Advanced Computer Science #2: Concurrency

CS SCHOLARS – PROGRAMMING

Learning Goals

Define and understand the differences between the following types of concurrency: **circuit-level concurrency, multitasking, multiprocessing,** and **distributed computing**

Create **concurrency trees** to increase the efficiency of complex operations by executing sub-operations at the same time

Recognize certain problems that arise while multiprocessing, such as **difficulty of design** and **deadlock**

Create **pipelines** to increase the efficiency of repeated operations by executing sub-steps at the same time

Use the **MapReduce pattern** to design and code **parallelized algorithms** for distributed computing

Moore's Law: Computers Keep Getting Faster

You've probably noticed that the computer you use now is much faster than the computer you used ten years ago. That's because of a technology principle known as **Moore's Law**.

Moore's Law basically states that the power of a computer doubles **every two years**. If you buy a computer designed in 2020, it should be **twice as powerful** as a computer made in 2018.

Note: Moore's Law is an observation, not an actual law of nature. But how does it work?

Moore's Law – The number of transistors on integrated circuit chips (1971-2018)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are linked to Moore's law.

Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldInData.org. There you find more visualizations and research on this topic.

Licensed under CC-BY-SA by the author Max Roser.

Transistors Provide Electronic Switching

Recall the lecture on gates and circuits. How does the computer send data to different circuits for different tasks?

This is accomplished using a **transistor,** a small device that makes it possible to switch electric signals. In other words, adding a transistor to a circuit gives the computer a **choice** between two different actions. Gates are partially made out of transistors.

When we make transistors smaller, we can decrease the distance between them (reducing communication time) and increase the number that fit on a chip. Smaller transistors also use less current. This makes the computer **faster**.

Moore's Law: Double the Transistors

A more precise statement of Moore's Law is that the number of transistors on a computer chip will double every two years. This provides the increase in computing power, and the speed-up.

Originally, engineers were able to double the number of transistors by making them smaller every year, to fit twice as many transistors on a single computer chip. But around 2010 it became physically impossible to make the transistors smaller at such a rapid rate (due to electronic leakage).

Now engineers attempt to follow Moore's Law by using **parallelization** instead. In other words, your computer may contain multiple processing units, and may run more than one block of instructions **at the same time**.

Levels of Concurrency

Concurrency and Parallelization

In general, when we refer to the term **concurrency**, we mean that multiple programs are running at exactly the same time.

We will also refer to **parallelization** as the process of taking an algorithm and breaking it up so that it can run across multiple concurrent processes at the same time.

We'll start by discussing four different levels at which concurrency occurs. Then we'll discuss broad approaches for implementing parallel algorithms.

Four Levels of Concurrency

The four levels of concurrency are:

Circuit-Level Concurrency: concurrent actions on a single CPU

Multitasking: seemingly-concurrent programs on a single CPU

Multiprocessing: concurrent programs across multiple CPUs

Distributed Computing: concurrent programs across multiple computers

A CPU Manages Computation

A **CPU** (or Central Processing Unit) is the part of a computer's hardware that actually runs the actions taken by a program. It's composed of a large number of circuits.

The CPU is made up of several parts. It has a **control unit**, which maps the individual steps taken by a program to specific circuits. It also has many **registers**, which store information and act as temporary memory.

CPUs Have Many Logic Units

For our purpose, the most interesting part is the **logic units**. These are a set of circuits that can perform basic arithmetic operations (like addition or multiplication).

Importantly, the CPU has many **duplicates** of these- it might have hundreds of logic units that all perform addition.
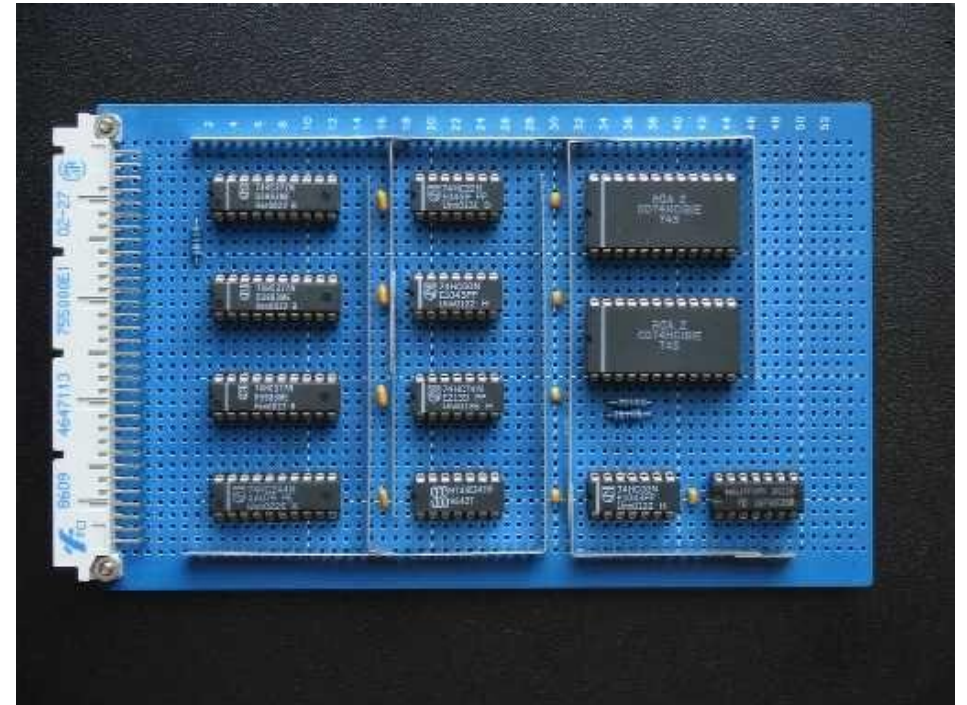
1: Circuit-Level Concurrency

The first level of concurrency happens within a single CPU, or **core**. Because the CPU has many arithmetic units, it can break up complex mathematical operations so that subparts of the operation run on separate logic units **at the same time**.

For example, if a computer needs to compute (2 + 3) * (5 + 7), it can send (2 + 3) and (5 + 7) to two different addition units **simultaneously**. Once it gets the results, it can then send them to the multiplication unit. This only takes **two time steps**, instead of three.

Concurrency Trees

A **concurrency tree** is a tree that shows how a complex operation can be broken down into the fewest possible time steps.

Actions which occur simultaneously are written as nodes at the same **level** of the tree. Nodes are on the same level when they are the same distance from the root.

The **total** number of steps is the number of non-leaf nodes in the tree. This example tree has three total steps.

The number of **time-steps** is the number of non-leaf **levels** in the tree. This example tree has two time-steps.

```
        (2+3) * (5+7)
             ↑
        time-step 2
       /            \
    2+3              5+7
   time-step 1
   /    \          /    \
  2      3        5      7
```
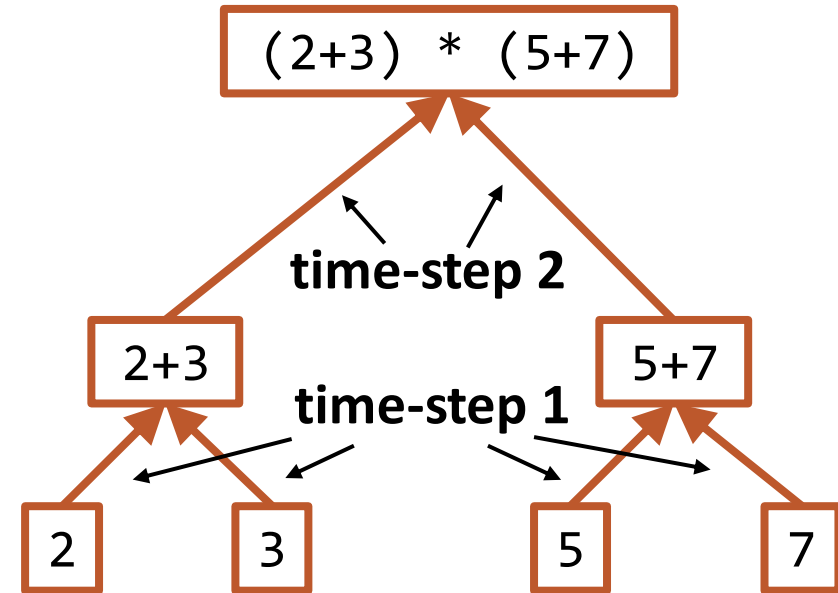
Example Concurrency Tree

For example, let's make a concurrency tree for

(a*b + c*(d/4)) * (g + f*h)

In the first time-step, we can compute a*b, d/4, and f*h.

The next time-step contains the operations that **required** those computations to be done already – c*(d/4) and g + f*h.

In general, the operations at each level could not be done any earlier in the process.

This tree has **seven total steps** and **four time-steps**.
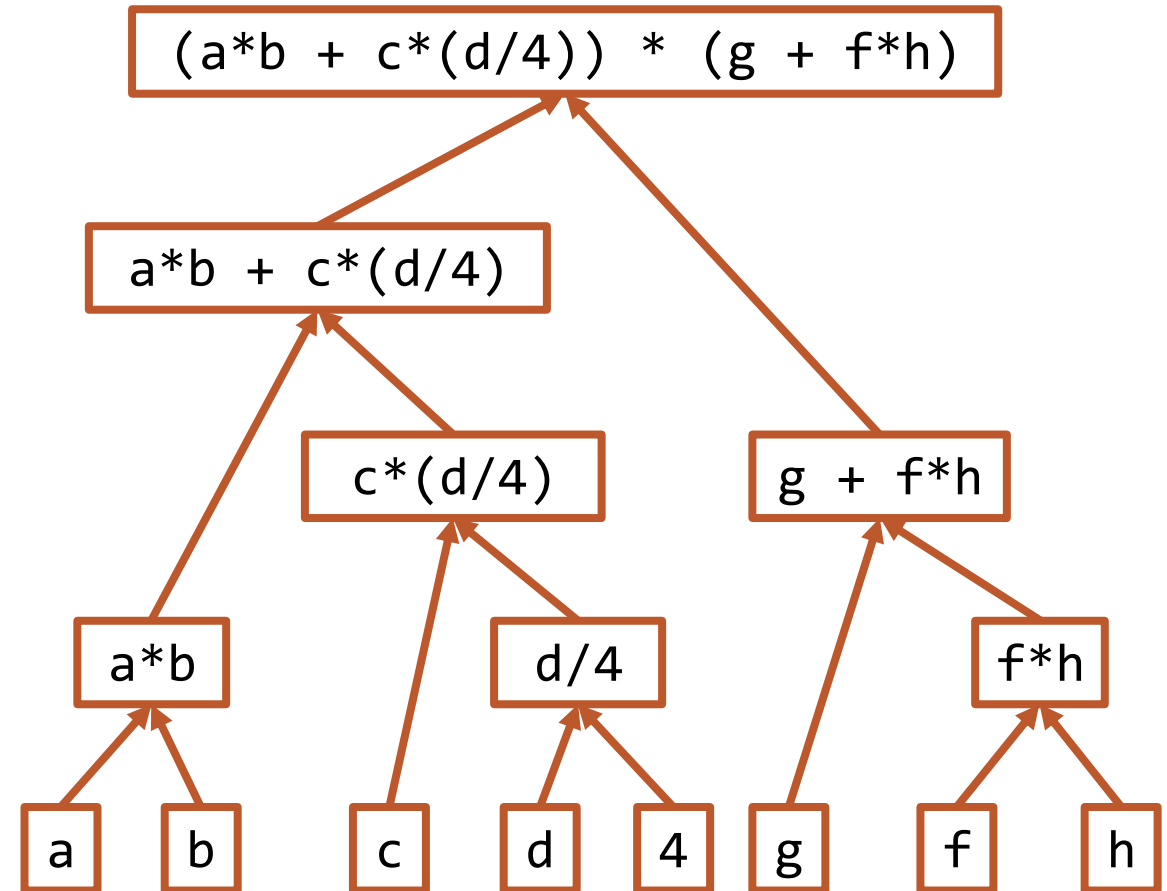
Activity: Count Equation Steps

Consider the following equation:

```
((a*b + 1) - a) + ((c/2) * (d*e + f))
```

How many **total steps** does it take to compute this equation?

How many **time-steps** does it take to compute this equation?

Hint: If you aren't sure, try drawing a concurrency tree!

Activity Solution

Total steps: 8

Time steps: 4
- a*b, c/2, d*e
- a*b + 1, d*e + f
- (a*b + 1) - a, (c/2) * (d*e + f)
- ((a*b + 1) - a) + ((c/2) * (d*e + f))

2: Multitasking

The second level of concurrency is **multitasking**.

This level is very different from the others in that it doesn't actually run multiple actions at the same time. Instead, it creates the **appearance** of concurrent actions.

CPU Schedulers Arrange Programs

Multitasking is accomplished by a part of the operating system called a **scheduler**. This is a component that decides which program action will happen next in the CPU.

When your computer is running multiple applications at the same time – like your browser, and a word editor, and Pyzo – the scheduler decides which program gets to use the CPU at any given point.

Multitasking with a Scheduler

When multiple applications are running at the same time, the scheduler can make them **seem** to run at the same time by breaking each application's process into steps, then alternating between the steps rapidly.

If this alternation happens quickly enough, it looks like true concurrency to the user, even though only one process is running at any given point in time.

Process 1:

step1   step 2   step3

run   run   run

Process 2:

step1   step 2

run   run
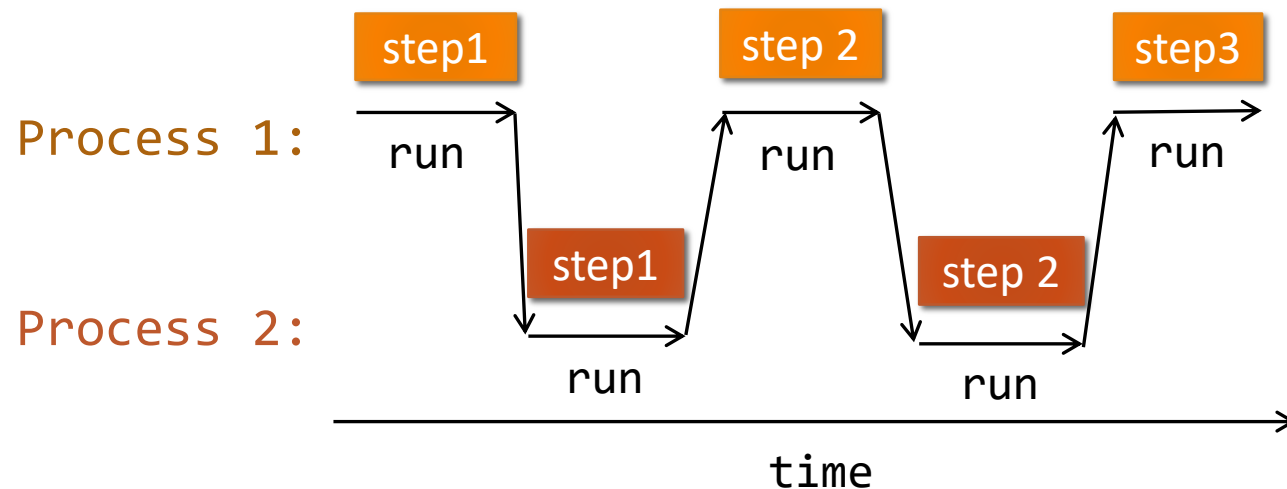
time

Schedulers Can Choose Any Order

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, then switch to one step of another, then run all the steps of a third. It might even choose to put a process on hold for a long time, if it isn't a priority.

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.

Process 1:

Process 2:

Process 3:

step1   step 2

run   run

step1   step 2

run   run

step1

run

time
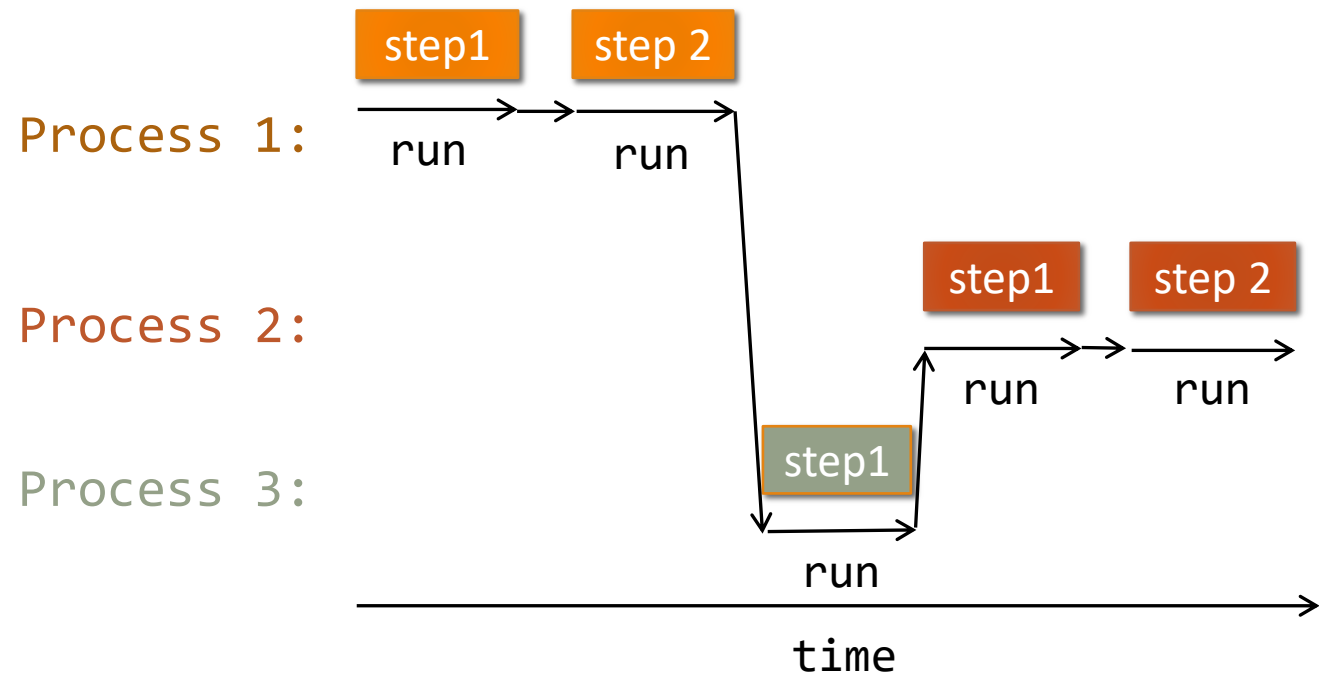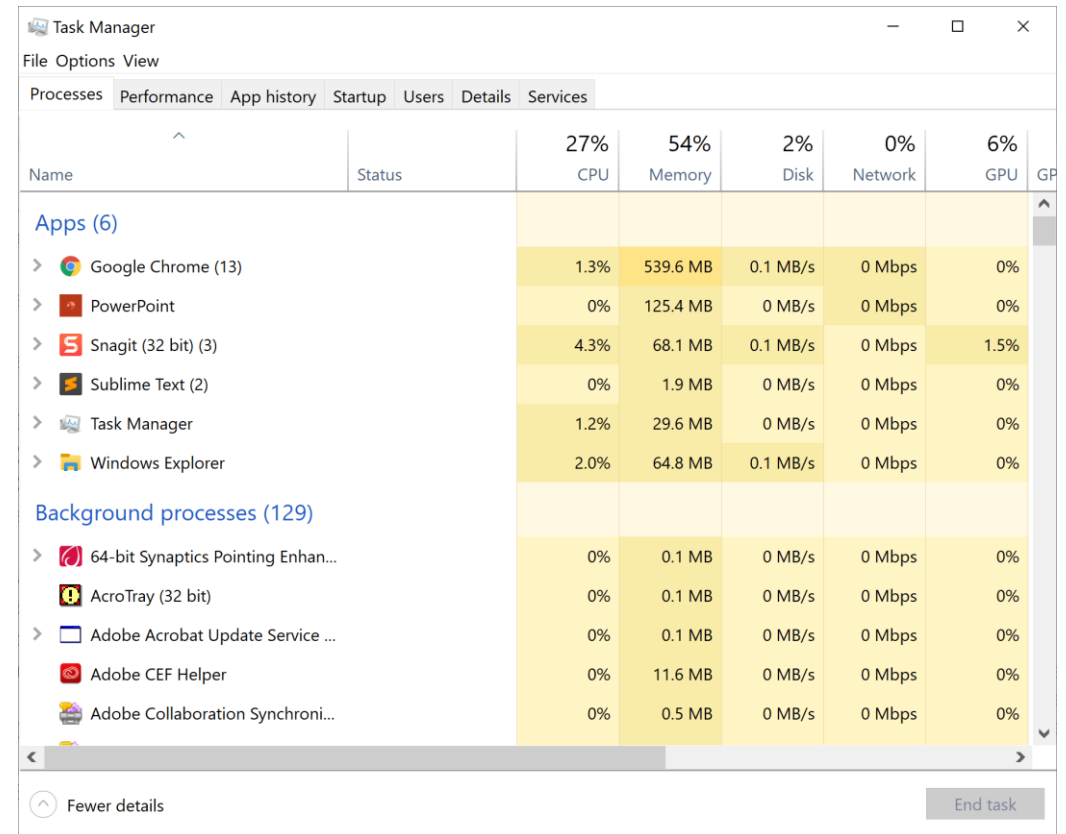
Your Computer Multitasks

Your computer uses multitasking to manage all of the applications you run, as well as the background processes needed to make your operating system work.

You can see all the applications your computer's scheduler is managing by going to your process manager (Task Manager on Windows, Activity Monitor on Macs). 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

3: Multiprocessing

The third level of concurrency, **multiprocessing**, can run multiple applications **at the exact same time** on a single computer.

To make this possible, we put **multiple CPUs** inside a single computer, then run different applications on different CPUs at the same time.

By multiplying the number of actions we can run at a point in time, we multiply the speed of the computer.

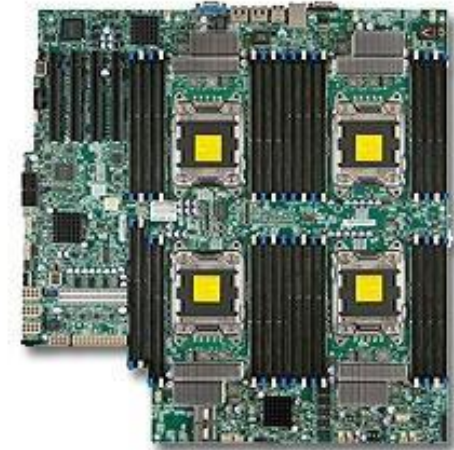# Multiple Processor vs. Multi-Core

Technically there are two ways to put several CPUs into a single machine.

The first is to insert more than one processor chip into the computer. This is called **multiple processors**.
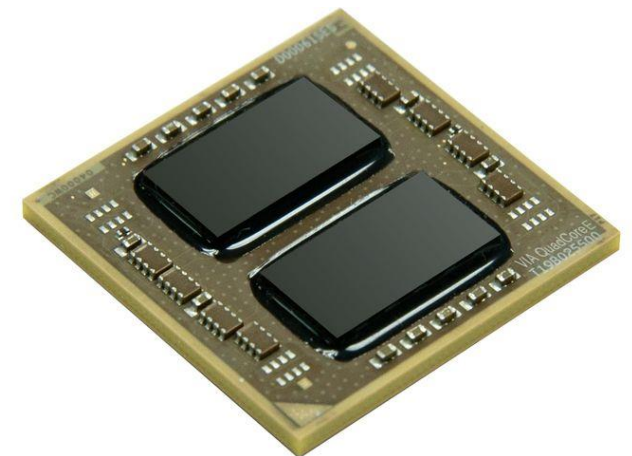
The second is to put multiple 'cores' on a single chip. Each core can manage its own set of actions. This is called **multi-core**.

There are slight differences between these two approaches in terms of how quickly the CPUs can work together and how they access memory. For this class, we'll treat them as the same.
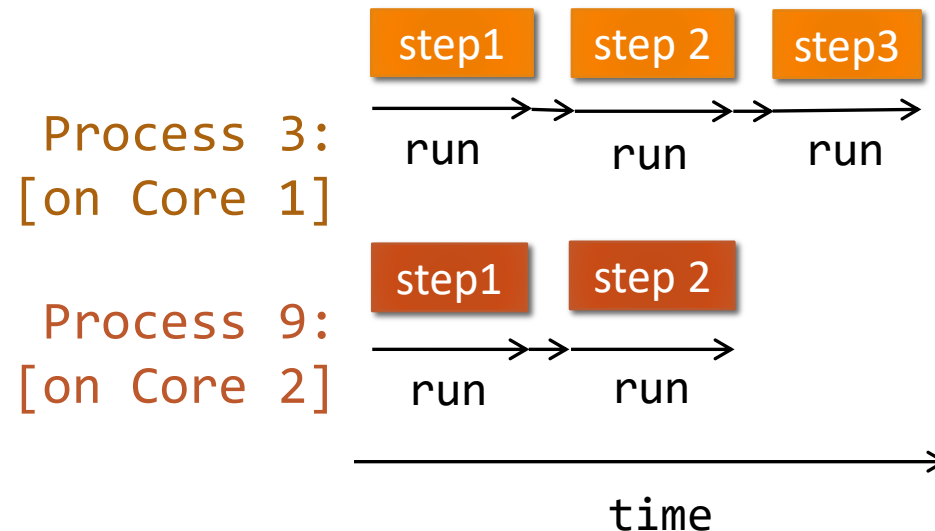
Multiple Processors

Multi-Core

Scheduling with Multiprocessing

When we use multiple cores and multiprocessing, we can run our applications simultaneously by assigning them to different cores.

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

Process 3:
[on Core 1]

step1   step 2   step3
run   run   run

Process 9:
[on Core 2]

step1   step 2
run   run

time

Simplified Scheduling

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

Core 1   Microsoft Word

Core 2   Firefox

Core 3   Pyzo

Core 4   Zoom
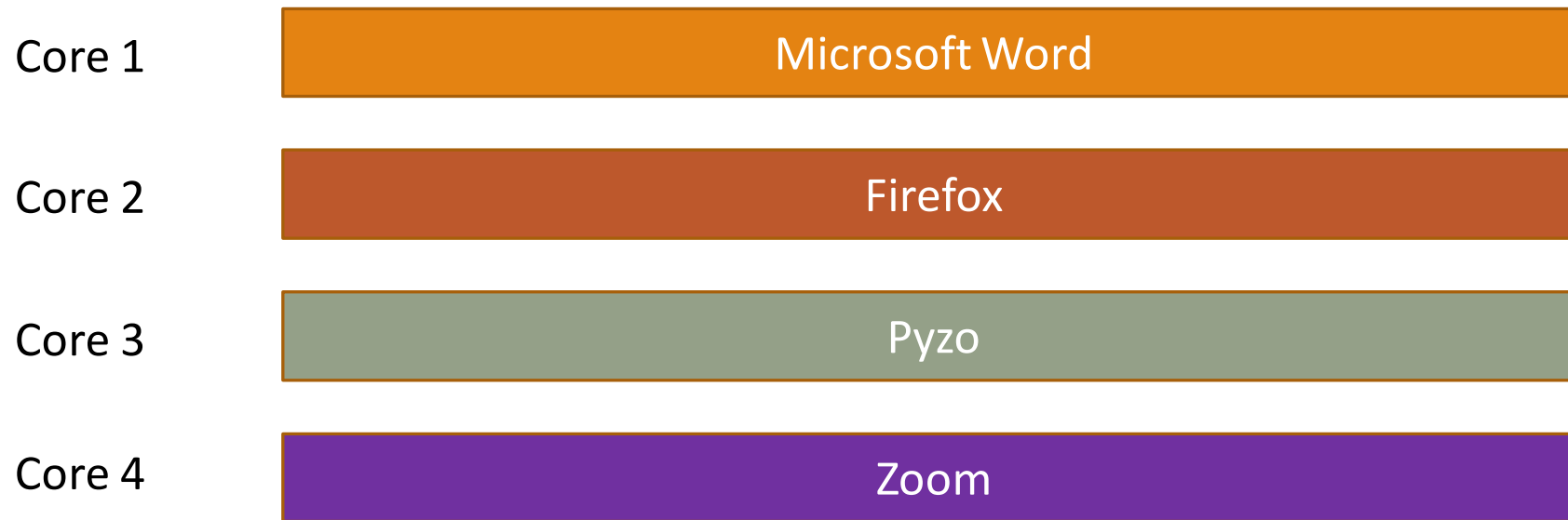
Multiprocessing and Multitasking

The number of cores we have on a single computer is usually still limited. Most modern computers use somewhere between 2-8 cores. If you run more than 2-8 applications at the same time, 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.

You do: look up how many cores your computer has!

Scheduling with Multiprocessing and Multitasking

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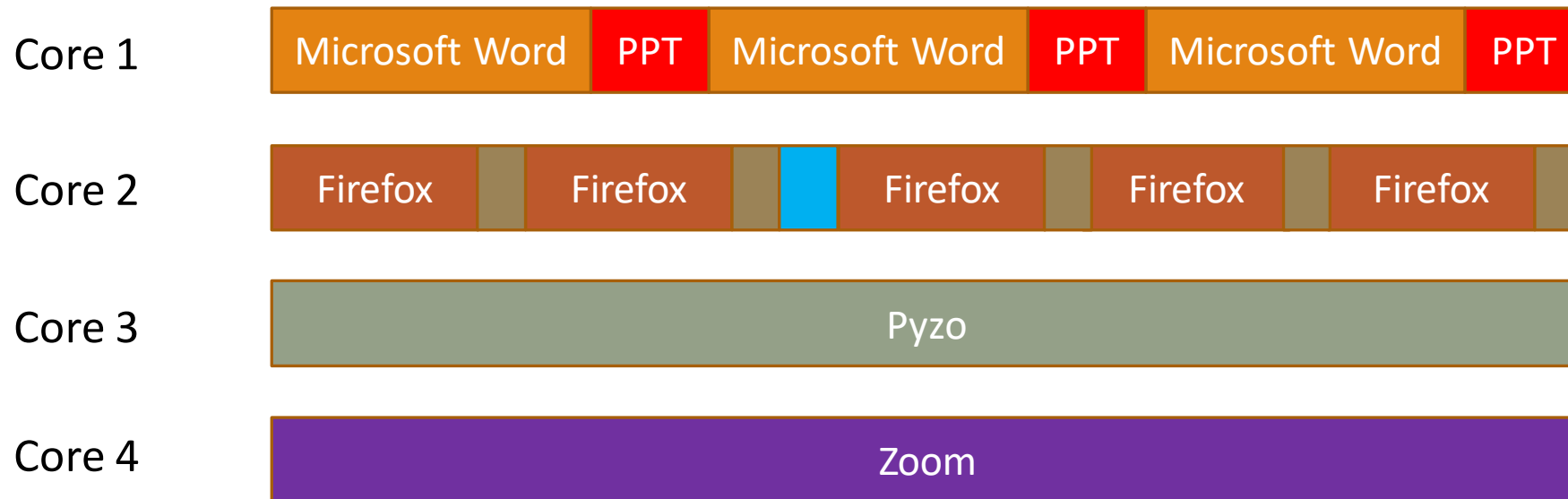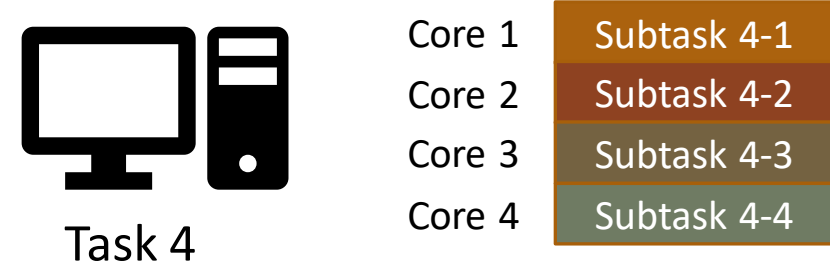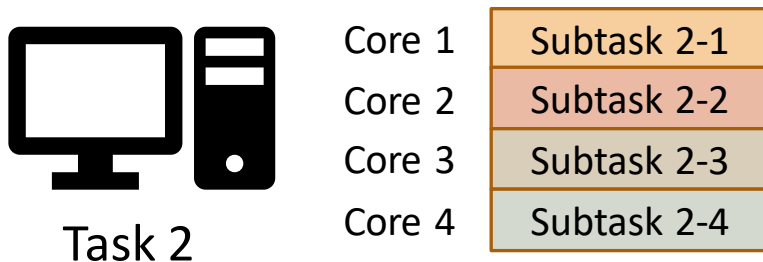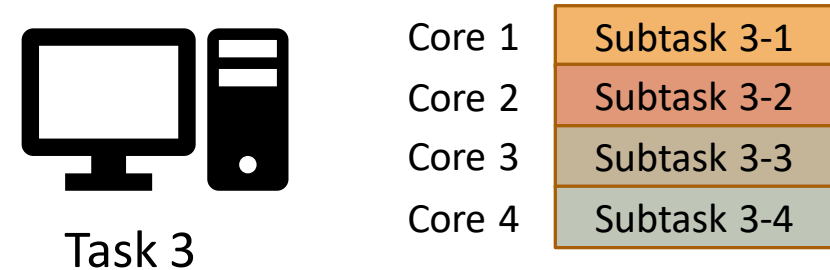
4: Distributed Computing

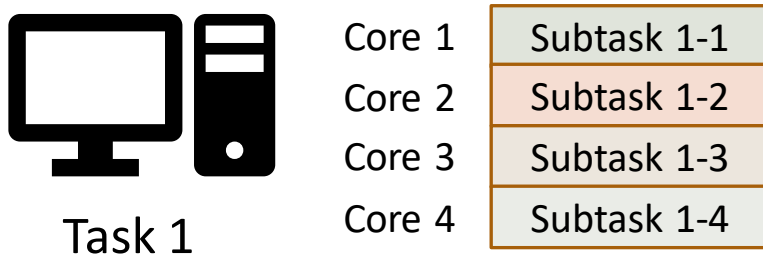The final level of concurrency, **distributed computing**, goes beyond using a single machine.

If we have access to several computers (each with its own set of CPUs), we can network them together and use them all to perform advanced computations by assigning different subtasks to different computers.

By multiplying the number of computers that are working on a single problem, we can multiply the speed of a difficult computation.

Scheduling with Distributed Computing

Each computer in the network can take a single task, break it up into further subtasks, and assign those subtasks to its cores. This makes it possible for us to attempt to solve problems which would take a long time to solve on a single processor.

Task 1

Core 1	Subtask 1-1
Core 2	Subtask 1-2
Core 3	Subtask 1-3
Core 4	Subtask 1-4

Task 2

Core 1	Subtask 2-1
Core 2	Subtask 2-2
Core 3	Subtask 2-3
Core 4	Subtask 2-4

Task 3

Core 1	Subtask 3-1
Core 2	Subtask 3-2
Core 3	Subtask 3-3
Core 4	Subtask 3-4

Task 4

Core 1	Subtask 4-1
Core 2	Subtask 4-2
Core 3	Subtask 4-3
Core 4	Subtask 4-4

Companies Use Distributed Computing

Distributed computing is used by big tech companies (like Google and Amazon) both to manage thousands of customers simultaneously and to process complex actions quickly.

This is where the term 'server farm' comes from- these companies will construct large buildings full of thousands of computers which are all networked together and ready to process information.

A **supercomputer** is very similar to distributed computing. It's a computer with a *huge* number of processors connected together. The main difference is that all the processors are located in the same place.

Distributed Computing Must Be Fault Tolerant

When using distributed computing, it's very important that algorithms are designed to be **fault tolerant**.

The probability that a computer randomly crashes while running a program is low (maybe 1 in 10,000). But server farms regularly run far more than 10,000 computers at the same time.

Algorithms that run on distributed systems must be designed to have checks in place to make sure that no work is left unfinished. Typically, storage is also backed up on multiple machines, to make sure no data is lost if a single machine goes down.

Difficulties in Parallelization

Designing Concurrent Algorithms

Now let's discuss how design algorithms so that they can run concurrently. This is often referred to as **parallel programming**.

We won't actually write parallelized code in this lecture (apart from a bit of MapReduce code where the parallelization is provided for us), but we will discuss common problems and algorithms in the field.
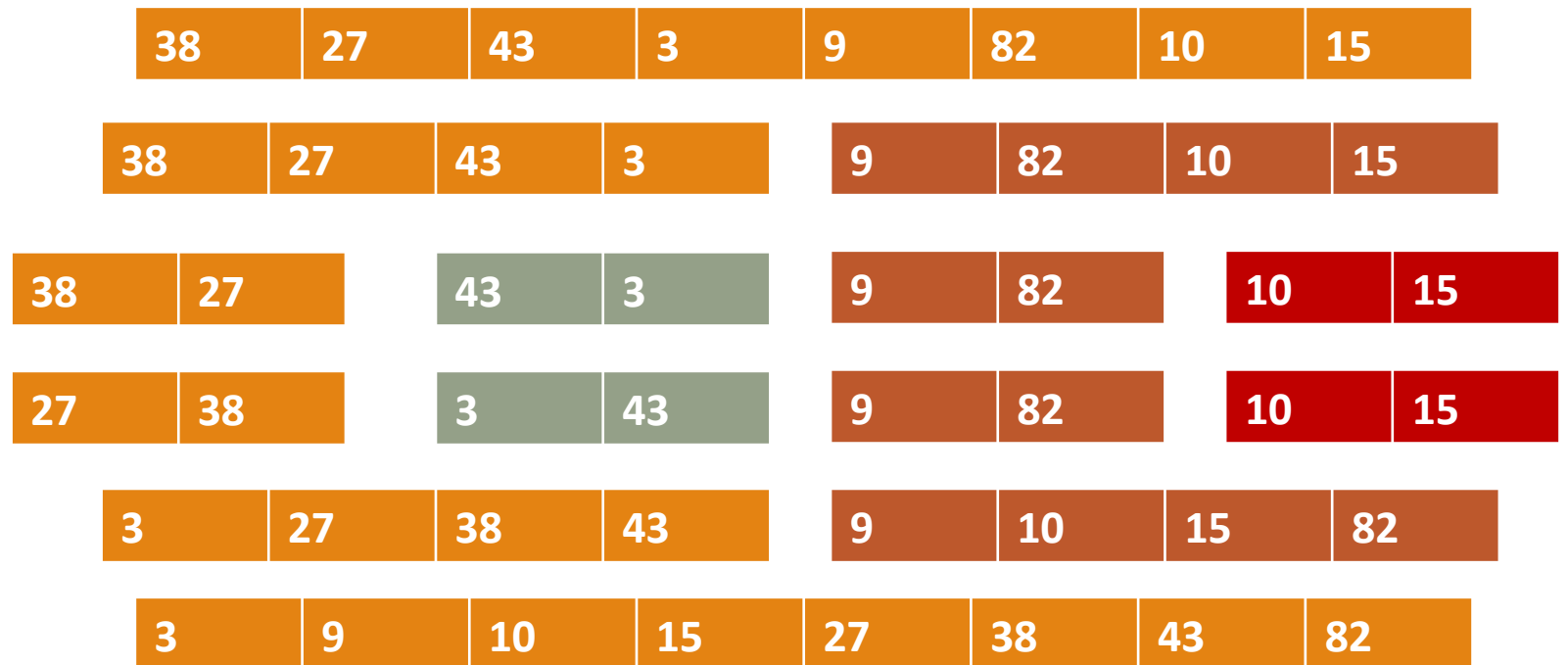
Difficulty of Design

Parallel programming is more difficult than regular programming because it forces us to think in new ways and adds new constraints to the problems we try to solve.

First, we must figure out how to design algorithms that can be split across multiple processes. This varies greatly in difficulty based on the problem we're solving!

Making Merge Sort Concurrent

Let's start with an easy example – the algorithm **Merge Sort**, which sorts a list by splitting the list in half, recursively sorting each half of the list, and then merging the two halves.

This adapts nicely to concurrency; instead of running mergeSort on the two halves of the lists sequentially, it runs them **concurrently**. Then it sends the results back to a single core to be merged.

| 38 | 27 | 43 | 3 | 9 | 82 | 10 | 15 |

| 38 | 27 | 43 | 3 | | 9 | 82 | 10 | 15 |

| 38 | 27 | | 43 | 3 | | 9 | 82 | | 10 | 15 |

| 27 | 38 | | 3 | 43 | | 9 | 82 | | 10 | 15 |

| 3 | 27 | 38 | 43 | | 9 | 10 | 15 | 82 |

| 3 | 9 | 10 | 15 | 27 | 38 | 43 | 82 |

Making Loops Concurrent

It's easy to make recursive problems like merge sort concurrent if they make multiple recursive calls. It's harder to think concurrently when writing programs that use 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.

```
def search(lst, target):
    for item in lst:
        if item == target:
            return True
    return False
```

```
def getSum(lst):
    sum = 0
    for item in lst:
        sum = sum + item
    return sum
```

```
def powersOf2(n):
    i = 2
    while i < n:
        print(i)
        i = i * 2
```

Sharing Resources

The next difficulty of writing parallel programs comes from the fact that 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.

Locking and Yielding Resources

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 Stalls the System

In general, this system of locking and yielding fixes most cases where programs might try to use a resource at the same time. But there are some situations where it can cause trouble.

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**.
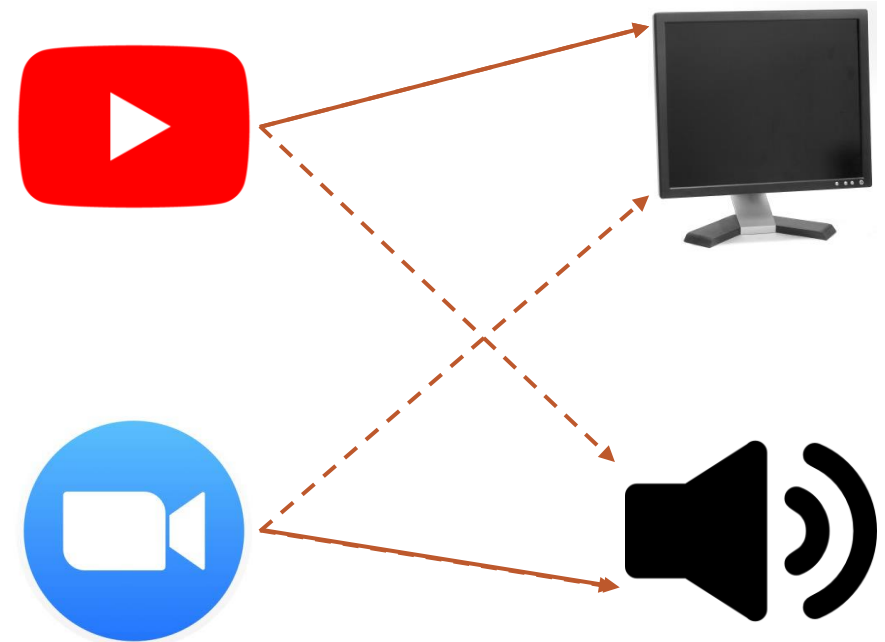
Deadlock Definition

In general, we say that deadlock occurs when two or more processes are all waiting for some resource that other processes in the group already hold. This will cause all processes to wait forever without proceeding.

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.
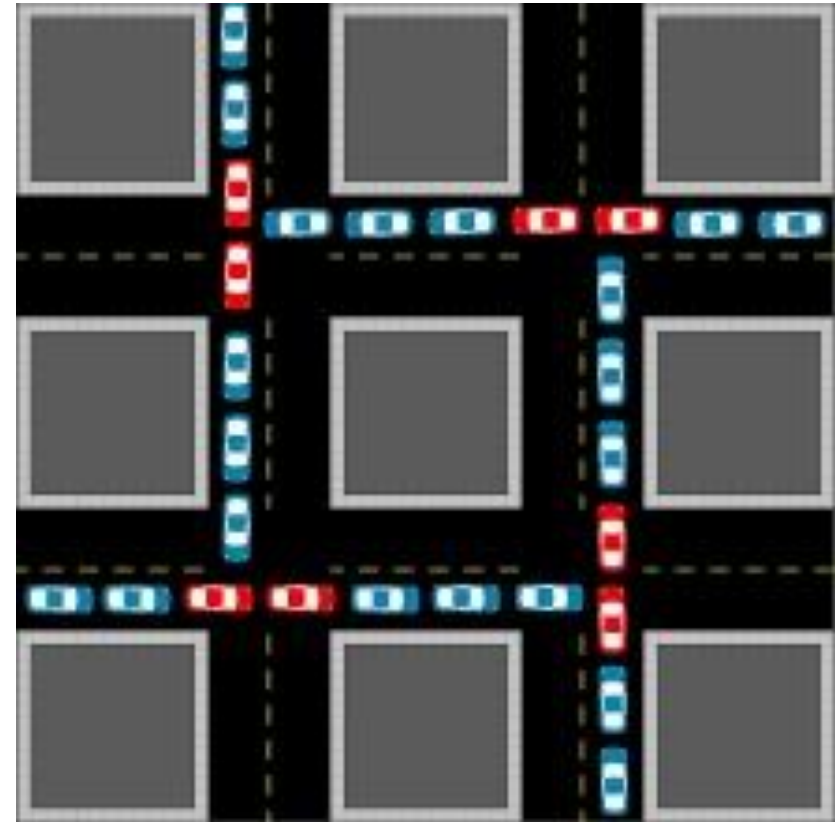
Fix Deadlock With Ordered Resources

In order to fix deadlock, 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.
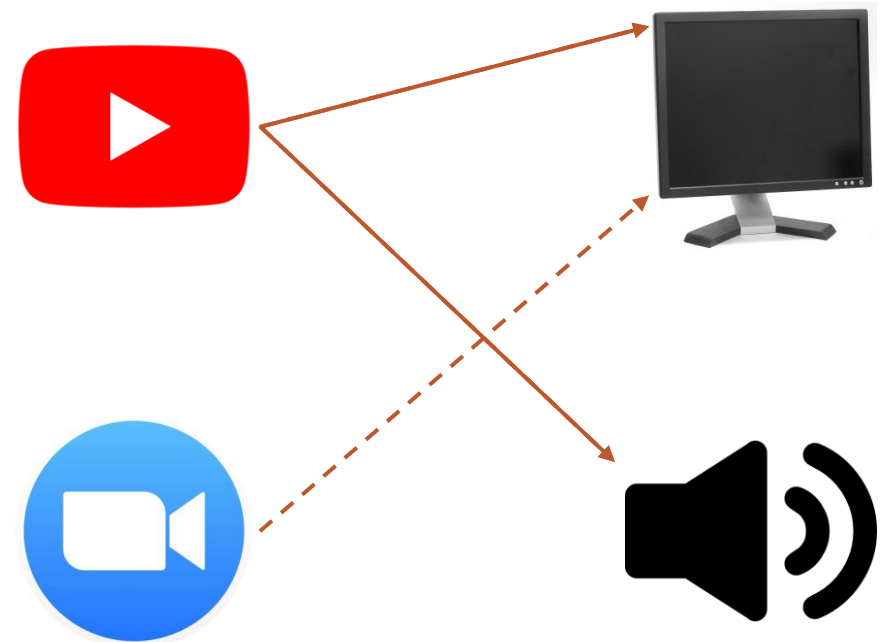
Activity: Dining Philosophers

Another example of deadlock occurs in the Dining Philosophers problem.

Several philosophers sit down at a circular table to eat. Each thinks for a while, then picks up their left fork, then picks up their right fork, then eats a bit. Then they put down the forks to think some more, then eat some more, etc.

You do: How can these philosophers get into deadlock? How can we solve that deadlock?

Socrates

Aristotle

Plato

Homer
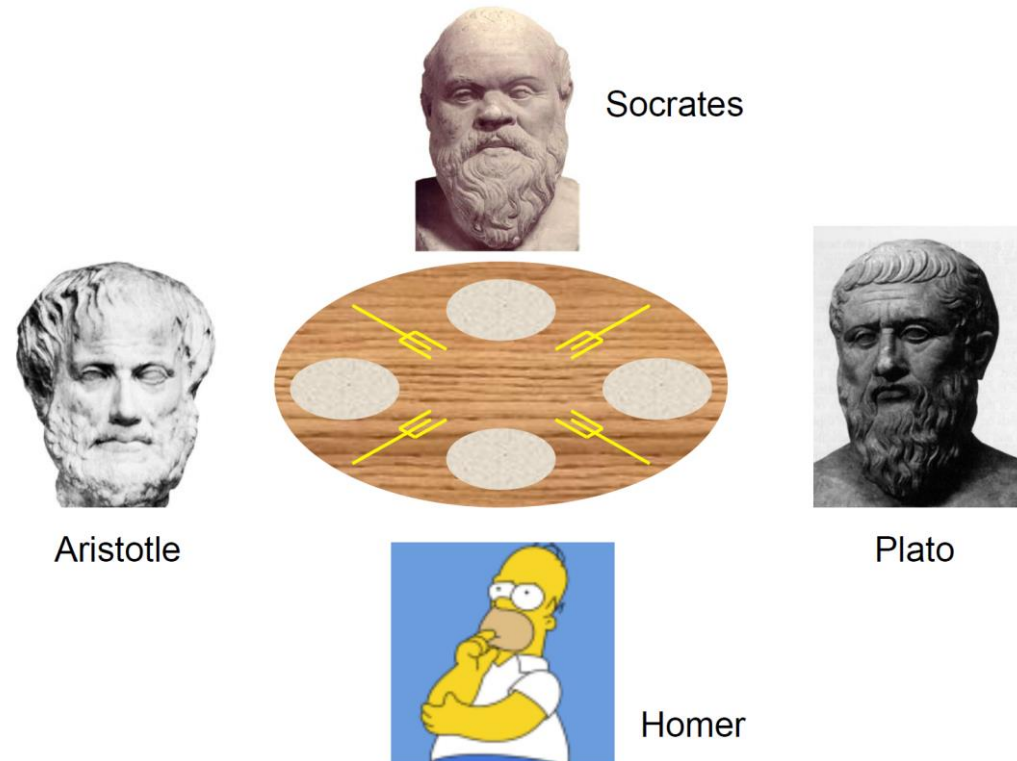
Activity Solution

Problem: if every philosopher picks up their left fork, no one will be able to eat.

Solution: number the forks. Every philosopher picks up the fork with the earlier number first. For most philosophers this will still be the left fork, but for one (the last philosopher) it will be the right.

Communication in Concurrency

Some Processes Need to Communicate

We can't always guarantee that the processes running concurrently on a computer are independent. Sometimes a single program is split into multiple tasks that run concurrently instead.

These tasks might need to share partial results as they run. They'll need a way to **communicate** with each other.

Processes Pass Messages to Share Data

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: consider merge sort. Once a core has finished splitting it will need to wait for the result of the alternate core to merge the two halves of the list together.

Pipelining and MapReduce

Writing algorithms that can pass messages is tricky. We'll discuss two approaches that make it easier: **pipelining** and **MapReduce**.

The core idea behind pipelining is that you can parallelize an algorithm by splitting up the **algorithm** into a series of **consecutive steps**.

The core idea behind MapReduce is that you can parallelize an algorithm by splitting up the **data** into **many many small parts**.

Message Passing Example: Line Cooking

Let's introduce our two algorithms 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:
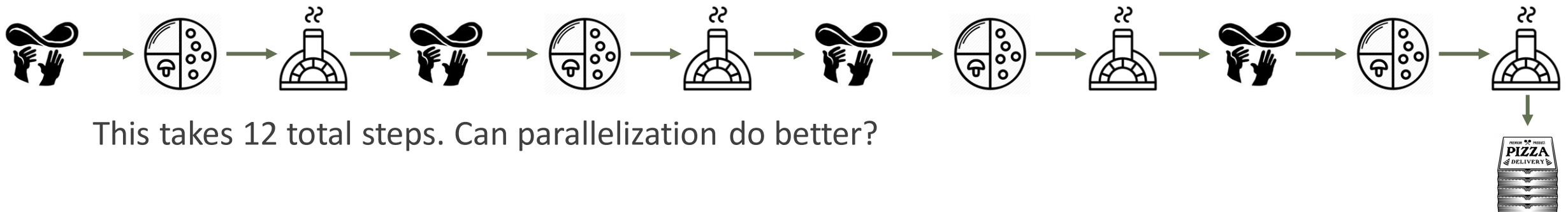
This takes 12 total steps. Can parallelization do better?

Pipelining

Pipelining Definition

One algorithmic process that simplifies parallel algorithm design is **pipelining**. In this process, you start with a task that repeats the same procedure over many different pieces of data.

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.
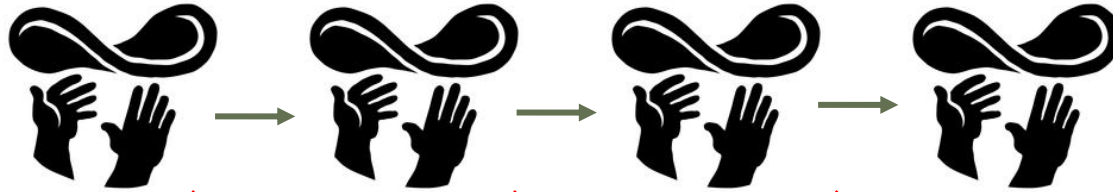
Just like in an assembly line, the cores can run **multiple pieces of data simultaneously** by starting new computations while the others are still in progress.
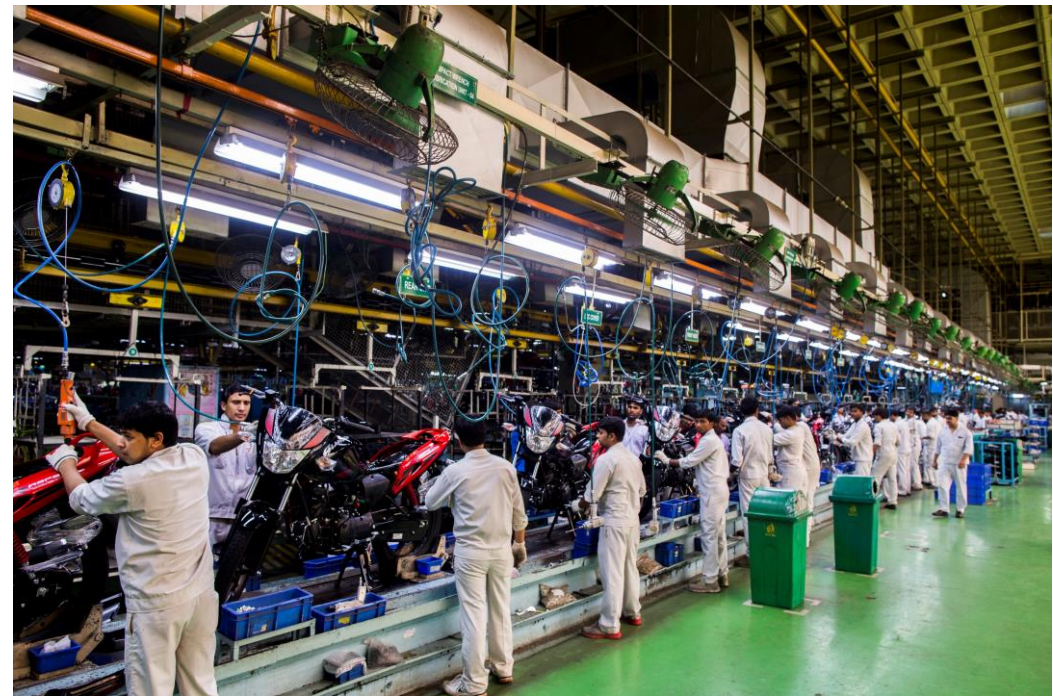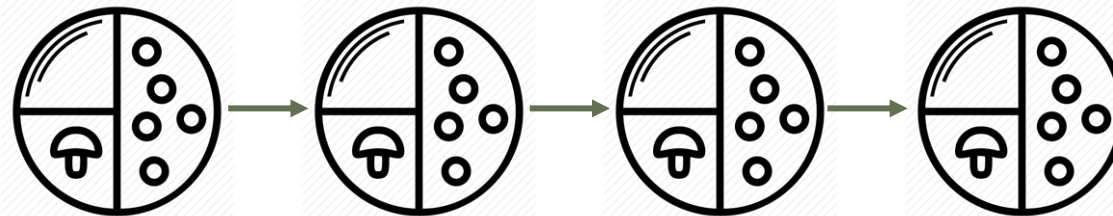
Pizza Pipelining —
3 workers, 1 oven, 6 time-steps

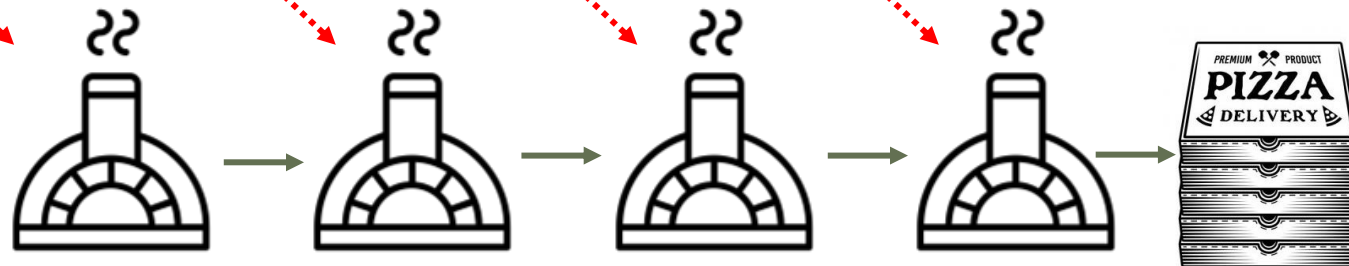Each worker has **one task**. #1 flattens dough, #2 arranges toppings, #3 bakes in the oven.

There are still 12 total steps, but there are only 6 **time-steps**.

Worker 1:

Worker 2:

Worker 3:

Rules for Pipelining

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.

Benefits of Pipelining

Pipelining is most useful when the number of shared resources is **limited**. For example, in pizza-making we may have only one oven; using pipelining ensures that we are constantly making use of the oven without wasting time.

Pipelining is also useful for tasks that require setup time, but then can run many times without further setup - maybe for flattening, the cook only has to clean the counter and flour it once.

Another Example: Laundry Without Pipelining

You probably already use pipelining when you do laundry. Let's look at an example where we assume you need to wash, dry, and fold several loads of laundry. Washing [W] takes 30 minutes; drying [D] takes 45; folding [F] takes 15.

If you don't use pipelining and wait until a load of laundry is folded before starting the next one, doing four loads of laundry takes **six hours**.
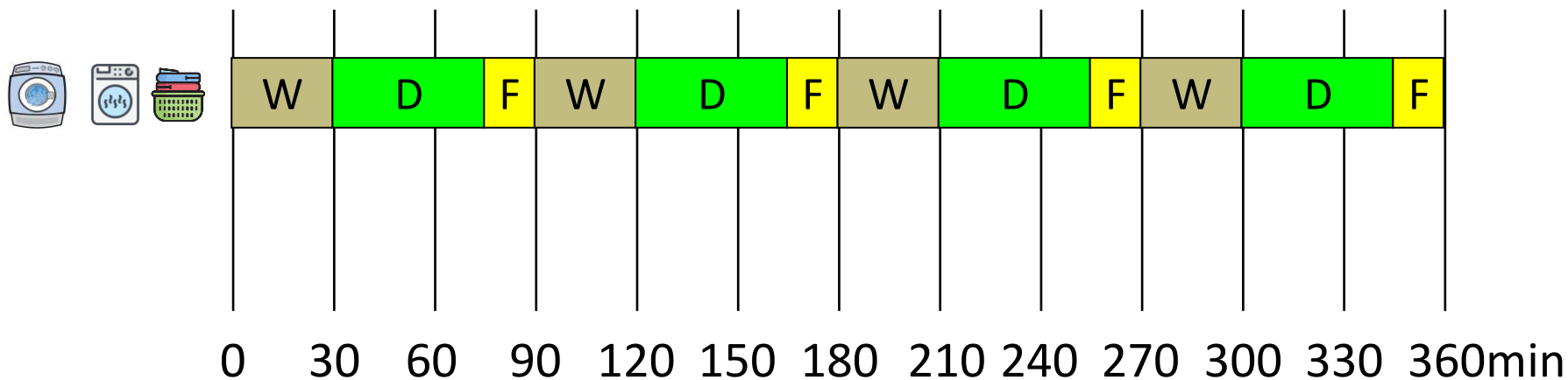
Example: Laundry With Pipelining

To use pipelining, split the three steps of the laundry process across three workers: the washer, dryer, and folder. Each worker has a lock on the **shared resource**.

With pipelining, four loads of laundry only takes **3 hours and 45 minutes**. Much faster!

[In reality, you alternate between these tasks and the machines do the work; you just start the machines. So the **machines** are the workers in this scenario.]
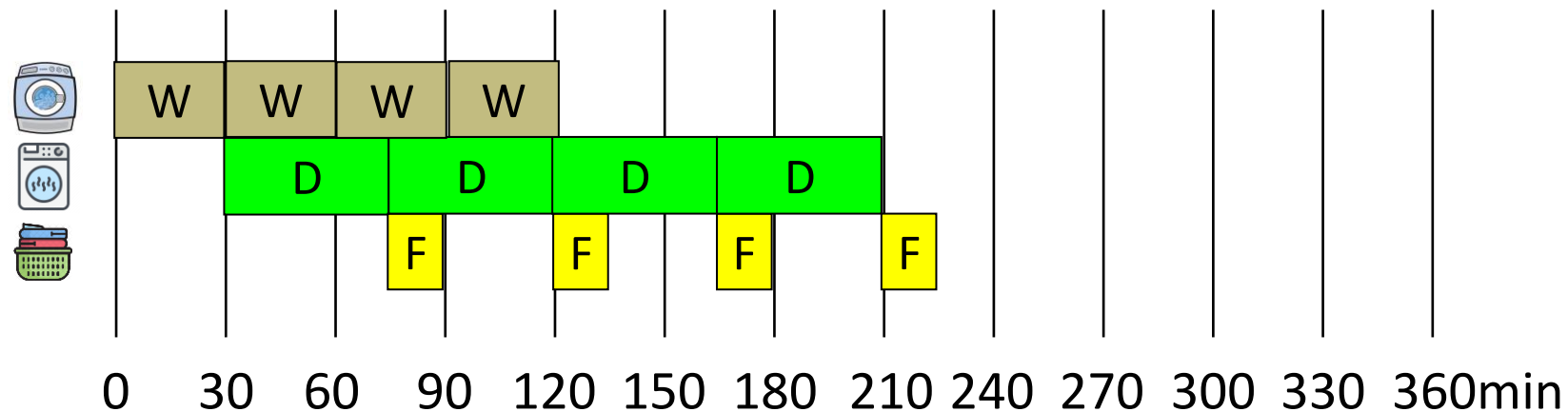
Activity: Design a Pipeline

The process of writing a thank-you card has three sequential steps: **Writing** the note [**10min**], **Adding** the address to the envelope [**6min**], and **Stuffing** the envelope [**6min**]. Because you hate writing thank-you cards, you've decided to hire two helpers (your younger siblings) to help with the work.

You need to **write all the notes yourself**, to make sure they're personalized, but you can outsource the other tasks to the helpers once the card has been written

By yourself, you can write 2 full thank-you cards in an hour (plus part of a third). If you use pipelining and the three workers (yourself + two helpers), **how many completed thank-you cards can you make in an hour**?

Hint: try drawing this out the way we drew out the washer/dryer/folder example, but with writer/adder/stuffer as the three roles.

Activity Solution

Answer: **four letters**

It takes 22min to finish a single letter, and you start writing a new letter every ten minutes. Because writing is the longest task, the other two tasks will need to wait on it.

Letter 1: 00:00-00:22

Letter 2: 00:10-00:32

Letter 3: 00:20-00:42

Letter 4: 00:30-00:52

Letter 5 is partially completed at the hour mark (00:40-00:62)

Pipelining in Computer Science

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

Pipelining is often combined with **multiprocessing** to split the operations being performed across multiple cores. This helps ensure that no core goes unused.
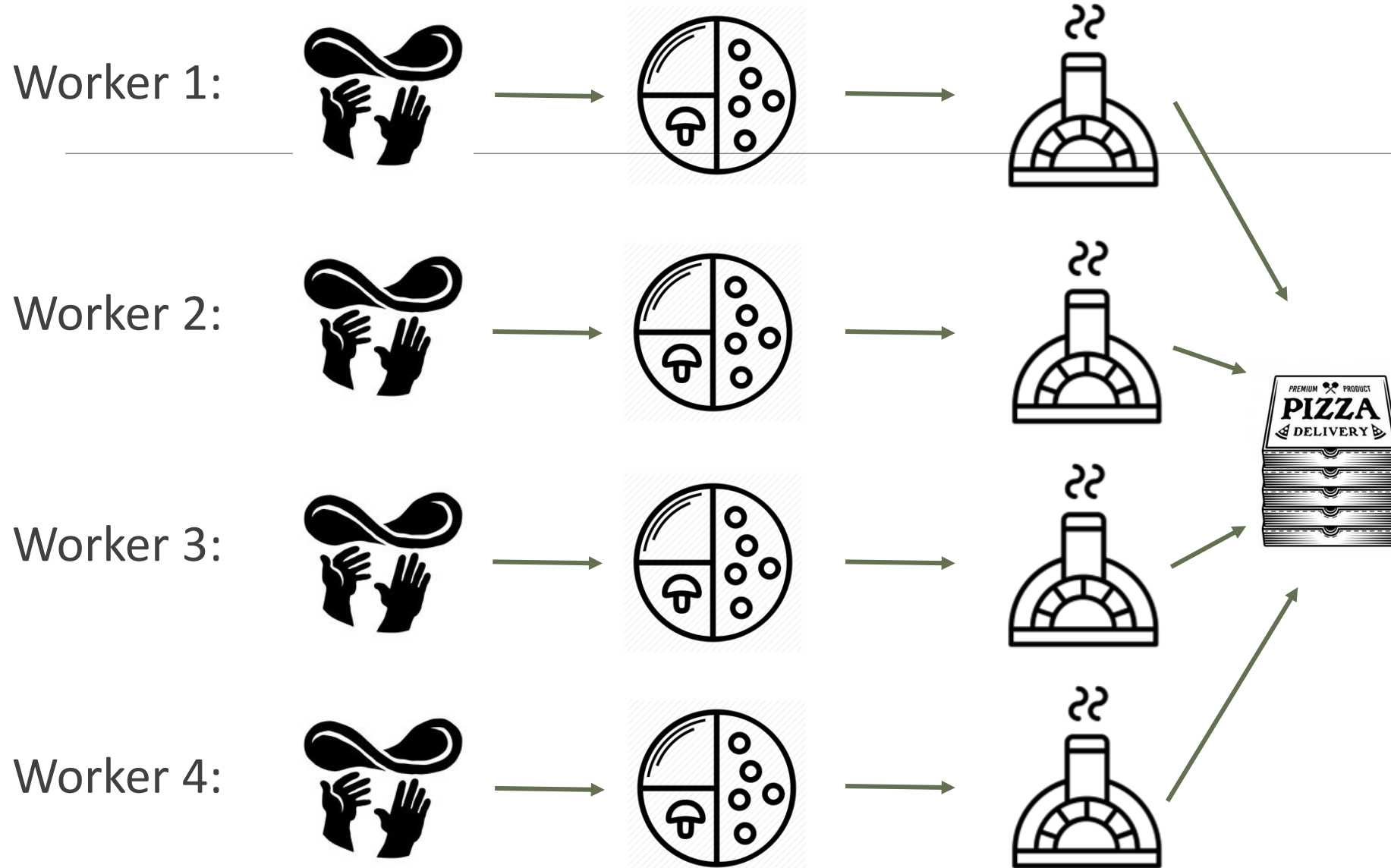
MapReduce

MapReduce Organizes Concurrency

Another popular algorithm for organizing parallelized programs is called **MapReduce**. Instead of breaking up a procedure's steps across different cores, this algorithm takes a **large data set** and breaks up the data itself across the cores.

This is a really effective approach if you have a lot of cores to work with (like in distributed computing). It's also a great approach for any problem over **big data** – that is, giant data sets that take far too long to process sequentially.

MapReduce - 4 workers, 4 Ovens, 3 time-steps

Worker 1:

Worker 2:

Worker 3:

Worker 4:

Each worker makes one pizza instead of doing one task repeatedly.

If we have infinite ovens and infinite workers, we can make as many pizzas as we want in just 3 time-steps!

Making MapReduce Algorithms

The MapReduce approach is simple enough that we can discuss how to build algorithms that actually use it.

A MapReduce algorithm is composed of three parts.

The **mapper** takes a piece of data, processes it, and finds a partial result

The **reducer** takes a set of results and combines them together

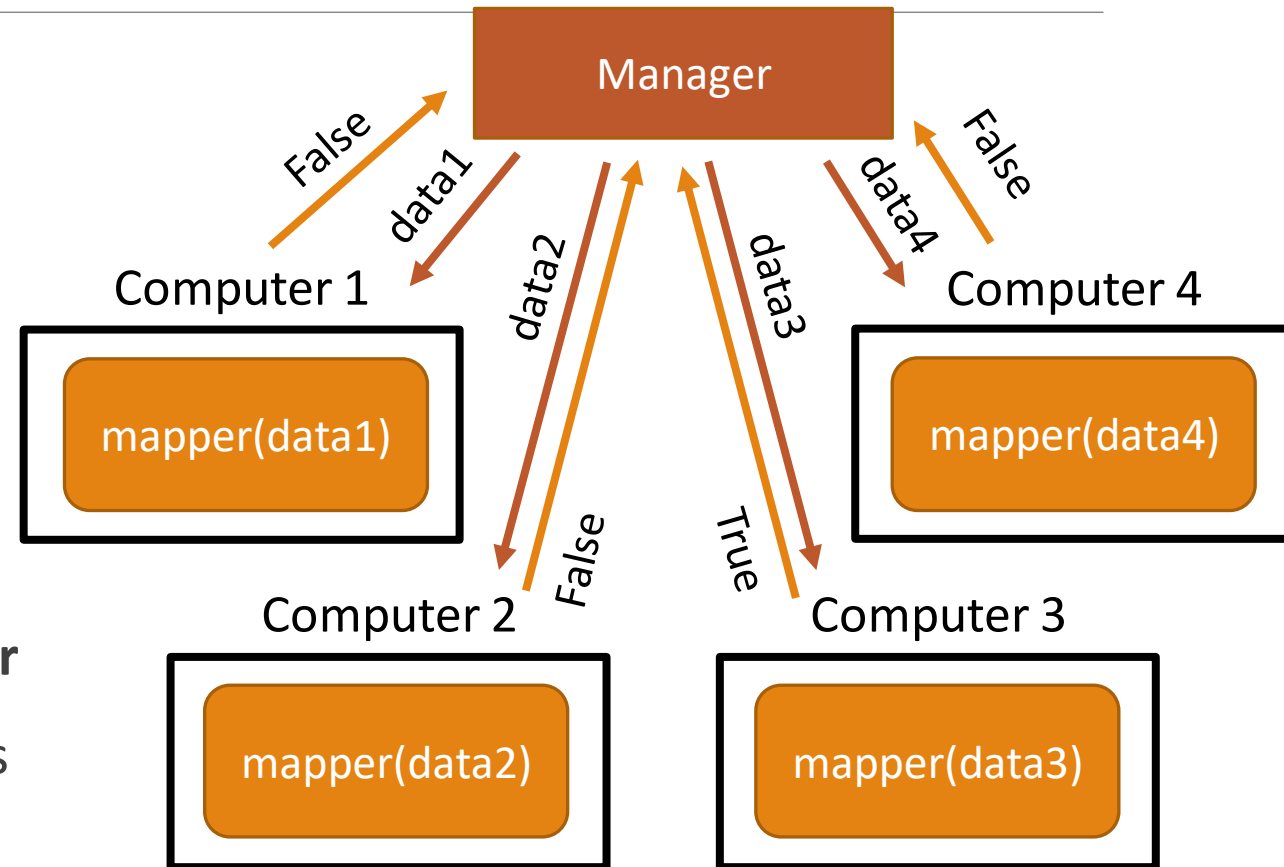The **manager** moves data through the process and outputs the final result
- Splits up data, sends to mappers, get results back
- Combines results together, sends to the reducer
- Gets the final result, outputs it

MapReduce Example: Search – Mapper

Let's say we want to search a book for a specific word. How can we split up this task?

First, the **manager** divides the book into many small parts- maybe one page per part. It sends each page to a different computer.

Each computer runs its copy of the **mapper** on its page. It returns True if it finds the result, and False otherwise. These results are sent back to the **manager**.

Manager

False

data1

data2

False

data3

True

data4

False

Computer 1

mapper(data1)

Computer 4

mapper(data4)

Computer 2

mapper(data2)

Computer 3

mapper(data3)

MapReduce Examples: Search – Reducer

Once all the mappers have returned their results the **manager** puts them all in a list and sends that list to the **reducer(s)**. The reducer combines the results together in some way.

There can be more than one reducer if there are lots of results to combine or if we're checking multiple things (like searching for more than one word). For now, we'll just use one.

Our reducer will check all of the results and send True back to the **manager** if any of them are True.

Manager

[False, False, True, False]

True

Computer 1

reducer(result)
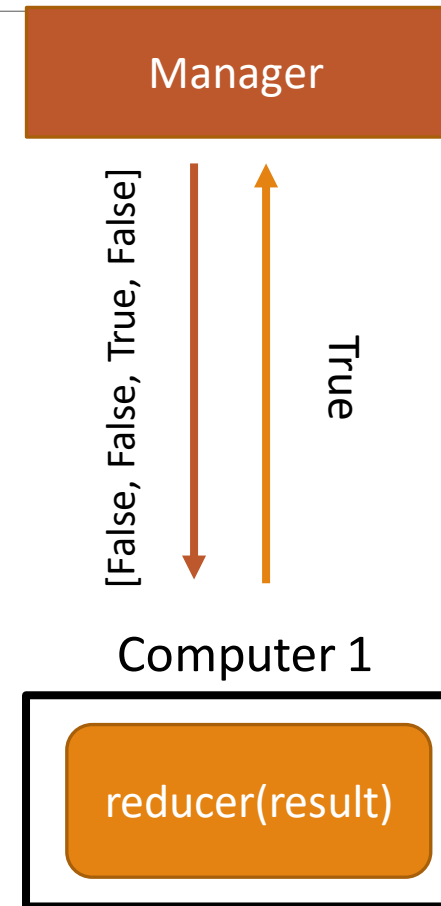
Coding MapReduce

We've provided a version of the MapReduce manager on the course website that uses multiprocessing to run the algorithm on several cores at the same time.

That makes implementing MapReduce easy- we just need to write code for the mapper and the reducer.

It's hard to tell that the system uses multiprocessing, but we can print out partial work to show what's happening. You need to end the process (by clicking the 'Terminate and restart the interpreter' button) to see what was printed in the individual calls.

```python
# Assume the page is in a file
def mapper(filename, target):
    # don't worry about reading/cleaning files
    # yet – we'll get there soon!
    text = cleanFile(readFile(filename))
    words = text.split(" ")
    for i in range(len(words)):
        word = words[i]
        if word == target:
            print("file", f, "found on word #", i)
            return True
    print("file", f, didn't find it")
    return False


# If the word is on any page, return True
def reducer(lst):
    print("reducer is checking", lst)
    for pageResult in lst:
        if pageResult == True:
            return True
    return False
```

MapReduce Efficiency

MapReduce can process huge data sets and get results quickly because it takes a list of length N and breaks it up into **constant-size parts**.

The core assumption is that we have enough computers to make the data pieces really small. If we process 1 million data points with 100,000 computers, each computer only needs to handle 100 data points.

Runtime of Search with N={1000,1M,1B,1T} items

LinearSearch   MapReduceSearch
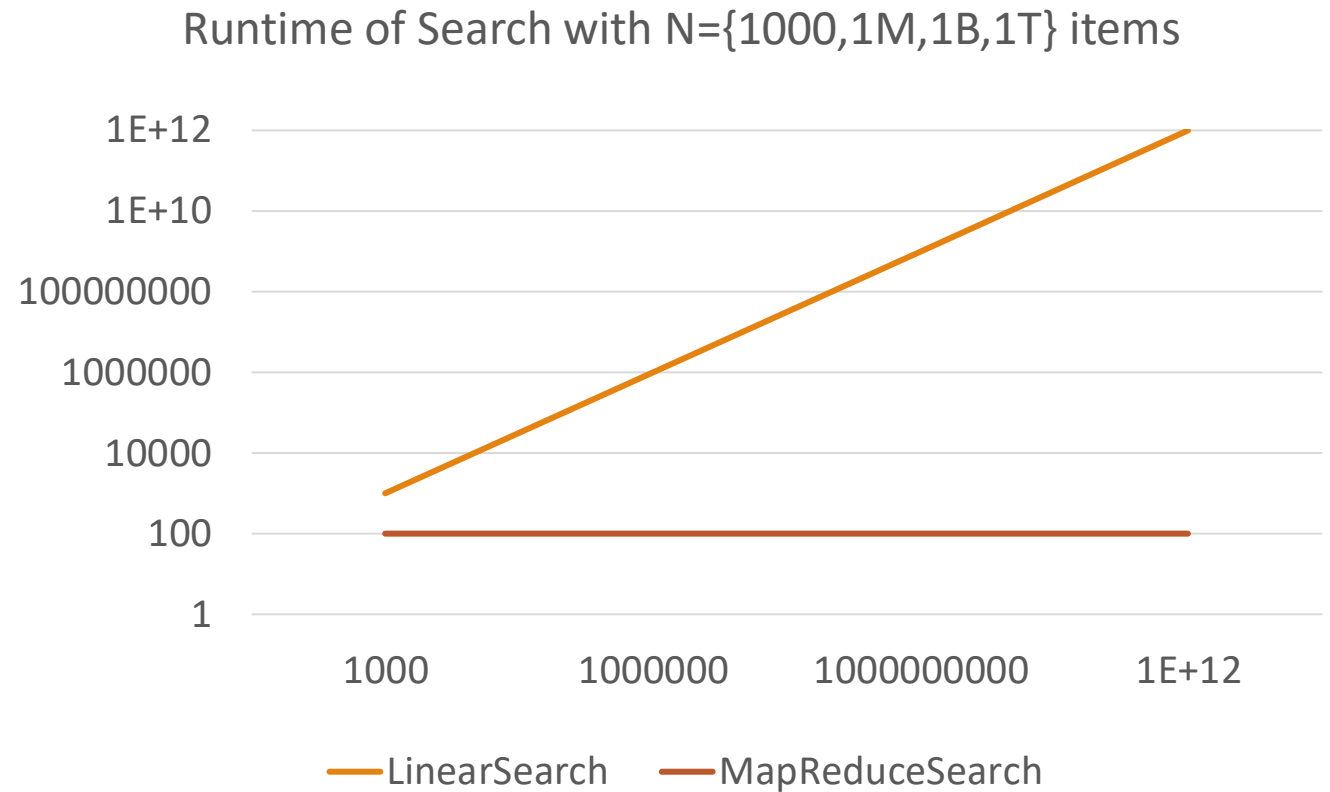
Another Example: Counting

What if we instead wanted to count the number of words across all of Wikipedia?

First, the **manager** breaks up the data- maybe each Wikipedia entry goes to a computer.

The **mapper** can take a single page and count all the words on it.

The **manager** takes all those counts and puts them in a list.

The **reducer** takes the list of numbers and returns their sum.

Learning Goals

Define and understand the differences between the following types of concurrency: **circuit-level concurrency, multitasking, multiprocessing,** and **distributed computing**

Create **concurrency trees** to increase the efficiency of complex operations by executing sub-operations at the same time

Recognize certain problems that arise while multiprocessing, such as **difficulty of design** and **deadlock**

Create **pipelines** to increase the efficiency of repeated operations by executing sub-steps at the same time

Use the **MapReduce pattern** to design and code **parallelized algorithms** for distributed computing