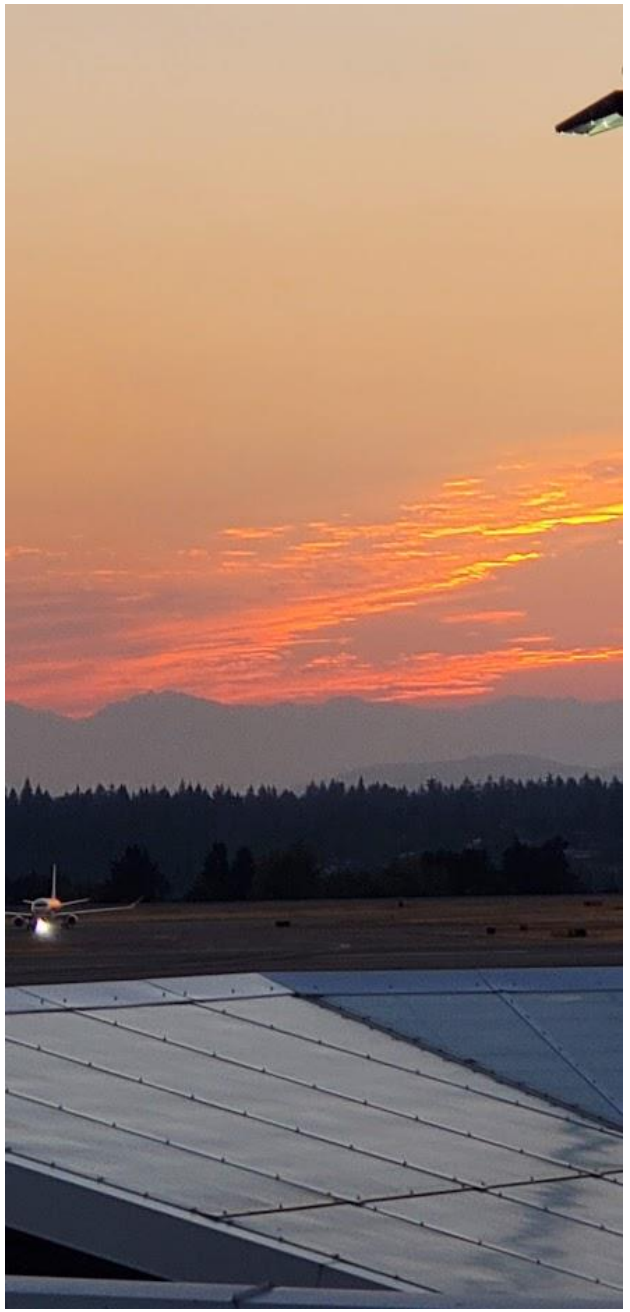
Intro to the File System

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



pollev.com/tqm

- Which operating system do you think is the most popular?

Logistics

- Penn-Shell due tomorrow!
  - Late due-date of Friday @ Midnight
  - Penn-shell Demo's this Friday! Please sign up on Ed if you haven't already!
- Quiz! In first half of class on Tuesday 9/29!
  - Policies posted today
- penn-fat
  - Goes out this weekend!
  - Sign up with your partner on Canvas + Make the Github Repo
  - If you don't sign up with a partner on Canvas by Thursday of Fall Break, we will randomly assign you.

Any Questions?

Lecture Outline

- Intro to File System
  - User Perspective
  - Blocks
- File Allocation
  - Contiguous
  - Linked List
- File Allocation Table (FAT)
  - FAT Walkthrough
  - PennFAT

Files

- You have interacted with files before.
- Files have names to identify them e.g. "Hello.txt"
- Files can be opened, read, written to, saved, deleted, etc..
- A file can store image data, programs, text, etc.
- Files can also be called non-volatile storage
  - This data persists when the computer is powered off, as long as the data is actually written to the file
  - Data that is in memory is volatile.
    - it is lost if the power goes out.
    - If you shoot your computer
    - Your sibling trips over the power chord

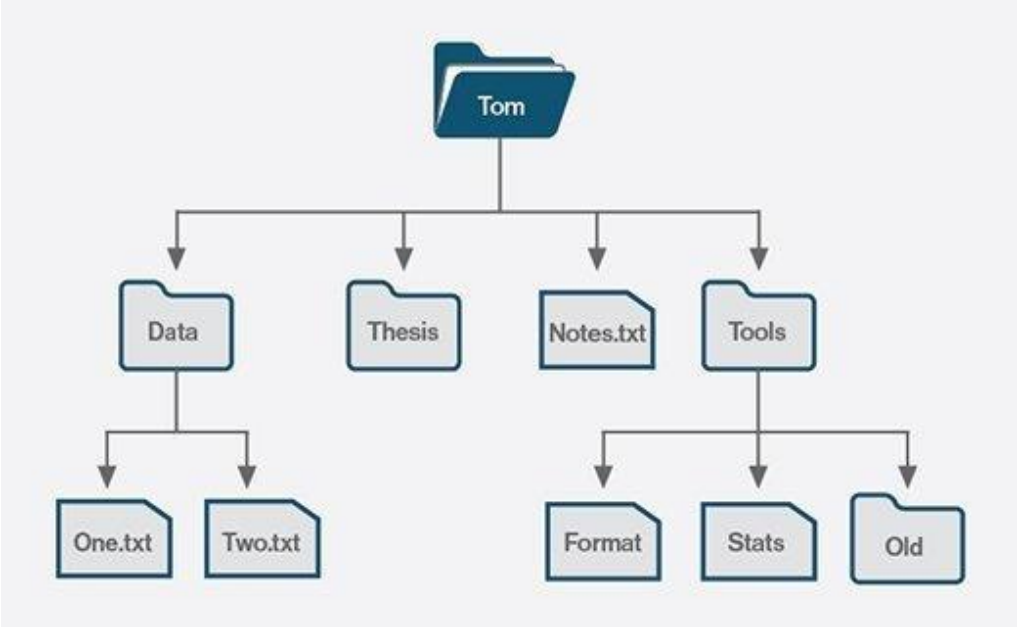
Directories

- A directory is a special type of file that contains a list of other files (and directories) that are “inside” of it
- A directory is also named a “Folder”
- For most cases, we can use the word Directory and Folder interchangeably

Hierarchical File System

- Files on a computer are structured as a **Hierarchical File System**
  - **Introduced:**
    - 1969, Multics (UNIX predecessor). First OS to have this!
    - September, 1985 by Apple
    - March, 1983 in MS-DOS 2.0

- Directories can contain other Directories
  - **Subdirectory** is used to describe a directory contained in another
  - **Parent** and **Child** are often used to describe the relationship between a subdirectory and the directory it is in.
  - With one directory being the “overall root” or “overall parent”



File System: User Level STD API

- C `stdio` API: core functionalities (***with File Structs***)

```
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

- C `stdio` API: core functionalities (***with File Structs***)

```
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);
```

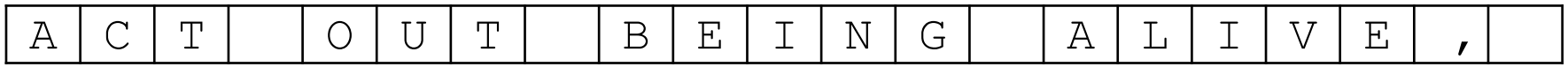
In addition to the above, we also have another common feature: moving to an arbitrary position in the file

```
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” or sequence 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



User Perspective: A stream of bytes

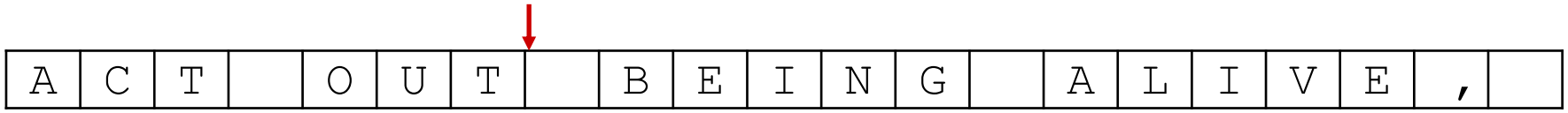
- As a user, we have the idea of a file as being a “stream” or sequence of bytes.
  - a continuous sequence of data made available over time.
  - There are many kinds of streams, for now we are talking about files

- 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” or sequence 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
    - Now we know we are changing the cursor for that open file…



User Perspective: A stream of bytes

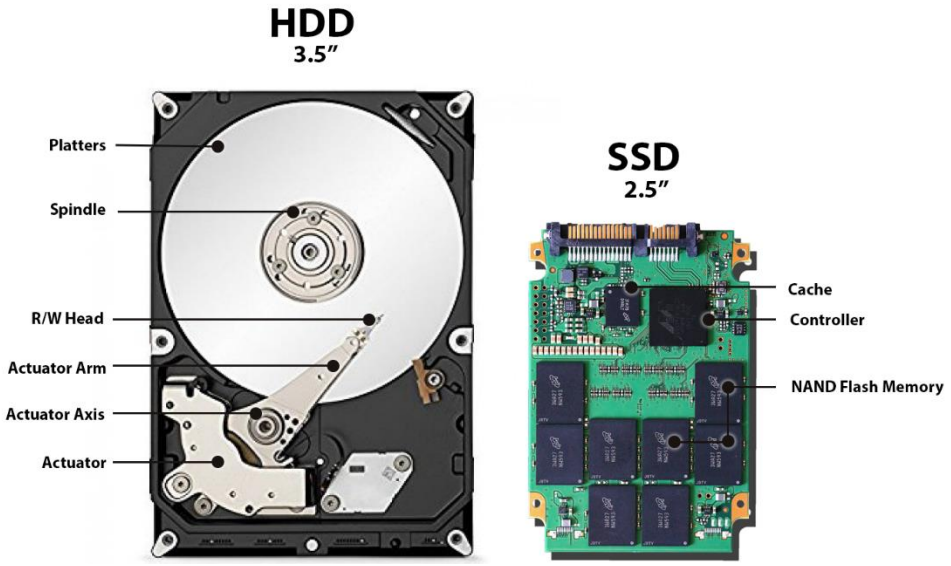
- As a user, we have the idea of a file as being a “stream” or sequence of bytes.
  - a continuous sequence of data made available over time.
  - There are many kinds of streams, for now we are talking about files

- 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.
  - It is a ***hardware thing***

File System

- **File System**: A system composed of algorithms and data structures for how data is stored, organized & retrieved from a storage medium.
  - E.g. how the operating system organizes the physical medium (Hard Disk, SSD, Tape, Floppy Disk, etc) to make the interface/abstraction we saw in the previous slides



The File System Foundations

- So, we have this complicated system of:
  - various files of different lengths
  - Files that can be written, read, extended, shrunk, deleted, copied…
  - Directories that contain files and other directories which can contain other directories etc.
    - Directories can be of various sizes
  - Files can have different permissions (executable, read, write)….
  - Files of the same name can exist in different directories
  - We want to try and support all of this, and have it run relatively fast
- What does the operating system get to implement this?

```
int the_filesystem[REALY_REALLY_BIG];
```

Not quite just an array of ints..

- From the OS perspective, it has to create and manage a file system with this

```
int the_filesystem[REALY_REALLY_BIG];
```

- This is not fully true
  - The "unit" size of elements in the array is not an int (typically 4 bytes) but instead a **block**
    - 512 or 4096 bytes, depending on the implementation and hardware

```
BLOCK the_filesystem[REALY_REALLY_BIG];
```

- **The OS does not get to directly index into the array**, it invokes functions (components either installed onto the kernel or alongside it) that can read or write specific blocks.
  - These functions are part of the hardware's device driver.
  - A device driver is just the software that controls a specific piece of hardware.
    Disk had different drivers than SSD. Two SSDs may have different drivers if they aren't the same model/manufacturer.

Storage Mediums Interface: Blocks

- A block is a fixed number of contiguous bytes
  - Usually, 4096 bytes or 512 bytes
  - Configurable (You will support this)
- **Storage Mediums can be thought of as a giant collection of blocks.**
  - The file system must organize these blocks (and the bytes inside of them) to make the abstractions we talked about. Otherwise, there would just be data with no clear separation of files
- A block is the unit of work for a file system
  - Read and write operations to storage mediums (e.g. disk) are done in multiples of their respective block size
    - So even if you want to change (1 byte) within a file, you must write an entire block of the file
    - The smallest space a file takes up on disk is 1 block*

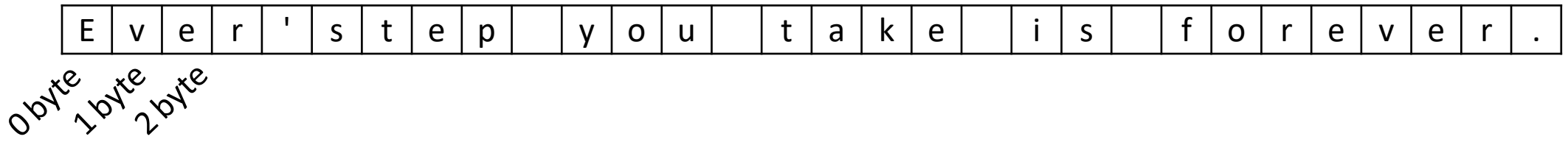
Operating System Perspective: Blocks

- The stream model is ***very convenient*** for user level programs, but hardware works in terms of blocks.

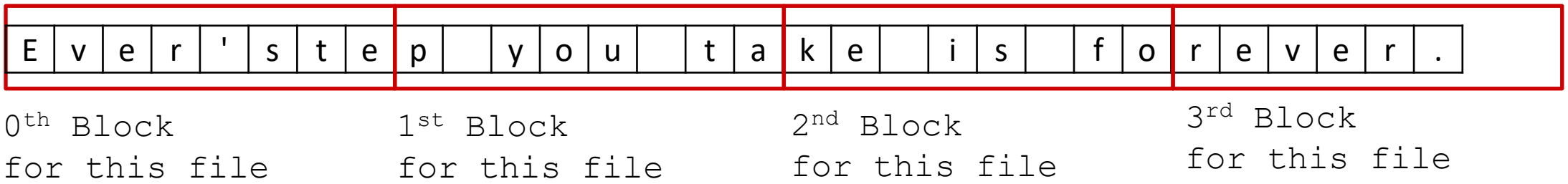
- The file system breaks files up into **blocks** so that it can be stored into the storage hardware.
  - When the operating system interfaces with hardware, it works in terms of blocks.
  - When the OS operates on a file, it reads/writes an entire block at a time
  - The user still sees the file as a stream abstraction, can work with bytes instead of blocks
    - You asked for 1 byte, the kernel read a block worth of information. And just gives you 1 byte. Stores the rest for later in the off chance you read more (which you will).

Operating System Perspective: Blocks

- User perspective: A sequence of bytes



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



These blocks are logically next to each other, but may not be contiguous physically.

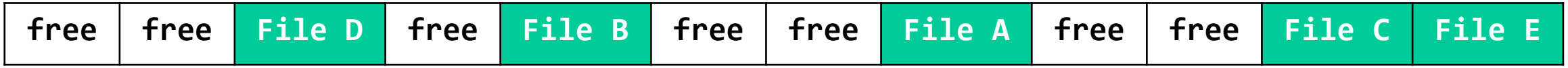
Lecture Outline

- Intro to File System
  - User Perspective
  - Blocks
- File Allocation
  - Contiguous
  - Linked List
- File Allocation Table (FAT)
  - FAT Walkthrough
  - PennFAT

Building up to a full filesystem

- Lets start with a simple abstraction:
  - We have disk that contains many blocks
  - We want to store a few files and just one block per file (so each file is at max ~4096 bytes)

Disk:



- How do we know where a certain file is on disk?
- How do we know which blocks are free?

Solution: Directories

- We can solve one of these problems with the introduction of directories.

- A directory is essentially a special type of file
  - We will store its data on disk within blocks (like a file)

- ***The layout of the directory content is known to the file system.***
  - The file system might maintain a list of directory entries
  - Each directory entry contains the name of the file, the first block number of the file, and some other information

Solution: Directories

- The directory content format is known to the file system.
  - Contains a list of directory entries
  - Each directory entry contains the name of the file, the first block number of the file, and some other information

free	free	File D	free	File B	free	free	File A	free	free	File C	File E
Block 0	Block 1	Block 2	Block 3	Block 4	Block 5	Block 6	Block 7	Block 8	Block 9	Block 10	Block 11

Directory:

File Name	Block Number
A	7
B	4
C	10
D	2
E	11

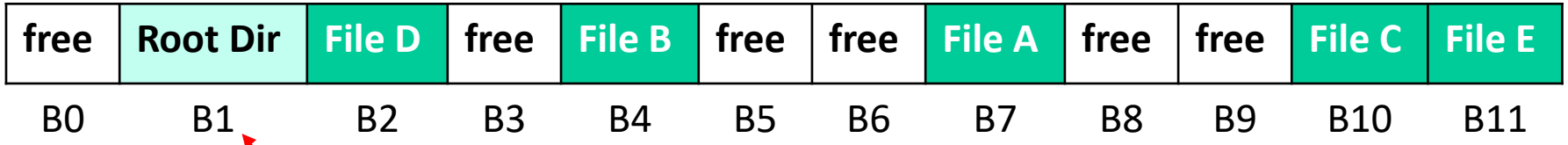
Where does this directory go?
Where do we store its information?
How do we know where the directory is in disk?

Remember: a directory stores its data in blocks in disk too

Solution: Root Directory

- Solution: we have an overall root directory that we always put in the same place (Block 1 or Block 0)

Disk:



Directory:

File Name	Block Number
A	7
B	4
C	10
D	2
E	11

How do we know which blocks are free?

Bitmap

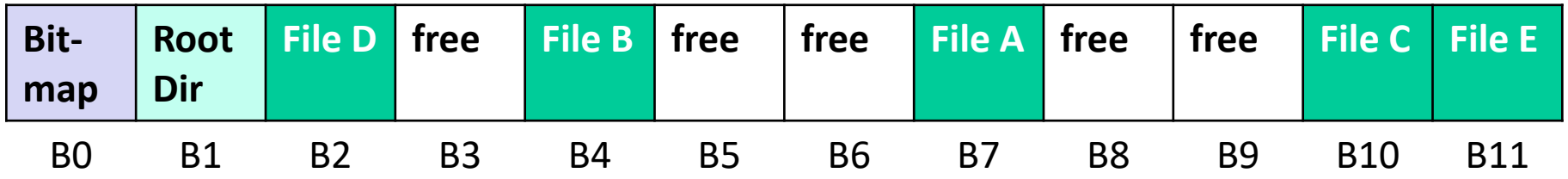
- We can have a bitmap (similar to a bitset) stored in disk to keep track of which blocks are free and which ones are not.
- If we have N blocks, then we need N bits (1 bit per block) to keep track of this information. If a bit is 1 the corresponding block is free, 0 means it is in use.

Here's an example with 16 blocks:

```
1 0 0 1
0 1 1 0
0 1 1 0
1 0 0 1
```

- It is also useful to stick this in the front of the disk, at a fixed location

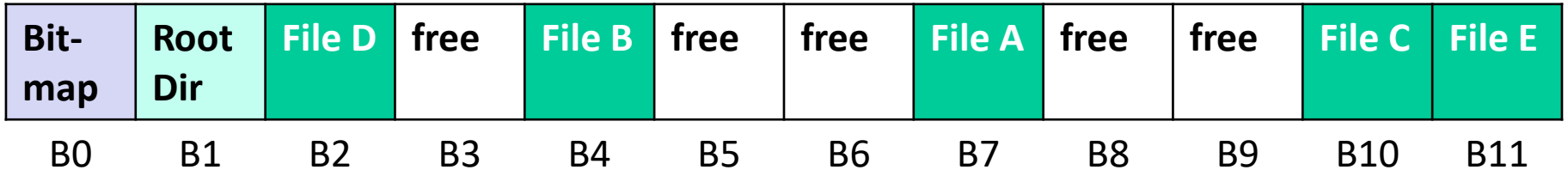
Disk:



Expanding on our model

- What we have works but what happens if we want files that are more than 1 block?

Disk:



- Let's say File B wants to be two blocks long instead of 1 block long

- What is the simplest thing we can do?

Contiguous Allocation

- Solution: let B expand into the block next to it on disk. It is a free block and we can take it

Disk:

- Only other change we need to make is probably have each directory entry also store the number of blocks in the file

Directory:

File Name	Block #	length
…	…	…
B	4	2
…	…	…

- This way of allocating blocks to a file is called Contigious allocation. Each file occupies a contiguous region of blocks

Contiguous Allocation: Random Access

- What if wanted to read the second block of File B?
  - How many blocks would we need to read from disk?
    - Assume we have not read anything in to the OS yet

Disk:

Bit-map	Root Dir	File D	free	File B	Also File B	free	File A	free	free	File C	File E
B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11

Directory:

File Name	Block #	length
…	…	…
B	4	2
…	…	…

Poll Everywhere

pollev.com/tqm

- What if wanted to read the second block of File B?
  - How many blocks would we need to read from disk?
    - **Assume we have not read anything into the OS yet.**

Disk:

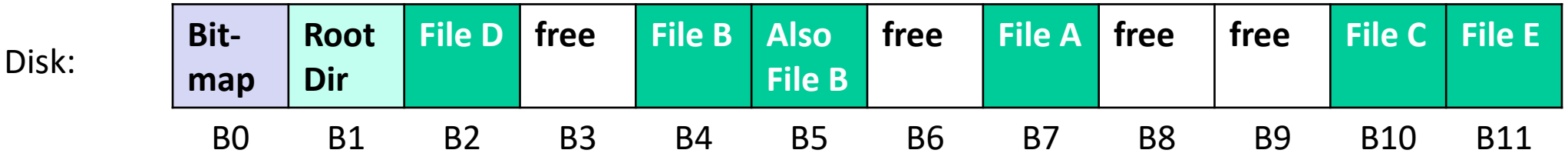
Bit-map	Root Dir	File D	free	File B	Also File B	free	File A	free	free	File C	File E
B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11

Directory:

File Name	Block #	length
…	…	…
B	4	2
…	…	…

Contiguous allocation: problems

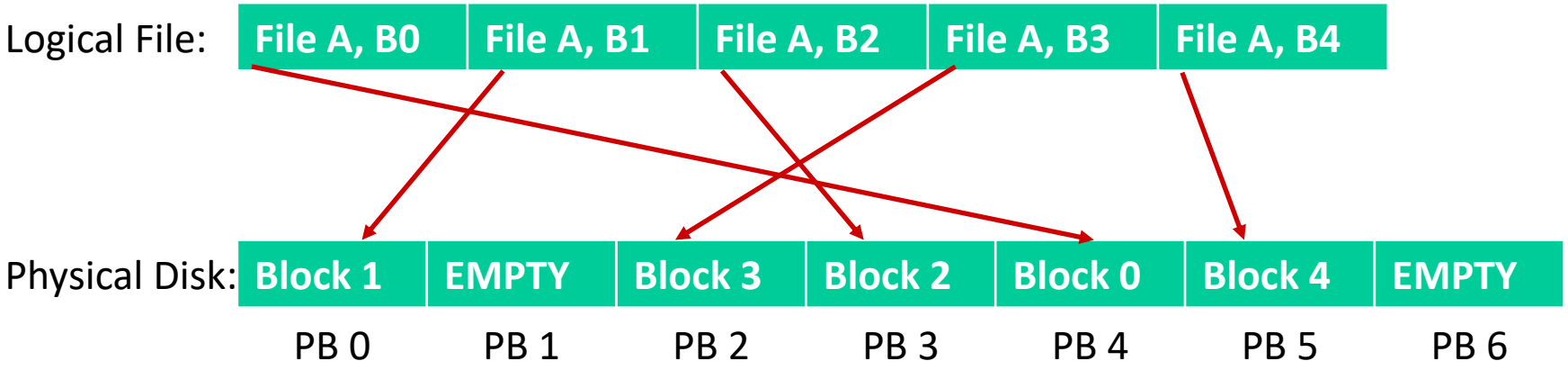
- Let's say File C wants to be two blocks long instead of 1 block long
  - What do we do?



- What if instead File D wants to be 5 blocks long?
  - If we wanted to extend the file but the next block is taken, we either give up or have to rearrange other files in the file system.
- Analysis: this doesn't work very well for files that may grow over time. There is fragmentation that can't be used unless we move files around, which takes a lot of time :/
  - Fragmentation: when we have unused portions ("Holes") of storage that can't easily be re-used, which end up being wasted space.

Do blocks need to be contiguous?

- Logically (from the user view) a file is contiguous.
- The user never directly interfaces with disk, the operating system just has to provide the data in the blocks in order



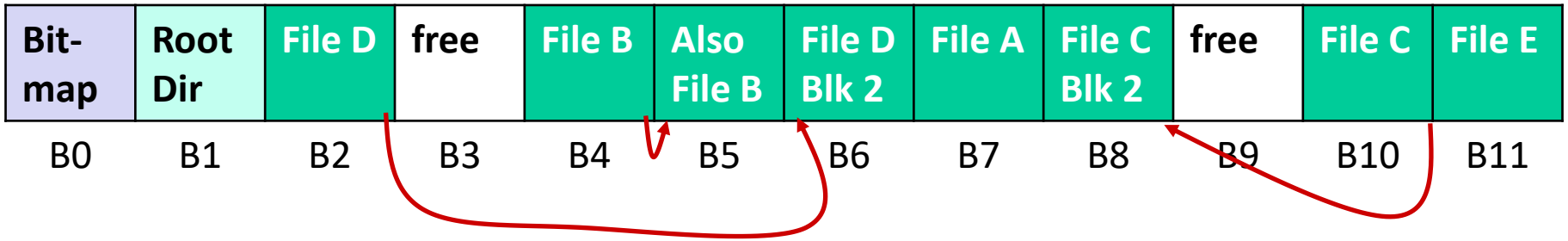
- **The operating system is maintaining the abstraction for the user.** The user asks for the 3rd block of a file, and the operating system will figure out which physical block it is.

Implicit Linked List Allocation

- We can have each block reserve some bits at the end that are pointers to the next block in the file,
  - or a special value to mark that there is no "next block"

- **NOTE:** when we say "*pointer*" here, it is not the same as a memory pointer. This is a "*disk pointer*", meaning it refers to a place in disk and **NOT** a place in memory

Disk:

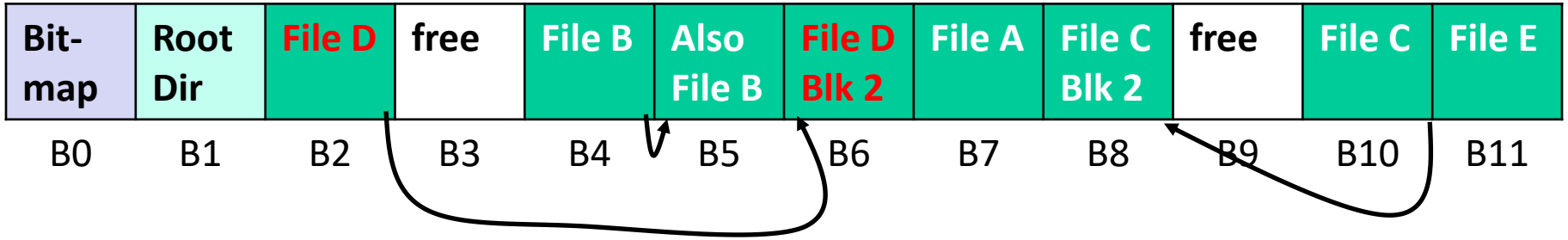


- Root directory still holds the first block number for a file in that file's file entry.

Implicit Linked List Allocation

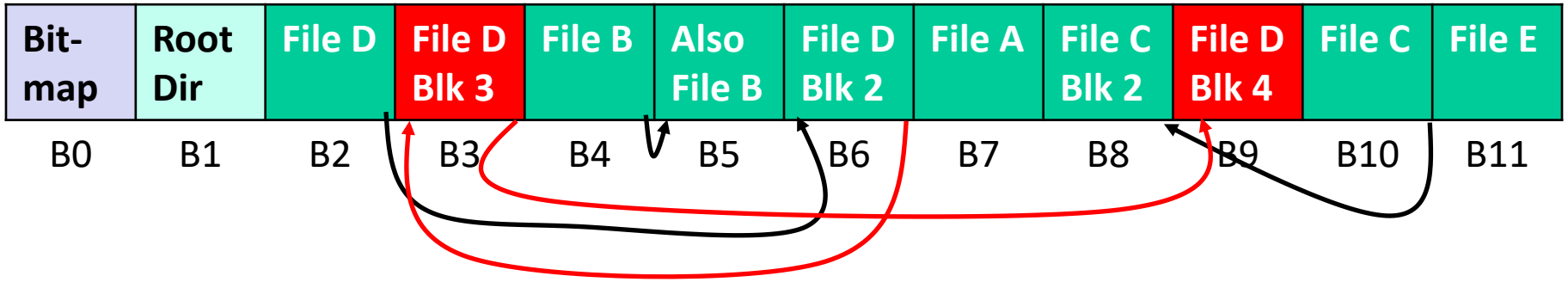
- What if I want to grow File D by 2 blocks?

Disk:



- Scan the bitmap to find which blocks are free
- Allocate the blocks and set up pointers to them

Disk:

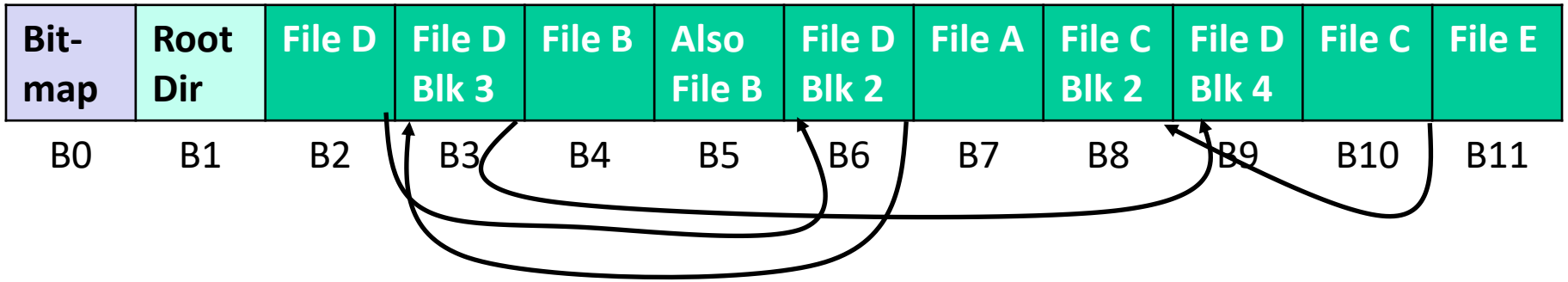


Poll Everywhere

pollev.com/tqm

- Let's say I wanted to read the 4th block of file D.
  How many block reads would be needed? Why?
  - You can assume we already know where the file begins (we have already read the directory entry for the file)

Disk:



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

Linked Allocation Analysis

- Linked List Pros:
  - Growing a file is more feasible
  - Fragmentation issues are less present

- Linked List Cons:
  - Reading can take a lots of seeks to different parts of disk.
    Seeks take up time ☹ (On a Hard Drive Disk)
  - This con is big enough to warrant a different allocation scheme.
    Computer science typically cares A LOT about how quick something is

Lecture Outline

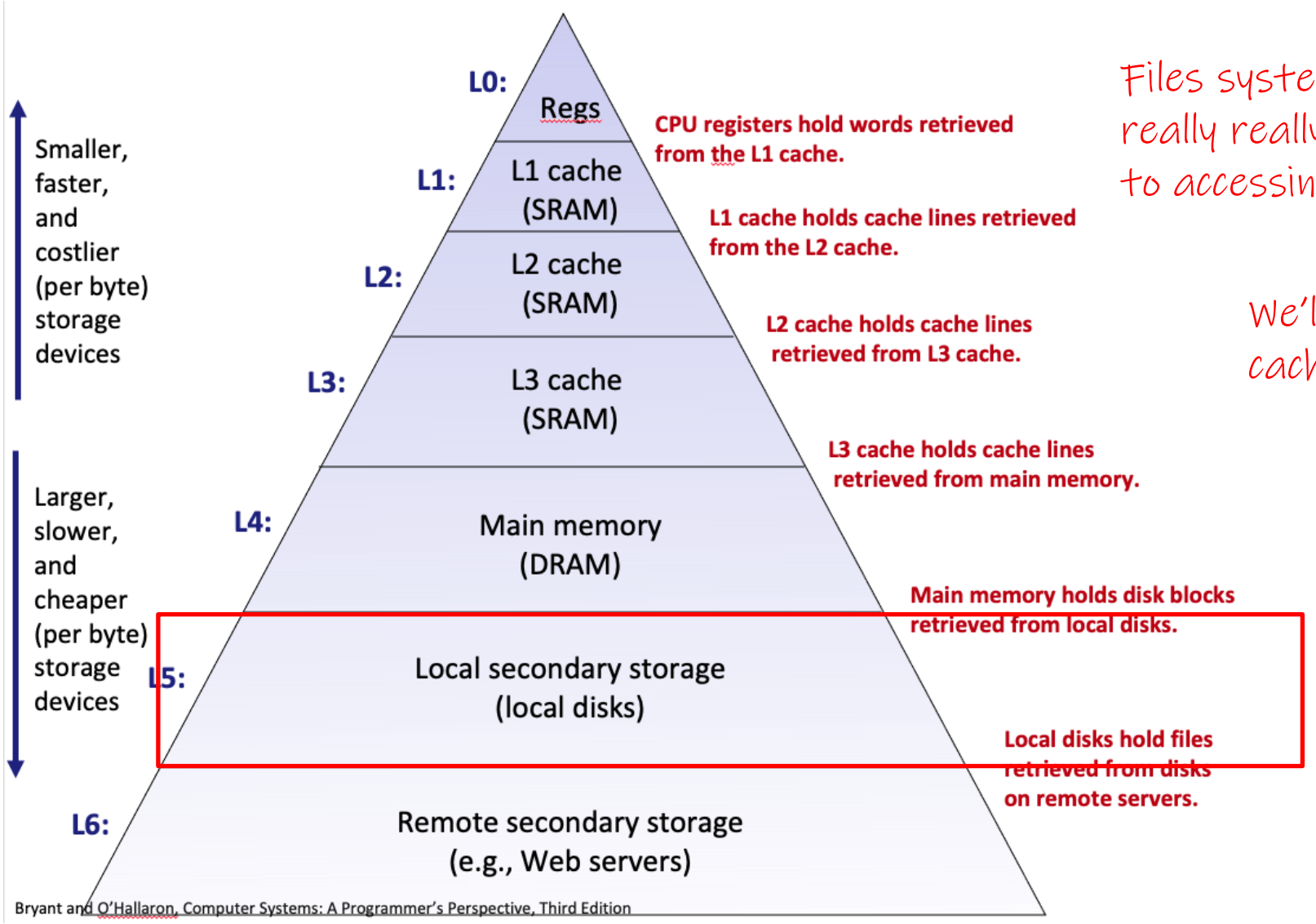
- Intro to File System
  - User Perspective
  - Blocks
- File Allocation
  - Contiguous
  - Linked List
- File Allocation Table (FAT)
  - FAT Walkthrough
  - PennFAT

Linked List via FAT

- We can still have a linked-list “style” approach, we just need a way to make looking up the blocks of a file quicker. We don’t want to access disk so many times if we can help it. O(N) look up to traverse all blocks in the file…

- What can we do instead of accessing disk?
  - What if we could access memory instead?

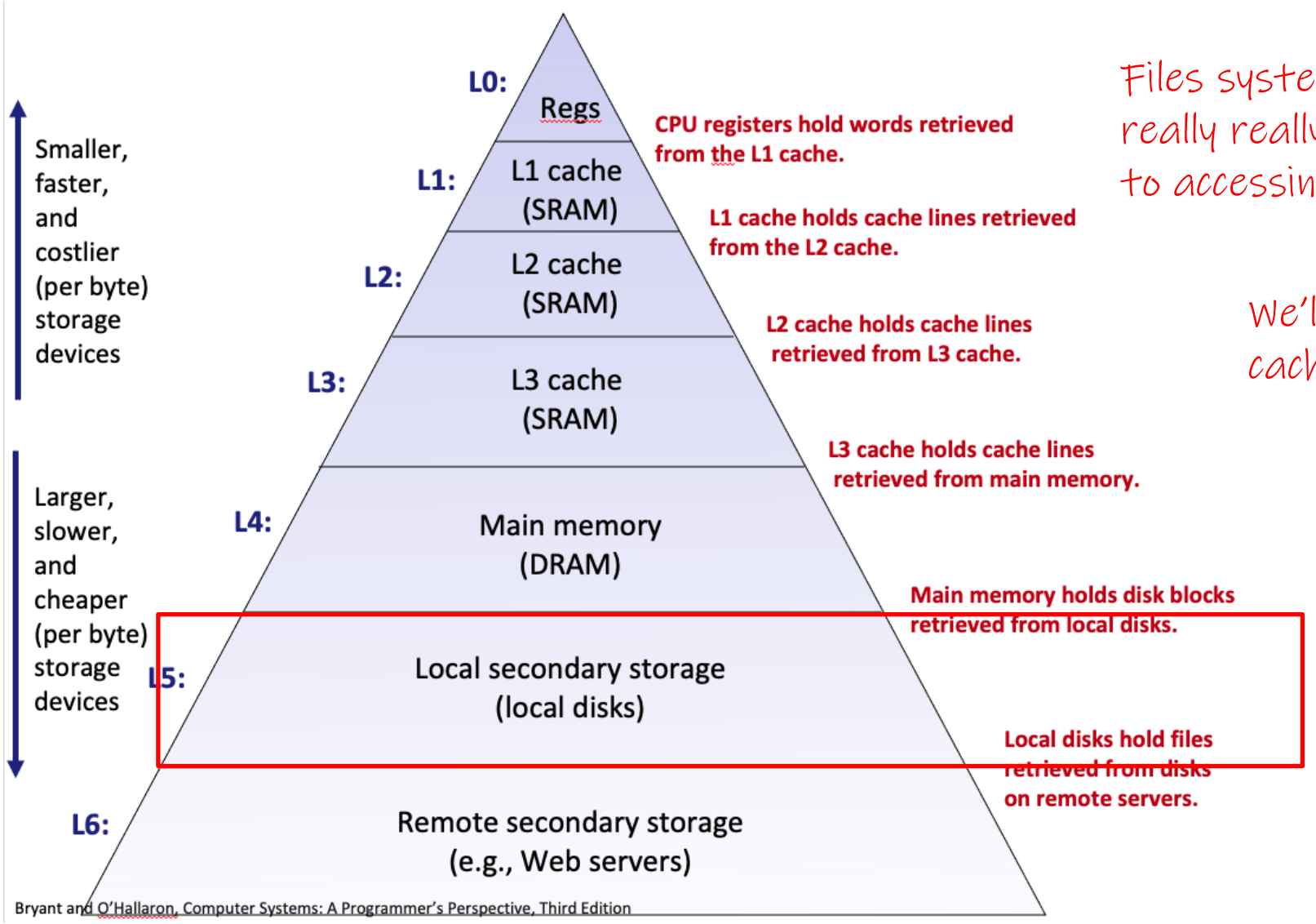
Memory Hierarchy



Files systems are really really really slow compared to accessing memory

We'll talk about caches later

THIS IS REALLY IMPORTANT (discussed later in the sem)



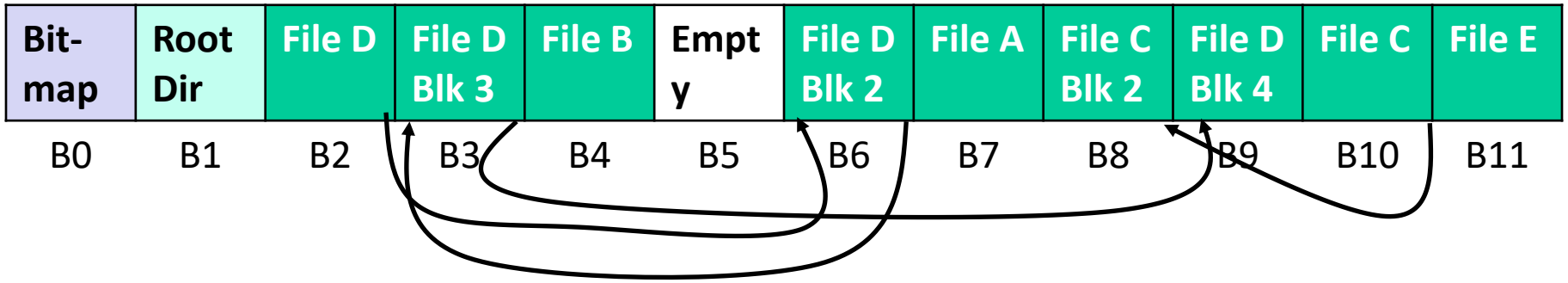
Files systems are really really really slow compared to accessing memory

We'll talk about caches later

FAT (File Allocation Table)

- Instead of this:

Disk:



- We can instead store the pointers or “links” in a table in memory to get…

FAT (File Allocation Table)

- This table is called the
  File **A**llocation **T**able (FAT)
- This table is in memory (RAM) when it is running
- Table stored in disk initially, loaded into memory when computer is booted.
- Replaces the bitmap
  - Why can it do that?

Block #	Next
0	BITMAP/SPECIAL
1	END
2	6
3	9
4	END
5	EMPTY / UNUSED
6	3
7	END
8	END
9	END
10	8
11	END

Disk:

FAT	Root Dir	File D	File D Blk 3	File B	Empty	File D Blk 2	File A	File C Blk 2	File D Blk 4	File C	File E
B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11

Poll Everywhere

pollev.com/tqm

- Let's say I wanted to read the 4th block of file D.
  How many block reads would be needed? Why?
  - You can assume we already know where the file begins (we have already read the directory entry for the file)

Block #	Next
0	BITMAP/SPECIAL
1	END
2	6
3	9
4	END
5	EMPTY / UNUSED
6	3
7	END
8	END
9	END
10	8
11	END

FAT	Root Dir	File D	File D Blk 3	File B	Empty	File D Blk 2	File A	File C Blk 2	File D Blk 4	File C	File E
B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11

FAT Walkthrough

- The FAT is the reason why the operating system knows which block is used for which purpose
- If we wanted to read the 4th block from file D:

Block #	Next
0	BITMAP/SPECIAL
1	END
2	6
3	9
4	END
5	EMPTY / UNUSED
6	3
7	END
8	END
9	END
10	8
11	END

FAT	Root Dir	???	???	???	???	???	???	???	???	???	???
B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11

FAT Walkthrough

- The FAT is the reason why the operating system knows which block is used for which purpose
- If we wanted to read the 4th block from file D:
  - Read the directory entry for File D (from the root dir) to see that it starts at block 2

Block #	Next
0	BITMAP/SPECIAL
1	END
2	6
3	9
4	END
5	EMPTY / UNUSED
6	3
7	END
8	END
9	END
10	8
11	END

FAT	Root Dir	File D Blk 0	???	???	???	???	???	???	???	???	???
B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11

FAT Walkthrough

- The FAT is the reason why the operating system knows which block is used for which purpose
- If we wanted to read the 4th block from file D:
  - Lookup next block in the FAT. We go to FAT entry #2 and the "next" says where the next block is (physical block 6)

Block #	Next
0	BITMAP/SPECIAL
1	END
2	6
3	9
4	END
5	EMPTY / UNUSED
6	3
7	END
8	END
9	END
10	8
11	END

FAT	Root Dir	File D Blk 0	???	???	???	File D Blk 1	???	???	???	???	???
B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11

FAT Walkthrough

- The FAT is the reason why the operating system knows which block is used for which purpose
- If we wanted to read the 4th block from file D:
  - Lookup next block in the FAT. We go to FAT entry #6 and the "next" says where the next block is (physical block 3)

Block #	Next
0	BITMAP/SPECIAL
1	END
2	6
3	9
4	END
5	EMPTY / UNUSED
6	3
7	END
8	END
9	END
10	8
11	END

FAT	Root Dir	File D Blk 0	File D Blk 2	???	???	File D Blk 1	???	???	???	???	???
B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11

FAT Walkthrough

- The FAT is the reason why the operating system knows which block is used for which purpose
- If we wanted to read the 4th block from file D:
  - Lookup next block in the FAT. We go to FAT entry #3 and the "next" says where the next block is (physical block 9)

Block #	Next
0	BITMAP/SPECIAL
1	END
2	6
3	9
4	END
5	EMPTY / UNUSED
6	3
7	END
8	END
9	END
10	8
11	END

FAT	Root Dir	File D Blk 0	File D Blk 2	???	???	File D Blk 1	???	???	File D Blk 3	???	???
B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11

FAT Walkthrough

- The FAT is the reason why the operating system knows which block is used for which purpose
- If we wanted to read the 4th block from file D:
  - The FAT entry for block 9 has a special value for "next" to indicate it is the last block in the file

Block #	Next
0	BITMAP/SPECIAL
1	END
2	6
3	9
4	END
5	EMPTY / UNUSED
6	3
7	END
8	END
9	END
10	8
11	END

FAT	Root Dir	File D Blk 0	File D Blk 2	???	???	File D Blk 1	???	???	File D Blk 3	???	???
B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11

Linked List via FAT

- FAT is logically very similar as a linked list, we just store the links somewhere else that can be conveniently stored in memory
- Since the links are in memory, we can find the N^{th} block of a file with much fewer disk accesses
- Disk accesses take a long time, so this is good ☺

Poll Everywhere

pollev.com/tqm

- If we want to extend a file in FAT what steps do we need to take?

- Hint: FAT is in memory, what are the big differences between Disk and Memory?

FAT is great ☺*

- FAT has allowed us to have non-contiguous blocks for a file.
- At the same time, we only need one disk read to access the Nth block of a file

- What could go wrong with this?
  - FAT existed in an era when storage devices were smaller. FAT was big, but not too big..
  - ***Then storage devices got bigger, so FAT got really big and is in memory, so memory consumption goes up*** ☹
  - Still used in some cases today!

FAT (The Table) size

- A FAT is similar to a bitmap
  - A bitmap needs 1 bit per block
  - A FAT needs ~16-bits per block ☹
- At least we don't need a bitmap anymore!

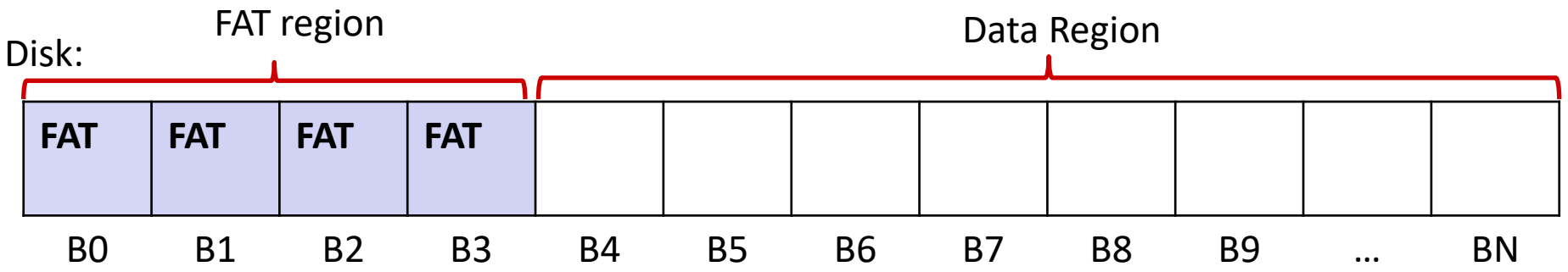
- Grows a lot as the size of disk grows
  - As the disk grows, there are more blocks in the disk. We need more FAT entries, and each entry needs more bits. (To hold the block number. # of bits for block # grows to support more blocks)
  - **The File Allocation Table may be bigger than one block**
  - ***we need to keep the FAT in memory to keep accesses fast***, memory consumption goes up!
  - FAT got fazed out for I-nodes (next lecture) because of this (thank god I hate FAT)

Poll Everywhere

pollev.com/tqm

- When you create a file system with PennFAT, you specify the number of blocks the File Allocation Table takes up and the size of a block.

- Let's say I want to create a FAT that spans 4 blocks, a block is 4096 (2^{12}) bytes, and an entry in the table is 2 bytes.
  - How many entries in the table do I have?



PennOS FAT Details

- If we have N entries in the File Allocation Table, we only have N – 1 references to data blocks in the FAT

- The first File Allocation Table entry **`FAT[0]`** holds meta data about the FAT, so it doesn't refer/point to a "real" block

- An entry is 16-bits, which is 2 bytes, short, or uint16_t, etc.
- Consider the example 2-byte value: 0x2004
  - We can split this into two bytes for **`FAT[0]`**
  - The MSB (Most Significant Byte) 0x20 -> 32 in decimal
  - The LSB (Least Significant Byte) 0x04 -> 4 in decimal

PennOS FAT[0] MSB

- The first FAT entry `FAT[0]` holds meta data about the FAT, so it doesn't correspond to a "real" block
- Consider the example 2-byte value: 0x2004
  - We can split this into two bytes
  - The MSB (Most Significant Byte) 0x20 -> 32 in decimal
  - The LSB (Least Significant Byte) 0x04 -> 4 in decimal

- The MSB is size of the File Allocation Table in units of blocks
  - in this example, the FAT is 32 blocks

PennOS FAT[0] LSB

- The first FAT entry `FAT[0]` holds meta data about the FAT, so it doesn't correspond to a "real" block
- Consider the example 2-byte value: 0x2004
  - We can split this into two bytes
  - The MSB (Most Significant Byte) 0x20 -> 32 in decimal
  - The LSB (Least Significant Byte) 0x04 -> 4 in decimal

- The LSB is between 0 and 4, and specifies the size of the blocks for the file system

LSB	Block Size
0	256
1	512
2	1,024
3	2,048
4	4,096

PennOS FAT Entry Special Values

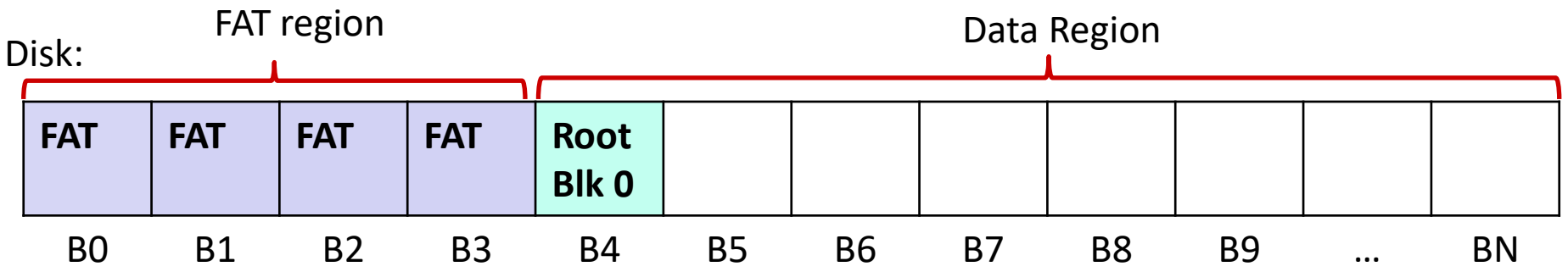
- A PennFAT entry is 16-bits and only contains the block number of the next block in the file.

- There are two special values a PennFAT entry can hold

- 0x0000 (0 in decimal)
  - Indicate the block is free.
  - We start indexing into our blocks in the data region starting with index 1 🤢🤢 😫😫😫
- 0xFFFF (65535 as unsigned, -1 as signed)
  - Indicates that there is no block after this logically in the file
  - That this is the last block in the file

PennOS root Directory

- PennFAT has a special value for `FAT[1]` as well.
- It still corresponds to a data block, but that data block is the first block of the root directory
- This means we always know where the root directory starts. (at index 1 into the data region), and from there we can find all other files
  - *…pathname resolution soon…*



Think About How You'd like to Design your Penn-FAT ☺

- Next time: I-Nodes and more! ☺

Lecture Outline

- Intro to File System
  - User Perspective
  - Blocks
- File Allocation
  - Contiguous
  - Linked List
- File Allocation Table (FAT)
  - FAT Walkthrough
  - PennFAT
- Poll Everywhere Solutions

Poll Everywhere

pollev.com/tqm

- What if wanted to read the second block of File B?
  - How many blocks would we need to read from disk?

2 Blocks! Why?

Second, we read the Block of Interest

Disk:

First, we read the Root Dir Block.

Directory:

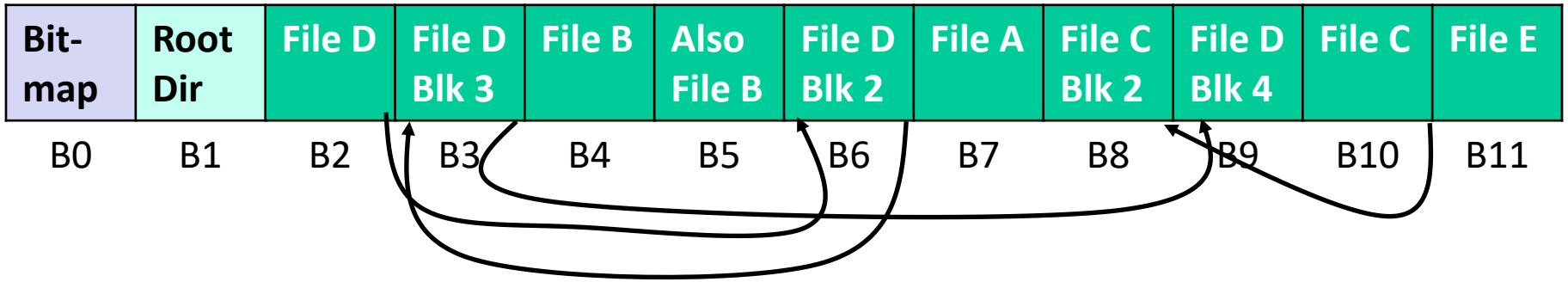
File Name	Block #	length
…	…	…
B	4	2
…	…	…

Poll Everywhere

pollev.com/tqm

- Let's say I wanted to read the 4th block of file D.
  How many block reads would be needed? Why?
  - You can assume we already know where the file begins (we have already read the directory entry for the file)

Disk:



- **4 block reads ☹**
- **We need to read each block to find where the next block is located. ☹**

Poll Everywhere

pollev.com/tqm

- Let's say I wanted to read the 4th block of file D.
  How many block reads would be needed? Why?
  - You can assume we already know where the file begins (we have already read the directory entry for the file)

Just one block!!!

Block #	Next
0	BITMAP/SPECIAL
1	END
2	6
3	9
4	END
5	EMPTY / UNUSED
6	3
7	END
8	END
9	END
10	8
11	END

We can do this traversal in software land, no need to read these blocks every time.

FAT	Root Dir	File D	File D Blk 3	File B	Empty	File D Blk 2	File A	File C Blk 2	File D Blk 4	File C	File E
B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11

Poll Everywhere

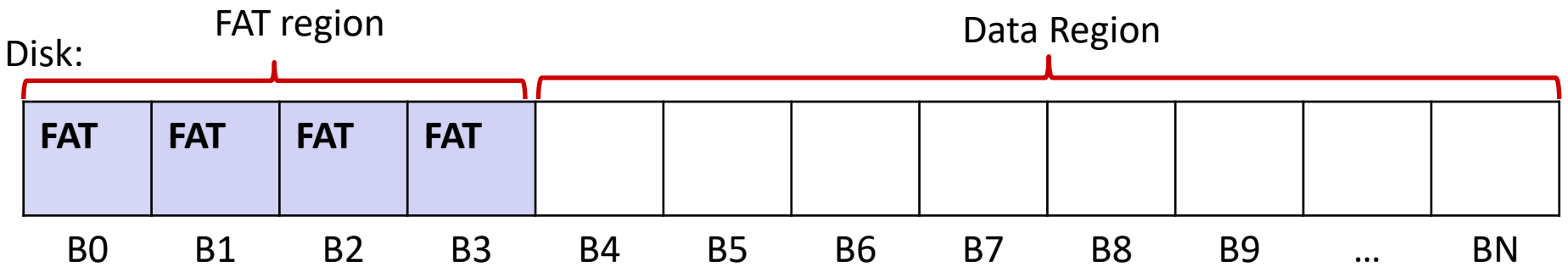
pollev.com/tqm

- If we want to extend a file in FAT what steps do we need to take?
  - Lookup a free block in the FAT, mark it as a last block
  - Lookup the last block in the file, change its FAT entry to think the newly allocated block is the new “last”
  - …
  - Write the FAT table to disk, memory is volatile storage

- Hint: FAT is in memory, what are the big differences between Disk and Memory?

Poll Everywhere

pollev.com/tqm

- When you create a file system with PennFAT, you specify the number of blocks the File Allocation Table takes up and the size of a block.

- Let's say I want to create a FAT that spans 4 blocks, a block is 4096 (2^{12}) bytes, and an entry in the table is 2 bytes.
  - How many entries do I have? $4 * 2^{12} / 2 = [2^{13}]$