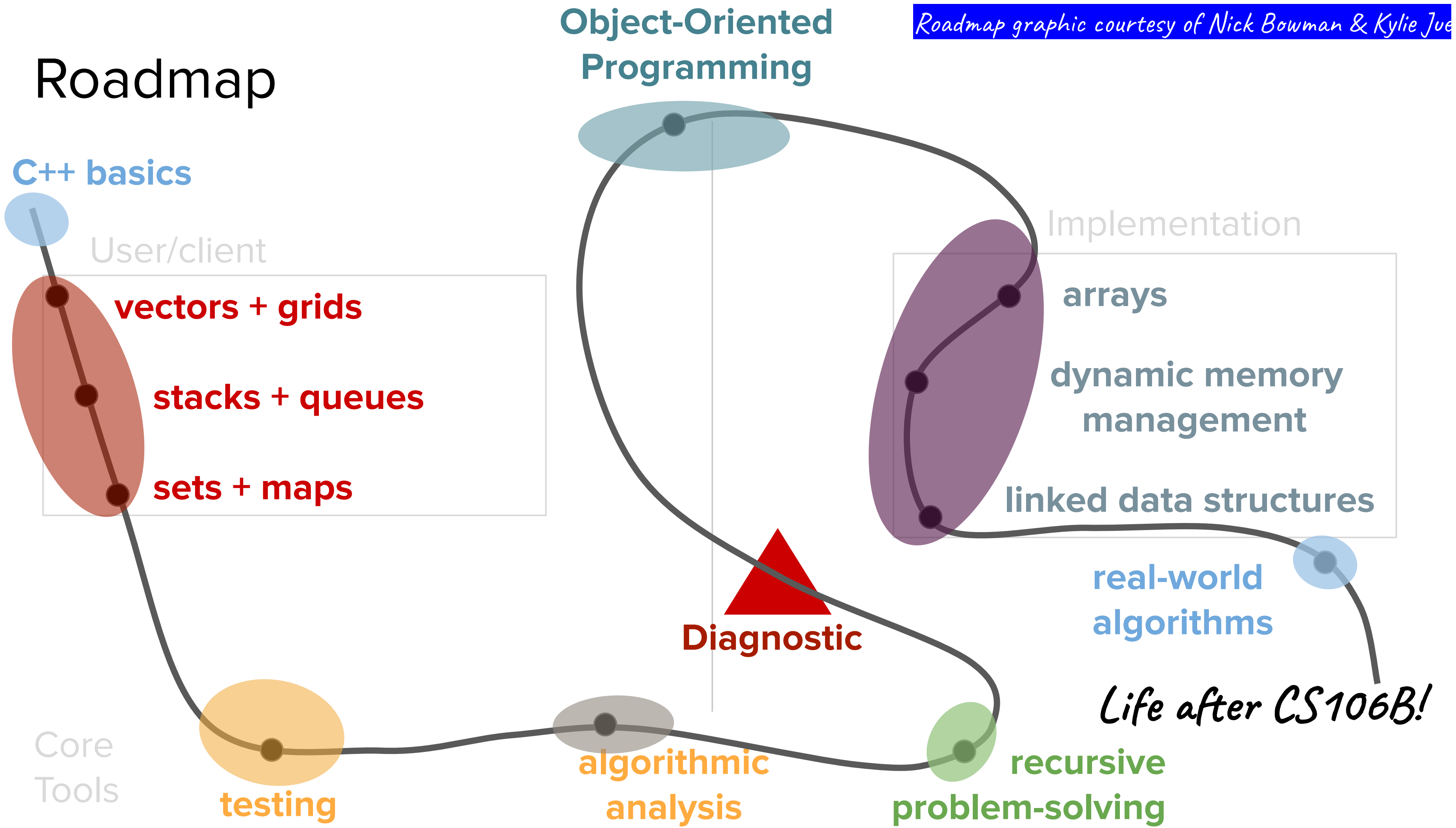
Multithreading and Parallel Computing

What kinds of multitasking do you do every day?

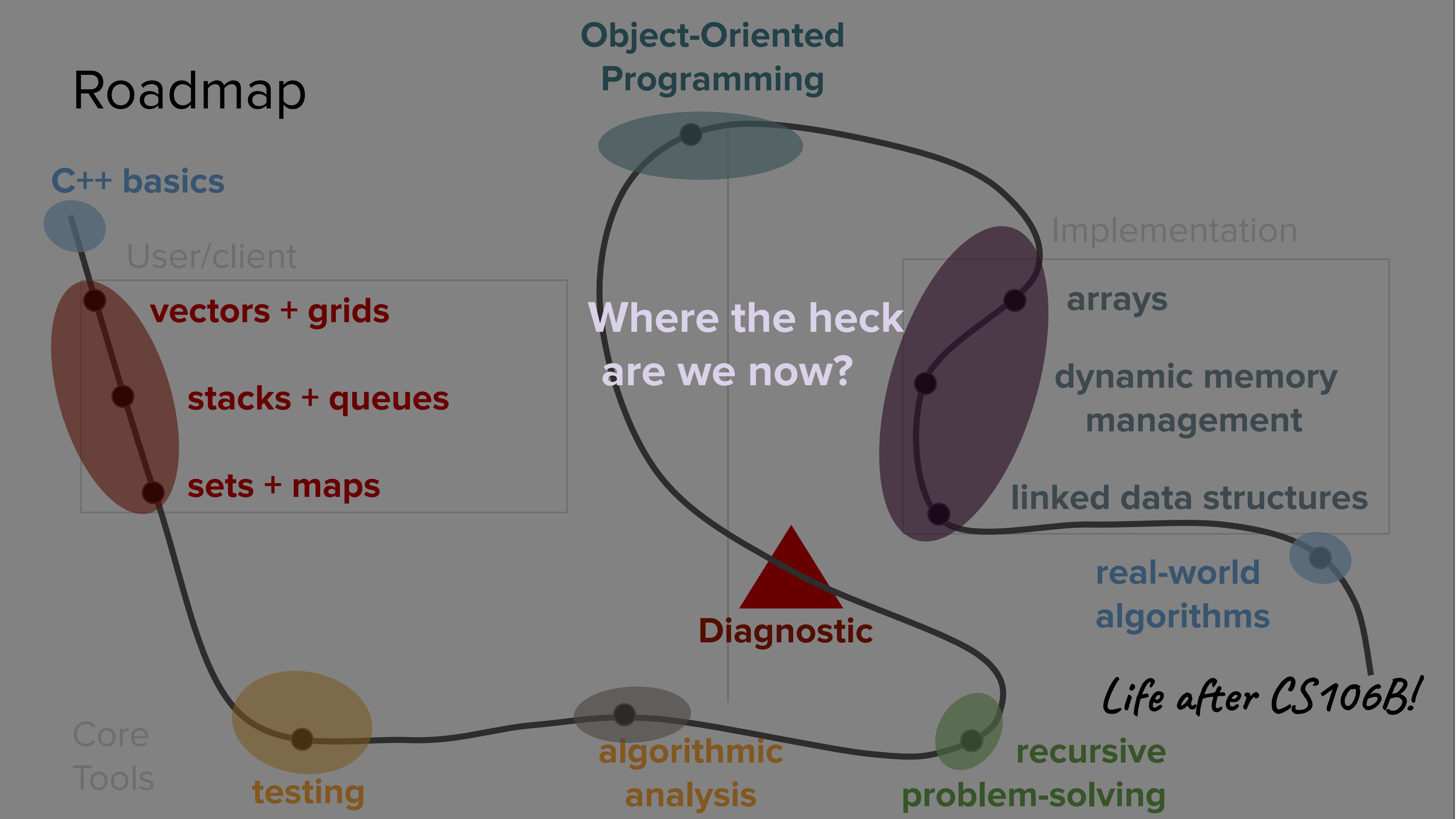
tinyurl.com/lecture25

What kinds of multitasking do you do every day?

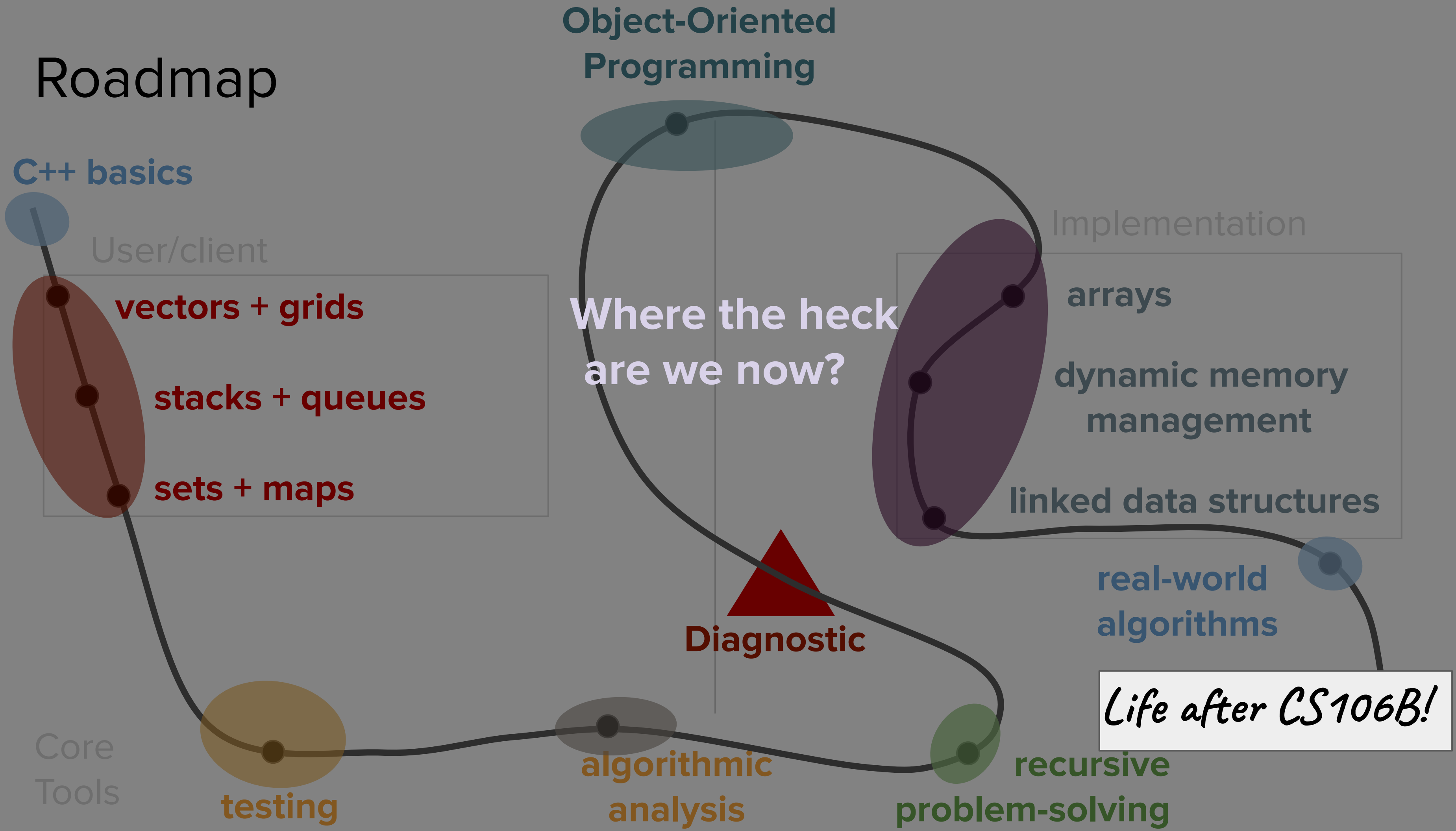
Roadmap


Roadmap graphic courtesy of Nick Bowman & Kylie Jue
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Core
Tools
testing
algorithmic
analysis
Object-Oriented
Programming
Diagnostic
recursive
problem-solving
Implementation
arrays
dynamic memory
management
linked data structures
real-world
algorithms
Life after CS106B!

Roadmap

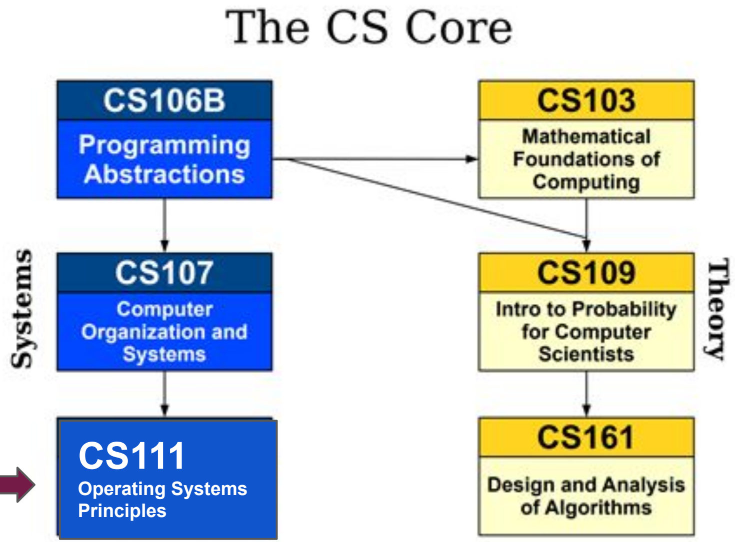

Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Where the heck are we now?
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

Roadmap



Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Where the heck are we now?
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

A glimpse of the future...



Multithreading!

Today's question

How can we harness the cores in our computer in order to parallelize a workload safely?

woah.

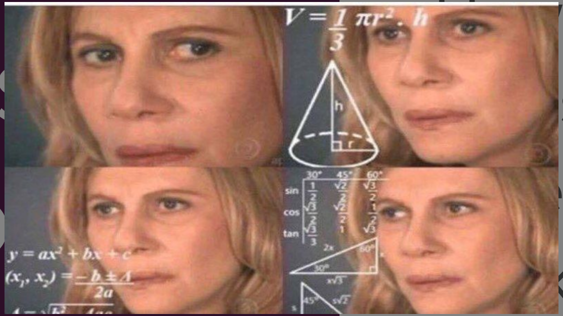
Today's question

How can we harness the cores in our computer in order to parallelize a workload safely?

Multiple cores?

Parallelize work??

Today's questio



...can we harness the
...s in our computer in
...er to parallelize a
...kload safely?

Today's topics

1. How code is run (short!)
2. An intro to Computer Architecture (Threads & Processors)
3. Thread Safety

Review

[Hashing]

HashMap the Hamster, courtesy of Lucía

Definition

hash function

A function that takes in arbitrary inputs and maps them to a fixed set of outputs.

Using Hash Functions

Our bucket rule:

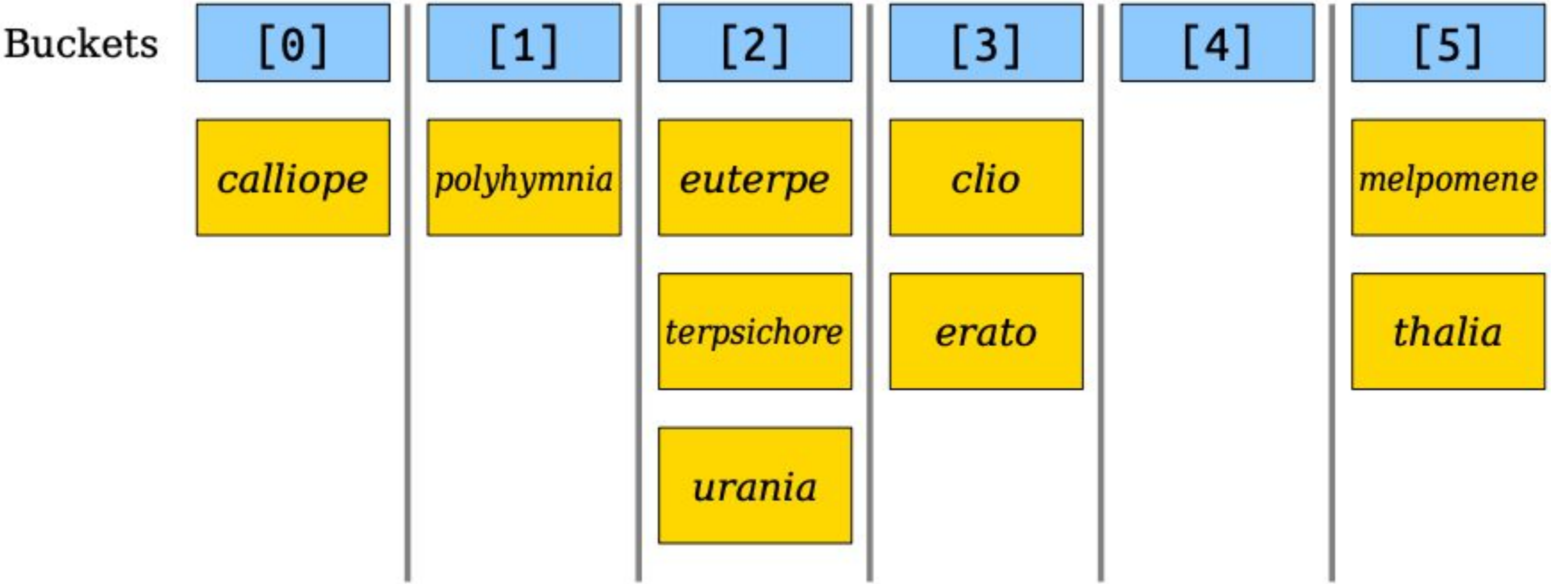
```
bucket = hash(input) % numBuckets;
```

Using Hash Functions

Main idea: We can turn any data into a sufficiently "random" index in an array of any size!

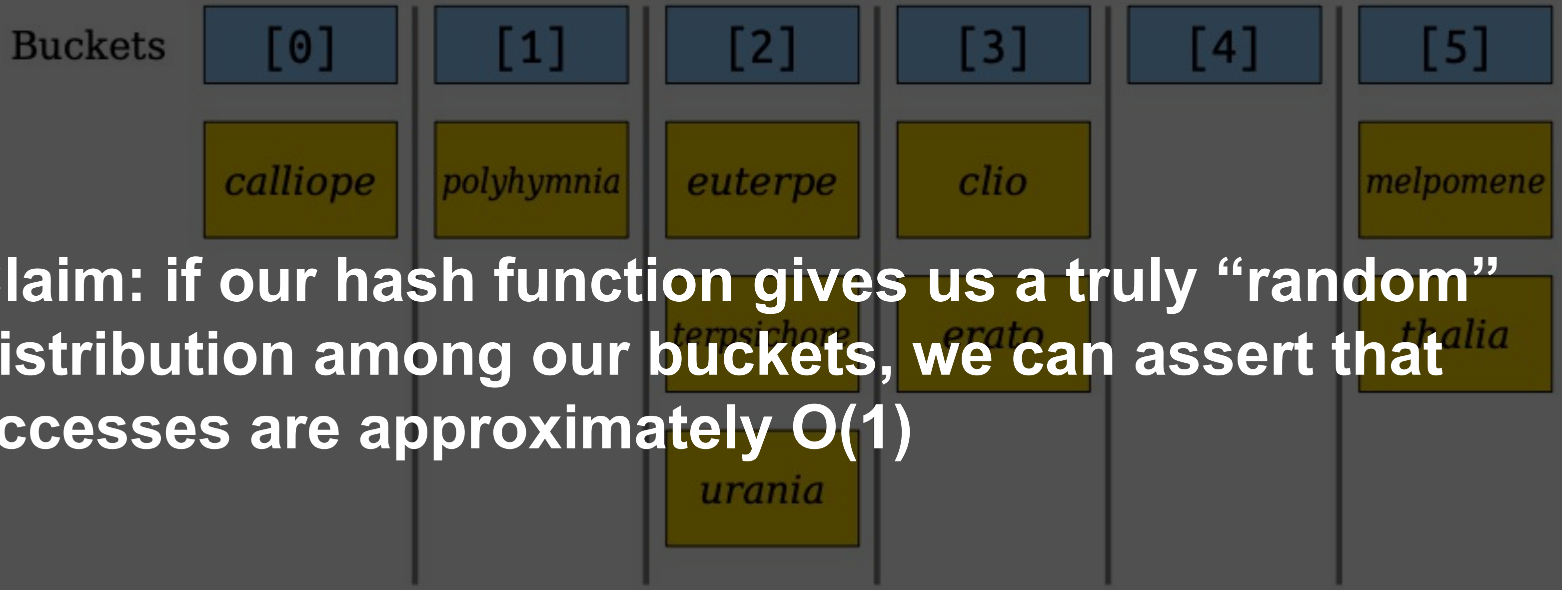
Our bucket rule:

```
bucket = hash(input) % numBuckets;
```



Our bucket rule:

```
bucket = hash(input) % numBuckets;
```



Claim: if our hash function gives us a truly "random" distribution among our buckets, we can assert that accesses are approximately O(1)

Our bucket rule:

```
bucket = hash(input) % numBuckets;
```

Other notes about Hashing

- If two elements are ‘hashed’ to the same bucket, we call that a ***collision***. Depending on how you implement your data structure, you can either allow the bucket to “grow,” (called chaining), or you can implement other clever schemes if you want to keep your buckets to 1 element (look up Linear Probing or Robin Hood Hashing!

Other notes about Hashing

- If two elements are 'hashed' to the same bucket, we call that a ***collision***. Depending on how you implement your data structure, you can either allow the bucket to "grow," (called chaining), or you can implement other clever schemes if you want to keep your buckets to 1 element (look up Linear Probing or Robin Hood Hashing!
- When your hashed ADT starts to fill up, you may need to ***rehash***, which is a process of expanding your array size and then recomputing the bucket index for each element with the new array size. Programmers often *rehash* when their ***load factor*** (number of elements / number of buckets) exceeds a certain threshold (like 0.75).
  - A cool optimization problem is figuring out how much to increase your hashed ADT without wasting space or sacrificing performance – you want to keep your load factor in a "sweet spot"

Back to content...

How do we run code? (short!)

(simple code flow)

How code is run

- How does the computer read and run your code?
  - Logically, **it** should read your code from top to bottom!

```cpp
int main () {

    int yeet = 9338;
    double foo = 2.4;

    doSomeMath(yeet);

    cout << "time to go home!" << endl;

    return 0;
}
```

How code is run

- How does the computer read and run your code?
  - Logically, **it** should read your code from top to bottom!

```
int main () {

    int yeet = 9338;
    double foo = 2.4;

    doSomeMath(yeet);

    cout << "time to go home!" << endl;

    return 0;
}
```

...but *who* is **it**? What's the thing that encapsulates and runs your code?

Definition

thread

An abstraction that represents a sequential execution of code.

Definition

thread

An abstraction that represents a sequential execution of code.

Line by line, top to bottom!

Definition

thread

An abstraction that represents a sequential execution of code.

Anything that's code!

Thread Examples

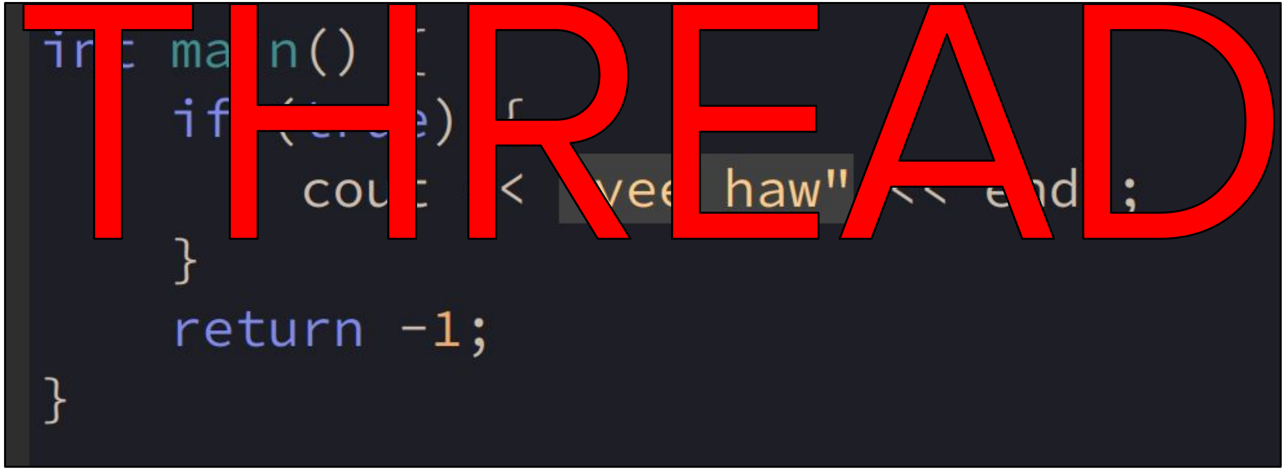
- Right now, your computer probably has a few threads running right now!
    - What are some examples of threads running on your PC?

Thread Examples

- Do you have QT creator open right now?

Thread Examples

- Do you have QT creator open right now?

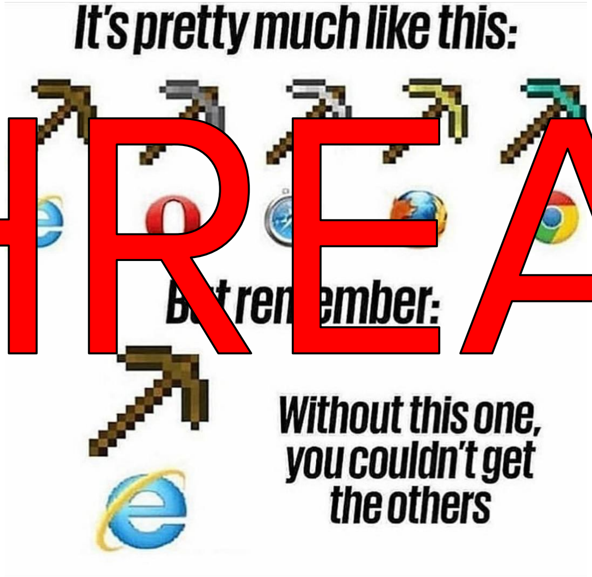


Thread Examples

- Do you have a web browser open? (Chrome, Safari?)

Thread Examples

- Do you have a web browser open?



THREAD

*unless you're using Chrome, sort of.

Thread Examples

- Are you watching TikToks during lecture?

Thread Examples

- Are you watching TikToks during lecture?

Stanford announces Charli D'Ameli as 2020 Commencement Speaker

THREAD

"Charli undeniably captures the same spirit of ingenuity we try to cultivate at our different schools," said President Marc Tessier-Lavigne. (Photo Edit: RICHARD COCA/The Stanford Daily)

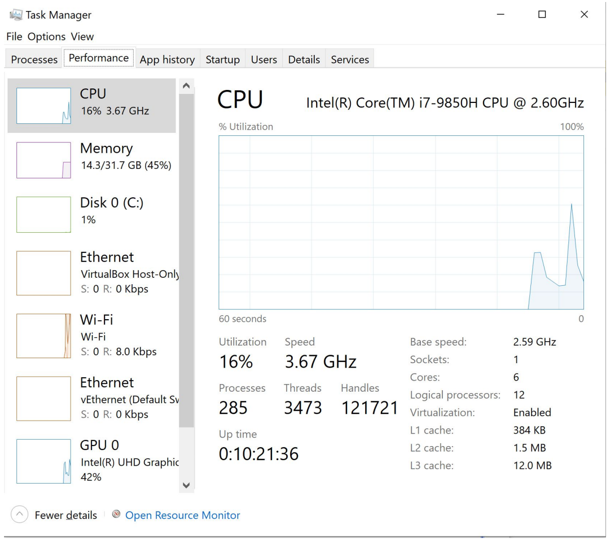
I have been told Ms. D'Amelio is a TikTok #influencer

Question:

How many threads do you think my computer had active when I was making this slide?

Thread examples

- Right now, your computer is executing a bunch of threads!
  - At the time of making this slide show, my computer was handline 3473 threads!
- Many large programs (your web browsers!) need **multiple threads** to run. That's because they have so many moving parts!



Question:

When you run a program in Qt Creator, is a thread executing your code?

Answer:

Er... Yes, sort of!

Answer:

Er... Yes, sort of!

Yes, when you run a program in Qt, a thread encapsulating your code is being **executed**.

Answer:

Er... Yes, sort of!

Yes, when you run a program in Qt, a thread encapsulating your code is being **executed**.

However, a thread alone isn't enough to run your code!

Definitions

software
Programs and and abstractions (code). Not a physical entity.

hardware
Physical parts of a computer.

The hardware-software boundary

- A thread **alone** cannot run your program.
    - A thread is just **software** that is an **abstraction** for some code.
- A thread needs to work with the computer's **hardware** in order to run the code it encapsulates!

The hardware-software boundary

- A thread **alone** cannot run your program.
  - A thread is just **software** that is an **abstraction** for some code.
- A thread needs to work with the computer’s **hardware** in order to run the code it encapsulates!

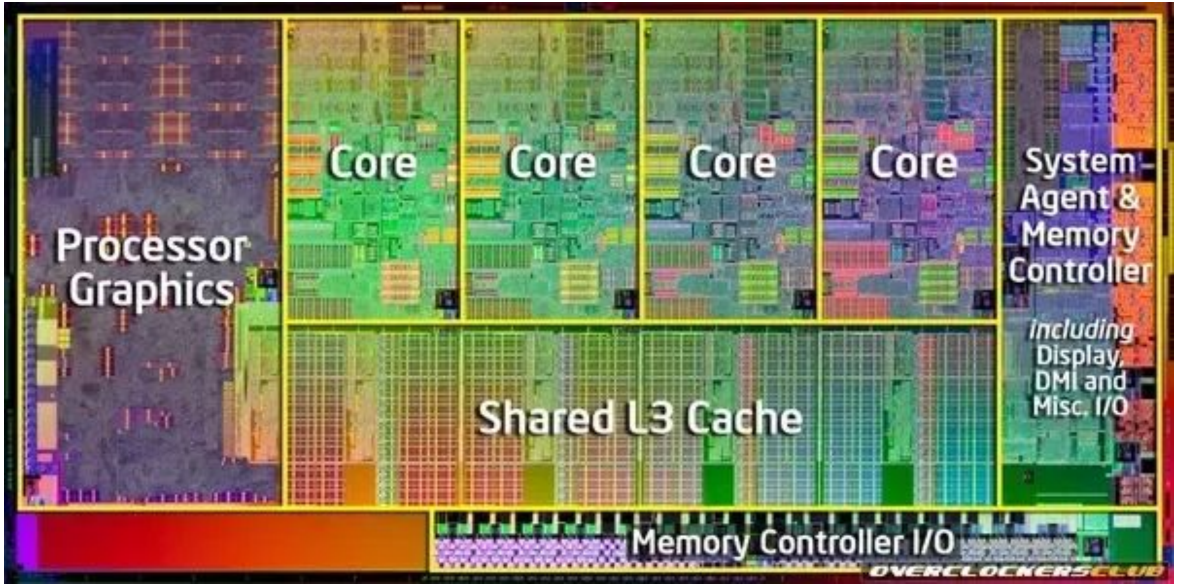
… but what piece of hardware does this?

Definitions

CPU (Central Processing Unit)
A piece of hardware responsible for executing instructions that make up a computer program

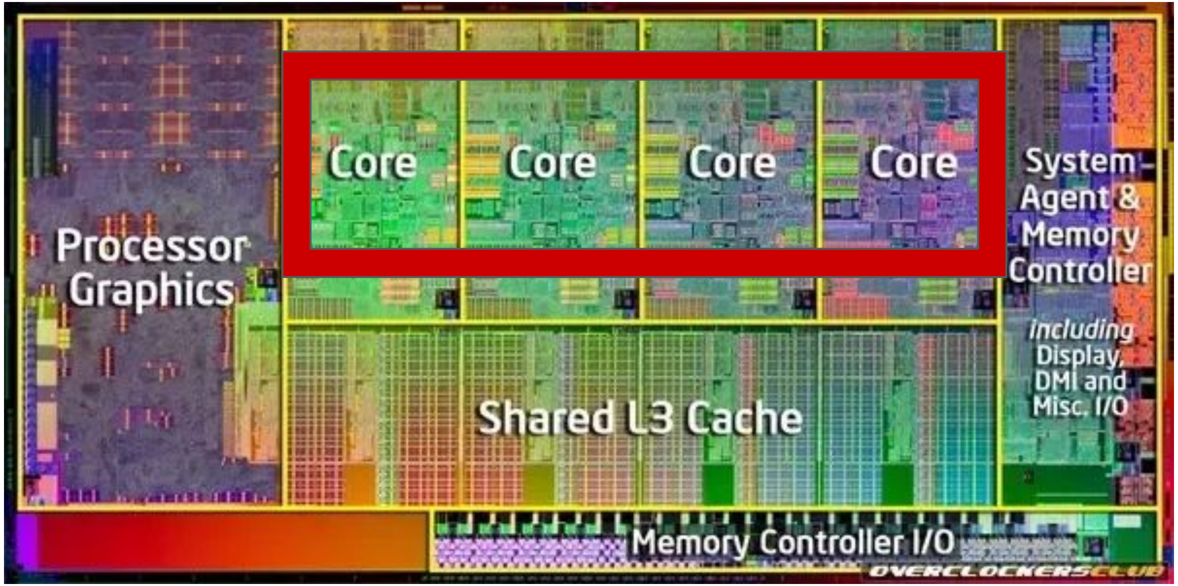
Core
An individual processor inside of a **CPU**. Each **core** is able to execute code independently of other **cores**.

Inside a CPU...



Don't worry about the other stuff -- we just care about the **cores!**

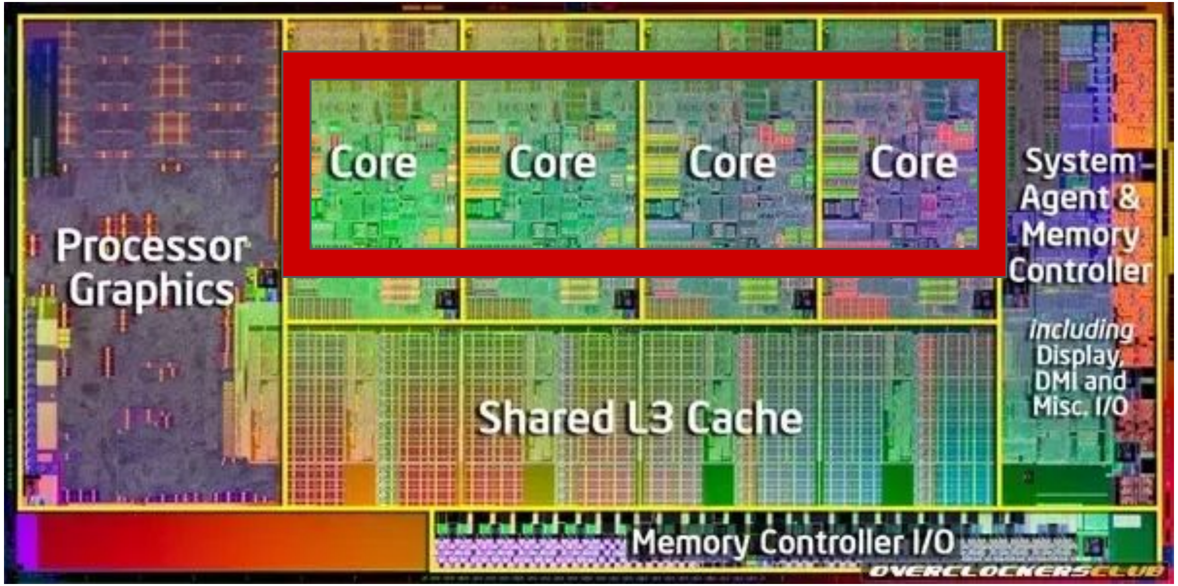
Inside a CPU…



Don’t worry about the other stuff -- we just care about the **cores!**

Inside a CPU...

How many concurrent programs can this CPU run?



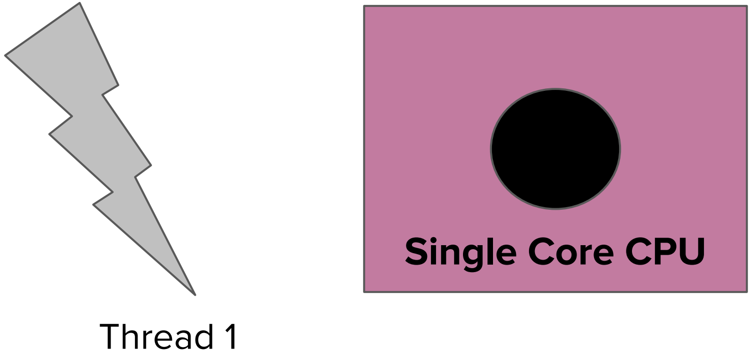
Don’t worry about the other stuff -- we just care about the **cores!**

Threads ‘n cores

- In order for a **thread** to be able to execute some code, it must be running on a **CPU core.**
- If all **cores** are currently busy, a thread must **wait** for a **core** to free up before it can hop on that **core** and begin executing its own code!

Threads 'n cores

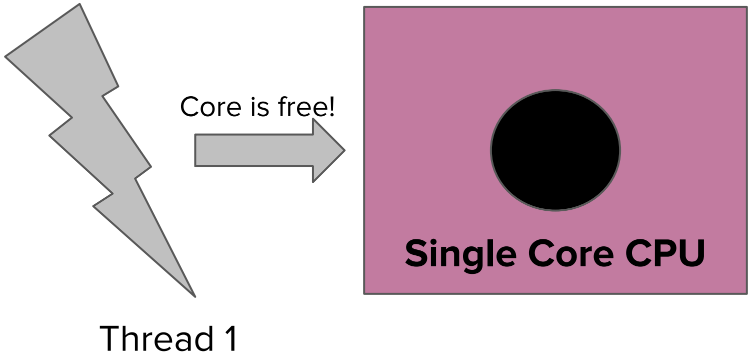
- In order for a **thread** to be able to execute some code, it must be running on a **CPU core.**
- If all **cores** are currently busy, a thread must **wait** for a **core** to free up before it can hop on that **core** and begin executing its own code!



Let's assume this computer has a CPU with only **one core.**

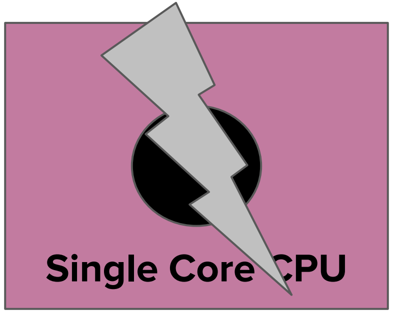
Threads ‘n cores

- In order for a **thread** to be able to execute some code, it must be running on a **CPU core.**
- If all **cores** are currently busy, a thread must **wait** for a **core** to free up before it can hop on that **core** and begin executing its own code!



Threads ‘n cores

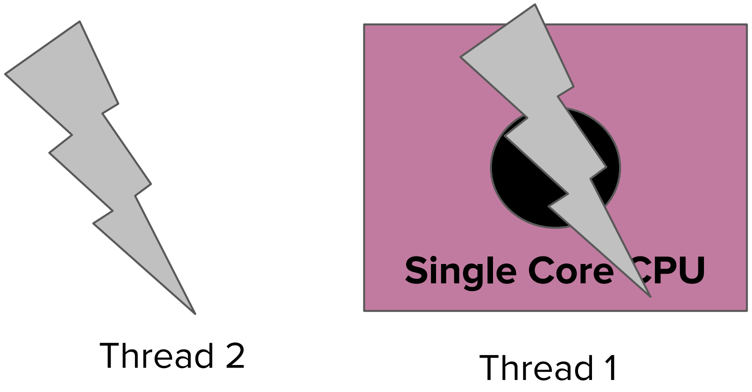
- In order for a **thread** to be able to execute some code, it must be running on a **CPU core.**
- If all **cores** are currently busy, a thread must **wait** for a **core** to free up before it can hop on that **core** and begin executing its own code!



Thread 1

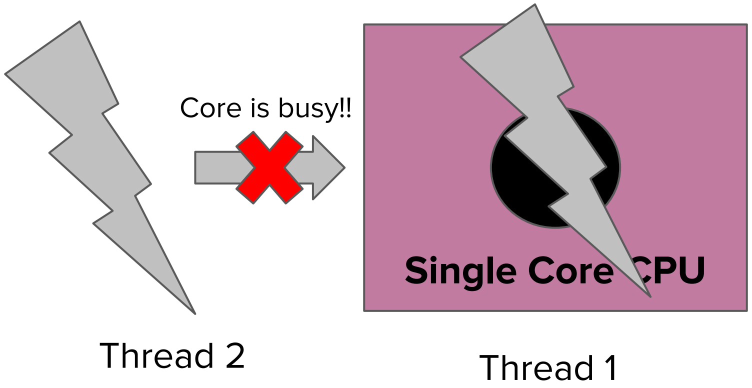
Threads ‘n cores

- In order for a **thread** to be able to execute some code, it must be running on a **CPU core.**
- If all **cores** are currently busy, a thread must **wait** for a **core** to free up before it can hop on that **core** and begin executing its own code!



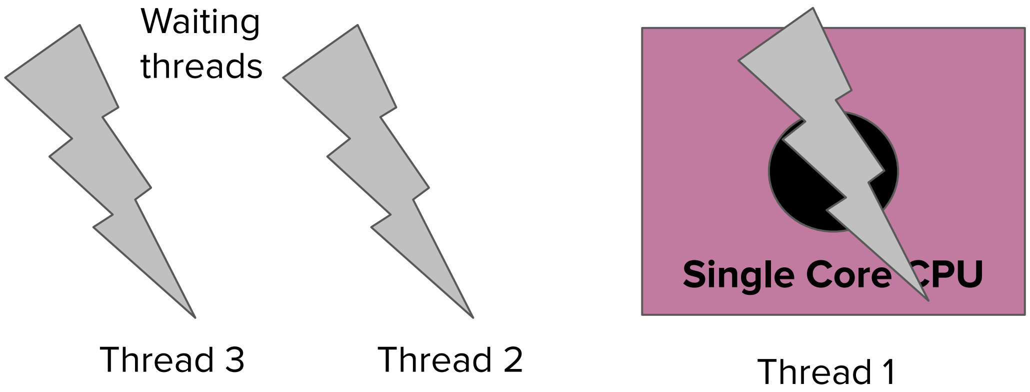
Threads ‘n cores

- In order for a **thread** to be able to execute some code, it must be running on a **CPU core.**
- If all **cores** are currently busy, a thread must **wait** for a **core** to free up before it can hop on that **core** and begin executing its own code!



Threads ‘n cores

- In order for a **thread** to be able to execute some code, it must be running on a **CPU core.**
- If all **cores** are currently busy, a thread must **wait** for a **core** to free up before it can hop on that **core** and begin executing its own code!



Question:

Who decides how long a thread should be able to run on a processor? Who decides which thread should run next?

What was running when the single-core was free in the example???

Definition

Operating System

Code that manages the relationship between a computer's **hardware** and **software.**

Thread Scheduling

- The **Operating System**, determines both **how long a thread should run** on a core, AND **which thread should run next.**
  - Want to learn how to implement these strategies? Take **an OS class like CS111!**
- For the purposes of this lecture, let's assume that a **thread** will run on a **core** until its program terminates or it is **forced off** the **processor**.
  - There are many reasons why a **thread** may be booted from a **core**: sometimes the **operating system** deems a thread needs to vacate its spot, and other times a thread will voluntarily yield its core.

Code example

- Let’s take a break from all of this low-level jazz and write a simple program!
- Let’s say you wanted to revise your **A2 Search Engine** program by ~~cheating and~~ making it ping the internet with actual queries.
    - Such a task is called **I/O Bound**, because the performance bottleneck is the waiting that happens between sending your request and getting your data! (We call this, and anything involving communication with the outside world, **I/O**)

Code example

- Let's write a program that repeatedly executes the below **I/O bound function**. (Forget the search engine thing; that's just an example of such a task).

```
static void task (int input);
```

- I've already implemented **task** for you; all you need to do is call it repeatedly! Let's do it!

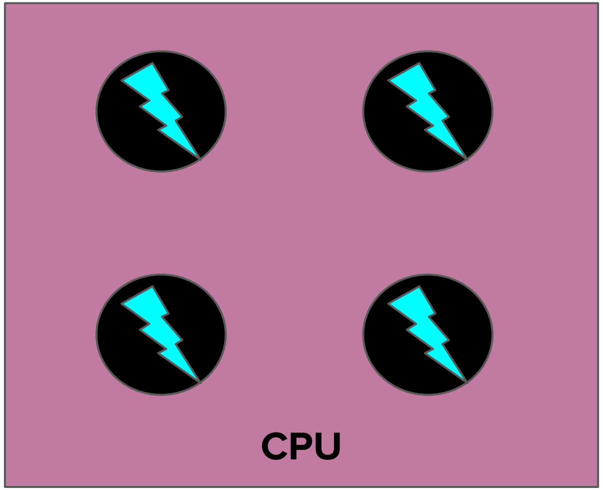
Code example

- What happened there?

Code example

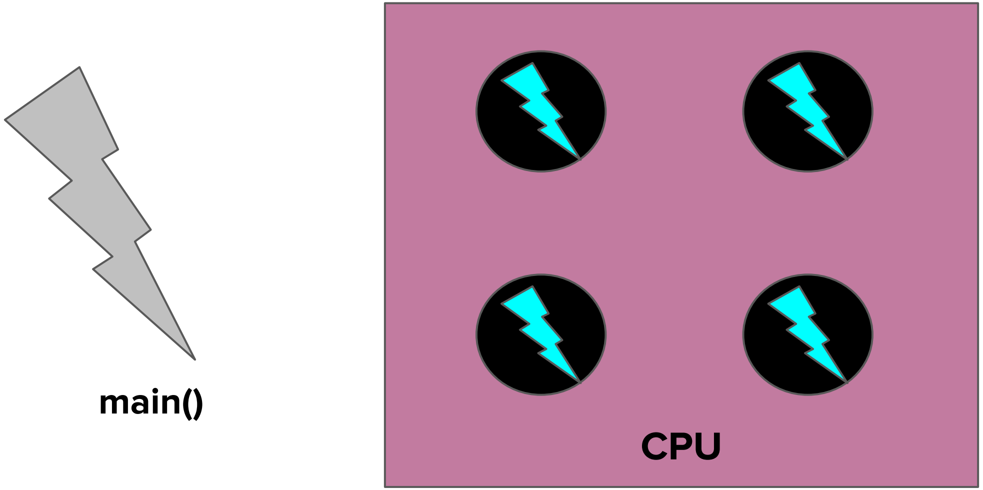
- What happened there?
  - Our code was slow as heck! This shouldn’t be surprising, however. Here’s what happened:

Code example: what happened?

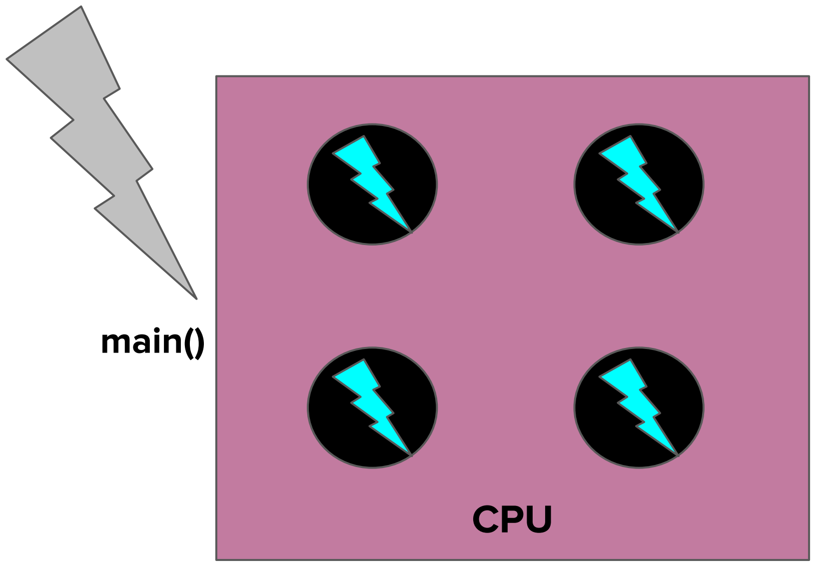


Before you run your program, your **CPU** is probably chugging away at other tasks!

Code example: what happened?

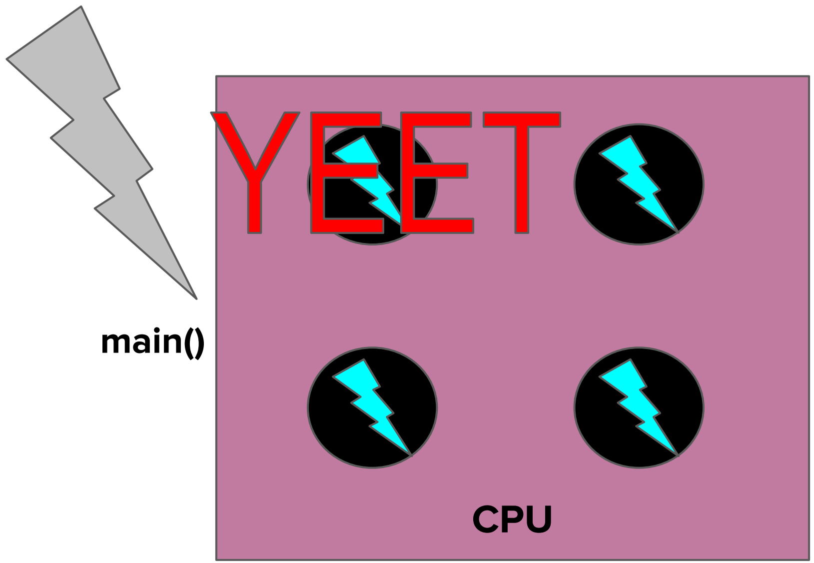


Code example: what happened?



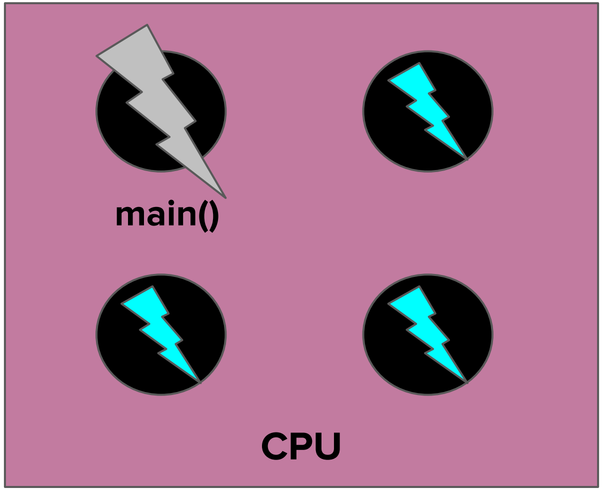
main() is a pretty important thread, so it has the power to boot another thread off a core!

Code example: what happened?



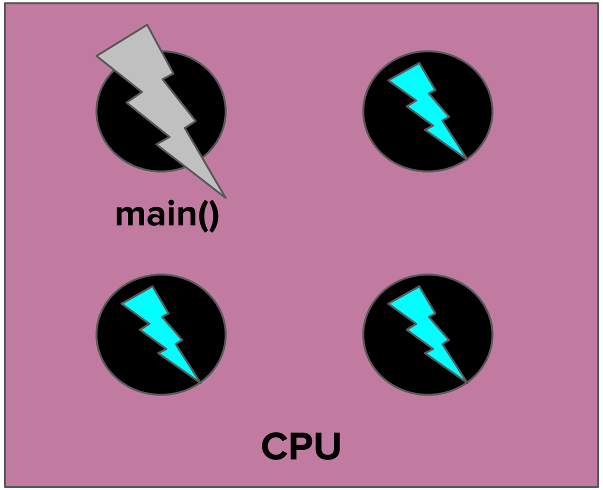
This transition is where your tuition money is going...

Code example: what happened?



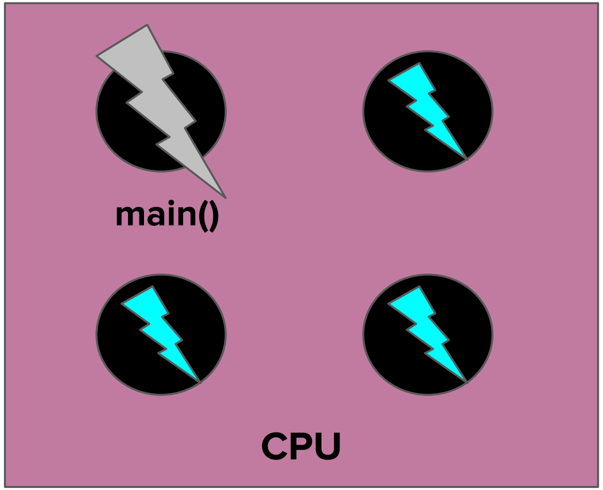
Code example: what happened?

- When you call the **I/O bound** function **task()** from **main()**, the **thread** will remove itself from the processor, as it is waiting on an **I/O** and therefore unable to do any work. Another **thread** will take its place immediately.



Code example: what happened?

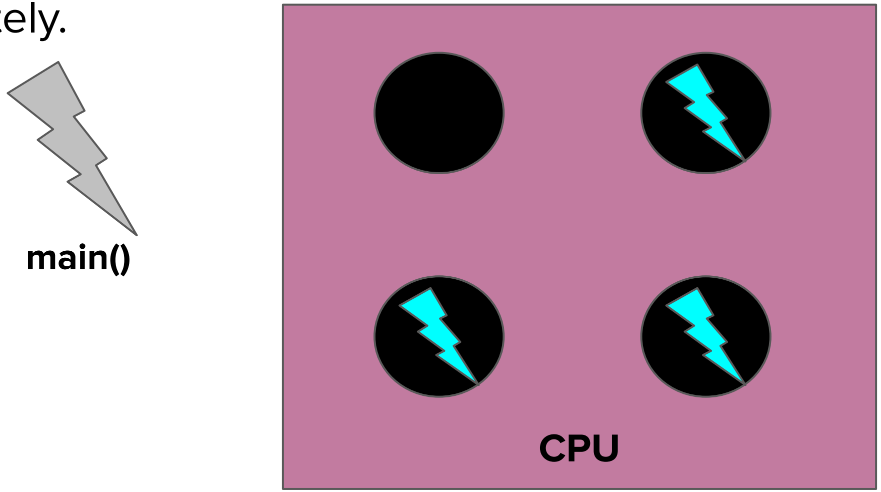
- When you call the **I/O bound** function **task()** from **main()**, the **thread** will remove itself from the processor, as it is waiting on an **I/O** and therefore unable to do any work. Another **thread** will take its place immediately.



Question for yourselves: why does self-removal make sense here?

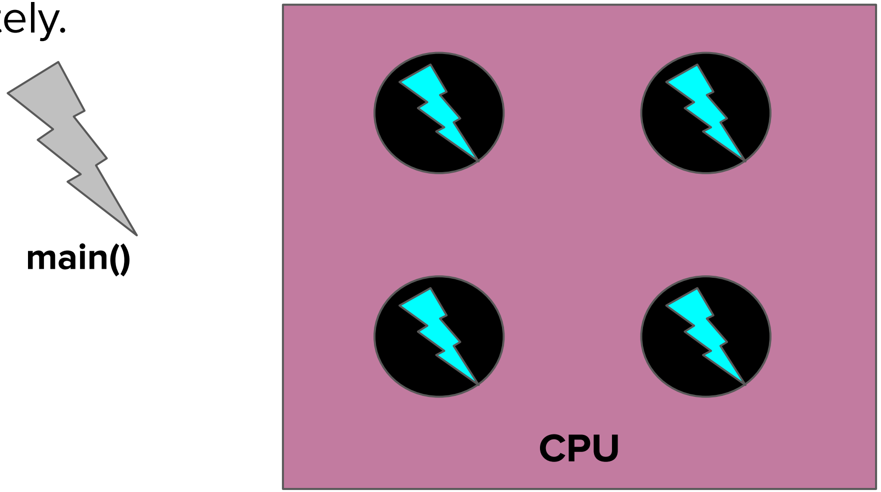
Code example: what happened?

- When you call the **I/O bound** function **task()** from **main()**, the **thread** will remove itself from the processor, as it is waiting on an **I/O** and therefore unable to do any work. Another **thread** will take its place immediately.



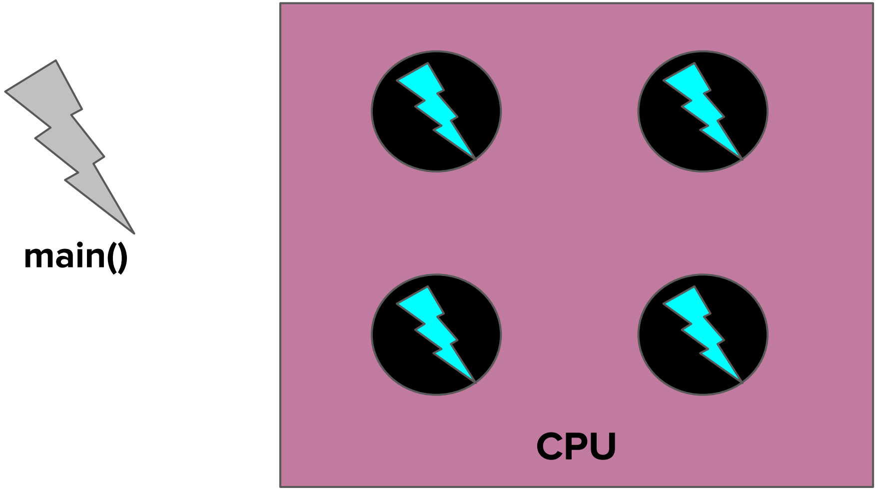
Code example: what happened?

- When you call the **I/O bound** function **task()** from **main()**, the **thread** will remove itself from the processor, as it is waiting on an **I/O** and therefore unable to do any work. Another **thread** will take its place immediately.



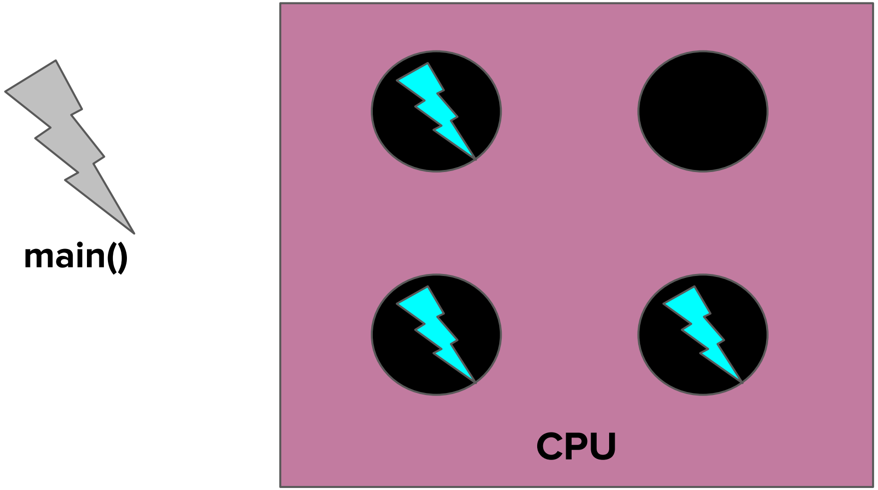
Code example: what happened?

- When the **I/O bound** task completes, your **thread** will attempt to get back on a core as soon as possible in order to continue (but its order in line is up to your **Operating System**)



Code example: what happened?

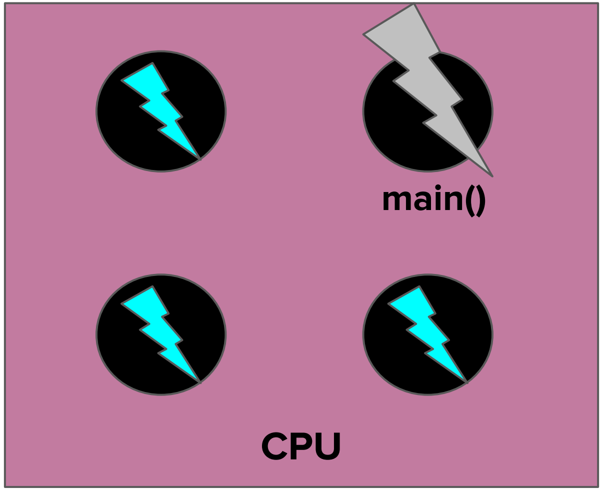
- When the **I/O bound** task completes, your **thread** will attempt to get back on a core as soon as possible in order to continue (but its order in line is up to your **Operating System**)



A vacancy!

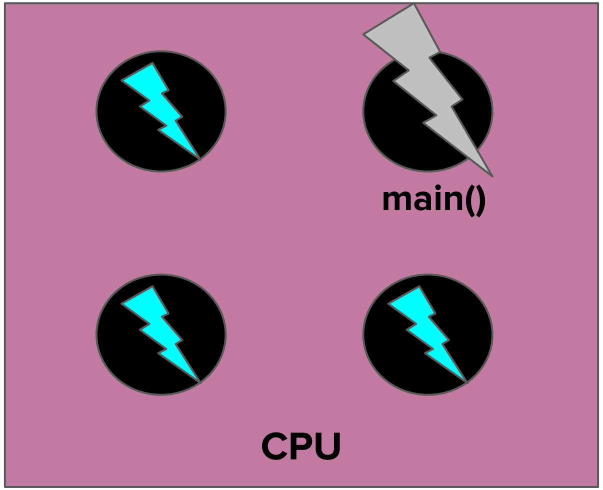
Code example: what happened?

- When the **I/O bound** task completes, your **thread** will attempt to get back on a core as soon as possible in order to continue (but its order in line is up to your **Operating System**)



Note how we're **core agnostic**. This doesn't need to be the case in some OS schedulers.

Questions about these events?



Code example: what happened?

- This process of getting on a **core**, **removing ourselves and waiting**, and reacquiring **a core** happened **every time** we called **task()**
  - Can we do better?

Code example: what happened?

- This process of getting on a **core**, **removing ourselves and waiting**, and reacquiring **a core** happened **every time** we called **task()**
  - Can we do better?
  - But first...

Announcements

Announcements

- Assignment 6 is due **Wednesday at 11:59pm PDT**. Remember that this is a hard deadline and there is no grace period!

- **There is no official section this week,** but keep an eye out for an email from your SL's in case they are hosting an optional section, or if the section time is being used for Final Project presentations.

- Make sure you've schedule your Final Project presentation. note that you will need to prepare SEPARATELY for the presentation (aka it's not just reading off of your writeup) - slides are encouraged, etc. See the Final Project handout on the website for more info.
- A5 revisions are due on Wednesday 8/10 (same day as A6). This is because our grading deadline is coming up quickly, so please keep that in mind if you're working on revisions!

Back to the action!

Code example: what happened?

- This process of getting on a **core**, **removing ourselves and waiting**, and reacquiring **a core** happened **every time** we called **task()**
  - Can we do better?

Code example: what happened?

- This process of getting on a **core**, **removing ourselves and waiting**, and reacquiring **a core** happened **every time** we called **task()**
  - Can we do better?
- In the words of a sectionee some years ago...
  - *"Let's parallelize this bad boy"*

Multithreading

- Let’s try and implement this same routine using **multithreading**.
    - That means we’ll try and use multiple threads instead of one in order to **parallelize** the workflow!

Multithreading

- Let's try and implement this same routine using **multithreading**.
    - That means we'll try and use multiple threads instead of one in order to **parallelize** the workflow!
- Before you can make threads, you'll **first** need to:

```
#include <thread>
```

- Bonus points: this is a **standard c++** library, so no Stanford-only woes!

Multithreading

- To instantiate a thread, it's pretty simple!

```
thread newthread = thread(funcName);
```

- This should look like a normal object declaration, except for the parameter!
    - *funcName* is the name of a the function you want to execute!

Multithreading

- To instantiate a thread, it's pretty simple!

```
thread newthread = thread(funcName);
```

- This should look like a normal object declaration, except for the parameter!
    - *funcName* is the name of a the function you want to execute!
    - In that case, let's create some `thread(task)'s!!`

Thread joining

- Woah woah woah, hold your horses, eager beaver.
- **As soon as you instantiate a thread, it begins to run**.

Thread joining

- Woah woah woah, hold your horses, eager beaver. Two things to think about:
  - **As soon as you instantiate a thread, it begins to run**. Be sure you’re ready before you dispatch them.
  - Threads are quite resource intensive, so when we dispatch them, we need to keep track of them so that we can clean up their memory once they’ve completed.
    - This is very much like the **new** and **delete** keywords you’ve used!

Thread joining

- After you’ve spawned a thread, simply call **threadName.join()** to clean it up.
  - This usually requires storing your threads in a collection! **Note**: Stanford’s Vector can’t store threads because it needs an update :(

Questions about creating / joining threads?

- You can call join() from your **main()** thread immediately after spawning the thread. Don't worry: join() will wait for your thread to finish :).

Questions about creating / joining threads?

- You can call join() from your **main()** thread immediately after spawning the thread. Don't worry: join() will wait for your thread to finish :).
- To pass params to a thread, just include them as the subsequent parameters.

```
thread newthread = thread(funcName, arg1, …);
```

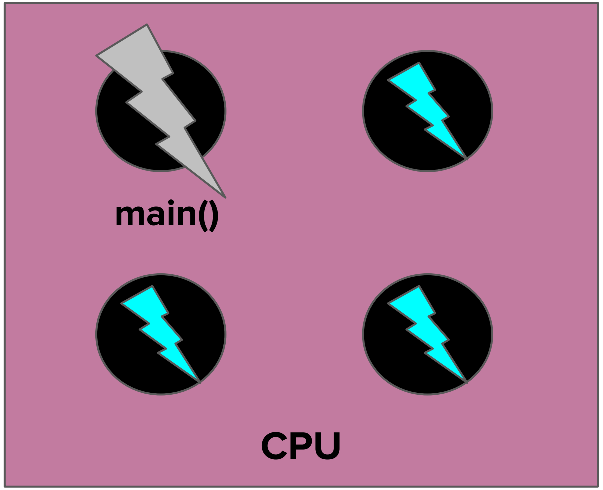
Let’s Parallelize!

What happened?

- Wow, that was super fast!

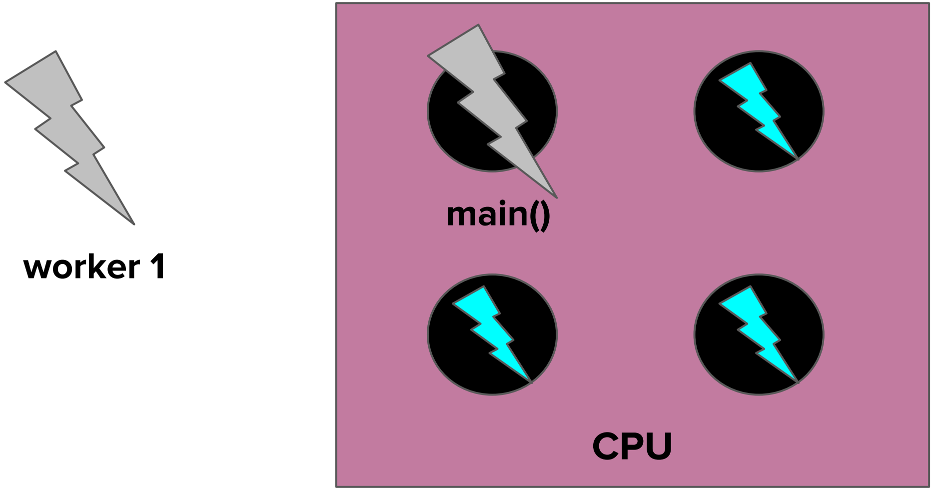
What happened?

- When our **main()** thread spawned up a new **thread**, the **new thread** might have taken a new core on the processor!
  - note* we don't know exactly what happened, but it could have done this!



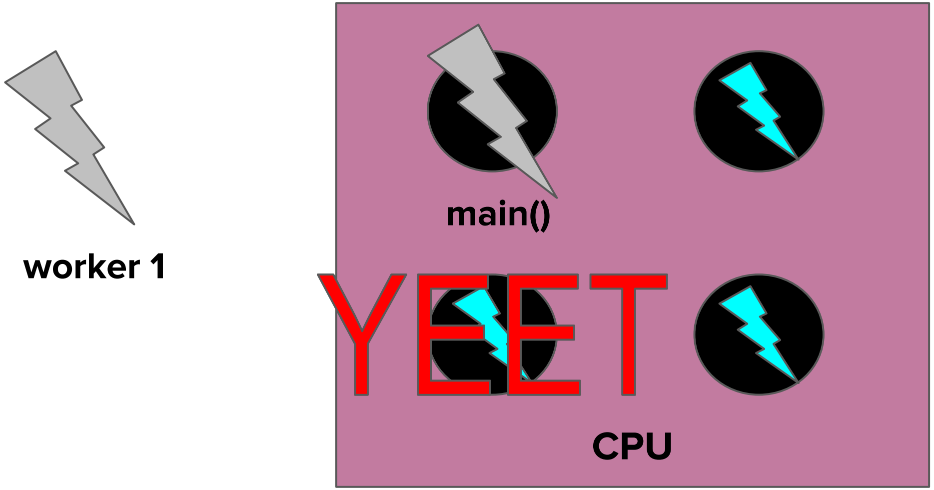
What happened?

- When our **main()** thread spawned up a new **thread**, the **new thread** might have taken a new core on the processor!
    - note* we don't know exactly what happened, but it could have done this!



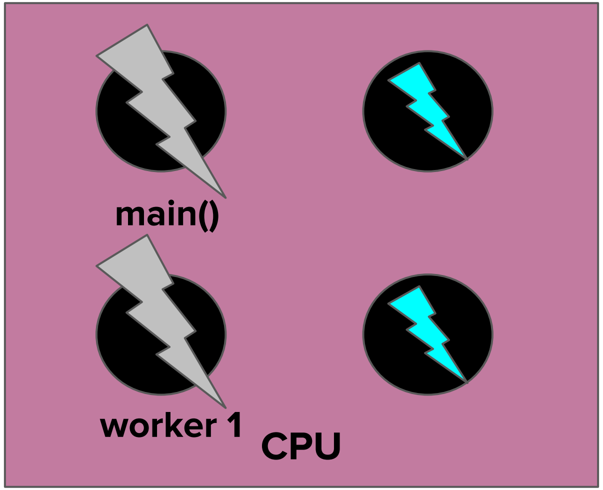
What happened?

- When our **main()** thread spawned up a new **thread**, the **new thread** might have taken a new core on the processor!
  - note* we don't know exactly what happened, but it could have done this!



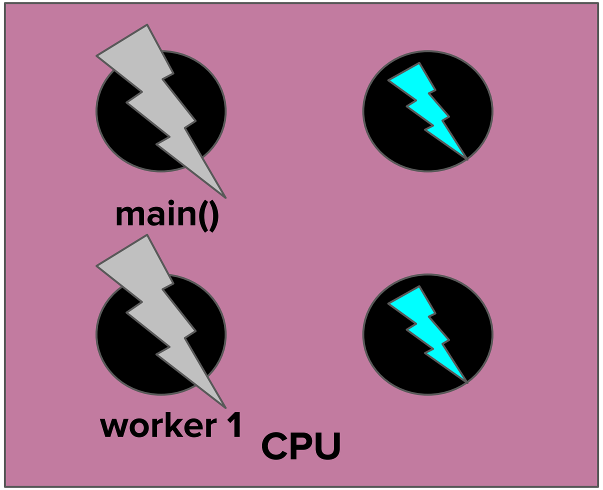
What happened?

- When our **main()** thread spawned up a new **thread**, the **new thread** might have taken a new core on the processor!
  - note* we don’t know exactly what happened, but it could have done this!



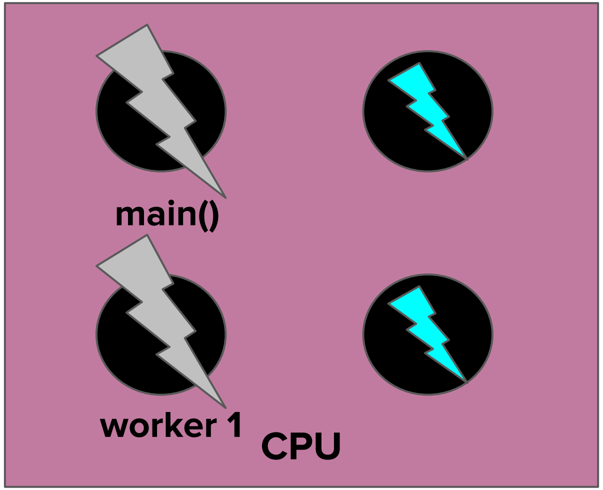
What happened?

- Note now that both **main()** and **worker 1** are running **concurrently!**



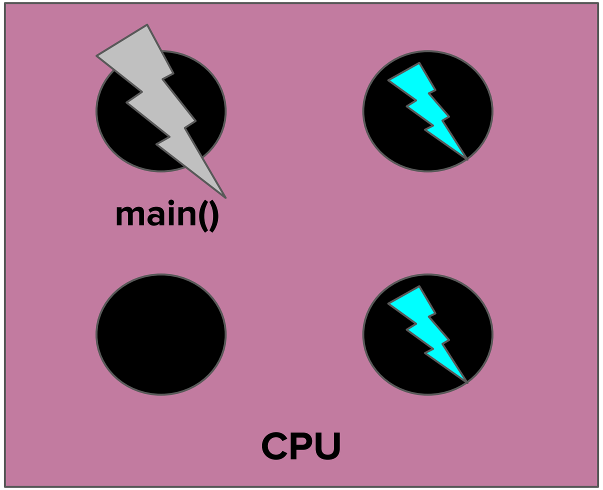
What happened?

- **Worker 1** will start its **I/O** and **remove itself from the core, getting replaced**



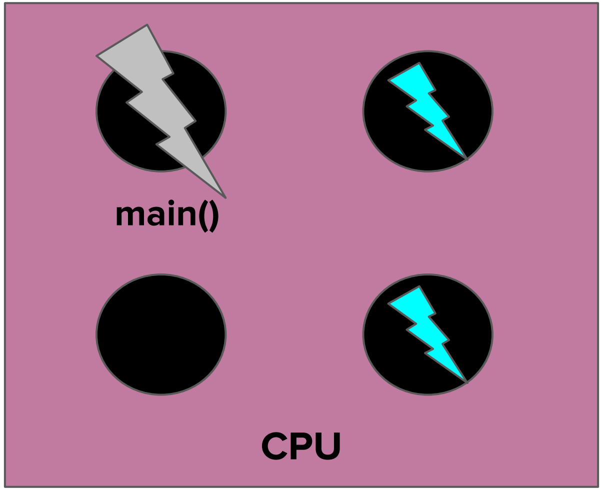
What happened?

- **Worker 1** will start its **I/O** and **remove itself from the core, getting replaced**



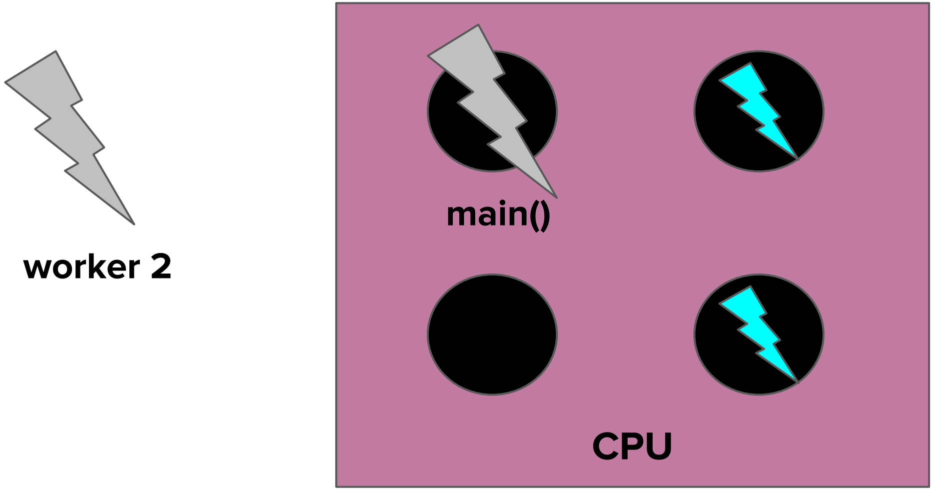
What happened?

- **Worker 1** will start its **I/O** and **remove itself from the core, getting replaced**
- But lo! Who is that in the distance?



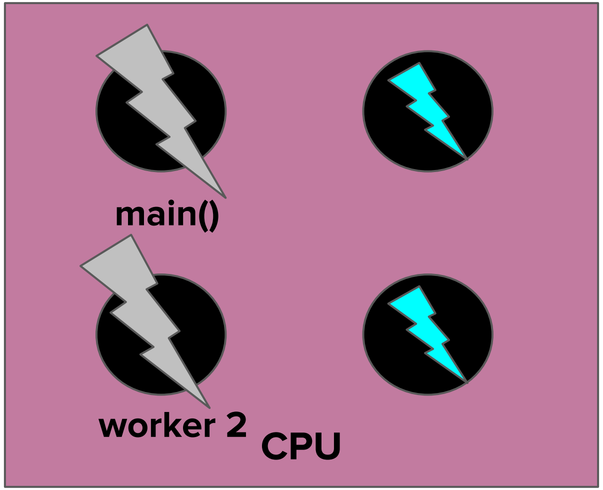
What happened?

- **Worker 1** will start its **I/O** and **remove itself from the core, getting replaced**
- But lo! Who is that in the distance?



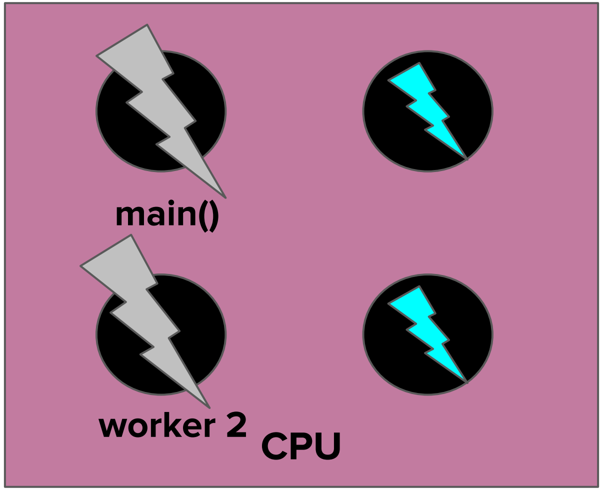
What happened?

- **Worker 1** will start its **I/O** and **remove itself from the core, getting replaced**
- But lo! Who is that in the distance?
- While **worker 1** was waiting for its I/O, **main()** was busy spinning up new threads!



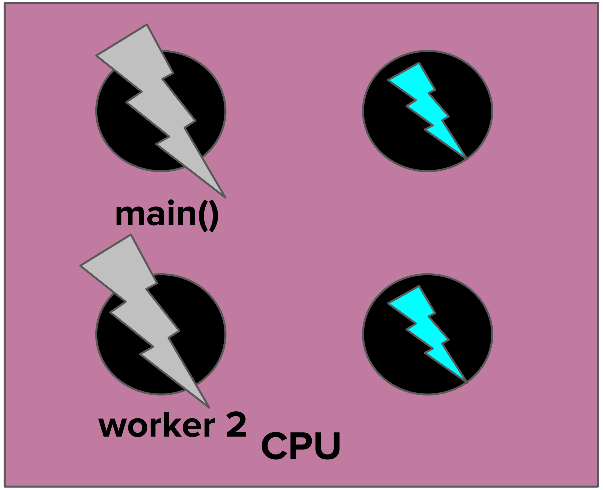
What happened?

- This process will continue -- each **worker thread** will only need to be on a core for a fraction of a second, just to set up the **I/O**, and then it can leave the processor and let a new **worker thread** set up its **I/O**.



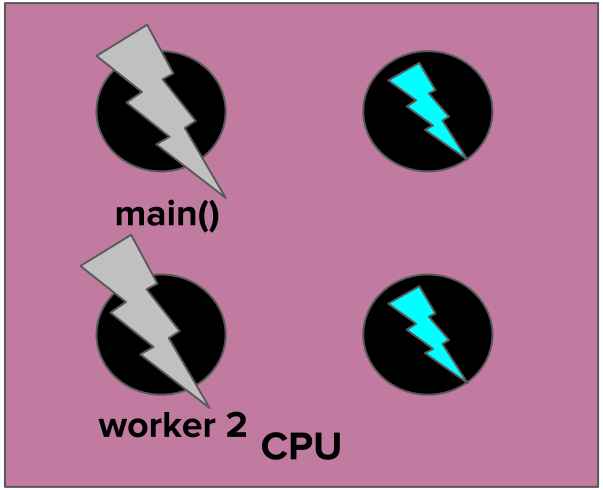
What happened?

- A similar thing will happen at completion time!
    - Each **thread** will be able to retake a core, but the core will only be needed for a few instructions! Then the **task()** will finish, and a new **thread** will try and complete!



What happened?

- A fair warning -- you can't predict which worker thread will begin working first! It might seem like **worker 1** should always start first, but the OS and CPU work in unpredictable ways!



What happened?

- The example you saw was blazing fast because the **task** at hand only needed to be on the processor for a **short period of time**.
- As you can see, the process of yielding a core to another worker takes an almost imperceptible amount of time!
  - That's because your OS is doing it constantly :o
- Parallelization is less successful when you don't have long **I/O** waits.

Questions?

Race Conditions

- Remember when I said that we can’t really determine the order that threads will run in? Let’s show that!
- Let’s add **logging** to our code to show the order that threads show up!
- It’s easy! Just add a print statement inside **task()** and keep a counter variable!

Race Conditions

- Remember when I said that we can’t really determine the order that threads will run in? Let’s show that!
- Let’s add **logging** to our code to show the order that threads show up!
- It’s easy! Just add a print statement inside **task()** and keep a counter variable!

- Let’s try it!

What happened??

Attendance ticket:

https://tinyurl.com/lec25ticket

Please don’t send this link to students who are not here. It’s on your honor!

Definition

Race Condition

A bug that is the product of two threads "racing" against each other and operating on the same state in the incorrect order.

Bonus: Race Conditions

- Congratulations, you’ve experienced your first **race condition!**
- It turns out that **cout** is not **thread-safe**, meaning that it will not behave predictably if you have multiple threads calling it at the same time!
    - Every time you printed to the console, you had some jumbling of all 10 cout statements!

Bonus: Race Conditions

- Congratulations, you've experienced your first **race condition!**
- It turns out that **cout** is not **thread-safe**, meaning that it will not behave predictably if you have multiple threads calling it at the same time!
    - Every time you printed to the console, you had some jumbling of all 10 cout statements!

- How can we fix this?

Definition

Atomic

A state that can only be observed or superseded **before** or **after** an operation occurs, **not during.**

Mutex

- To make code **atomic**, we can use something called a **mutex.**
    - Sounds like Mut(ual) Ex(clusion)!

Mutex

- To make code **atomic**, we can use something called a **mutex.**
  - Sounds like Mut(ual) Ex(clusion)!
- To make a mutex, you'll need this library:

```
#include <mutex>
```

- and you'll want to declare a single mutex like this:

```
mutex m;
```

Mutex

- You'll want to make a **single mutex**, and pass it as a **pointer** to your worker threads. (This is because threads need to **share** the **mutex**)

```
thread t = thread (funcName, &mutexName);
```

- In order to make code **atomic**, all you need to do is wrap the code in question around these two statements:

```
mutexName->lock();

mutexName->unlock();
```

Mutex

- In order to make code **atomic**, all you need to do is wrap the code around these two statements:

```
mutexName->lock();

mutexName->unlock();
```

- When you **lock** a **mutex**, any other threads trying to lock that **mutex** will be forced to wait until you **unlock** it.
    - Once you **unlock**, the **Operating System** decides which thread can **lock** the **mutex** next!

Let’s try it!

We’re still not done!?

- Why is everything 10?

```
Setting the program up...
Let's process 10 numbers!
Starting in 3...
2...
1...
GO!!
Hello from worker 10
Hello from worker 10
Hello from worker 10
Hello from worker 10
Hello from worker 10
Hello from worker 10
Hello from worker 10
Hello from worker 10
Hello from worker 10
Hello from worker 10
All done! The total time spent working was 1353 milliseconds (roughly 1 second!)
```

We’re still not done!?

- Remember how we passed **id** by reference? (using a pointer)
- The problem is that **the threads** **share** the variable “i”
- This actually indicates that **main()** finished the for loop that created **all ten threads** (therefore increasing **i** to the max value) before a **single worker could complete**.
  - This should make sense because even the first worker had to wait a full second before it could print anything!

We're still not done!?

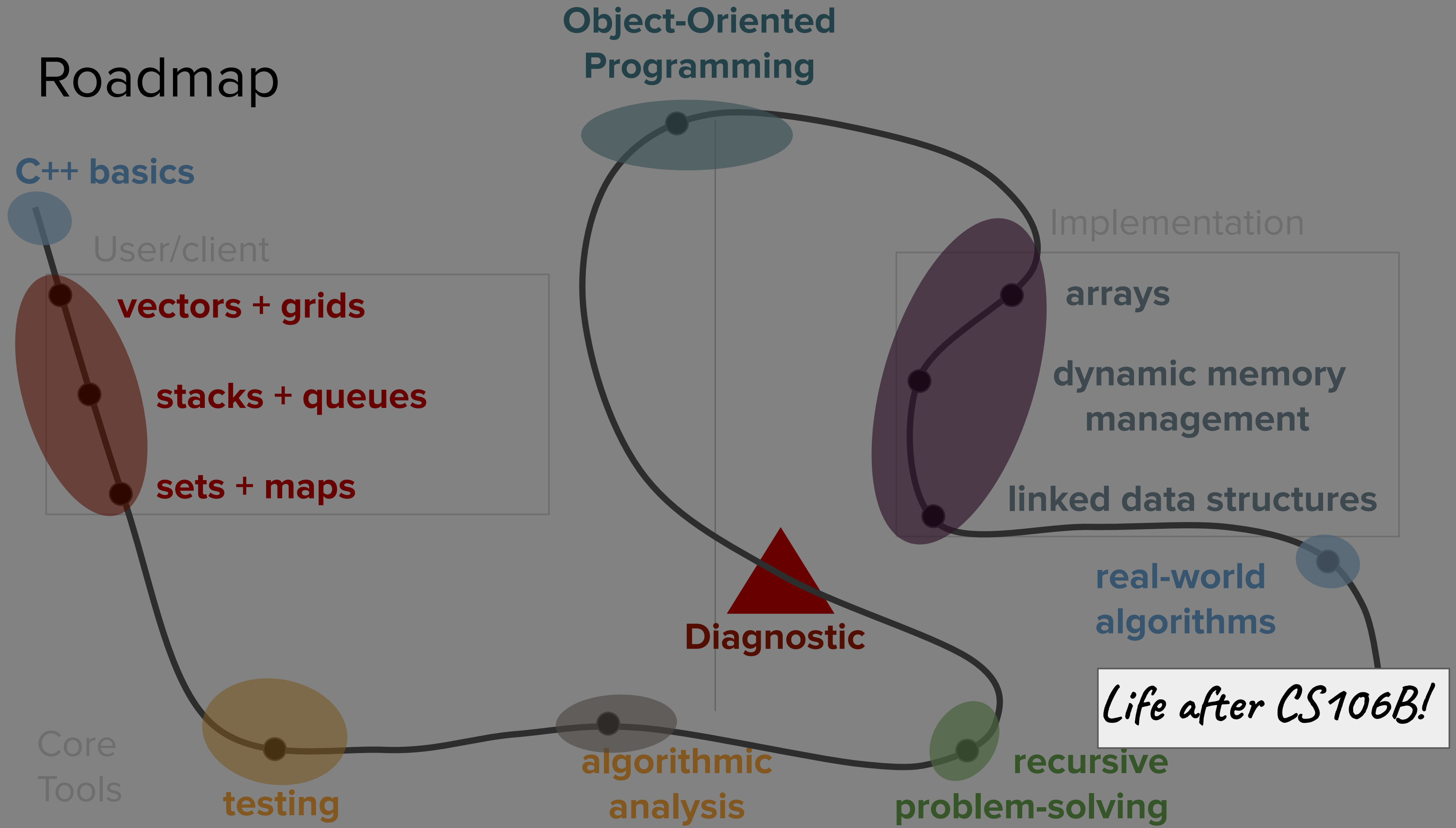
- Remember how we passed **id** by reference? (using a pointer)
- The problem is that **the threads share** the variable "i"
- This actually indicates that **main()** finished the for loop that created **all ten threads** (therefore increasing **i** to the max value) before a **single worker could complete**.
  - This should make sense because even the first worker had to wait a full second before it could print anything!
- How do we fix this?

Final thoughts

- Multithreading is an incredibly powerful tool that lets you parallelize work among your CPU’s cores.
- Threads are a fundamental building block of computing that play an important role in **Operating Systems!**
- When using multiple threads, be wary of **any** data that is **shared** between them.
  - Using a **mutex** allows you to enforce **atomicity** in sections of code, but sometimes even that isn’t enough!
  - If all of your code is **atomic**, there’s no parallelization at all!
- If you liked this topic, take Computer Systems courses to go into more depth :)

What’s next?

Roadmap



Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving