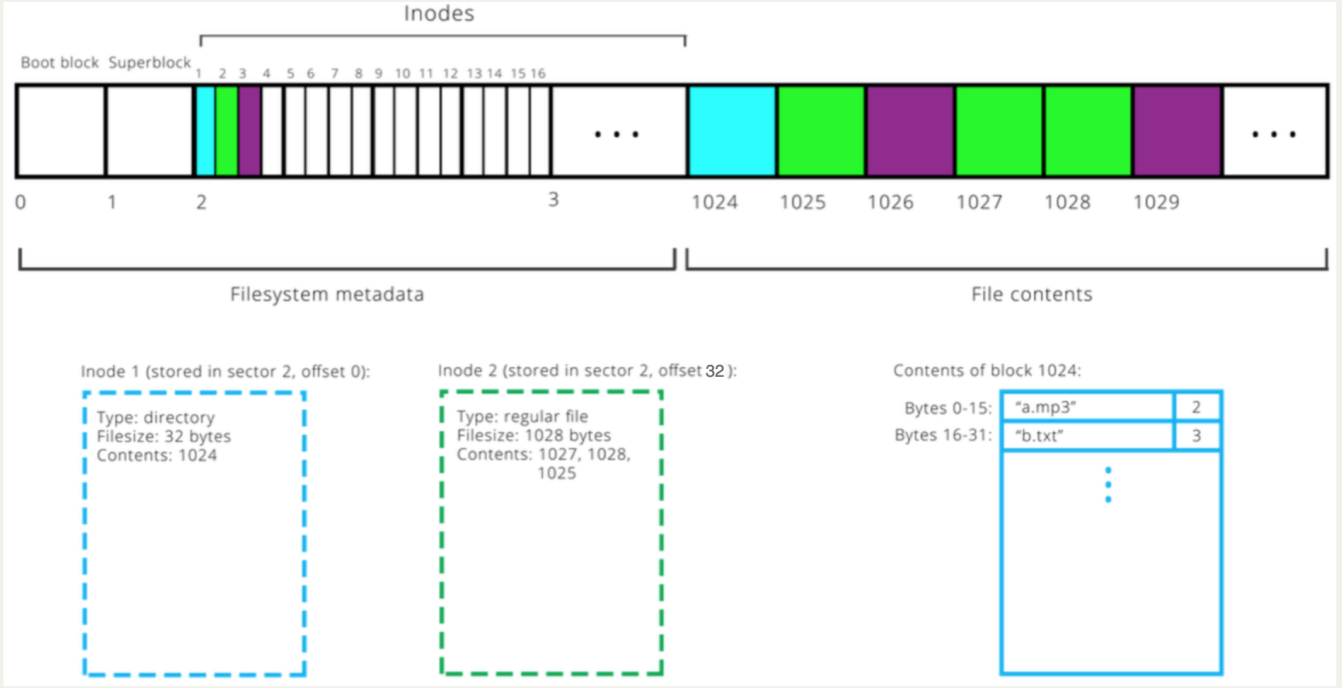
# CS110 Lecture 2: Filesystem Design, Part 1

**CS110: Principles of Computer Systems**

Winter 2021-2022

Stanford University

**Instructors**: Nick Troccoli and Jerry Cain




PDF of this presentation

# Asking Questions

- Feel free to raise your hand at any time with a question
- If you are more comfortable, you can post a question in the Ed forum thread for each day's lecture (optionally anonymously)
- We will monitor the thread throughout the lecture for questions


Visit Ed (or access via Canvas):

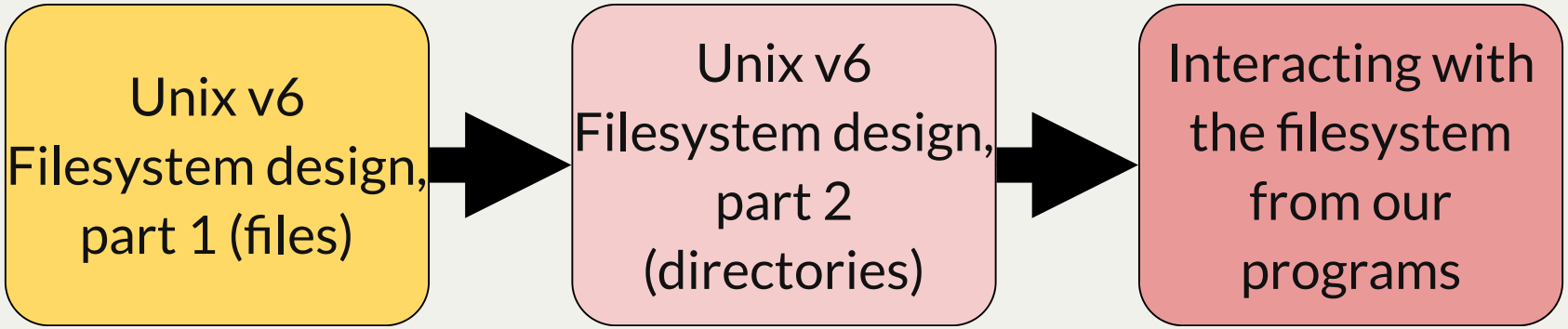
https://edstem.org/us/courses/16701/discussion/

Today's thread:

https://edstem.org/us/courses/16701/discussion/981967

**CS110 Topic 1:** How can we design filesystems to store and manipulate files on disk, and how can we interact with the filesystem in our programs?

# Learning About Filesystems




assign2: implement portions of a filesystem!

# Learning Goals

- Learn about the differences in how data is stored in memory vs. on disk
- Understand the design of the Unix v6 filesystem in how it represents files
- Understand the tradeoffs and limitations in filesystem design

# Lecture Plan

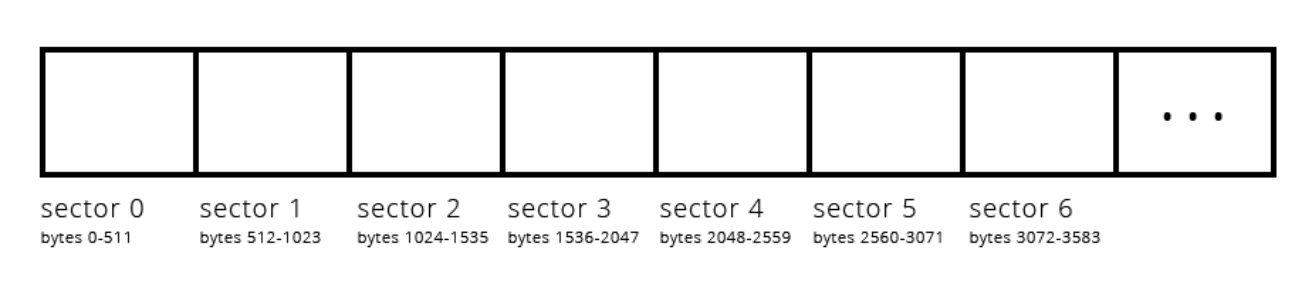
- Data Storage and Access
- Filesystem goals
- **Case Study:** The Unix v6 Filesystem
  - Sectors/Blocks
  - Inodes
  - Large files
- Practice

# Lecture Plan

- **Data Storage and Access**
- Filesystem goals
- **Case Study:** The Unix v6 Filesystem
  - Sectors/Blocks
  - Inodes
  - Large files
- Practice

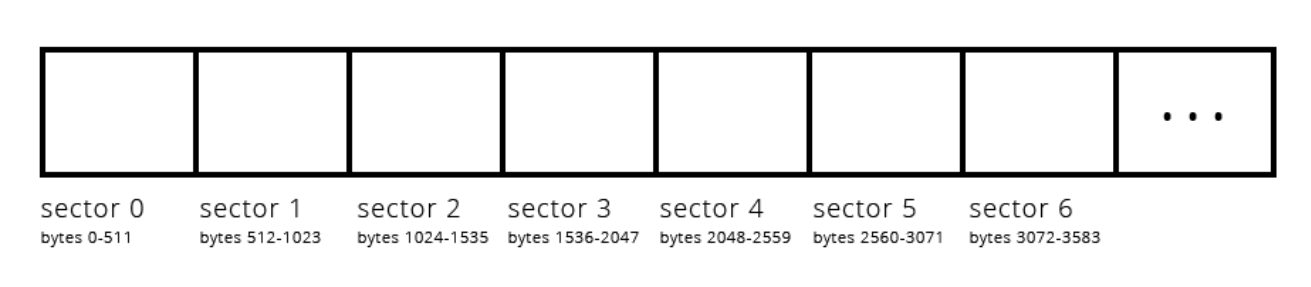
# Data Storage and Access

- The stack, heap and other segments of program data live in **memory (RAM)**
  - fast
  - *byte-addressable:* can quickly access any byte of data by address, but not individual bits by address
  - not persistent - cannot store data between power-offs
- The filesystem lives on **disk (eg. hard drives)**
  - slower
  - persistent - stores data between power-offs
  - *sector-addressable:* cannot read/write just one byte of data - can only read/write "sectors" of data



# Data Storage and Access

A hard disk is *sector-addressable:* cannot read/write just one byte of data - can only read/write "sectors" of data. (we will work with a sector size of 512; but size is determined by the physical drive).




Let's imagine that the hard disk creators provide software to let us interface with the disk.

```
void readSector(size_t sectorNumber, void *data);
void writeSector(size_t sectorNumber, const void *data);
```

This is all we get! We have to layer functions on top of these to ultimately allow us to read, write, lookup, and modify entire files.

# Data Storage and Access

Let's imagine that the hard disk creators provide software to let us interface with the disk.

```
void readSector(size_t sectorNumber, void *data);
void writeSector(size_t sectorNumber, const void *data);
```

How do we use **readSector**? Here are some examples:

```
1 char text[512];
2 readSector(5, text);
3
4 // Now text contains the contents of sector 5
5
6 int nums[512 / sizeof(int)];
7 readSector(6, nums);
8
9 // Now nums contains the contents of sector 6
```

# Data Storage and Access

Let's imagine that the hard disk creators provide software to let us interface with the disk.

```
void readSector(size_t sectorNumber, void *data);
void writeSector(size_t sectorNumber, const void *data);
```

How do we use **writeSector**? Here are some examples:

```
1  char text[512] = "Hello, world!";
2  writeSector(5, text);
3
4  // Now sector 5 contains "Hello, world!" (and \0) followed by garbage values.
5
6  int nums[512 / sizeof(int)];
7  readSector(6, nums);
8  nums[15] = 22;
9  writeSector(6, nums);
10
11 // Now sector 6 is updated to change its 16th number to be 22.
```

# Filesystem Goals

We want to read/write file on disk and have them persist even when the device is off.

This may include operations like:

- creating a new file on disk
- looking up the location of a file on disk
- reading all or part of an existing file from disk
- editing part of an existing file from disk
- creating folders on disk
- getting the contents of folders on disk
- ...

# Lecture Plan

- Data Storage and Access
- Filesystem goals
- **Case Study: The Unix v6 Filesystem**
  - Sectors/Blocks
  - Inodes
  - Large files
- Practice

# Case Study: Unix V6 Filesystem

We will use the Unix Version 6 Filesystem to see an example of filesystem design.

- From around 1975; well-designed, open-source filesystem
- Great example of a well-thought-out, layered engineering design
- Not the only filesystem design - each has tradeoffs. Modern file systems (particularly for Linux) are, in general, descendants of this file system, but they are more complex and geared towards high performance and fault tolerance.
- Details we discuss (e.g. "size of a sector") are specific to this filesystem design, but general principles apply to modern operating systems
- Some other filesystems are open source and viewable if you're interested (e.g., the ext4 file system, which is the most common Linux file system right now)
- Our discussion will highlight various design questions as we go. Consider the pros/cons of this approach vs. alternatives!

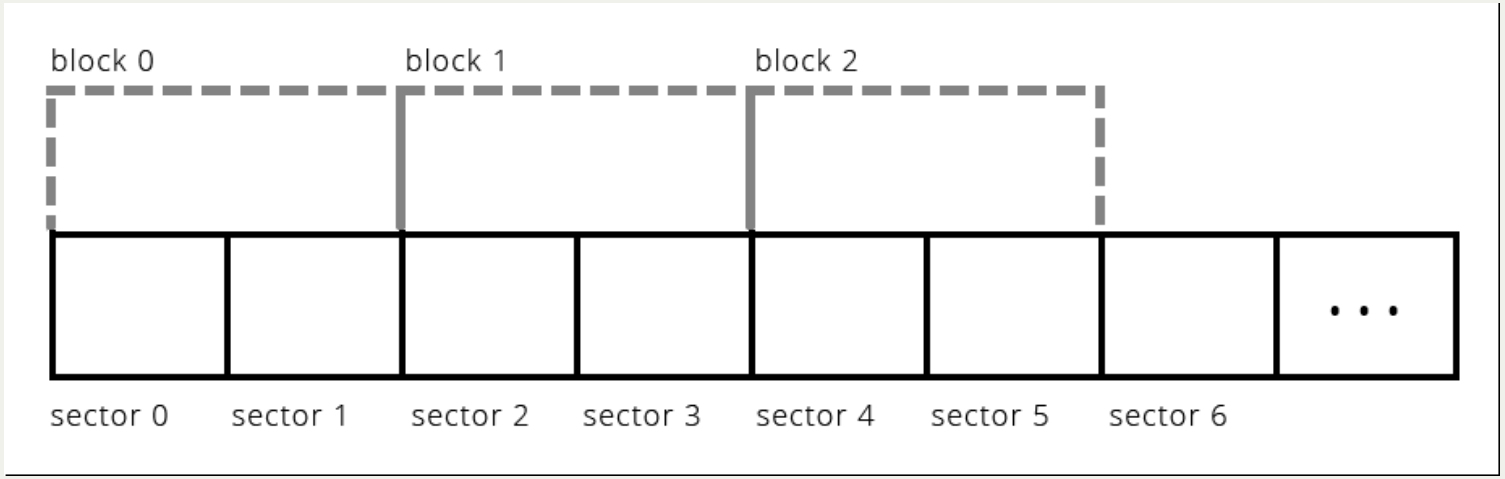
# Sectors and Blocks

A filesystem generally defines its own unit of data, a "block," that it reads/writes at a time.

- "Sector" = hard disk storage unit
- "Block" = filesystem storage unit (1 or more sectors) - software abstraction

Pros of larger block size?  Smaller block size?

Example: the block size could be defined as two sectors




The Unix V6 Filesystem defines a block to be 1 sector (so they are interchangeable).

# Storing Data on Disk

Two types of data we will be working with:

1. file payload data - contents of files (e.g. text in documents, pixels in images)
2. file metadata - information about files (e.g. name, size)

**Key insight:** *both* of these must be stored on the hard disk. Otherwise, we will not have it across power-offs! (E.g. without storing metadata we would lose all filenames after shutdown). *This means some blocks must store data other than payload data.*

# Storing Data on Disk

Two types of data we will be working with:

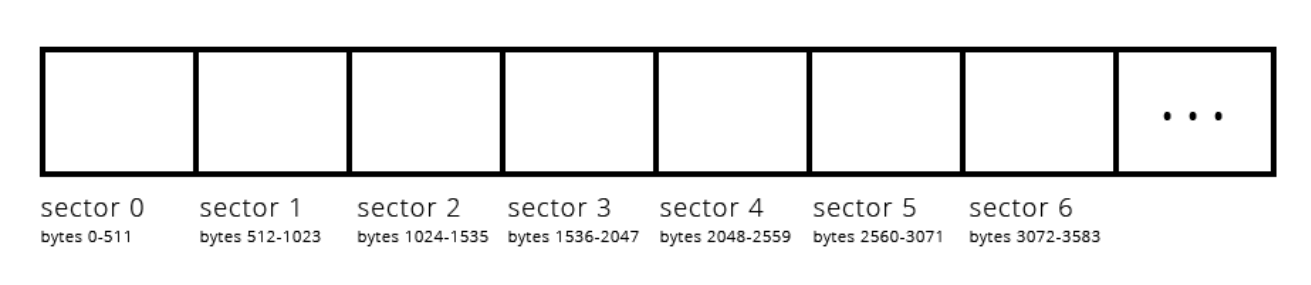
1. **file payload data - contents of files (e.g. text in documents, pixels in images)**
2. file metadata - information about files (e.g. name, size)

**Key insight:** *both* of these must be stored on the hard disk. Otherwise, we will not have it across power-offs! (E.g. without storing metadata we would lose all filenames after shutdown). *This means some blocks must store data other than payload data.*

# File Payload Data

Two types of data we will be working with:

1. **file payload data - contents of files (e.g. text in documents, pixels in images)**
2. file metadata - information about files (e.g. name, size)



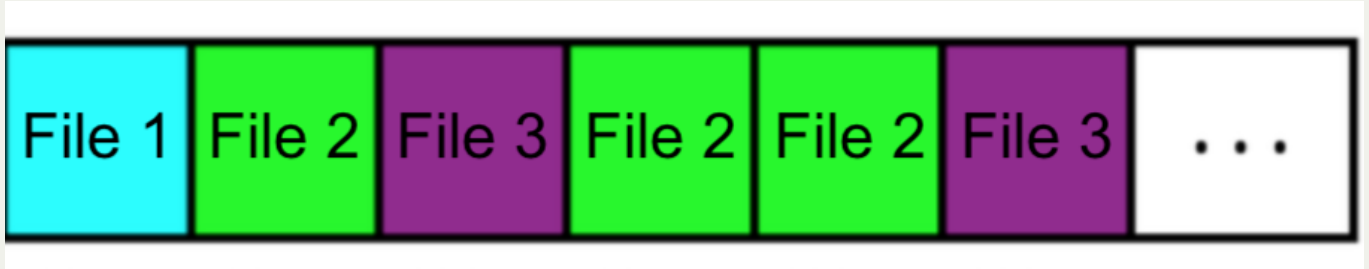

Design questions to consider:

- how do we handle small files < 512 bytes?
- for files spanning multiple blocks, must their blocks be adjacent?

# File Payload Data

Design questions to consider:

- how do we handle small files < 512 bytes? **Still reserve entire block (most do this)**
  - reserving partial blocks may better utilize space, but more complex to implement
- for files spanning multiple blocks, must their blocks be adjacent? **No.**

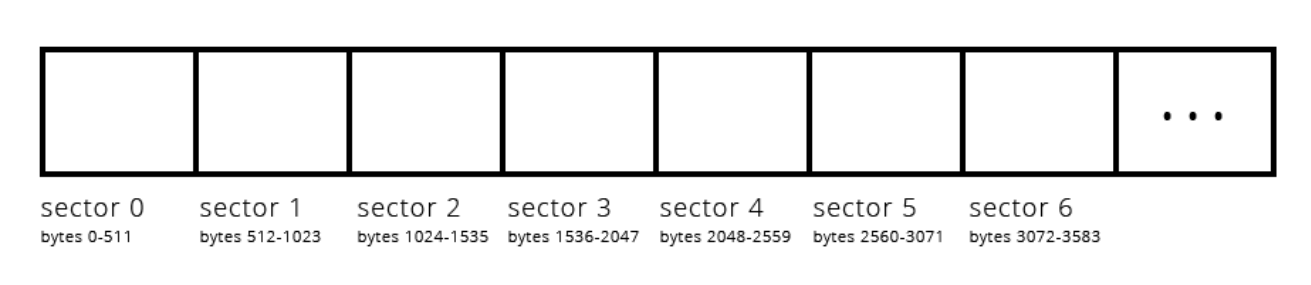



**Problem:** how do we know what block numbers store a given file's data?

# Storing Data on Disk

Two types of data we will be working with:

1. file payload data - contents of files (e.g. text in documents, pixels in images)
2. **file metadata - information about files (e.g. name, size)**




**Problem:** how do we know what block numbers store a given file's data?

We need somewhere to store information about each file, such as which block numbers store its payload data. Ideally, this data would be easy to look up as needed.

# Inodes

An **inode** ("index node") is a grouping of data about a single file. It stores things like:

- file size
- ordered list of block numbers that store file payload data

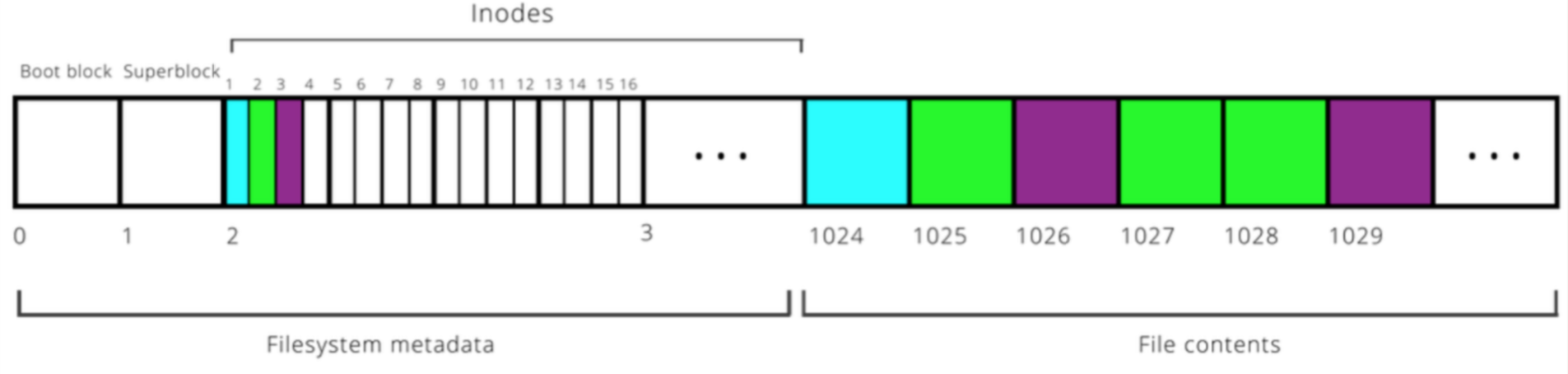
The full definition of an inode has much more; but we focus just on size (**i_size0** and **i_size1**) and block numbers (**i_addr[8]**). An inode is 32 bytes big in this filesystem.

The filesystem stores inodes on disk together in the **inode table** for quick access.

```
struct inode {
  uint16_t  i_mode;     // bit vector of file
                        //   type and permissions
  uint8_t   i_nlink;    // number of references
                        //   to file
  uint8_t   i_uid;      // owner
  uint8_t   i_gid;      // group of owner
  uint8_t   i_size0;    // most significant byte
                        //  of size
  uint16_t  i_size1;    // lower two bytes of size
                        //  (size is encoded in a
                        //  three-byte number)
  uint16_t  i_addr[8];  // device addresses
                        //   constituting file
  uint16_t  i_atime[2]; // access time
  uint16_t  i_mtime[2]; // modify time
};
```

# Inodes

- The filesystem stores inodes on disk together in the **inode table** for quick access.
- inodes are stored in a reserved region starting at block 2 (block 0 is "boot block" containing hard drive info, block 1 is "superblock" containing filesystem info). Typically at most 10% of the drive stores metadata.
- 16 inodes fit in a single block here.




Filesystem goes from **filename** to **inode number** ("inumber") to **file data**. (Demo time!)

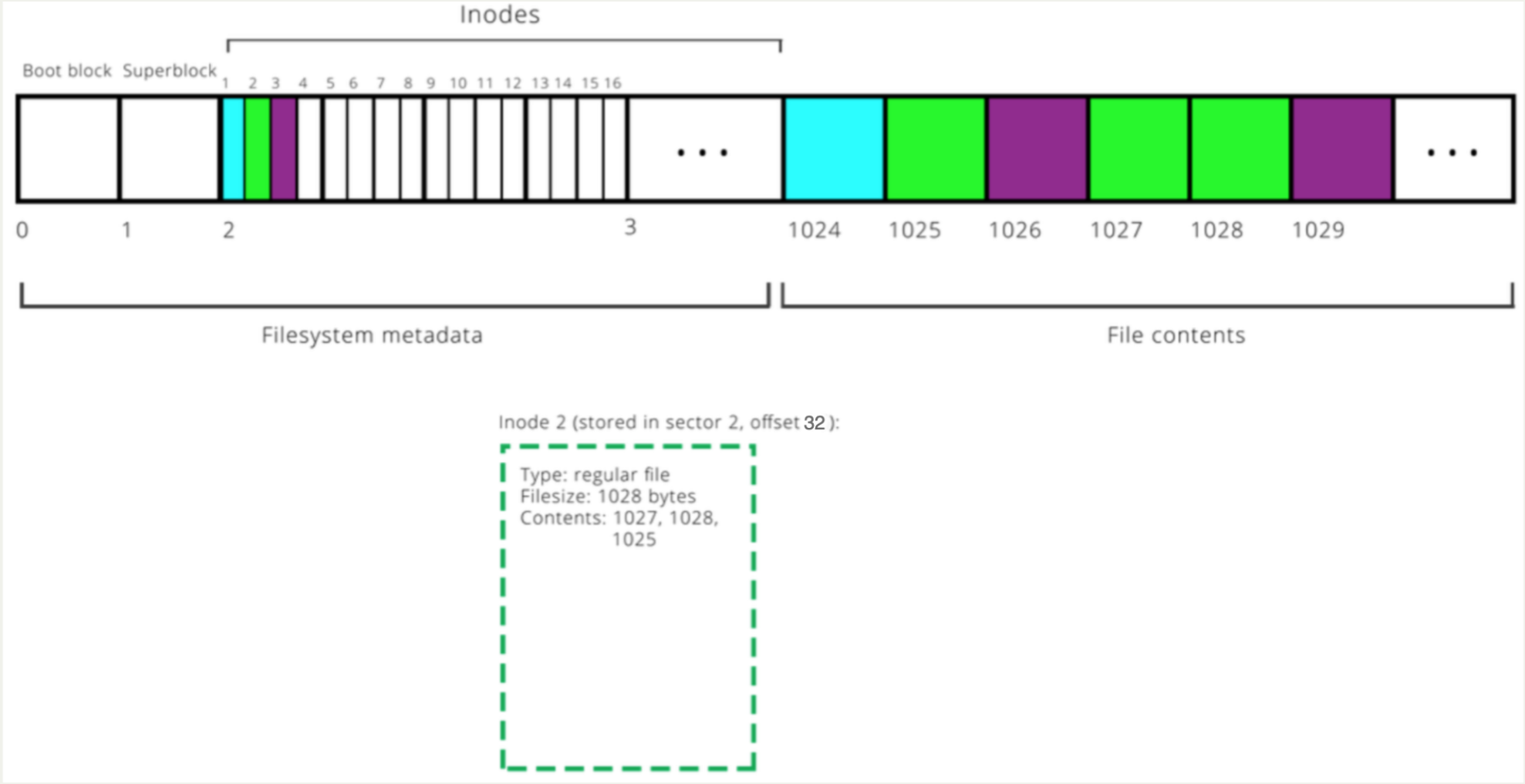
# Inodes

We need inodes to be a fixed size, and not too large. So how should we store the block numbers? How many should there be?

1. if variable number, there's no fixed inode size
2. if fixed number, this limits maximum file size

**The inode design here has space for 8 block numbers.** But we will see later how we can build on this to support very large files.

# Inodes


Inodes
Boot block
Superblock
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
0
1
2
3
1024
1025
1026
1027
1028
1029
Filesystem metadata
File contents
Inode 2 (stored in sector 2, offset 32 ):
Type: regular file
Filesize: 1028 bytes
Contents: 1027, 1028, 1025

# Practice #1: Inodes

Let's say we have an inode with the following information (remember 1 block = 1 sector = 512 bytes):

**file size: 600 bytes**

**block numbers: 56, 122**

How many bytes of block 56 store file payload data?

How many bytes of block 122 store file payload data?

# Practice #2: Inodes

Let's say we have an inode with the following information (remember 1 block = 1 sector = 512 bytes):

**file size: 2000 bytes**

**block numbers: 56, 122, 45, 22**

Which block number stores the 2000th byte of the file?

Which block number stores the 1500th byte of the file?

*Bytes 0-511 reside within block 56, bytes 512-1023 within block 122, bytes 1024-1535 within block 45, and bytes 1536-1999 at the front of block 22.*

**Note:** inodes live on disk. But we can read them into memory where we can represent them as structs.

# Inodes

Let's imagine that the hard disk creators provide software to let us interface with the disk.

```
void readSector(size_t sectorNumber, void *data);
void writeSector(size_t sectorNumber, const void *data);
```

How do we access inodes? Here are some examples:

```
typedef struct inode {
  uint16_t  i_addr[8];  // device addresses
                        //   constituting file
  ...
} inode;

// Loop over each inode in sector 2
inode inodes[512 / sizeof(inode)];
readSector(2, inodes);
for (size_t i = 0; i < sizeof(inodes) / sizeof(inodes[0]); i++) {
        ...
}
```

# Lecture Plan

- Data Storage and Access
- Filesystem goals
- **Case Study:** The Unix v6 Filesystem
  - Sectors/Blocks
  - Inodes
  - **Large files**
- Practice

# File Size

**Problem**: with 8 block numbers per inode, the largest a file can be is 512 * 8 = 4096 bytes (~4KB). That definitely isn't realistic!

Let's say a file's payload is stored across 10 blocks:

*45, 42, 15, 67, 125, 665, 467, 231, 162, 136*

Assuming that the size of an inode is fixed, where can we put these block numbers?

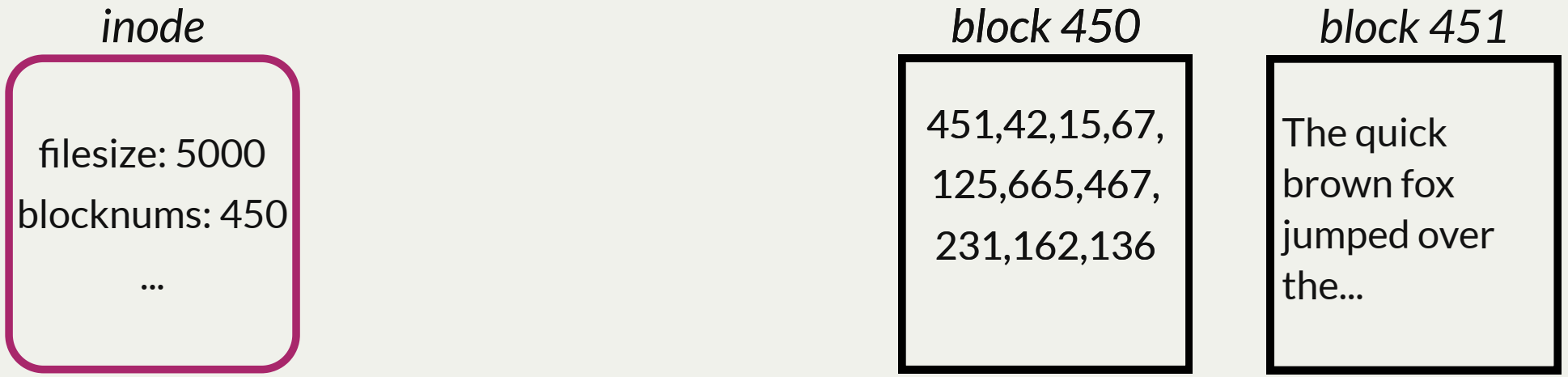
**Solution:** let's store them *in a block*, and then store *that* block's number in the inode!

# Indirect Addressing

Let's say a file's payload is stored across 10 blocks:

*451, 42, 15, 67, 125, 665, 467, 231, 162, 136*

**Solution:** let's store them *in a block*, and then store *that* block's number in the inode! This approach is called *indirect addressing*.

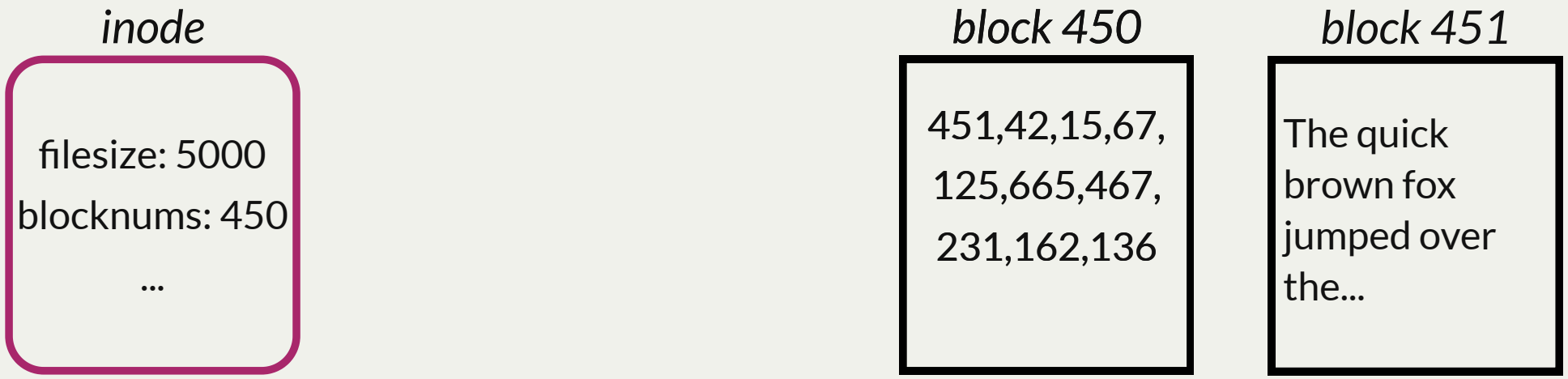


# Indirect Addressing

Design questions:

- should we make *all* the block numbers in an inode use indirect addressing?
- should we use this approach for all files, or just large ones?

*Indirect addressing is useful, but means that it takes more steps to get to the data, and we may use more blocks than we need.*

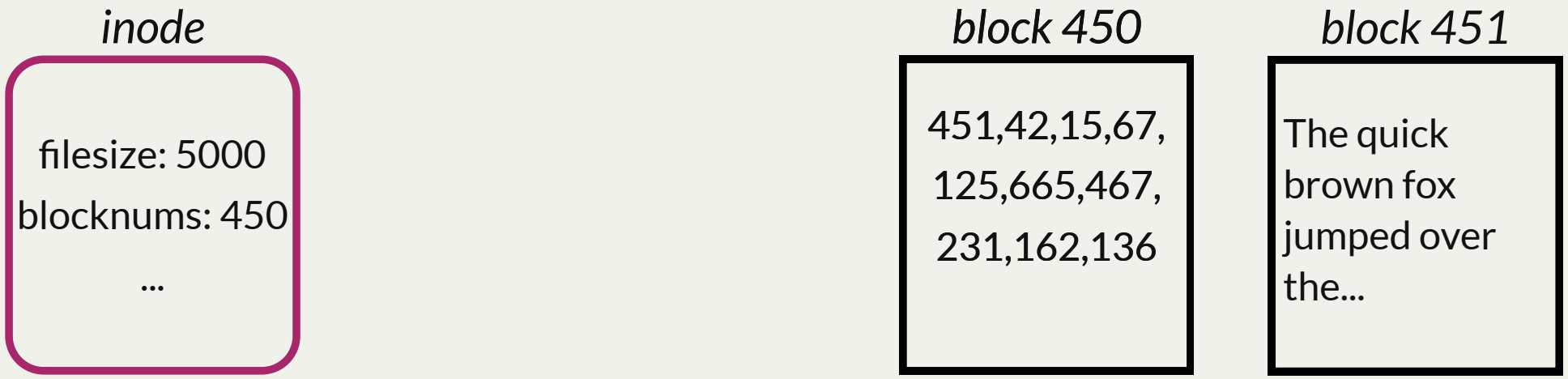


# Indirect Addressing

Design questions:

- should we make *all* the block numbers in an inode use indirect addressing? **just some.**
- should we use this approach for all files, or just large ones? **just large ones.**

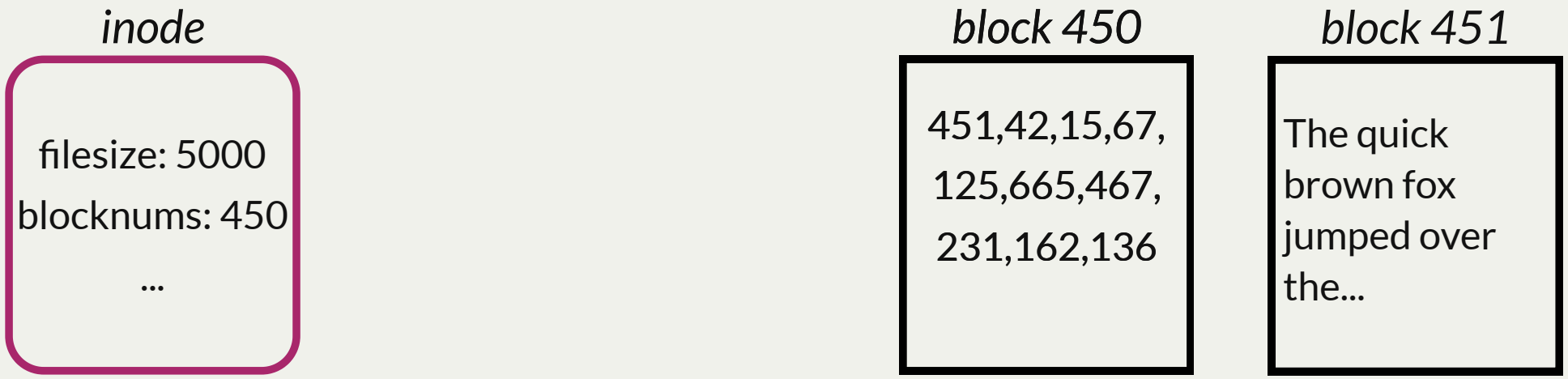
*Indirect addressing is useful, but means that it takes more steps to get to the data, and we may use more blocks than we need.*



# Singly-Indirect Addressing

The Unix V6 filesystem uses *singly-indirect addressing* (blocks that store payload block numbers) just for large files.

- check flag or size in inode to know whether it is a small file (direct addressing) or large one (indirect addressing)
  - If small, each block number in the inode stores payload data
  - If large, **first 7 block numbers** in the inode stores block numbers for payload data
  - 8th block number? we'll get to that :)



# Singly-Indirect Addressing

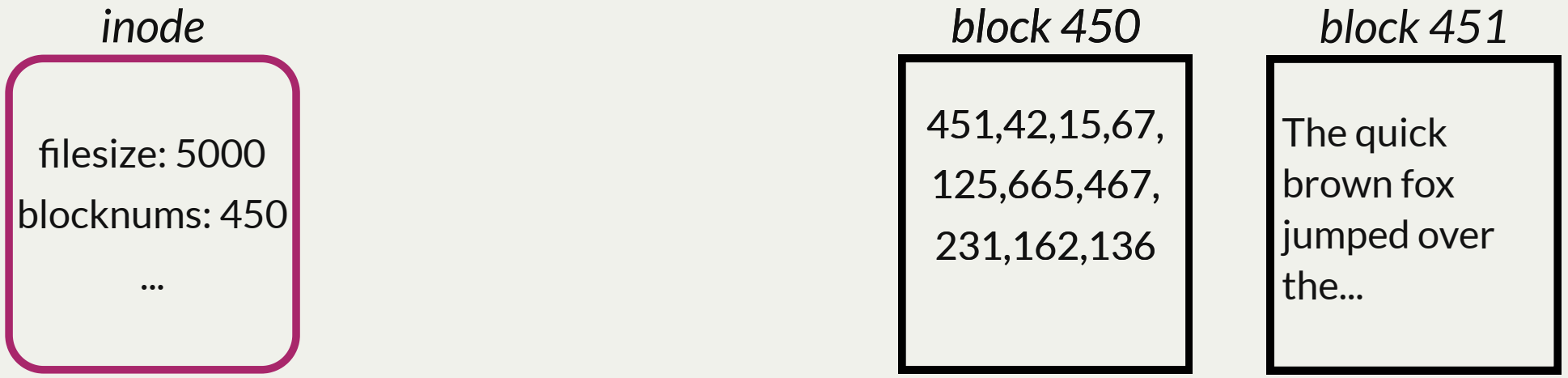
Let's assume for now that an inode for a large file uses all 8 block numbers for singly-indirect addressing. What is the largest file size this supports? Each block number is 2 bytes big.

*8* block numbers in an inode x

*256* block numbers per singly-indirect block x

*512* bytes per block

= ~1MB



# Practice: Singly-Indirect Addressing

Let's say we have an inode with the following information (remember 1 block = 1 sector = 512 bytes, and block numbers fit i):

**file size: 200,000 bytes**

**block numbers: 56, 122**

Which singly-indirect block stores the block number holding the 150,000th byte of the file?

*Bytes 0-131,071 reside within blocks whose block numbers are in block 56. Bytes 131,072 (256*512) - 199,999 reside within blocks whose block numbers are in block 122.*

# File Size

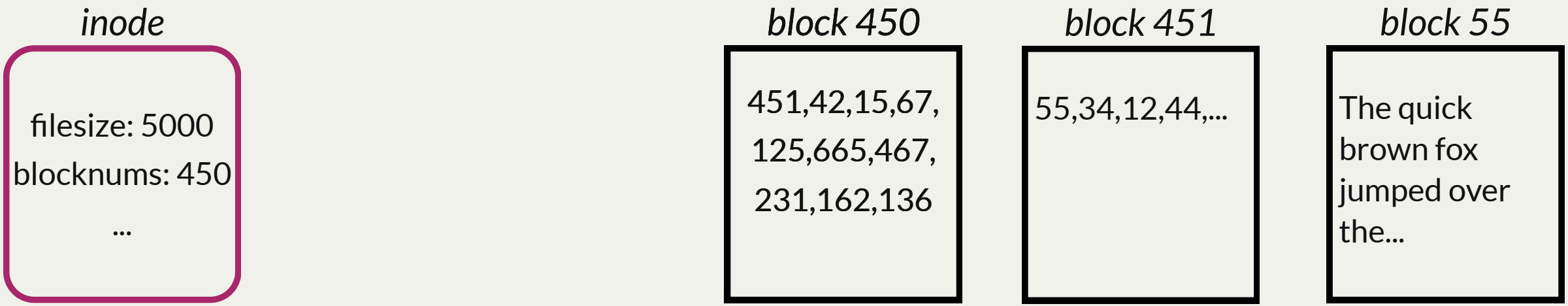
**Problem**: even with singly-indirect addressing, the largest a file can be is 8 * 256 * 512 = 1,048,576 bytes (~1MB). That still isn't realistic!

**Solution:** let's use *doubly-indirect addressing*; store a block number for a block that contains *singly-indirect block numbers*.

# File Size

**Solution:** let's use *doubly-indirect addressing*; store a block number for a block that contains *singly-indirect block numbers*.

Allows even larger files, but data takes even more steps to access. How do we employ this idea?



# Indirect Addressing

The Unix V6 filesystem uses *singly-indirect addressing* (blocks that store payload block numbers) just for large files.  It also uses *doubly-indirect addressing* (blocks that store singly-indirect block numbers).

- check flag or size in inode to know whether it is a small file (direct addressing) or large one (indirect addressing)
  - If small, each block number in the inode stores payload data
  - If large, **first 7 block numbers** in the inode stores block numbers for payload data
  - NEW: If large, **8th block number** in the inode stores singly-indirect block numbers

# Indirect Addressing

- If small, each block number in the inode stores payload data
- If large, **first 7 block numbers** in the inode stores block numbers for payload data
- If large, **8th block number** in the inode stores singly-indirect block numbers

**In other words; a file can be represented using at most 256 + 7 = 263 singly-indirect blocks. The first seven are stored in the inode. The remaining 256 are stored in a block whose block number is stored in the inode.**

# Indirect Addressing

An inode for a large file stores 7 singly-indirect block numbers and 1 doubly-indirect block number. What is the largest file size this supports? Each block number is 2 bytes big.

*263* singly-indirect block numbers total x

*256* block numbers per singly-indirect block x

*512* bytes per block

= ~34MB

# Indirect Addressing

An inode for a large file stores 7 singly-indirect block numbers and 1 doubly-indirect block number. What is the largest file size this supports? Each block number is 2 bytes big.

OR:

*(7 * 256 * 512) + (256 * 256 * 512) ~ 34MB*

*(singly indirect) + (doubly indirect )*

Better! still not sufficient for today's standards, but perhaps in 1975. Moreover, since block numbers are 2 bytes, we can number at most 2^16 - 1 = 65,535 blocks, meaning the entire filesystem can be at most 65,535 * 512 ~ 32MB.

# Indirect Addressing Summary

- If small (<= 4096 bytes), each block number in the inode stores payload data
- If large:
  - **first 7 block numbers** in the inode stores block numbers for payload data
  - **8th block number** in the inode stores singly-indirect block numbers

Not all the block numbers may be used. E.g.

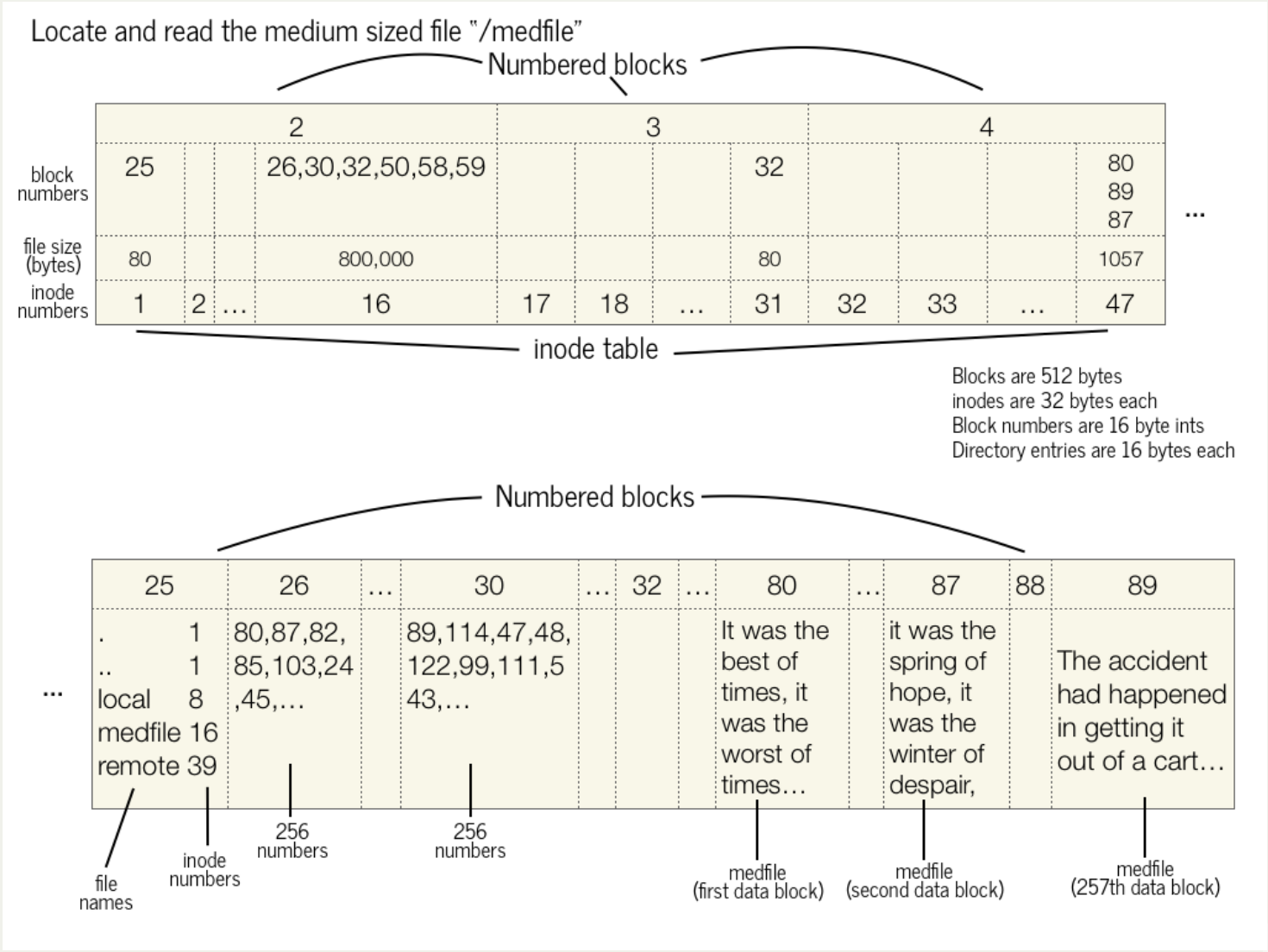
- 8th block number may be unused
- Or only the first X singly-indirect block numbers may be used
- Or a singly-indirect block may not be completely filled with block numbers

# Lecture Plan

- Data Storage and Access
- Filesystem goals
- **Case Study:** The Unix v6 Filesystem
  - Sectors/Blocks
  - Inodes
  - Large files
- **Practice**

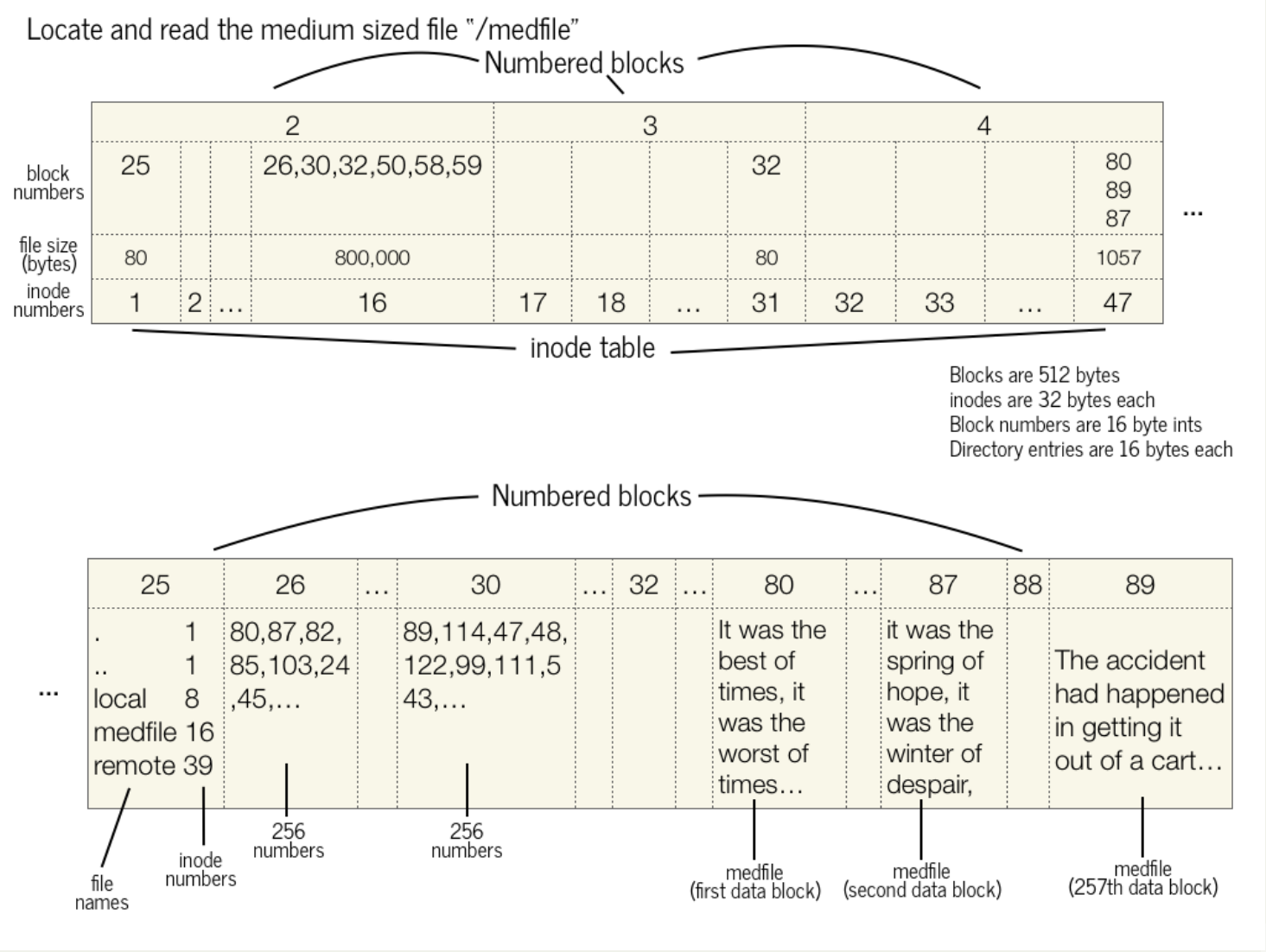
# Unix V6 Filesystem Practice #1

Assume we have a large file with inumber 16. How do we find the block containing the start of its payload data? How about the remainder of its payload data?



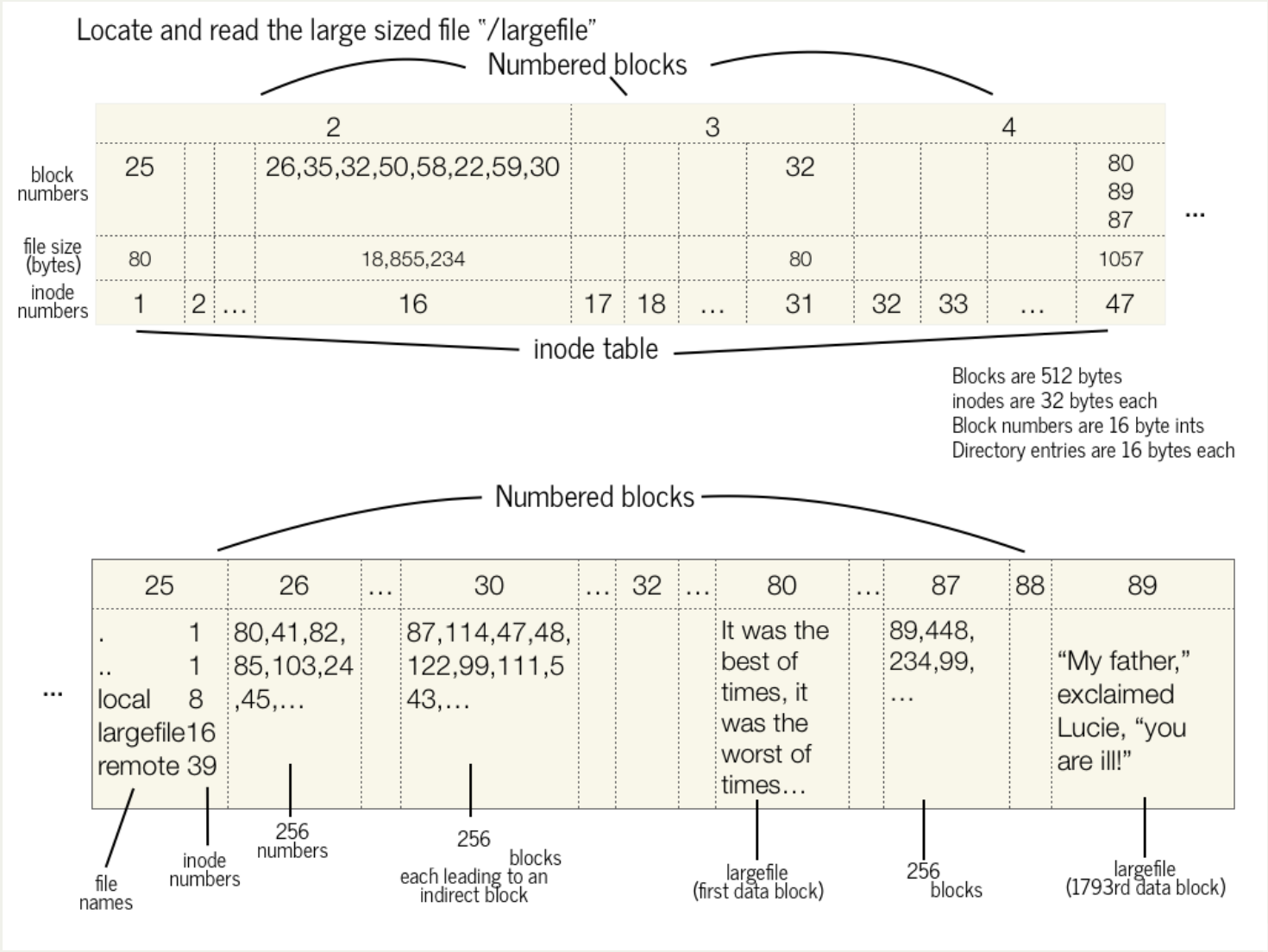
# Unix V6 Filesystem Practice #1

1. Go to block 26, and start reading block numbers. For the first number, 80, go to block 80 and read the beginning of the file (the first 512 bytes). Then go to block 87 for the next 512 bytes, etc.
2. After 256 blocks, go to block 30, and follow the 256 block numbers to 89, 114, etc. to read the 257th-511th blocks of data.
3. Continue with all indirect blocks, 32, 50, 58, 59 to read all 800,000 bytes.



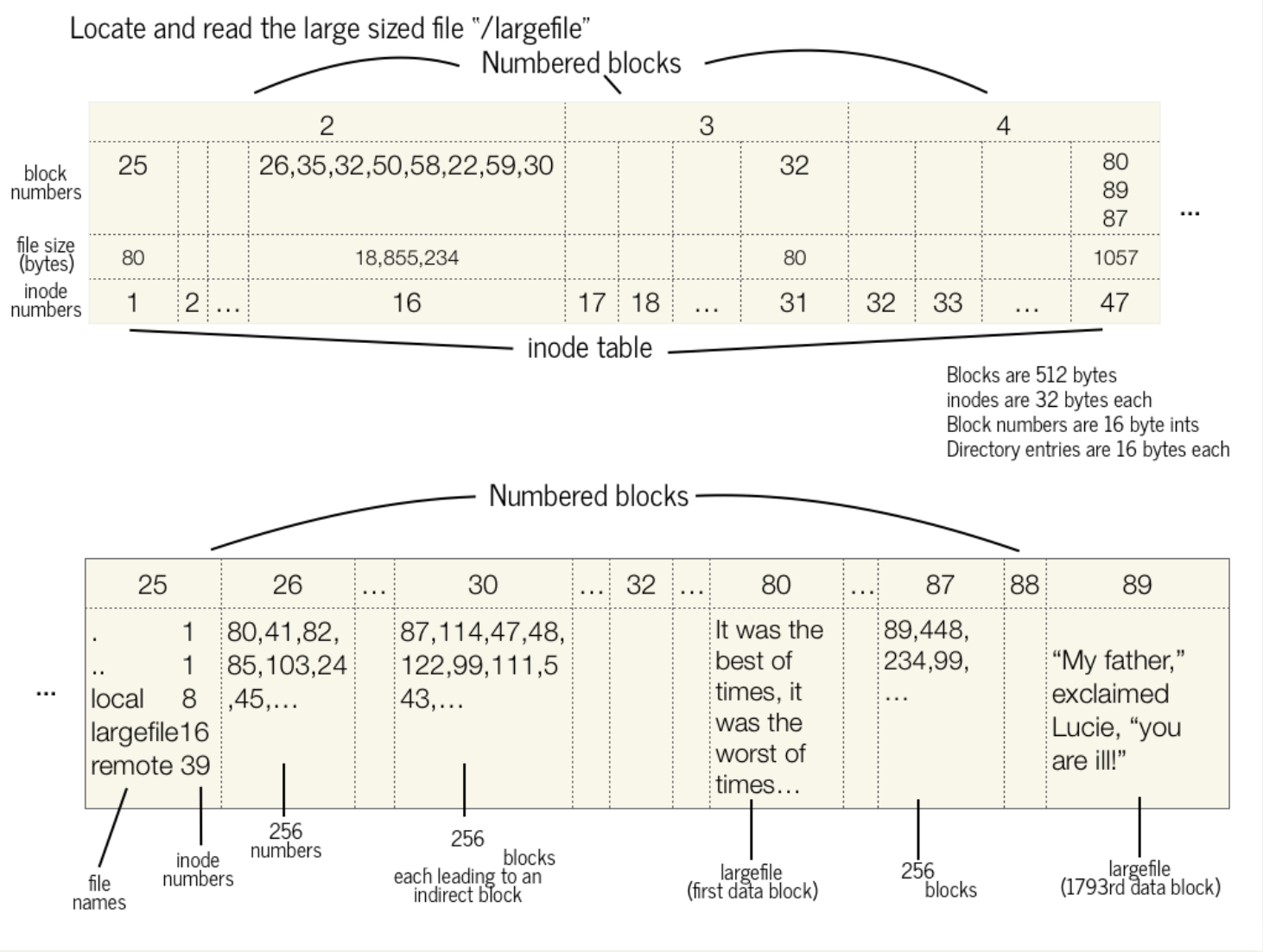
# Unix V6 Filesystem Practice #2

Assume we have a large file with inumber 16. How do we find the block containing the start of its payload data? How about the remainder of its payload data?



# Unix V6 Filesystem Practice #2

1. Go to block 26, and start reading block numbers. For the first number, 80, go to block 80 and read the beginning of the file (the first 512 bytes). Then go to block 41 for the next 512 bytes, etc.
2. After 256 blocks, go to block 35, repeat the process. Do this a total of 7 times, for blocks 26, 35, 32, 50, 58, 22, and 59, reading 1792 blocks.
3. Go to block 30, which is a doubly-indirect block. From there, go to block 87, which is an indirect block. From there, go to block 89, which is the 1793rd block.



# Recap

- Data Storage and Access
- Filesystem goals
- **Case Study:** The Unix v6 Filesystem
  - Sectors/Blocks
  - Inodes
  - Large files
- Practice

**Next time:** how can we update our filesystem design to support directories?