File System (Fin.) && Scheduling Start

Operating Systems, Fall 2026

Instructors: Travis McGaha

Head TAs: Shriya Sane

Arnuv Batra
Khoi Dinh
Sydney Fitzgerald
Rania Souissi
Jeffrey Oduman
Abdullah Goher
Monisha Krothapalli
Veer Kakar
Kevin Xue
Shreya Mukunthan
Daniel Price
Wesley Liu
Leo Zhang

Logistics

- Penn-FAT is out!
  - It is due October 29th in pairs!
  - It is a HUGE assignment. And make sure to put in a lot of elbow grease.
  - Some questions on the midterm will ask about concepts enforced by this assignment.
  - Recitation this week is designed to be helpful for it!

- Penn-fat milestone details posted soon!

- Final exam schedule posted:
  - Tuesday, December 15, 2026: 3:00pm to 5:00pm in JMHH G06

Any Questions?

Lecture Outline

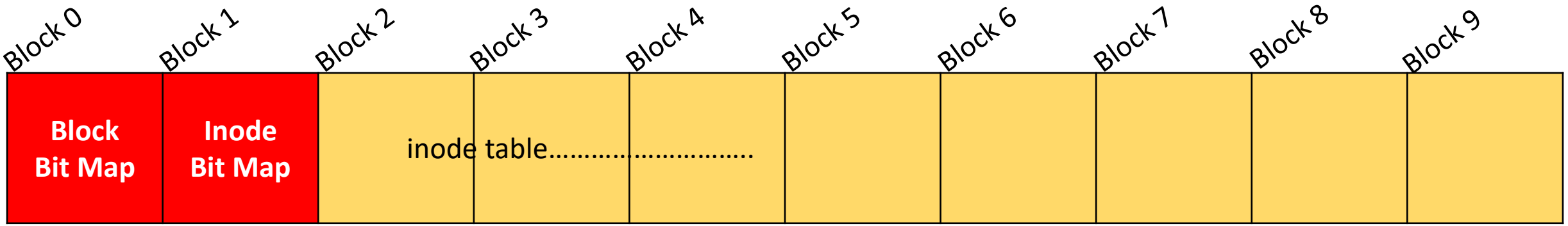
- **File System: Putting It All Together**
  - **Bitmaps**
  - **Super Block**
  - **Boot Block**
- High Level: Scheduling
- Non-Preemptive Algorithms
  - First Come First Serve (FCFS)
  - Shortest Job First

A more realistic rendition of the Linux Filesystem

- When the filesystem is formatted (set up), the entire inode table is created with a set number of inodes.
- Thus, the number of blocks that can hold file data is also limited and can be tracked.
- We need two things:
  - A bit map for the inodes to correspond to allocated and unallocated inodes.
  - A bit map for the blocks to correspond to allocated and unallocated blocks.
  - These bitmaps should take up at least one block each ***prior to the inode table.***

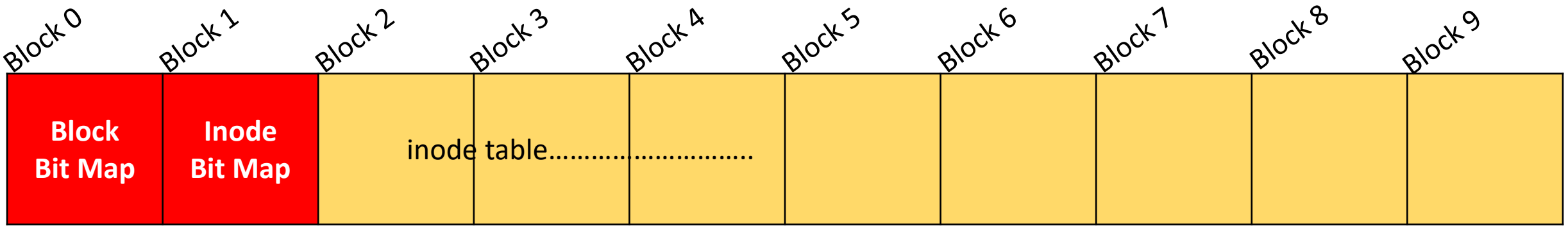
A more realistic rendition of the Linux Filesystem

- We need two things:
  - A bit map for the inodes to correspond to allocated and unallocated inodes.
  - A bit map for the blocks to correspond to allocated and unallocated blocks.
  - These bitmaps take up at least one block each ***prior to the inode table.***



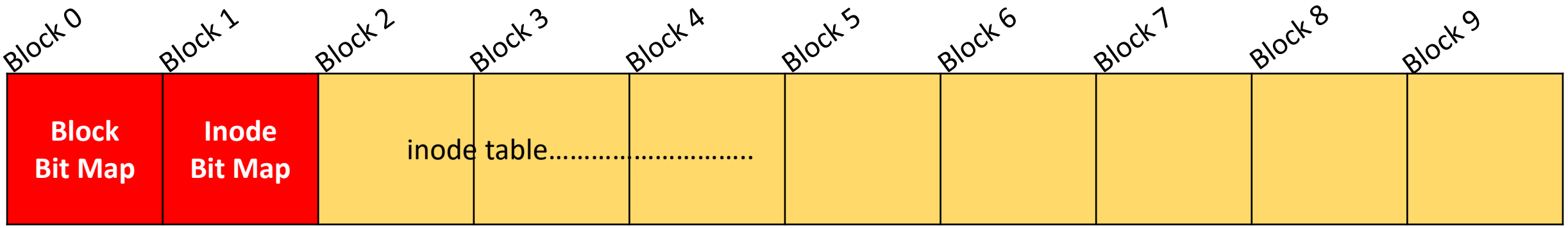
A much more realistic rendition of the Linux Filesystem

- We have the bit maps now, what else do we need?
  - ***We don't have any information about the file system itself. Some questions we need answers to.***
    - How large are the blocks, how many inodes in the inode table, how many free blocks in total, how many blocks are allocated to data, how many blocks are hidden from the user (just for the os usage), what type of file system is this, what is the size of each inode, which is the first nonspecial inode, is the file system in a valid state, and much much more.



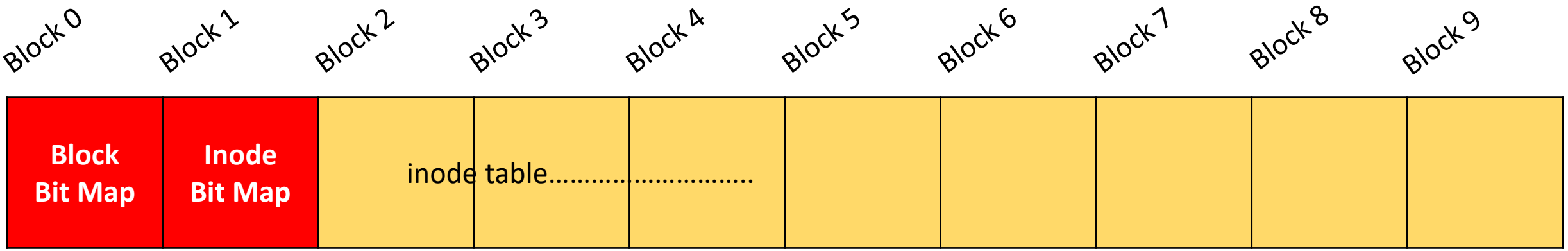
A much more realistic rendition of the Linux Filesystem

- **The Super Block**
  - Is a block in the filesystem that contains metadata ***about*** the file system itself.
  - Used by the operating system to maintain the file system
    - Because it is so important, many copies of the super block are maintained within the file system.
    - Just incase the super block the kernel has becomes corrupted
    - Required overhead, as the superblock is written back to disk frequently.



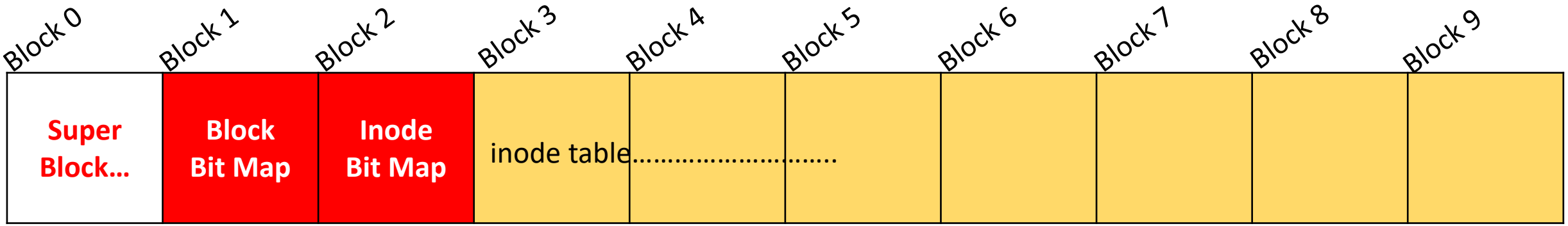
A much more realistic rendition of the Linux Filesystem

- Linux ext2: Super Block Values
  - **inode_count, block_count, reserved_blocks for the kernel**
    - **(why is this important?)**
  - **free_inode_count, free_block_count, first_data_block, block_size,**
  - **filesystem_magic_number, error_no**
  - **first_real_inode** (version 0 set to 11, can be set to any in future versions)
  - **inode_table_start**
  - **There are many many more; 1024 bytes worth of information.**



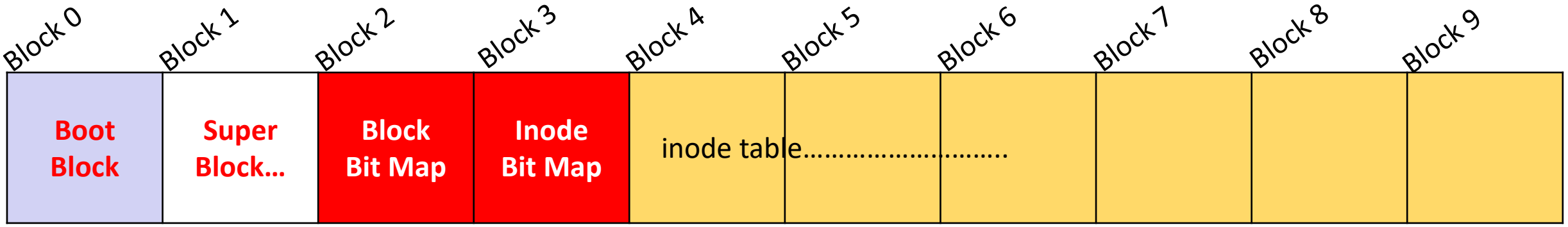
A much more realistic rendition of the Linux Filesystem

- To make room for the Super Block, let's scoot everything over one block.
- ***Can you believe we're still missing one thing?***
  - ***Probably the most important piece…***



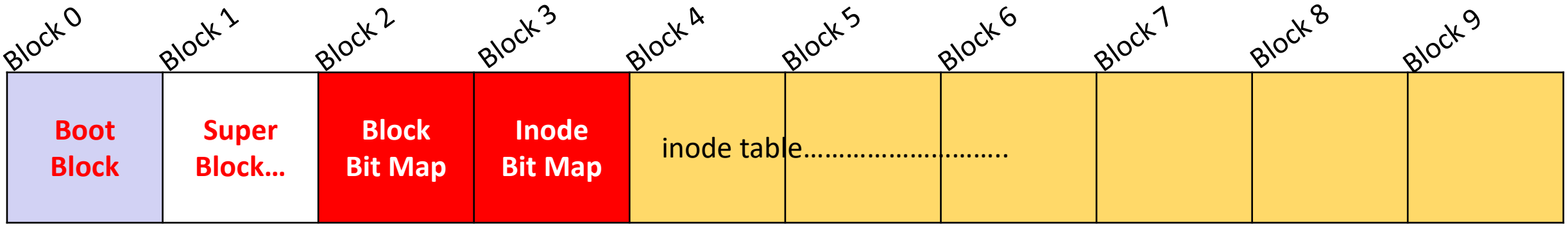
A much more realistic rendition of the Linux Filesystem

- WHERE IS THE OPERATING SYSTEM/KERNAL?
  - It is ***not just going to appear on the computer.***
- ***The code for the operating system is stored within the filesystem itself!***
- It is stored at the start of the file system device and "only" takes up 1024 bytes; so let's scoot everything over.
- Ah, finally, a more realistic design.



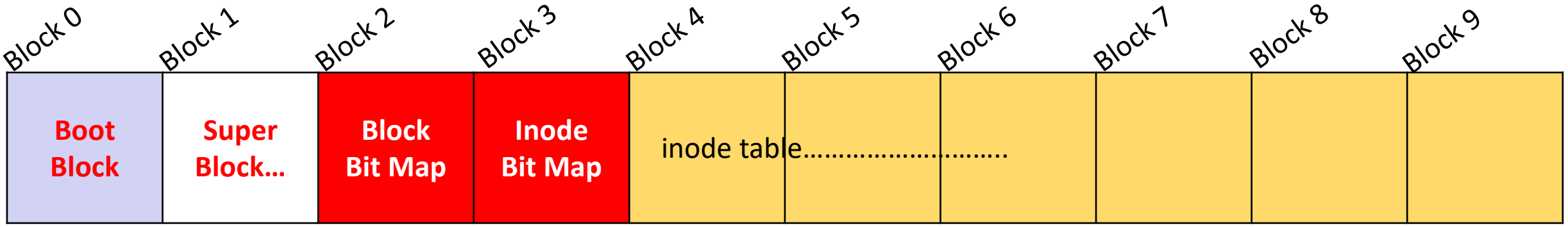
A much more realistic rendition of the Linux Filesystem

- The boot block
  - Contains the initial segments of code necessary to bootstrap the operating system
    - (that is, so that the operating system can ***install itself***)
    - 1024 bytes might seem like a little bit, but really, the boot code is much smaller than this.
      - **512 bytes minimum needed for the boot block, *with 446 bytes dedicated to the actual bootloader itself. Talk about optimization.***
      - ***Why 512? (legacy support).***



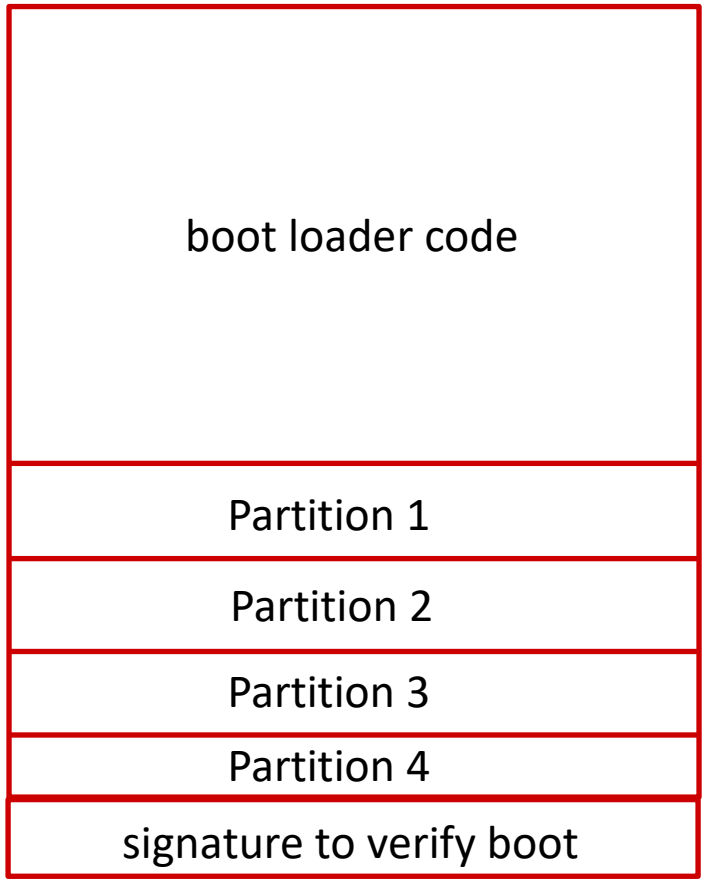
A much more realistic rendition of the Linux Filesystem

- The boot block
  - 512 bytes minimum needed for the boot block, with 446 bytes dedicated to the actual bootloader itself. *Talk about optimization.*
  - The boot block also contains information not about the operating system itself, but also about the physical disk itself. It tells it (the cpu) ***where to find the rest of the OS code***. Because, the linux kernal is not going to fit in 446 bytes, that would be silly.



A closer look at the boot block

- The boot block
  - 512 bytes minimum needed for the boot block, with 446 bytes dedicated to the actual bootloader itself. *Talk about optimization.*



The bootloader/pre-kernal code tells the cpu

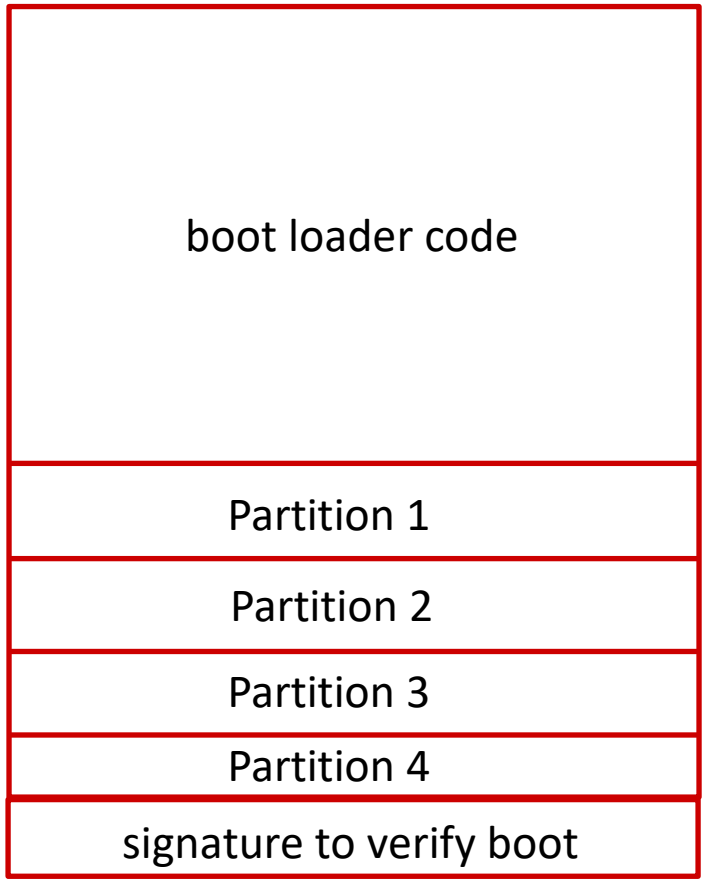
What it should do to prepare itself to install the actual kernel.

1. ***How to set up registers.***
2. ***How to set up memory allocation***
   A. ***(literally, a small stack and text segment, and others)***
3. ***Validate the file system***
4. ***Establish permissions (software execution permission)***
5. ***Examines the CPU to denote which type it is and what other hardware it can use.***
6. ***Tells it where to locate the rest of the kernel is within the filesystem.***

It is an extremely intricate processes that itself could be an entire class.

A closer look at the boot block

- The boot block
  - 512 bytes minimum needed for the boot block, with 446 bytes dedicated to the actual bootloader itself. *Talk about optimization.*

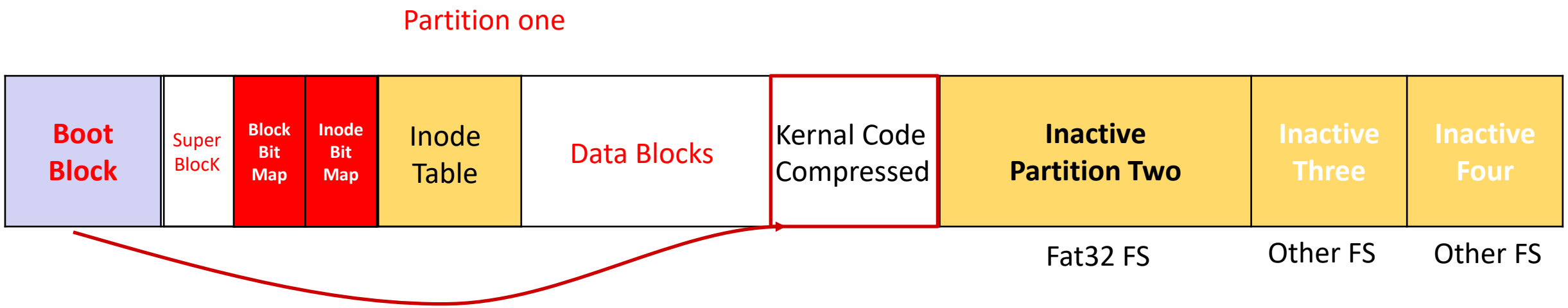


What are these partitions?

They tell the boot loader "mini kernel" which portions of the file system denote which is 'active', that is, which contains the rest of the ***kernel code***. If you have both windows and macos on your machine, then you have two sperate partitions that are active and can be activated by you manually/by the firmware (BIOS).

They can also denote that you have other filesystems on the same disks (hence that it is partitioned).

A real realistic rendition of the Linux Filesystem



You press the power button; what happens?

The hardware searches for attached storage devices (CDs, hard drives, flash drives, etc.) to find a **boot block**.

1. The **boot block** is always at the start of the device or contains a pointer to the actual location of the bootloader or kernel.
2. The **boot block** provides the CPU with its first non-firmware instructions, which are loaded into RAM and executed. This process sets up the system to install the full kernel, which is stored in the file system.
3. After setting up memory, initializing registers, and handling other necessary configurations, the CPU loads the kernel code from the location specified by the bootloader in the active partition.
4. The kernel then decompresses itself and loads into memory (RAM), either all at once or in parts.
5. It loads in the super block for the file system and configures everything (that it needs to).
6. And finally, you are running the operating system for the first time, and it spawns our first processes
   A. **init** on linux, ***launchd*** on mac, ***sessionmanager*** on windows which are ***daemons***
7. Yeah, I know. ☺

And that was file systems! Truly a miracle.

Lecture Outline

- File System: Putting It All Together
  - Bitmaps
  - Super Block
  - Boot Block
- **High Level: Scheduling**
- Non-Preemptive Algorithms
  - First Come First Serve (FCFS)
  - Shortest Job First

Aside: Threads

- Threads will be introduced next week.
- For now, you can assume that threads are the “core of execution” of a process.
  - Or for now you can assume thread == process (tho be ready to unlearn this next week)

OS as the Scheduler

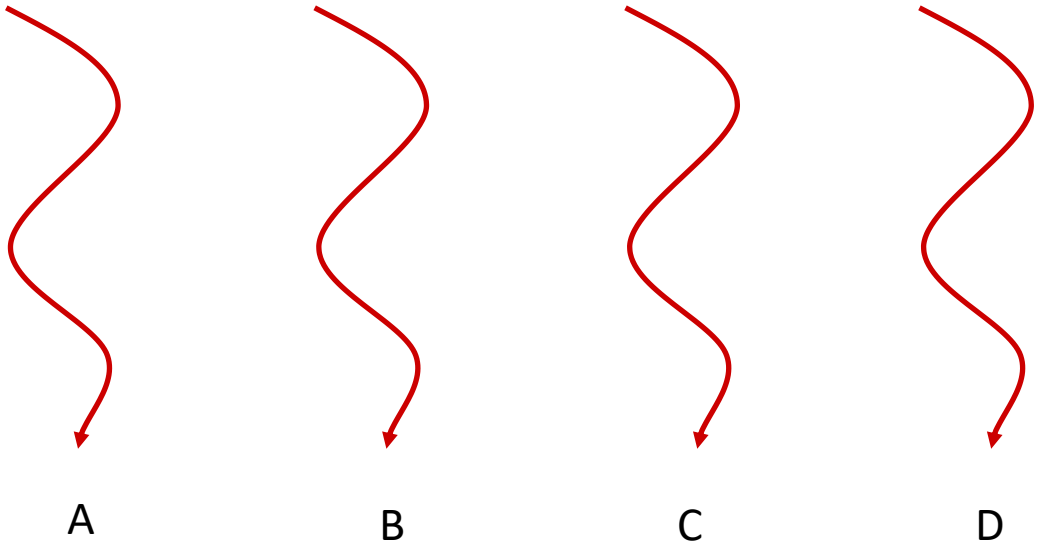
- The scheduler is code that is part of the kernel (OS)

- The scheduler runs when a thread:
  - Starts ("arrives to be scheduled")
  - Finishes
  - Blocks (e.g., waiting on something, usually some form of I/O)
  - Has run for a certain amount of time

- It is responsible for scheduling threads
  - Choosing ***which*** one to run
  - Deciding ***how*** long to run it

Scheduling Threads

- Scheduler: It is responsible for scheduling threads
- From 4 threads, **which do we select to run**?



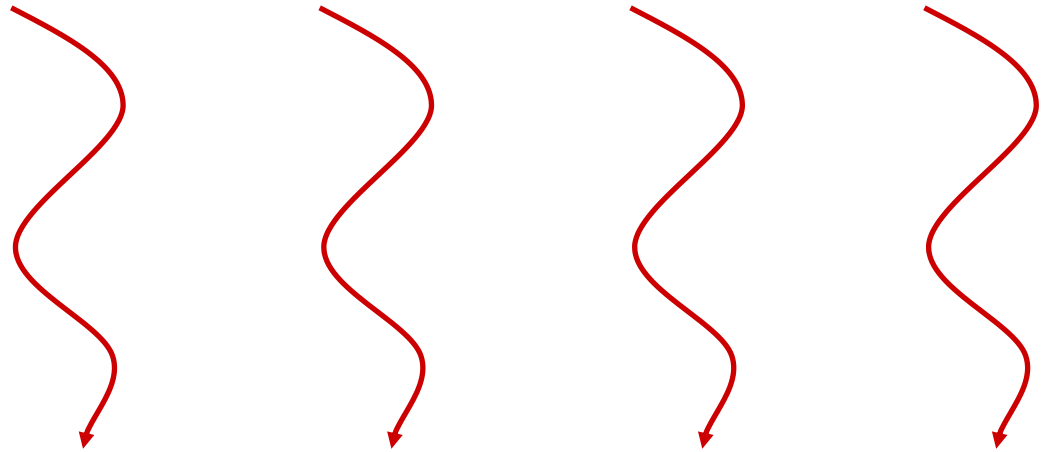
Time Needed at Minimum To Complete Task

A	**5**
B	**5**
C	**5**
D	**5**

Seems like we could just choose any; no preference.

Scheduling Threads

- Scheduler: It is responsible for scheduling threads
- From 4 threads, **which do we select to run**?



A
WIFI daemon

B
Bluetooth daemon

C
Discord

D
Spotify

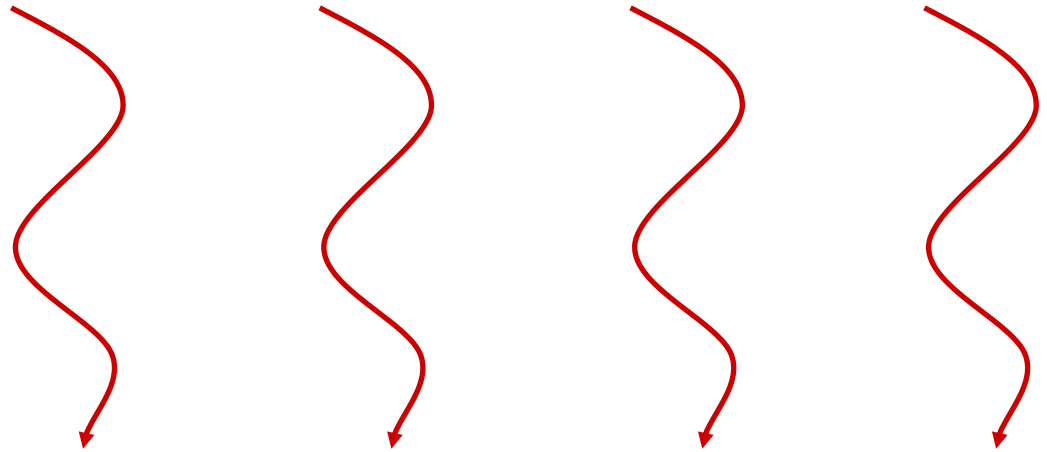
Time Needed at Minimum To Complete Task

A	**5**
B	**5**
C	**5**
D	**5**

Now, the question doesn't seem as straightforward.

Scheduling Threads

- Scheduler: It is responsible for scheduling threads
- From 4 threads, **how long do we select each thread to run**?



A
WIFI daemon

B
Bluetooth daemon

C
Discord

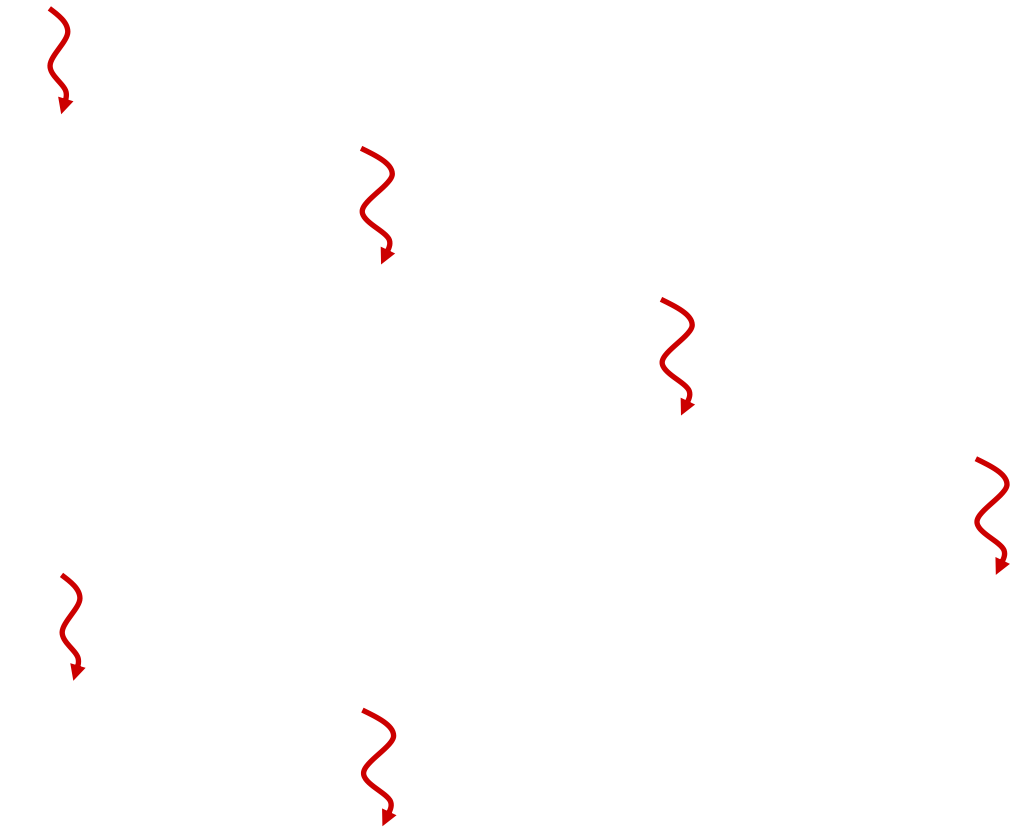
D
Spotify

Time Needed at Minimum To Complete Task

A	**5**
B	**5**
C	**5**
D	**5**

We don't need to run threads until they finish.

Scheduling Threads: A Smaller Unit of Time



- Allows for finer interleaving patterns of threads
- Might result in quicker response times
  - *E.g.* need to handle network responses ASAP so that we don't lose packets

- Much is still missing from the conversation here…

= 1 unit of time on CPU

Scheduler Goals

- Not as straightforward, scheduler needs to balance many things such as:
  - **Minimizing wait time**
    - Starting threads as soon as possible
  - **Minimizing latency**
    - Quick response times and task completions are preferred
  - **Maximizing throughput**
    - Do as much work as possible per unit of time
  - **Maximizing fairness**
    - Make sure every thread can execute fairly

- These goals depend on the system and can conflict with each other!

Scheduler Terminology

- The scheduler utilizes a **scheduling algorithm** to decide what runs next.

- Algorithms are designed to consider many factors:
  - Fairness: Every program gets to run
  - Liveness: That “something” will eventually happen
  - Throughput: Amount of work completed over an interval of time
  - Wait time: Average time a “task” is “alive” but not running
  - Turnaround time: Time between task being ready and completing
  - Response time: Time it takes between task being ready and when it can take user input
  - Etc

Scheduling: Other Considerations

- It takes time to context switch between threads
  - Could get more work done if thread switching is minimized

It takes ~1-2ns to context switch but really depends on machine.

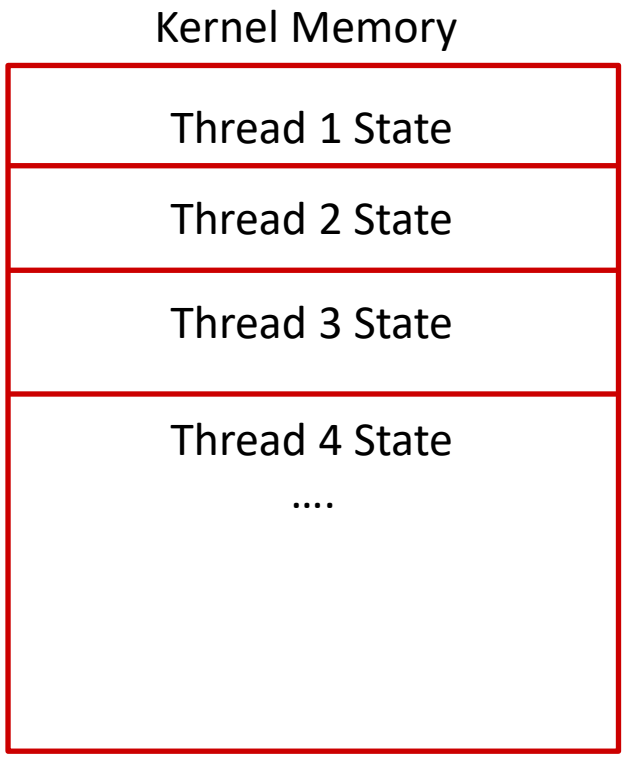
Save Thread A's registers to memory

Load Thread B's registers from memory

These are the General Purpose Registers in RISC-V, but this extends to all architectures

Scheduling: Other Considerations

- Scheduling takes resources
  - It takes time to decide which thread to run next
  - It takes space to hold the required data structures



You don't want to ***waste*** valuable time looping through all threads to find the most optimal – selecting the next thread should be a O(1) operation!

Scheduling: Other Considerations

- It takes time to context switch between threads
  - Could get more work done if thread switching is minimized
- Scheduling takes resources
  - It takes time to decide which thread to run next
  - It takes space to hold the required data structures
- Different tasks have different priorities
  - Higher priority tasks should finish first

Lecture Outline

- File System: Putting It All Together
  - Bitmaps
  - Super Block
  - Boot Block
- High Level: Scheduling
- **Non-Preemptive Algorithms**
  - **First Come First Serve (FCFS)**
  - **Shortest Job First**

Types of Scheduling Algorithms

- **Non-Preemptive:** if a thread is running, it continues to run until it completes or until it gives up the CPU
  - First come first serve (FCFS)
  - Shortest Job First (SJF)

- Also called "Co-Operative" threading because the threads co-operate with each other ☺

First Come First Serve (FCFS)

- **Idea**: Whenever a thread is ready, schedule it to run until it is finished (or blocks).

- Maintain a queue of ready threads
  - Threads go to the back of the queue when it arrives or becomes unblocked
  - The thread at the front of the queue is the next to run

Example of FCFS

- Example workload with three “jobs”

1 CPU
Job 2 arrives slightly after job 1
Job 3 arrives slightly after job 2

Time to Finish

Job 1	**24**
Job 2	**3**
Job 3	**3**

- FCFS schedule:

Example of FCFS

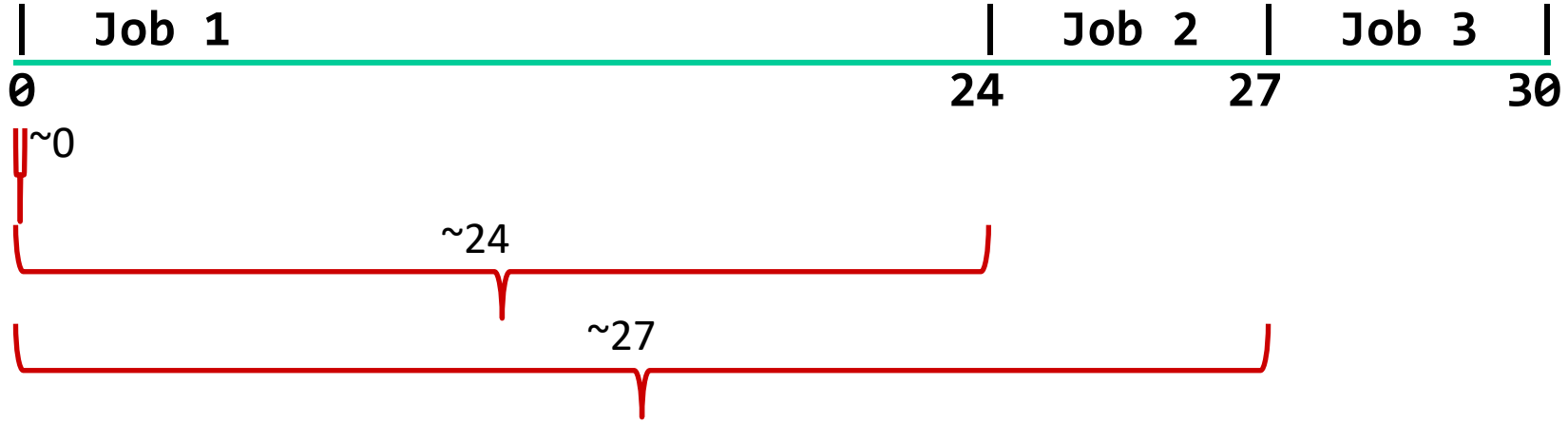
- Wait Time: from ready to running

1 CPU
Job 2 arrives slightly after job 1
Job 3 arrives slightly after job 2

Time to Finish

Job 1	**24**
Job 2	**3**
Job 3	**3**

- FCFS schedule:



- Total wait time: 0 + 24 + 27 = 51
- Average wait time: 51/3 = 17

Example of FCFS

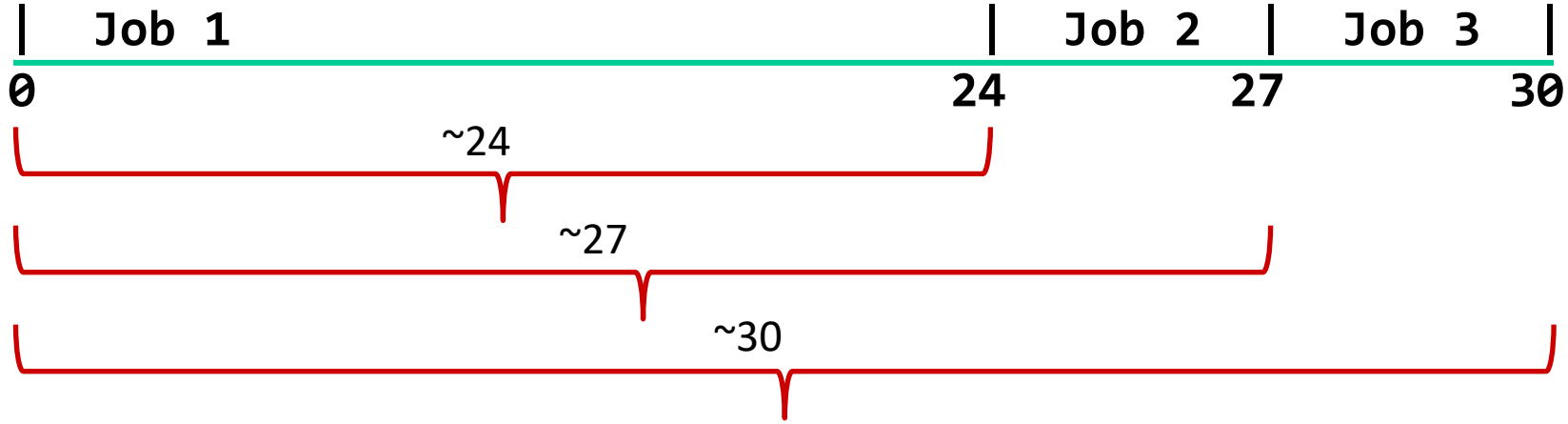
- Turnaround Time: from arrival to finished

1 CPU
Job 2 arrives slightly after job 1
Job 3 arrives slightly after job 2

Time to Finish

Job 1	**24**
Job 2	**3**
Job 3	**3**

- FCFS schedule:



- Total turnaround time: 24 + 27 + 30 = 81
- Average turnaround time: 81/3 = 27

Poll Everywhere

Discuss

- What are the advantages/disadvantages/concerns with **First Come First Serve**

- Things a scheduler should prioritize:
  - Minimizing wait time
  - Minimizing latency
  - Maximizing fairness
  - Maximizing throughput
  - Task priority
  - Cost to schedule things
  - Cost to context switch

- Imagine we have 1 core, and tasks of various (finite) lengths

FCFS Analysis

- Advantages:
  - Simple, low overhead
  - Hard to mess up the implementation
  - Each thread will *definitely* run eventually

- Disadvantages
  - Doesn't work well for interactive systems
  - Throughput can be low due to long threads
  - Large fluctuations in average turn around time
  - Priority not taken into considerations

Shortest Job First (SJF)

- **Idea**: variation on FCFS, but have the tasks with the smallest CPU-time requirement run first
  - Arriving jobs are instead put into the queue depending on their run time, shorter jobs being towards the front
  - Scheduler selects the shortest job (1st in queue) and runs till completion

Example of SJF

- Example workload with three "jobs"

1 CPU
Job 1, 2, & 3 arrive at the same time

	Time to Finish
Job 1	**24**
Job 2	**3**
Job 3	**3**

- SJF schedule:

- Total wait time: 0 + 3 + 6 = 9
- Average wait time: 9/3 = 3
- Total turnaround time: 3 + 6 + 30 = 39
- Average turnaround time: 39/3 = 13

Poll Everywhere

Discuss

- What are the advantages/disadvantages/concerns with **Shortest Job First**

- Things a scheduler should prioritize:
  - Minimizing wait time
  - Minimizing latency
  - Maximizing fairness
  - Maximizing throughput
  - Task priority
  - Cost to schedule things
  - Cost to context switch

- Imagine we have 1 core, and tasks of various (finite) lengths

SJF Analysis

- Advantages:
  - Still relatively simple, low overhead
  - perhaps minimal average turnaround time

- Disadvantages
  - Starvation possible
    - If quick jobs keep arriving, long jobs will keep being pushed back and won't execute
  - How do you know how long it takes for something to run?
    - You can't! You can use a history of past behavior to make a guess.
  - Priority not taken into considerations