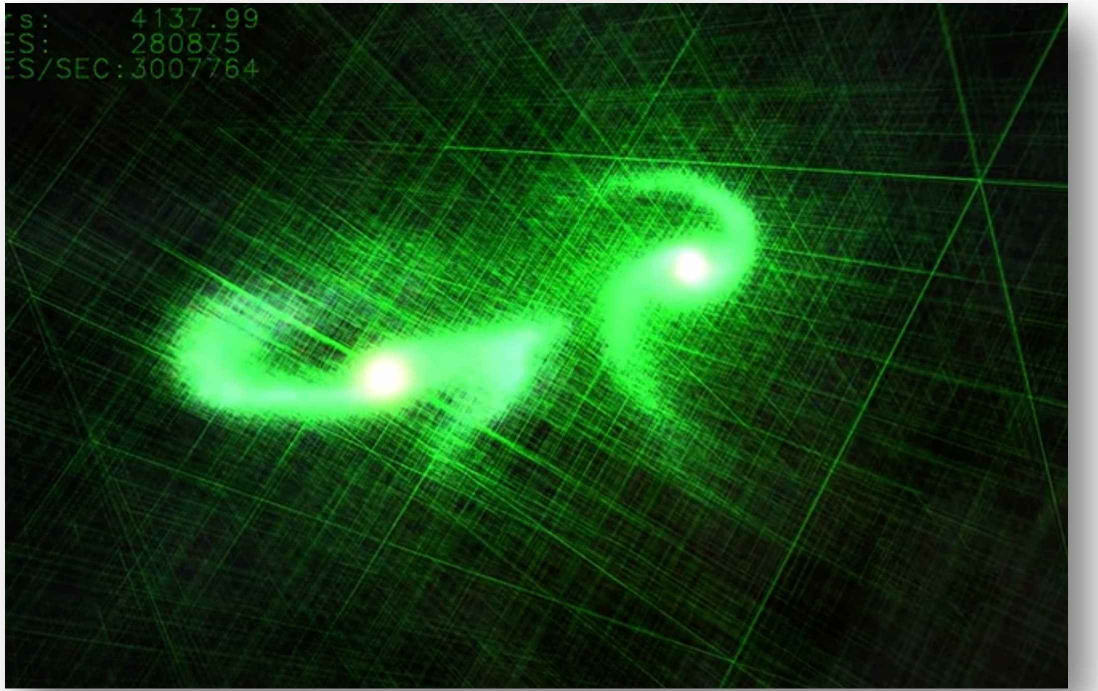
# CSE113: Parallel Programming

April 27, 2021

- **Topic**: Concurrent Objects
  - Motivation
  - Bank Account Example
  - Specification
    - Sequentially consistent
    - Linearizability




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

# Announcements

- Homework was due
  - we are going to start grading, I will keep you posted about ETA for grades
- New homework posted
  - Benchmarking questions; don't share timing until next week
  - Bonus questions for those looking for extra
- Office hours are as advertised this week

# Announcements

- Midterm assigned on Thursday
    - It will provided both as a MS word document and PDF
    - Your submission should be a PDF
    - My suggestion:
        - complete using a combination of a word processor and some problems using pencil/paper.
    - Make sure to give yourself time to juggle both homework and midterm!

# Announcements

- Poll, mid class break:
  - Do we want a 5 minute break in the middle of class?

# Announcements

- Speaking of polls:
  - There seems to be some cases where students are only logging in for the attendance points.
  - Please don’t do this.
  - It is a small portion of your grade. You get 2 excused absences in the quarter
  - If we continue to see inconsistent patterns we will move to a more accurately attendance mechanism.

# Quiz

- If you aren’t planning on staying for the whole lecture, don’t submit the quiz.

- Don’t submit the quiz if you are not listening to the lecture live.

# Quiz

- Discuss answers

# Lecture schedule

- Concurrent object motivation
- Concurrent object example with bank account
- Concurrent object specifications
  - sequential specification
  - concurrent specification - sequential consistency

# Lecture schedule

- **Concurrent object motivation**
- Concurrent object example with bank account
- Concurrent object specifications
  - sequential specification
  - concurrent specification - sequential consistency

# 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

```
//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;
}
```

simple example:
We can understand this!

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
  - Compossible

# 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
  - Compossible
- 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

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

**What can go wrong?**

We end up with duplicates

eggs
carrots
tortillas


Consider two people splitting the work.

# Conceptual examples

- Shopping list: Going shopping with roommates

**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

eggs
carrots
tortillas


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!

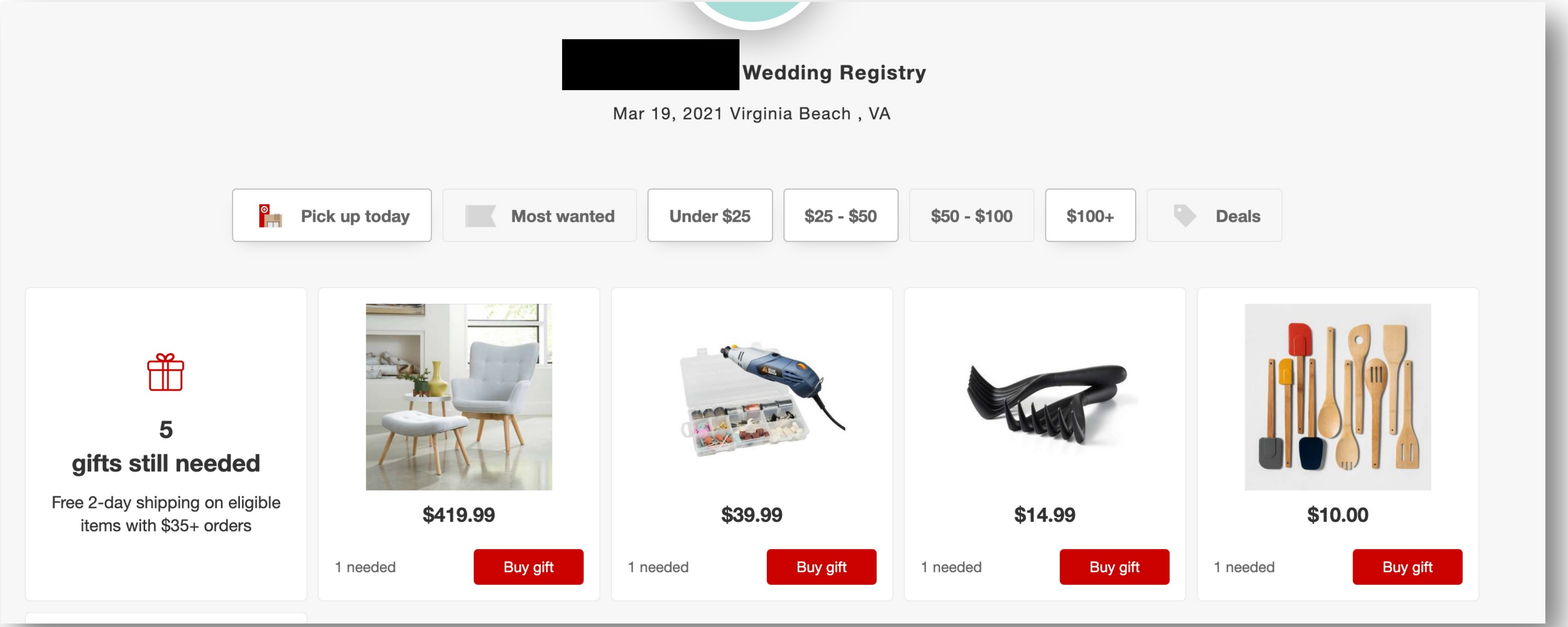

Consider two people splitting the work.

# 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
```

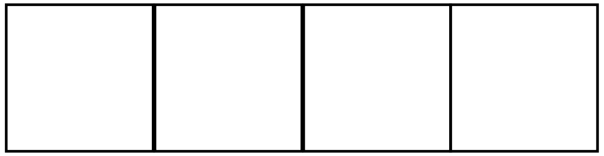
# Shared memory concurrent objects

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

*How does it actually work?*

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

concurrent queue



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

./a.out

terminal display

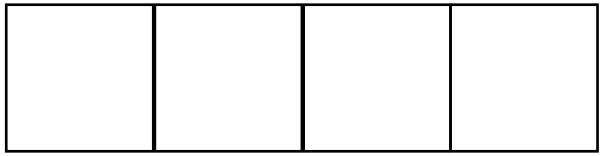
# Shared memory concurrent objects

```
printf(“hello world\n”);
```

*How does it actually work?*

```
printf(“h”);
printf(“e”);
printf(“l”);
printf(“l”);
printf(“o”);
```

concurrent queue



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

./a.out

terminal display

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

# Shared memory concurrent objects

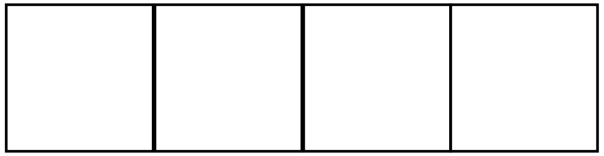
```
printf(“hello world\n”);
```

Show example

*How does it actually work?*

```
printf(“h”);
printf(“e”);
printf(“l”);
printf(“l”);
printf(“o”);
```

concurrent queue



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

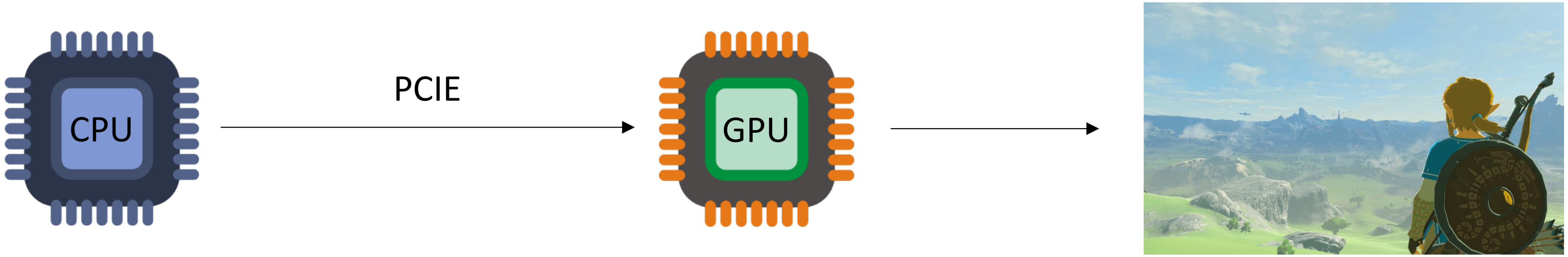
./a.out

terminal display

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

# Shared memory concurrent objects

- Graphics programming



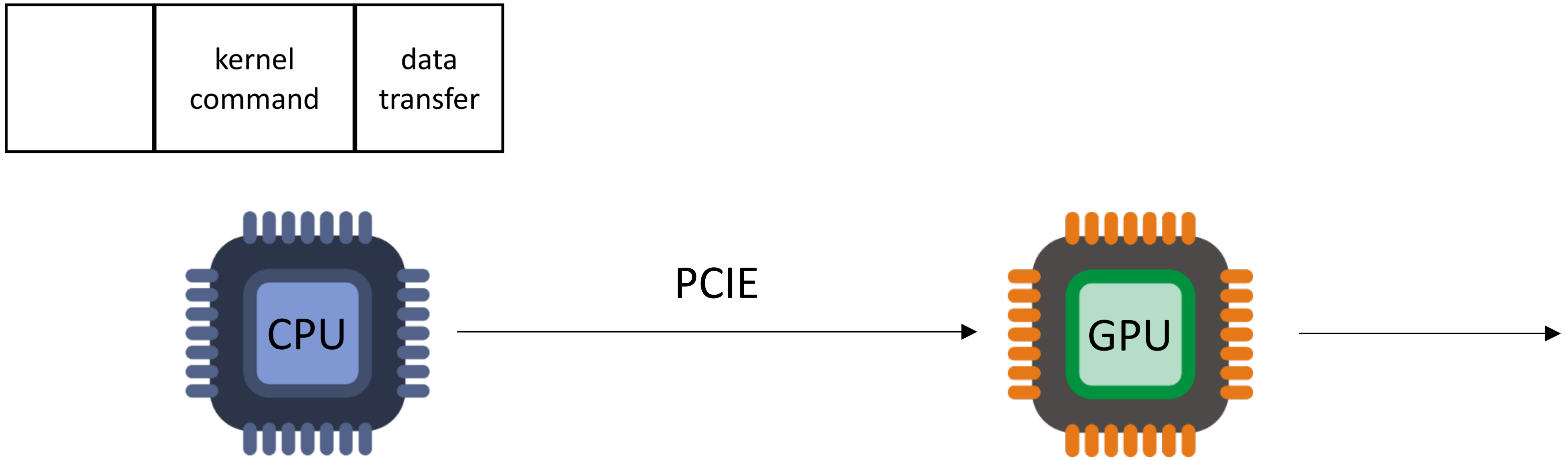

*loop:*
update data (data transfer)
graphics computation (kernel)

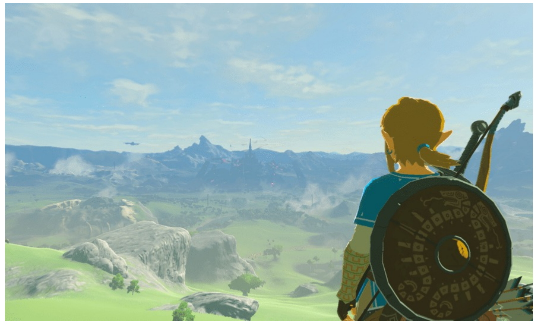
Nintendo: breath of the Wild

# Shared memory concurrent objects

- Graphics programming

Vulkan/OpenCL CommandQueue



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

Nintendo: breath of the Wild

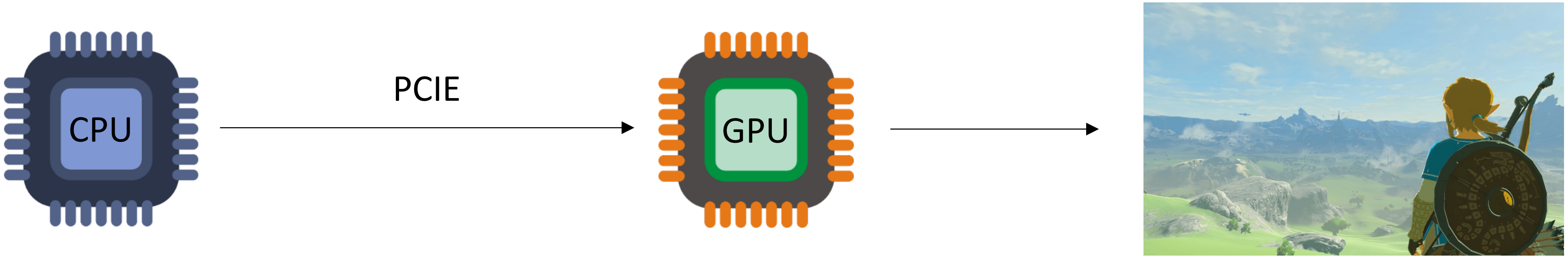
# Shared memory concurrent objects

- Graphics programming

Vulkan/OpenCL CommandQueue

| | | |
|---|---|---|
| | kernel command | data transfer |

*GPU driver concurrently reads from the queue*




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

Nintendo: breath of the Wild

# Shared memory concurrent objects

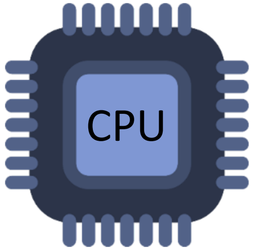
- Graphics programming

Vulkan/OpenCL CommandQueue

| | | |
|---|---|---|
| | kernel command | data transfer |

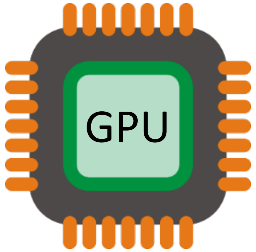
*GPU driver concurrently reads from the queue*

this concurrent queue enables an efficient graphics pipeline

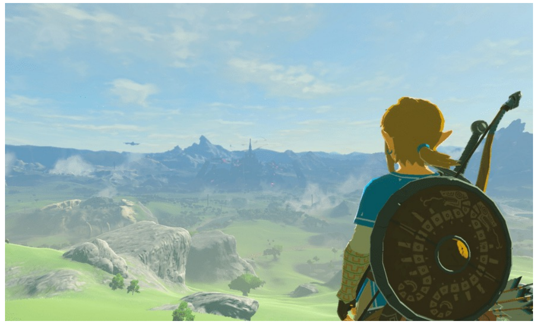



PCIE

Transferring data for scene 2




Computation for scene 1



Scene 0

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

Nintendo: breath of the Wild

# Shared memory concurrent objects

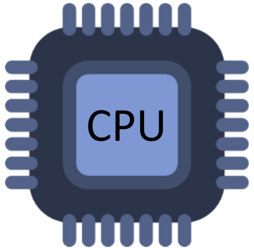
- Graphics programming

Single writer, single reader
Like in `Printf`

Vulkan/OpenCL CommandQueue

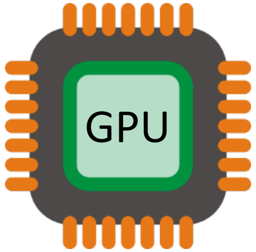
| | | |
|---|---|---|
| | kernel command | data transfer |

*GPU driver concurrently reads from the queue*

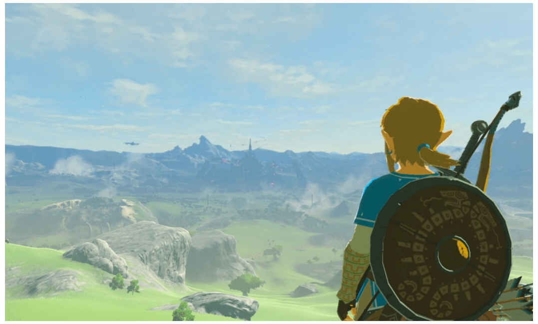



PCIE

Transferring
data for scene 2




Computation
for scene 1



Scene 0

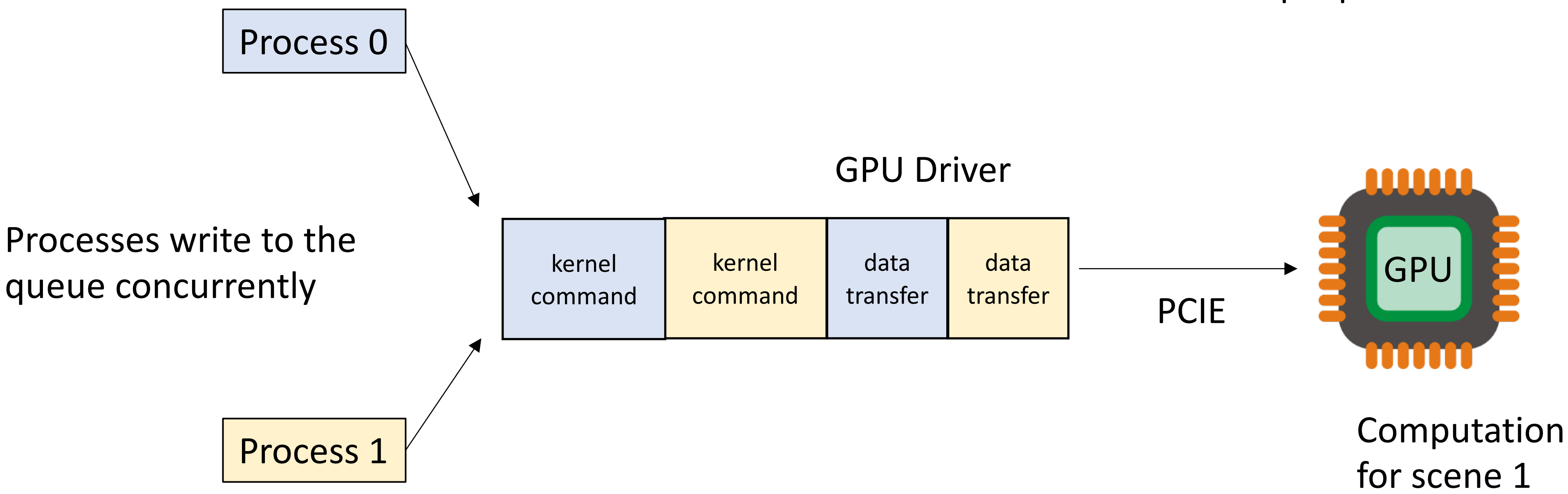
*loop:*
update data (data transfer)
graphics computation (kernel)

Nintendo: breath of the Wild

# Shared memory concurrent objects

- Graphics programming

Multiple producers




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

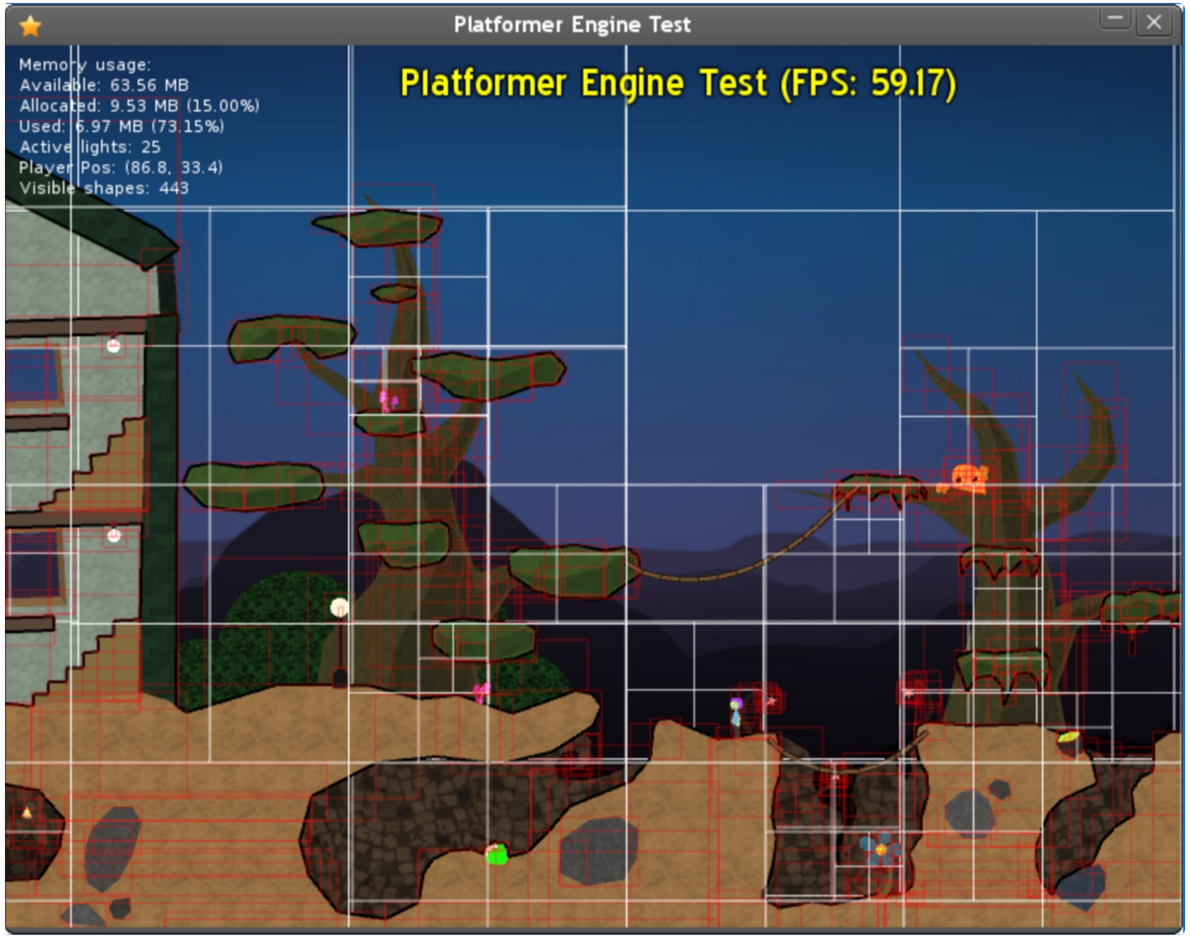
# Intro to concurrent objects

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

- They also exist in standalone applications

# Shared memory concurrent objects

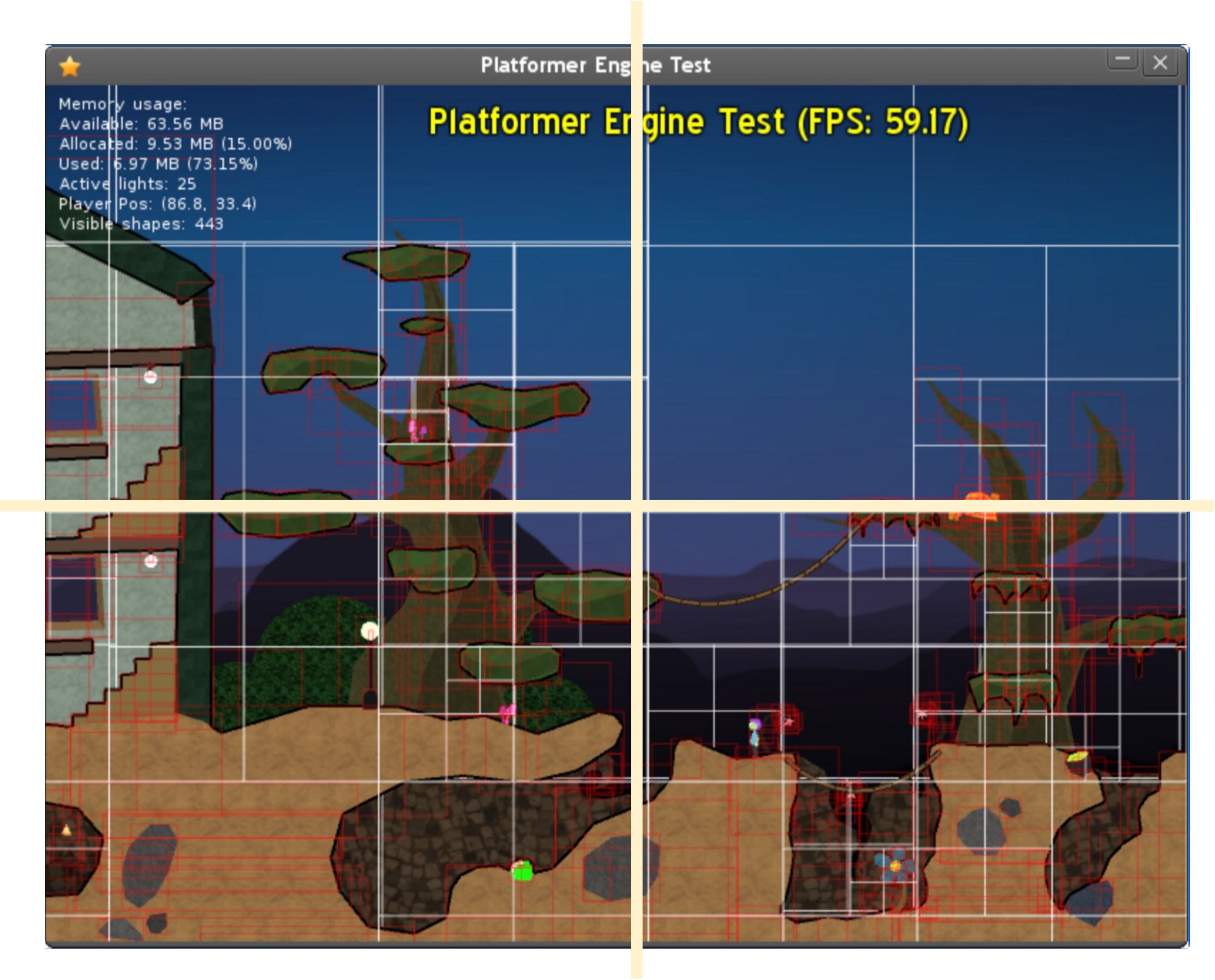
- Quadtree/Octree




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

# Shared memory concurrent objects

- Quadtree/Octree



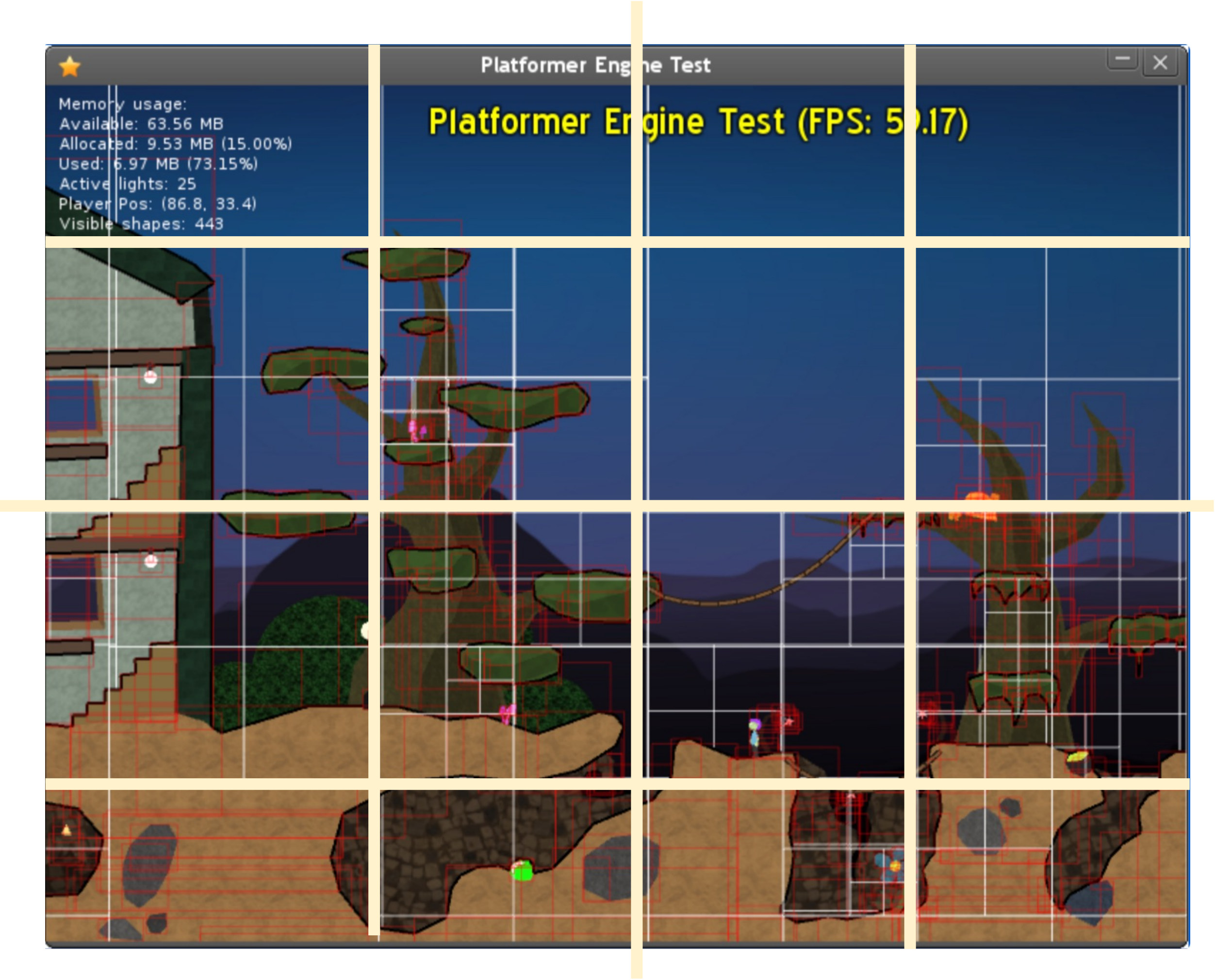

recursively divide
the scene giving more
detail to “interesting”
areas

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

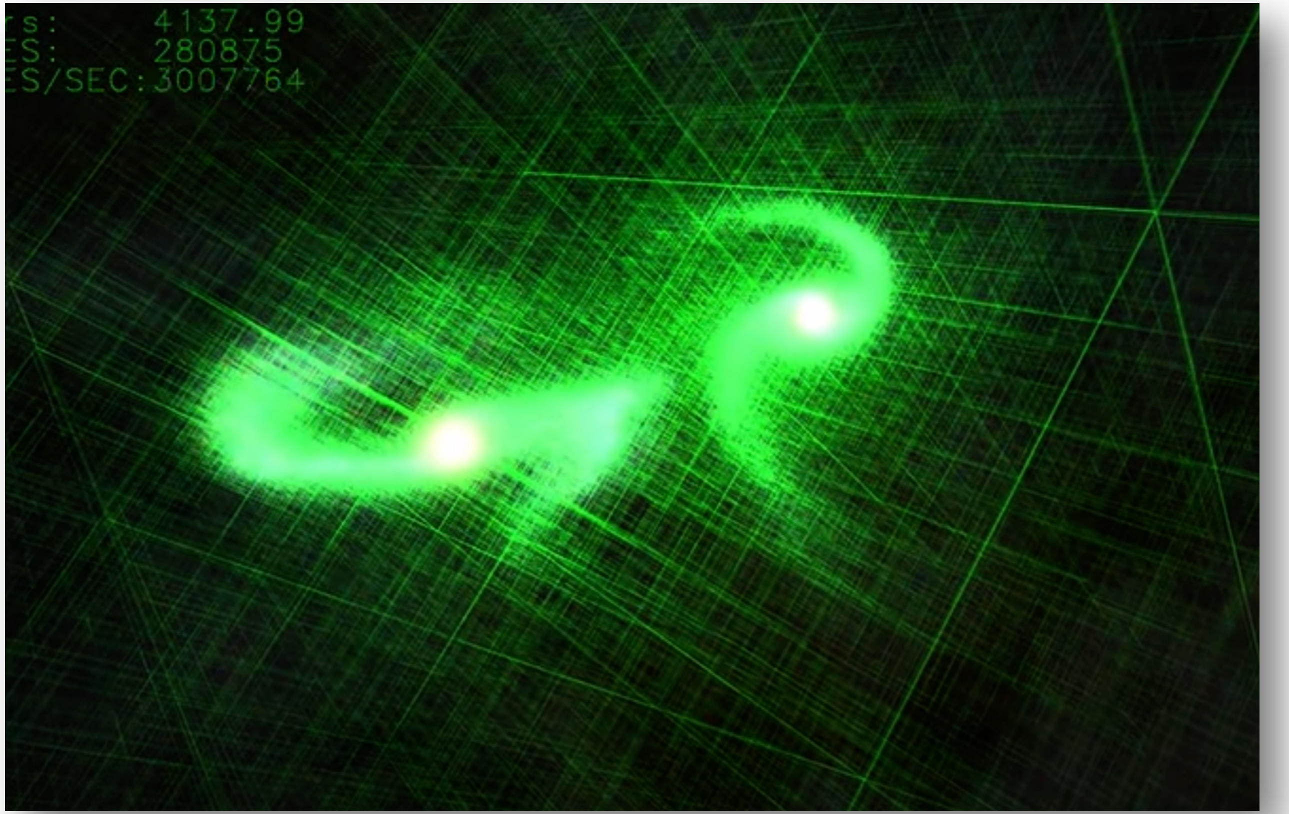
# Octree example

- From GTC 2012 (almost 10 years ago)
  - Simulation of 2 galaxies colliding
  - 280K stars


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

# Octree example

- From GTC 2012 (almost 10 years ago)
  - Simulation of 2 galaxies colliding
  - 280K stars




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

# Lecture schedule

- Concurrent object motivation
- Concurrent object example with bank account
- Concurrent object specifications
  - sequential specification
  - concurrent specification - sequential consistency

# Lecture schedule

- Concurrent object motivation
- **Concurrent object example with bank account**
- Concurrent object specifications
  - sequential specification
  - concurrent 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

We might decide to wrap my bank account in an object

*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

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

    void buy_coffee() {
      balance -= 1;
    }

    void get_paid() {
      balance += 1;
    }

  private:
    int balance;
};
```

# Bank account example

We might decide to wrap my bank account in an object

*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!

```
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 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
}
```

# Recall atomic RMWs cannot interleave

*Buying coffee*

```
atomic_fetch_add(&account, -1);
```

*Getting paid*

```
atomic_fetch_add(&account, 1);
```

time

time

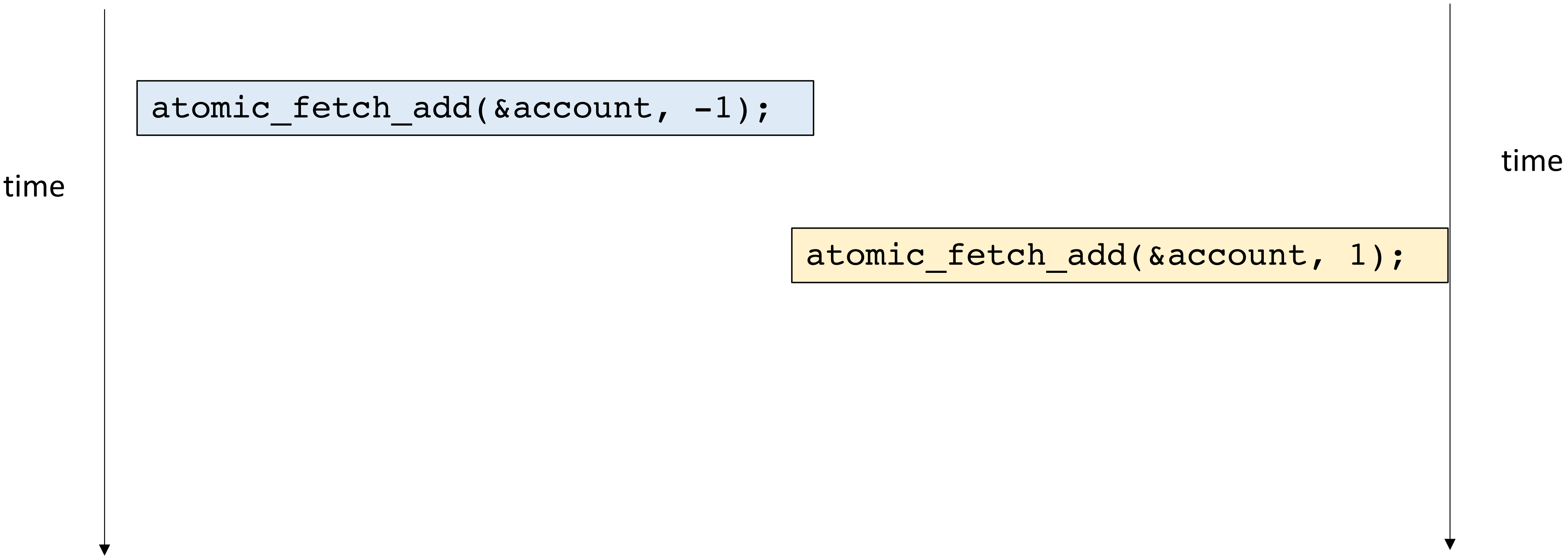
# Recall atomic RMWs cannot interleave

*Buying coffee*

```
atomic_fetch_add(&account, -1);
```

*Getting paid*

```
atomic_fetch_add(&account, 1);
```



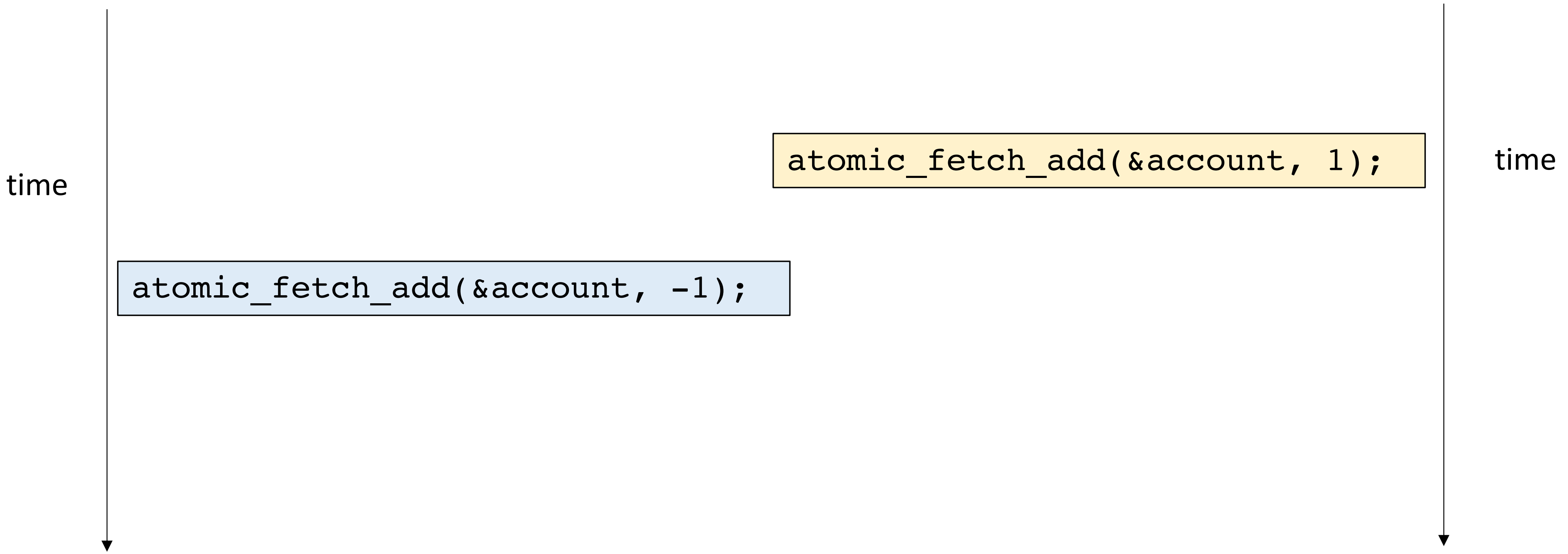
# Recall atomic RMWs cannot interleave

*Buying coffee*

```
atomic_fetch_add(&account, -1);
```

*Getting paid*

```
atomic_fetch_add(&account, 1);
```



# Bank account example

*global variables:*

```
bank_account tylers_account;
```

*Tyler's coffee addiction:*

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

*Tyler's employer*

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

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

    void buy_coffee() {
      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(&account, -1);
    }

    void get_paid() {
      atomic_fetch_add(&account, 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
- **Progress**:
  - What do we expect from the OS scheduler?
  - Under what conditions can concurrent objects deadlock
- **Performance**:
  - How to make things fast fast fast!

# Lecture schedule

- Concurrent object motivation

- Concurrent object example with bank account

- Concurrent object specifications
  - sequential specification
  - concurrent specification - sequential consistency

# Lecture schedule

- **Concurrent object motivation**

- Concurrent object example with bank account

- **Concurrent object specifications**
  - sequential specification
  - concurrent specification - sequential consistency

# 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();
```

# 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: *None*

# 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&lt;int&gt; q;
```

*Thread 0:*

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

*Thread 1:*

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

what can be stored in t after this concurrent program?

Can t be 256?

# Thinking about a concurrent queue

*Global variable:*

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

*Thread 0:*

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

*Thread 1:*

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

what can be stored in t after this concurrent program?

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

*Global variable:*

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

*Thread 0:*

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

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

*Thread 1:*

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

*Global variable:*

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

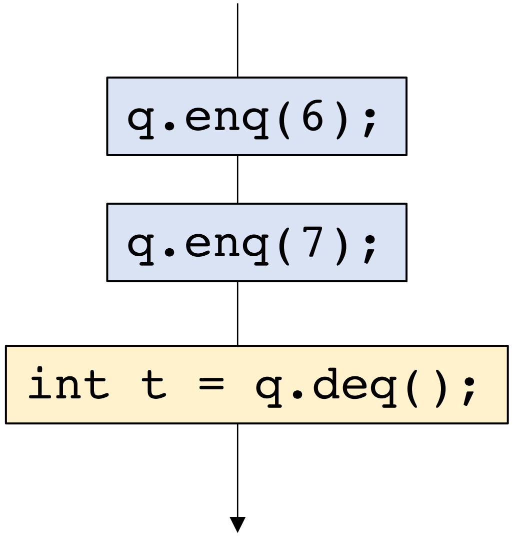
*Thread 0:*

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

*Thread 1:*

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

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




t is 6

*Global variable:*

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

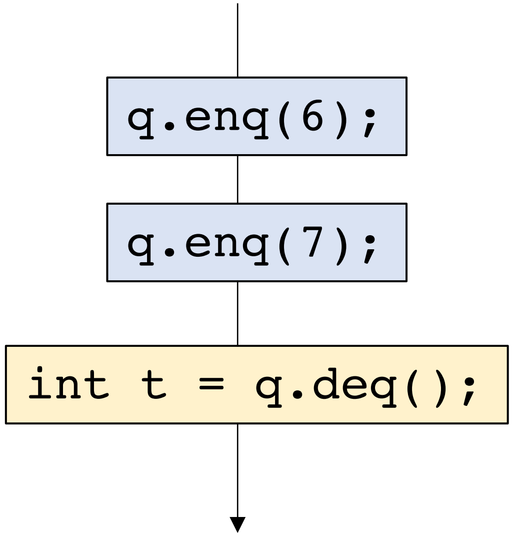
*Thread 0:*

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

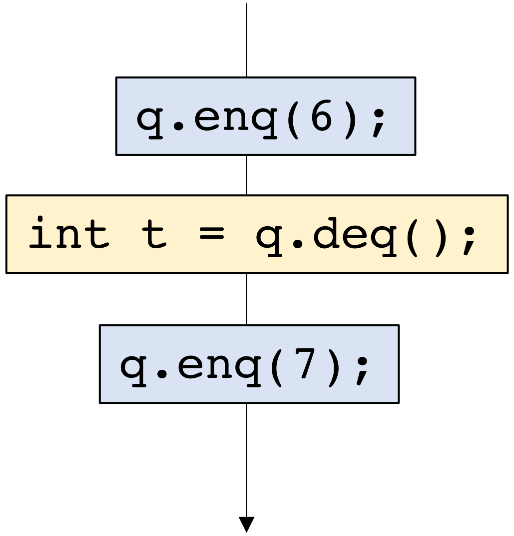
*Thread 1:*

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

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




t is 6




t is 6

*Global variable:*

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

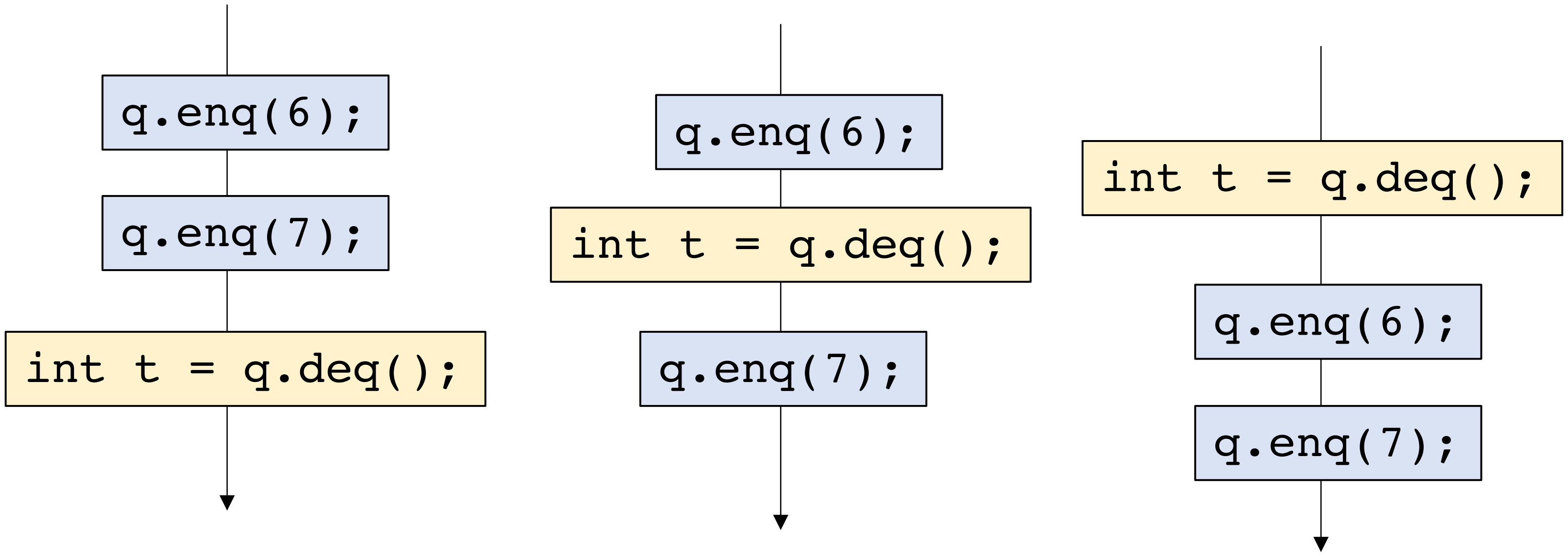
*Thread 0:*

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

*Thread 1:*

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

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




t is 6    t is 6    t is None

*Global variable:*

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

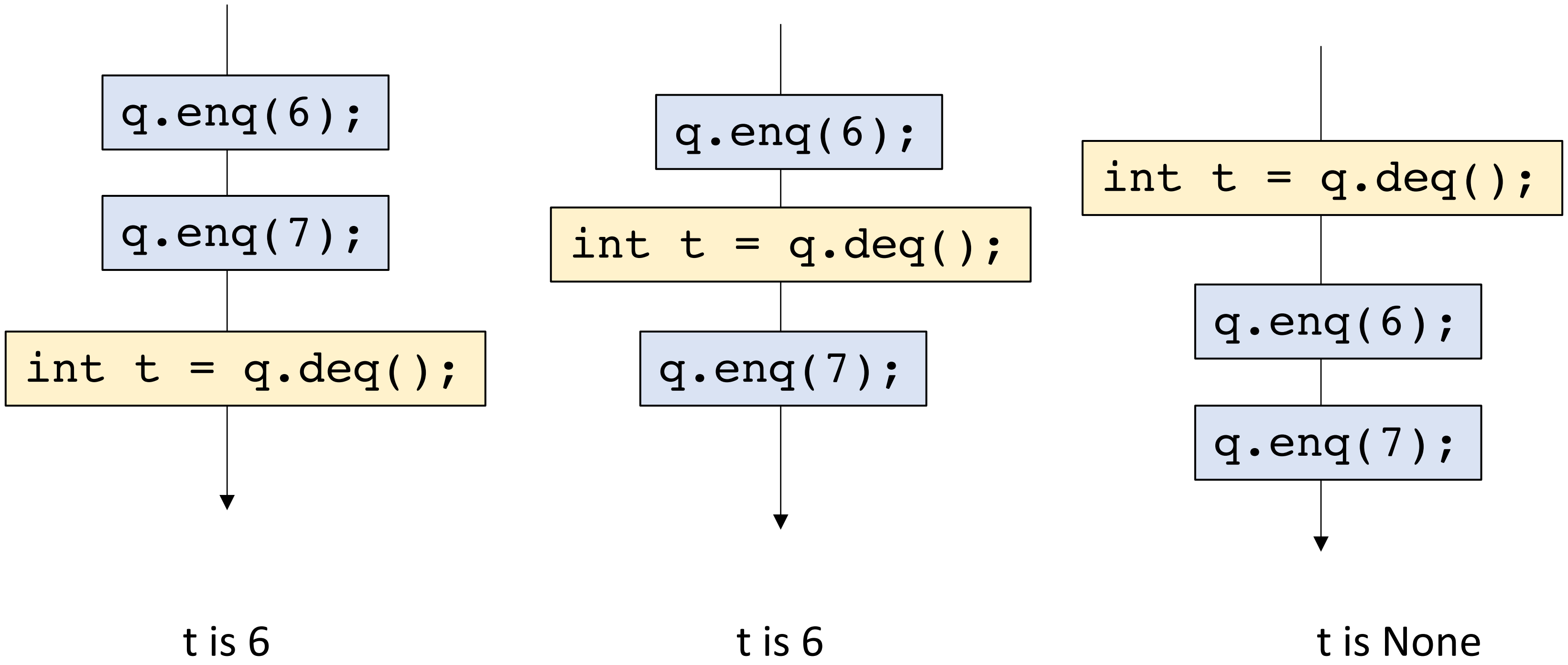
*Thread 0:*

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

*Thread 1:*

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

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




*Can t ever be 7?*

*Global variable:*

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

*Thread 0:*

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

*Thread 1:*

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

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

*Can t ever be 7?*

*Global variable:*

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

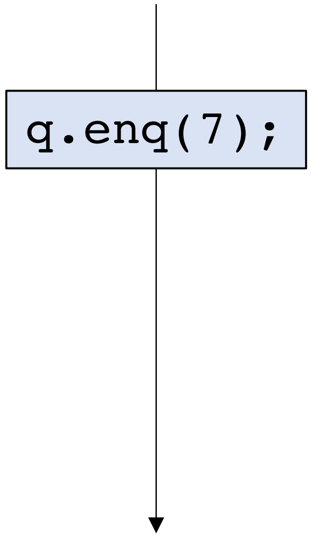
*Thread 0:*

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

*Thread 1:*

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

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




*Can t ever be 7?*

*Global variable:*

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

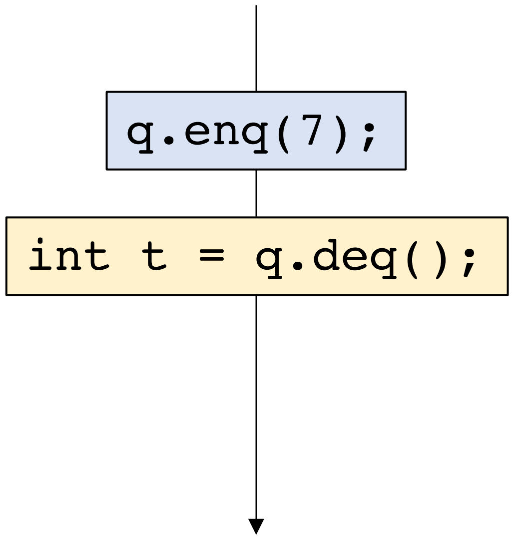
*Thread 0:*

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

*Thread 1:*

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

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




*Can t ever be 7?*

*Global variable:*

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

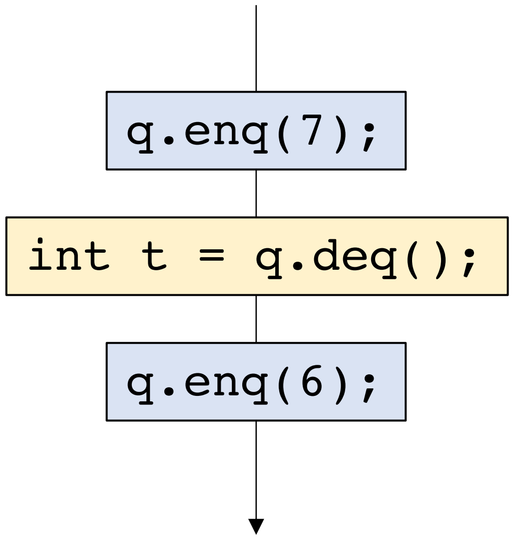
*Thread 0:*

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

*Thread 1:*

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

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




*Can t ever be 7?*

*Global variable:*

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

*Thread 0:*

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

*Thread 1:*

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

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

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

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

*This should not be allowed!*

*Can t ever be 7?*

# Sequential Consistency

- Valid executions correspond a sequentialization of object method
- 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
- The sequentialization must respect per-thread ”program order”, the order in which the object method calls occur in the thread
- Events across threads can interleave in any way possible

How many possible interleavings?
Combinatorics question:

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

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

# Sequential Consistency

How many possible interleavings?
Combinatorics question:

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

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

Reminder that N and M are events, not instructions

# Sequential Consistency

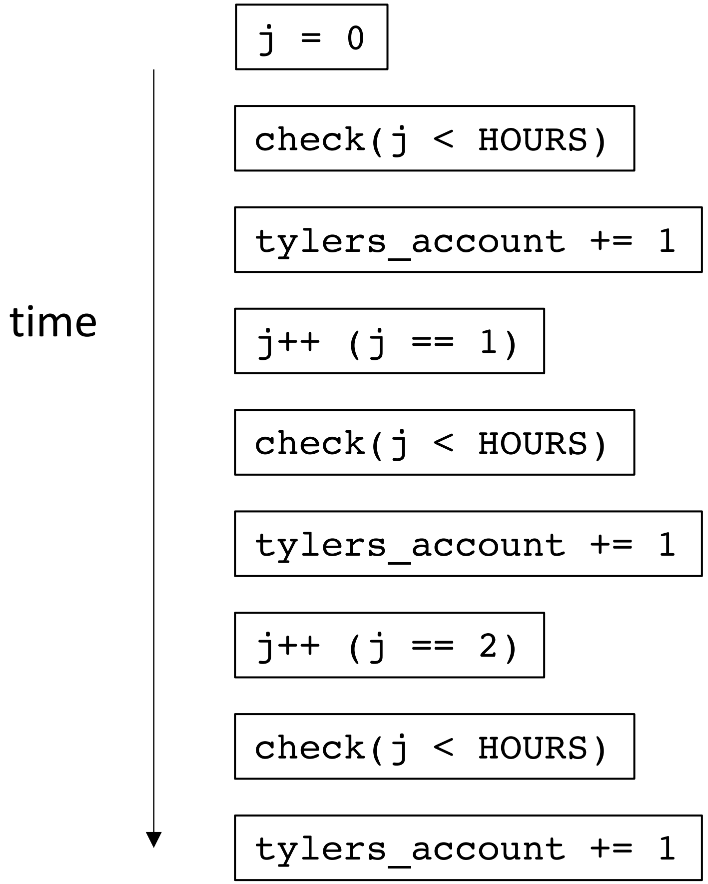
*Tyler's employer*

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

How many possible interleavings?
Combinatorics question:

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

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




Reminder that N and M are events, not instructions

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

# Don’t think about all possible interleavings!

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

# Reasoning about concurrent objects

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

*Global variable:*

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

*Thread 0:*

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

*Thread 1:*

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

Can t0 == 0 and t1 == 6?

*Global variable:*

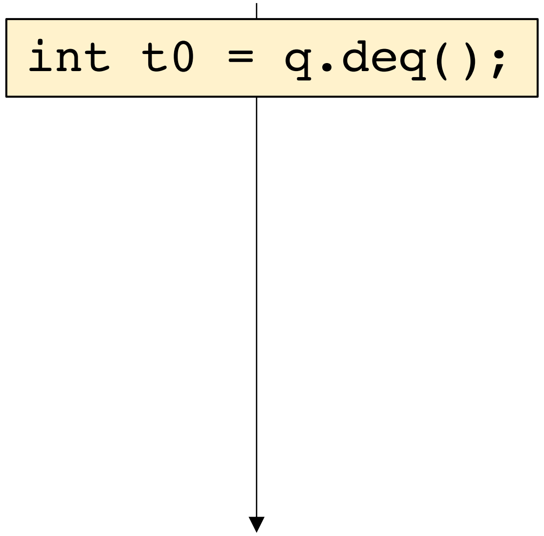
```
CQueue<int> q;
```

*Thread 0:*

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

*Thread 1:*

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




Can t0 == 0 and t1 == 6?

*Global variable:*

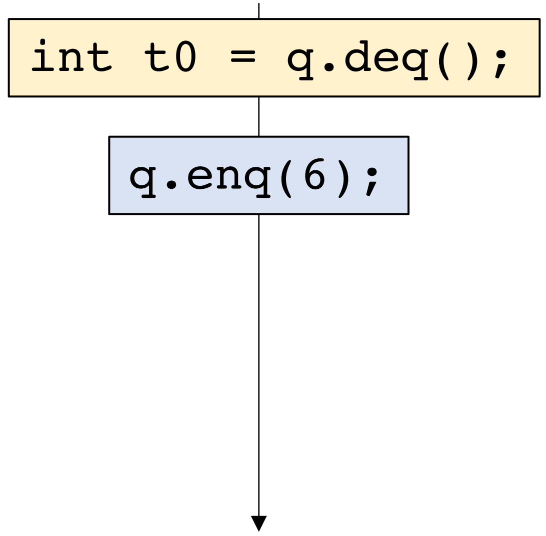
```
CQueue<int> q;
```

*Thread 0:*

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

*Thread 1:*

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




Can t0 == 0 and t1 == 6?

*Global variable:*

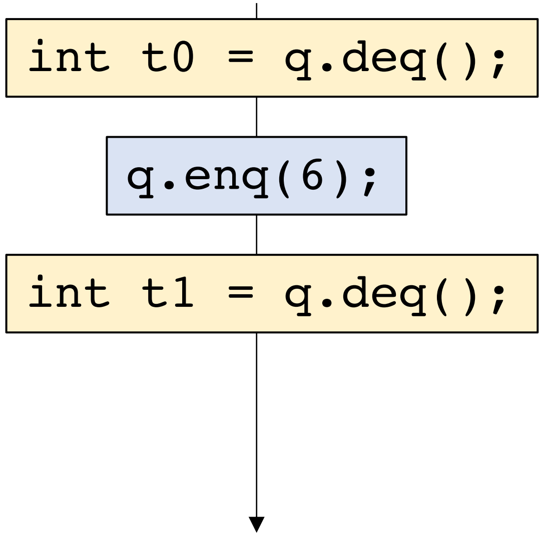
```
CQueue<int> q;
```

*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();
```

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

Can t0 == 0 and t1 == 6?

Valid execution!

Are there others?

*Global variable:*

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

Lets do another!

*Thread 0:*

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

*Thread 1:*

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

Can t0 == 6 and t1 == 7?

*Global variable:*

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

*Thread 0:*

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

*Thread 1:*

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

Can t0 == 6 and t1 == 7?

*Global variable:*

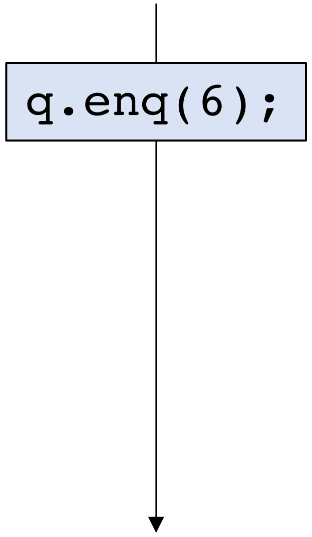
```
CQueue<int> q;
```

*Thread 0:*

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

*Thread 1:*

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




Can t0 == 6 and t1 == 7?

*Global variable:*

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

*Thread 0:*

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

*Thread 1:*

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

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

Can t0 == 6 and t1 == 7?

*Global variable:*

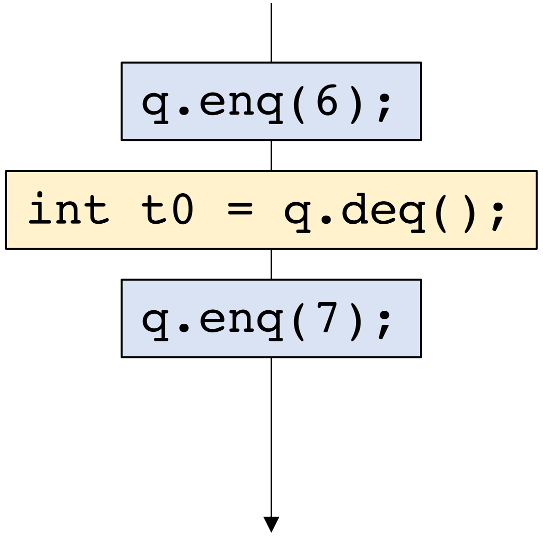
```
CQueue<int> q;
```

*Thread 0:*

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

*Thread 1:*

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




Can t0 == 6 and t1 == 7?

*Global variable:*

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

*Thread 0:*

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

*Thread 1:*

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

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

Can t0 == 6 and t1 == 7?

Found one! Are there others?

# Reasoning about concurrent objects

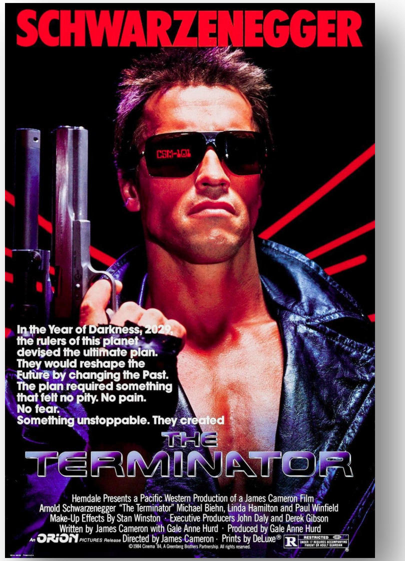
To show that an outcome is possible, simply construct the sequential sequence

To show that an outcome is ***impossible*** show that the outcome would require time travel!

# 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 the outcome would require time travel!

*Global variable:*

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

*Thread 0:*

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

*Thread 1:*

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

Can t0 == 0 and t1 == 7?

*Global variable:*

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

*Thread 0:*

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

*Thread 1:*

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

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

Can t0 == 0 and t1 == 7?

*Global variable:*

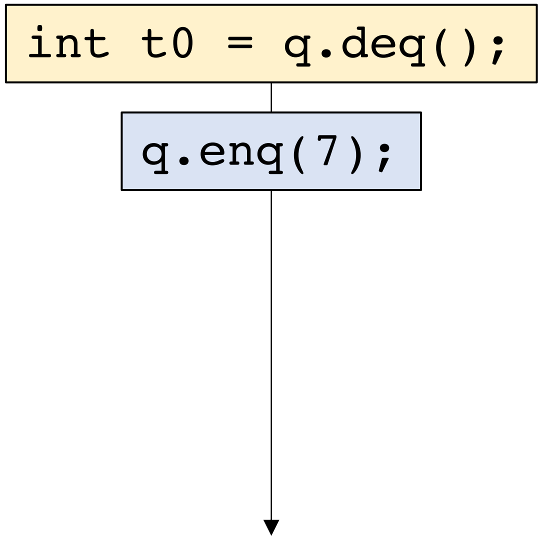
```
CQueue<int> q;
```

*Thread 0:*

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

*Thread 1:*

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




Can t0 == 0 and t1 == 7?

*Global variable:*

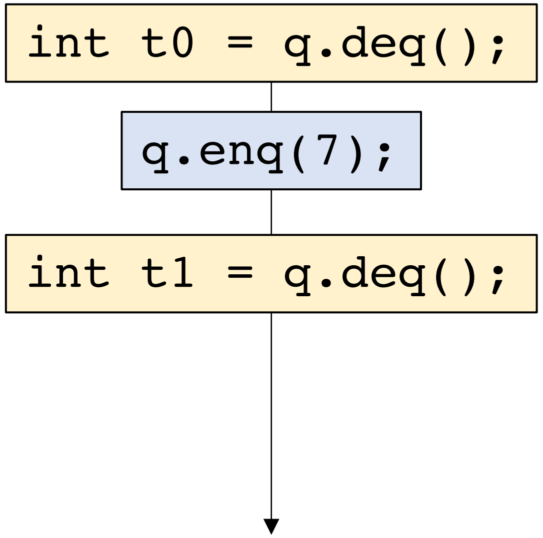
```
CQueue<int> q;
```

*Thread 0:*

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

*Thread 1:*

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




Can t0 == 0 and t1 == 7?

*Global variable:*

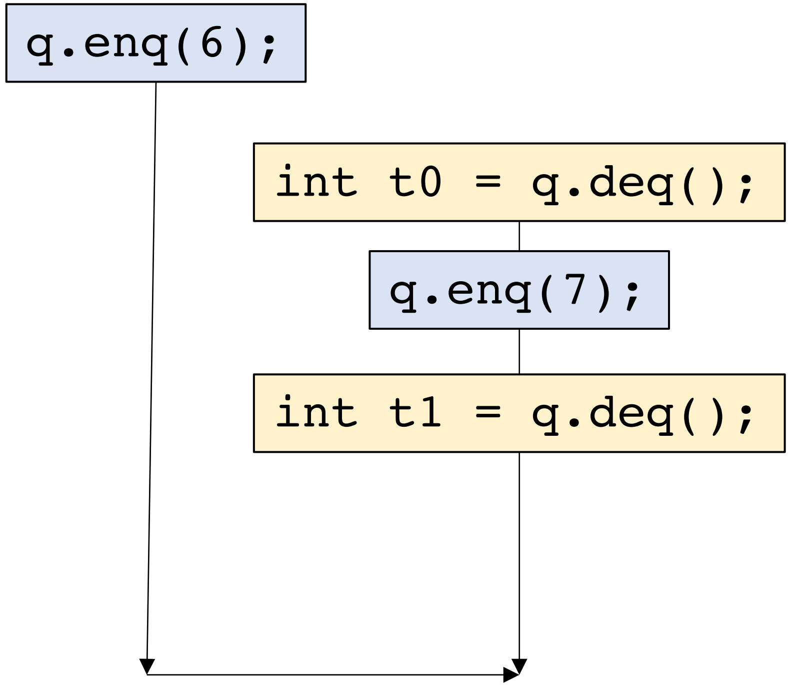
```
CQueue<int> q;
```

*Thread 0:*

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

*Thread 1:*

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



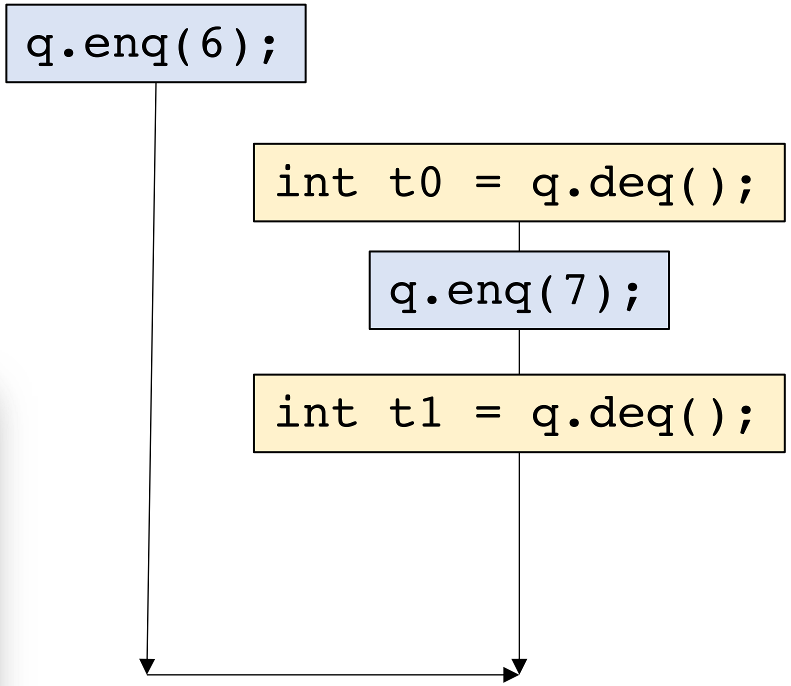

Can t0 == 0 and t1 == 7?

*Global variable:*

`CQueue<int> q;`

*Thread 0:*

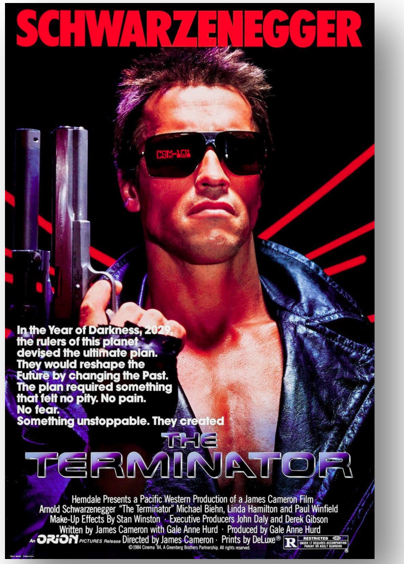
```
q.enq(6);
q.enq(7);
```




*Thread 1:*

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

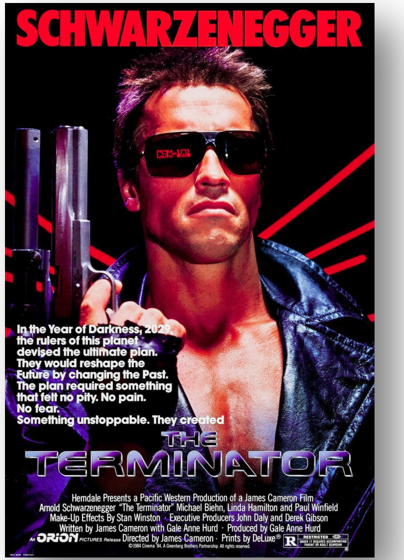
Can t0 == 0 and t1 == 7?




Time travel in our specifications should not be allowed!

# What does that cycle mean?

- Justify your current state with something you will do in the future:
  - I have my phone right now because I will give it to myself tomorrow
  - Causality cycles: The past influences the future, the future can't influence the past


Lets do one more examples

*Global variable:*

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

*Thread 0:*

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

*Thread 1:*

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

Is it possible for t0 == 6
but the queue to contain 7
after the program?

*Global variable:*

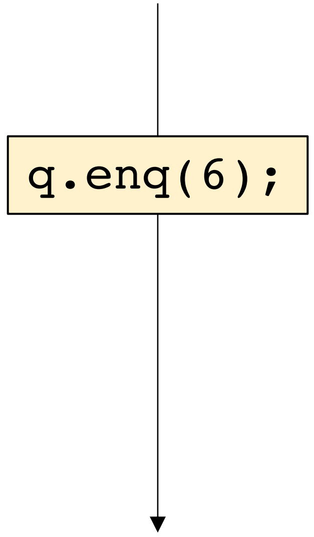
```
CQueue<int> q;
```

*Thread 0:*

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

*Thread 1:*

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




Is it possible for t0 == 6
but the queue to contain 7
after the program?

*Global variable:*

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

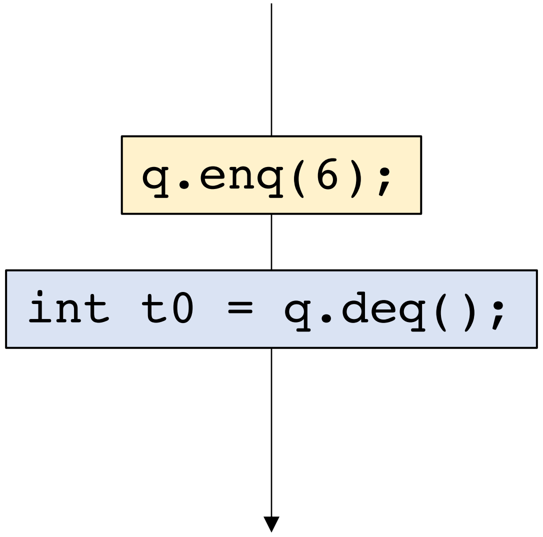
*Thread 0:*

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

*Thread 1:*

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

Is it possible for t0 == 6
but the queue to contain 7
after the program?



*Global variable:*

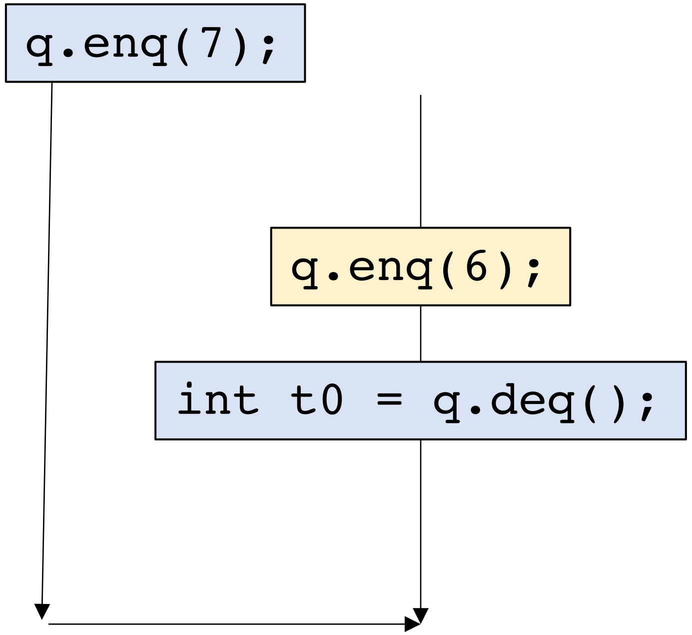
```
CQueue<int> q;
```

*Thread 0:*

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

*Thread 1:*

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




Is it possible for t0 == 6
but the queue to contain 7
after the program?

*Global variable:*

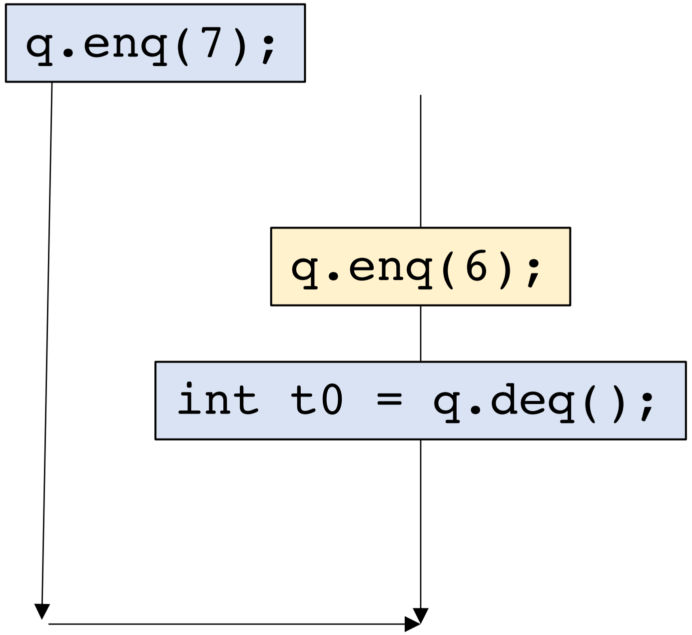
```
CQueue<int> q;
```

*Thread 0:*

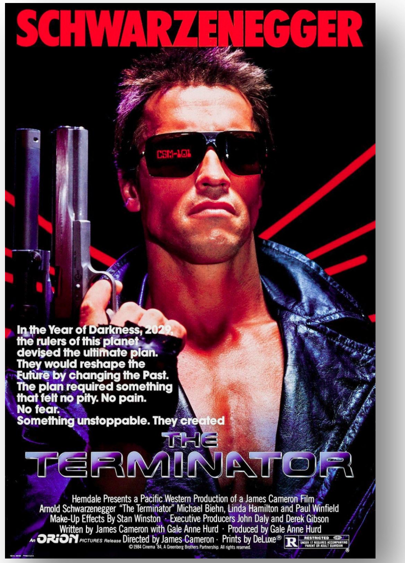
```
q.enq(7);
int t0 = q.dec();
```

*Thread 1:*

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




Is it possible for t0 == 6
but the queue to contain 7
after the program?




time travel!
not allowed!

# Do we have our specification?

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

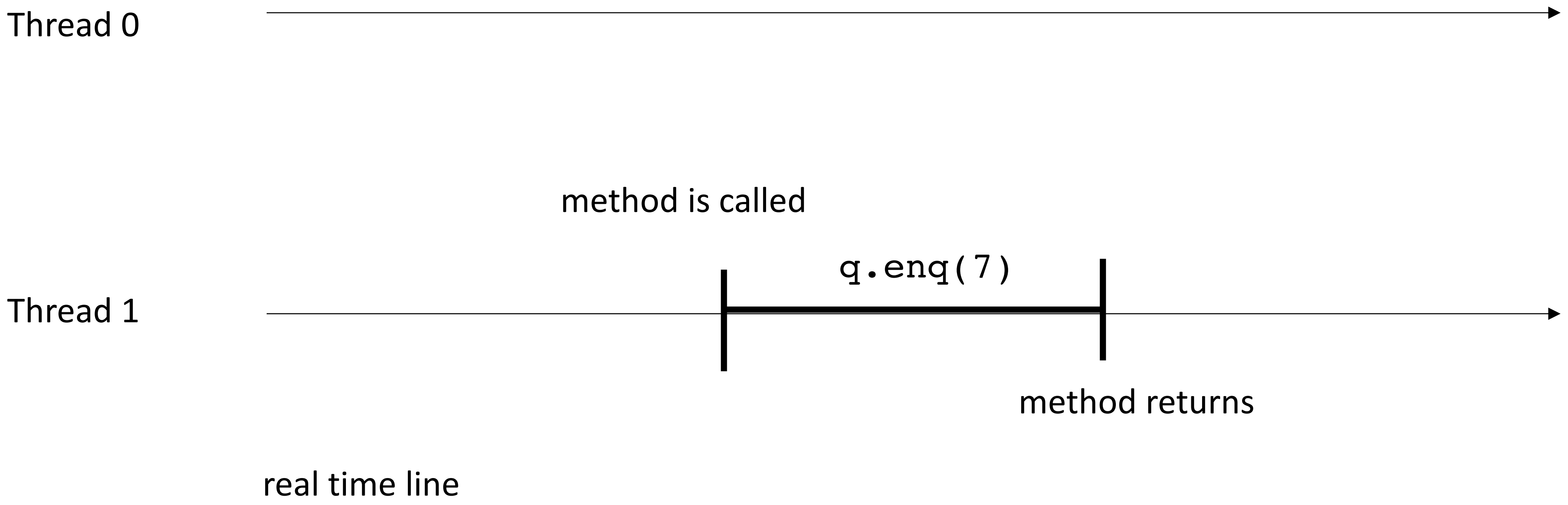
- It's a good first step, but relative timing (happens-before) interacts strangely with concrete time.

- We will need something stronger.

# Sequential consistency and real time

- Add in real time:

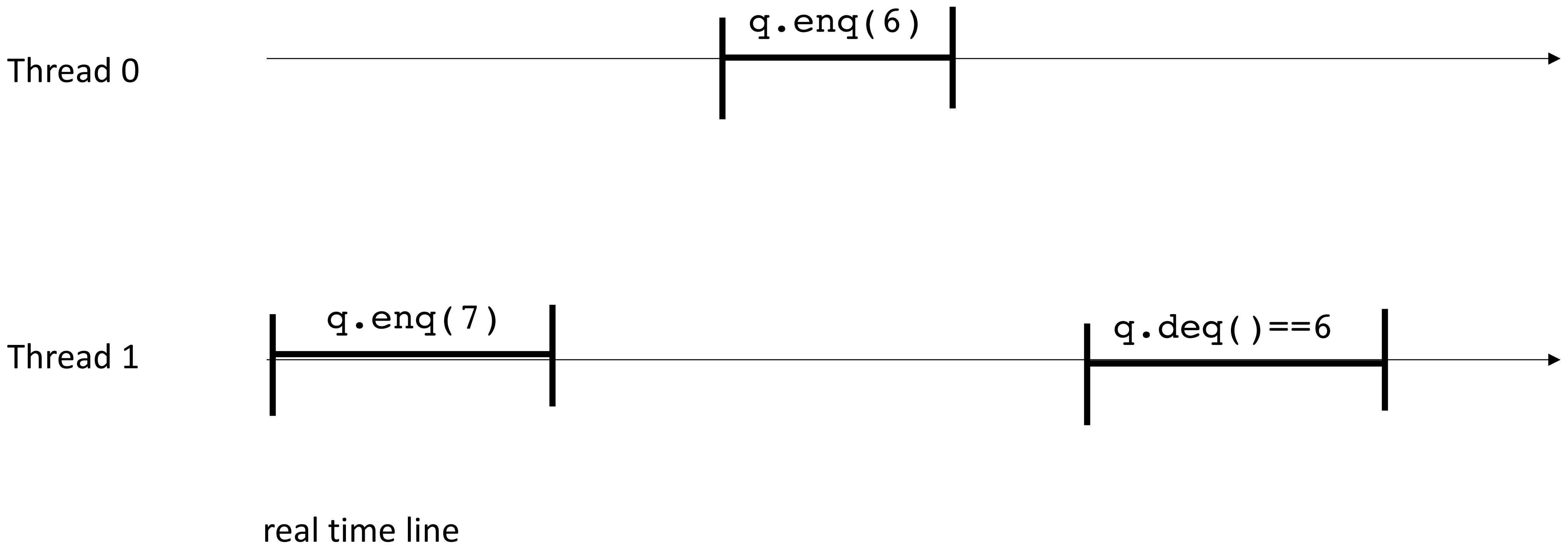
each method as a start, and end time stamp



# Sequential consistency and real time

- Add in real time:

This timeline seems strange...

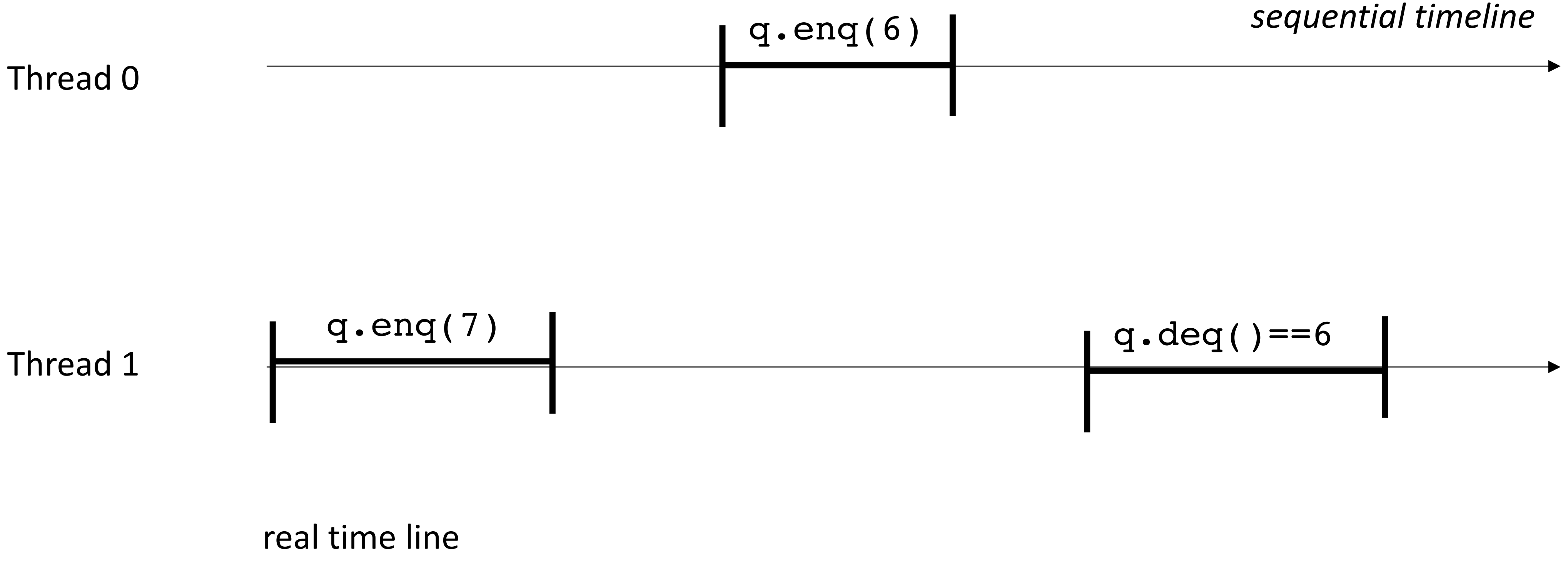


# Sequential consistency and real time

- 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*

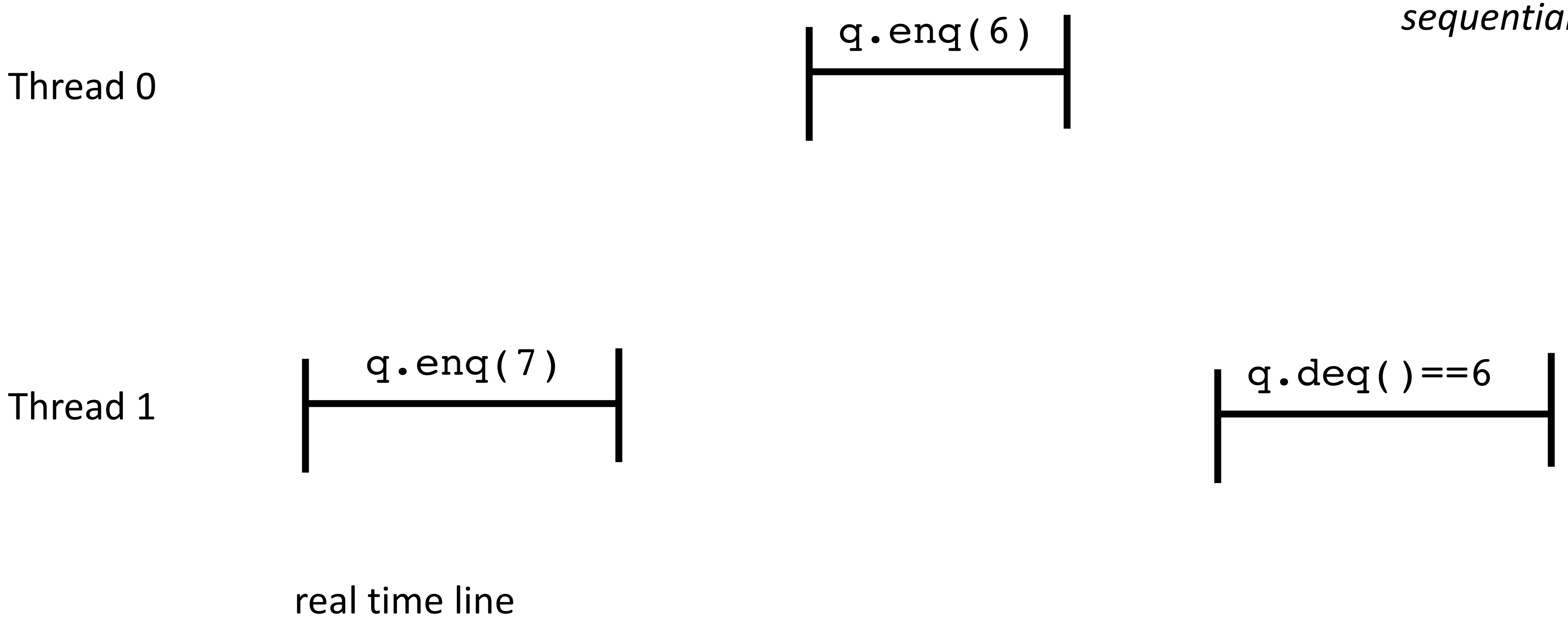


# Sequential consistency and real time

- 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*

`q.enq(6)`

Thread 0

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

Thread 1

real time line

# Sequential consistency and real time

- Add in real time:

*This execution is allowed in sequential consistency!*

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

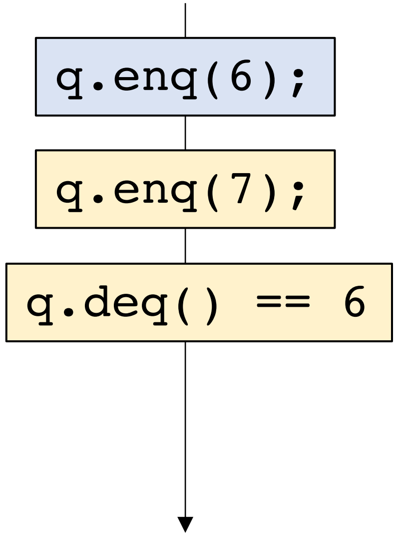
`q.enq(6)`

Thread 0

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

Thread 1

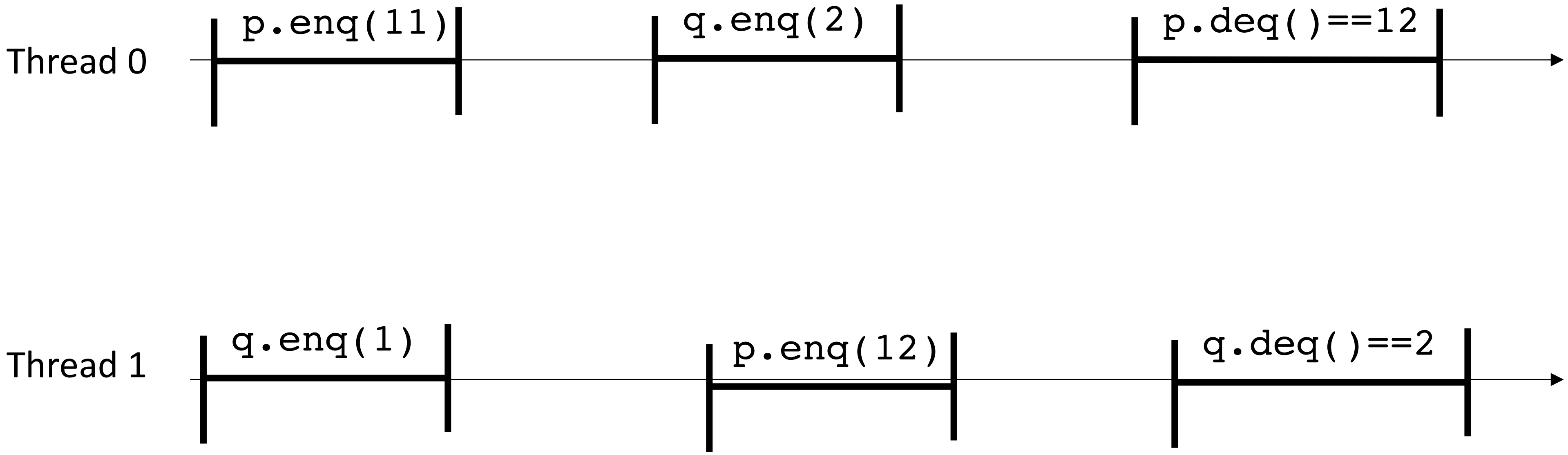
real time line



# Sequential consistency and real time

- Add in real time:

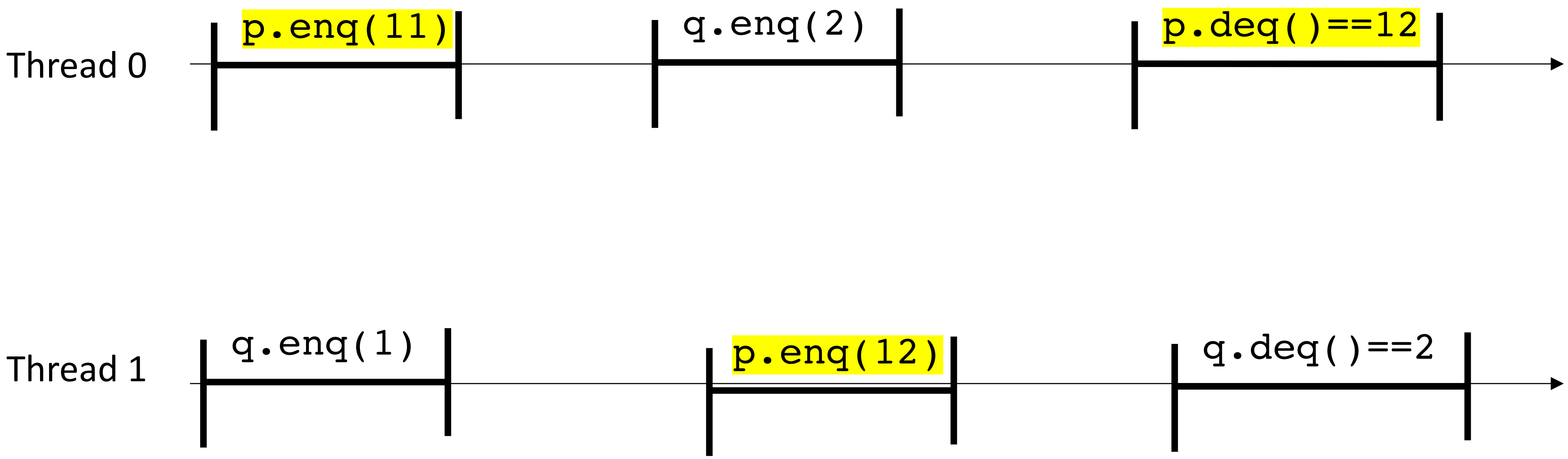
2 objects now: p and q



# Sequential consistency and real time

- Add in real time:

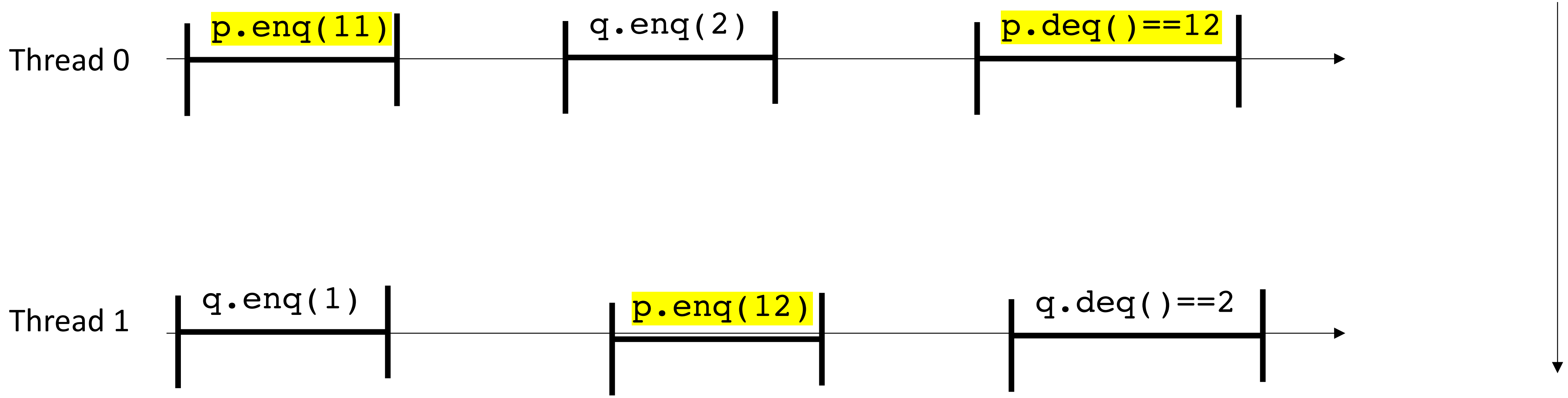
2 objects now: p and q
Consider each object in isolation



# Sequential consistency and real time

- Add in real time:

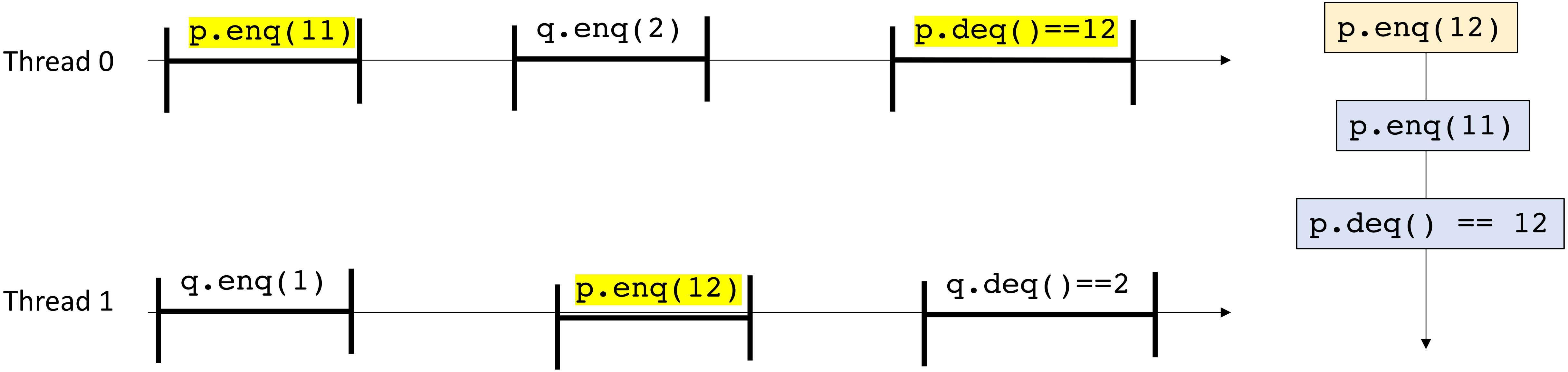
2 objects now: p and q
Consider each object in isolation



# Sequential consistency and real time

- Add in real time:

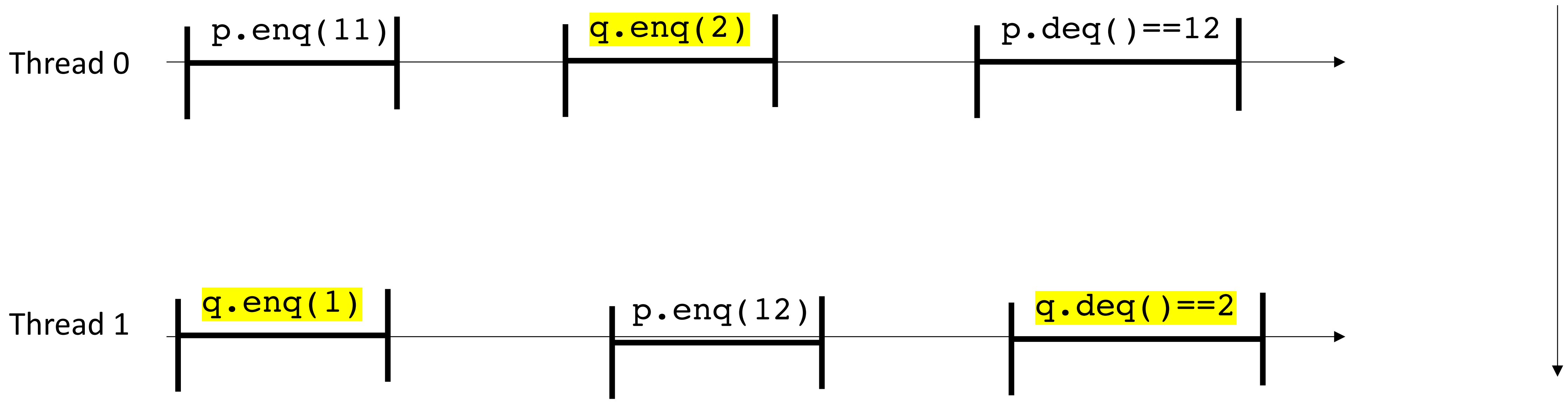
2 objects now: p and q
Consider each object in isolation



# Sequential consistency and real time

- Add in real time:

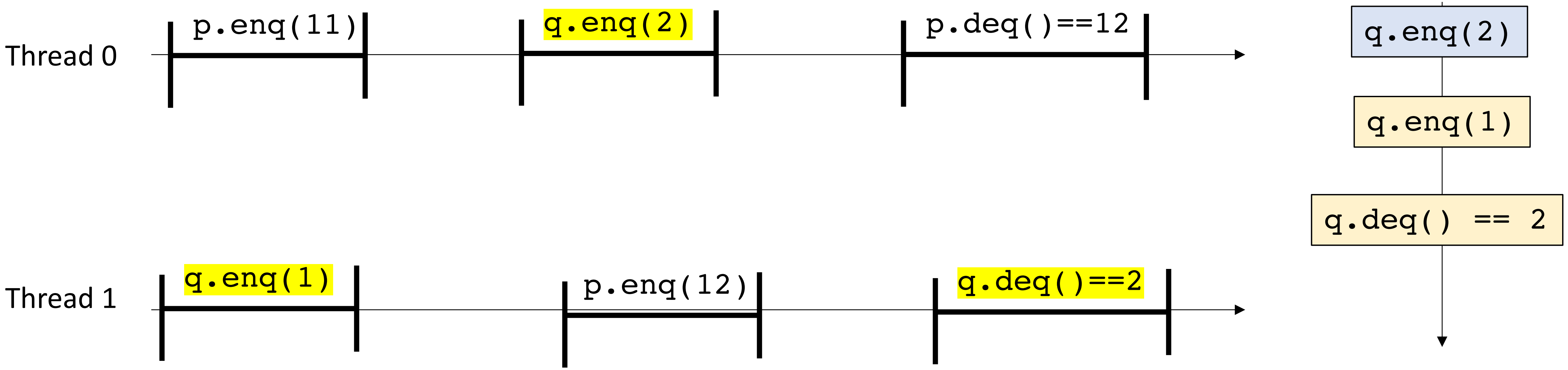
2 objects now: p and q
Consider each object in isolation



# Sequential consistency and real time

- Add in real time:

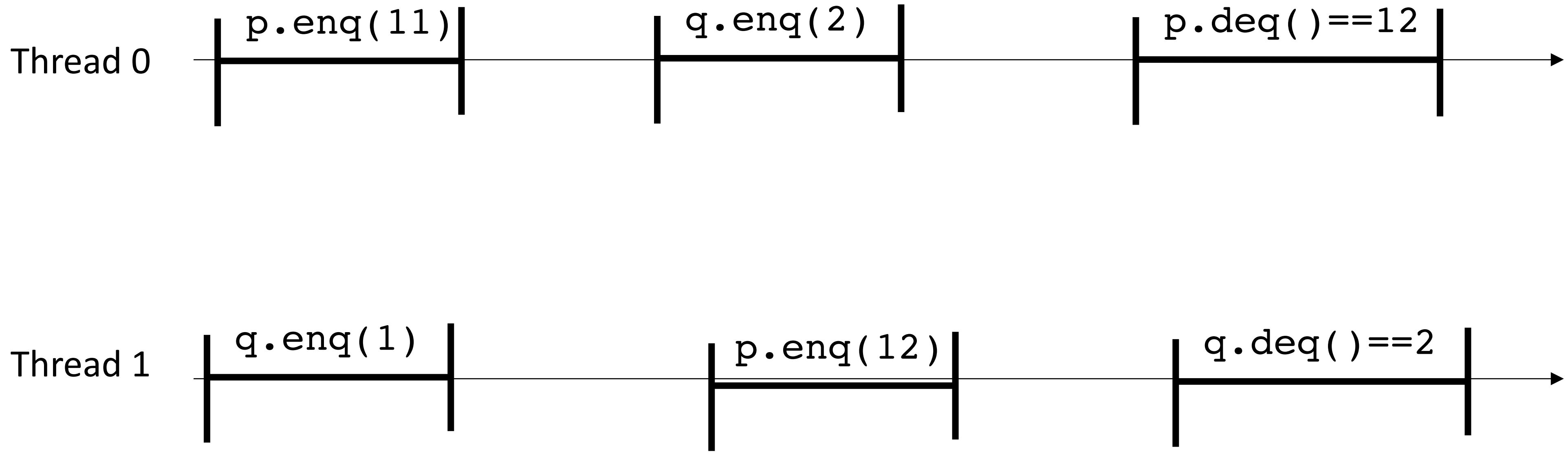
2 objects now: p and q
Consider each object in isolation



# Sequential consistency and real time

- Add in real time:

Now consider them all together



*Global variable:*

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

*Thread 0:*

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

*Thread 1:*

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

*Global variable:*

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

*Thread 0:*

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

*Thread 1:*

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

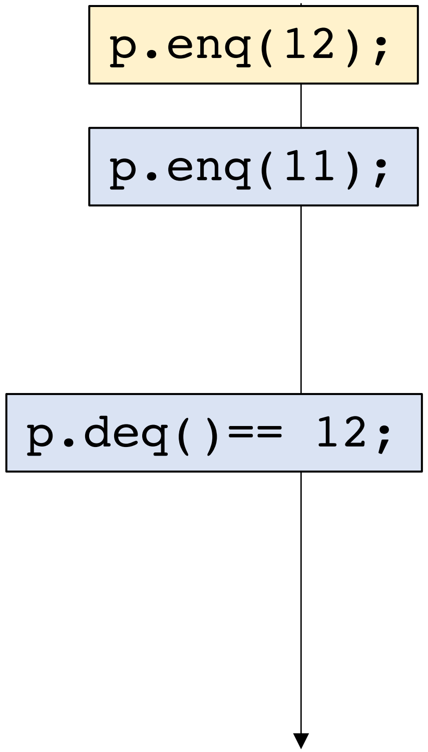
```
p.deq()== 12;
```

*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:*

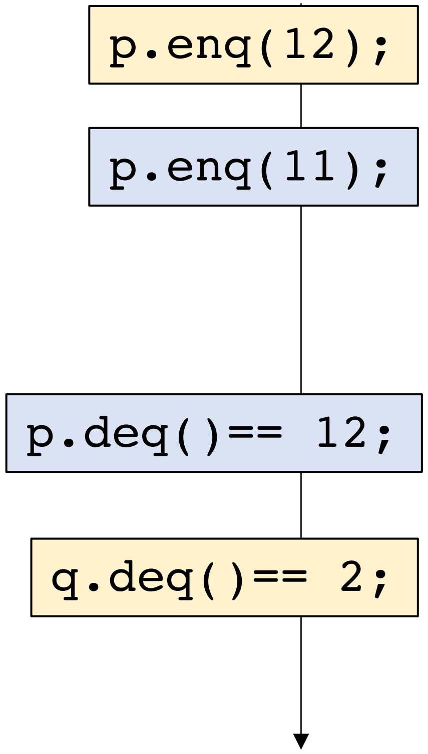
```
CQueue<int> p,q;
```

*Thread 0:*

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

*Thread 1:*

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



*Global variable:*

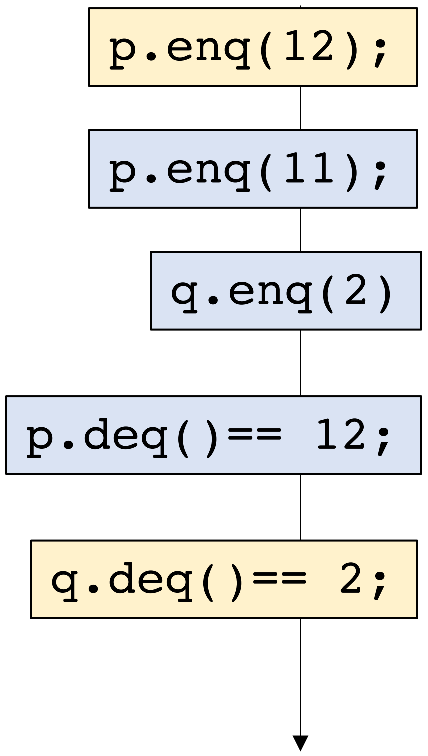
CQueue<int> p,q;

*Thread 0:*

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

*Thread 1:*

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



*Global variable:*

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

*Thread 0:*

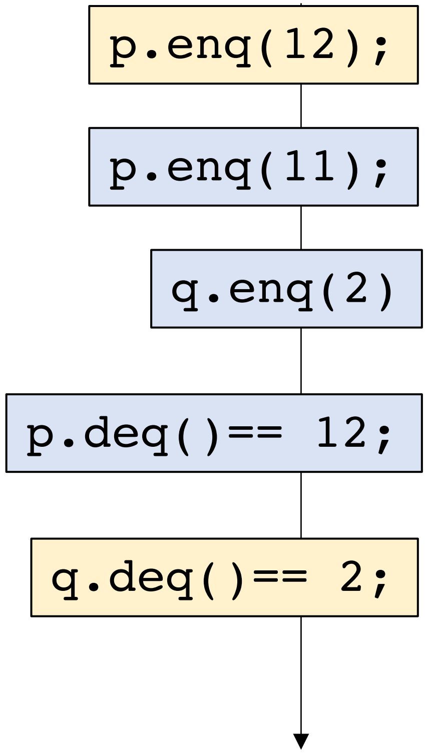
```
p.enq(11)
q.enq(2)
p.deq()==12
```

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

where to put this?

*Thread 1:*

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

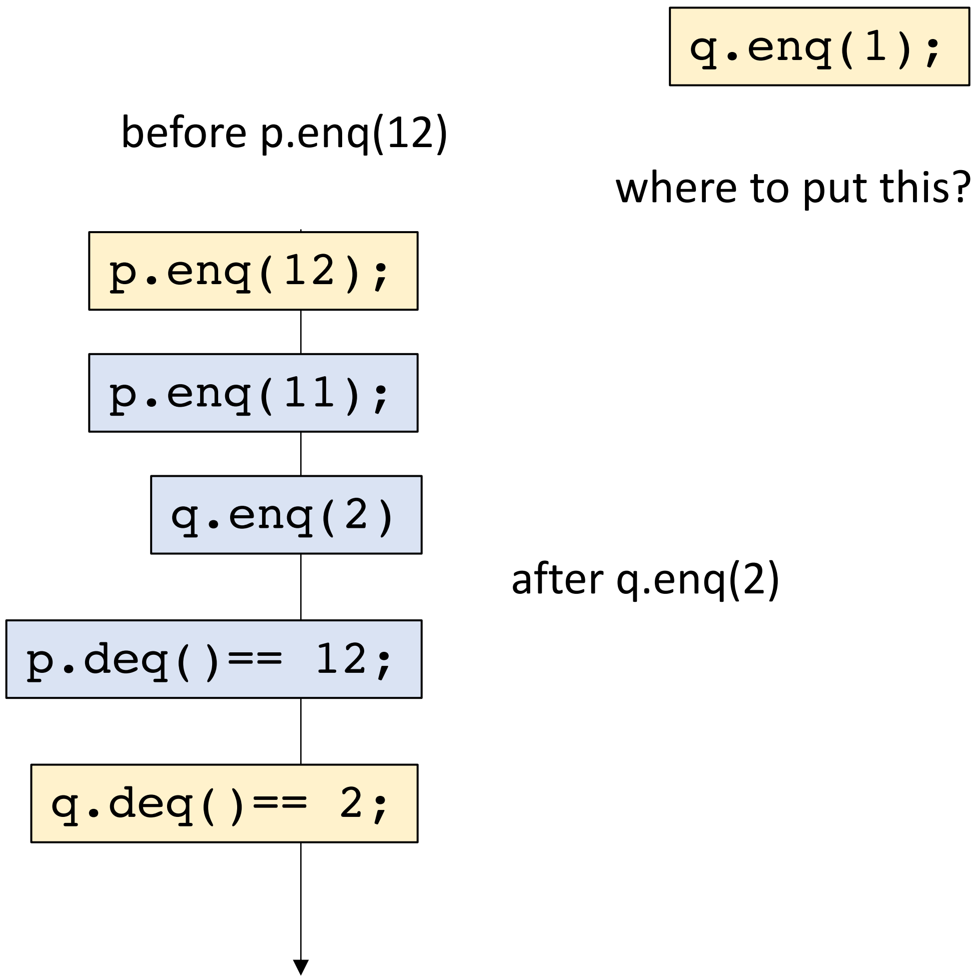


*Global variable:*

```
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
```

# 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!

# Next week

- A strong specification: Linearizability
  - Strictly stronger than sequential consistency
  - Reasoning about sequential consistency is still incredibly valuable
- Progress properties of concurrent objects
- Start looking at how to implement a linked list