CS111, Lecture 2
Introduction to Filesystems

Optional reading:

Operating Systems: Principles and Practice (2nd Edition): Chapter 11, Section 12.1, 12.2 and Section 13.3 (up through page 567)

While you're waiting – get set up with PollEverywhere!

Visit pollev.stanford.edu to set up your account.

😷 masks required

PollEverywhere

- Today we're doing a "trial run" of using PollEverywhere for poll questions
  - Not counted for attendance (that starts Friday), just a chance to get a feel for the system
  - Participation info posted on Canvas Gradebook after lecture so you can confirm your responses were recorded
- Responses not anonymized, but we don't look at specific responses, just aggregated results and participation totals
- Visit pollev.stanford.edu to log in (or use the PollEverywhere app) and sign in with your **@stanford.edu email – NOT your personal email!**
- You can use any device with a web browser, or download the PollEverywhere app, or respond via text – **however, to respond via text you must first log in via a web browser and add your phone number to your profile.**
- Whenever we reach a poll question in the slides, it will automatically activate the poll and allow you to respond.

How are you doing? (This is an open ended question, answer however you like!)

Announcements

- Remember to input your section preferences by 5PM Sat! Link is on the course website (under “Sections”).
- Helper Hours scheduled and starting this week!
- Please let us know about OAE accommodations and midterm conflicts as soon as you can

Topic 1: Filesystems - How can we design filesystems to manage files on disk, and what are the tradeoffs inherent in designing them? How can we interact with the filesystem in our programs?

CS111 Topic 1: Filesystems

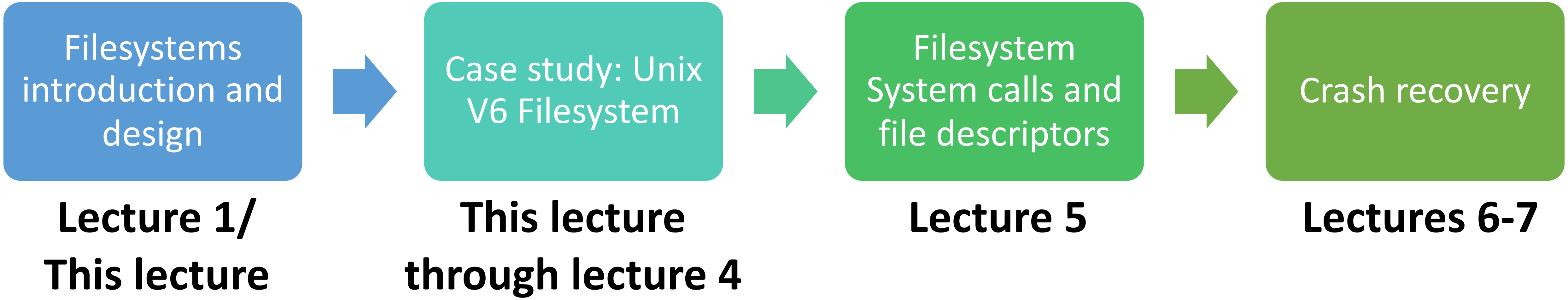
Filesystems - *How can we design filesystems to manage files on disk, and what are the tradeoffs inherent in designing them? How can we interact with the filesystem in our programs?*

Why is answering this question important?

- Helps us understand what filesystems do (today and next time)
- Provides insight into the challenges and tradeoffs in designing large systems (next few lectures)
- Shows us how we can directly manipulate files in our programs (next week)

assign1: implement layers of the Unix v6 filesystem to read a file from disk given its path.

CS111 Topic 1: Filesystems


Filesystems introduction and design
Lecture 1/ This lecture
Case study: Unix V6 Filesystem
This lecture through lecture 4
Filesystem System calls and file descriptors
Lecture 5
Crash recovery
Lectures 6-7

assign1: implement portions of the Unix v6 filesystem!

Learning Goals

- Understand the key responsibilities and requirements of a filesystem
- Get practice identifying tradeoffs in different filesystem designs
- Explore the design of the Unix V6 filesystem

Plan For Today

- Filesystems Introduction
- Methods for Storing Files
  - Contiguous Allocation
  - Linked Files
  - Windows FAT
  - Multi-level indexes
- The Unix V6 Filesystem
  - Inodes

Plan For Today

- **Filesystems Introduction**
- Methods for Storing Files
  - Contiguous Allocation
  - Linked Files
  - Windows FAT
  - Multi-level indexes
- The Unix V6 Filesystem
  - Inodes

Filesystems

A **filesystem** is the portion of the OS that manages the disk.

- A hard drive (or, more commonly these days, flash storage) is persistent storage – it can store data between power-offs.

Memory (RAM)	Disk
• Fast, but less space • *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	• Slower, but more space • *Sector-addressable:* cannot read/write just one byte of data – can only read/write "sectors" of data at a time • Persistent: stores data between power-offs

Filesystem Functionality

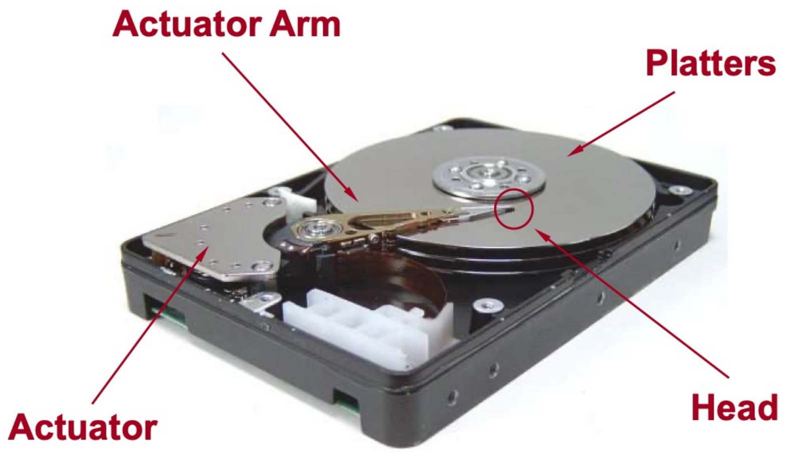
We want to read/write file on disk and have them persist even when the device is off. This may include operations like:

- creating a new file on disk
- looking up the location of a file on disk
- Reading/editing all or part of an existing file from disk – e.g., <u>sequential</u>/<u>random access</u>
- creating folders on disk
- getting the contents of folders on disk
- …

Hard Drives

Magnetic disks (hard drives) have been the standard storage mechanism for files.

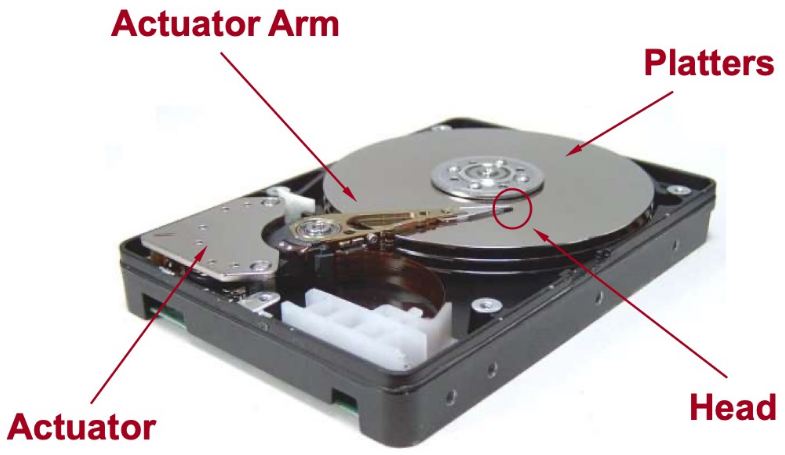
- Spinning, magnetically-coated platters
- Actuator arm positions *heads*, which can read and write data on the magnetic surfaces
- Moving parts means risk of damage from sudden movement, dust, etc.

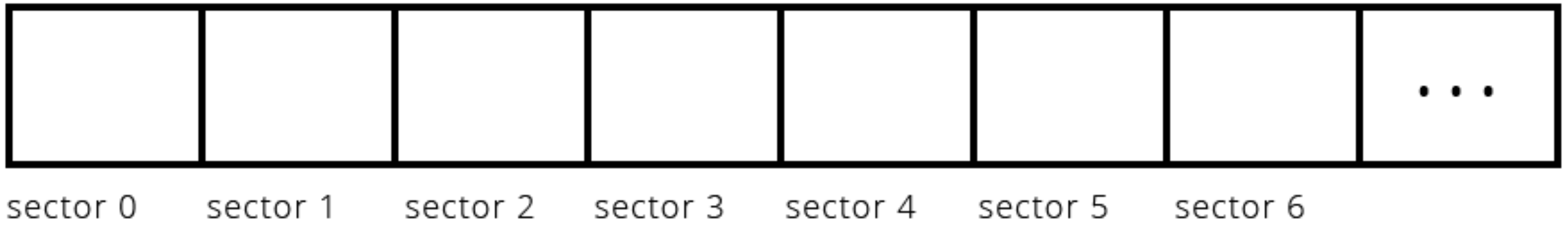


Hard Drives

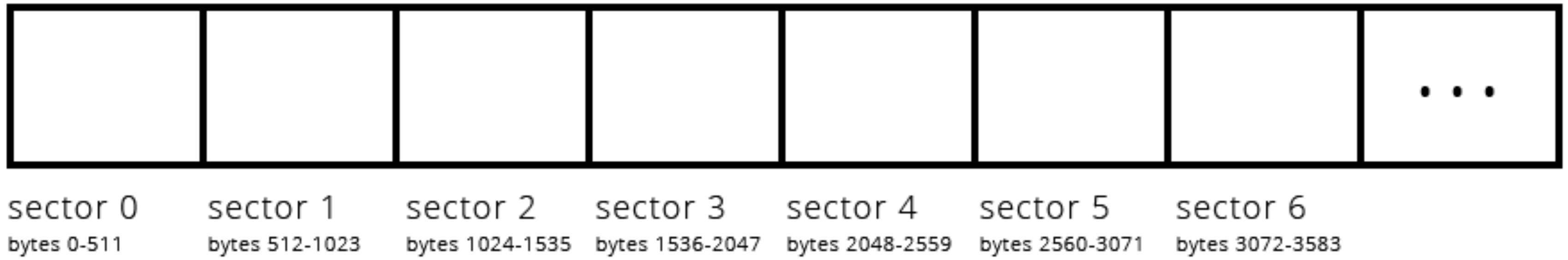
Hard drives have peculiar performance characteristics that have a big impact on how we build filesystems.

- Reading and writing requires *seeking* (moving arm to position heads over desired track) and waiting for desired location to pass underneath. Want to minimize this time.
- We can only read data in chunks of *sectors.* Example of *virtualization*; making one thing look like another.





Hard Disks are Sector-Addressable



If we are the OS, the hard disk creators might provide this API (“application programming interface”) – a set of public functions - to 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 (the OS) must build a filesystem by layering functions on top of these to ultimately allow us to read, write, lookup, and modify entire files.

Filesystems

Functions for user programs to read/write files

Filesystem

`readSector` *and* ***`writeSector`***

Filesystem Challenges

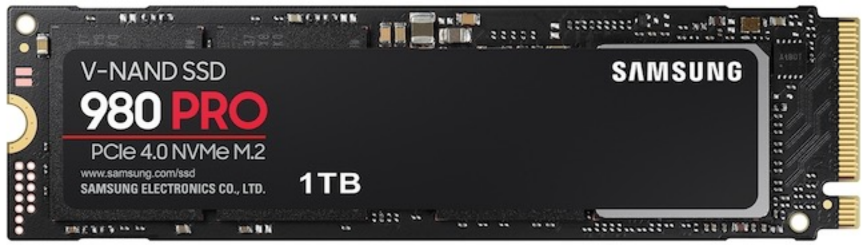
Problems addressed by modern file systems:

- **Disk space management**:
  - Fast access to files (minimize seeks)
  - Sharing space between users
  - Efficient use of disk space
- **Naming**: how do users select files?
- **Reliability**: information must survive OS crashes and hardware failures.
- **Protection**: isolation between users, controlled sharing.

Flash Storage

Recently, flash storage ("SSD") has become more popular and commonplace, especially with the growth in mobile devices.

- Much faster (100x faster access), but more expensive (5-10x higher cost/bit than disk)
- No moving parts, so more reliable
- Issues with *wear-out*; once a chunk of the drive has been erased many times (~100k), it no longer stores info reliably.
- Typically, still only support reading/writing in units of sectors.



https://www.samsung.com/us/computing/memory-storage/solid-state-drives/980-pro-pcie-4-0-nvme-ssd-1tb-mz-v8p1t0b-am/

Plan For Today

- Filesystems Introduction
- **Methods for Storing Files**
    - Contiguous Allocation
    - Linked Files
    - Windows FAT
    - Multi-level indexes
- The Unix V6 Filesystem
    - Inodes

Sectors and Blocks

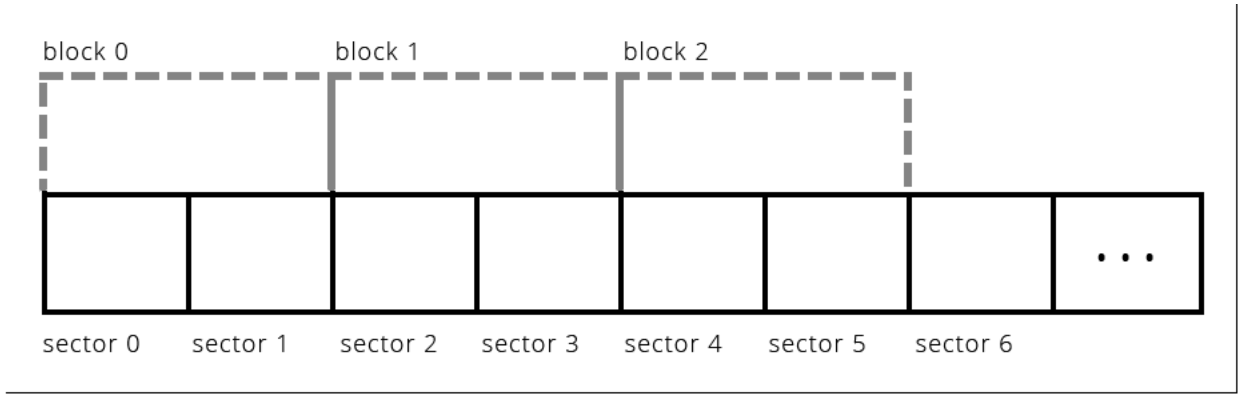
A filesystem generally defines its own unit of data, a "block," that it reads/writes at a time.

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

Pros of larger block size? Smaller block size?

- E.g. fewer transfer operations if larger, but smaller files may read in more data than necessary

Example: the block size could be defined as two sectors



Storing Files 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* 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 Files on Disk

Two types of data we will be working with:

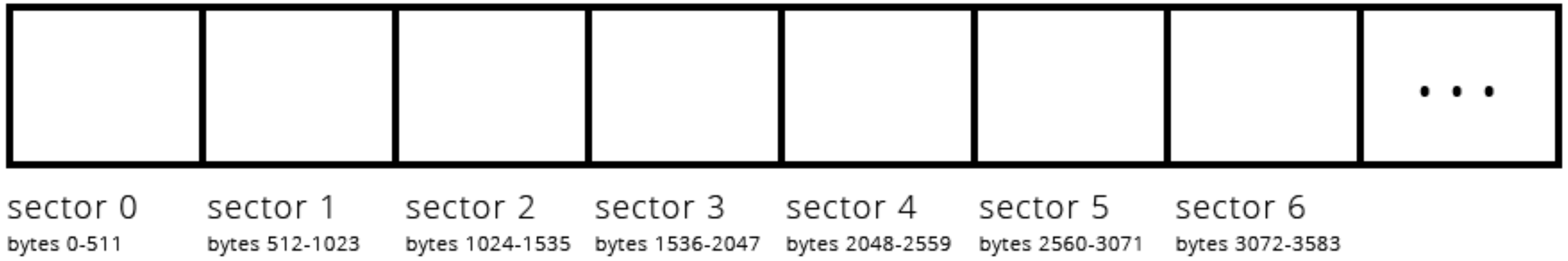
1. **file payload data - contents of files (e.g. text in documents, pixels in images)**
2. file metadata - information about files (e.g. name, size)

Key insight: *both* 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.*

Contiguous Allocation

First key question: should we store files contiguously on disk? What would it look like if we did?

- Called *contiguous allocation* – allocate a file in one contiguous group of blocks
- For each file, keep track of the number of its first sector and its length
- Keep a free list of unused areas of the disk
- Example: IBM OS/360
- Advantages/drawbacks?



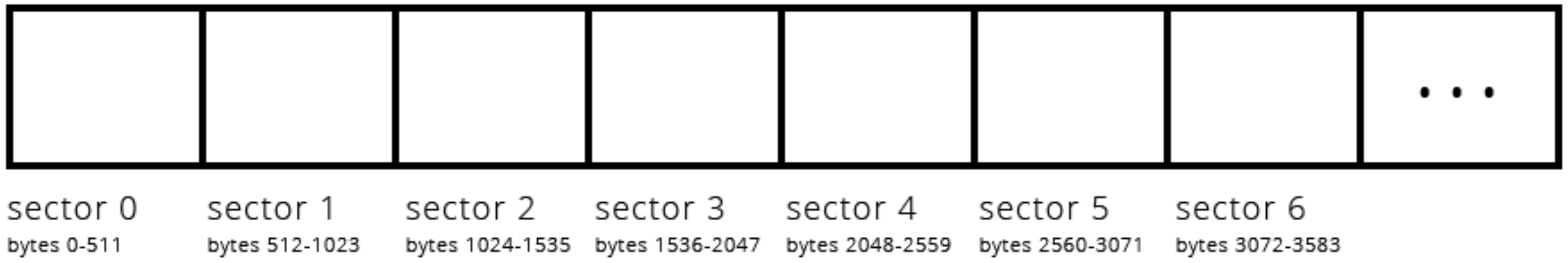
Contiguous Allocation

First key question: should we store files contiguously on disk? What would it look like if we did?

- Called *contiguous allocation* – allocate a file in one contiguous group of blocks

Advantages:

- simple
- can read sequentially or easily jump to any location in file ("random access")
- all data in one place (few seeks)



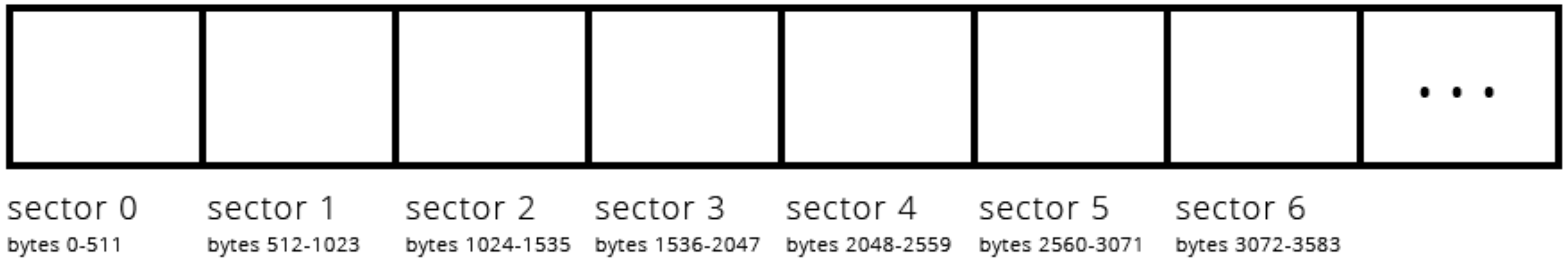
Contiguous Allocation

First key question: should we store files contiguously on disk? What would it look like if we did?

- Called *contiguous allocation* – allocate a file in one contiguous group of blocks

<u>Disadvantages:</u>

- hard to grow files
- hard to lay out files on disk – we may not be able to squeeze a new file in a block of free space (*fragmentation* – occurs when we have space on disk, but can't use it to store files)



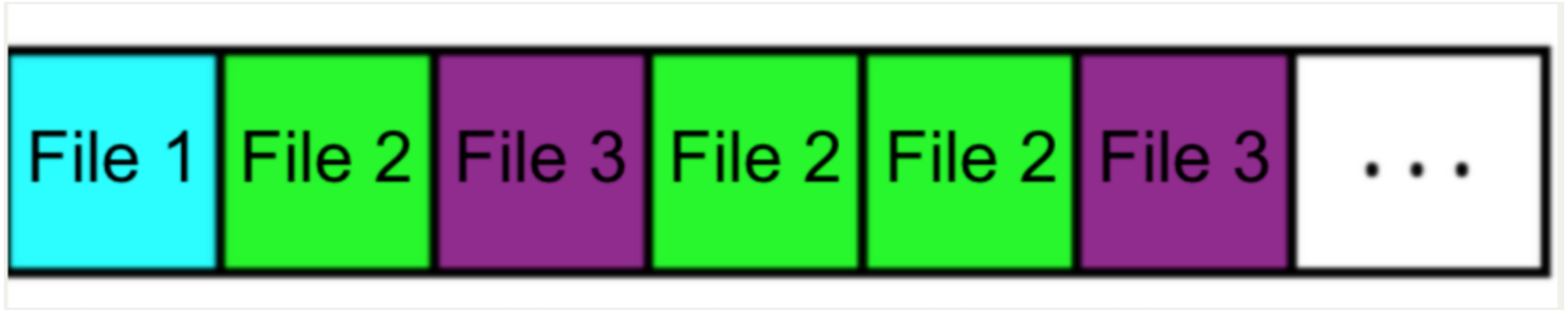
Linked Files

First key question: should we store files contiguously on disk? What would it look like if we ***didn't***?

- Problem: we need to know what blocks are associated with what files

One idea: *linked files* – like a linked list

- Each block contains file data *as well as* the location of the next block
- For each file, keep track of the number of its first block
- Approximate examples: TOPS-10, Xerox Alto
- Advantages/drawbacks?



Linked Files

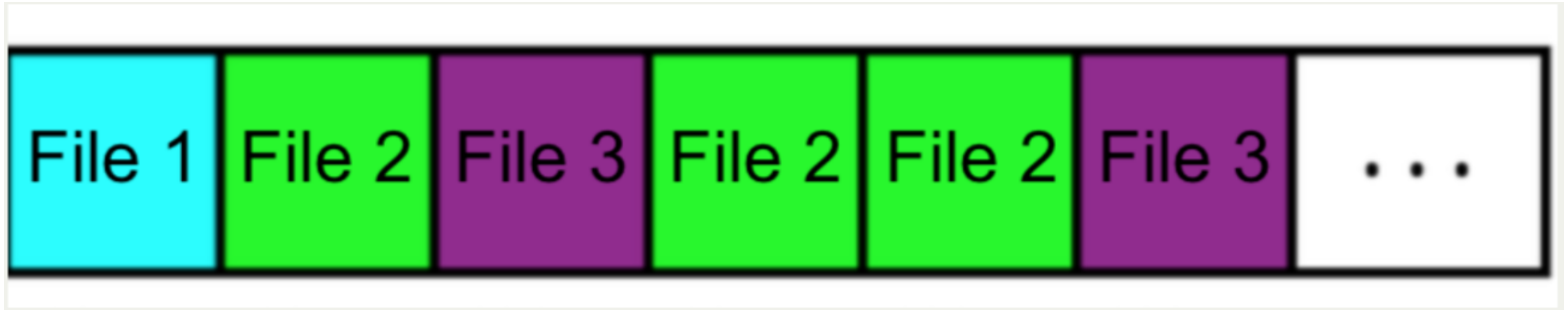
First key question: should we store files contiguously on disk? What would it look like if we ***didn't***?

One idea: *linked files* – like a linked list

- Each block contains file data *as well as* the location of the next block

Advantages:

- Easy to grow files
- Easier to fit files in available space – less *fragmentation*
- Still supports simple sequential access



Linked Files

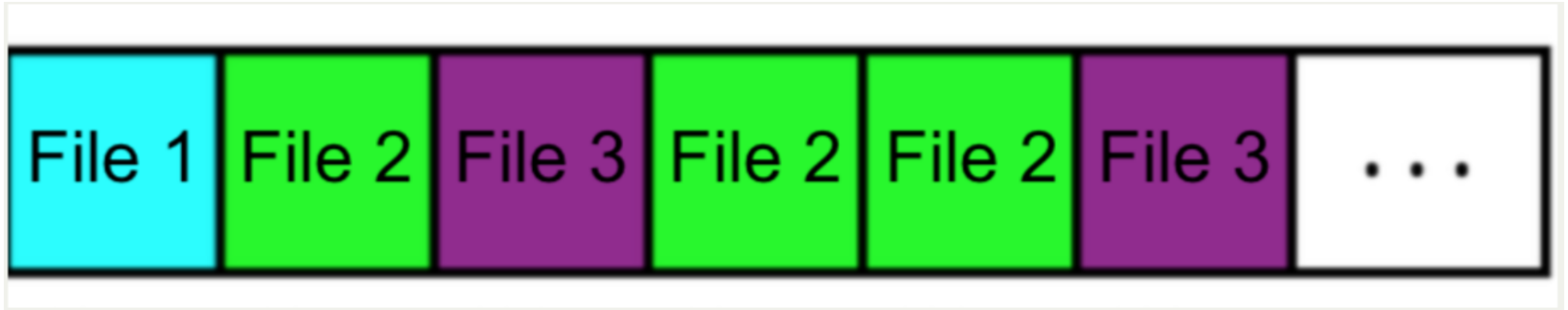
First key question: should we store files contiguously on disk? What would it look like if we ***didn't***?

One idea: *linked files* – like a linked list

- Each block contains file data *as well as* the location of the next block

Disadvantages:

- Can't easily jump to any arbitrary location in the file
- Data scattered throughout disk (more seeks)



Linked Files

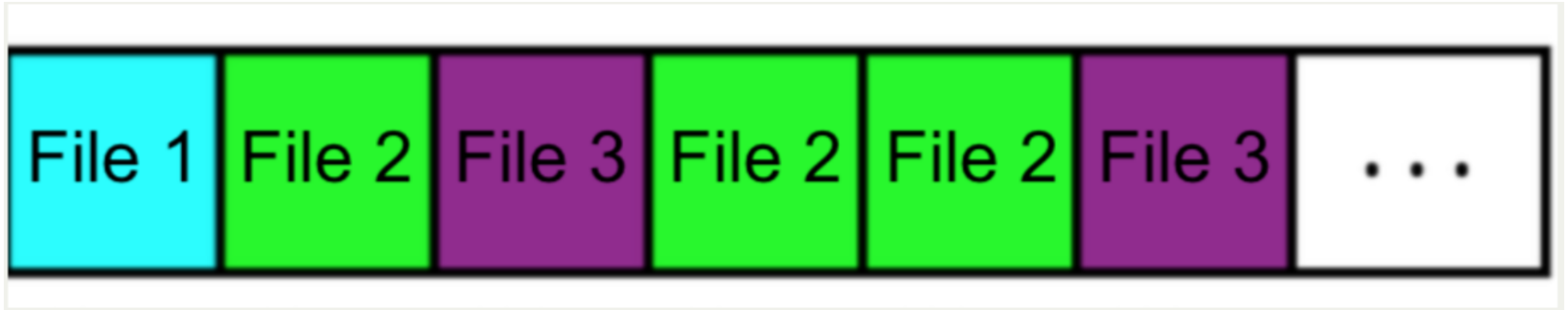
First key question: should we store files contiguously on disk? What would it look like if we ***didn't***?

One idea: *linked files* – like a linked list

- Each block contains file data *as well as* the location of the next block

Disadvantages:

- **Can't easily jump to any arbitrary location in the file**
- Data scattered throughout disk (more seeks)

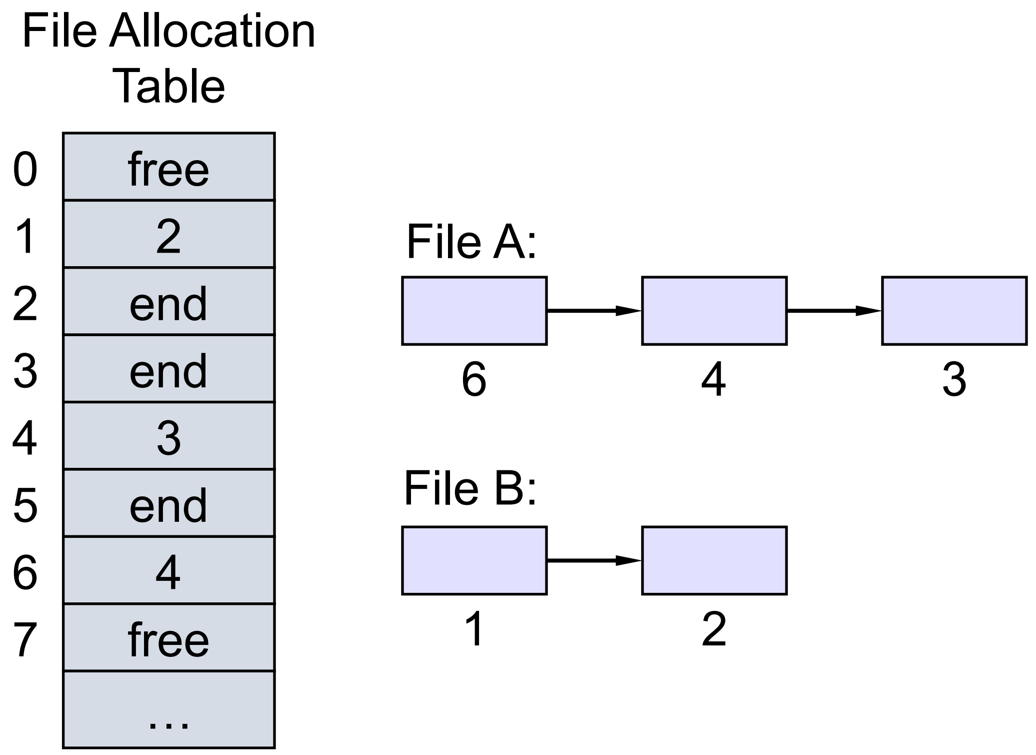


Windows FAT

First key question: should we store files contiguously on disk? What would it look like if we ***didn't***?

Interesting idea: what if we stored all the links in one big table in memory?

- *Windows (DOS) FAT*: like linked allocation, except links aren't in blocks, they are in a "file allocation table" in memory and disk
- Still keep track of each file's first block
- (Still used today for flash sticks, digital cameras, many embedded devices)
- Advantages/Disadvantages?



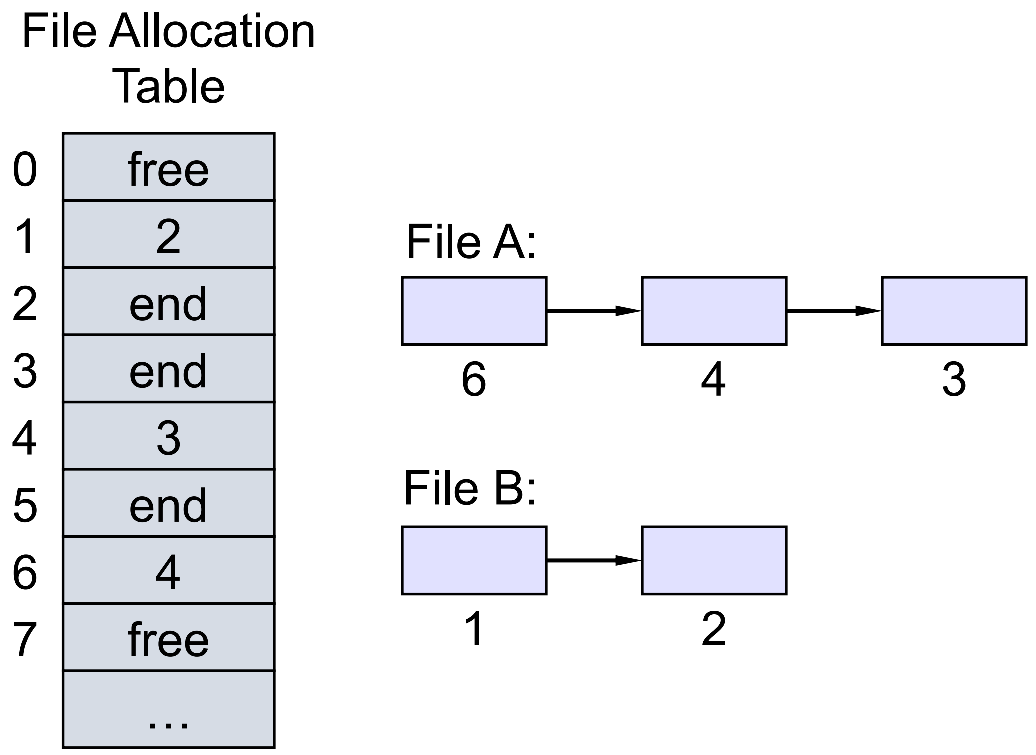
Windows FAT

First key question: should we store files contiguously on disk? What would it look like if we ***didn't***?

- *Windows (DOS) FAT*: like linked allocation, except links aren't in blocks, they are in a "file allocation table" in memory

Advantages:

- Can more quickly jump to various locations in a file
- Still supports easy sequential access



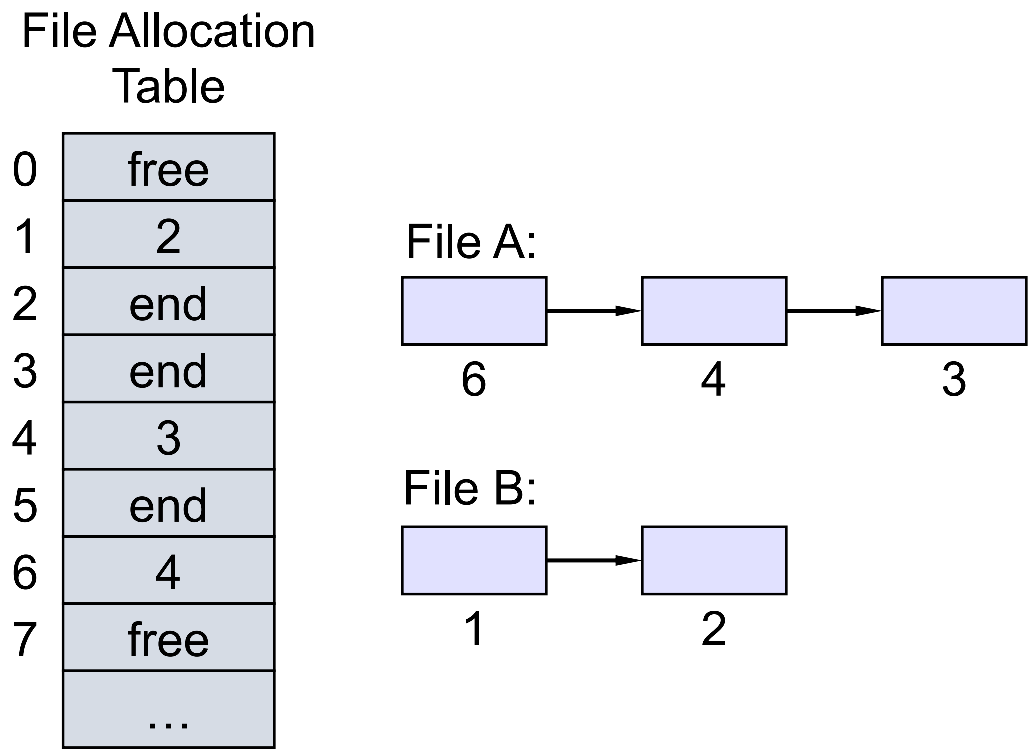
Windows FAT

First key question: should we store files contiguously on disk? What would it look like if we ***didn't***?

- *Windows (DOS) FAT*: like linked allocation, except links aren't in blocks, they are in a "file allocation table" in memory

Disadvantages:

- Data scattered throughout disk (more seeks)
- Still need to jump through table to get to an arbitrary location in the file
- Must store table in memory



0	free
1	2
2	end
3	end
4	3
5	end
6	4
7	free
	…

File Payload Data

First key question: should we store files contiguously on disk? What would it look like if we ***didn't***?

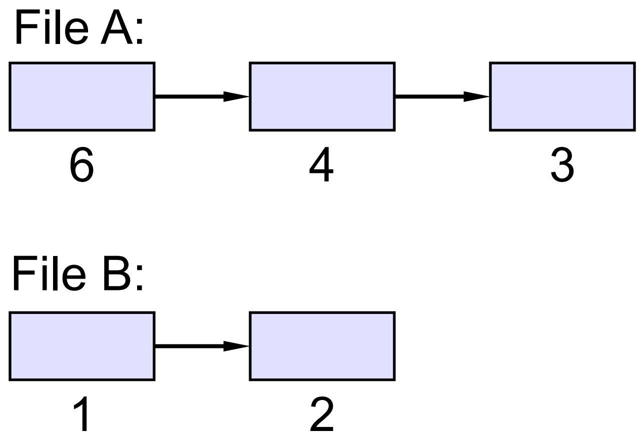
- *Windows (DOS) FAT*: like linked allocation, except links aren't in blocks, they are in a "file allocation table" in memory

Disadvantages:

- Data scattered throughout disk (more seeks)
- **Still need to jump through table to get to an arbitrary location in the file**
- Must store table in memory

File Allocation Table

0	free
1	2
2	end
3	end
4	3
5	end
6	4
7	free
	…

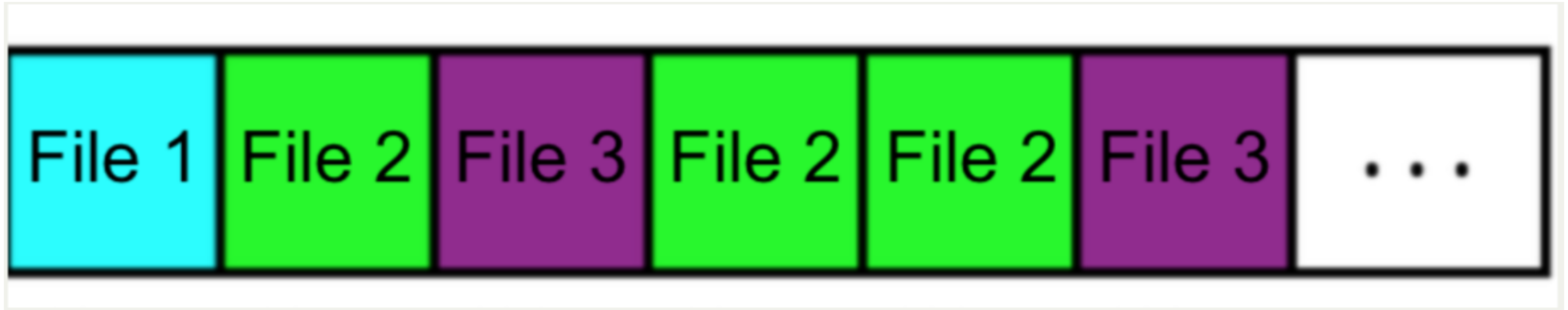


File Payload Data

First key question: should we store files contiguously on disk? What would it look like if we ***didn't***?

Interesting idea: what if we stored all the block numbers for a file? That way we could quickly jump to any point in the file.

- *Multi-level indexes*: store all block numbers for a given file (but how?)
- Example: 4.3 BSD Unix, Unix V6 Filesystem (~1975)



Plan For Today

- Filesystems Introduction
- Methods for Storing Files
  - Contiguous Allocation
  - Linked Files
  - Windows FAT
  - Multi-level indexes
- **The Unix V6 Filesystem**
  - Inodes

Unix V6 Filesystem

Key Idea: files don’t need to be stored contiguously on disk, but we want to store all the block numbers that make up the data for a file.

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

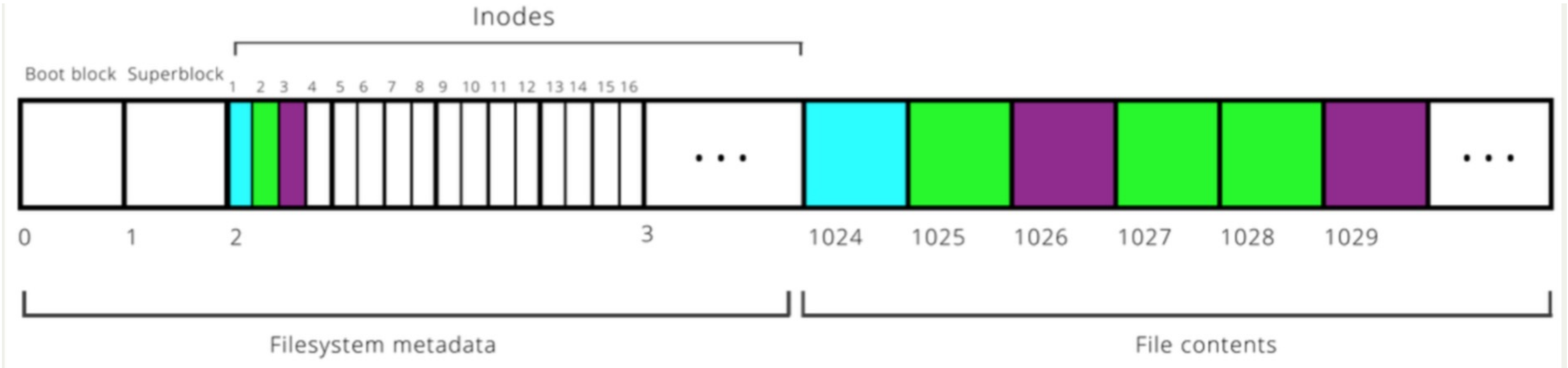
An **inode** ("index node") is a grouping of data about a single file. It’s stored on disk, but we can read it into memory when the file is open.

- Some other filesystems (e.g., contiguous allocation/linked files, but not FAT) store file metadata in inodes, too
- For Unix v6, they store things like file size and an ordered list of block numbers that store file payload data.
- The Unix v6 filesystem stores inodes on disk together in the **inode table** for quick access.

Unix V6 Inodes

The Unix v6 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.
- Inodes are 32 bytes big, and 1 block = 1 sector = 512 bytes, so 16 inodes/block.
- Filesystem goes from **filename** to **inode number** ("inumber") to **file data**.



Unix V6 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, which are stored in order. (i.e. first block number stores first chunk of file, etc.). But we will see later how we can build on this to support very large files.

Recap

- Filesystems Introduction
- Methods for Storing Files
  - Contiguous Allocation
  - Linked Files
  - Windows FAT
  - Multi-level indexes
- The Unix V6 Filesystem
  - Inodes

Next time: more about the Unix v6 Filesystem

Lecture 2 takeaway: Filesystems need to store both file metadata and payload data. There are various ways to store payload data, each with different pros/cons. The Unix V6 filesystem uses inodes to store file data, including block numbers.