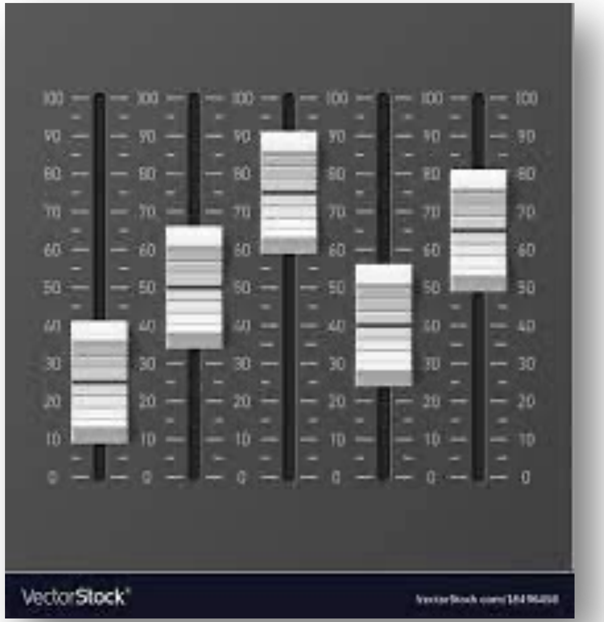
CSE113: Parallel Programming

Feb. 10, 2023

- **Topics**:
  - Concurrent data structure specifications
    - Sequential consistency
    - Breaking sequential consistency
    - Linearizability



Announcements

- Expect HW1 grades by Moday
  - Let us know if there are any issues ASAP
- Homework 2 was due, but you have until Monday
  - Please use office hours or piazza if you have questions
  - Remember, nights and weekends have no guarantees of responses.
- Homework 3 is scheduled for Monday release

Announcements

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

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

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;
};
```

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"

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();
}
```

The object is "thread safe"

```
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;
};
```

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;
};
```

The object is "non-locking"

3 dimensions for concurrent objects

- **Correctness**:
  - How should concurrent objects behave (Specification)
- **Performance**:
  - How to make things fast fast fast!
- **Fairness**:
  - Under what conditions can concurrent objects deadlock

Sequential Consistency

- Our first specification

Lets think about a Queue

What is a queue?

We consider 2 API functions:

- enq(value v) - enqueues the value v
- deq() - returns the value at the front of the queue

```
Queue<int> q;
q.enq(6);
int t = q.deq();
```

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

```
Queue<int> q;
q.enq(6);
q.enq(7);
int t = q.deq();
int t1 = q.deq();
```

Lets think about a Queue

What is a queue?

We consider 2 API functions:

- enq(value v) - enqueues the value v
- deq() - returns the value at the front of the queue

```
Queue<int> q;
int t = q.deq();
```

Let’s say: *value of 0*

Lets think about a Queue

This is called a sequential specification:

The sequential specification is nice! We want to base our concurrent specification on the sequential specification

We will have to deal with the non-determinism of concurrency

Thinking about a concurrent queue

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

Thinking about a concurrent queue

Global variable:

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

Lets call our concurrent queue “CQueue”

Thread 0:

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

Thinking about a concurrent queue

Global variable:

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

Thread 0:

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

Thread 1:

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

what can be stored in t after this concurrent program?

Thinking about a concurrent queue

Global variable:

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

Thread 0:

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

Thread 1:

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

what can be stored in t after this concurrent program?

Can t be 256?

Thinking about a concurrent queue

Global variable:

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

Thread 0:

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

Thread 1:

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

what can be stored in t after this concurrent program?

Can t be 256? it should be one of {None, 6, 7}

Global variable:

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

Thread 0:

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

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

Thread 1:

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

Global variable:

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

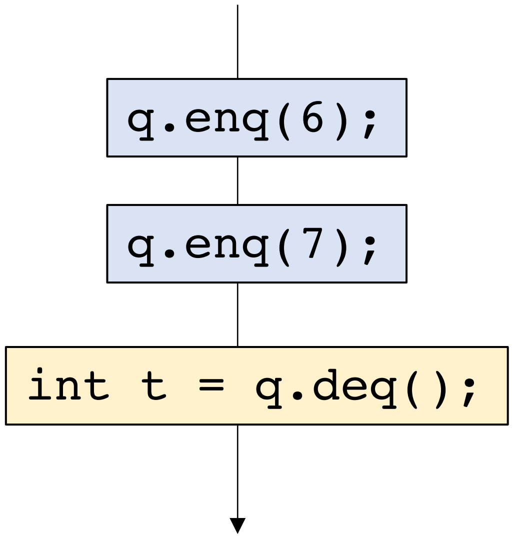
Thread 0:

```
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

Global variable:

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

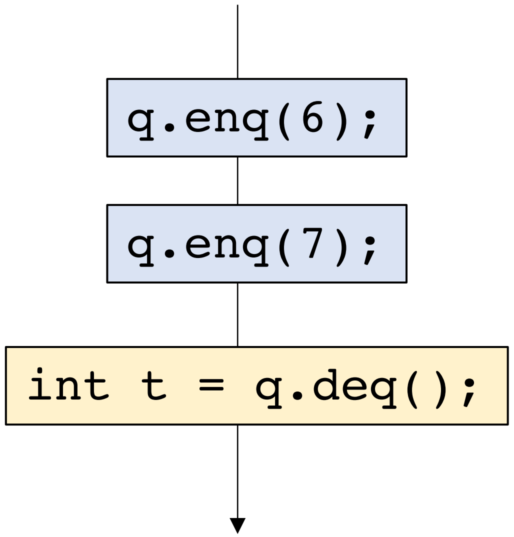
Thread 0:

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

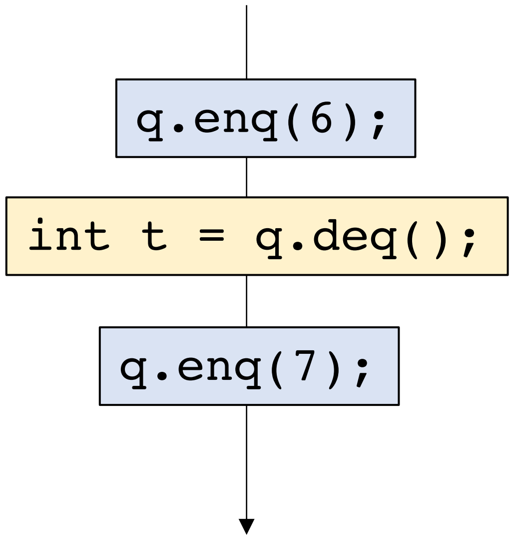
Thread 1:

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

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



t is 6



t is 6

Global variable:

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

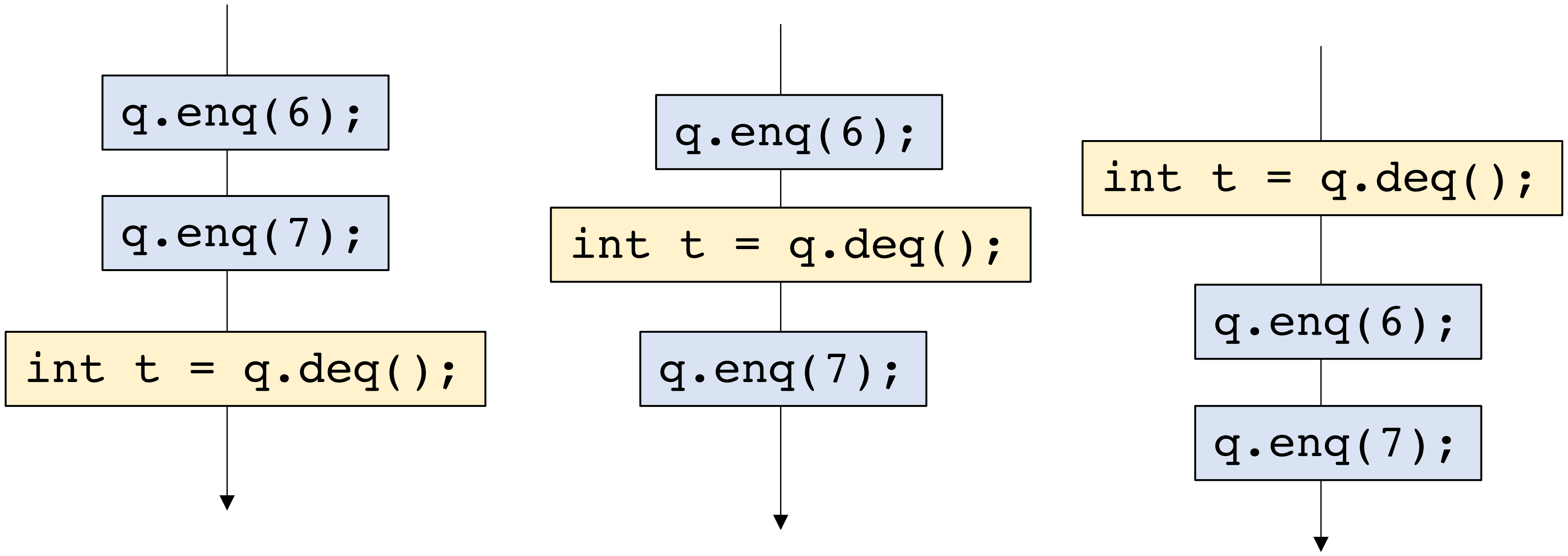
Thread 0:

```
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

Global variable:

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

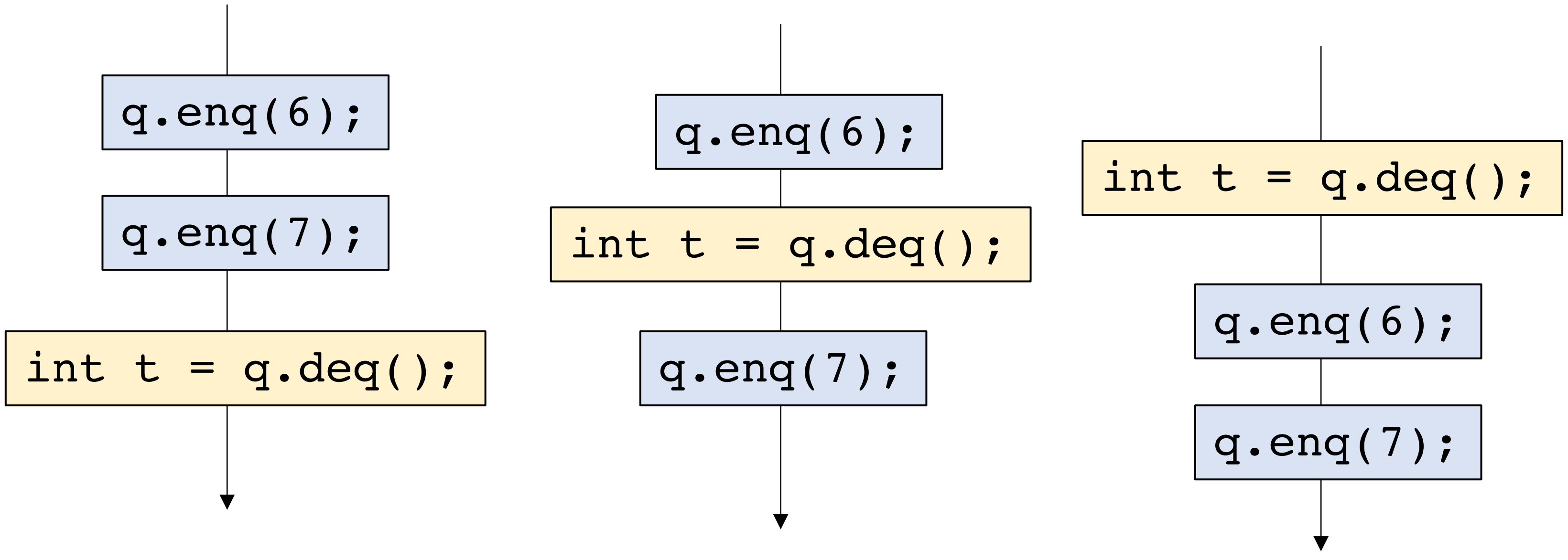
Thread 0:

```
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?

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:

Can t ever be 7?

Global variable:

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

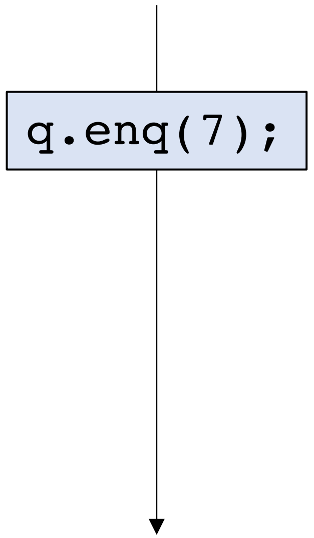
Thread 0:

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

Thread 1:

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

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



Can t ever be 7?

Global variable:

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

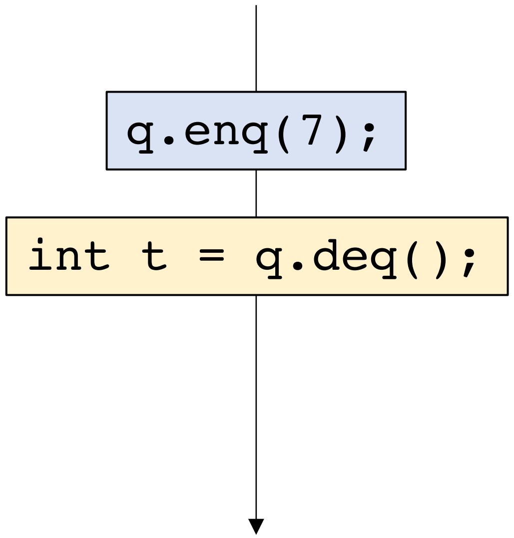
Thread 0:

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

Thread 1:

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

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



Can t ever be 7?

Global variable:

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

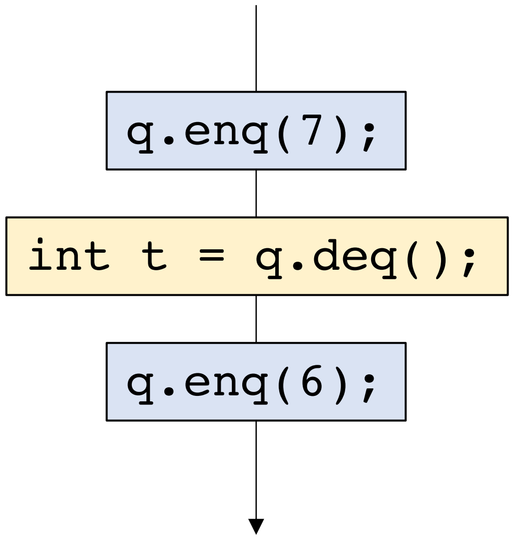
Thread 0:

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

Thread 1:

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

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



Can t ever be 7?

Global variable:

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

Thread 0:

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

Thread 1:

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

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

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

The events of Thread 0 don't appear in the same order of the program!

This should not be allowed!

Can t ever be 7?

Sequential Consistency

- Valid executions correspond a sequentialization of object method calls

- The sequentialization must respect per-thread ”program order”, the order in which the object method calls occur in the thread

- Events across threads can interleave in any way possible

Sequential Consistency

- Valid executions correspond a sequentialization of object method calls

- The sequentialization must respect per-thread ”program order”, the order in which the object method calls occur in the thread

- Events across threads can interleave in any way possible

How many possible interleavings?
Combinatorics question:

if Thread 0 has N events
if Thread 1 has M events

$$\frac{(N+M)!}{N!\,M!}$$

Sequential Consistency

How many possible interleavings?
Combinatorics question:

if Thread 0 has N events
if Thread 1 has M events

$$\frac{(N+M)!}{N!\,M!}$$

Reminder that N and M are events, not instructions

Sequential Consistency

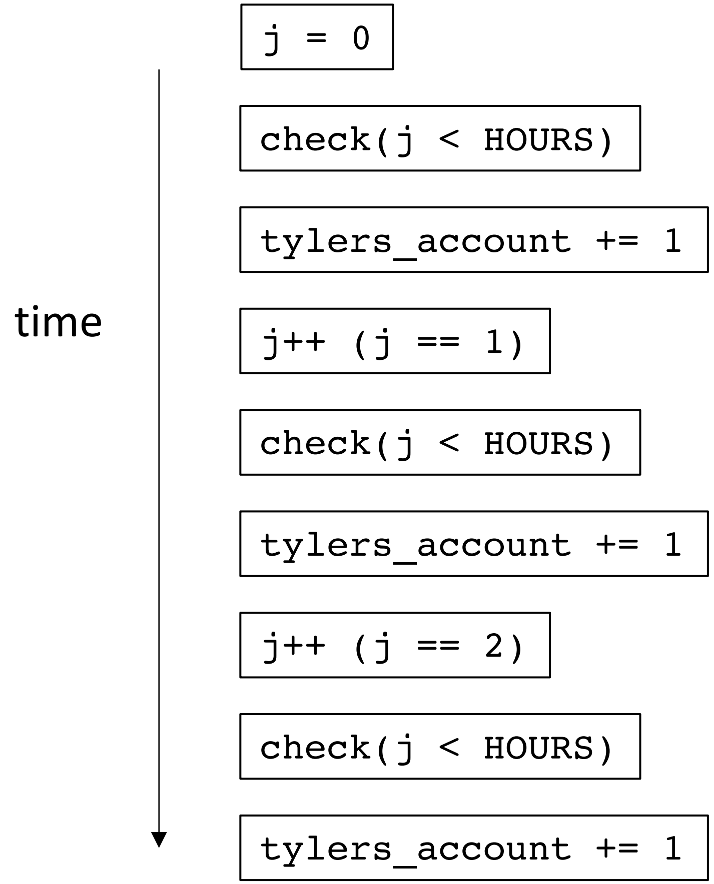
Tyler's employer

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

How many possible interleavings?
Combinatorics question:

if Thread 0 has N events
if Thread 1 has M events

$$\frac{(N+M)!}{N!\,M!}$$



Reminder that N and M are events, not instructions

If N and M execute 150 events each, there are more possible executions than particles in the observable universe!

Don’t think about all possible interleavings!

- Higher-level reasoning:
  - I get paid 100 times and buy 100 coffees, I should break even
  - If you enqueue 100 elements to a queue, you should be able to dequeue 100 elements
- Reason about a specific outcome
  - Find an interleaving that allows the outcome
  - Find a counter example

Reasoning about concurrent objects

To show that an outcome is possible, simply construct the sequential sequence

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();
```

```
int t0 = q.deq();
q.enq(6);
q.enq(7);
int t1 = q.deq();
```

Can t0 == 0 and t1 == 6?

Valid execution!

Are there others?

Global variable:

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

Lets do another!

Thread 0:

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

Thread 1:

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

Can t0 == 6 and t1 == 7?

Global variable:

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

Lets do another!

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 == 6 and t1 == 7?

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);
int t0 = q.deq();
q.enq(7);
int t1 = q.deq();
```

Found one! Are there others?

Can t0 == 6 and t1 == 7?

Reasoning about concurrent objects

To show that an outcome is possible, simply construct the sequential sequence

To show that an outcome is ***impossible*** show that there is no possible sequential sequence

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 == 7?

Global variable:

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

Thread 0:

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

Thread 1:

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

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

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

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

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

Can t0 == 0 and t1 == 7?

Global variable:

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

Thread 0:

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

Thread 1:

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

```
int t0 = q.deq();
q.enq(7);
int t1 = q.deq();
```

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

No place for this event to go!

Can t0 == 0 and t1 == 7?

One more example

Global variable:

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

Thread 0:

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

Thread 1:

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

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

Global variable:

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

Thread 0:

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

Thread 1:

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

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

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

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

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

Do we have our specification?

- Is sequential consistency a good enough specification for concurrent objects?

- It’s a good first step, but relative timing interacts strangely with absolute time.

- We will need something stronger.

Schedule

- **Problems with sequential consistency**

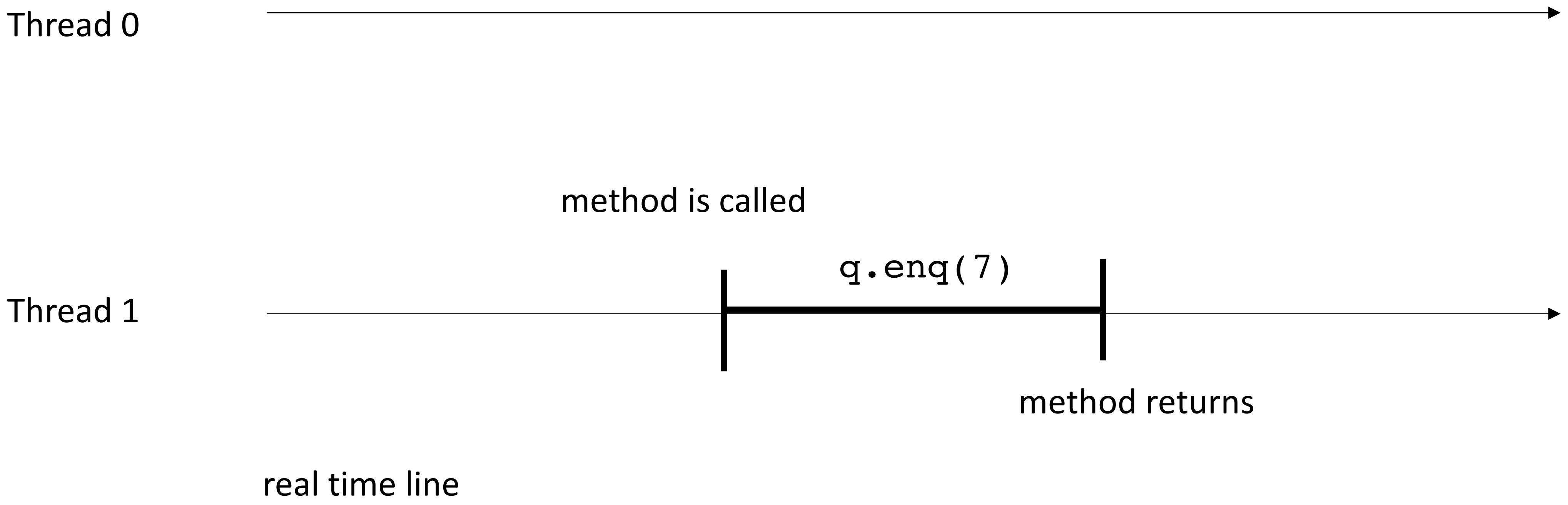
- Linearizability

- Specialized concurrent queues

Sequential consistency and real time

- Add in real time:

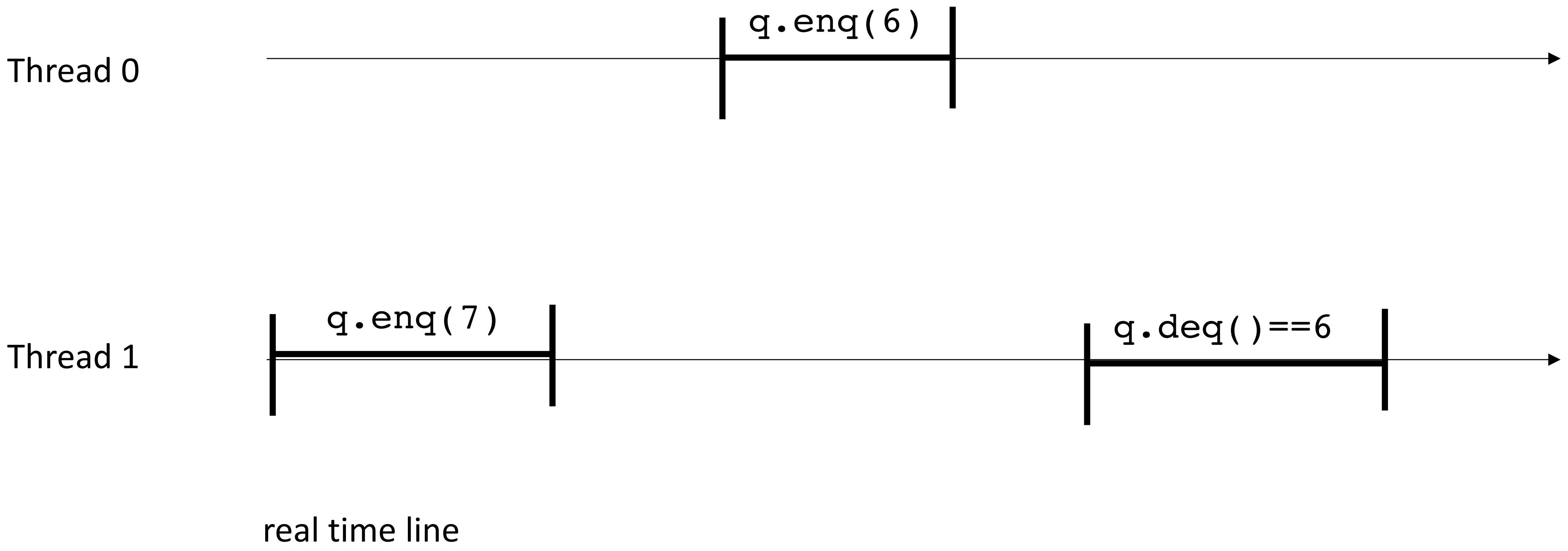
each method as a start, and end time stamp



Sequential consistency and real time

- Add in real time:

This timeline seems strange...

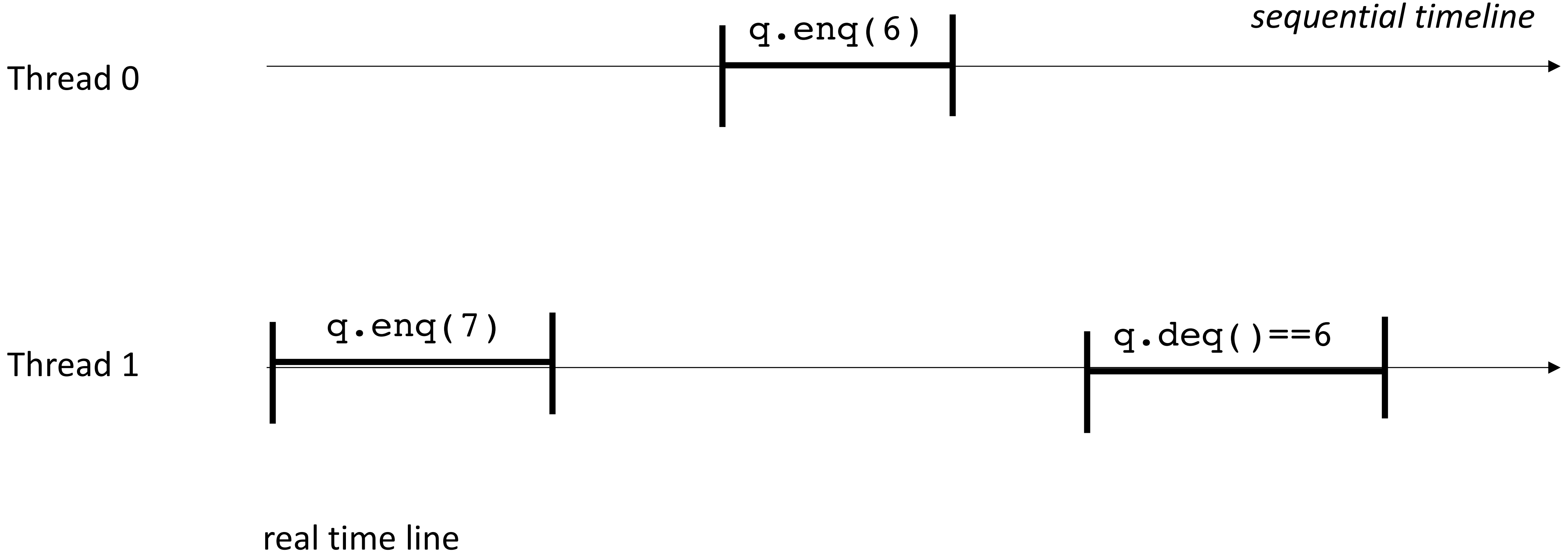


Sequential consistency and real time

- 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

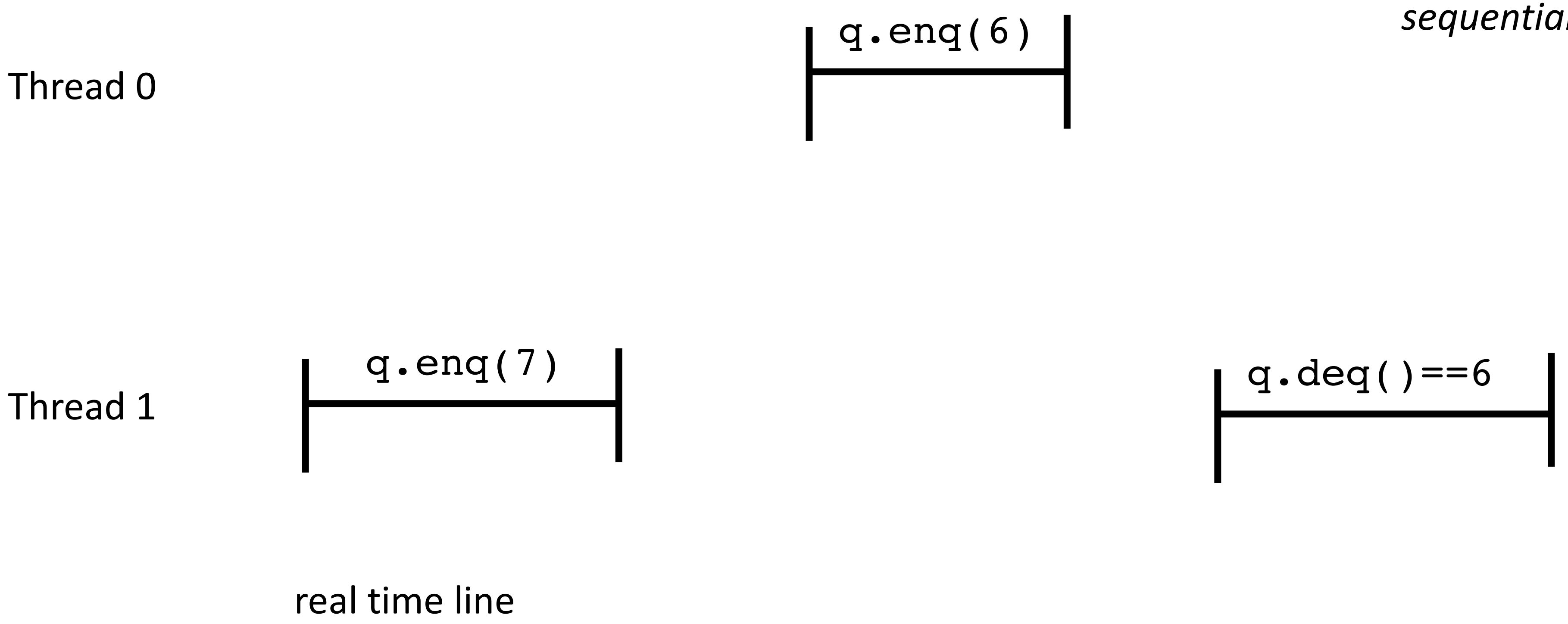


Sequential consistency and real time

- 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

`q.enq(6)`

Thread 0

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

Thread 1

real time line

Sequential consistency and real time

- Add in real time:

This execution is allowed in sequential consistency!

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

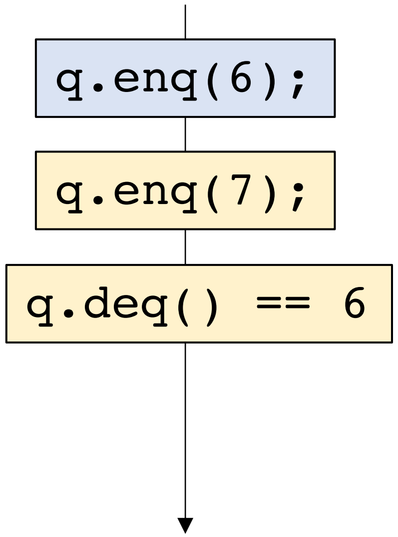
`q.enq(6)`

Thread 0

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

Thread 1

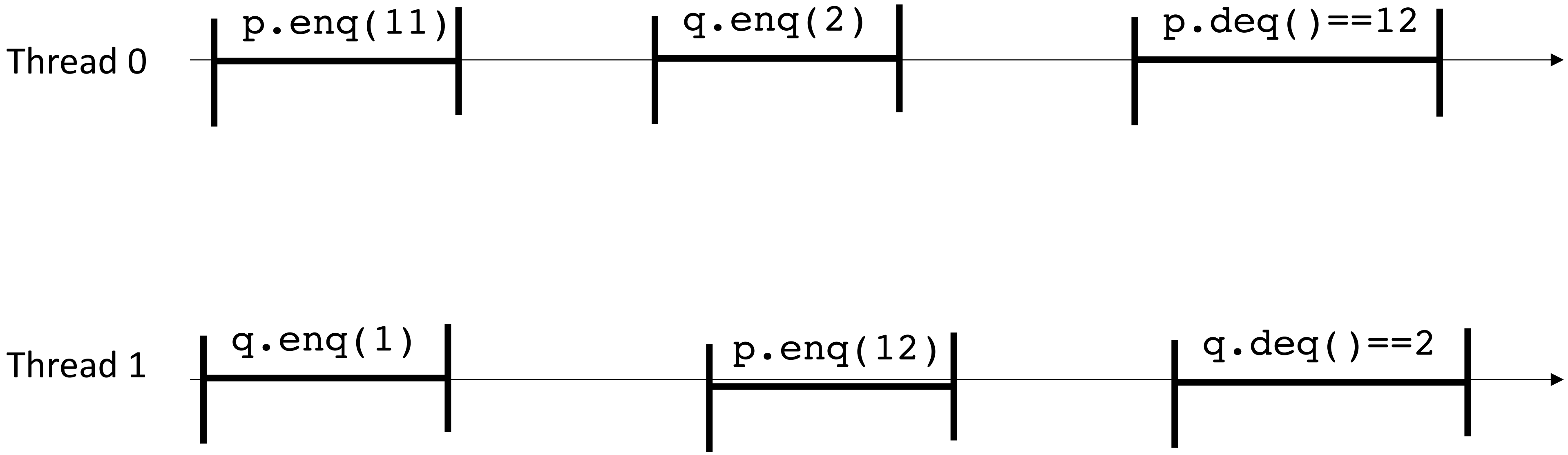
real time line



Sequential consistency and real time

- Add in real time:

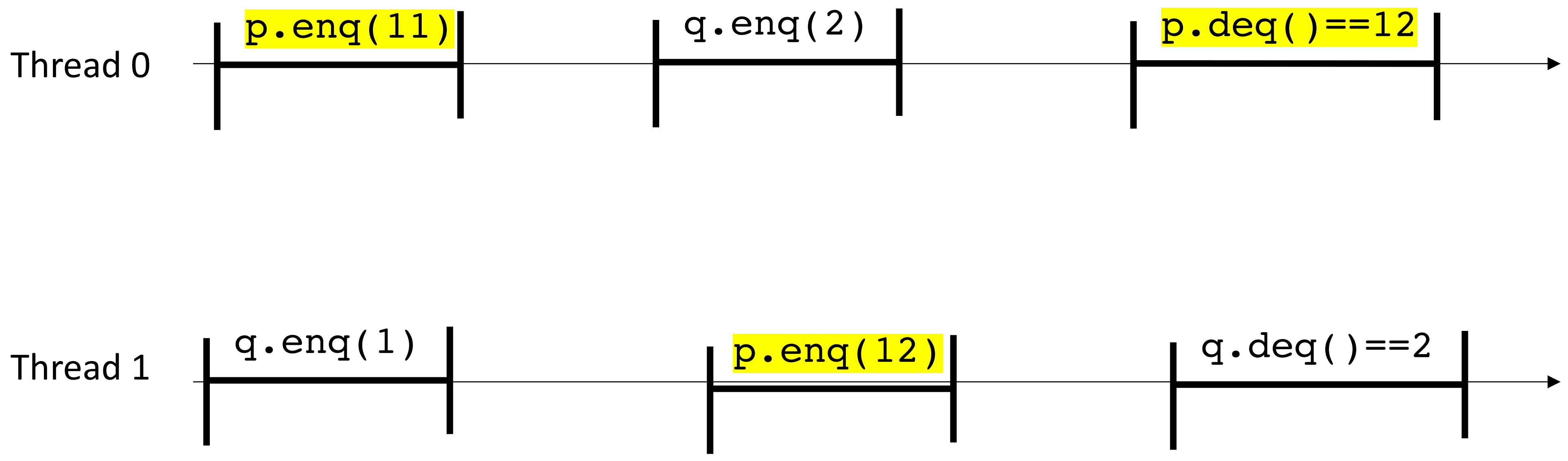
2 objects now: p and q



Sequential consistency and real time

- Add in real time:

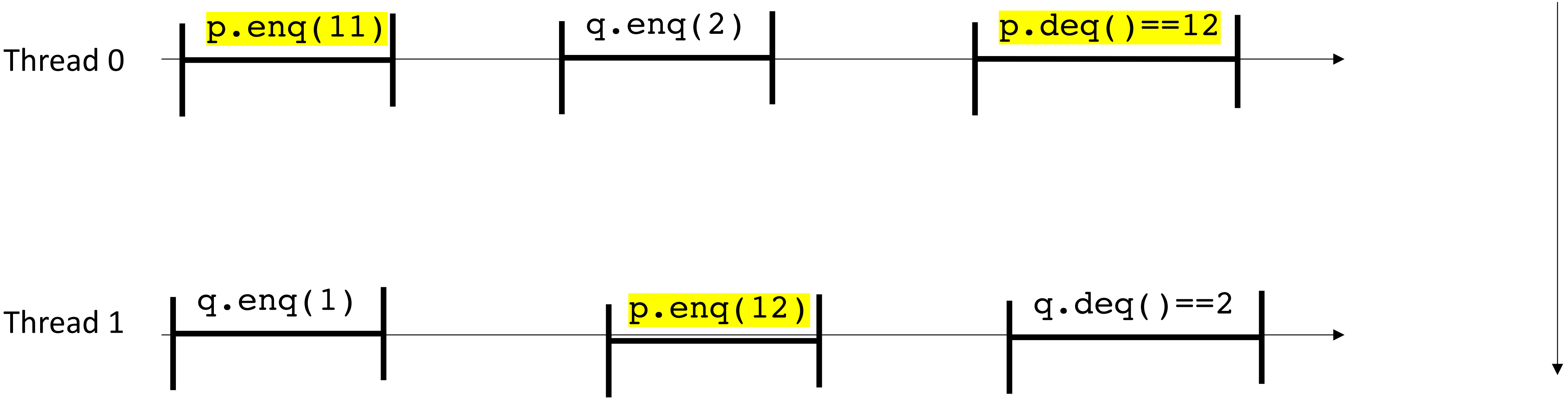
2 objects now: p and q
Consider each object in isolation



Sequential consistency and real time

- Add in real time:

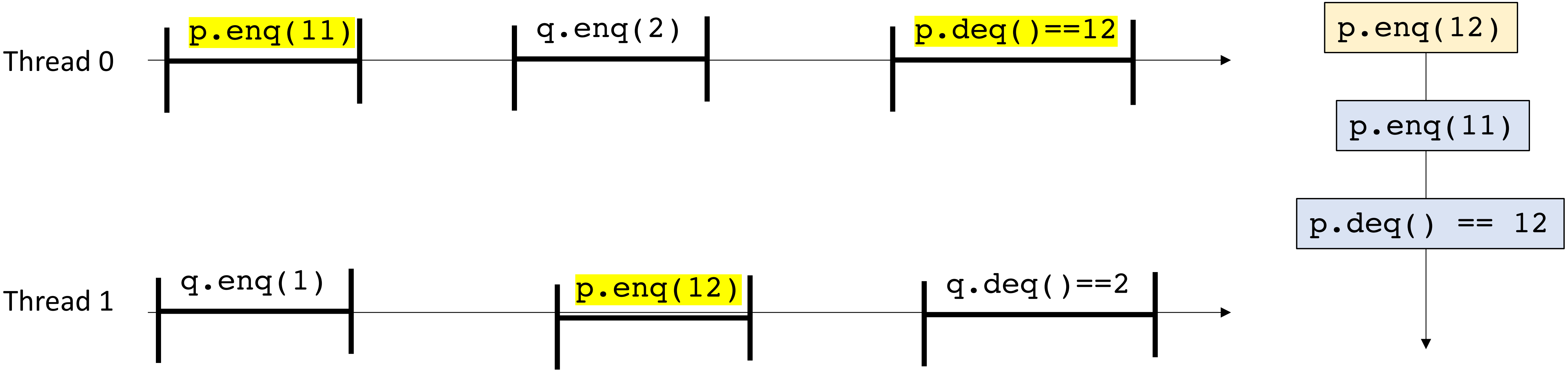
2 objects now: p and q
Consider each object in isolation



Sequential consistency and real time

- Add in real time:

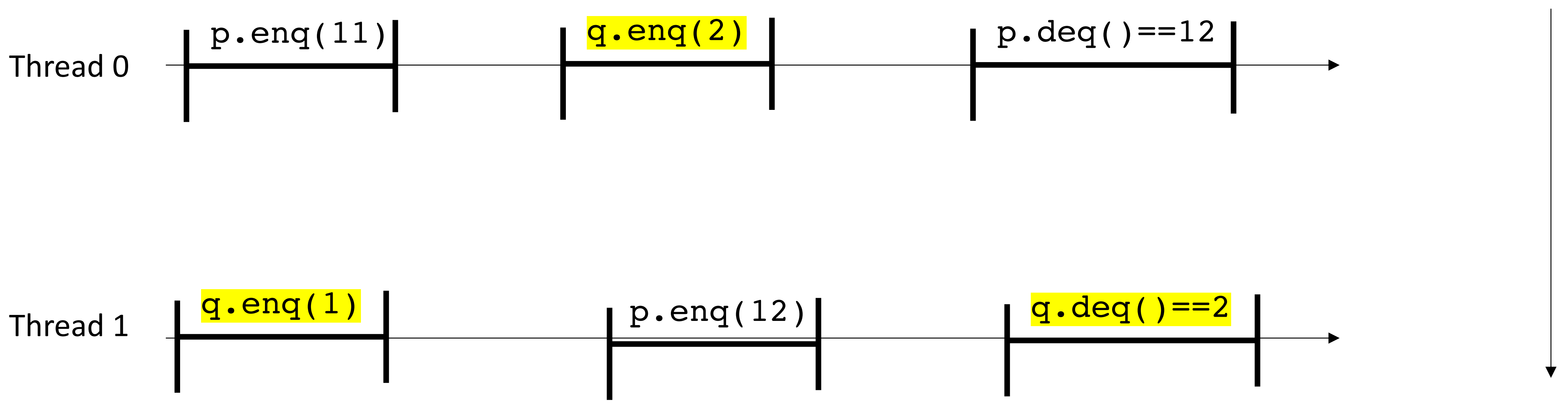
2 objects now: p and q
Consider each object in isolation



Sequential consistency and real time

- Add in real time:

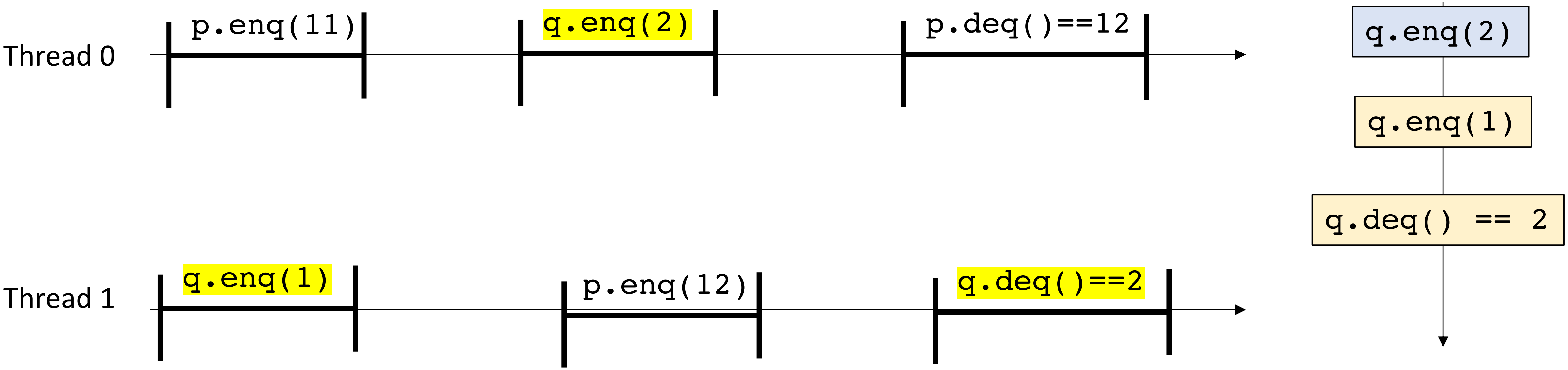
2 objects now: p and q
Consider each object in isolation



Sequential consistency and real time

- Add in real time:

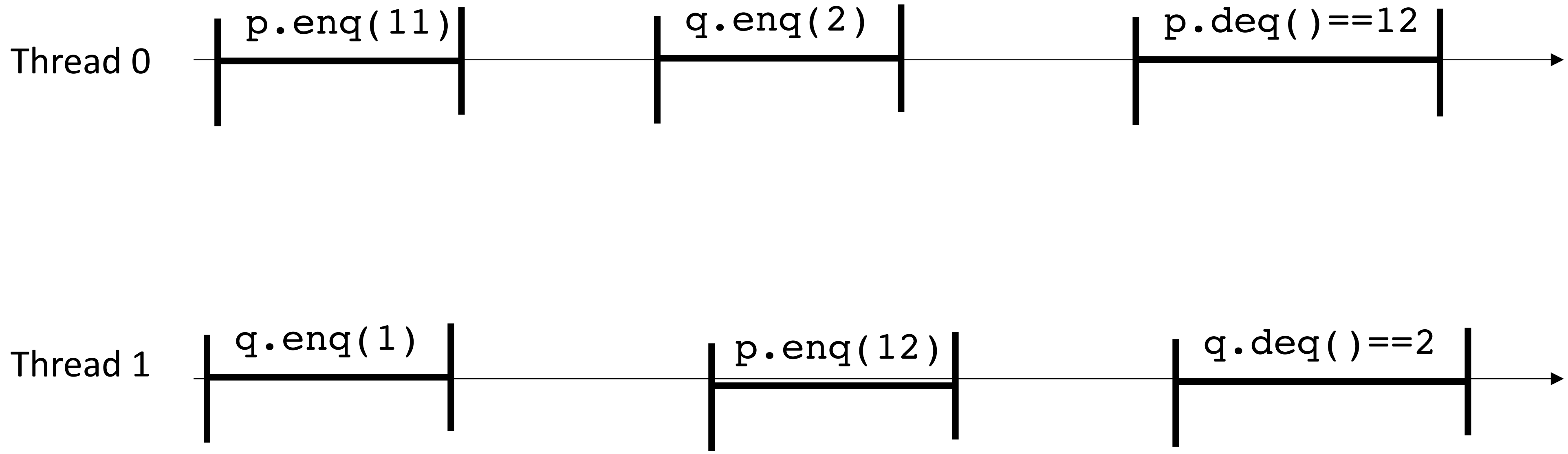
2 objects now: p and q
Consider each object in isolation



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

```
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!

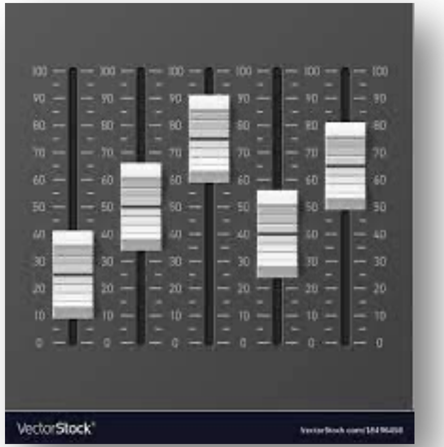
Schedule

- 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

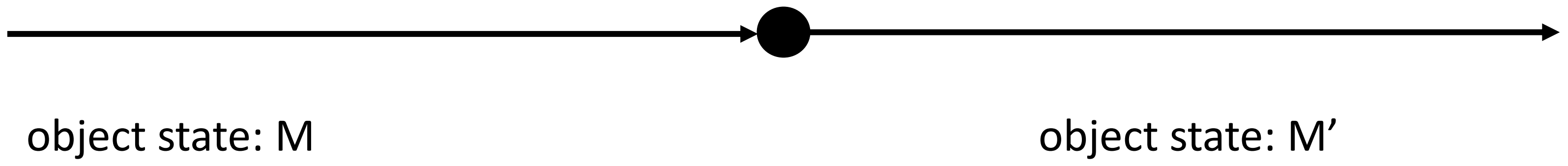
each operation has a linearizability point

- 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

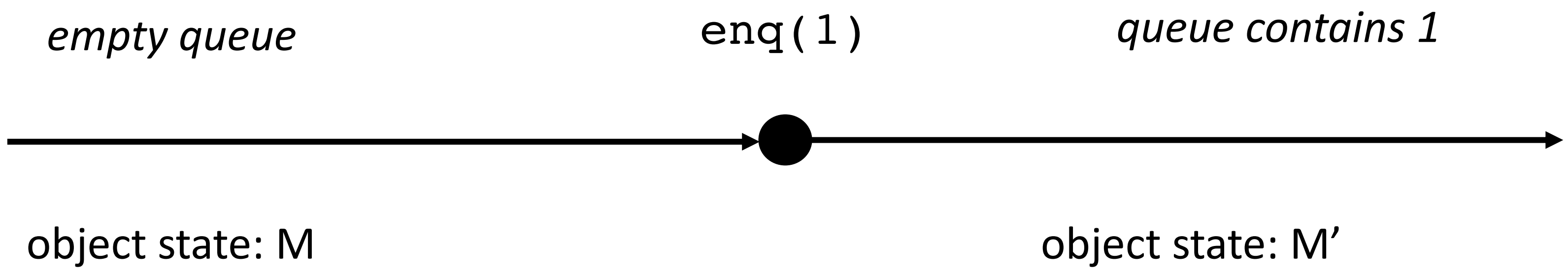
- does not overlap with other with other linearizability points
- indivisible computation (critical section, atomic RMW, atomic load, atomic store)
- object update (or read) occurs exactly at this point



Linearizability

each operation has a linearizability point

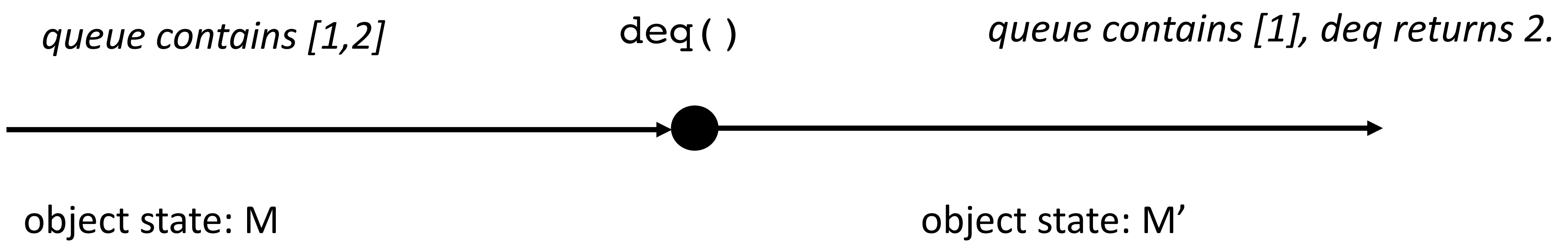
- does not overlap with other with other linearizability points
- indivisible computation (critical section, atomic RMW, atomic load, atomic store)
- object update (or read) occurs exactly at this point



Linearizability

each operation has a linearizability point

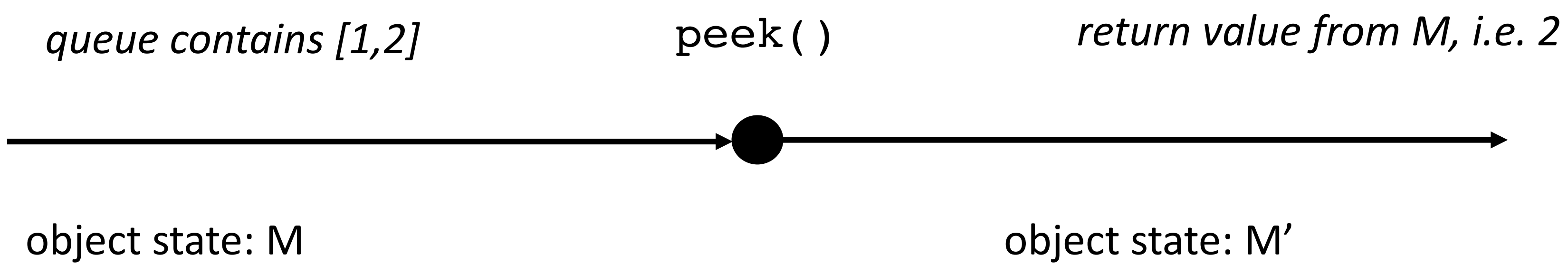
- does not overlap with other with other linearizability points
- indivisible computation (critical section, atomic RMW, atomic load, atomic store)
- object update (or read) occurs exactly at this point



Linearizability

each operation has a linearizability point

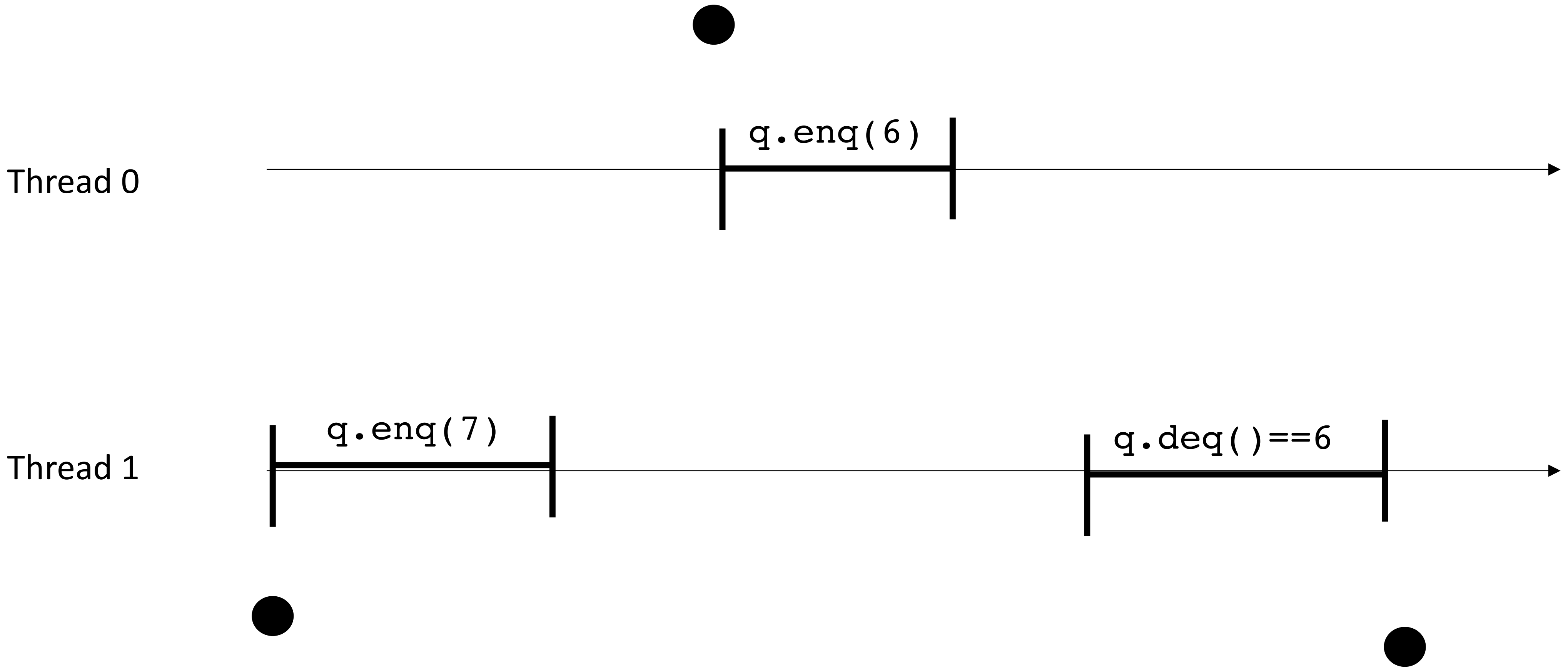
- does not overlap with other with other linearizability points
- indivisible computation (critical section, atomic RMW, atomic load, atomic store)
- object update (or read) occurs exactly at this point



Linearizability

each command gets a linearization point.

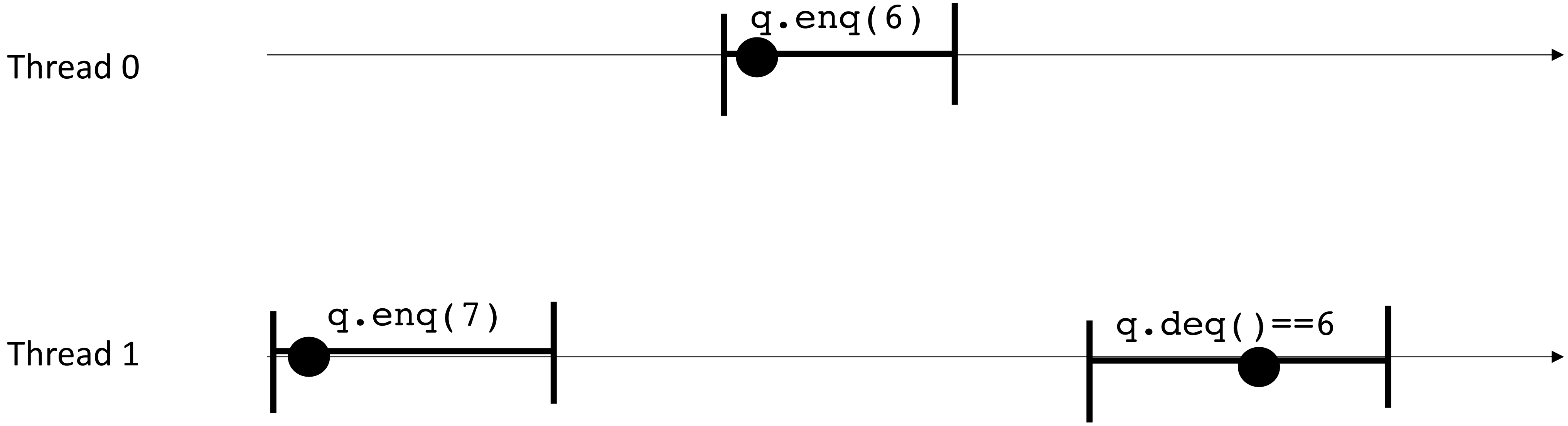
You can place the point any where between its innovation and response!



Linearizability

each command gets a linearization point.

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

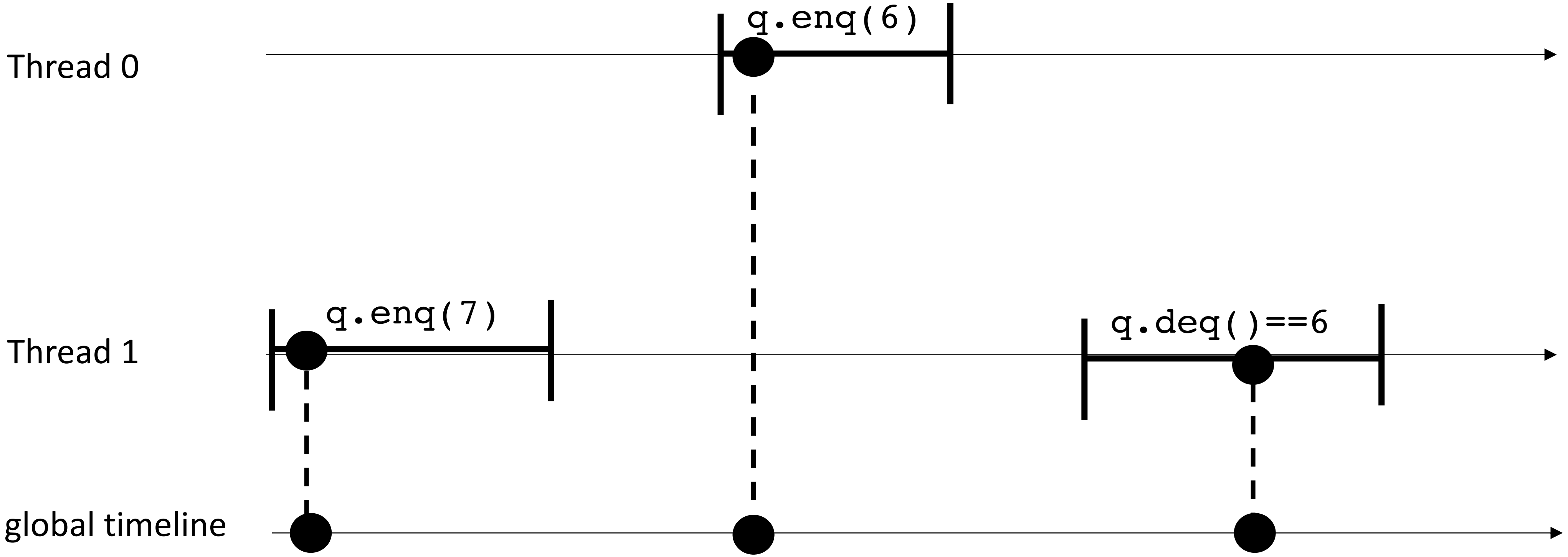


Linearizability

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



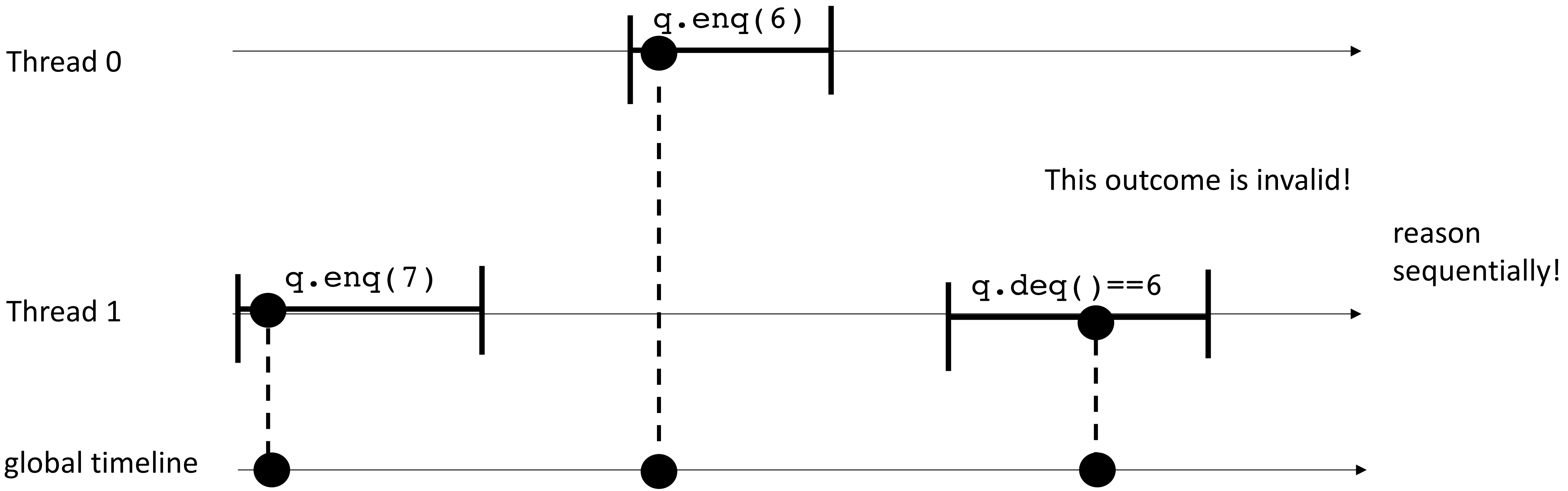
reason sequentially!

Linearizability

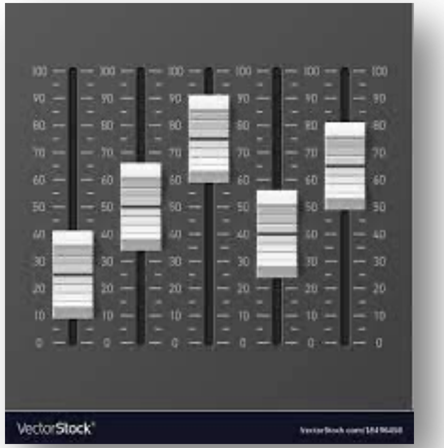
each command gets a linearization point.

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!

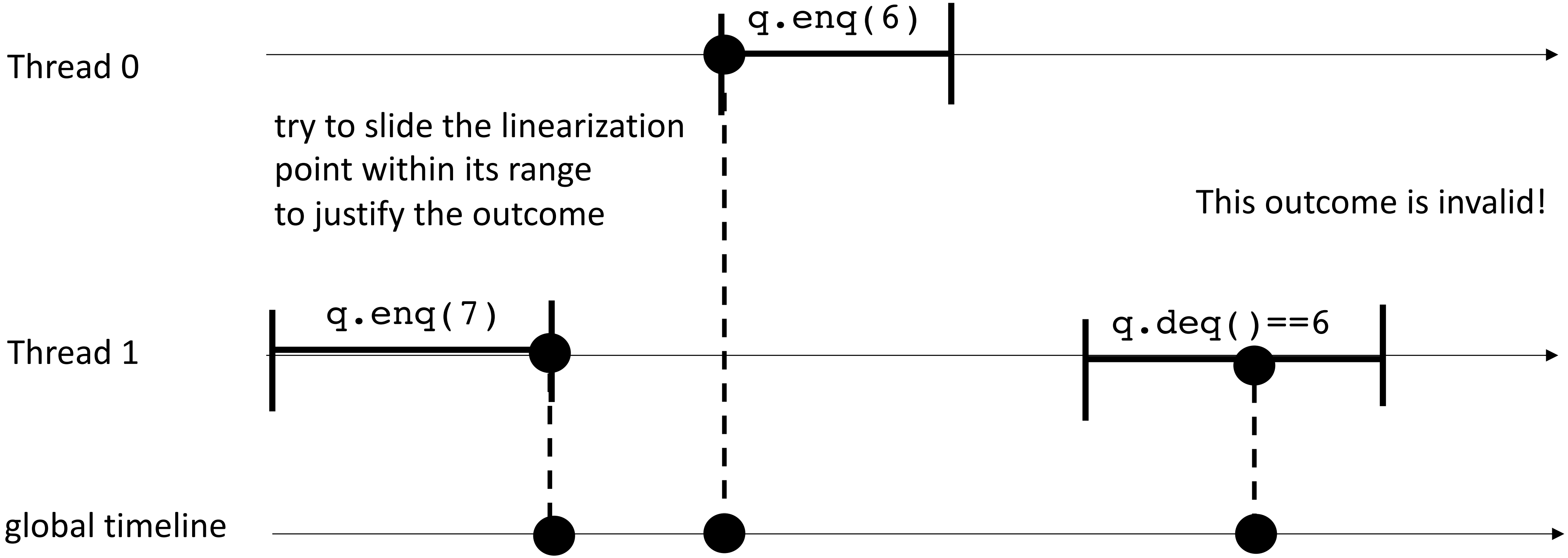
Project the linearization points to a global timeline

slider game!

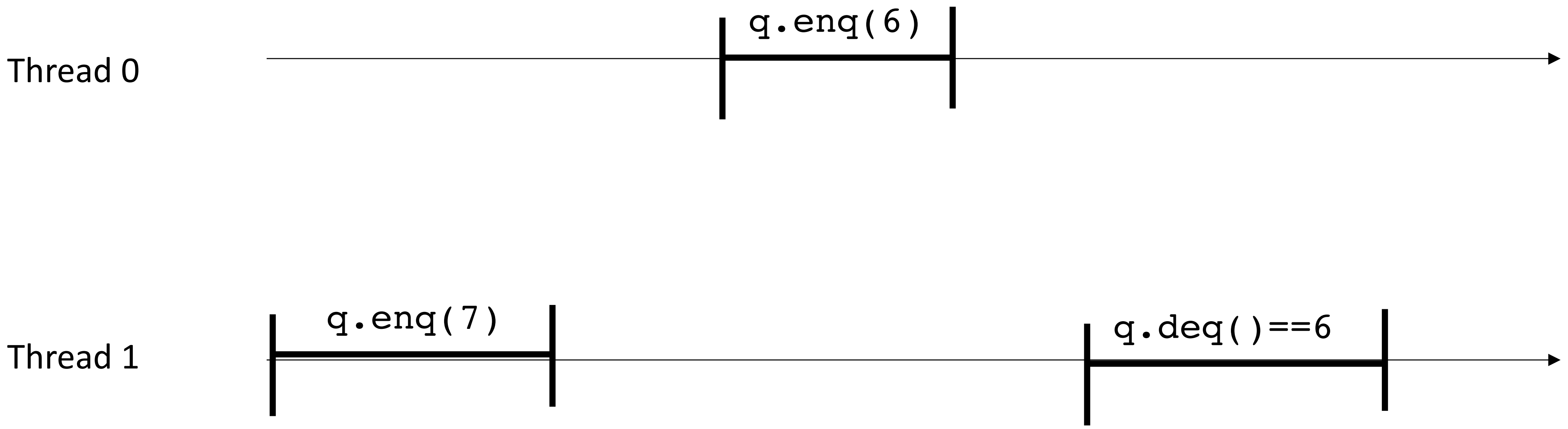
try to slide the linearization
point within its range
to justify the outcome

This outcome is invalid!

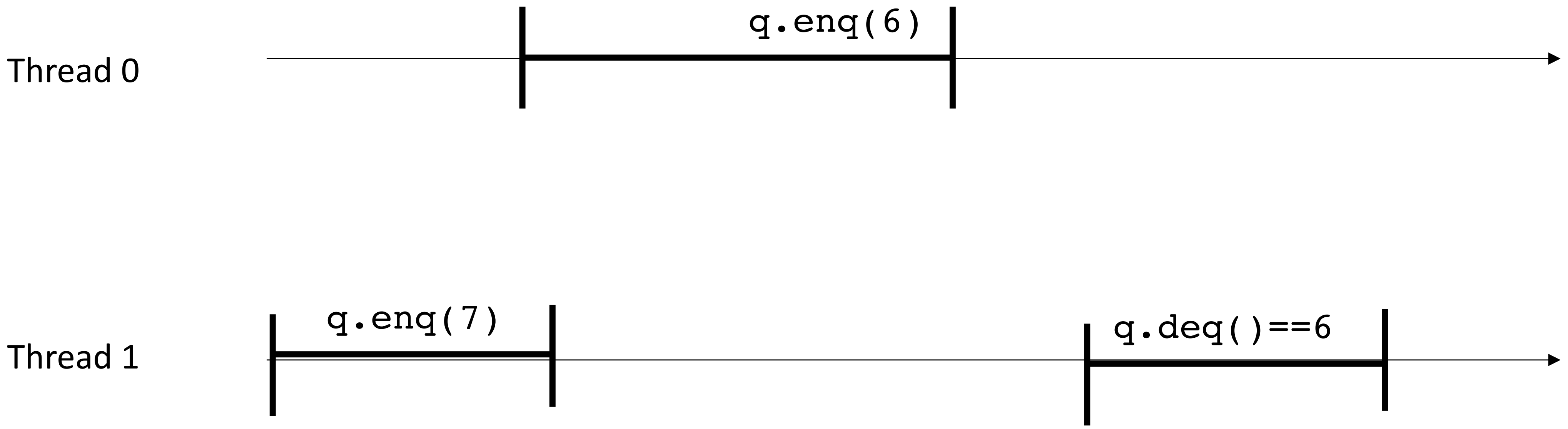
reason sequentially!



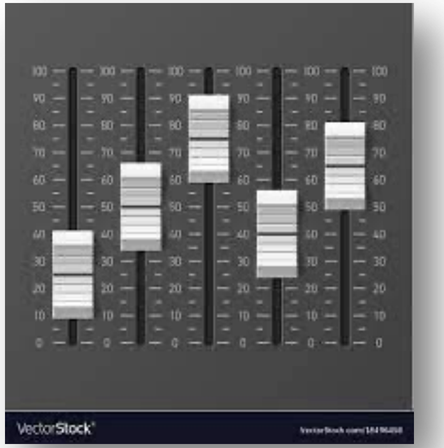
Linearizability

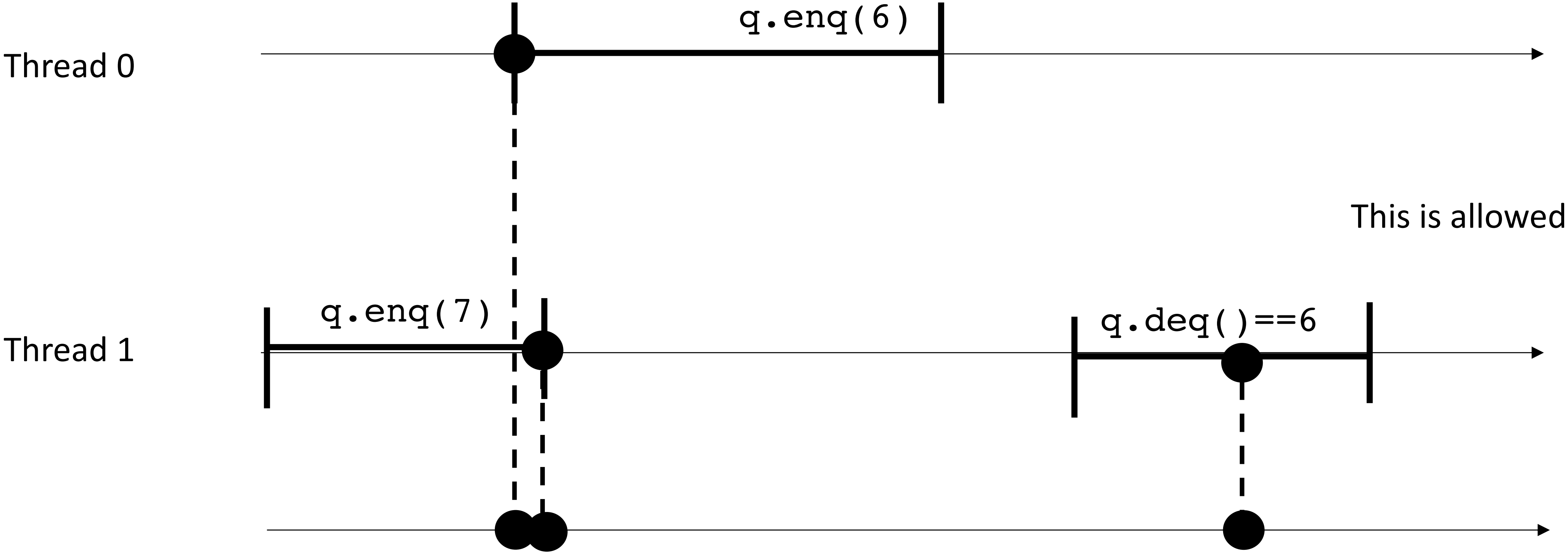


Linearizability



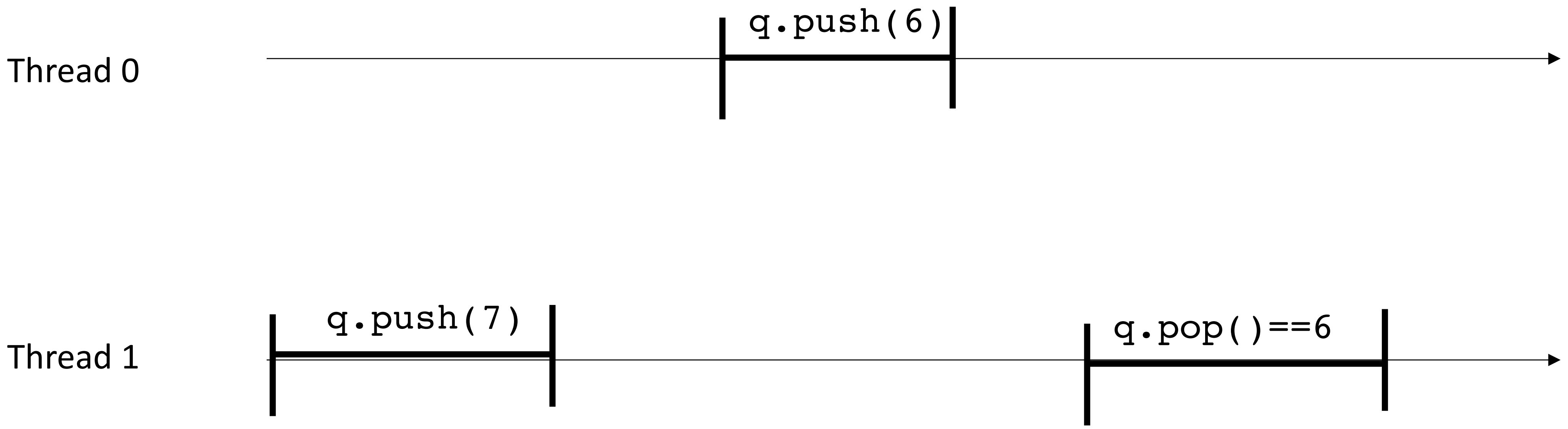
Linearizability



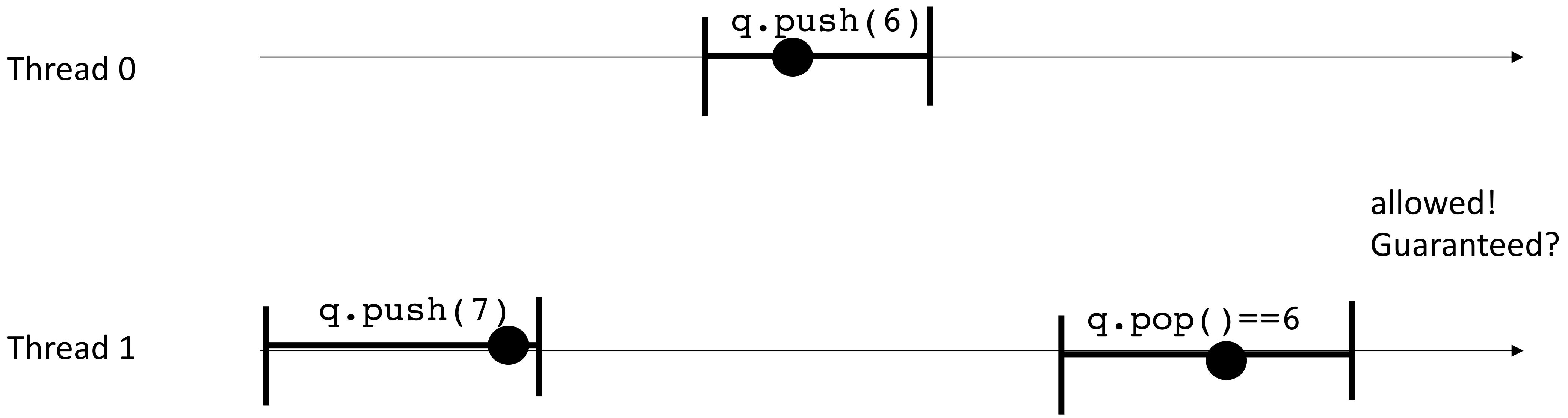


This is allowed now!

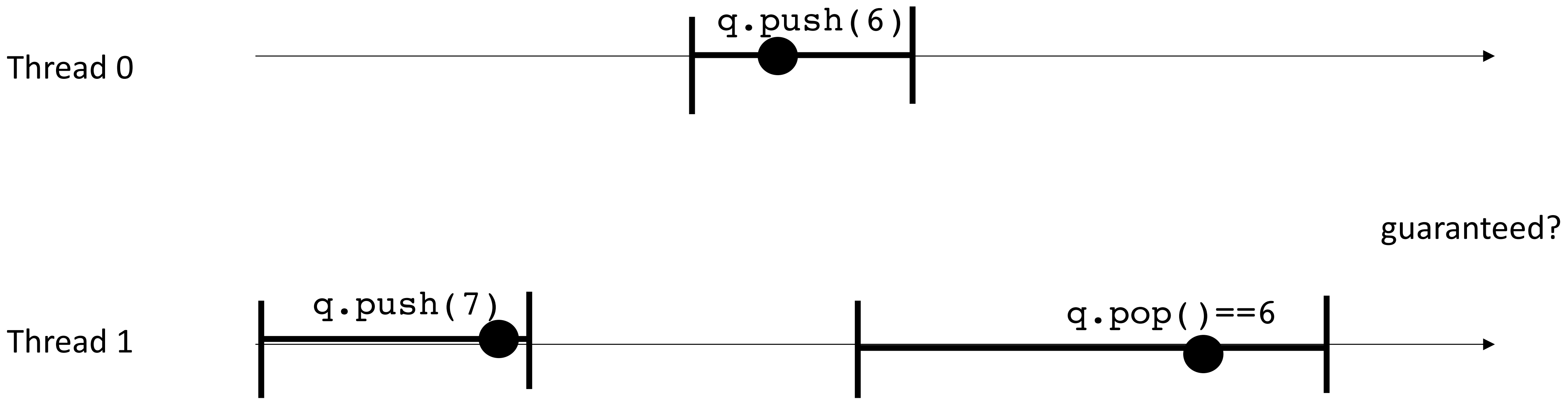
Linearizability



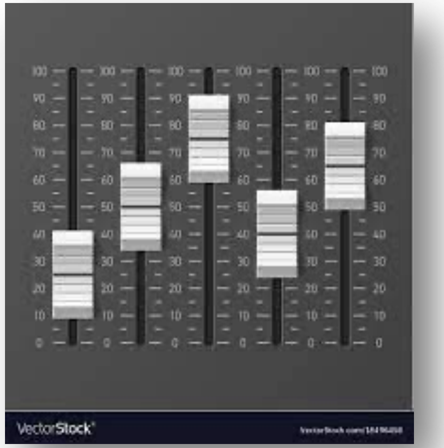
Linearizability

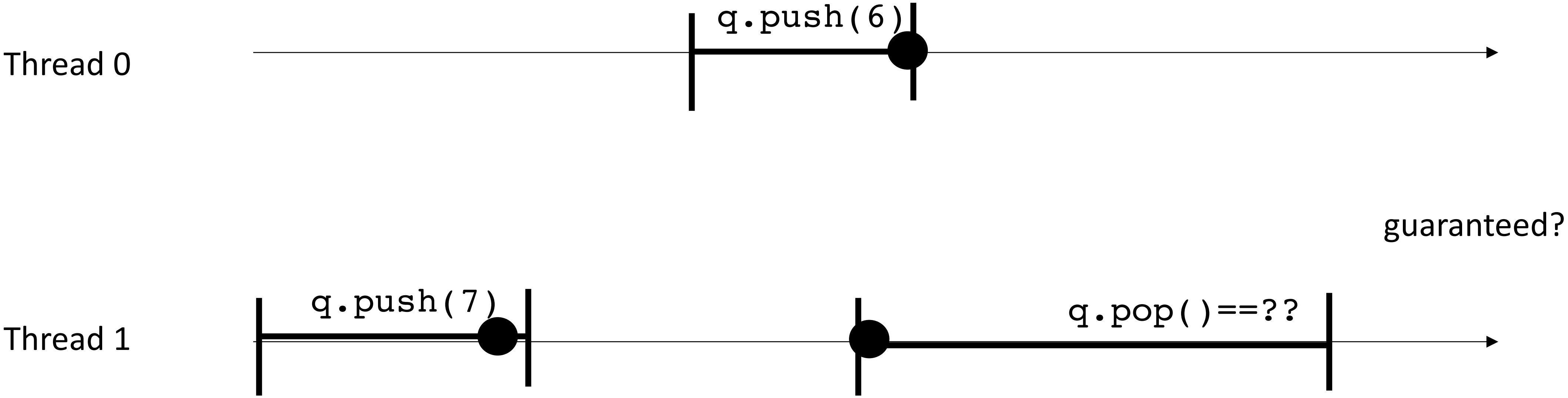


Linearizability



Linearizability





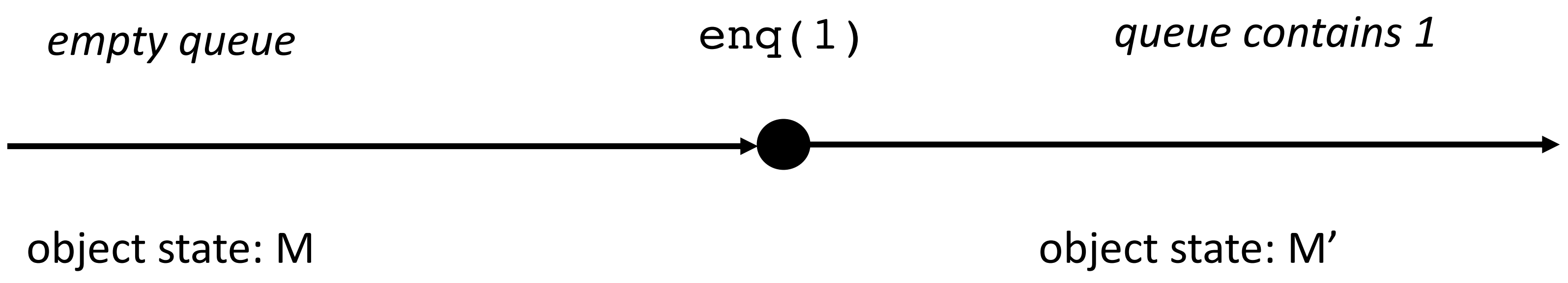
Linearizability

- 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.

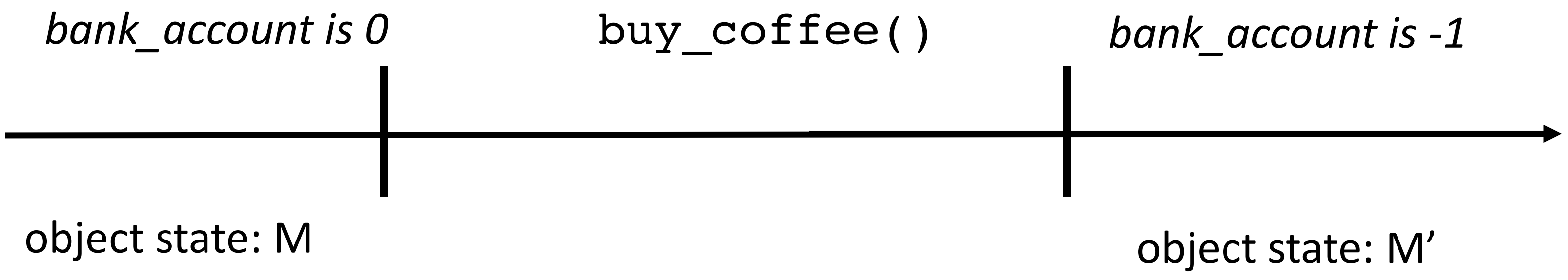
Linearizability

- 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.

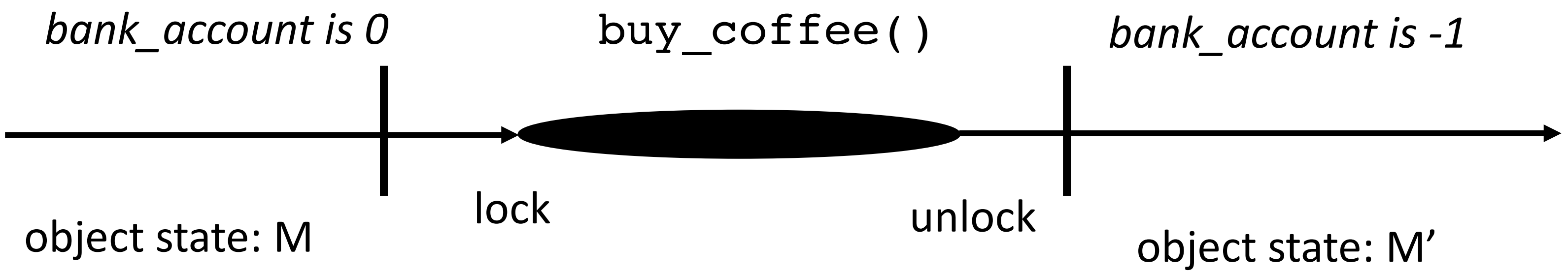



```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    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

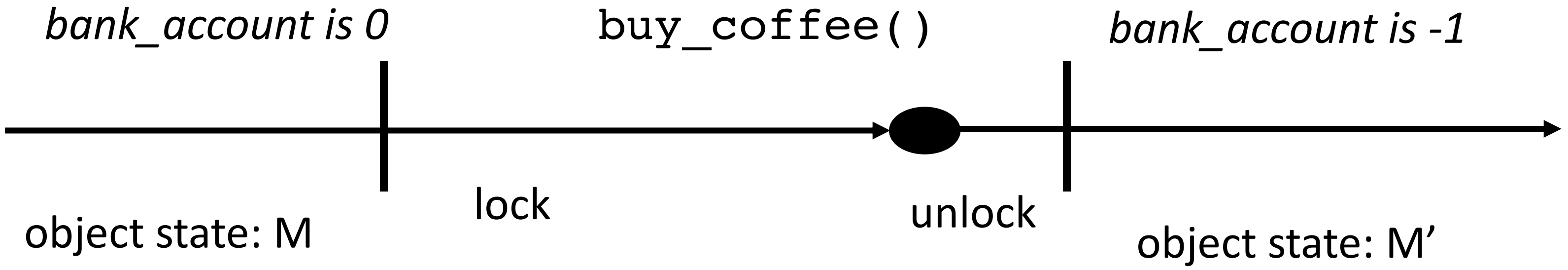



```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    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.

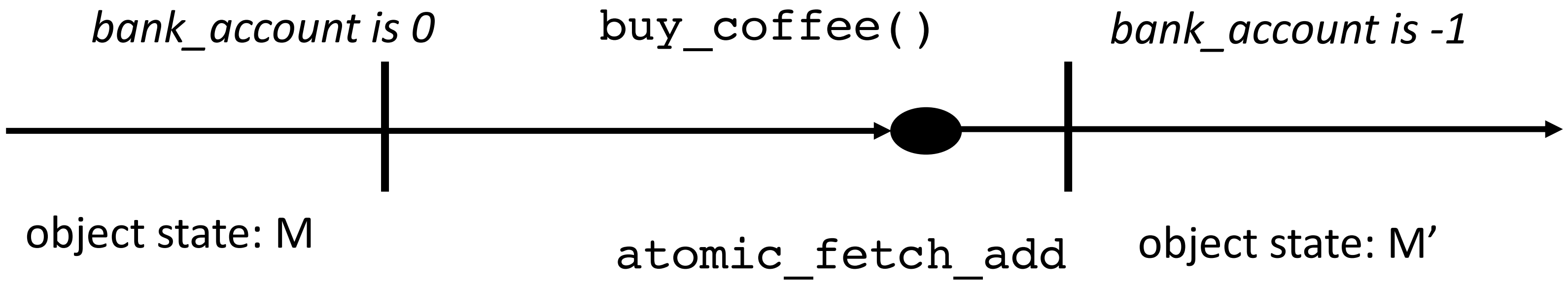



```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

    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;
};
```



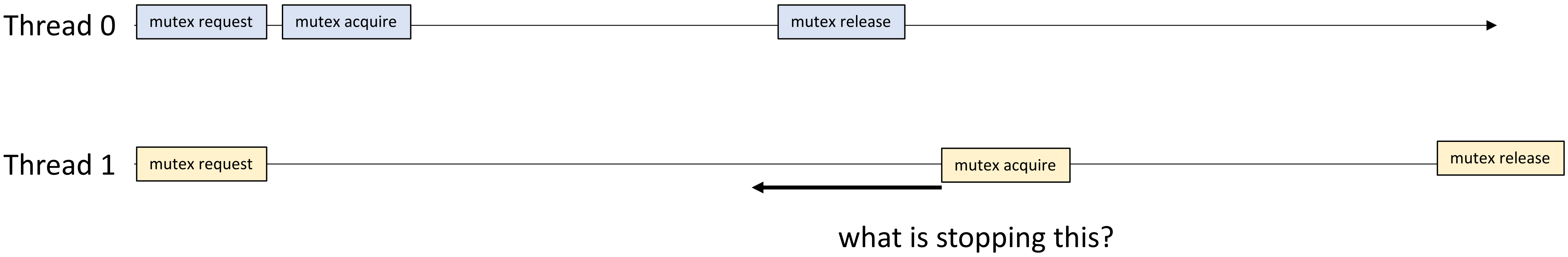
Lecture schedule

- Linearizablity
- **Progress Properties**
- Implementing a set

Progress properties

- Going back to specifications:

Recall the mutex

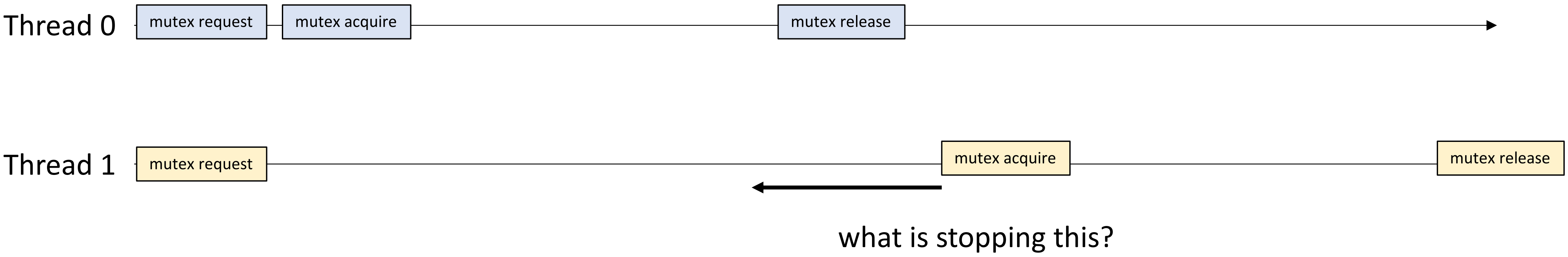


Progress properties

- 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

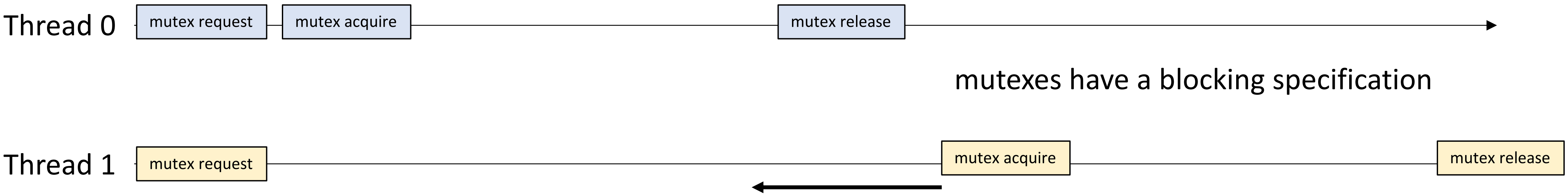


Progress properties

- 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



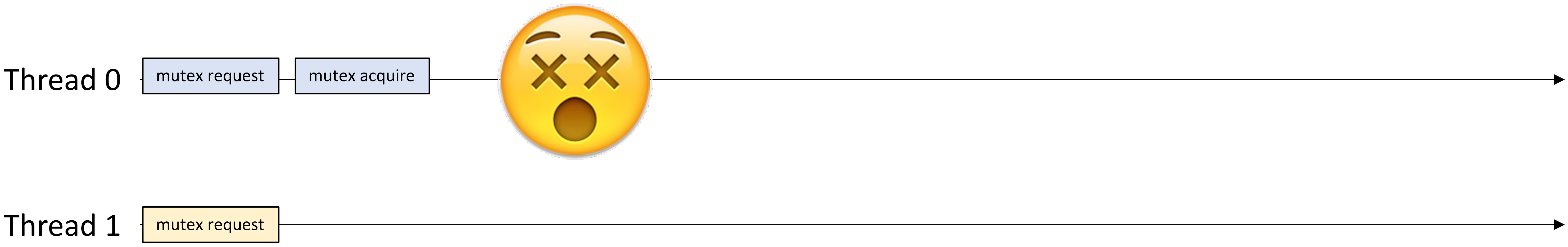
what is stopping this?

Progress properties

- Going back to specifications:

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

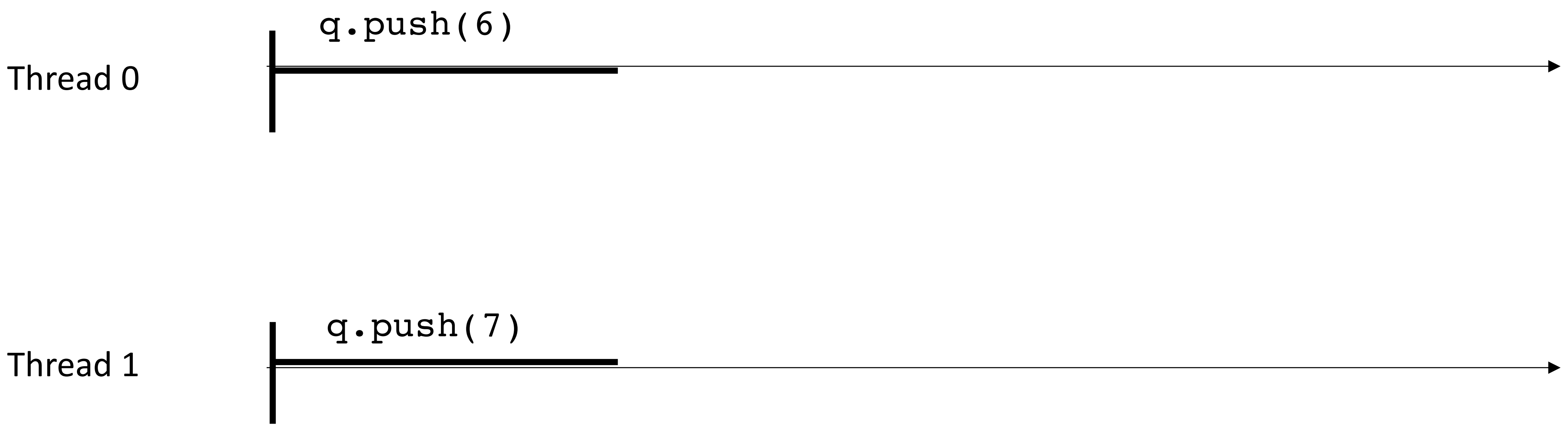
Recall the mutex



What now?!

Linearizability

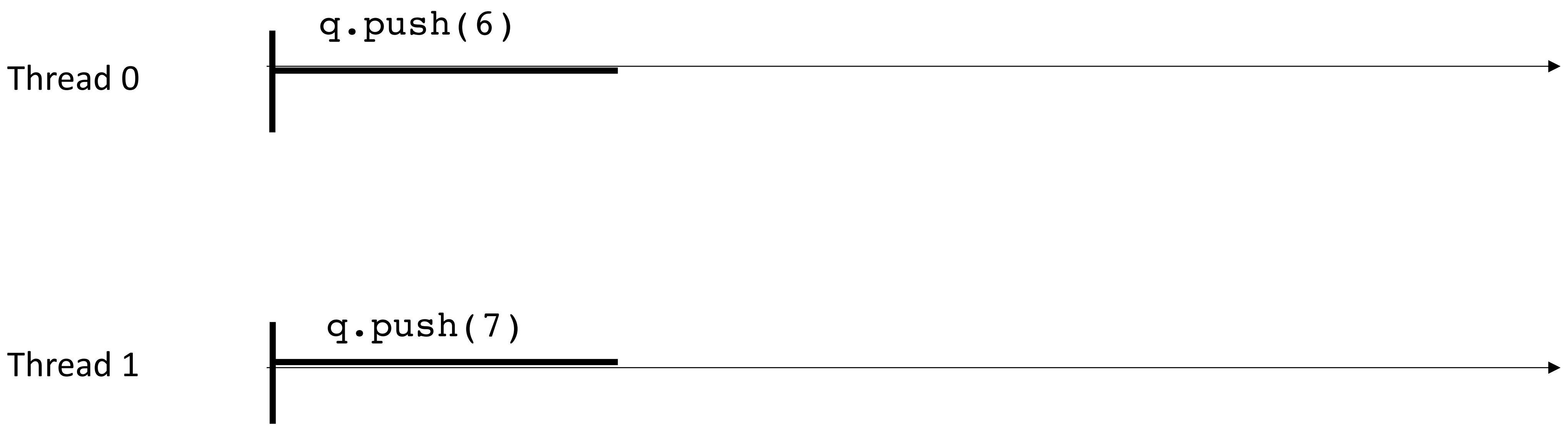
Two unfinished commands.



Linearizability

Two unfinished commands.

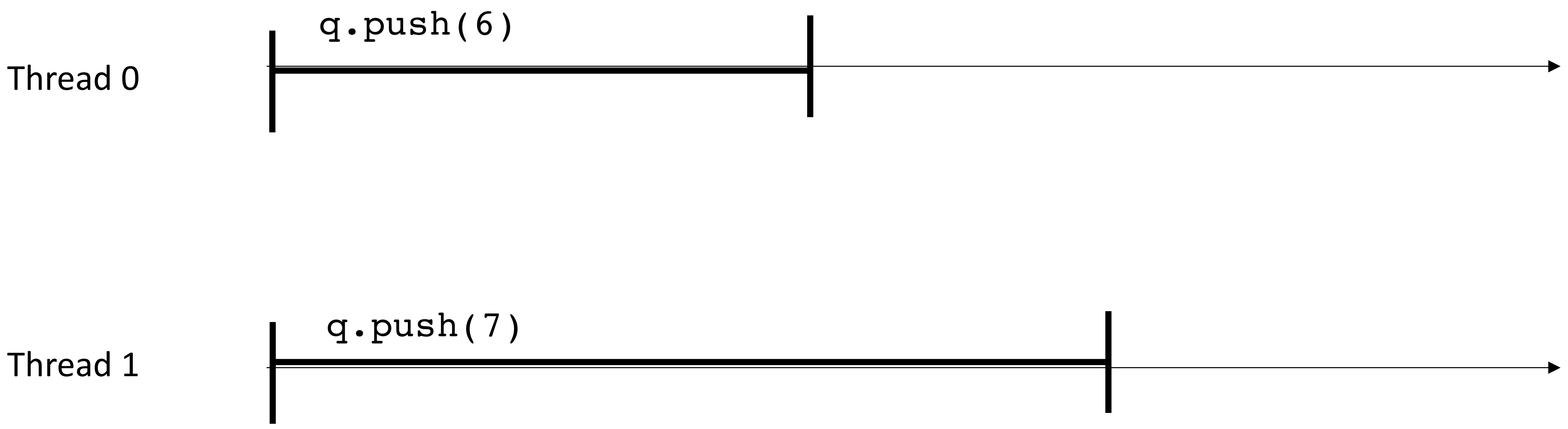
Linearizability does not dictate that one needs to wait for another



Linearizability

Two unfinished commands.

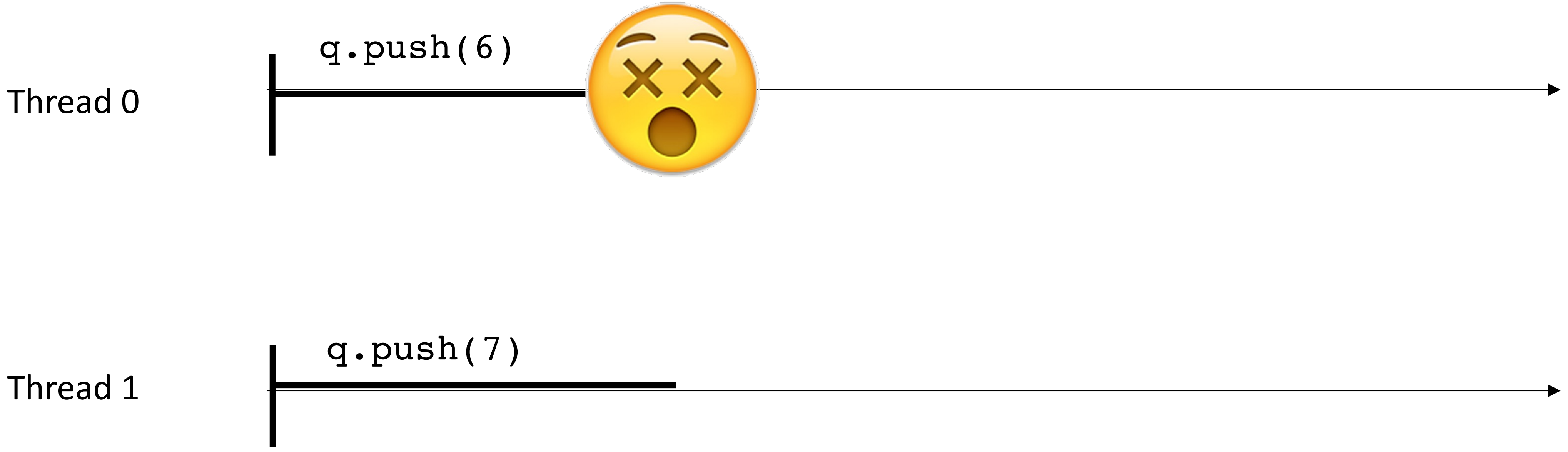
Linearizability does not dictate that one needs to wait for another



Linearizability

Two unfinished commands.

Linearizability does not dictate that one needs to wait for another

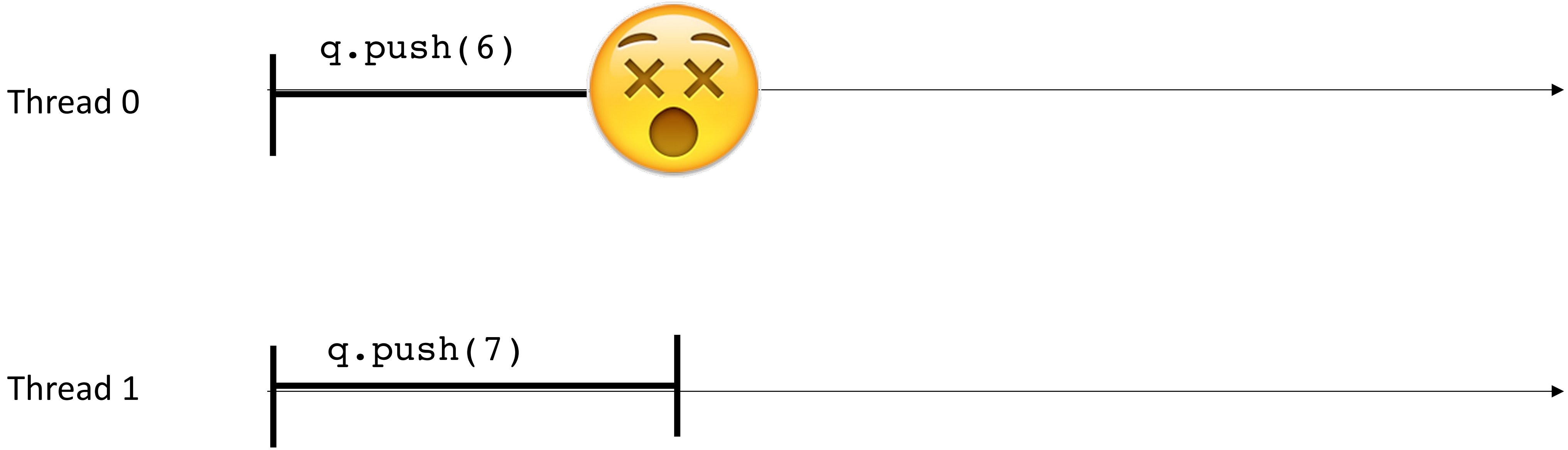


for mutexes, the specification required that the system hang.

Linearizability

Two unfinished commands.

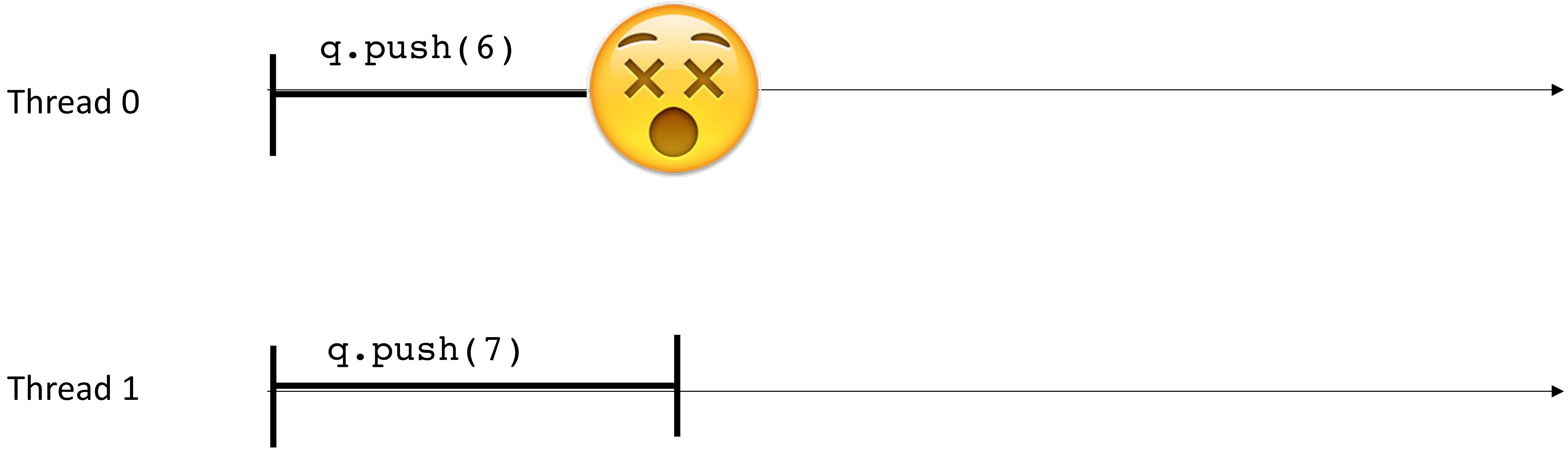
Linearizability does not dictate that one needs to wait for another



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

Linearizability

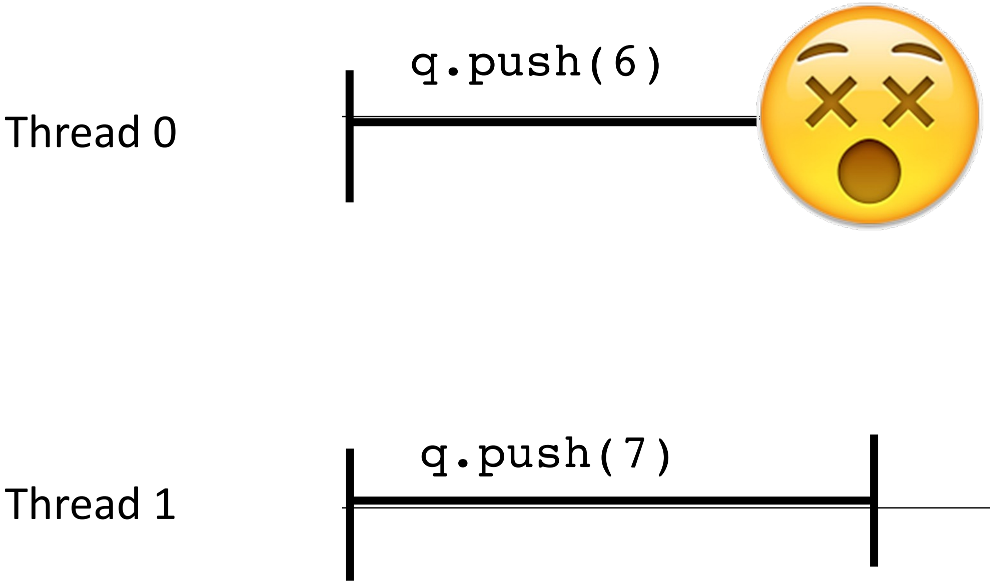
Non-blocking specification:
Every thread is allowed to continue executing
REGARDLESS of the behavior of other threads



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

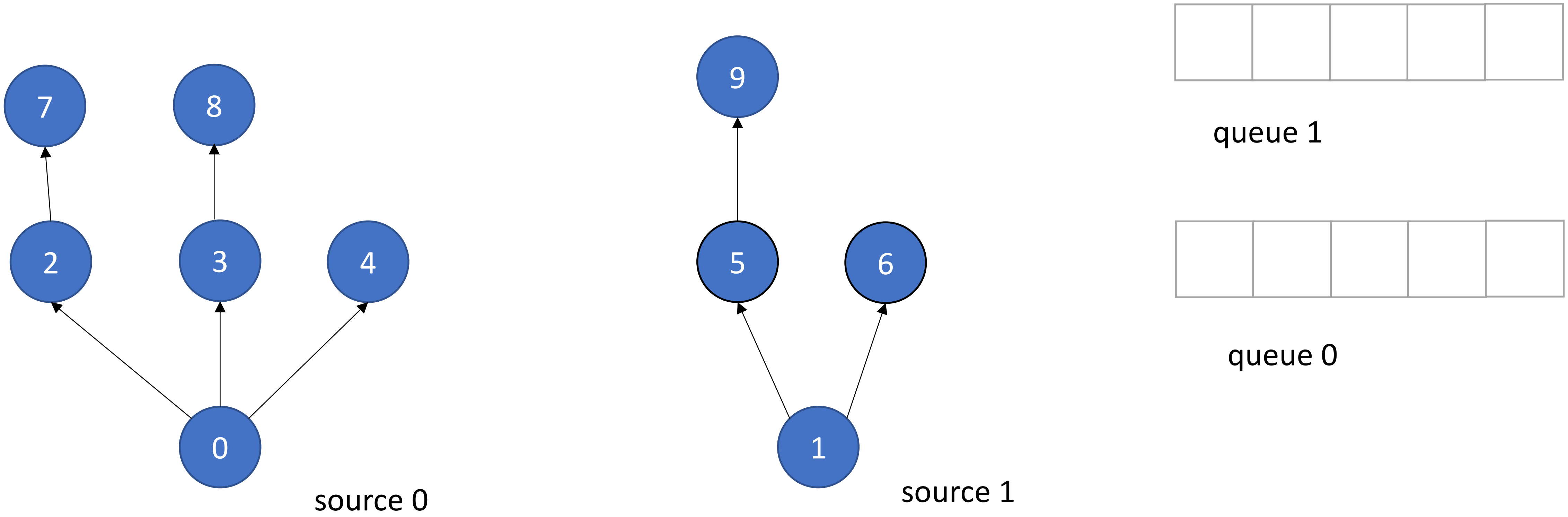
- Queue in which multiple threads read (deq), or write (enq), but not both.
- Why would we want a thing?
- Computation done in phases:
  - First phase prepares the queue (by writing into it)
  - All threads join
  - Second phase reads values from the queue.

Input/Output Queues

- Example: Information flow in graph applications:

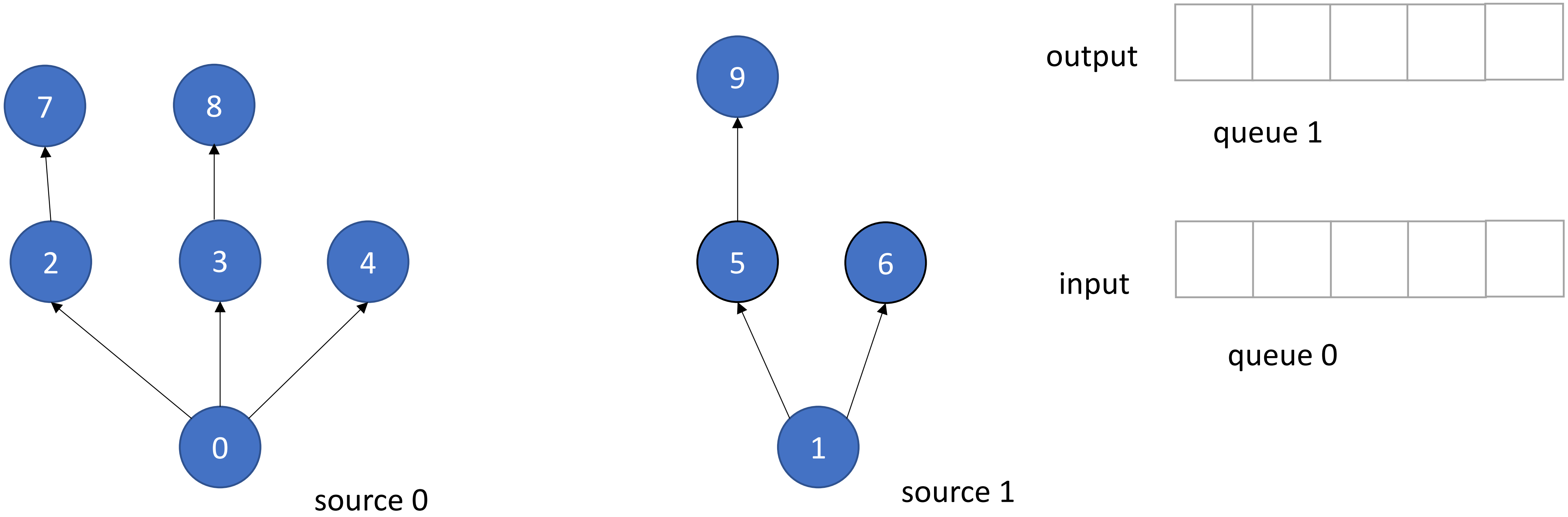
Input/Output Queues

- Example: Information flow in graph applications:



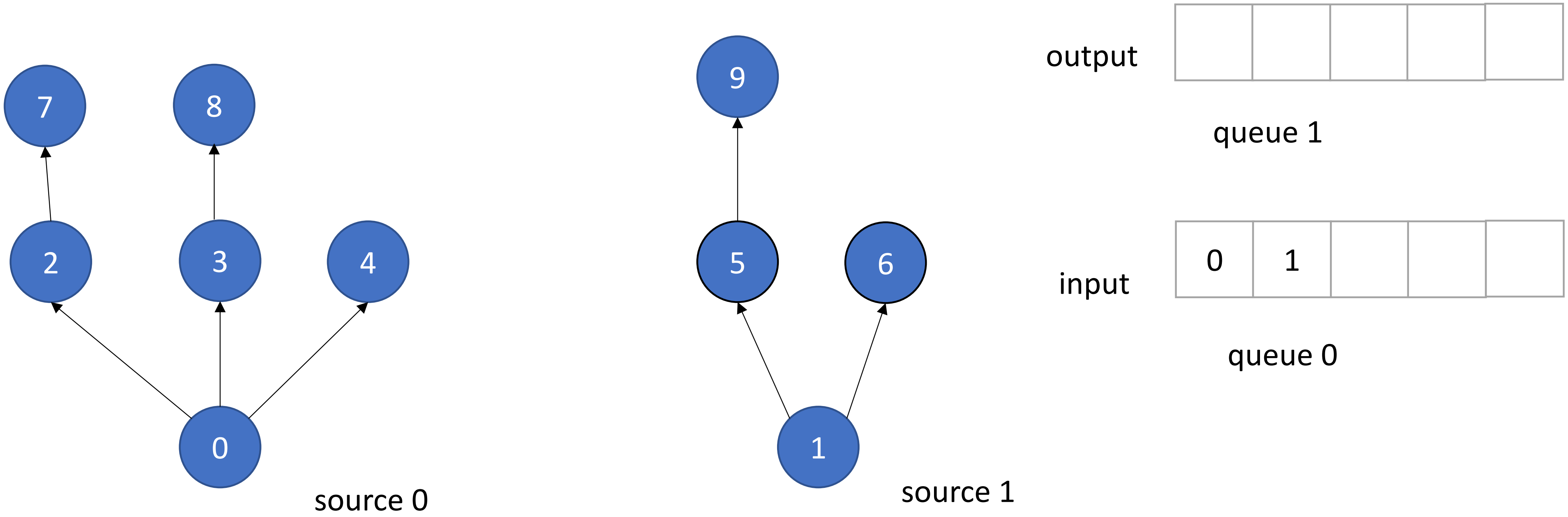
Input/Output Queues

- Example: Information flow in graph applications:



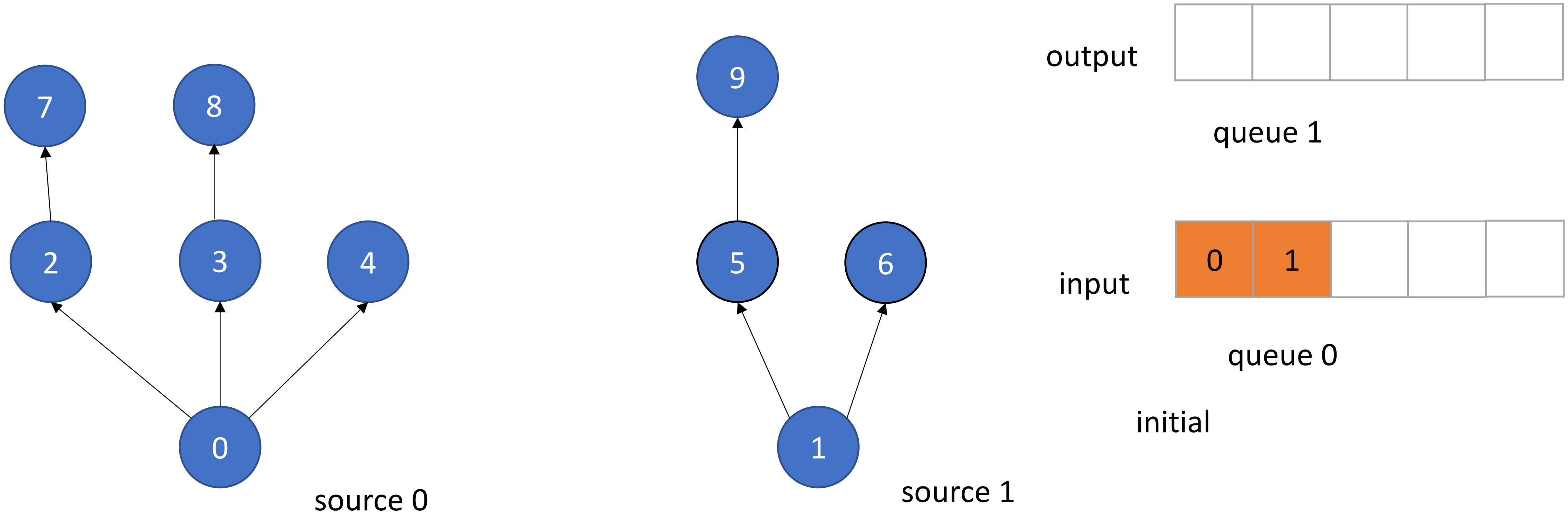
Input/Output Queues

- Example: Information flow in graph applications:



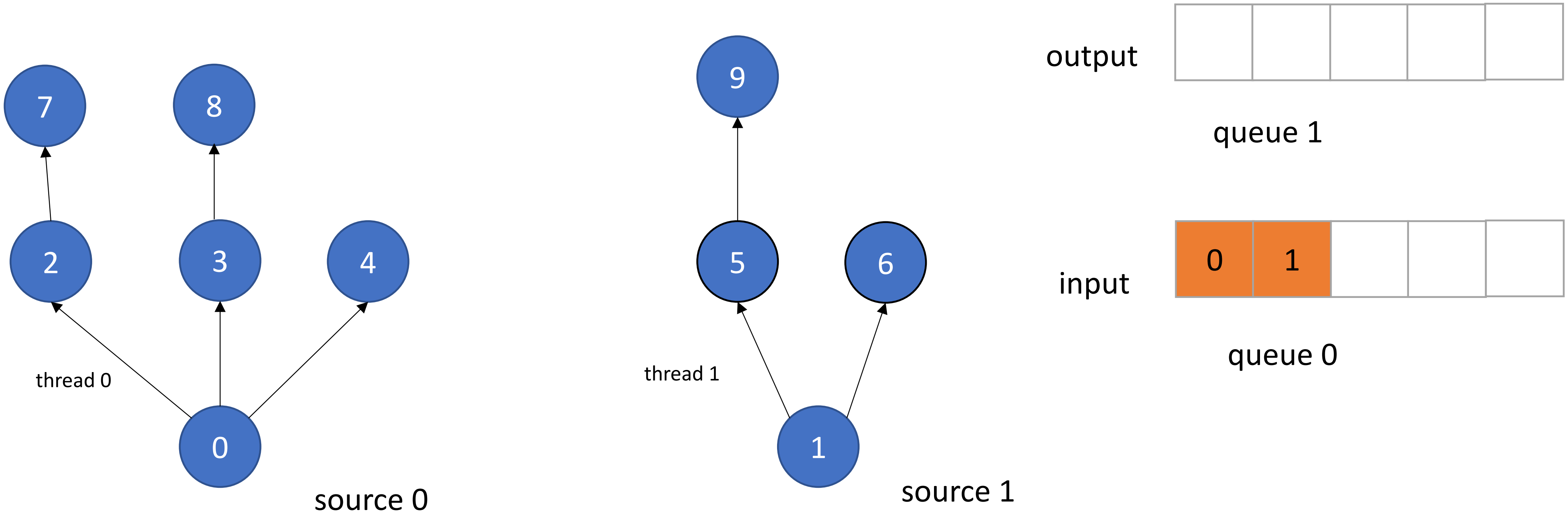
Input/Output Queues

- Example: Information flow in graph applications:



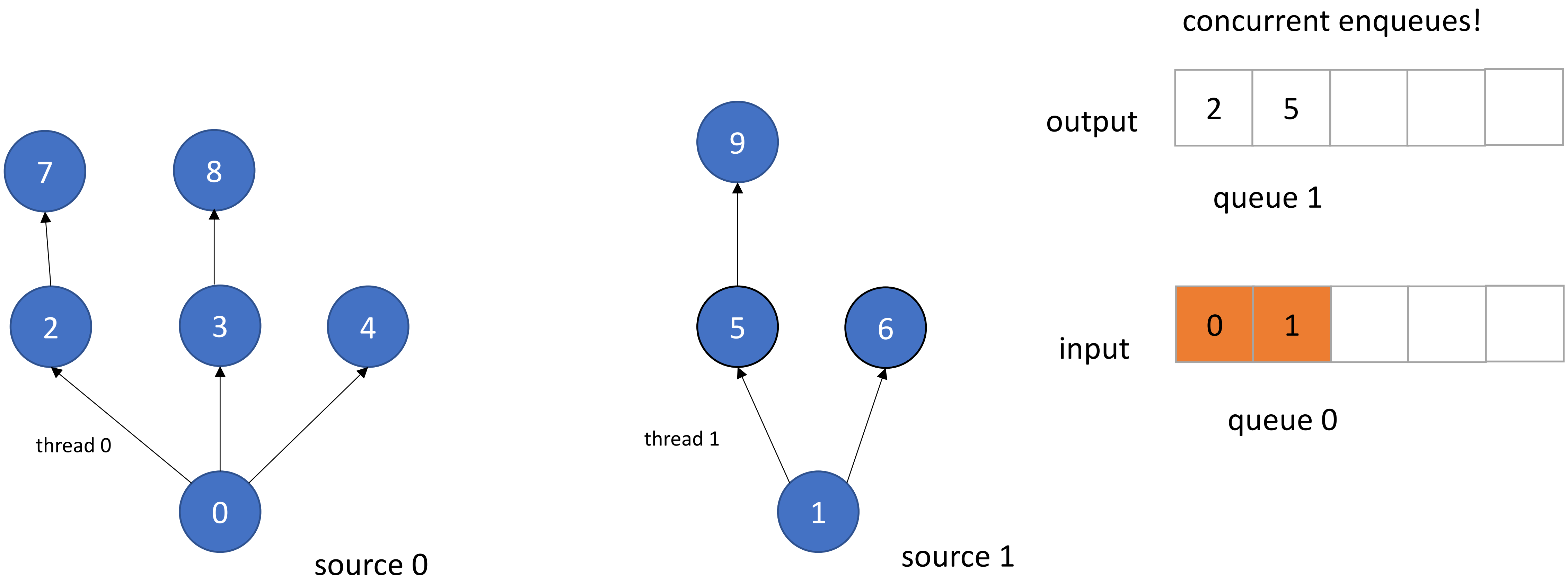
Input/Output Queues

- Example: Information flow in graph applications:



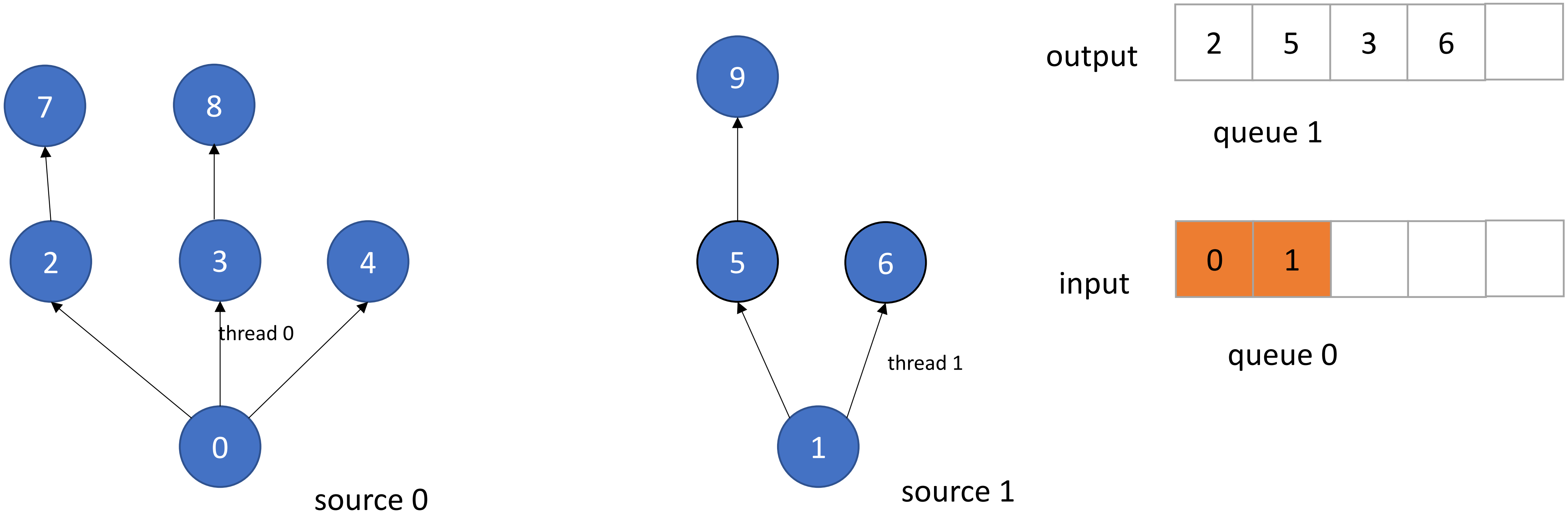
Input/Output Queues

- Example: Information flow in graph applications:



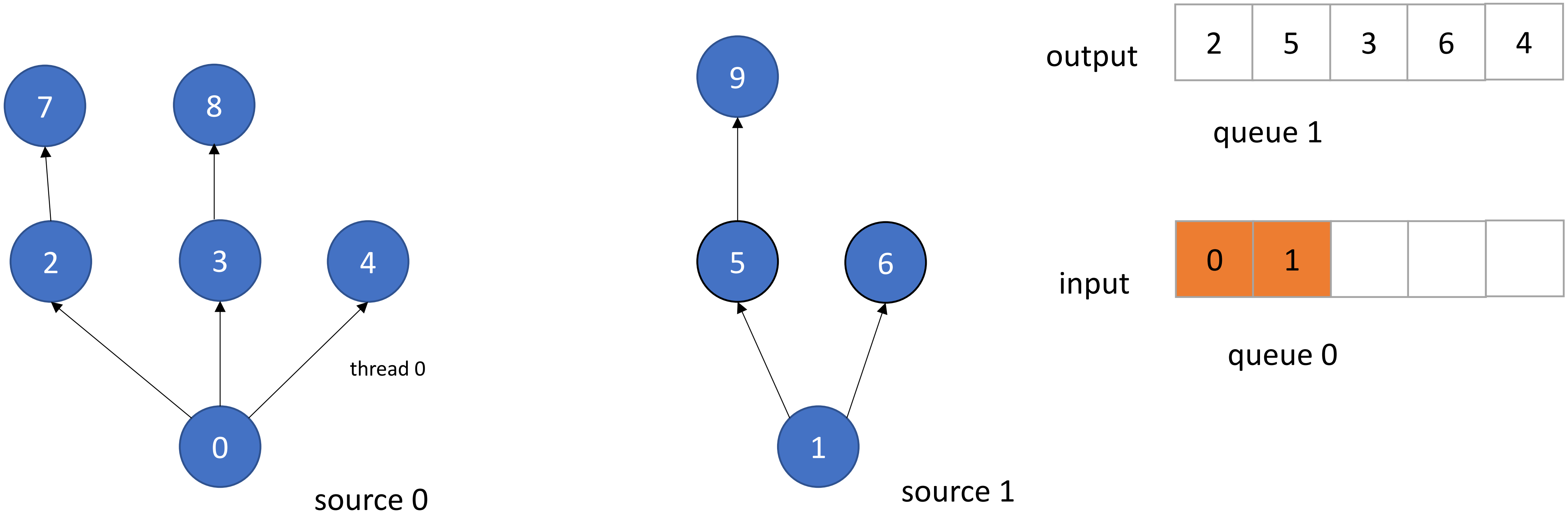
Input/Output Queues

- Example: Information flow in graph applications:



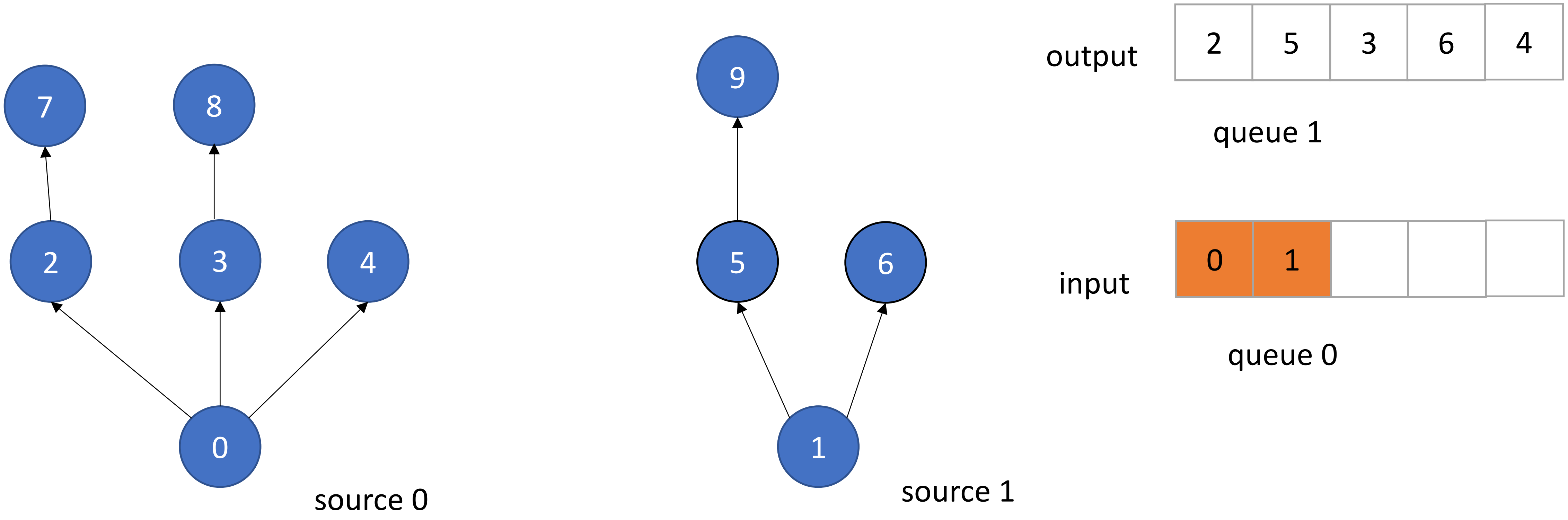
Input/Output Queues

- Example: Information flow in graph applications:



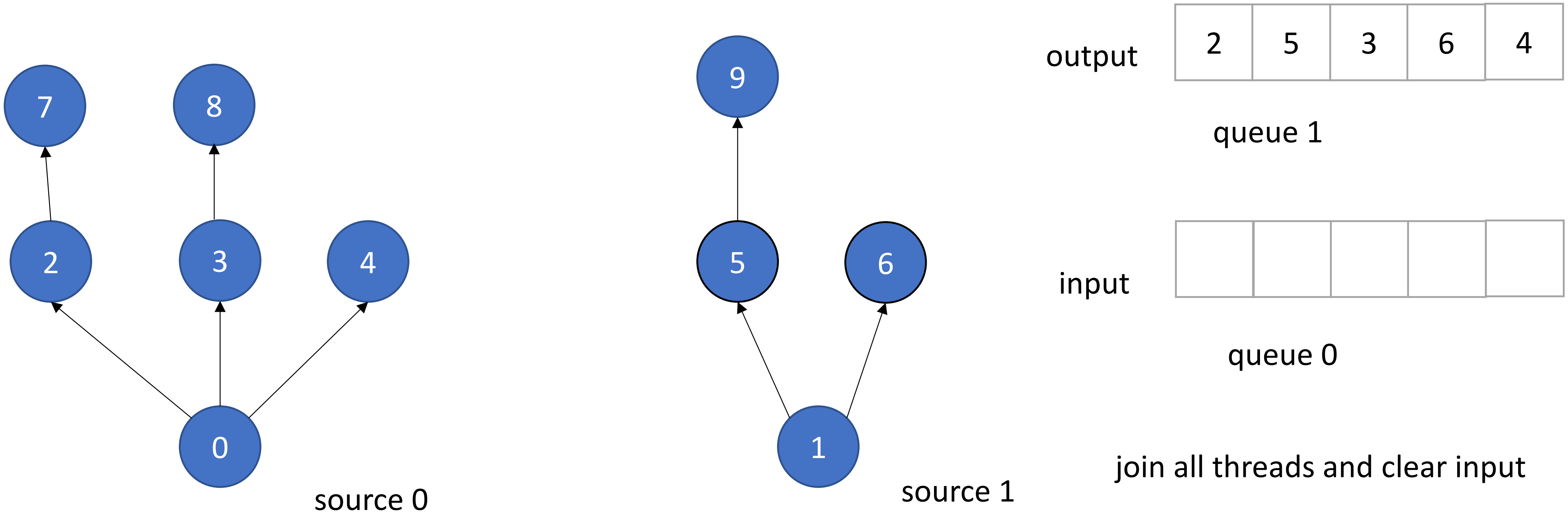
Input/Output Queues

- Example: Information flow in graph applications:



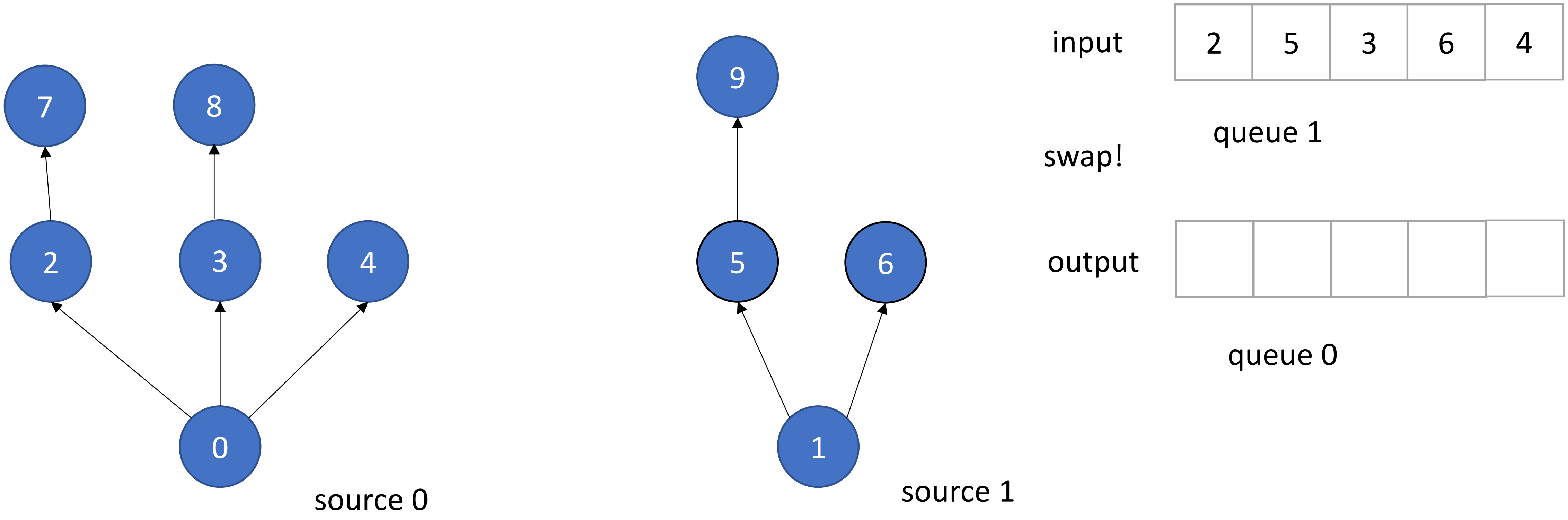
Input/Output Queues

- Example: Information flow in graph applications:



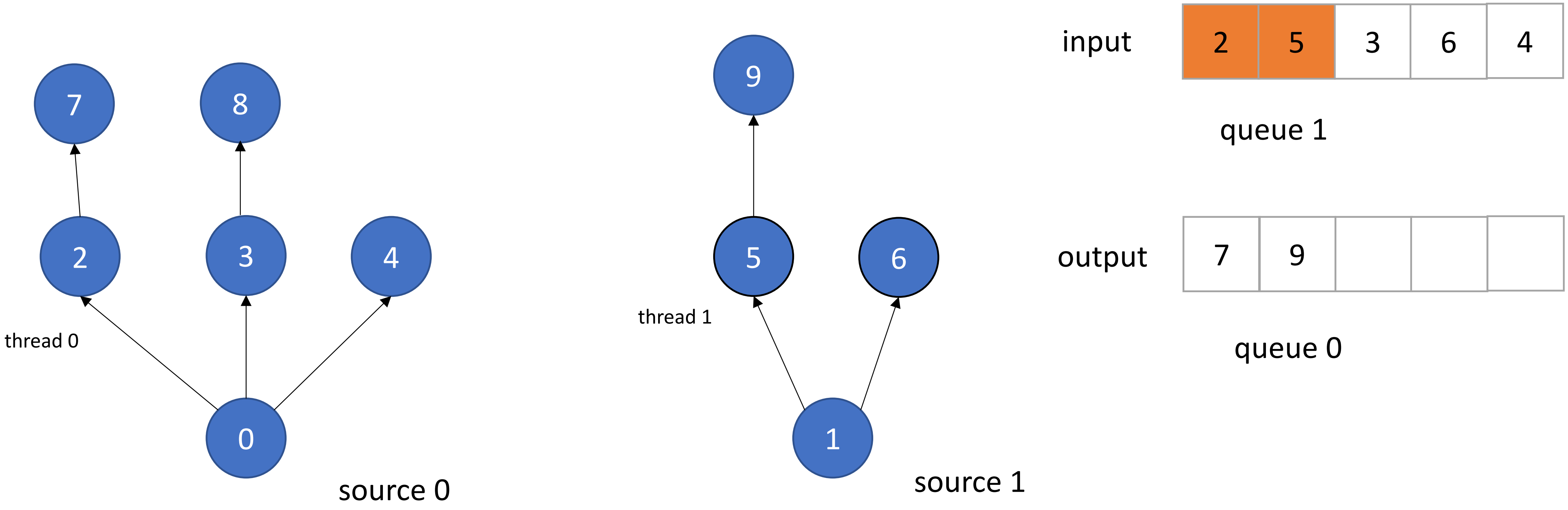
Input/Output Queues

- Example: Information flow in graph applications:



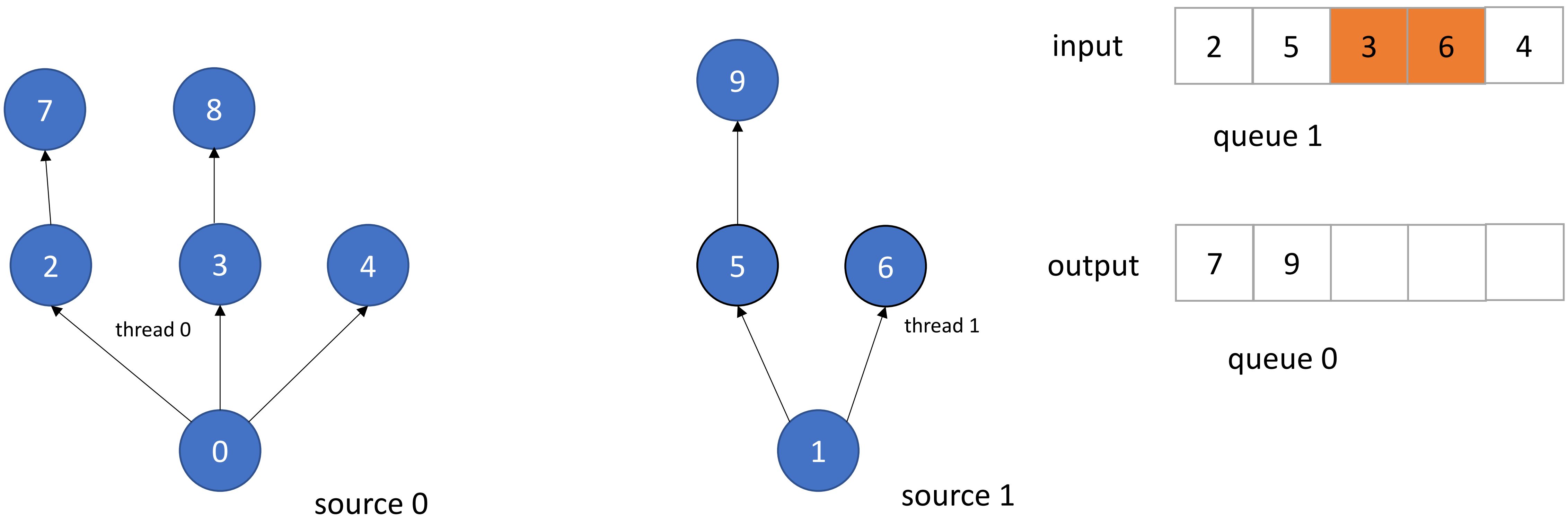
Input/Output Queues

- Example: Information flow in graph applications:



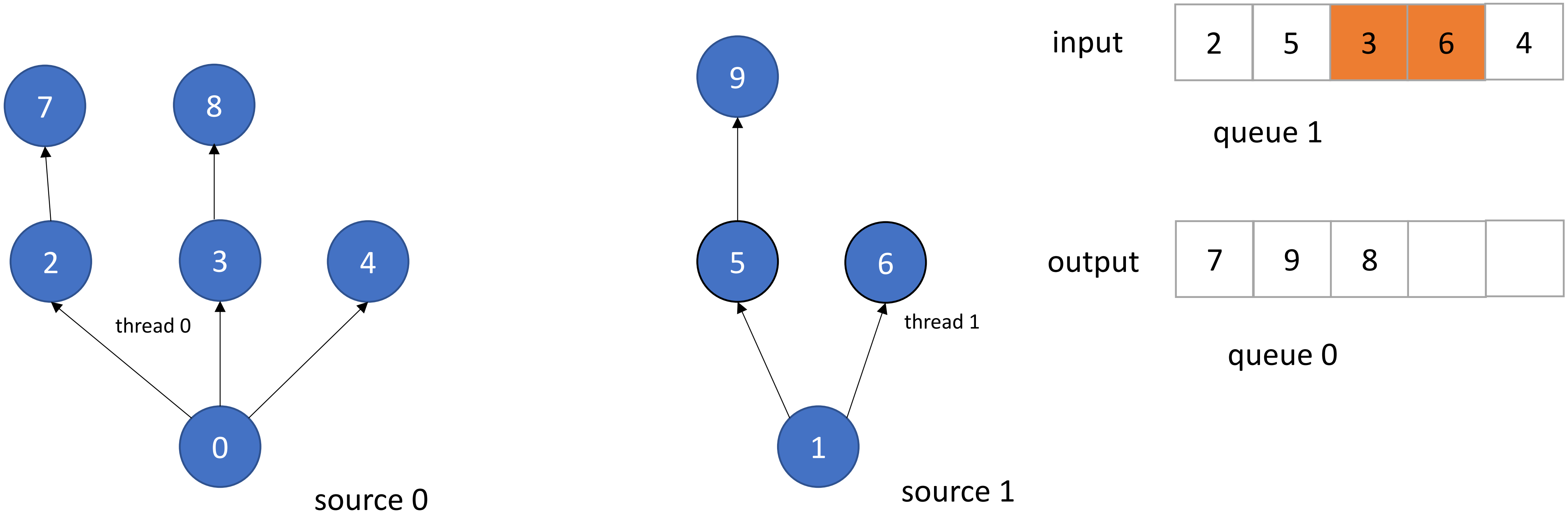
Input/Output Queues

- Example: Information flow in graph applications:



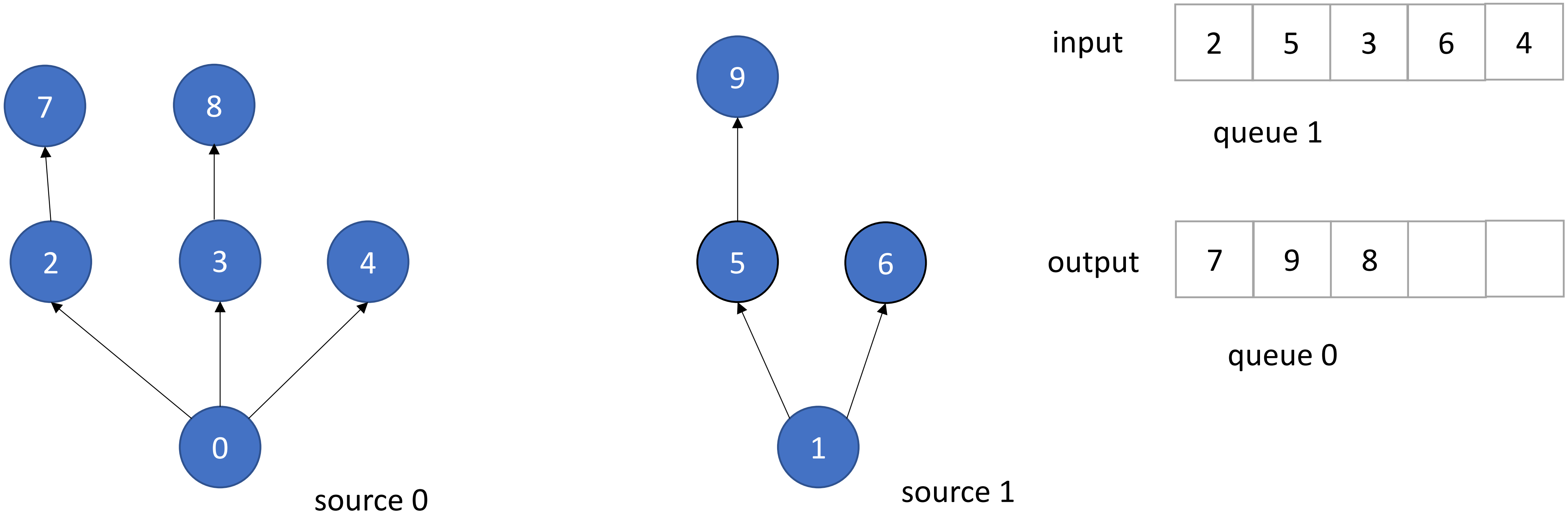
Input/Output Queues

- Example: Information flow in graph applications:



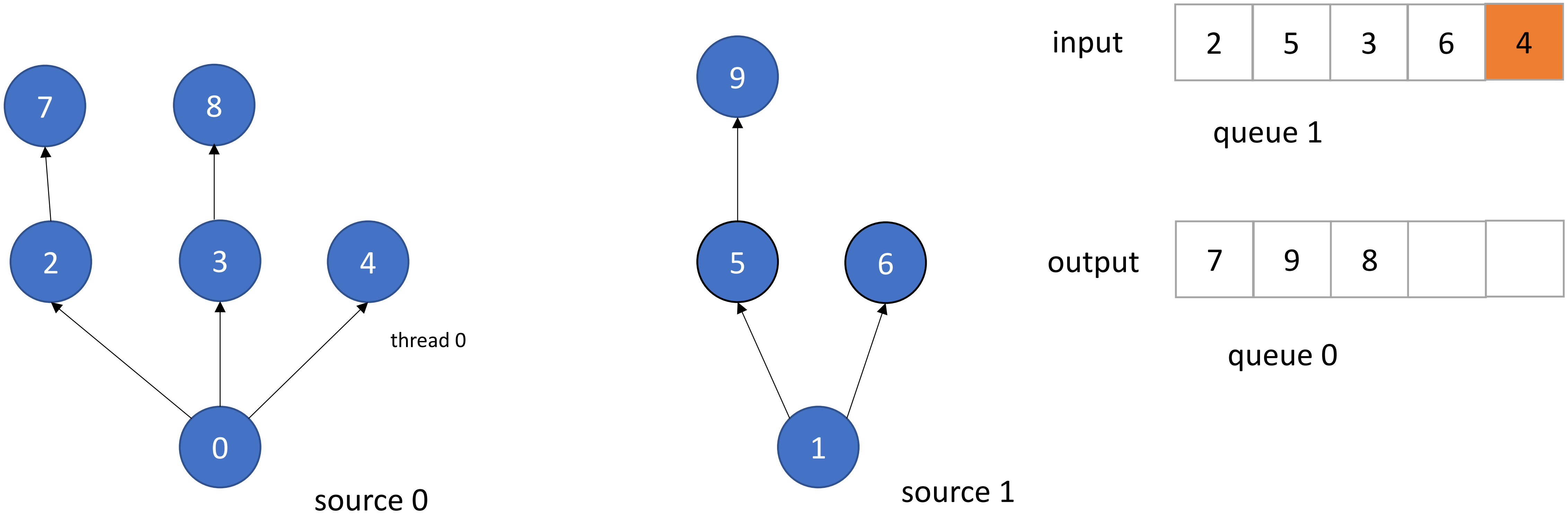
Input/Output Queues

- Example: Information flow in graph applications:



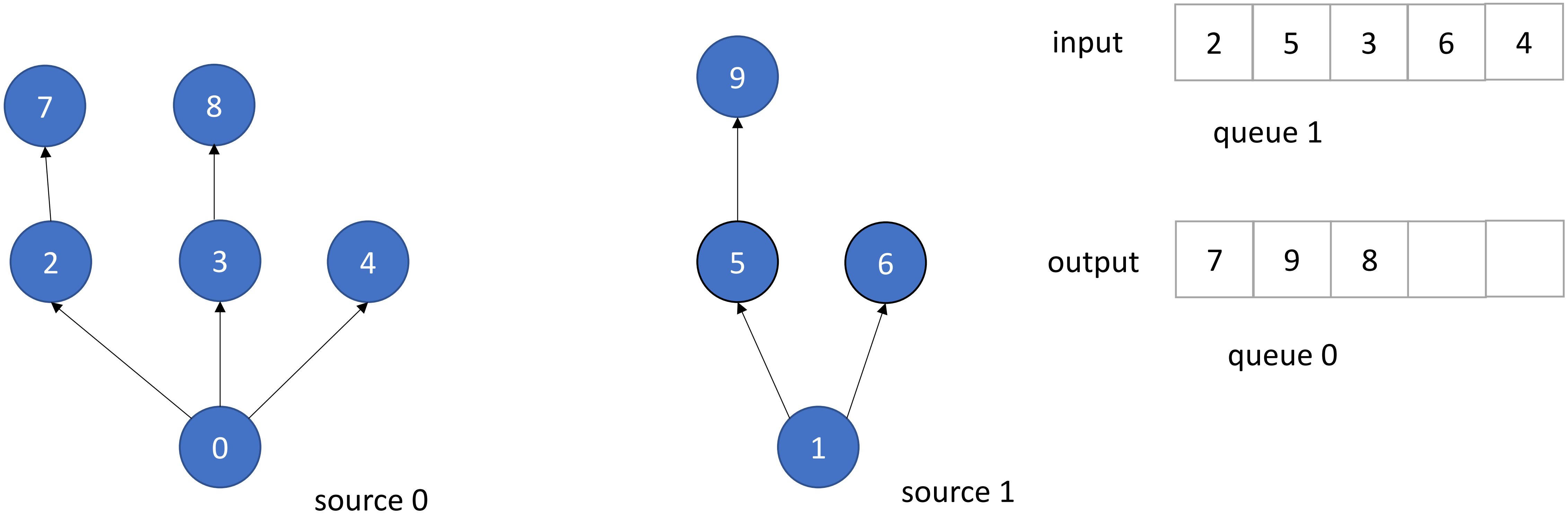
Input/Output Queues

- Example: Information flow in graph applications:



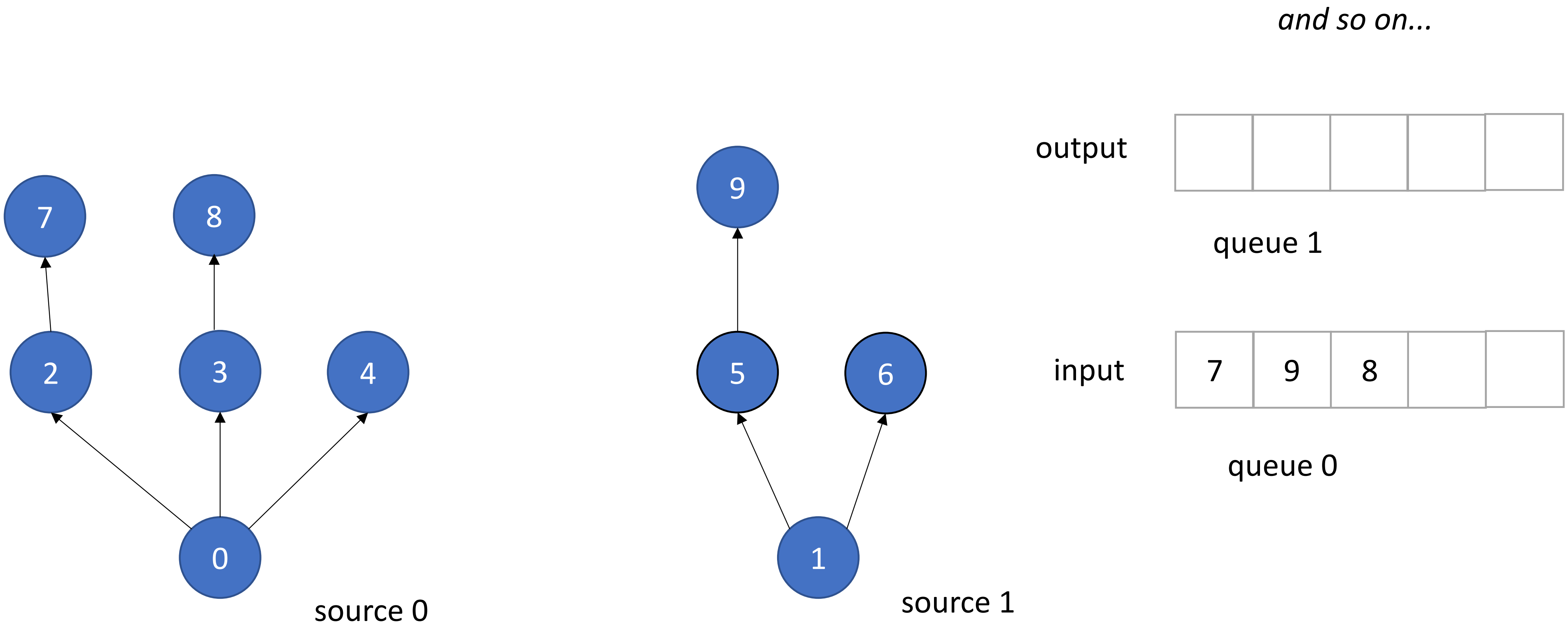
Input/Output Queues

- Example: Information flow in graph applications:



Input/Output Queues

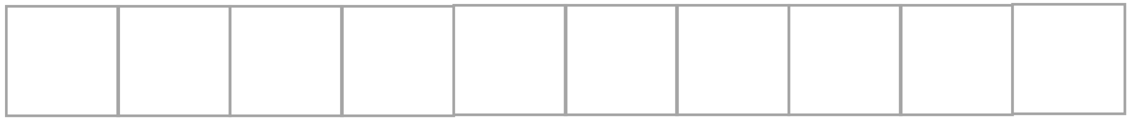
- Example: Information flow in graph applications:



Implementation

Implementation

Allocate a contiguous array

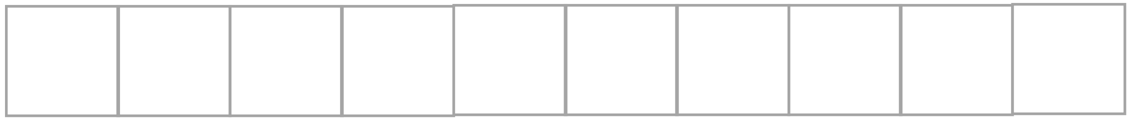


Pros:
?

Cons:
?

Implementation

Allocate a contiguous array



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

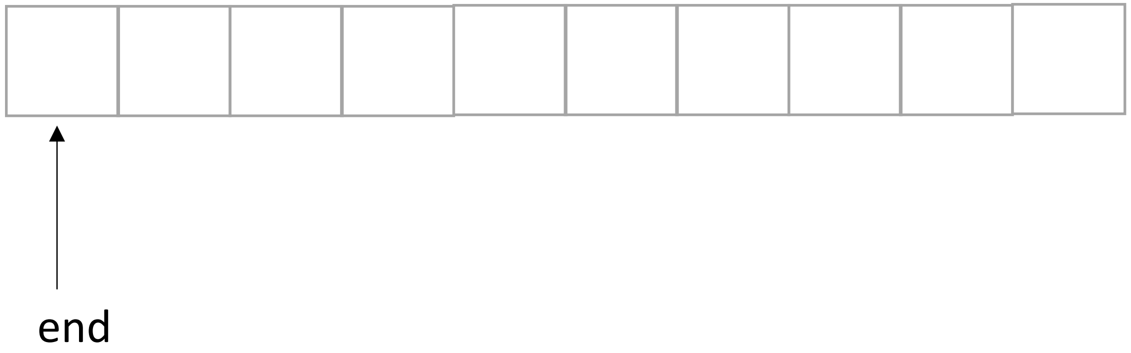
Cons:
- need to reason about overflow!

Note on terminology

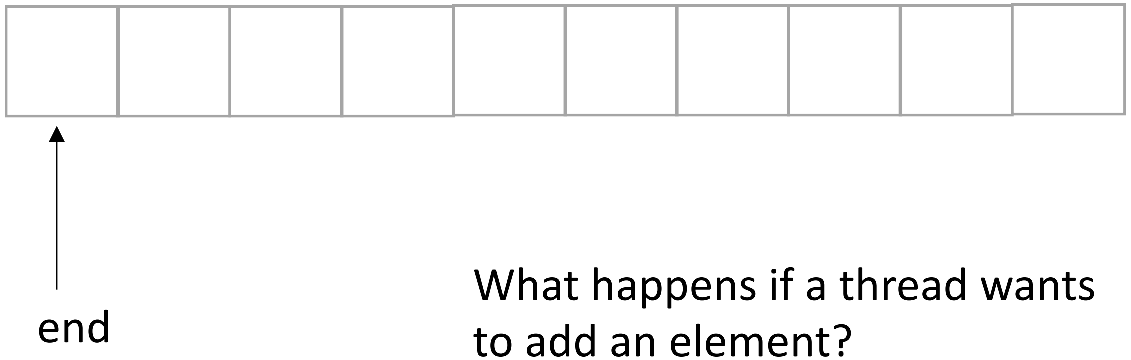
- Head/tail - often used in queue implementations, but switches when we start doing circular buffers.

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

Implementation

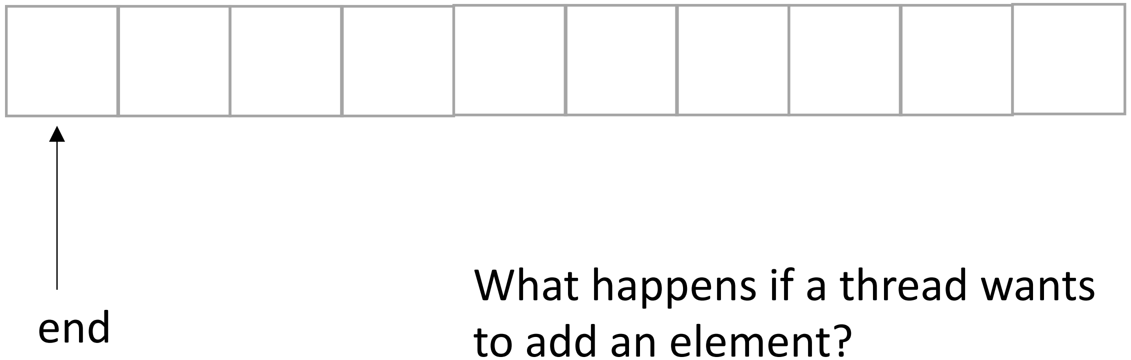


Implementation



What happens if a thread wants to add an element?

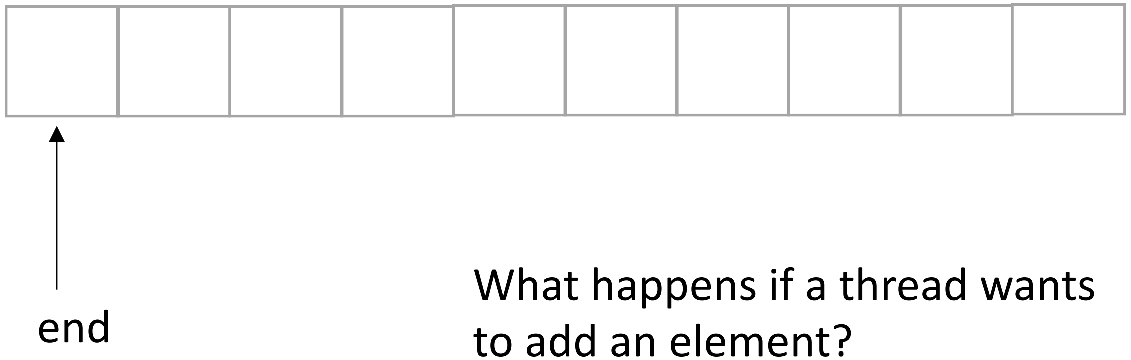
Implementation



What happens if a thread wants to add an element?

Think sequentially:

Implementation

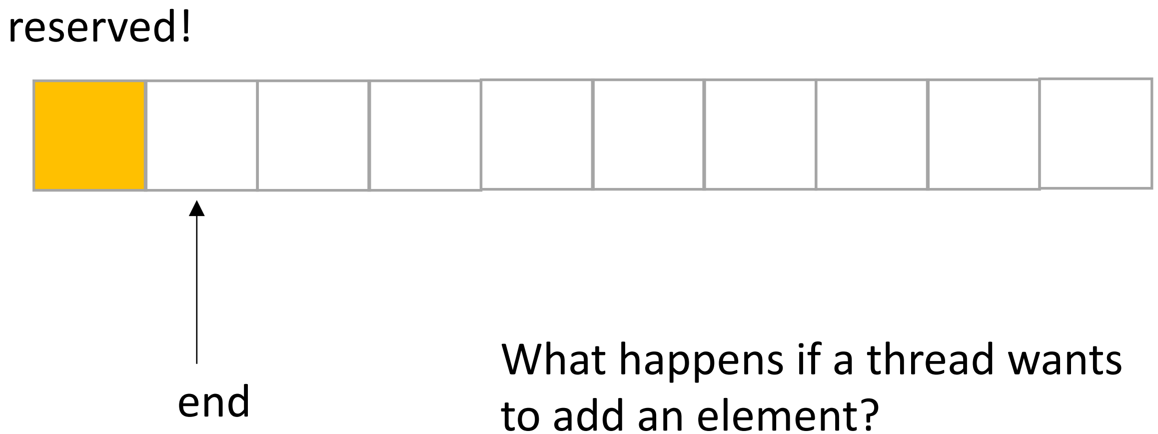


What happens if a thread wants to add an element?

Think sequentially:

*reserve a space - increment end

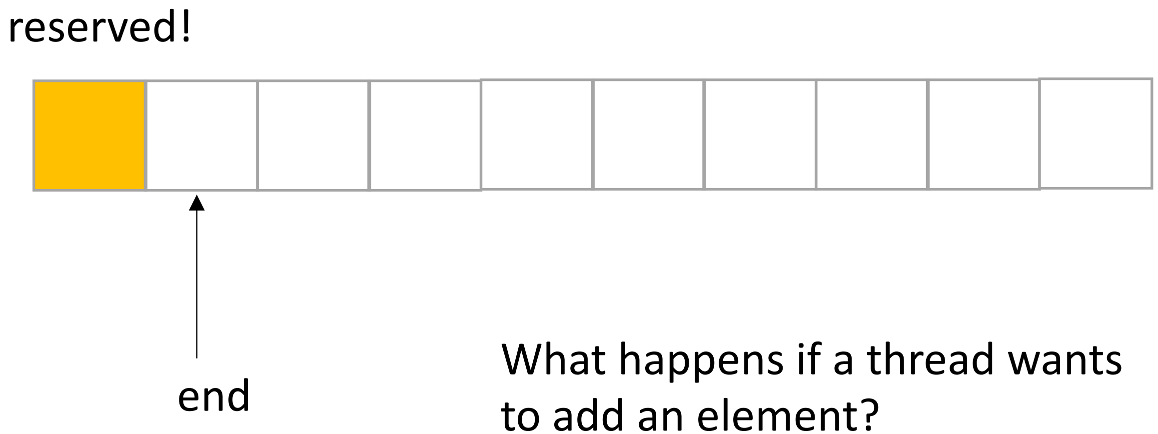
Implementation



What happens if a thread wants to add an element?

Think sequentially:
*reserve a space - increment end

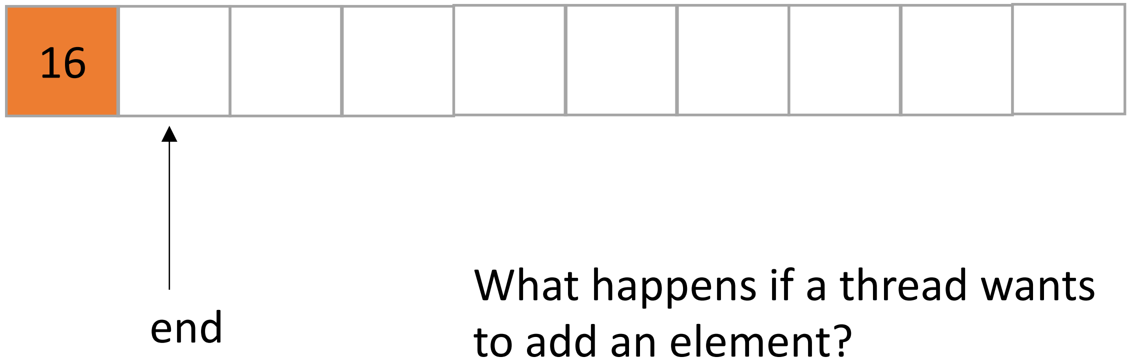
Implementation



What happens if a thread wants to add an element?

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

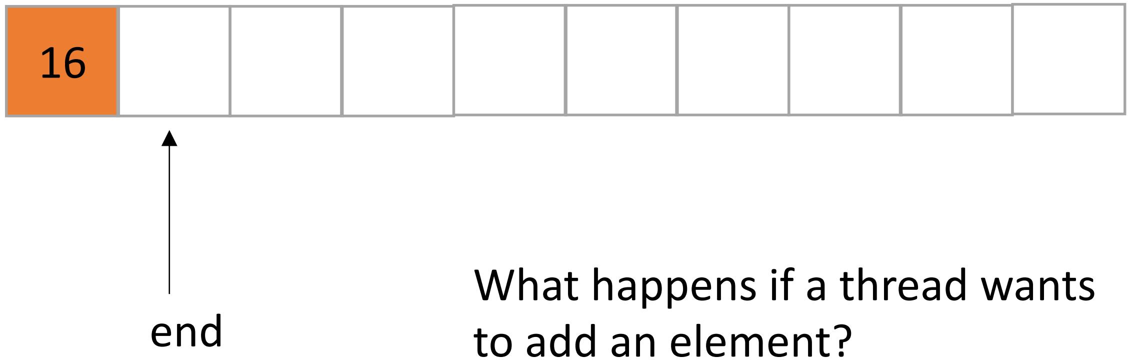
Implementation



What happens if a thread wants to add an element?

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

Implementation

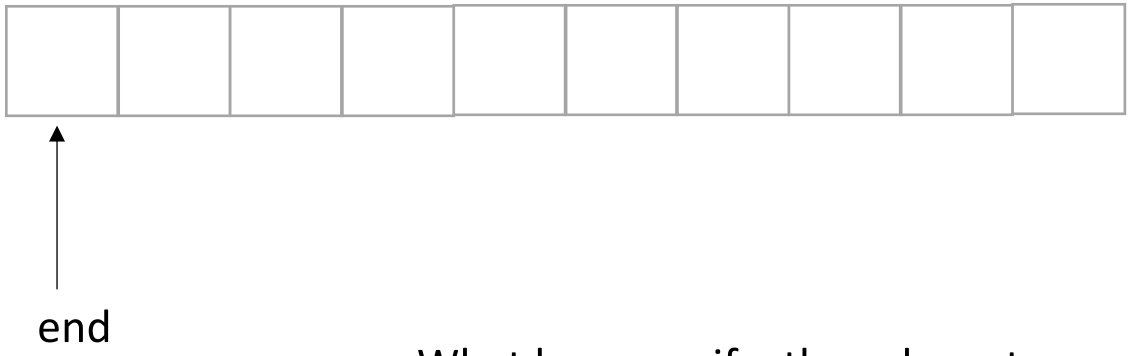


What happens if a thread wants to add an element?

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

done!

Implementation

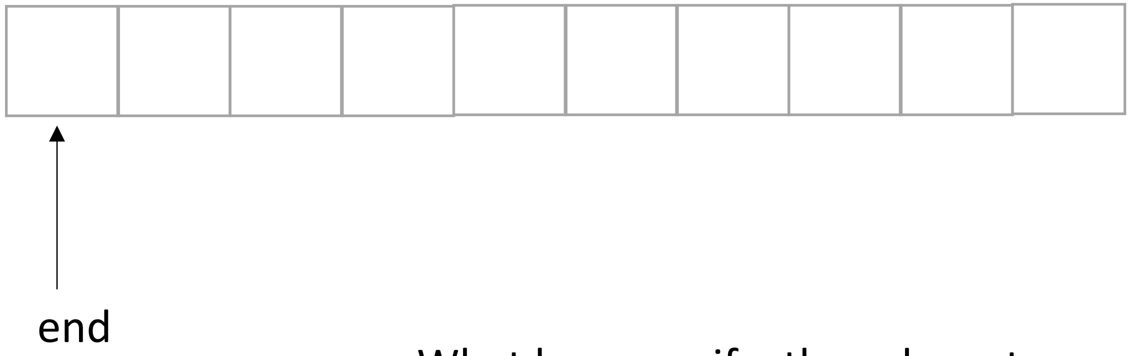


What happens if a thread wants to add an element?

Think concurrently:

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

Implementation

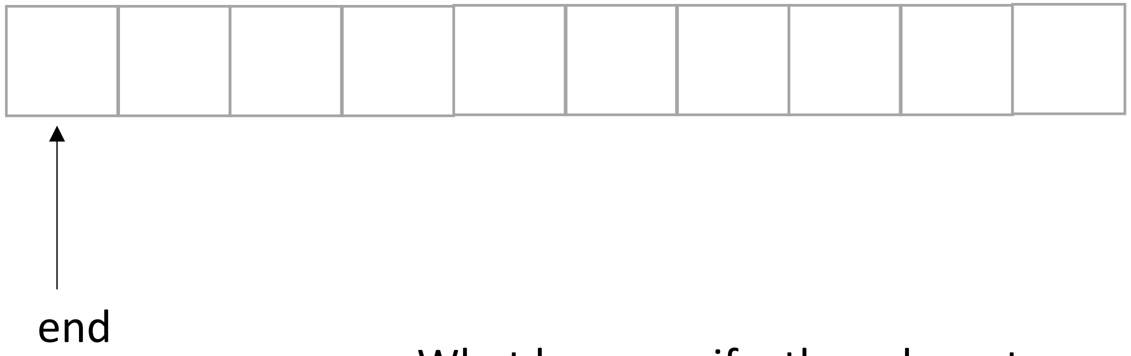


What happens if a thread wants to add an element?

Think concurrently:

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

Implementation



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

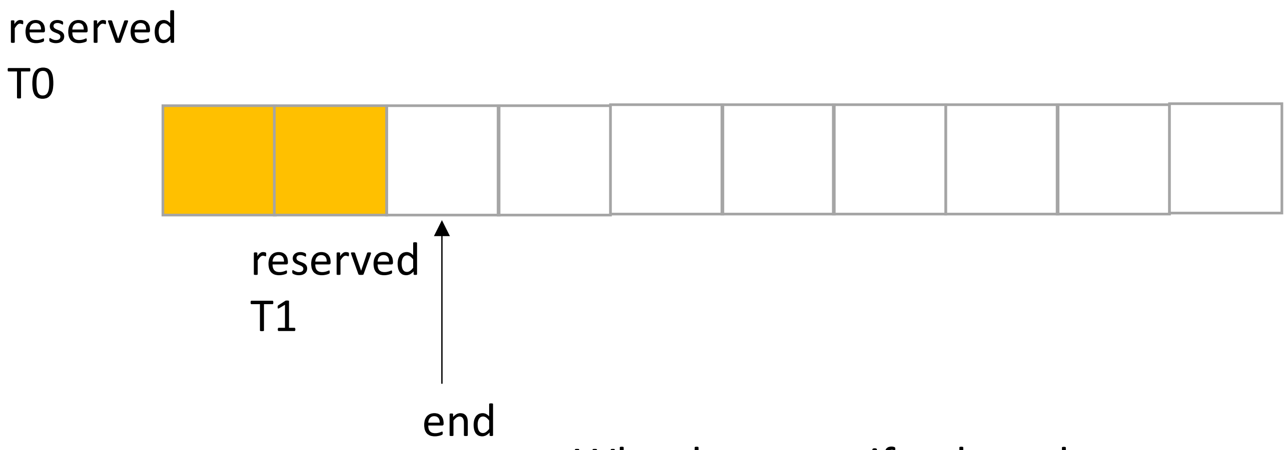
<u>Thread 1:</u>
enq(7);

What happens if a thread wants to add an element?

Think concurrently:

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

Implementation



Thread 0:
enq(6);

Thread 1:
enq(7);

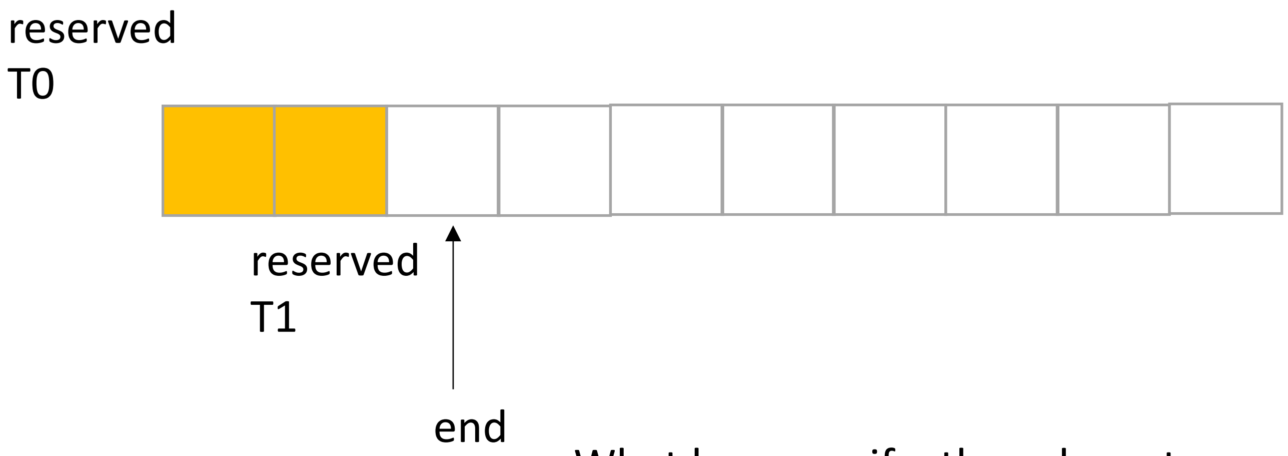
What happens if a thread wants to add an element?

Think concurrently:

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

Implementation

does it matter which order threads add their data?



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

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

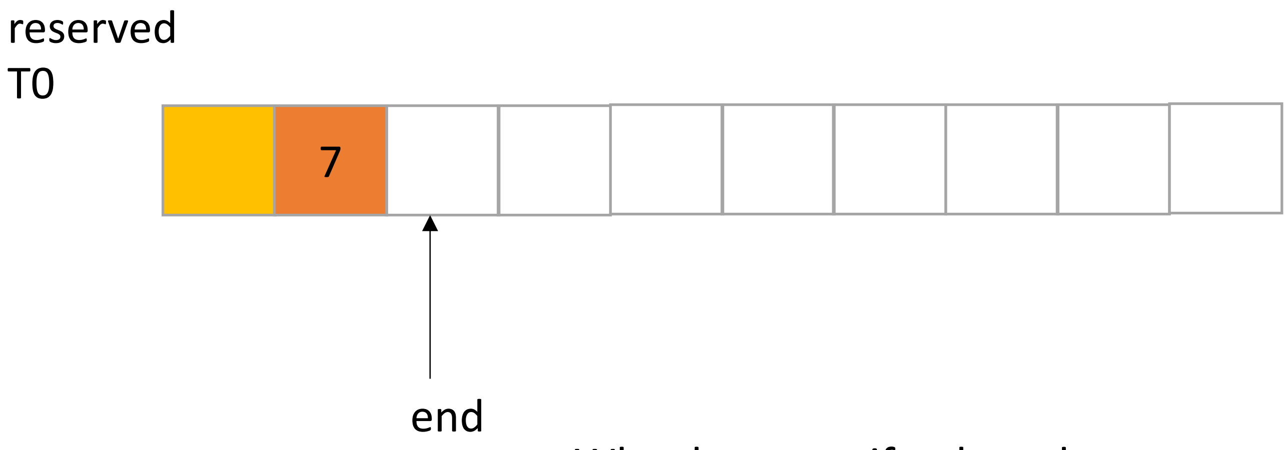
What happens if a thread wants to add an element?

Think concurrently:

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

Implementation

does it matter which order threads add their data?



What happens if a thread wants to add an element?

Think concurrently:

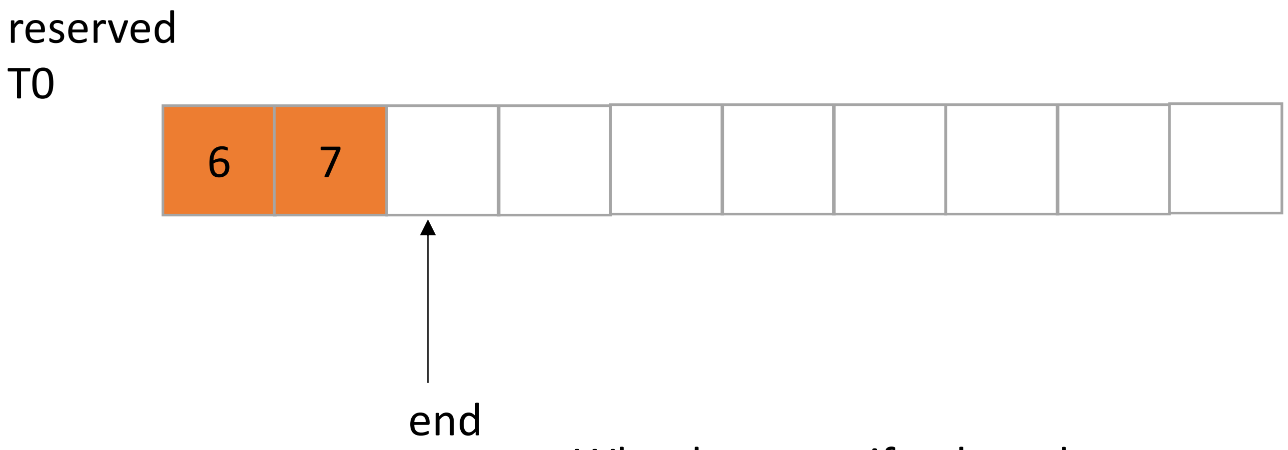
```
reserved_index = atomic_fetch_add(&end, 1);
```

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

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

Implementation

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



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

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

What happens if a thread wants to add an element?

Think concurrently:

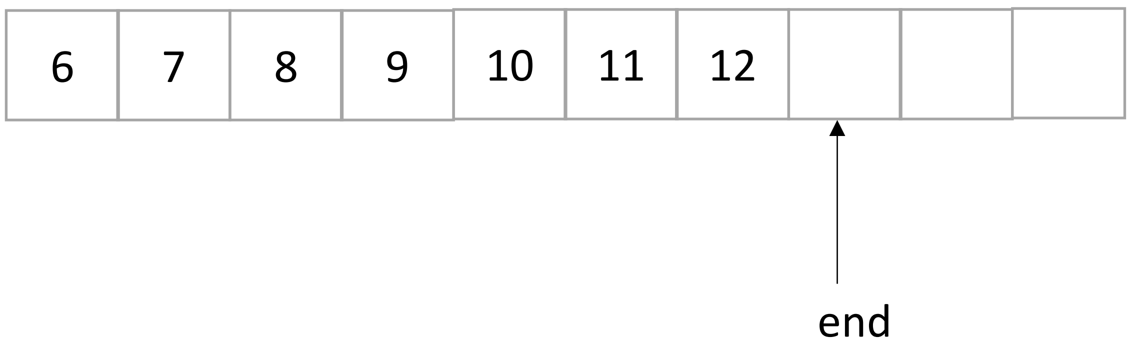
```
reserved_index = atomic_fetch_add(&end, 1);
```

```
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?

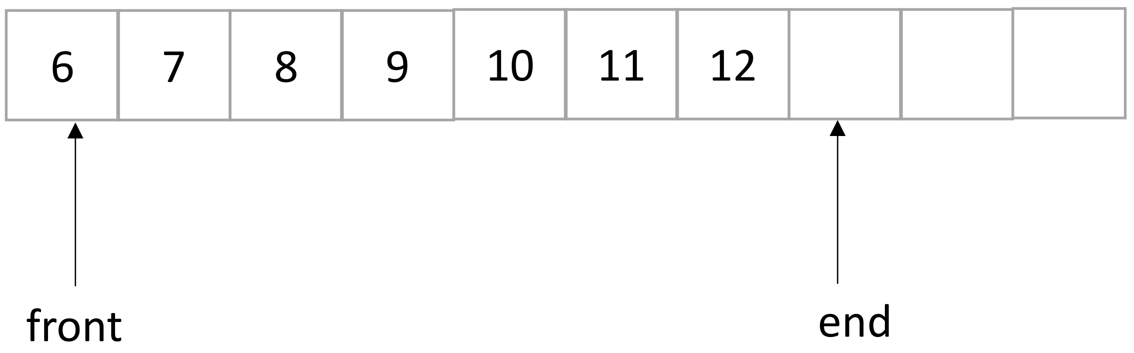
What about Input?

- Now we only do deqs



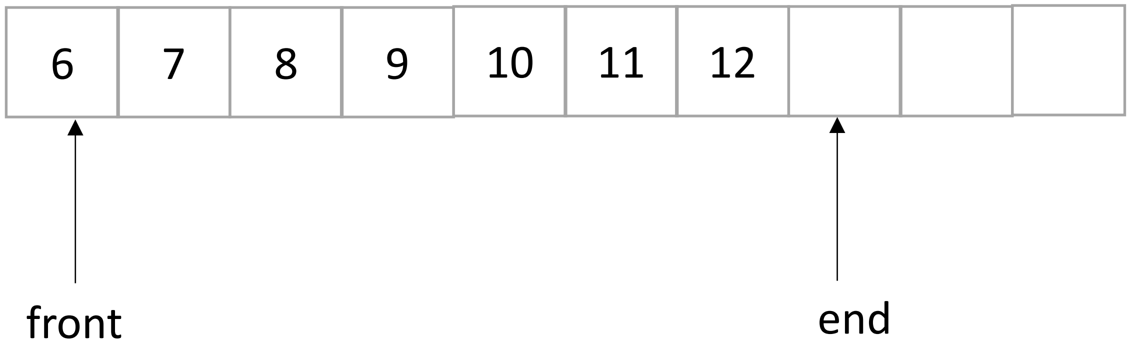
What about Input?

- Now we only do deqs



What about Input?

- Now we only do deqs



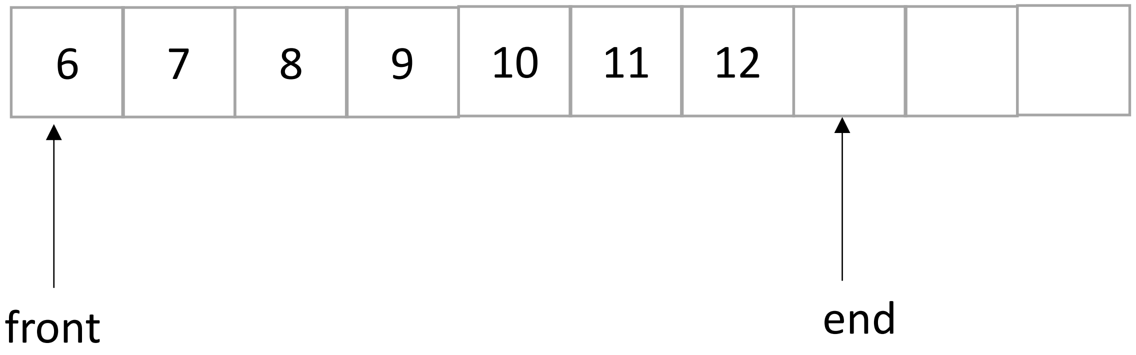
What happens if a thread wants to add an element?

Think concurrently:

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

What about Input?

- Now we only do deqs



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

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

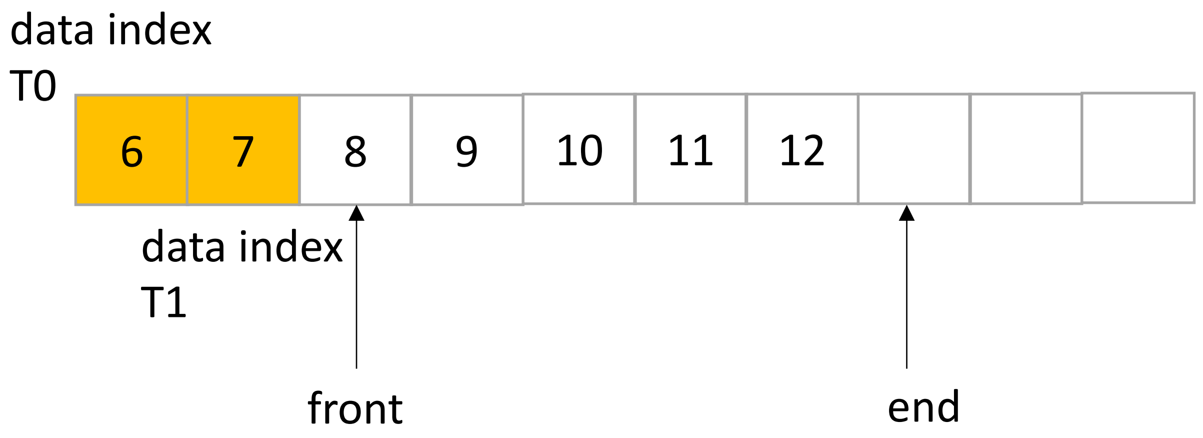
What happens if a thread wants to add an element?

Think concurrently:

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

What about Input?

- Now we only do deqs



Thread 0:
deq();

Thread 1:
deq();

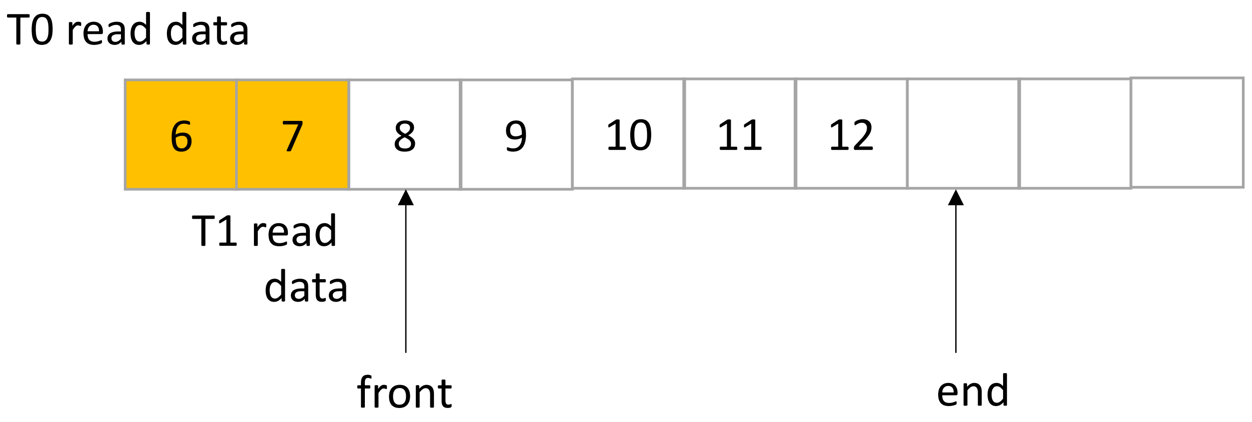
What happens if a thread wants to add an element?

Think concurrently:

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

What about Input?

- Now we only do deqs



<u>Thread 0:</u>
deq(); // reads 6

<u>Thread 1:</u>
deq(); // reads 7

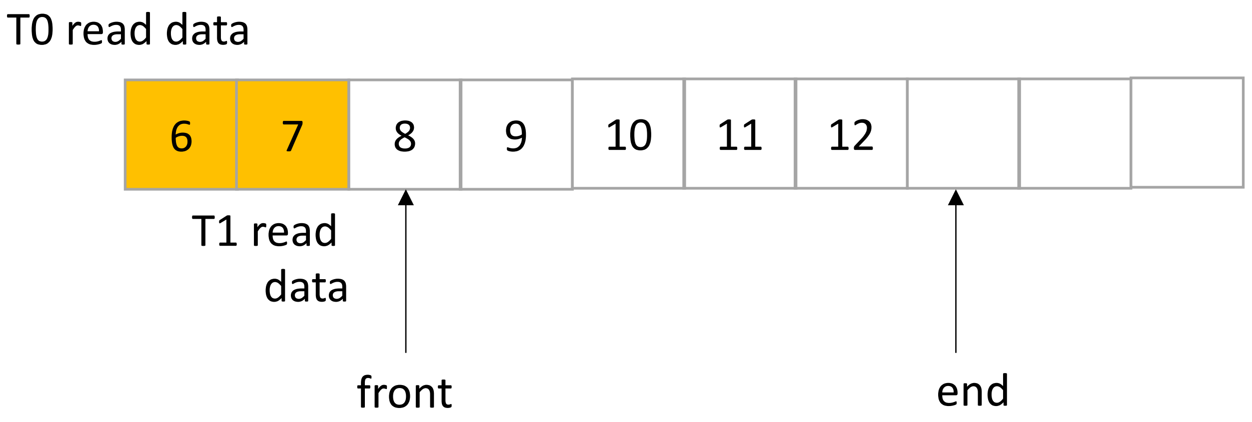
What happens if a thread wants to add an element?

Think concurrently:

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

What about Input?

- Now we only do deqs



How to implement a stack?

<u>Thread 0:</u>
deq(); // reads 6

<u>Thread 1:</u>
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 Monday!

- Get HW 2 submitted!