Scheduling & File System Intro

Computer Operating Systems, Fall 2023

Instructor: Travis McGaha

Head TAs: Nate Hoaglund & Seungmin Han

TAs:

Andy Jiang	Haoyun Qin	Kevin Bernat	Ryoma Harris
Audrey Yang	Jason hom	Leon Hertzberg	Shyam Mehta
August Fu	Jeff Yang	Maxi Liu	Tina Kokoshvili
Daniel Da	Jerry Wang	Ria Sharma	Zhiyan Lu
Ernest Ng	Jinghao Zhang	Rohan Verma	

Administrivia

- Mid Semester Feedback Survey Releases Tonight!
  - Is anonymous, but as a result is on canvas ☹
  - Lots of questions and opportunities to give long answers.
    Your answers will help us shape the course for future semesters
  - As long as you submit you should get the credit
  - Worth about 1 check-in, no penalty for not doing it. Can think of it as a “make-up” check-in?

- PennOS specification released! (We are modifying the due dates)
  - Milestone 0 due in ~1 week
    - It is just making sure you have a group, read the specification, understand ucontext, and have a rough plan :P
  - Milestone 1 is due in ~2-ish weeks
  - Whole thing due in (~1 month)

Administrivia

- Recitation After lecture will be preparing you for PennOS and refreshing you on Makefiles

- Lecture today will be scheduling and a brief intro to File Systems. The goal is to introduce you enough to have a vague understanding of what to do for PennOS.

- Lecture on Thursday will be entirely a PennOS TA demonstration presentation.
  - Please read the specification and review this lecture, come to class with questions

Administrivia

- GROUPS MUST BE MADE BY THE END OF **SATURDAY**
  - Same canvas group sign-up steps as before. Should be an ed post about it.
  - Submit to gradescope to make a repository
  - remaining people will be randomly assigned partners
- MILESTONE 0 IS DUE Friday 11/3 @ MIDNIGHT

Poll Everywhere

pollev.com/tqm

- Any questions, comments or concerns from last lecture?

Lecture Outline

- **Scheduling**
  - **FCFS**
  - **SJF**
  - **RR**
  - **RR Variants**
- Intro to File System
- Disk Allocation
  - Contiguous
  - Linked List
  - FAT

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

Scheduler Terminology

- The scheduler has 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...

Goals

- The scheduler will have various things to prioritize
- Some examples:
- Minimizing wait time
  - Get threads started 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

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

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)

- **Preemptive:** the thread may be interrupted after a given time and/or if another thread becomes ready
  - Round Robin
  - Priority Round Robin
  - …

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

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

- Example workload with three “jobs”:
  Job 1: 24 time units; Job 2: 3 units; Job 3: 3 units

- FCFS schedule:

```
|   Job 1                    |   Job 2   |   Job 3   |
 0                          24          27          30
```

- Total waiting time: 0 + 24 + 27 = 51
- Average waiting time: 51/3 = 17
- Total turnaround time: 24 + 27 + 30 = 81
- Average turnaround time: 81/3 = 27

Poll Everywhere

pollev.com/tqm

- 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 lengths…

FCFS Analysis

- Advantages:
  - Simple, low overhead
  - Hard to screw up the implementation
  - Each thread will DEFINITELY get to 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

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

- Same example workload with three “jobs”:

  Job 1: 24 time units; Job 2: 3 units; Job 3: 3 units

- FCFS schedule:

```
| Job 2 | Job 3  |    Job 1                        |
0       3        6                                 30
```

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

Poll Everywhere

pollev.com/tqm

- 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 lengths…

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)

- **Preemptive:** the thread may be interrupted after a given time and/or if another thread becomes ready
  - Round Robin
  - Priority Round Robin
  - …

Round Robin

- Sort of a preemptive version of FCFS
  - Whenever a thread is ready, add it to the end of the queue.
  - Run whatever job is at the front of the queue

- BUT only led it run for a fixed amount of time (quantum).
  - If it finishes before the time is up, schedule another thread to run
  - If time is up, then send the running thread back to the end of the queue.

Example of Round Robin

- Same example workload:
  Job 1: 24 units, Job 2: 3 units, Job 3: 3 units
- RR schedule with time quantum=2:

```
|Job 1|Job 2|Job 3|Job 1|Jo2|Jo3|Job 1| …     |Job 1|
0     2     4     6     8   9   10    12,14…        30
```

- Total waiting time: (0 + 4 + 2) + (2 + 4) + (4 + 3) = 19
  - Counting time spent waiting between each “turn” a job has with the CPU
- Average waiting time: 19/3 (~6.33)
- Total turnaround time: 30 + 9 + 10 = 49
- Average turnaround time: 49/3 (~16.33)

Poll Everywhere

pollev.com/tqm

- What are the advantages/disadvantages/concerns with **Round Robin**

- 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 lengths…

Round Robin Analysis

- Advantages:
  - Still relatively simple
  - Can works for interactive systems

- Disadvantages
  - If quantum is too small, can spend a lot of time context switching
  - If quantum is too large, approaches FCFS
  - Still assumes all processes have the same priority.

- Rule of thumb:
  - Choose a unit of time so that most jobs (80-90%) finish in one usage of CPU time

RR Variant: PennOS Scheduler

- In PennOS you will have to implement a priority scheduler based mostly off of round robin.

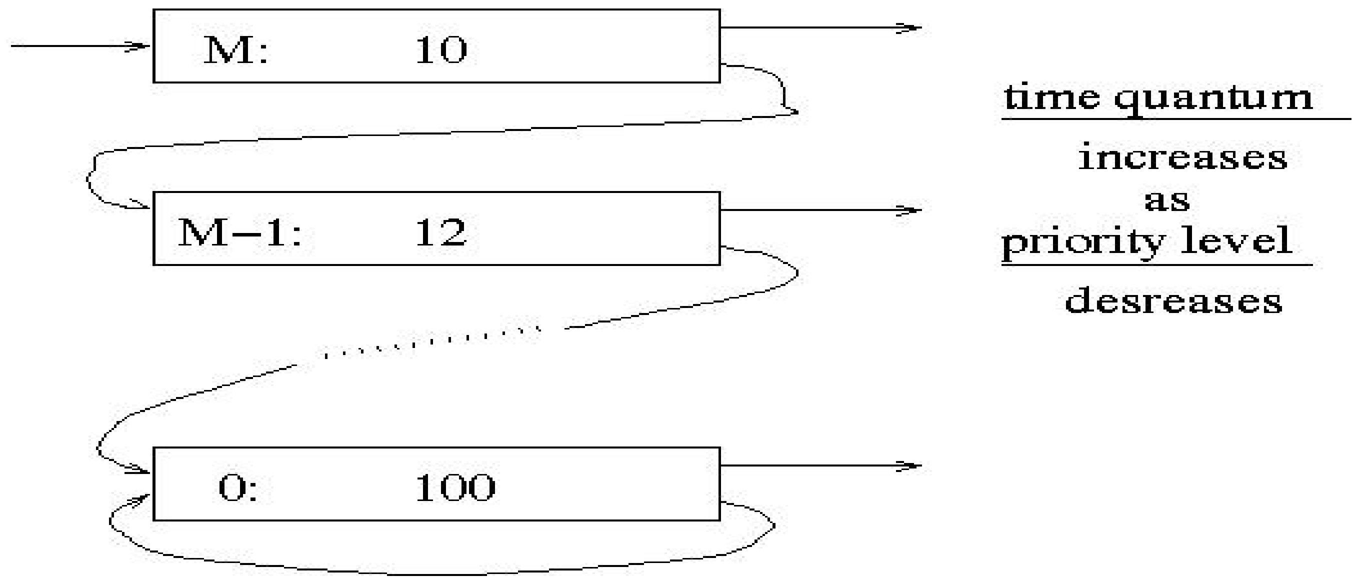
- You will have 3 queues, each with a different priority (-1, 0, 1)
  - Each queue acts like normal round robin within the queue

- You spend time quantum processing each queue proportional to the priority
  - Priority -1 is scheduled 1.5 times more often than priority 0
  - Priority 0 is scheduled 1.5 times more often than priority 1

RR Variant: Priority Round Robin

- Same idea as round robin, but with multiple queues for different priority levels.
- Scheduler chooses the first item in the highest priority queue to run
- Scheduler only schedules items in lower priorities if all queues with higher priority are empty.

RR Variant: Multi Level Feedback



- Each priority level has a ready queue, and a time quantum
- Thread enters highest priority queue initially, and lower queue with each timer interrupt
- If a thread voluntarily stops using CPU before time is up, it is moved to the end of the current queue
- Bottom queue is standard Round Robin
- Thread in a given queue not scheduled until all higher queues are empty

Multi Level Feedback Analysis

- Threads with high I/O bursts are preferred
  - Makes higher utilization of the I/O devices
  - Good for interactive programs (keyboard, terminal, mouse is I/O)
- Threads that need the CPU a lot will sink to lower priority, giving shorter threads a chance to run
- Still have to be careful in choosing time quantum
- Also have to be careful in choosing how many layers

Multi Level Feedback Variants: Priority

- Can assign tasks different priority levels upon initiation that decide which queue it starts in
  - E.g. the scheduler should have higher priority than HelloWorld.java
- Update the priority based on recent CPU usage rather than overall cpu usage of a task
  - Makes sure that priority is consistent with recent behavior
- Many others that vary from system to system

Why did we talk about this?

- Scheduling is fundamental towards how computer can multi-task

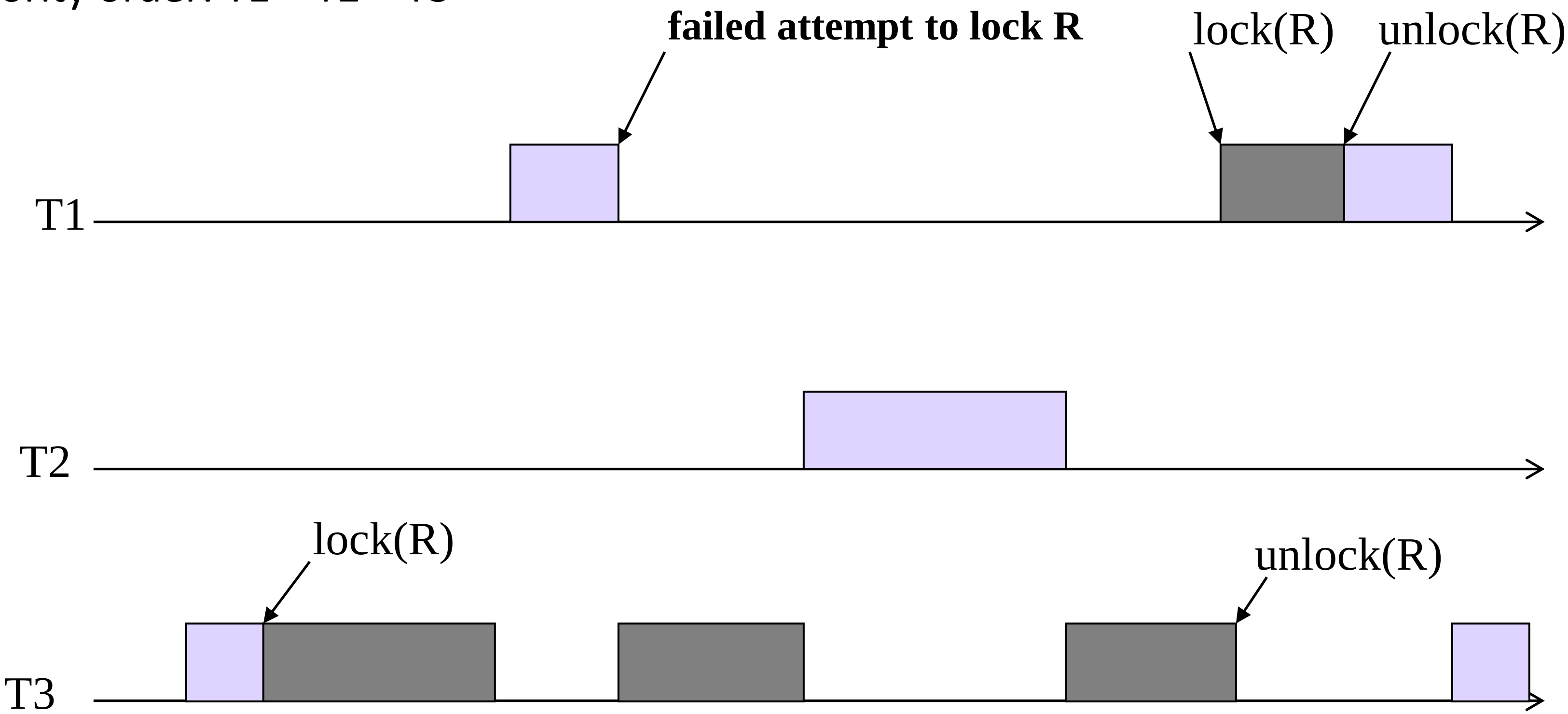
- This is a great example of how "systems" intersects with algorithms :)

- It shows up occasionally in the real world :)
  - Scheduling threads with priority with shared resources can cause a priority inversion, potentially causing serious errors.

What really happened on Mars Rover Pathfinder, Mike Jones.
http://www.cs.cornell.edu/courses/cs614/1999sp/papers/pathfinder.html

The Priority Inversion Problem

Priority order: T1 > T2 > T3



T2 is causing a higher priority task T1 wait !

More

- For those curious, there was a LOT left out

- RTOS (Real Time Operating Systems)
  - For real time applications
  - CRITICAL that data and events meet defined time constraints
  - Different focus in scheduling. Throughput is de-prioritized

- Fair-share scheduling
  - Equal distribution across different users instead of by processes

- Etc.

Lecture Outline

- Scheduling
  - FCFS
  - SJF
  - RR
  - RR Variants
- **Intro to File System**
- Disk Allocation
  - Contiguous
  - Linked List
  - FAT

File System: User Level STD API

- C stdio API: core functionalities
  - ```
    FILE* fopen(char *pathname, char *mode);
    ```
  - ```
    size_t fread(void *ptr, size_t size,
                 size_t nmemb, FILE* stream);
    ```
  - ```
    size_t fwrite(void *ptr, size_t size,
                  size_t nmemb, FILE* stream);
    ```
  - ```
    int fclose(FILE *stream);
    ```

- These core functionality of these functions should be self-explanatory. If you need to use these, use man pages to lookup the exact details

File System: User Level STD API again

- C stdio API: core functionalities
  - ```
    FILE* fopen(char *pathname, char *mode);
    ```
  - ```
    size_t fread(void *ptr, size_t size,
                 size_t nmemb, FILE* stream);
    ```
  - ```
    size_t fwrite(void *ptr, size_t size,
                  size_t nmemb, FILE* stream);
    ```
  - ```
    int fclose(FILE *stream);
    ```
- In addition to the above, we also have another common feature: moving to an arbitrary position in the file

```
int fseek(FILE *stream, long offset, int whence);
```

User Perspective: A stream of bytes

- As a user, we have the idea of a file as being a "stream" of bytes.
  - a continuous sequence of data made available over time.
  - There are many kinds of streams, for now we are talking about files

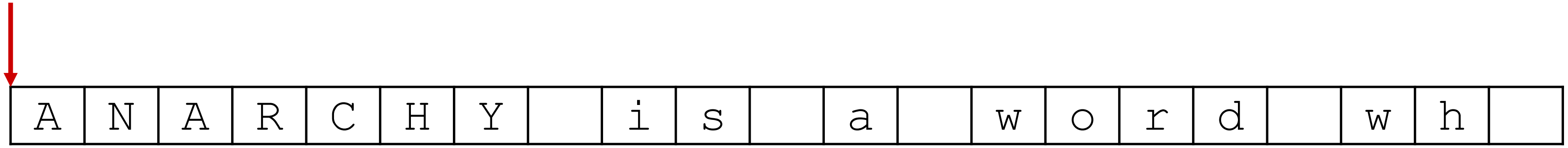
- From our perspective, a **file** stream looks like this:
  - A sequence of characters that come one after the other

A	N	A	R	C	H	Y		i	s		a		w	o	r	d		w	h	

User Perspective: A stream of bytes

- As a user, we have the idea of a file as being a "stream" of bytes.
  - a continuous sequence of data made available over time.
  - There are many kinds of streams, for now we are talking about files

- From our perspective, a **file** stream looks like this:
  - A sequence of characters that come one after the other
  - When we open a file, we start at the beginning of the file stream



User Perspective: A stream of bytes

- As a user, we have the idea of a file as being a "stream" of bytes.
  - a continuous sequence of data made available over time.
  - There are many kinds of streams, for now we are talking about files

- From our perspective, a **file** stream looks like this:
  - A sequence of characters that come one after the other
  - When we open a file, we start at the beginning of the file stream
  - As we read chars, we "move forward" to the next chars in the file

A	N	A	R	C	H	Y		i	s		a		w	o	r	d		w	h	

User Perspective: A stream of bytes

- As a user, we have the idea of a file as being a "stream" of bytes.
  - a continuous sequence of data made available over time.
  - There are many kinds of streams, for now we are talking about files

- From our perspective, a **file** stream looks like this:
  - A sequence of characters that come one after the other
  - When we open a file, we start at the beginning of the file stream
  - As we read chars, we "move forward" to the next chars in the file

- **This is not just a C thing**; this is probably what you have done in Java and other languages.

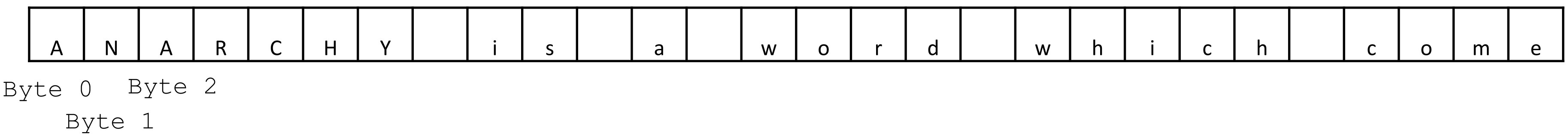
Operating System Perspective: Blocks

- The stream model is very convenient for user level programs to access files. How data is stored on disk is more complicated
  - File data is not necessarily contiguous in hardware.

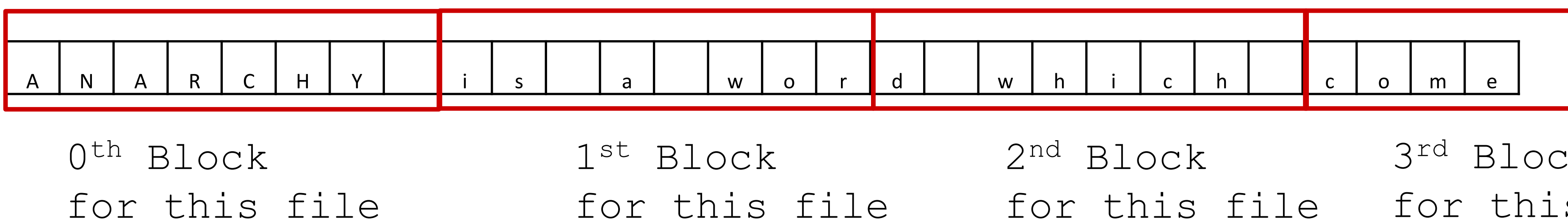
- Files can be broken up into units called **blocks**
  - Blocks are a fixed-size of contiguous bytes in the disk.
  - When the operating system interfaces with hardware, it works in terms of blocks.
  - When the OS operates on a file, it reads/writes an entire block at a time

Operating System Perspective: Blocks

- User perspective: A sequence of bytes



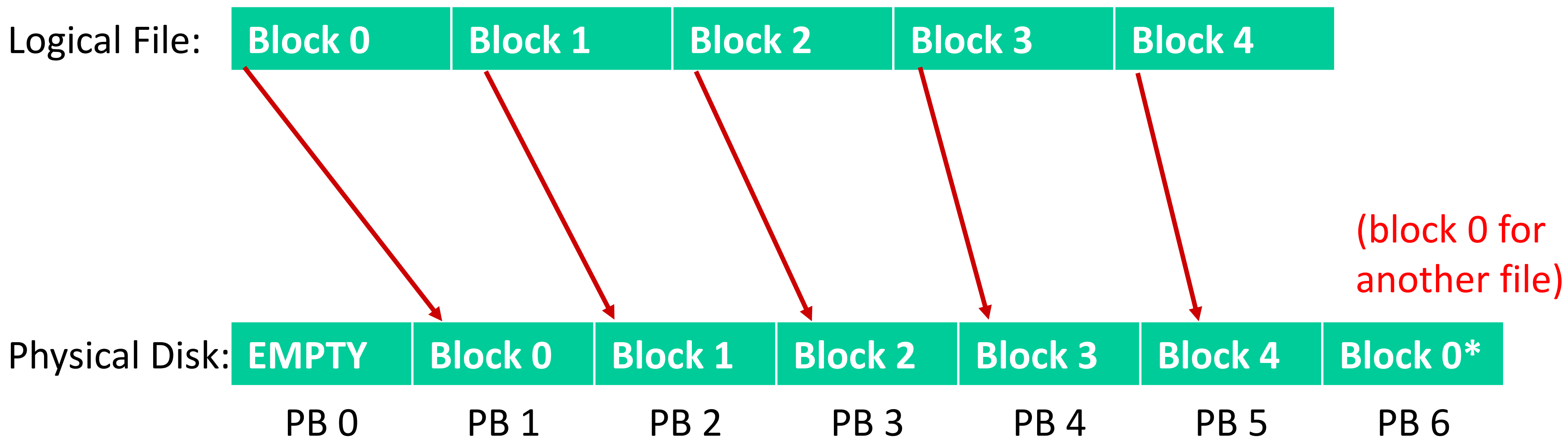
- More details: these bytes are broken up into a series of logical blocks



Block numbers are not 0, 1, 2 3. More on this in a few slides

High Level: View

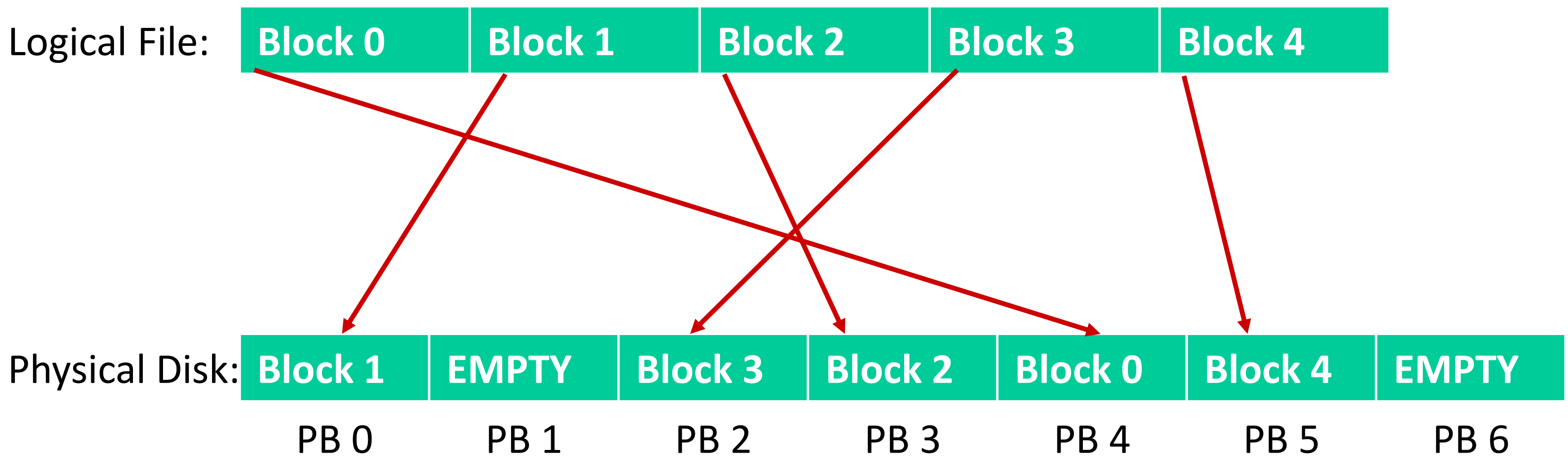
- A file is a sequence of bytes that can be split into blocks



- Disk can be thought of as an array of **physical blocks** that contain file blocks, metadata or are empty
- The file system allocates blocks to files and translates from a “logical” block number to a physical block number

High Level: View

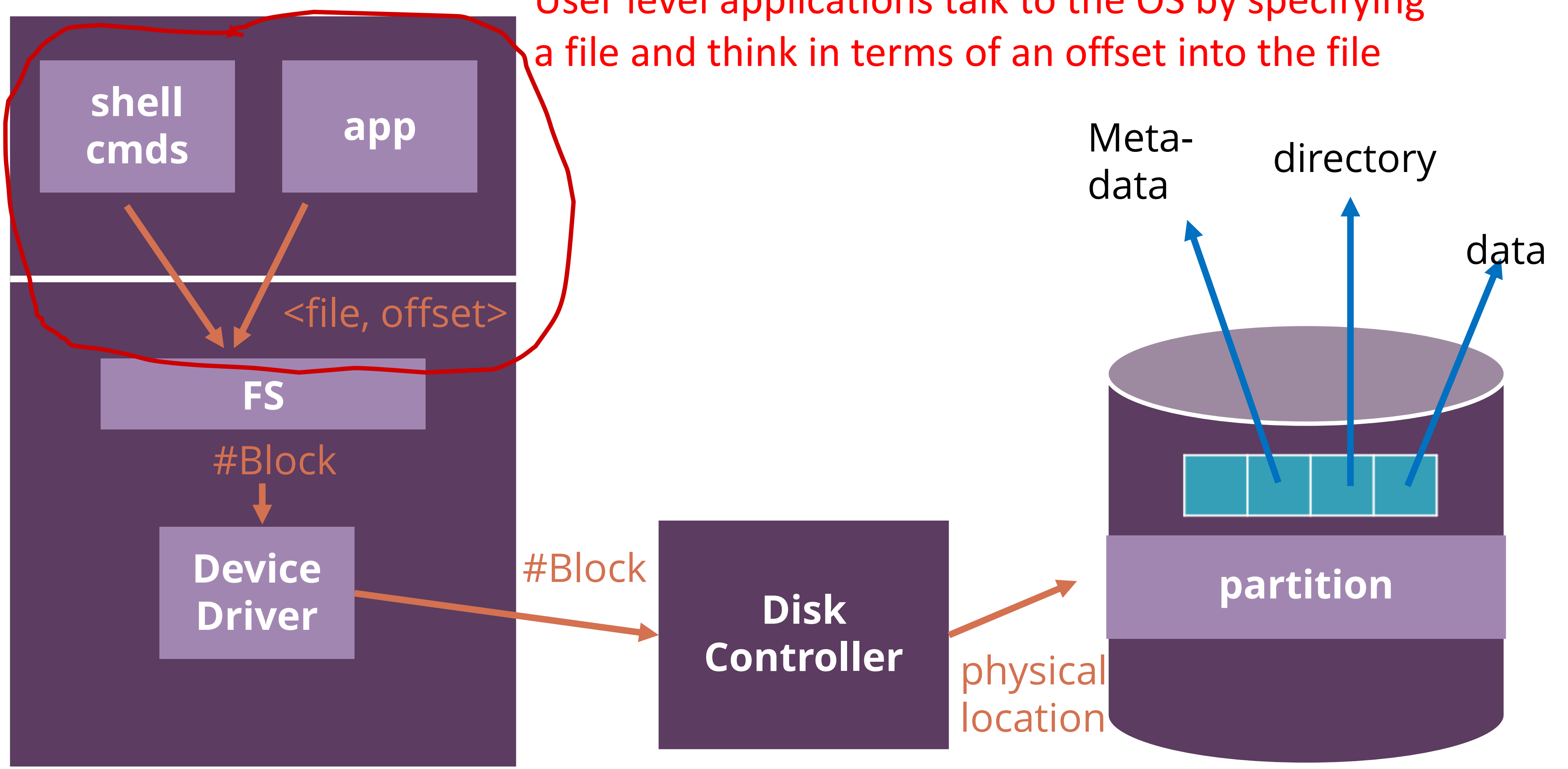
- A file is a sequence of bytes that can be split into blocks



- Disk can be thought of as an array of **physical blocks** that contain file blocks, metadata or are empty
- Note: blocks that are logically next to each other in a file may not be contiguous in hardware

Operating System Perspective: Hardware

User level applications talk to the OS by specifying a file and think in terms of an offset into the file

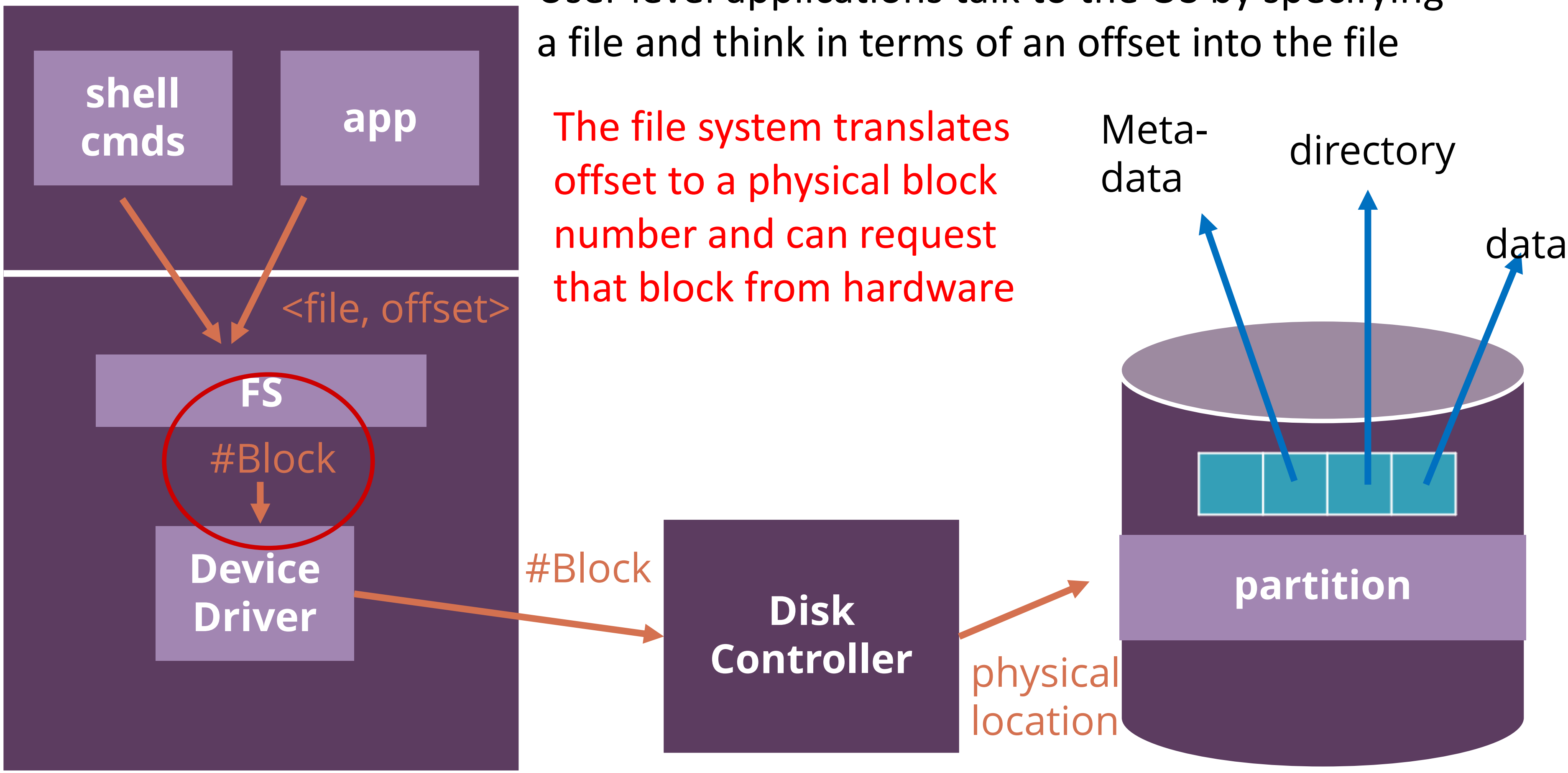


QUICK PERSPECTIVE TO GET YOU READY FOR PENNOS, SLIGHLY MORE DETAIL ANOTHER LECTURE

Operating System Perspective: Hardware

User level applications talk to the OS by specifying a file and think in terms of an offset into the file

The file system translates offset to a physical block number and can request that block from hardware



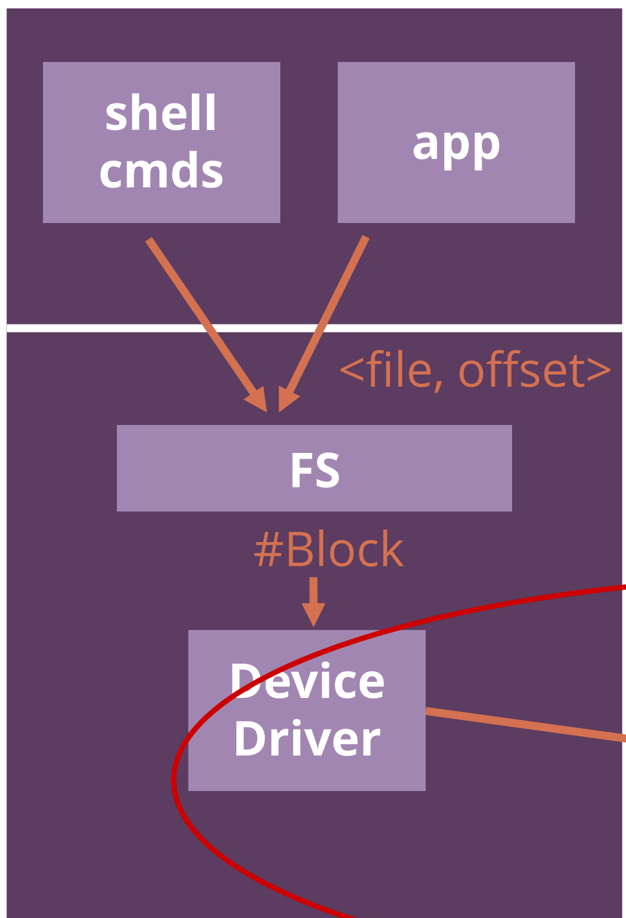
QUICK PERSPECTIVE TO GET YOU READY FOR PENNOS, SLIGHLY MORE DETAIL ANOTHER LECTURE

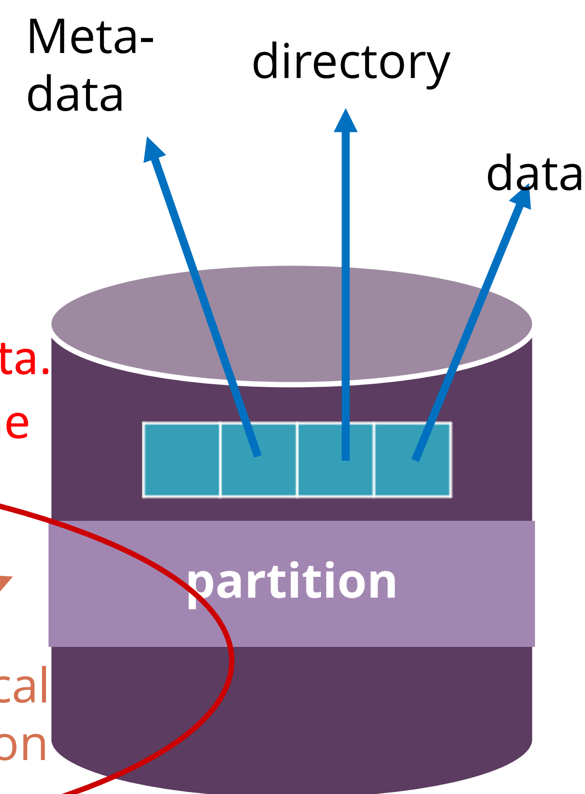
Operating System Perspective: Hardware

User level applications talk to the OS by specifying a file and think in terms of an offset into the file

The file system translates offset to a physical block number and can request that block from hardware

Hardware will find the physical location and read/write the data. Interrupt the OS when it is done





<u>**QUICK PERSPECTIVE TO GET YOU READY FOR PENNOS, SLIGHLY MORE DETAIL ANOTHER LECTURE**</u>

Block Size

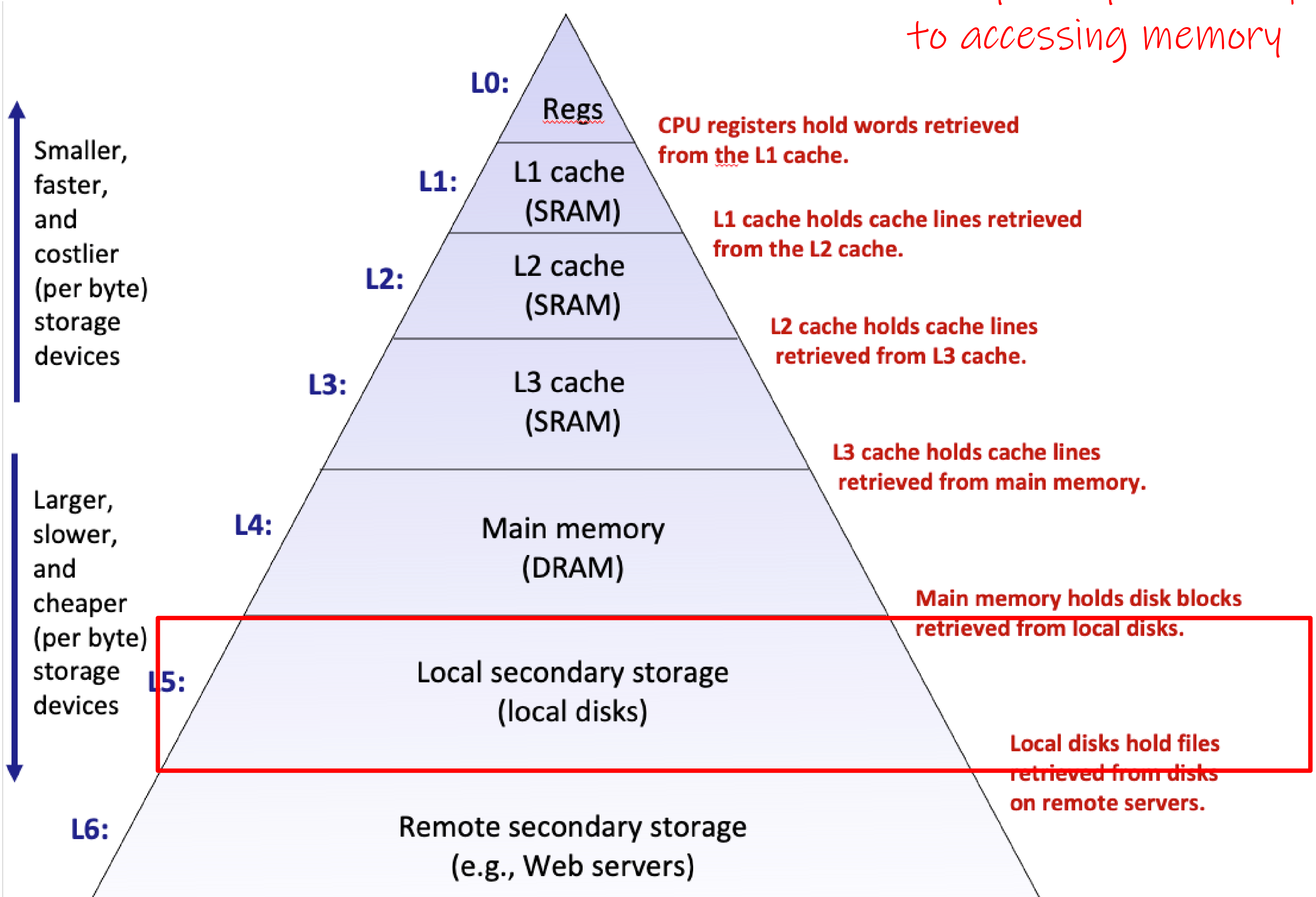
- The Block size is dependent on hardware and some file systems can have multiple different block sizes

- Typically, 2Kib – 16KiB in size

- Tradeoff in size:
  - If blocks are small, then we need to go to disk more often
  - If block sizes are too big, then we may have more internal fragmentation. (e.g., a file containing the letters “hi” will take up more space)

Memory Hierarchy (again)

Files systems are really really really slow compared to accessing memory



Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Seek Time

- To seek in a file is to move to a different position in the file. If we want to move from one place on the hardware to another, that takes a VERY long time (relatively)

- HDD (Hard Disk Drives) consist of a spinning disk and an arm that hovers over the disk to read data

- Video: https://yewtu.be/watch?v=p-JJp-oLx58
  - Start at 6:48 ish
- Since this is a physical operation, much slower (relatively) than electronic operations

HDD vs SSD

- SSD’s (Solid State Drives) are another piece of hardware that is gaining a lot of popularity

- Compare to HDDs
  - Much faster read & seek time
  - Lower energy requirements
  - Smaller
  - Etc.

HDD’s are still ~80-90% of what data centers use to store data

Personal & Mobile devices use SSD, they are really nice ☺

- Downsides:
  - HDD’s are still cheaper per bit than SSD
  - SDD’s degrade quicker on reads & writes.

Lecture Outline

- Scheduling
  - FCFS
  - SJF
  - RR
  - RR Variants
- Intro to File System
- **Disk Allocation**
  - **Contiguous**
  - **Linked List**
  - **FAT**

Disk vs Memory Allocation

- Disk and memory allocation looks very similar

- Big difference:
  - Disk access speed is different than memory accesses.
    Memory has quick random access, while disk needs to seek to the correct position first
  - Access pattern for Disk can be different than Memory

- Same goals:
  - Fast sequential & Random Access
  - Minimize fragmentation
  - Be able to extend files when needed

Contiguous Allocation

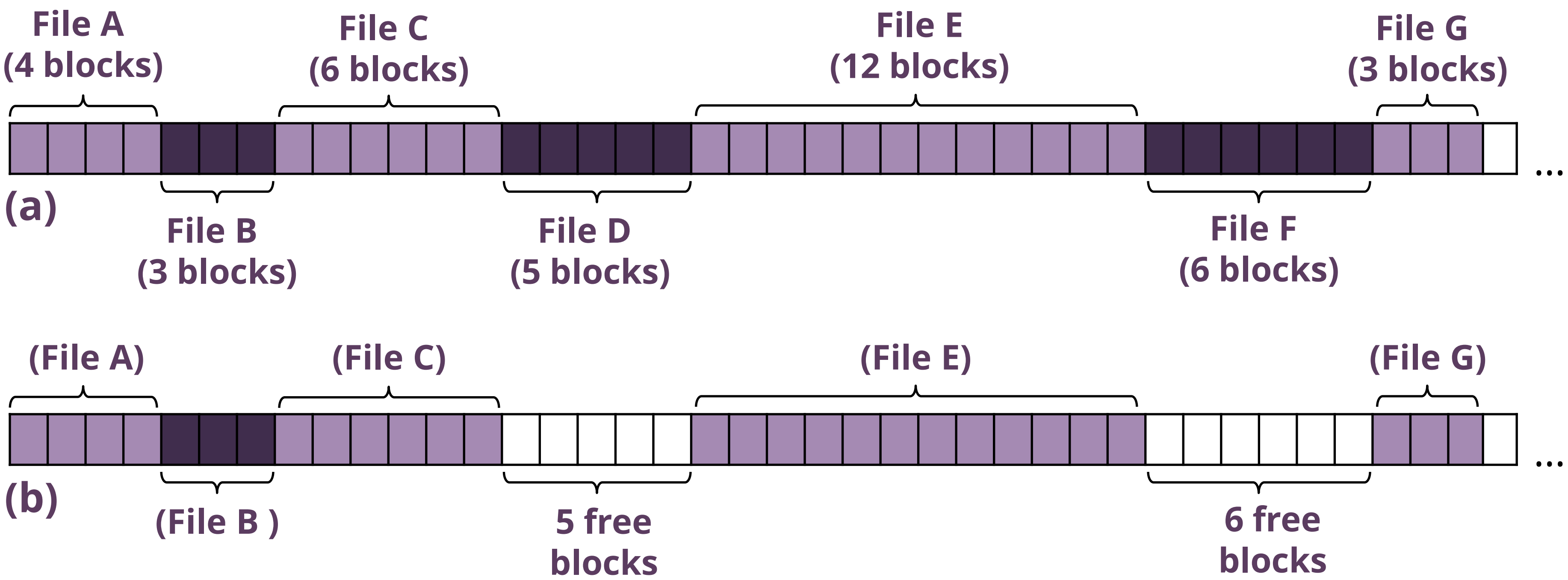
- Each file occupies a contiguous region of blocks
  - Fast random access (only one seek to the beginning needed)
- Useful when read-only devices or small devices
  - CD-ROMs, DVD-ROMs and other optical media
  - Embedded/personal devices
- Management is easy but inflexible
  - Directory entry of a file needs to specify its size and start location

Contiguous Allocation

- Fragmentation is a problem if deletes are allowed, or if files grow to need more space

- After disk is full, new files need to fit into any “holes”
  - Requires advanced declaration of size at the time of creation

Contiguous Allocation



(a) Contiguous allocation of disk space for 7 files

(b) The state of the disk after files D and F have been removed

Fragmentation after the deletes. If I wanted to allocate a file of size 8, I couldn't without rearranging file allocations

Contiguous Allocation Analysis

- Pros
  - Quick and simple ☺

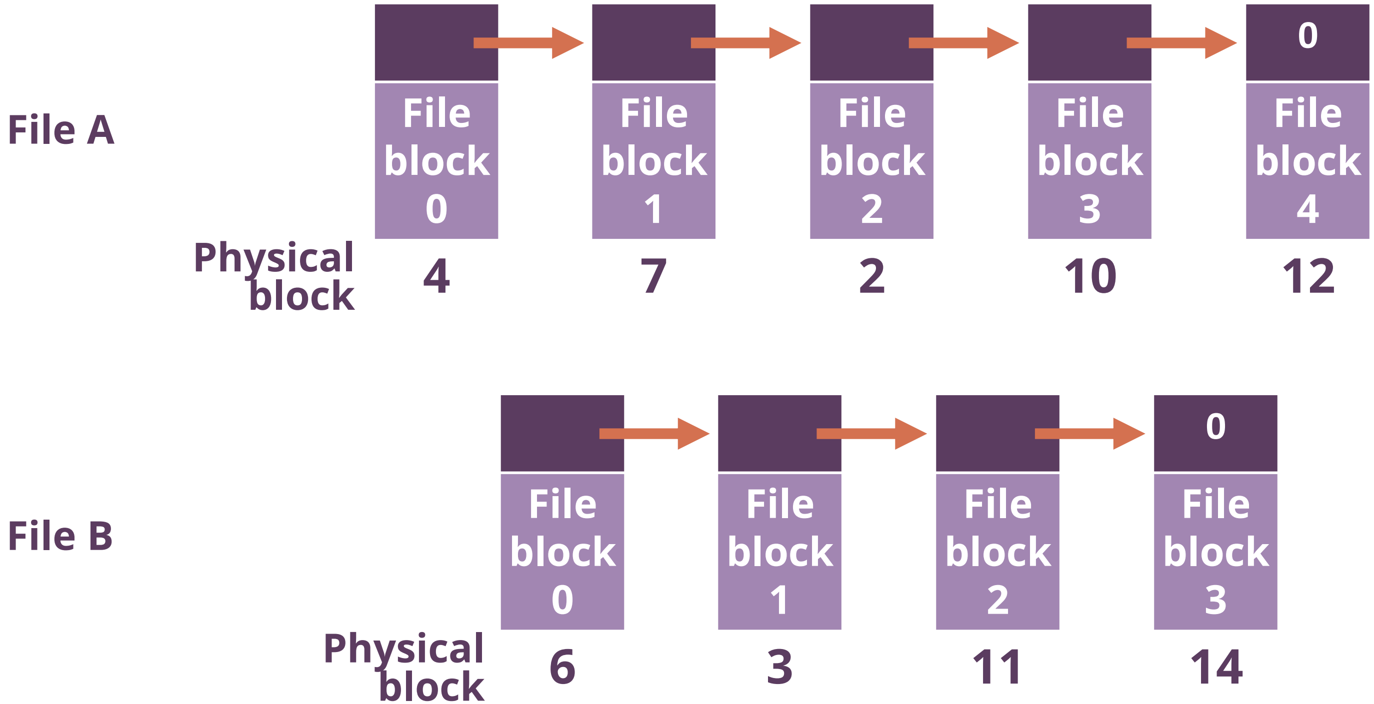
- Cons
  - Issues with fragmentation when we delete files
  - Can’t extend the size of files easily ☹

Linked List Allocation

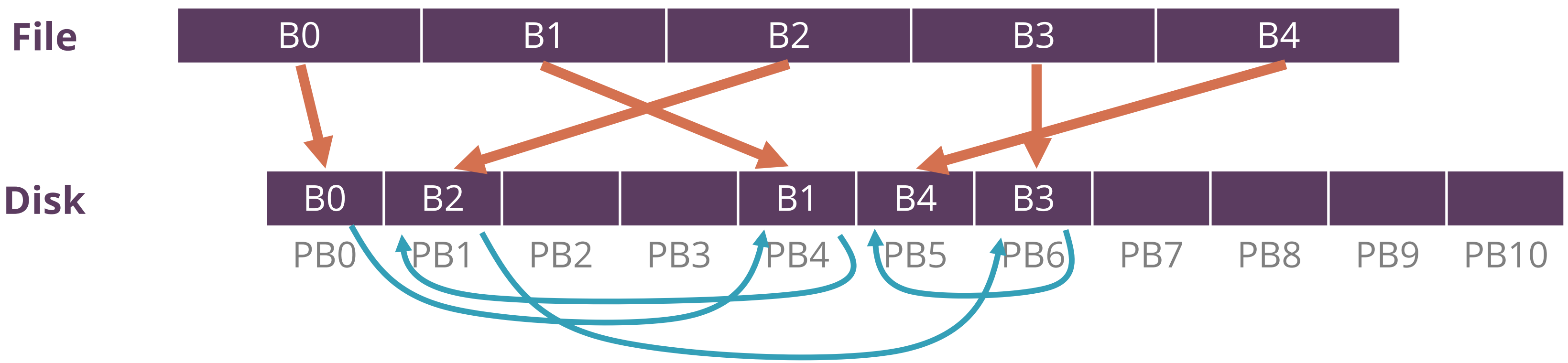
- Similar idea to the free-list idea we had for memory allocation
- Each file in our file system has the block number of its first block.
- Each physical block on disk contains a pointer to the next block or NULL to mark this is the end of the file

Linked List Allocation

- Storing a file as a linked list of disk blocks



Linked List Allocation



Poll Everywhere

pollev.com/tqm

- What are the advantages/disadvantages/concerns with **Linked List Allocation**

- Consider:
  - Space needed
  - Fragmentation
  - Time to scan the whole file
  - Time to access the last block in the file

Linked List Allocation Analysis

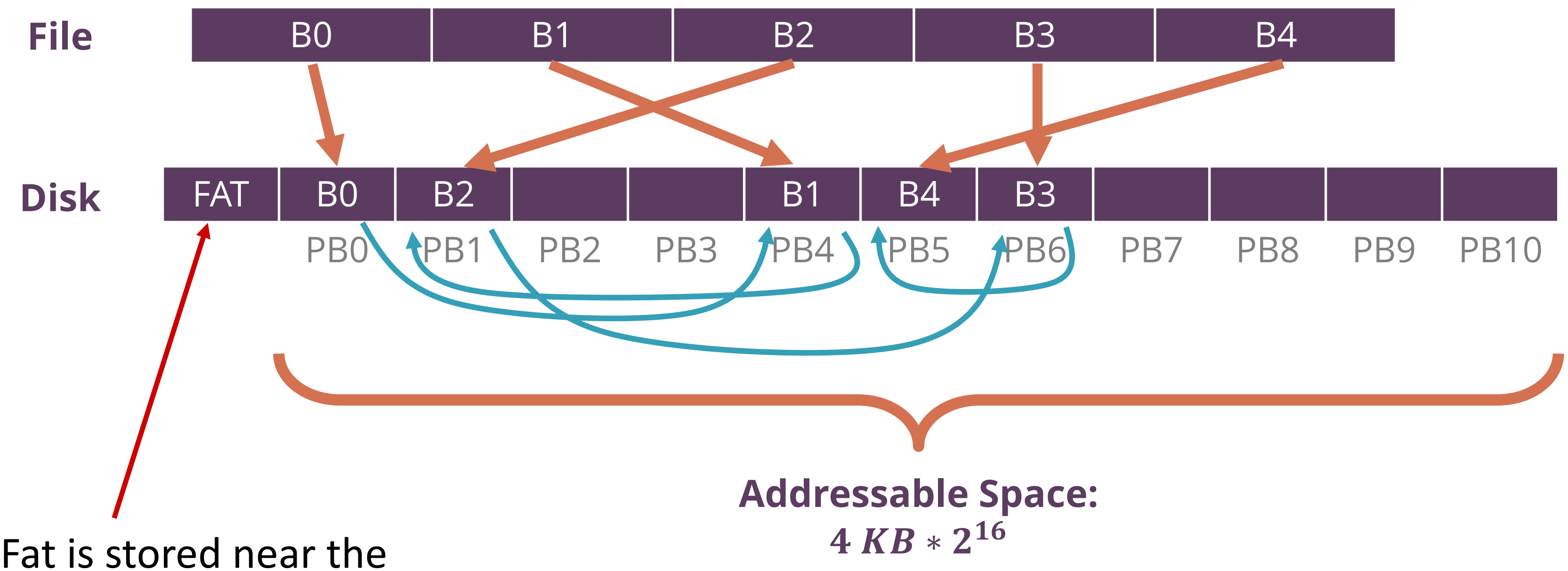
- Pros
  - No External Fragmentation! As long as there is free space, we can put a block there
- Cons
  - Extra Space needed for the "next" pointers
  - Sequential access can be rough
  - Random access can be rough

File Allocation Table (FAT)

- Linked List: to access a block we need to traverse the entire list. Each node in the list could require a new disk seek operation, which takes a long time…

- Idea: store the pointers somewhere else so we don't have to traverse disk to find where the N^{th} block is.
  - Store a table in memory so that it is quick to access.

File Allocation Table (FAT)



Fat is stored near the beginning of disk and loaded into memory when we boot it up

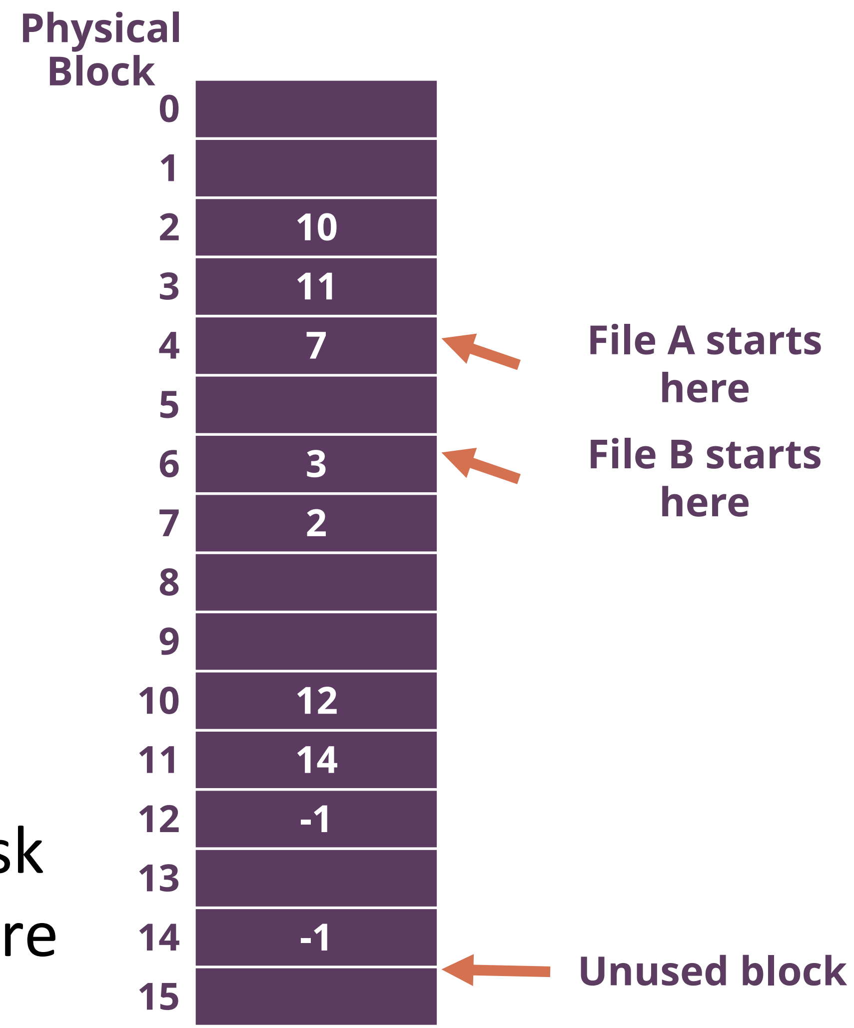
The translation (orange arrows) are done with the FAT table

The blue arrows are stored in the FAT table

FAT Table

- The FAT table consists of an array.
  - The index into the array is the physical block number
  - The value is the block number of the next block in the file or -1 to indicate end of file

- Array must have one entry per physical block on the disk even if some of the blocks are unused.

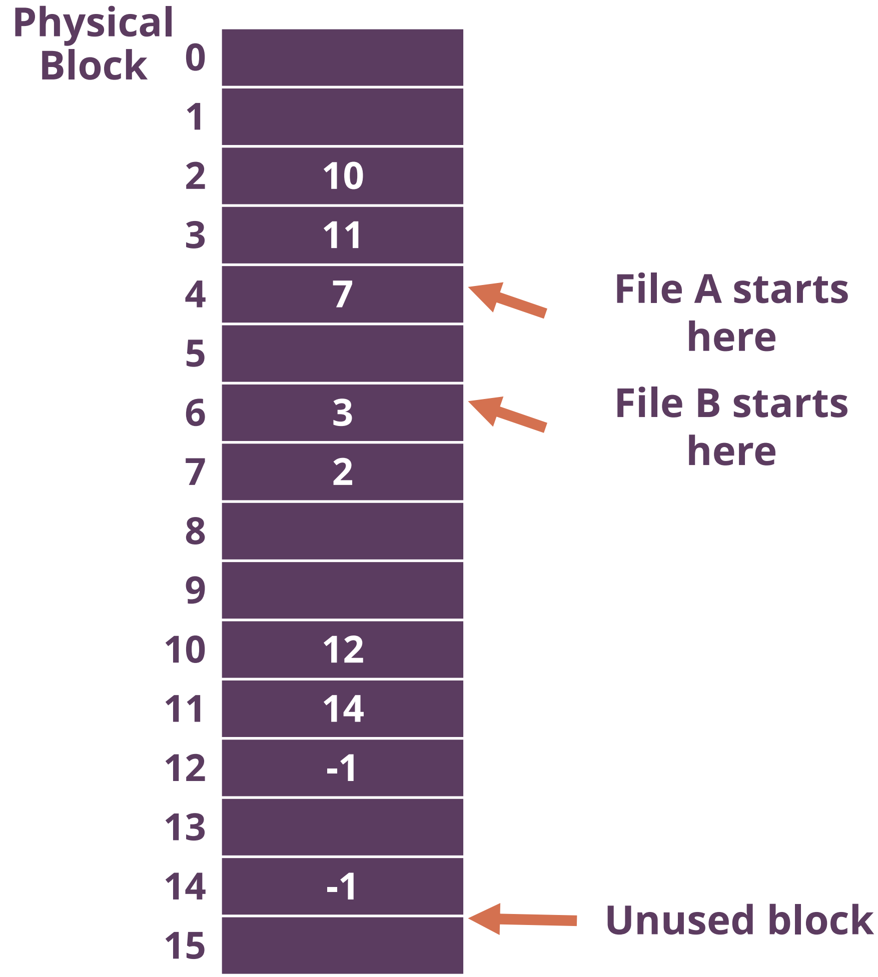


Poll Everywhere

pollev.com/tqm

- What are the blocks that make up file A?

- What about file B?



FAT Allocation Analysis

- Pros
  - No External Fragmentation! As long as there is free space, we can put a block there
  - Random access is a lot quicker!

- Cons
  - Extra Space needed for the FAT table in both memory and disk
  - As Disk spaces increases, the size of the FAT table increases.
  - Sequential access can be rough still ☹

Defragmentation

- Accessing data as it would appear sequentially in a file can still take some time
  - if the blocks are not contiguous on disk, a seek must occur to access the next block and that seek could be far away

- Even with FAT (and other allocation schemes), it is best to keep data of the same file next to each other.

- Defragmentation or "De-fragging" a drive involves rearranging blocks on hardware to be next to each other
  - Can take a while to do, but can speed up disk usage.