# Lecture 13: Filesystem Principles

CS343 – Operating Systems

Branden Ghena – Fall 2025

Some slides borrowed from:
Stephen Tarzia (Northwestern), Shivaram Venkataraman (Wisconsin), and UC Berkeley CS162

# Administrivia

- Best time to start Paging Lab was last week
  - Second-best time is right now

- This is the hardest lab and will take quite a bit of work

- Due one week from today

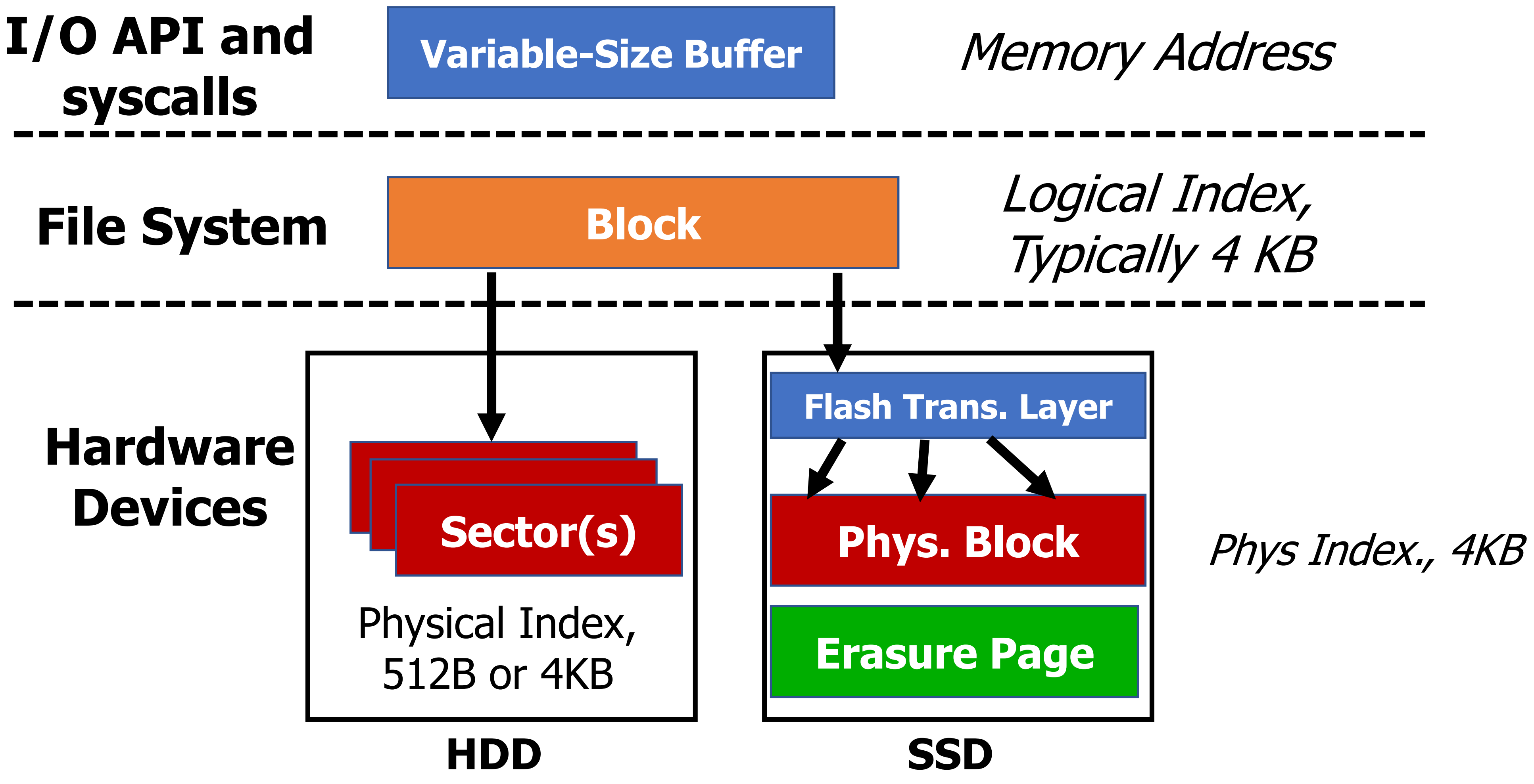
# Today's Goals

- Introduce the general concerns of filesystems.
- Revisit application-level view of filesystems.
- Explore tradeoffs in how filesystems track which blocks are available and which blocks are in use by which files.
- Generally, understand the "design space" of filesystems.
    - Implementations will be selections of these.

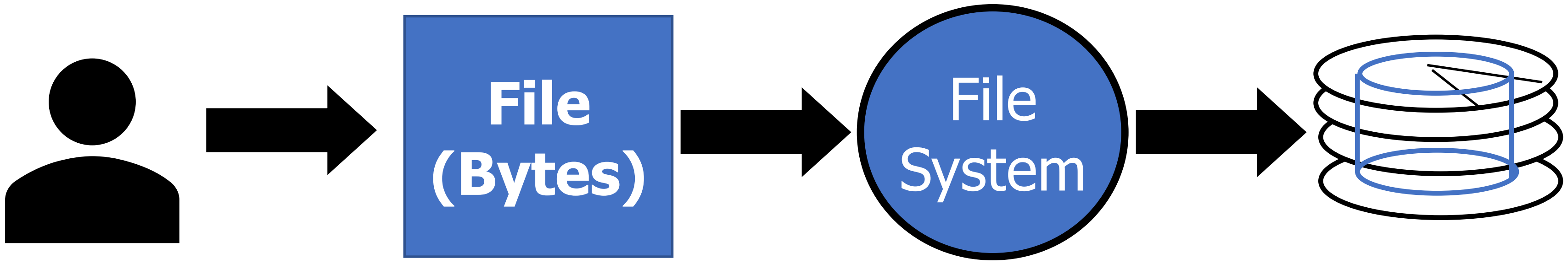
# Outline

- **Introduction to filesystems**
- Application view
- Parts of a file system
  - Managing disk
  - Handling file data
  - Tracking files
- Whole filesystem example

# Introducing file systems



# Translation from user to system view




What happens if user says: “give me bytes 2 – 12?”

- Fetch block corresponding to those bytes
- Return just the correct portion of the block

- What about writing bytes 2 – 12?
  - Fetch block, modify relevant portion, write out block

Everything inside file system is in terms of whole-size blocks

- Actual disk I/O happens in blocks
- `read`/`write` smaller than block size needs to translate and buffer

# Classic OS situation

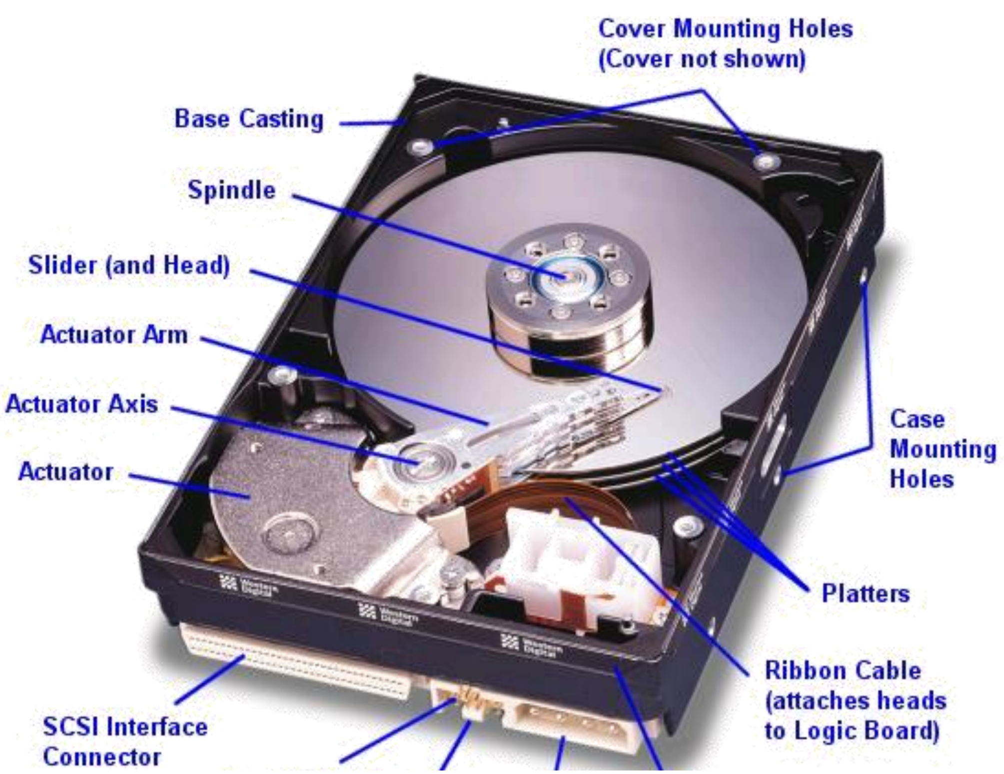
- Take limited hardware interface (array of blocks) and provide a more convenient/useful interface with:
  1. Naming: Find file by name, not block numbers
  2. Translation: Map files to blocks
  3. Organization: Tree-based directory structure which holds all files
  4. Protection: Enforce access restrictions
  5. Reliability: Keep files intact despite crashes, hardware failures, etc.

- We combine all of this to create a filesystem
  - Many different approaches and tradeoffs
  - FAT32, NTFS, ext4, ZFS, etc.

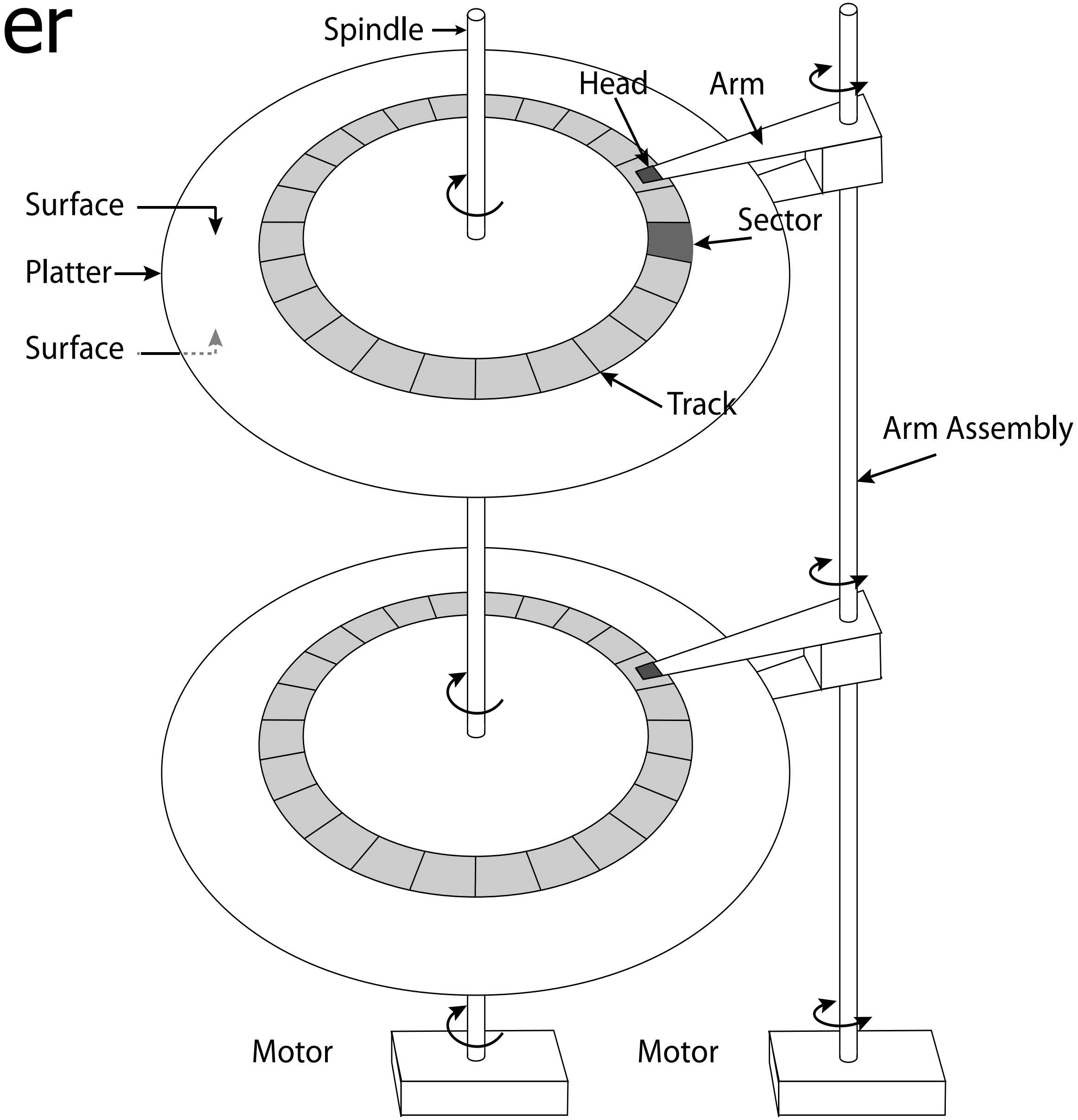
# Filesystem challenges

- Disk performance
  - Sequential access is fast; random access is slow (for HDDs)
- Persistence of data
  - Needs to tolerate sudden power loss without corruption
- Free space management
  - Files are created and deleted
  - Files grow and shrink in size

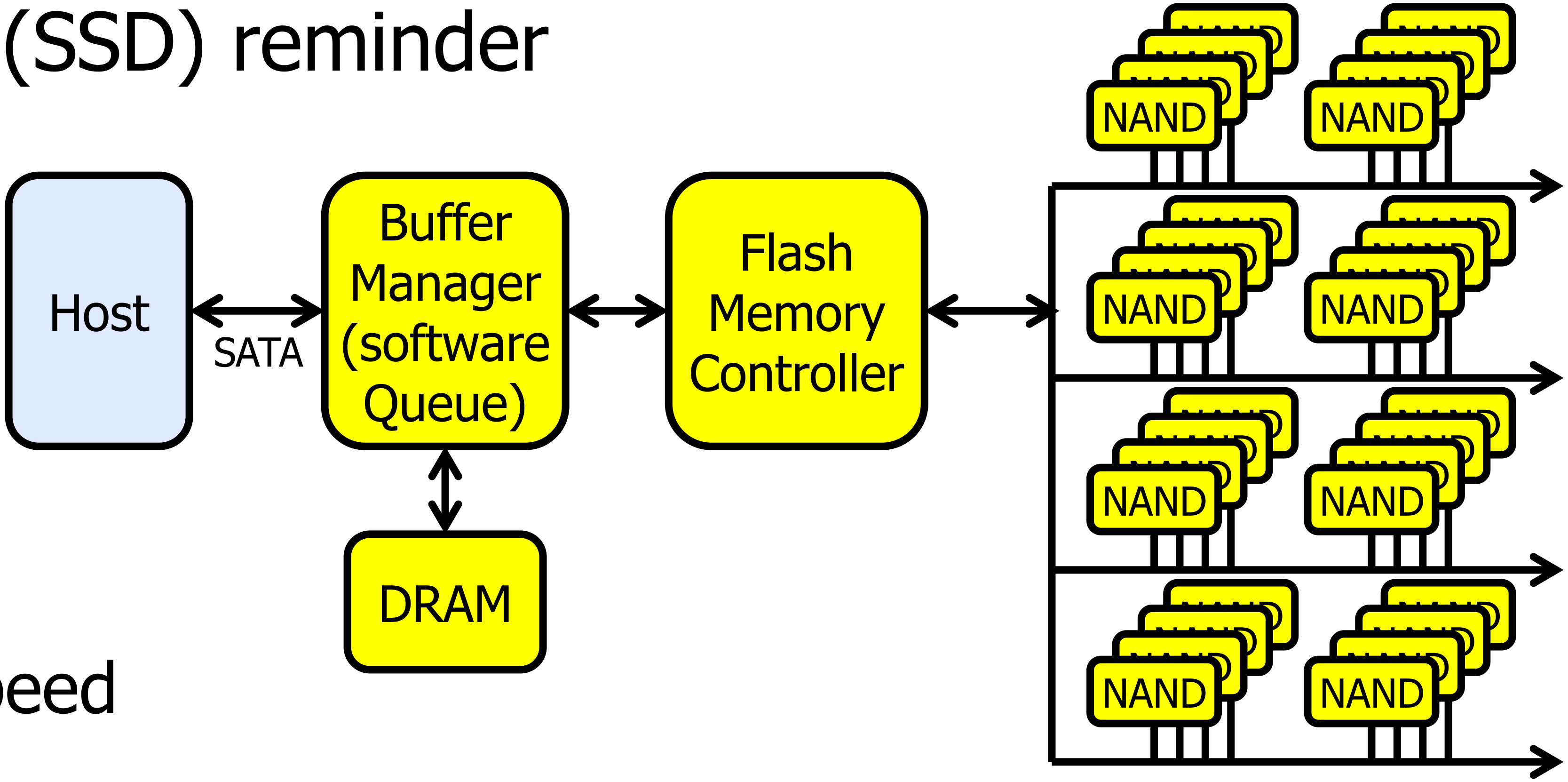
# Hard drive disk (HDD) reminder



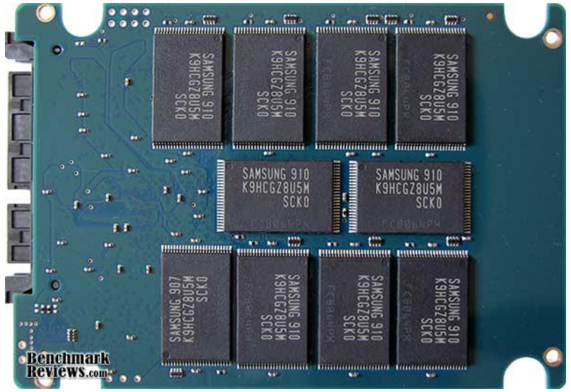

Western Digital Drive
http://www.storagereview.com/guide/



# Solid state drive (SSD) reminder




- Flash memory
- No issues with random access speed
- Writes are a concern though!
  - Writes 10x slower than reads
  - Limited write lifetime (~1-10k writes per page)



# Outline

- Introduction to filesystems
- **Application view**
- Parts of a file system
  - Managing disk
  - Handling file data
  - Tracking files
- Whole filesystem example

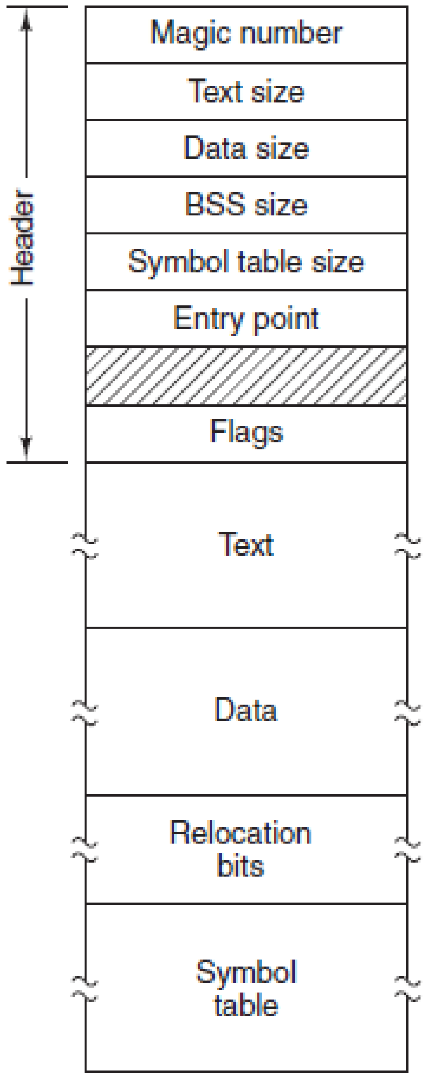
# Application view of file system

- Directories
  - Which are just a file where the data is pointers to other files

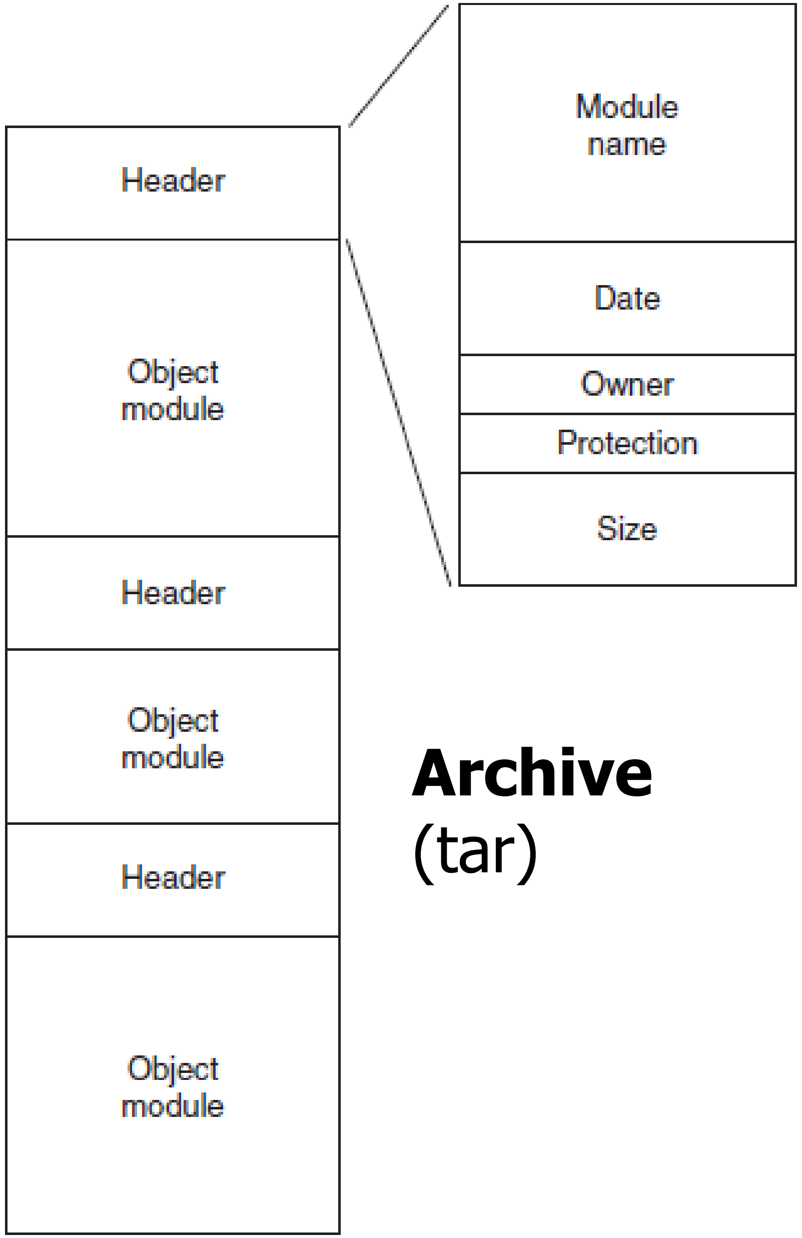
- Regular Files
  - A handle with associated data
  - "Type" of the file comes down to the data within it
    - Reminder: "File extensions" in name of file are a convention, not a necessity

- Special files
  - Devices can be accessed through "files" in some operating systems (character and block devices)

# Binary file examples




**Executable File**




**Archive** (tar)

# Syscalls to interact with files

- `open` (or create) a file with a given *path* (directories & name) and set the file pointer to the beginning of the file
- `read` up to a certain number of bytes from an open file, and move the file pointer for the next read.
- `write` an array of bytes to an open file (and move the pointer)
- `close` an open file
- `lseek` to move the file pointer to a certain index in the file
- `fsync` to push changes to disk immediately (flush dirty data)

# Additional file syscalls

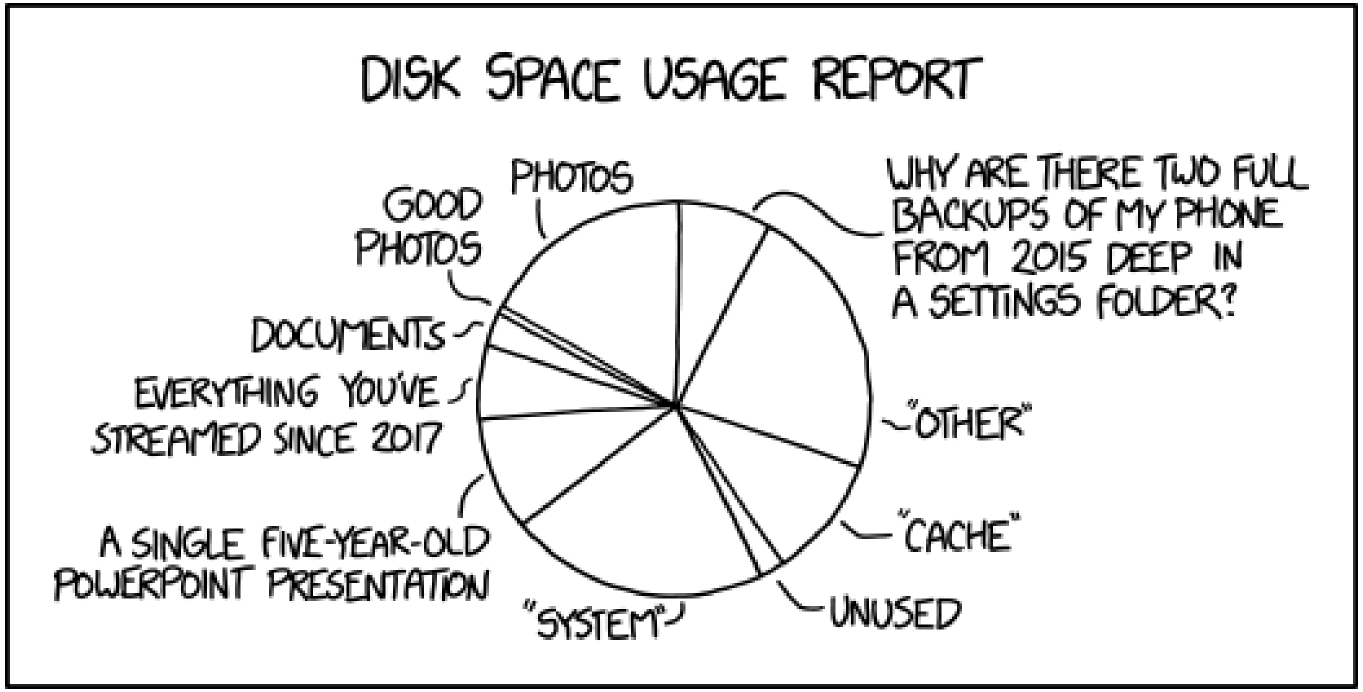
- `rename` to move a file
- `unlink` to remove a file
- `mkdir` to make a directory
- Linux:
  - `getdents` to list the contents of a directory
    - “get directory entries”
    - Because “read” would be filesystem-specific to interpret
- `stat/fstat` gets file metadata (data about the data)

# File/directory metadata

- Files also have *attributes*: readable, writeable, access time, etc.

```
struct stat {
    dev_t     st_dev;          /* ID of device containing file */
    ino_t     st_ino;          /* Inode number (low-level name) */
    mode_t    st_mode;         /* File type and mode (permissions) */
    nlink_t   st_nlink;        /* Number of hard links */
    uid_t     st_uid;          /* User ID of owner */
    gid_t     st_gid;          /* Group ID of owner */
    dev_t     st_rdev;         /* Device ID (if special file) */
    off_t     st_size;         /* Total size, in bytes */
    blksize_t st_blksize;      /* Block size for filesystem I/O */
    blkcnt_t  st_blocks;       /* Number of 512B blocks allocated */
    struct timespec st_atim;   /* Time of last access */
    struct timespec st_mtim;   /* Time of last modification */
    struct timespec st_ctim;   /* Time of last status change */
};
```

# Break + **double** xkcd




https://xkcd.com/2143/


https://xkcd.com/2531/

# Outline

- Introduction to filesystems
- Application view
- **Parts of a file system**
  - **Managing disk**
  - Handling file data
  - Tracking files
- Whole filesystem example

# Data structures on disk

- A bit different than data structures in memory

- Access must be in units of blocks at a time
  - Can’t efficiently read/write a single word
  - Instead must read/write entire block containing it
  - Ideally want sequential access patterns (sequential accesses are fast)

- Durability
  - File system *hopefully* should be in a meaningful state upon shutdown
  - But, shutdown could happen *at any moment* so that guarantee is tricky!

# Disk partitions

Disk A

| |
|---|
| Initial Boot Code |
| Partition Table |
| Partition 1: /boot |
| Partition 2: / |
| Partition 3: swap |
| Partition 4: Windows C:\ |

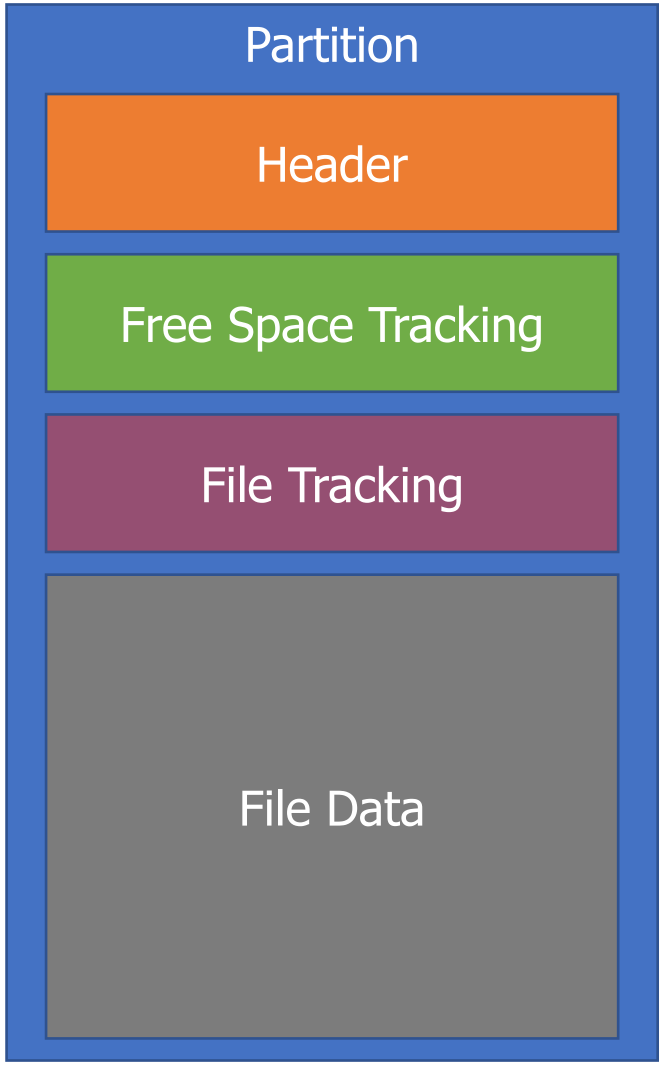
*(not drawn to scale)*

- Many computers have one physical disk,
  - But they may require multiple filesystems.
- A disk partition is a contiguous chunk of the disk that can be formatted to store a filesystem.
- At left, we have:
  - Three different Linux partitions: /boot, swap, /
  - A Windows partition.
  - Each of the partitions may be formatted differently.
- At bootup, initial boot code will present user with a menu to choose Windows or Linux boot.

# What does the filesystem need to track?

- Track free disk blocks (within partition)
  - Need to know which are available for new data
- Track blocks containing data for files
  - Need to know where to read a file from
- Track files in a directory
  - Need to be able to walk the directory hierarchy to find files
- All this needs to be maintained in data structures on the disk itself

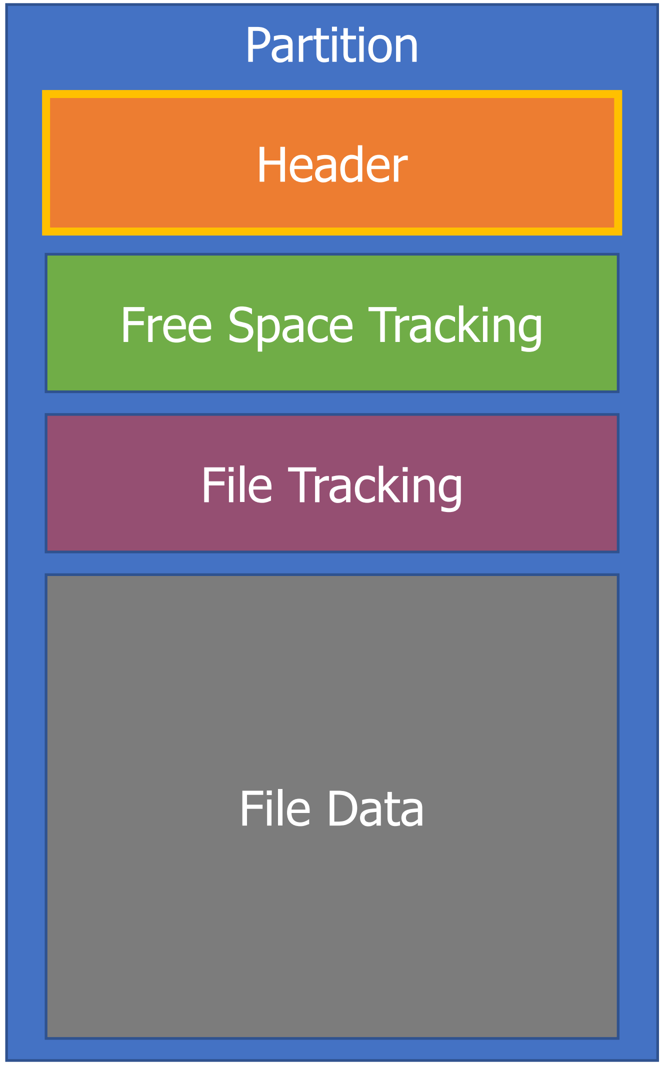
# What goes within a partition?




- Generic view of any filesystem
  - We'll talk about specifics next lecture

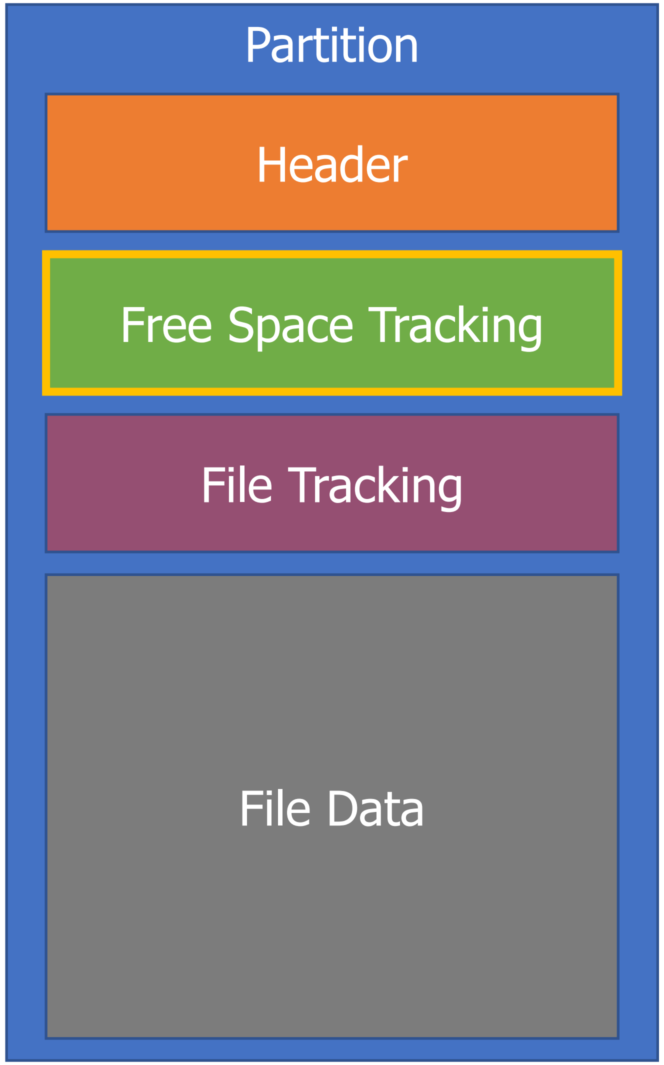
1. Header
2. Free Space Tracking
3. File Tracking
4. File Data

# What goes within a partition?




- Generic view of any filesystem
  - We'll talk about specifics next lecture
- Header (Superblock)
  - Details about which filesystem this is
  - Metadata about the filesystem

# Tracking available blocks on a disk




- Free Space Tracking
  - Track which blocks in “File Data” are in use

- List of block addresses could make sense…
  - Assume block address is 32-bits and 4 KB block
  - 1 TB disk -> 250,000,000 blocks
  - = 1 GB of block addresses
  - More complex but space-efficient data structures are possible
    - Space-efficient is nice because we really want to limit reads to disk

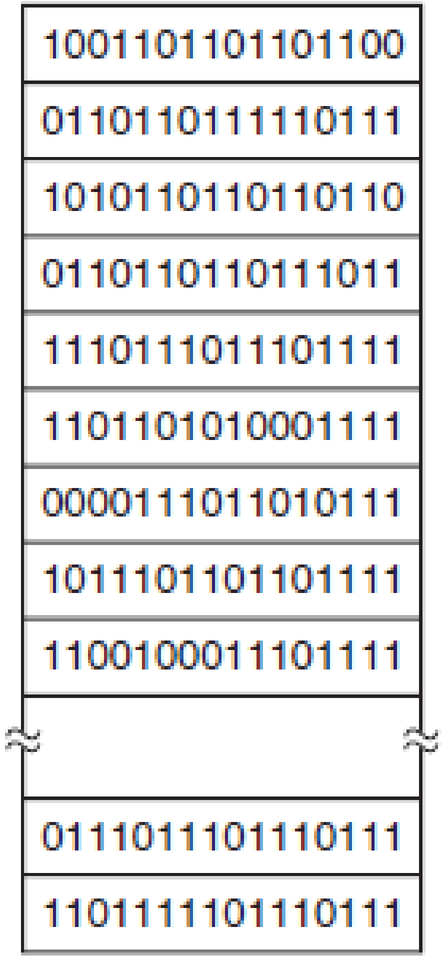
# Bitmaps are a more space efficient tracking option

- Each block on disk is represented by a single bit
  - 1 means free and 0 means used (or vice versa)
  - Every block is listed in order

- 1 TB disk -> 250 million blocks -> 250 million bits -> 30 MB

- Bitmaps for tracking free blocks are usually a constant size for a given disk size
  - Upside: easy to work with
  - Downside: more complex data structures could compress runs of free/used blocks
    - Depends whether disk is expected to be fragmented or not

- Bitmaps are typically used in practice

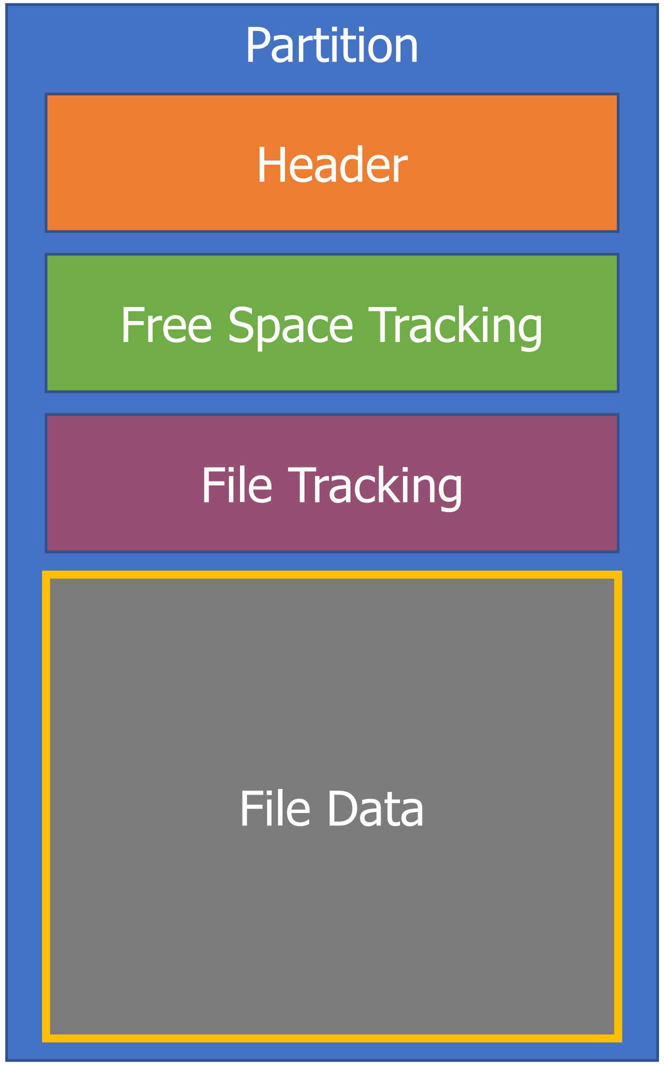



A bitmap

# Outline

- Introduction to filesystems
- Application view
- **Parts of a file system**
  - Managing disk
  - **Handling file data**
  - Tracking files
- Whole filesystem example

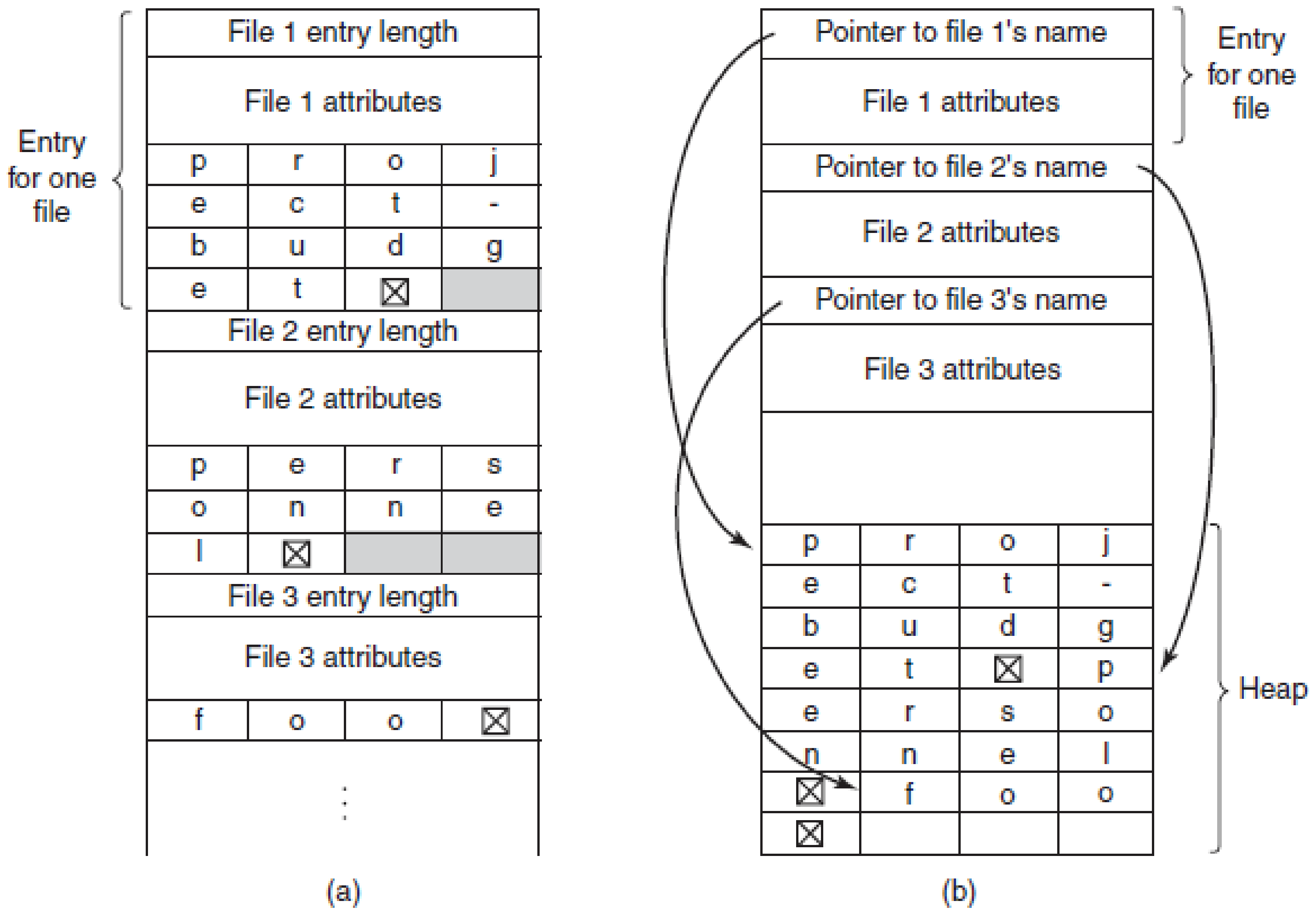
# What goes in the file data?




- Regular file data
  - Just the file's contents directly
  - Attributes already handled in inode
- Directory data
  - Structure listing files within this directory
    - File name, permissions, first data block
    - OR File name, inode
  - Obvious implementation leads to a fixed maximum file name size
    - 8 characters in MS-DOS plus 3 for extension
    - 14 characters in Unix v7
    - This is the route of much evil abbreviation

# Directory data structures

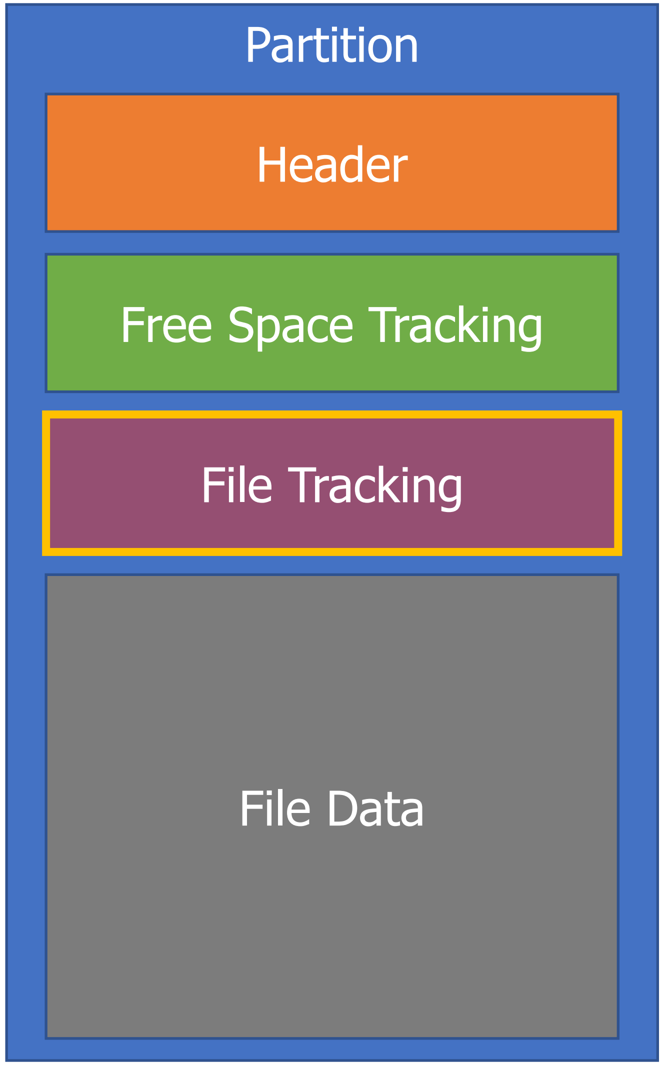
- (a) uses variable-length structures for each file
- (b) contains an extra heap section for holding filenames
- File attributes could also go here instead of in the inode



# Outline

- Introduction to filesystems
- Application view
- **Parts of a file system**
  - Managing disk
  - Handling file data
  - **Tracking files**
- Whole filesystem example

# Tracking available blocks on a disk



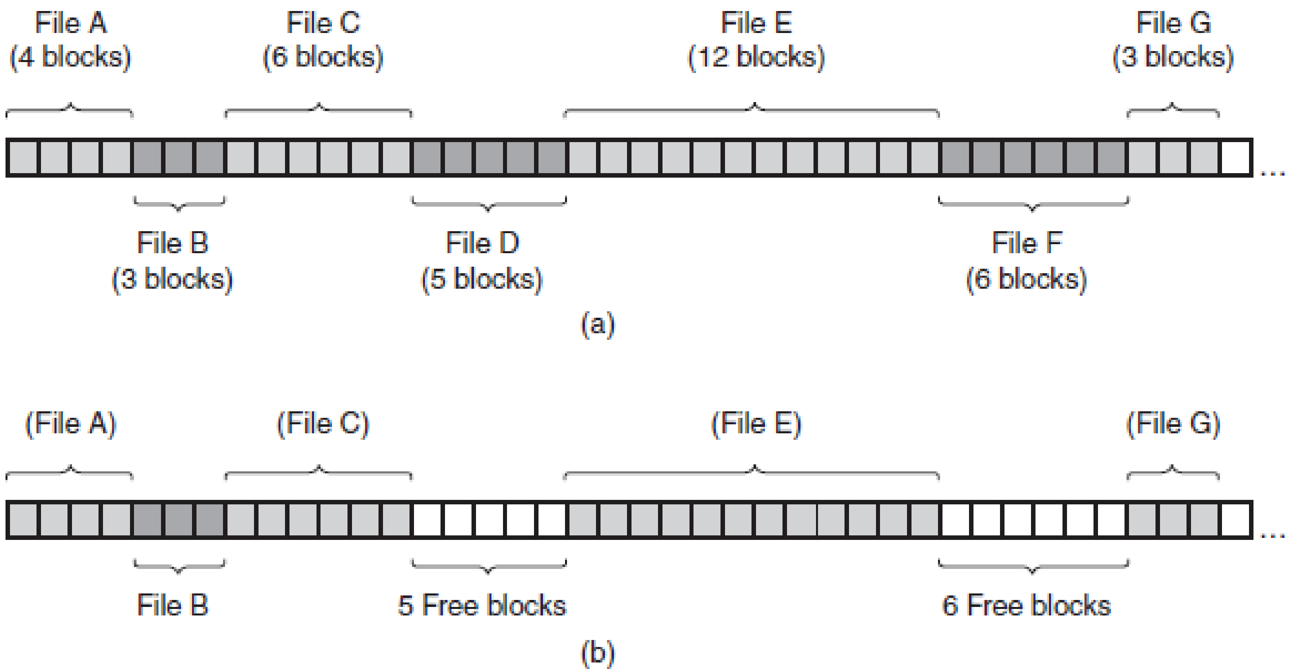

- File Tracking
  - File attributes
  - Ordered blocks where the file data is located

- Two common implementations
  - **Allocation Table**
    - FAT32

  - **Index Nodes** (inodes)
    - Unix File System, Fast File System,
    - ext3/ext4, NTFS

# Requiring contiguous blocks won't work

- Need ability to map random blocks to file
  - Files in contiguous blocks is definitely nice
    - Sequential reads are fast
  - But *requiring* it leads to lots of fragmentation (unusable gaps in disk)



# Allocation table – an example file tracking solution

- Build a linked list of data blocks for each file
  - But distribute that linked list across an array

- Treat "File Tracking" block as an array of block pointers
  - Index into this array is the block number
  - Read tracking block first to determine which data blocks for the file
    - Then you can only read the actual data blocks you want

Next Block

Example: File "A" starts at Block 4

Block Number

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **0** | -1 | **4** | 7 | **8** | -1 | **12** | -1 |
| **1** | -1 | **5** | -1 | **9** | -1 | **13** | -1 |
| **2** | 10 | **6** | 3 | **10** | 12 | **14** | -1 |
| **3** | 11 | **7** | 2 | **11** | 14 | **15** | -1 |

# Allocation table

- Build a linked list of data blocks for each file
  - But distribute that linked list across an array

- Treat “File Tracking” block as an array of block pointers
  - Index into this array is the block number
  - Read tracking block first to determine which data blocks for the file
    - Then you can only read the actual data blocks you want

Next Block

Example: File “A” starts at Block 4

Block Number

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **0** | -1 | **4** | 7 | **8** | -1 | **12** | -1 |
| **1** | -1 | **5** | -1 | **9** | -1 | **13** | -1 |
| **2** | 10 | **6** | 3 | **10** | 12 | **14** | -1 |
| **3** | 11 | **7** | 2 | **11** | 14 | **15** | -1 |

# Allocation table

- Build a linked list of data blocks for each file
  - But distribute that linked list across an array
- Treat "File Tracking" block as an array of block pointers
  - Index into this array is the block number
  - Read tracking block first to determine which data blocks for the file
    - Then you can only read the actual data blocks you want

Next Block

Example: File "A" starts at Block 4

Block Number

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **0** | -1 | **4** | 7 | **8** | -1 | **12** | -1 |
| **1** | -1 | **5** | -1 | **9** | -1 | **13** | -1 |
| **2** | 10 | **6** | 3 | **10** | 12 | **14** | -1 |
| **3** | 11 | **7** | 2 | **11** | 14 | **15** | -1 |

# Allocation table

- Build a linked list of data blocks for each file
  - But distribute that linked list across an array

- Treat "File Tracking" block as an array of block pointers
  - Index into this array is the block number
  - Read tracking block first to determine which data blocks for the file
    - Then you can only read the actual data blocks you want

Next Block

Example: File "A" starts at Block 4

Block Number

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **0** | -1 | **4** | 7 | **8** | -1 | **12** | -1 |
| **1** | -1 | **5** | -1 | **9** | -1 | **13** | -1 |
| **2** | 10 | **6** | 3 | **10** | 12 | **14** | -1 |
| **3** | 11 | **7** | 2 | **11** | 14 | **15** | -1 |

# Allocation table

- Build a linked list of data blocks for each file
  - But distribute that linked list across an array

- Treat “File Tracking” block as an array of block pointers
  - Index into this array is the block number
  - Read tracking block first to determine which data blocks for the file
    - Then you can only read the actual data blocks you want

Example: File “A” starts at Block 4

| Block Number | Next Block | Block Number | Next Block | Block Number | Next Block | Block Number | Next Block |
|---|---|---|---|---|---|---|---|
| **0** | -1 | **4** | 7 | **8** | -1 | **12** | -1 |
| **1** | -1 | **5** | -1 | **9** | -1 | **13** | -1 |
| **2** | 10 | **6** | 3 | **10** | 12 | **14** | -1 |
| **3** | 11 | **7** | 2 | **11** | 14 | **15** | -1 |

5 total blocks
{4, 7, 2, 10, 12}

# allocation-table-based system breakdown

- The allocation table tracks next pointers for each data block
  - Contains one entry for each data block
- Each directory has zero or more data blocks
  - The data blocks contain (filename, permissions, first data block) tuples for each file it contains
- Each file has zero or more data blocks
  - The data blocks contain the actual file data

# Break + Check your understanding – Allocation table size

- If each block address is 32 bits, and blocks are 4 kB in size, how big is the Allocation Table for a 2 TB drive?

# Break + Check your understanding – Allocation table size

- If each block address is 32 bits, and blocks are 4 kB in size, how big is the Allocation Table for a 2 TB drive?

- 2 TB / 4 KB = 500,000,000 blocks * 4 bytes = 2 GB

# Problem 1: we really want the allocation table to fit in RAM

- Accessing the allocation table on disk would slow us down
  - File blocks are not necessarily sequential
  - You might end up having to load in multiple blocks worth of File Tracking
- Instead, at boot, load allocation table into RAM
  - File accesses will require scanning the linked list in RAM, but only a single disk access
  - Writes should be sent back to disk occasionally for safety
- But 2 GB is a bit big to leave in RAM all the time…

# Problem 2: file attributes should be more accessible

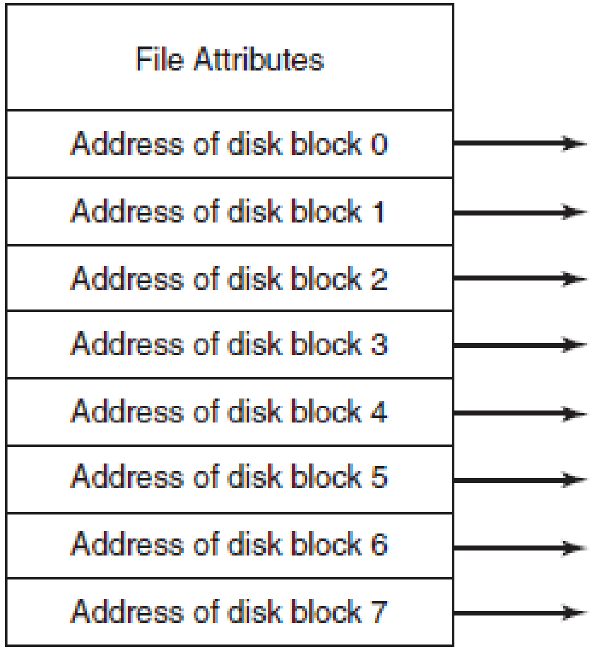
- Unclear where attributes should go for a file with allocation table
  - Either in the first block of the file
  - Or in the directory data

- Separation of attributes from block pointers is undesirable
  - Would be nice to have both of them in a single disk read
  - Or less than one read if they're already in RAM

# Alternative file-tracking mechanism: inodes

- An inode (Index Node) tracks details for a single file
  - Blocks it uses
  - Permissions
- Each file has one and only one inode
- We can hold inodes in memory just for the files that we're actively using!
  - Instead of holding all file-tracking in memory,
    we hold the currently-useful portion of file-tracking in memory

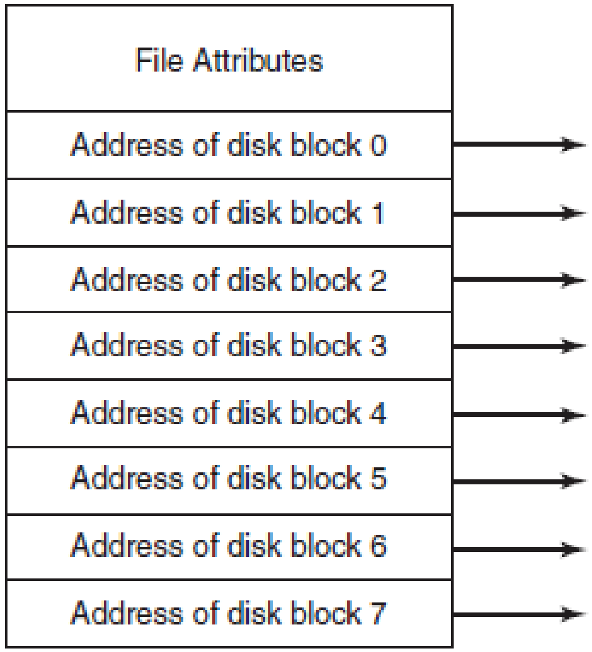
**inode**



# Inode details

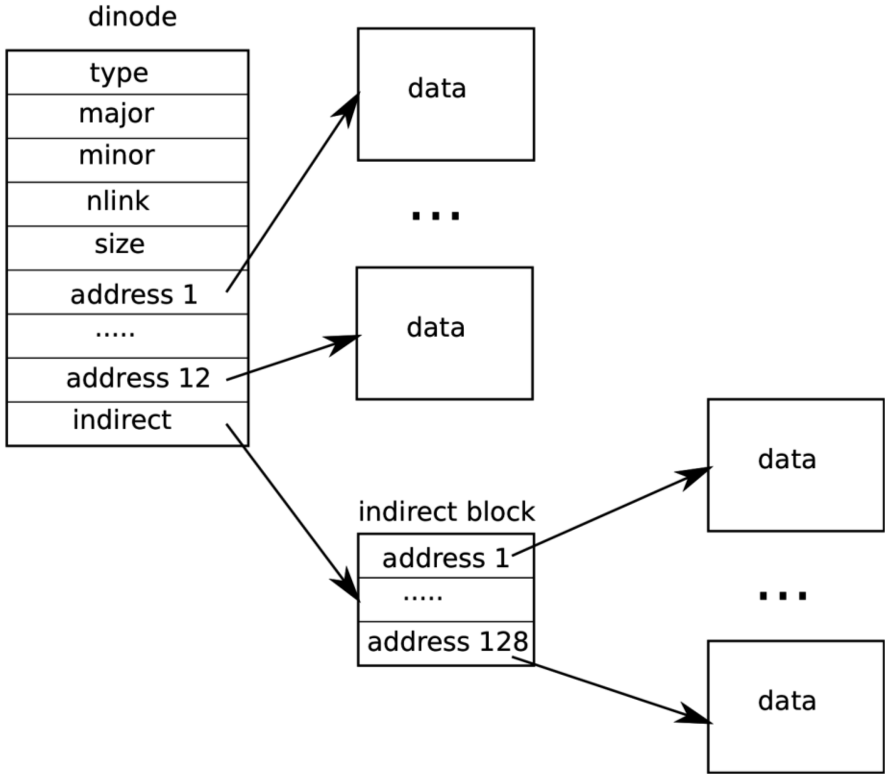
- Treat “File Tracking” as an array of inodes
    - Each inode corresponds to a single file
    - More files means more inodes
- inode contents
    - File attributes
    - Ordered list of pointers to data blocks for the file
- Many improvements have sprung up
    - Optimization: coalesce contiguous blocks
    - Optimization: for very small files, put data right in the inode!

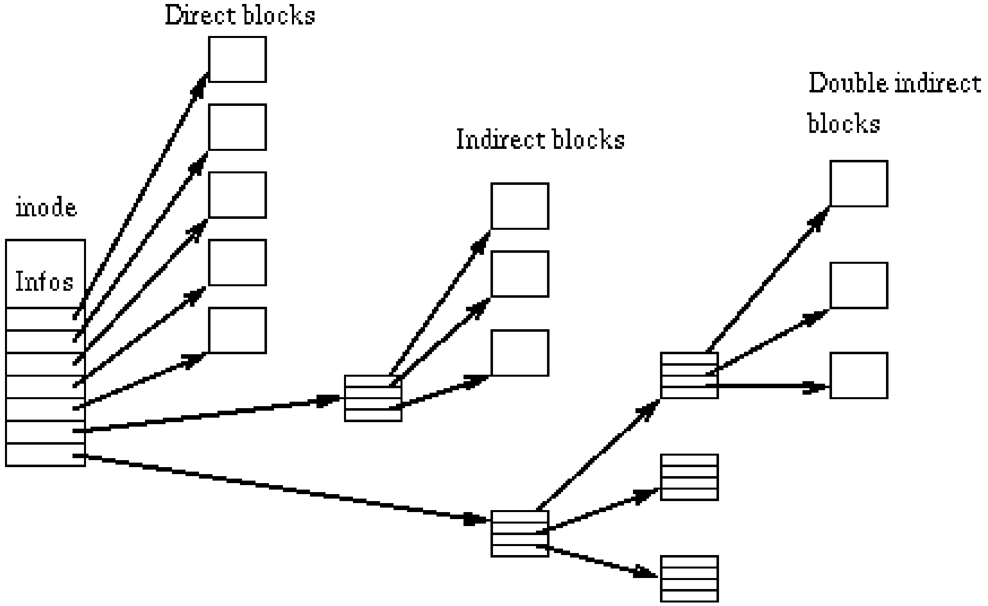
**inode**



# Hierarchical inodes allow for larger file sizes

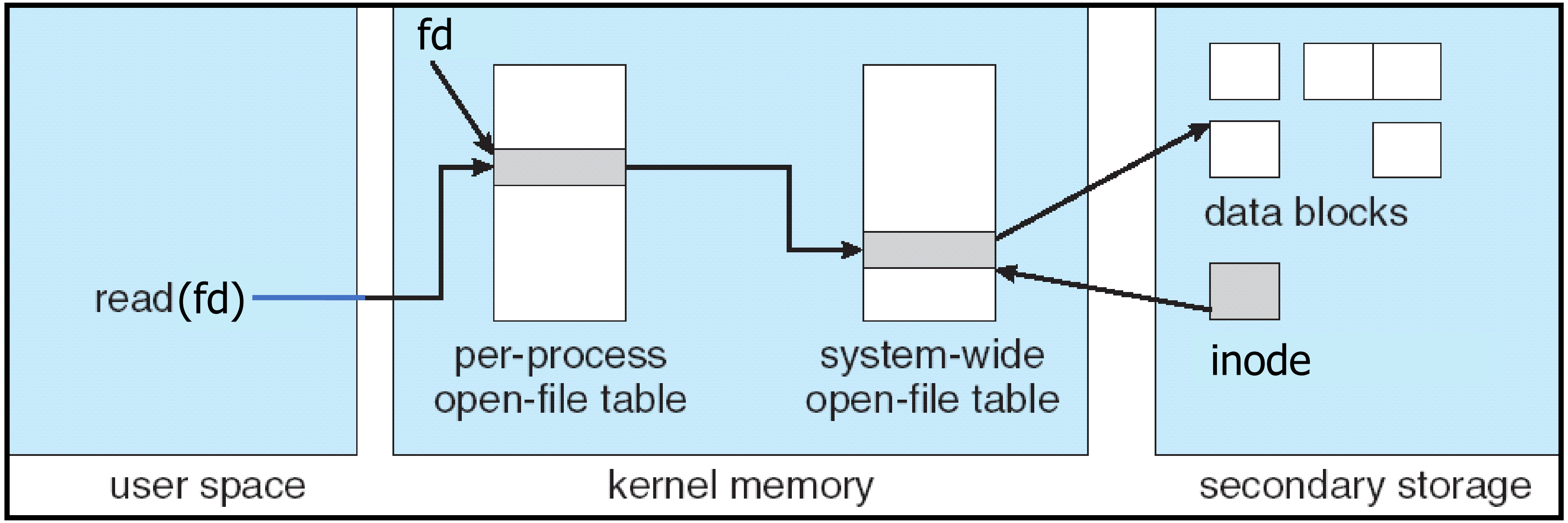
- Each inode is ≤ one block in size
  - So there is a limit to how many block pointers we can fit in an inode
  - Apply tree structure to block pointers to solve this



# File system access with inodes

- Open syscall: find inode and load it into memory
- Read/write syscalls: reference inode by file descriptor



# Full inode-based system breakdown

- Each file (regular file or directory) has one inode
  - The inode has permissions and pointers to data blocks
  - If the file is really big, the inode points to indirect blocks (which contain additional data block pointers)

- Each directory has zero or more data blocks
  - The data blocks contain (filename, inode) pairs for each file it contains

- Each file has zero or more data blocks
  - The data blocks contain the actual file data

# What can we observe about real-world file systems?

## A Five-Year Study of File-System Metadata

NITIN AGRAWAL
University of Wisconsin, Madison
and
WILLIAM J. BOLOSKY, JOHN R. DOUCEUR, and JACOB R. LORCH
Microsoft Research

2007

# 1. Most files are small

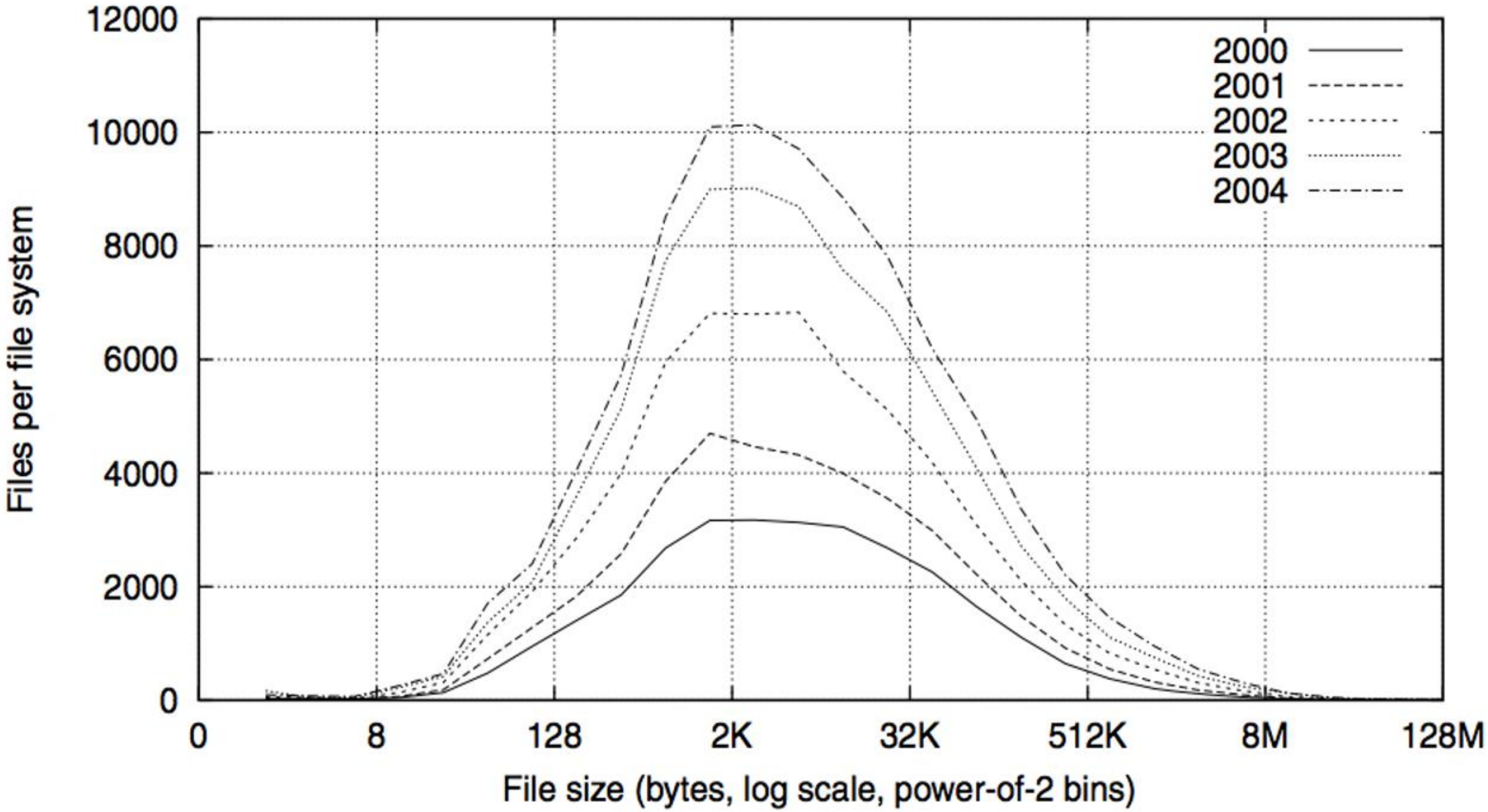



Fig. 2. Histograms of files by size.

# 2. Most bytes are spent on a few large files

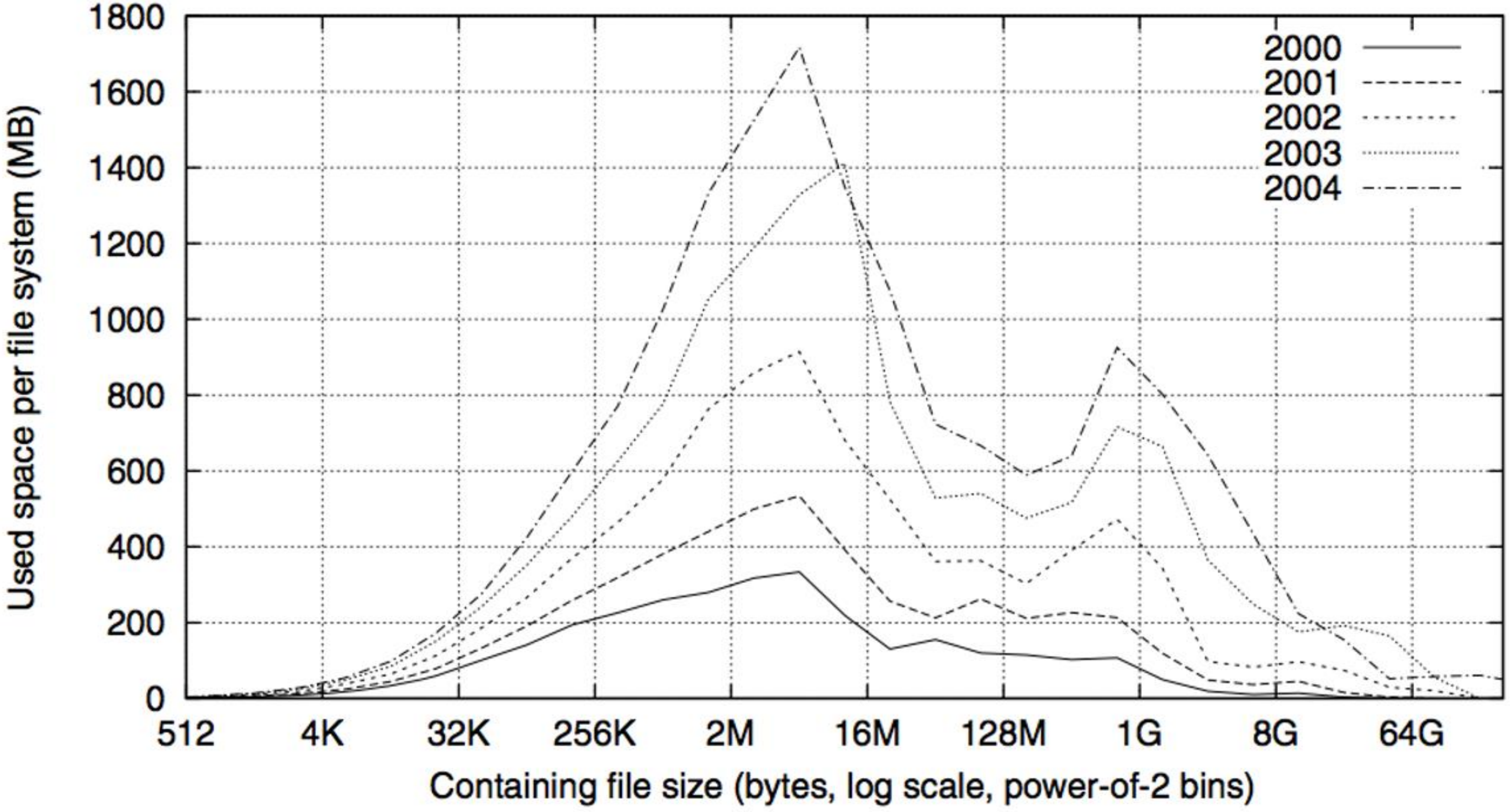



Fig. 4. Histograms of bytes by containing file size.

# Break + Broader Thinking

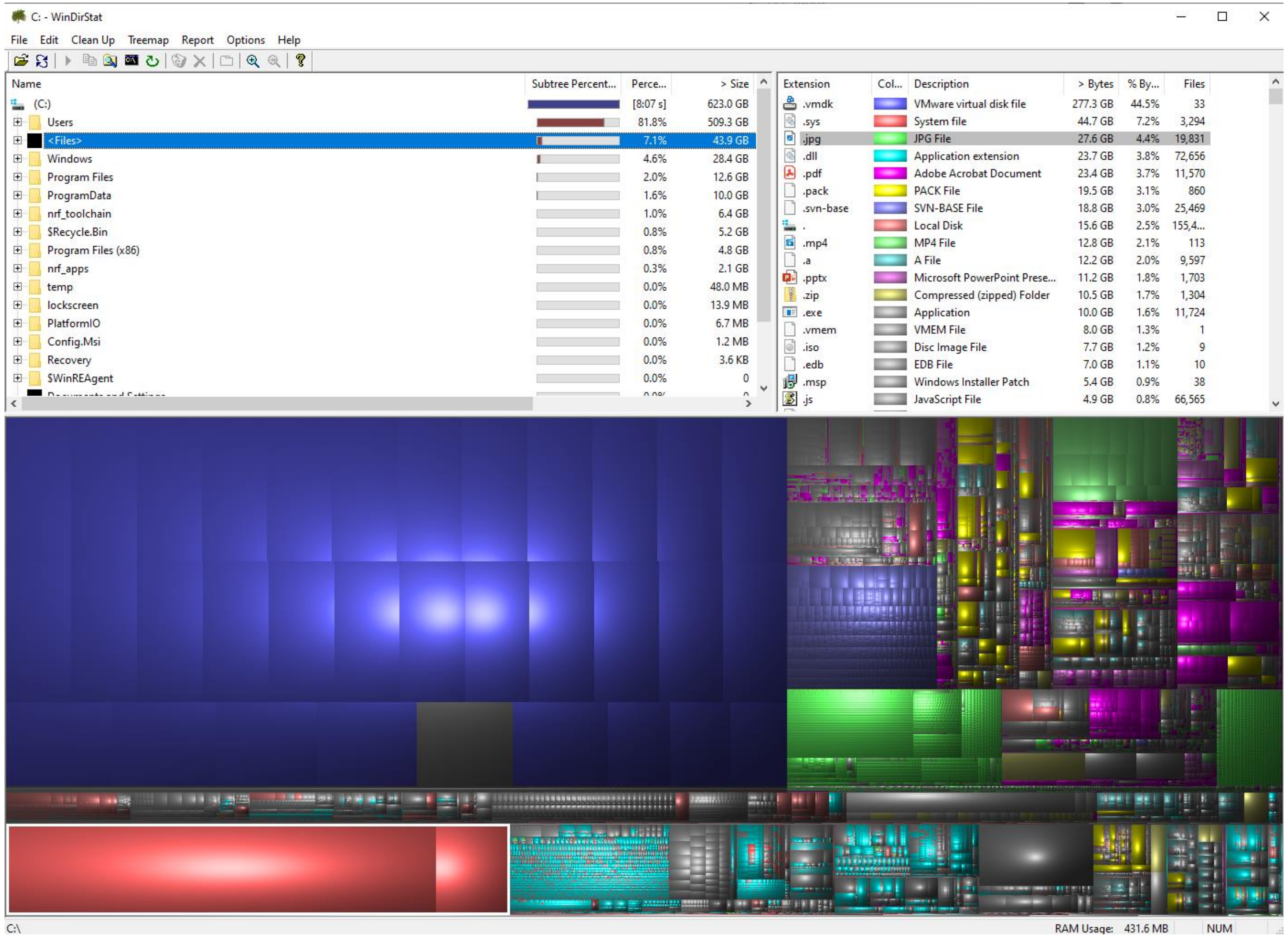
- Study was on 60,000 Windows PC file systems in a large corporation from 2000-2004 (over twenty years ago)

1. Does this still apply today? Why or why not?

2. Can you think of systems where it especially might not apply?

# Storage trace of my work desktop

- 4% JPGs
  - 19,000 total
  - Roughly 1.5 MB per image
- 4% PDFs
- 2% MP4s
- 2% PPTX
- 2% Zips
- 45% for a VM
  (which is really just a partition in disguise)



# Outline

- Introduction to filesystems
- Application view
- Parts of a file system
  - Managing disk
  - Handling file data
  - Tracking files
- **Whole filesystem example**

# A trace through the filesystem

- Now we have enough knowledge to walk through an entire filesystem access

- Here we assume
  - Bitmap for marking free data blocks
  - Bitmap for marking free inode blocks
  - Inode for each file/directory
  - One or more data blocks for each file/directory

- Exact ordering of some of these steps are up for debate
  - Different implementations may swap ordering of some parts

# Open and read example

time

| | data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data | bar data[0] | bar data[1] | bar data[1] |
|---|---|---|---|---|---|---|---|---|---|---|
| open("/foo/bar") | | | read | | | | | | | |
| | | | | | | read | | | | |
| | | | | read | | | | | | |
| | | | | | | | read | | | |
| | | | | | read | | | | | |
| read() | | | | | read | | | | | |
| | | | | | | | | read | | |
| | | | | | write | | | | | |
| read() | | | | | read | | | | | |
| | | | | | | | | | read | |
| | | | | | write | | | | | |
| read() | | | | | read | | | | | |
| | | | | | | | | | | read |
| | | | | | write | | | | | |

find file inode

# Open and read example

| | data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data | bar data[0] | bar data[1] | bar data[1] |
|---|---|---|---|---|---|---|---|---|---|---|
| open("/foo/bar") | | | **read** | | | | | | | |
| | | | | | | **read** | | | | |
| | | | | **read** | | | | | | |
| | | | | | | | **read** | | | |
| | | | | | **read** | | | | | |
| read() | | | | | **read** | | | | | |
| | | | | | | | | **read** | | |
| | | | | | **write** | | | | | |
| read() | | | | | **read** | | | | | |
| | | | | | | | | | **read** | |
| | | | | | **write** | | | | | |
| read() | | | | | **read** | | | | | |
| | | | | | | | | | | **read** |
| | | | | | **write** | | | | | |

time

update attributes

# Open and read example

time

| | data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data | bar data[0] | bar data[1] | bar data[1] |
|---|---|---|---|---|---|---|---|---|---|---|
| open("/foo/bar") | | | **read** | | | | | | | |
| | | | | | | **read** | | | | |
| | | | | **read** | | | | | | |
| | | | | | | | **read** | | | |
| | | | | | **read** | | | | | |
| read() | | | | | **read** | | | | | |
| | | | | | | | | **read** | | |
| | | | | | **write** | | | | | |
| read() | | | | | **read** | | | | | |
| | | | | | | | | | **read** | |
| | | | | | **write** | | | | | |
| read() | | | | | **read** | | | | | |
| | | | | | | | | | | **read** |
| | | | | | **write** | | | | | |

inode reads/writes from here
occur in memory rather than disk

# Open and read example

| | data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data | bar data[0] | bar data[1] | bar data[1] |
|---|---|---|---|---|---|---|---|---|---|---|
| open("/foo/bar") | | | **read** | | | | | | | |
| | | | | | | **read** | | | | |
| | | | | **read** | | | | | | |
| | | | | | | | **read** | | | |
| | | | | | **read** | | | | | |
| read() | | | | | **read** | | | | | |
| | | | | | | | | **read** | | |
| | | | | | **write** | | | | | |
| read() | | | | | **read** | | | | | |
| | | | | | | | | | **read** | |
| | | | | | **write** | | | | | |
| read() | | | | | **read** | | | | | |
| | | | | | | | | | | **read** |
| | | | | | **write** | | | | | |

time

read next file block

# Create and write a file

| | data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data | bar data[0] | bar data[1] | bar data[1] |
|---|---|---|---|---|---|---|---|---|---|---|
| create (/foo/bar) | | | read | | | | | | | |
| | | | | | | read | | | | |
| | | | | read | | | | 1 | | |
| | | | | | | | read | | | |
| | | read | 2 | | | | | | | |
| | | write | | | | | | | | |
| | | | | | | | write | 3 | | |
| | | | | | write | 4 | | | | |
| | | | | write | | | | | | |
| write() | | | | | read | | | | | |
| | read | | | | | | | | | |
| | write | | | | | | | | | |
| | | | | | | | | write | | |
| | | | | | write | | | | | |
| write() | | | | | read | | | | | |
| | read | | | | | | | | | |
| | write | | | | | | | | | |
| | | | | | | | | | write | |
| | | | | | write | | | | | |
| write() | | | | | read | | | | | |
| | read | | | | | | | | | |
| | write | | | | | | | | | |
| | | | | | | | | | | write |
| | | | | | write | | | | | |

## Create:

1. First, read the parent directory to ensure that name is not already used.
2. Find & claim a free inode.
3. Add <"bar", inode#> to parent directory.
4. Fill-in file metadata.

# Create and write a file

| | data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data | bar data[0] | bar data[1] | bar data[1] | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| create (/foo/bar) | | | read | | | | | | | | 1 |
| | | | | | | read | | | | | 1 |
| | | | | read | | | | | | | 1 |
| | | | | | | | read | | | | 1 |
| | | read | | | | | | | | | 2 |
| | | write | | | | | | | | | 2 |
| | | | | | | | write | | | | 3 |
| | | | | | write | | | | | | 4 |
| | | | | write | | | | | | | 4 |
| write() | | | | | read | | | | | | 1 |
| | read | | | | | | | | | | 2 |
| | write | | | | | | | | | | 2 |
| | | | | | | | | write | | | 3 |
| | | | | | write | | | | | | 4 |
| write() | | | | | read | | | | | | |
| | read | | | | | | | | | | |
| | write | | | | | | | | | | |
| | | | | | | | | | write | | |
| | | | | | write | | | | | | |
| write() | | | | | read | | | | | | |
| | read | | | | | | | | | | |
| | write | | | | | | | | | | |
| | | | | | | | | | | write | |
| | | | | | write | | | | | | |

## Create:

1. First, read the parent directory to ensure that name is not already used.
2. Find & claim a free inode.
3. Add <"bar", inode#> to parent directory.
4. Fill-in file metadata.

## Write:

1. Look for remaining space in existing blocks first.
2. Find & claim a new data block.
3. Write data to new block
4. Point to it in inode

# Outline

- Introduction to filesystems
- Application view
- Parts of a file system
  - Managing disk
  - Tracking files
  - Handling file data
- Whole filesystem example