# CSE113: Parallel Programming

Jan. 31, 2022

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

# Announcements

- Expect HW1 grades by Friday
  - Let us know if there are any issues ASAP
- Homework 2 is due on Friday
  - Please use office hours or piazza if you have questions
- Midterm is released in 1 week
  - asynchronous, 1 week (no time limit)
  - Open note, open internet (to a reasonable extent: no googling exact questions or asking questions on forums)
  - do not discuss with classmates AT ALL while the test is active
  - **No late tests will be accepted.**

# Returning to in-person

- Welcome!
  - Kresge 327
  - Figuring out lecture capture
  - Quizzes (attendance) will maintain the same format, please do them!

# Today’s Quiz

- Due Monday by class time

# Previous quiz

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

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

# Previous quiz

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

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

With this being the end of module two, please write a few sentence about how you found this module. For example, was the material clear? was the material interesting? What did you find surprising? What was something that was unclear?

# Review

# Reader-Writer Mutex

Global variable: `int tylers_account`

```
void buy_coffee() {
    tylers_account--;
}
```

```
void get_paid() {
    tylers_account++;
}
```

```
int check_balance() {
    return tylers_account;
}
```

# Reader-Writer Mutex

Global variable: `int tylers_account`

```
void buy_coffee() {
    m.lock();
    tylers_account--;
    m.unlock();
}
```

```
void get_paid() {
    m.lock();
    tylers_account++;
    m.unlock();
}
```

```
int check_balance() {
    m.reader_lock();
    int t = tylers_account;
    m.reader_unlock();
    return t;
}
```

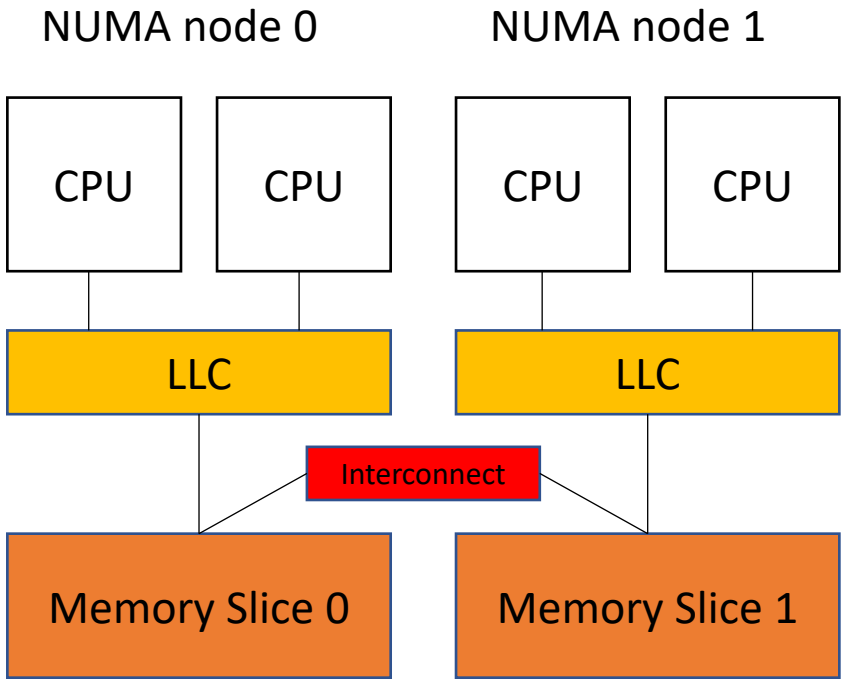
# Reader-Writer Mutex Implementation

```
void lock() {
  bool acquired = false;
  while (!acquired) {
    internal_mutex.lock();
    if (!writer && num_readers == 0) {
      acquired = true;
      writer = true;
    }
    internal_mutex.unlock();
  }
}

void unlock() {
  internal_mutex.lock();
  writer = false;
  internal_mutex.unlock();
}
```

```
void reader_lock() {
  bool acquired = false;
  while (!acquired) {
    internal_mutex.lock();
    if (!writer) {
      acquired = true;
      num_readers++;
    }
    internal_mutex.unlock();
  }
}

void reader_unlock() {
  internal_mutex.lock();
  num_readers--;
  internal_mutex.unlock();
}
```

# Hierarchical Locks

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



# Hierarchical Locks

```
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;
  }
}
```

Sleep longer for threads that are in different NUMA nodes

# Starvation for Hierarchical Locks

- Tune sleep times. You shouldn’t starve the other nodes!

- Advanced: have internal mutex state that counts how long the mutex has stayed with in the NUMA node.

# Data Conflicts in the Real World

- Be careful when writing concurrent code!
  - Data conflicts have led to real world catastrophes
  - Data conflicts lead to bugs that remain in code bases for very long times
    - they are difficult to trigger/find
  - Use Mutexes carefully
    - Using them more often can slow down code, but is more safe
    - Use optimized mutexes (Sleeping, peaking, fairness, RW locks)
  - Use tools to help you!
    - Thread sanitizer!

# On to new stuff!

# Schedule

- Intro to concurrent data structures
- Bank account example
- 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

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
  - 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

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!

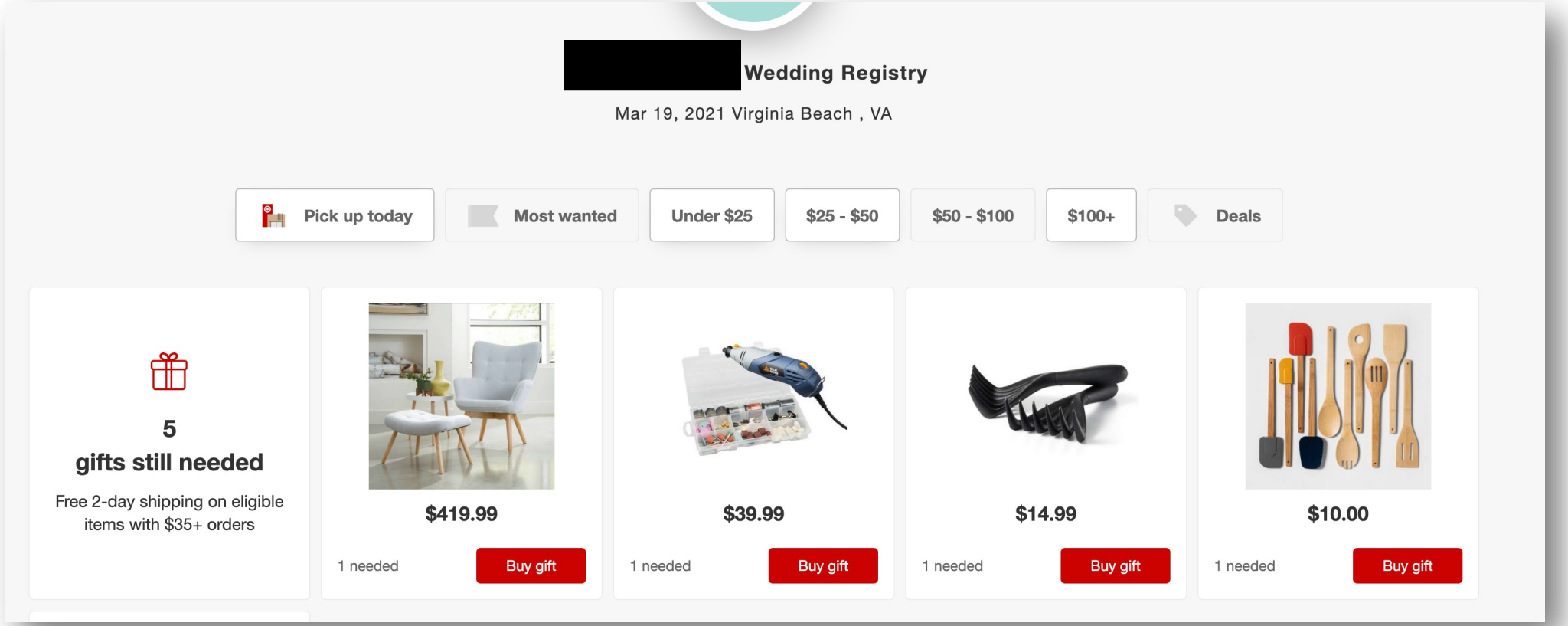

Consider two people splitting the work.

# 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");
```

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
```

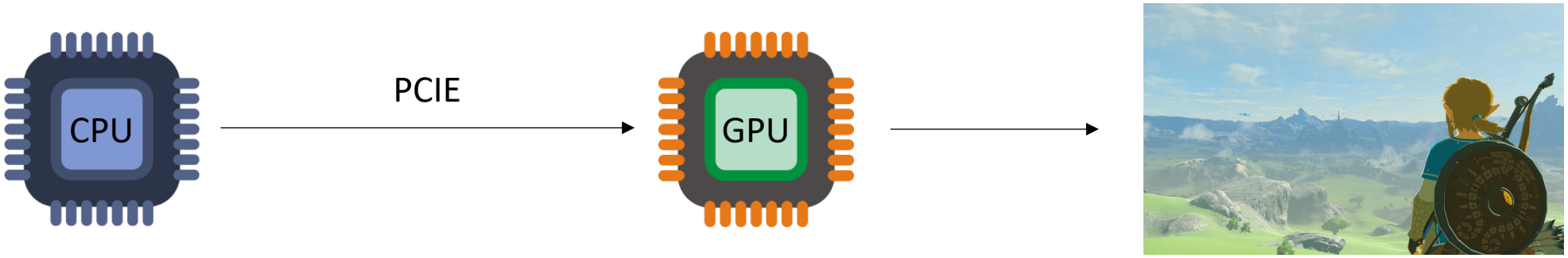
./a.out

terminal display

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

# Shared memory concurrent objects

- Graphics programming



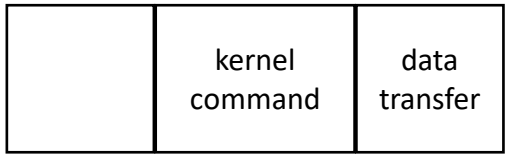

*loop:*
update data (data transfer)
graphics computation (kernel)

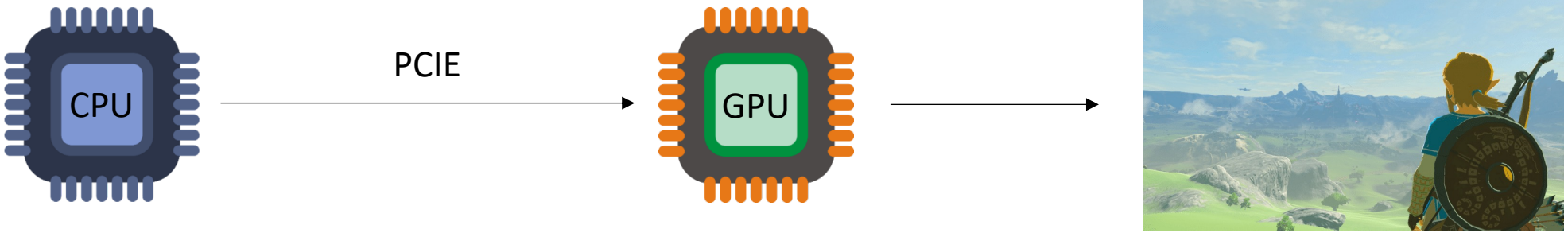
Nintendo: breath of the Wild

# Shared memory concurrent objects

- Graphics programming

Vulkan/OpenCL CommandQueue



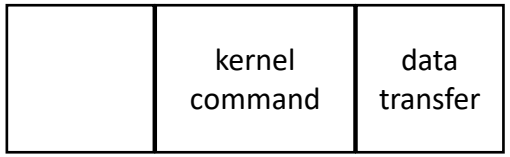

*loop:*
update data (data transfer)
graphics computation (kernel)

Nintendo: breath of the Wild

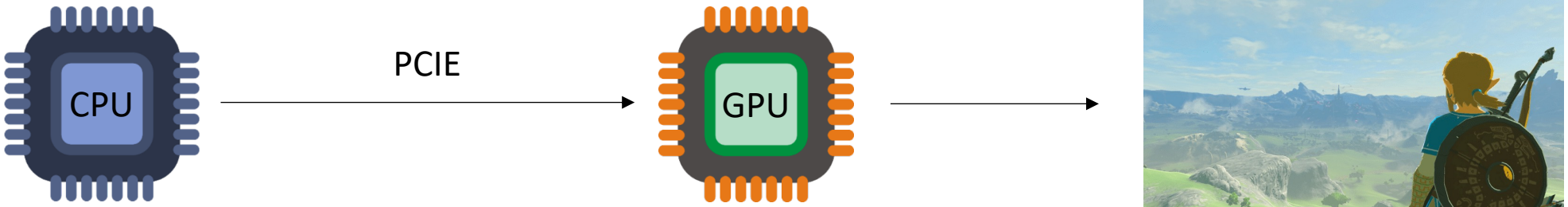
# Shared memory concurrent objects

- Graphics programming

Vulkan/OpenCL CommandQueue




*GPU driver concurrently reads from the queue*



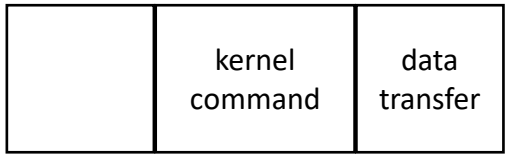

*loop:*
update data (data transfer)
graphics computation (kernel)

Nintendo: breath of the Wild

# Shared memory concurrent objects

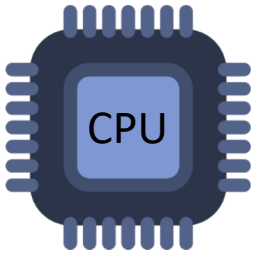
- Graphics programming

Vulkan/OpenCL CommandQueue



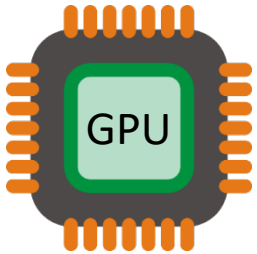

*GPU driver concurrently reads from the queue*

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

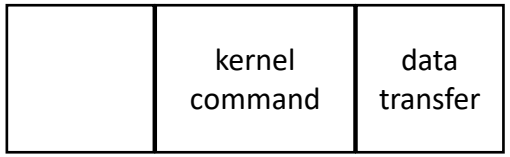
Scene 0

Nintendo: breath of the Wild

# Shared memory concurrent objects

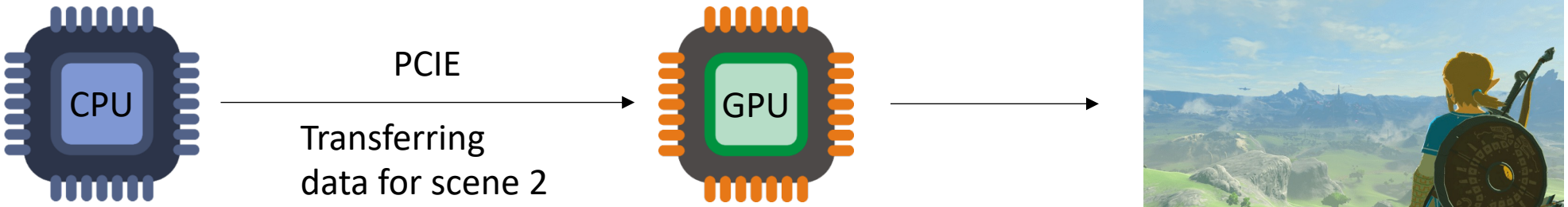
- Graphics programming

Vulkan/OpenCL CommandQueue




*GPU driver concurrently reads from the queue*

Single writer, single reader
Like in `Printf`






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

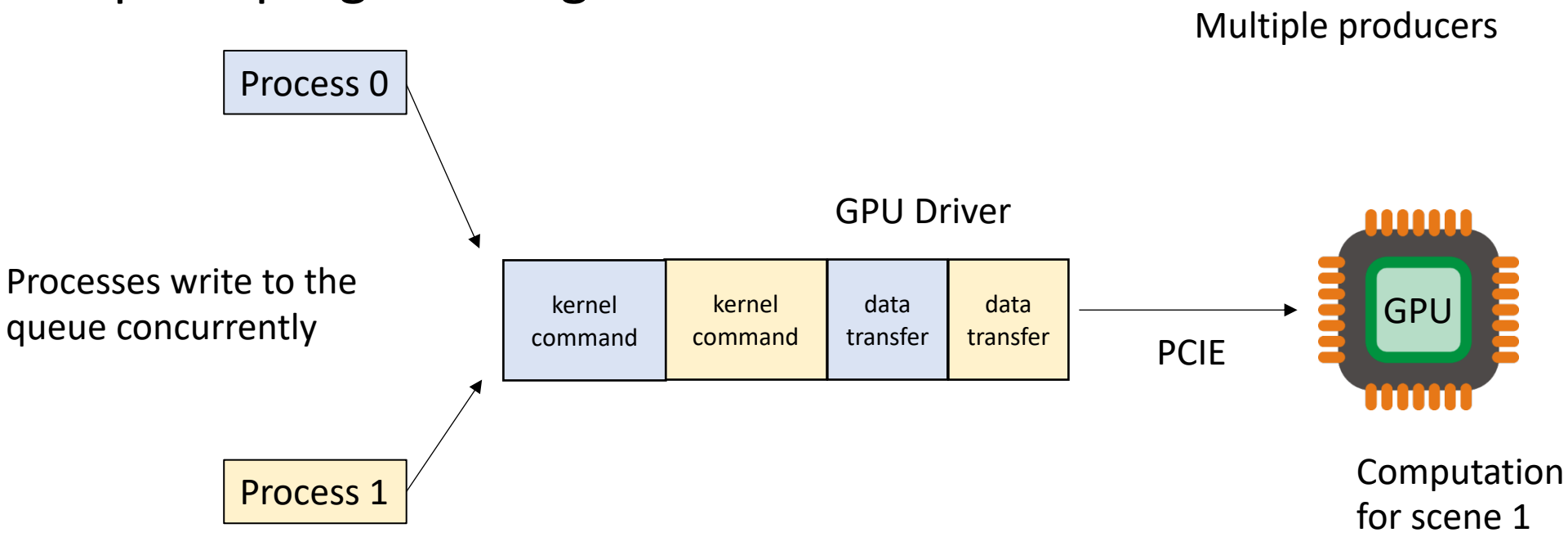
Computation for scene 1

Scene 0

Nintendo: breath of the Wild

# Shared memory concurrent objects

- Graphics programming




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

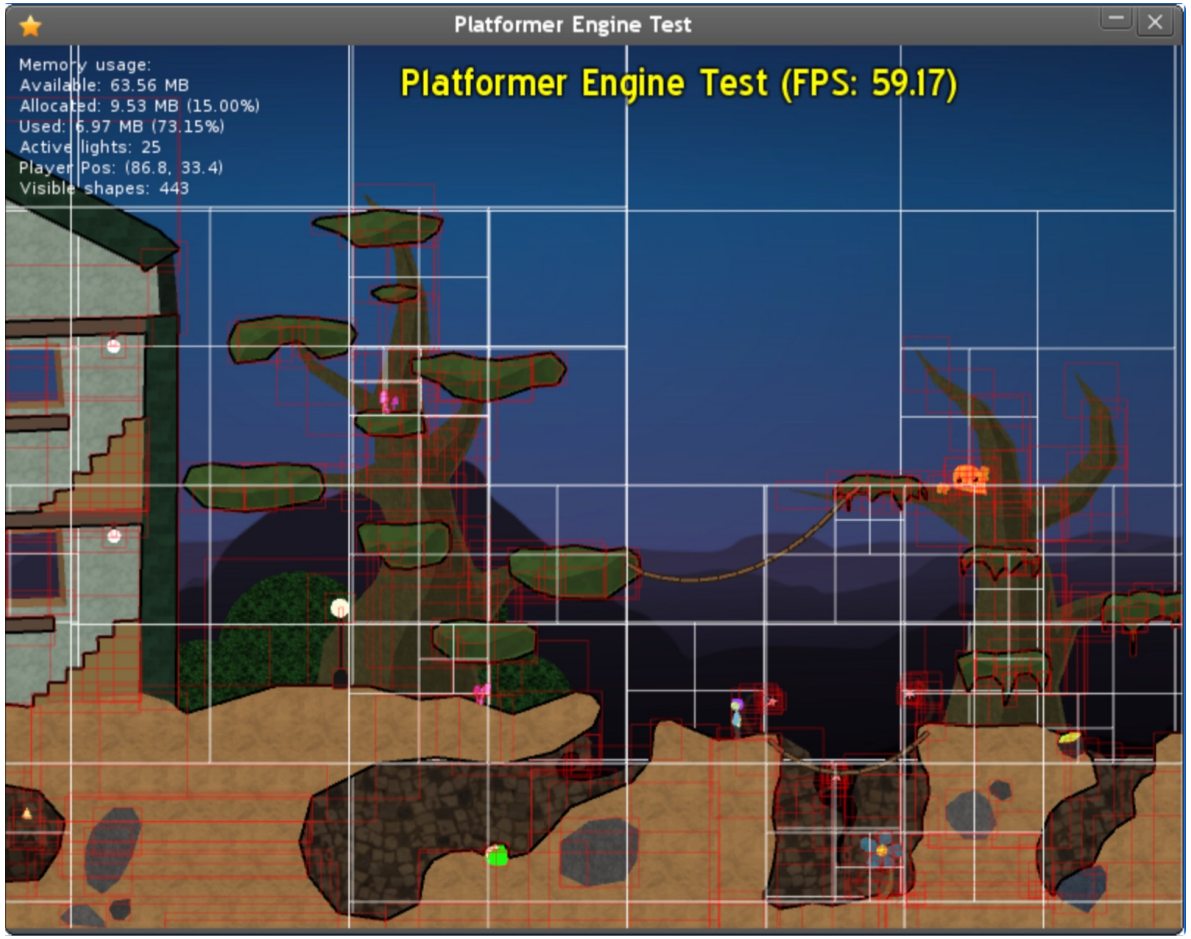
# Intro to concurrent objects

- 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

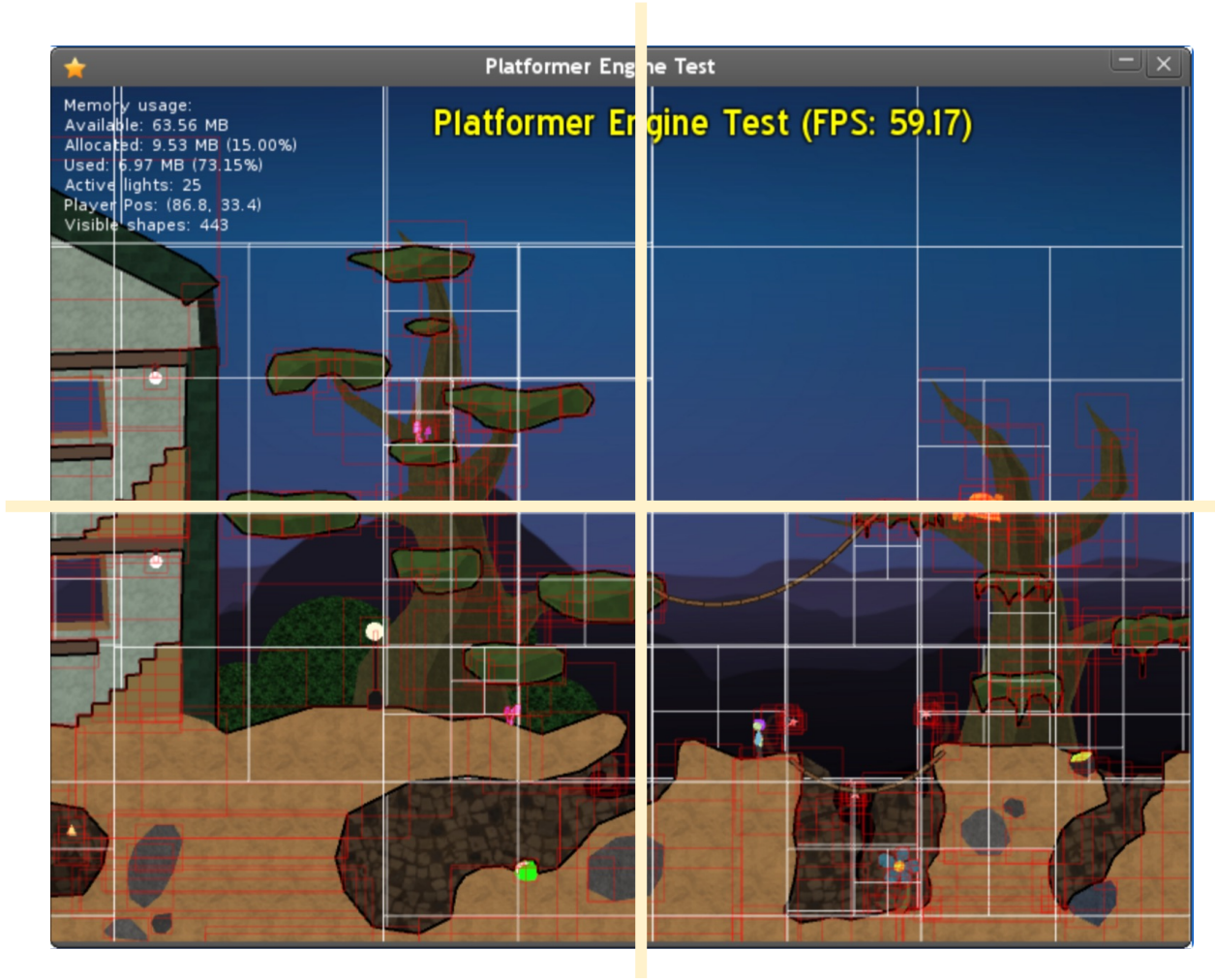



https://medium.datadriveninvestor.com/partitioning-2d-spaces-an-introduction-to-quadtrees-d95728856613

# Shared memory concurrent objects

- Quadtree/Octree

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

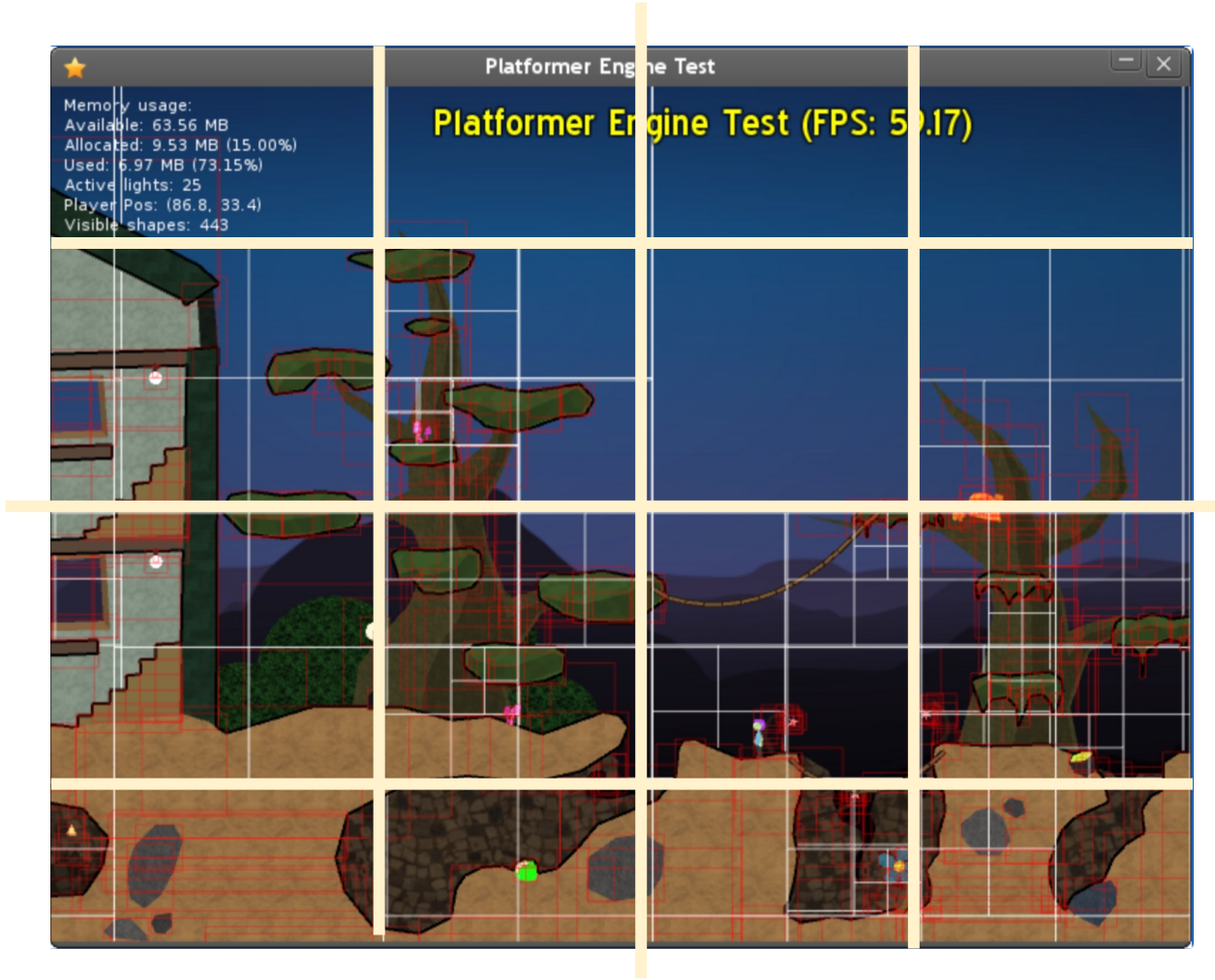



https://medium.datadriveninvestor.com/partitioning-2d-spaces-an-introduction-to-quadtrees-d95728856613

# 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

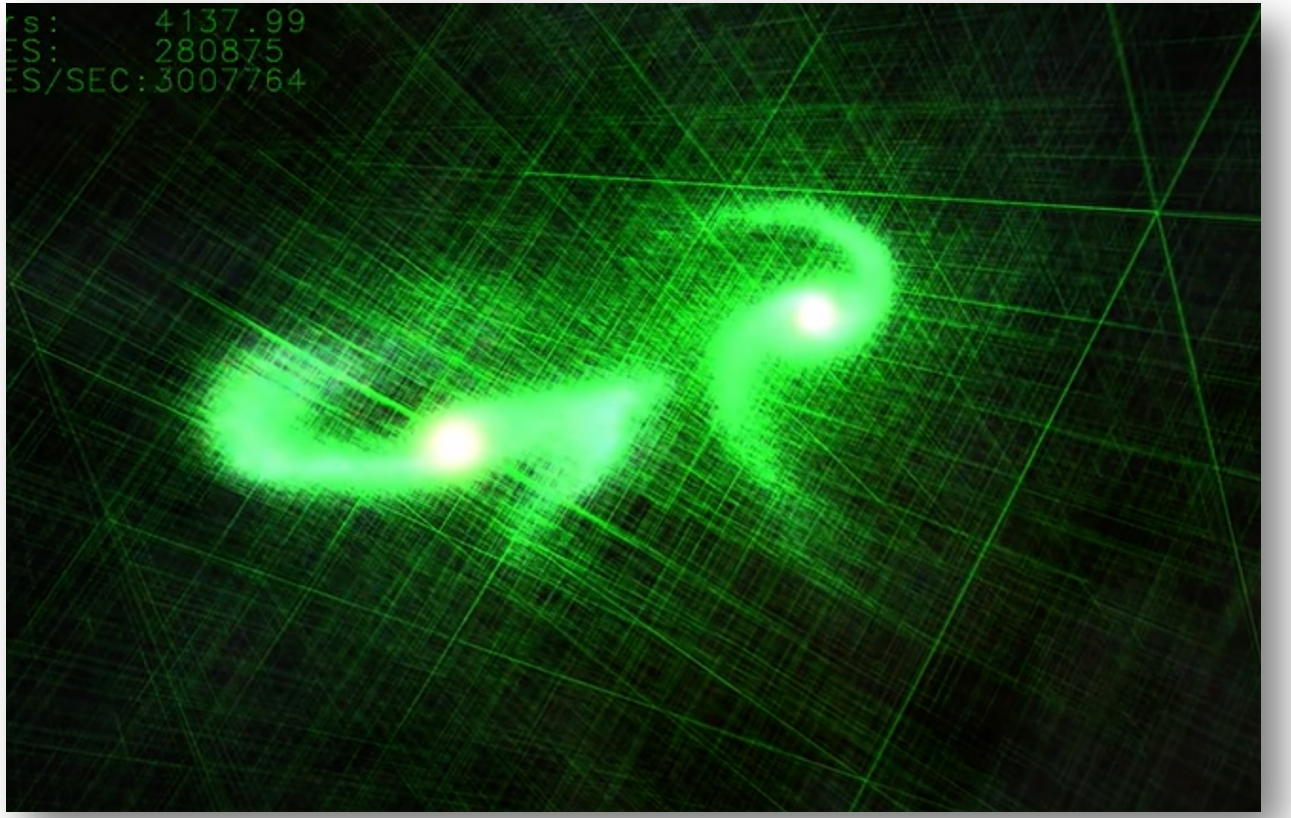
# Octree example

- From GTC 2012 (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

# 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 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!
- **Progress**:
  - What do we expect from the OS scheduler?
  - Under what conditions can concurrent objects deadlock

# Schedule

- Intro to concurrent data structures
- Bank account example
- 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: *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);
```

*Thread 1:*

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

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

*Global variable:*

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

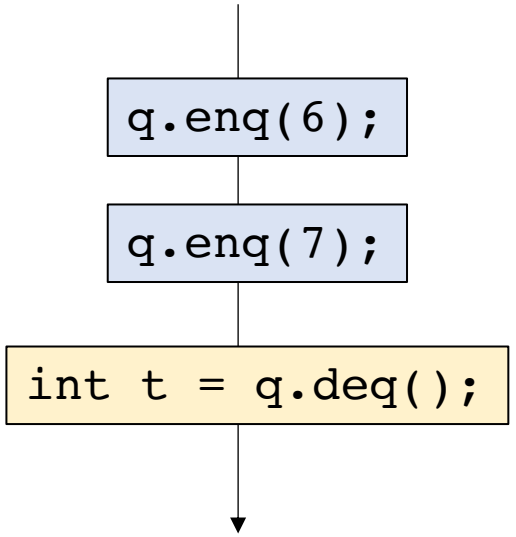
*Thread 0:*

```
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

*Global variable:*

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

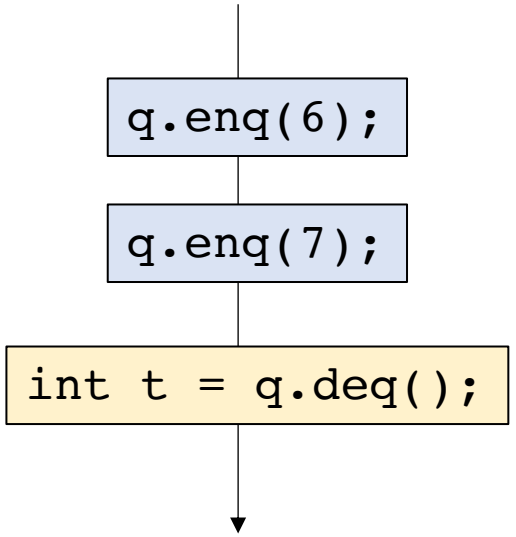
*Thread 0:*

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

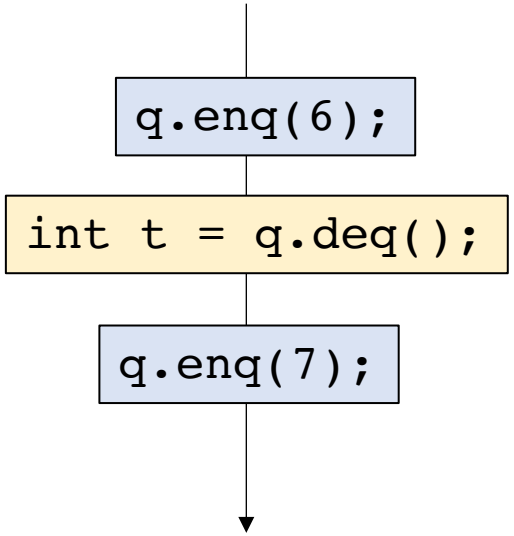
*Thread 1:*

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

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




t is 6




t is 6

*Global variable:*

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

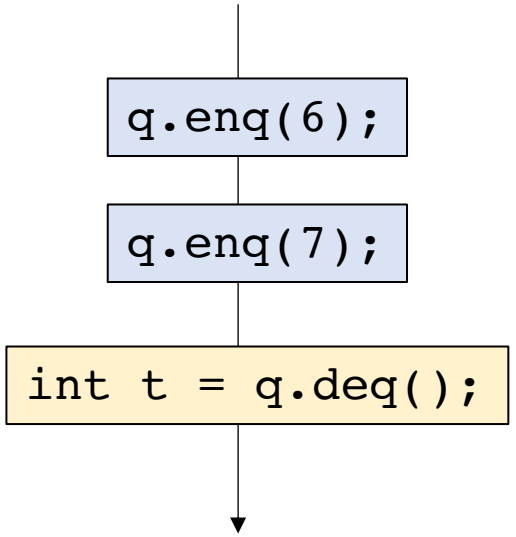
*Thread 0:*

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

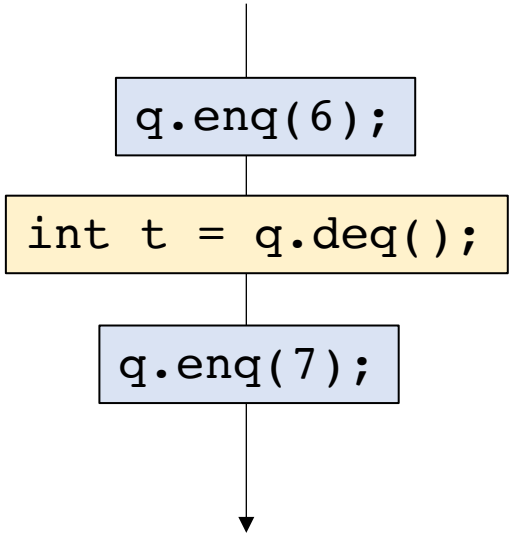
*Thread 1:*

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

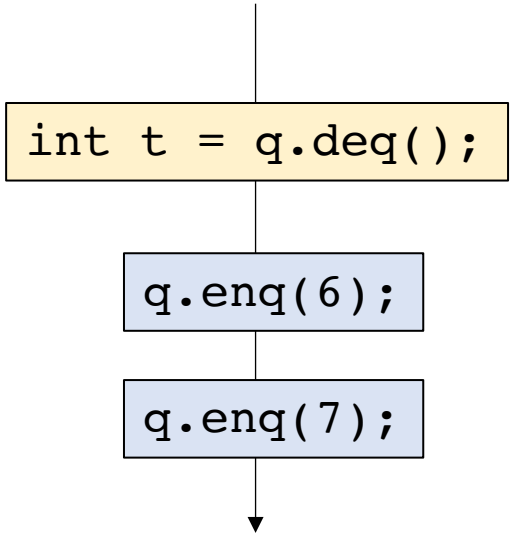
*Construct a sequential timeline of API calls*
*Any sequence is valid:*




t is 6




t is 6




t is None

*Global variable:*

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

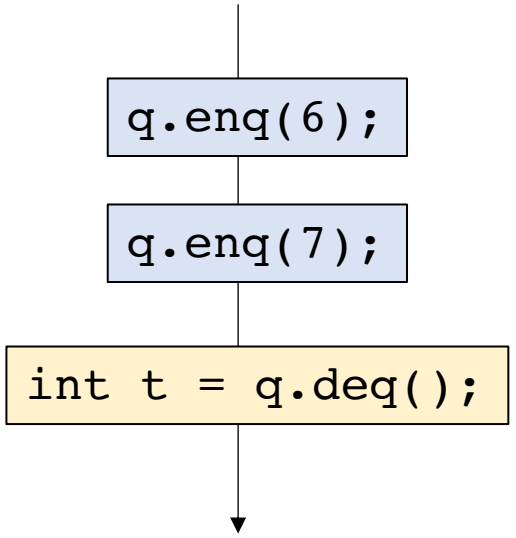
*Thread 0:*

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

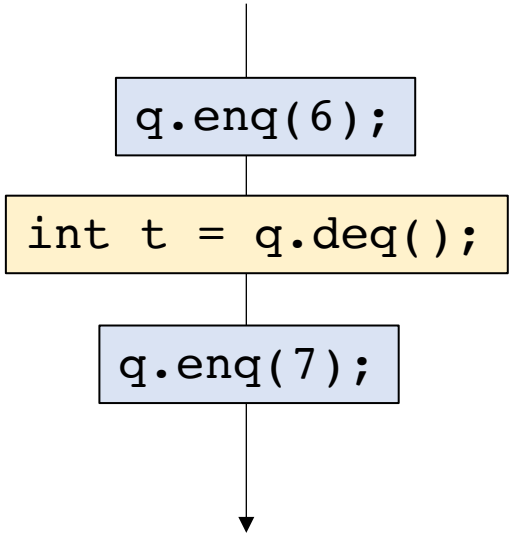
*Thread 1:*

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

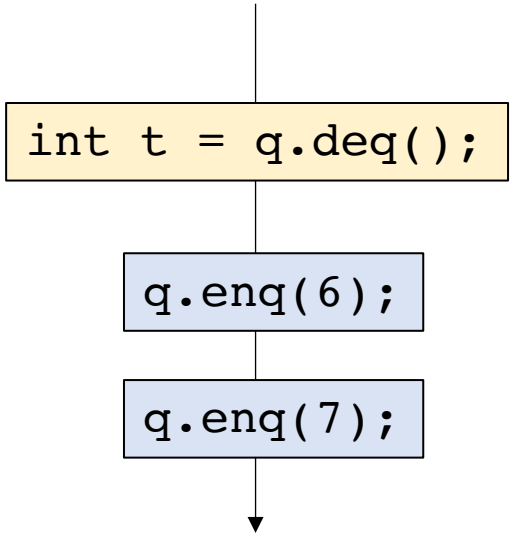
*Construct a sequential timeline of API calls*
*Any sequence is valid:*




t is 6




t is 6




t is None

*Can t ever be 7?*

*Global variable:*

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

*Thread 0:*

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

*Thread 1:*

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

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



*Can t ever be 7?*

*Global variable:*

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

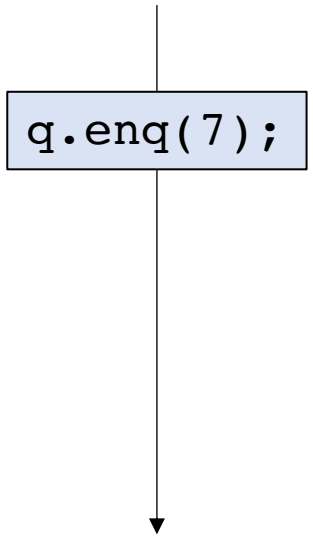
*Thread 0:*

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

*Thread 1:*

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

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




*Can t ever be 7?*

*Global variable:*

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

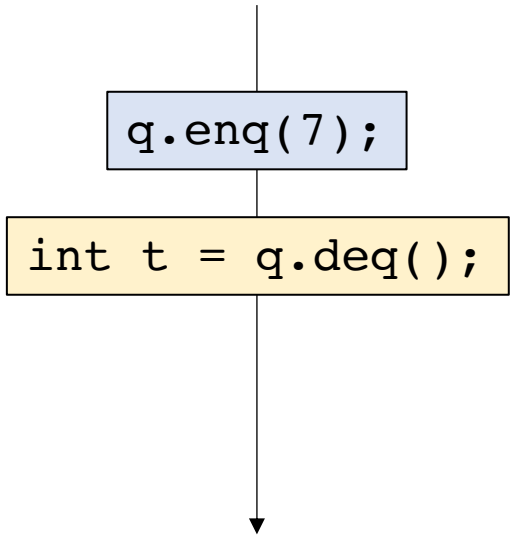
*Thread 0:*

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

*Thread 1:*

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

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




*Can t ever be 7?*

*Global variable:*

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

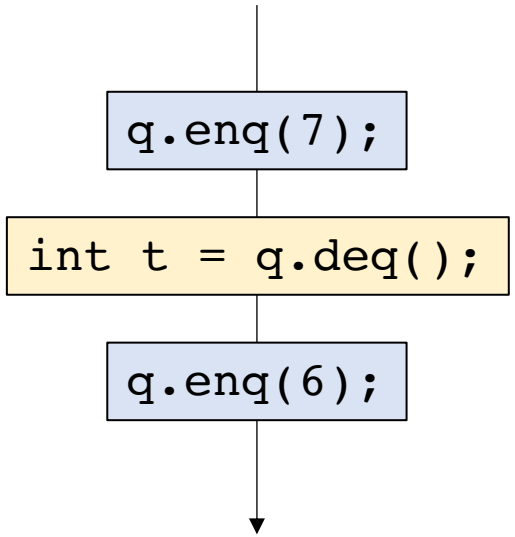
*Thread 0:*

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

*Thread 1:*

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

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




*Can t ever be 7?*

*Global variable:*

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

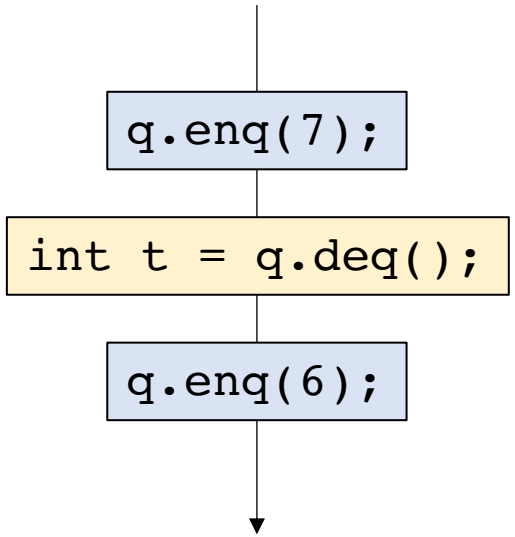
*Thread 0:*

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

*Thread 1:*

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

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




*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

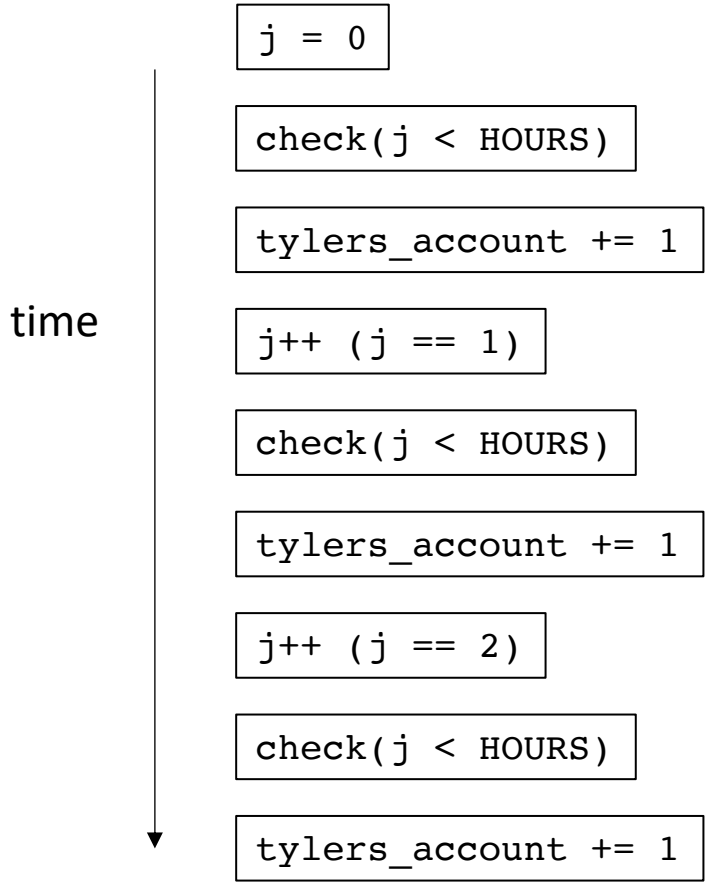
$$\frac{(N+M)!}{N!\,M!}$$

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:*

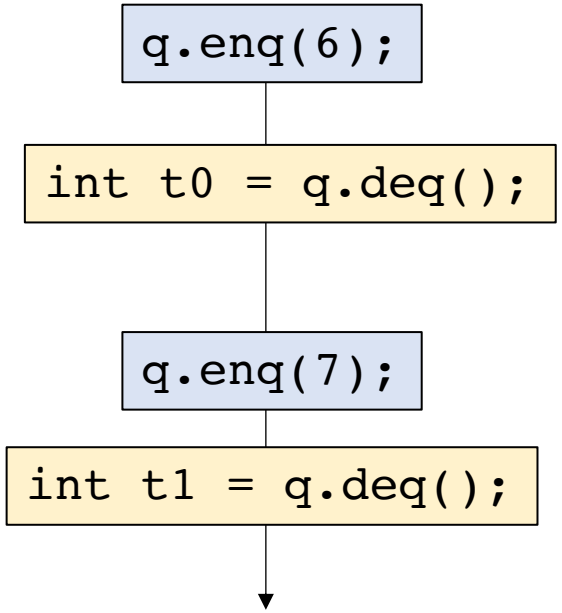
```
CQueue<int> q;
```

*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);
```

Lets do another!

*Thread 1:*

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

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

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



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

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

Can t0 == 6 and t1 == 7?

*Global variable:*

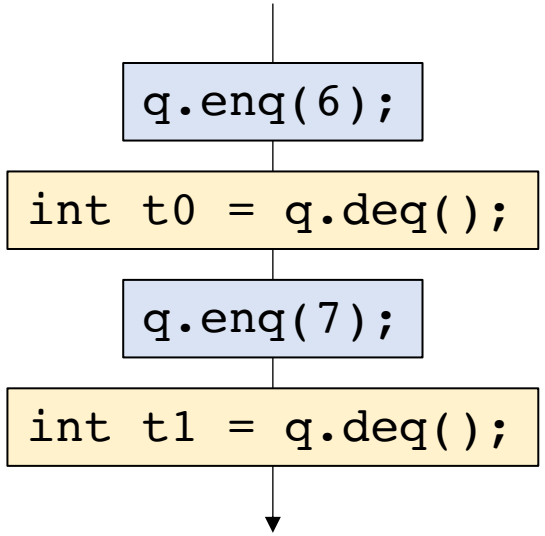
```
CQueue<int> q;
```

*Thread 0:*

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

*Thread 1:*

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




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:*

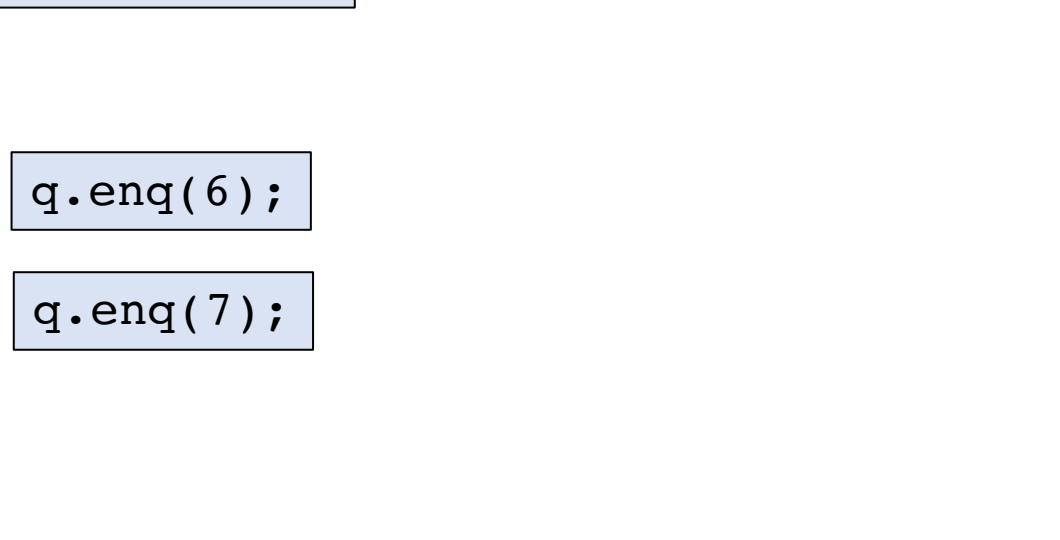
```
CQueue<int> q;
```

*Thread 0:*

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

*Thread 1:*

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




Can t0 == 0 and t1 == 7?

*Global variable:*

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

*Thread 0:*

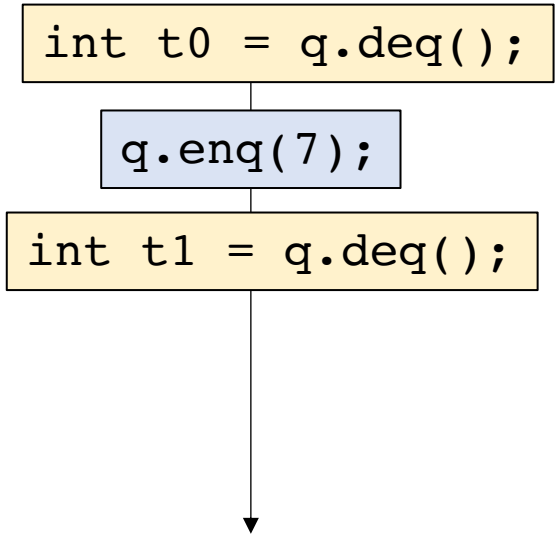
```
q.enq(6);
q.enq(7);
```

*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:*

```
CStack<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.

# Next week

- Work on HW 2
  - Visit office ours if you need to
  - Ask questions on piazza