# Inodes, Directories, mmap()

## 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

- MILESTONE 0 IS DUE around Friday 11/3 @ MIDNIGHT
  - You should already be in a group
  - I sent an email to everyone in a group that had some amount of random assignment
  - Please meet with your TA, you should have been contacted by them soon.

# Administrivia

- I synched a bunch of grades to canvas. **PLEASE CHECK THAT THEY ARE ACCURATE**
  - All check-ins
  - Project 0 & peer-eval

- Midterm grades to be released soon
  - There will be a period where you can submit regrade requests
  - More info on Ed soon
  - Solutions will be posted shortly afterwards

- Will also post some example PennOS filesystem files after lecture

# Poll Everywhere

**pollev.com/tqm**

- Any questions, comments or concerns from last lecture?

# Lecture Outline

- **Inodes**
- Directories
- Block Caching
- mmap & PennOS stuff

# Poll Everywhere

pollev.com/tqm

- What was the big downside of using FAT?

# Poll Everywhere

pollev.com/tqm

- What was the big downside of using FAT?

- **Big memory consumption, one entry needed for every block in the file system, and that all needs to be in memory.**
  - **A FAT likely spans multiple blocks**
  - **This size also grows as disk grows :/**

# Poll Everywhere

pollev.com/tqm

- Could we instead store FAT blocks on disk and only load into memory the parts that are used for looking up files that are currently open/being used?

# Poll Everywhere

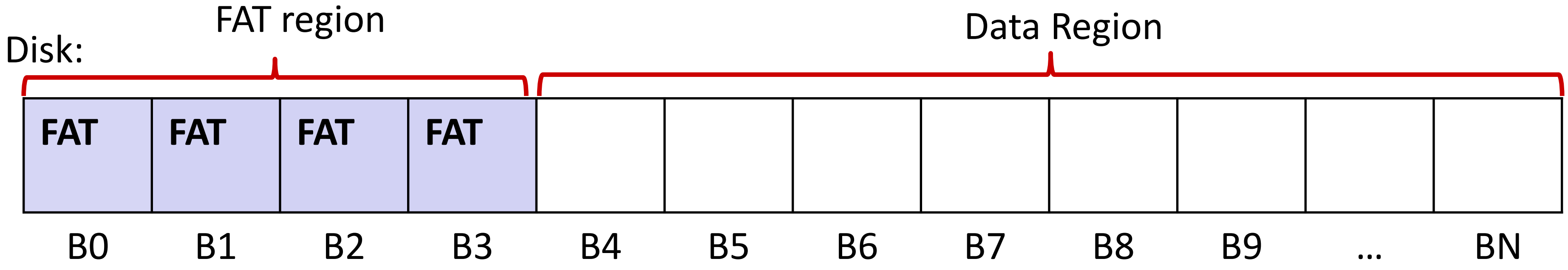
pollev.com/tqm

- Could we instead store FAT blocks on disk and only load into memory the parts that are used for looking up files that are currently open/being used?

- **Yes, but the blocks of a file could be spread out across disk. We may have to load all FAT blocks to lookup a file anyways**

# Explanation

- Blocks of a file could be spread out across disk. We may have to load all FAT blocks to lookup a file anyways
- Small example:
  - consider block size 256,
  - FAT entry 2 bytes, so 128 entries per FAT block
  - FAT takes up 4 blocks
- **Reminder: FAT region is separate from the data region (blocks it manages)**

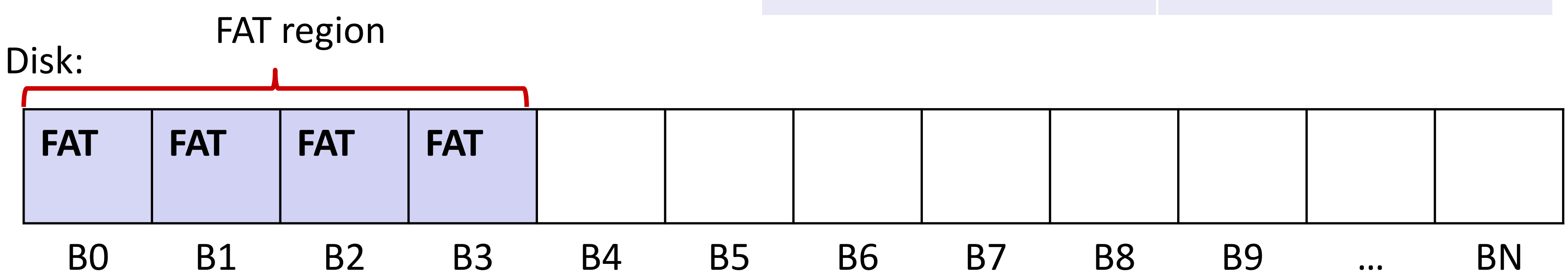


# Explanation

**Consider we have a file that starts at block 2 into the data region**

- Blocks of a file could be spread out across disk. We may have to load all FAT blocks to lookup a file anyways
- Small example:
  - consider block size 256,
  - FAT entry 2 bytes, so 128 entries per FAT block
  - FAT takes up 4 blocks

| Block # | Next |
|---|---|
| … | |
| 2 | 128 |
| … | |
| 128 | 256 |
| … | |
| 256 | 500 |
| … | |
| 500 | |



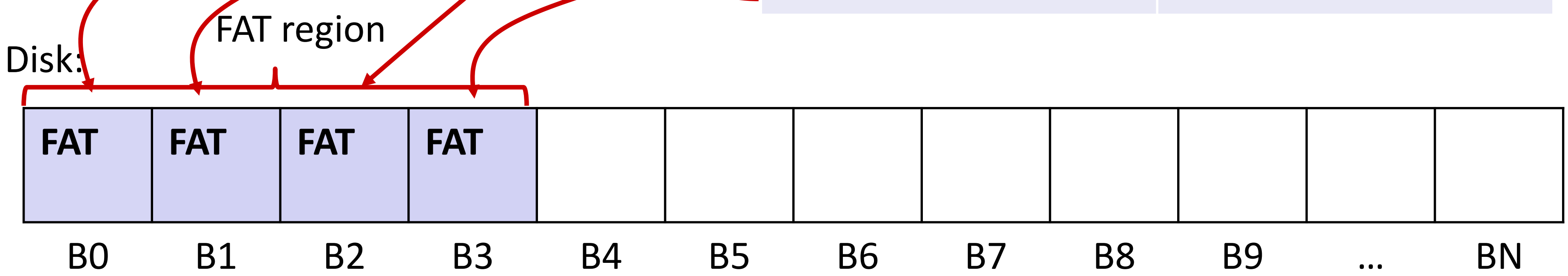
# Explanation

**Consider we have a file that starts at block 2 into the data region**

**We would need to read in the whole FAT just to look up this file**

- Blocks of a file could be spread out across disk. We may have to load all FAT blocks to lookup a file anyways
- Small example:
  - consider block size 256,
  - FAT entry 2 bytes, so 128 entries per FAT block
  - FAT takes up 4 blocks

| Block # | Next |
|---|---|
| … | |
| 2 | 128 |
| … | |
| 128 | 256 |
| … | |
| 256 | 500 |
| … | |
| 500 | |



# Inode motivation

- Idea: we usually don't care about ALL blocks in the file system, just the blocks for the currently open files

- Can we group the block numbers of a file together?

- Yes: we call these inodes:
  - Contains some metadata about the file and 12 physical block numbers corresponding to the first 12 logical blocks of a file

| |
|---|
| meta data |
| $0^{th}$ phys block # |
| $1^{st}$ phys block # |
| $2^{nd}$ phys block # |
| $3^{rd}$ phys block # |
| $4^{th}$ phys block # |
| … |
| $12^{th}$ phys block # |

# Inode layout

- Inodes contain:
  - some metadata about the file
    - Owner of the file
    - Access permissions
    - Size of the file
    - Time of last change
  - 12 physical block numbers corresponding to the first 12 logical blocks of a file
- In C struct format:

```
struct inode_st {
  attributes_t metadata;
  block_no_t blocks[12];
  // more fields to be shown
  // on later slides
};
```

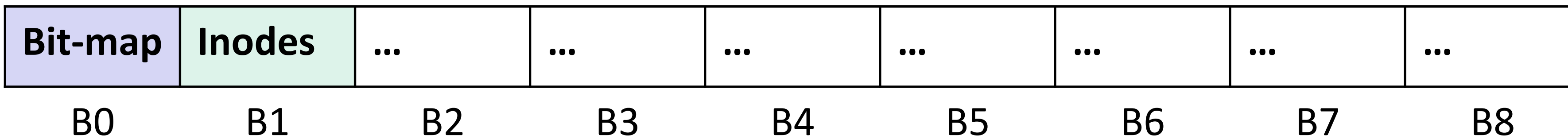
# Inodes Disk Layout

- When we use Inodes instead of FAT, we get something like this instead:

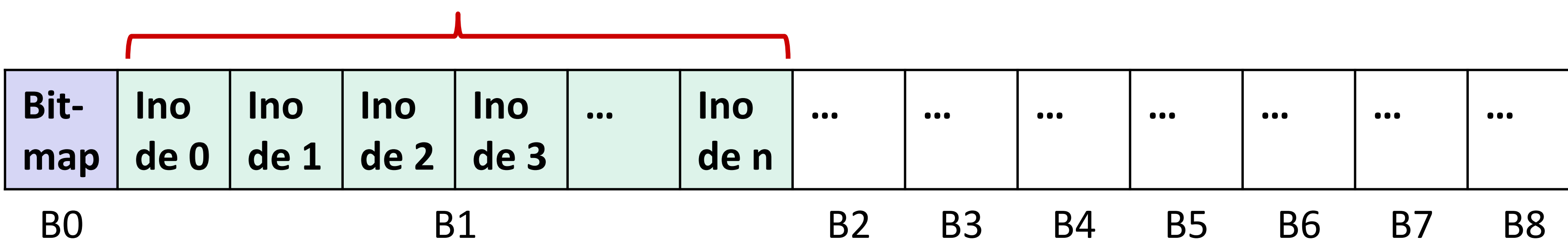
| **Bit-map** | **Inodes** | **…** | **…** | **…** | **…** | **…** | **…** | **…** |
|---|---|---|---|---|---|---|---|---|
| B0 | B1 | B2 | B3 | B4 | B5 | B6 | B7 | B8 |

# Inodes Disk Layout

- When we use Inodes instead of FAT, we get something like this instead:




- Inodes are smaller than a block, can fit multiple inodes in a single block
- Each Inode is numbered



# Example File Block Lookup

- Each File will have an Inode number
- Suppose that we wanted to look up a file that is made of 4 blocks.
  - First, we need the Inode number for the file (lets assume it is 2)

| Bit-map | Ino de 0 | Ino de 1 | Ino de 2 | Ino de 3 | … | Ino de n | … | … | … | … | … | … | … |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| B0 | | | B1 | | | | B2 | B3 | B4 | B5 | B6 | B7 | B8 |

# Example File Block Lookup

- Each File will have an Inode number
- Suppose that we wanted to look up a file that is made of 4 blocks.
  - First, we need the Inode number for the file (lets assume it is 2)
  - We can read the Inode to see which blocks makeup the file

| | |
|---|---|
| meta data | … |
| $0^{th}$ phys block # | 0 |
| $1^{st}$ phys block # | 5 |
| $2^{nd}$ phys block # | 3 |
| $3^{rd}$ phys block # | 2 |
| … | |

The block numbers in the Inode are indexes relative to the start of the data region.

You will be doing this in PennOS too

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Bit-map** | **Inode 0** | **Inode 1** | **Inode 2** | **Inode 3** | **…** | **Inode n** | **…** | **…** | **…** | **…** | **…** | **…** | **…** |
| B0 | | | B1 | | | | B2 | B3 | B4 | B5 | B6 | B7 | B8 |

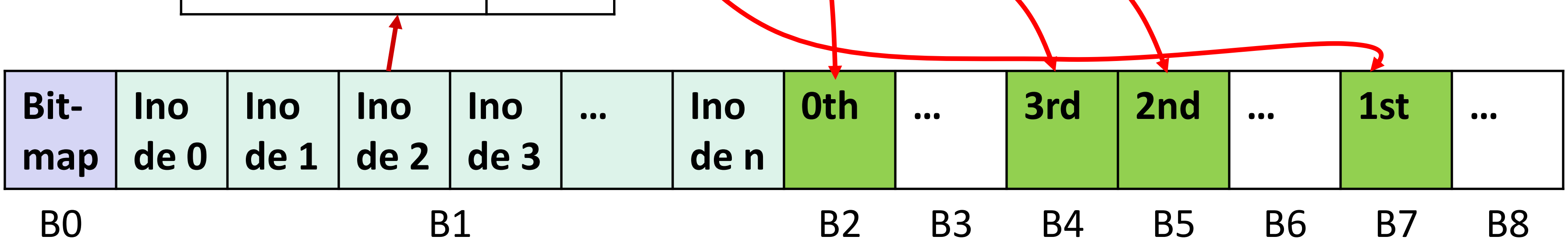
# Example File Block Lookup

- Each File will have an Inode number
- Suppose that we wanted to look up a file that is made of 4 blocks.
  - First, we need the Inode number for the file (lets assume it is 2)
  - We can read the Inode to see which blocks makeup the file

| meta data | … |
|---|---|
| 0th phys block # | 0 |
| 1st phys block # | 5 |
| 2nd phys block # | 3 |
| 3rd phys block # | 2 |
| … | |

The block numbers in the Inode are indexes relative to the start of the data region.

You will be doing this in PennOS too



# File Sizes with Inode

- So with Inodes, how many blocks can we have per file?
  - So far: 12 blocks per file (this is not enough, way too small!
- We can allocate a ***block*** to hold more block numbers
  - This block can hold 128 block numbners

| | |
|---|---|
| meta data | … |
| $0^{th}$ phys block # | 0 |
| $1^{st}$ phys block # | 5 |
| … | … |
| $11^{th}$ phys block # | 2 |
| Block of ptrs | |
| … | |

| | |
|---|---|
| $12^{th}$ phys block # | -- |
| $13^{st}$ phys block # | -- |
| … | … |
| $139^{th}$ phys block # | -- |

# File Sizes with Inode

- So with Inodes, how many blocks can we have per file?
  - So far: 12 blocks per file (this is not enough, way too small!

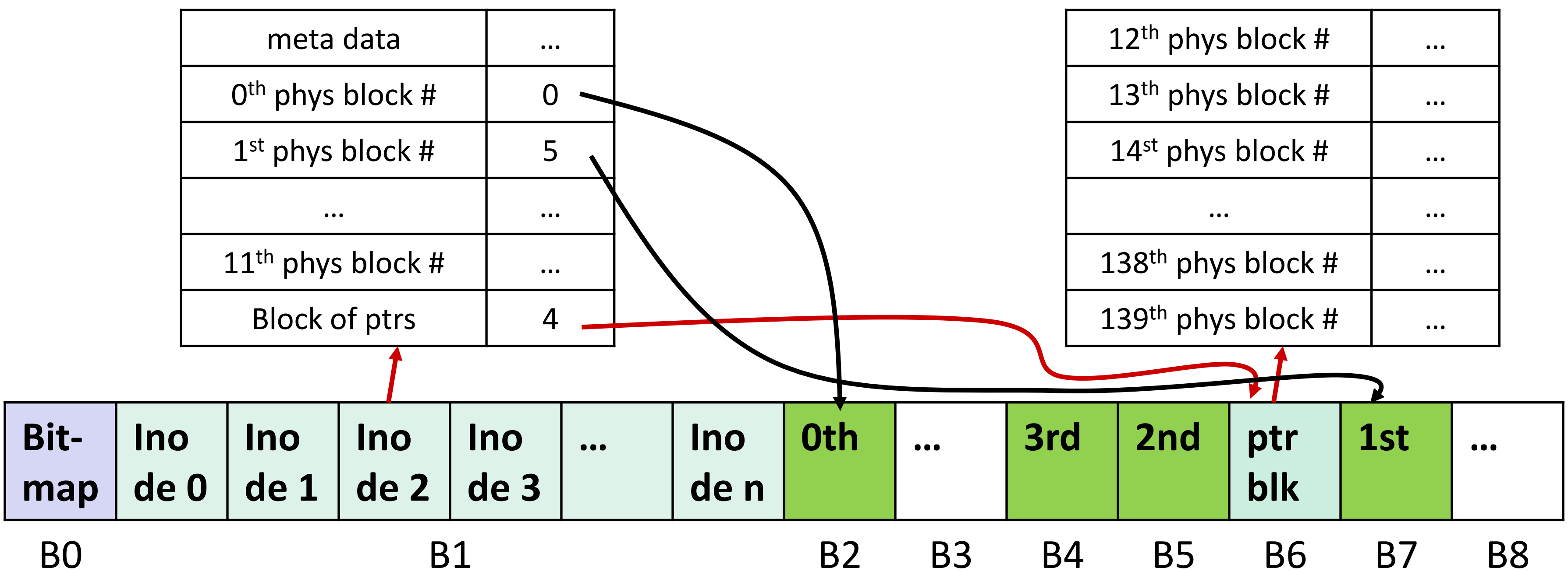
- We can allocate a block to hold more block numbers

```
struct inode_st {
  attributes_t metadata;
  block_no_t blocks[12];
  block_no_t more_pointers;
  // more fields to be shown
  // on later slides
};
```

# File Sizes with Inode

- So with Inodes, how many blocks can we have per file?
  - So far: 12 blocks per file (this is not enough, way too small!

- We can allocate a block to hold more block numbers



# We need moreeeeee

- What if a file needs more than 140 blocks?
- Add another field to the inode that refers to a block that refers to other blocks that refer to data blocks

| | |
|---|---|
| meta data | … |
| $0^{th}$ phys block # | 0 |
| $1^{st}$ phys block # | 5 |
| … | … |
| $11^{th}$ phys block # | 2 |
| Block of ptrs | |
| Indirect block | |
| … | |

| | |
|---|---|
| $12^{th}$ phys block # | -- |
| $13^{st}$ phys block # | -- |
| … | … |
| $139^{th}$ phys block # | -- |

| | |
|---|---|
| Block for 140-267 | -- |
| Block for 268- | -- |
| … | … |
| …. | -- |

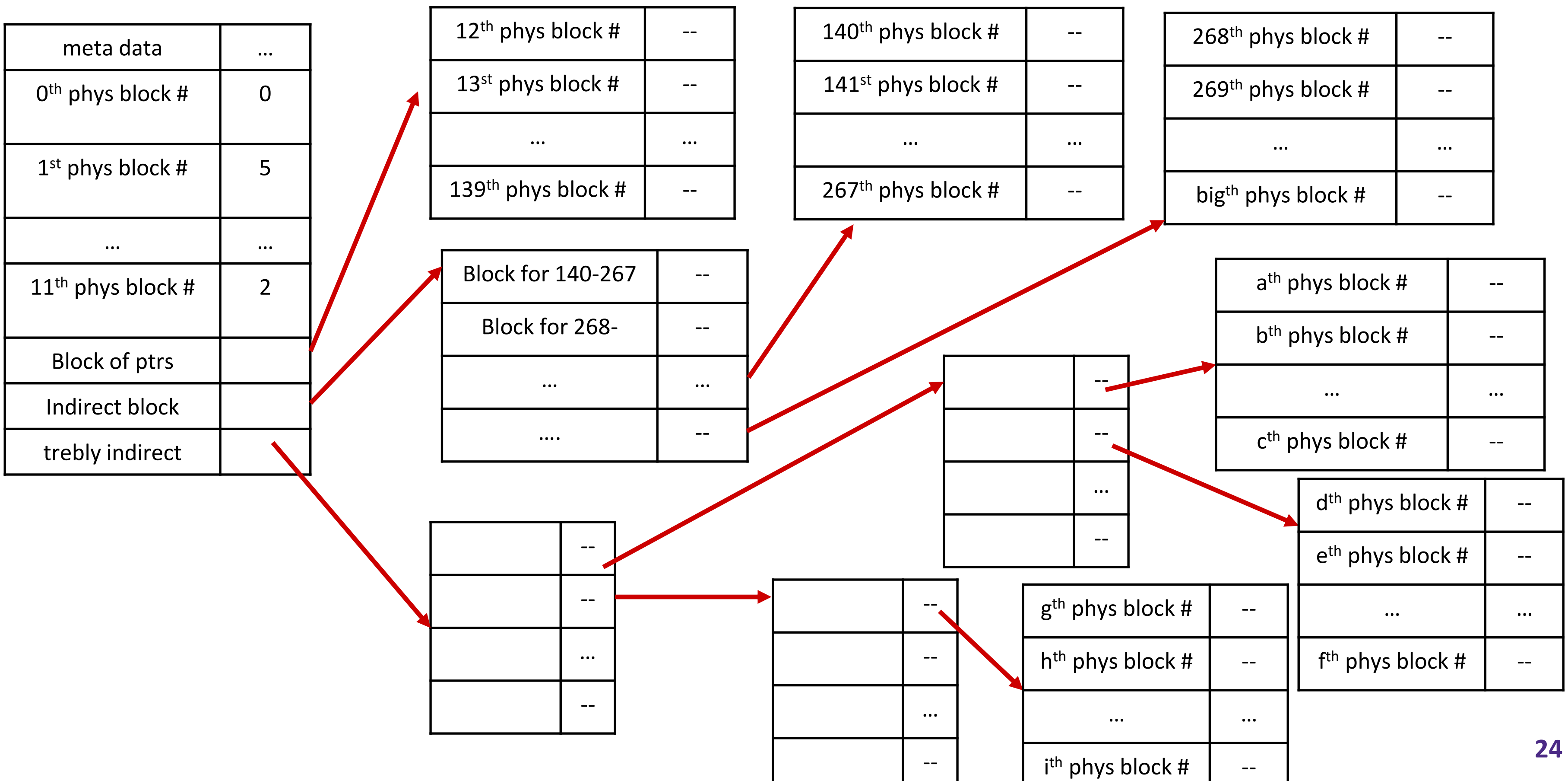
| | |
|---|---|
| $140^{th}$ phys block # | -- |
| $141^{st}$ phys block # | -- |
| … | … |
| $267^{th}$ phys block # | -- |

| | |
|---|---|
| $268^{th}$ phys block # | -- |
| $269^{th}$ phys block # | -- |
| … | … |
| $big^{th}$ phys block # | -- |

This block does NOT directly refer to blocks containing file data

# MORE MORE MORE MORE MORE MORE MOR

- What if our file needs more than that?
  - We can add another field to our Inode that refers to a pointer block that refers to pointer blocks that refer to data blocks…



# More?

- No more (at least on ext2)
- If you need more space than this, the operating system will tell you no

- Boon did the math on this: this is already enough for a file that is

$$(128 \times 512) + 10 \times 512\ Bytes$$
$$\left(128^2 \times 512\right) + (128 \times 512) + (10 \times 512)\ Bytes$$
$$\left(128^3 \times 512\right) + \left(128^2 \times 512\right) + (128 \times 512) + (10 \times 512)\ Bytes$$

- Big enough

# Poll Everywhere

pollev.com/tqm

- How is this better than FAT?

## Poll Everywhere

pollev.com/tqm

- How is this better than FAT?

- Inodes keep all the information of a file near each other
- if we wanted to store in memory only the information of open files, we could do that with les memory consumption

- In other words: only need to store in memory the inodes of the open files instead of the whole FAT

# Lecture Outline

- Inodes
- **Directories**
- Block Caching
- mmap & PennOS stuff

# Directory Entries with Inodes

- With FAT we said a directory entry had:
  - The file name
  - The number of the first block of the file

- With Inodes, we instead store the inode number for the file in the directory entry

# Reminder: Directories

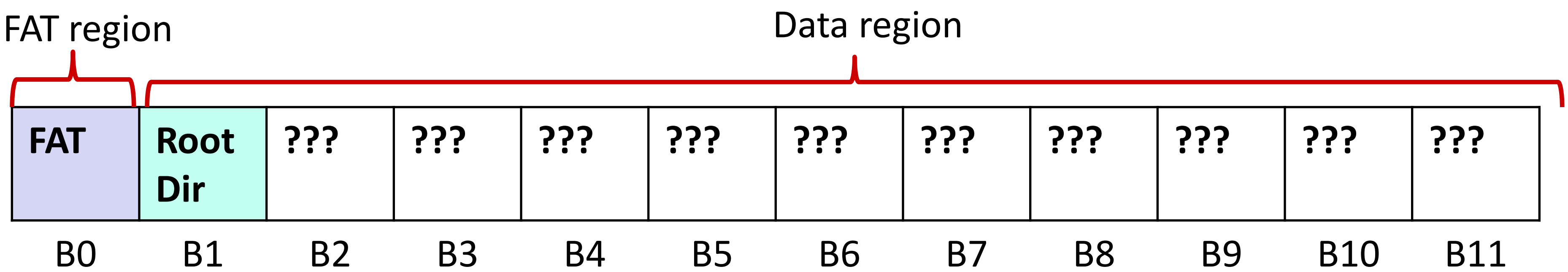
- A directory is essentially like a file
  - We will store its data on disk inside of blocks (like a file)

- 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, some metadata and…
    - If using Inodes, the inode for the file
    - If using FAT, the first block number of the file

  - I know we just said Inodes are better and more modern, but PennOS uses FAT so my examples will follow that, it is not much different for Inodes though

# Review: Directories

- In FAT our file system looked something like this:
  - 2 regions, and assuming FAT is just 1 block

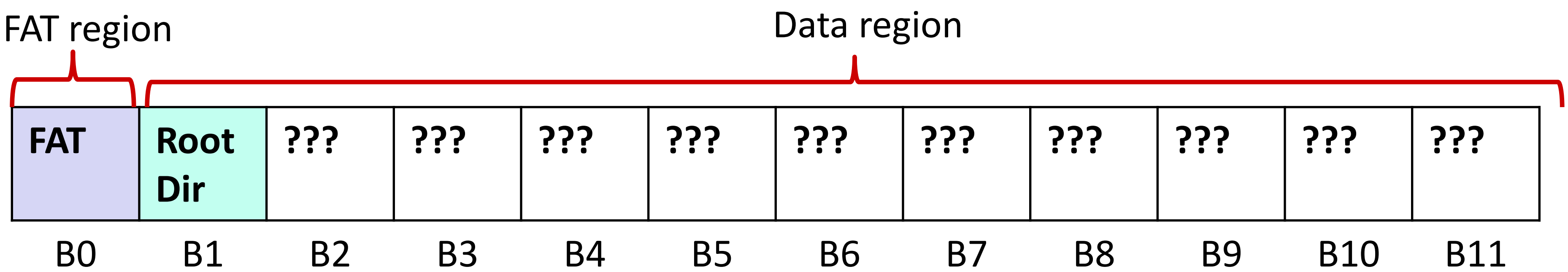



- And the root Directory contains a list of directory entries

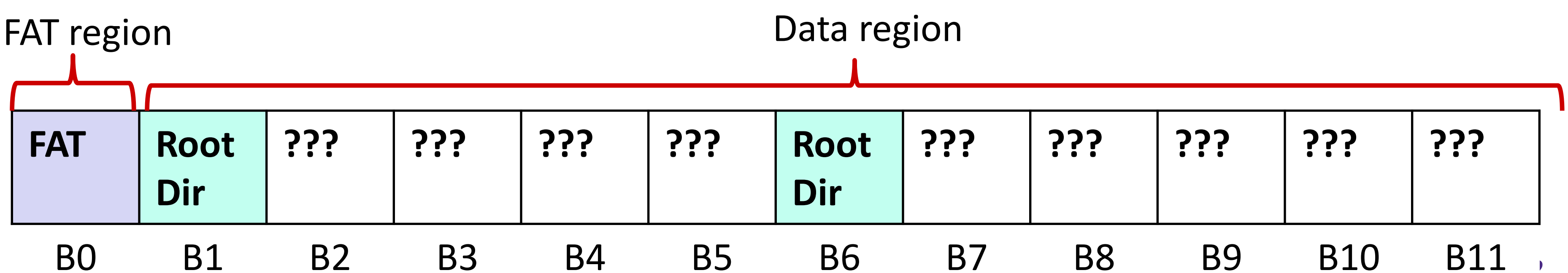
| File Name | Block Number |
|---|---|
| A | 7 |
| B | 4 |
| C | 9 |
| D | 2 |
| E | 10 |

# Growing a Directory

- In FAT our file system looked something like this:
  - 2 regions, and assuming FAT is just 1 block




- What happens if the root directory starts filling up?
  - The root directory is itself a file, it can expand to another block



# Growing a Directory

- We would also need to update the FAT to account for this change.
  - Root directory in PennFAT starts at index 1 into the data region
  - Index 1 into the data region is the first block in the data region 😖

| Block # (FAT Index) | Next (FAT value) |
|---|---|
| 0 | METADATA |
| 1 | END |
| … | … |
| …. | … |
| … | … |
| 6 | EMPTY |
| 7 | EMPTY |
| … | … |

→

| Block # (FAT Index) | Next (FAT value) |
|---|---|
| 0 | METADATA |
| 1 | 6 |
| … | … |
| …. | … |
| … | … |
| 6 | END |
| 7 | END |
| … | … |

# Question is not good format for pollev ☹

**Discuss**

- Let's say PennFAT is 4 blocks
- What are value of the remaining blocks in the diagram?

Root DIR

| File Name | Block Number |
|---|---|
| A | 7 |
| B | 2 |
| C | 6 |

FAT

| Block # (FAT Index) | Next (FAT value) |
|---|---|
| 0 | METADATA |
| 1 | 4 |
| 2 | 8 |
| 3 | END |
| 4 | END |
| 5 | EMPTY |
| 6 | END |
| 7 | END |
| 8 | 3 |
| … | … |

FAT region: B0–B3; Data region: B4–B11

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

# Question is not good format for pollev ☹

**Discuss**

- Let's say PennFAT is 4 blocks
- What are value of the remaining blocks in the diagram?

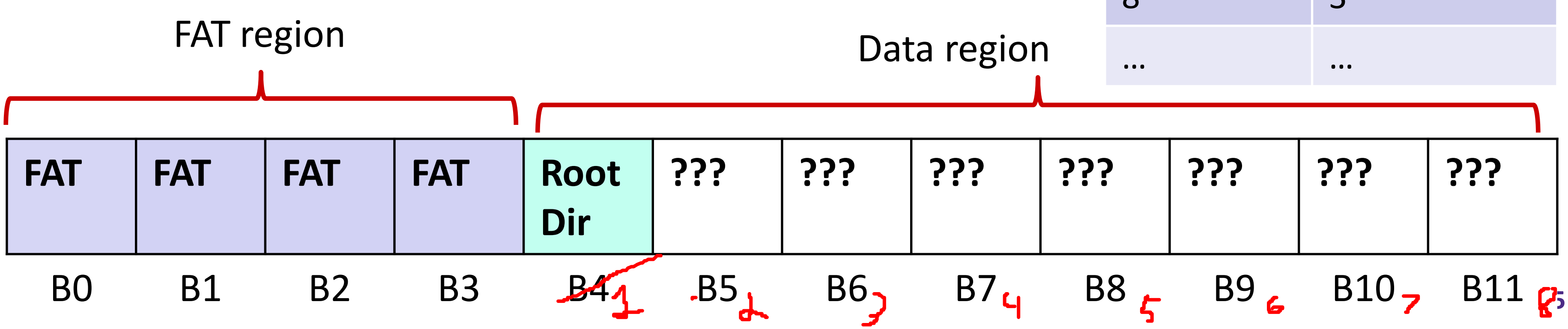
**Hint: Index into data region starting at index 1**

Root DIR

| **File Name** | **Block Number** |
|---|---|
| A | 7 |
| B | 2 |
| C | 6 |

FAT

| **Block # (FAT Index)** | **Next (FAT value)** |
|---|---|
| 0 | METADATA |
| 1 | 4 |
| 2 | 8 |
| 3 | END |
| 4 | END |
| 5 | EMPTY |
| 6 | END |
| 7 | END |
| 8 | 3 |
| … | … |



## Question is not good format for pollev ☹

**Discuss**

- Let's say PennFAT is 4 blocks

- What are value of the remaining blocks in the diagram?

**Hint: Index into data region starting at index 1**

Root DIR

| File Name | Block Number |
|---|---|
| A | 7 |
| B | 2 |
| C | 6 |

FAT

| Block # (FAT Index) | Next (FAT value) |
|---|---|
| 0 | METADATA |
| 1 | 4 |
| 2 | 8 |
| 3 | END |
| 4 | END |
| 5 | EMPTY |
| 6 | END |
| 7 | END |
| 8 | 3 |
| … | … |

FAT region: B0–B3; Data region: B4–B11

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

# Question is not good format for pollev ☹

**Discuss**

- Let's say PennFAT is 4 blocks
- What are value of the remaining blocks in the diagram?

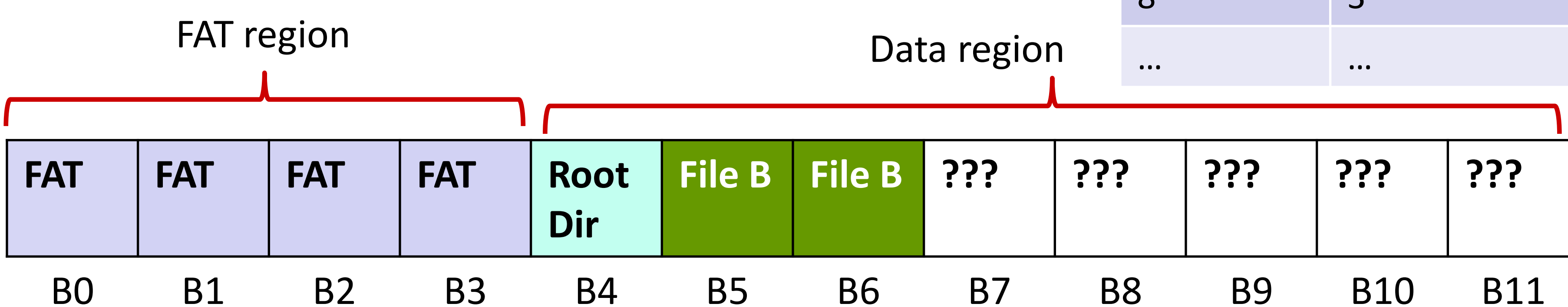
**Hint: Index into data region starting at index 1**

Root DIR

| File Name | Block Number |
|---|---|
| A | 7 |
| B | 2 |
| C | 6 |

FAT

| Block # (FAT Index) | Next (FAT value) |
|---|---|
| 0 | METADATA |
| 1 | 4 |
| 2 | 8 |
| 3 | END |
| 4 | END |
| 5 | EMPTY |
| 6 | END |
| 7 | END |
| 8 | 3 |
| … | … |



## Question is not good format for pollev ☹

**Discuss**

- Let's say PennFAT is 4 blocks
- What are value of the remaining blocks in the diagram?

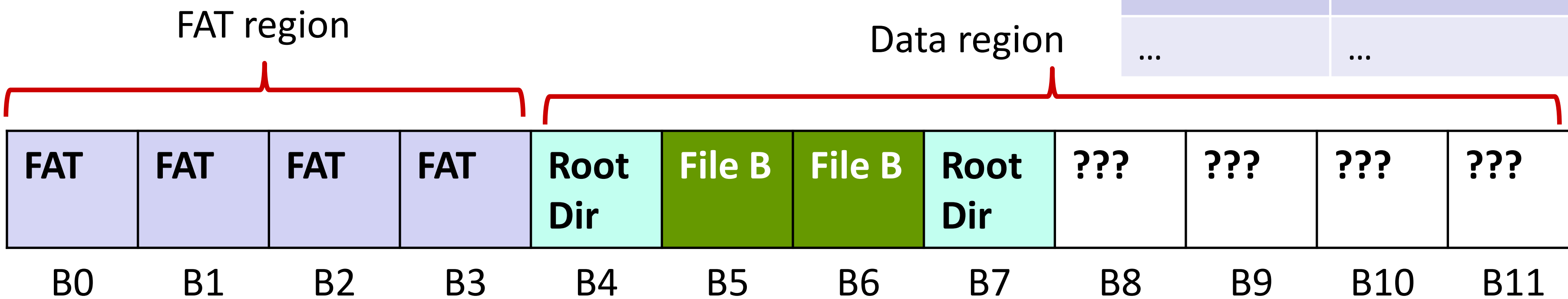
**Hint: Index into data region starting at index 1**

Root DIR

| File Name | Block Number |
|---|---|
| A | 7 |
| B | 2 |
| C | 6 |

FAT

| Block # (FAT Index) | Next (FAT value) |
|---|---|
| 0 | METADATA |
| 1 | 4 |
| 2 | 8 |
| 3 | END |
| 4 | END |
| 5 | EMPTY |
| 6 | END |
| 7 | END |
| 8 | 3 |
| … | … |



# Question is not good format for pollev ☹

**Discuss**

- Let's say PennFAT is 4 blocks
- What are value of the remaining blocks in the diagram?

**Hint: Index into data region starting at index 1**

Root DIR

| File Name | Block Number |
|---|---|
| A | 7 |
| B | 2 |
| C | 6 |

FAT

| Block # (FAT Index) | Next (FAT value) |
|---|---|
| 0 | METADATA |
| 1 | 4 |
| 2 | 8 |
| 3 | END |
| 4 | END |
| 5 | EMPTY |
| 6 | END |
| 7 | END |
| 8 | 3 |
| … | … |

FAT region: B0–B3; Data region: B4–B11

| B0 | B1 | B2 | B3 | B4 | B5 | B6 | B7 | B8 | B9 | B10 | B11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| FAT | FAT | FAT | FAT | Root Dir | File B | File B | Root Dir | EMPTY | ??? | ??? | ??? |

# Question is not good format for pollev ☹

**Discuss**

- Let's say PennFAT is 4 blocks
- What are value of the remaining blocks in the diagram?

**Hint: Index into data region starting at index 1**

FAT

| Block # (FAT Index) | Next (FAT value) |
|---|---|
| 0 | METADATA |
| 1 | 4 |
| 2 | 8 |
| 3 | END |
| 4 | END |
| 5 | EMPTY |
| 6 | END |
| 7 | END |
| 8 | 3 |
| … | … |

Root DIR

| File Name | Block Number |
|---|---|
| A | 7 |
| B | 2 |
| C | 6 |

FAT region (B0–B3) | Data region (B4–B11)

| B0 | B1 | B2 | B3 | B4 | B5 | B6 | B7 | B8 | B9 | B10 | B11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| FAT | FAT | FAT | FAT | Root Dir | File B | File B | Root Dir | EMPTY | File C | ??? | ??? |

## Question is not good format for pollev ☹

**Discuss**

- Let's say PennFAT is 4 blocks
- What are value of the remaining blocks in the diagram?

**Hint: Index into data region starting at index 1**

Root DIR

| File Name | Block Number |
|---|---|
| A | 7 |
| B | 2 |
| C | 6 |

FAT

| Block # (FAT Index) | Next (FAT value) |
|---|---|
| 0 | METADATA |
| 1 | 4 |
| 2 | 8 |
| 3 | END |
| 4 | END |
| 5 | EMPTY |
| 6 | END |
| 7 | END |
| 8 | 3 |
| … | … |

FAT region (B0–B3) · Data region (B4–B11)

| B0 | B1 | B2 | B3 | B4 | B5 | B6 | B7 | B8 | B9 | B10 | B11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| FAT | FAT | FAT | FAT | Root Dir | File B | File B | Root Dir | EMPTY | File C | File A | ??? |

# Question is not good format for pollev ☹

**Discuss**

- Let's say PennFAT is 4 blocks
- What are value of the remaining blocks in the diagram?

**Hint: Index into data region starting at index 1**

Root DIR

| File Name | Block Number |
|---|---|
| A | 7 |
| B | 2 |
| C | 6 |

FAT

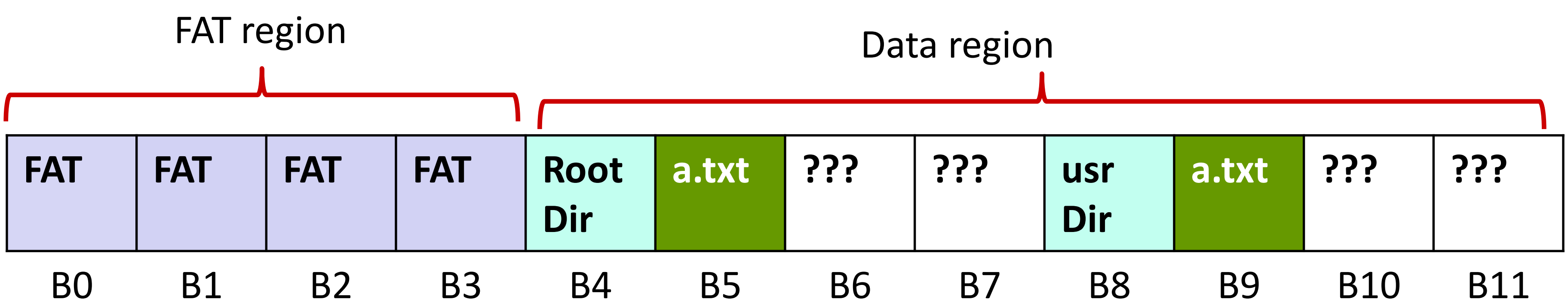
| Block # (FAT Index) | Next (FAT value) |
|---|---|
| 0 | METADATA |
| 1 | 4 |
| 2 | 8 |
| 3 | END |
| 4 | END |
| 5 | EMPTY |
| 6 | END |
| 7 | END |
| 8 | 3 |
| … | … |

FAT region (B0–B3) | Data region (B4–B11)

| B0 | B1 | B2 | B3 | B4 | B5 | B6 | B7 | B8 | B9 | B10 | B11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| FAT | FAT | FAT | FAT | Root Dir | File B | File B | Root Dir | EMPTY | File C | File A | File B |

# Sub Directories

- In PennOS, we are only required to deal with 1 directory, but you can implement sub-directories.
  - Sub directories are just other (special) files
- Consider we have the following two directories and files
  - /a.txt
  - /usr/a.txt
  - Above are two separate files!



# Sub Directories

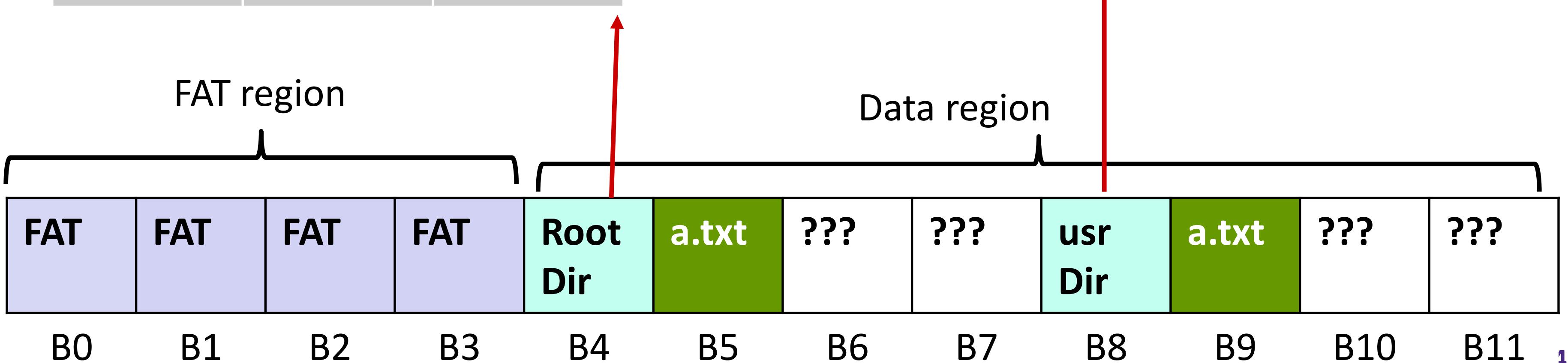
- We would also have some information in a directory entry to specify what kind of file it is

Root DIR

| File Name | Block Number | File Type |
|---|---|---|
| a.txt | 2 | Regular |
| usr/ | 5 | directory |
| … | .. | |

usr DIR

| File Name | Block Number | File Type |
|---|---|---|
| a.txt | 6 | Regular |
| … | .. | |



# . and ..

- It would be useful to support . and . .
  - . Refers to the current directory, . . refers to parent directory

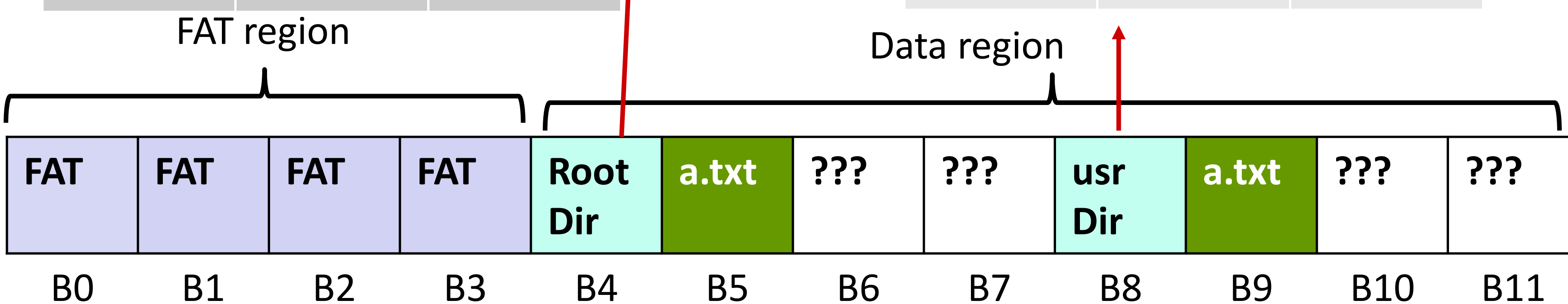
root DIR

| File Name | Block Number | File Type |
|---|---|---|
| . | 1 | directory |
| .. | 1 | directory |
| a.txt | 2 | Regular |
| usr/ | 5 | directory |
| ... | .. | |

Has no parent, refers to self

usr DIR

| File Name | Block Number | File Type |
|---|---|---|
| . | 5 | directory |
| .. | 1 | directory |
| a.txt | 6 | Regular |
| ... | .. | |



# Lecture Outline

- Inodes
- Directories
- **Block Caching**
- mmap & PennOS stuff

# Block Caching

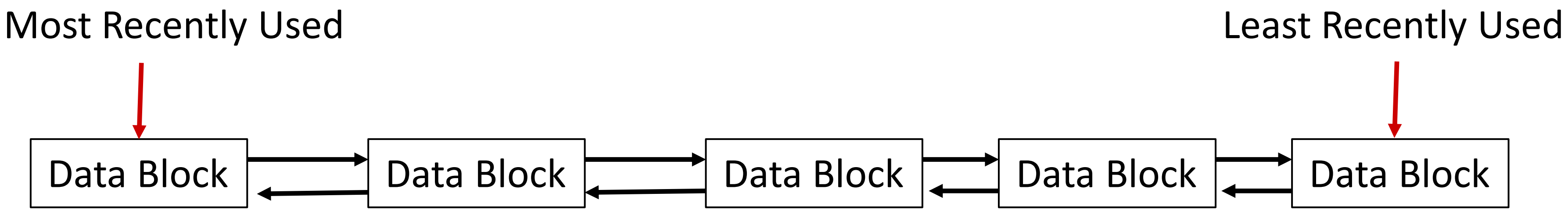
- Disk I/O is really slow (relative to accessing memory)

- What can we do instead to make it faster?
  - Keep data that we want to access in memory ☺
  - We already did this with FAT and Inodes for open files

- We can do the same for data blocks we think we may use again in the future

# Block Caching Data Structure

- We can use a linked list to store blocks in LRU

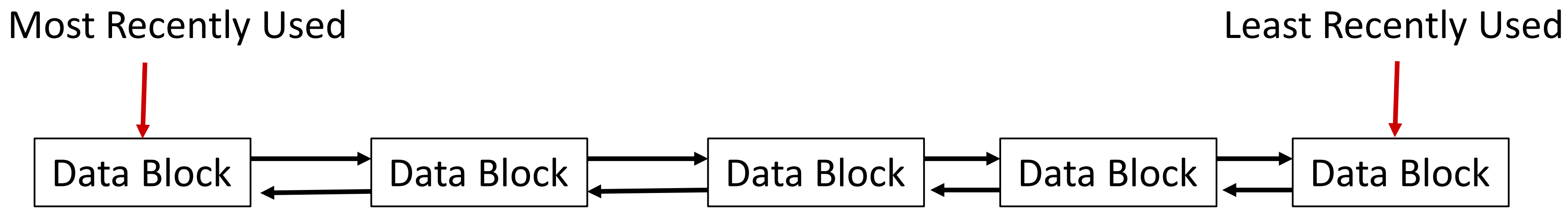



- What is the algorithmic runtime analysis to:
  - lookup a specific block?
  - Removal time?
  - Time to move a block to the front or back?

Discuss

# Block Caching Data Structure

- We can use a linked list to store blocks in LRU



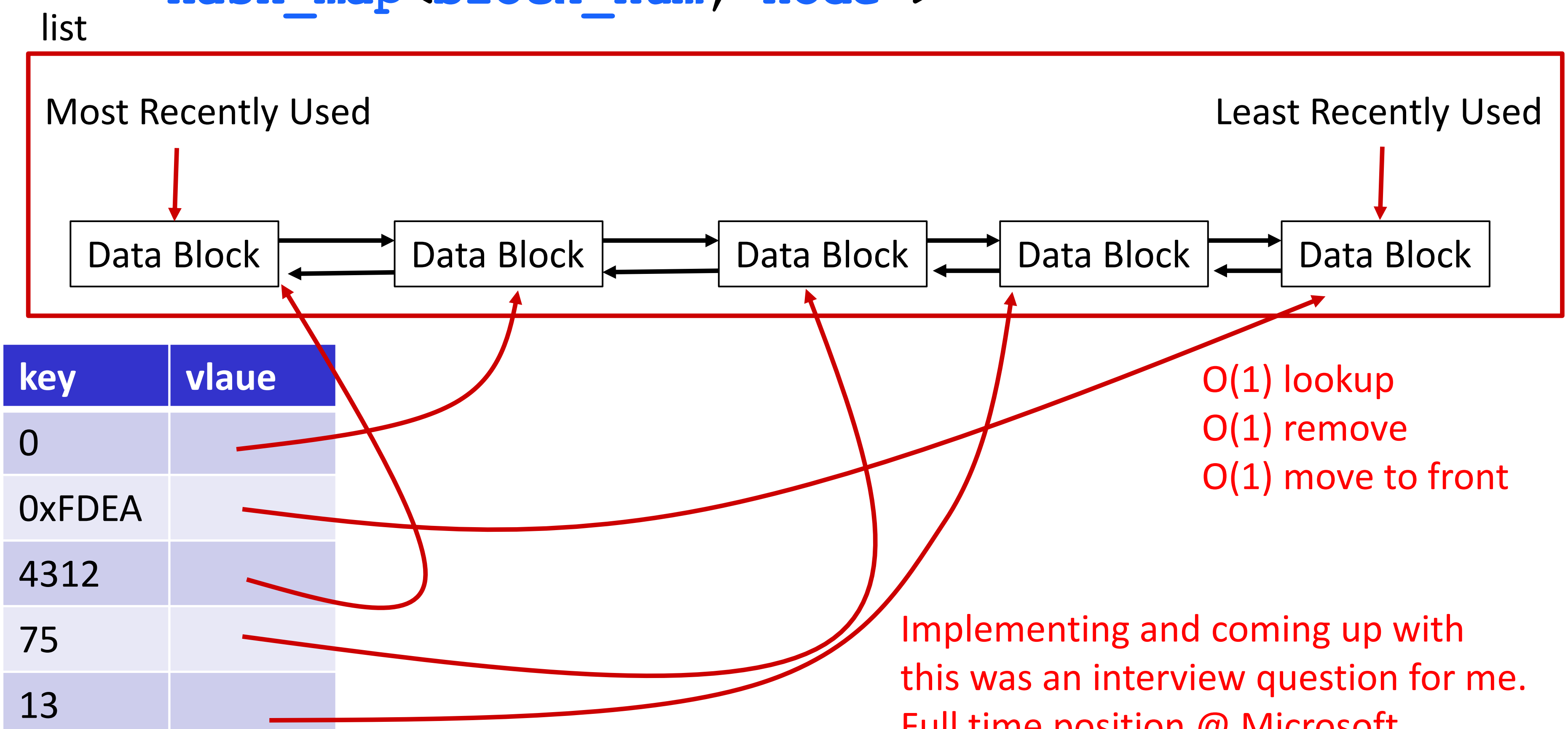

- What is the algorithmic runtime analysis to:
  - lookup a specific block? O(n)
  - Removal time? O(1)
  - Time to move a block to the front or back? O(1)

Discuss

Is there a structure we know of that has O(1) lookup time?

# Chaining Hash Map

- We can use a combination of two data structures:
  - `linked_list<block>`
  - `hash_map<block_num, node*>`




O(1) lookup
O(1) remove
O(1) move to front

Implementing and coming up with this was an interview question for me. Full time position @ Microsoft

# Lecture Outline

- Inodes
- Directories
- Block Caching
- **mmap & PennOS stuff**

# mmap

```
void* mmap(void* addr, size_t length, int prot,
           int flags, int fd, off_t offset);
```

- Maps part of a virtual address space of a calling process. This mapping could be to a file, so reading/writing to memory also updates the file.
  - **addr**: Hint at the address to create the mapping at. Use **NULL** to let linux kernel decide for you
  - **length**: the length of the mapping
  - **prot**: desired memory protection (readable, writable, etc.)
  - **flags**: specify attributes of the mapping, we will use **MAP_SHARED**
  - **fd**: the file we want to map into memory
  - **offset**: the offset we want to start at in the file must be a multiple of the page size (we will use 0)

# mmap demo & Pennos Test FS

```
void* mmap(void* addr, size_t length, int prot,
           int flags, int fd, off_t offset);
```

- `mmap.c` loads a fs FAT with **`mmap`** and prints the first FAT entry
  - Note: when we try to read from fakefs (empty) we get an error
  - You may want to use **`read`** or **`fread`** to create the fat

- On the website in a zip file:
  - minfs fat_blocks = 1 block_size = 512
  - testfs: fat_blocks = 1 block_size = 256
  - maxfs: fat_blocks = 32 block_size = 4096

- You should be able to download these and use these to test your fat implementation to some extent

# PennOS FAT clarification

- The specification says:
  - **If a <u>user level program</u> is calling read(2), then you are doing something wrong.**

- Your PennFAT implementation can use `read()`, **mmap()** etc for implementing the file system
  - We are using a file on the host operating system as our storage medium (as a fake disk)

- PennOS users should only call your user level functions, like `f_read()`, they should not interact with our FAT or “disk” directly

# Common Mistakes/Questions in PennOS

- why do we need to implement process related things like kill and fork. Can we call the linux things?
  - Answer: we can't do that since we are working with ucontext to mimic processes. If we called kill or fork, it would affect/duplicate the entire PennOS Process
  - **Calling fork and similar functions will get you a ZERO**

- Be prepared for race conditions:
  - you may enter the scheduler cooperatively instead of an alarm. Being in the scheduler can be interrupted by an ALARM
  - Be careful and away this may happen
  - alarm handlers are in their own temporary context, which makes things weird

# Common Mistakes/Questions in PennOS

- If you are splitting up work: **don't integrate too late**

- Be sure to upload the abstraction user vs kernel
  - it cuts off a percentage of the points if you do, not a flat deduction. Can particularly affect your grade.

- Don’t leave companion document till the end
  - If you can, try to use doxygen, it saves a lot of time