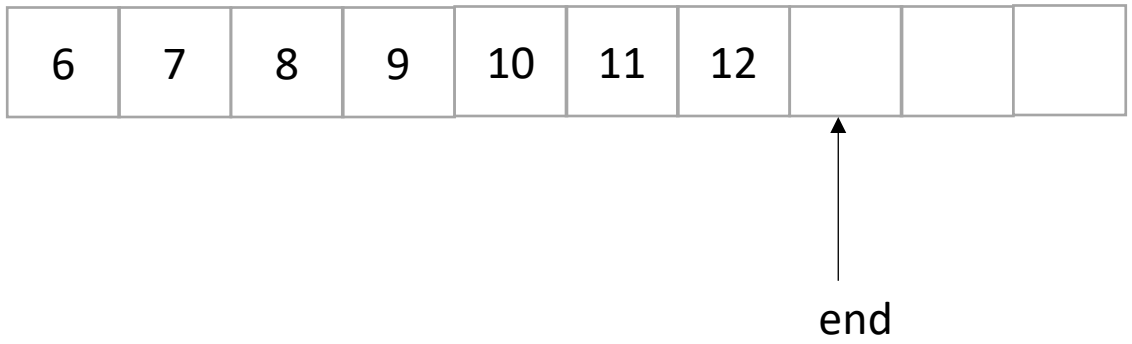
```
reserved_index = atomic_fetch_add(&end, 1);
```

```
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?

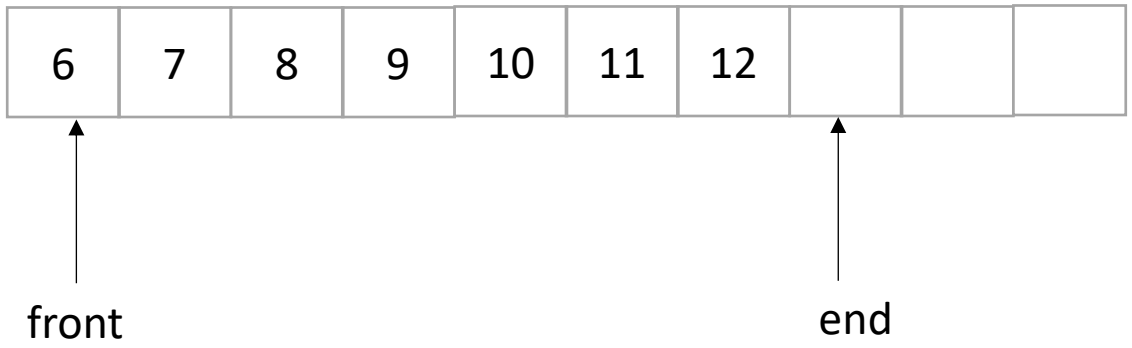
What about Input?

- Now we only do deqs



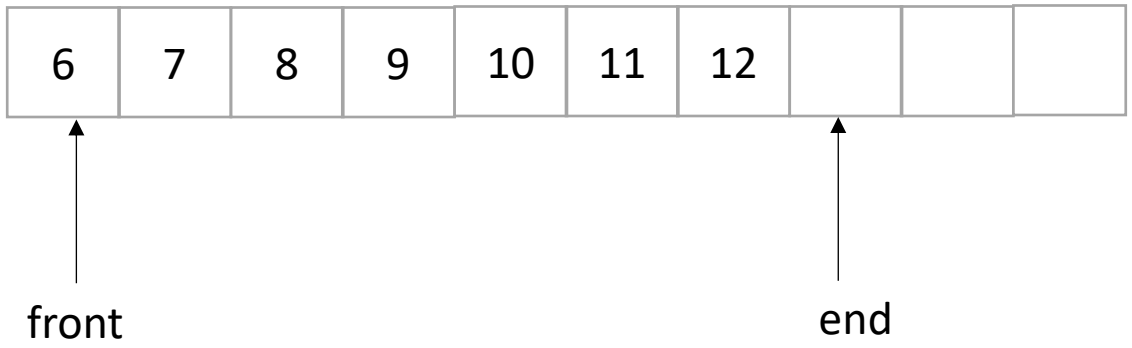
What about Input?

- Now we only do deqs



What about Input?

- Now we only do deqs



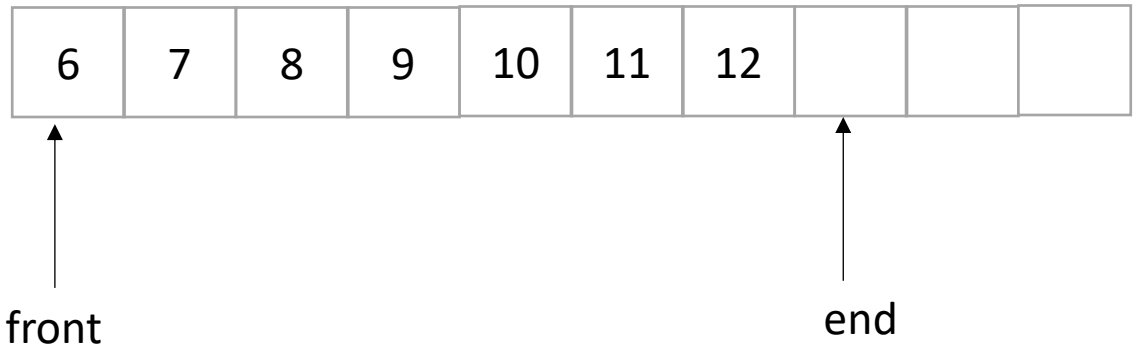
What happens if a thread wants to add an element?

Think concurrently:

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

What about Input?

- Now we only do deqs



Thread 0:
deq();

Thread 1:
deq();

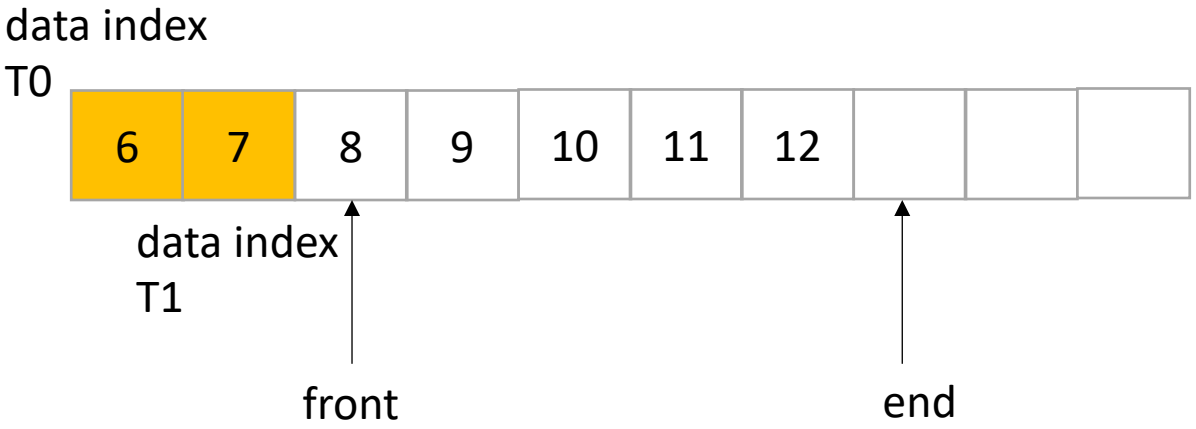
What happens if a thread wants to add an element?

Think concurrently:

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

What about Input?

- Now we only do deqs



Thread 0:
deq();

Thread 1:
deq();

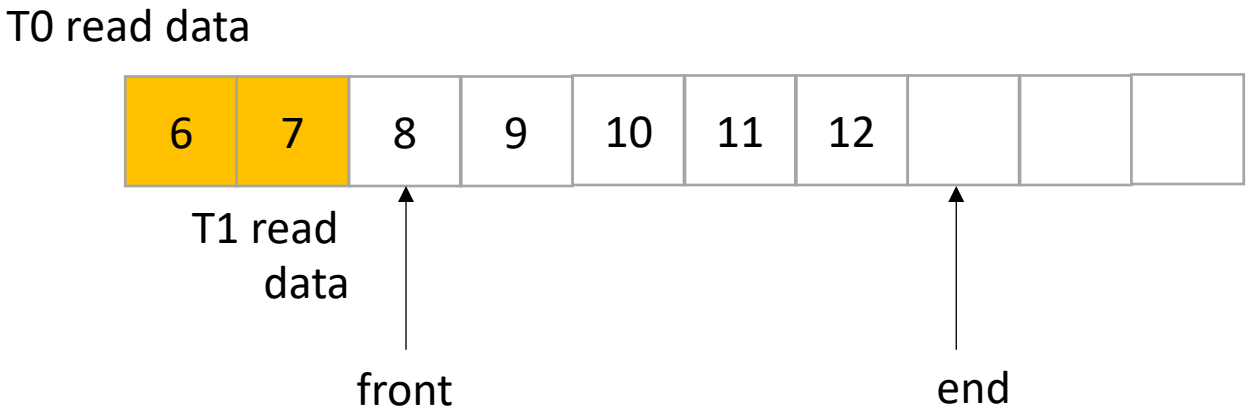
What happens if a thread wants to add an element?

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

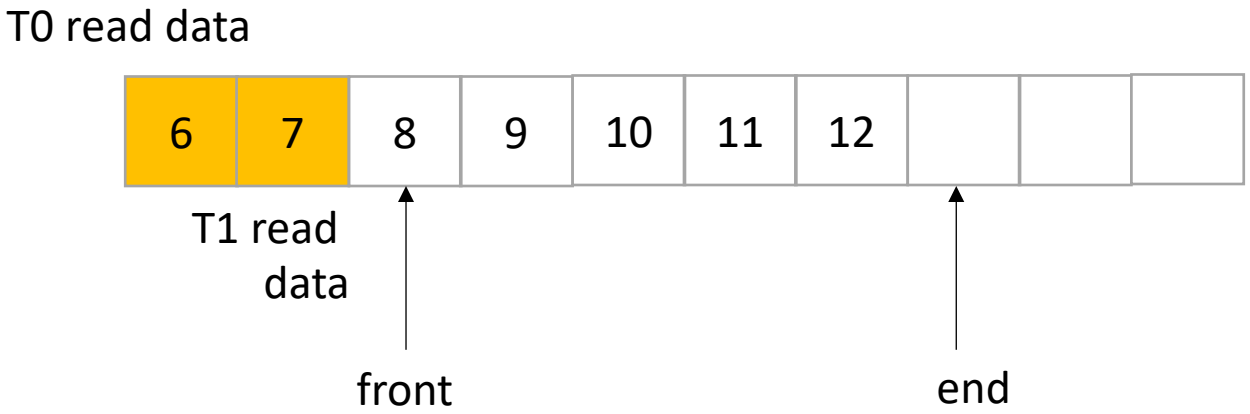
What happens if a thread wants to add an element?

Think concurrently:

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

What about Input?

- Now we only do deqs



How to implement a stack?

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?

See you on Friday!

- Work on HW 2!
  - Let us know if you need help! Piazza, office hours, etc.

- My office hours are tomorrow
  - They are hybrid (remote or in-person)
  - E2 - 233 (no nameplate still)
  - Sign-up sheet is still the format