CSE113: Parallel Programming

Feb, 7, 2024

- **Topics**:
  - Intro to concurrent data structures
  - Bank account example
  - Specification: Sequential consistency

Announcements

- Homework 2 is due on Monday
  - Plus 3 free late days

- Plenty of office hours to get help
  - I have hours tomorrow
  - Piazza
  - Etc.

- You can share throughput numbers with each other (server and local). But don’t share code.

Announcements

- Homework 1
  - Still working on grading
  - If you see a 0, please let us know ASAP!

- Homework 3 released next Wednesday

Announcements

- New grader/tutor: Ryan Nelson

- Expect more tutoring hours and faster grading turn around

Announcements

- Midterm is next class period (Feb 12)
  - In-person test
  - 3 pages of notes front and back (but no memorization questions)
  - 10% of your grade
  - 5 or 6 short answer questions (e.g., how to reverse a linked list but with parallel programming questions)
  - Please bring a pencil/pen!

Announcements

- Starting module 3 today!

Previous quiz + Review

Previous quiz + Review

Which of the following are NOT ways that mutex implementations can encourage fair access?

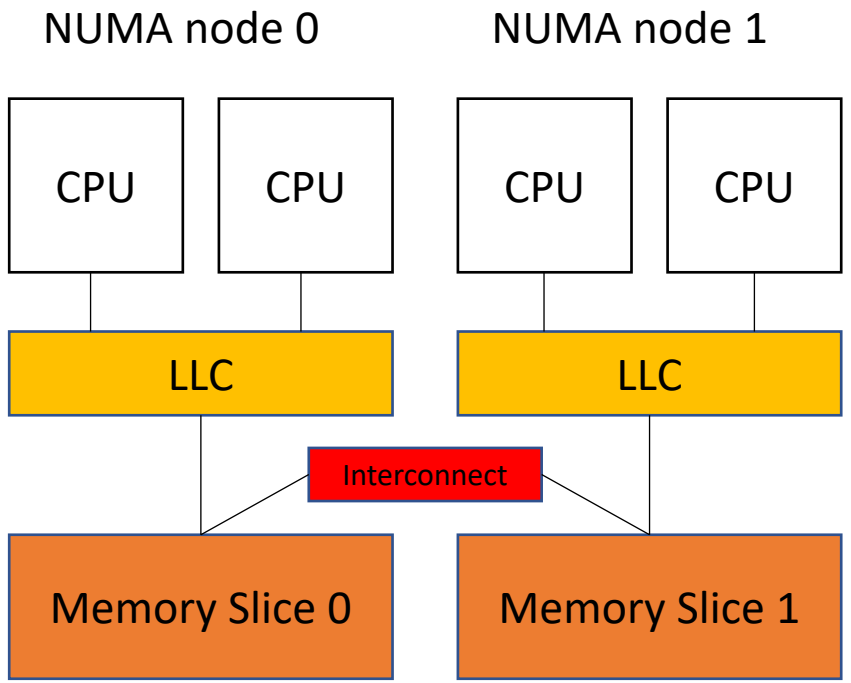
- ◯ Sleeping
- ◯ Yielding
- ◯ Using a ticket lock
- ◯ relaxed peeking

Optimizations: backoff

```
void lock(int thread_id) {
  bool e = false;
  bool acquired = false;
  while (!acquired) {
    while (flag.load(memory_order_relaxed) == true) {
      this_thread::yield();
    }
    e = false;
    acquired = atomic_compare_exchange_strong(&flag, &e, true);
  }
}
```

Optimization: Hierarchical locks

- Any sort of communication is very expensive:
  - Spinning triggers expensive coherence protocols.
  - cache flushes between NUMA nodes is expensive (transferring memory between critical sections)



```
void lock(int thread_id) {
  int e = -1;
  bool acquired = false;
  while (acquired == false) {
    acquired = atomic_compare_exchange_strong(&m_owner, &e, thread_id);

    if (thread_id/2 != e/2) {
      this_thread::sleep_for(10ms);
    }
    else {
      this_thread::sleep_for(1ms);
    }
    e = -1;
  }
}
```

Previous quiz + Review

A reader-writer mutex allows multiple readers in the critical section, mutliple writers in the critical section, but never a combination of readers and writers.

- ◯ True
- ◯ False

Previous quiz + Review

If you are an expert in how your code will compile to machine instructions, it is okay to have data conflicts in your code.

- ◯ True
- ◯ False

Previous quiz + Review

Why is the compare-and-swap operation required after the relaxed peeking sees that the mutex is available?

Optimizations: backoff

```
void lock(int thread_id) {
  bool e = false;
  bool acquired = false;
  while (!acquired) {
    while (flag.load(memory_order_relaxed) == true) {
      this_thread::yield();
    }
    e = false;
    acquired = atomic_compare_exchange_strong(&flag, &e, true);
  }
}
```

Final thoughts on Module 2

Data conflicts

- Data conflicts are undefined
  - Compiler can do crazy things
  - interleavings cause bugs that are extremely rare
- Your code should use mutexes to avoid data conflicts!
- What happens when you don’t?

Horrible data conflicts in the real world

Therac 25: a radiation therapy machine

- Between 1987 and 1989 a software bug caused 6 cases where radiation was massively overdosed
- Patients were seriously injured and even died.
- Bug was root caused to be a data conflict.

- https://en.wikipedia.org/wiki/Therac-25
- https://www.youtube.com/watch?v=Ap0orGCiou8

Horrible data conflicts in the real world

2003 NE power blackout

- second largest power outage in USA history: 55 million people were effected
- NYC was without power for 2 days, estimated 100 deaths
- Root cause was a data conflict

- https://en.wikipedia.org/wiki/Northeast_blackout_of_2003

But checking for data conflicts is hard...

- Tools are here to help (Professor Flanagan is famous in this area)

How do they work?

- Two approaches
- **Happens-before**: build a partial order of mutex lock/unlocks. Any memory access that can’t be ordered in this partial order is a conflict.
- **Lockset**: Every shared memory location has is associated with a set of locks. Refine the lockset for every access and evaluate the final result.

Dynamic Analysis

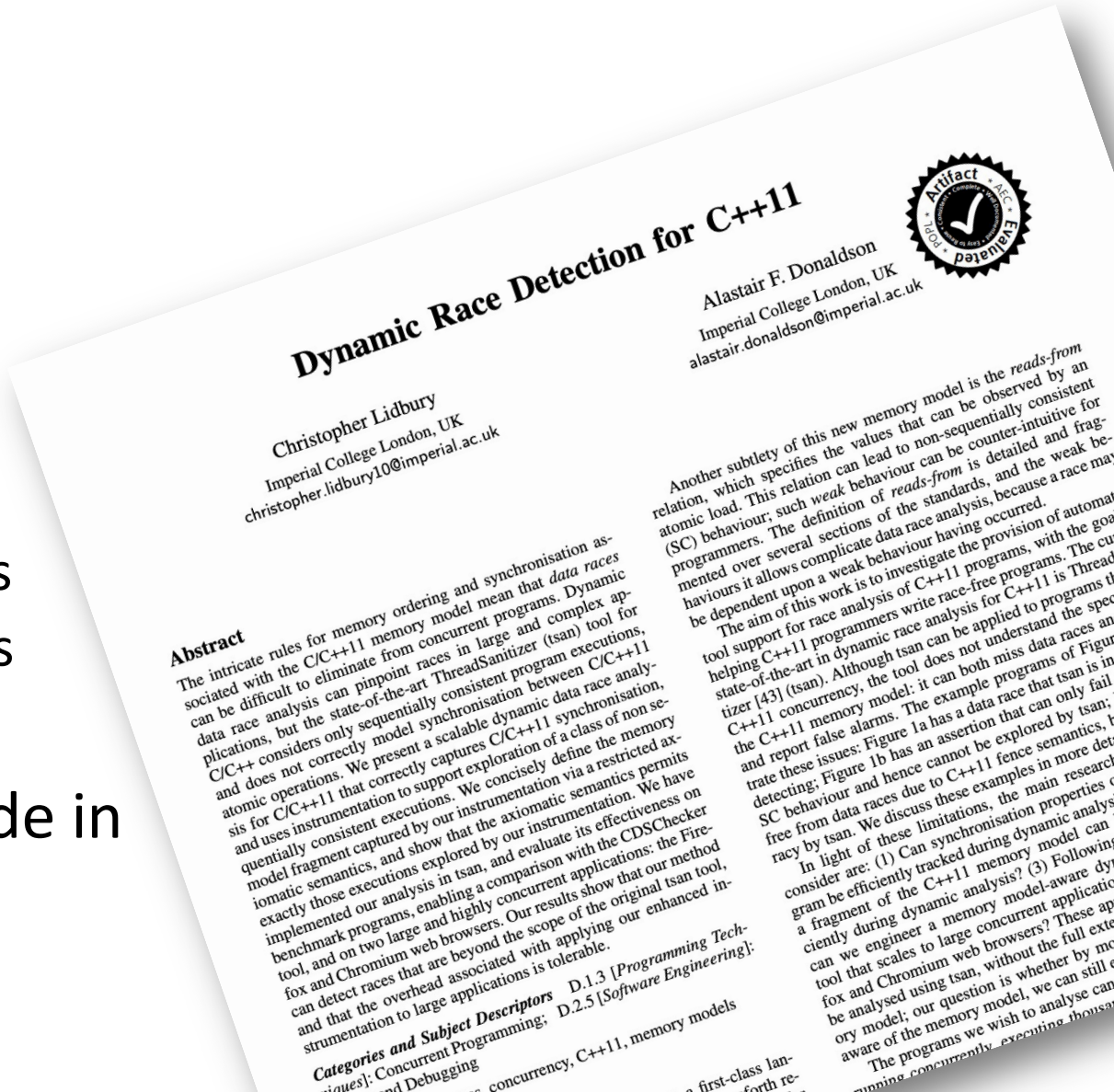
- Thread sanitizer:
  - a compiler pass built into Clang
  - About 10x overhead when you run the program
  - Identifies data conflicts and deadlocks

Static Analysis

- Facebook Infer:
  - Statically checks for many issues (memory safety, assertions)
  - Can check for races in concurrent classes
  - Main support is for Java, although they claim support for C++

Current state of data conflicts

- A recent tool:
  - Checks for C++ races
  - Scales to large programs
- Reports:
  - Chrome has 6 unresolved data-conflicts
  - Firefox has 52 unresolved data-conflicts
- Difficult to fix! 6.7 million lines of code in Chrome



Dynamic Race Detection for C++11

Christopher Lidbury
Imperial College London, UK
christopher.lidbury10@imperial.ac.uk

Alastair F. Donaldson
Imperial College London, UK
alastair.donaldson@imperial.ac.uk

Abstract

The intricate rules for memory ordering and synchronisation associated with the C/C++11 memory model mean that *data races* can be difficult to eliminate from concurrent programs. Dynamic data race analysis can pinpoint races in large and complex applications, but the state-of-the-art ThreadSanitizer (tsan) tool for C/C++ considers only sequentially consistent program executions, and does not correctly model synchronisation between C/C++11 atomic operations. We present a scalable dynamic data race analysis for C/C++11 that correctly captures C/C++11 synchronisation, and uses instrumentation to support exploration of a class of non sequentially consistent executions. We concisely define the memory model fragment captured by our instrumentation via a restricted axiomatic semantics, and show that the axiomatic semantics permits exactly those executions explored by our instrumentation. We have implemented our analysis in tsan, and evaluate its effectiveness on benchmark programs, enabling a comparison with the CDSChecker tool, and on two large and highly concurrent applications: the Firefox and Chromium web browsers. Our results show that our method can detect races that are beyond the scope of the original tsan tool, and that the overhead associated with applying our enhanced instrumentation to large applications is tolerable.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.2.5 [*Software Engineering*]: Debugging

concurrency, C++11, memory models

Another subtlety of this new memory model is the *reads-from* relation, which specifies the values that can be observed by an atomic load. This relation can lead to non-sequentially consistent (SC) behaviour; such *weak* behaviour can be counter-intuitive for programmers. The definition of *reads-from* is detailed and fragmented over several sections of the standards, and the weak behaviours it allows complicate data race analysis, because a race may be dependent upon a weak behaviour having occurred.

The aim of this work is to investigate the provision of automat
tool support for race analysis of C++11 programs, with the goa
helping C++11 programmers write race-free programs. The cu
state-of-the-art in dynamic race analysis for C++11 is Thread
tizer [43] (tsan). Although tsan can be applied to programs th
the C++11 concurrency, the tool does not understand the spec
C++11 memory model: it can both miss data races an
and report false alarms. The example programs of Figur
trate these issues: Figure 1a has a data race that tsan is in
detecting; Figure 1b has an assertion that can only fail
SC behaviour and hence cannot be explored by tsan;
free from data races due to C++11 fence semantics, b
racy by tsan. We discuss these examples in more det

In light of these limitations, the main research
consider are: (1) Can synchronisation properties c
gram be efficiently tracked during dynamic analys
a fragment of the C++11 memory model can
ciently during dynamic analysis? (3) Following
can we engineer a memory model-aware dy
tool that scales to large concurrent applicatio
fox and Chromium web browsers? These ap
be analysed using tsan, without the full exte
ory model; our question is whether by mo
aware of the memory model, we can still e

The programs we wish to analyse can

Summary

- Avoid data conflicts! They can cause serious bugs that trigger very very very rarely. (heisenbugs).
  - Better to use too many mutexes than not enough

- Use tools to help you!
  - Infer can helps with Java
  - Thread sanitizer helps with C++

On to new stuff!

Concurrent object motivation

- Programming basics cover a set of primitives:
  - types: ints, floats, bools
  - functions: call stacks, recursion

Concurrent object motivation

- Programming basics cover a set of primitives:
  - types: ints, floats, bools
  - functions: call stacks, recursion

simple example:
We can understand this!

```c
//Fibonacci Series using Recursion
#include<stdio.h>
int fib(int n)
{
   if (n <= 1)
      return n;
   return fib(n-1) + fib(n-2);
}

int main ()
{
  int n = 9;
  printf("%d", fib(n));
  getchar();
  return 0;
}
```

https://www.geeksforgeeks.org/c-program-for-n-th-fibonacci-number/

Concurrent object motivation

- How does it look moving into a more complicated setting?

Concurrent object motivation

- How does it look moving into a more complicated setting?
  - Hello world Android app:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Log.d("MainActivity", "Hello World");
}
```

https://developer.android.com/codelabs/android-training-hello-world#7

Concurrent object motivation

- How does it look moving into a more complicated setting?
  - Hello world Android app:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Log.d("MainActivity", "Hello World");
}
```

what the heck is a bundle?

https://developer.android.com/codelabs/android-training-hello-world#7

Concurrent object motivation

- How does it look moving into a more complicated setting?
  - Hello world Android app:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Log.d("MainActivity", "Hello World");
}
```

what is this?

https://developer.android.com/codelabs/android-training-hello-world#7

Concurrent object motivation

- How does it look moving into a more complicated setting?
  - Hello world Android app:

- These are objects!

https://developer.android.com/codelabs/android-training-hello-world#7

Concurrent object motivation

- Objects are user-specified abstractions:
  - A collection of data (state) and methods (behavior) representing something more complicated than primitive types can express.

Concurrent object motivation

- Objects are user-specified abstractions:
  - A collection of data (state) and methods (behavior) representing something more complicated than primitive types can express.

- Examples:
  - Writing a video game? objects for enemies and players
  - Writing an IOS app? objects for buttons

Concurrent object motivation

- Objects are user-specified abstractions:
  - A collection of data (state) and methods (behavior) representing something more complicated than primitive types can express.

- Examples:
  - Writing a video game? objects for enemies and players
  - Writing an IOS app? objects for buttons

- Objects allow programmer productivity:
  - Modular
  - Encapsulation
  - Composable

Concurrent object motivation

- Objects are user-specified abstractions:
  - A collection of data (state) and methods (behavior) representing something more complicated than primitive types can express.
- Examples:
  - Writing a video game? objects for enemies and players
  - Writing an IOS app? objects for buttons
- Objects allow programmer productivity:
  - Modular
  - Encapsulation
  - Composable
- We would like objects in the concurrent setting!

Concurrent object motivation

- Note:
  - The foundations in this lecture are general, and can be widely applied to many different types of objects
  - We will focus on ”container” objects, lists, sets, queues, stacks.
  - These are:
    - Practical - used in many applications
    - Well-specified - their sequential behavior is agreed on
    - Interesting implementations - great for us to study!

Conceptual examples

- Shopping list: Going shopping with roommates

eggs
carrots
tortillas

Best case:
2x as fast (so we can get back to CSE113 homework)

Consider two people splitting the work.

Conceptual examples

- Shopping list: Going shopping with roommates

eggs
carrots
tortillas

Best case:
2x as fast (so we can get back to CSE113 homework)

What can go wrong?

Consider two people splitting the work.

Conceptual examples

- Shopping list: Going shopping with roommates

eggs
carrots
tortillas

Best case:
2x as fast (so we can get back to CSE113 homework)

What can go wrong?

We end up with duplicates

Consider two people splitting the work.

Conceptual examples

- Shopping list: Going shopping with roommates

eggs
carrots
tortillas

Best case:
2x as fast (so we can get back to CSE113 homework)

What can go wrong?

We end up with duplicates

We end up missing an item

Consider two people splitting the work.

Conceptual examples

- Shopping list: Going shopping with roommates

eggs
carrots
tortillas

Best case:
2x as fast (so we can get back to CSE113 homework)

What can go wrong?

We end up with duplicates

We end up missing an item

If my roommate decides to go surfing, then I could get stranded!

Consider two people splitting the work.

Conceptual examples

- Shopping list: Going shopping with roommates

What kind of object is the list?

eggs
carrots
tortillas

Best case:
2x as fast (so we can get back to CSE113 homework)

What can go wrong?

We end up with duplicates

We end up missing an item

If my roommate decides to go surfing, then I could get stranded!

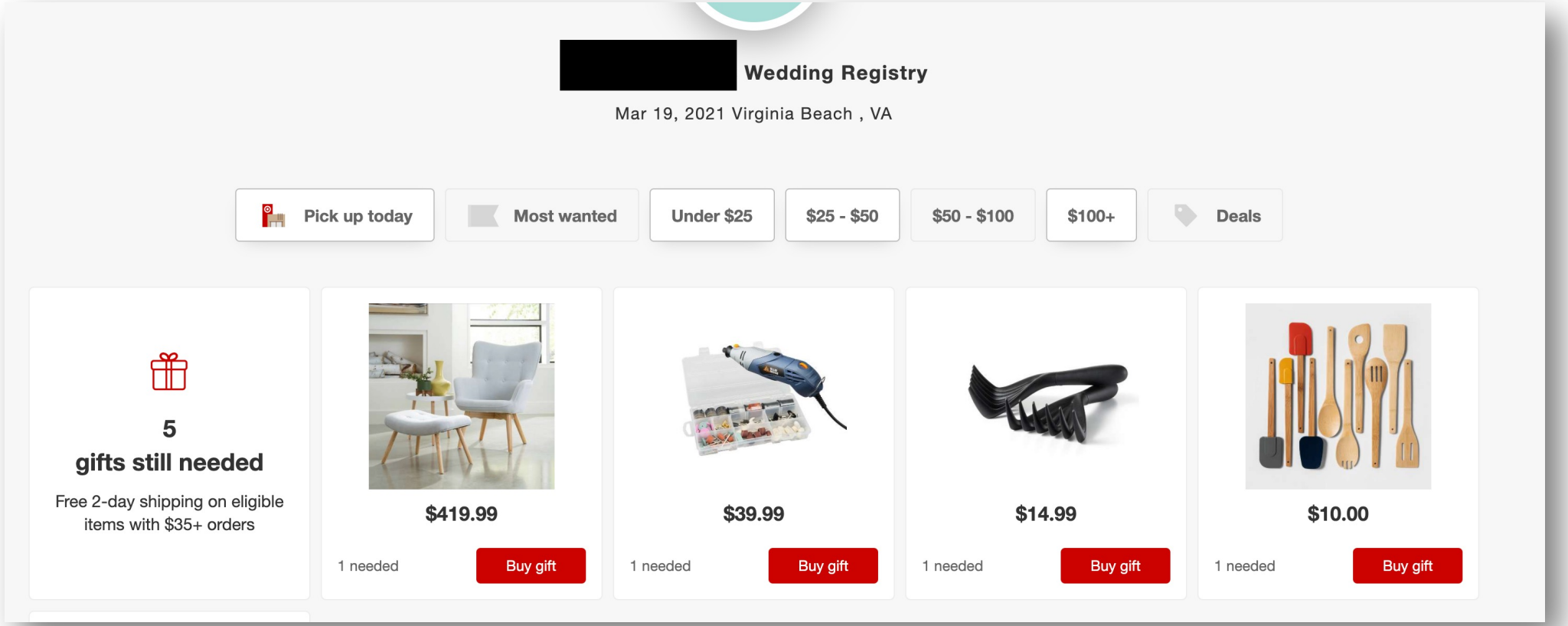
Consider two people splitting the work.

Conceptual examples

- Physically shopping with roommates is a nice conceptual example, but the example also occurs in automated systems

Conceptual examples

- Physically shopping with roommates is a nice conceptual example, but the example also occurs in automated systems



Shared memory concurrent objects

- Lets ground this even more in a shared memory system.

- Shopping cart examples mostly occur in a distributed system setting where there are many different concerns
  - Consider taking a class from Prof. Kuper or Prof. Alvaro!

Shared memory concurrent objects

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

how do we envision printf to work?

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

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

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

./a.out

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

terminal display

Shared memory concurrent objects

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

How does it actually work?

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

./a.out

concurrent queue

terminal display

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

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

Shared memory concurrent objects

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

Show example

How does it actually work?

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

concurrent queue

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

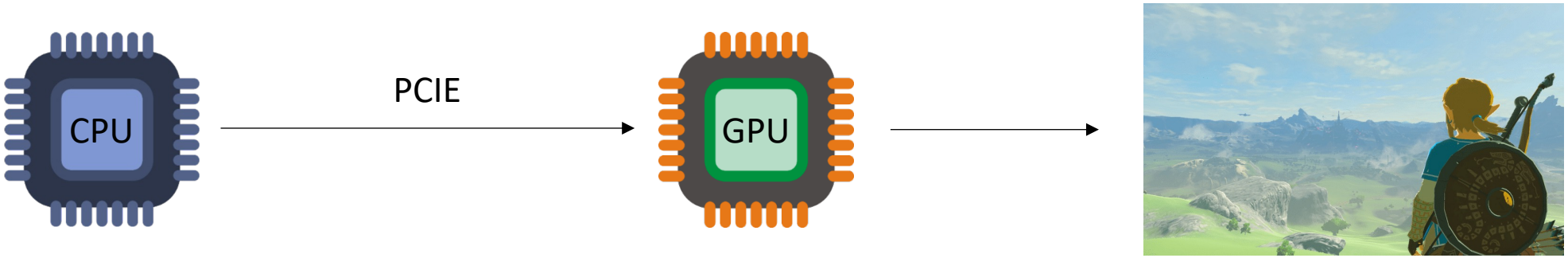
./a.out

terminal display

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

Shared memory concurrent objects

- Graphics programming



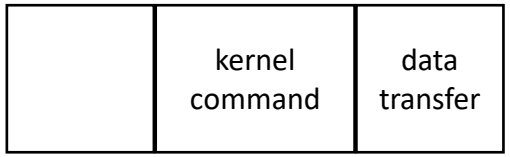
loop:
update data (data transfer)
graphics computation (kernel)

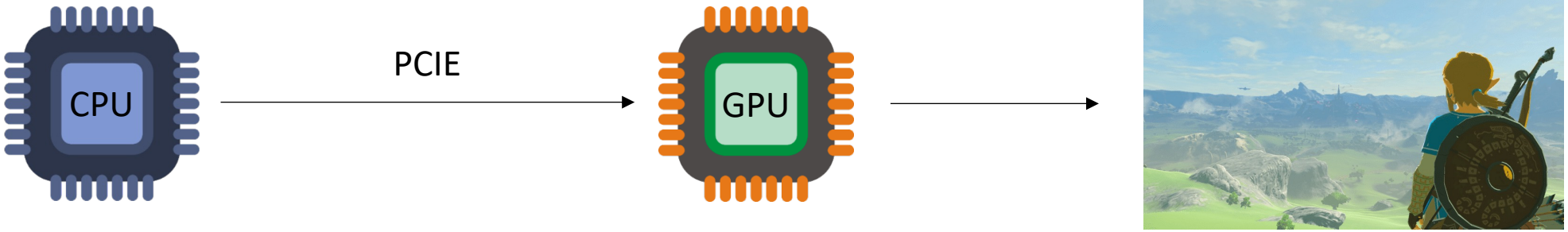
Nintendo: breath of the Wild

Shared memory concurrent objects

- Graphics programming

Vulkan/OpenCL CommandQueue





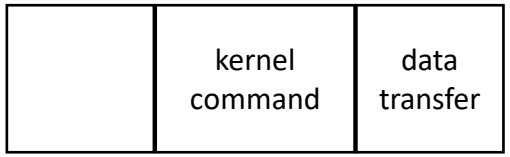
loop:
update data (data transfer)
graphics computation (kernel)

Nintendo: breath of the Wild

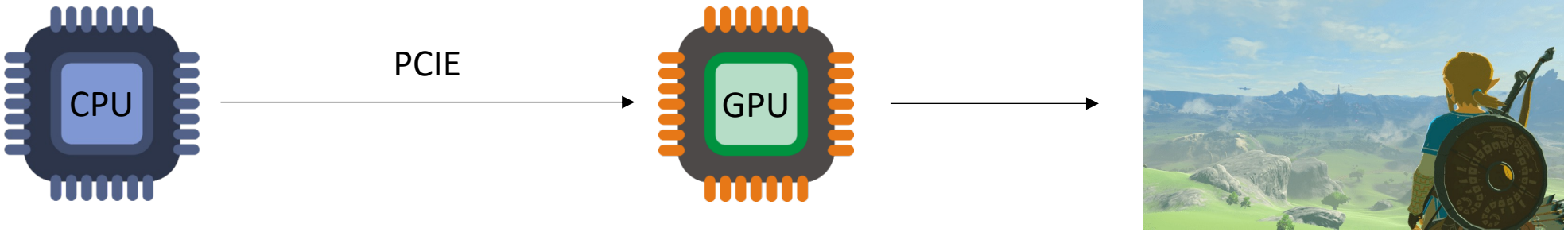
Shared memory concurrent objects

- Graphics programming

Vulkan/OpenCL CommandQueue



GPU driver concurrently reads from the queue



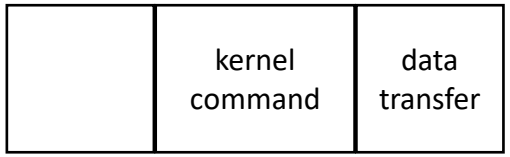
loop:
update data (data transfer)
graphics computation (kernel)

Nintendo: breath of the Wild

Shared memory concurrent objects

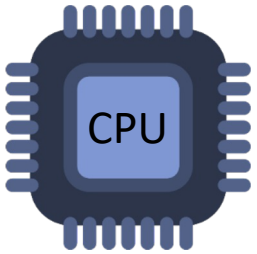
- Graphics programming

Vulkan/OpenCL CommandQueue



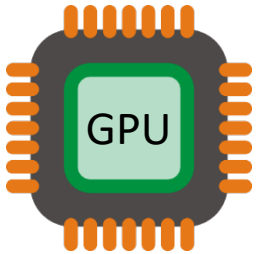
GPU driver concurrently reads from the queue

this concurrent queue enables an efficient graphics pipeline



PCIE

Transferring data for scene 2



loop:
update data (data transfer)
graphics computation (kernel)

Computation for scene 1

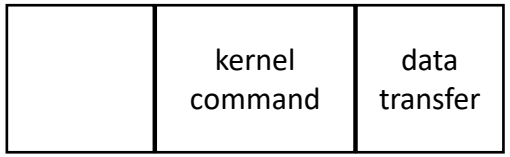
Scene 0

Nintendo: breath of the Wild

Shared memory concurrent objects

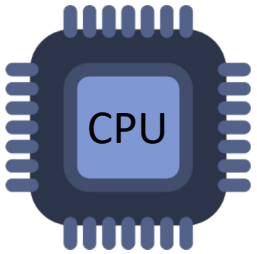
- Graphics programming

Vulkan/OpenCL CommandQueue



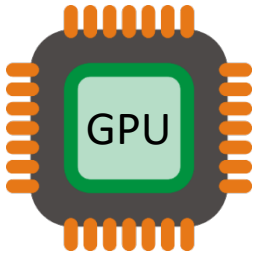
GPU driver concurrently reads from the queue

Single writer, single reader
Like in `printf`



PCIE

Transferring data for scene 2



loop:
update data (data transfer)
graphics computation (kernel)

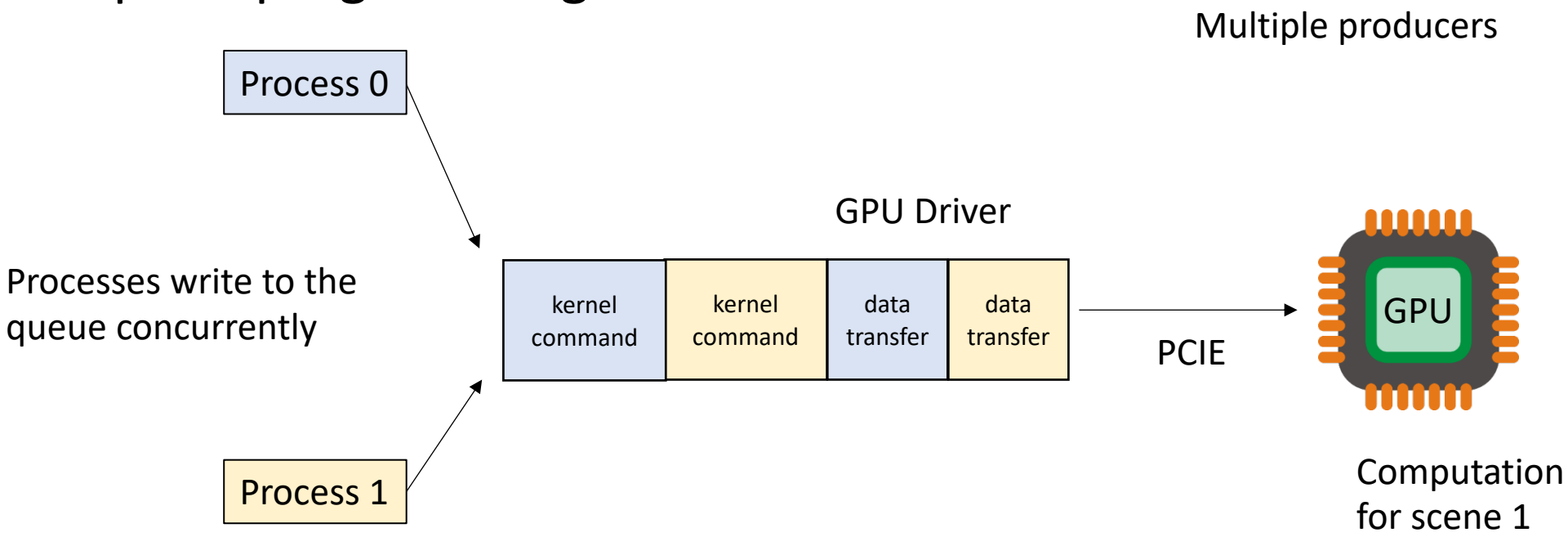
Computation for scene 1

Scene 0

Nintendo: breath of the Wild

Shared memory concurrent objects

- Graphics programming



Each process:
loop:
update data (data transfer)
graphics computation (kernel)

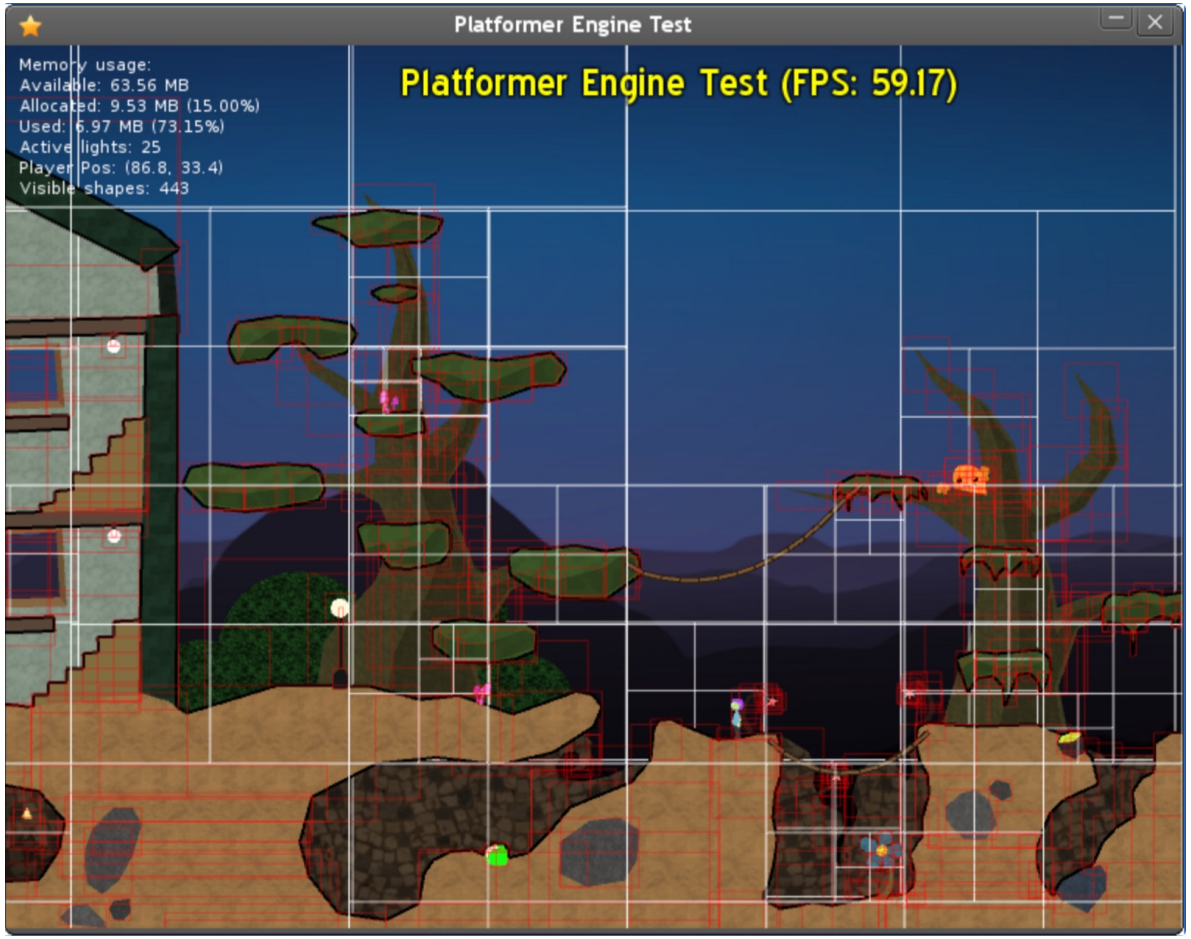
Intro to concurrent objects

- Prior examples have been infrastructural:
  - things happening behind the scenes, drivers, OS, etc.

- They also exist in standalone applications

Shared memory concurrent objects

- Quadtree/Octree

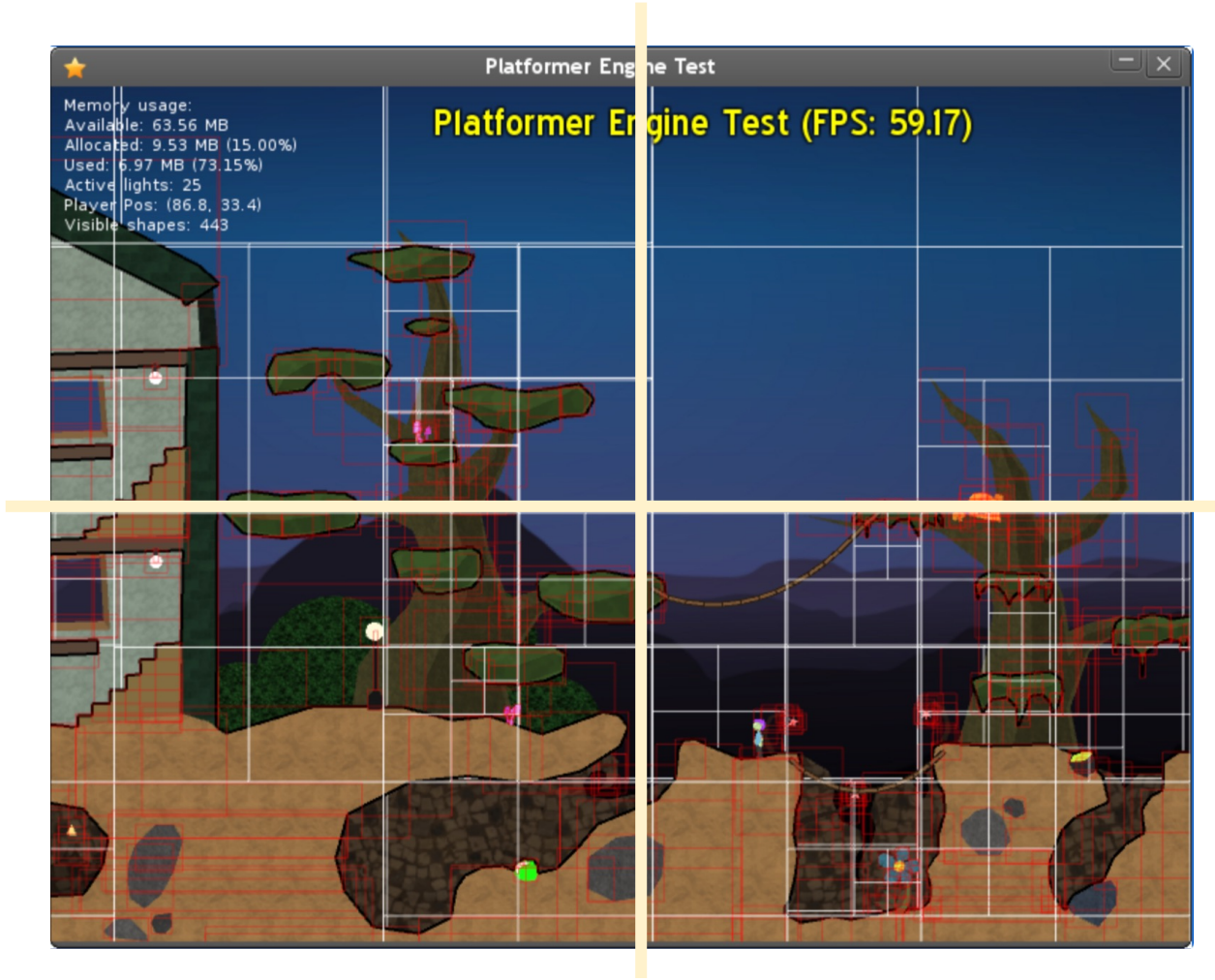


https://medium.datadriveninvestor.com/partitioning-2d-spaces-an-introduction-to-quadtrees-d95728856613

Shared memory concurrent objects

- Quadtree/Octree

recursively divide the scene giving more detail to “interesting” areas

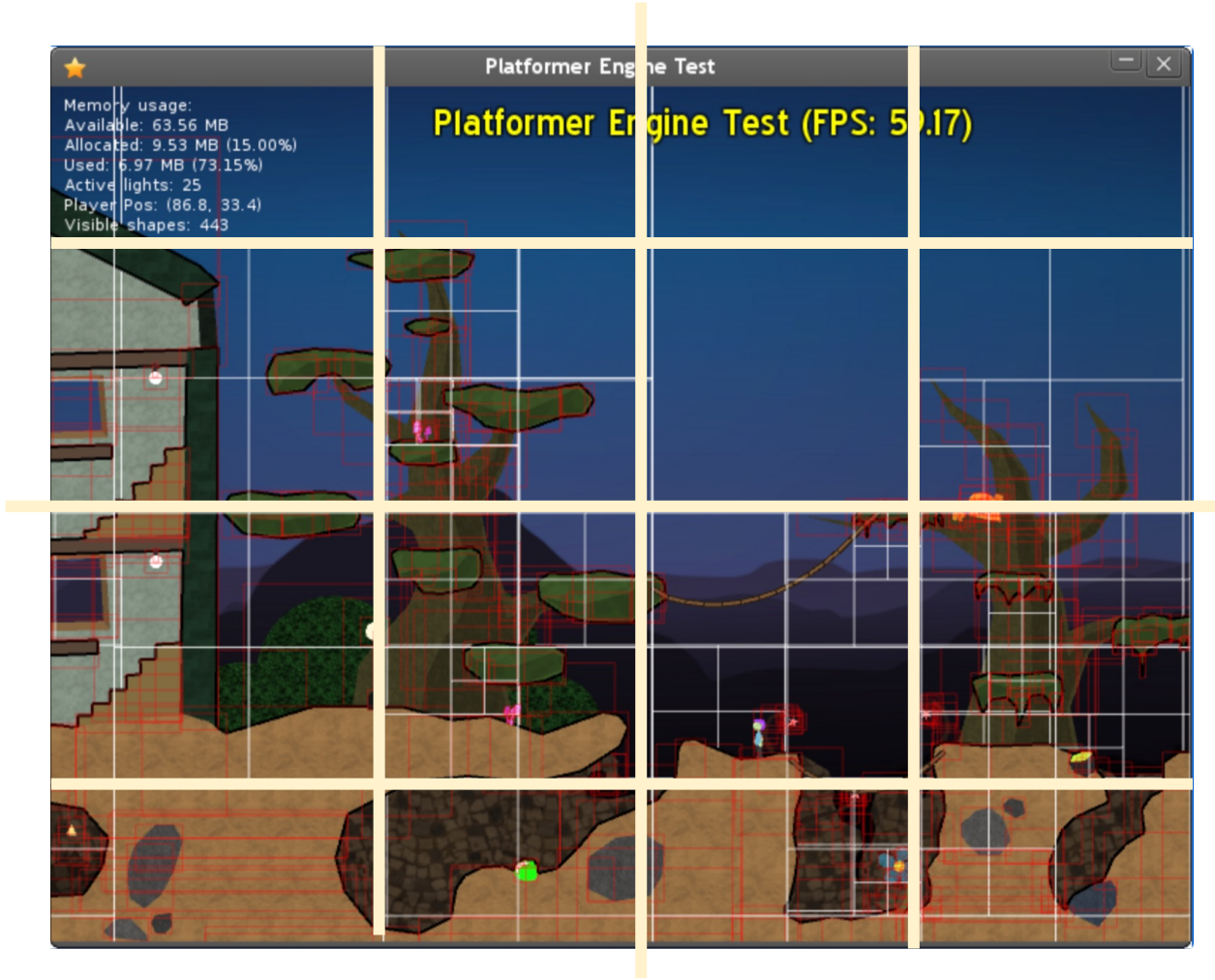


https://medium.datadriveninvestor.com/partitioning-2d-spaces-an-introduction-to-quadtrees-d95728856613

Shared memory concurrent objects

- Quadtree/Octree

recursively divide the scene giving more detail to “interesting” areas



https://medium.datadriveninvestor.com/partitioning-2d-spaces-an-introduction-to-quadtrees-d95728856613

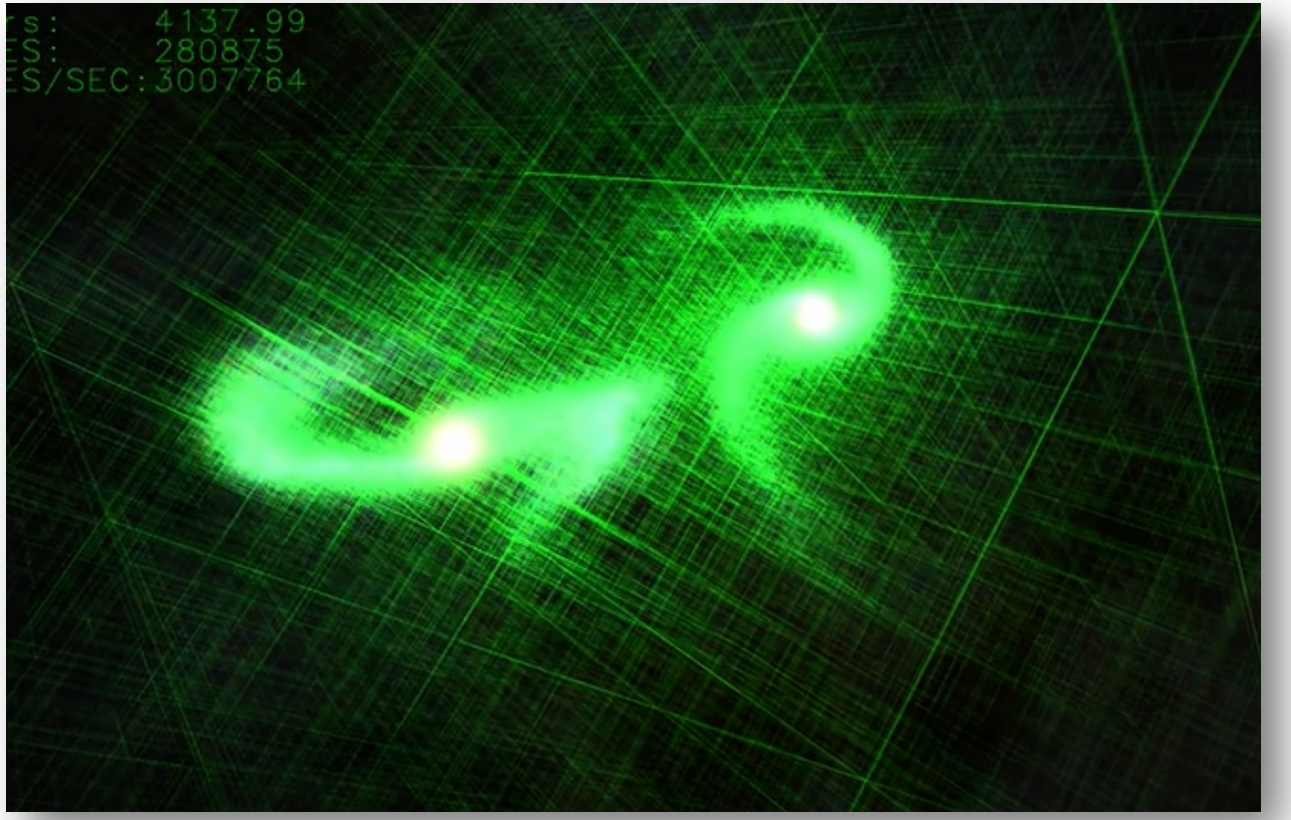
Octree example

- From GTC 2012
  - Simulation of 2 galaxies colliding
  - 280K stars

https://www.youtube.com/watch?v=aByz-mxOXJM

Octree example

- From GTC 2012
  - Simulation of 2 galaxies colliding
  - 280K stars



https://www.youtube.com/watch?v=aByz-mxOXJM

Take away

- Concurrent data structures are everywhere!
  - Infrastructure
  - Applications

Schedule

- Intro to concurrent data structures
- **Bank account example**
- Specification: Sequential consistency

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:

```
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;
}
```

We might decide to wrap my bank account in an object

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

    void buy_coffee() {
      balance -= 1;
    }

    void get_paid() {
      balance += 1;
    }

  private:
    int balance;
};
```

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();
}
```

We might decide to wrap my bank account in an object

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

    void buy_coffee() {
      balance -= 1;
    }

    void get_paid() {
      balance += 1;
    }

  private:
    int balance;
};
```

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?

Example

We might decide to wrap my bank account in an object

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

    void buy_coffee() {
      balance -= 1;
    }

    void get_paid() {
      balance += 1;
    }

  private:
    int balance;
};
```

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?

Example

C++ will not magically make your objects concurrent!

We might decide to wrap my bank account in an object

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

    void buy_coffee() {
      balance -= 1;
    }

    void get_paid() {
      balance += 1;
    }

  private:
    int balance;
};
```

The object is not "thread safe"

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();
}
```

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

We might decide to wrap my bank account in an object

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

    void buy_coffee() {
      balance -= 1;
    }

    void get_paid() {
      balance += 1;
    }

  private:
    int balance;
};
```

The object is not "thread safe"

global variables:

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

what if you have multiple objects?

Tyler's coffee addiction:

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

Tyler's employer

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

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

We might decide to wrap my bank account in an object

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

    void buy_coffee() {
      balance -= 1;
    }

    void get_paid() {
      balance += 1;
    }

  private:
    int balance;
};
```

The object is not "thread safe"

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();
}
```

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

client has to manage locks

We might decide to wrap my bank account in an object

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

    void buy_coffee() {
      balance -= 1;
    }

    void get_paid() {
      balance += 1;
    }

  private:
    int balance;
};
```

The object is not "thread safe"

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();
}
```

we can encapsulate a mutex in the object.

The API stays the same!

```
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;
};
```

Thread safe objects

- An object is thread-safe if you can call it concurrently
- Otherwise you must provide your own locks!

Lock free programming

- An object is “lock free” if it is:
  - thread safe
  - does not use a lock in its underlying implementation.

- We can make a lock free bank account

```
atomic_fetch_add(atomic_int * addr, int value) {
   int tmp = *addr; // read
   tmp += value;    // modify
   *addr = tmp;     // write
}
```

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;
};
```

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:
    atomic_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() {

      balance -= 1;

    }

    void get_paid() {

      balance += 1;

    }

  private:
    atomic_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();
}
```

```
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;
};
```

How does it perform

How does it perform

- Noticeably better!
  - Mutexes reduce parallelism
  - Mutexes require many RMW operations
- Straight forward to do with the bank account, we will apply this to more objects
  - This performance matters in frameworks!

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

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: *Error 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:

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

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

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

t is 6

Global variable:

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

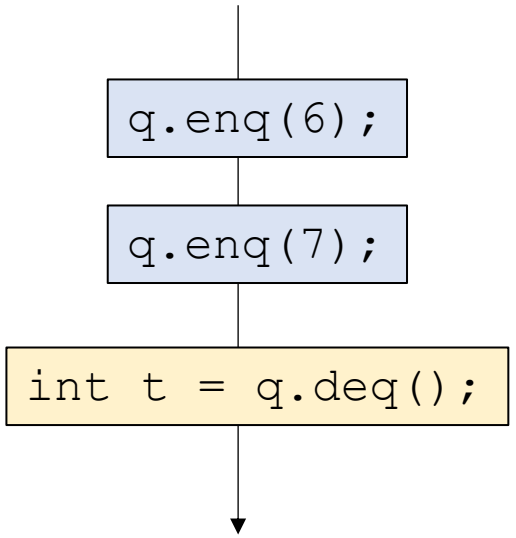
Thread 0:

```
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

Global variable:

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

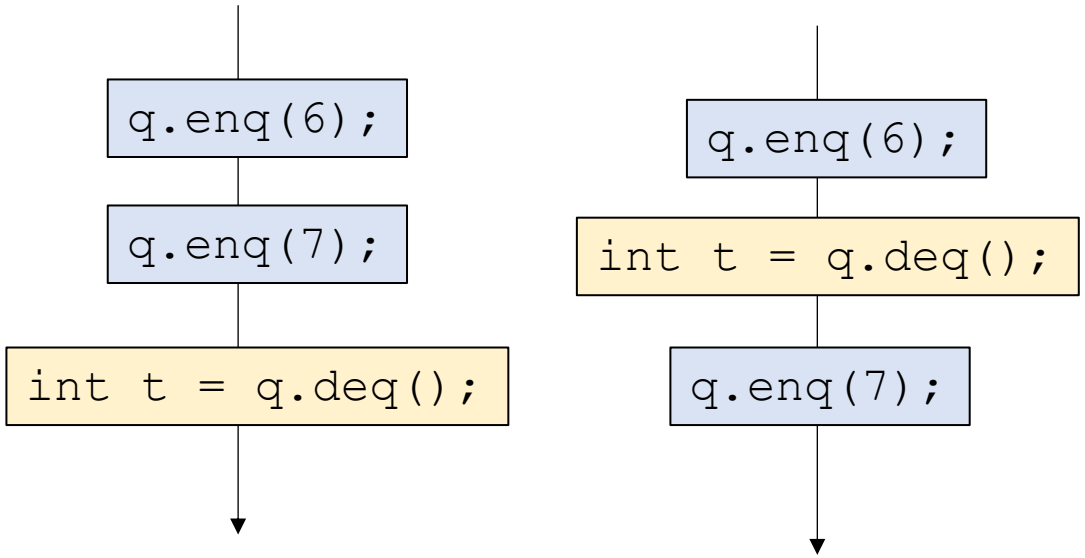
Thread 0:

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

Thread 1:

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

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



Global variable:

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

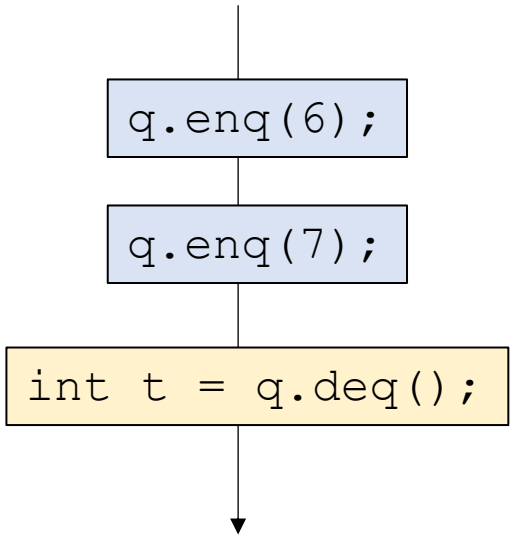
Thread 0:

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

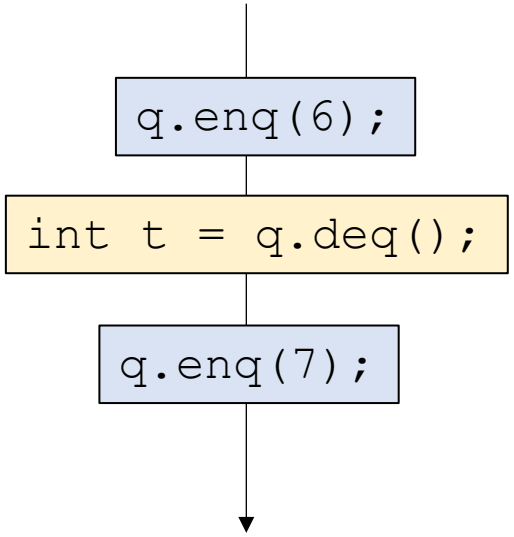
Thread 1:

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

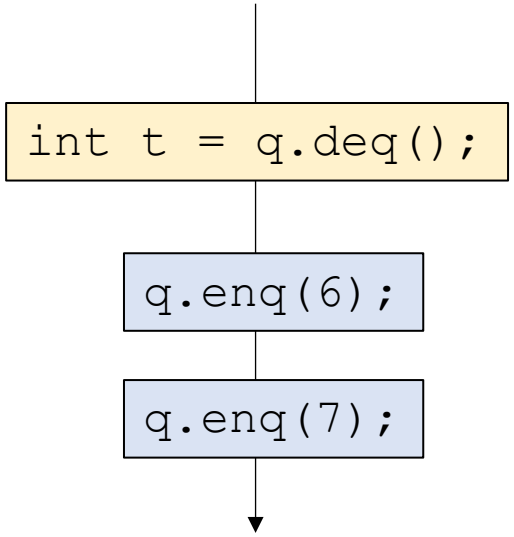
Construct a sequential timeline of API calls
Any sequence is valid:



t is 6



t is 6



t is None

Global variable:

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

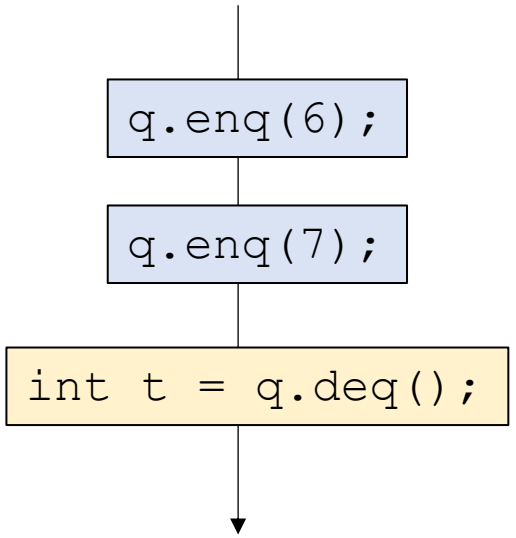
Thread 0:

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

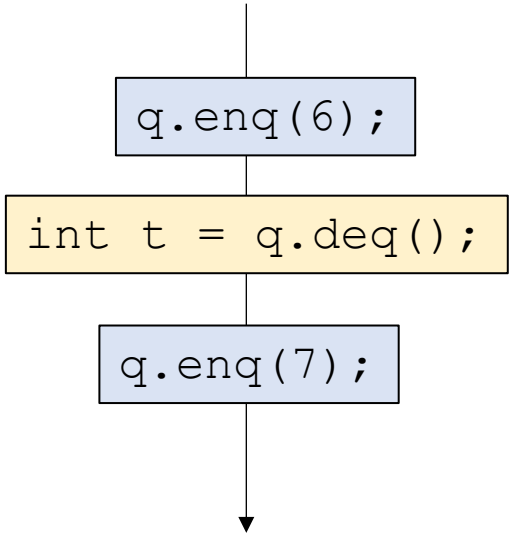
Thread 1:

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

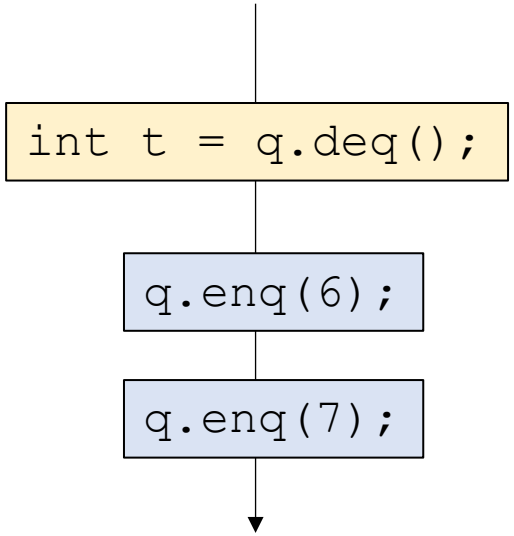
Construct a sequential timeline of API calls
Any sequence is valid:



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;
```

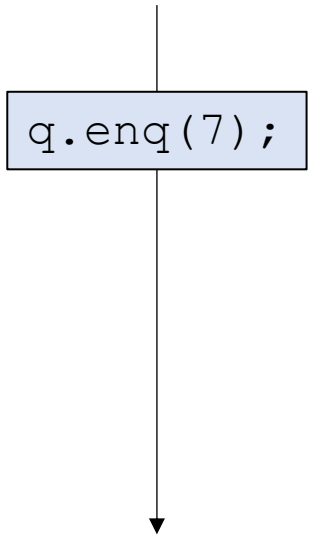
Thread 0:

```
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;
```

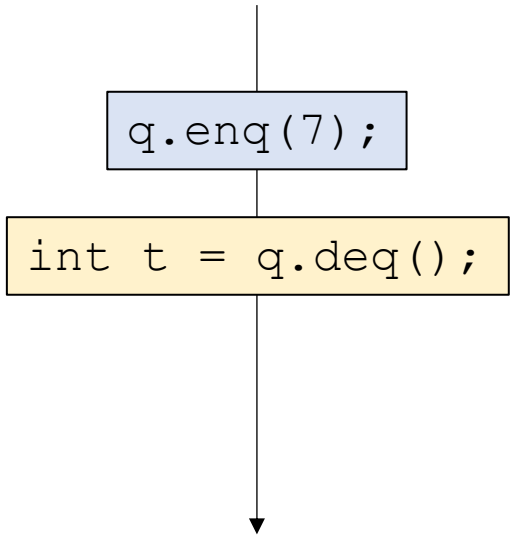
Thread 0:

```
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;
```

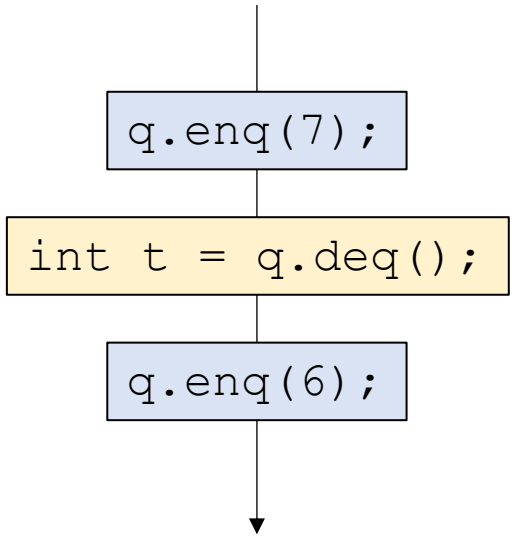
Thread 0:

```
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;
```

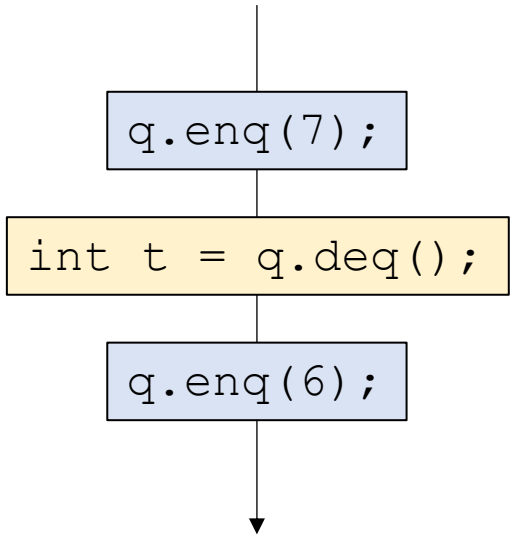
Thread 0:

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

Thread 1:

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

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



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

How many possible interleavings?
Combinatorics question:

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

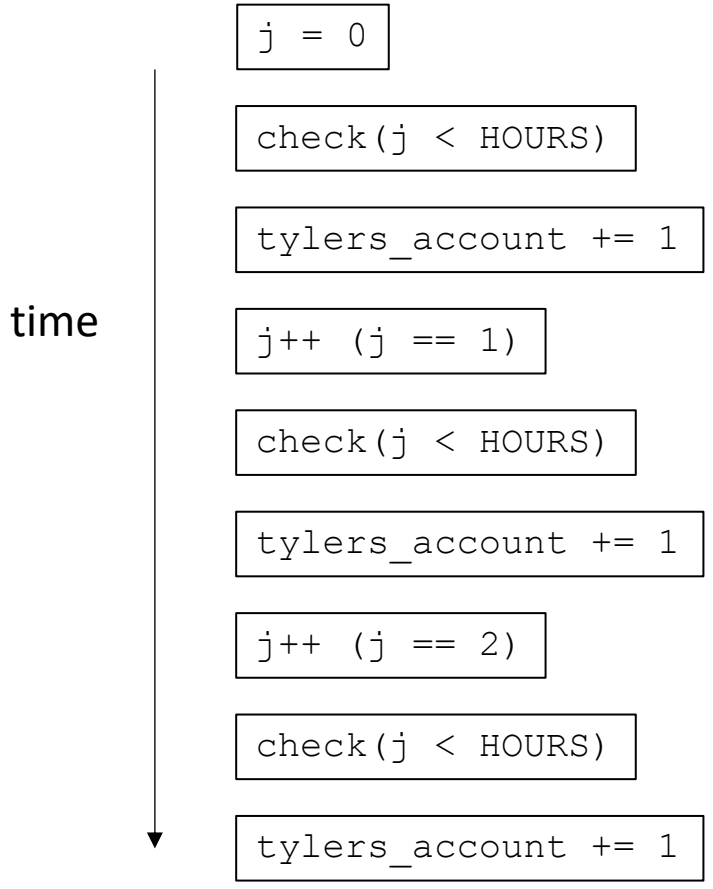
$$\frac{(N+M)!}{N!\,M!}$$

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!

Tyler's employer

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



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:

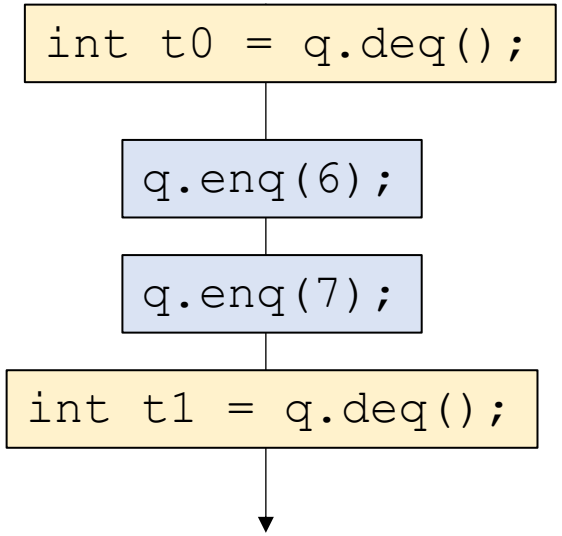
```
CQueue<int> q;
```

Thread 0:

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

Thread 1:

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



Can t0 == 0 and t1 == 6?

Valid execution!

Are there others?

Global variable:

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

Thread 0:

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

Lets do another!

Thread 1:

```
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);
```

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

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

Lets do another!

Thread 1:

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

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

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

Can t0 == 6 and t1 == 7?

Global variable:

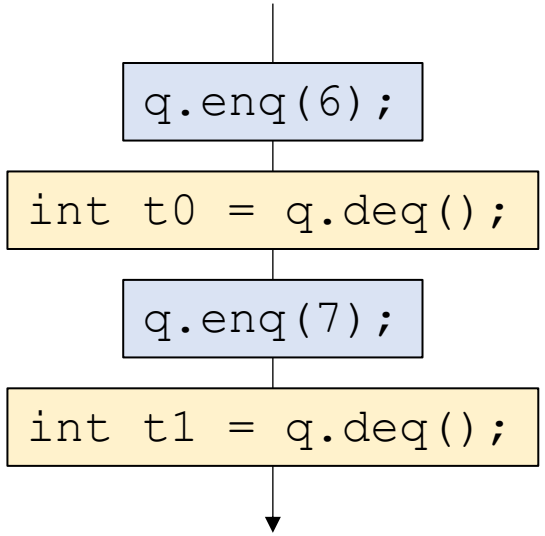
```
CQueue<int> q;
```

Thread 0:

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

Thread 1:

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



Can t0 == 6 and t1 == 7?

Found one! Are there others?

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();
```



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

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

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

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

Can t0 == 0 and t1 == 7?

Global variable:

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

Thread 0:

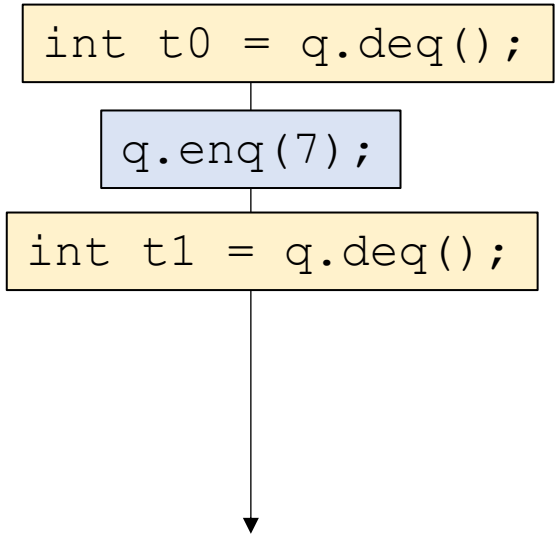
```
q.enq(6);
q.enq(7);
```

Thread 1:

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

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

No place for this event to go!



Can t0 == 0 and t1 == 7?

One more example

Global variable:

```
CStack<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?

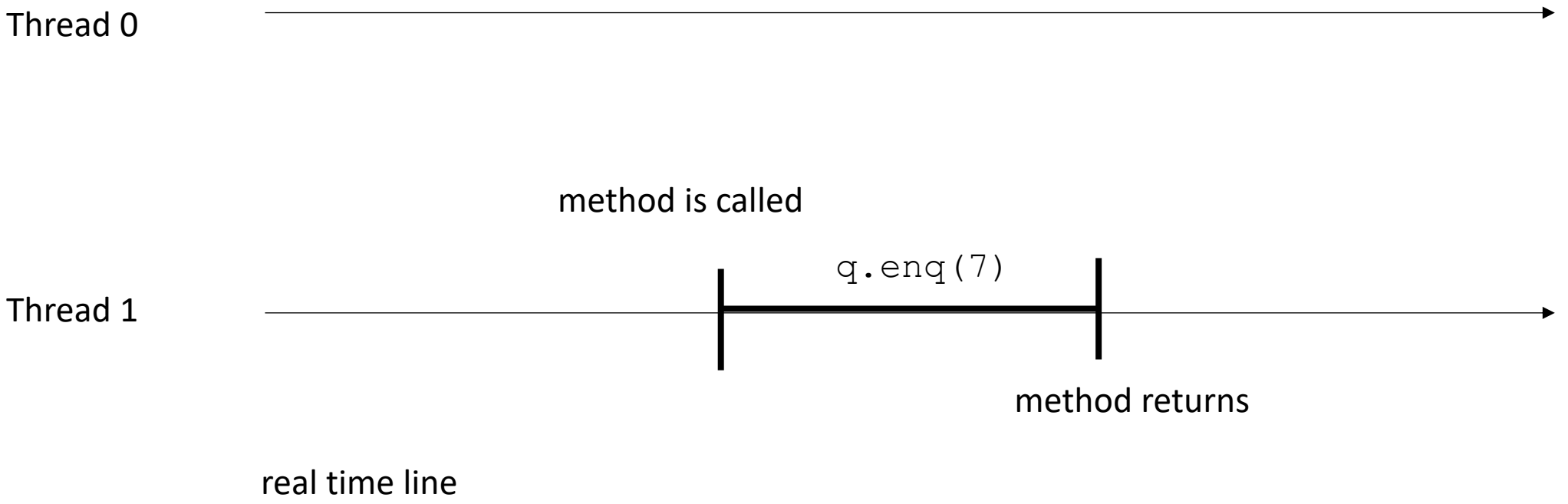
Do we have our specification?

- 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.

Sequential consistency and real time

- Add in real time:

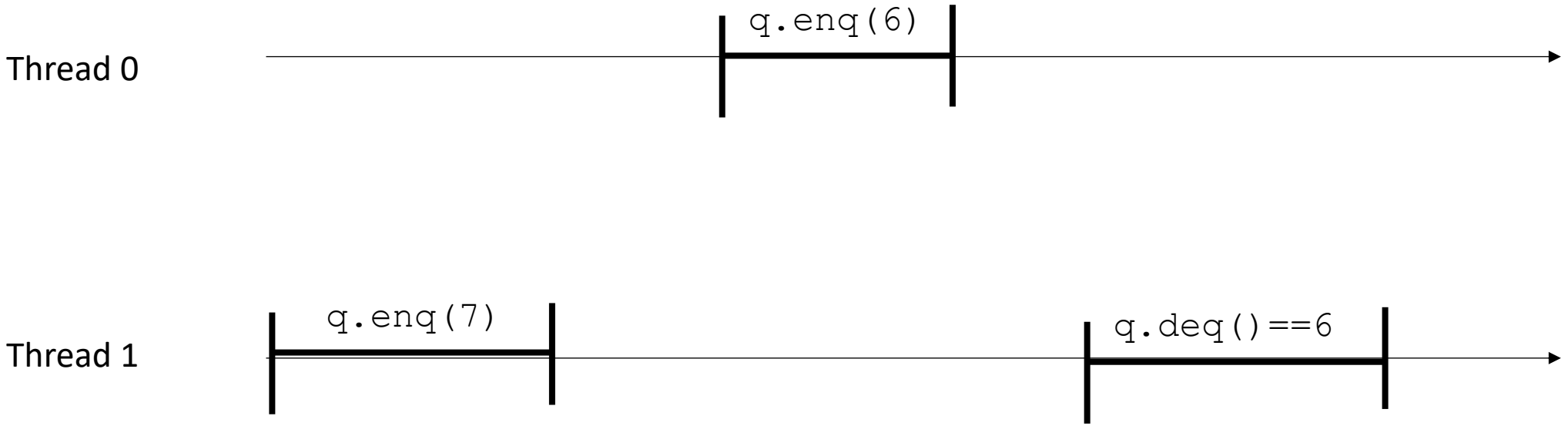
each method as a start, and end time stamp



Sequential consistency and real time

- Add in real time:

This timeline seems strange...

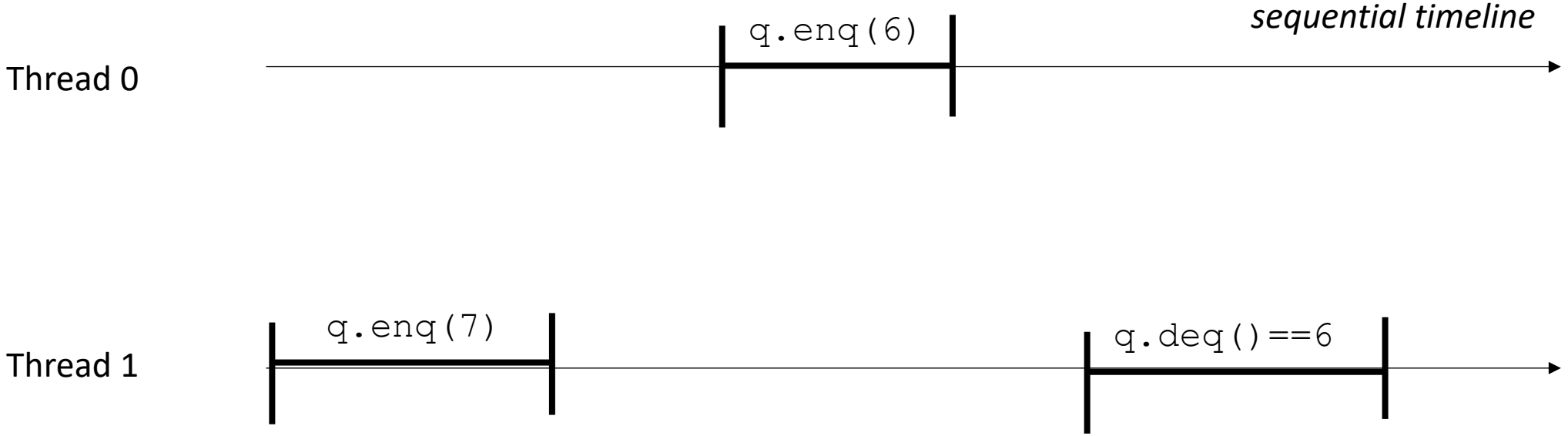


Sequential consistency and real time

- 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

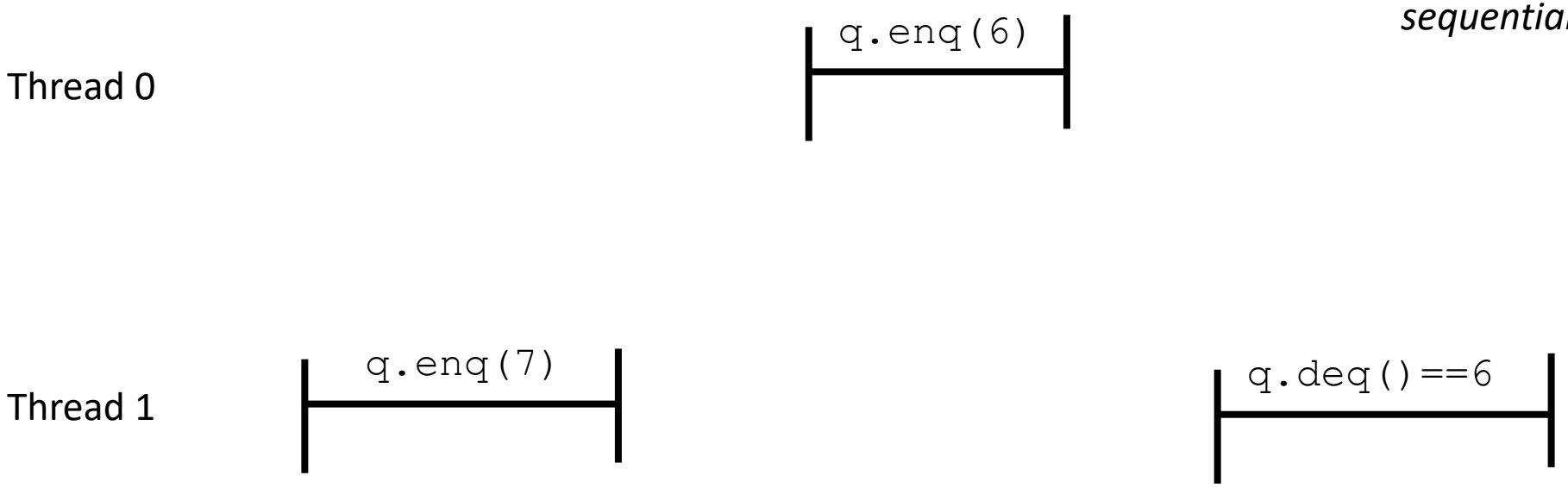


Sequential consistency and real time

- 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

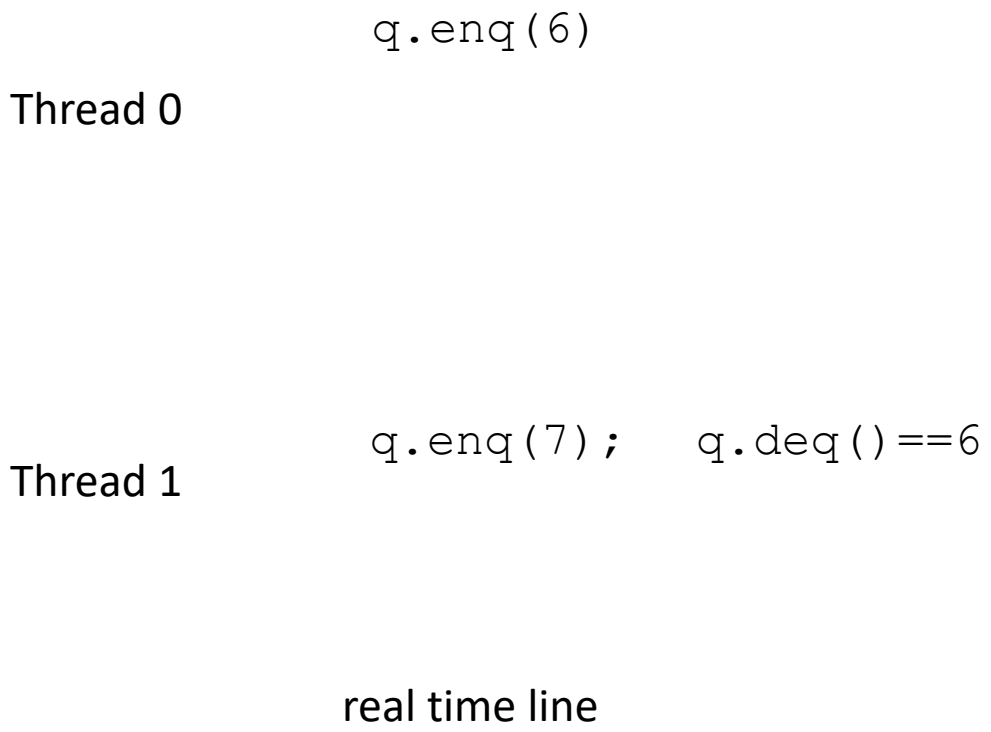


Sequential consistency and real time

- 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



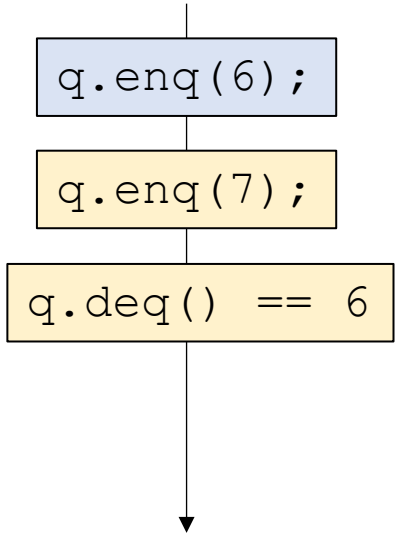
Sequential consistency and real time

- 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!

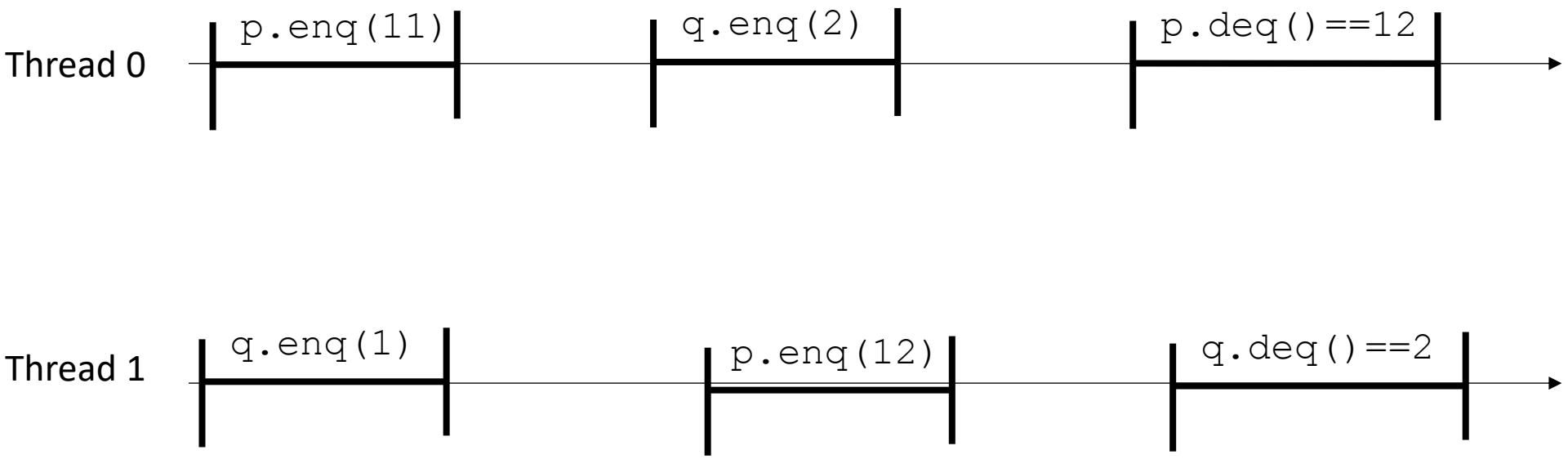
SC doesn't care about real time, only if it can construct its virtual sequential timeline



Sequential consistency and real time

- Add in real time:

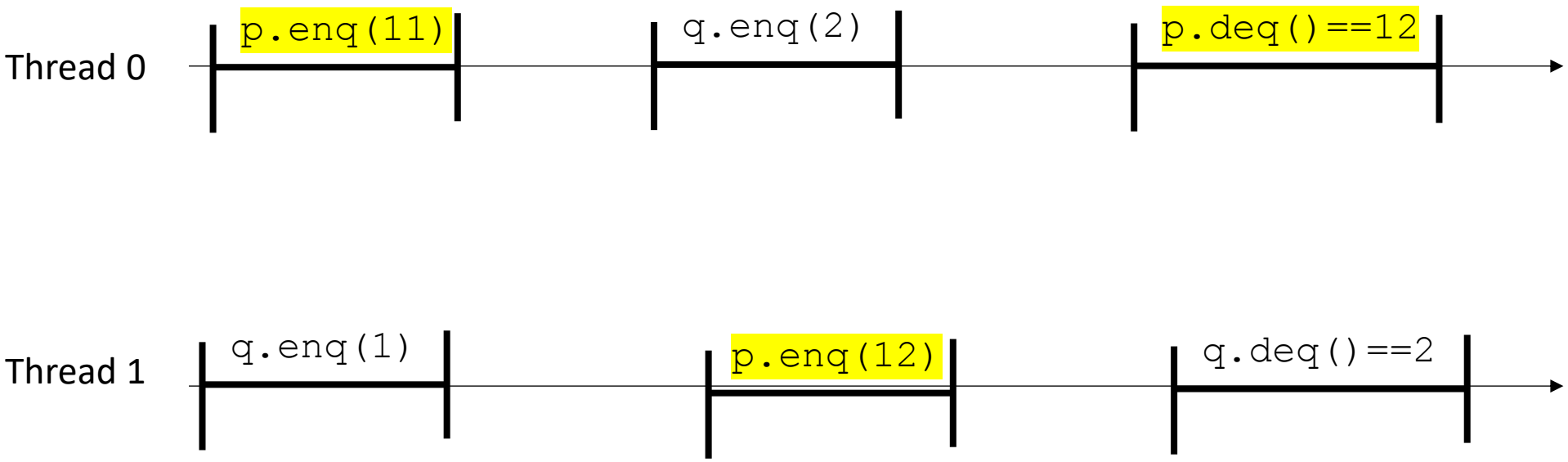
2 objects now: p and q



Sequential consistency and real time

- Add in real time:

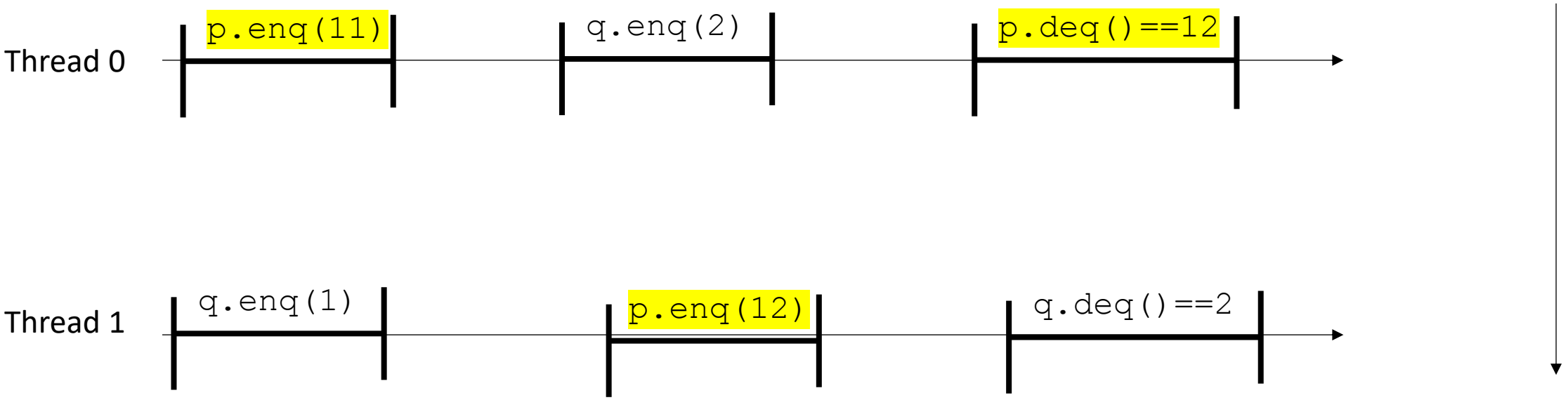
2 objects now: p and q
Consider each object in isolation



Sequential consistency and real time

- Add in real time:

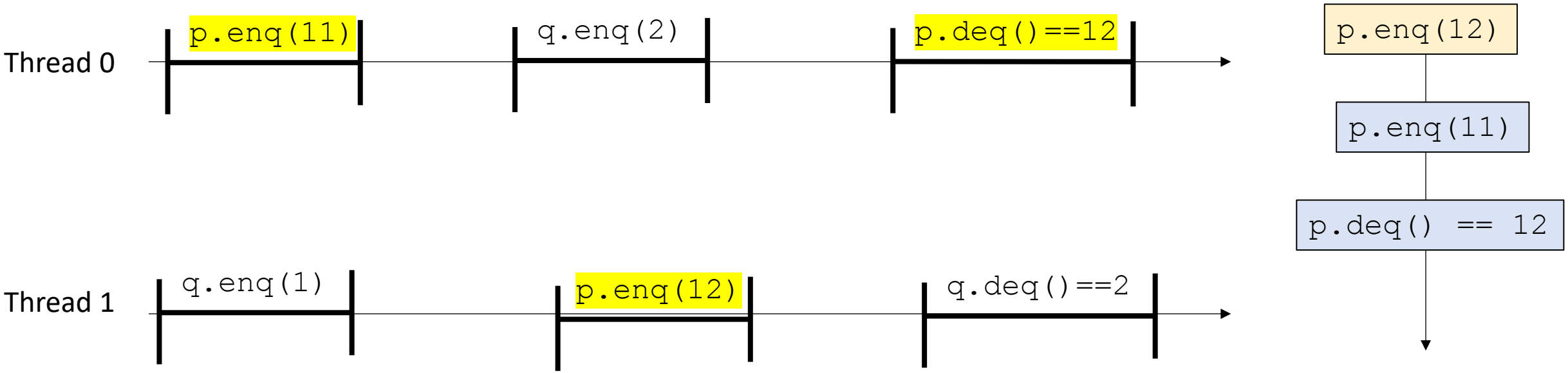
2 objects now: p and q
Consider each object in isolation



Sequential consistency and real time

- Add in real time:

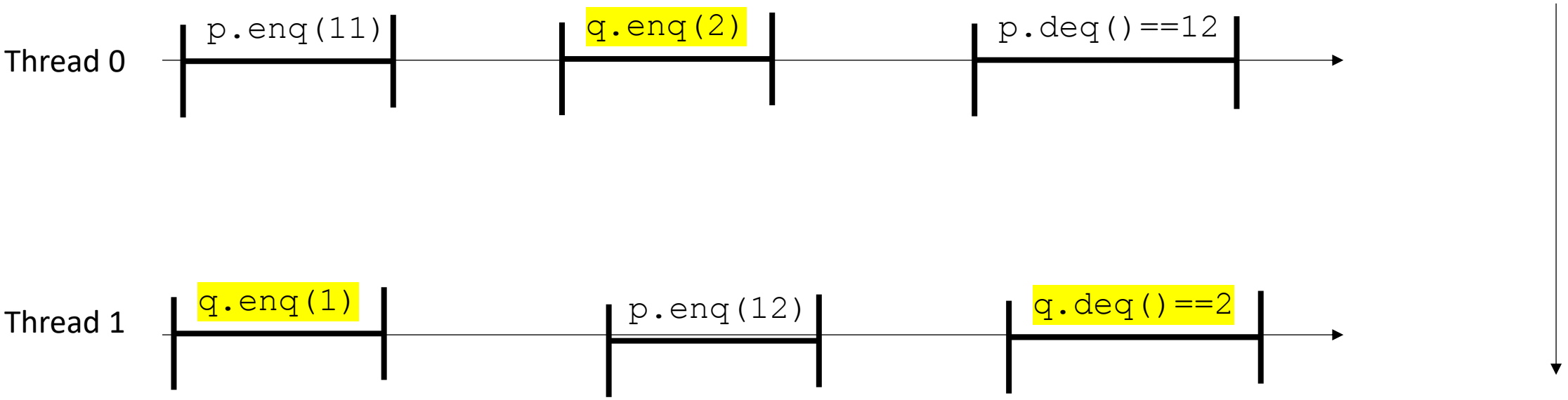
2 objects now: p and q
Consider each object in isolation



Sequential consistency and real time

- Add in real time:

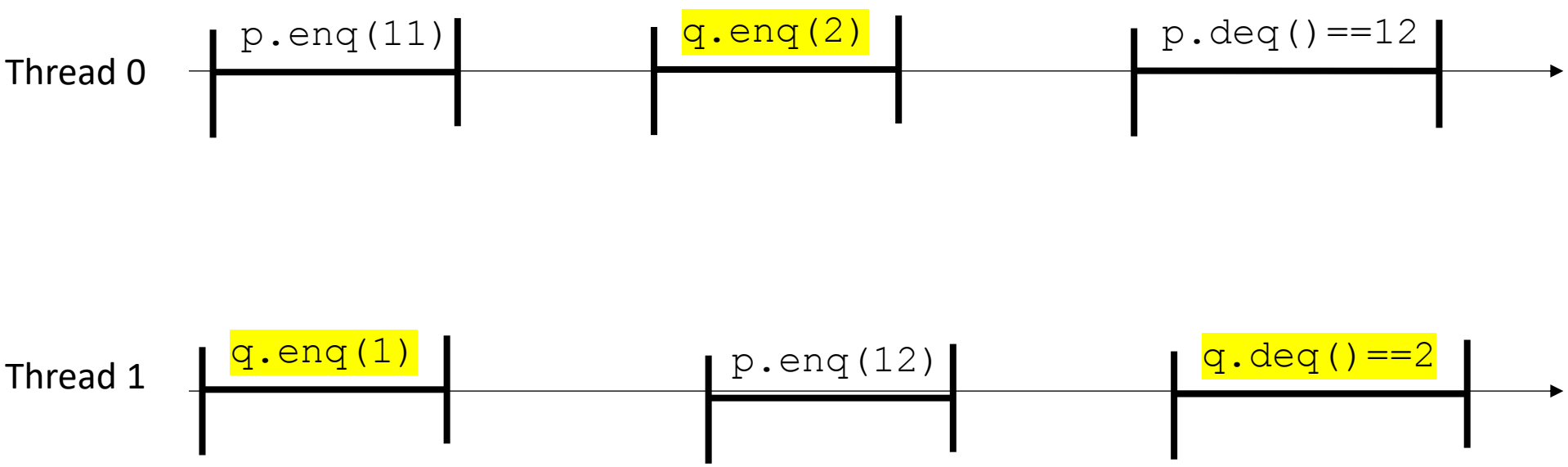
2 objects now: p and q
Consider each object in isolation

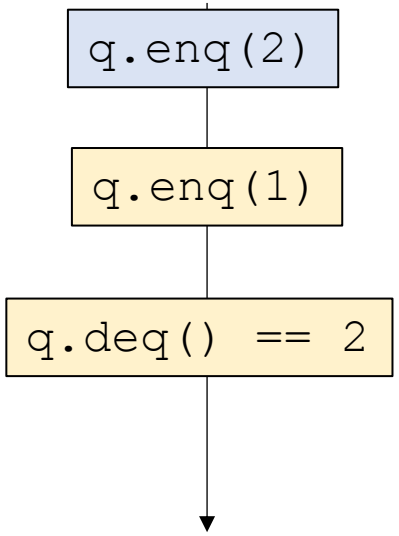


Sequential consistency and real time

- Add in real time:

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

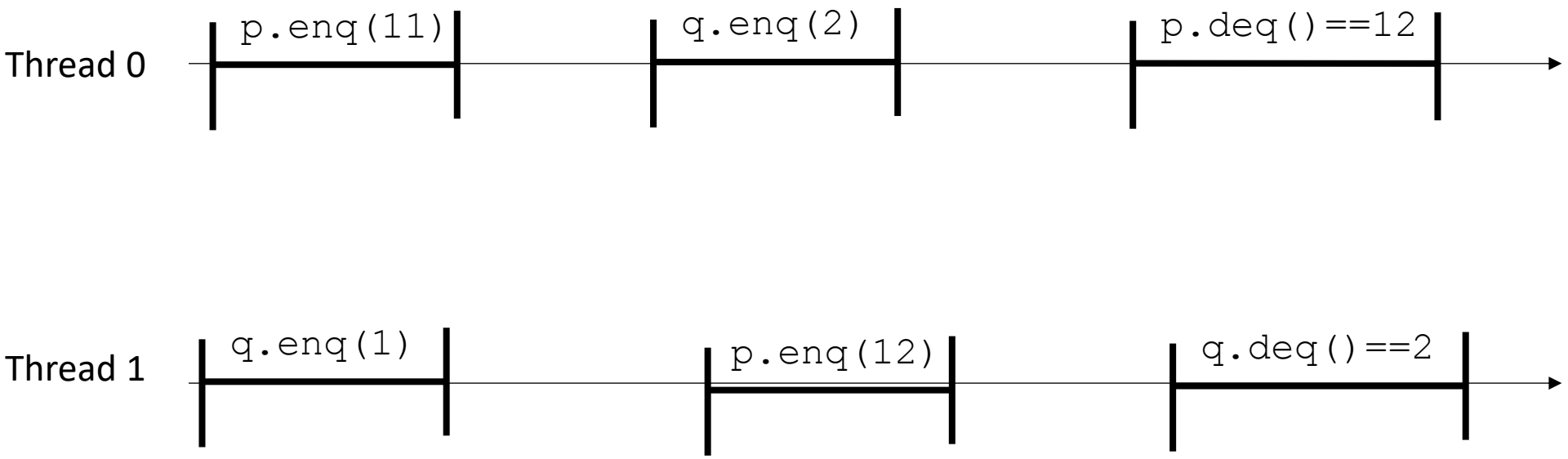




Sequential consistency and real time

- 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:

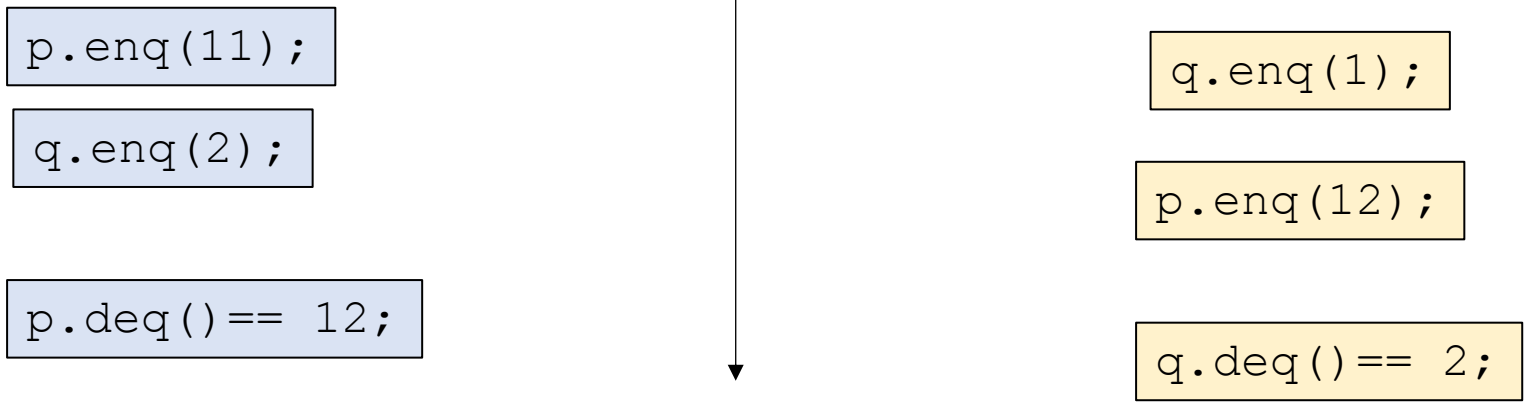
```
CQueue<int> p,q;
```

Thread 0:

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

Thread 1:

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

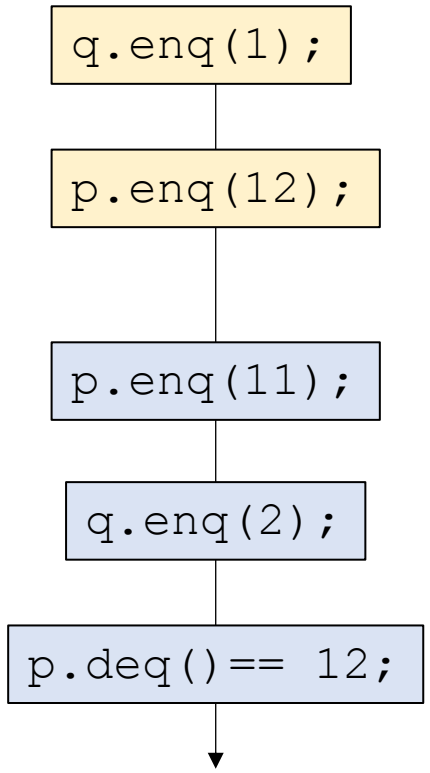


Global variable:

```
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!

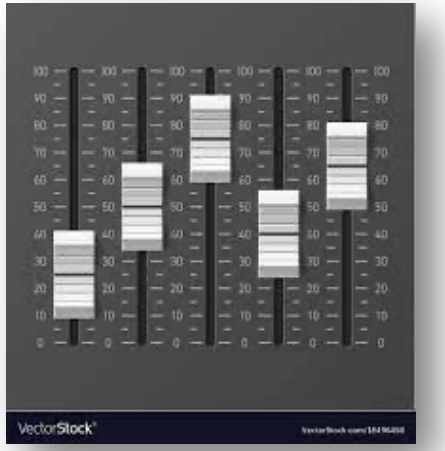
What does this mean?

- 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!

Linearizability

- Linearizability
  - Defined in term of real-time histories
  - We want to ask if an execution is allowed under linearizability

- Slightly different game:
  - sequential consistency is a game about stacking lego bricks
  - linearizability is about sliders



Linearizability

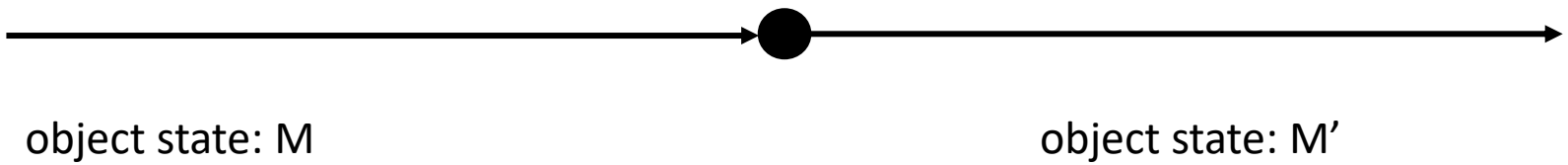
each operation has a linearizability point

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

Linearizability

each operation has a linearizability point

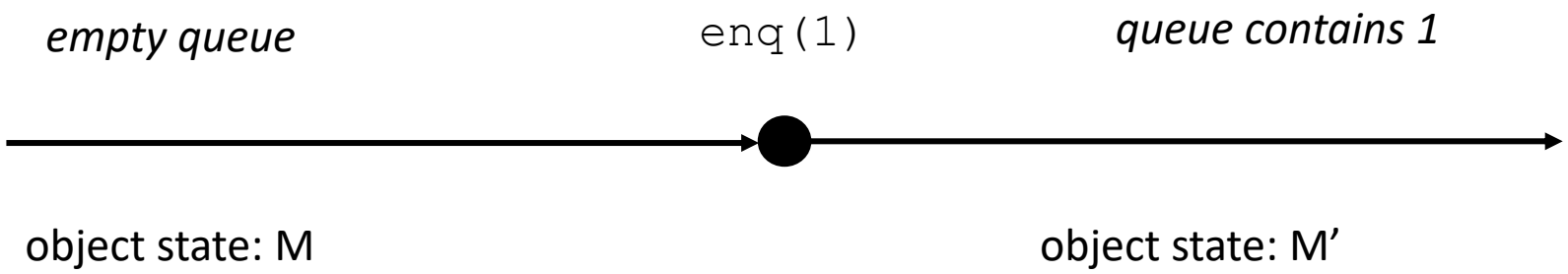
- does not overlap with other with other linearizability points
- indivisible computation (critical section, atomic RMW, atomic load, atomic store)
- object update (or read) occurs exactly at this point



Linearizability

each operation has a linearizability point

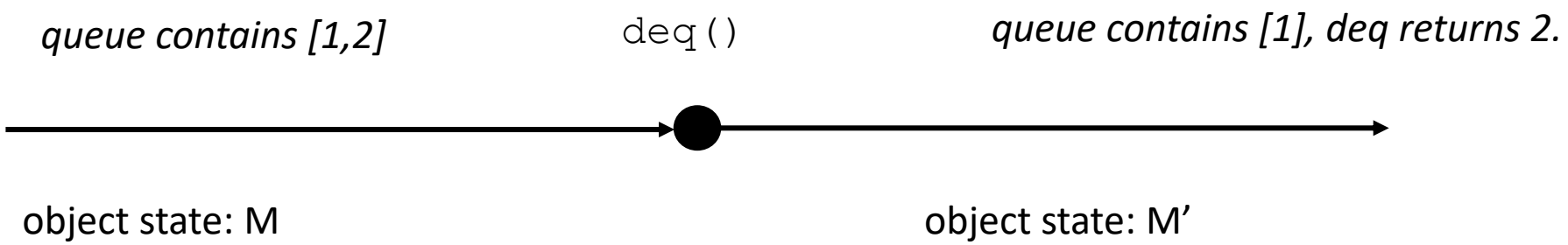
- does not overlap with other with other linearizability points
- indivisible computation (critical section, atomic RMW, atomic load, atomic store)
- object update (or read) occurs exactly at this point



Linearizability

each operation has a linearizability point

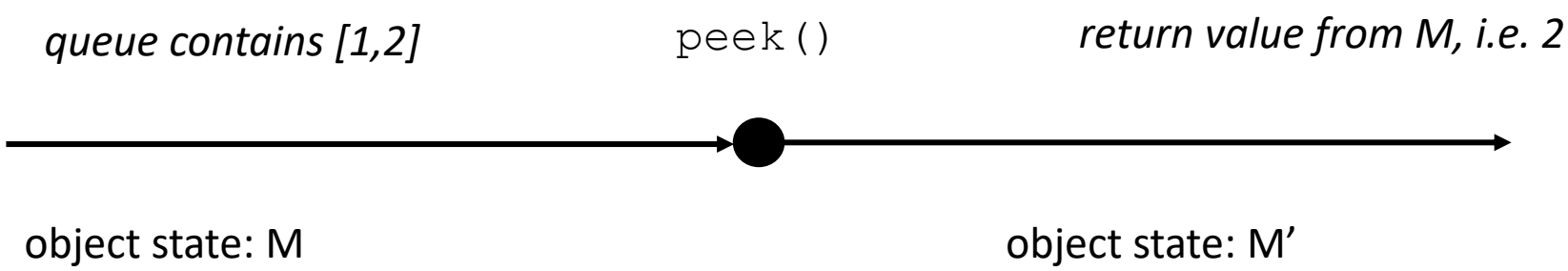
- does not overlap with other with other linearizability points
- indivisible computation (critical section, atomic RMW, atomic load, atomic store)
- object update (or read) occurs exactly at this point



Linearizability

each operation has a linearizability point

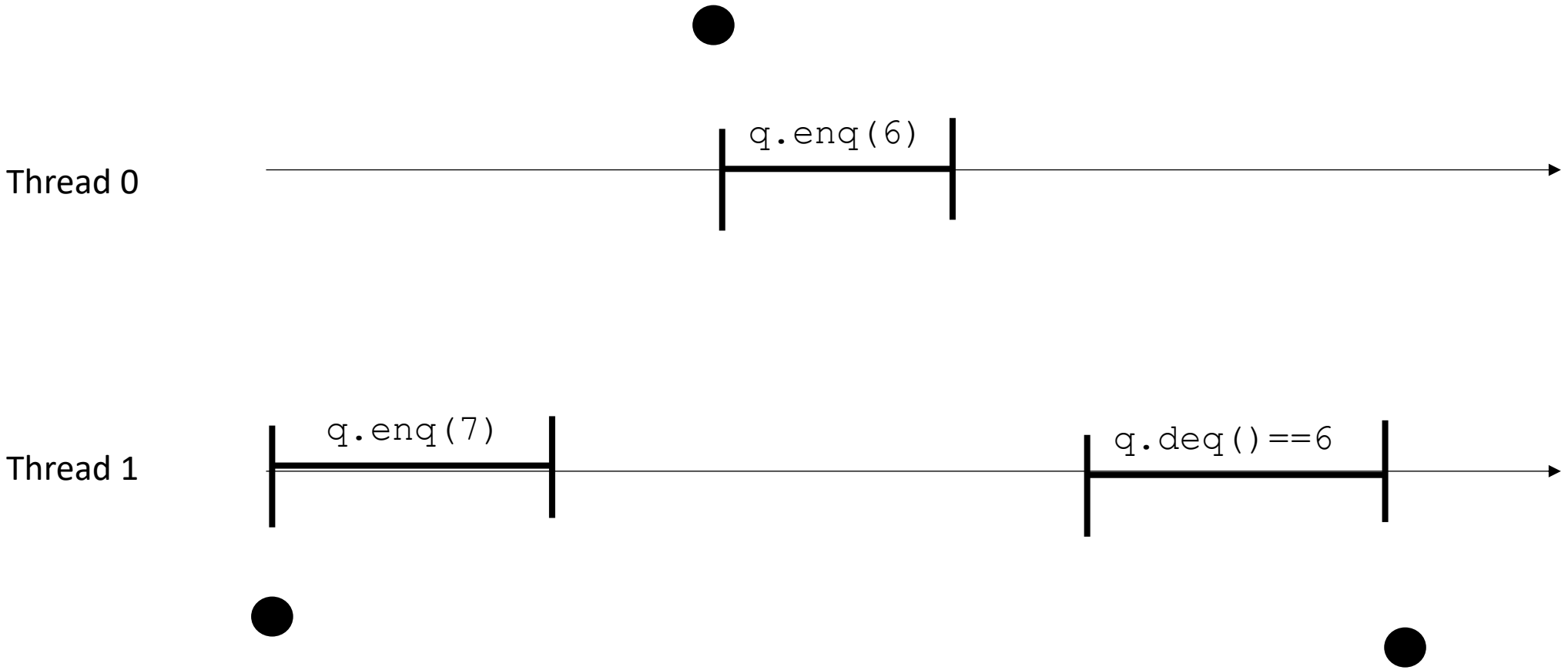
- does not overlap with other with other linearizability points
- indivisible computation (critical section, atomic RMW, atomic load, atomic store)
- object update (or read) occurs exactly at this point



Linearizability

each command gets a linearization point.

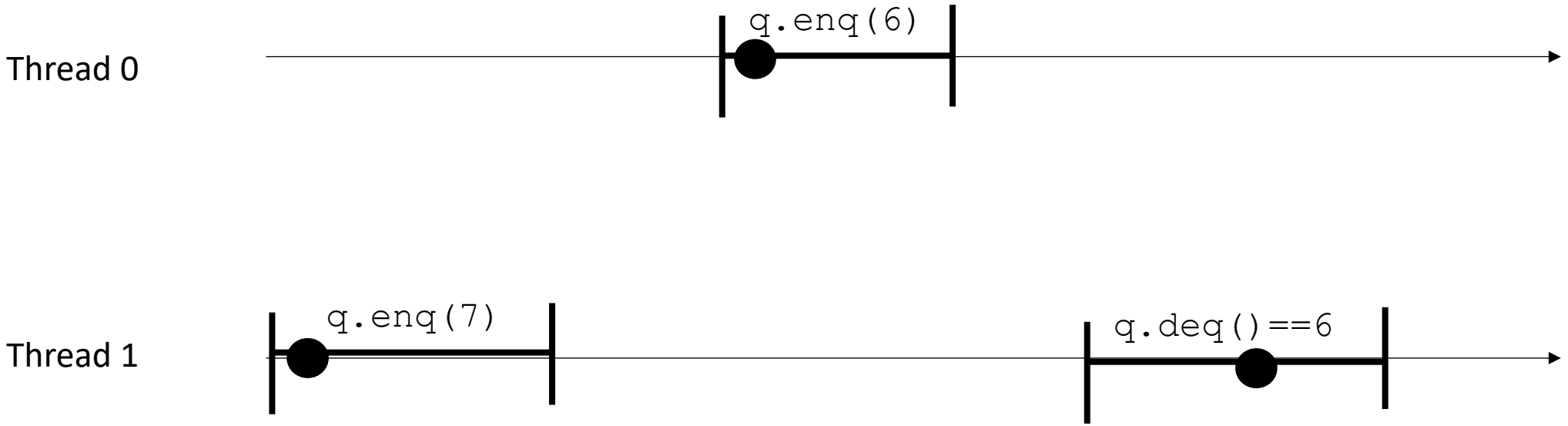
You can place the point any where between its innovation and response!



Linearizability

each command gets a linearization point.

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

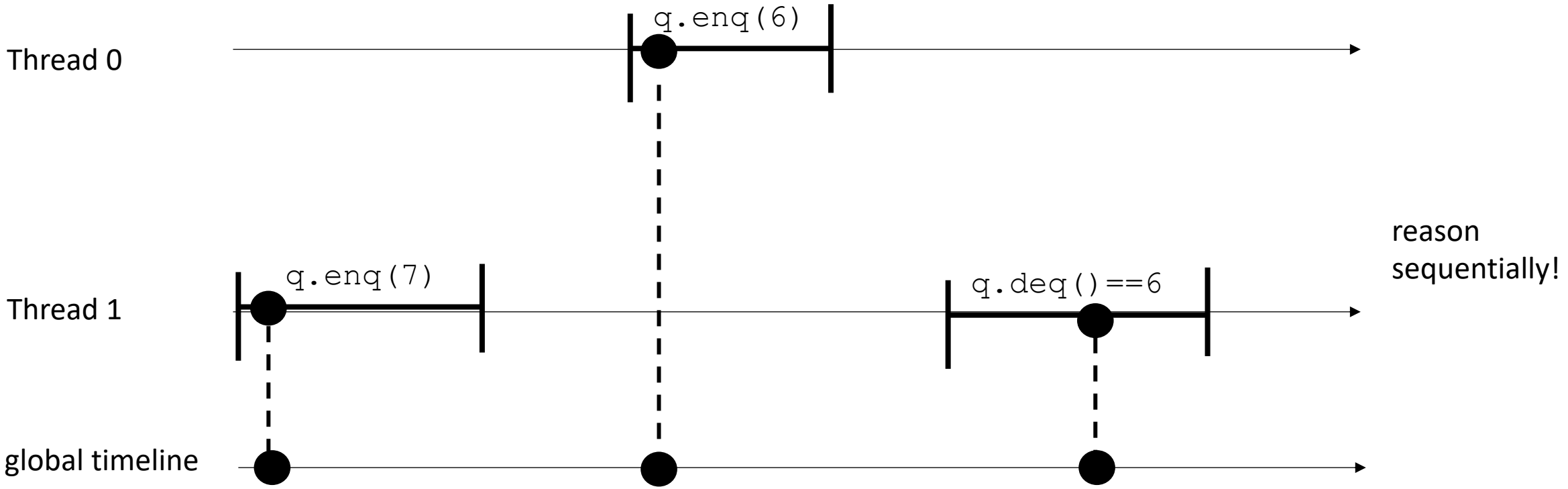


Linearizability

each command gets a linearization point.

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

Project the linearization points to a global timeline



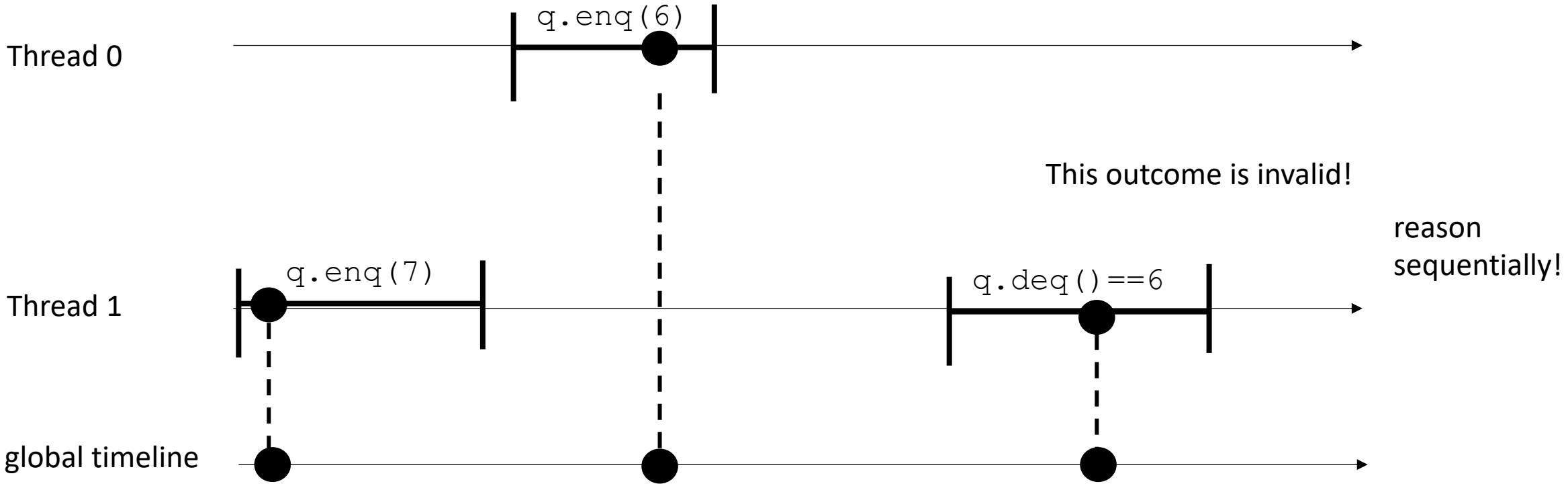
reason sequentially!

Linearizability

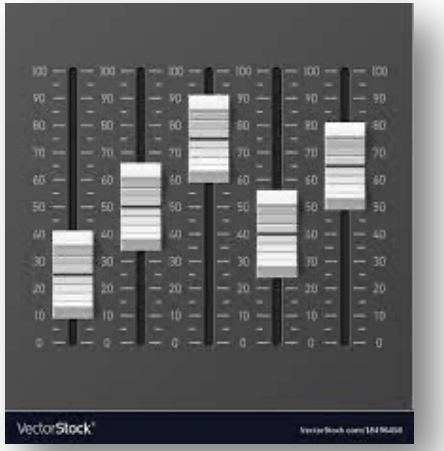
each command gets a linearization point.

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

Project the linearization points to a global timeline



Linearizability

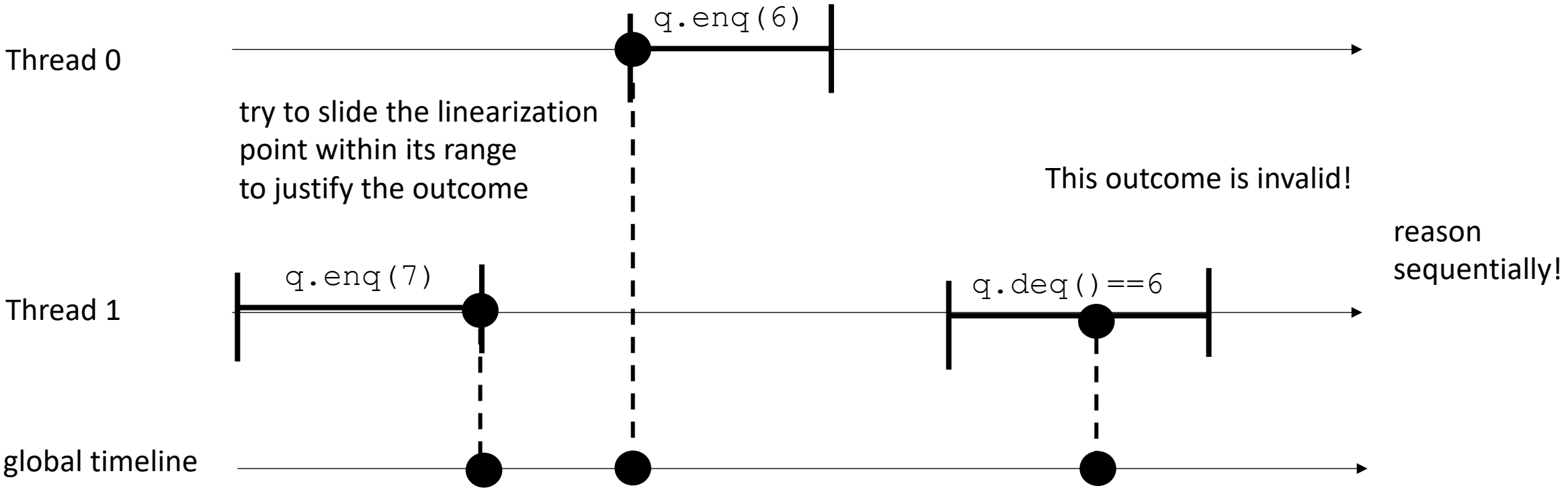


each command gets a linearization point.

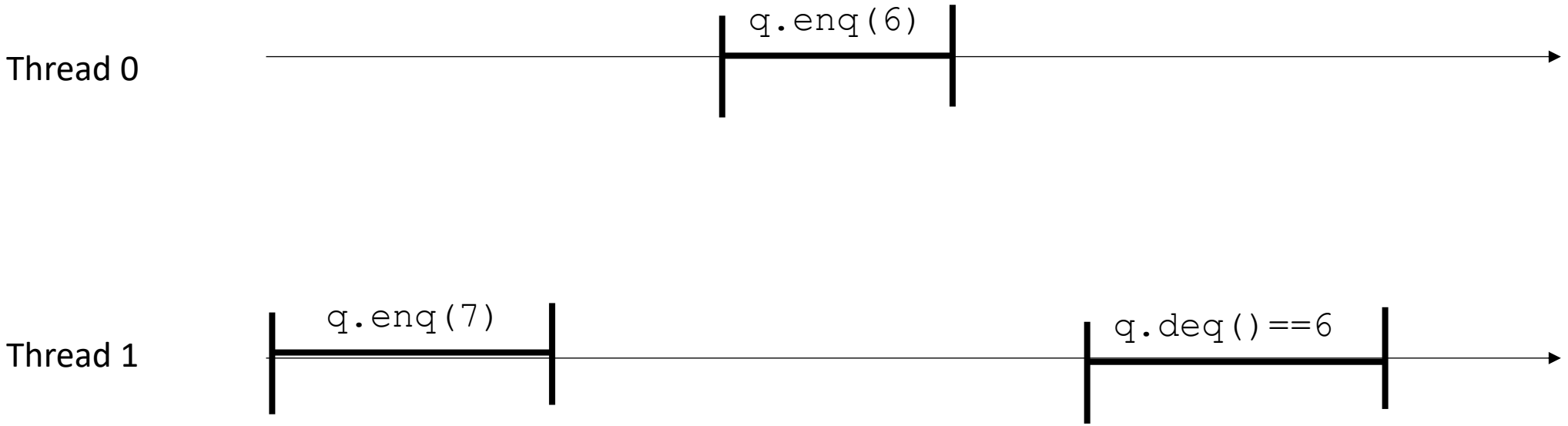
You can place the point any where between its innovation and response!

Project the linearization points to a global timeline

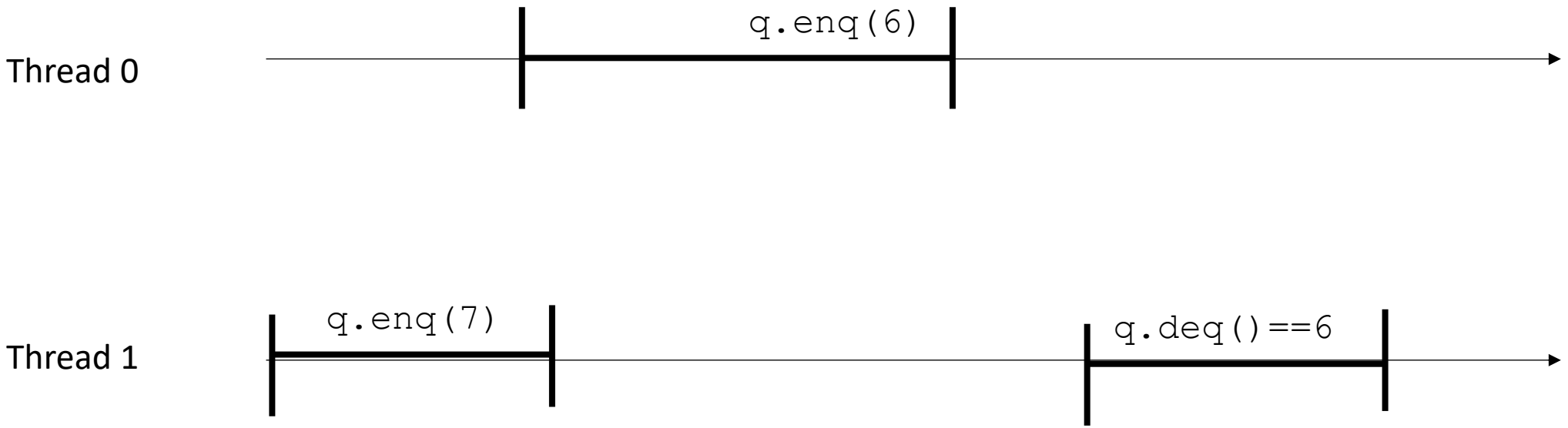
slider game!



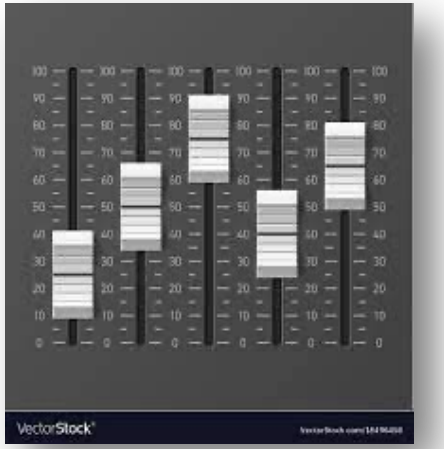
Linearizability

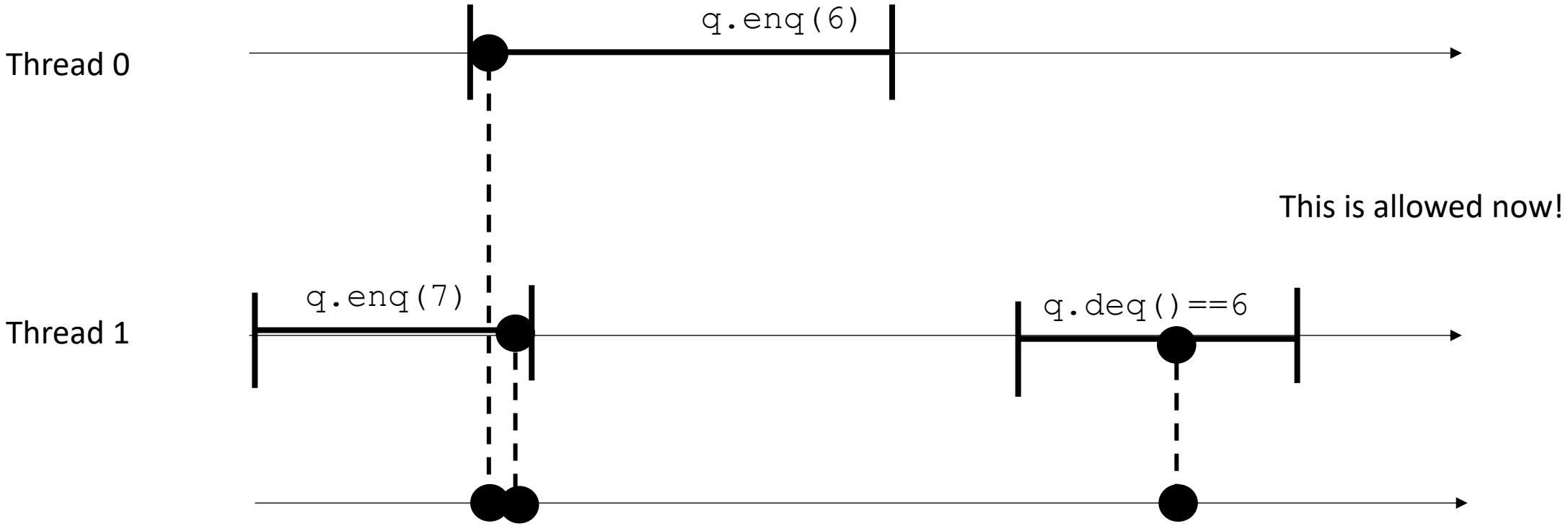


Linearizability

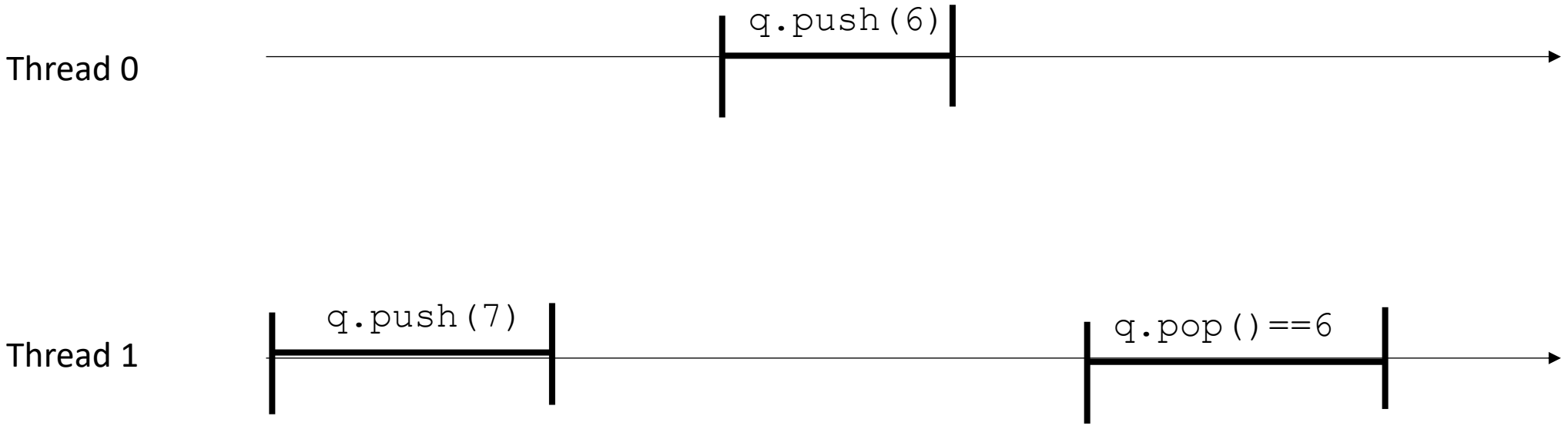


Linearizability

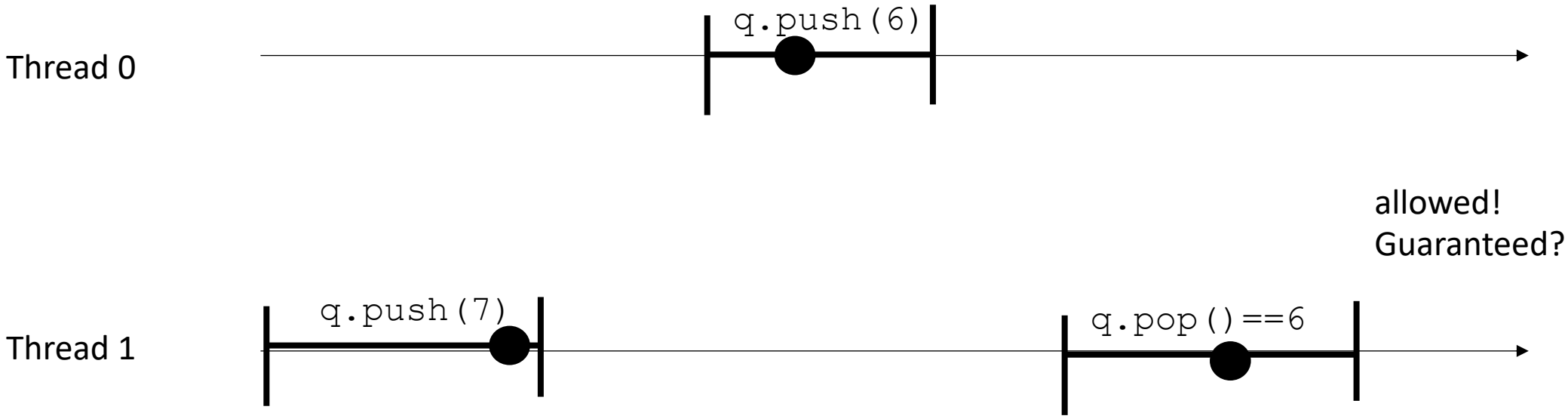




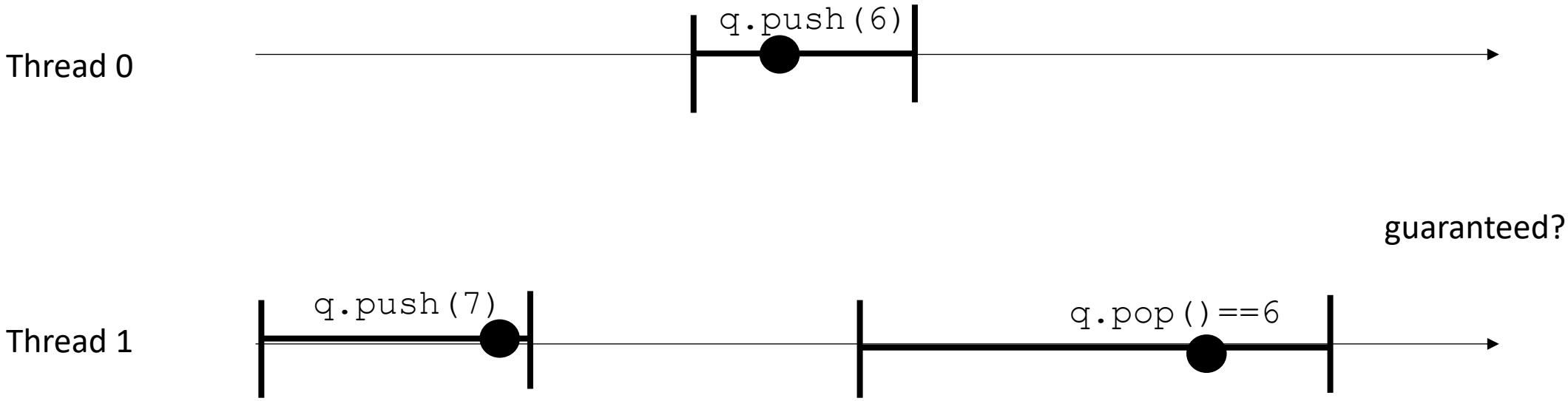
Linearizability



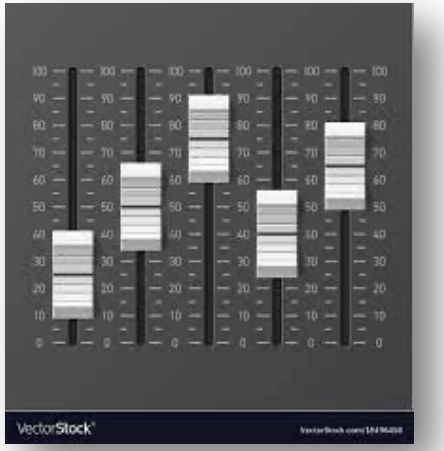
Linearizability

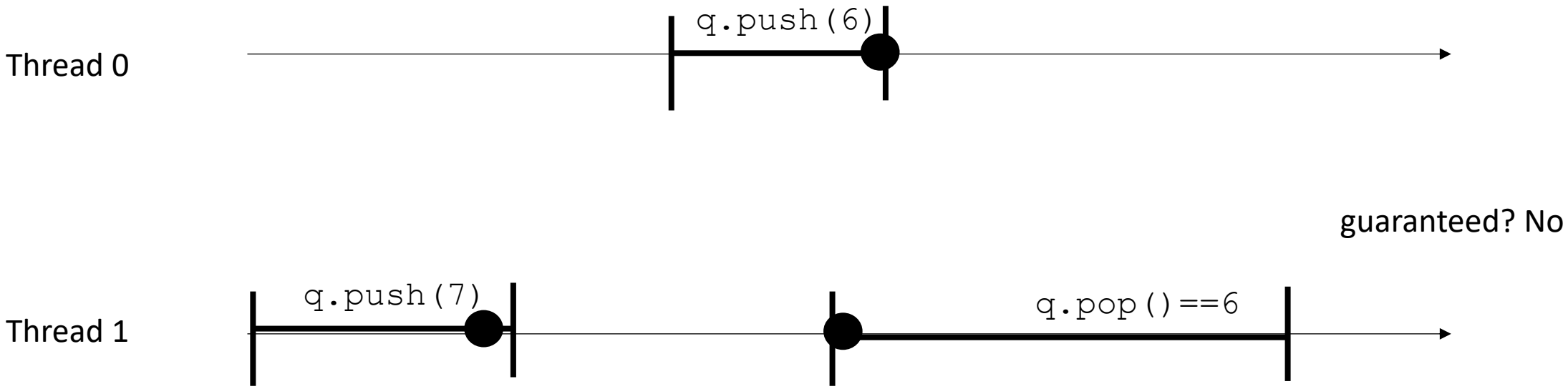


Linearizability



Linearizability





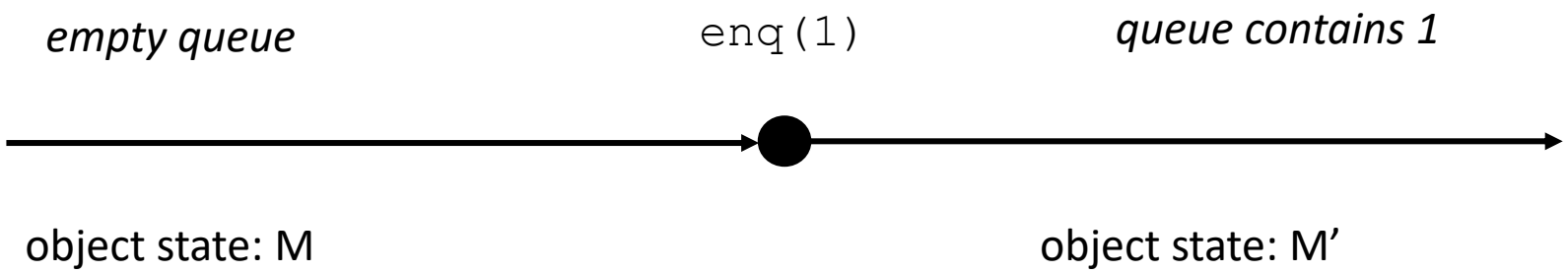
guaranteed? No

Linearizability

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

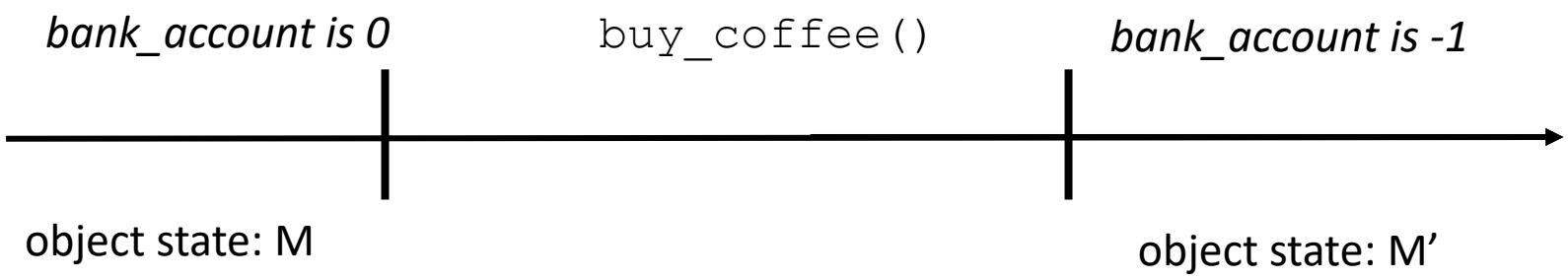
Linearizability

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



Linearizability

- Locked data structures are linearizable.

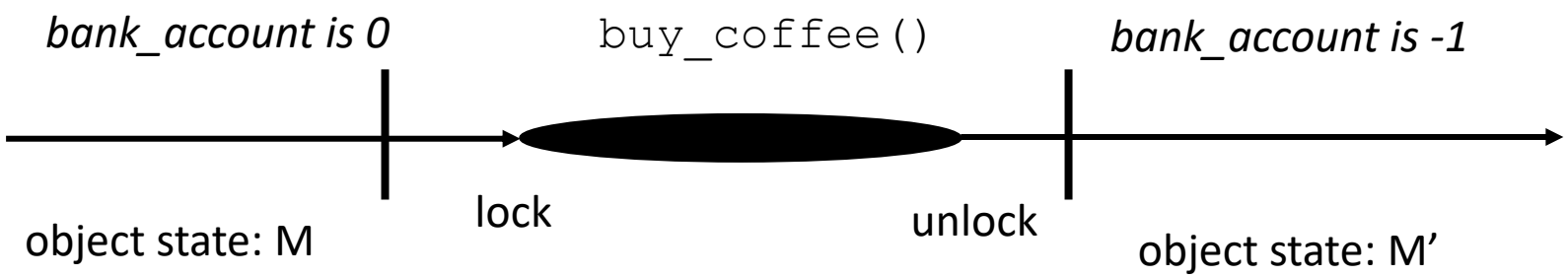



```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

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

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

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

Linearizability

- Locked data structures are linearizable.

typically modeled as the point the lock is acquired or released

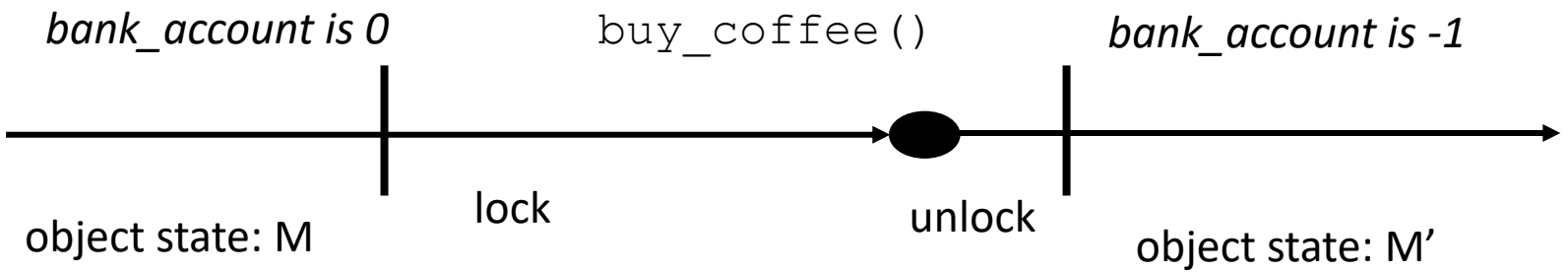



```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

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

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

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

Linearizability

- Locked data structures are linearizable.

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

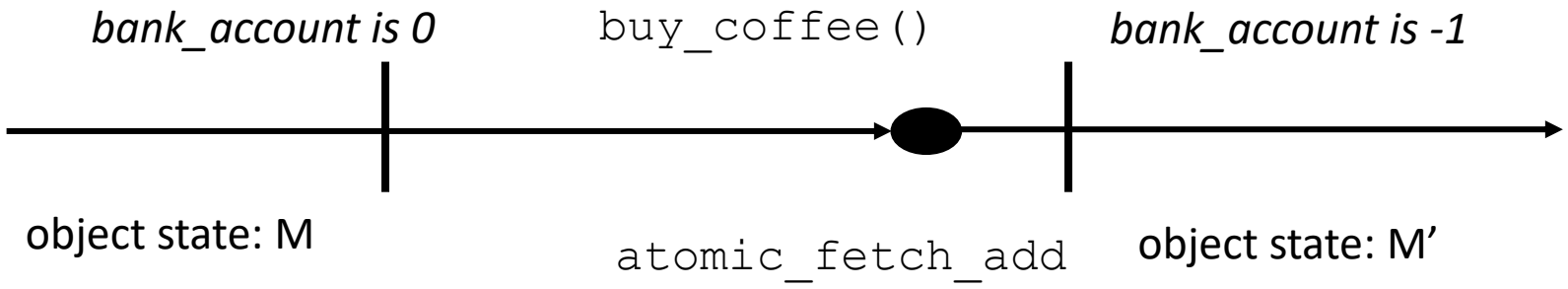



```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

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

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

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

Linearizability

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

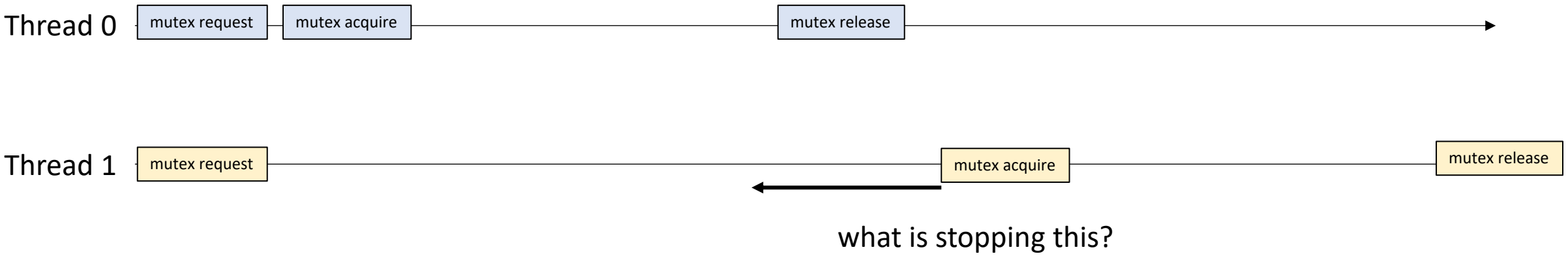
```
class bank_account {
  public:
    bank_account() {
      balance = 0;
    }

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

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

  private:
    atomic_int balance;
};
```



Progress properties

- Going back to specifications:

Recall the mutex

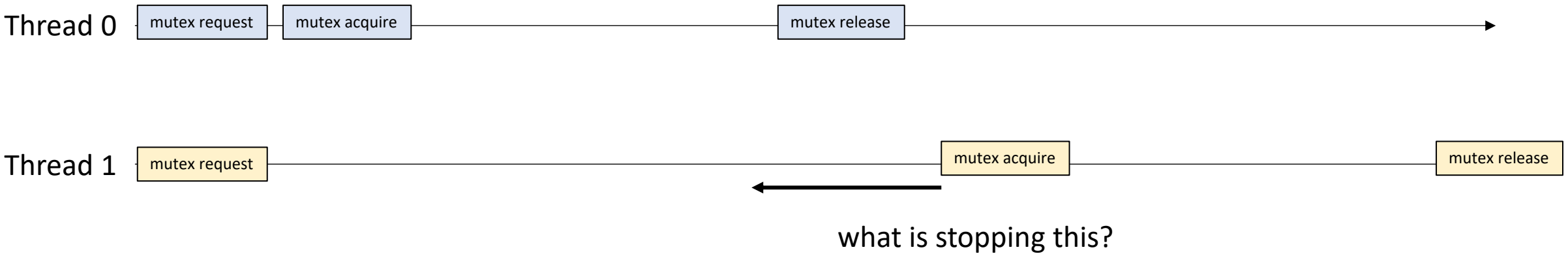


Progress properties

- Going back to specifications:

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

Recall the mutex

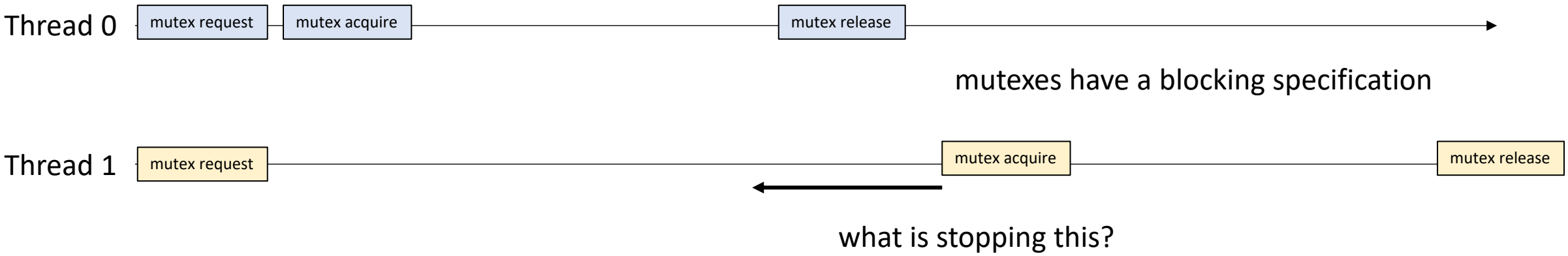


Progress properties

- Going back to specifications:

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

Recall the mutex

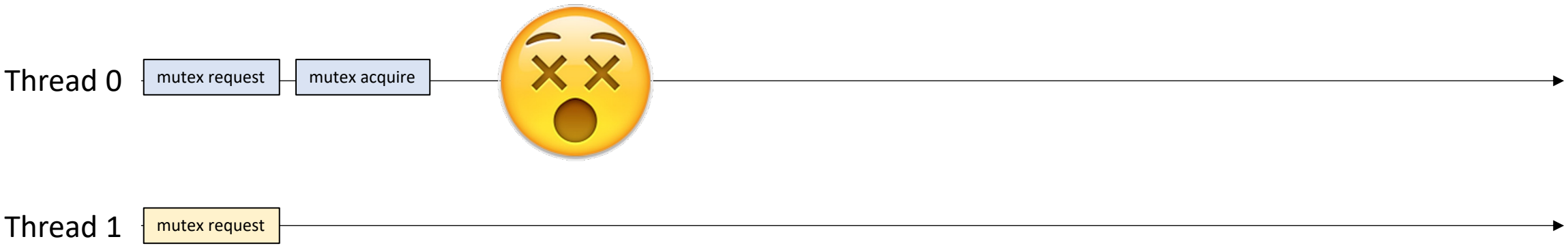


Progress properties

- Going back to specifications:

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

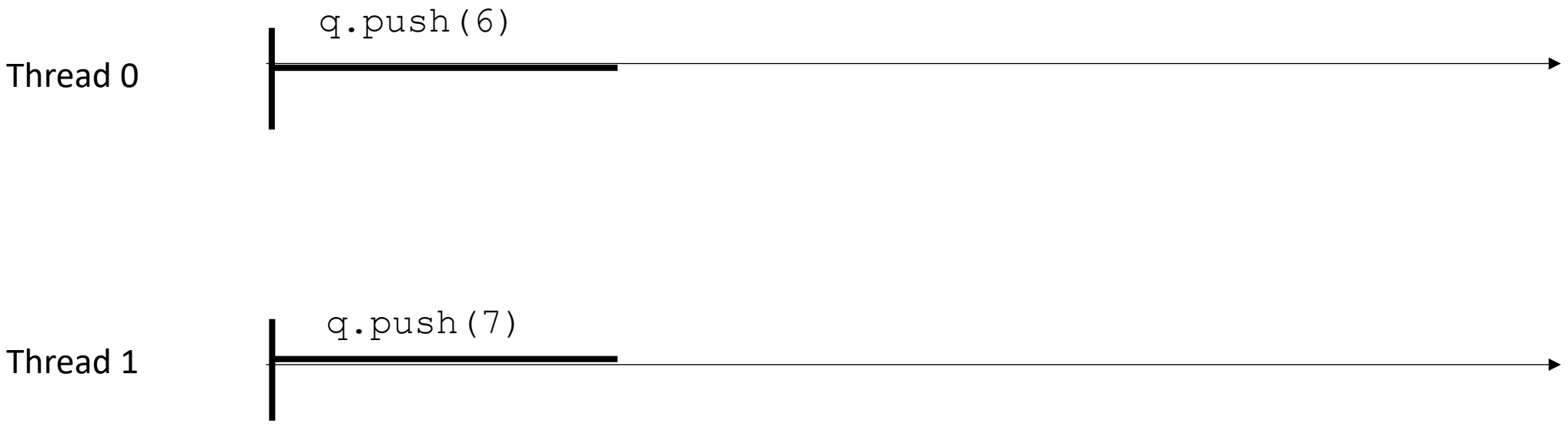
Recall the mutex



What now?!

Linearizability

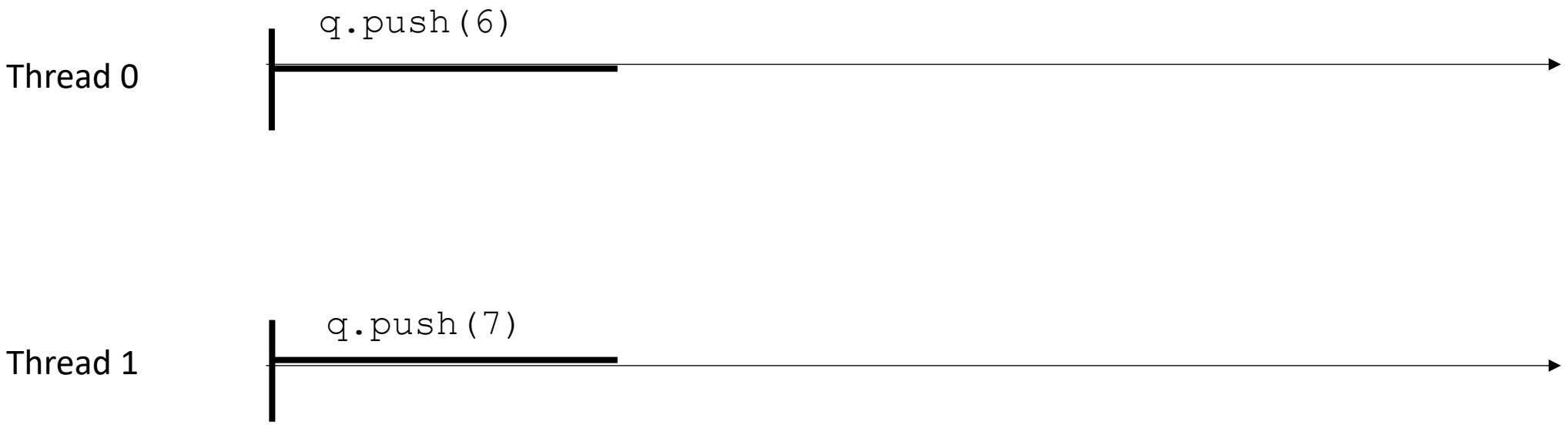
Two unfinished commands.



Linearizability

Two unfinished commands.

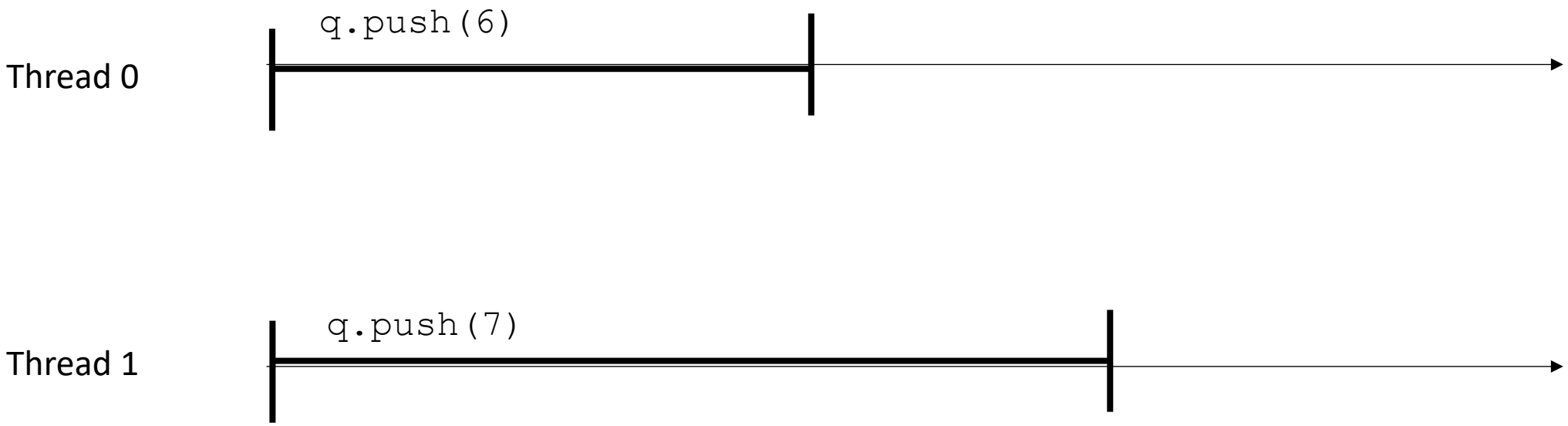
Linearizability does not dictate that one needs to wait for another



Linearizability

Two unfinished commands.

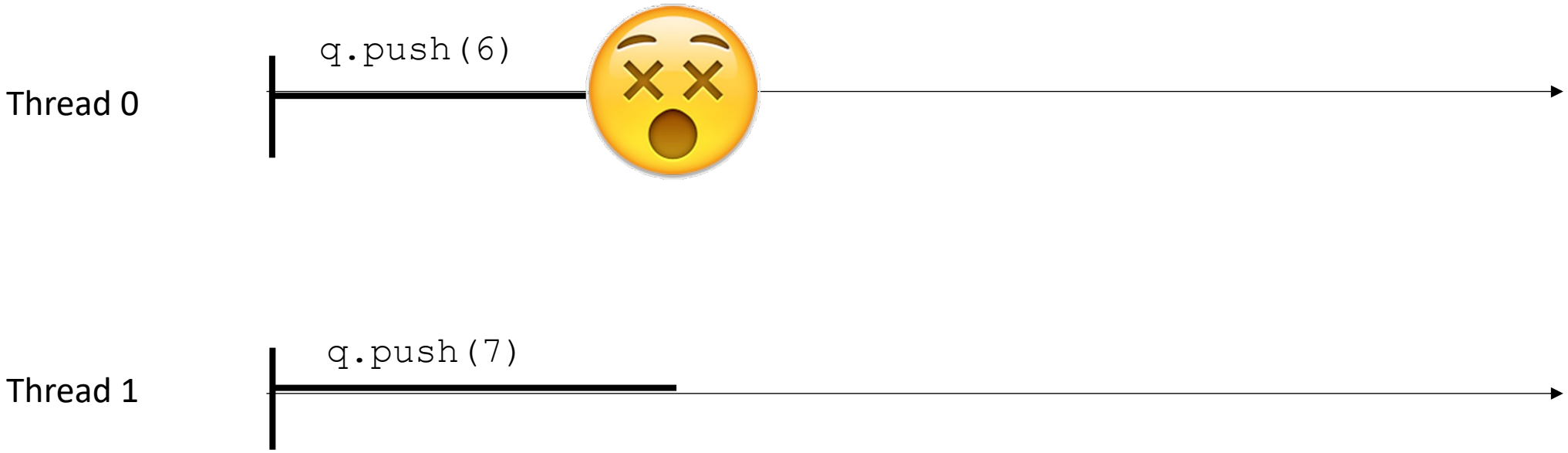
Linearizability does not dictate that one needs to wait for another



Linearizability

Two unfinished commands.

Linearizability does not dictate that one needs to wait for another

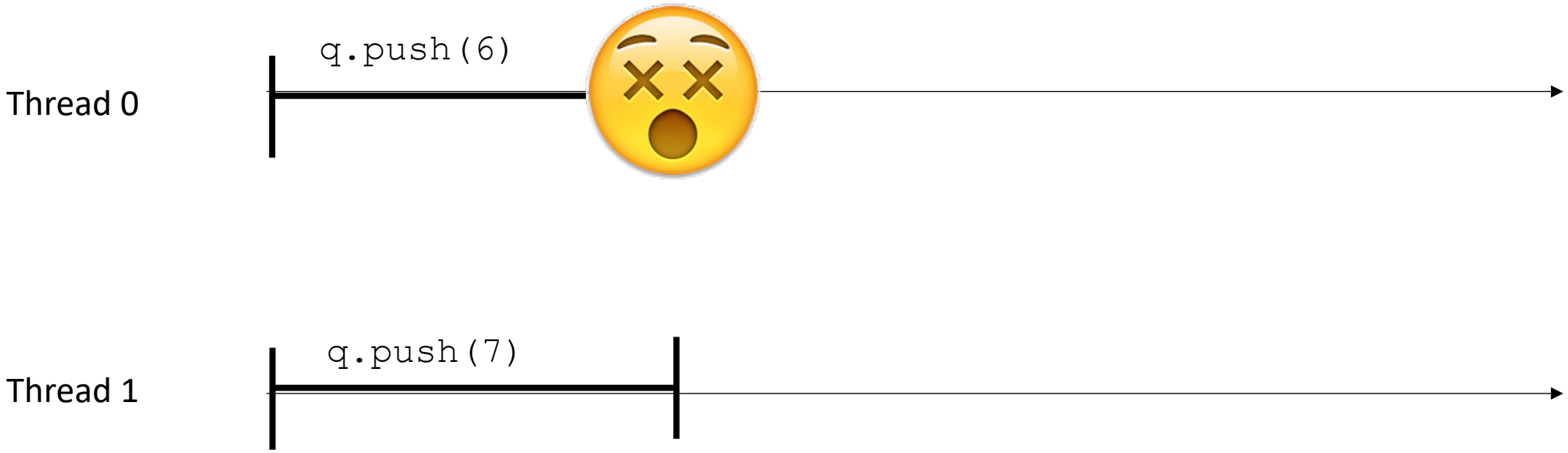


for mutexes, the specification required that the system hang.

Linearizability

Two unfinished commands.

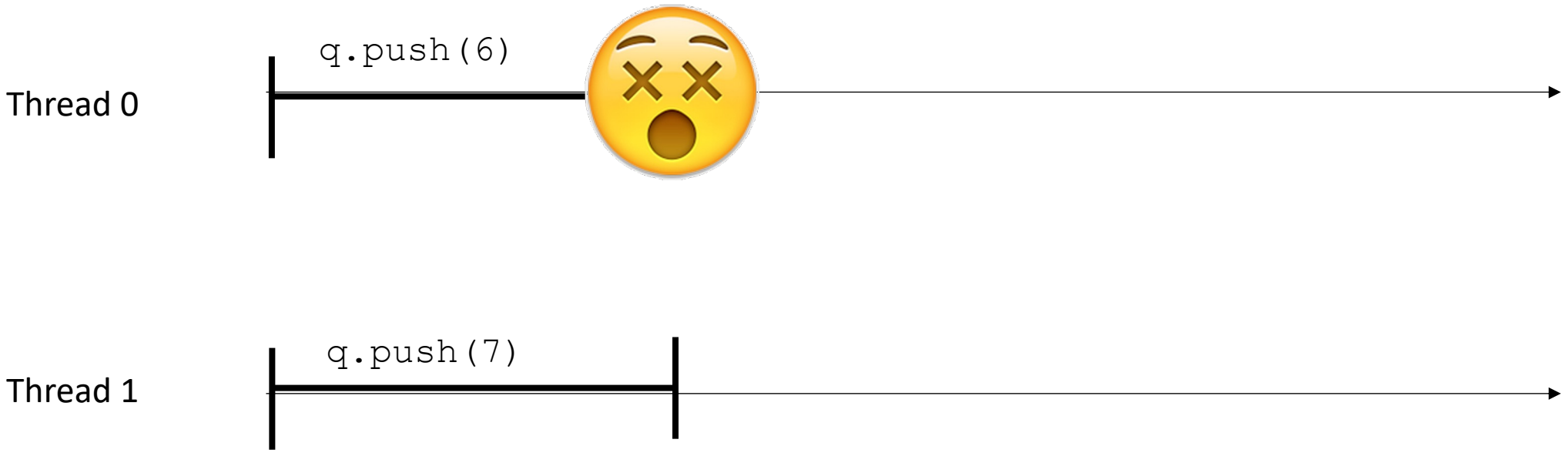
Linearizability does not dictate that one needs to wait for another



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

Linearizability

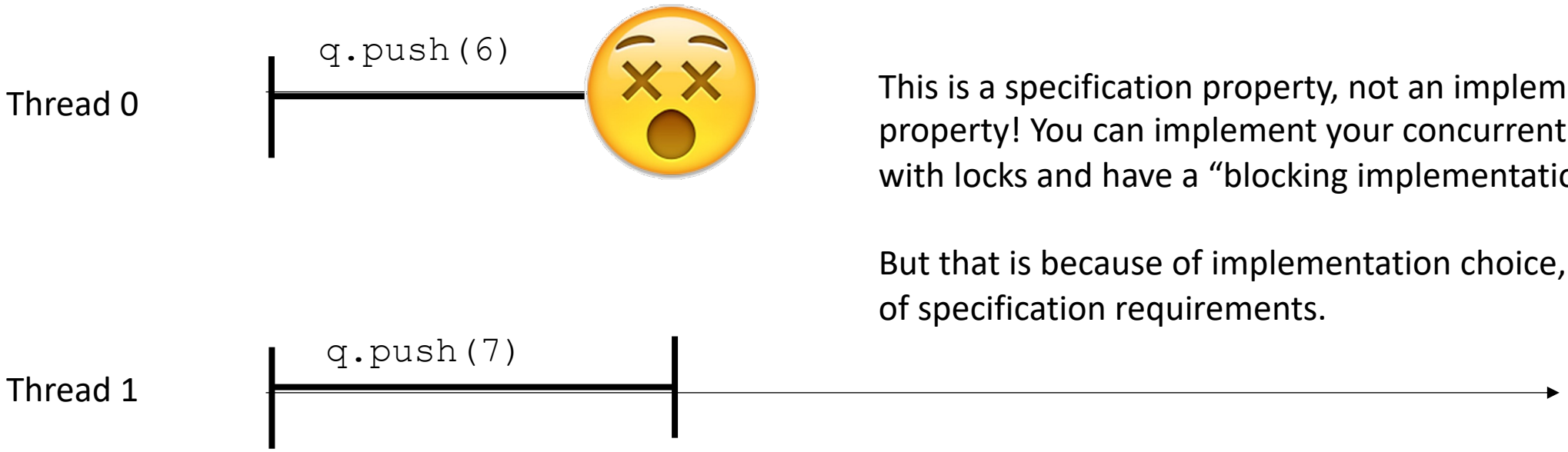
Non-blocking specification:
Every thread is allowed to continue executing
REGARDLESS of the behavior of other threads



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

Linearizability

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



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

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

Terminology overview

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

Terminology overview

- Sequential consistency:
- Linearizability:
- Linearizability point:

Starting simple

Concurrent Queues

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

Concurrent Queues

- General implementation given in Chapter 10 of the book.
- Similar types of reasoning as the linked list
  - Lots of reasoning about node insertion, node deletion
  - Using atomic RMWs (CAS) in clever ways

- We will think about specialized queues
  - Implementations can be simplified!

Input/Output Queues

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

- Why would we want a thing?

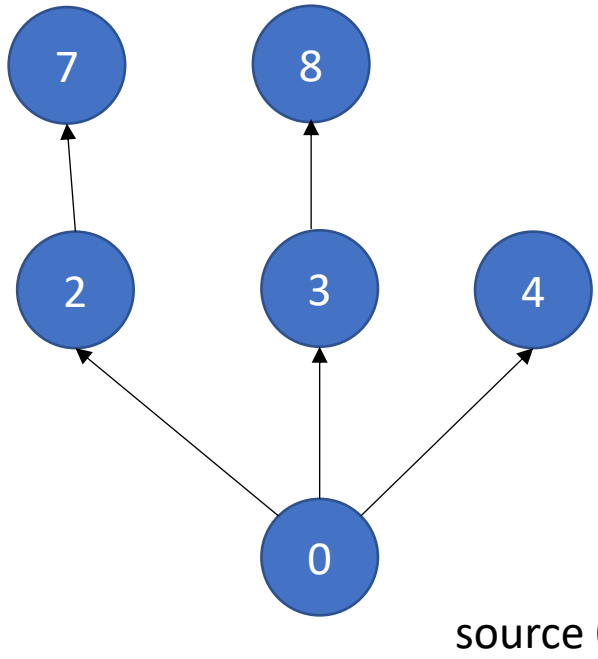
- Computation done in phases:
  - First phase prepares the queue (by writing into it)
  - All threads join
  - Second phase reads values from the queue.

Input/Output Queues

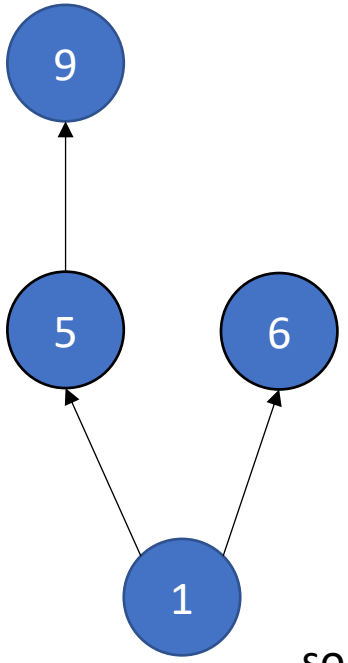
- Example: Information flow in graph applications:

Input/Output Queues

- Example: Information flow in graph applications:



source 0



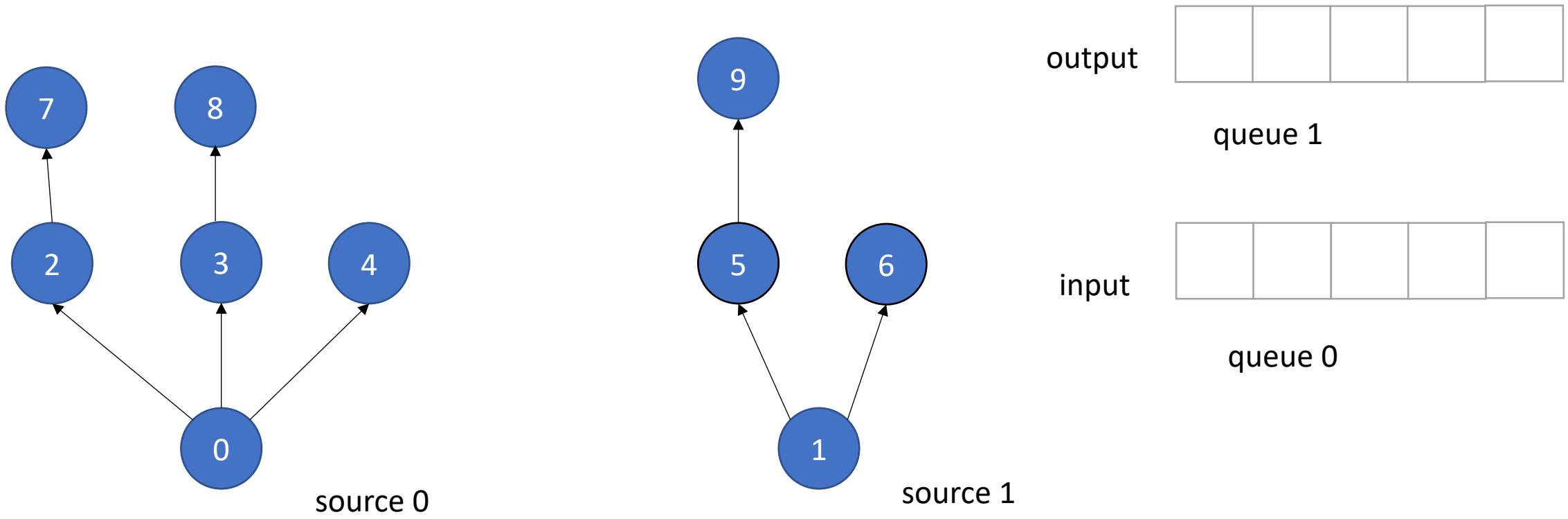
source 1

queue 1

queue 0

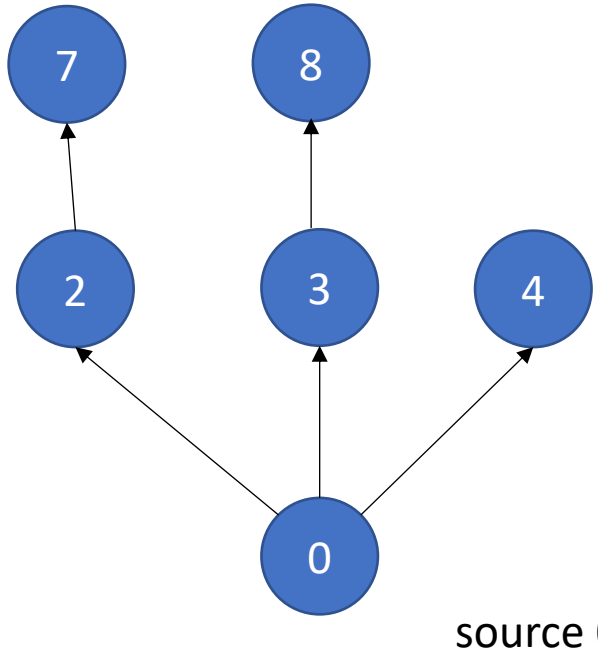
Input/Output Queues

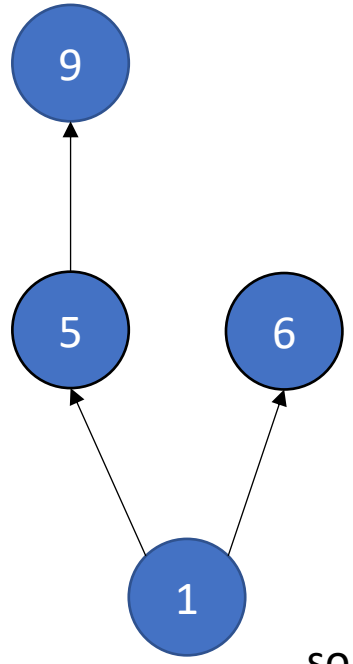
- Example: Information flow in graph applications:

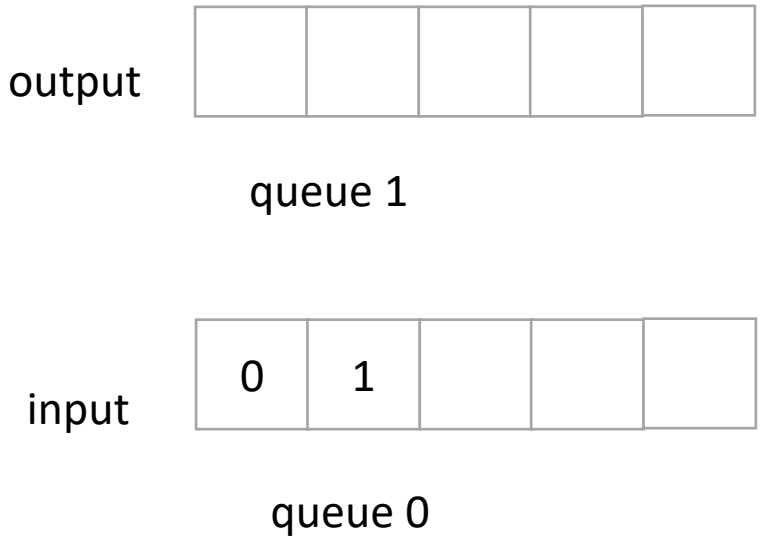


Input/Output Queues

- Example: Information flow in graph applications:

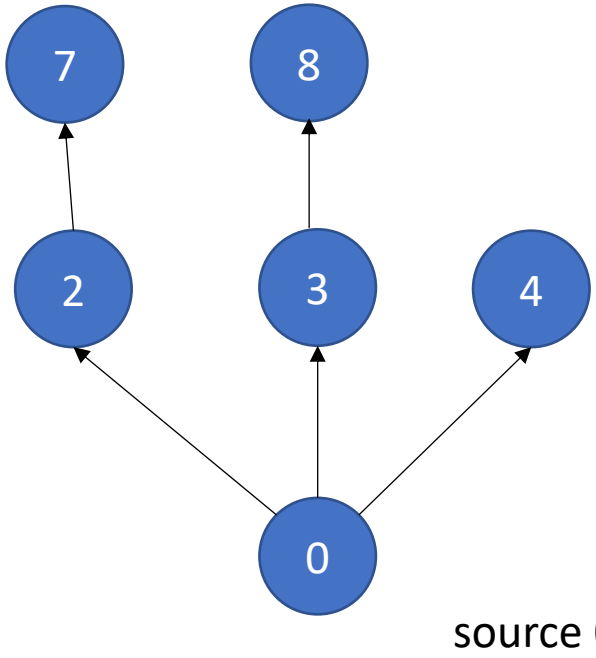


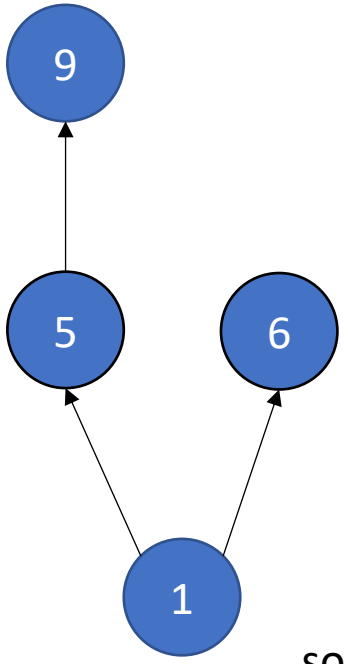


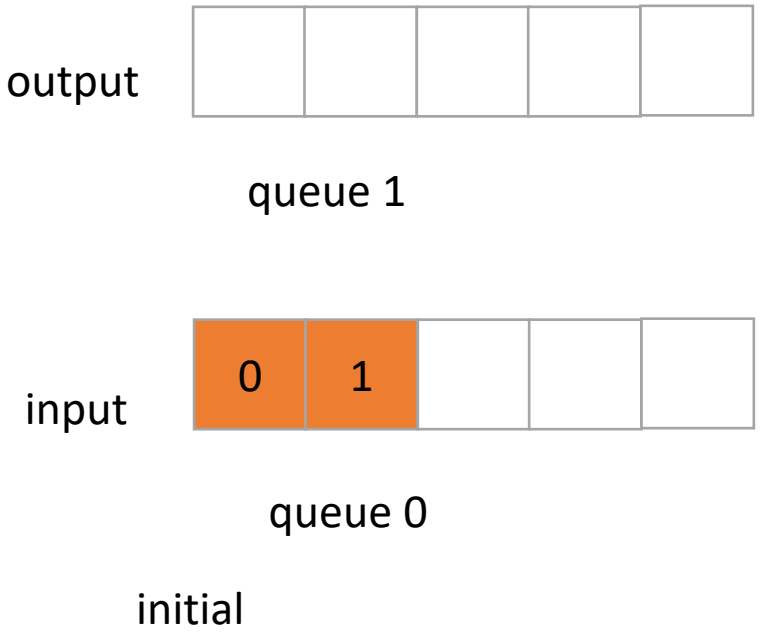


Input/Output Queues

- Example: Information flow in graph applications:

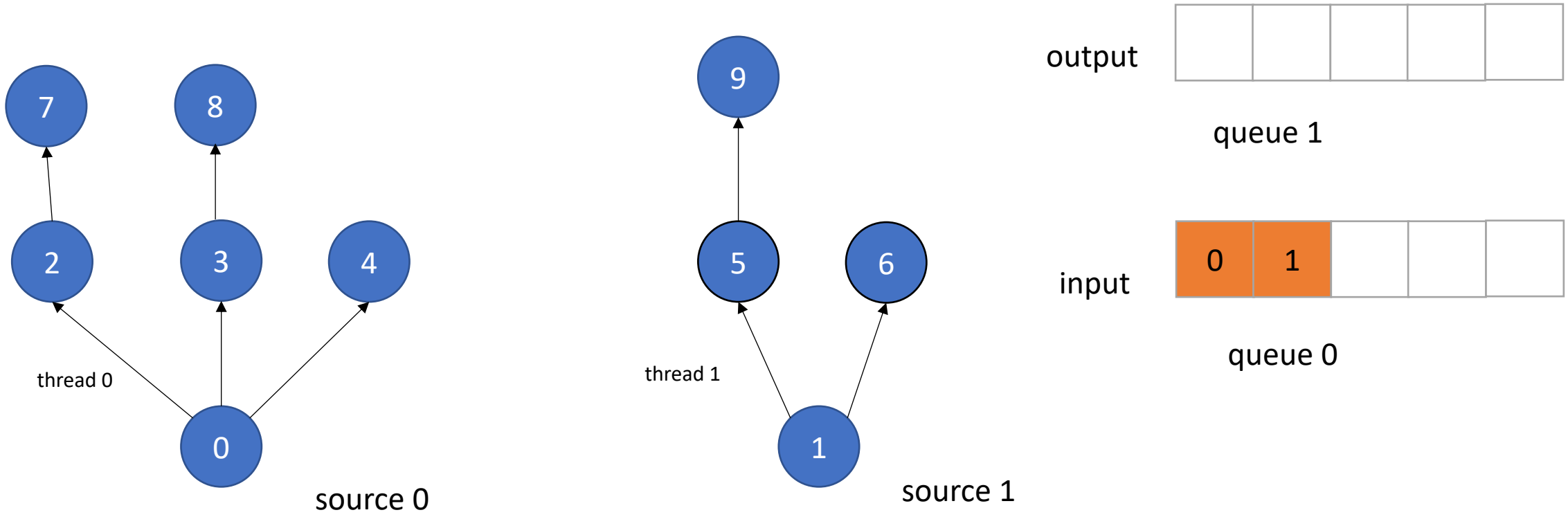






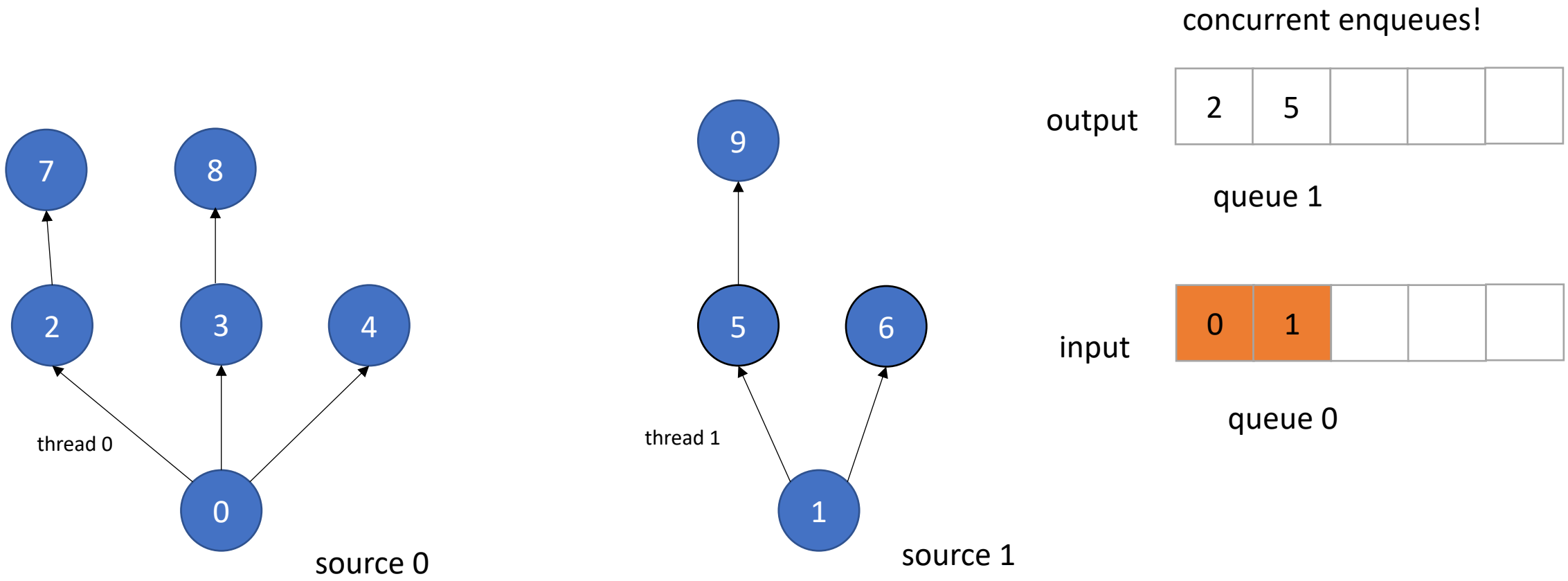
Input/Output Queues

- Example: Information flow in graph applications:



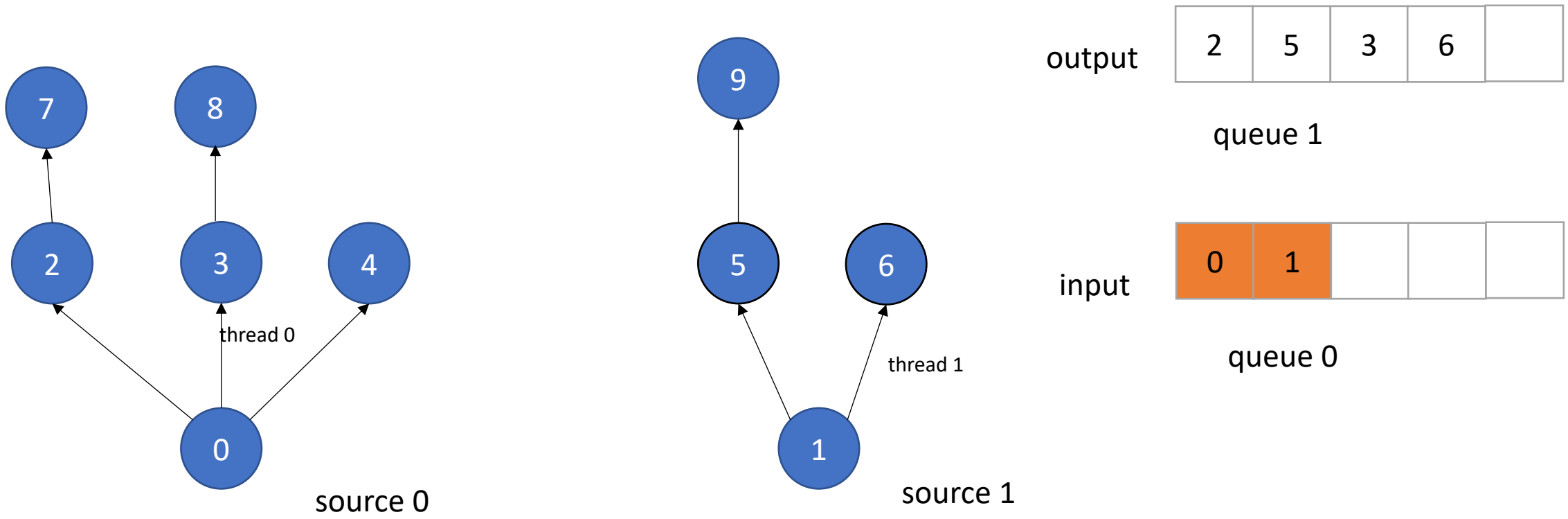
Input/Output Queues

- Example: Information flow in graph applications:



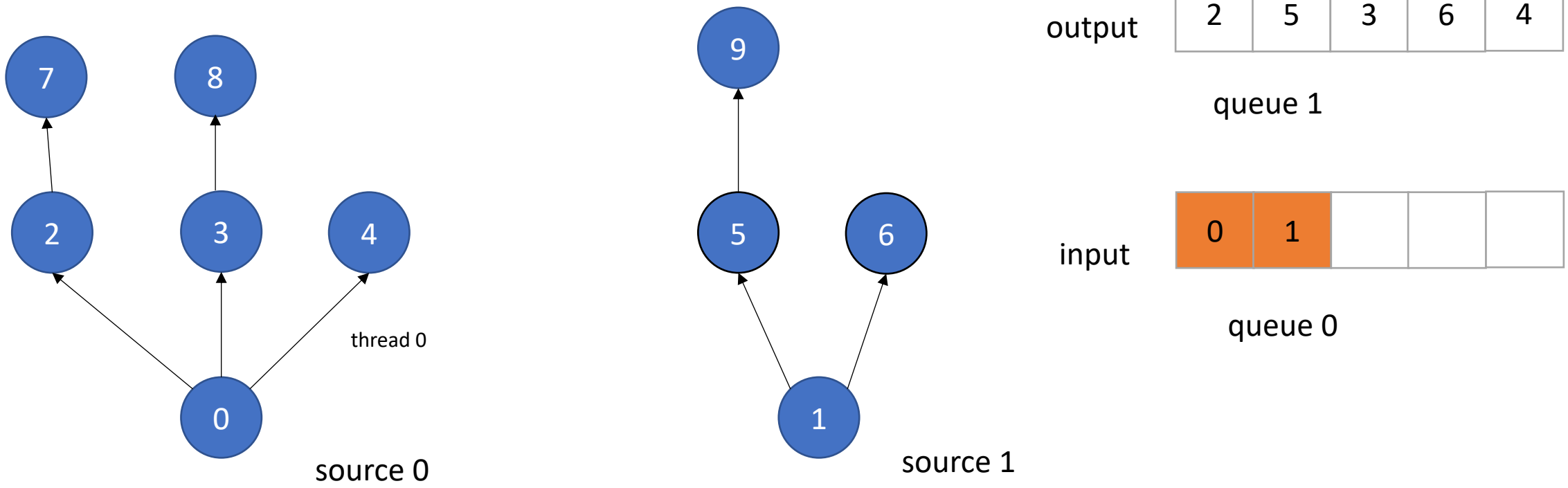
Input/Output Queues

- Example: Information flow in graph applications:



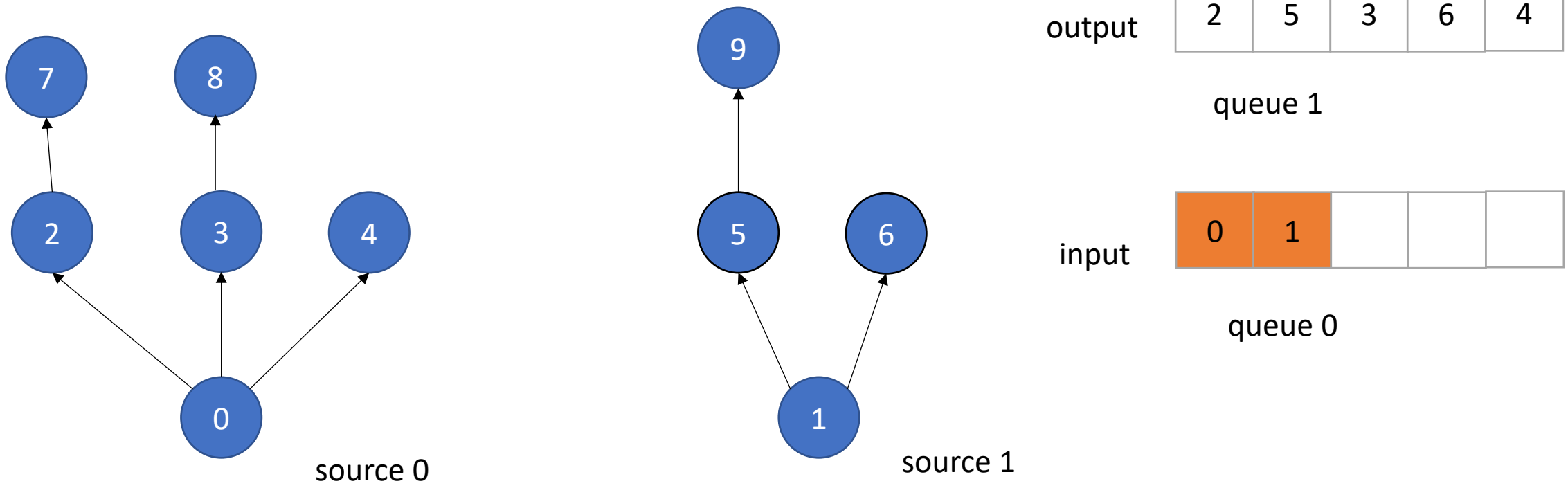
Input/Output Queues

- Example: Information flow in graph applications:



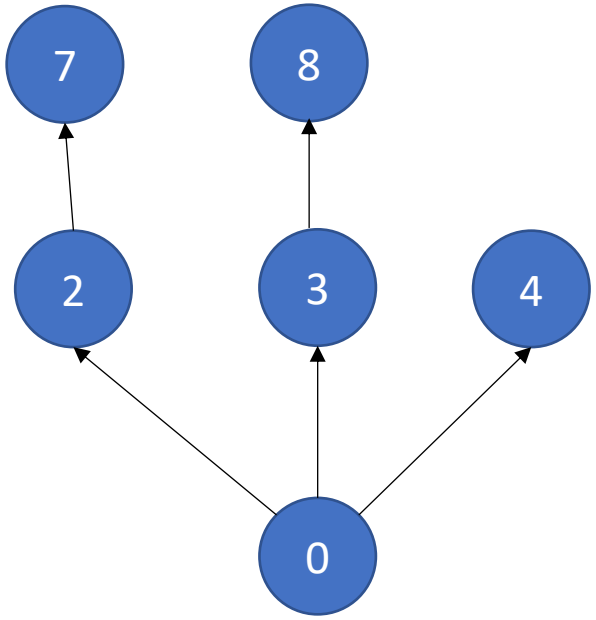
Input/Output Queues

- Example: Information flow in graph applications:

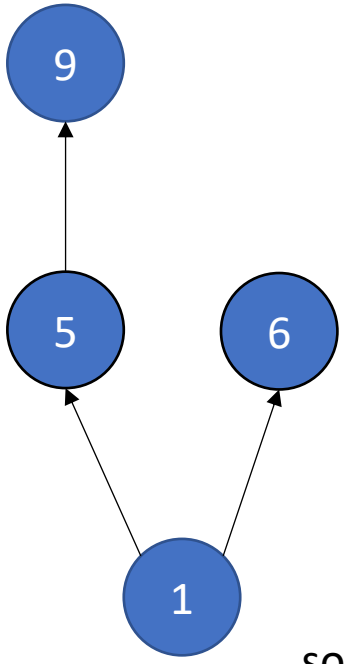


Input/Output Queues

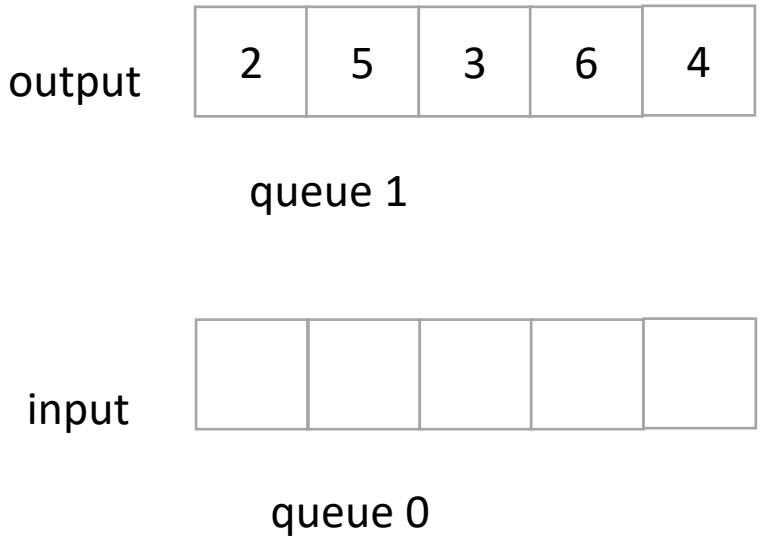
- Example: Information flow in graph applications:



source 0



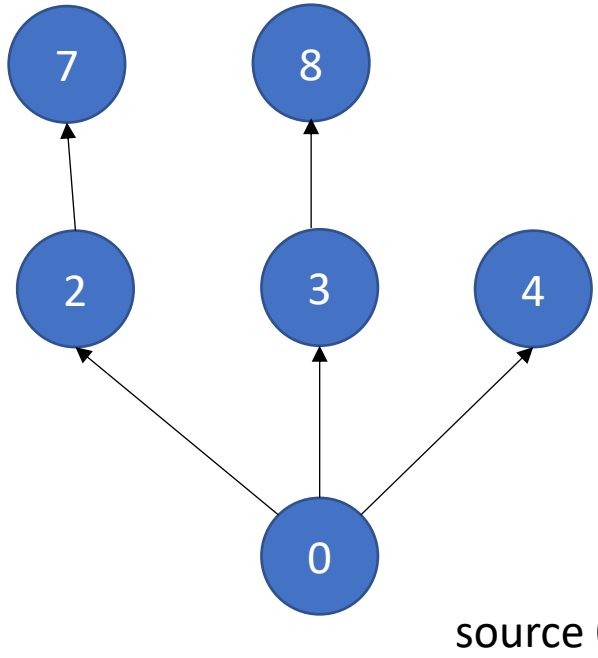
source 1

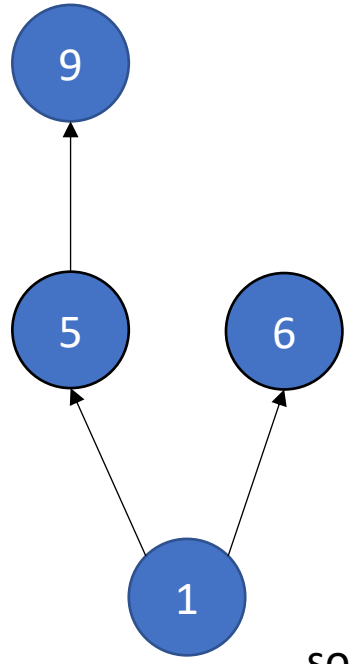


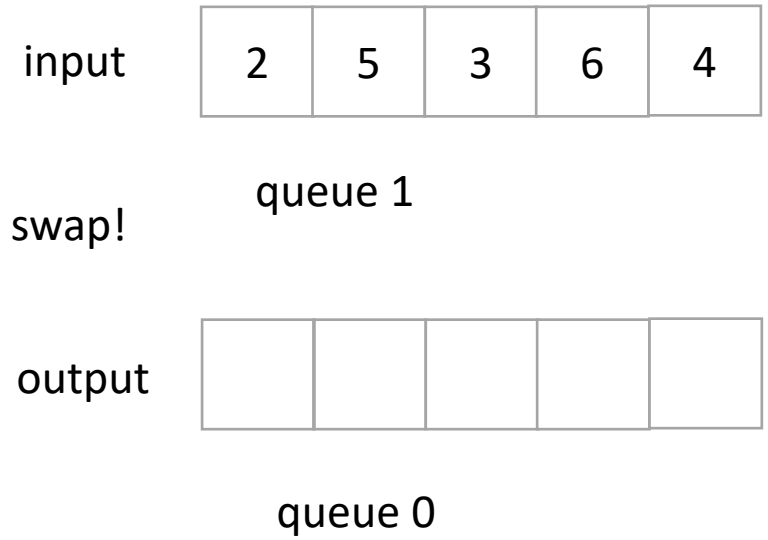
join all threads and clear input

Input/Output Queues

- Example: Information flow in graph applications:

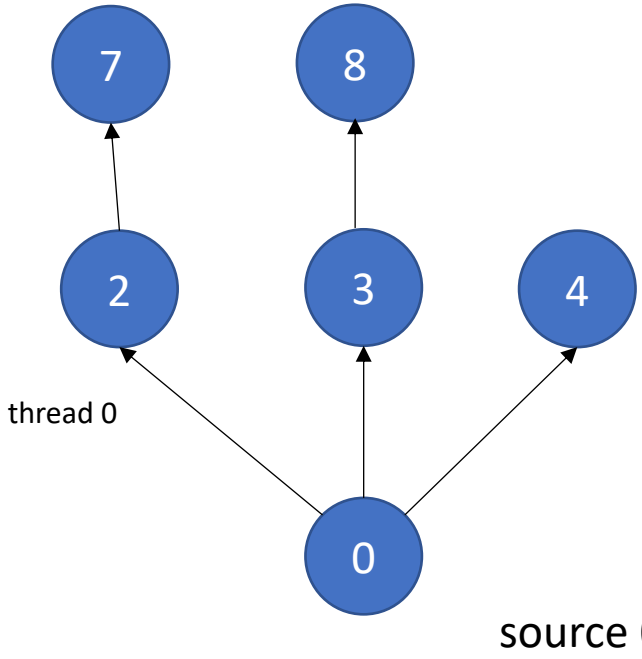


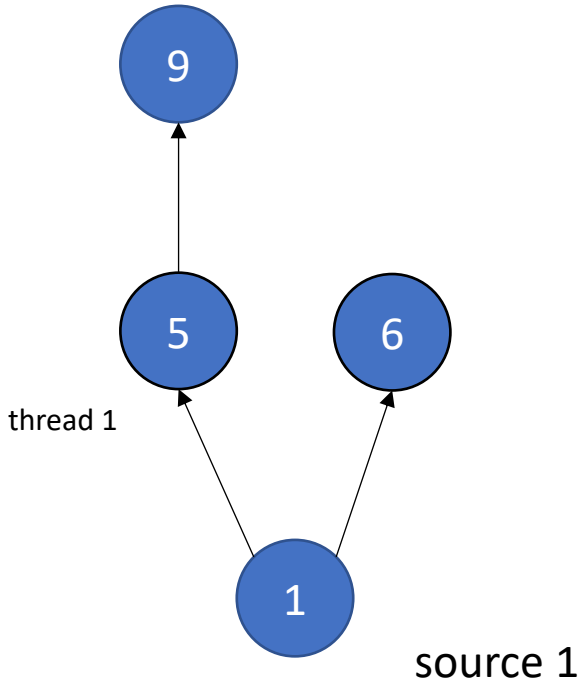


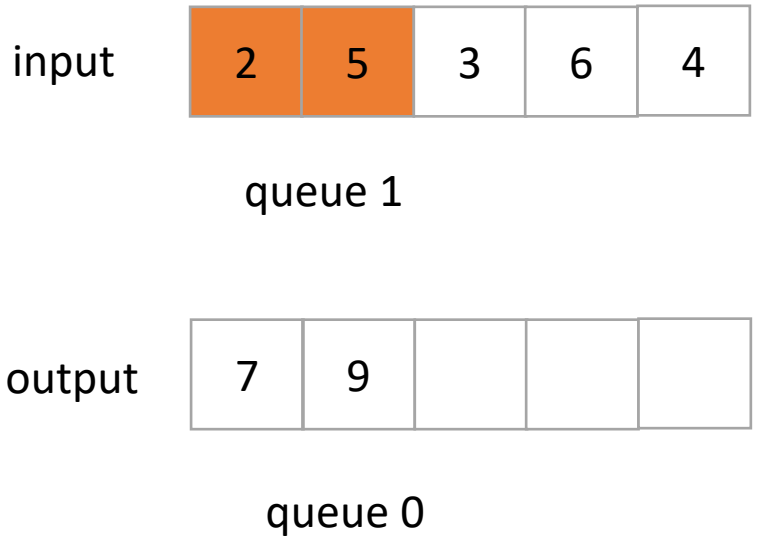


Input/Output Queues

- Example: Information flow in graph applications:

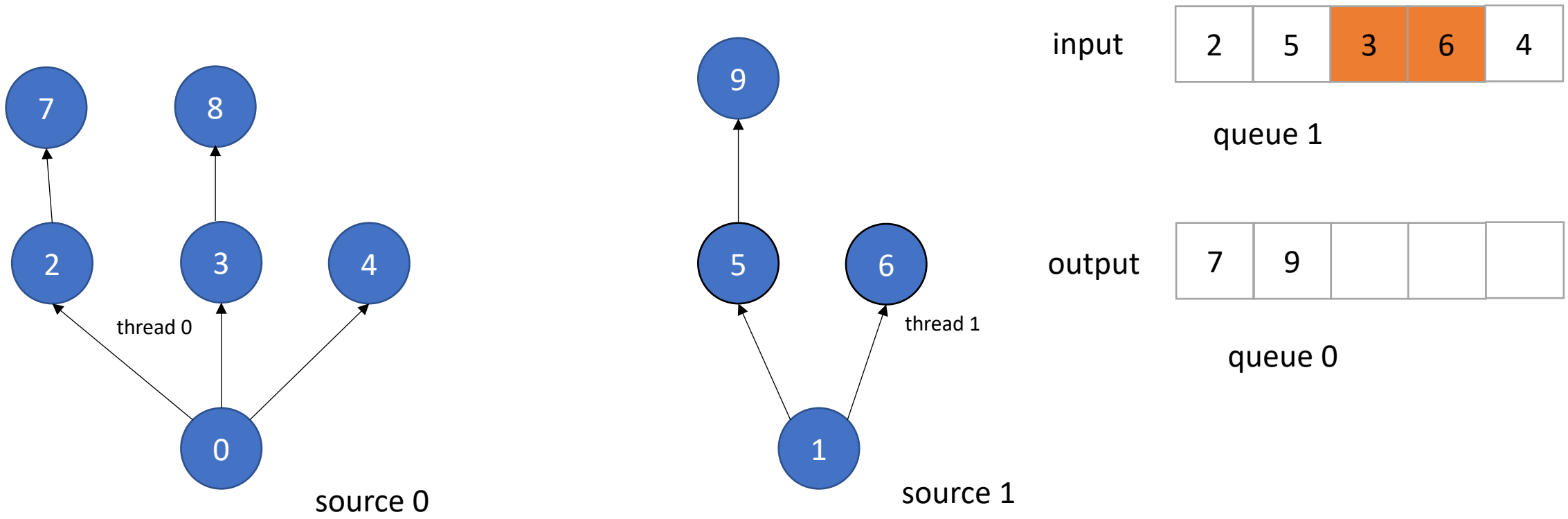






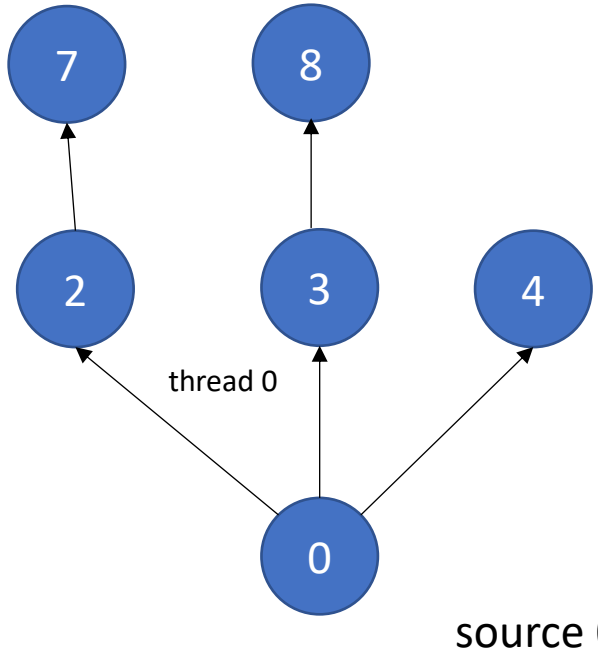
Input/Output Queues

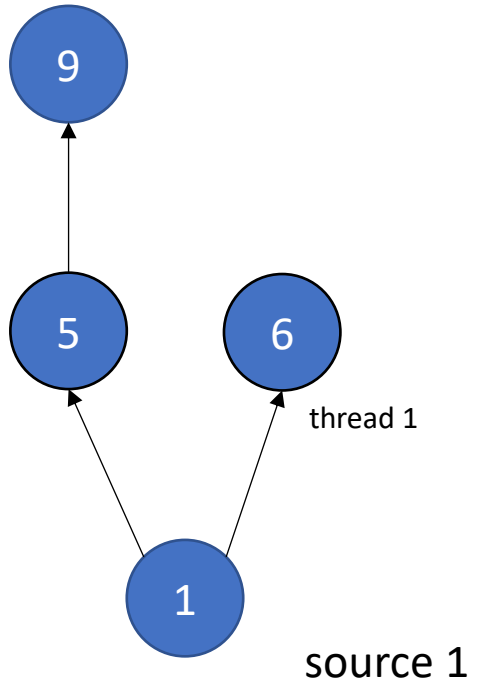
- Example: Information flow in graph applications:

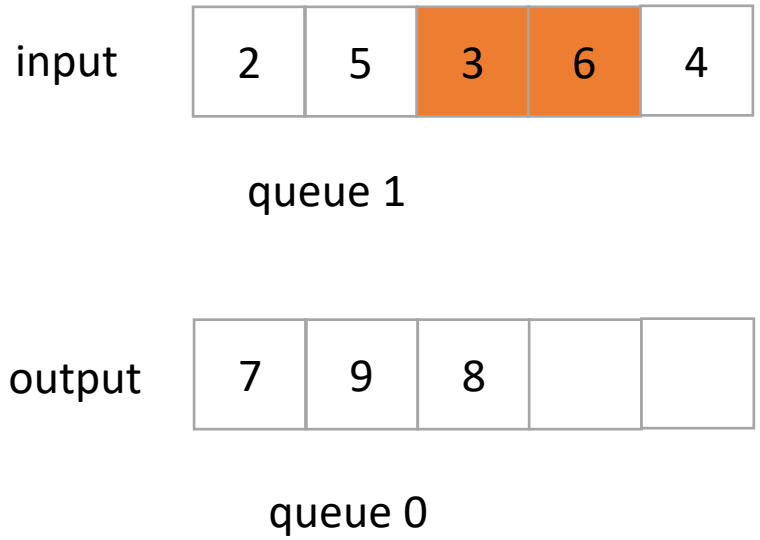


Input/Output Queues

- Example: Information flow in graph applications:

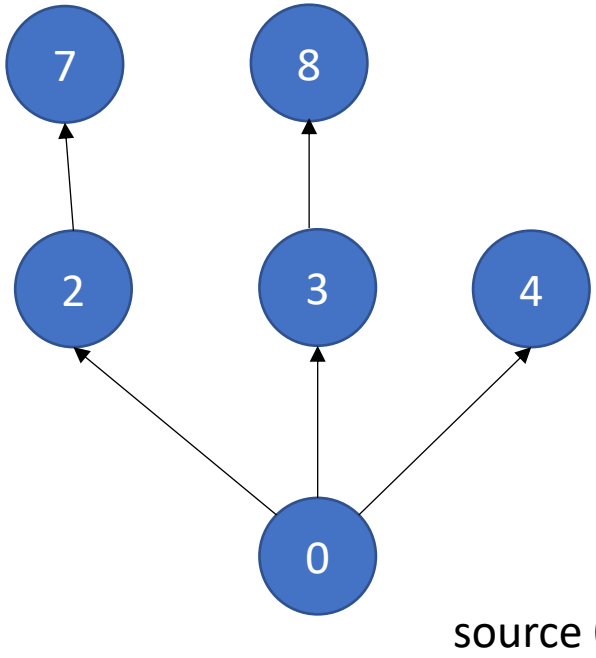


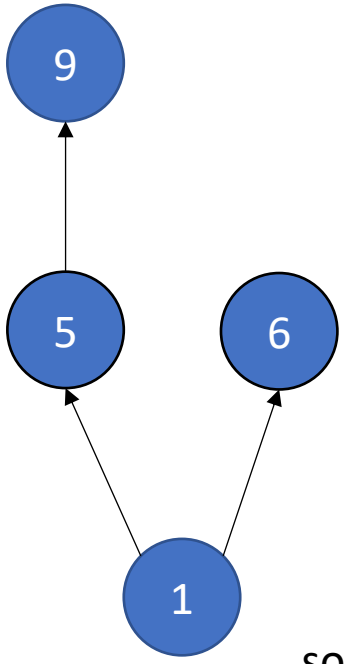


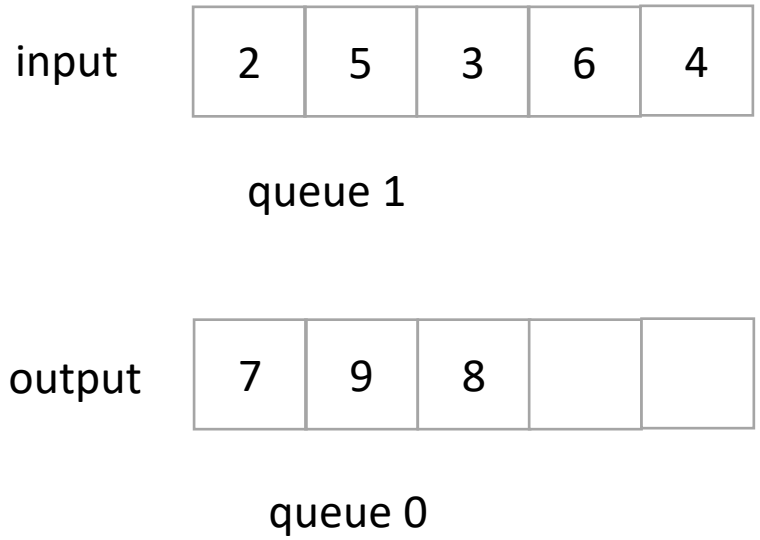


Input/Output Queues

- Example: Information flow in graph applications:

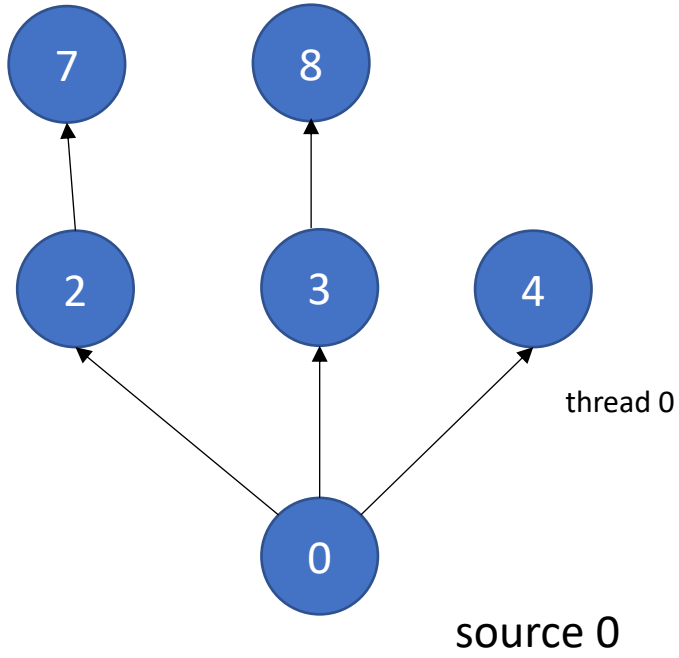


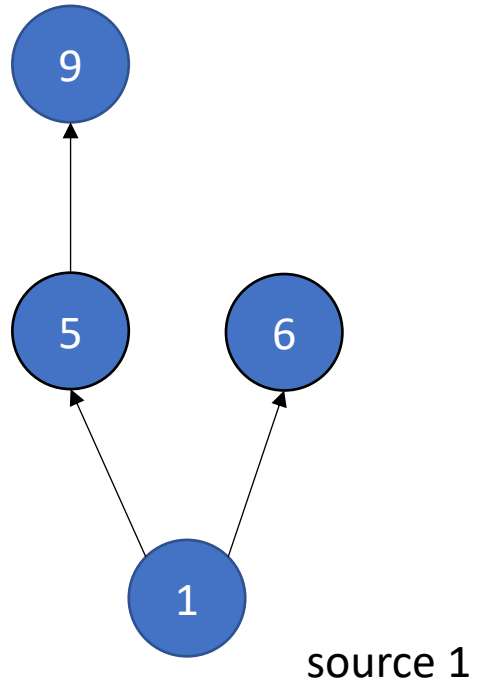


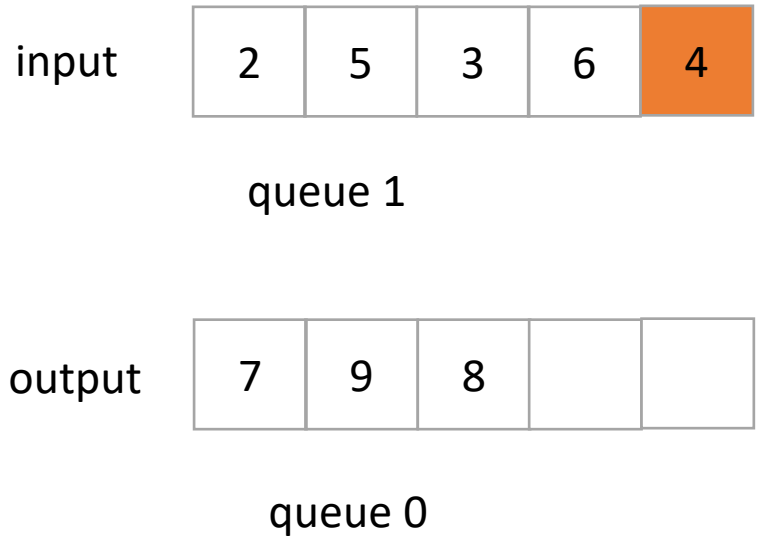


Input/Output Queues

- Example: Information flow in graph applications:

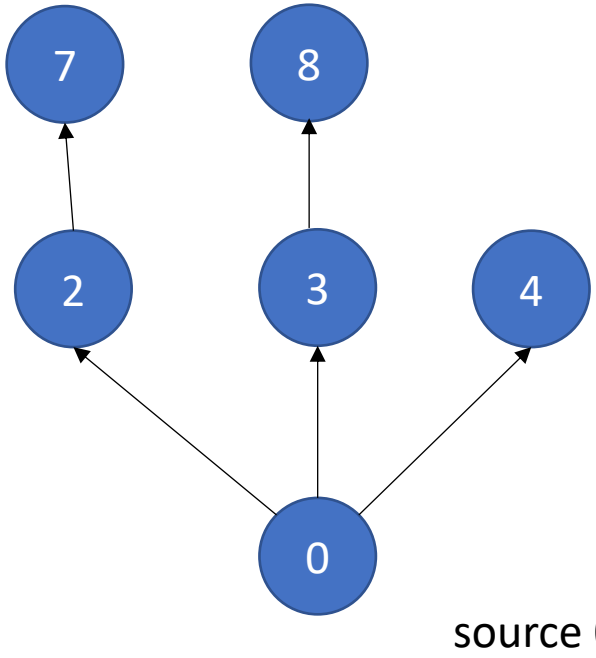


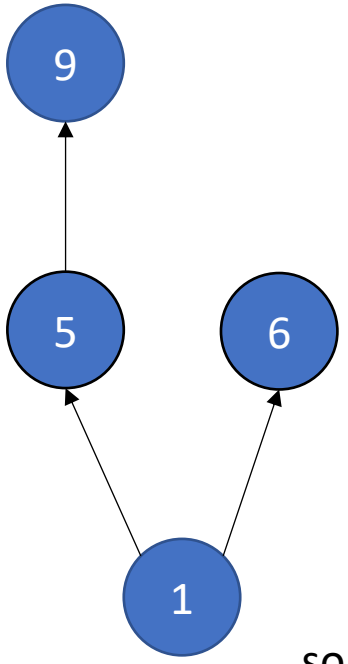


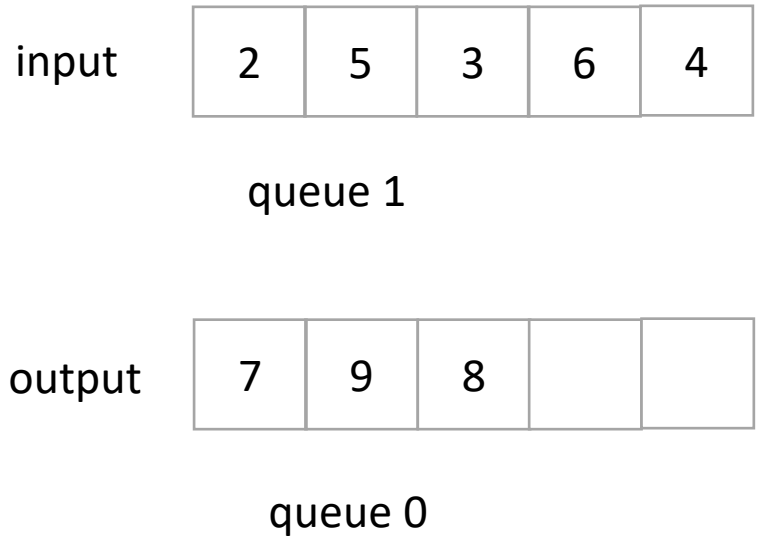


Input/Output Queues

- Example: Information flow in graph applications:

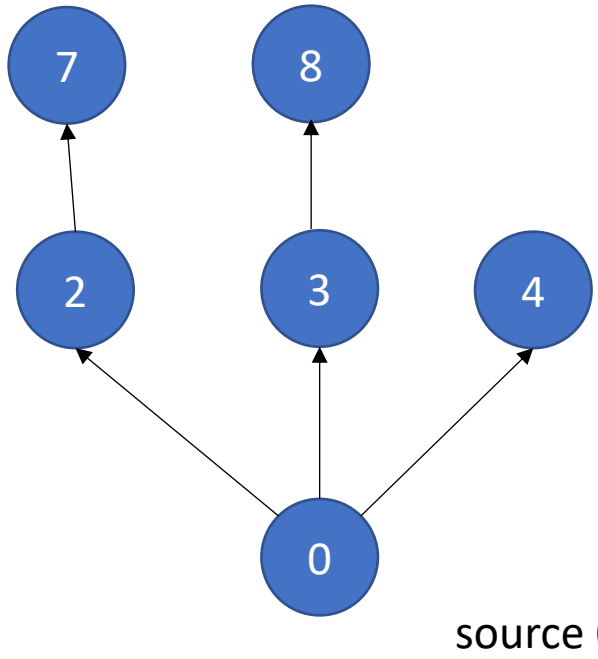


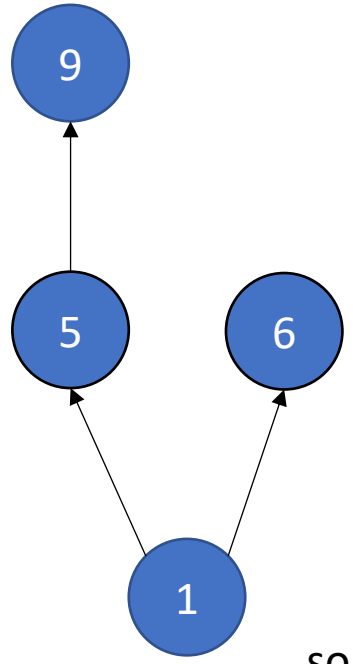




Input/Output Queues

- Example: Information flow in graph applications:





and so on...

output

queue 1

input

7 9 8

queue 0

Implementation

Implementation

Allocate a contiguous array

Pros:
?

Cons:
?

Implementation

Allocate a contiguous array

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

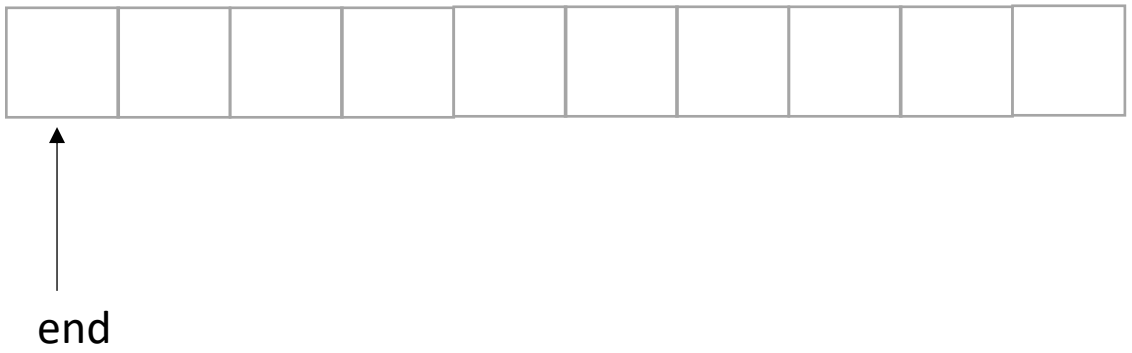
Cons:
- need to reason about overflow!

Note on terminology

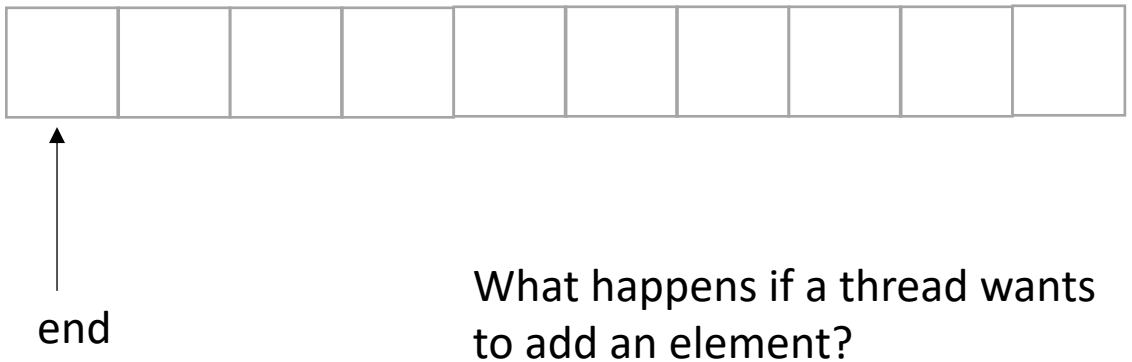
- Head/tail - often used in queue implementations, but switches when we start doing circular buffers.

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

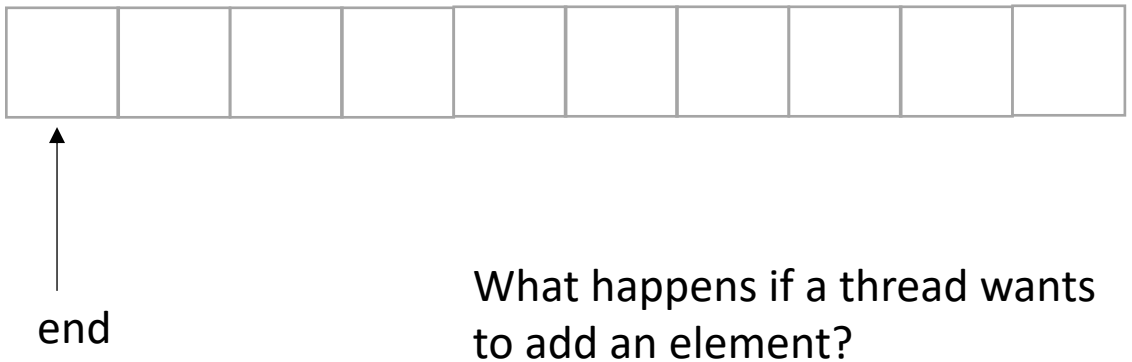
Implementation



Implementation



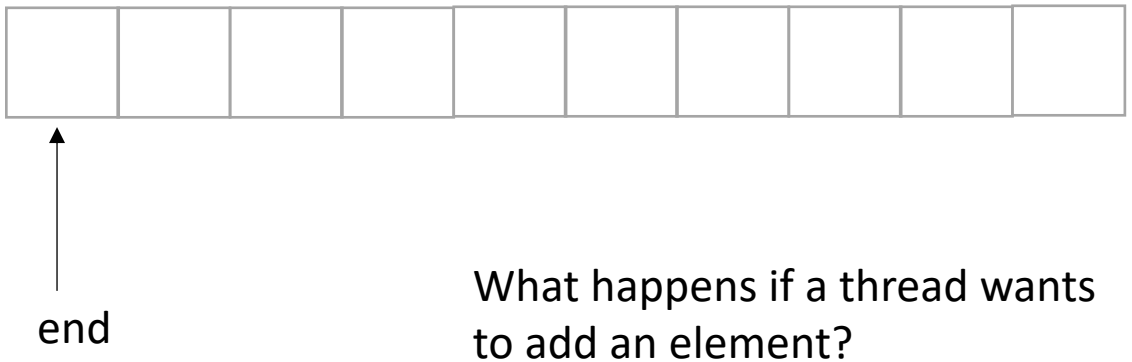
Implementation



What happens if a thread wants to add an element?

Think sequentially:

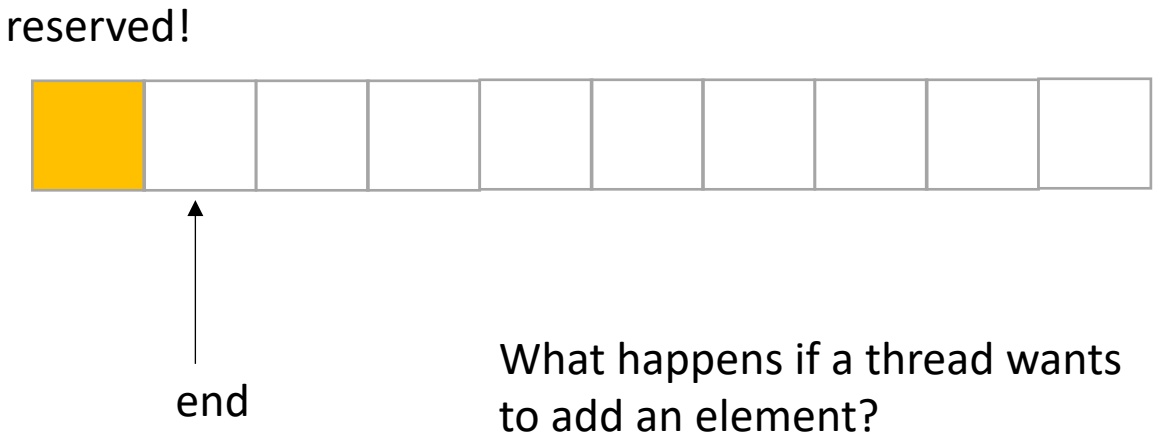
Implementation



What happens if a thread wants to add an element?

Think sequentially:
*reserve a space - increment end

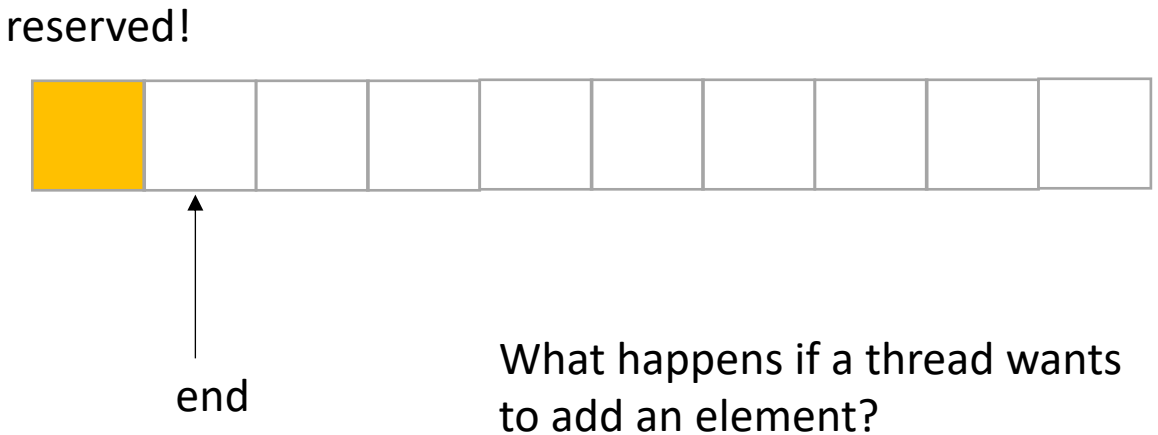
Implementation



What happens if a thread wants to add an element?

Think sequentially:
*reserve a space - increment end

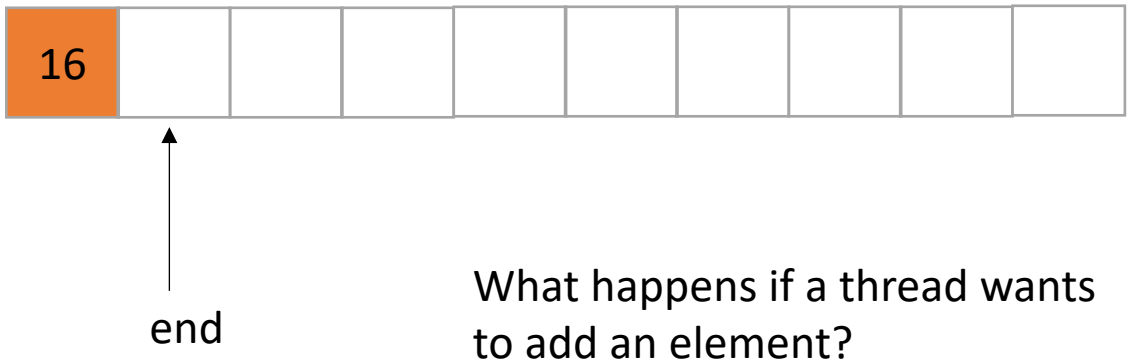
Implementation



What happens if a thread wants to add an element?

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

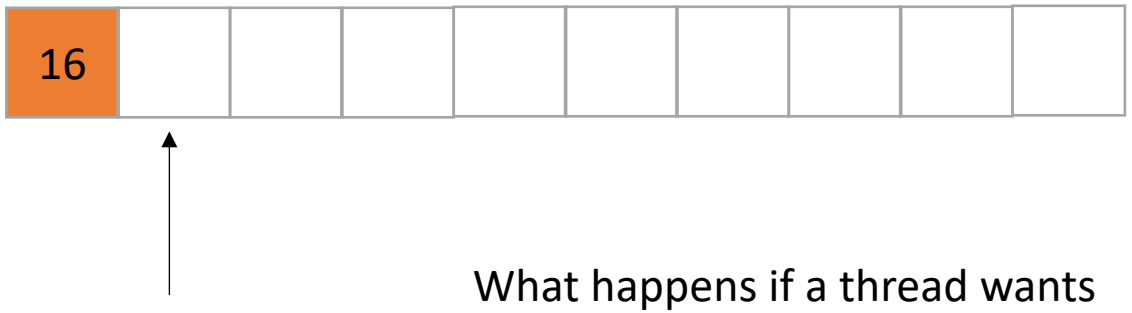
Implementation



What happens if a thread wants to add an element?

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

Implementation

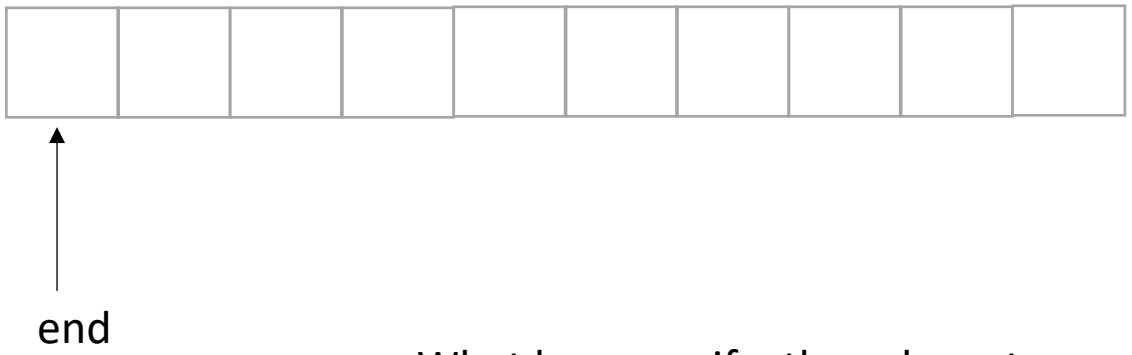


What happens if a thread wants to add an element?

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

done!

Implementation

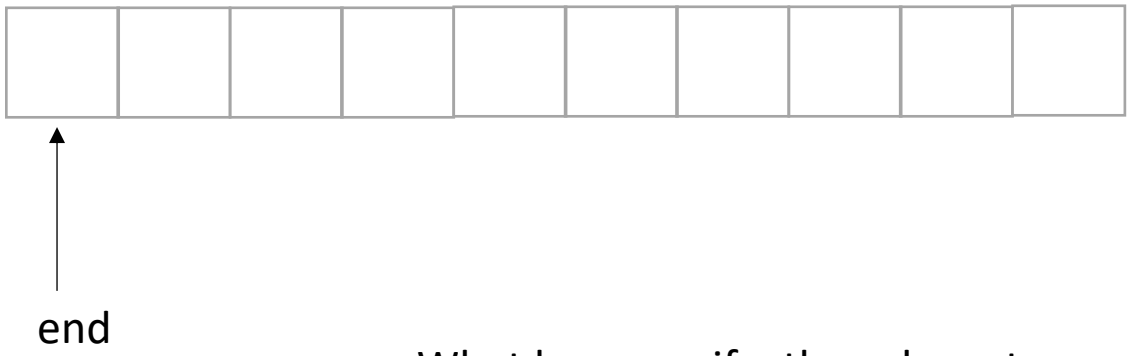


What happens if a thread wants
to add an element?

Think concurrently:

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

Implementation

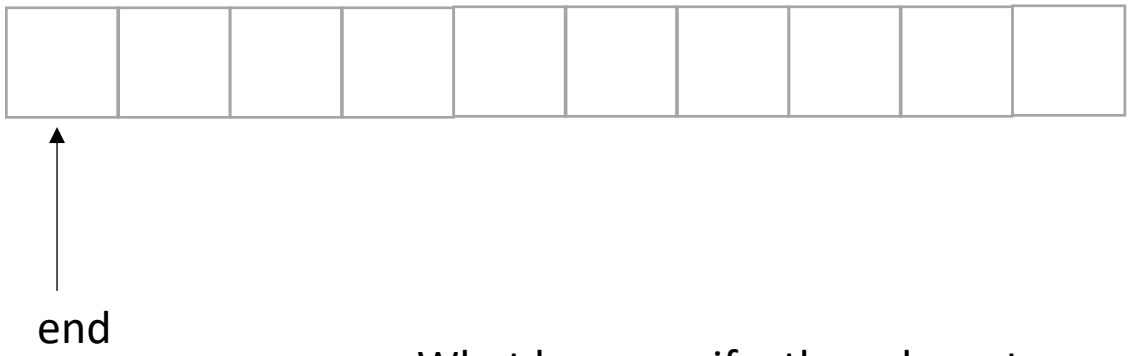


What happens if a thread wants to add an element?

Think concurrently:

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

Implementation



Thread 0:
enq(6);

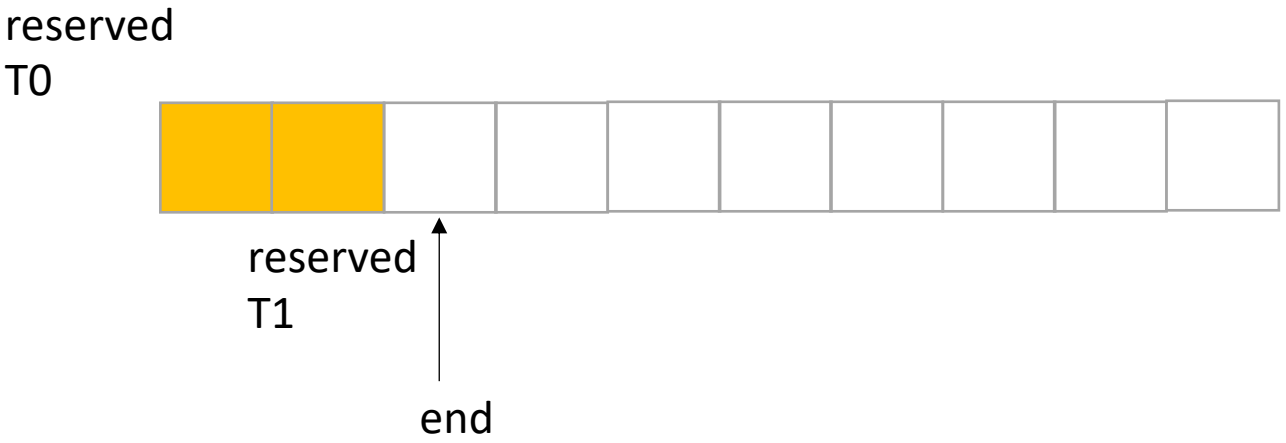
Thread 1:
enq(7);

What happens if a thread wants to add an element?

Think concurrently:

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

Implementation



Thread 0:
enq(6);

Thread 1:
enq(7);

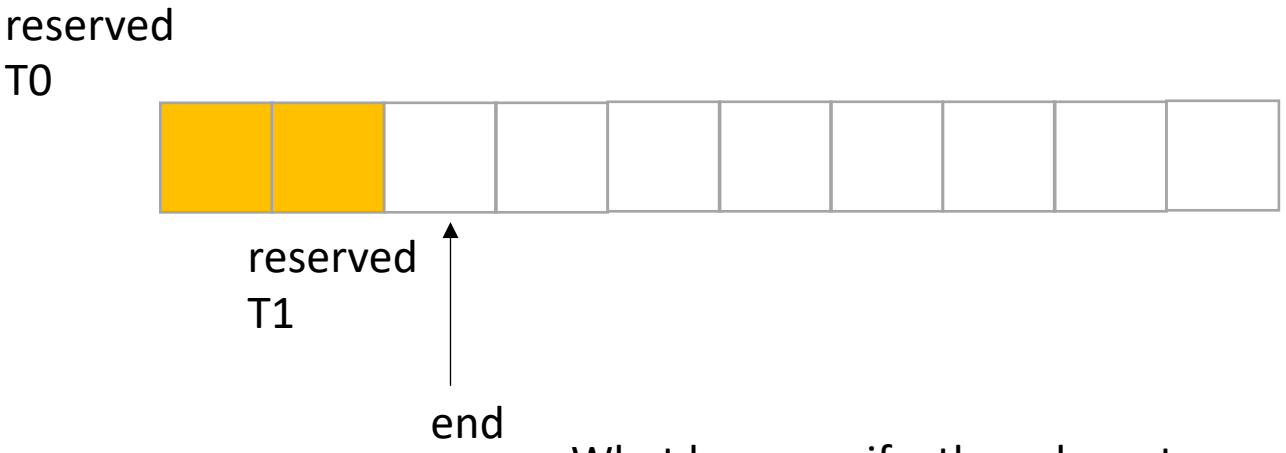
What happens if a thread wants to add an element?

Think concurrently:

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

Implementation

does it matter which order threads add their data?



Thread 0:
enq(6);

Thread 1:
enq(7);

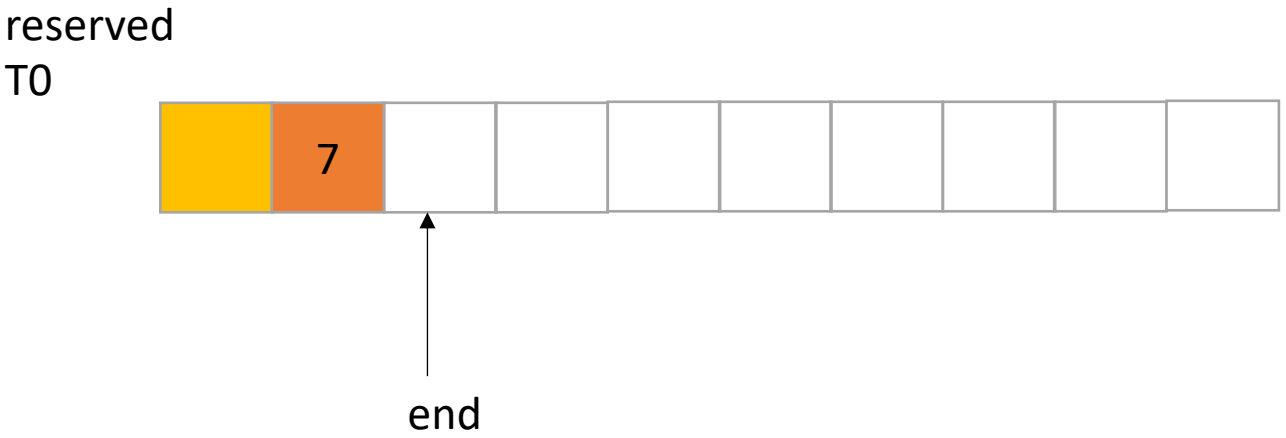
What happens if a thread wants to add an element?

Think concurrently:

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

Implementation

does it matter which order threads add their data?



Thread 0:
enq(6);

Thread 1:
enq(7);

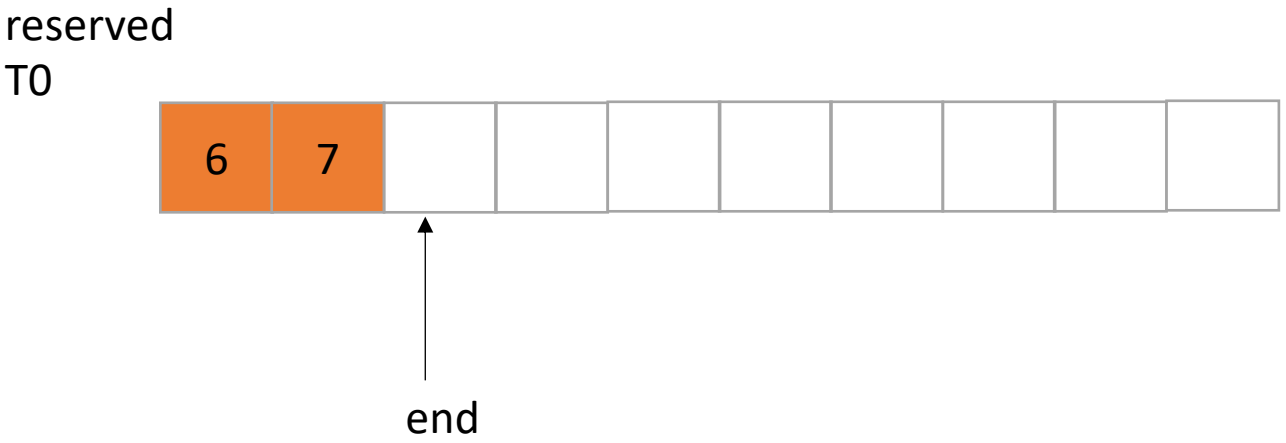
What happens if a thread wants to add an element?

Think concurrently:

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

Implementation

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



Thread 0:
enq(6);

Thread 1:
enq(7);

What happens if a thread wants to add an element?

Think concurrently:

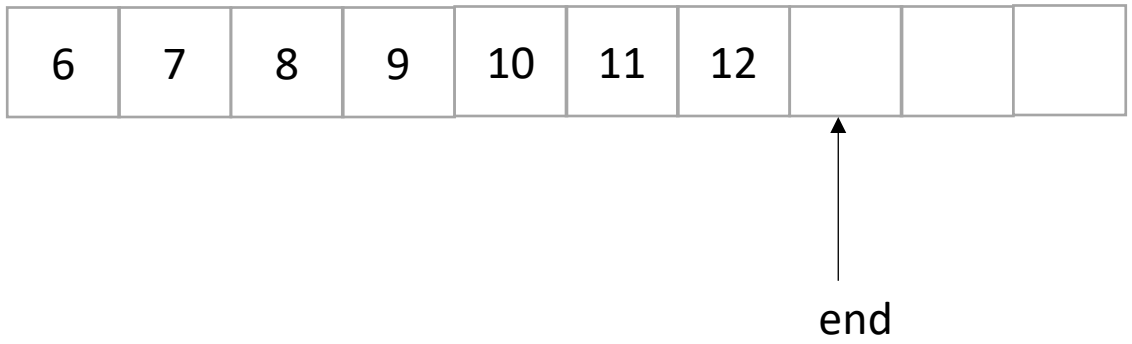
```
reserved_index = atomic_fetch_add(&end, 1);
```

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

  public:
    InputOutputQueue() {
        end = 0;
     }

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

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

How to protect against overflows?

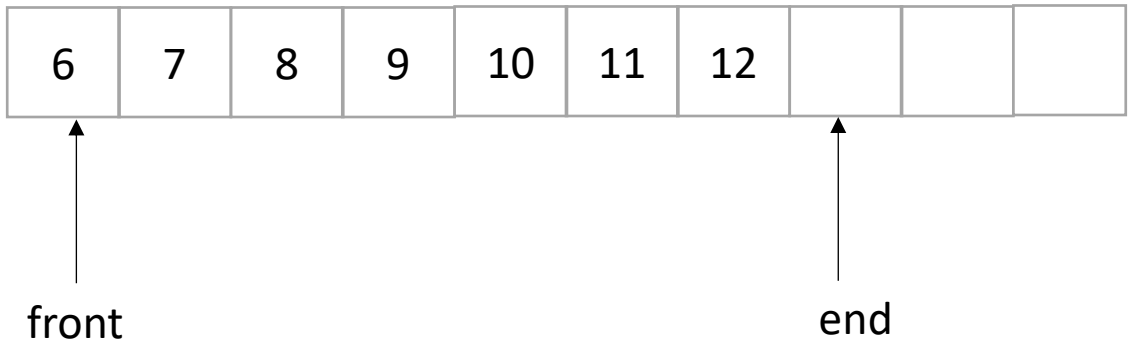
What about Input?

- Now we only do deqs



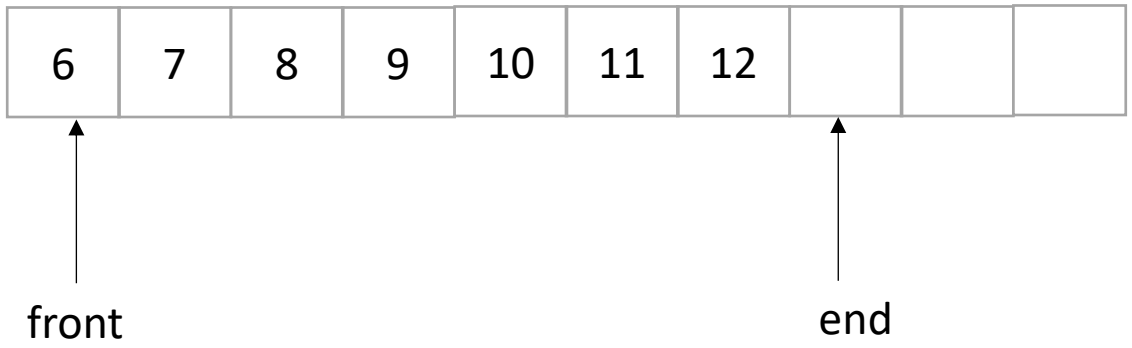
What about Input?

- Now we only do deqs



What about Input?

- Now we only do deqs



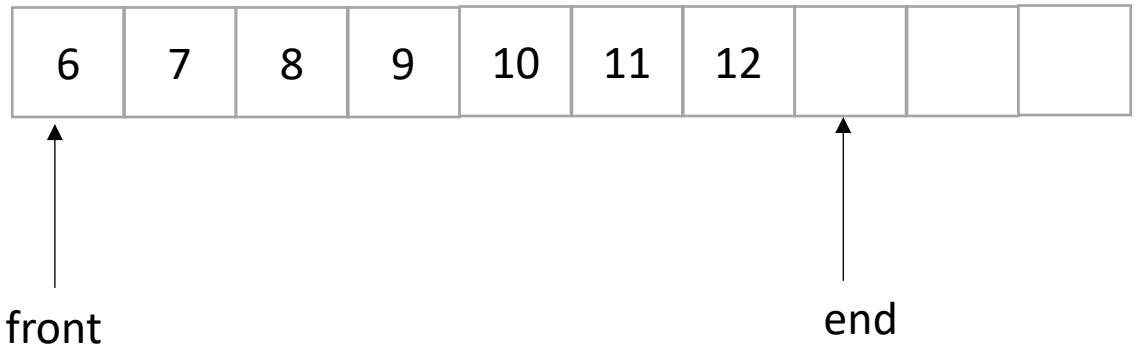
What happens if a thread wants to add an element?

Think concurrently:

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

What about Input?

- Now we only do deqs



Thread 0:
deq();

Thread 1:
deq();

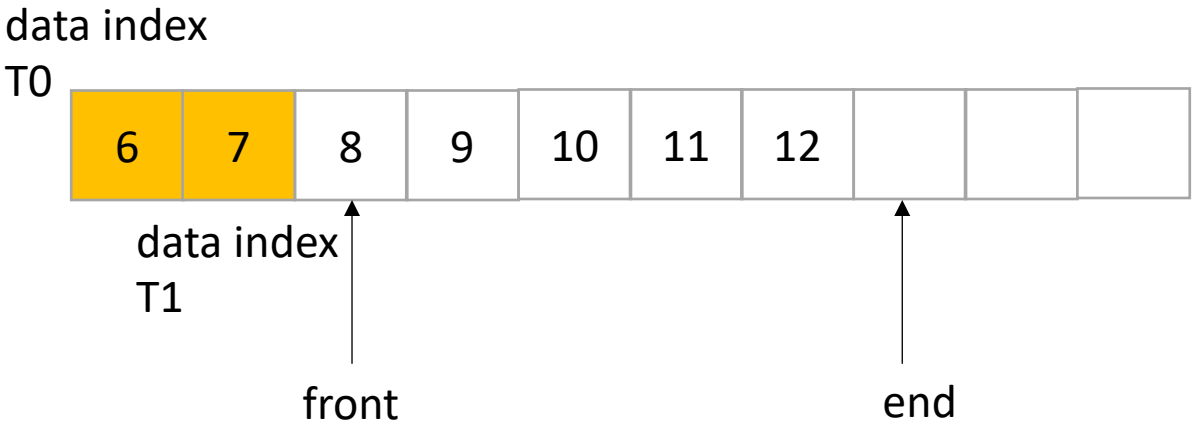
What happens if a thread wants to add an element?

Think concurrently:

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

What about Input?

- Now we only do deqs



Thread 0:
deq();

Thread 1:
deq();

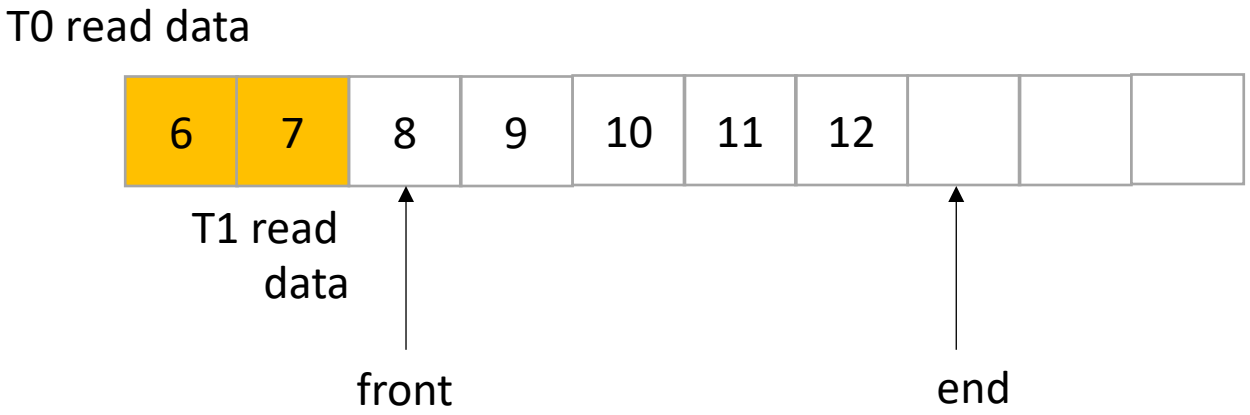
What happens if a thread wants to add an element?

Think concurrently:

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

What about Input?

- Now we only do deqs



Thread 0:
deq(); // reads 6

Thread 1:
deq(); // reads 7

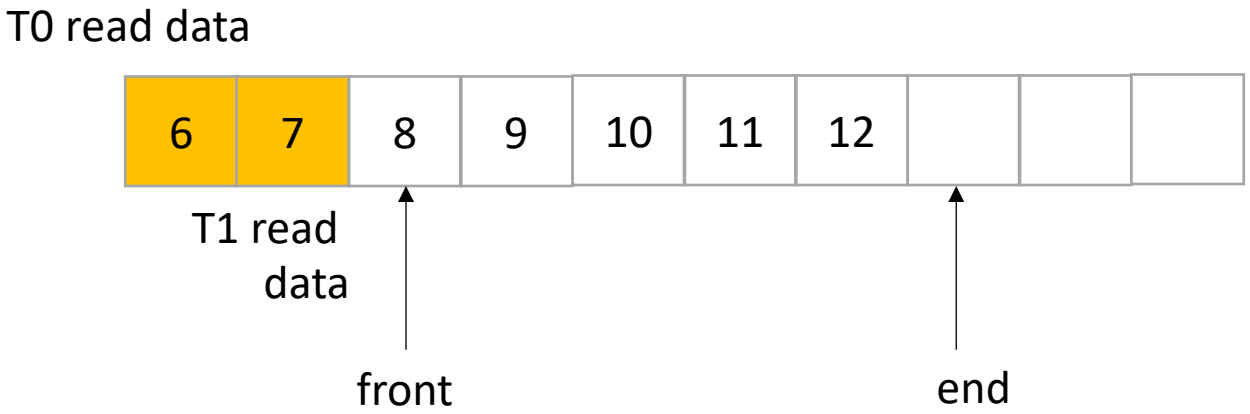
What happens if a thread wants to add an element?

Think concurrently:

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

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?