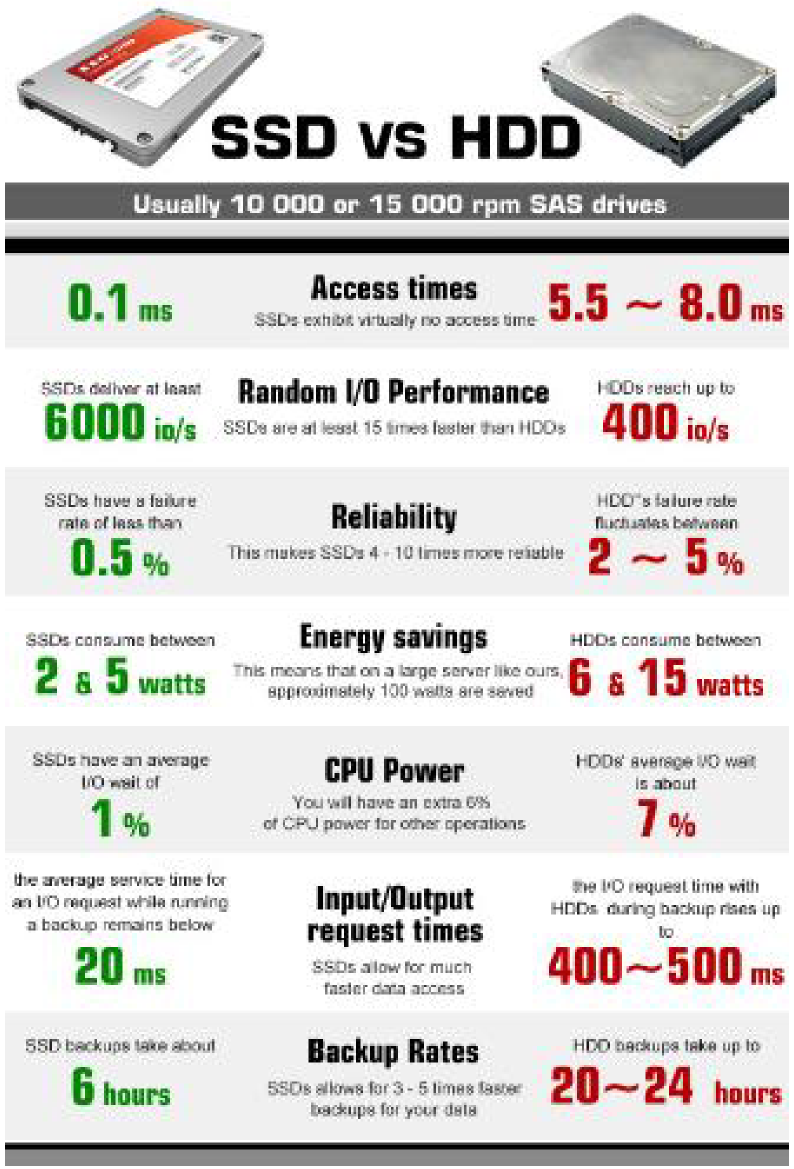
CS162
Operating Systems and Systems Programming
Lecture 19

File Systems

Professor Natacha Crooks
https://cs162.org/

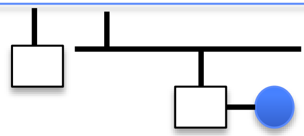
Slides based on prior slide decks from David Culler, Ion Stoica, John Kubiatowicz, Alison Norman and Lorenzo Alvisi

Recall: HDD vs. SSD Comparison

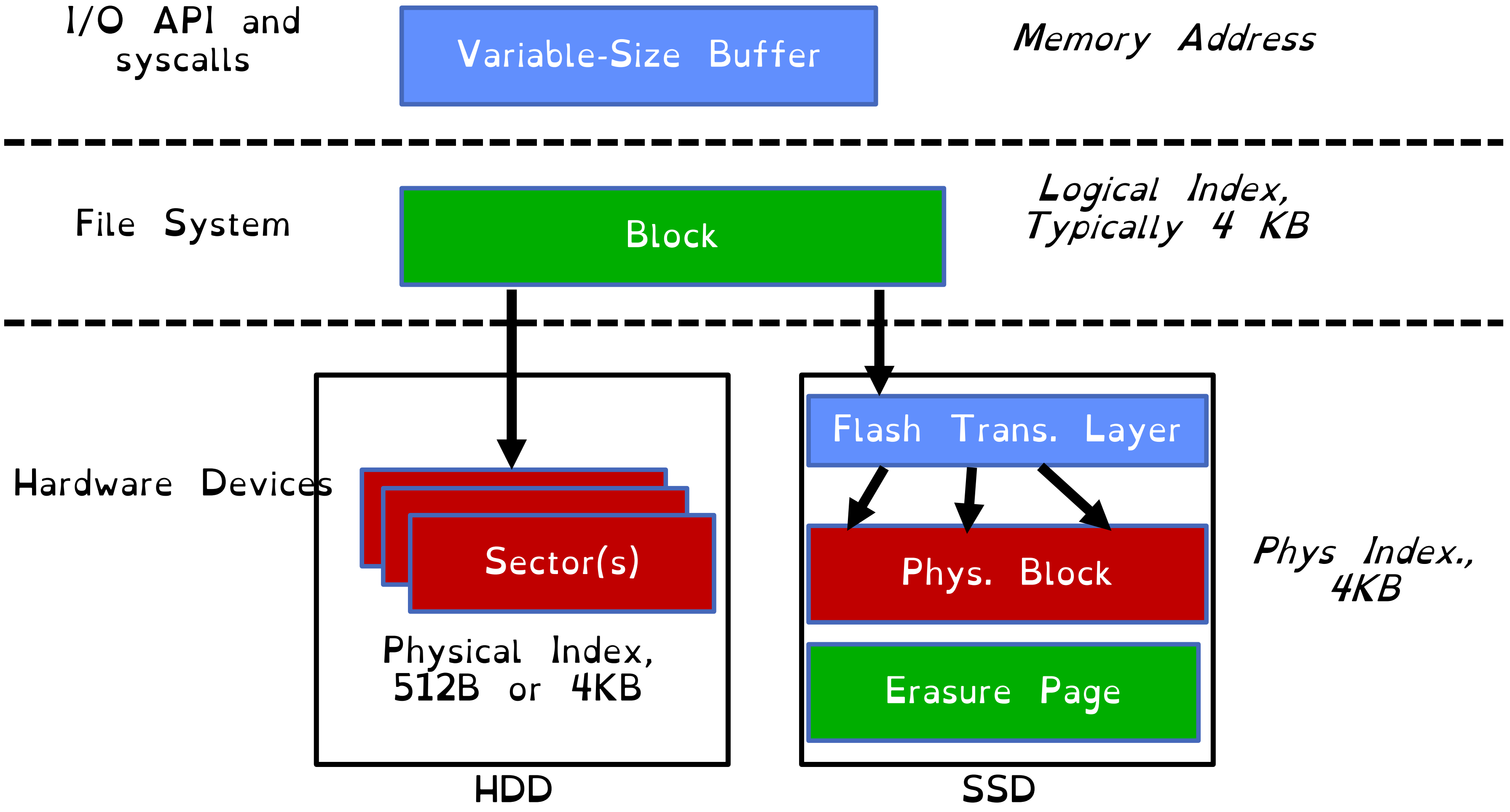


HDD	SDD
Require seek + rotation	No seeks
Not parallel (one head)	Parallel
Brittle (moving parts)	No moving parts
Random reads take 10s milliseconds	Random reads take 10s microseconds
Slow (Mechanical)	Wears out
Cheap/large storage	Expensive/smaller storage

Recall: I/O and Storage Layers

High Level I/O	*Streams*
Low Level I/O	*File Descriptors*
Syscall	*open(), read(), write(), close(), …*
	Open File Descriptions
File System	*Files/Directories/Indexes*
I/O Driver	*Commands and Data Transfers*
	*Disks, Flash, Controllers, DMA*

From Storage to File Systems



Building a File System

Layer of OS that transforms block interface of disks (or other block devices) into Files, Directories, etc.

Building a File System

Take limited hardware interface (array of blocks) and provide a more convenient/useful interface with:

Naming: Find file by name, not block numbers

Structure: Organize file names with directories

Organization: Map files to blocks

Protection: Enforce access restrictions

Reliability: Keep files intact despite crashes, failures, etc.

User vs. System View of a File

User's view:

Durable Data Structures

System's view (system call interface):

Collection of Bytes (UNIX)

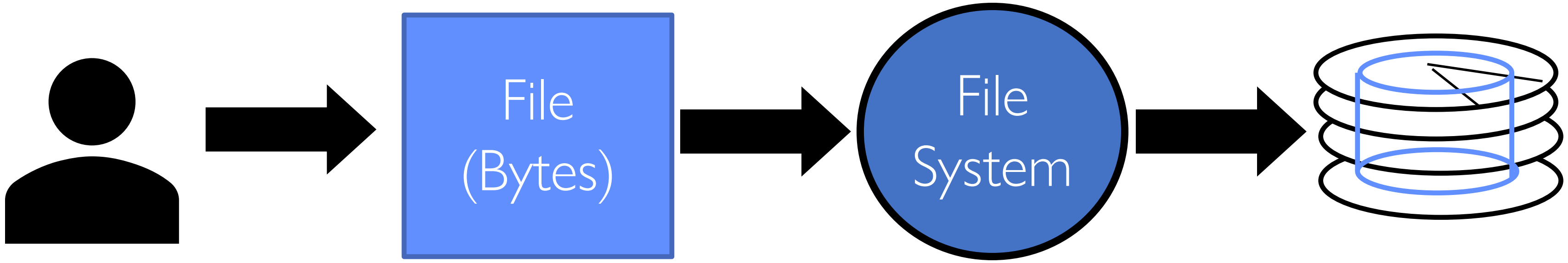
Doesn't matter to system what kind of data structures you want to store on disk!

System's view (inside OS):

Collection of blocks (a block is a logical transfer unit, while a sector is the physical transfer unit)

Block size ≥ sector size; in UNIX, block size is 4KB

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

Disk Management

File:
User-visible group of blocks arranged sequentially in logical space

Directory:
User-visible index mapping names to files

Logical Block Addressing (LBA)

The disk is accessed as linear array of sectors
Every sector has integer address
Controller translates from address $\Rightarrow$ physical position
Shields OS from structure of disk

What Does the File System Need?

Track free disk blocks

Need to know where to put newly written data

Track which blocks contain data for which files

Need to know where to read a file from

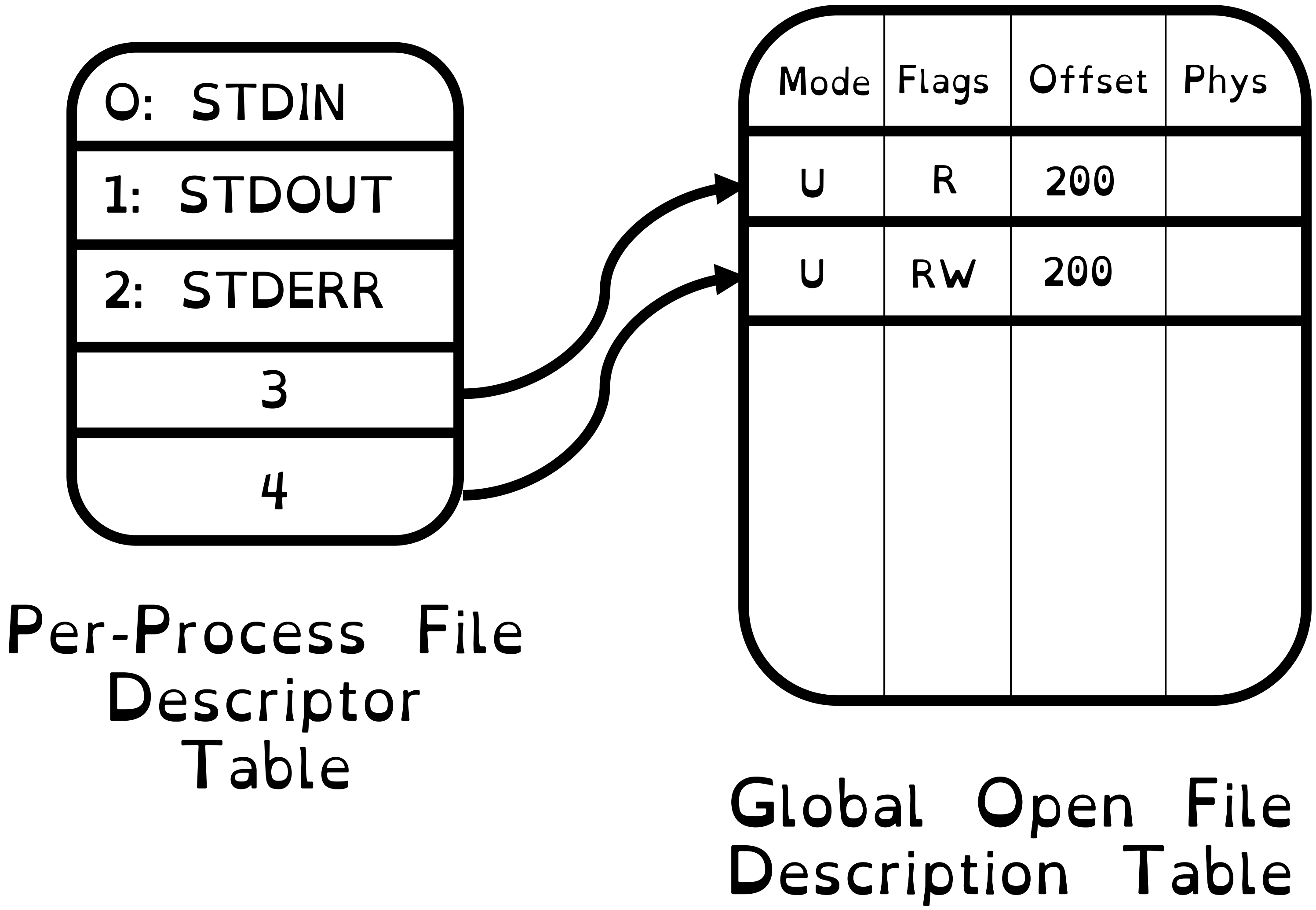
Track files in a directory

Find list of file's blocks given its name

Where do we maintain all of this?

Somewhere on disk

Recall: FD & File Descriptors



Critical Factors in File System Design

(Hard) Disks Performance !!!

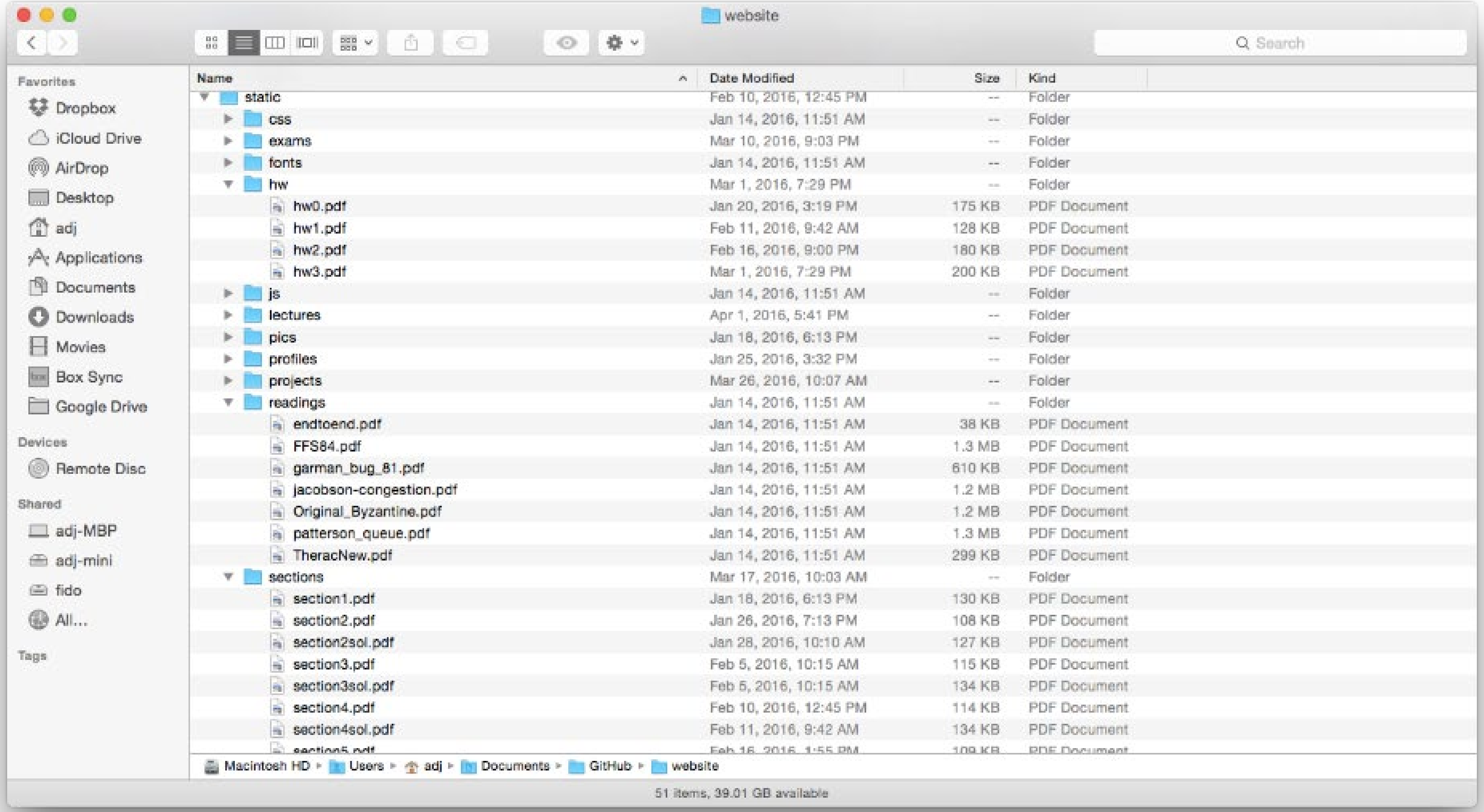
Open before Read/Write

Size is determined as they are used !!!

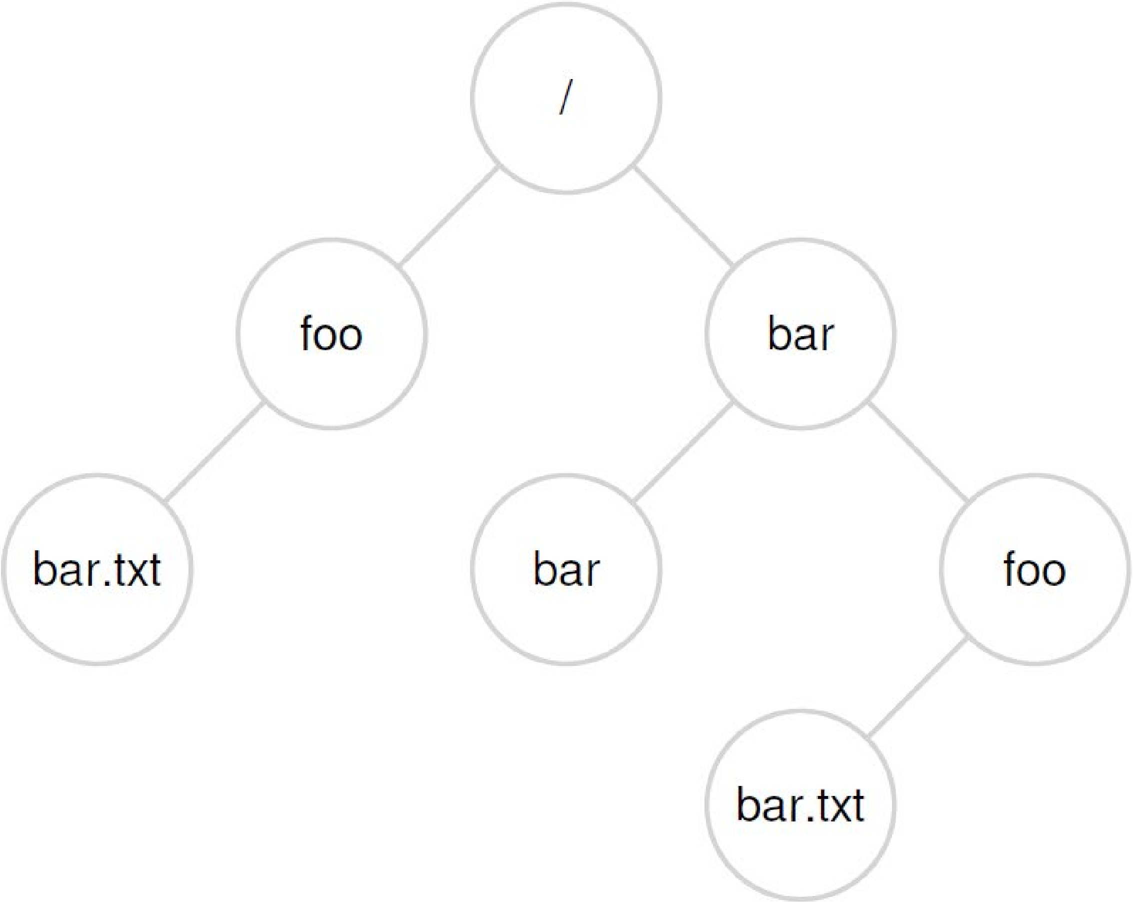
Organized into directories

Need to carefully allocate / free blocks

Files & Directories



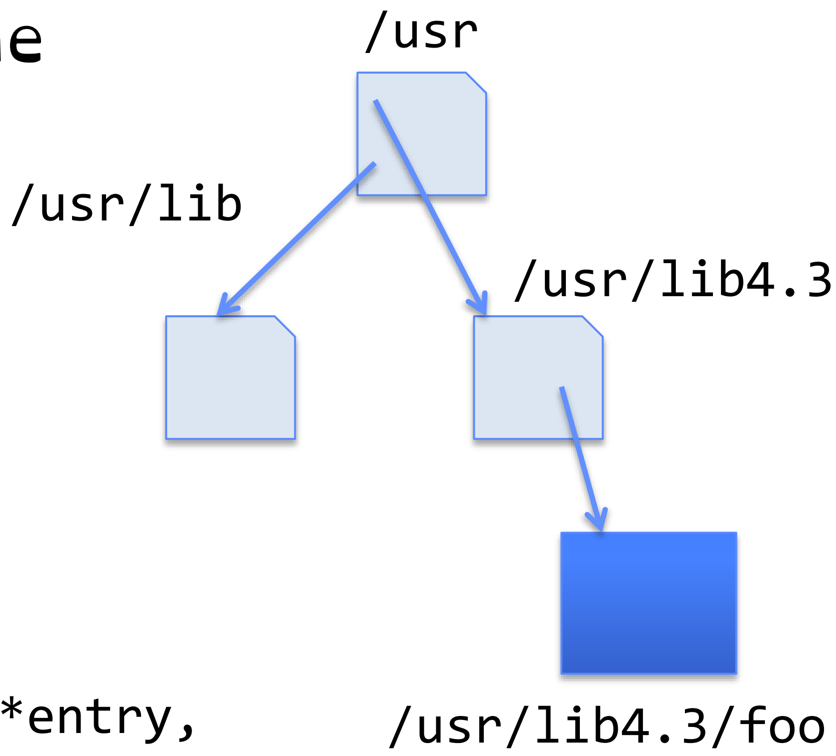
Files & Directories



Manipulating directories

System calls to access directories

- open / creat / readdir traverse the structure
- mkdir / rmdir add/remove entries
- link / unlink (rm)



libc support

- DIR * opendir (const char *dirname)
- struct dirent * readdir (DIR *dirstream)
- int readdir_r (DIR *dirstream, struct dirent *entry, struct dirent **result)

Components of a File System

Superblock object: information about file system

Free bitmaps: what is allocated/not allocated

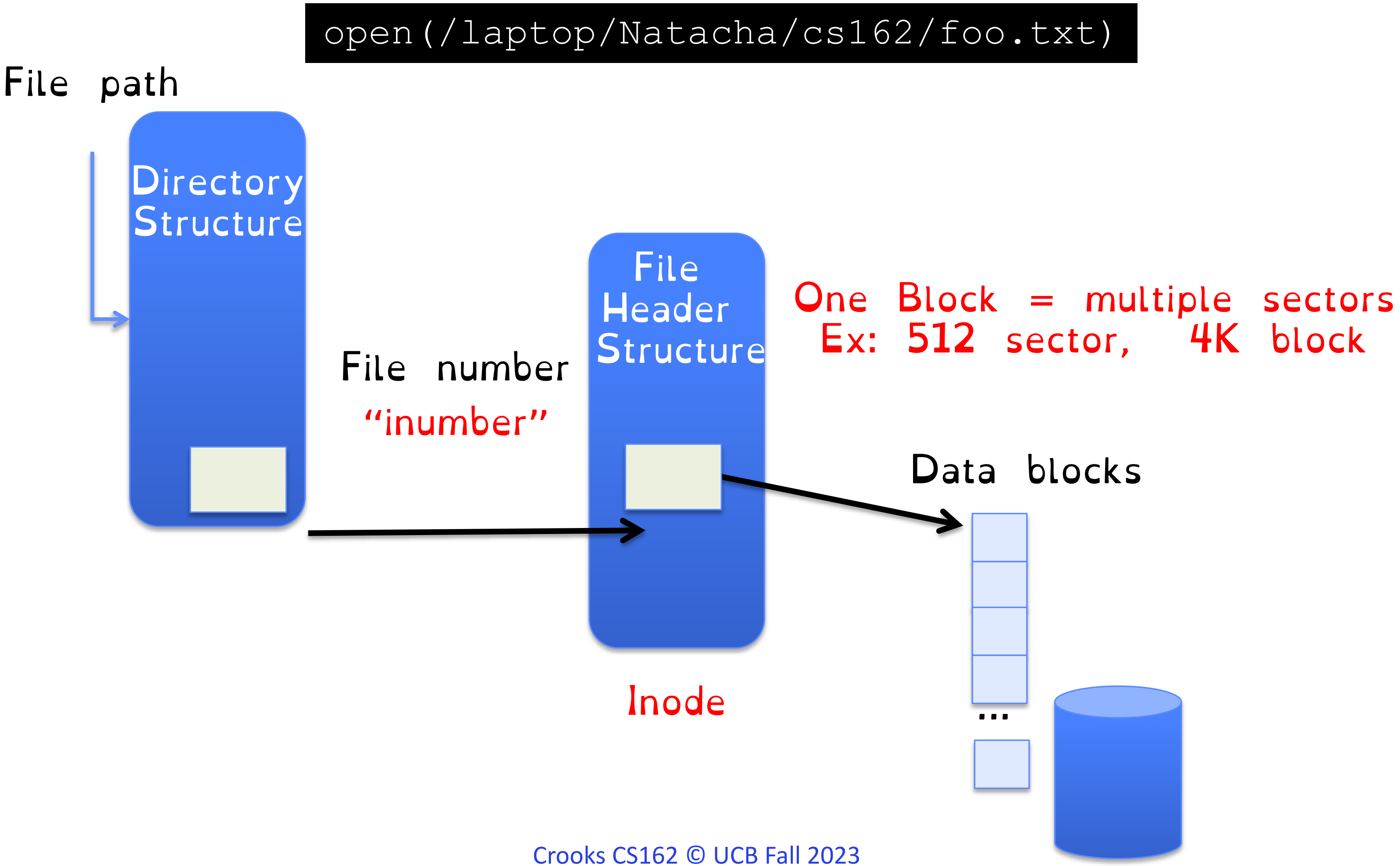
Inode object: represents a specific file

Dentry object: directory entry, single component of a path

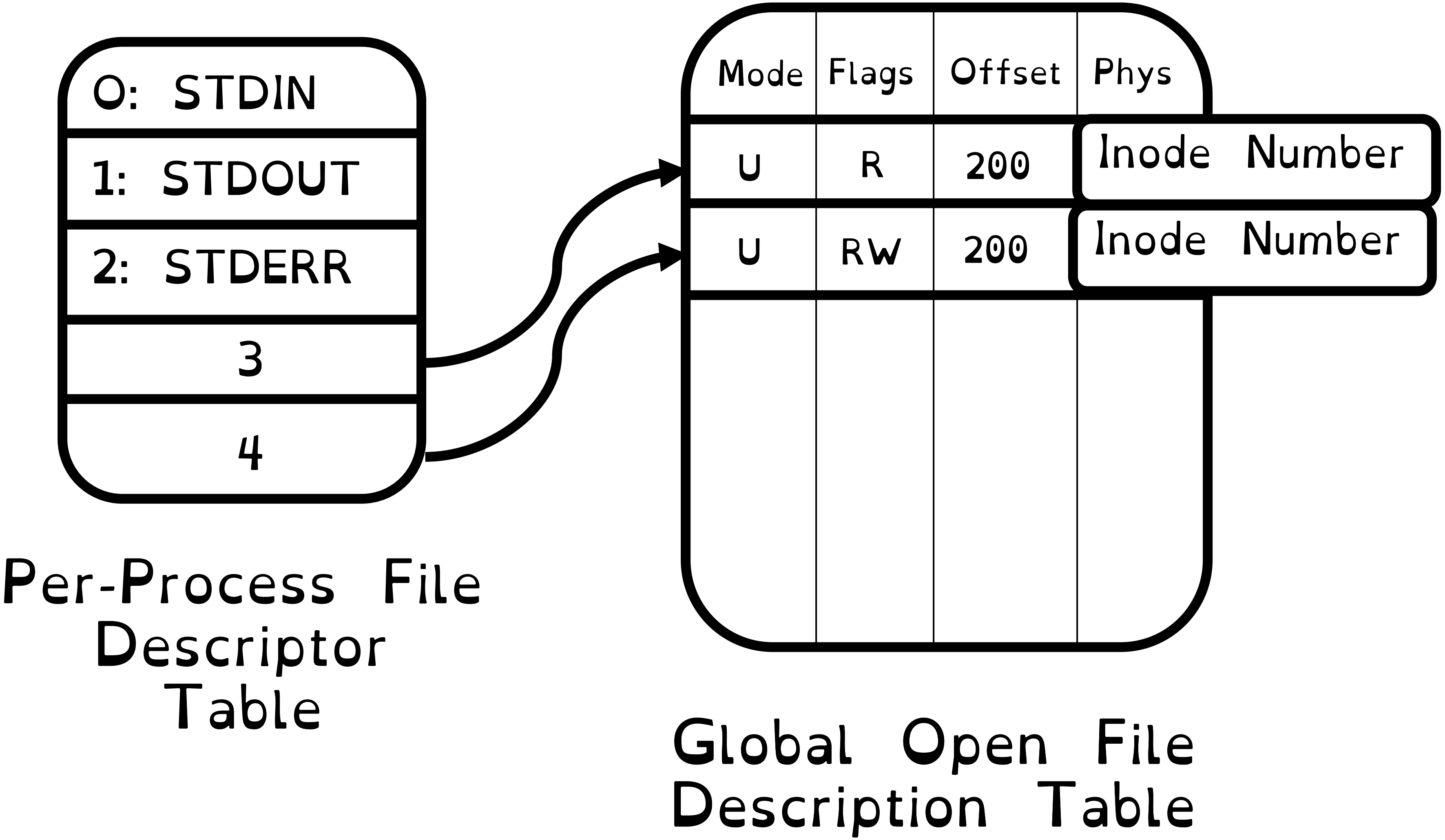
File object: open file associated with a process.

Blocks: How files are stored on disk

Components of a File System



The (In)famous Inode



How to get the Inode number?

Look up in directory structure

Directory is a specialised file containing <file_name : inode number> mappings

File number could be a file or another directory

Each <file_name : inode> mapping is called a directory entry

How to read a file from disk

Let's read file /foo/bar.txt (Time goes downwards)

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data [0]	bar data [1]	bar data [2]
open(bar)			**read**							
						read				
				read						
							read			
					read					
read()					**read**					
								read		
					write					
read()					**read**					
									read	
					write					
read()					**read**					
										read
					write					

Characteristics of Files

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

Published in FAST 2007

Observation #1: Most Files Are Small

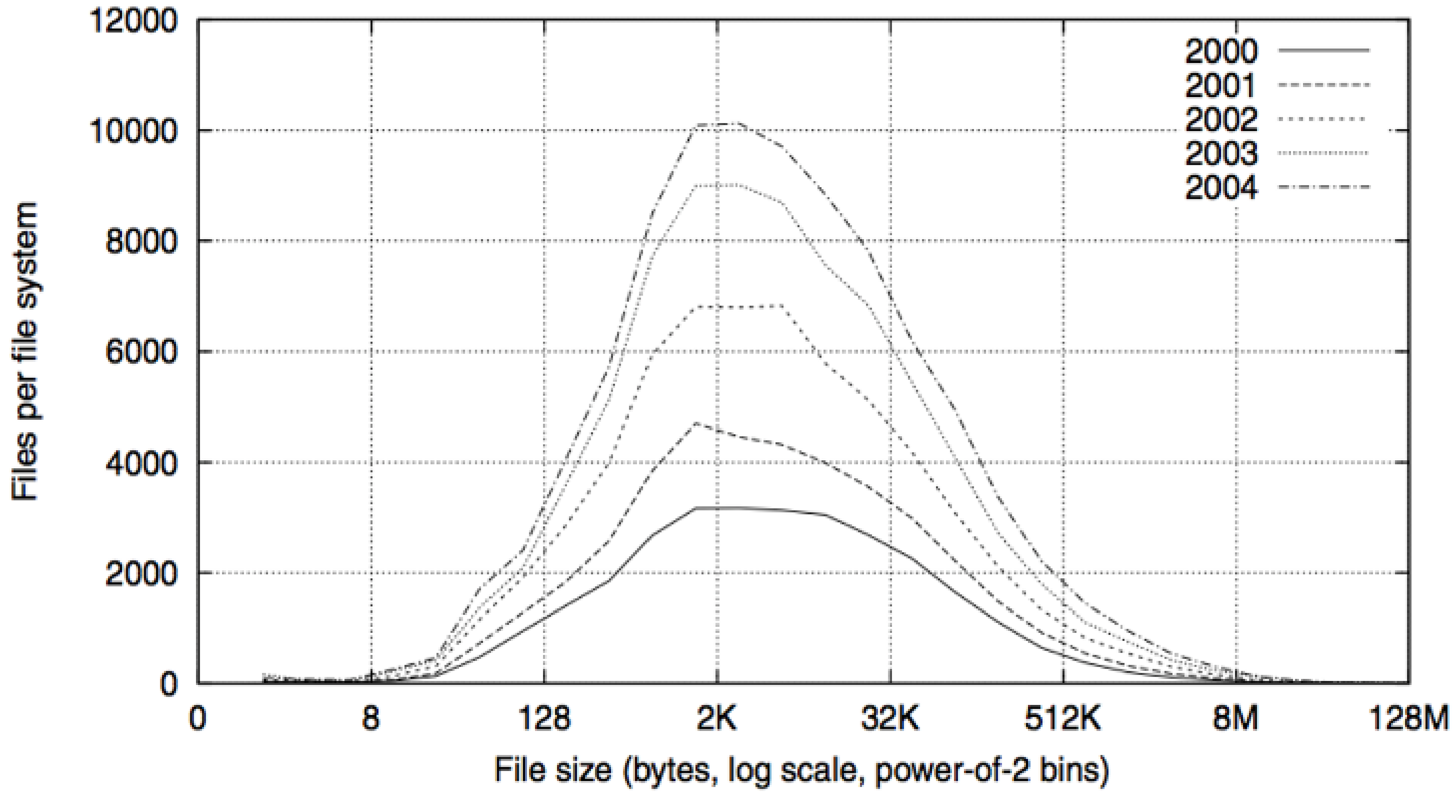


Fig. 2. Histograms of files by size.

Observation #2: Most Bytes are in Large Files

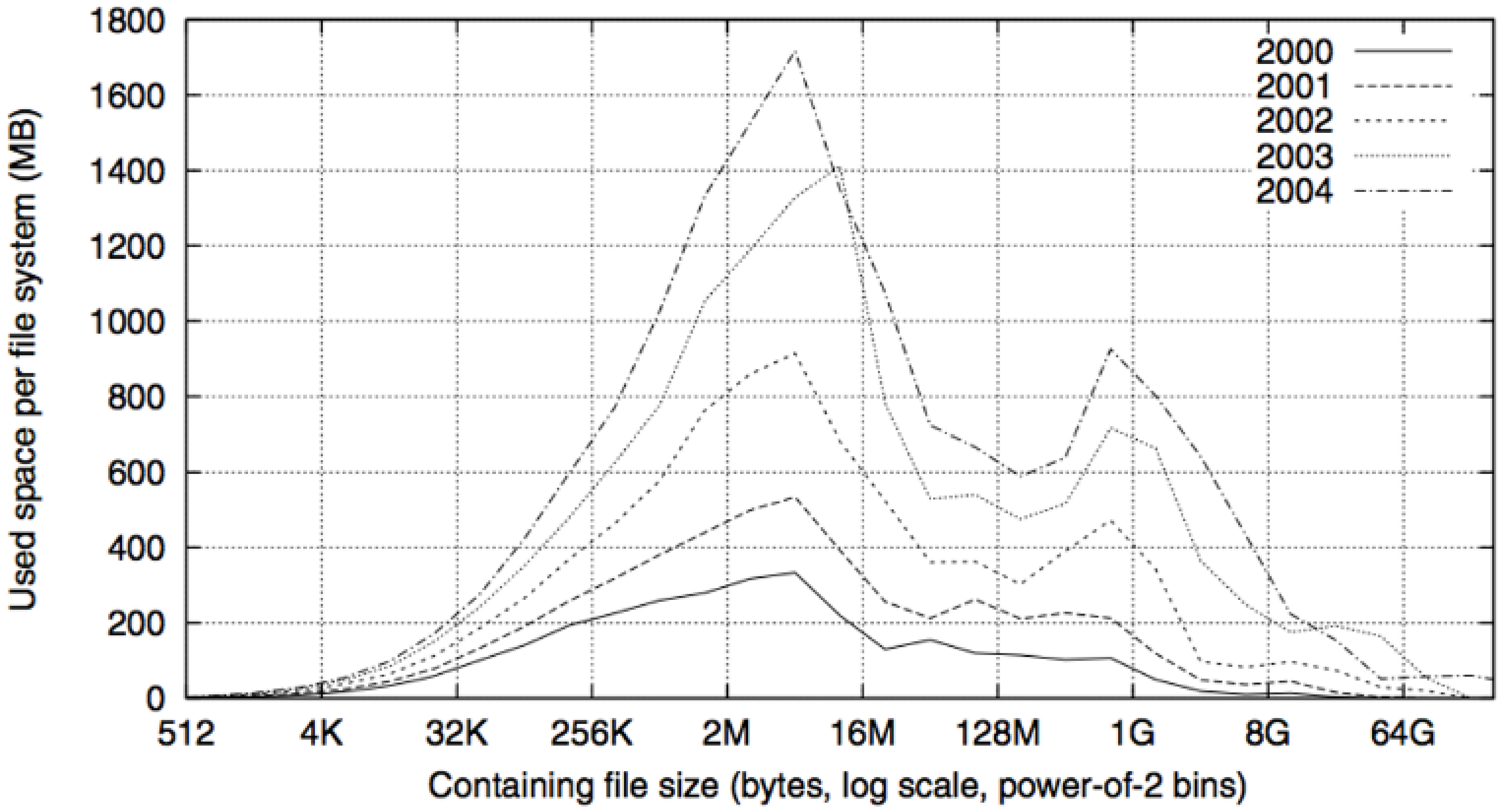


Fig. 4. Histograms of bytes by containing file size.

The key to it all: the Inode

File Number is index into set of inode arrays

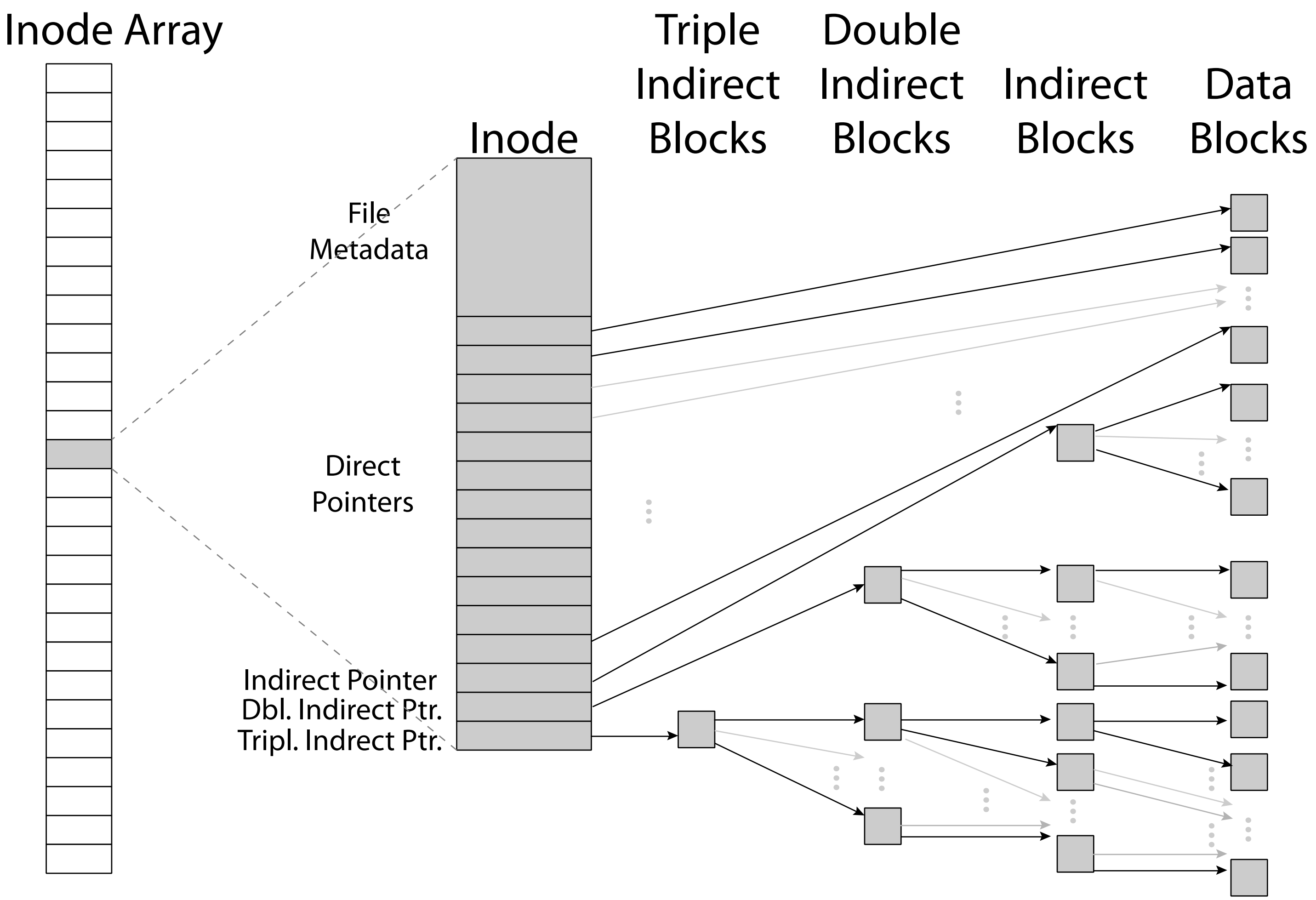
Index structure is an array of *inodes*

Each inode corresponds to a file and contains its metadata

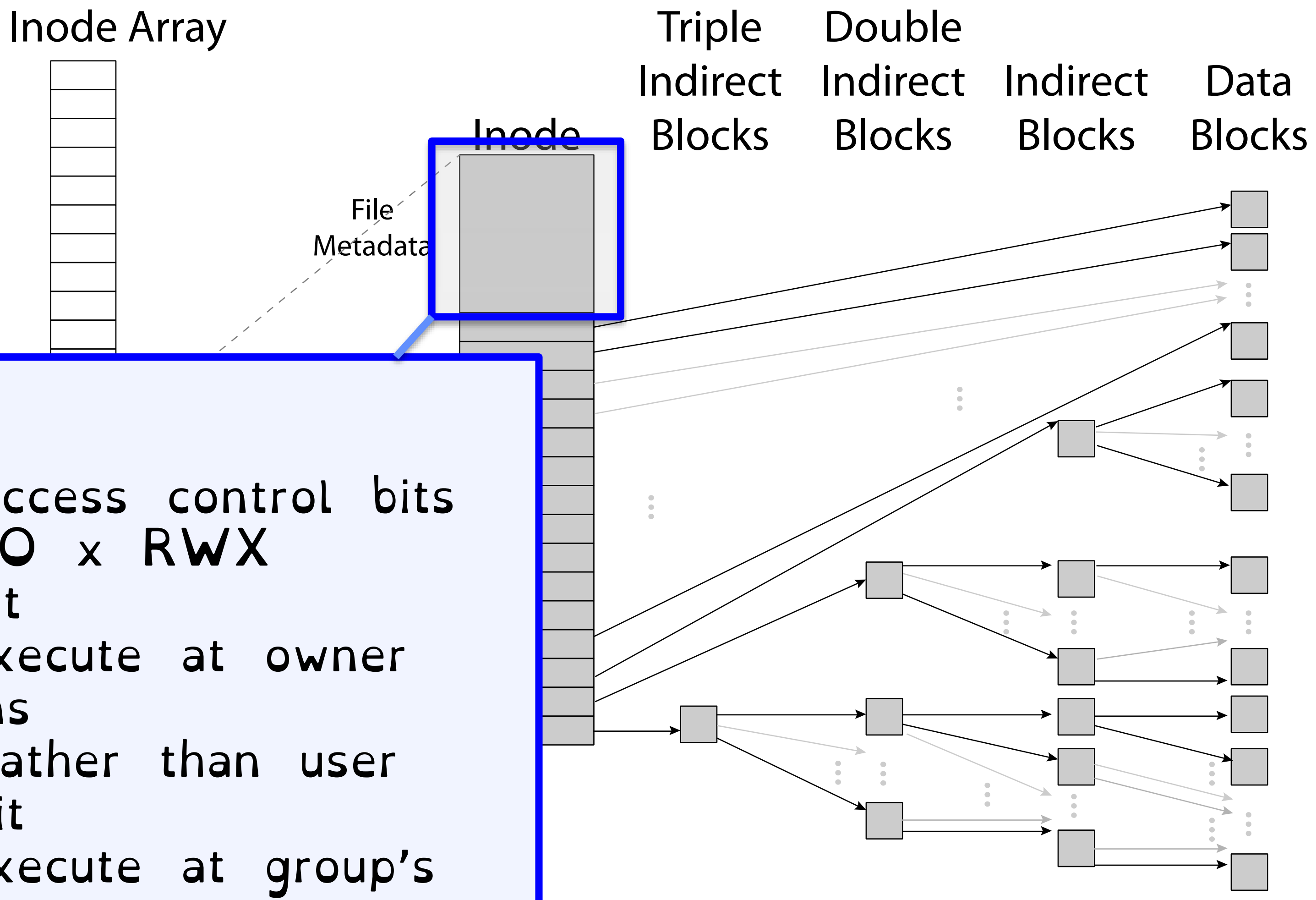
Inode maintains a multi-level tree structure to find storage blocks for files

Original *inode* format appeared in BSD 4.1
Berkeley Standard Distribution Unix!

Inode Structure



File Attributes



User
Group
9 basic access control bits
- UGO x RWX
SetUID bit
- execute at owner permissions rather than user
SetGID bit
- execute at group's permissions

Direct Pointers

Direct pointers

4kB blocks ⇒ sufficient for files up to 48KB

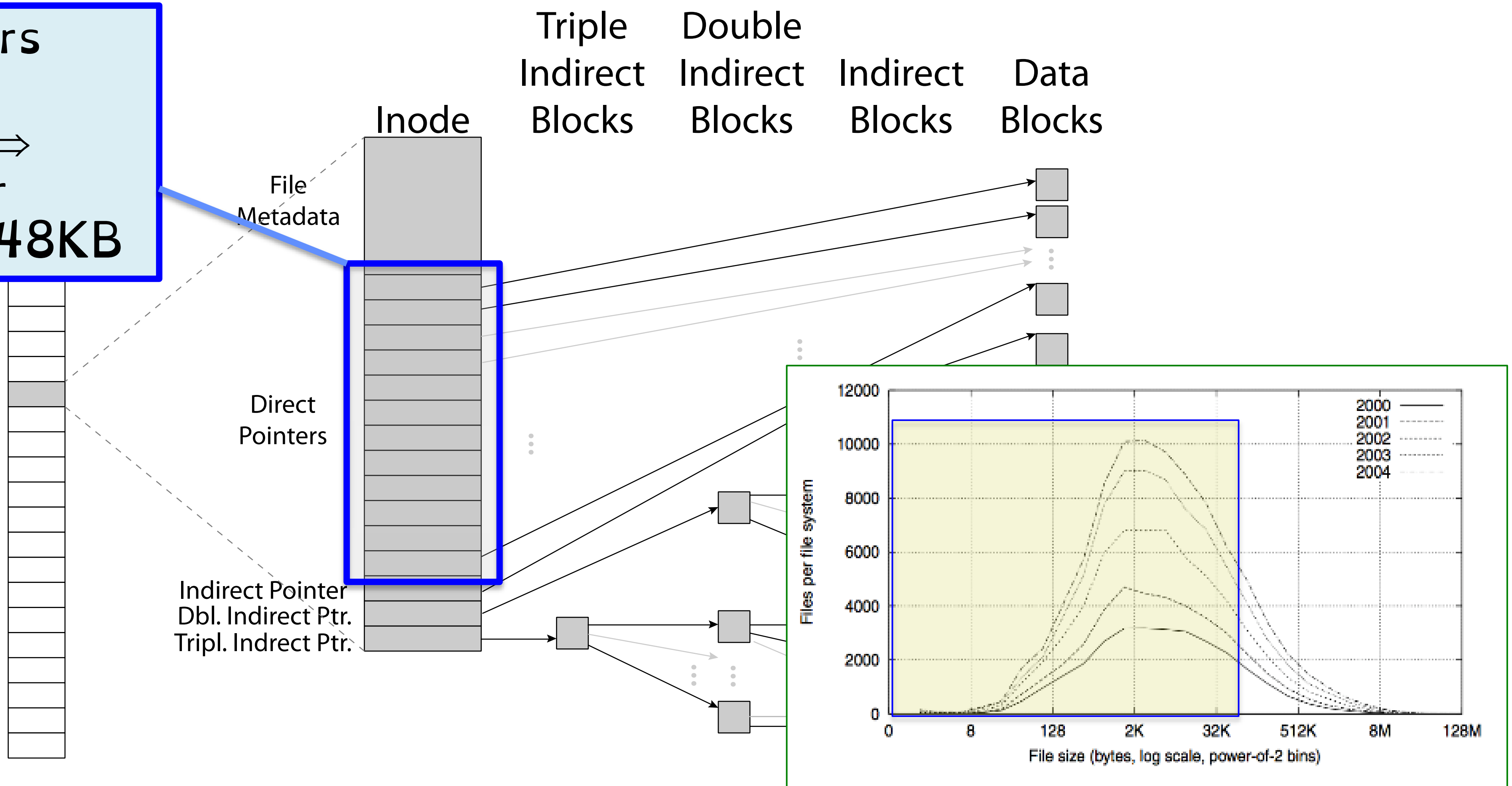
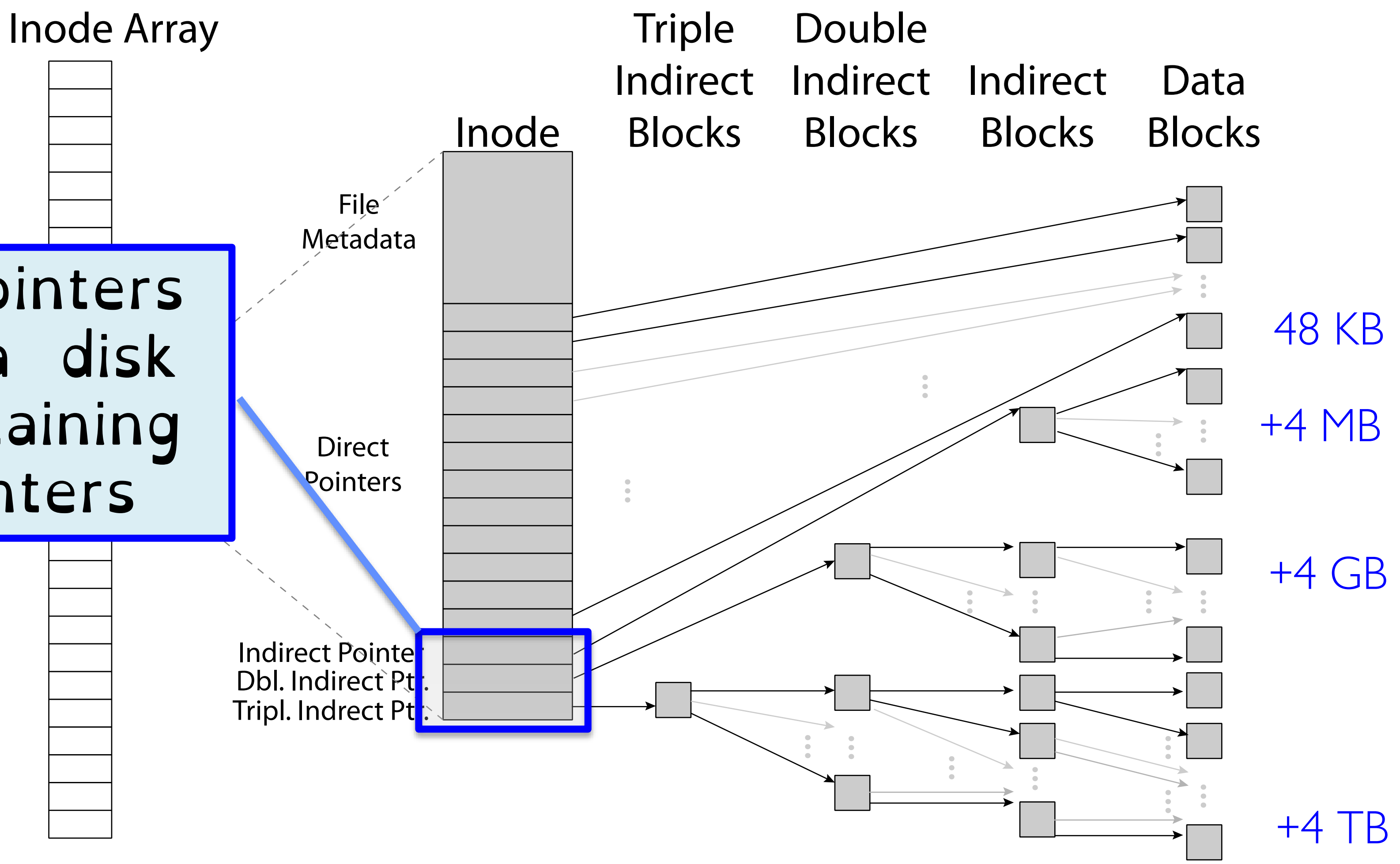


Fig. 2. Histograms of files by size.

Indirect Pointers

Indirect pointers point to a disk block containing only pointers



Indirect Pointers

Assume 4KB blocks

What is the maximum size of a file with only direct pointers?

12 * 4 KB = 48 KB

What is the maximum size of a file with one indirect pointer?

12 * 4 KB + 1024 * 4KB = 4.1MB

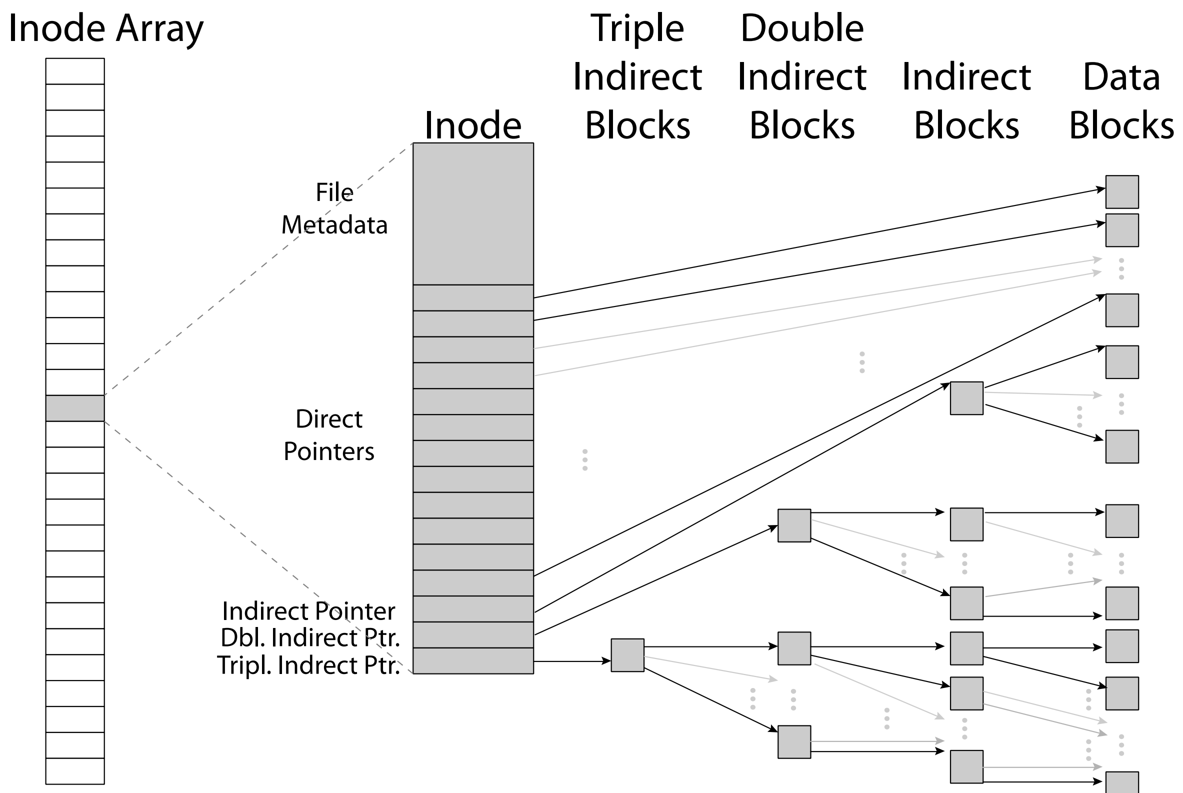
What is the maximum size of a file with double indirect pointers?

12 * 4KB + 1024 * 4KB + 1024 * 1024 * 4KB = 4.6 GB

Inodes form an on-disk index

Sample file in multilevel indexed format:

- 12 direct ptrs, 4K blocks
- How many accesses for block #23? (assume file header accessed on open)?
  - » Two: One for indirect block, one for data
- How about block #5?
  - » One: One for data
- Block #340?
  - » Three: double indirect block, indirect block, and data



Creating new files

Inodes are (logically) stored in an inode table

File system stores a bitmap of free inodes and free blocks

On creating a new file,

1) Check which inode is free/where that inode is stored

2) Check which data blocks are free

Putting it together

/cs162/natacha.txt (60KB)
/cs162/natasha.txt (4KB)

Each block is 4KB
Inode is 256 Bytes

Putting it together

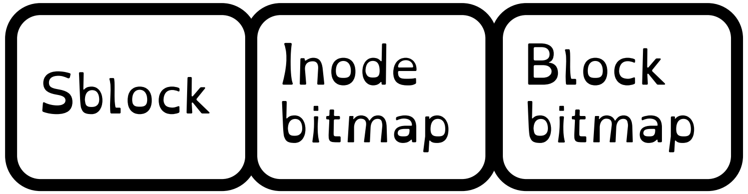
/cs162/natacha.txt (60KB)
/cs162/natasha.txt (4KB)

Sblock

Putting it together

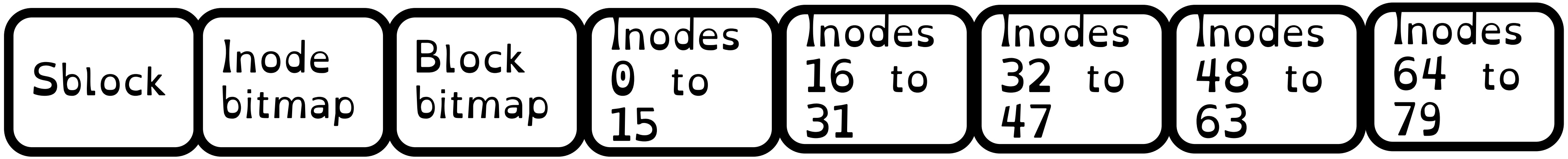
/cs162/natacha.txt (60KB)

/cs162/natasha.txt (4KB)



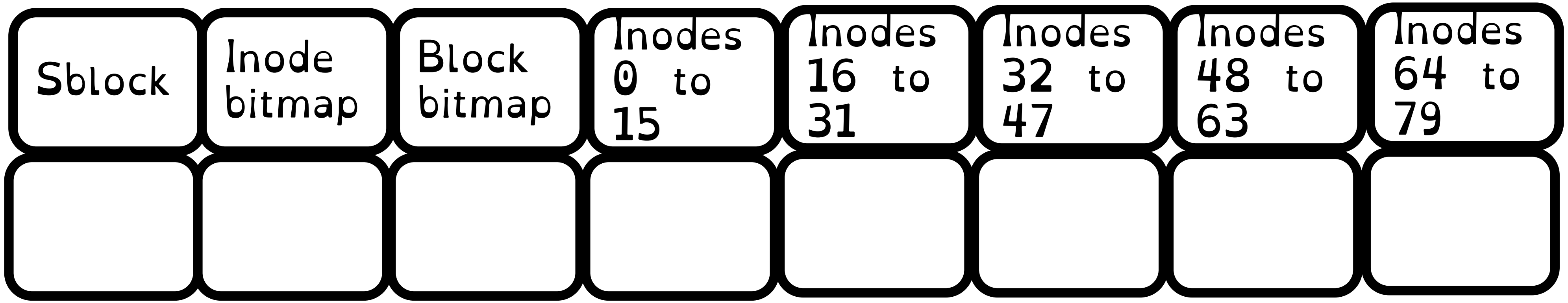
Putting it together

/cs162/natacha.txt (60KB)
/cs162/natasha.txt (4KB)



Putting it together

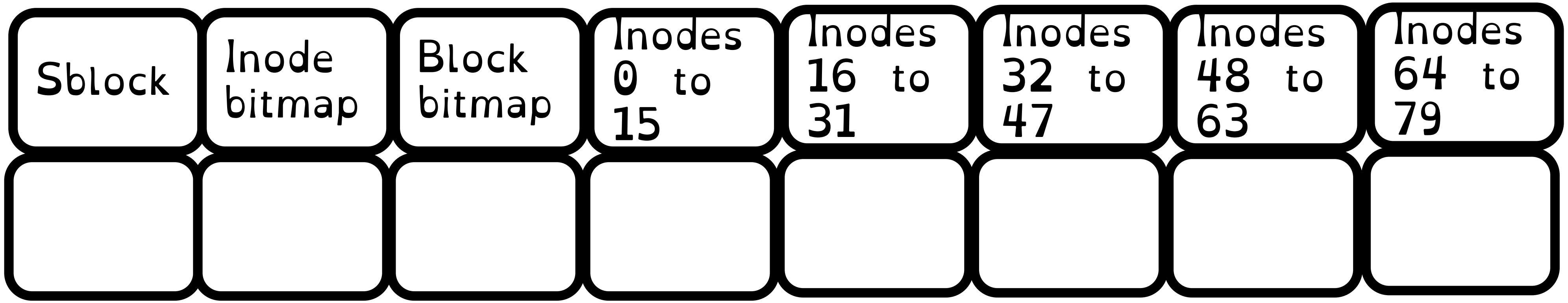
/cs162/natacha.txt (60KB)
/cs162/natasha.txt (4KB)



Putting it together

/

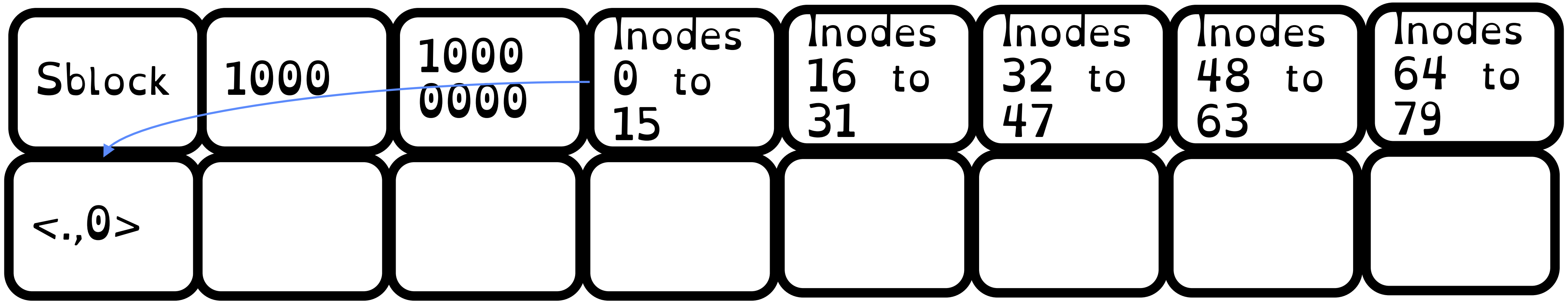
Allocate inode 0
Create data block



Putting it together

/

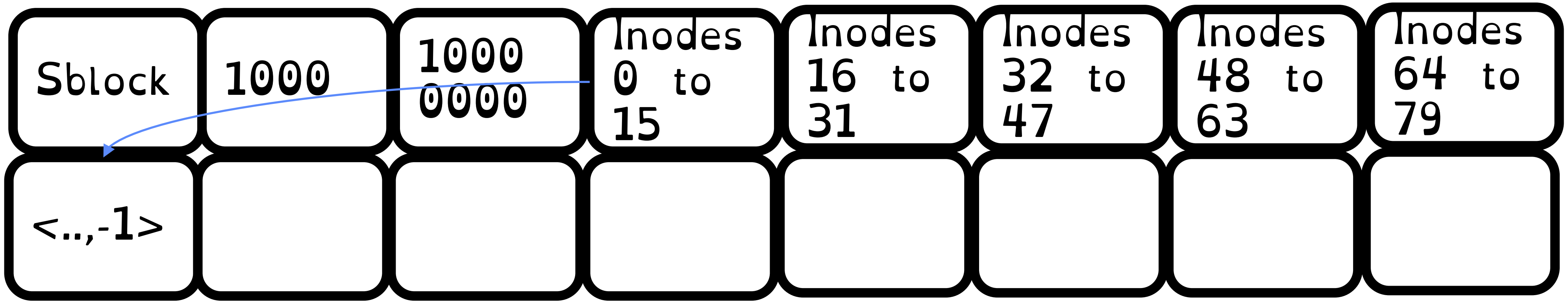
Allocate inode 0
Create data block



Putting it together

/cs162

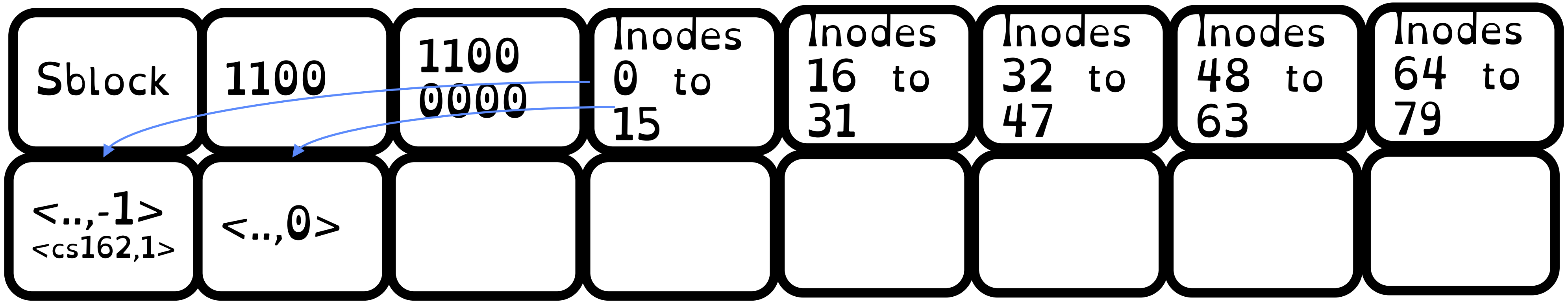
Allocate inode 1
Update direntry for /
Create data block



Putting it together

/cs162

Allocate inode 1
Update direntry for /
Create data block



Putting it together

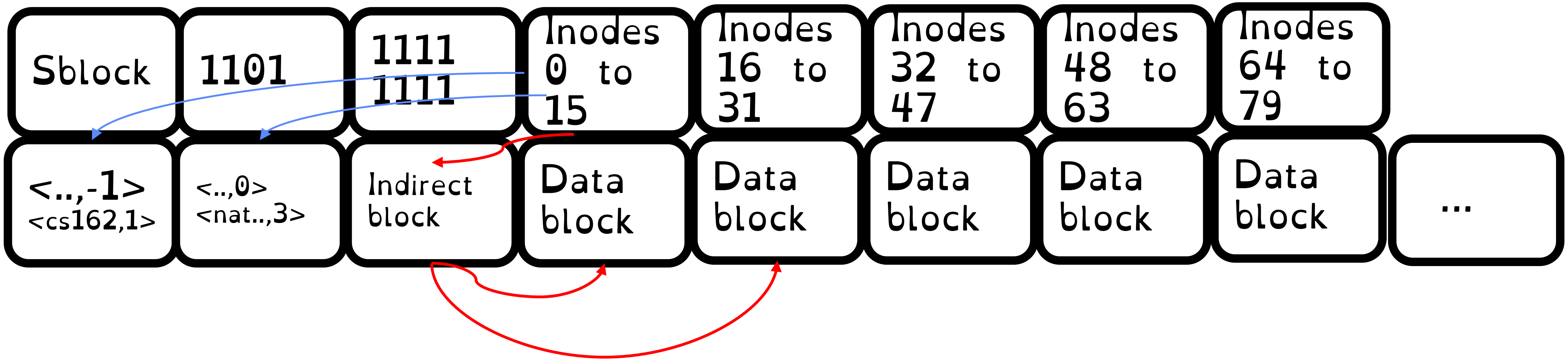
/cs162/natacha.txt (60KB)

Allocate inode 3

Update direntry

Create indirect block

Create datablocks



Unix File System (Berkeley FFS)

Introducing Disk Awareness

A Fast File System for UNIX*

Marshall Kirk McKusick, William N. Joy†,
Samuel J. Leffler‡, Robert S. Fabry

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720

ABSTRACT

A reimplementation of the UNIX file system is described. The reimplementation provides substantially higher throughput rates by using more flexible allocation policies that allow better locality of reference and can be adapted to a wide range of peripheral and processor characteristics. The new file system clusters data that is sequentially accessed and provides two block sizes to allow fast access to large files while not wasting large amounts of space for small files. File access rates of up to ten times faster than the traditional UNIX file system are experienced. Long needed enhancements to the pro-

Recall: Critical Factors in File System Design

(Hard) Disk Performance !!!
Maximize sequential access, minimize seeks

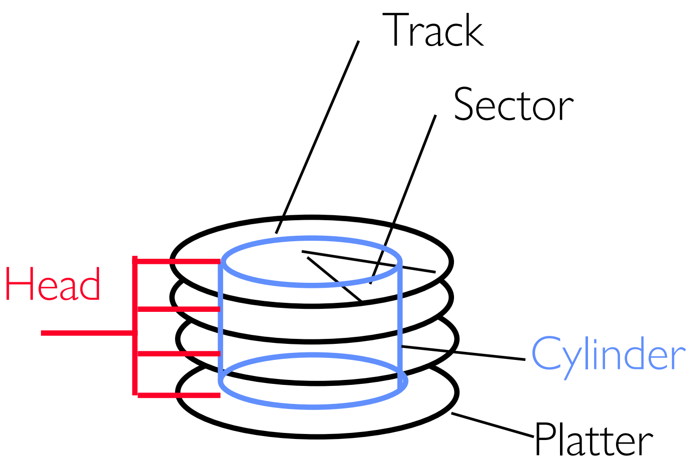
Open before Read/Write
- Can perform protection checks and look up where the actual file resource are, in advance

Size is determined as they are used !!!
- Can write (or read zeros) to expand the file
- Start small and grow, need to make room

Organized into directories
- What data structure (on disk) for that?

Need to carefully allocate / free blocks
- Such that access remains efficient

Recall: Magnetic Disks



Cylinders: all the tracks under the head at a given point on all surfaces

Read/write data is a three-stage process:

- Seek time: position the head/arm over the proper track
- Rotational latency: wait for desired sector to rotate under r/w head
- Transfer time: transfer a block of bits (sector) under r/w head

Fast File System (BSD 4.2, 1984)

Same inode structure as in BSD 4.1
- same file header and triply indirect blocks like we just studied
- Some changes to block sizes from 1024⇒4096 bytes for performance

Optimization for Performance and Reliability:
- Distribute inodes among different tracks to be closer to data
- Uses bitmap allocation in place of freelist
- Attempt to allocate files contiguously
- 10% reserved disk space
- Skip-sector positioning (mentioned later)

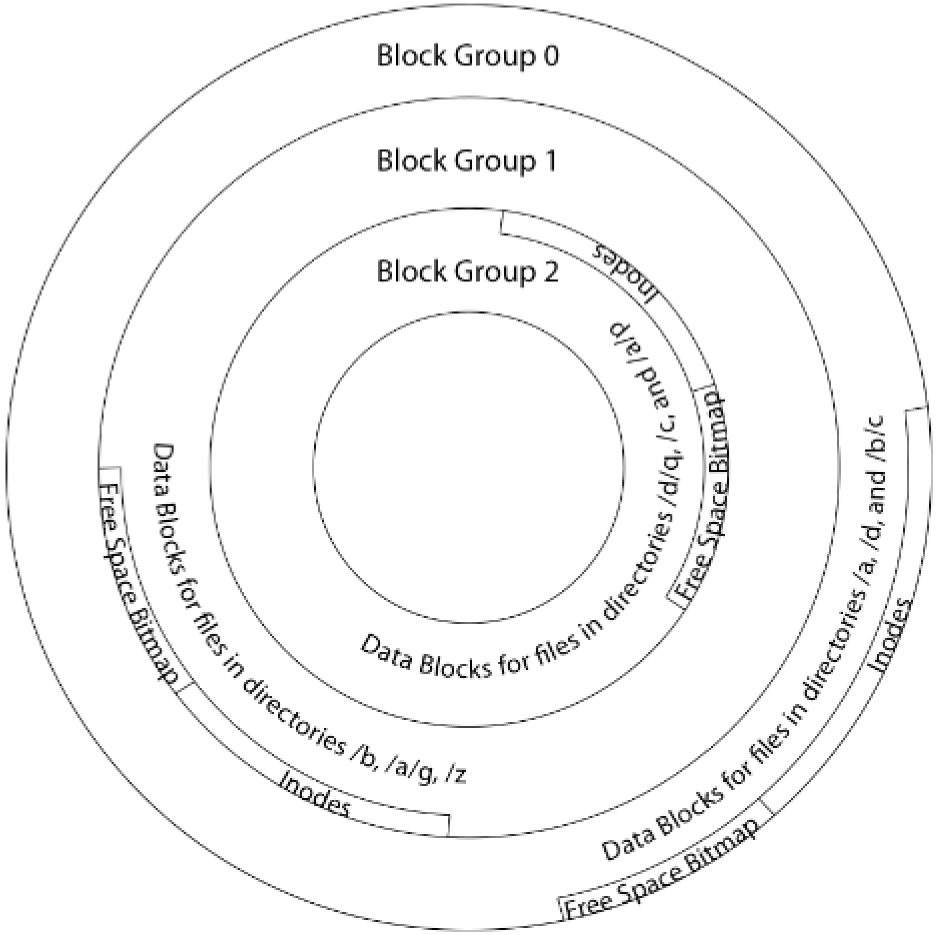
FFS Locality: Block Groups

Distribute header information (inodes) closer to the data blocks, in same "cylinder group"

File system volume divided into set of block groups

Data blocks, metadata, and free space
interleaved within block group

Put directory and its files in common block group



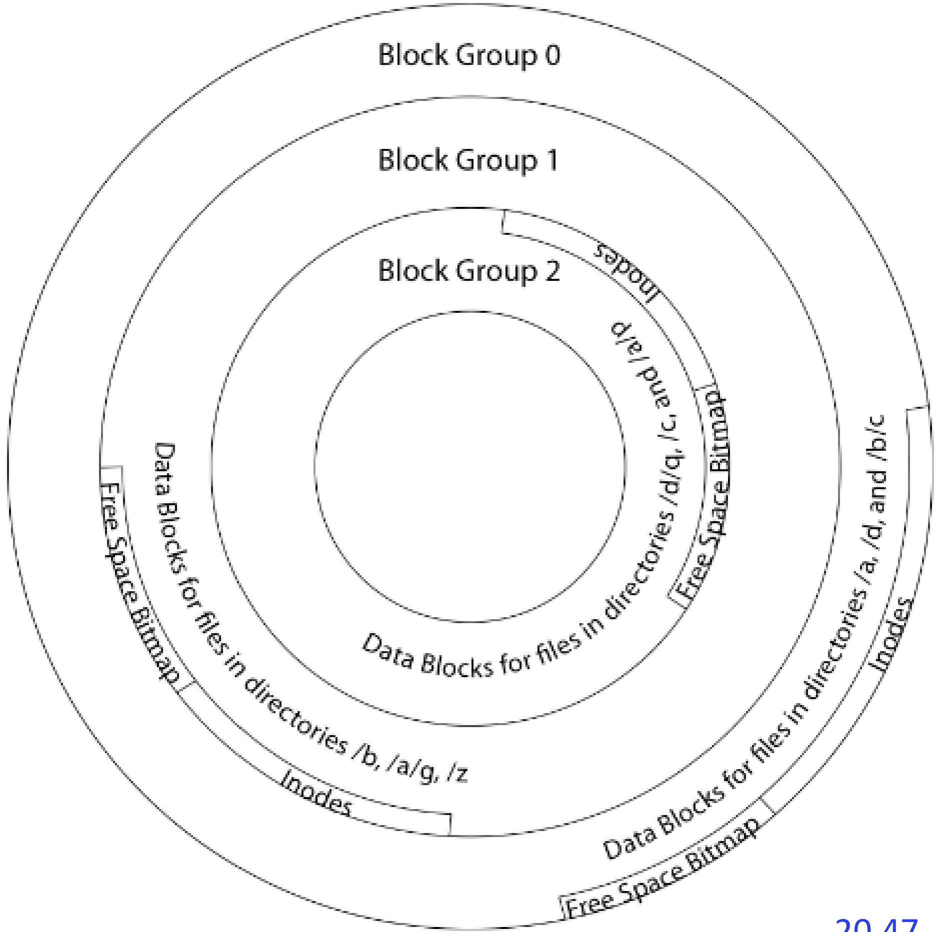
FFS Locality: Block Groups

First-Free allocation of new file blocks

- To expand file, first try successive blocks in bitmap, then choose new range of blocks
- Few little holes at start, big sequential runs at end of group
- Avoids fragmentation
- Sequential layout for big files

Important: keep **10%** or more free!

- Reserve space in the Block Group



Attack of the Rotational Delay

Missing blocks due to rotational delay

Issue: Read one block, do processing, and read next block. In meantime, disk has continued turning: missed next block! Need 1 revolution/block!

Attack of the Rotational Delay

Solution 1: Skip sector positioning ("interleaving")

- » Place the blocks from one file on every other block of a track: give time for processing to overlap rotation
- » Can be done by OS or in modern drives by the disk controller

Solution 2: Read ahead: read next block right after first, even if application hasn't asked for it yet

- » This can be done either by OS (read ahead)
- » By disk itself (track buffers) - many disk controllers have internal RAM that allows them to read a complete track

UNIX 4.2 BSD FFS

Pros
- Efficient storage for both small and large files
- Locality for both small and large files
- Locality for metadata and data
- No defragmentation necessary!

Cons
- Inefficient for tiny files (a 1 byte file requires both an inode and a data block)
- Inefficient encoding when file is mostly contiguous on disk
- Need to reserve 10-20% of free space to prevent fragmentation

What about other file systems?

FAT:
File Allocation Table

(MS-DOS,1977)

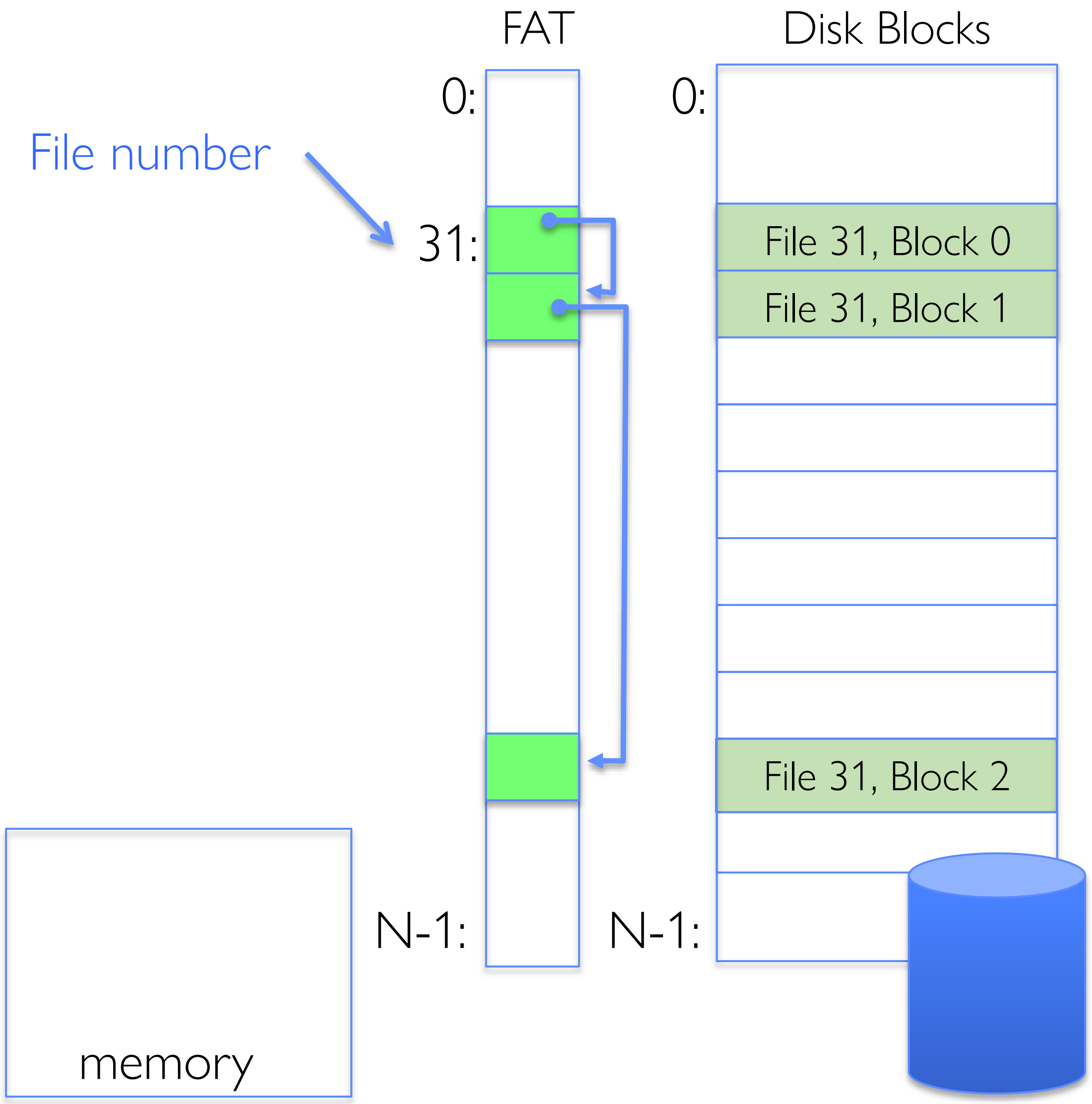
Windows NTFS

FAT (File Allocation Table)

Assume (for now) we have a way to translate a path to a "file number"
- i.e., a directory structure

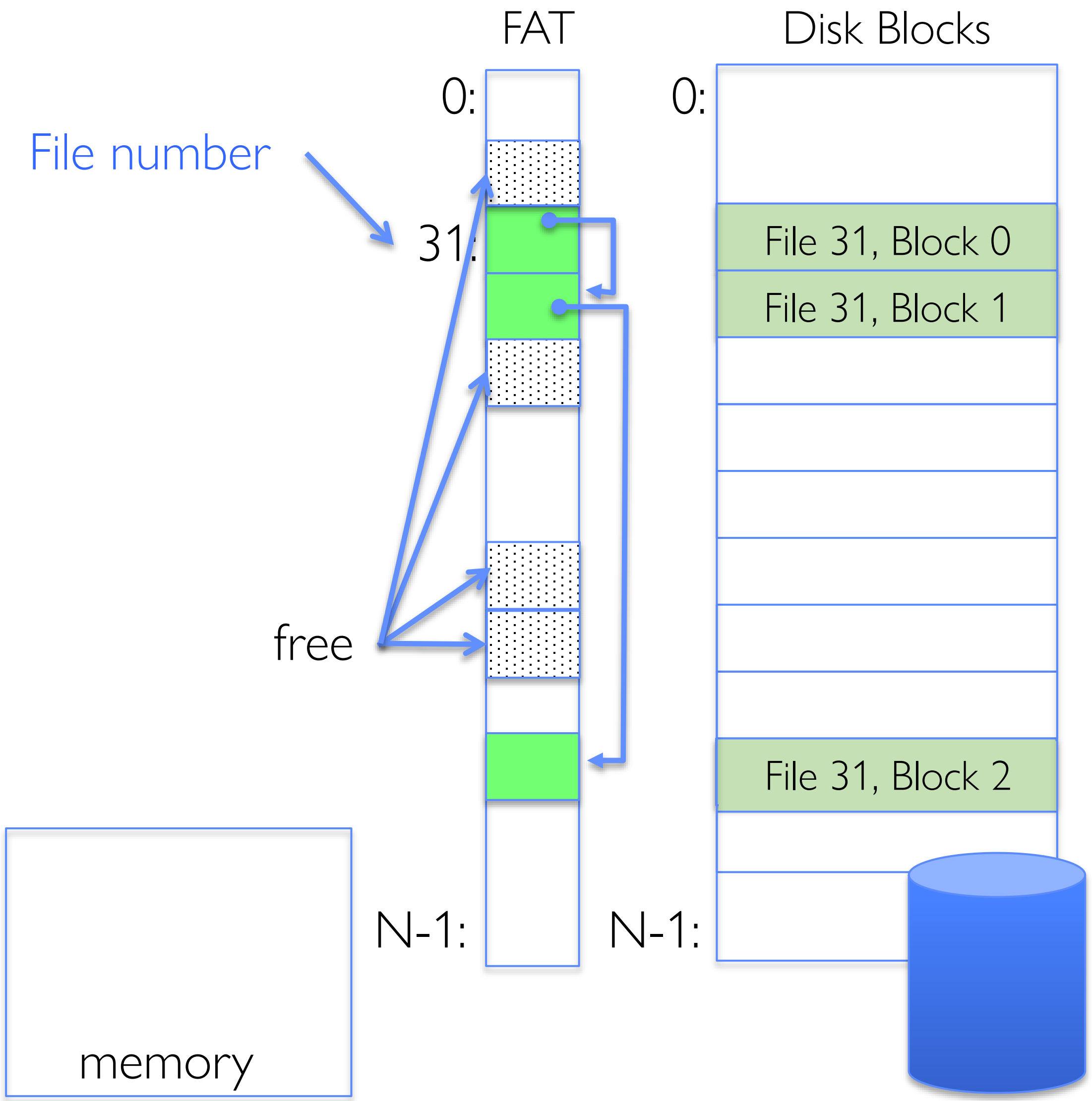
Disk Storage is a collection of Blocks
- Just hold file data
  (offset o = < B, x >)

Example: file_read **31**, < **2**, x >
- Index into FAT with file number
- Follow linked list to block
- Read the block from disk into memory



FAT (File Allocation Table)

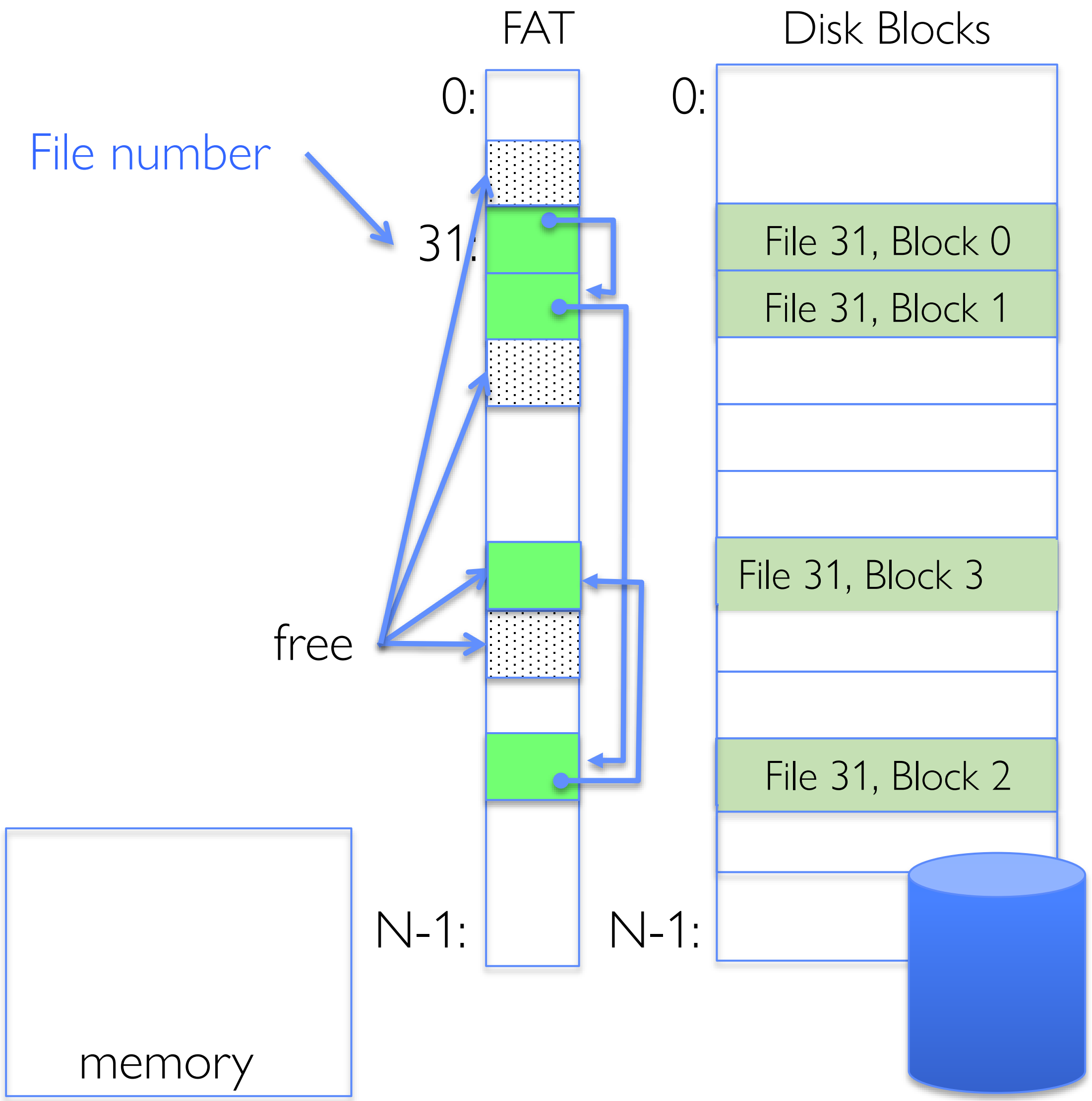
- File is a collection of disk blocks
- FAT is linked list 1-1 with blocks
- File number is index of root of block list for the file
- File offset: block number and offset within block
- Follow list to get block number
- Unused blocks marked free
  - Could require scan to find
  - Or, could use a free list



FAT (File Allocation Table)

file_write(31, < 3, y >)

- Grab free block
- Linking them into file



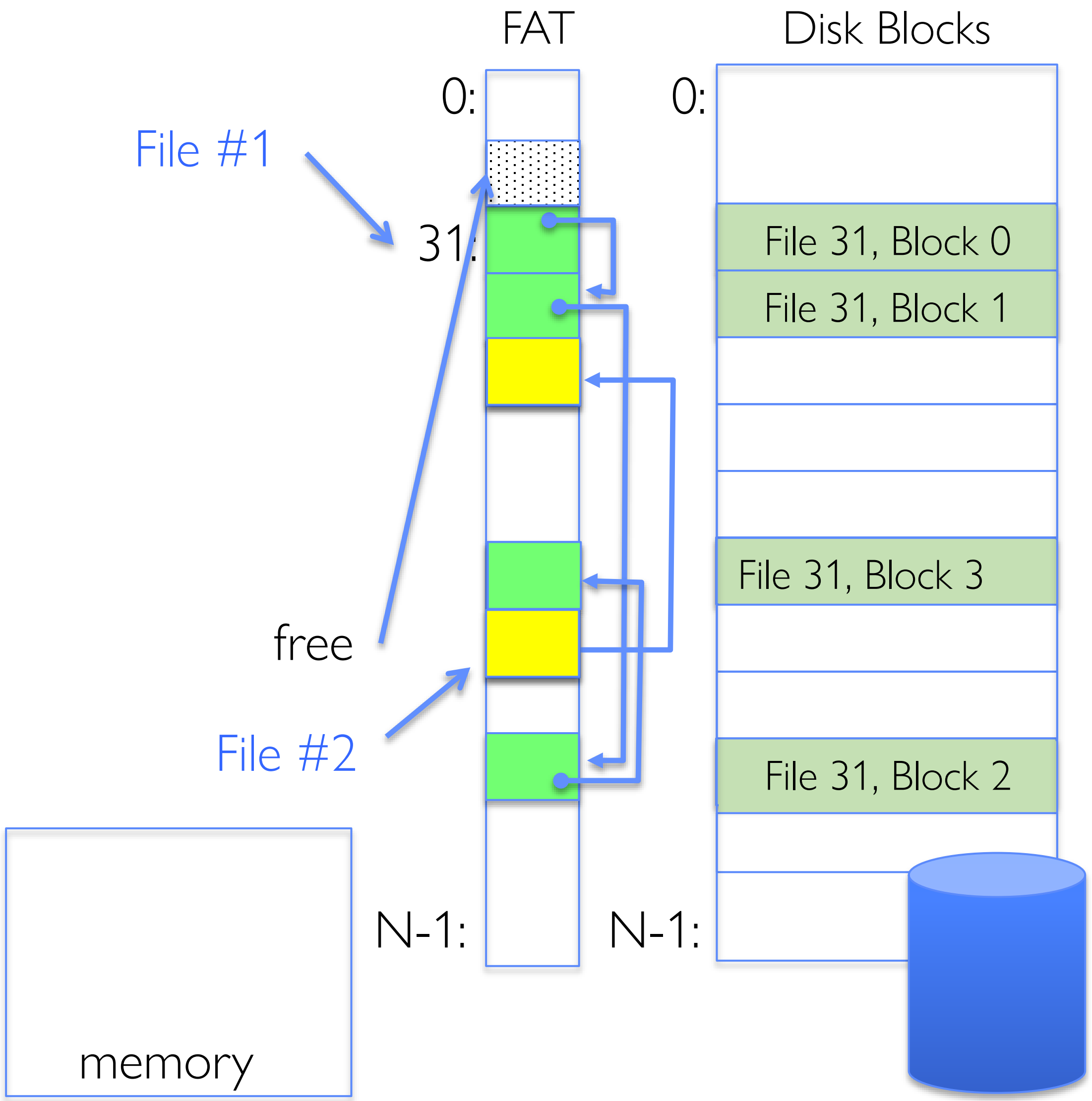
FAT (File Allocation Table)

Where is FAT stored?
- On disk

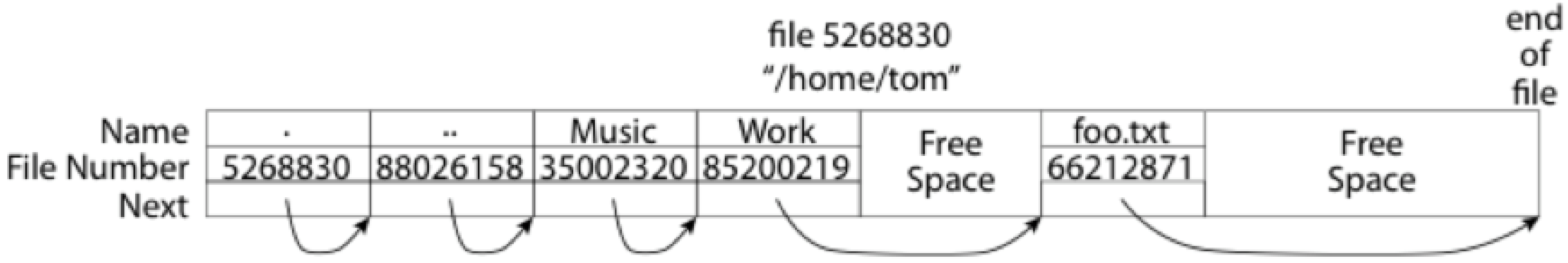
How to format a disk?
- Zero the blocks, mark FAT entries "free"

How to quick format a disk?
- Mark FAT entries "free"

Simple: can implement in device firmware



FAT: Directories



- A directory is a file containing <file_name: file_number> mappings
- In FAT: file attributes are kept in directory (!!!)
  - Not directly associated with the file itself
- Each directory a linked list of entries
  - Requires linear search of directory to find particular entry
- Where do you find root directory ("/")?
  - At well-defined place on disk
  - For FAT, this is at block 2 (there are no blocks 0 or 1)

FAT Discussion

Suppose you start with the file number:

- Time to find block?
- Block layout for file?
- Sequential access?
- Random access?
- Fragmentation?
- Small files?
- Big files?