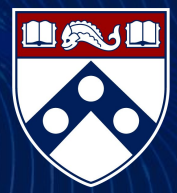


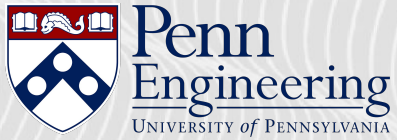
Penn Engineering
University of Pennsylvania

CIS 5480 Recitation 4

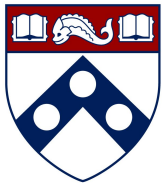
File Systems and pennfat

2026-10-09

Agenda



- Inodes Recap
- Q1: Designing open()
- FAT Recap
- Hexdumps
- Q2: Hexdumps are Scary(?)
- pennfat demo



Penn
Engineering
University of Pennsylvania

Inodes Recap

FAT vs. Inodes

Penn Engineering
University of Pennsylvania

FAT

Pros:

- Simple, just one array of "next block" pointers
  - Easy to grow a file

Cons:

- Slow for random access, you need to traverse an entire block chain
- Lots of memory consumed

Inodes

Pros:

- Fast random access via direct & indirect pointers
- Less memory consumed, only the inodes of open files are in memory

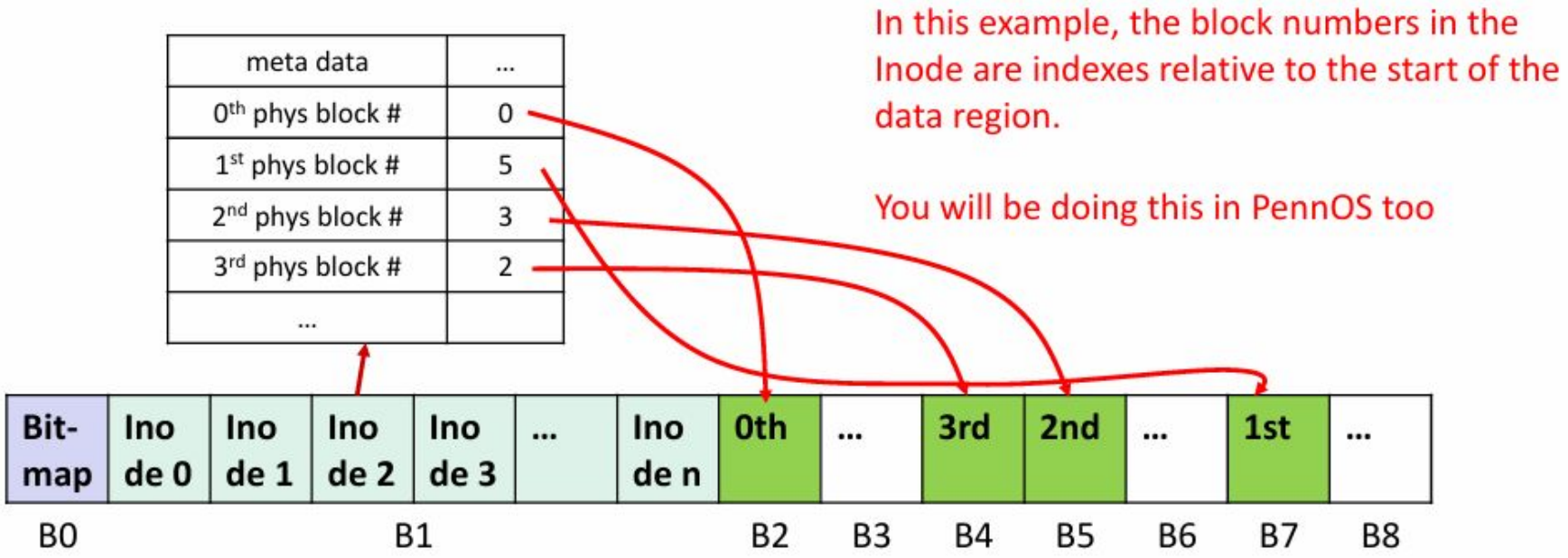
Cons:

- Harder to implement (that is why you are creating a FAT :D)

Inodes: Basics

Idea: index node (inode) is 1-1 with file and holds (most) file info.

Loading inodes for open files is (usually) smaller than loading FAT!



Inodes: Directories

Consider the dirent for pennfat. How should we change this to work with inodes?

Change firstBlock to inodeNum!

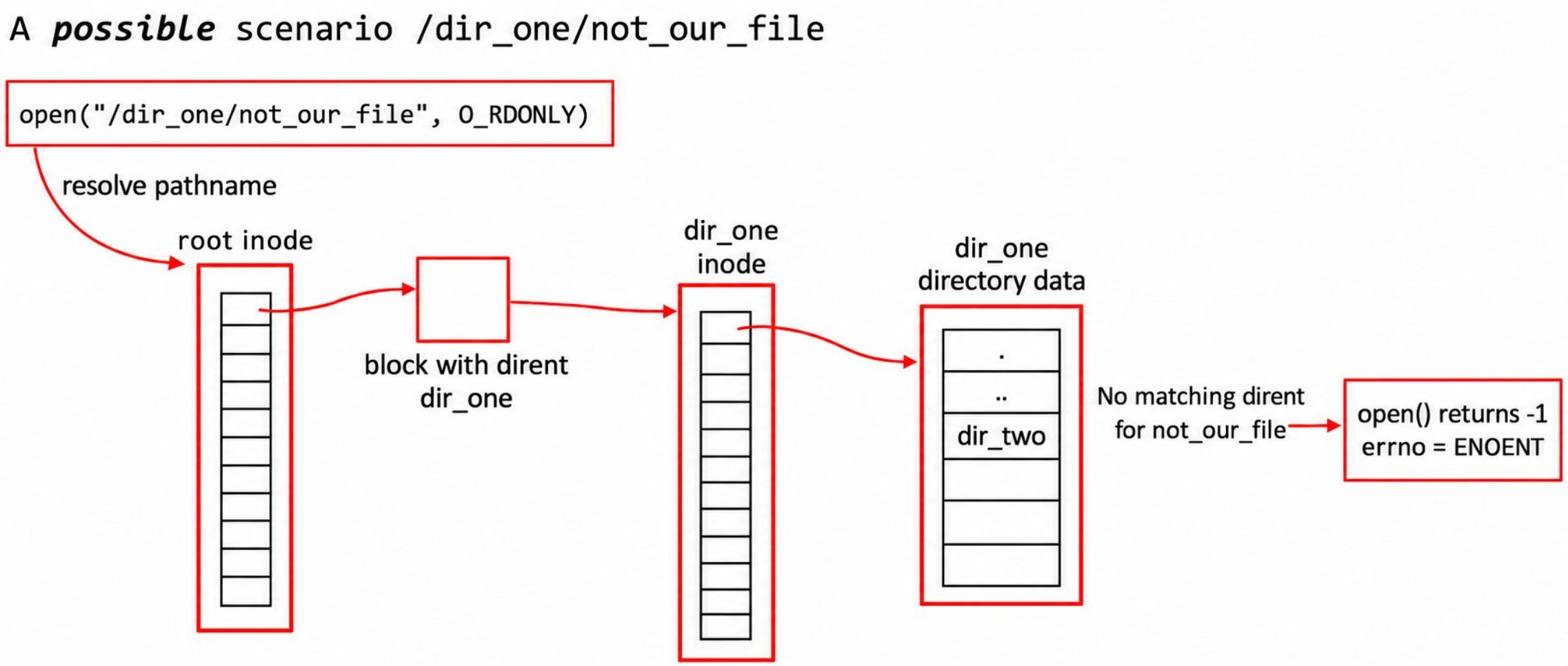
FAT: files 1-1 with first blocks

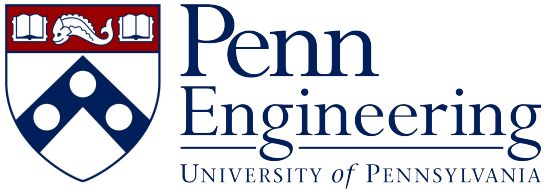
Inodes: files 1-1 with inode nums

(Also, we can move some of the file's metadata to the inode.. it has space reserved for it!)

```
struct pennos_dirent {
  char name[32];
  uint32_t size;
  uint16_t firstBlock;
  uint8_t type;
  uint8_t perm;
  time_t mtime;
  // remaining 16 bytes reserved (e.g. for extensions)
}
```

Inodes: Traversing





Q1: Designing open()

BTW: you will need this for your pennfat milestone ;)

Q1: Premise

Penn Engineering
University of Pennsylvania

1. Designing open

You are now tasked with designing open for an inode-based filesystem. From the user's perspective, calling open returns a file descriptor that can later be used with read and write. From the filesystem's perspective, open must resolve a path name, locate an inode, and initialize per-open state.

Assume:

- Directories store mappings from file names to inode numbers
- The filesystem maintains a global open file table (OFT)
- Each process maintains a per-process file descriptor table

Q1.1

Outline the sequence of steps required to implement 'open(path, flags)', including path traversal, inode lookup, permission checks, allocation of open-file state. This is what you will do in PennFAT.

1. Starting at root, find dirent for the next part of path; if subdirectory, enter it
2. Find inode number in dirent for the file
3. Check this inode to see if flags' permissions match
4. If creating a new file (no dirent found), allocate an inode and dirent with all block pointers in inode zeroed (no data blocks)
5. Allocate an OFT entry with file info, inode number, ref count, access mode
6. Allocate a FD that points to the OFT entry; return this

Q1.2

Where should the current file offset be stored: the inode, the directory entry, the open file table entry, or the file descriptor table? Justify your answer by considering what should happen when the same file is opened multiple times.

The OFT entry. This is our “ticket” to access the file, so it specifies the permissions and offset.

Each call to open() creates its own OFT entry with its own offset; a file has only one inode and one dirent, so the offset can't go there without every open() sharing it.

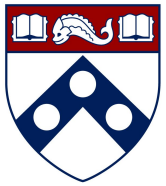
FDs created by dup() or fork() point to the same OFT entry and should share an offset, so storing it per-FD would mean syncing it by hand; keeping it in the OFT prevents this annoying extra work.

Q1.3

Suppose two processes simultaneously call write(path, ...). Identify one race condition that can occur and describe one mechanism the filesystem could use to prevent it. (We haven't learned about these yet so just speculate what you should do.)

If the writes overlap, meaning they try to write to the same data, the final value there is based on the last process which wrote there - this is non-deterministic due to the scheduler.

Blunt force solution: each file may have at most 1 writer at a time (we have to enforce this! Think about how you might do so).



Penn
Engineering
University of Pennsylvania

FAT Recap

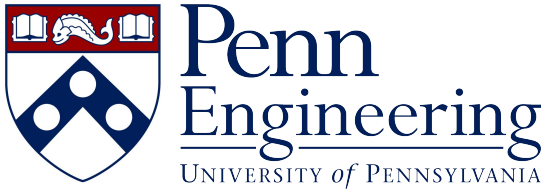
FAT

FAT[block] =

- if unused, 0x0000
- else if end of the file, 0xFFFF
- otherwise, next block

Idea: first block corresponds 1-1 with file.

Store size of block and number of blocks in FAT in FAT[0], so LSB encodes size and MSB encodes number. FAT[0] is always used for this, so data blocks are 1-indexed.



Hexdump

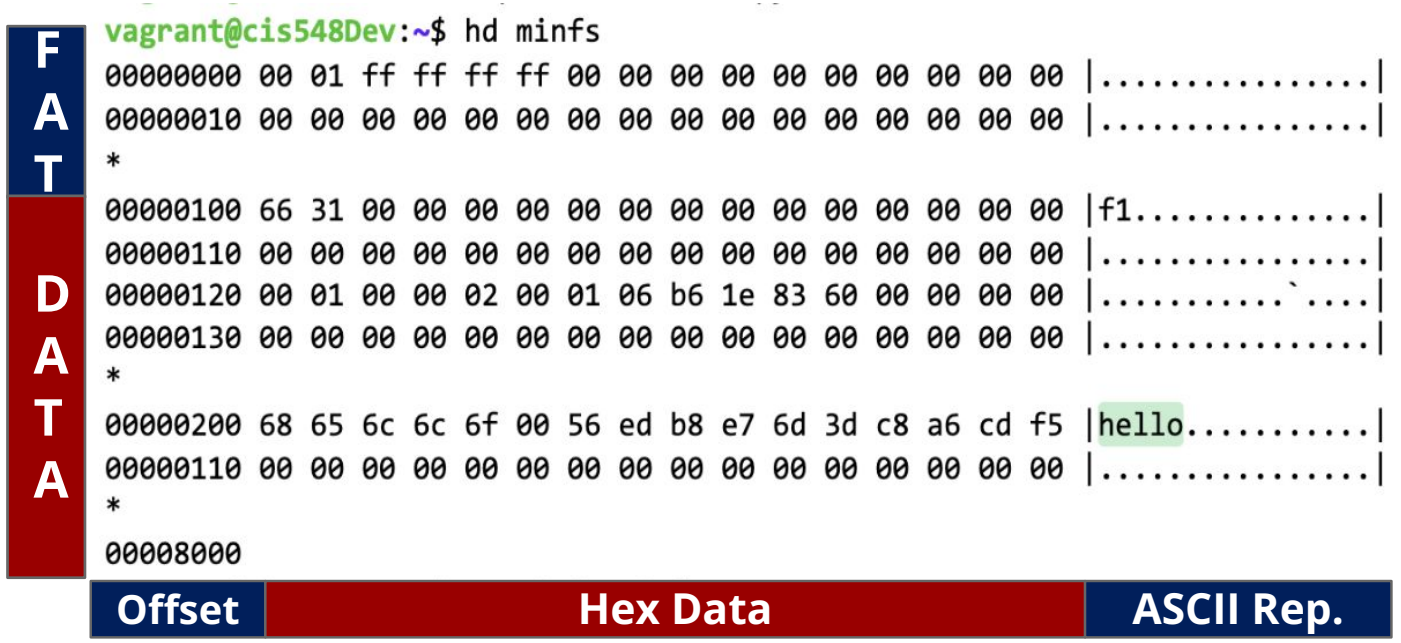
Hexdump

Program which prints a file in hexadecimal.

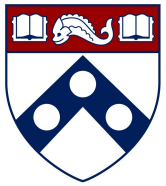
hexdump [file] or hd [file]

3 components:

- Offset
- Hex data
- ASCII rep



minfs is a FS with 1 FAT block, block size 256B, and a file named f1 containing “hello”.



Penn
Engineering
University of Pennsylvania

Q2: Hexdumps are Scary(?)

Q2: Premise

Penn Engineering
University of Pennsylvania

2. Hexdumps are Scary(?)

Let's practice reverse engineering a file system directly from its raw bytes so we don't get so scared anymore.

In this question, you will be shown two PennFAT hexdumps, one above the other. Each hexdump represents the beginning of a different PennFAT file system image.Remember, each PennFAT entry is 16 bytes.

Your task is to extract structural information purely from the hex.

Q2.1

2.1 Before We Begin: Endianness

Sometimes, we look at integers displayed in pure hex and are confused by their representation in the terminal. Let's go ahead and disambiguate this.

Questions:

- What is the difference between big endian and little endian?

Big endian: order bytes MSB…LSB; little endian: order bytes LSB…MSB.

Why little endian? It makes integer downcasts easy. To cast an 8B int to a 4B int, start at the same spot and look at the first 4 bytes.

On most machines, we store multi-byte values in little endian. And the hex dump will print values from the lowest address first, from left to right.

If a 2-byte value appears in the start of the hexdump as:

```
0x00 0x10
```

What does this tell us about the block sizes and the number of FAT blocks?

Block size: 2^{8+0x00}B = 256B

Number of FAT blocks: 0x10 = 16

Q2.2

Penn Engineering
University of Pennsylvania

2.2 PennFAT Hexdumps

The following are two PennFAT file system images shown as hexdumps. They represent two different file system configurations.

PennFAT Image A

```
00000000 04 20 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 |................|
*
1001e000
```

PennFAT Image B

```
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
```

Throughout the questions below, clearly indicate whether you are referring to Image A or Image B.

Q2.3

Penn Engineering
University of Pennsylvania

2.3 Reverse Engineering the FAT Region

Questions:

How many blocks does the FAT occupy for both hexdumps?

A: 0x20 = 32

B: 0x01 = 1

What is the size (in bytes) of each block?

A: 4096B (2^{8+0x04}B = 4096B)

B: 256B (2^{8+0x00}B = 256B)

What **mkfs-style command** would you run in the PennFAT assignment to create each of these two file systems?

A: mkfs A 32 4

B: mkfs B 1 0

Q2.4

Penn Engineering
University of Pennsylvania

2.4 The FAT Finale

Now, take a look at the following hex dump of the FAT after some modifications. There is only one block in the FAT but you should have known that by now.

Based on this, how many files are on the file system, how large is the largest file? You will need to implement something just identical to this, so it is imperative that you understand how to read these hexdumps.

Q2.4

2.4 The FAT Finale

Now, take a look at the following hex dump of the FAT after some modifications. There is only one block in the FAT but you should have known that by now.

```
00000000 00 01 ff ff ff ff 04 00 05 00 06 00 07 00 08 00
00000010 09 00 0a 00 0b 00 0c 00 0d 00 0e 00 0f 00 10 00
00000020 11 00 12 00 13 00 14 00 15 00 16 00 17 00 18 00
00000030 19 00 1a 00 1b 00 1c 00 1d 00 1e 00 1f 00 20 00
00000040 21 00 22 00 23 00 24 00 25 00 26 00 27 00 28 00
00000050 29 00 2a 00 2b 00 2c 00 2d 00 2e 00 2f 00 30 00
00000060 31 00 32 00 33 00 34 00 35 00 36 00 37 00 38 00
00000070 39 00 3a 00 3b 00 3c 00 3d 00 3e 00 3f 00 40 00
00000080 41 00 42 00 43 00 44 00 45 00 46 00 47 00 48 00
00000090 49 00 4a 00 4b 00 4c 00 4d 00 4e 00 4f 00 50 00
000000a0 51 00 52 00 53 00 54 00 55 00 56 00 57 00 58 00
000000b0 59 00 5a 00 5b 00 5c 00 5d 00 5e 00 5f 00 60 00
000000c0 61 00 62 00 63 00 64 00 65 00 66 00 67 00 68 00
000000d0 69 00 6a 00 6b 00 6c 00 6d 00 6e 00 6f 00 70 00
000000e0 71 00 72 00 73 00 74 00 75 00 76 00 77 00 78 00
000000f0 79 00 7a 00 7b 00 7c 00 7d 00 7e 00 7f 00 ff ff
```

Based on this, how many files are on the file system, how large is the largest file? You will need to implement something just identical to this, so it is imperative that you understand how to read these hexdumps.

First, interpret FAT[0]: 0x0100

- 1 FAT block
- 256B block size

(256 / 2 = 128 entries in the FAT)

Q2.4

Penn Engineering
University of Pennsylvania

2.4 The FAT Finale

Now, take a look at the following hex dump of the FAT after some modifications. There is only one block in the FAT but you should have known that by now.

```
00000000 00 01 ff ff ff ff 04 00 05 00 06 00 07 00 08 00
00000010 09 00 0a 00 0b 00 0c 00 0d 00 0e 00 0f 00 10 00
00000020 11 00 12 00 13 00 14 00 15 00 16 00 17 00 18 00
00000030 19 00 1a 00 1b 00 1c 00 1d 00 1e 00 1f 00 20 00
00000040 21 00 22 00 23 00 24 00 25 00 26 00 27 00 28 00
00000050 29 00 2a 00 2b 00 2c 00 2d 00 2e 00 2f 00 30 00
00000060 31 00 32 00 33 00 34 00 35 00 36 00 37 00 38 00
00000070 39 00 3a 00 3b 00 3c 00 3d 00 3e 00 3f 00 40 00
00000080 41 00 42 00 43 00 44 00 45 00 46 00 47 00 48 00
00000090 49 00 4a 00 4b 00 4c 00 4d 00 4e 00 4f 00 50 00
000000a0 51 00 52 00 53 00 54 00 55 00 56 00 57 00 58 00
000000b0 59 00 5a 00 5b 00 5c 00 5d 00 5e 00 5f 00 60 00
000000c0 61 00 62 00 63 00 64 00 65 00 66 00 67 00 68 00
000000d0 69 00 6a 00 6b 00 6c 00 6d 00 6e 00 6f 00 70 00
000000e0 71 00 72 00 73 00 74 00 75 00 76 00 77 00 78 00
000000f0 79 00 7a 00 7b 00 7c 00 7d 00 7e 00 7f 00 ff ff
```

Based on this, how many files are on the file system, how large is the largest file? You will need to implement something just identical to this, so it is imperative that you understand how to read these hexdumps.

FAT[1]: 0xFFFF

- First file (root directory)
- Already at the end of the chain, so first block of the file is also the last block
- All data for this file is in physical block 1 (0x100, directly after the FAT)

Q2.4

2.4 The FAT Finale

Now, take a look at the following hex dump of the FAT after some modifications. There is only one block in the FAT but you should have known that by now.

```
00000000 00 01 ff ff ff ff 04 00 05 00 06 00 07 00 08 00
00000010 09 00 0a 00 0b 00 0c 00 0d 00 0e 00 0f 00 10 00
00000020 11 00 12 00 13 00 14 00 15 00 16 00 17 00 18 00
00000030 19 00 1a 00 1b 00 1c 00 1d 00 1e 00 1f 00 20 00
00000040 21 00 22 00 23 00 24 00 25 00 26 00 27 00 28 00
00000050 29 00 2a 00 2b 00 2c 00 2d 00 2e 00 2f 00 30 00
00000060 31 00 32 00 33 00 34 00 35 00 36 00 37 00 38 00
00000070 39 00 3a 00 3b 00 3c 00 3d 00 3e 00 3f 00 40 00
00000080 41 00 42 00 43 00 44 00 45 00 46 00 47 00 48 00
00000090 49 00 4a 00 4b 00 4c 00 4d 00 4e 00 4f 00 50 00
000000a0 51 00 52 00 53 00 54 00 55 00 56 00 57 00 58 00
000000b0 59 00 5a 00 5b 00 5c 00 5d 00 5e 00 5f 00 60 00
000000c0 61 00 62 00 63 00 64 00 65 00 66 00 67 00 68 00
000000d0 69 00 6a 00 6b 00 6c 00 6d 00 6e 00 6f 00 70 00
000000e0 71 00 72 00 73 00 74 00 75 00 76 00 77 00 78 00
000000f0 79 00 7a 00 7b 00 7c 00 7d 00 7e 00 7f 00 ff ff
```

Based on this, how many files are on the file system, how large is the largest file? You will need to implement something just identical to this, so it is imperative that you understand how to read these hexdumps.

FAT[2]: 0xFFFF

- Next file
- Already at the end of the chain, so first block of the file is also the last block
- All data for this file is in physical block 2 (0x200)

Q2.4

2.4 The FAT Finale

Now, take a look at the following hex dump of the FAT after some modifications. There is only one block in the FAT but you should have known that by now.

```
00000000 00 01 ff ff ff ff 04 00 05 00 06 00 07 00 08 00
00000010 09 00 0a 00 0b 00 0c 00 0d 00 0e 00 0f 00 10 00
00000020 11 00 12 00 13 00 14 00 15 00 16 00 17 00 18 00
00000030 19 00 1a 00 1b 00 1c 00 1d 00 1e 00 1f 00 20 00
00000040 21 00 22 00 23 00 24 00 25 00 26 00 27 00 28 00
00000050 29 00 2a 00 2b 00 2c 00 2d 00 2e 00 2f 00 30 00
00000060 31 00 32 00 33 00 34 00 35 00 36 00 37 00 38 00
00000070 39 00 3a 00 3b 00 3c 00 3d 00 3e 00 3f 00 40 00
00000080 41 00 42 00 43 00 44 00 45 00 46 00 47 00 48 00
00000090 49 00 4a 00 4b 00 4c 00 4d 00 4e 00 4f 00 50 00
000000a0 51 00 52 00 53 00 54 00 55 00 56 00 57 00 58 00
000000b0 59 00 5a 00 5b 00 5c 00 5d 00 5e 00 5f 00 60 00
000000c0 61 00 62 00 63 00 64 00 65 00 66 00 67 00 68 00
000000d0 69 00 6a 00 6b 00 6c 00 6d 00 6e 00 6f 00 70 00
000000e0 71 00 72 00 73 00 74 00 75 00 76 00 77 00 78 00
000000f0 79 00 7a 00 7b 00 7c 00 7d 00 7e 00 7f 00 ff ff
```

Based on this, how many files are on the file system, how large is the largest file? You will need to implement something just identical to this, so it is imperative that you understand how to read these hexdumps.

FAT[3]: 0x0004

- Third file
- Points us to FAT[4] for the next block in the chain
- Let's follow the chain to the end…

Q2.4

Penn Engineering
University of Pennsylvania

2.4 The FAT Finale

Now, take a look at the following hex dump of the FAT after some modifications. There is only one block in the FAT but you should have known that by now.

```
00000000 00 01 ff ff ff ff 04 00 05 00 06 00 07 00 08 00
00000010 09 00 0a 00 0b 00 0c 00 0d 00 0e 00 0f 00 10 00
00000020 11 00 12 00 13 00 14 00 15 00 16 00 17 00 18 00
00000030 19 00 1a 00 1b 00 1c 00 1d 00 1e 00 1f 00 20 00
00000040 21 00 22 00 23 00 24 00 25 00 26 00 27 00 28 00
00000050 29 00 2a 00 2b 00 2c 00 2d 00 2e 00 2f 00 30 00
00000060 31 00 32 00 33 00 34 00 35 00 36 00 37 00 38 00
00000070 39 00 3a 00 3b 00 3c 00 3d 00 3e 00 3f 00 40 00
00000080 41 00 42 00 43 00 44 00 45 00 46 00 47 00 48 00
00000090 49 00 4a 00 4b 00 4c 00 4d 00 4e 00 4f 00 50 00
000000a0 51 00 52 00 53 00 54 00 55 00 56 00 57 00 58 00
000000b0 59 00 5a 00 5b 00 5c 00 5d 00 5e 00 5f 00 60 00
000000c0 61 00 62 00 63 00 64 00 65 00 66 00 67 00 68 00
000000d0 69 00 6a 00 6b 00 6c 00 6d 00 6e 00 6f 00 70 00
000000e0 71 00 72 00 73 00 74 00 75 00 76 00 77 00 78 00
000000f0 79 00 7a 00 7b 00 7c 00 7d 00 7e 00 7f 00 ff ff
```

Based on this, how many files are on the file system, how large is the largest file? You will need to implement something just identical to this, so it is imperative that you understand how to read these hexdumps.

FAT[3]: 0x0004

- Third file
- Points us to FAT[4] for the next block in the chain
- Let's follow the chain to the end…

Q2.4

2.4 The FAT Finale

Now, take a look at the following hex dump of the FAT after some modifications. There is only one block in the FAT but you should have known that by now.

```
00000000 00 01 ff ff ff ff 04 00 05 00 06 00 07 00 08 00
00000010 09 00 0a 00 0b 00 0c 00 0d 00 0e 00 0f 00 10 00
00000020 11 00 12 00 13 00 14 00 15 00 16 00 17 00 18 00
00000030 19 00 1a 00 1b 00 1c 00 1d 00 1e 00 1f 00 20 00
00000040 21 00 22 00 23 00 24 00 25 00 26 00 27 00 28 00
00000050 29 00 2a 00 2b 00 2c 00 2d 00 2e 00 2f 00 30 00
00000060 31 00 32 00 33 00 34 00 35 00 36 00 37 00 38 00
00000070 39 00 3a 00 3b 00 3c 00 3d 00 3e 00 3f 00 40 00
00000080 41 00 42 00 43 00 44 00 45 00 46 00 47 00 48 00
00000090 49 00 4a 00 4b 00 4c 00 4d 00 4e 00 4f 00 50 00
000000a0 51 00 52 00 53 00 54 00 55 00 56 00 57 00 58 00
000000b0 59 00 5a 00 5b 00 5c 00 5d 00 5e 00 5f 00 60 00
000000c0 61 00 62 00 63 00 64 00 65 00 66 00 67 00 68 00
000000d0 69 00 6a 00 6b 00 6c 00 6d 00 6e 00 6f 00 70 00
000000e0 71 00 72 00 73 00 74 00 75 00 76 00 77 00 78 00
000000f0 79 00 7a 00 7b 00 7c 00 7d 00 7e 00 7f 00 ff ff
```

Based on this, how many files are on the file system, how large is the largest file? You will need to implement something just identical to this, so it is imperative that you understand how to read these hexdumps.

FAT[3]: 0x0004

- Third file
- Points us to FAT[4] for the next block in the chain
- Let's follow the chain to the end…

(Let's fast forward to the end of the chain…)

Q2.4

Penn Engineering
University of Pennsylvania

2.4 The FAT Finale

Now, take a look at the following hex dump of the FAT after some modifications. There is only one block in the FAT but you should have known that by now.

```
00000000 00 01 ff ff ff ff 04 00 05 00 06 00 07 00 08 00
00000010 09 00 0a 00 0b 00 0c 00 0d 00 0e 00 0f 00 10 00
00000020 11 00 12 00 13 00 14 00 15 00 16 00 17 00 18 00
00000030 19 00 1a 00 1b 00 1c 00 1d 00 1e 00 1f 00 20 00
00000040 21 00 22 00 23 00 24 00 25 00 26 00 27 00 28 00
00000050 29 00 2a 00 2b 00 2c 00 2d 00 2e 00 2f 00 30 00
00000060 31 00 32 00 33 00 34 00 35 00 36 00 37 00 38 00
00000070 39 00 3a 00 3b 00 3c 00 3d 00 3e 00 3f 00 40 00
00000080 41 00 42 00 43 00 44 00 45 00 46 00 47 00 48 00
00000090 49 00 4a 00 4b 00 4c 00 4d 00 4e 00 4f 00 50 00
000000a0 51 00 52 00 53 00 54 00 55 00 56 00 57 00 58 00
000000b0 59 00 5a 00 5b 00 5c 00 5d 00 5e 00 5f 00 60 00
000000c0 61 00 62 00 63 00 64 00 65 00 66 00 67 00 68 00
000000d0 69 00 6a 00 6b 00 6c 00 6d 00 6e 00 6f 00 70 00
000000e0 71 00 72 00 73 00 74 00 75 00 76 00 77 00 78 00
000000f0 79 00 7a 00 7b 00 7c 00 7d 00 7e 00 7f 00 ff ff
```

Based on this, how many files are on the file system, how large is the largest file? You will need to implement something just identical to this, so it is imperative that you understand how to read these hexdumps.

FAT[127]: 0xFFFF

- Finally reached the end of the chain for file 3
- File contains all data blocks from 0x0003 to 0x007f

For a total of 125 blocks in F3!

Q2.4

Penn Engineering
University of Pennsylvania

2.4 The FAT Finale

Now, take a look at the following hex dump of the FAT after some modifications. There is only one block in the FAT but you should have known that by now.

```
00000000 00 01 ff ff ff ff 04 00 05 00 06 00 07 00 08 00
00000010 09 00 0a 00 0b 00 0c 00 0d 00 0e 00 0f 00 10 00
00000020 11 00 12 00 13 00 14 00 15 00 16 00 17 00 18 00
00000030 19 00 1a 00 1b 00 1c 00 1d 00 1e 00 1f 00 20 00
00000040 21 00 22 00 23 00 24 00 25 00 26 00 27 00 28 00
00000050 29 00 2a 00 2b 00 2c 00 2d 00 2e 00 2f 00 30 00
00000060 31 00 32 00 33 00 34 00 35 00 36 00 37 00 38 00
00000070 39 00 3a 00 3b 00 3c 00 3d 00 3e 00 3f 00 40 00
00000080 41 00 42 00 43 00 44 00 45 00 46 00 47 00 48 00
00000090 49 00 4a 00 4b 00 4c 00 4d 00 4e 00 4f 00 50 00
000000a0 51 00 52 00 53 00 54 00 55 00 56 00 57 00 58 00
000000b0 59 00 5a 00 5b 00 5c 00 5d 00 5e 00 5f 00 60 00
000000c0 61 00 62 00 63 00 64 00 65 00 66 00 67 00 68 00
000000d0 69 00 6a 00 6b 00 6c 00 6d 00 6e 00 6f 00 70 00
000000e0 71 00 72 00 73 00 74 00 75 00 76 00 77 00 78 00
000000f0 79 00 7a 00 7b 00 7c 00 7d 00 7e 00 7f 00 ff ff
```

Based on this, how many files are on the file system, how large is the largest file? You will need to implement something just identical to this, so it is imperative that you understand how to read these hexdumps.

There are 3 files:

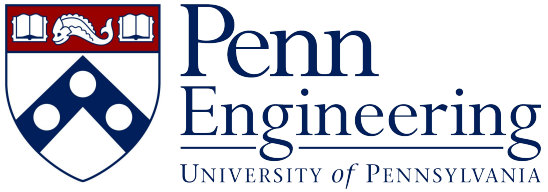
- F1: first (and last) block is 0x01
- F2: first (and last) block is 0x02
- F3: first block is 0x03

The largest file (F3) contains blocks 0x03–0x7f. This is **125 blocks**.

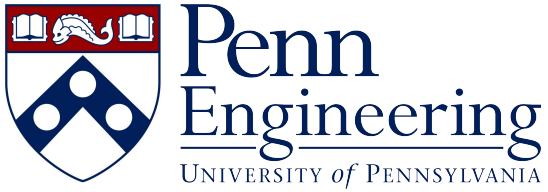
PennFat Reminders

- Headers for student-created .c files
- Repeating code can be abstracted away.
- Be careful with allocations.
- One giant source file isn’t great.
  - Each file should have a clear responsibility.
- Comment your code!

- Keep file system logic out of your shell.
- Implement helpers (directory scan, block traversal, allocation).
- Derive FAT size and block size from fat[0] at mount time.
- Test with files that cross block boundaries.



pennfat demo



Thank you!