Lecture 15: Filesystem Principles

CS343 – Operating Systems

Branden Ghena – Fall 2022

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

Administrivia – what's left in CS343?

- Lecture
  - Filesystems (two days), Embedded OS, Virtualization

- Paging Lab is due Tuesday *after* Thanksgiving
  - Get started on it early so you don't have to work over break

- Midterm exam 2 will be during exam week
  - Wednesday, December 7th at 12pm in Tech LR4
  - We'll do another review practice session during lecture beforehand

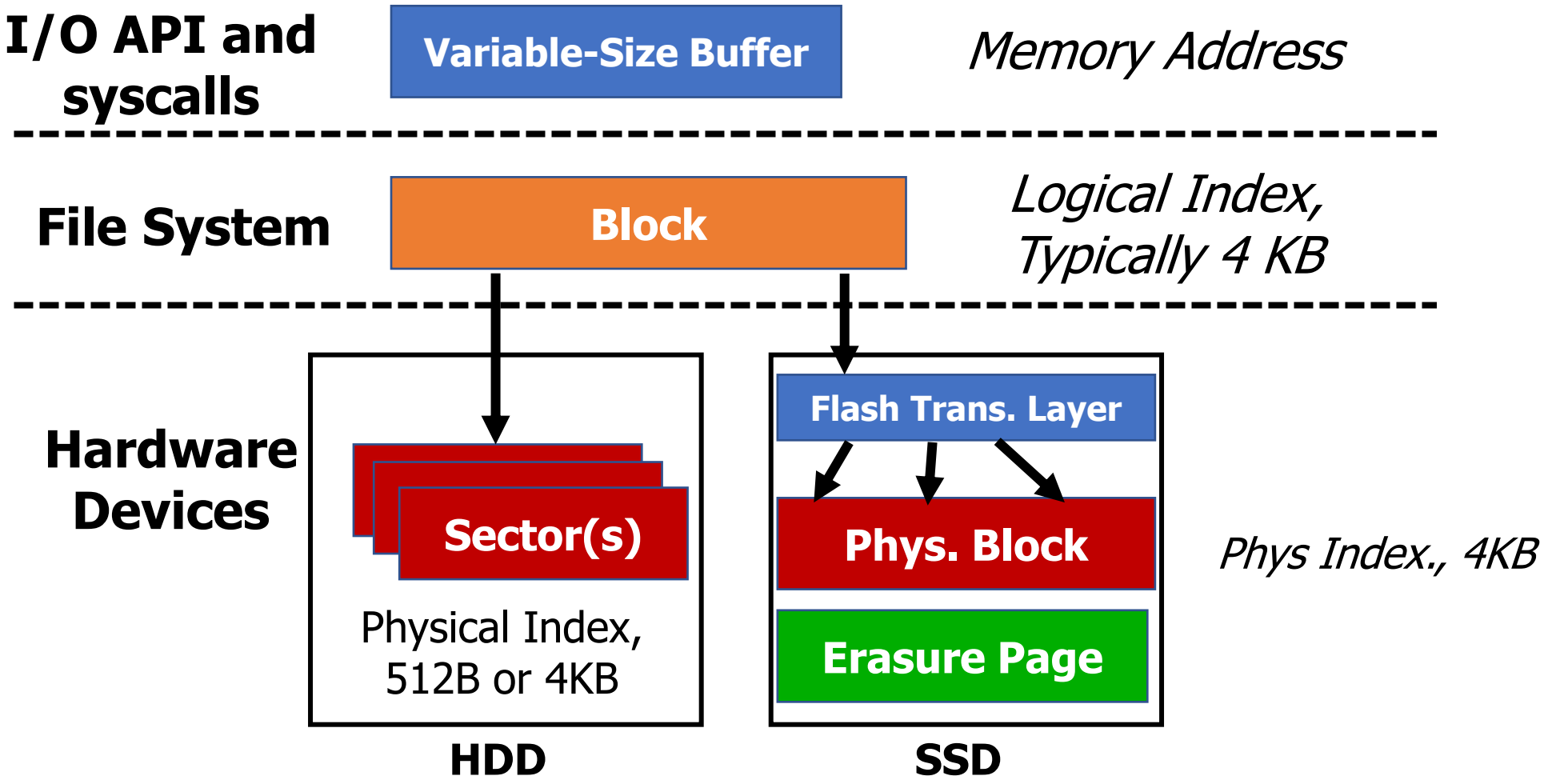
Today's Goals

- 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.

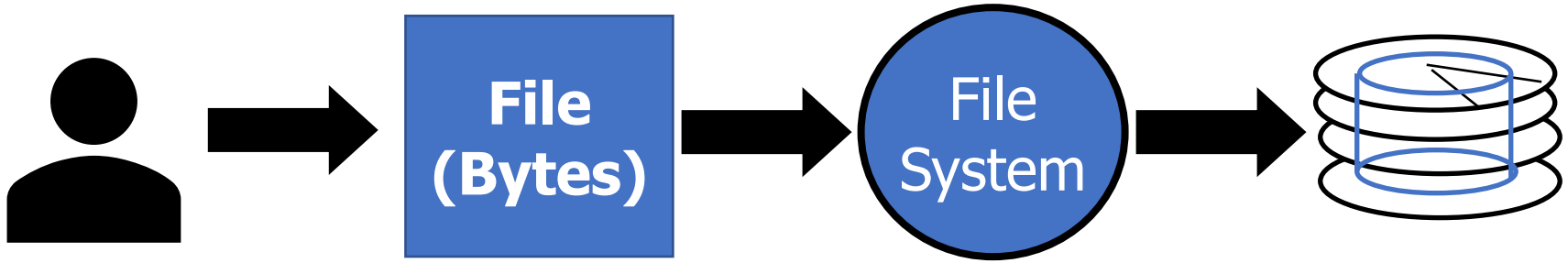
Outline

- **Introduction to filesystems**
- Application view
- Parts of a file system
  - Managing disk
  - Tracking files
  - Handling file data
- 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. Organization: Organize file names with directories
  3. Translation: Map files to blocks
  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

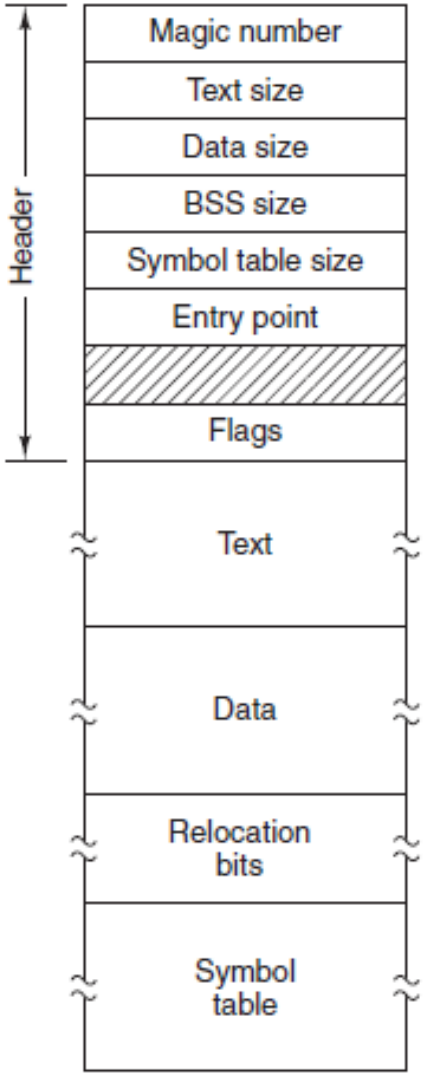
Outline

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

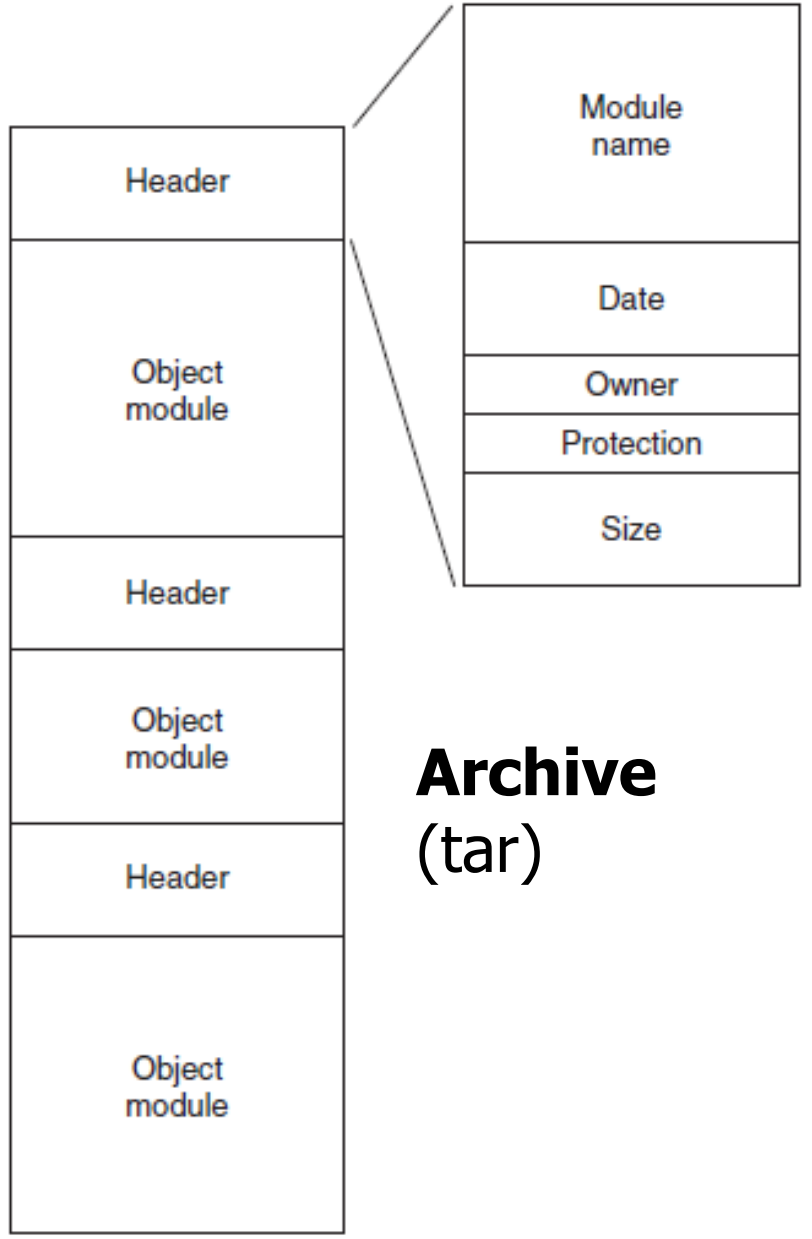
Application view of file system

- Directories
  - Which are just a file where the data is pointers to other files
- 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
  - Character and block devices!!

Binary file examples



Executable File



Archive
(tar)

File command

- **`file`** in Linux command line can determine the type of a file
  - https://github.com/file/file

```
[brghena@ubuntu nautilus] [paginglab *] $ ls
CODE_OF_CONDUCT.md  img          link             nautilus.iso                 scripts  Vagrantfile
configs             include      lua_script.txt   nautilus.syms                setups   xeon_phi
CONTRIBUTING.md     Kconfig      Makefile         PULL_REQUEST_TEMPLATE.md     src
doc                 lib          Makefile.x86_64  README.md                    tools
ENV                 LICENSE.txt  nautilus.bin     run                          user
[brghena@ubuntu nautilus] [paginglab *] $ file README.md
README.md: UTF-8 Unicode text, with very long lines
[brghena@ubuntu nautilus] [paginglab *] $ file nautilus.bin
nautilus.bin: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, with debug_info, no
t stripped
[brghena@ubuntu nautilus] [paginglab *] $ file nautilus.iso
nautilus.iso: DOS/MBR boot sector; GRand Unified Bootloader, stage1 version 0x79, boot drive 0xbb, stage2
 address 0x8e70, 1st sector stage2 0xb8db31c3, stage2 segment 0x201
```

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

- `stat/fstat` gets file metadata (data about the data)
- `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

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 */
};
```

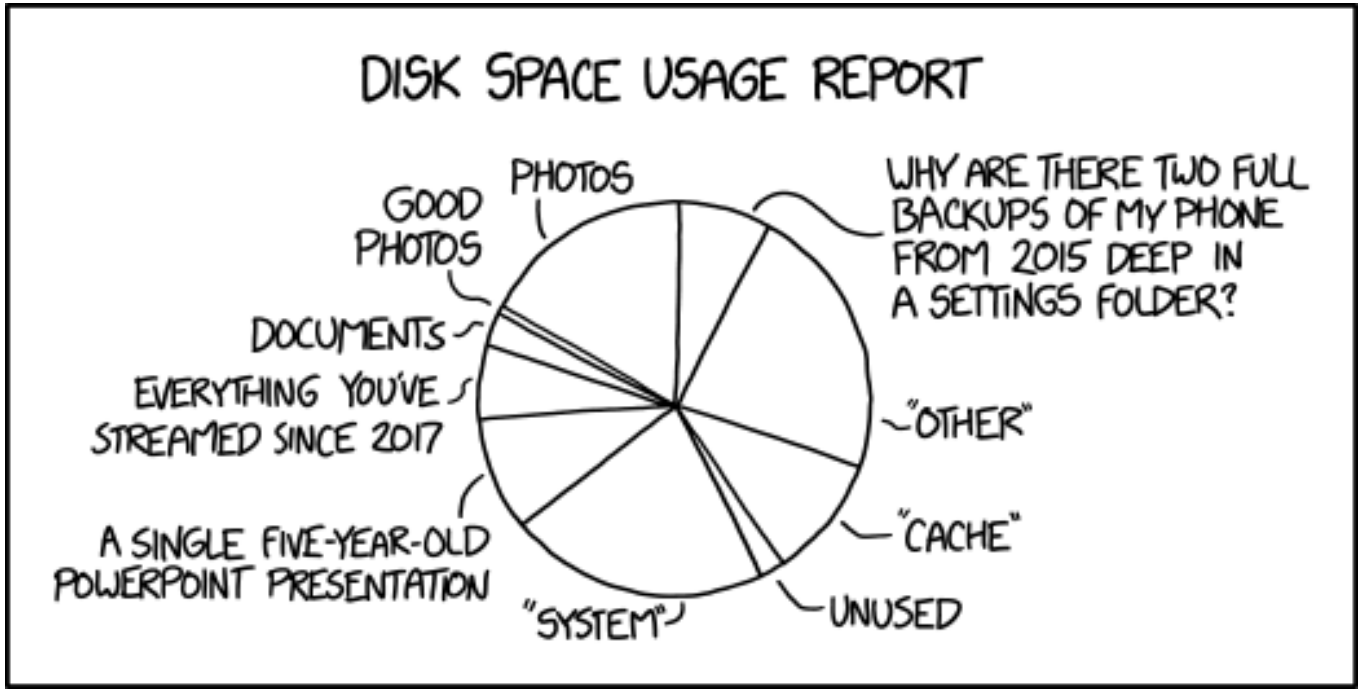
Filesystem links

- **`ln`** unix command creates a link to a file – like a pointer.
  - Allows a file to exist in multiple paths without wasting space

- ***Hard link*** creates another entry in a directory referring to the same disk address (inode number).

- ***Symbolic/Soft link*** is a special file whose contents is just the string path of another file.
  - Symlinks are much more common in modern practice (**`ln -s`**)
  - Allow referring to file in other filesystems
  - But may lead to a ***dangling reference*** – the referred-to file may be deleted

Break + **double** xkcd



https://xkcd.com/2143/

MY RESPONSE WHENEVER ANYONE ASKS ME TO MESS AROUND WITH FILESYSTEMS

https://xkcd.com/2531/

Outline

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

Data structures on disk

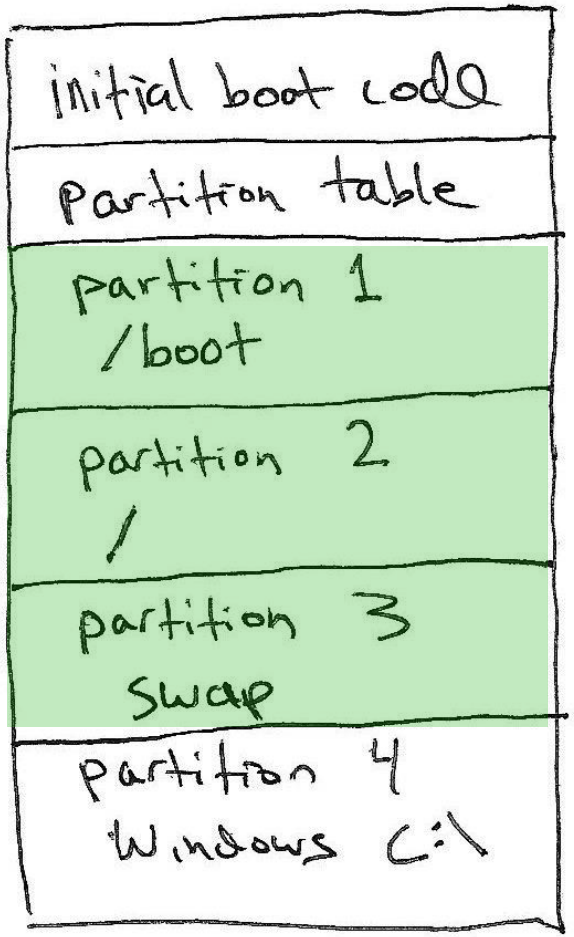
- A bit different than data structures in memory

- 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

Disk partitions

Disk A



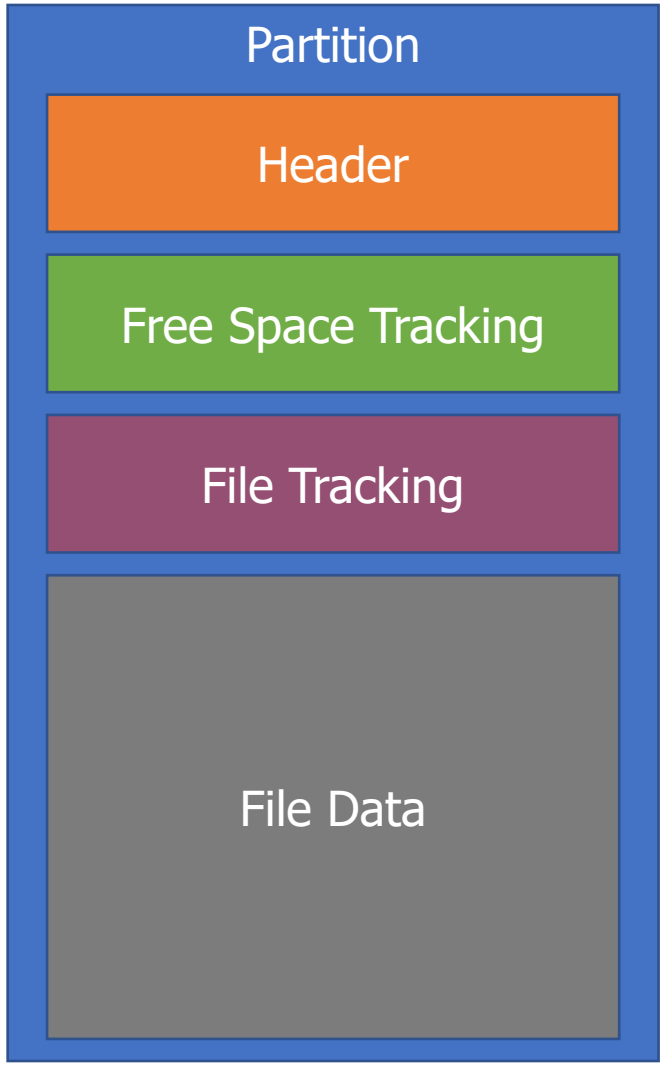
(not drawn to scale)

- Most 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

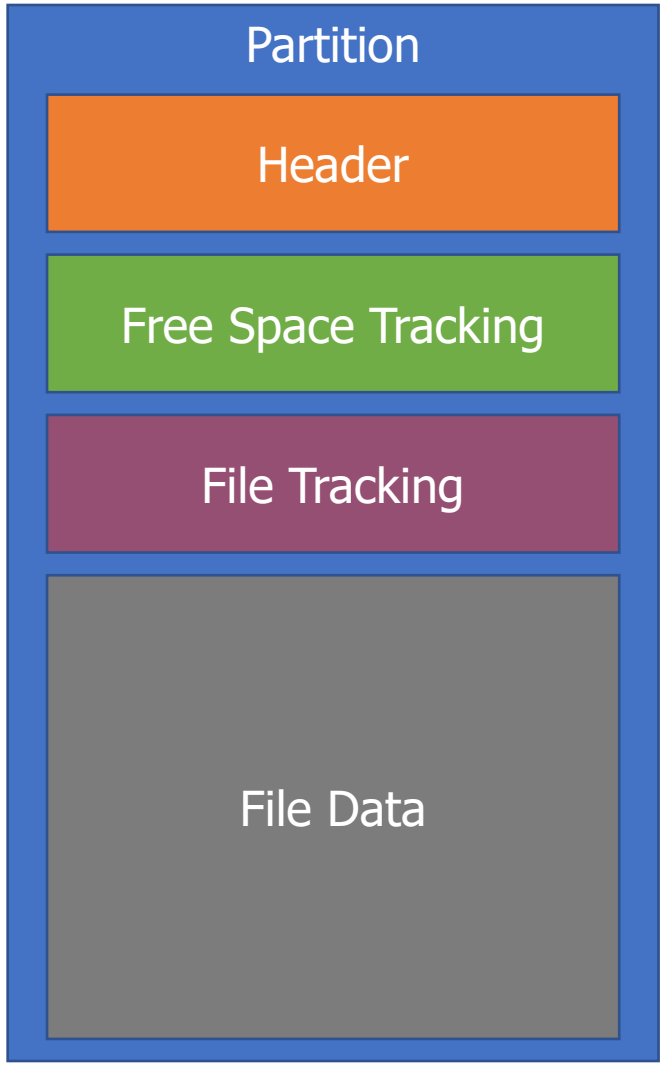
What goes within a partition?



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

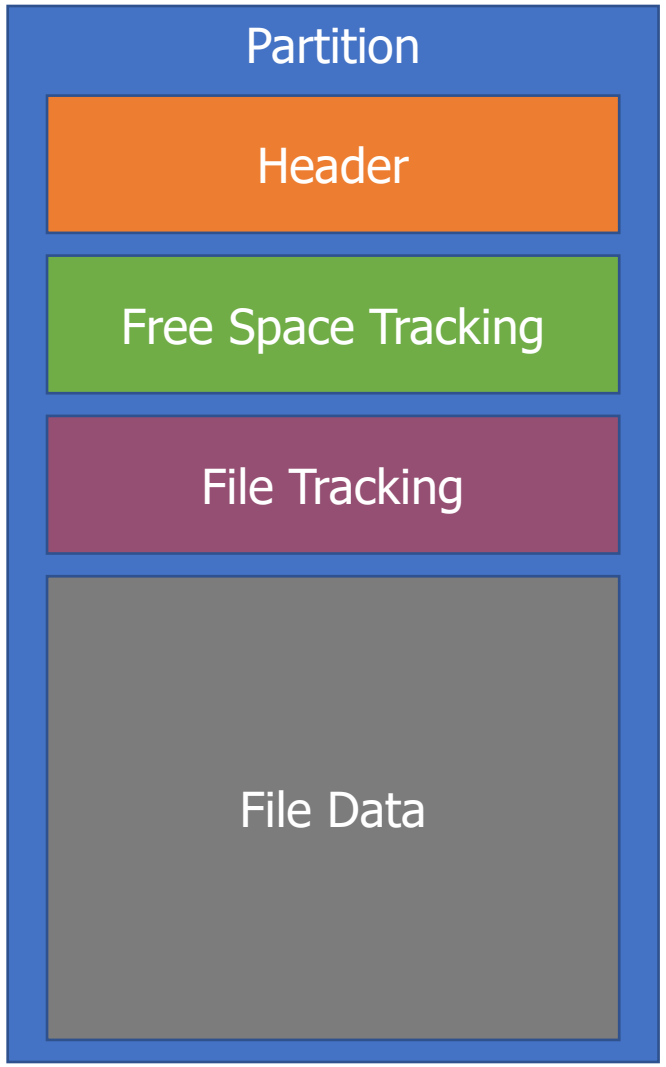
1. Header
2. Free Space Tracking
3. File Tracking
4. File Data

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

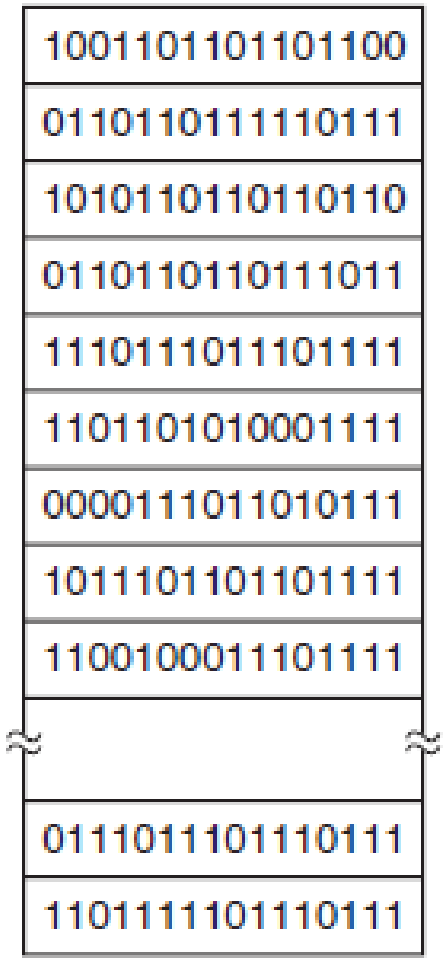
Tracking available blocks on a disk



- Free Space Tracking
  - Track which blocks in “File Data” are in use
- Could be a list of block addresses
  - 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
    - But 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 -> 250,000,000 blocks -> 250,000,000 bits -> 30 MB
- Bitmaps for tracking free blocks are a constant size for a disk
  - Upside: easy to work with
  - Downside: 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

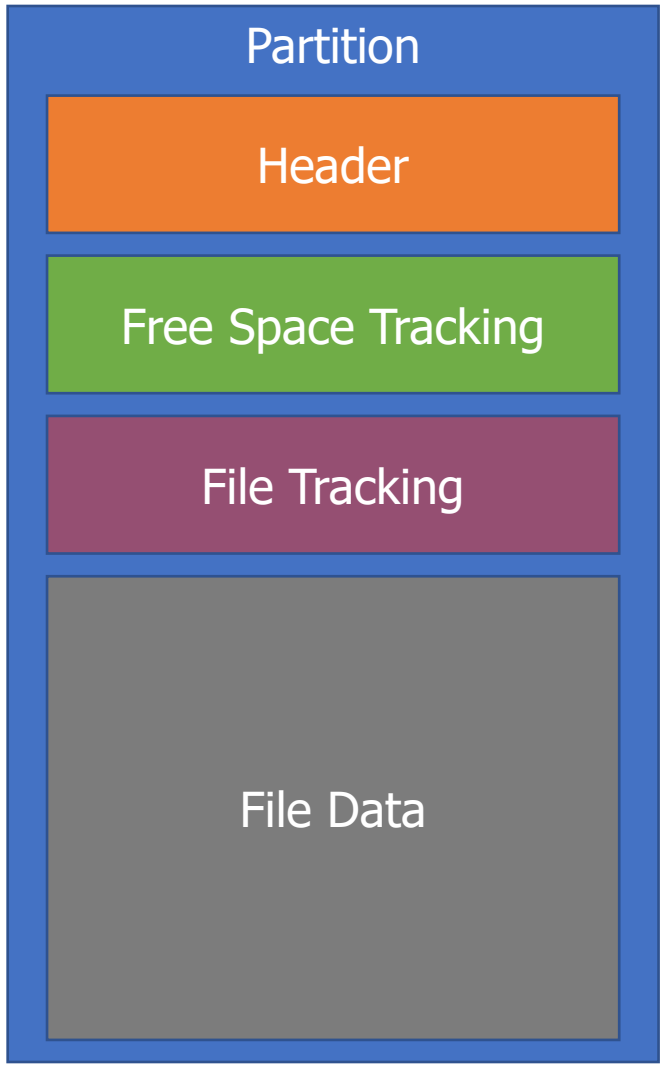


A bitmap

Outline

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

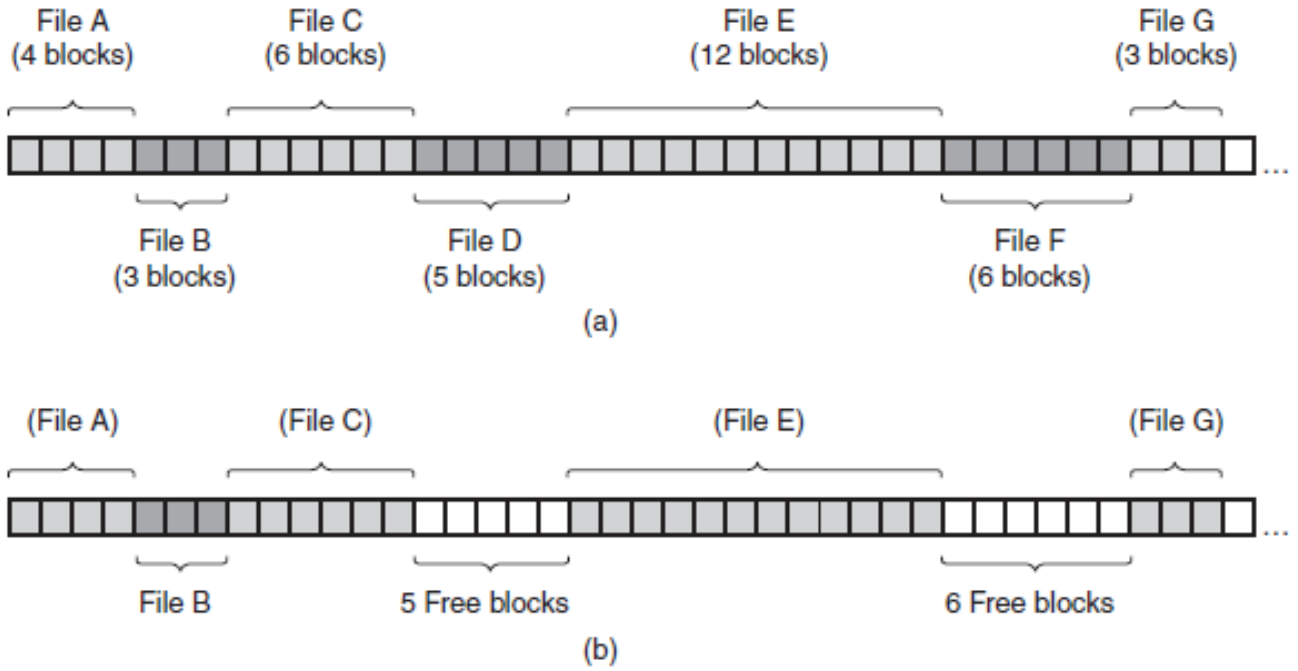
Tracking available blocks on a disk



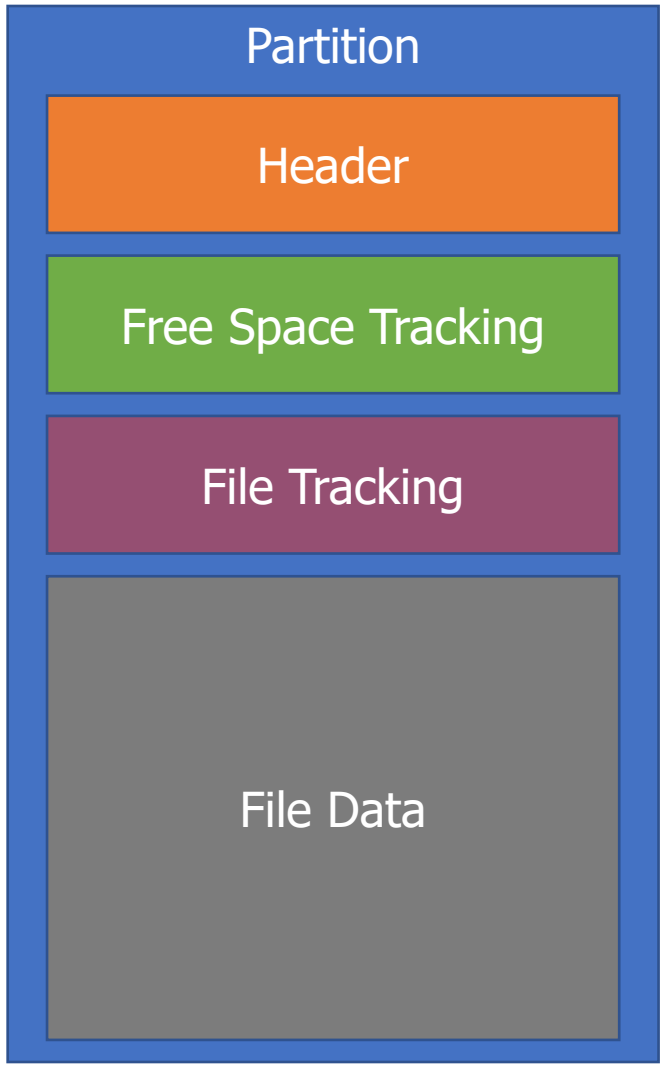
- File Tracking
  - File attributes
  - Ordered blocks where the file data is located
- Two common implementations
  - **Allocation Table**
    - FAT32
  - **Index Nodes** (inodes)
    - Unix File System, Fast File System,
    - ext3/ext4, NTFS

Requiring contiguous blocks won't work

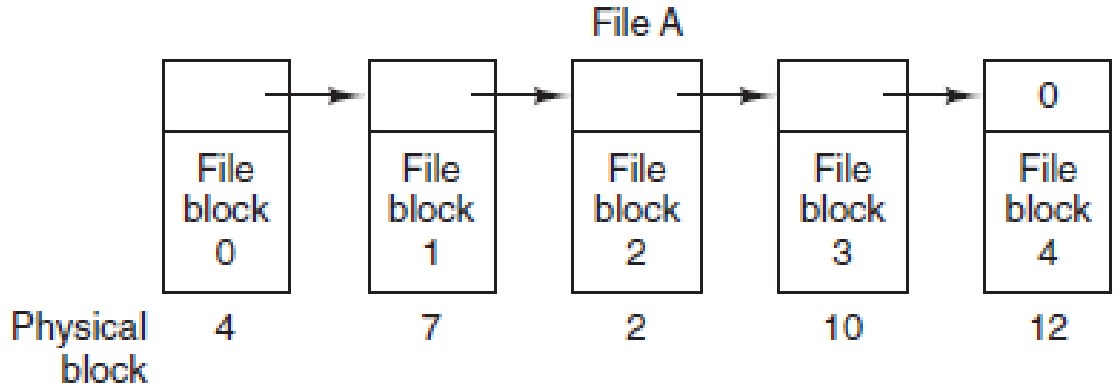
- Need ability to map random blocks to file
  - Files in contiguous blocks sounds nice
    - Sequential reads are fast
  - But *requiring* it leads to lots of fragmentation (unusable gaps in disk)



Forcing sequential access to data also won't work



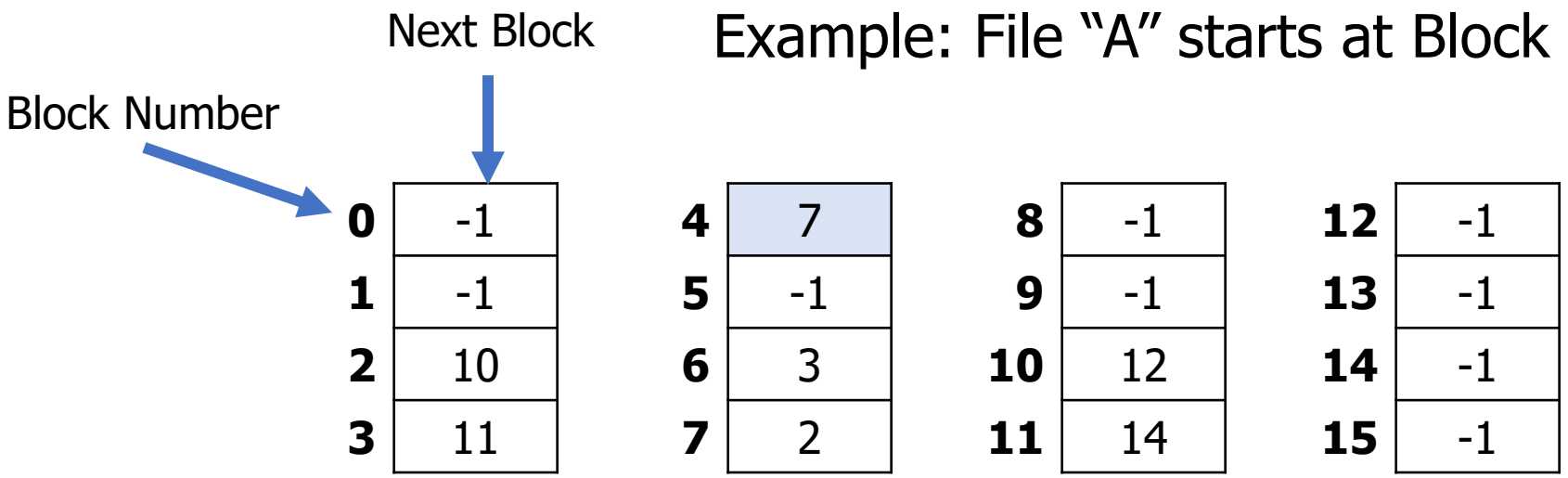
- Linked list in File Data is undesirable too
  - Must read **each block** in order to get next pointer
  - No random access to file
    - Appending requires reading through all of the file's blocks first



Allocation table – an example file tracking solution

- Keep the linked list idea, but keep a table of block pointers

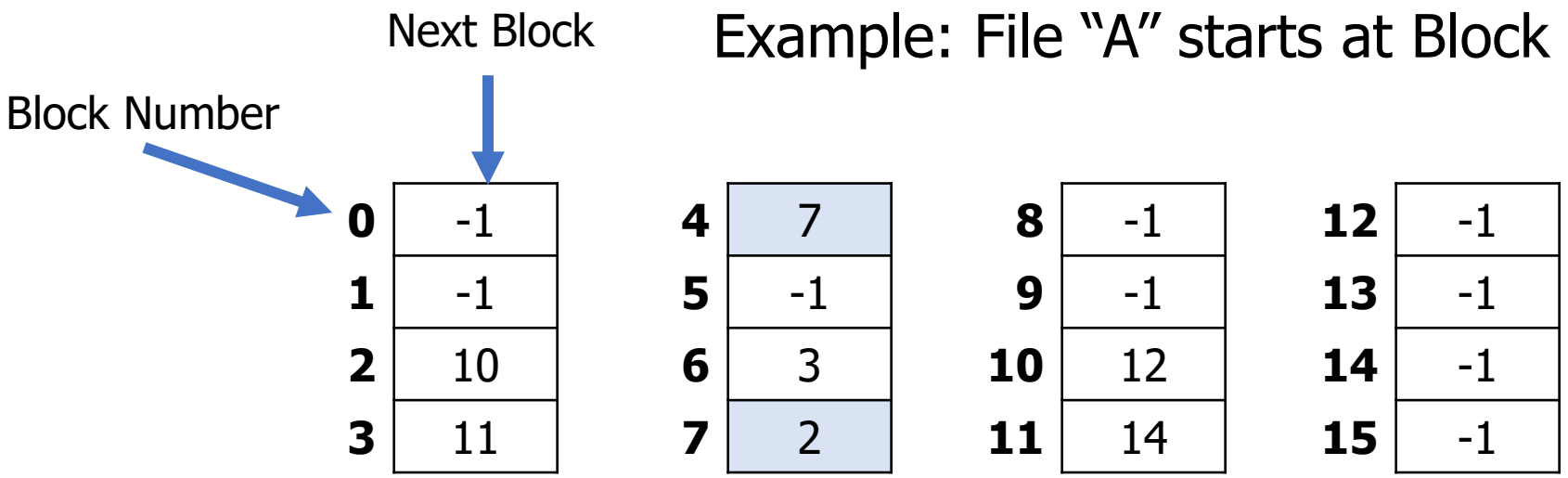
- Treat "File Tracking" block as an array of block pointers
  - Index into this array is the block number
  - Reading the file tracking block gets you all the pointers (so traversal is easy)



Allocation table

- Keep the linked list idea, but keep a table of block pointers

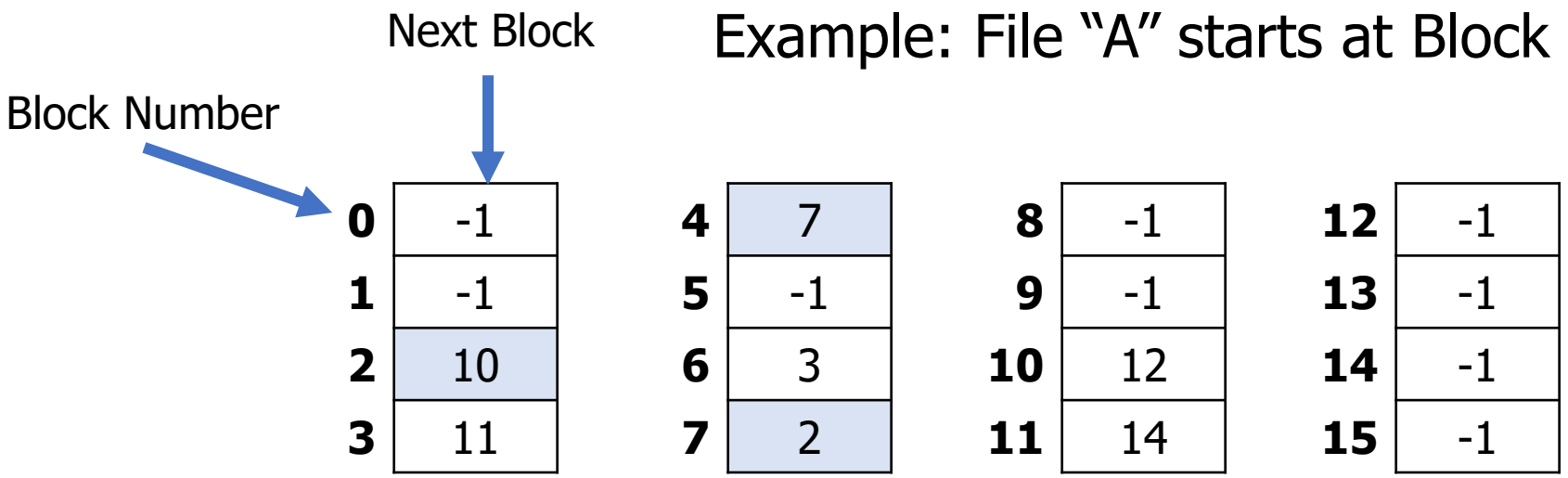
- Treat "File Tracking" block as an array of block pointers
  - Index into this array is the block number
  - Reading the file tracking block gets you all the pointers (so traversal is easy)



Allocation table

- Keep the linked list idea, but keep a table of block pointers

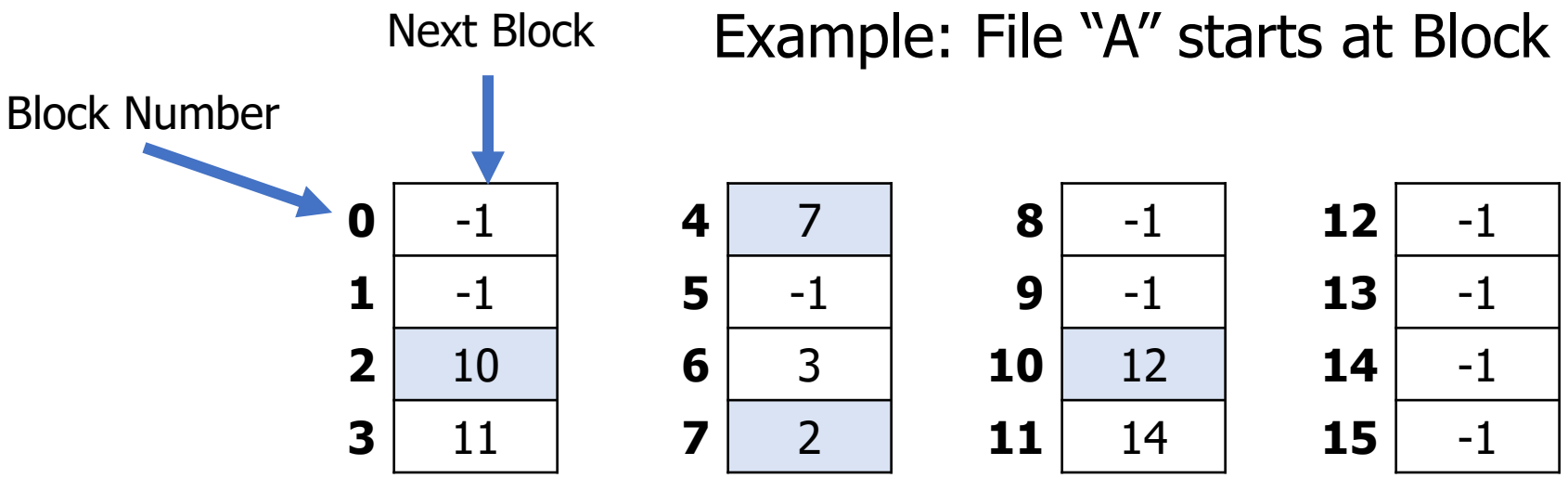
- Treat "File Tracking" block as an array of block pointers
  - Index into this array is the block number
  - Reading the file tracking block gets you all the pointers (so traversal is easy)



Allocation table

- Keep the linked list idea, but keep a table of block pointers

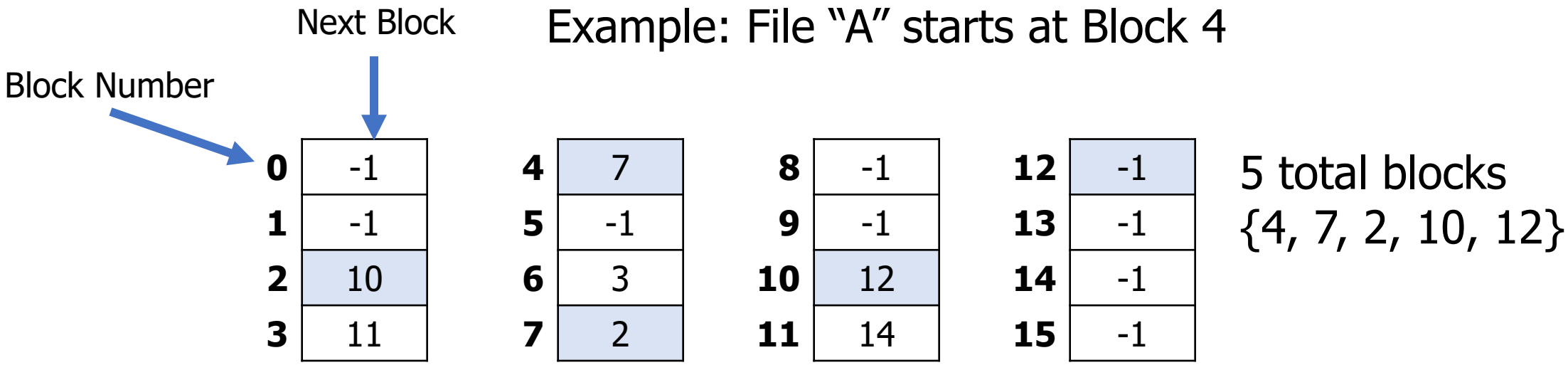
- Treat "File Tracking" block as an array of block pointers
  - Index into this array is the block number
  - Reading the file tracking block gets you all the pointers (so traversal is easy)



Allocation table

- Keep the linked list idea, but keep a table of block pointers

- Treat "File Tracking" block as an array of block pointers
  - Index into this array is the block number
  - Reading the file tracking block gets you all the pointers (so traversal is easy)



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

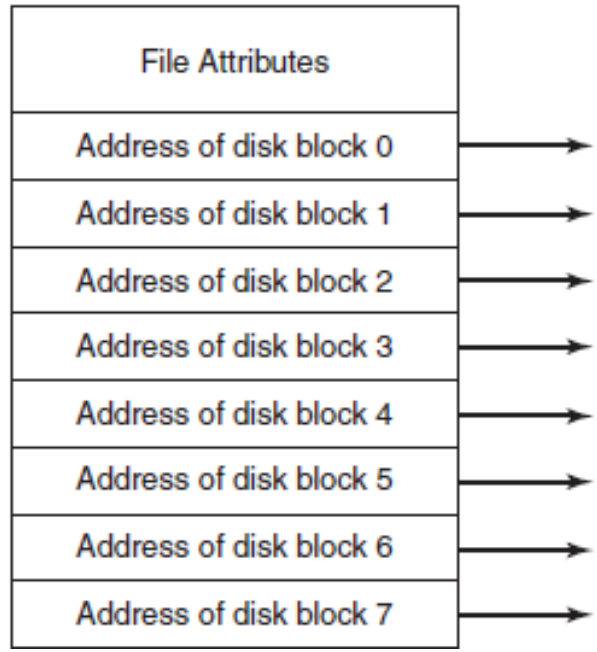
- Separation of attributes from block pointers is undesirable
  - Would be nice to have both of them in a single disk read
  - Or less than one read if they’re already in RAM

Index node (inode) – alternative file tracking solution

- Treat "File Tracking" as an array of inodes
  - Each inode corresponds to a single file
  - Size proportional to the number of files

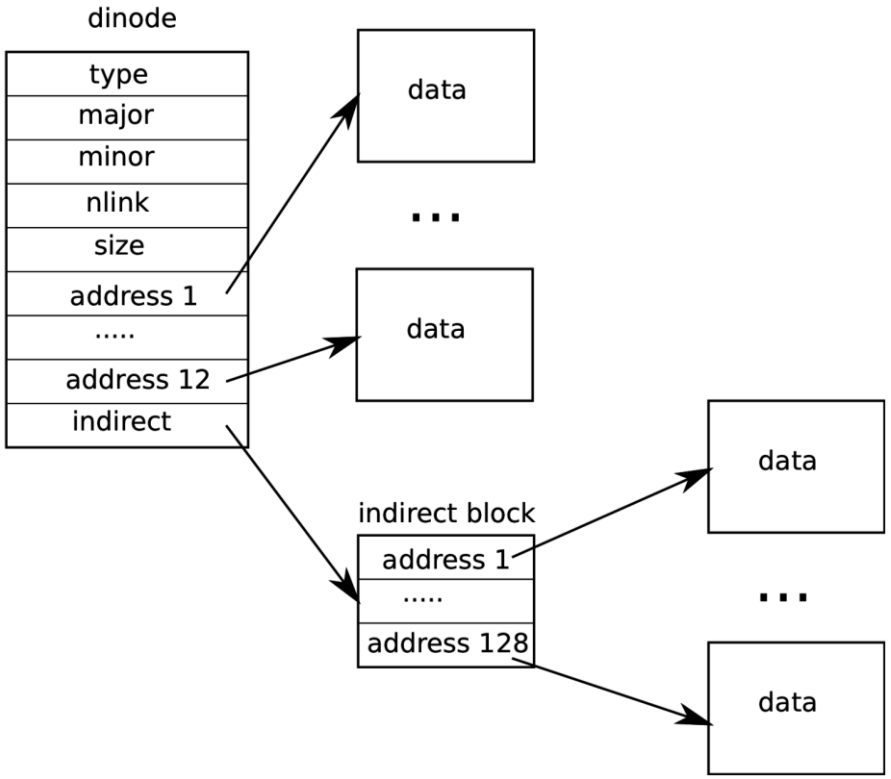
- inode contents
  - File attributes
  - Ordered list of pointers to data blocks for the file

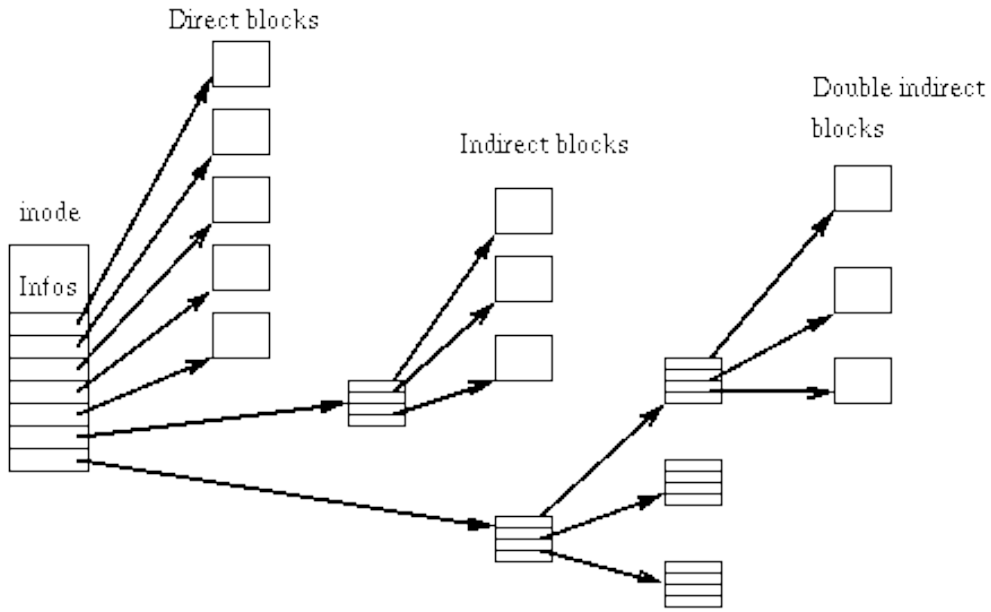
- Many improvements have sprung up
  - Optimization: coalesce contiguous blocks
  - Optimization: for very small files, put data right in the inode!



Hierarchical inodes allow for larger file sizes

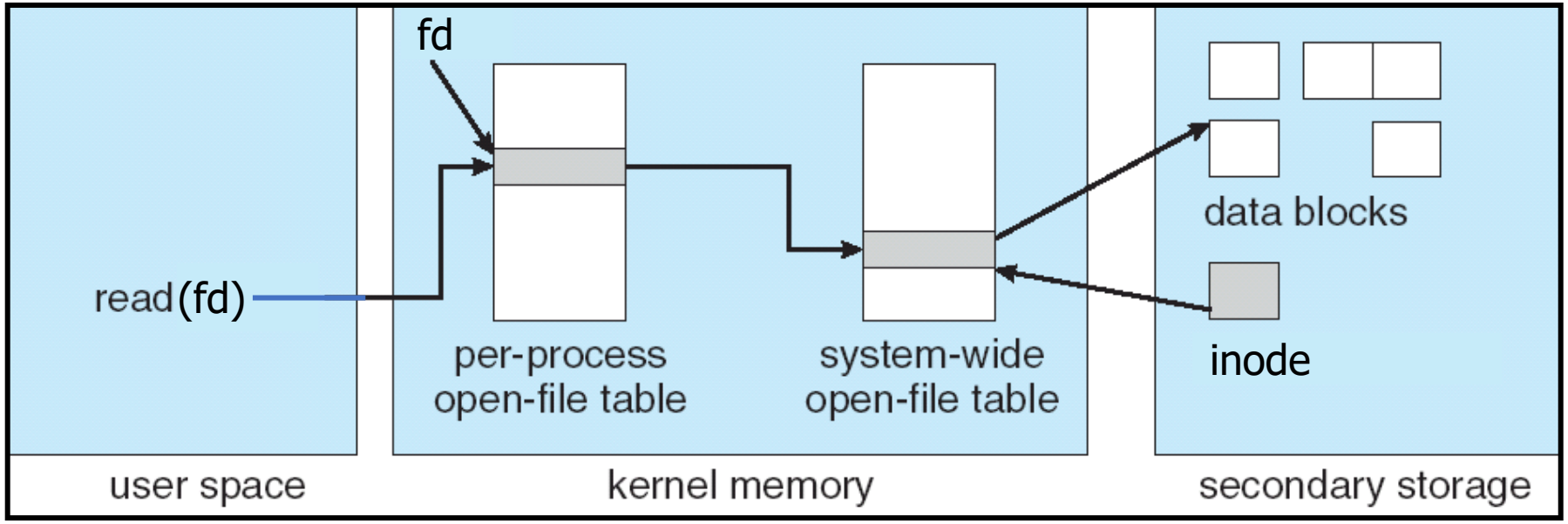
- Each inode is ≤ one block in size
  - So there would be a limit to how many blocks a file can have
  - Apply tree structure to block pointers to solve this





File system access with inodes

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



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

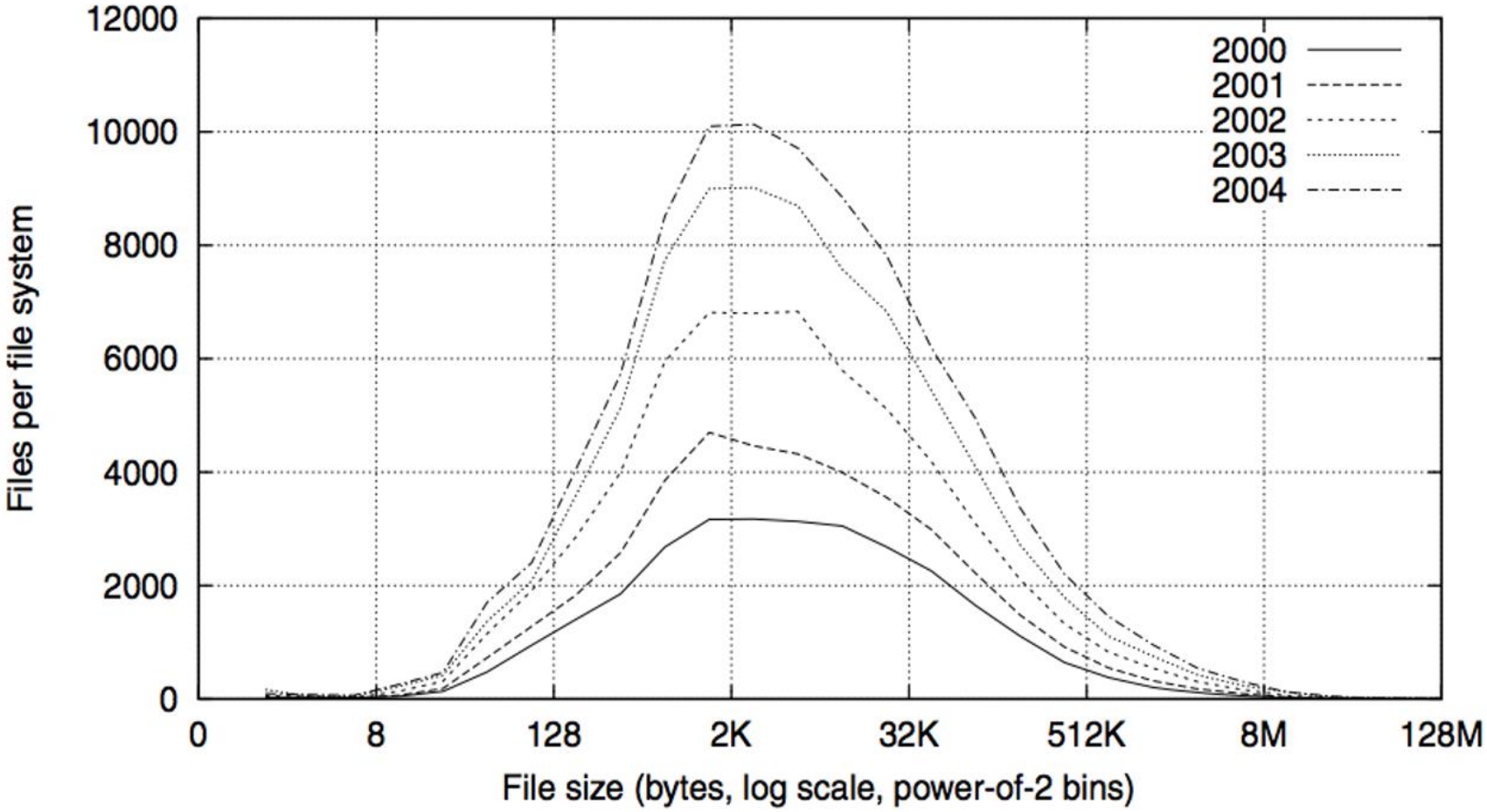


Fig. 2. Histograms of files by size.

2. Most bytes are spent on a few large files

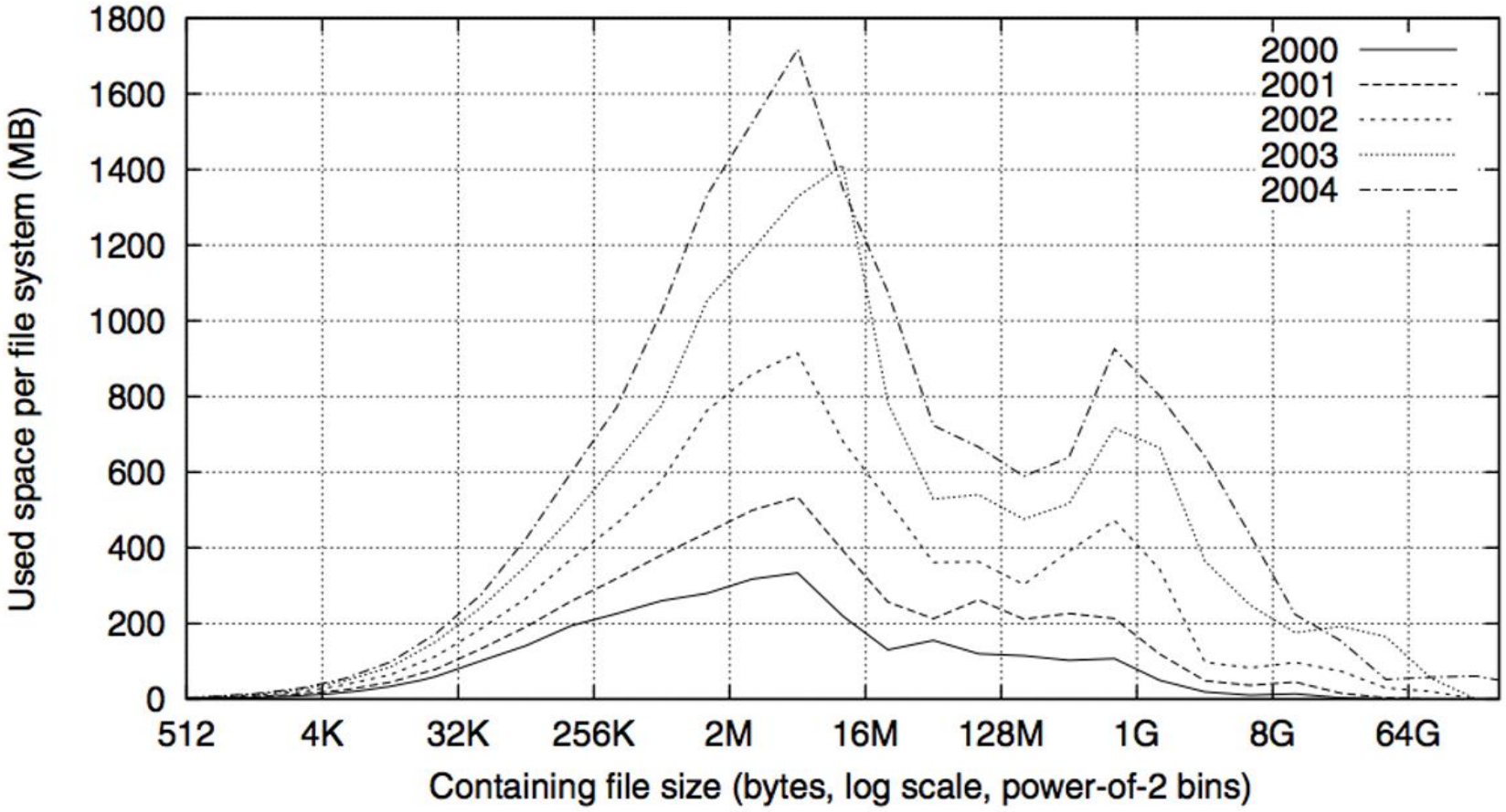


Fig. 4. Histograms of bytes by containing file size.

Break + Broader Thinking

- Study was on 60,000 Windows PC file systems in a large corporation from 2000-2004

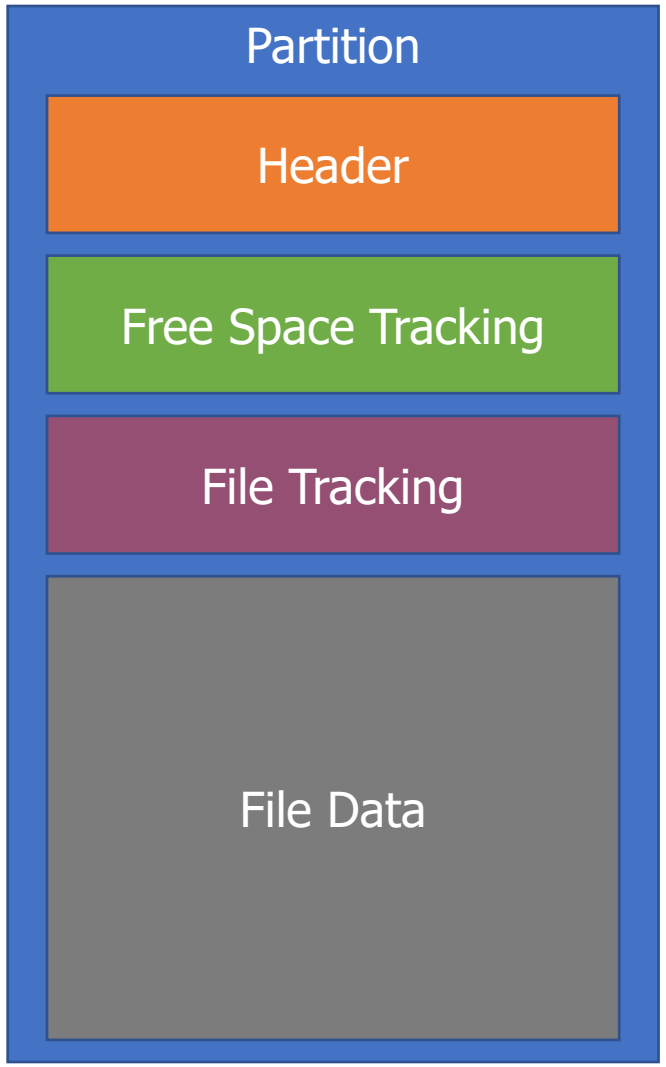
1. Does this still apply today? Why or why not?

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

Outline

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

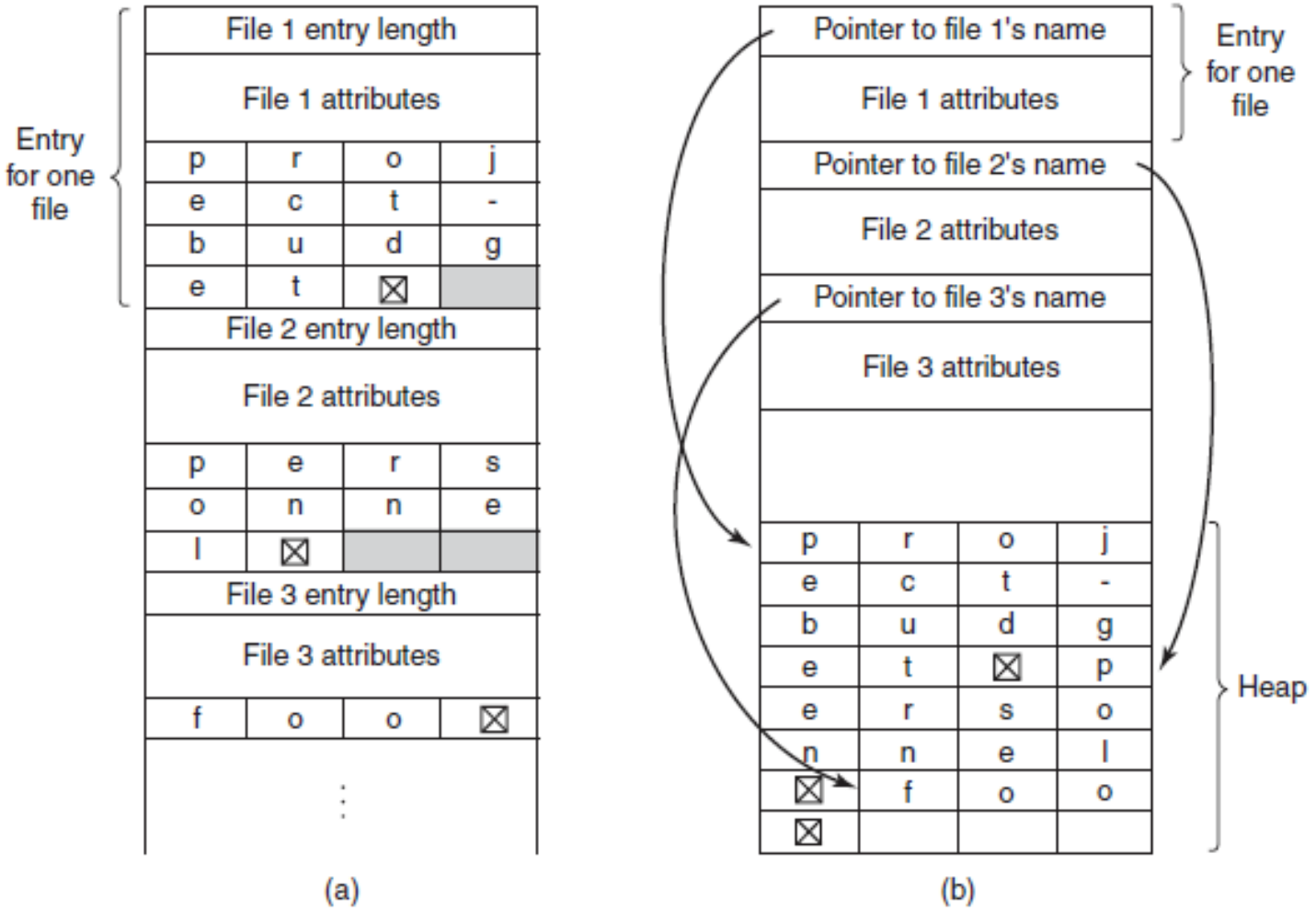
What goes in the file data?



- Normal files
  - Just the file's data
  - Attributes already handled in inode
- Directories
  - Structure listing files within this directory
    - 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

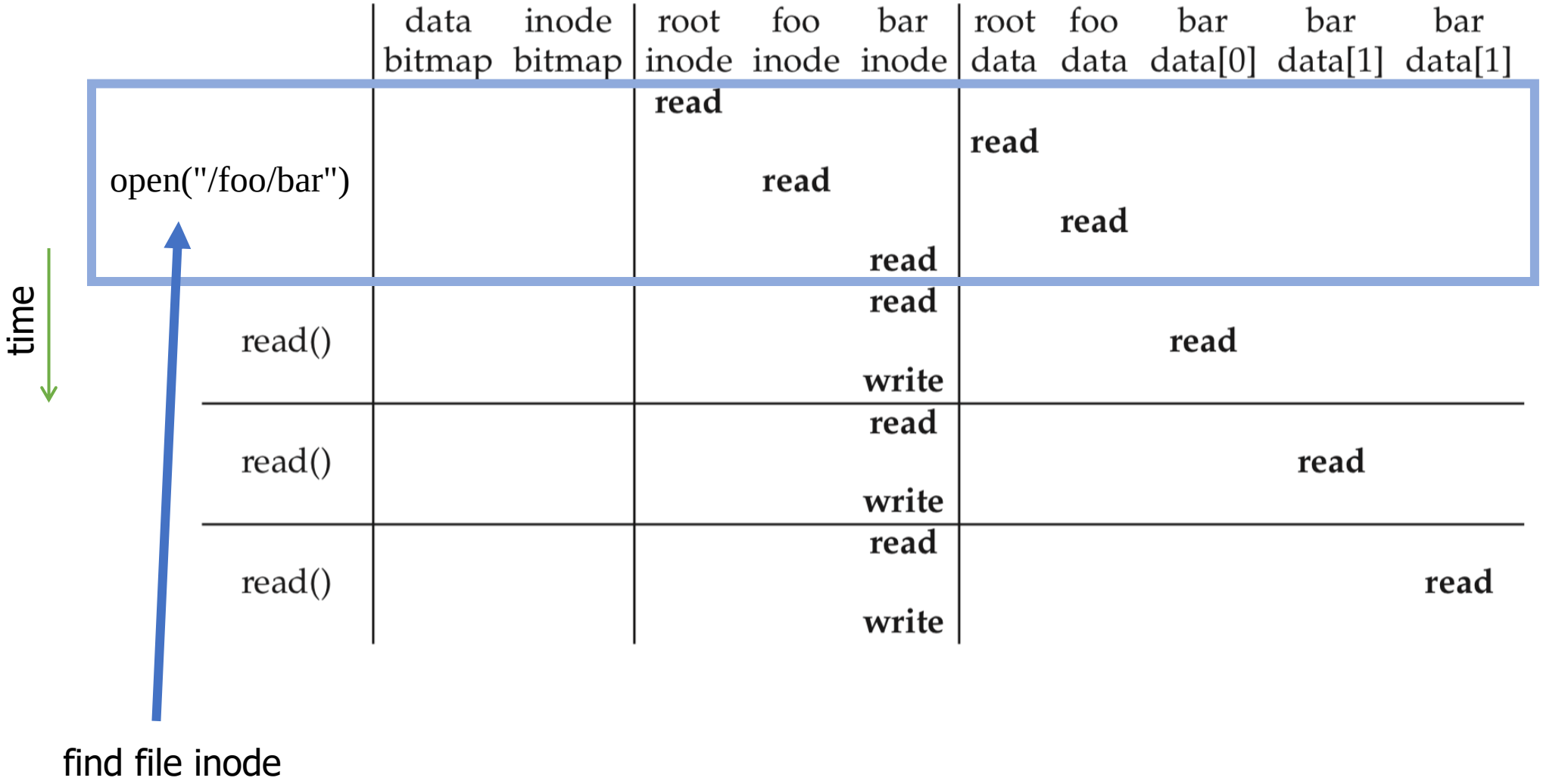
- Introduction to filesystems
- Application view
- Parts of a file system
  - Managing disk
  - Tracking files
  - Handling file data
- **Whole filesystem example**

A trace through the filesystem

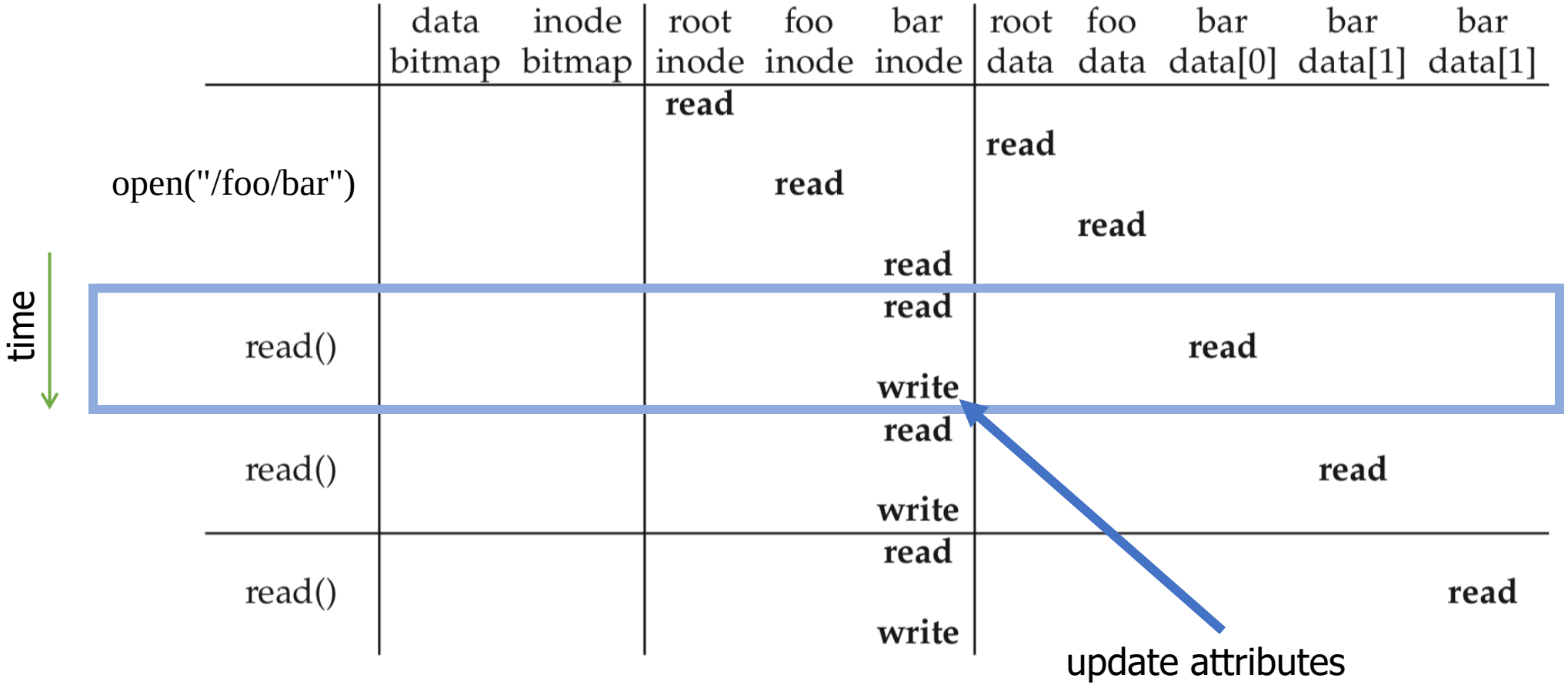
- Now we have enough knowledge to walk through an entire filesystem access

- 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

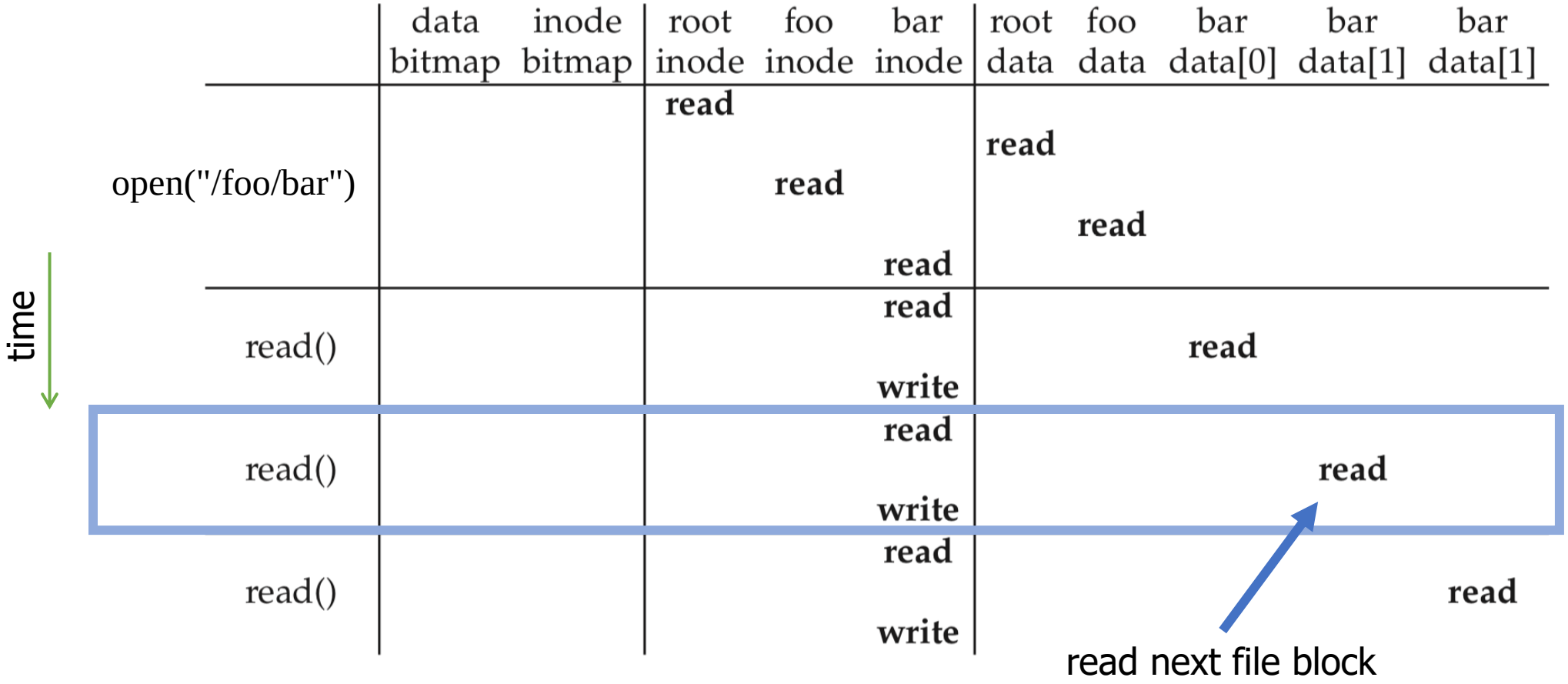
Open and read example



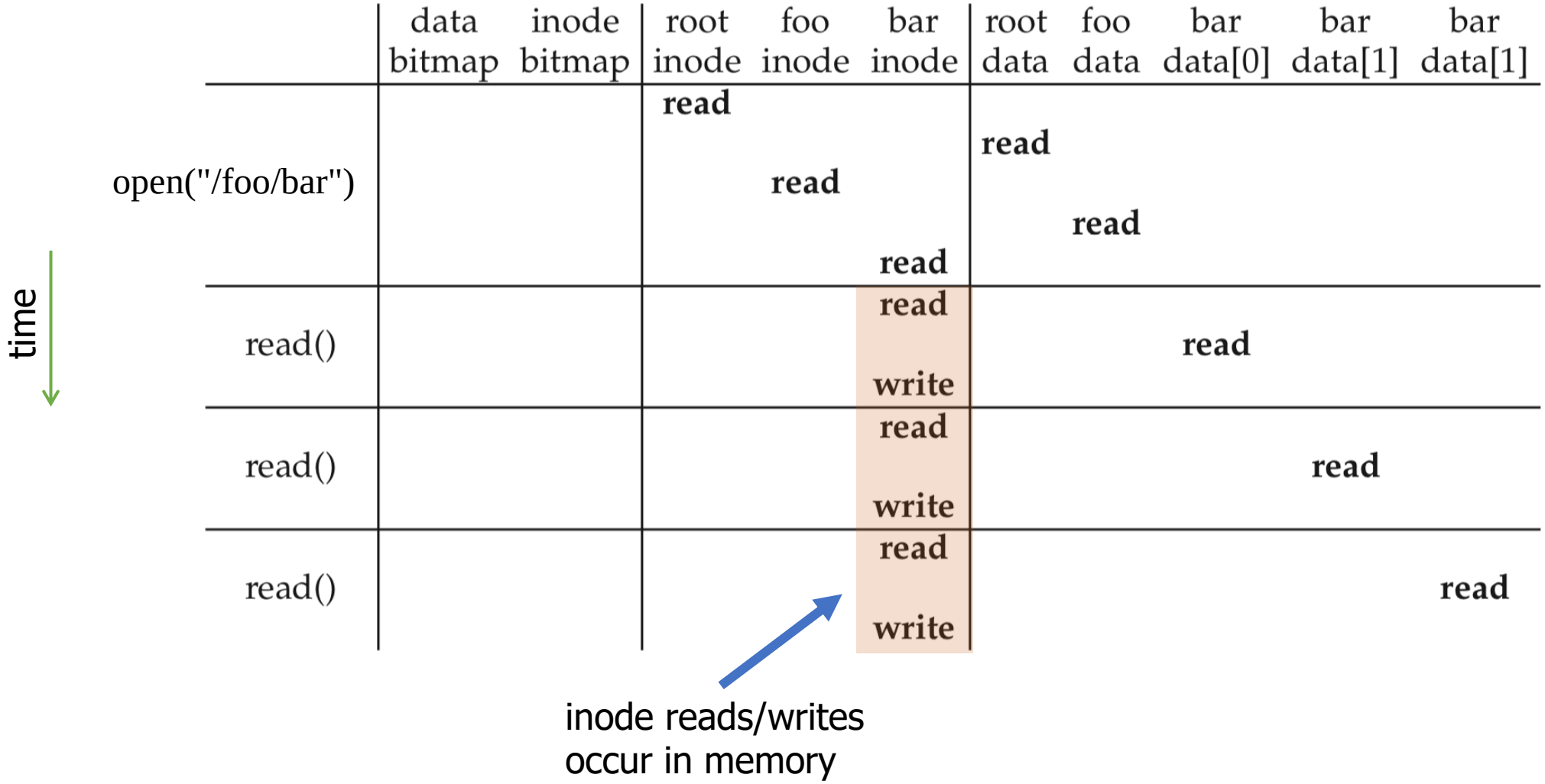
Open and read example



Open and read example



Open and read example



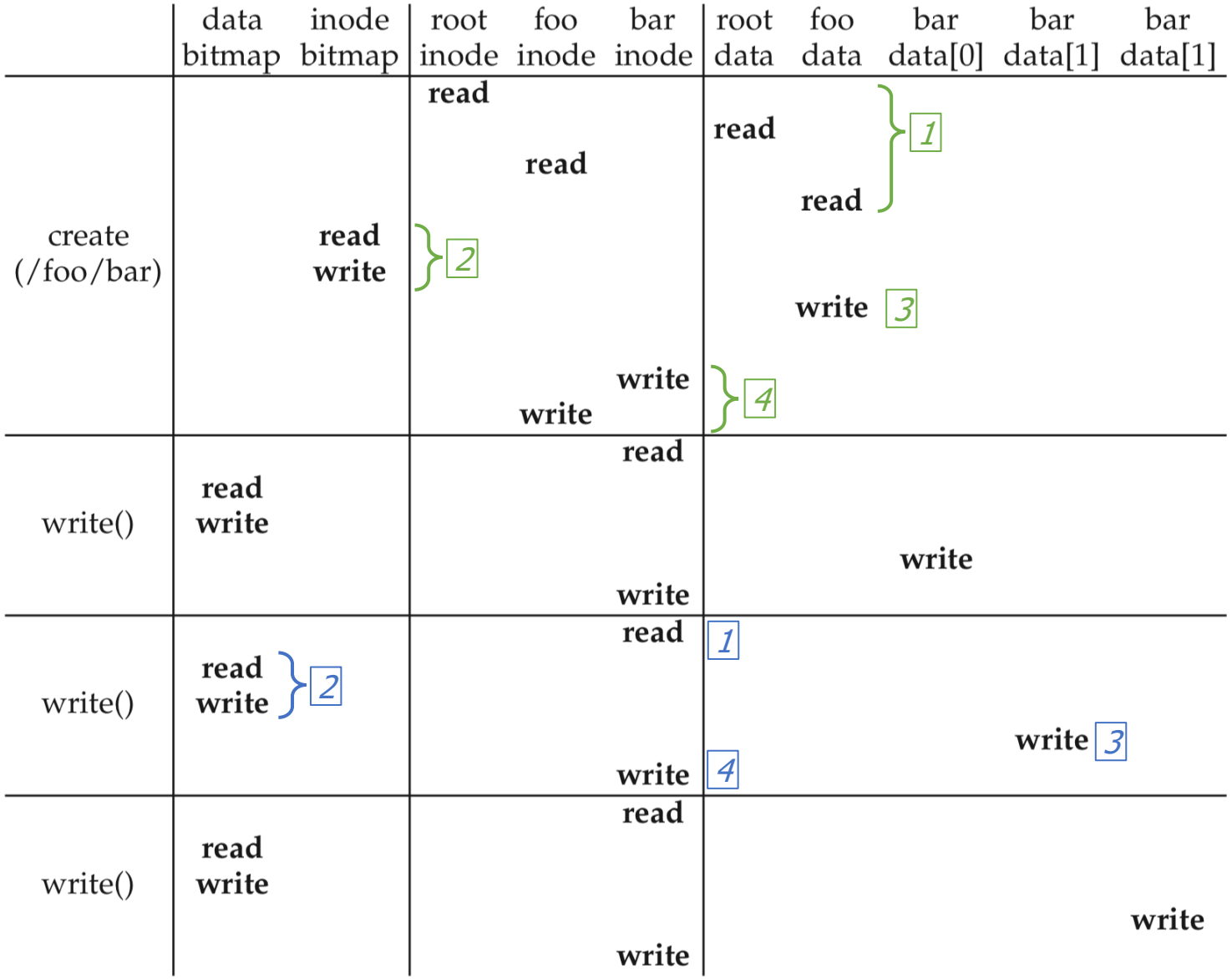
Create and write a file

	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 write	read 2	read write	write	read	read write	1 write 3		
write()	read write				read write			write		
write()	read write				read write				write	
write()	read write				read 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 <"name", inode#> to parent directory.
4. Fill-in file metadata.

Create and write a file



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