FAT, I-nodes

Computer Operating Systems, Spring 2024

Instructor: Travis McGaha

Head TAs: Nate Hoaglund & Seungmin Han

TAs:

Adam Gorka	Haoyun Qin	Kyrie Dowling	Ryoma Harris
Andy Jiang	Jeff Yang	Oliver Hendrych	Shyam Mehta
Charis Gao	Jerry Wang	Maxi Liu	Tom Holland
Daniel Da	Jinghao Zhang	Rohan Verma	Tina Kokoshvili
Emily Shen	Julius Snipes	Ryan Boyle	Zhiyan Lu

Poll Everywhere

pollev.com/tqm

- How is milestone 1 looking?

Administrivia

- Penn-shell is out!
  - **Full thing is due at the end of the week** (2/23 @ 11:59 pm)
  - Done in partners
  - Should have everything you need to complete the assignment in this class
  - Please add your partner to the gradescope submission if you can.
  - Autograder for full thing should be up today

Administrivia

- Midterm booked:
  - 5:15 - 7:15 pm in Meyerson B1
  - Thursday 2/29 (the Thursday before break)
  - Let me know if you conflicts

- Final Tentatively Booked
  - Tuesday May 7th, Noon – 2pm in Towne 100
  - Not confirmed yet, but this is likely it

- Travis is still a little sick, but probably be in-person for next lecture

Penn-Shell Compatibility

- From the signal(2) man page

```
Portability
    The only portable use of signal() is to set a signal's disposition to SIG_DFL or SIG_IGN.  The semantics when using signal()
    to establish a signal handler vary across systems (and POSIX.1 explicitly permits this variation); do not use  it  for  this
    purpose.
```

- If you want to have better help from TA's put this at the top of your file before you #include anything
  - This *should* get signals to behave as we expect, so TAs can better help
  - If you got it working another way, that is OK. Auto-grader *should* still accept it

```
#ifndef _POSIX_C_SOURCE
#define _POSIX_C_SOURCE 200809L
#endif

#ifndef _DEFAULT_SOURCE
#define _DEFAULT_SOURCE 1
#endif
```

Poll Everywhere

pollev.com/tqm

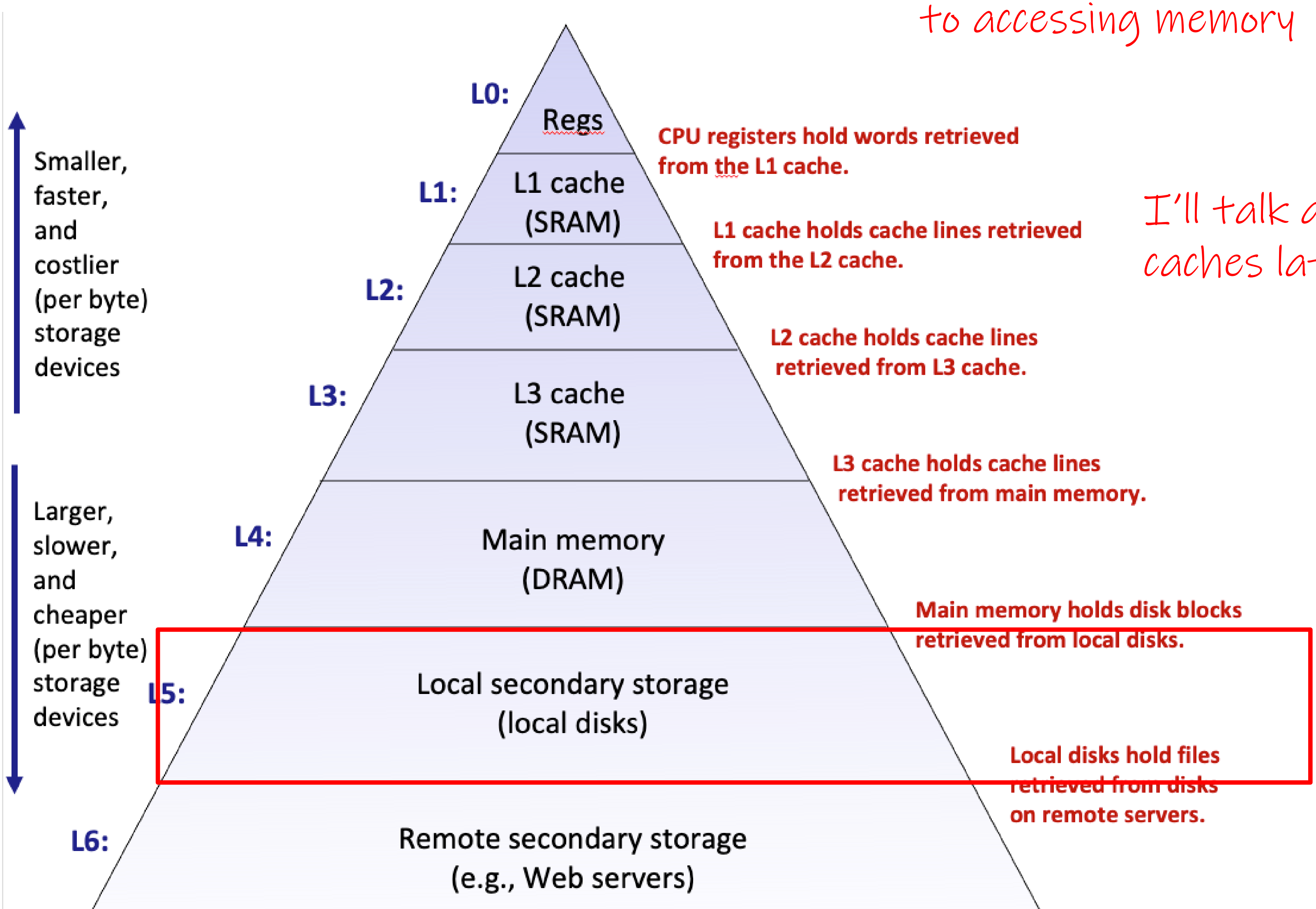
- How are you doing?

Lecture Outline

- **FAT & PennFAT wrap-up**
- Inodes
- Directories
- Block Caching

Memory Hierarchy

Files systems are really really really slow compared to accessing memory

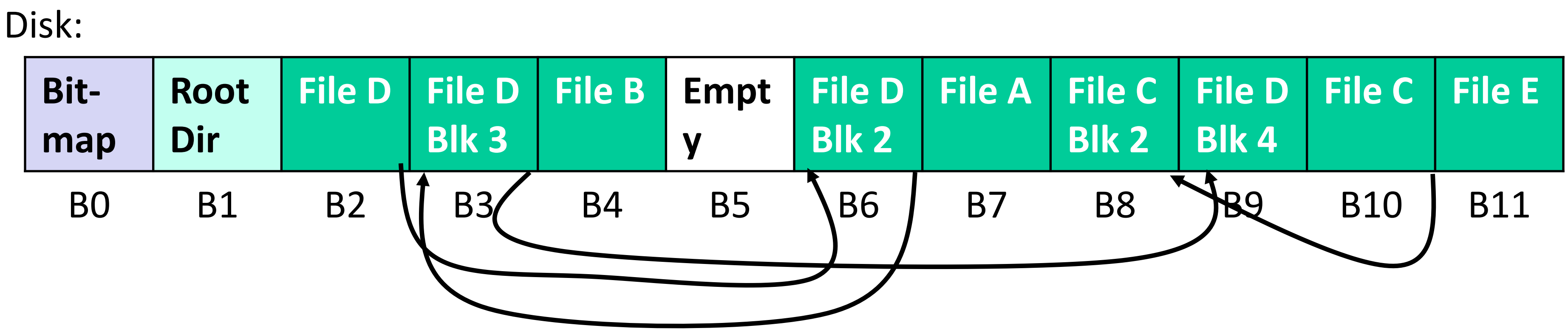


I'll talk about caches later

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

FAT (File Allocation Table)

- Instead of this:



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

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 ☺

FAT (File Allocation Table)

- This table is called the File Allocation Table (FAT)
- This table is in memory 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

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

Disk:

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 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

Disk:

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

Disk:

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

Disk:

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 4^{th} 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

Disk:

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

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

Disk:

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

Poll Everywhere

pollev.com/tqm

- What 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?

Poll Everywhere

pollev.com/tqm

- What 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?

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 is really big and is in memory, so memory consumption goes up ☹

FAT 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 bitmap anymore!

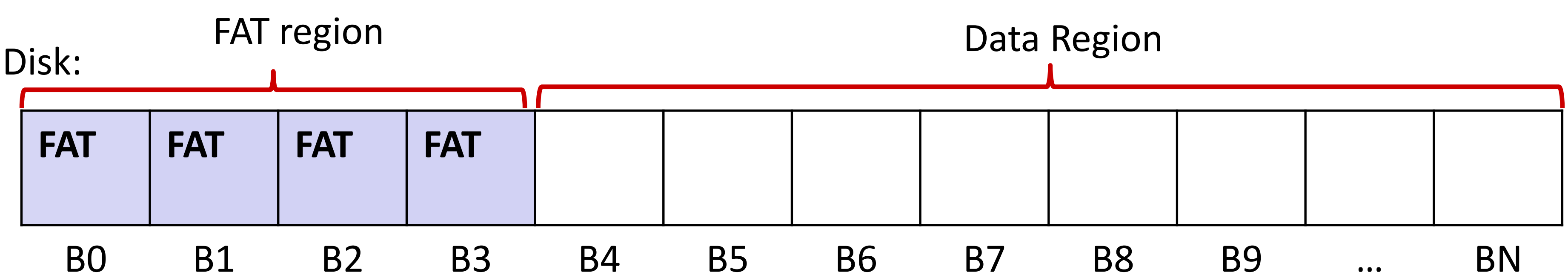
- Grows a lot as the size of disk grows
  - As the disk grows, there are more blocks in the disk. We need more FAT entries, and each entry needs more bits. (To hold the block number. # of bits for block # grows to support more blocks)
  - **A FAT may be bigger than one block**
  - Since we need to keep the FAT in memory, this increases our memory consumption as well
  - FAT got fazed out for I-nodes (next lecture) because of this

Poll Everywhere

pollev.com/tqm

- When you create a file system with PennFAT, you specify the number of blocks the FAT (this is just the 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 a FAT entry is 2 bytes.
  - How many entries do I have?
  - How many Blocks do we have that can store actual file data?

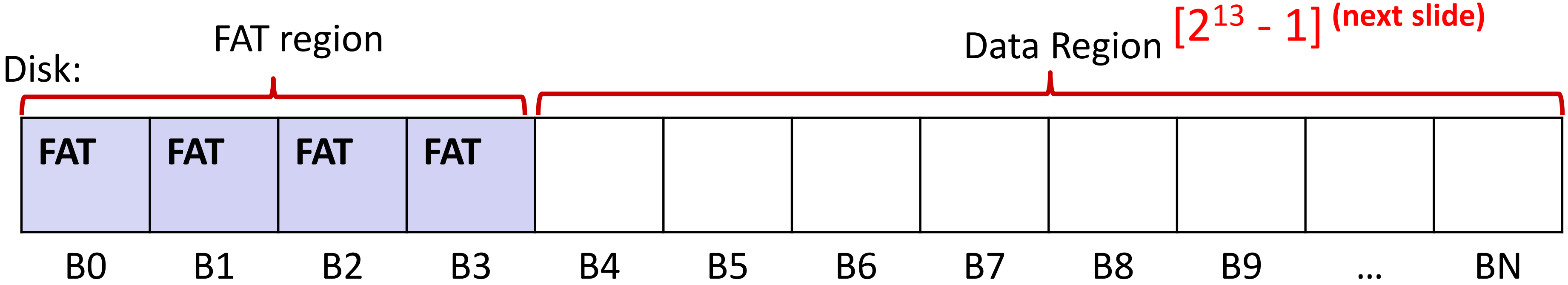


Poll Everywhere

pollev.com/tqm

- When you create a file system with PennFAT, you specify the number of blocks the FAT (this is just the 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 a FAT entry is 2 bytes.
  - How many entries do I have? $4 * 2^{12} / 2 = [2^{13}]$
  - How many Blocks do we have that can store actual file data? $[2^{13} - 1]$ **(next slide)**



PennOS FAT Details

- If we have N entries in the FAT, we only have N – 1 blocks in the FAT

- The first FAT entry `FAT[0]` holds meta data about the FAT, so it doesn't correspond to a "real" block

- An entry is 16-bits, which is 2 bytes.
- 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

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 number of blocks in the FAT
  - 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

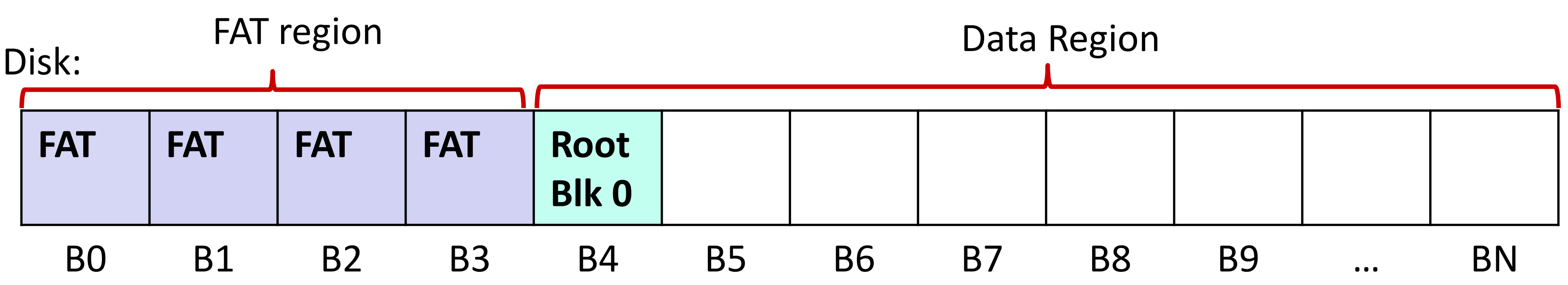
LSB	Block Size
0	256
1	512
2	1,024
3	2,048
4	4,096

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)



Lecture Outline

- FAT & PennFAT wrap-up
- **Inodes**
- Directories
- Block Caching

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

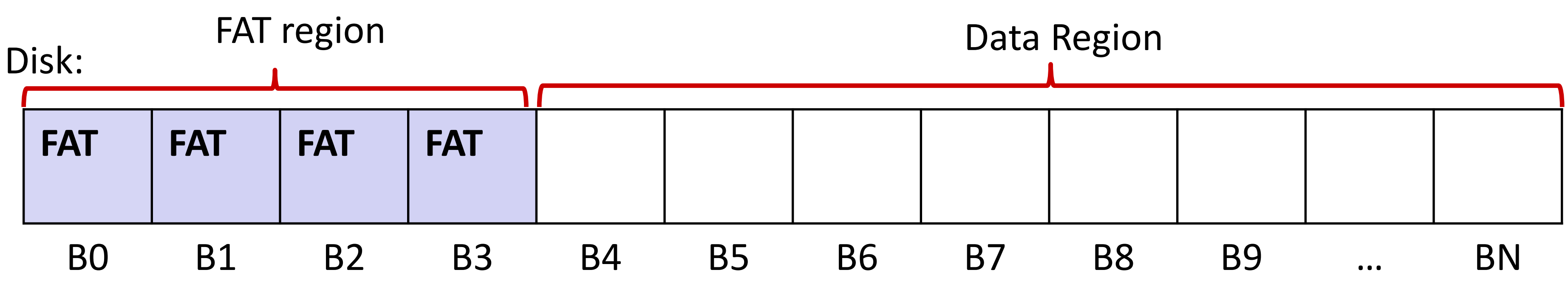
pollev.com/tqm

- 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)**

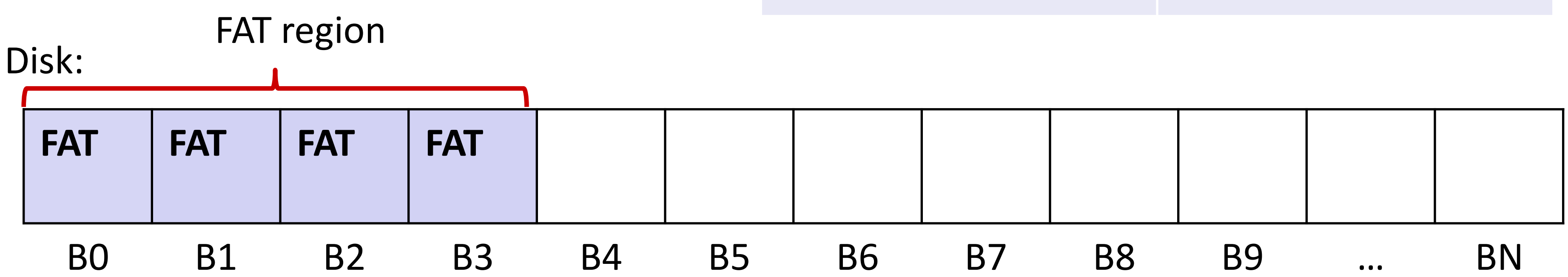


Explanation

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	



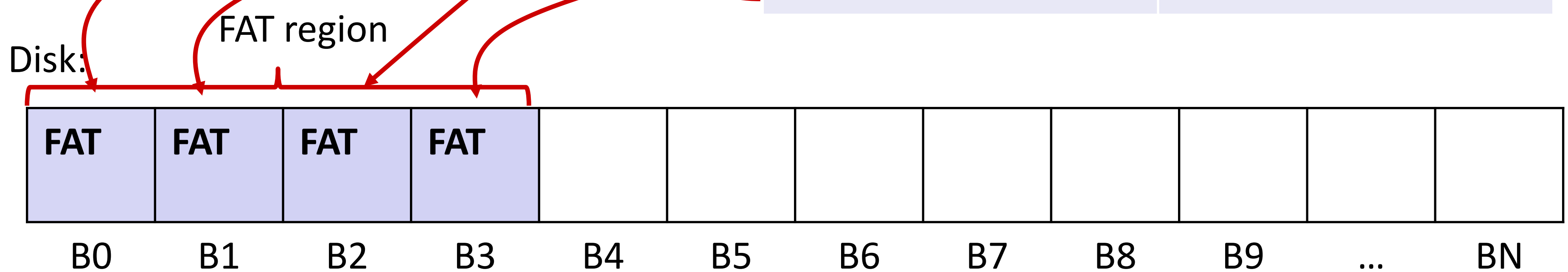
Explanation

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:

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

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

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)

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

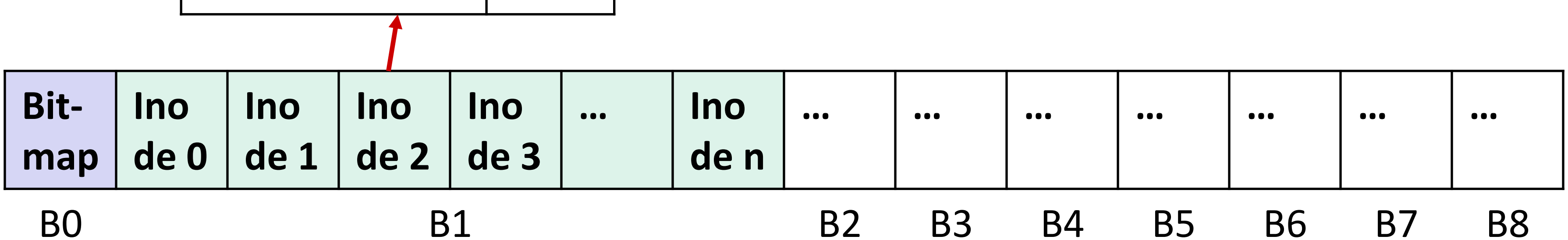
Example File Block Lookup

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

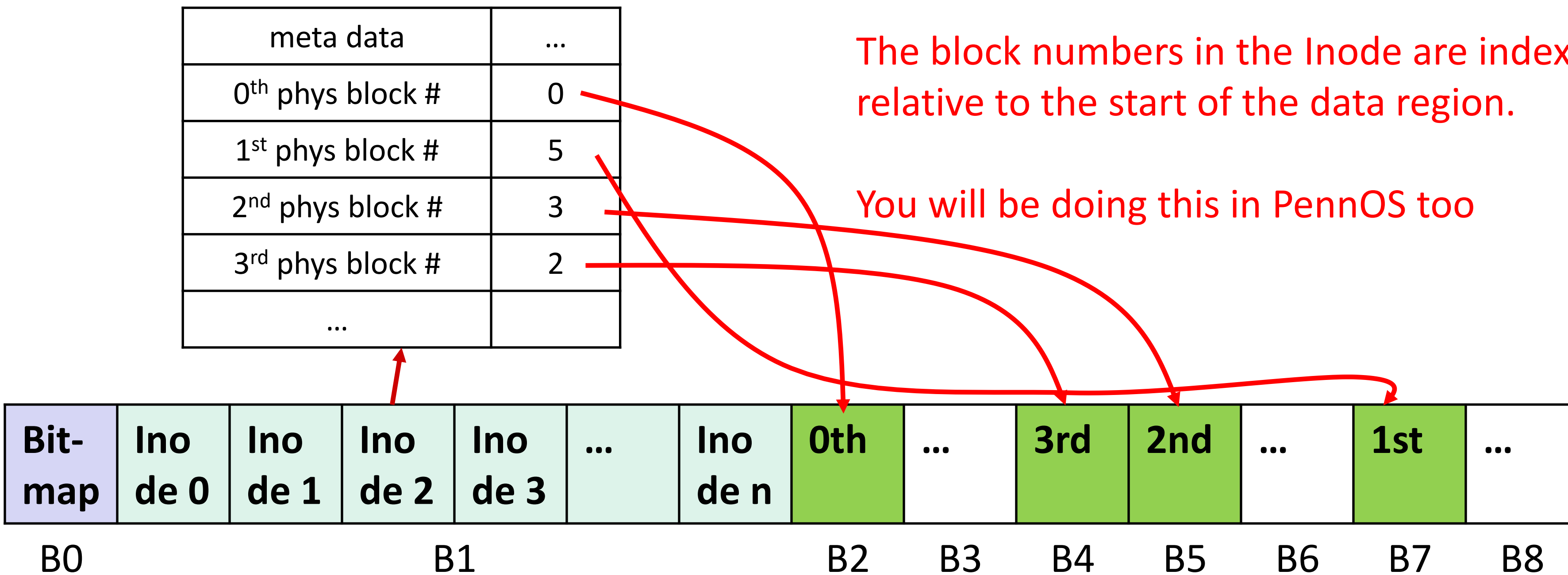
The block numbers in the Inode are indexes relative to the start of the data region.

You will be doing this in PennOS too



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



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

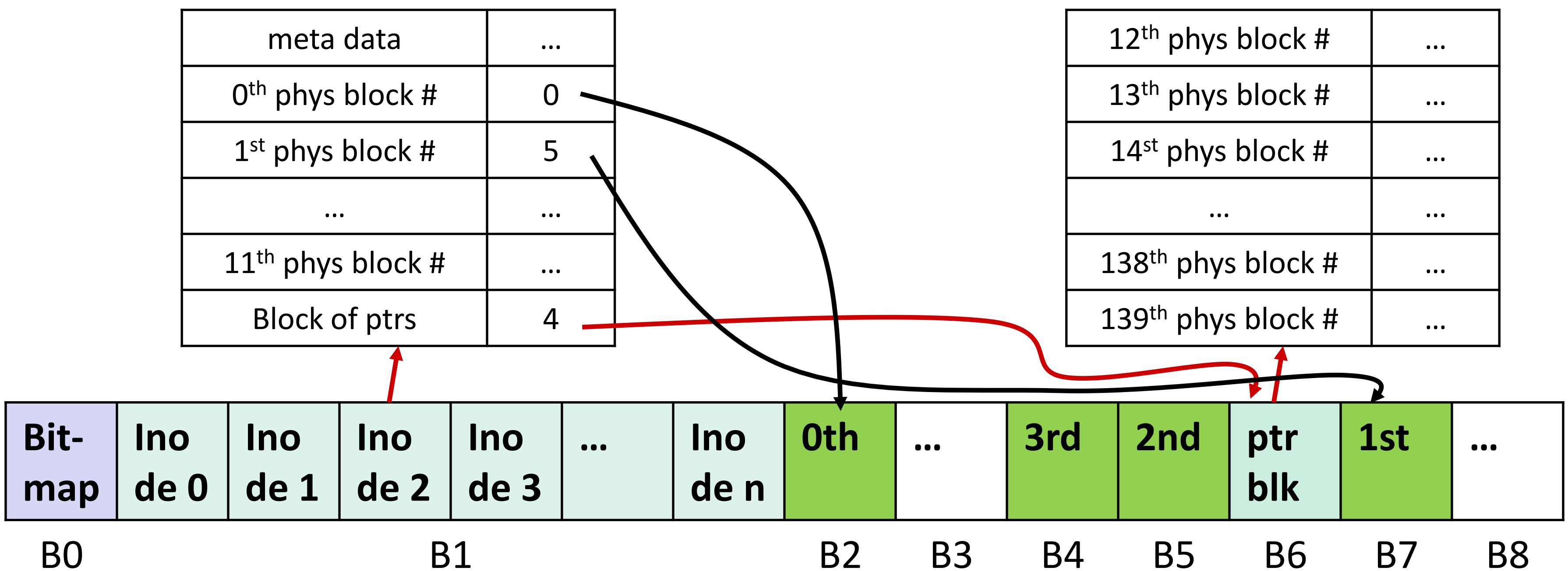
- So with Inodes, how many blocks can we have per file?
  - So far: 12 blocks per file (this is not enough, way too small!
- We can allocate a block to hold more block numbers

```
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

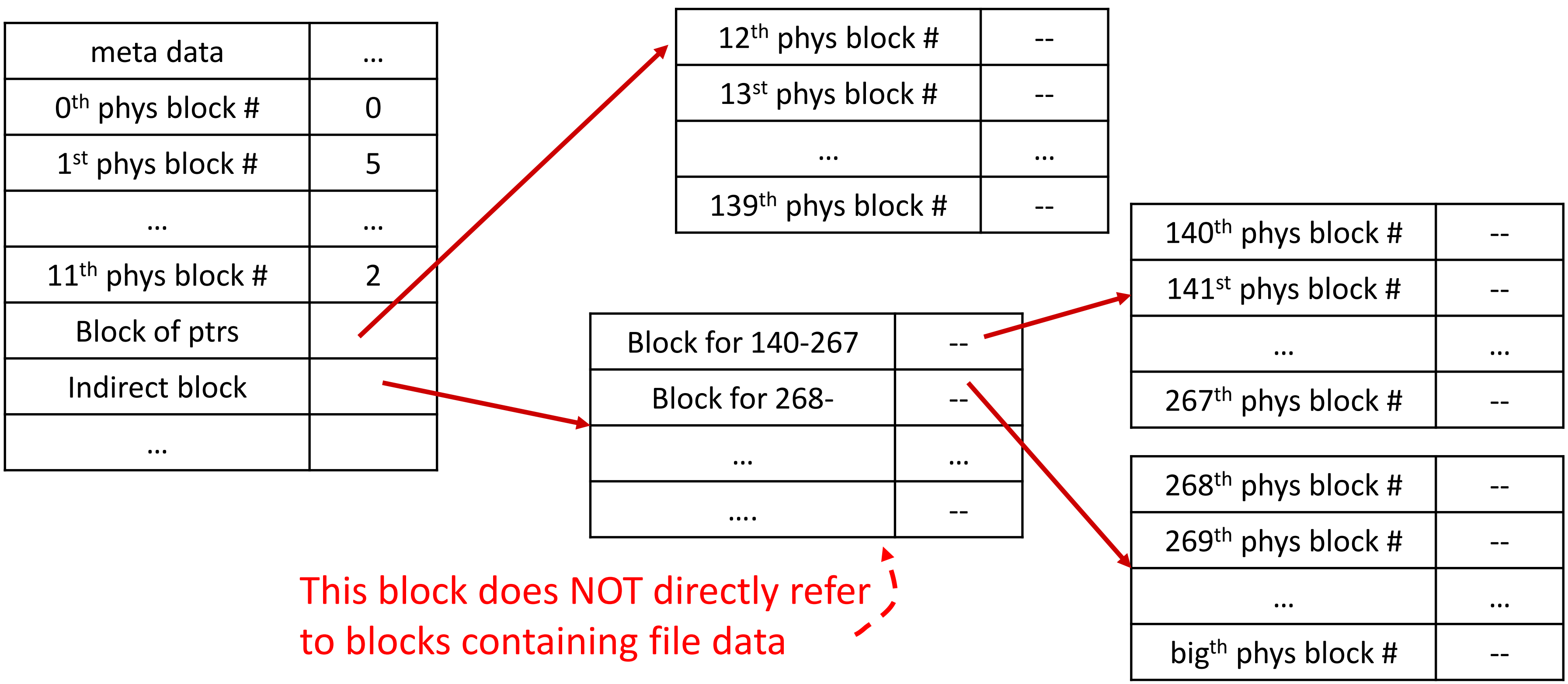
- So with Inodes, how many blocks can we have per file?
  - So far: 12 blocks per file (this is not enough, way too small!

- We can allocate a block to hold more block numbers



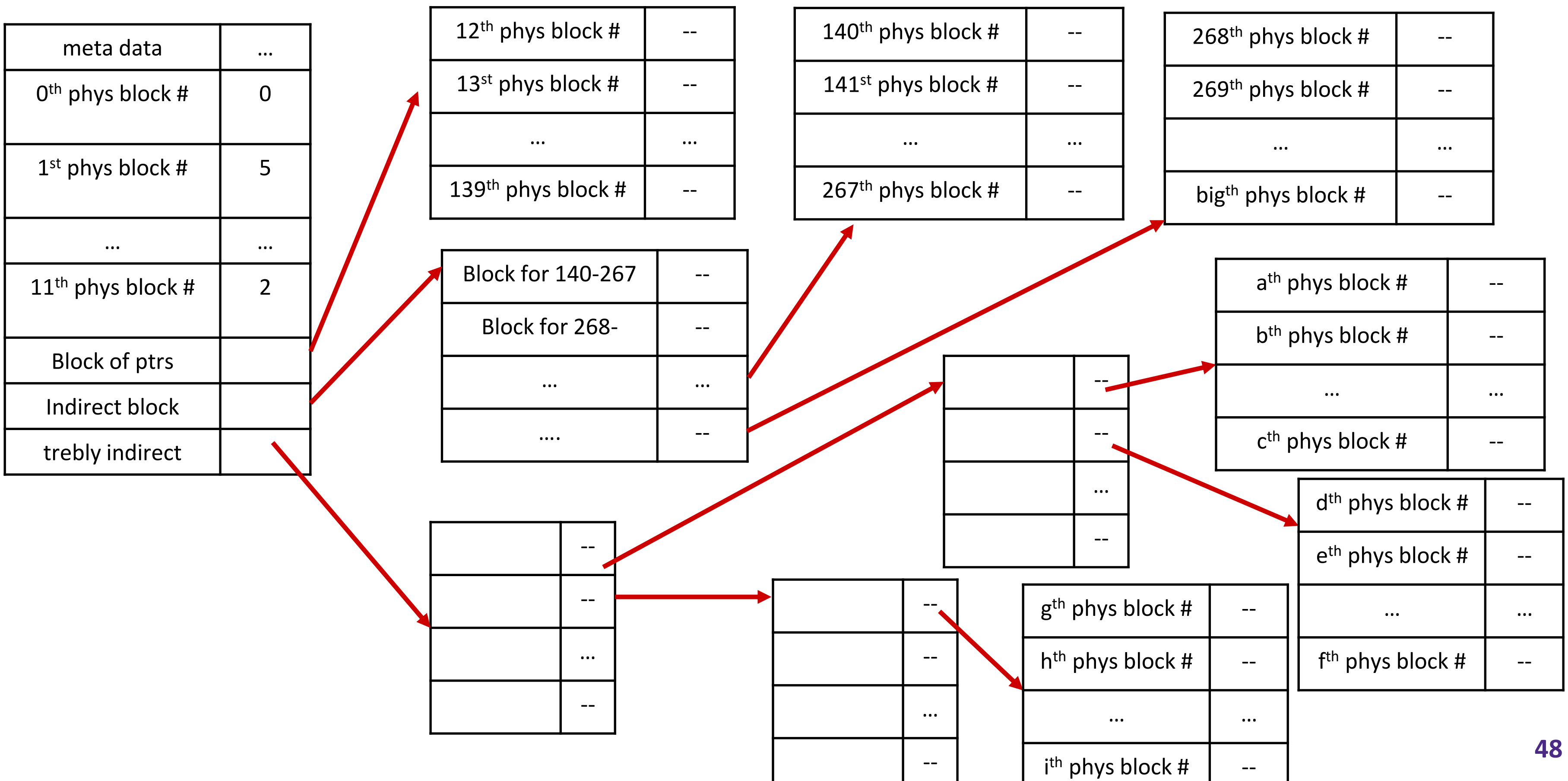
We need moreeeeee

- 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



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

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

- 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

- FAT & PennFAT wrap-up
- Inodes
- **Directories**
- Block Caching

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

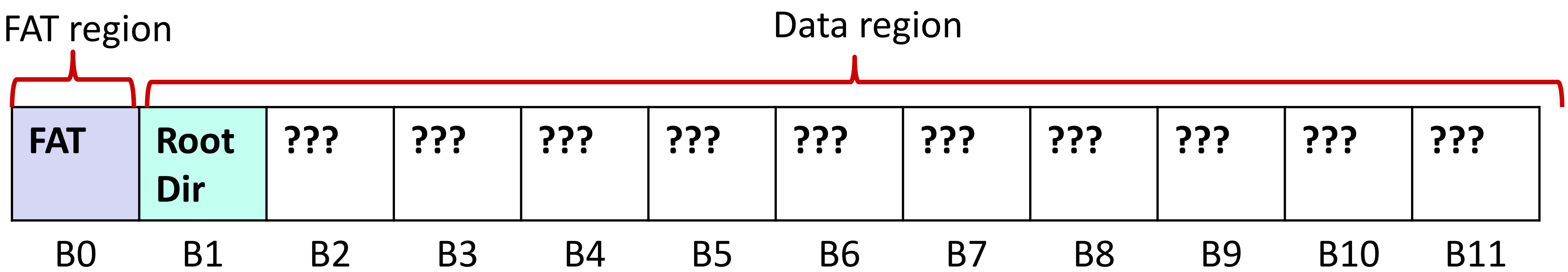
- A directory is essentially like a file
  - We will store its data on disk inside of blocks (like a file)

- 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

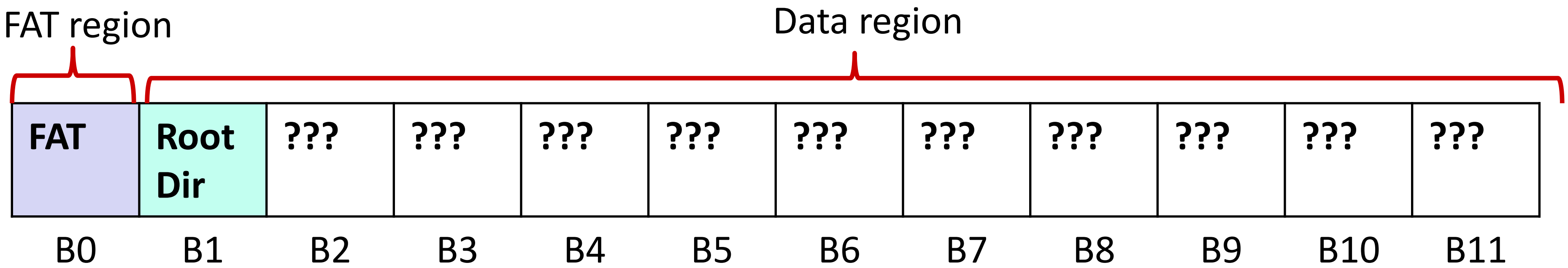


- And the root Directory contains a list of directory entries

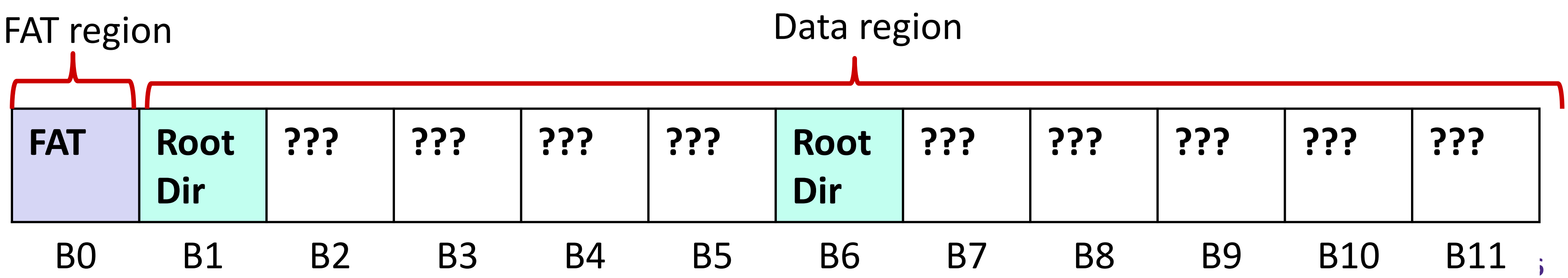
File Name	Block Number
A	7
B	4
C	9
D	2
E	10

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	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?

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)

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

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?

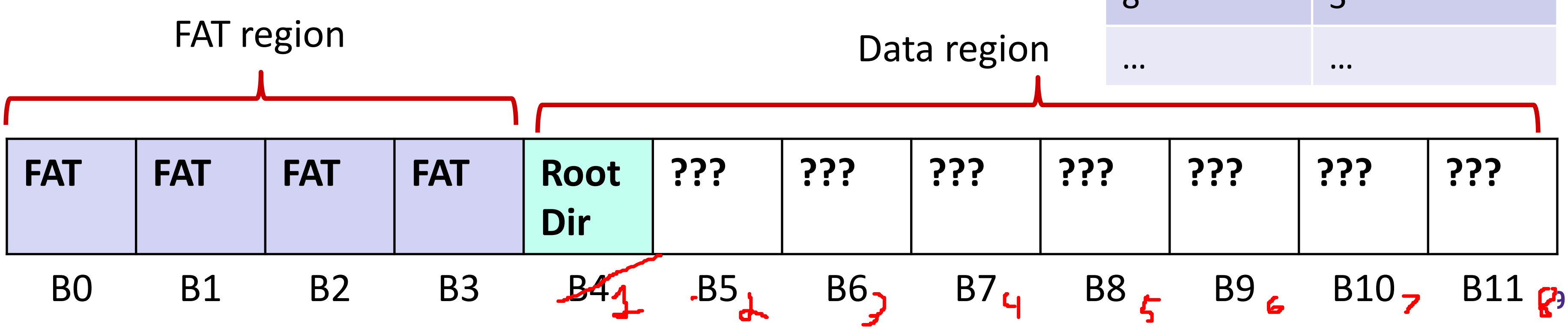
Hint: Index into data region starting at index 1

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?

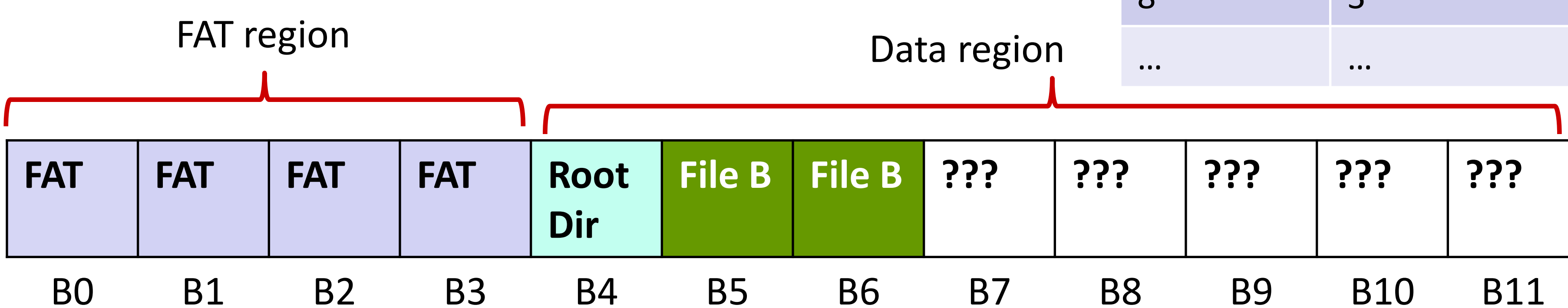
Hint: Index into data region starting at index 1

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?

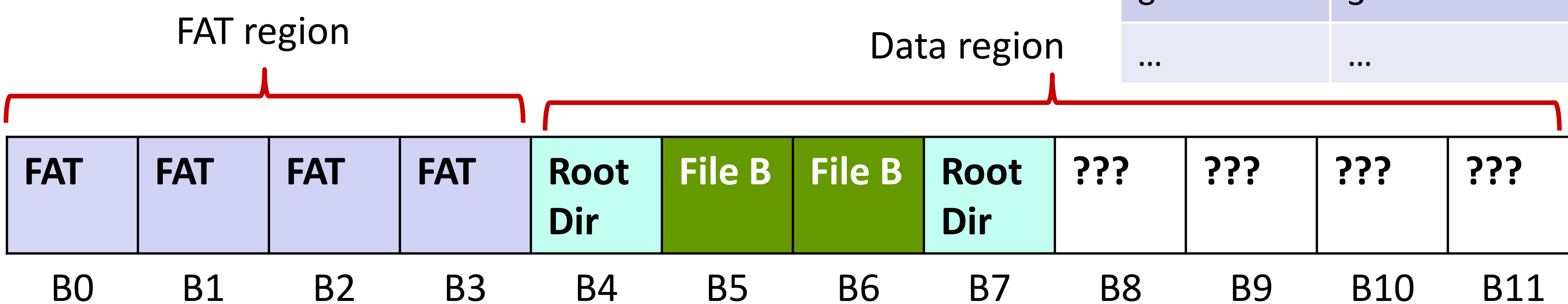
Hint: Index into data region starting at index 1

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	EMP TY	???	???	???

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?

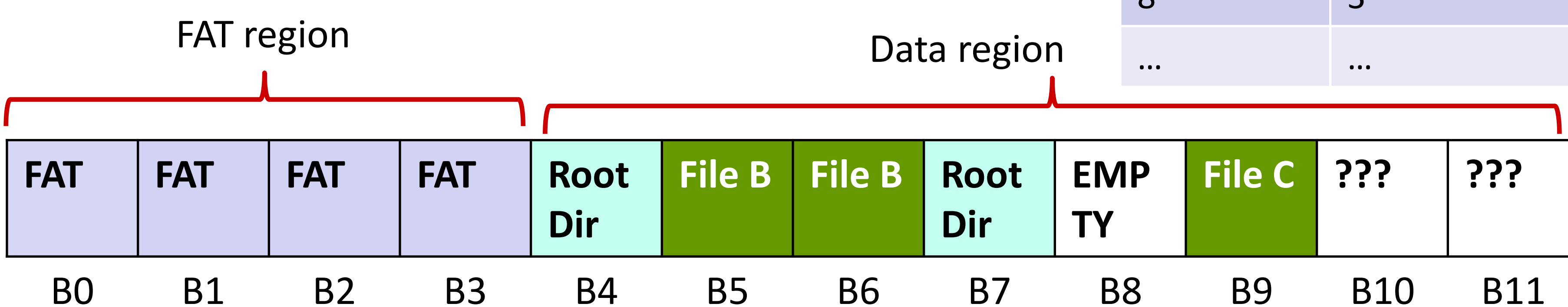
Hint: Index into data region starting at index 1

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?

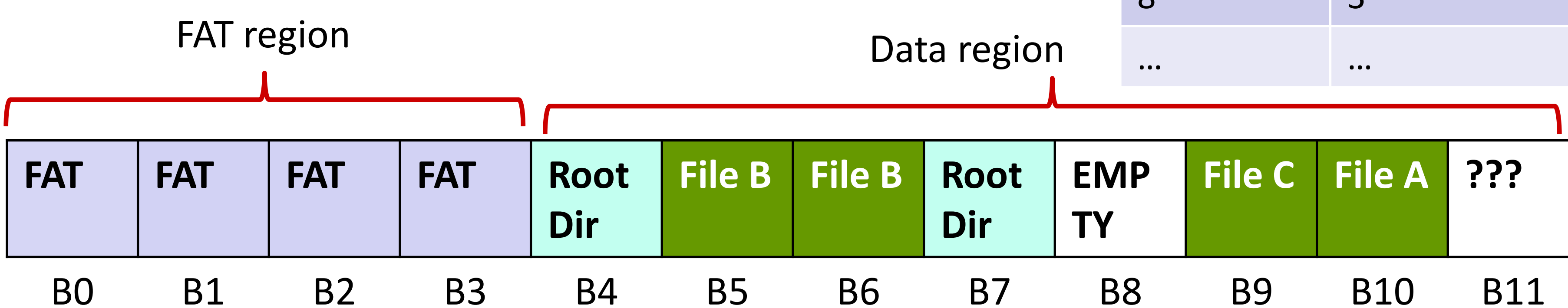
Hint: Index into data region starting at index 1

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

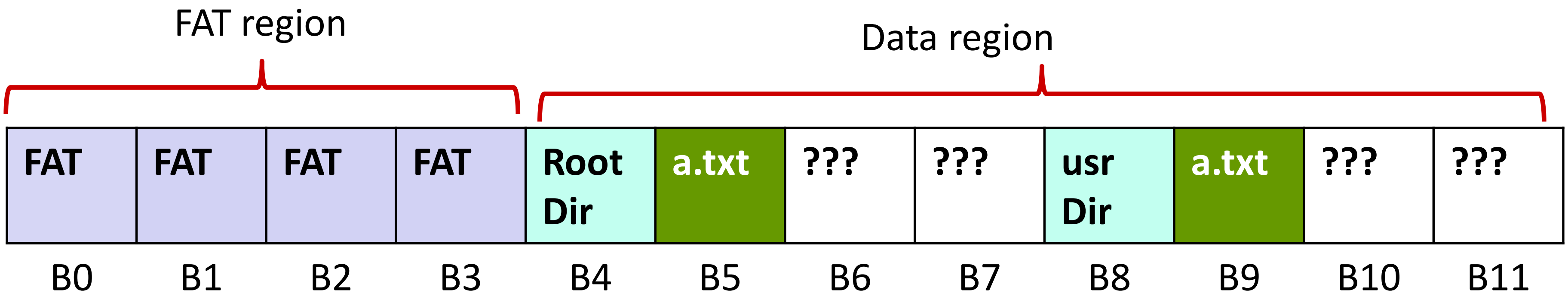
Block # (FAT Index)	Next (FAT value)
0	METADATA
1	4
2	8
3	END
4	END
5	EMPTY
6	END
7	END
8	3
…	…

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	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

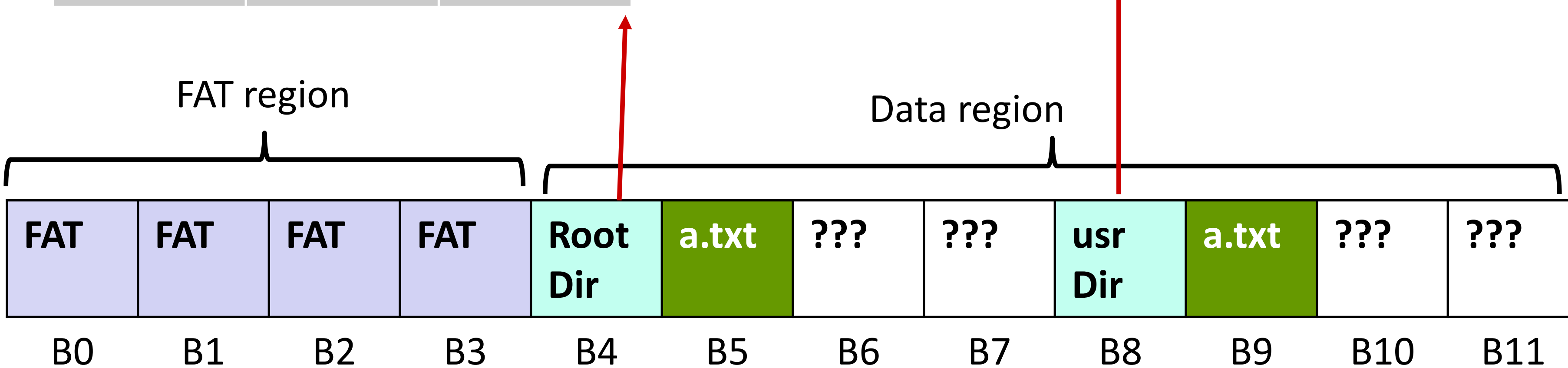
- We would also have some information in a directory entry to specify what kind of file it is

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

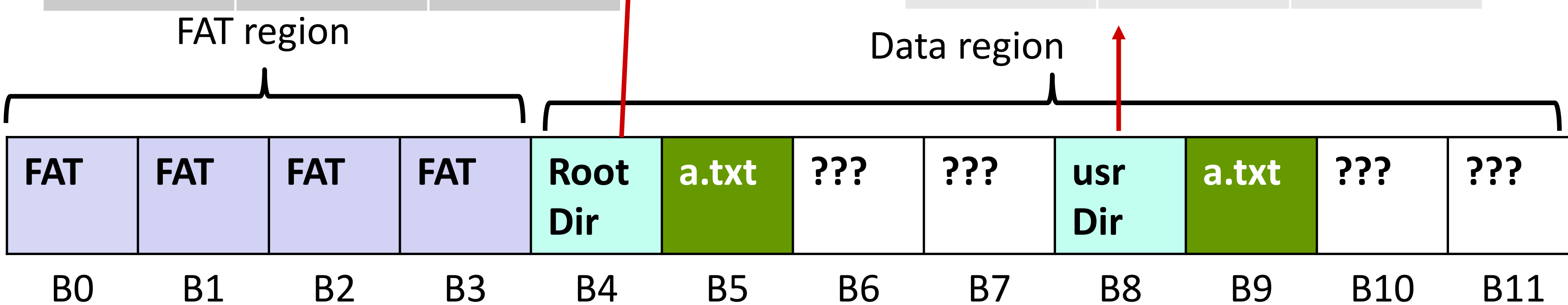
root DIR

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

- FAT & PennFAT wrap-up
- Inodes
- Directories
- **Block Caching**

Block Caching

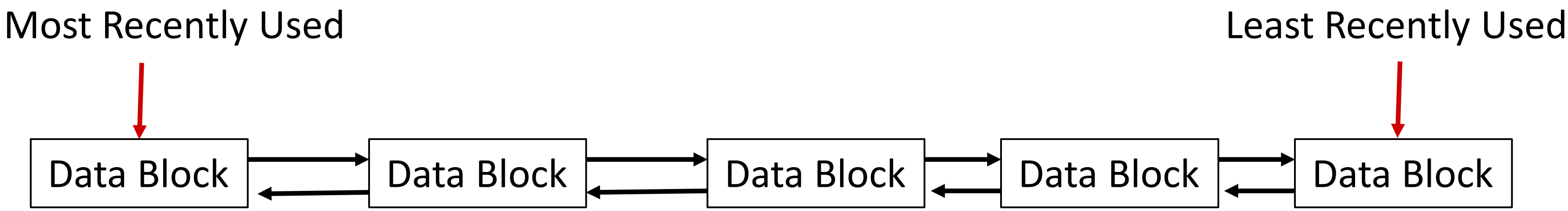
- Disk I/O is really slow (relative to accessing memory)

- 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

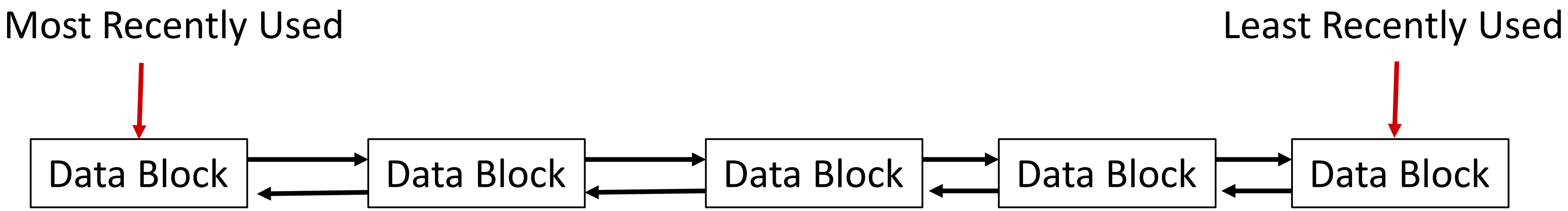
- We can use a linked list to store blocks in LRU



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

Block Caching Data Structure

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



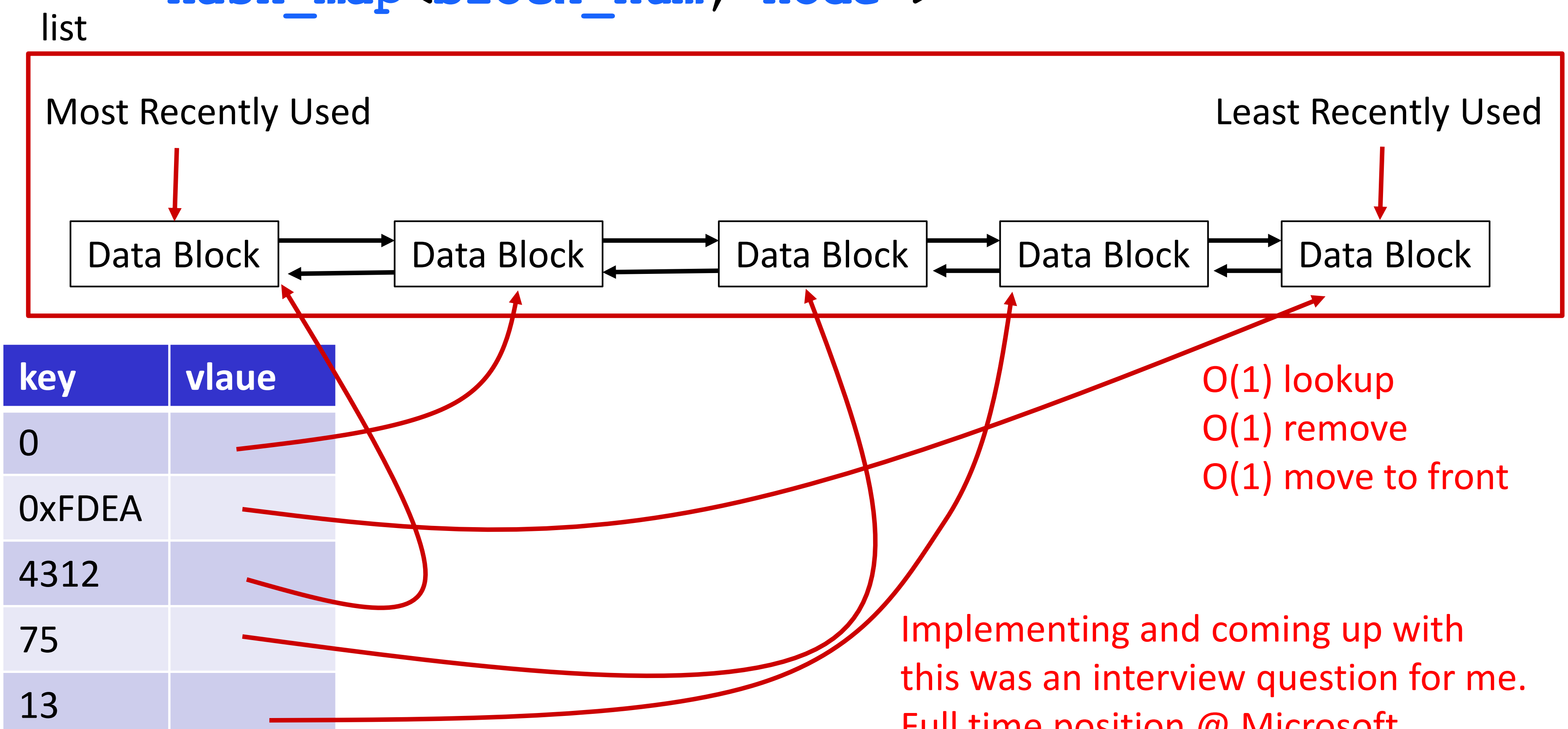
- What is the algorithmic runtime analysis to:
  - lookup a specific block? O(n)
  - Removal time? O(1)
  - Time to move a block to the front or back? O(1)

Discuss

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

Chaining Hash Cache

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



key	vlaue
0	
0xFDEA	
4312	
75	
13	

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