Parallel Programming

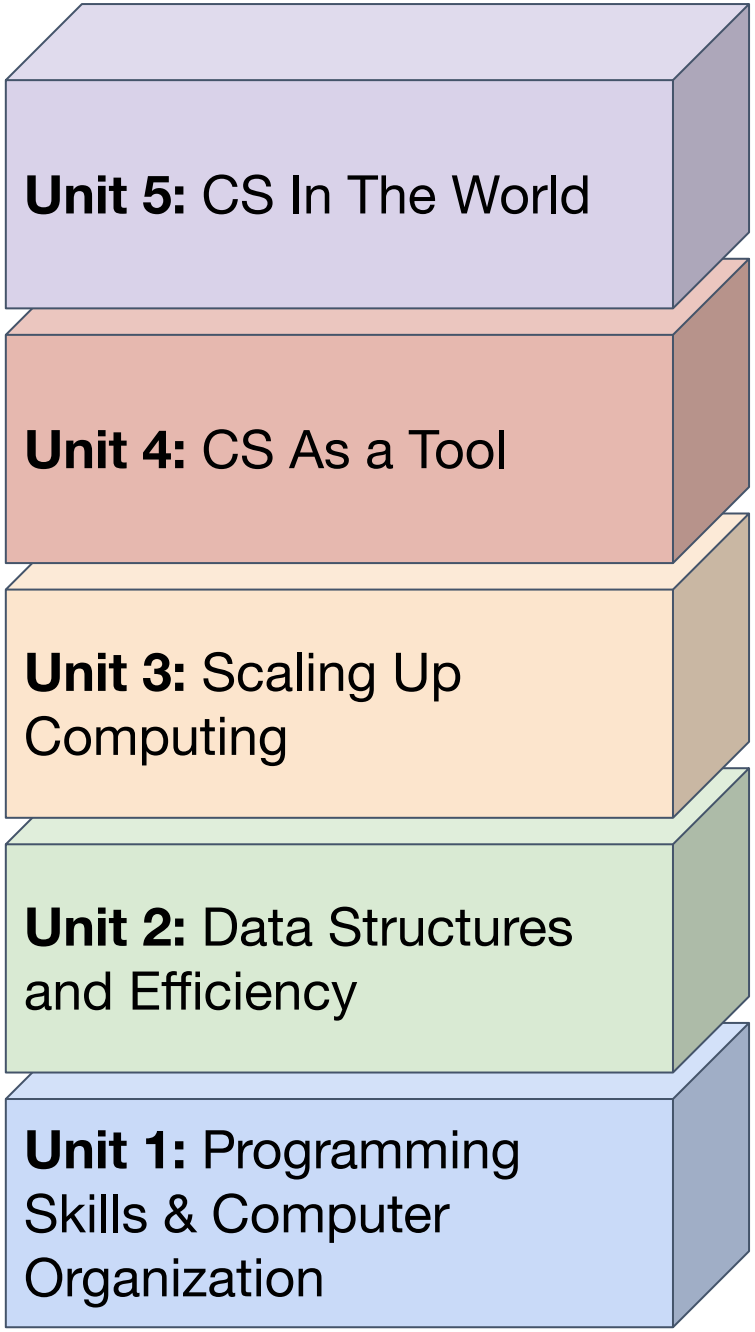
15-110 – Friday 10/25

Announcements

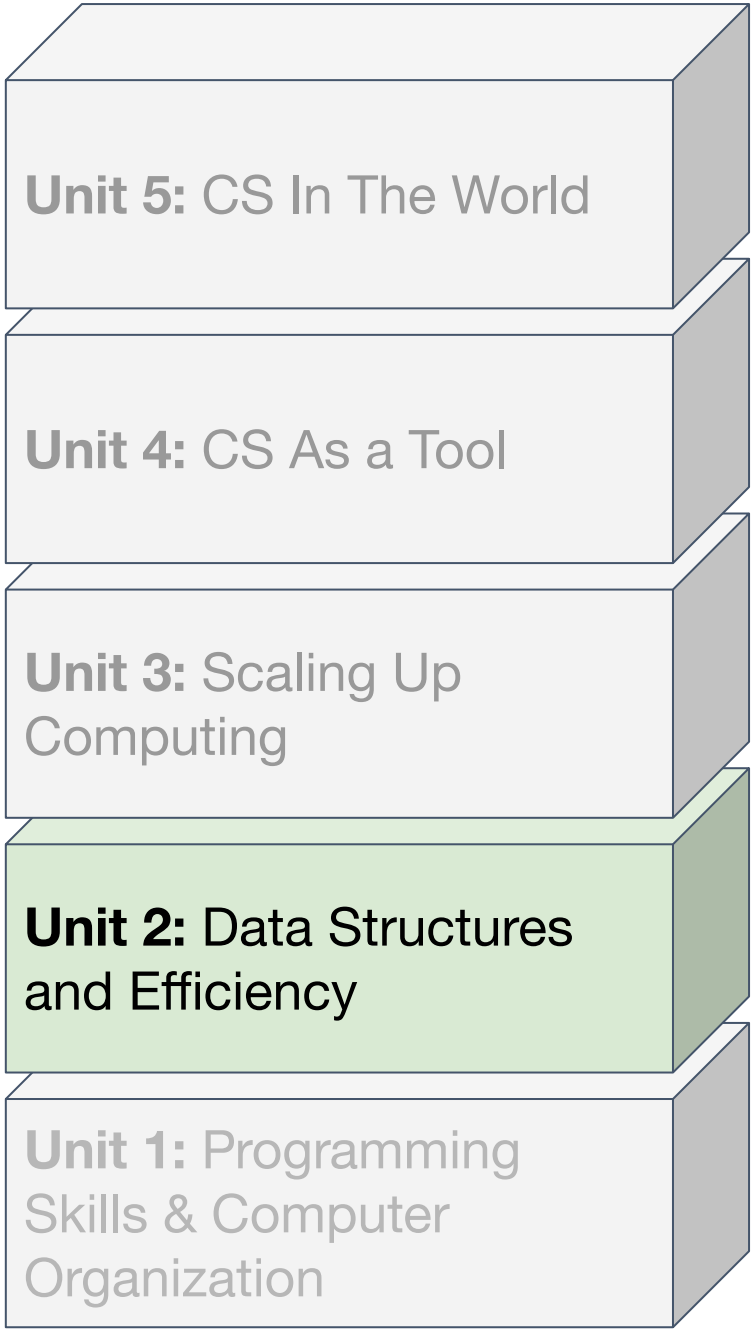
- **Hw4** due on Monday
  - Remember – it's large! Don't leave it until the last minute!

- Will post midsemester surveys tonight: completing them will by the deadline will be 1 bonus point on a quizlet!
  - Look out for a Piazza post

We’ve finished Unit 2!

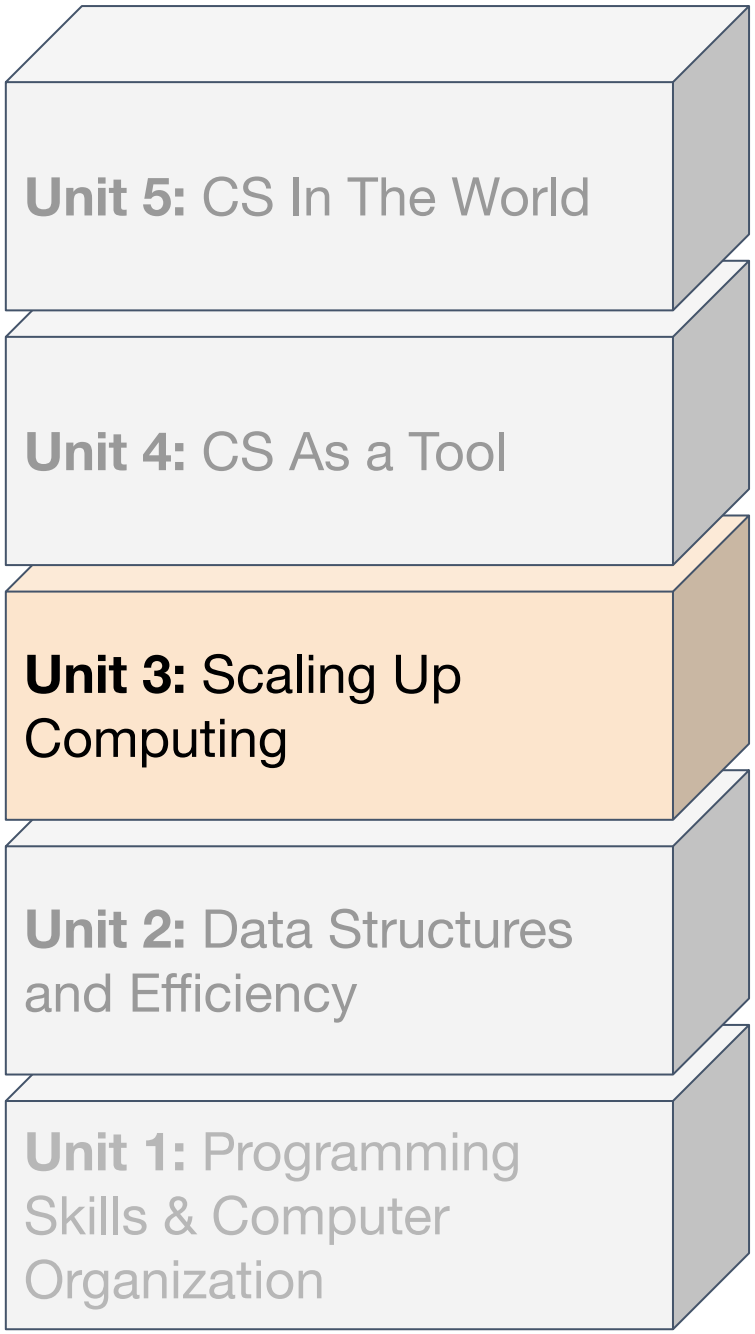


The topics we cover in this course build on each other!



Exam 2 covers Unit 2:

- Lists and Methods
- References and Memory
- Recursion
- Search Algorithms
- Dictionaries
- Runtime and Big-O Notation
- Trees
- Graphs
- Tractability



Unit 3 Topics:

Scaling Up Computing:

- How is it possible for complex algorithms on huge sets of data (like Google search) to run quickly?

- How can we write algorithms that work across multiple computers, instead of running on just one machine?

Learning Goals

- Recognize and define the following keywords: **concurrency**, **parallel programming**, **CPU**, **scheduler**, **throughput**, **multitasking**, **multiprocessing**, and **deadlock**

- Calculate the **total steps** and **time steps** taken by a parallel algorithm

- Create **pipelines** to increase the efficiency of repeated operations by splitting steps across cores

Transistors Provide Electronic Switching

Logical gates are made out of small electronic switches called **transistors**. Over time, we have learned to make transistors smaller and smaller, which allows us to:

- Run computers at faster clock speeds
- Add more complexity into a small device
- Use less power

This means we have much faster and more powerful computers than prior generations.

Switch From Smaller Transistors to Concurrency

For a while, engineers were able to double the speed of computers every two years by increasing the density of transistors in a computer.

However, around 2010 it became physically impossible to keep up this doubling rate because of physical limitations related to power and heat.

Alternate strategy: Create computers that can **run multiple smaller programs at the same time**. This is called **concurrency**.

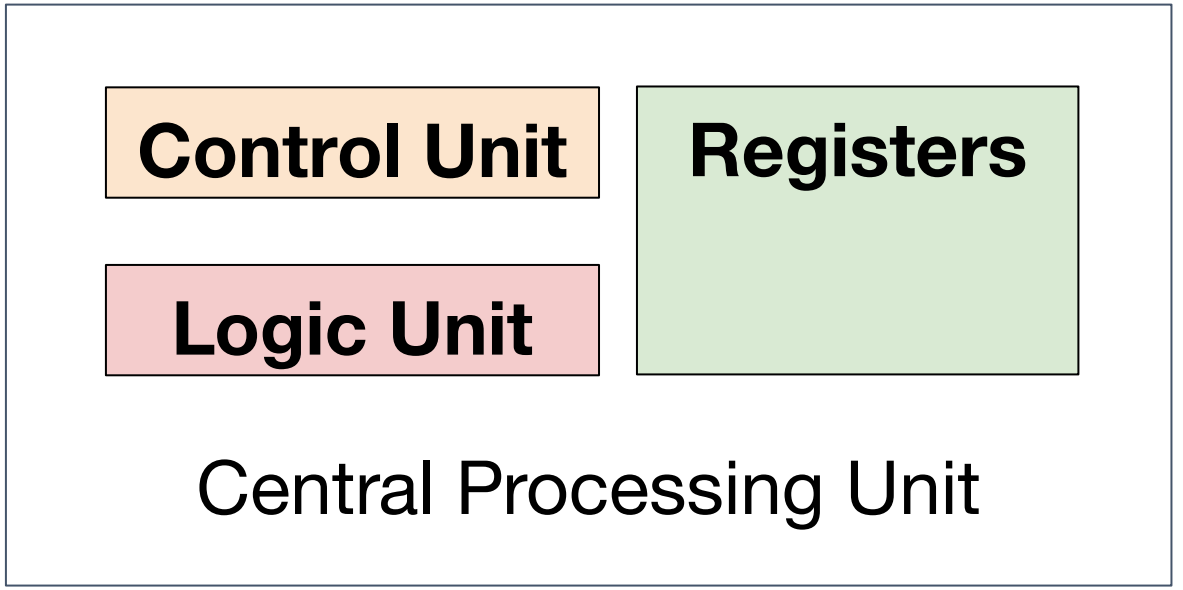
CPUs and Multitasking

A **CPU** (aka **core**), is composed of lots of circuits that actually run a program: **control unit**, **logic units**, and **registers**.

Control unit: maps the individual steps taken by a program to specific circuits.

Logic units: individual circuits that can perform simple operations (like addition and multiplication).

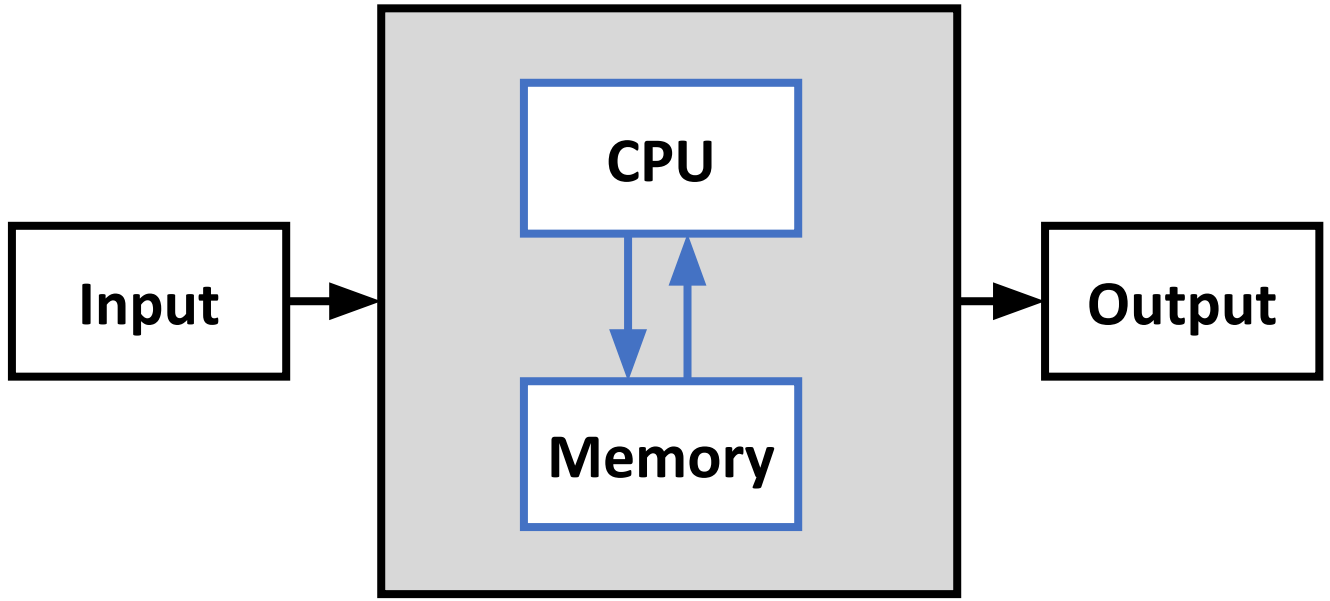
Registers: store information and act as very fast temporary memory.



Computers also have **memory** that the CPU can read from and write to.

Reading/writing to memory is **how the CPU can load instructions and save results**.

Combine a CPU with memory and basic mechanisms for input and output, and you've got a simple abstract computer!

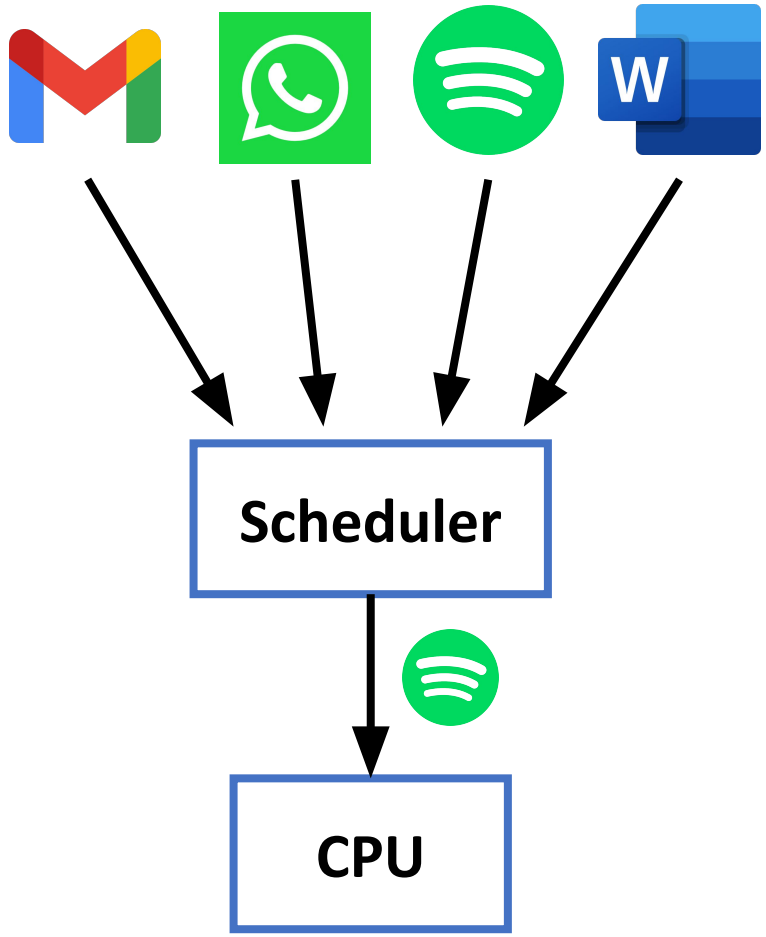


This organization of CPU, memory, input, and output is called a **von Neumann architecture**.

The CPU decided what action to take next using a **scheduler.**

When you use a computer, you likely have multiple applications open and running at any given moment. How does the CPU decide what action to take next?

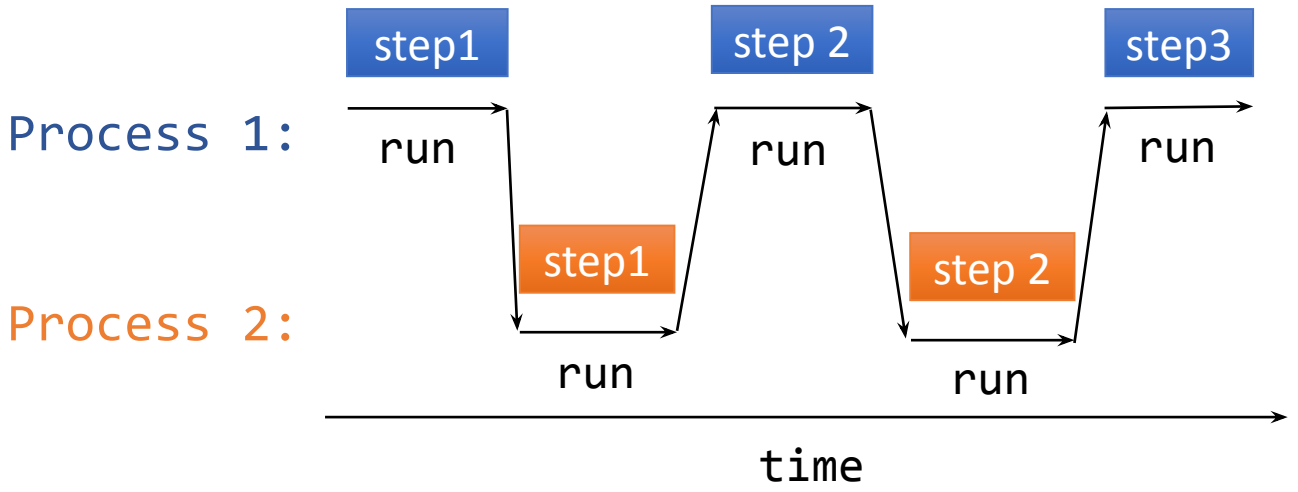
The **scheduler** is a computer component that takes information from the programs that are currently running and input from the user and decides which program gets to use the CPU.



The scheduler uses **multitasking** to make programs appear to run at the same time.

The scheduler can make programs **appear** to run at the same time by breaking each application's process into steps, then alternating between the steps rapidly.

This is called **multitasking**.

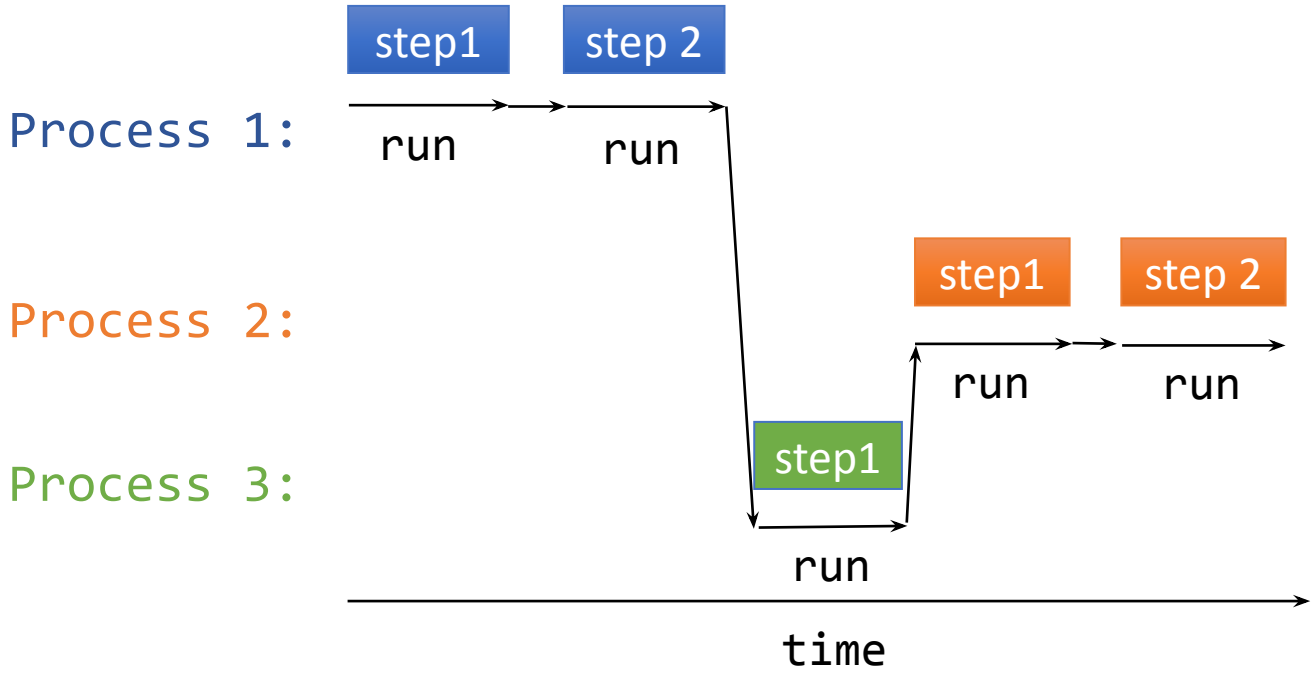


The scheduler chooses which order to run the steps in to **maximize throughput** for the user.

When two (or more) processes are running at the same time, the steps don't need to alternate perfectly.

The scheduler may choose to run several steps of one process before switching to another.

In general, the scheduler chooses which order to run the steps in to maximize throughput for the user. **Throughput** is the amount of work a computer can do during a set length of time.



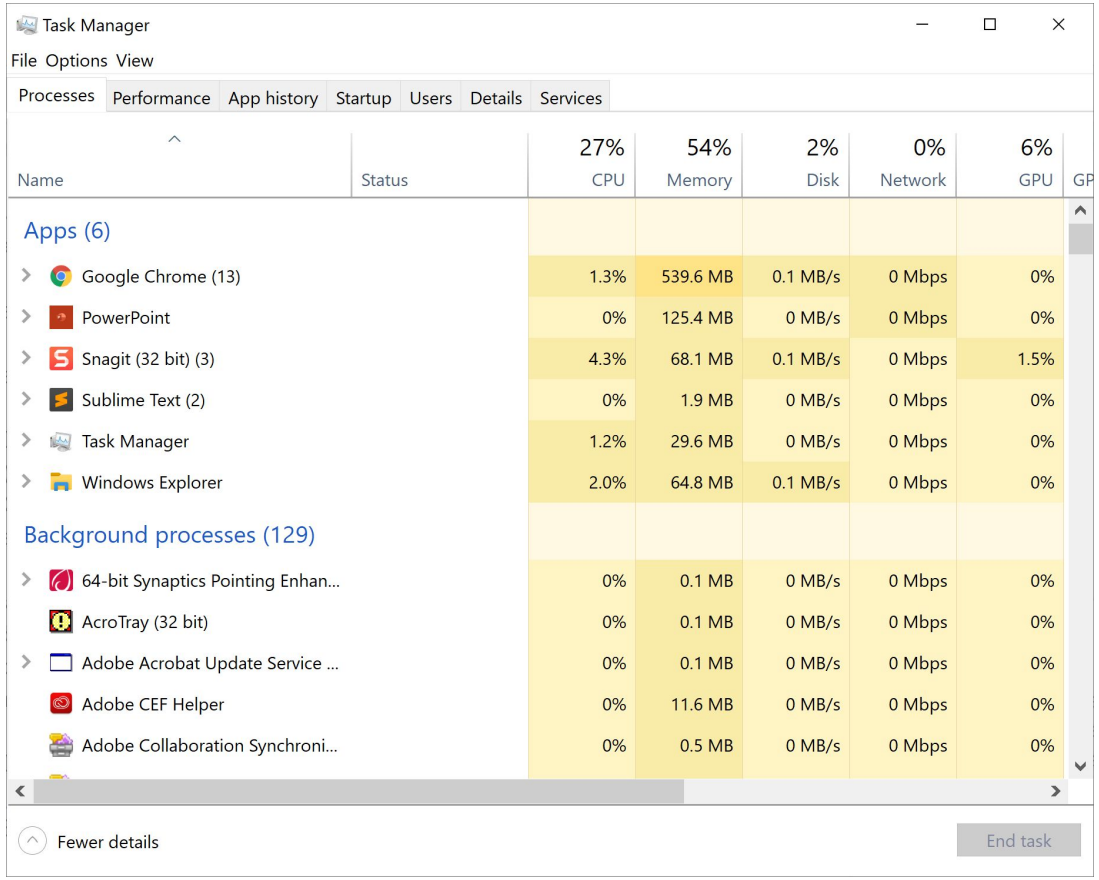
You can see **what processes are running** on your computer using a process monitor.

You can see all the applications and background processes your computer's scheduler is managing by going to your process manager:

- Windows: Task Manager
- Mac: Activity Monitor
- Linux: htop

You can even see how much time each process gets on the CPU!

You do: open your process manager now to see how much CPU time each application takes



Multitasking is very useful, but **it doesn't increase the speed** of a complex algorithm.

A single CPU can still only run one action at a time, so an algorithm is still limited to the CPU's maximum number of instructions per second.

How can we speed up algorithms? We can **use multiple CPUs!**

Multiprocessing and Parallel Programming

Multiprocessing is a method of concurrency where you run multiple actions at the **exact same time on a single computer**.

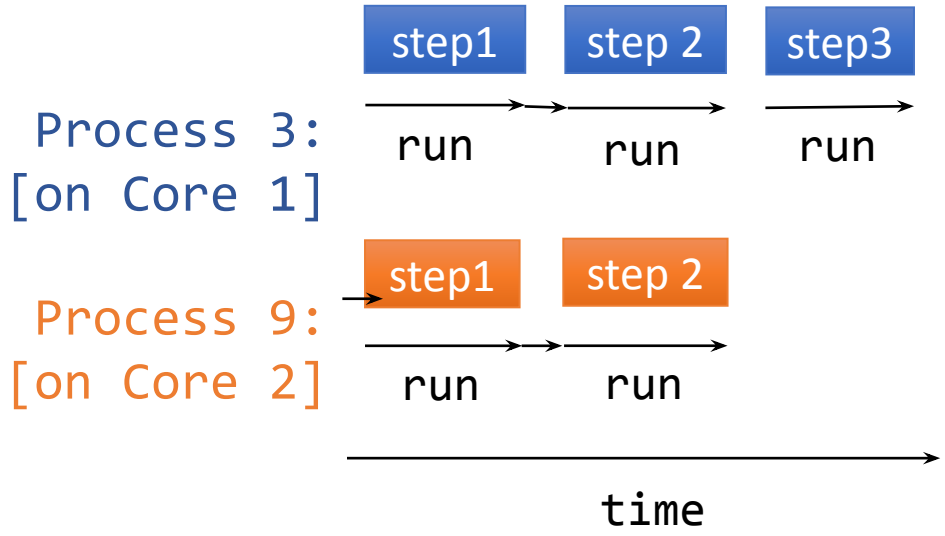
To make this possible, you put **multiple CPUs inside a single computer**. Then the computer can send different actions to different CPUs at the same time.

If you have two CPUs instead of one, you can theoretically double the speed of your computer. With four CPUs, you could quadruple it!

With multiprocessing we can run our applications simultaneously by **assigning them to different cores**.

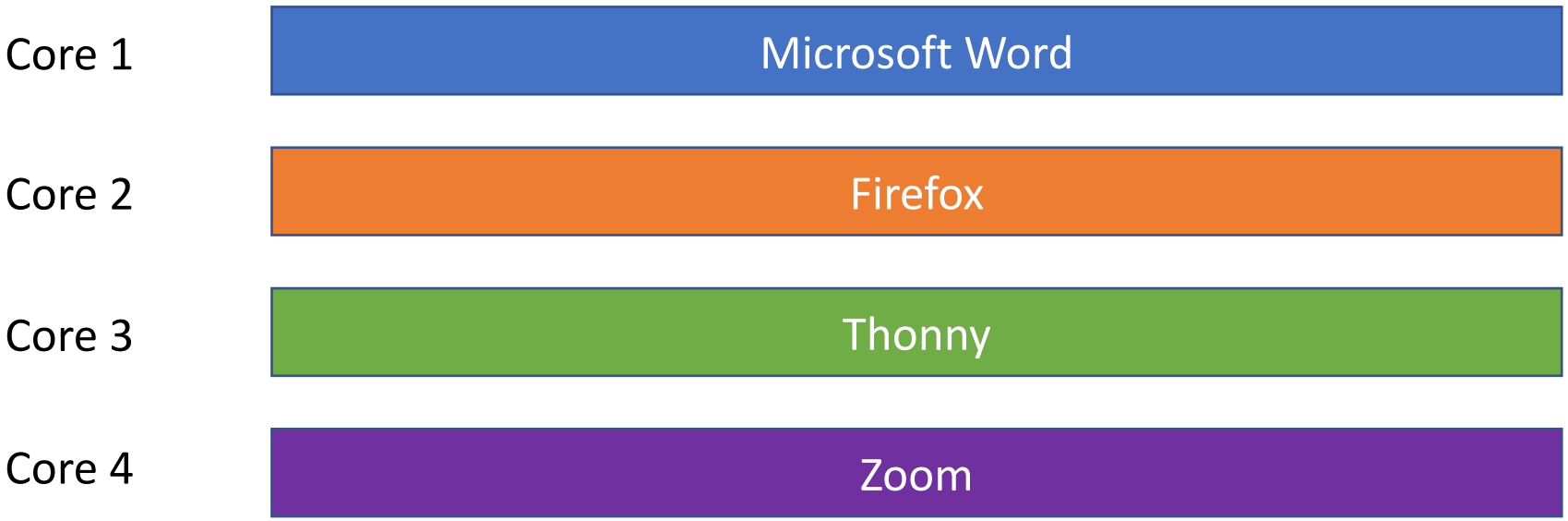
Each core has its own scheduler, so they can work independently.

Multiple cores and multiple processors are slightly different approaches in practice, but we'll treat them as the same in this class.



Simplified Scheduling

Here's a simplified visualization of scheduling with multiprocessing, where we condense all of the steps of an application into one block.



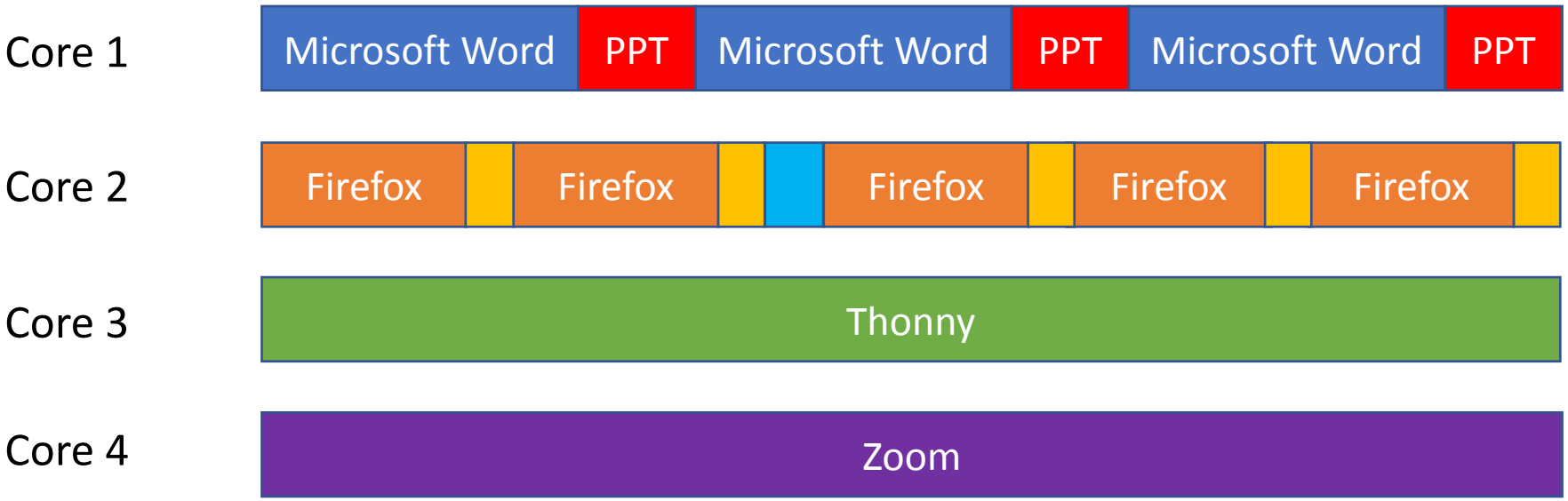
The number of cores we have on a single computer is **usually still limited**.

Most modern computers use somewhere between 2-16 cores. If you run more applications than you have cores, the cores use **multitasking** to make them appear to run concurrently.

You can check how many cores your own computer has! If you're on Windows, go back to the process manager and switch to the tab 'Performance'. If you're on a Mac, go to About This Mac > System Report > Hardware. On Linux, run `lscpu`.

Computers **combine schedulings with multiprocessing and multitasking** to try and achieve as much concurrency as possible.

Here's a simplified view of what scheduling might look like when we combine multiprocessing with multitasking.



Parallel programming divides a single program amongst multiple CPUs.

Parallel programming is a type of multiprocessing where we run a single process or program on multiple CPUs at the same time, so that the work can get done a lot faster!

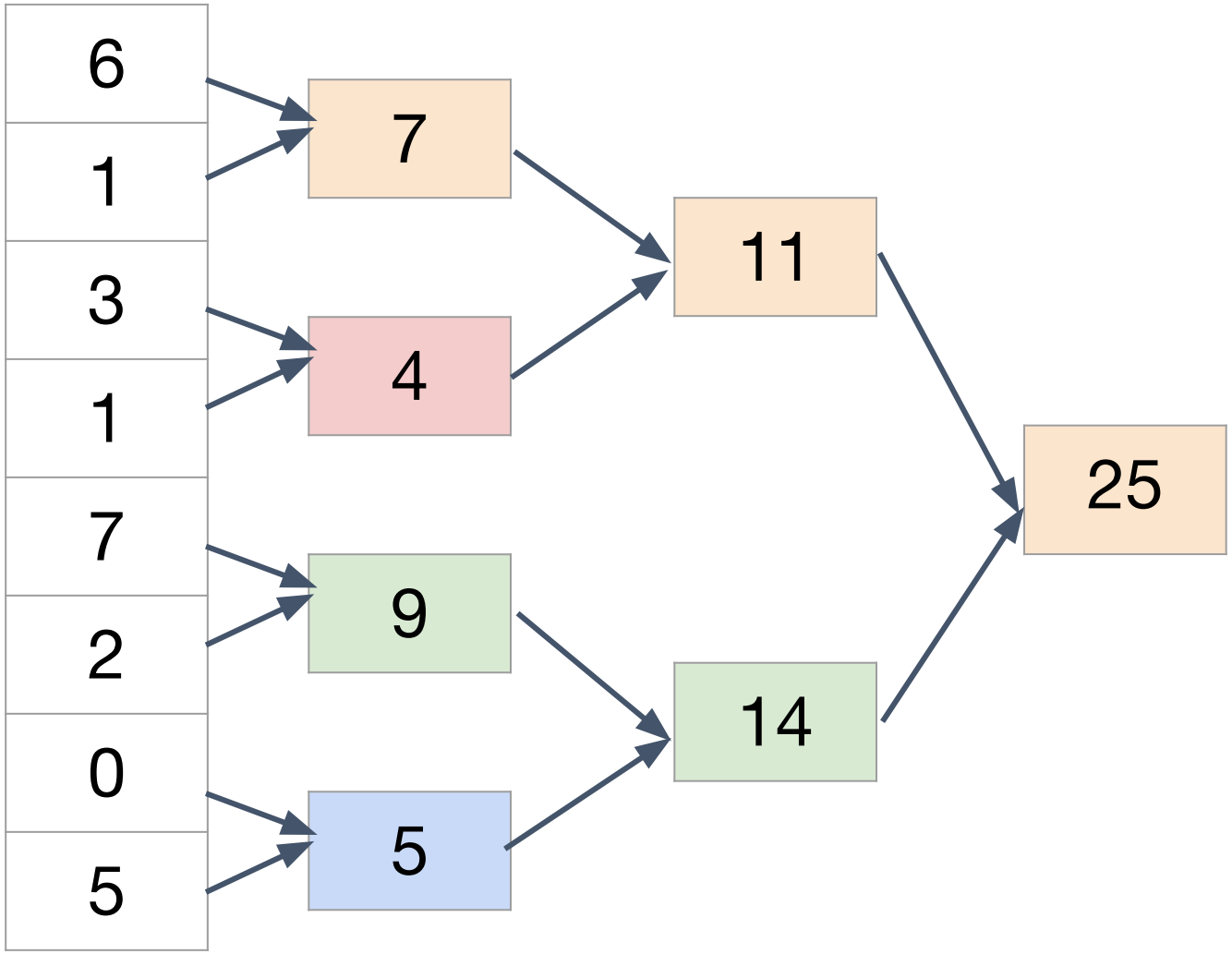
Unfortunately, **not all algorithms can be easily split** between multiple CPUs…

Parallel programming tends to be **more difficult than regular programming**.

To solve a problem using parallel programming, we must design algorithms that can be split across multiple processes. This varies greatly in difficulty based on the problem we're solving!

We won't write actual parallel programs in this class. But we will talk about common algorithmic approaches for writing parallel code.

We can sum a list **concurrently**.



To determine the **efficiency of a parallel algorithm**, it helps to compare the total number of steps to the number of time steps.

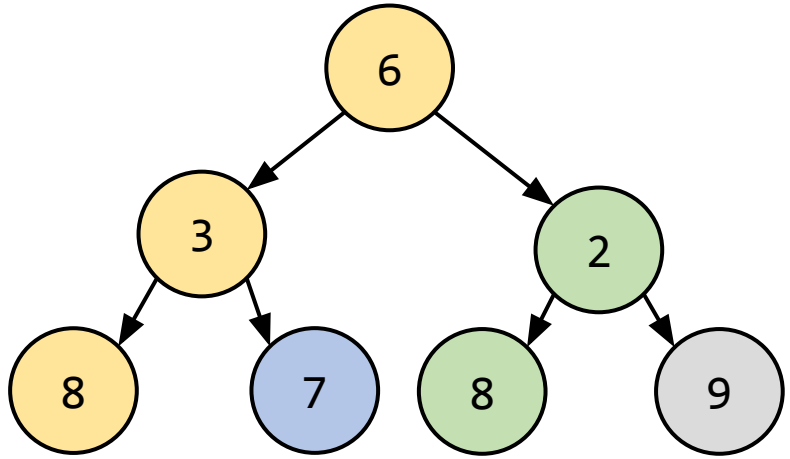
The **total number of steps** is just the number of actions taking place across all CPUs in the whole process. For summing the previous tree that is always 7 steps, whether or not we use parallelization.

The **number of time steps is the number of steps taken over time**. Multiple actions can be merged into a single time step when they happen at the same time. Summing the list sequentially takes seven time steps, but summing the tree concurrently only takes **three time steps**.

We can sum a tree **concurrently**.

We showed in class how to write a function that can sum all the nodes in a tree that ran in O(n) time sequentially, since **each node needs to be visited**. What if we do it concurrently?

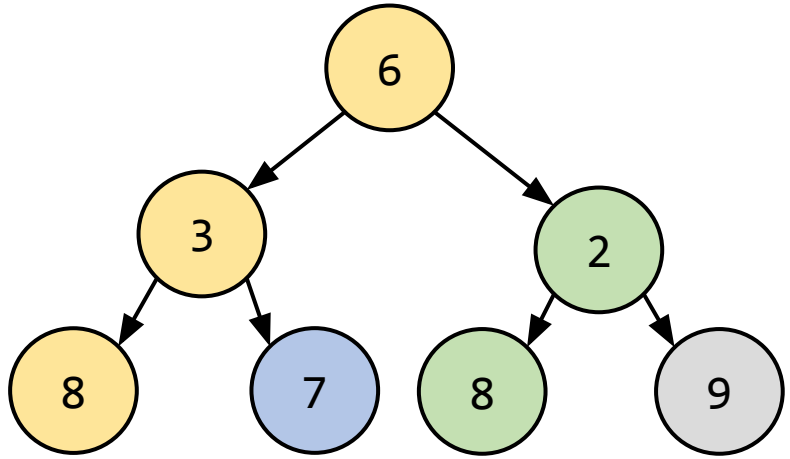
We do up to two recursive calls in each recursive case (one on the left child, one on the right). Call the left child recursively on the current core and **send the right child's call to a new core**. This lets us **do the two recursive calls concurrently**. In our example to the right, this is shown using different colors for each core.



Summing a tree concurrently is **O(log n)**.

How can we calculate the efficiency of concurrent tree summing? Consider the original core, which does the most steps. This will only do one call per level of the tree.

If the tree is balanced (and if we have enough cores), it will have log n levels. Concurrent tree-summing is O(log n).



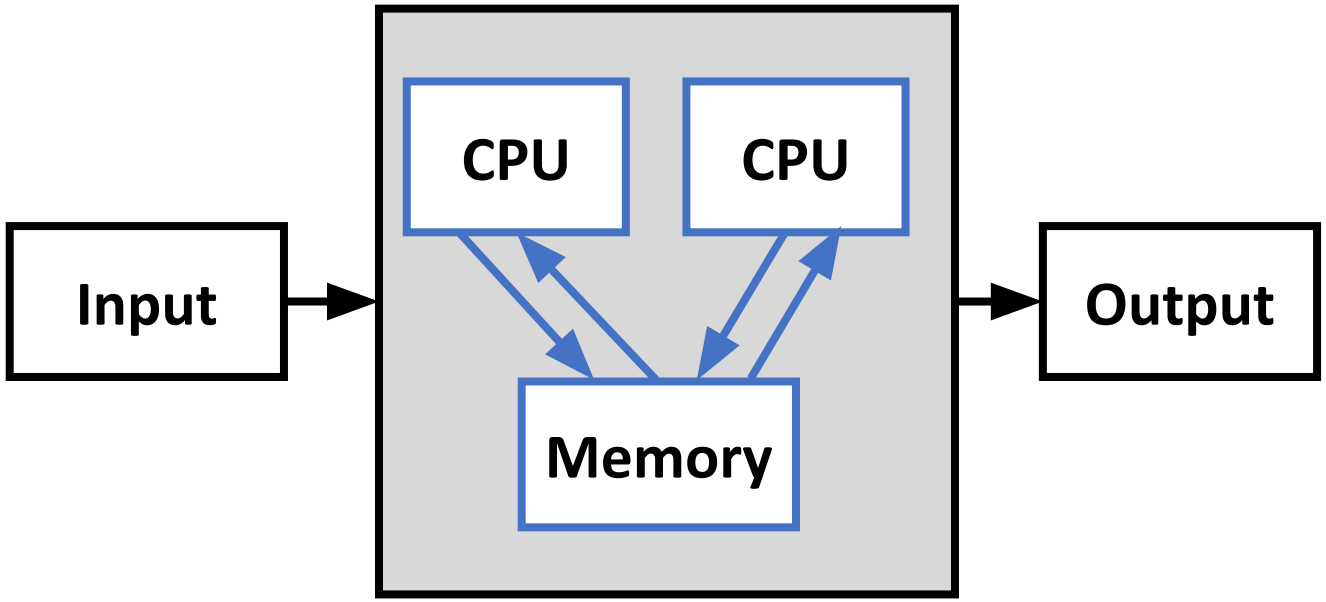
It is difficult to write parallel programs with **concurrent loops**.

We could plan to identify all the iterations of the loop and run each iteration on a separate core. But what if the results of all the iterations need to be combined? And what if each iteration depends on the result of the previous one? This gets even harder if we don't know how many iterations there will be overall, like when we use a while loop.

A bit later, we'll talk about how to use algorithmic plans to address these difficulties.

It is difficult to write parallel programs when **multiple cores need to share individual resources** on a single machine.

For example, two different programs might want to access the same part of the computer's memory at the same time. They might both want to update the computer's screen or play audio over the computer's speaker.



To avoid this making bad concurrent updates, programs put a lock on a shared resource when they access it.

We can't just let two programs update a resource simultaneously- this will result in garbled results that the user can't understand. For example, if one program wants to print "Hello World" to the console and the other wants to print "Good Morning", the user might end up seeing "Hello Good World Morning".

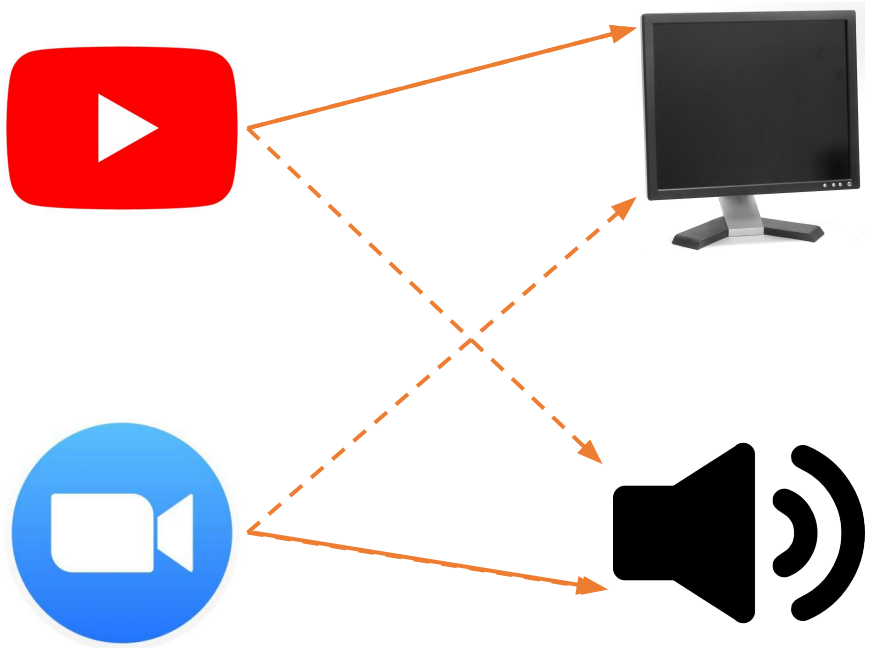
To avoid this situation, programs put a **lock** on a shared resource when they access it. While a resource is locked, no other program can access it.

Then, when a program is done with a resource, it **yields** that resource back to the computer system, where it can be sent to the next program that wants it.

Sidebar: if we want two programs to use a resource simultaneously, we usually use a third program to combine the actions together, and that third program is the one that accesses the resource. For example, if you listen to music while watching a lecture recording, your computer **mixes** the two audio tracks together and plays the combined result.

Deadlock occurs when the **system stalls** because multiple programs want to use the same resource at the same time.

For example: Two programs, Youtube and Zoom, both want to access the screen and audio. They put their requests in at the same time, and the computer gives the screen to Youtube and the audio to Zoom.

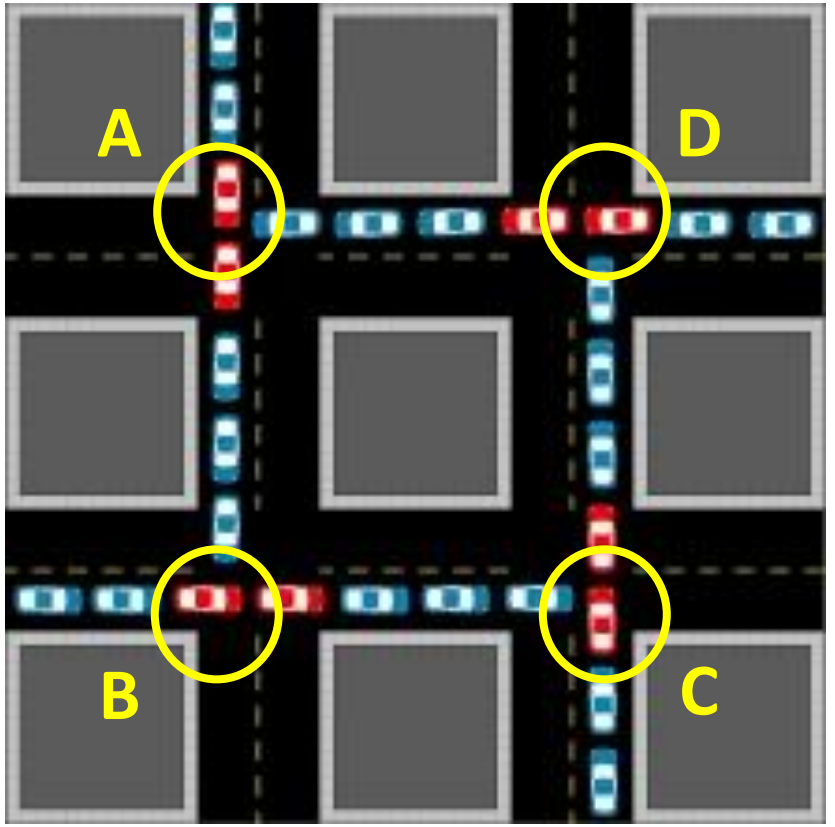


Both programs will lock the resource they have, then wait for the next resource to become available. **Since they're waiting on each other, they'll wait forever!** This is known as deadlock.

In general, deadlock occurs when two or more processes are all **waiting for some resource** that other processes in the group already hold.

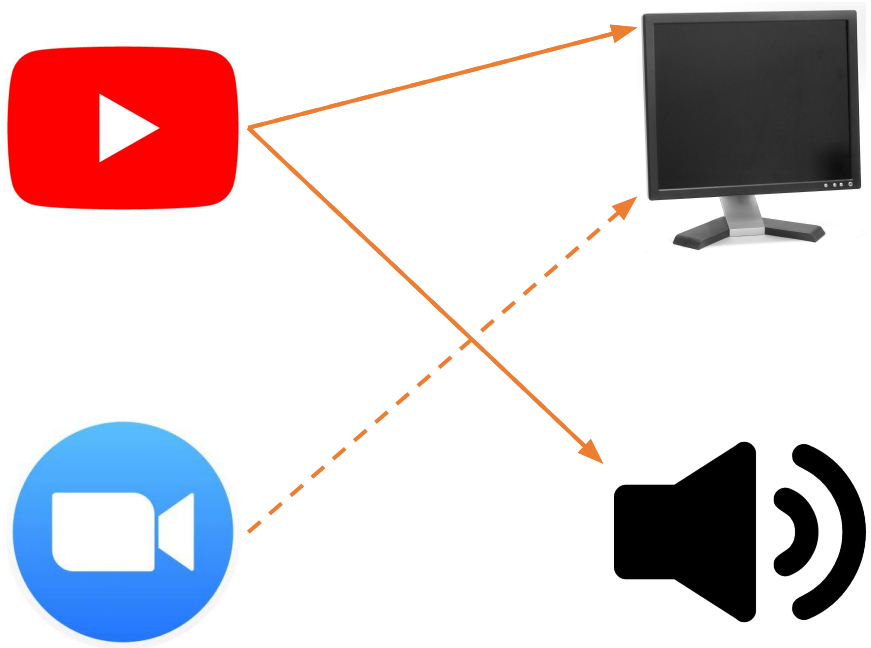
Deadlock can happen in real life! For example, if enough cars edge into traffic at four-way intersections, the intersections can get locked such that no one can move forward.

In the example to the right, each direction of traffic needs two of the intersection spots, but only has one. All four directions are blocked as a result.



To **fix deadlock** we can impose an order that programs always follow when requesting resources.

For example, maybe Youtube and Zoom must receive the screen lock before they can request the audio. When Youtube gets the screen, it can make a request for the audio while Zoom waits for its turn.



When Youtube is done, it will yield its resources and Zoom will be able to access them.

Sometimes processes **need to communicate**.

If a single program is split into multiple tasks that run concurrently instead, those tasks might need to share partial results as they run.

Data is shared between processes by **passing messages**. When one task has found a result, it may send it to the other process before continuing its own work.

If one process depends on the result of another, it may need to halt its work while it waits on the message to be delivered. This can slow down the concurrency, as it takes time for data to be sent between cores or computers. **Example:** in tree-summing, a core will need to wait for both calls to finish before it can sum the results.

There are **general algorithmic approaches** for message passing.

Writing algorithms that can pass messages is tricky. To make it easier, we use general algorithmic approaches that can be adapted for specific tasks.

We'll discuss one common approach today (**pipelining**) and another in the next lecture (**MapReduce**).

Pipelining

In **pipelining**, you start with a task that repeats the same procedure over many different pieces of data by **creating an assembly line**.

The steps of the procedure are split across different cores. Each core is like a single worker on an **assembly line**; when it is given a piece of data it executes the step, then passes the result to the next core.

Just like in an assembly line, the cores can run **multiple pieces of data simultaneously** by starting new computations while the others are still in progress.

Demo: Real-Life Pipelining

Let's compare pipelining to sequential work with a real-life race!

We need to generate ten greeting cards. We can divide the process of writing a greeting card into three steps:

1. Write 'Wish you were here!' on the paper
2. Fold the paper and put it inside an envelope
3. Seal the envelope

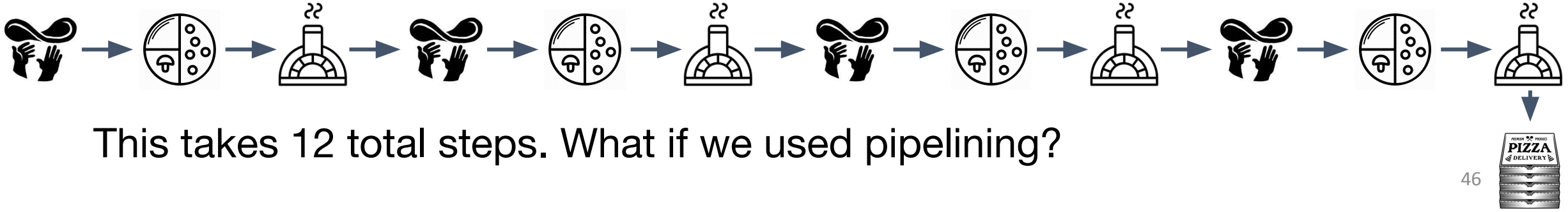
What happens if we have one process (person) complete all three tasks vs. having three processes (people) complete the tasks using a pipeline?

Sequential Pizza – 1 worker, 1 oven, 12 steps

Here's an example of pipelining through the lens of line cooking. To make a pizza, we must:

1. Flatten the dough
2. Apply the toppings
3. Bake in the oven

If we need to make four pizzas without parallelization, it will look like this:



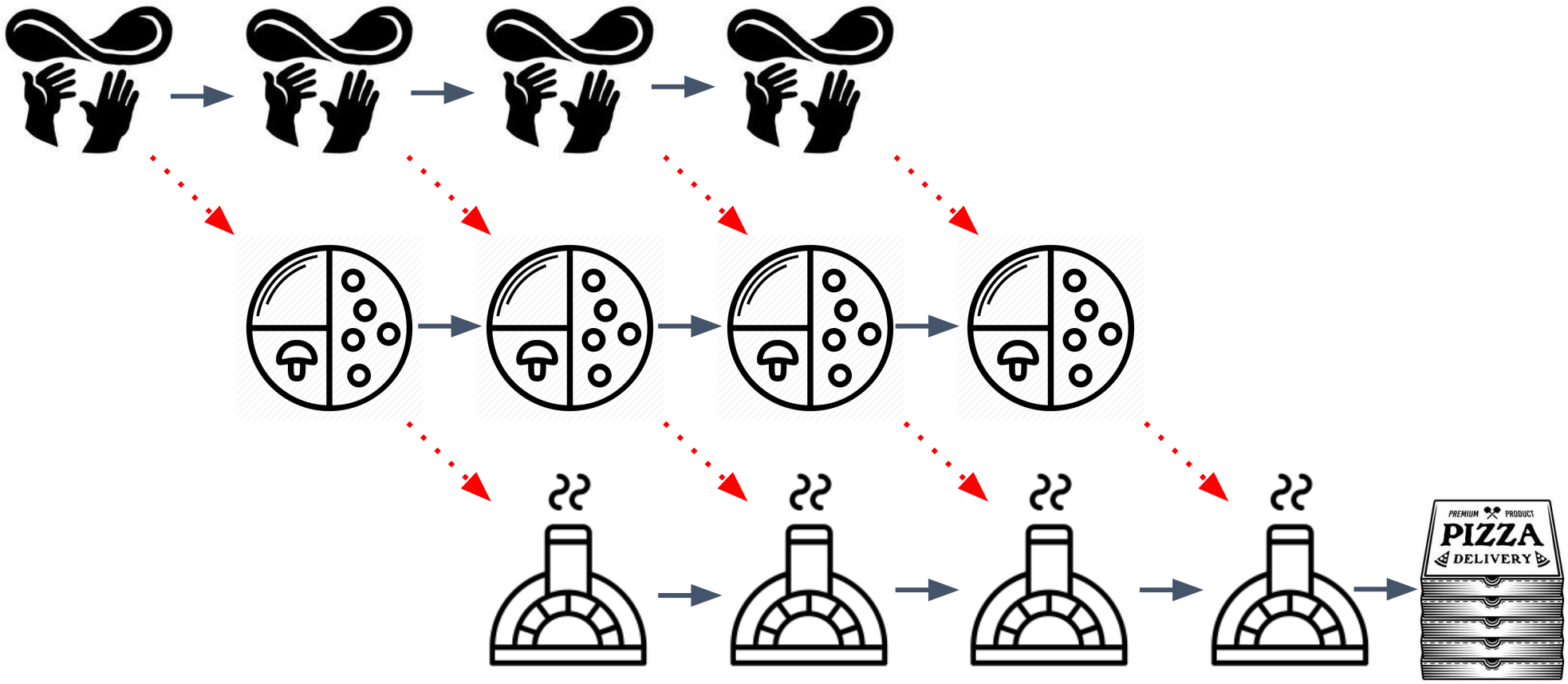
This takes 12 total steps. What if we used pipelining?

Pipelining Pizza - 3 workers, 1 oven, 6 steps

Worker 1:

Worker 2:

Worker 3:



Each worker has **one task**. #1 flattens dough, #2 arranges toppings, #3 bakes in the oven.
There are still 12 total steps, but only **6 time steps** occur.

There are **several rules** to remember when creating pipelines.

When designing a pipeline, it's important to remember that **each step relies on the step that came before it**. You cannot start applying toppings until the dough has been flattened.

Additionally, the length of time that the pipelining process takes **depends on the longest step**. If flattening dough and applying toppings are fast (maybe 5 minutes each) but cooking in the oven is slow (maybe 20 minutes), the whole process will have to wait on the slowest step to conclude.

Pipelining is most useful when the **number of shared resources is limited**.

For example, you probably use pipelining when doing laundry at home, because you have a limited number of washers and driers to work with!

In computer science, pipelining is used to increase the efficiency of certain operations, like matrix multiplication. It is also used in the Fetch-Decode-Execute cycle, which is how the CPU processes instructions.

Learning Goals

- Recognize and define the following keywords: **concurrency**, **parallel programming**, **CPU**, **scheduler**, **throughput**, **multitasking**, **multiprocessing**, and **deadlock**

- Calculate the **total steps** and **time steps** taken by a parallel algorithm

- Create **pipelines** to increase the efficiency of repeated operations by splitting steps across cores