Multithreading and Parallel Computing

What's an example of multitasking that you do in your everyday life?

Today's question

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

woah.

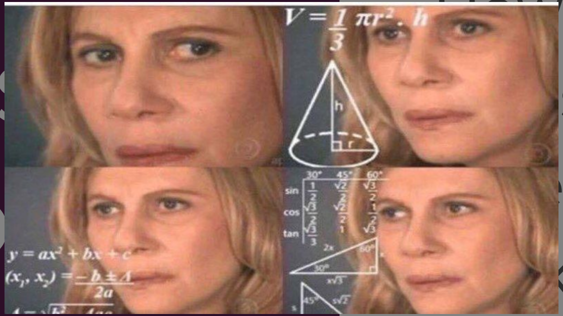
Today's question

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

Multiple cores?

Parallelize work??

Today's questio



How can we harness the
s in our computer in
r to parallelize a
kload safely?

Today's topics

1. Review (short!)
2. Some Computer Architecture (Threads & Processors)
3. Multithreading Perils (If we have time!)

Review (short!)

(simple code flow)

How code is run

- At a *high level*, how does the computer run your code?
  - Logically, **it** should interpret 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;
}
```

How code is run

- How does the computer read an[d] 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 abstraction that encapsulates and executes your main() function?

Definition

thread

An abstraction that represents a sequential execution of code.

Definition

Line by line, top to bottom!

thread

An abstraction that represents a sequential execution of code.

Definition

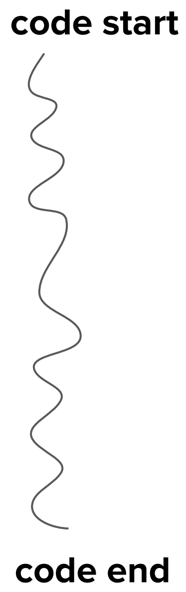
thread

An abstraction that represents a sequential execution of code.

Anything that's code!

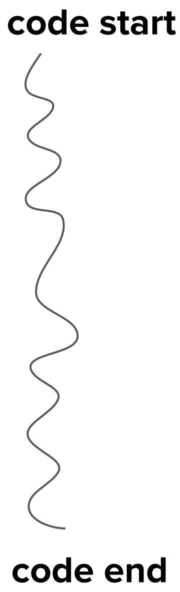
How to think about threads

- When talking about a **thread**, you'll very frequently see it referenced as a "**thread** of execution."



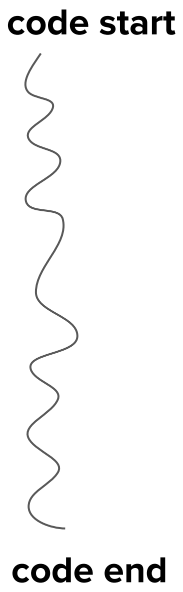
How to think about threads

- When talking about a **thread**, you'll very frequently see it referenced as a "**thread** of execution."
  - Think about the line on the right as a program's execution. You start at **main()**, which might call other functions, which might return to **main()** or call other helper functions. Although the execution flow of your program may involve many function calls, it will eventually go from the top of **main()** to the bottom.



How to think about threads

- When talking about a **thread**, you'll very frequently see it referenced as a "**thread** of execution."
  - Think about the line on the right as a program's execution. You start at **main()**, which might call other functions, which might return to **main()** or call other helper functions. Although the execution flow of your program may involve many function calls, it will eventually go from the top of **main()** to the bottom.
  - The flow would almost looks like a **thread**, or a piece of string!



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

- Are you on Zoom right now?

Thread Examples

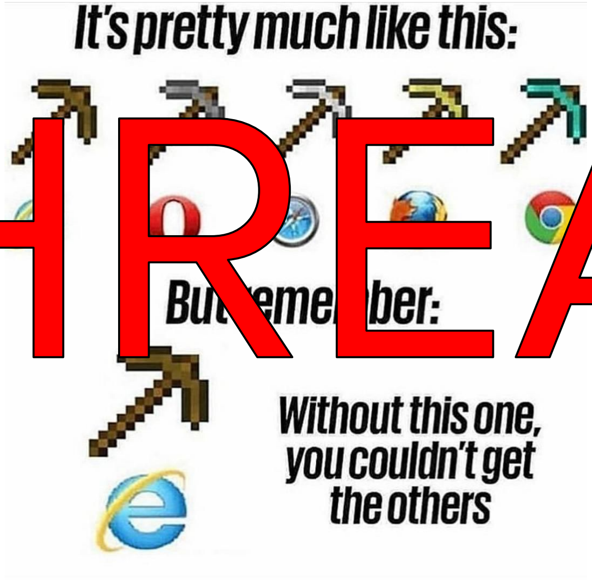
- Are you on Zoom right now?

Thread Examples

- Do you have a web browser open?

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)

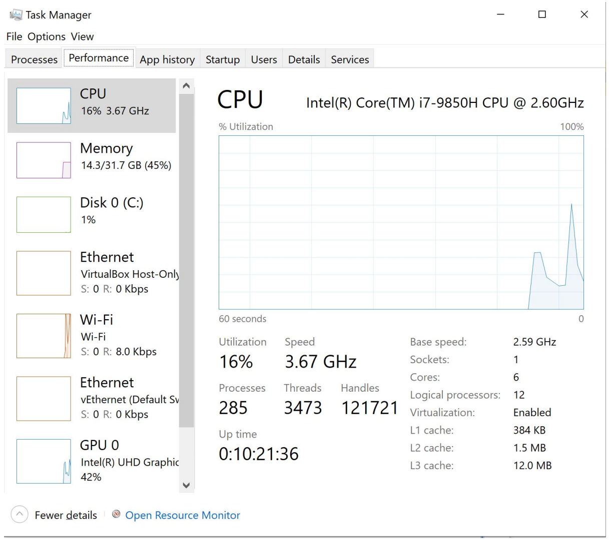
I have been told Ms. D'Ameli is a TikTok #influencer

Question:

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

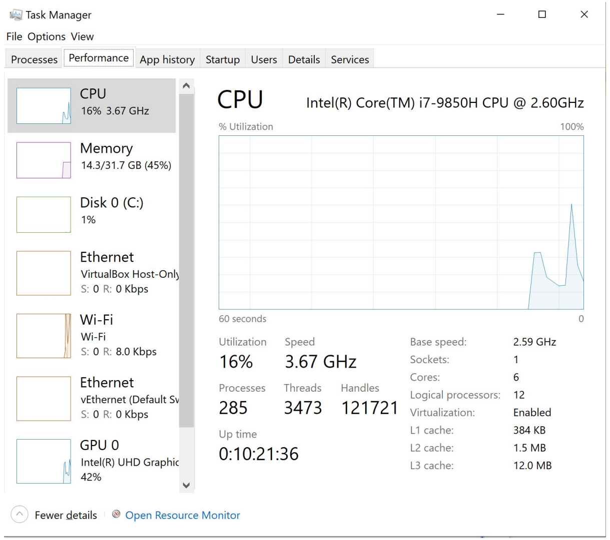
Thread examples

- Right now, your computer is executing a bunch of threads!
    - At the time of making this slide show, my computer was handline 3473 threads!



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 abstractions (code). Not a physical entity.

hardware
Physical parts of a computer.

The hardware-software boundary

- A thread **alone** cannot run your program.

The hardware-software boundary

- A thread **alone** cannot run your program.
  - A thread is just **software** that is an **abstraction** for some code.

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!

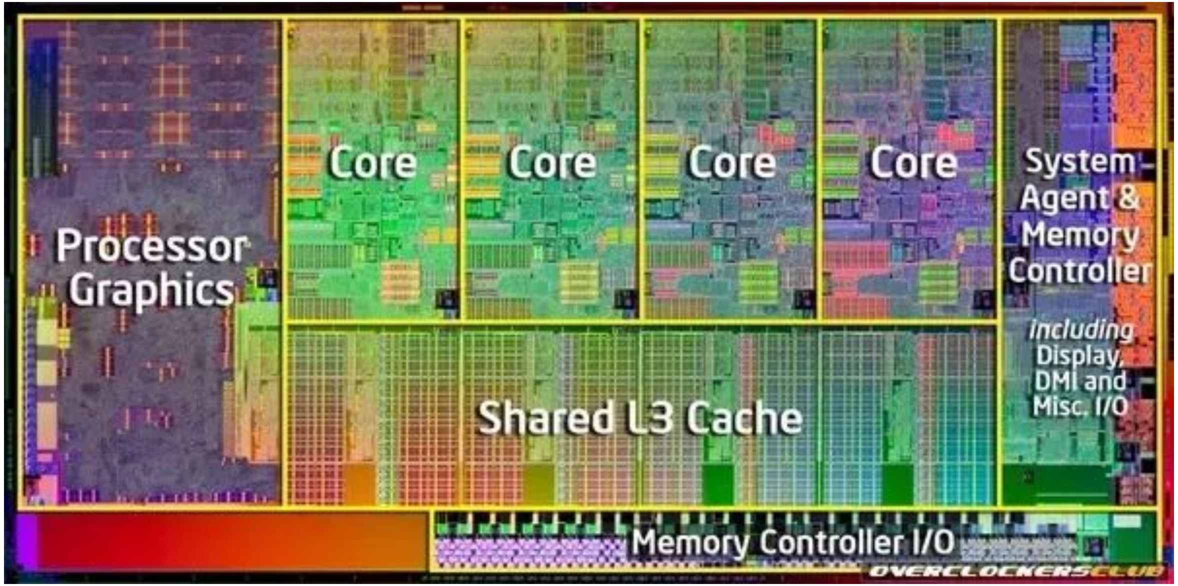
... but what piece of hardware does this?

Definitions

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

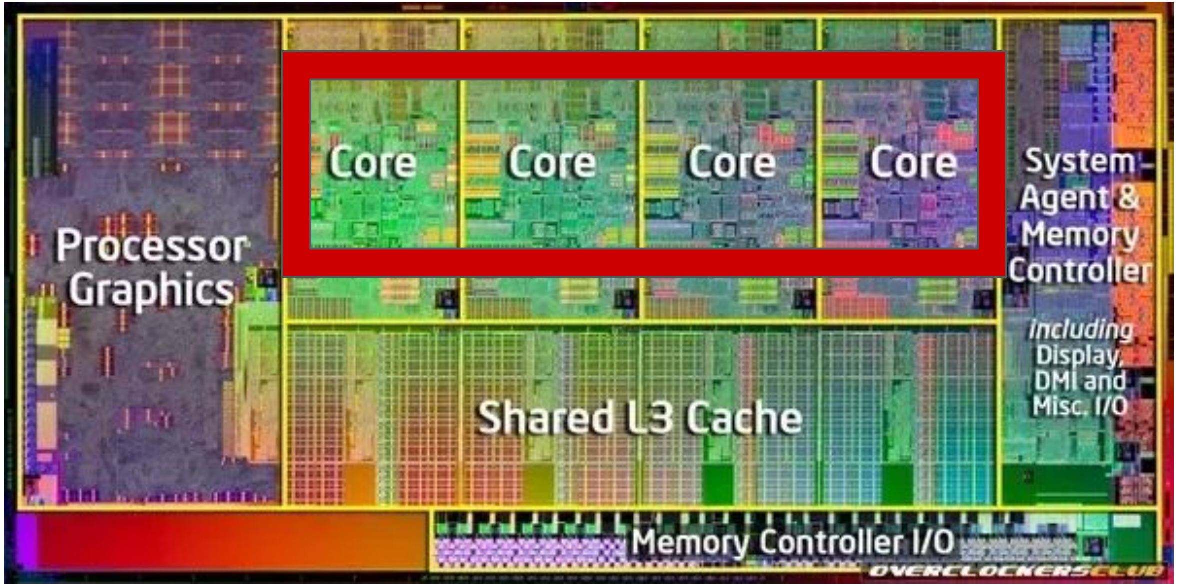
Core
An individual processor inside of a **CPU**. Each **core** is able to execute code independently of other **cores**.

Inside a CPU...



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

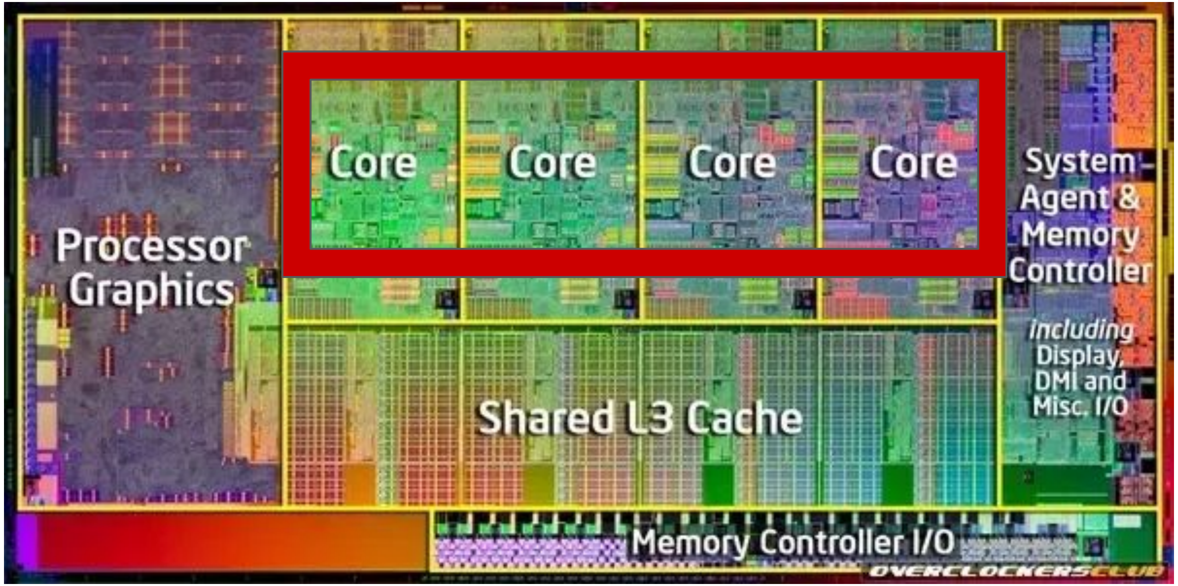
Inside a CPU…



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

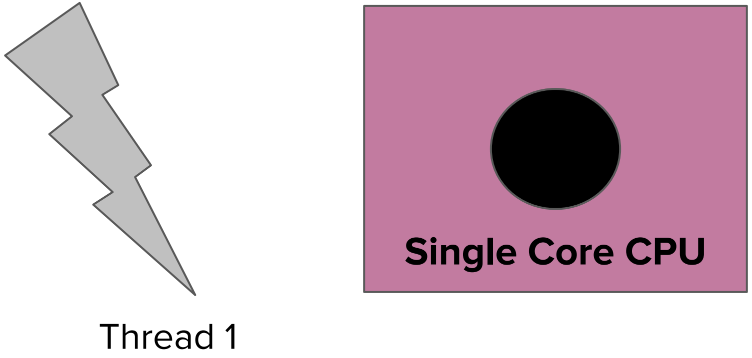
- In order for a **thread** to be able to execute some code, it must be running on a **CPU core.**

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

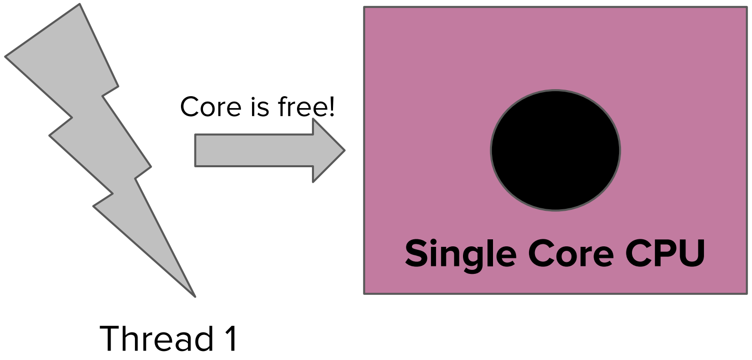
- In order for a **thread** to be able to execute some code, it must be running on a **CPU core.**
- If all **cores** are currently busy, a thread must **wait** for a **core** to free up before it can hop on that **core** and begin executing its own code!



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

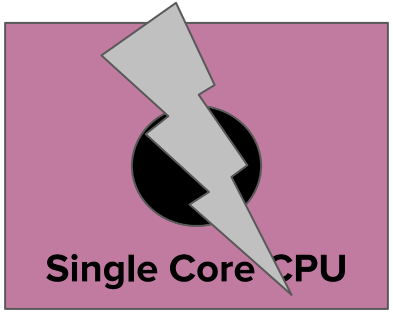
Threads ‘n cores

- 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

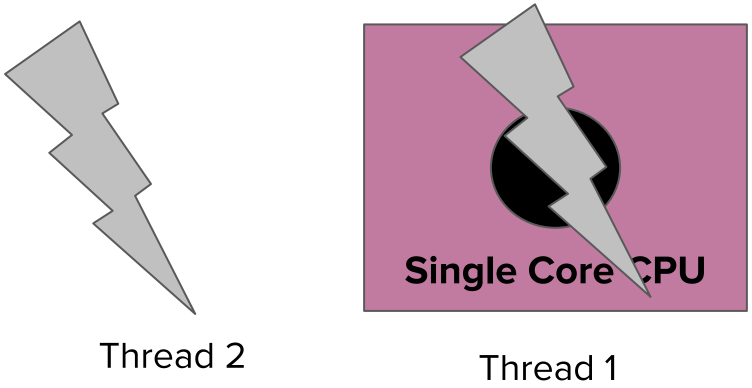
- In order for a **thread** to be able to execute some code, it must be running on a **CPU core.**
- If all **cores** are currently busy, a thread must **wait** for a **core** to free up before it can hop on that **core** and begin executing its own code!



Thread 1

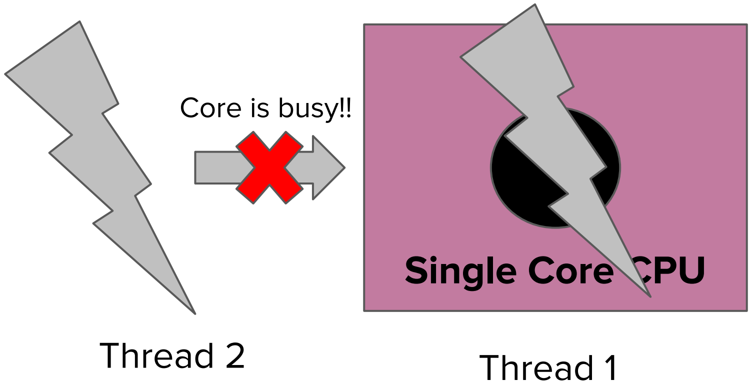
Threads ‘n cores

- 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!



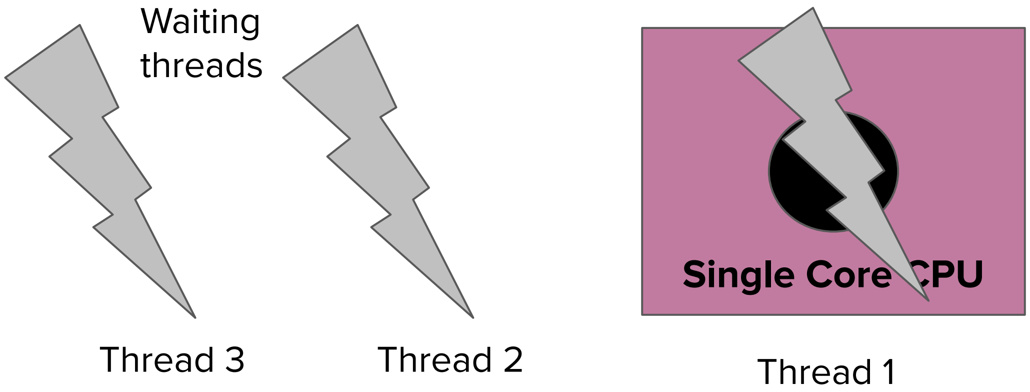
Threads ‘n cores

- 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 program 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.**

Thread Scheduling

- The **Operating System**, determines both **how long a thread should run** on a core, AND **which thread should run next.**

Thread Scheduling

- The **Operating System**, determines both **how long a thread should run** on a core, AND **which thread should run next.**
- A **thread** will run on a **core** until its program terminates or it is **forced off** the **processor** by the Operating System.

Thread Scheduling

- The **Operating System**, determines both **how long a thread should run** on a core, AND **which thread should run next.**
- A **thread** will run on a **core** until its program terminates or it is **forced off** the **processor** by the Operating System.
  - 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!

Code example

- Let’s take a break from all of this low-level jazz and write a simple program!
- Let’s say I wanted to call this **non-computational**, but **expensive** function a certain number of times:

```
void task (int id);
```

Code example

- Let’s take a break from all of this low-level jazz and write a simple program!
- Let’s say I wanted to call this **non-computational**, but **expensive** function a certain number of times:

```
void task (int id);
```

- This function sends some data to a server over the internet and waits for a response. This is called an **I/O Bound** task, because the slowness of the function does not depend on the speed of the CPU.

Code example

- Let’s take a break from all of this low-level jazz and write a simple program!
- Let’s say I wanted to call this **non-computational**, but **expensive** function a certain number of times:

```
void task (int id);
```

- This function sends some data to a server over the internet and waits for a response. This is called an **I/O Bound** task, because the slowness of the function does not depend on the speed of the CPU.

Code example

- Let’s take a break from all of this low level jazz and write a simple program!
- Let’s say I wanted to ca **utational**, but **expensive** function a certain numb

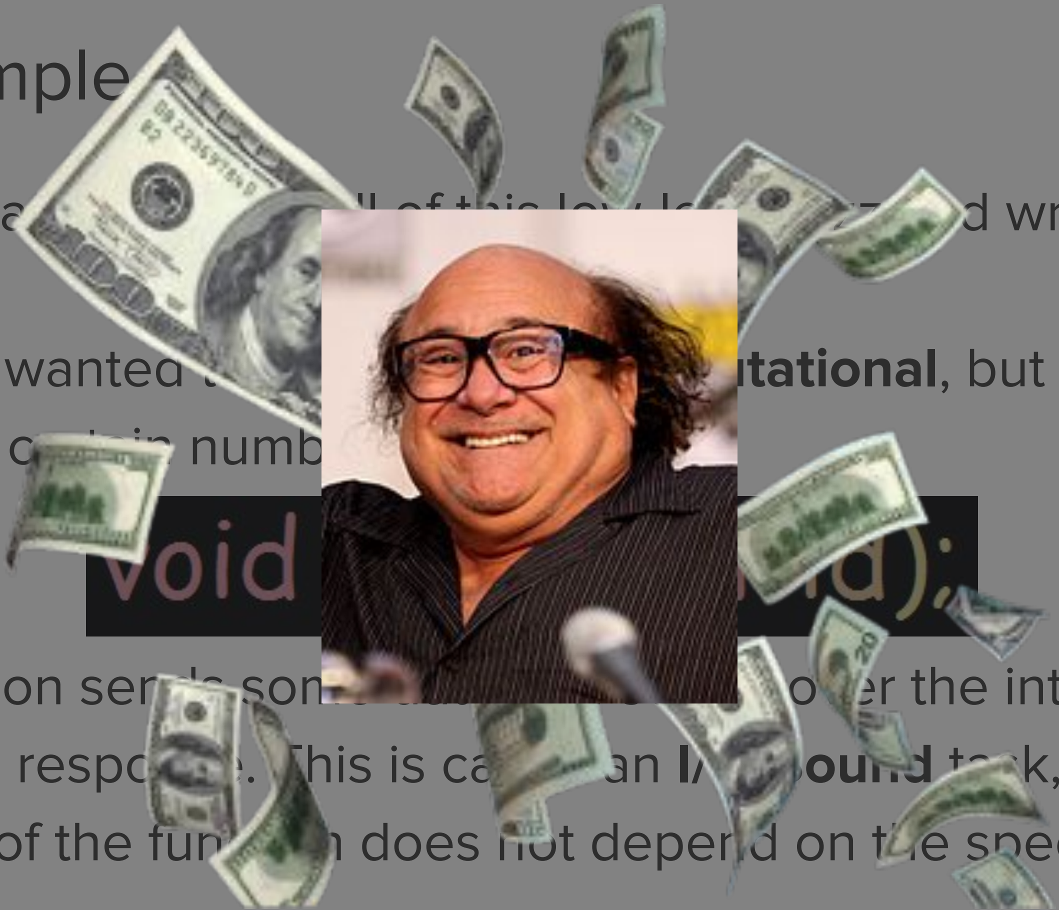
```
void  t id);
```

- This function sends som er over the internet and waits for a response. This is called an **I/O Bound** task, because the slowness of the function does not depend on the speed of the CPU.

Code example

- Let’s take a [illegible] d write a simple program!
- Let’s say I wanted t [illegible] utational, but **expensive** function a c [illegible] numb [illegible]

```
void [illegible] d);
```



- This function sen [illegible] er the internet and waits for a respo [illegible] e. This is ca [illegible] an **I/[illegible] bound** task, because the slowness of the fun [illegible] n does not depend on the speed of the CPU.

Code example

- I’ve already implemented **task** for you; all you need to do is call it repeatedly and see how long it takes!

```
void task (int id);
```

Code example

- I’ve already implemented **task** for you; all you need to do is call it repeatedly and see how long it takes!
- **Let’s code it up!**

```
void task (int id);
```

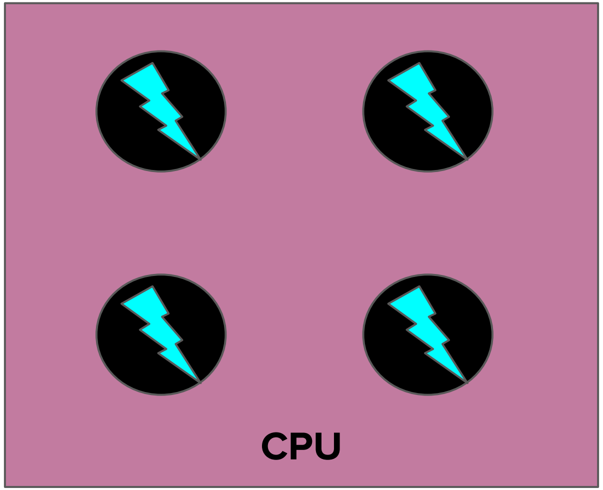
Code example

- What happened there?

Code example

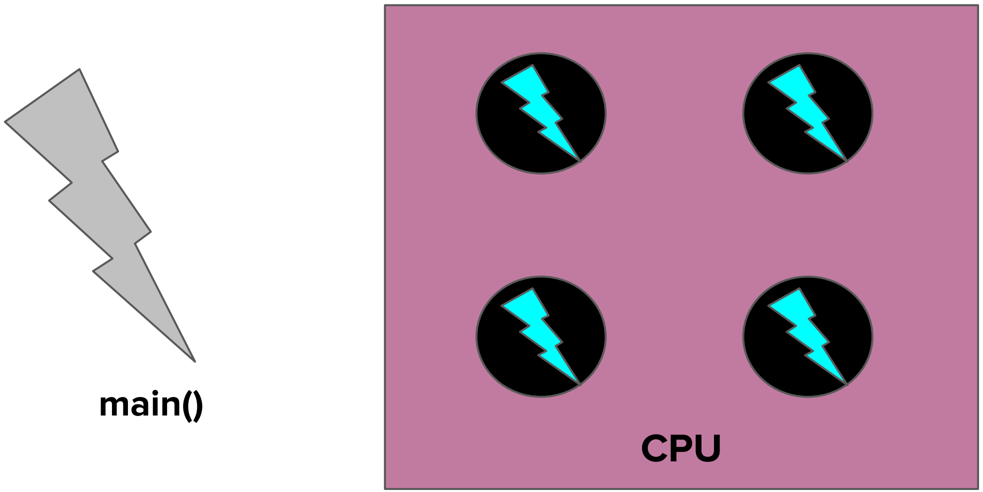
- What happened there?
  - Our code was slow as heck! This shouldn’t be surprising, however. Here’s what happened:

Code example: what happened?

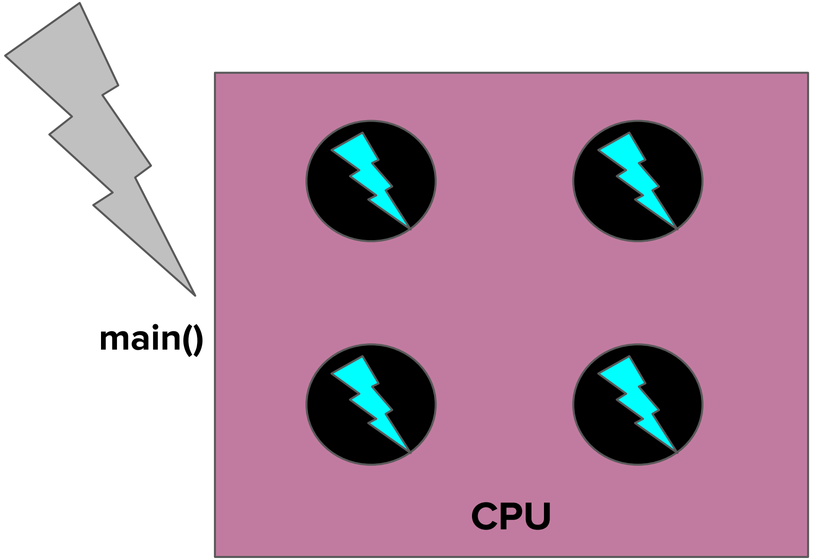


Before you run your program, your **CPU** is probably chugging away at other tasks!

Code example: what happened?

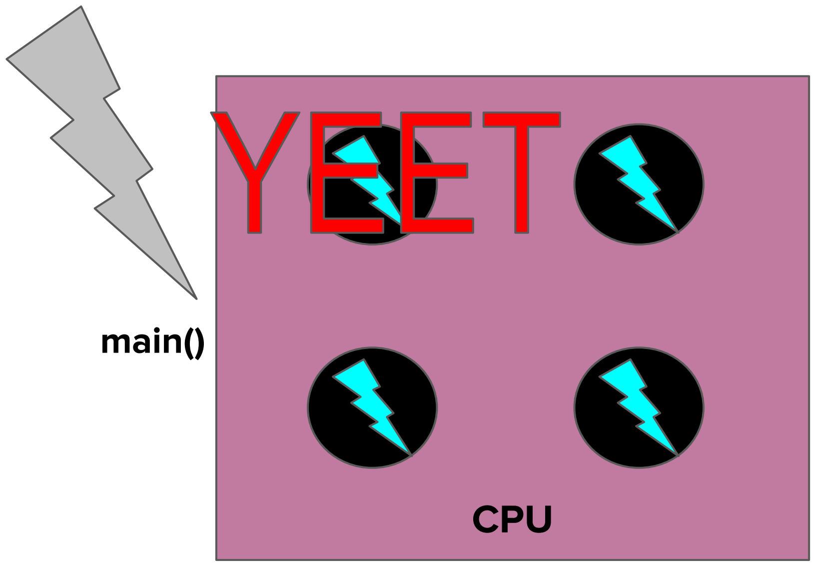


Code example: what happened?



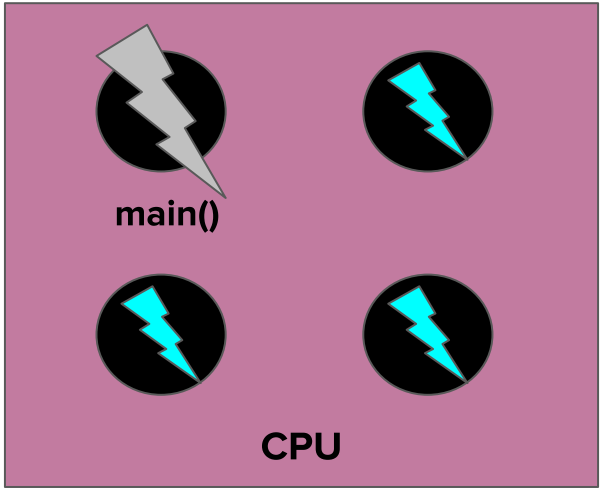
main() is a pretty important thread, so it has the power to boot another thread off a core!

Code example: what happened?



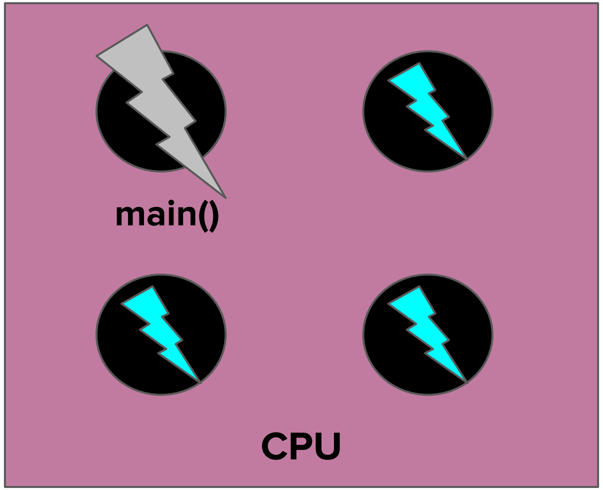
This transition is where your tuition money is going...

Code example: what happened?



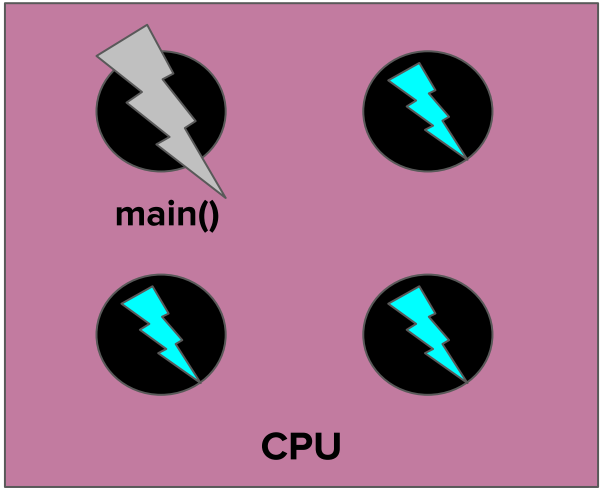
Code example: what happened?

- When you call the **I/O bound** function **task()** from **main()**, the **main() 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?

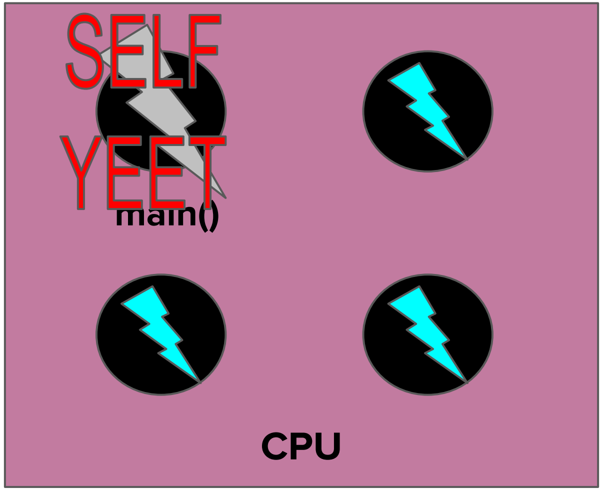
- When you call the **I/O bound** function **task()** from **main()**, the **main() thread** will remove itself from the processor, as it is waiting on an **I/O** and therefore unable to do any work. Another **thread** will take its place immediately.



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

Code example: what happened?

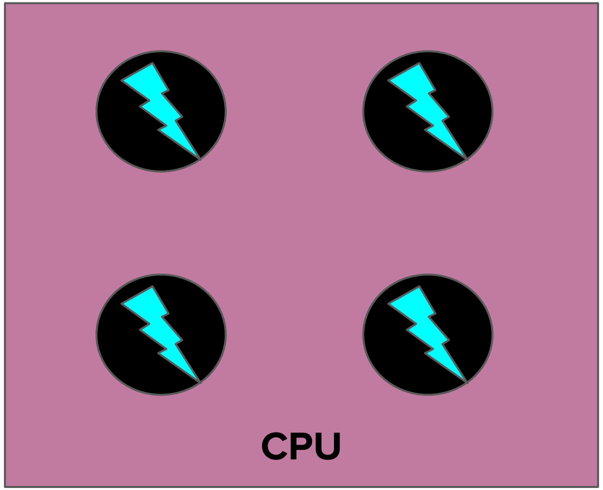
- When you call the **I/O bound** function **task()** from **main()**, the **main() thread** will remove itself from the processor, as it is waiting on an **I/O** and therefore unable to do any work. Another **thread** will take its place immediately.



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

Code example: what happened?

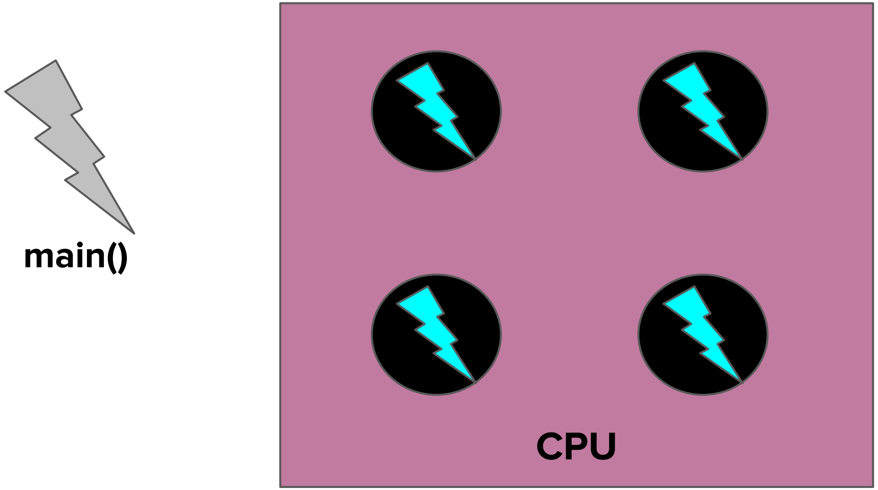
- When you call the **I/O bound** function **task()** from **main()**, the **main() thread** will remove itself from the processor, as it is waiting on an **I/O** and therefore unable to do any work. Another **thread** will take its place immediately.



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

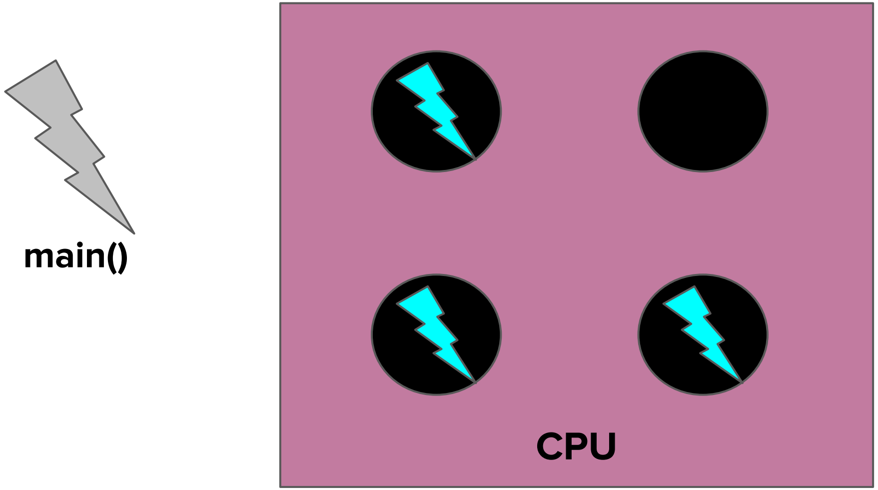
Code example: what happened?

- When the **I/O bound** task completes, the **main 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?

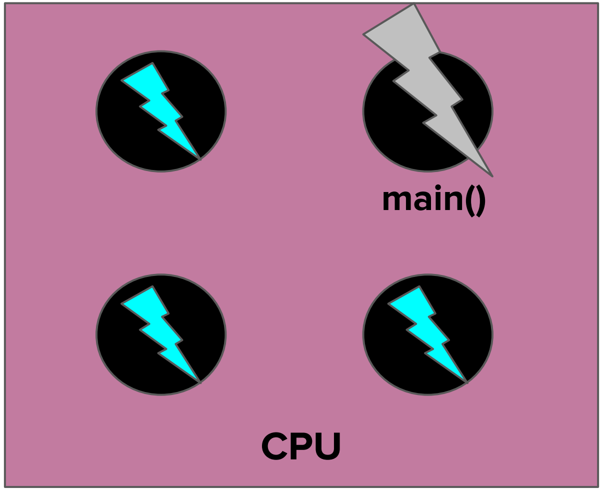
- When the **I/O bound** task completes, the **main thread** will attempt to get back on a core as soon as possible in order to continue (but its order in line is up to your **Operating System**)



A vacancy!

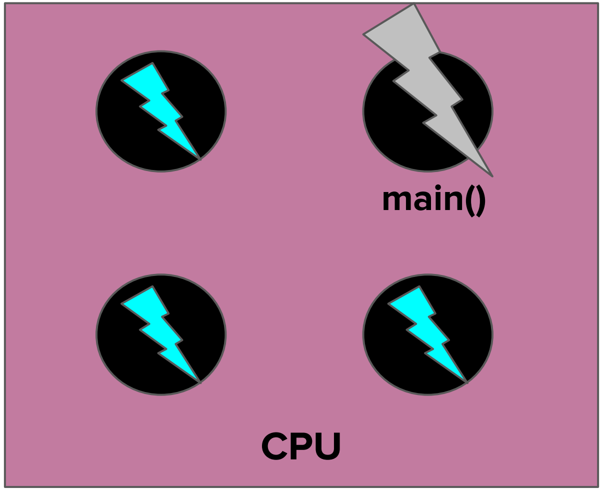
Code example: what happened?

- When the **I/O bound** task completes, the **main 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()**

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()**
- In other words, every time we call **task()** we have to deal with **I/O** wait times that don't depend on how fast our CPU is.

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()**
- In other words, every time we call **task()** we have to deal with **I/O** wait times that don't depend on how fast our CPU is.
  - Can we do better?

Idea: 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!

Idea: 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!

Idea: Multithreading

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

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

- This should look pretty vanilla, except for the parameter!
    - *funcName* is the name of a the function you want to execute!

Idea: Multithreading

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

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

- This should look pretty vanilla, except for the parameter!
  - *funcName* is the name of a the function you want to execute!
  - Let's make new threads that encapsulate **task()**, it's not that hard... right?

Thread joining

- Woah woah woah, hold your horses, eager beaver:

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:
- **As soon as you instantiate a thread, it begins to run**.
    - Be sure you're ready before you dispatch them.

Thread joining

- Woah woah woah, hold your horses, eager beaver:
- **As soon as you instantiate a thread, it begins to run**.
    - Be sure you’re ready before you dispatch them.
    - Threads are somewhat 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.

Thread joining

- Woah woah woah, hold your horses, eager beaver:
- **As soon as you instantiate a thread, it begins to run**.
    - Be sure you're ready before you dispatch them.
    - Threads are somewhat 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.

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 :(

More Threads

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

More Threads

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

Questions so far?

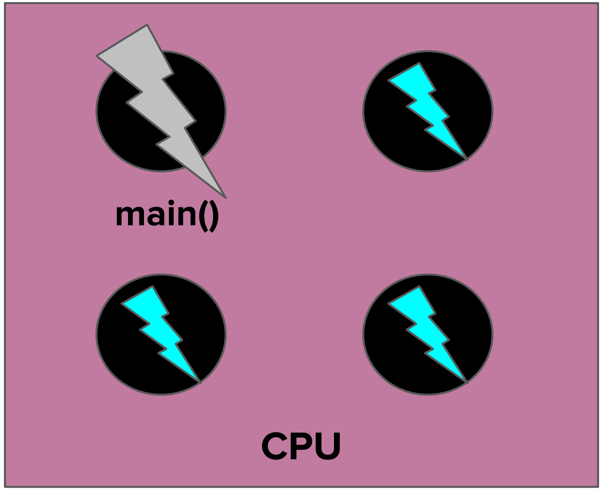
Let’s Parallelize!

What happened?

- Wow, that was super fast!

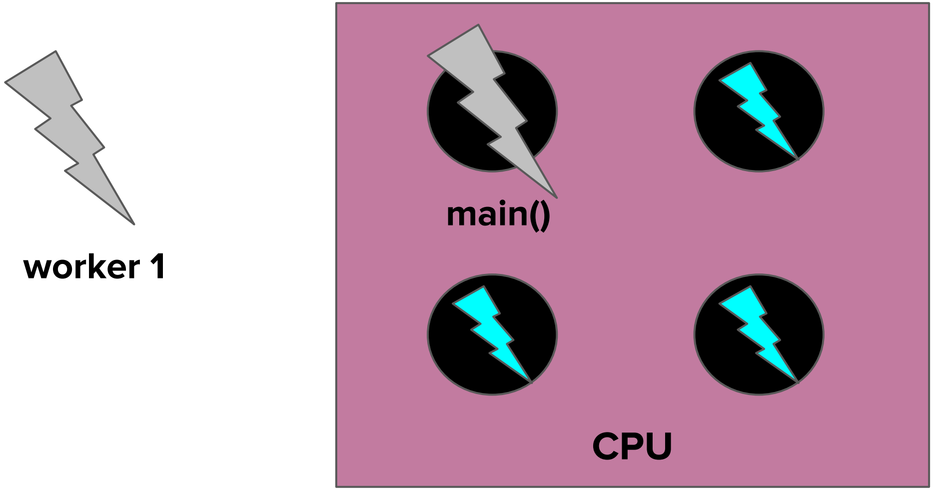
What happened?

- 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!



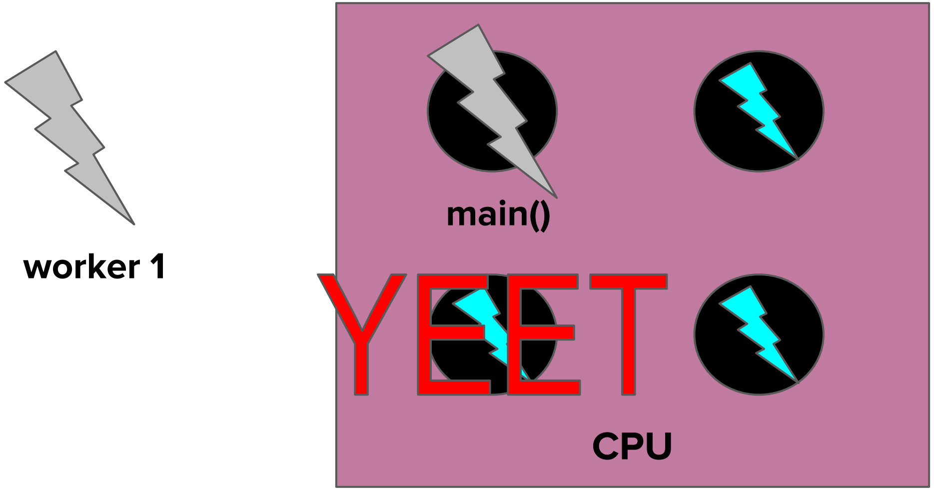
What happened?

- 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!



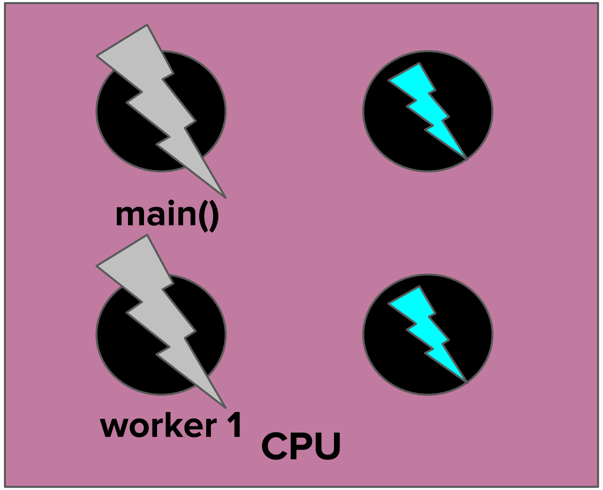
What happened?

- 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!



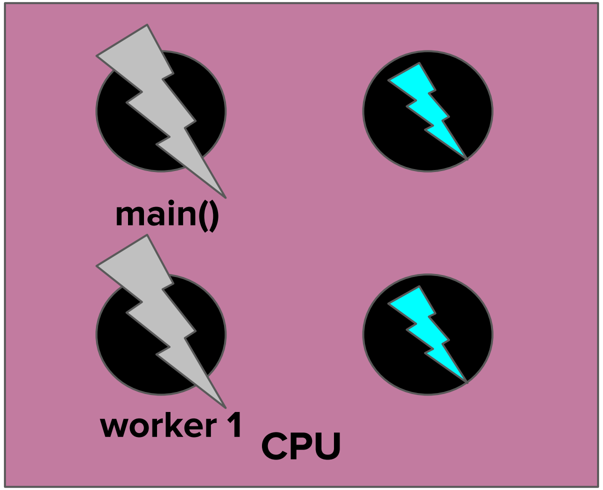
What happened?

- 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!



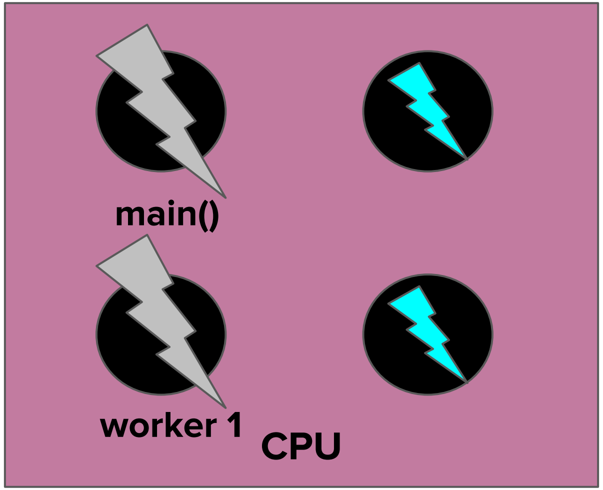
What happened?

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



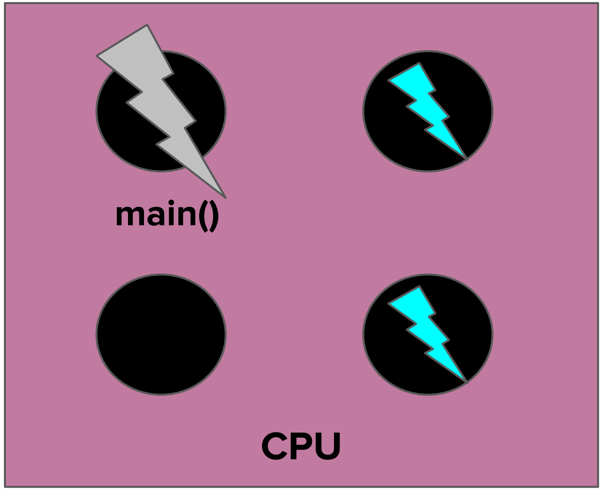
What happened?

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



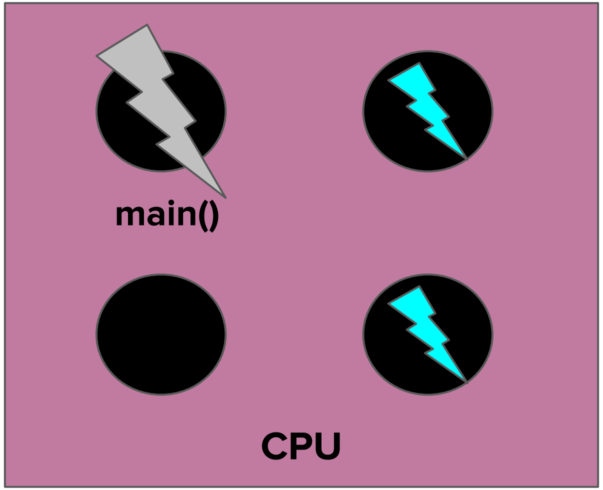
What happened?

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



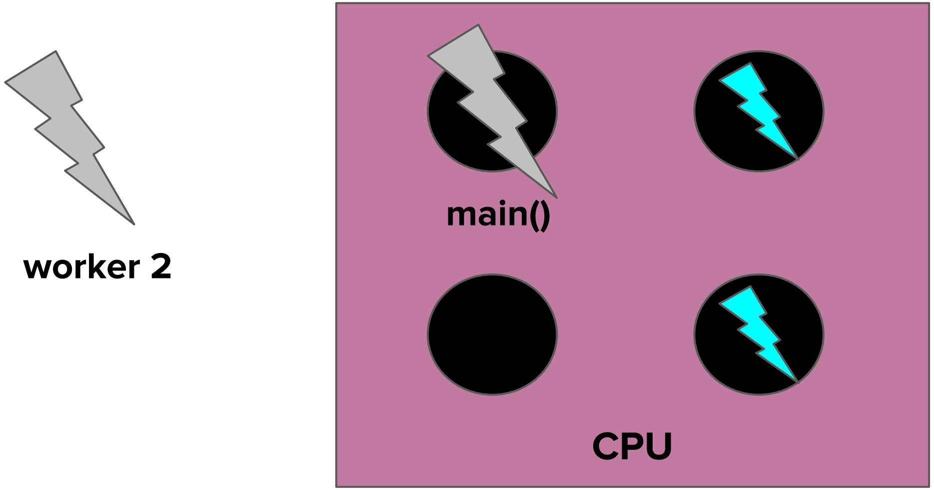
What happened?

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



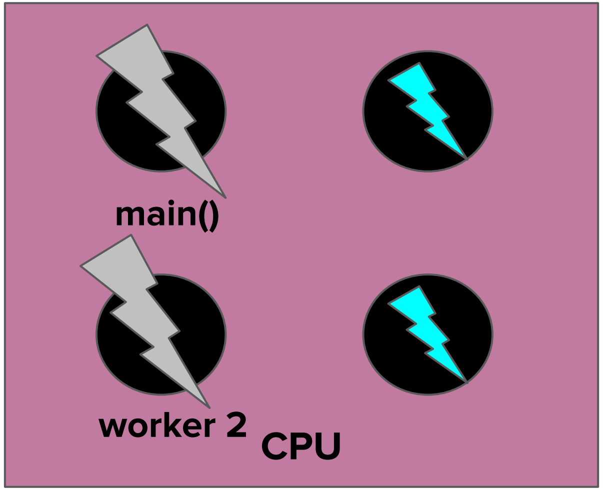
What happened?

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



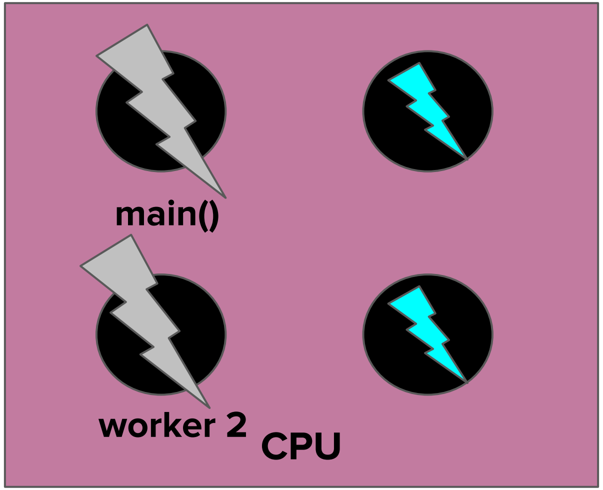
What happened?

- **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!



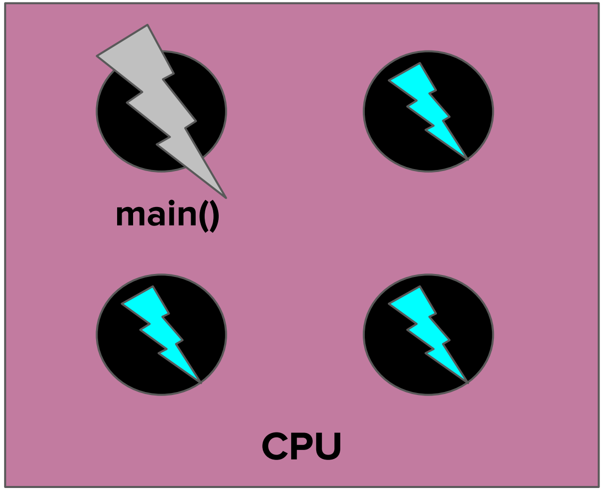
What happened?

- 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?

- At this point, we're **here** in the code:

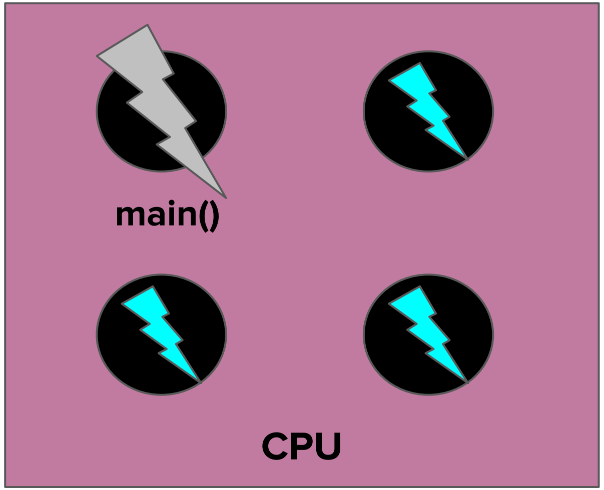



```
thread arr[kArrSize];
for (int i = 0; i < kArrSize; i++) {
    arr[i] = thread(task, i);
}
for (int i = 0; i < kArrSize; i++) {
    arr[i].join();
}
```

What happened?

- At this point, we're **here** in the code:

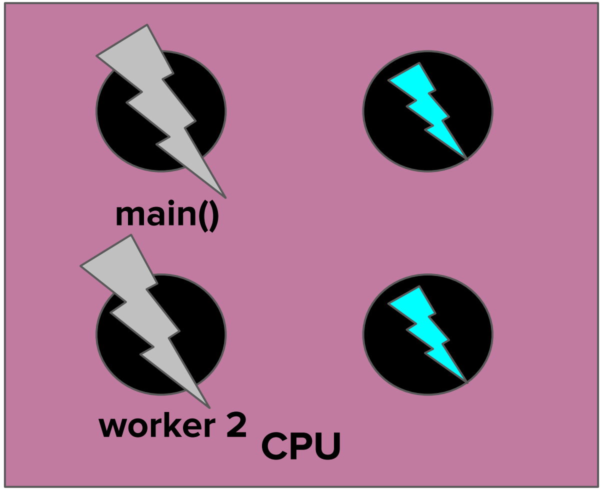
Check your understanding: *What are the worker threads doing right now?*




```
thread arr[kArrSize];
for (int i = 0; i < kArrSize; i++) {
    arr[i] = thread(task, i);
}
for (int i = 0; i < kArrSize; i++) {
    arr[i].join();
}
```

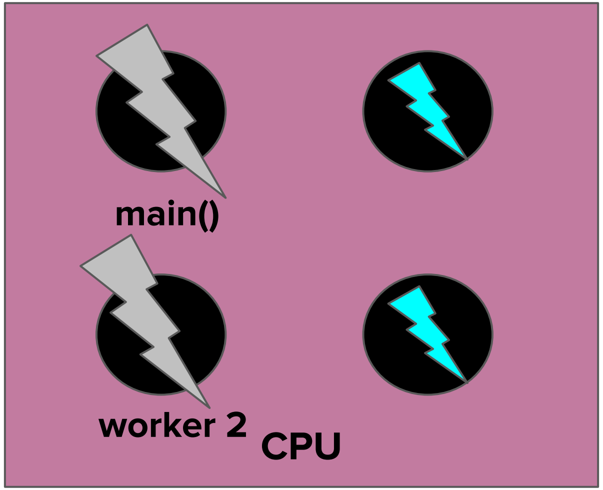
What happened?

- 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**.

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!

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

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, because then task completion depends on chip speed!
    - Take an Operating Systems class to find out more :)

Questions?

Bonus! Race Conditions

- Remember when I said that we can’t really determine the order that threads will run in? Let’s show that!

Bonus! 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!

Bonus! 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 an id variable!

Bonus! 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 an id variable!
  - Recall that we can add parameters to our thread instantiation by simply appending the parameter to our thread instantiation

```
thread newthread = thread(funcName, param1);
```

Bonus! 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 an id variable!
  - Recall that we can add parameters to our thread instantiation by simply appending the parameter to our thread instantiation

```
thread newthread = thread(funcName, param1);
```

- Let's try it!

woah...

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!**

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!

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?

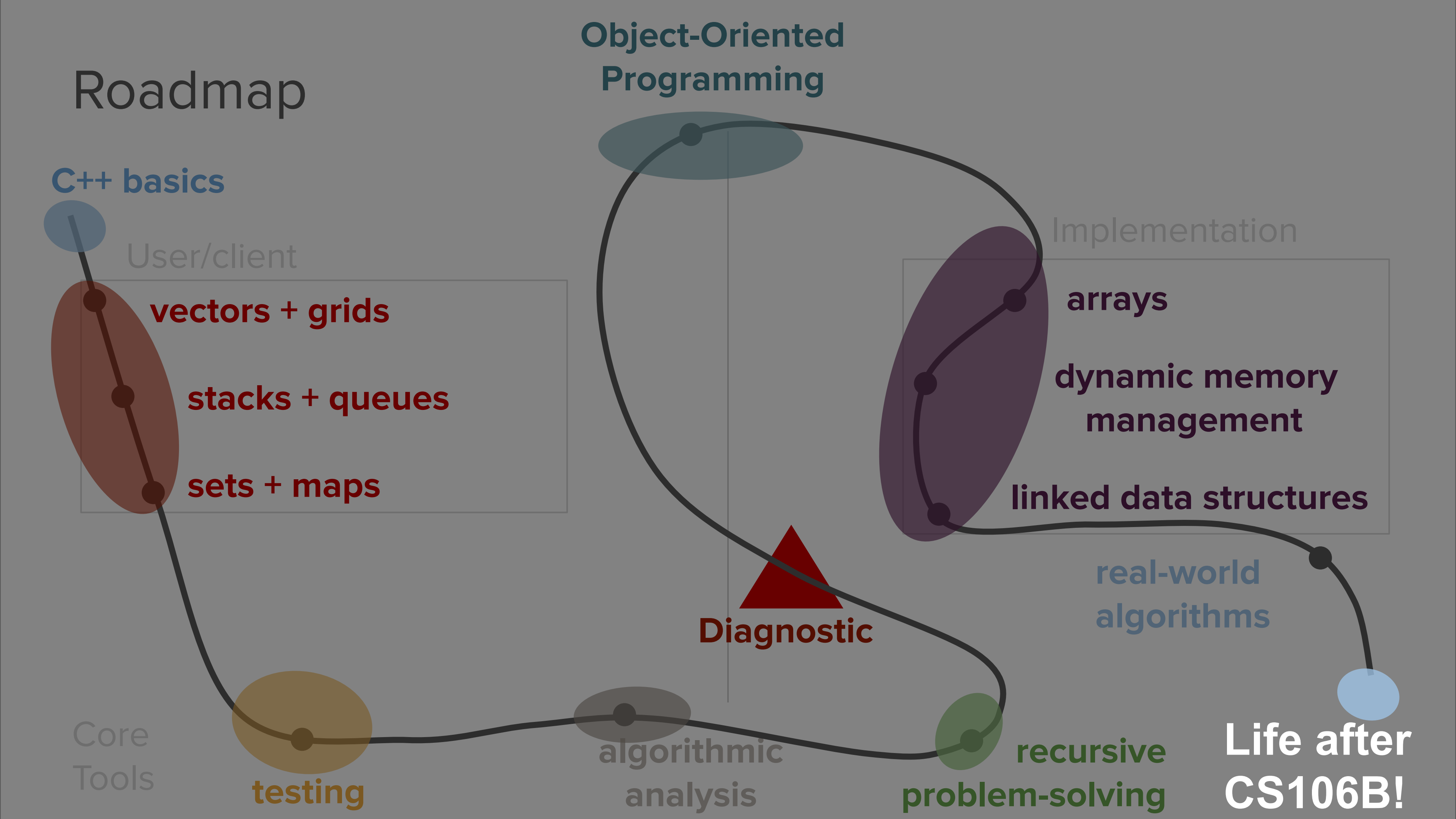
Bonus: Race Conditions

- 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? **Take a systems class :D**

Any Last Questions?

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
Core Tools
testing
algorithmic analysis
recursive problem-solving
Life after CS106B!