Recitation 3

Welcome, everybody!

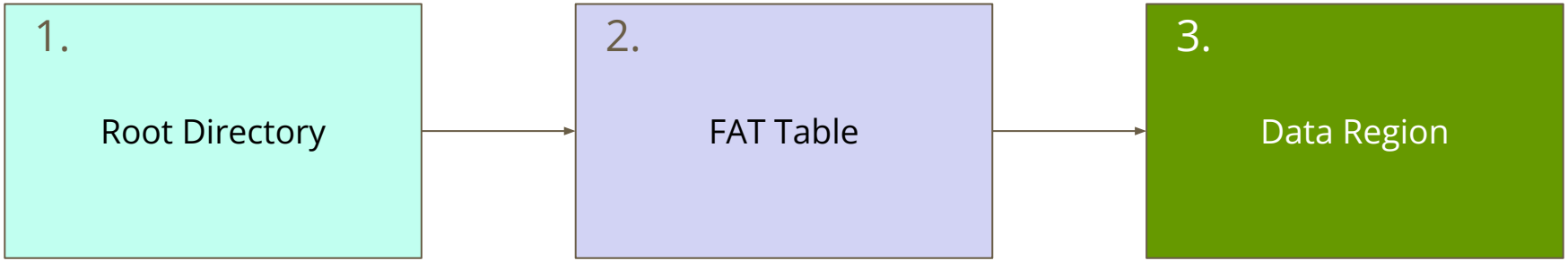
Topics Covered

- FAT File System
- FAT vs Inode
- Hexdump Guide (PennOS preview!)

FAT File System

How can I edit a file on a FAT based File System?

1. Does the file exist? Am I allowed to access it? Where can I find the file?
2. Where is the n^{th} byte of the file? That's where I want to begin editing.
3. Access the file via some block math.

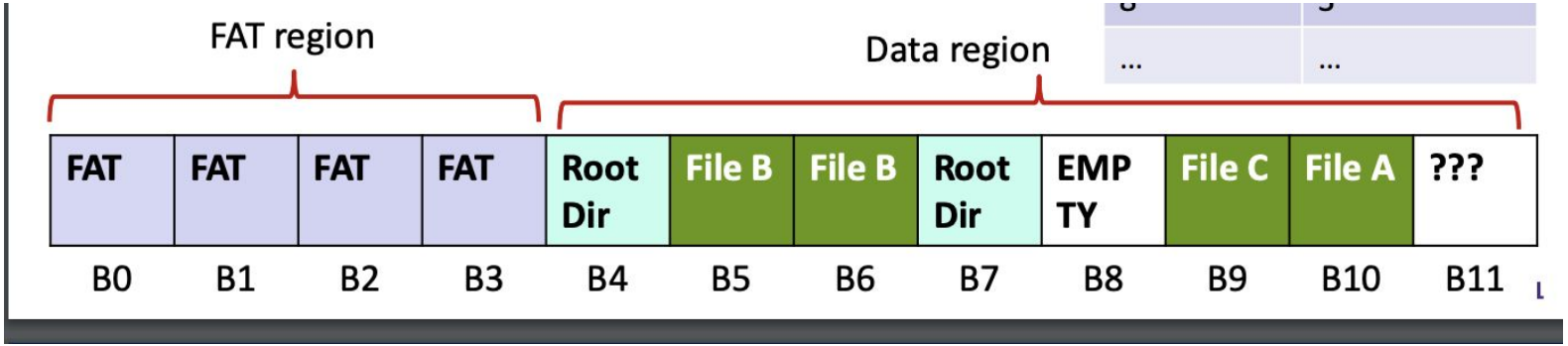


File Allocation Table

FAT system splits to two parts:

FAT region & Data Region

Data Region has *File Contents* & *Directory Contents*



Does the file exist? Can I access it? Where is the file?

The Root Directory starts at a *known* spot within the Data Region

Root directory stores info about other files.

1. Search by **name**
2. Verify **permissions**
3. Return **firstBlock**

Root Directory Entry (PennFAT)

```
char name[32];
uint32_t size;
uint16_t firstBlock;
uint8_t type;
uint8_t perm;
time_t mtime;
// The remaining 16 bytes are reserved
```

Navigating the FAT

FAT[current] = next

Index	Link
0	0x2004 <— MSB=0x20 (32 blocks in FAT), LSB=0x04 (4K-byte block size)
1	0xFFFF <— Block 1 is the only block in the root directory file
2	5 <— File A starts with Block 2 followed by Block 5
3	4 <— File B starts with Block 3 followed by Block 4
4	0xFFFF <— last block of File B
5	6 <— File A continues to Block 6
6	0xFFFF <— last block of File A
…	…

Each entry is 2 bytes (0xABCD)

[0] # of FAT blocks (MSB) & Block Size (LSB)

[1] **ROOT DIRECTORY.** First data block is the root dir.

(FAT[2].....FAT[MSB]) are file (or extended root directory) block numbers.

Finding the block address

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

e.g. **Block Size** = 4 KB; **# of Blocks in FAT** = 32

File: “Foo” has firstBlock at 2. I want to edit the 9,021 B

“Foo’s Block” = ceil (9,021 B / 4 KB) = **3rd**

Block Address = (Blocks in Fat) (Block Size) + (“Foo’s Block” - 1) (Block Size)

Skip FAT Region

1-indexing

Block Offset = 9,021 % Block Size = 829

Desired Byte Location = Block Address + Block Offset

= 164,669 “Bytes” from the start of the file system

This will be VERY helpful for PennFAT

INodes

Why use I-Nodes?

Recap: What is an Inode?

- An Inode (Index Node) is a structure that stores metadata about a file (like permissions, size, and owner) and **the locations of the file's data blocks**. Instead of requiring all blocks to be continuous, the inode points to wherever the blocks are scattered on the disk.

Feature	Contiguous Allocation	Inode-Based Allocation
Data Block Layout	All blocks for a file are stored sequentially in one continuous chunk on the disk.	Data blocks for a file can be scattered anywhere on the disk. The inode links them together.
Sequential Read Speed	**Very Fast.** Minimal disk head movement is required since all data is physically next to each other.	**Slower.** The disk head may need to jump to different locations to read all the blocks.
File Growth	**Hard.** If a file grows, a larger contiguous block of free space must be found, and the file must be copied over.	**Easy.** New blocks can be allocated from anywhere on the disk and simply added to the inode's list of pointers.
Metadata Storage	Typically stored in the directory entry.	Stored in the inode itself, separate from the directory entry.

Key Takeaways

1. **Contiguous Allocation:** Think **performance and simplicity** for files that don't change size. It excels at fast sequential reads but struggles with file growth and disk fragmentation.
2. **Inode-Based Allocation:** Think **flexibility and efficiency** for dynamic files. It handles file size changes and fragmentation gracefully at the cost of some performance overhead for sequential reads.

Comparison Question Pt 1)

Question: Which file allocation scheme is best for small, fixed-size files that are frequently read but sometimes written to: contiguous or Inodes? **Why?**

Answer:

- **Contiguous** because the size of the files stays the same, so we **do not need to reallocate**. In this case, we can specifically use **caching** to take advantage of **spatial locality** since we will never have to move existing files around given that they have constant size.

- Inodes would introduce unnecessary overhead for these specific use cases.

Comparison Question Pt. 2)

Question: If we need to store many very large files that stretch several blocks and are frequently modified including file extending/shortening and rearrangement, which allocation scheme is better: contiguous or Inodes? Why?

Answer:

- **Inode-based allocation** is better here because it is better at dealing with **dynamic file sizes** and **fragmentation**. The file's data blocks can be scattered across the disk, which allows efficient modification, extension, and truncation without needing to move the entire file.

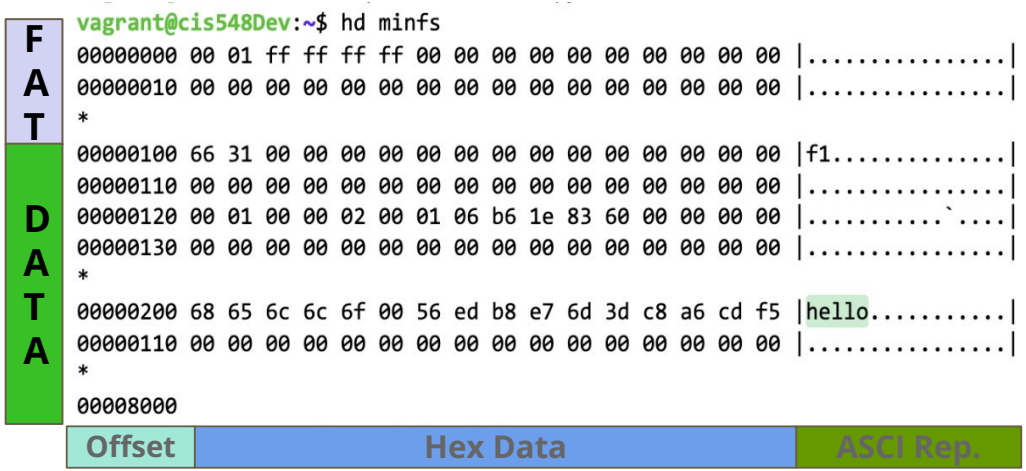
- Contiguous allocation would **not** work here because frequent length modifications would require reallocation to find a large enough block of free space.

Hexdump Example

What is a HexDump?

- A hexdump is a representation of binary data in hexadecimal format.
- The hd command shows the raw contents of a file or disk, byte by byte. We will use this a lot in Penn OS for the file system to check our progress and debug
- A hexdump has three parts: the offset, the hex data, and ASCII representation.

What does a Hexdump look like?



We will dive into the specifics next, but for some context, this is a dump of a file system that has 1 FAT block, a block size of 256B, and 1 file (f1) that just contains the word "hello".

Little Endian Explained

- Little-endian is a byte-ordering scheme where the **least significant byte comes first**. This is a common point of confusion we see when students read hexdumps.
- In the minfs FAT, 00 01 is read as 0x0100. The computer interprets the bytes in reverse order to form the full value. Which in this case is # of blocks in Fat, Block Size
- When we see the first block of f1 is 02 00, this means the first block is #2.

Diving into our Hexdump

- Let's look at the *minfs* hexdump which we use in PennOS.
- When we run mkfs minfs 1 0, it creates an empty file system that has **1 block in FAT** and a **block size of 256B**.
- Our first block is the FAT and our other 127 (256/2 - 1) blocks are reserved for Data
- The first two bytes of the FAT, FAT[0], are **important** because they give us the block size and # of blocks, which is essential for navigating our file system.

Minfs Hexdump

```
vagrant@cis548Dev:~$ pennfat
pennfat# mkfs minfs 1 0
vagrant@cis548Dev:~$ ls -l minfs maxfs
-rw-r--r-- 1 vagrant vagrant 268558336 Apr 23 19:21 maxfs
vagrant@cis548Dev:~$ hd minfs
00000000 00 01 ff ff 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
*
00008000
vagrant@cis548Dev:~$ cp minfs minfs.1
```

- FAT[0] = 00 01, which represents 0x0100 (**due to little-endian byte order**). This signifies a 1-block FAT and a block size of 256 bytes.
- FAT[1] = ff ff, which represents the File System ending at block 1.
  - Note: The File System always starts at block 1

Hexdump with a File Added

- Let's follow the file f1 through our hexdump. When we add f1, the file system updates the **FAT** and the **Directory Entry**.
- The **FAT** is updated to link the file's data blocks. Since f1's first block is #2, we see that the FAT entry for block #2 is ff ff, which means end of file.
- The directory entry is located at offset 00000100, which is the start of the root directory.
  - **66 31 00...:** This is the filename, "f1", padded with null bytes.
  - **00 01:** These bytes represent the size of the file, 0x0100, which is 1 block.
  - **02 00:** These bytes indicate the first block address of the file, 0x0002 which is block #2
  - **00 01 06 b6 1e 83 60 00 00 00 00 00 00 00 00 00:** This is the remaining metadata, including a timestamp and permissions.
- Finally, at offset 00000200 (block #1), we find the actual file data.
  - **68 65 6c 6c 6f 00:** These are the ASCII codes for "hello" followed by a null terminator. The ASCII representation on the far right confirms this.

File System with 1 File Added

```
vagrant@cis548Dev:~$ pennfat
pennfat# mount minfs
pennfat# touch f1
pennfat# cat -a f1
hello
pennfat# ls
  2 -rw-     6 Apr 23 19:23:34 2021 f1
pennfat# cp f1 -h demo2copy
pennfat#
vagrant@cis548Dev:~$ cmp demo2 demo2copy
vagrant@cis548Dev:~$ hd minfs
00000000 00 01 ff ff ff ff 00 00 00 00 00 00 00 00 00 00 |................|
00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
*
00000100 66 31 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |f1..............|
00000110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00000120 00 01 00 00 02 00 01 06 b6 1e 83 60 00 00 00 00 |...........`....|
00000130 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
*
00000200 68 65 6c 6c 6f 00 56 ed b8 e7 6d 3d c8 a6 cd f5 |hello.V...m=....|
00000110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
*
00008000
vagrant@cis548Dev:~$ cp minfs minfs.2
```

FAT

ROOT

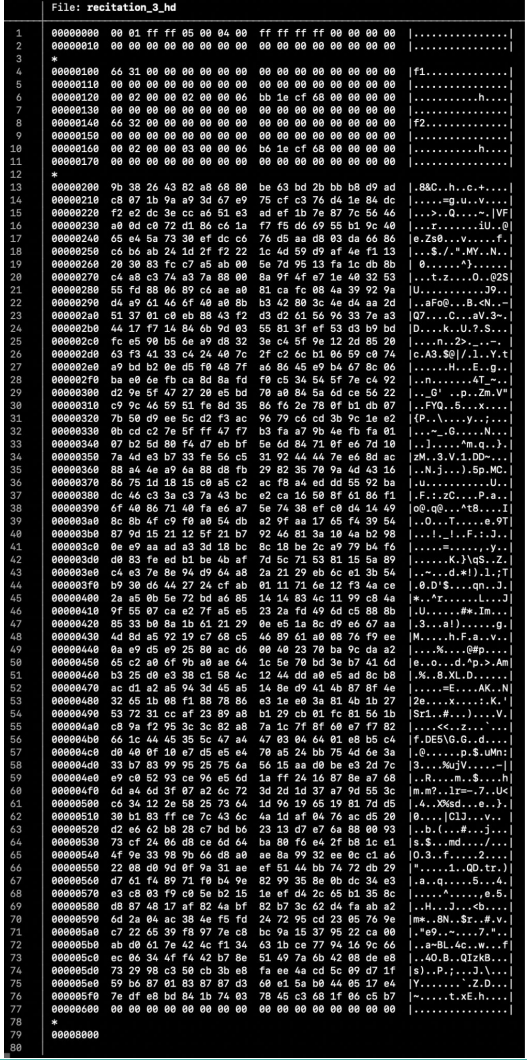
F1

On Gradescope →

Hexdump Navigation

1. What is the index of f1's second block?

   5

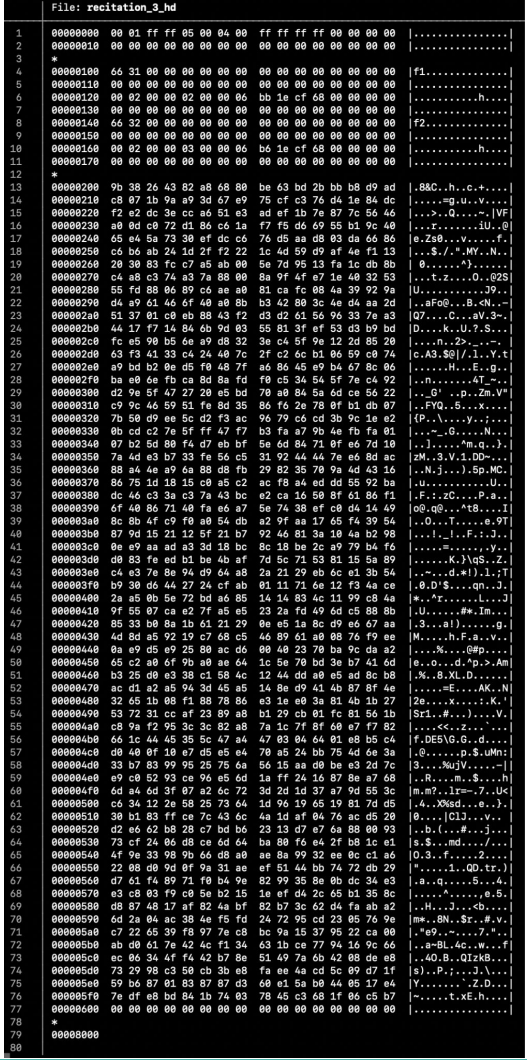


On Gradescope →

Hexdump Navigation

2. What is the block size of the hexdump?

256



On Gradescope →

Hexdump Navigation

3. What is the 404th byte of f2?

Block Size = 256 B, # Blocks in FAT = 1

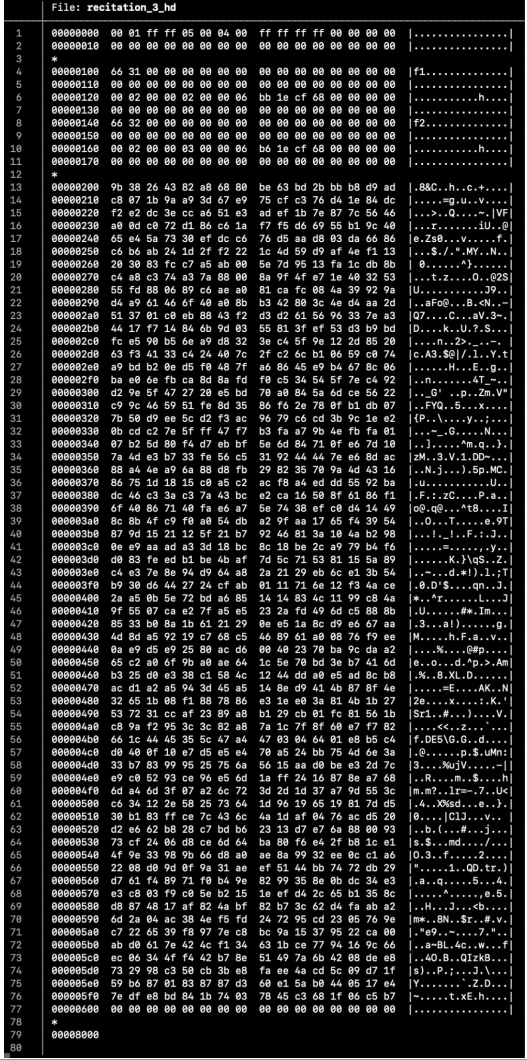
Desired Block = ceil(404/256) = 2nd ⇒ Block 4

Block Offset = 404 % 256 = 148 B

Byte Location = (Blocks in Fat) (Block Size) + (Desired Block - 1) (Block Size) + Block Offset

= (1)(256)+(4-1)(256) + 148

0x00000494 ⇒ 23



Hexdump Navigation

3. What is the 404th byte of f2?

0x494