CS162
Operating Systems and Systems Programming
Lecture 19

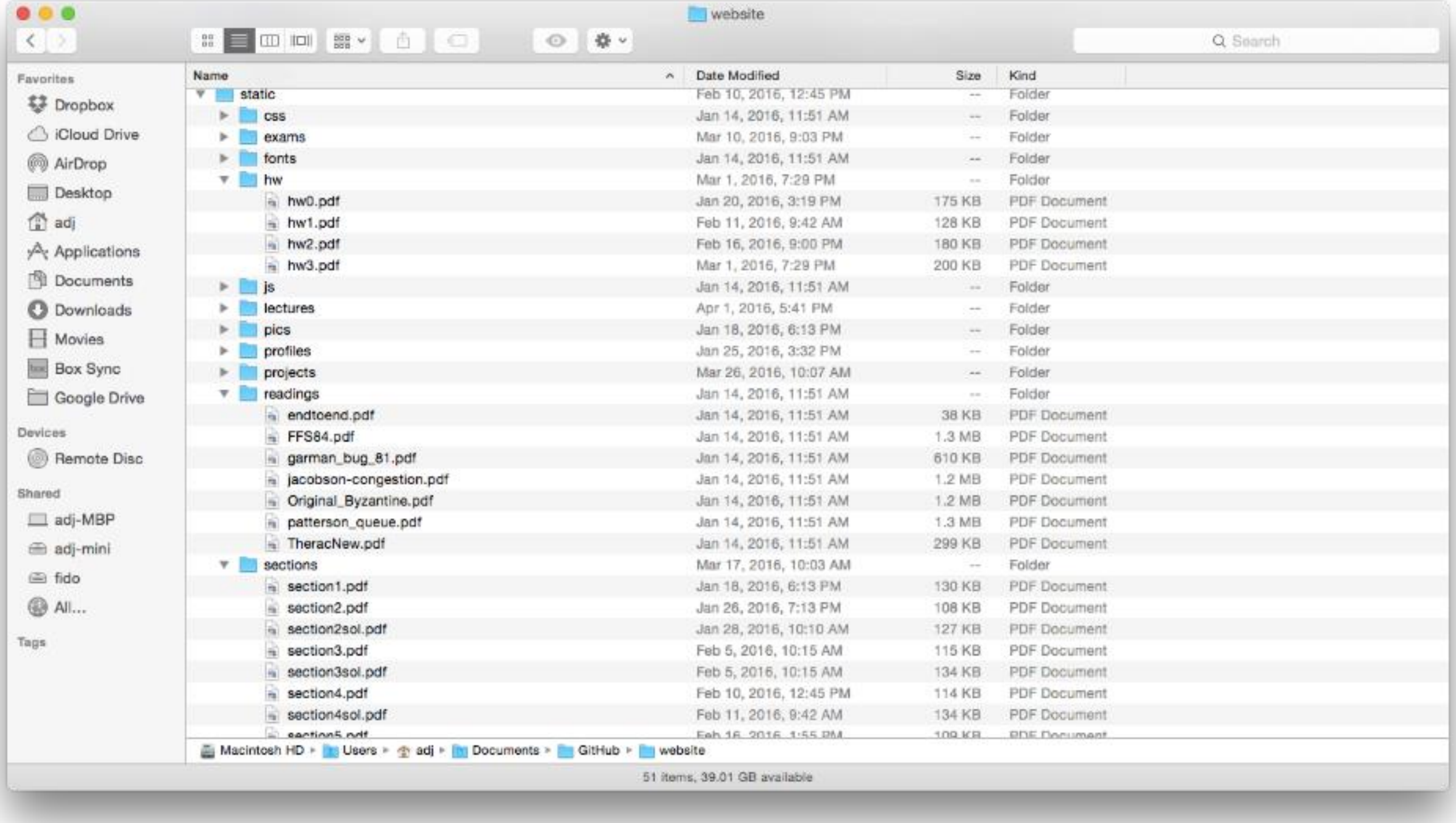
File Systems

Professor Natacha Crooks & Matei Zaharia

https://cs162.org/

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

Files & Directories

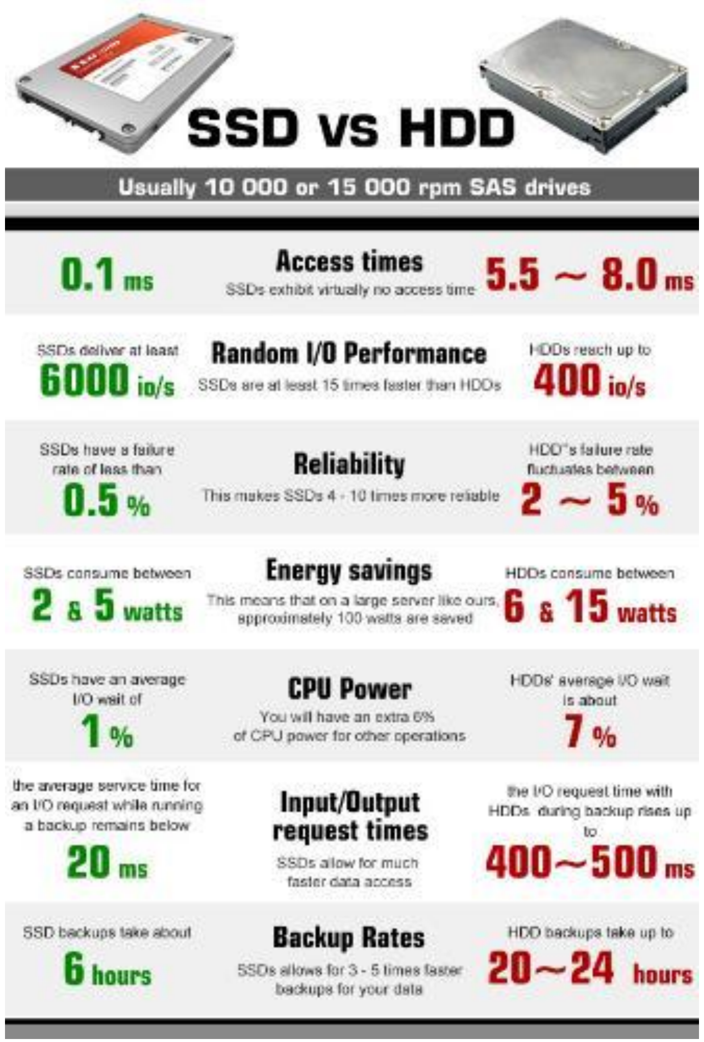


Name	Date Modified	Size	Kind
▼ static	Feb 10, 2016, 12:45 PM	--	Folder
▶ css	Jan 14, 2016, 11:51 AM	--	Folder
▶ exams	Mar 10, 2016, 9:03 PM	--	Folder
▶ fonts	Jan 14, 2016, 11:51 AM	--	Folder
▼ hw	Mar 1, 2016, 7:29 PM	--	Folder
hw0.pdf	Jan 20, 2016, 3:19 PM	175 KB	PDF Document
hw1.pdf	Feb 11, 2016, 9:42 AM	128 KB	PDF Document
hw2.pdf	Feb 16, 2016, 9:00 PM	180 KB	PDF Document
hw3.pdf	Mar 1, 2016, 7:29 PM	200 KB	PDF Document
▶ js	Jan 14, 2016, 11:51 AM	--	Folder
▶ lectures	Apr 1, 2016, 5:41 PM	--	Folder
▶ pics	Jan 18, 2016, 6:13 PM	--	Folder
▶ profiles	Jan 25, 2016, 3:32 PM	--	Folder
▶ projects	Mar 26, 2016, 10:07 AM	--	Folder
▼ readings	Jan 14, 2016, 11:51 AM	--	Folder
endtoend.pdf	Jan 14, 2016, 11:51 AM	38 KB	PDF Document
FFS84.pdf	Jan 14, 2016, 11:51 AM	1.3 MB	PDF Document
garman_bug_81.pdf	Jan 14, 2016, 11:51 AM	610 KB	PDF Document
jacobson-congestion.pdf	Jan 14, 2016, 11:51 AM	1.2 MB	PDF Document
Original_Byzantine.pdf	Jan 14, 2016, 11:51 AM	1.2 MB	PDF Document
patterson_queue.pdf	Jan 14, 2016, 11:51 AM	1.3 MB	PDF Document
TheracNew.pdf	Jan 14, 2016, 11:51 AM	299 KB	PDF Document
▼ sections	Mar 17, 2016, 10:03 AM	--	Folder
section1.pdf	Jan 18, 2016, 6:13 PM	130 KB	PDF Document
section2.pdf	Jan 26, 2016, 7:13 PM	108 KB	PDF Document
section2sol.pdf	Jan 28, 2016, 10:10 AM	127 KB	PDF Document
section3.pdf	Feb 5, 2016, 10:15 AM	115 KB	PDF Document
section3sol.pdf	Feb 5, 2016, 10:15 AM	134 KB	PDF Document
section4.pdf	Feb 10, 2016, 12:45 PM	114 KB	PDF Document
section4sol.pdf	Feb 11, 2016, 9:42 AM	134 KB	PDF Document
section5.pdf	Feb 16, 2016, 1:55 PM	109 KB	PDF Document

Macintosh HD ▸ Users ▸ adj ▸ Documents ▸ GitHub ▸ website

51 items, 39.01 GB available

Recall: HDDs and SSDs



SSD vs HDD

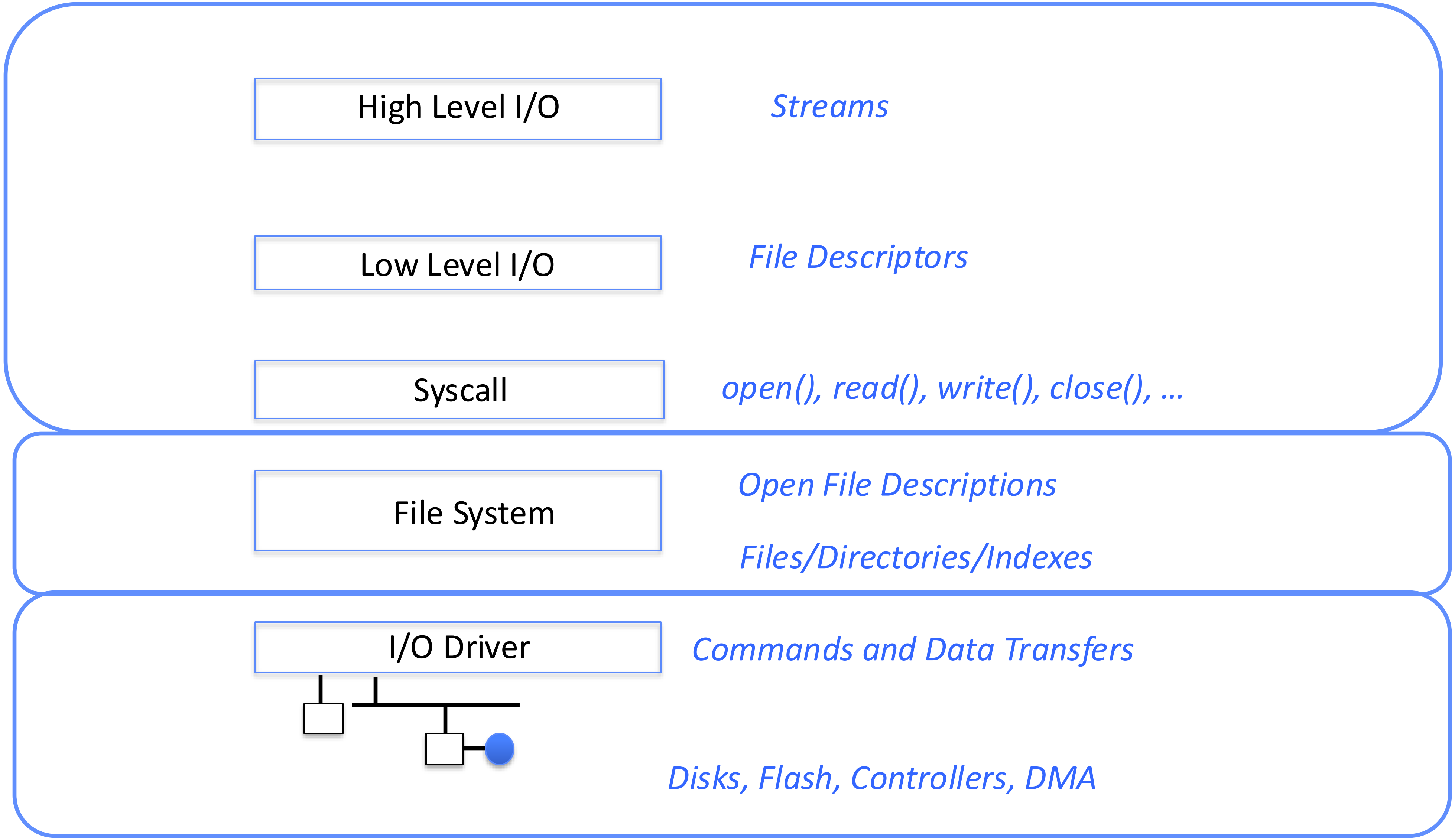
Usually 10 000 or 15 000 rpm SAS drives

0.1 ms	**Access times** SSDs exhibit virtually no access time	5.5 ~ 8.0 ms
SSDs deliver at least 6000 io/s	**Random I/O Performance** SSDs are at least 15 times faster than HDDs	HDDs reach up to 400 io/s
SSDs have a failure rate of less than 0.5 %	**Reliability** This makes SSDs 4 - 10 times more reliable	HDD"s failure rate fluctuates between 2 ~ 5 %
SSDs consume between 2 & 5 watts	**Energy savings** This means that on a large server like ours, approximately 100 watts are saved	HDDs consume between 6 & 15 watts
SSDs have an average I/O wait of 1 %	**CPU Power** You will have an extra 6% of CPU power for other operations	HDDs' average I/O wait is about 7 %
the average service time for an I/O request while running a backup remains below 20 ms	**Input/Output request times** SSDs allow for much faster data access	the I/O request time with HDDs during backup rises up to 400~500 ms
SSD backups take about 6 hours	**Backup Rates** SSDs allows for 3 - 5 times faster backups for your data	HDD backups take up to 20~24 hours

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

Both work better with larger reads & writes

Recall: I/O and Storage Layers



High Level I/O

Streams

Low Level I/O

File Descriptors

Syscall

open(), read(), write(), close(), …

File System

Open File Descriptions

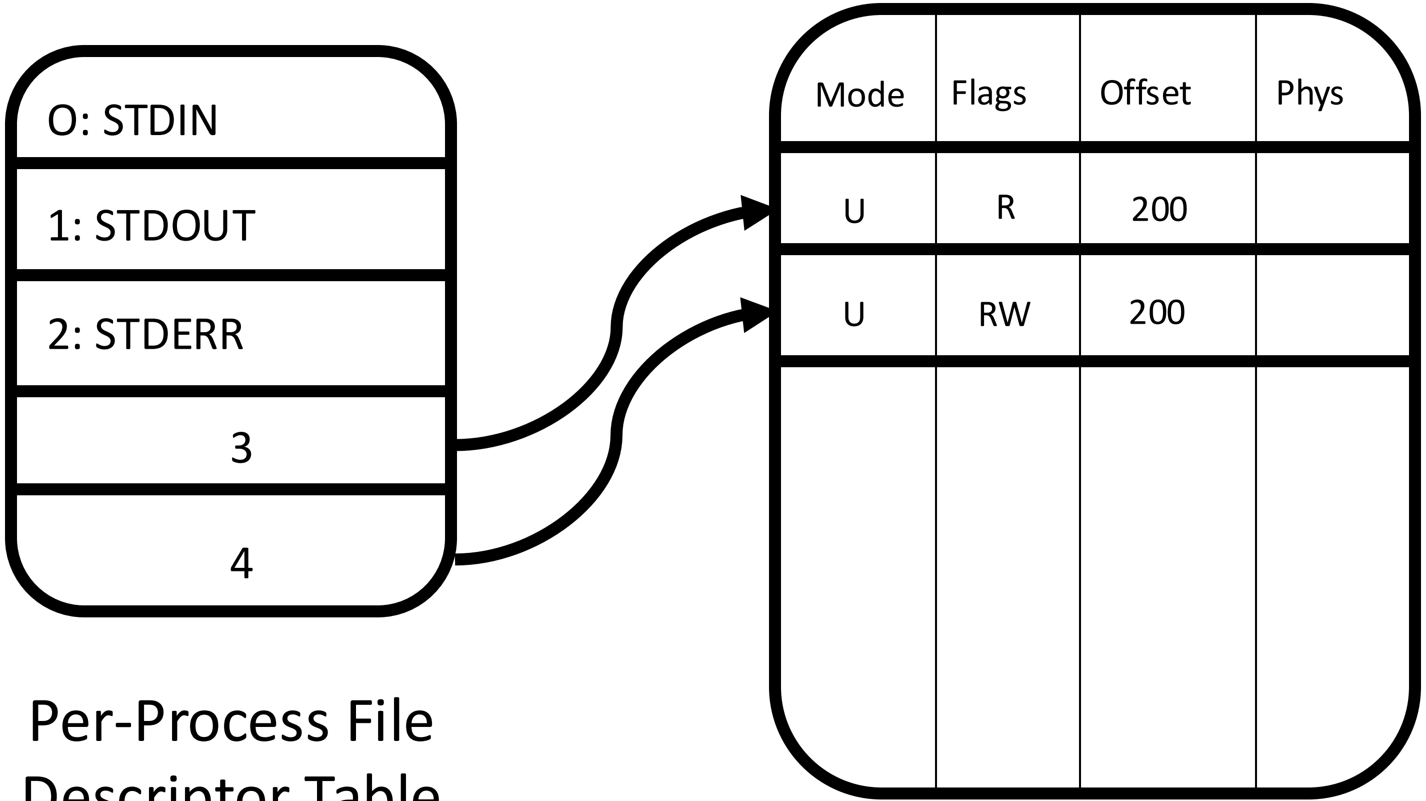
Files/Directories/Indexes

I/O Driver

Commands and Data Transfers

Disks, Flash, Controllers, DMA

Recall: FD & File Descriptors



O: STDIN

1: STDOUT

2: STDERR

3

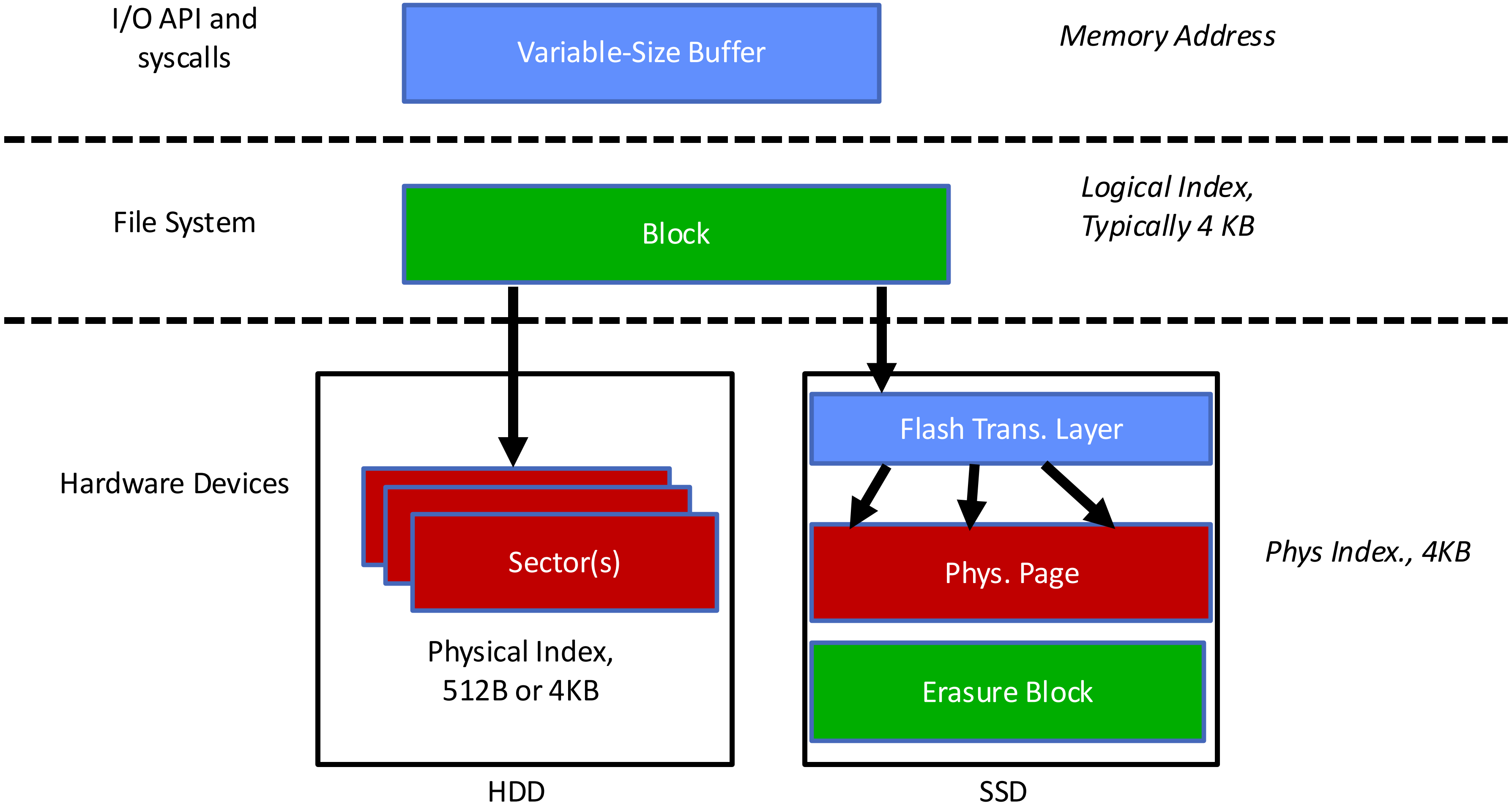
4

Mode	Flags	Offset	Phys
U	R	200	
U	RW	200	

Per-Process File Descriptor Table

Global Open File Description Table

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.

Purpose of 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

Organization: Organize file names within directories

Placement: 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)

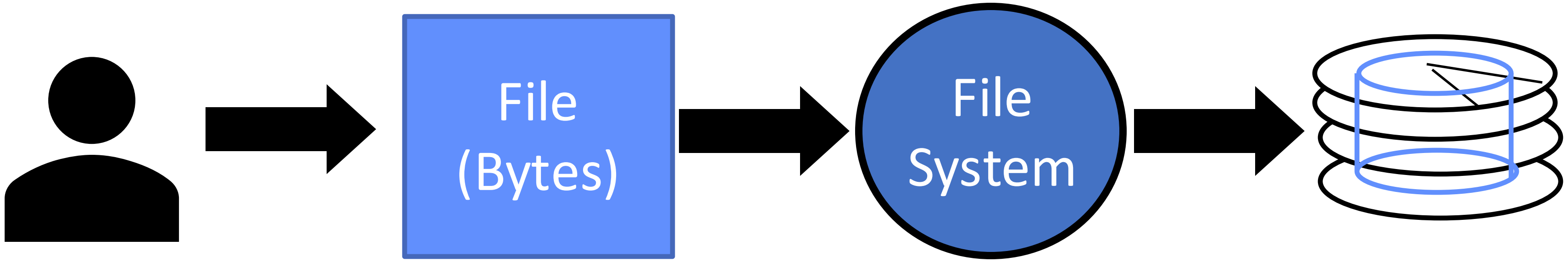
Doesn’t matter to system what kind of data structures you want to store on disk!

System’s view (inside OS):

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

Block size $\geq$ 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

What Does the File System Need to Do?

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

Critical Factors in File System Design

(Hard) Disk Performance !!!

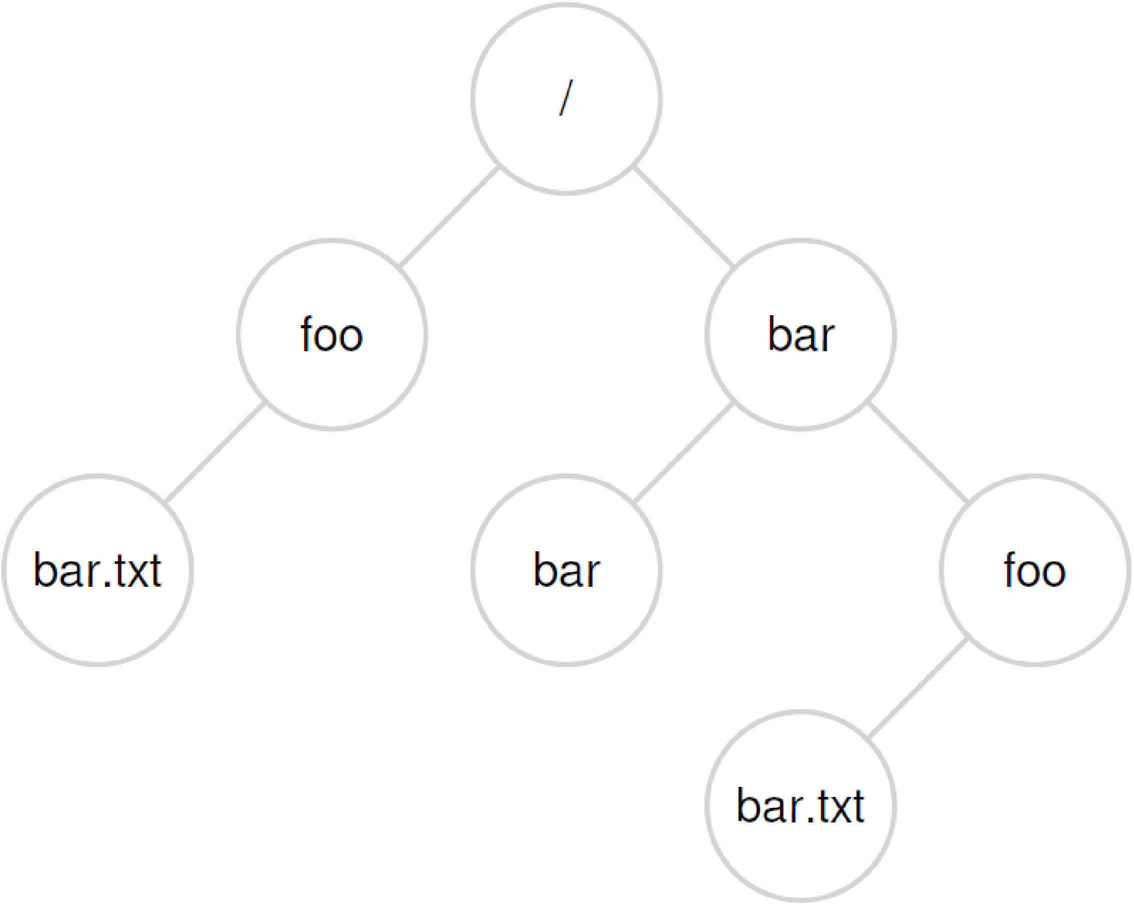
Open before read/write

Size is determined as files are used !!!

Organized into directories

Need to carefully allocate / free blocks

Files & Directories



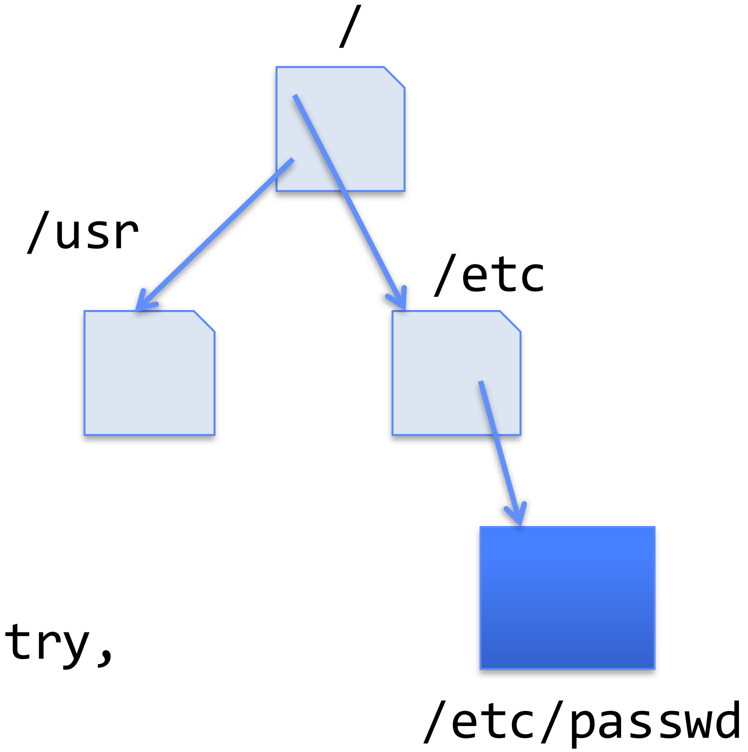
Manipulating Directories

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)



Example: Early Unix File System

Superblock object: information about file system

Free bitmaps: what is allocated/not allocated

Inode object: represents a specific file

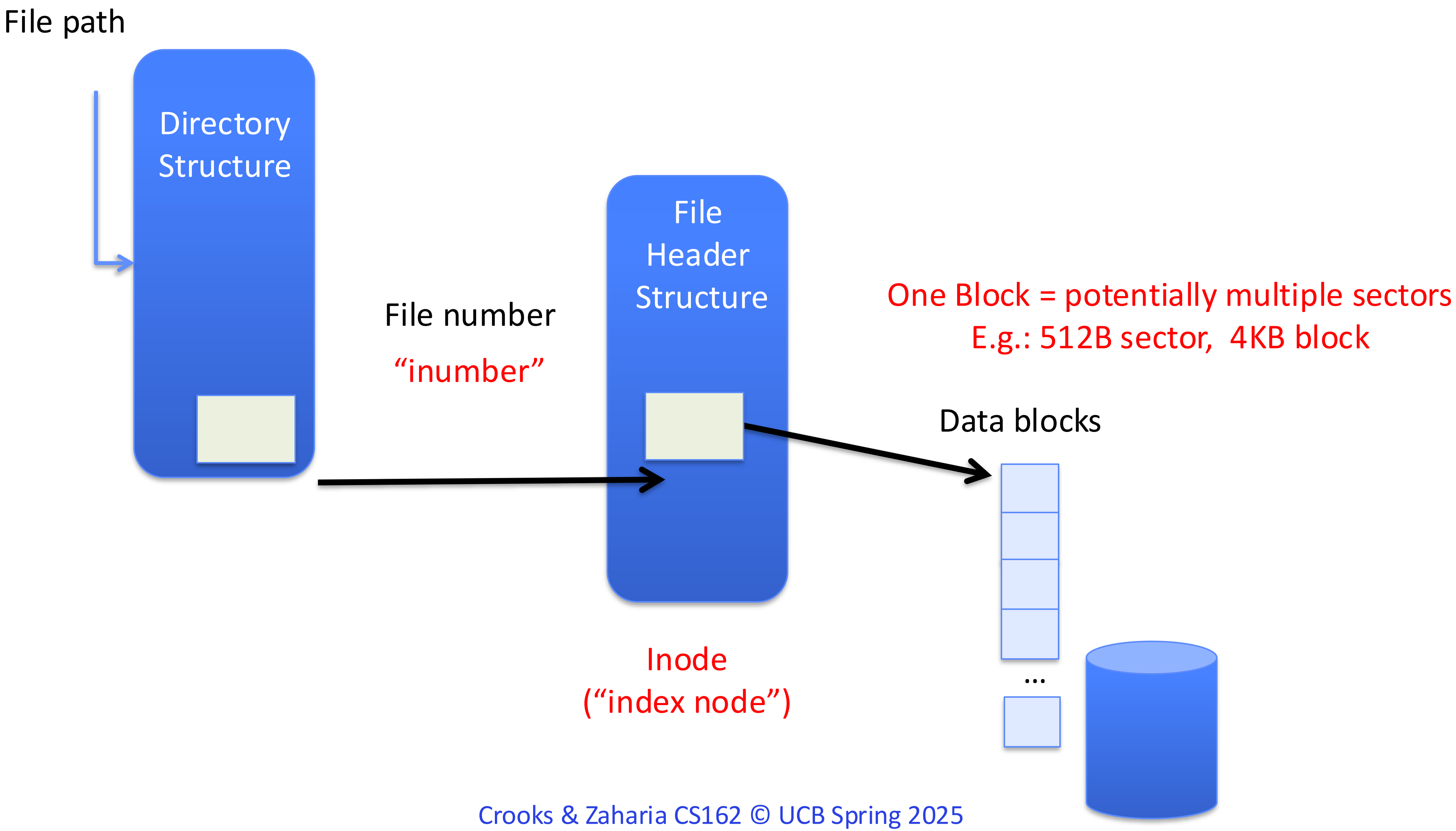
Dentry object: directory entry, single component of a path

Blocks: How files are stored on disk

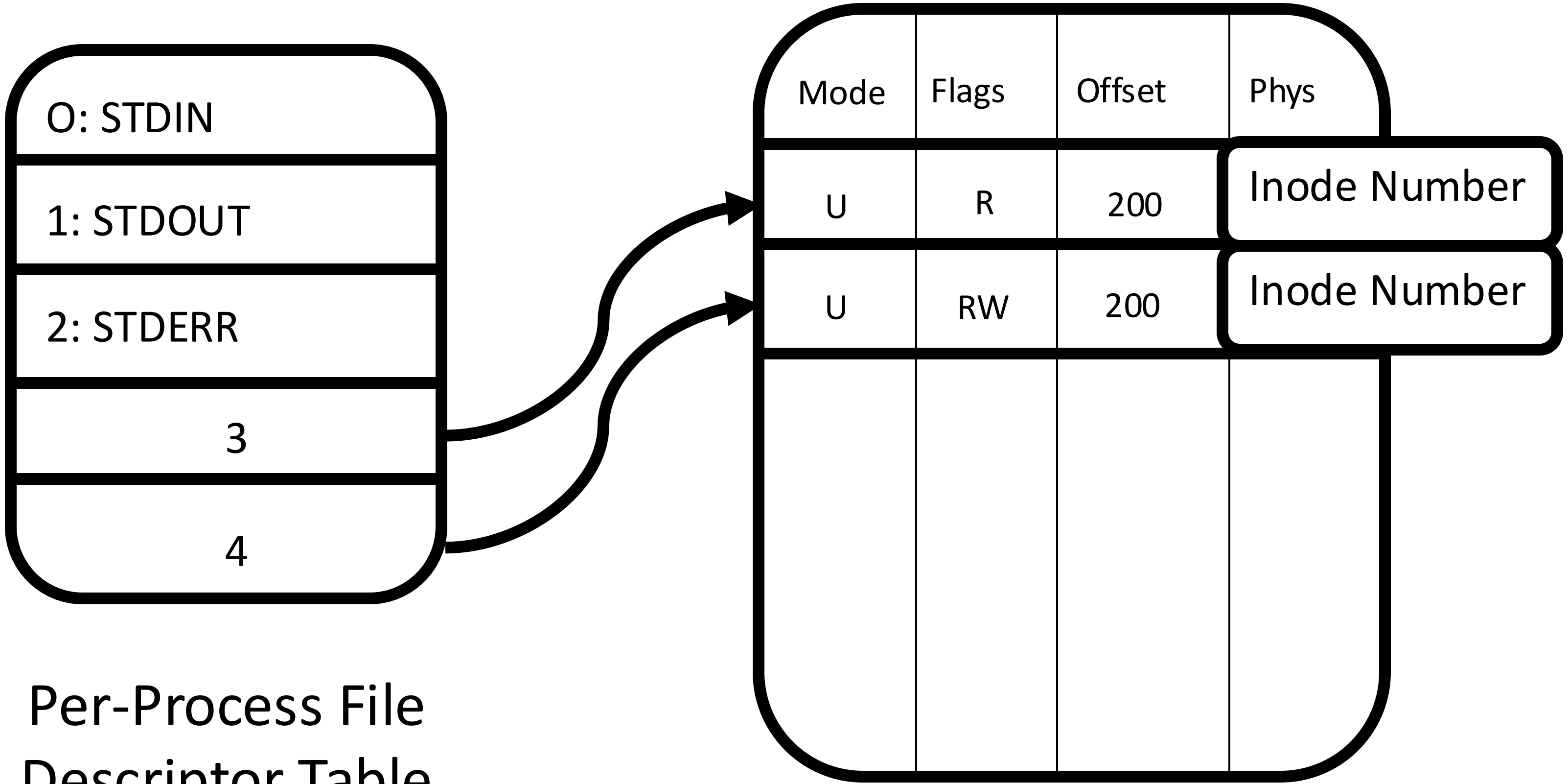
File object: open file associated with a process

Components of Unix File System

```
open("/home/matei/cs162/foo.txt")
```



The (In)famous Inode



O: STDIN
1: STDOUT
2: STDERR
3
4

Per-Process File Descriptor Table

Mode	Flags	Offset	Phys
U	R	200	Inode Number
U	RW	200	Inode Number

Global Open File Description Table

How to Find a File’s Inode Number?

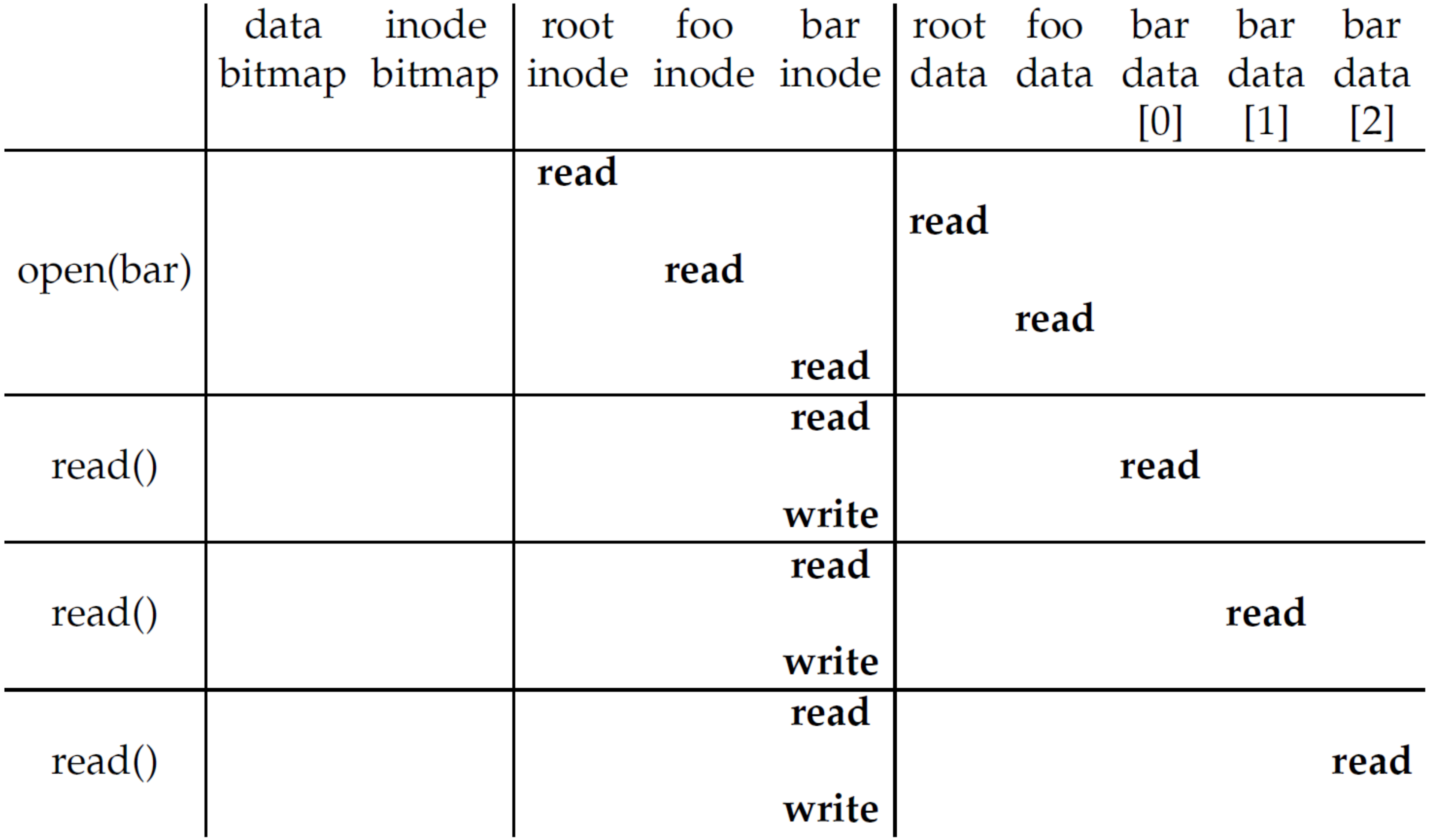
Look up through directory structure

A directory is a specialized file containing
<file_name : inode number> mappings

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

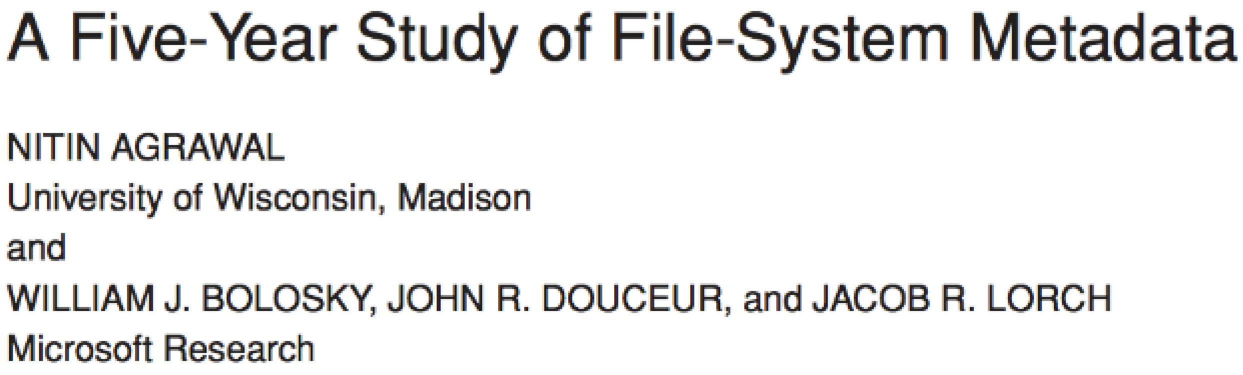
How to Read a File from Disk

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					

File System Workload Characteristics



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

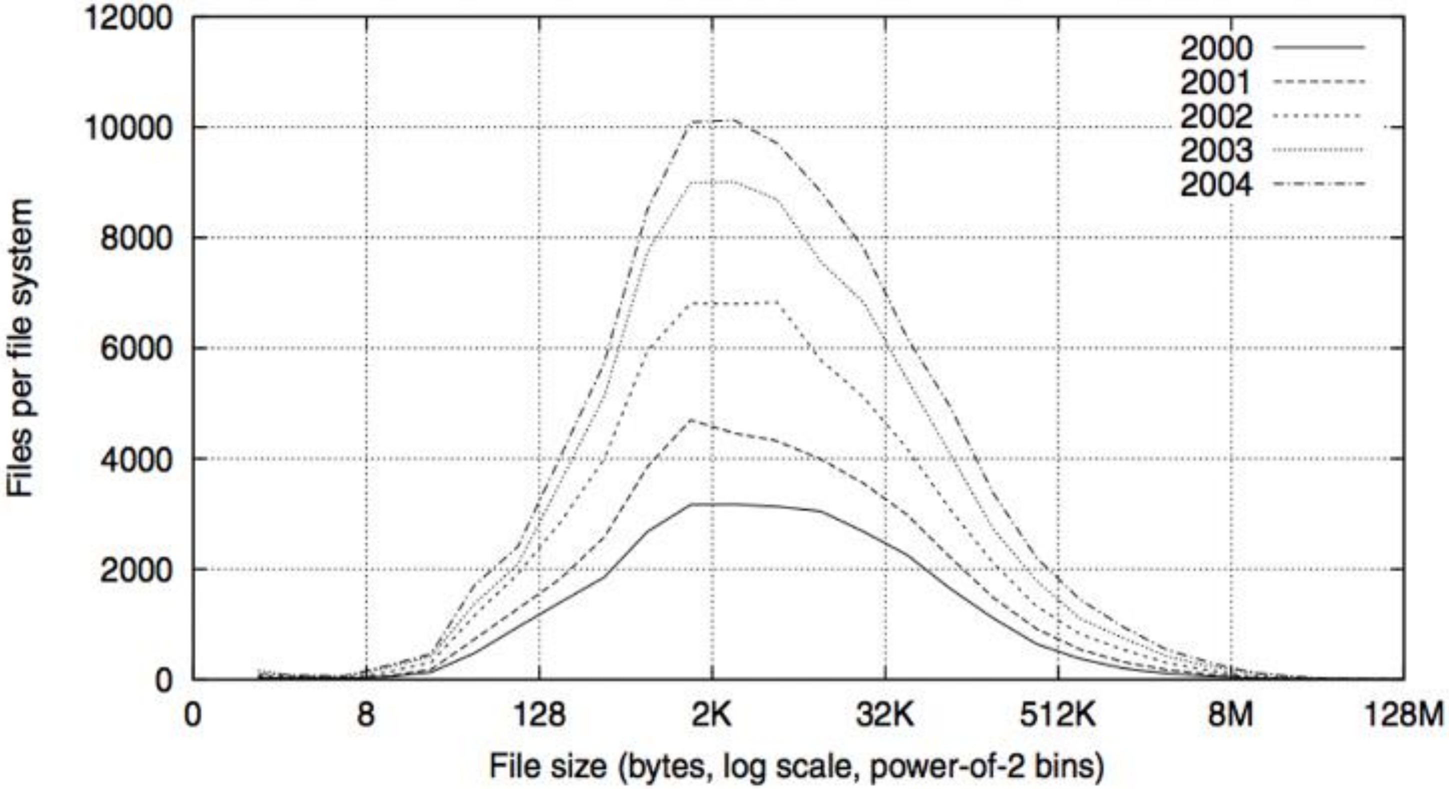


Fig. 2. Histograms of files by size.

Observation #2: Most Bytes are in Large Files

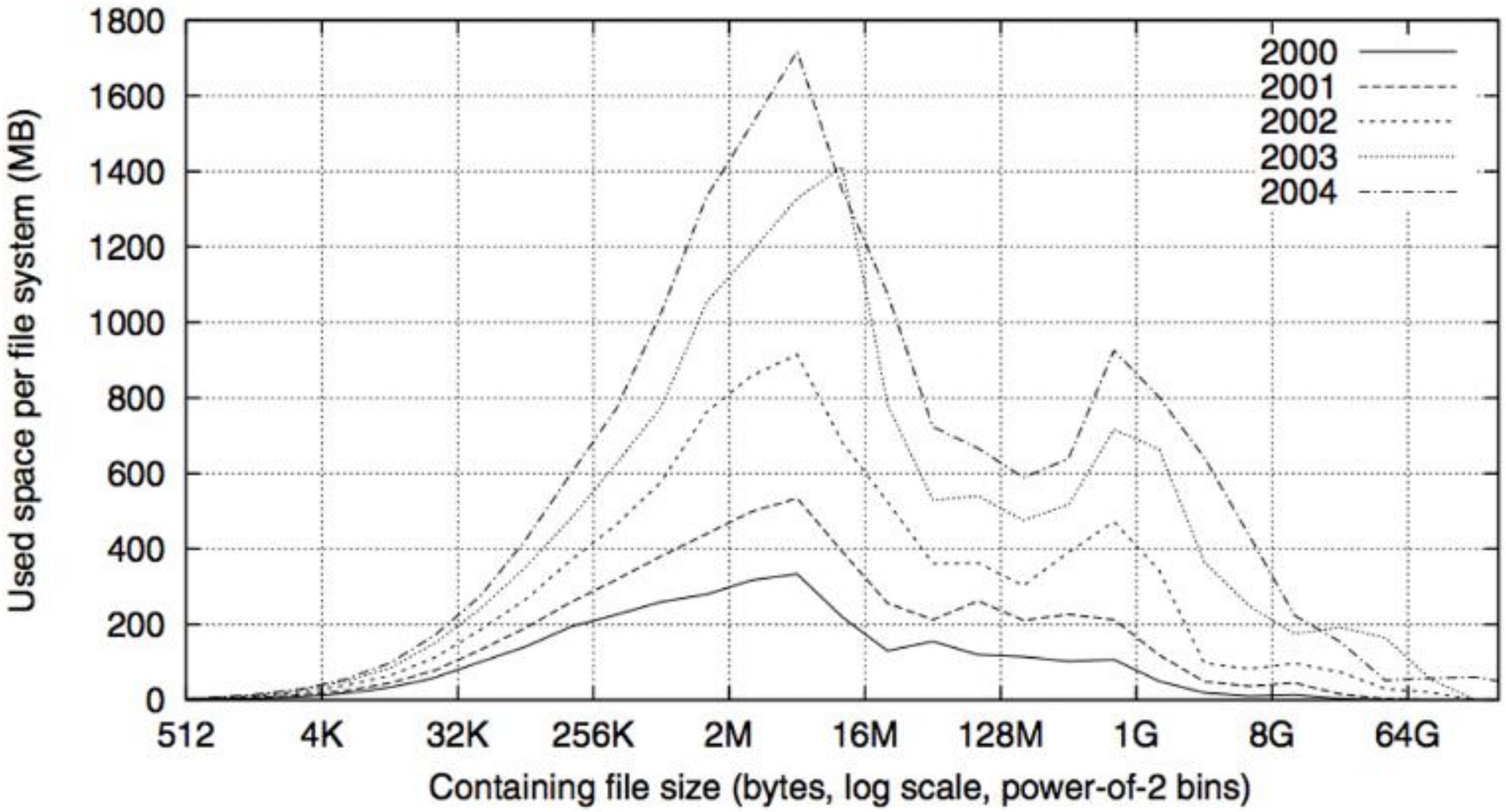


Fig. 4. Histograms of bytes by containing file size.

Unix Inode Structure

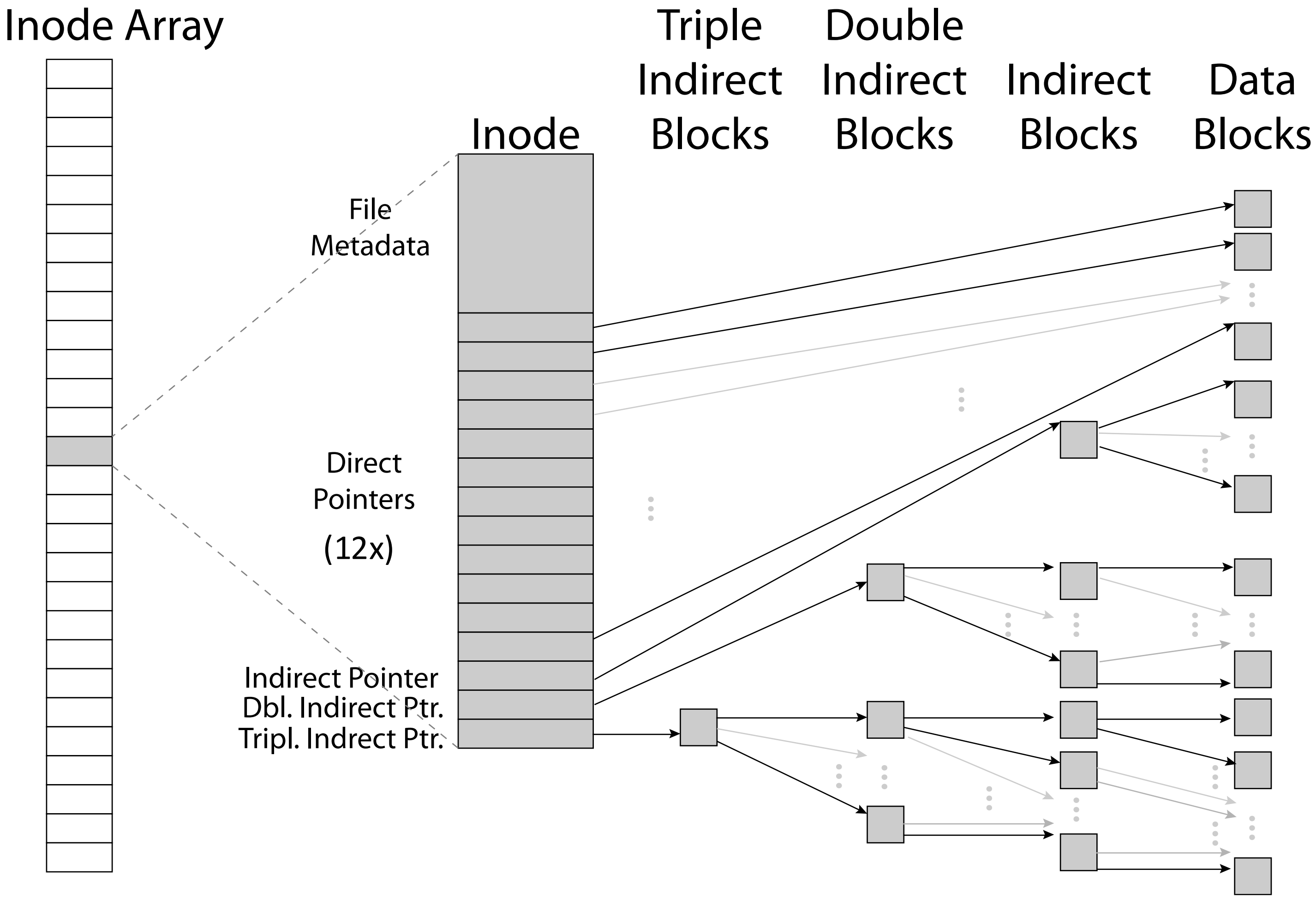
File Number is index into an array of *inode* structures

Each inode corresponds to a file and contains its metadata

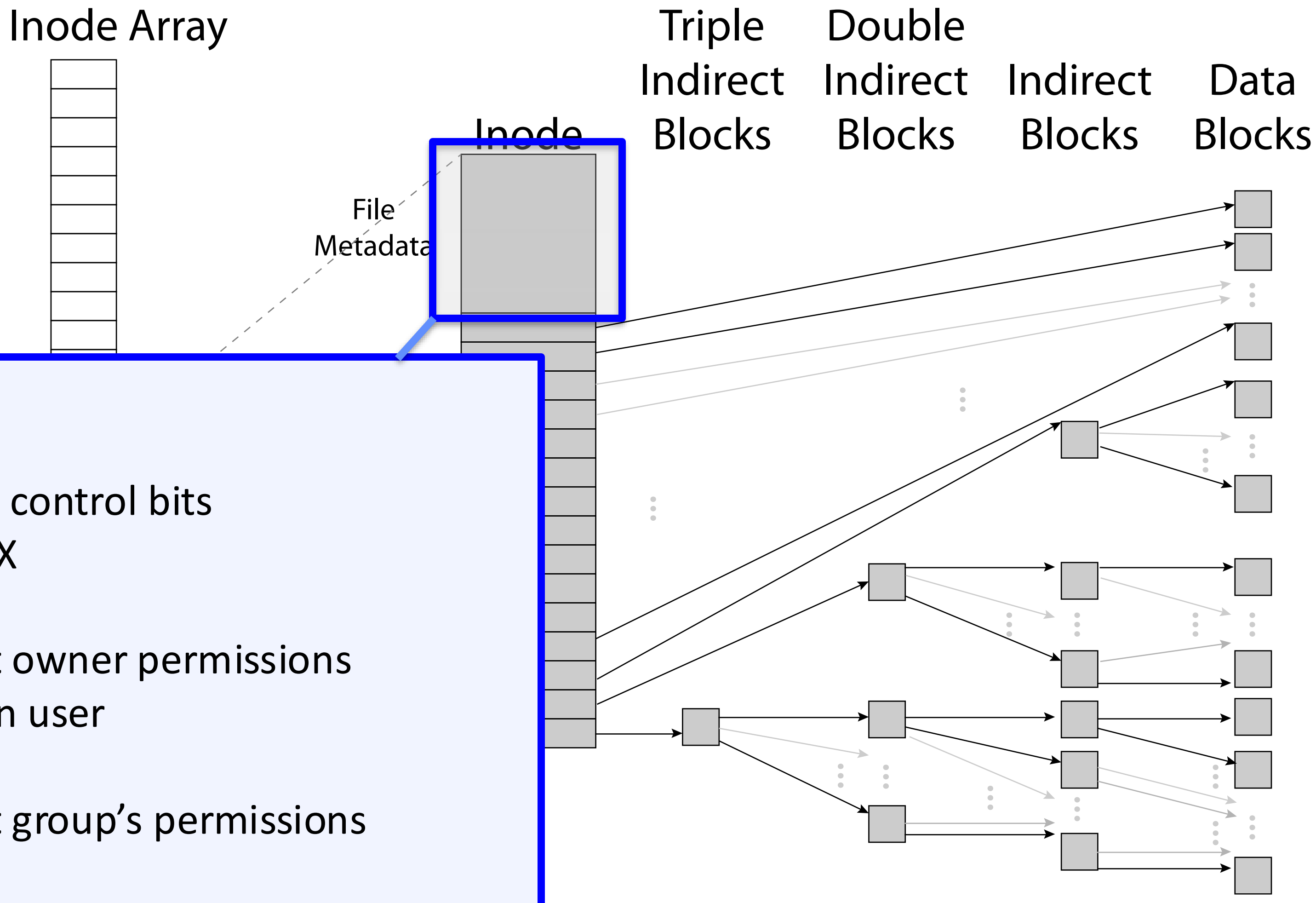
Inode maintains a multi-level tree to find storage blocks for files

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

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

12 Direct pointers

4kB blocks ⇒ sufficient for files up to 48KB

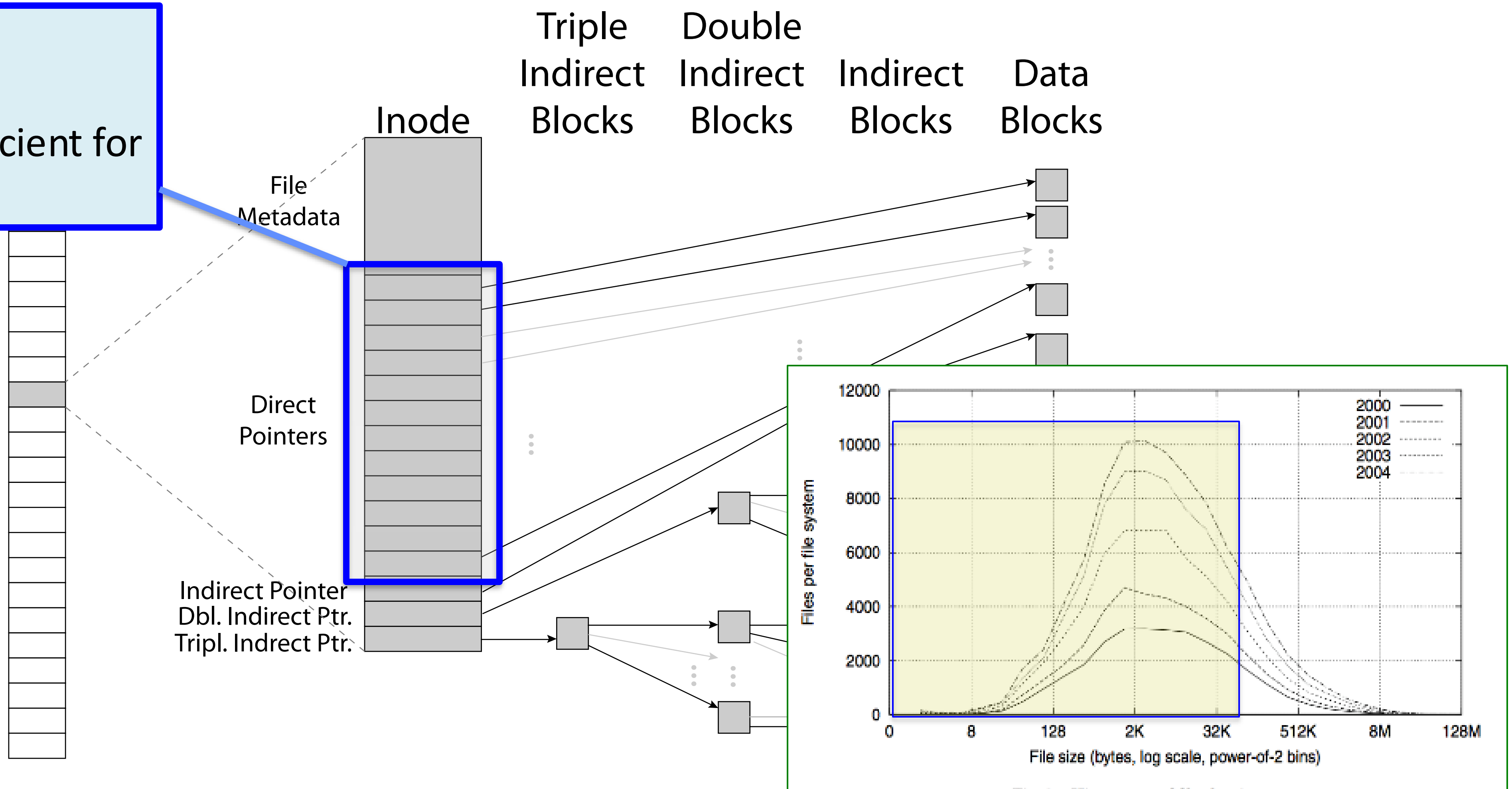
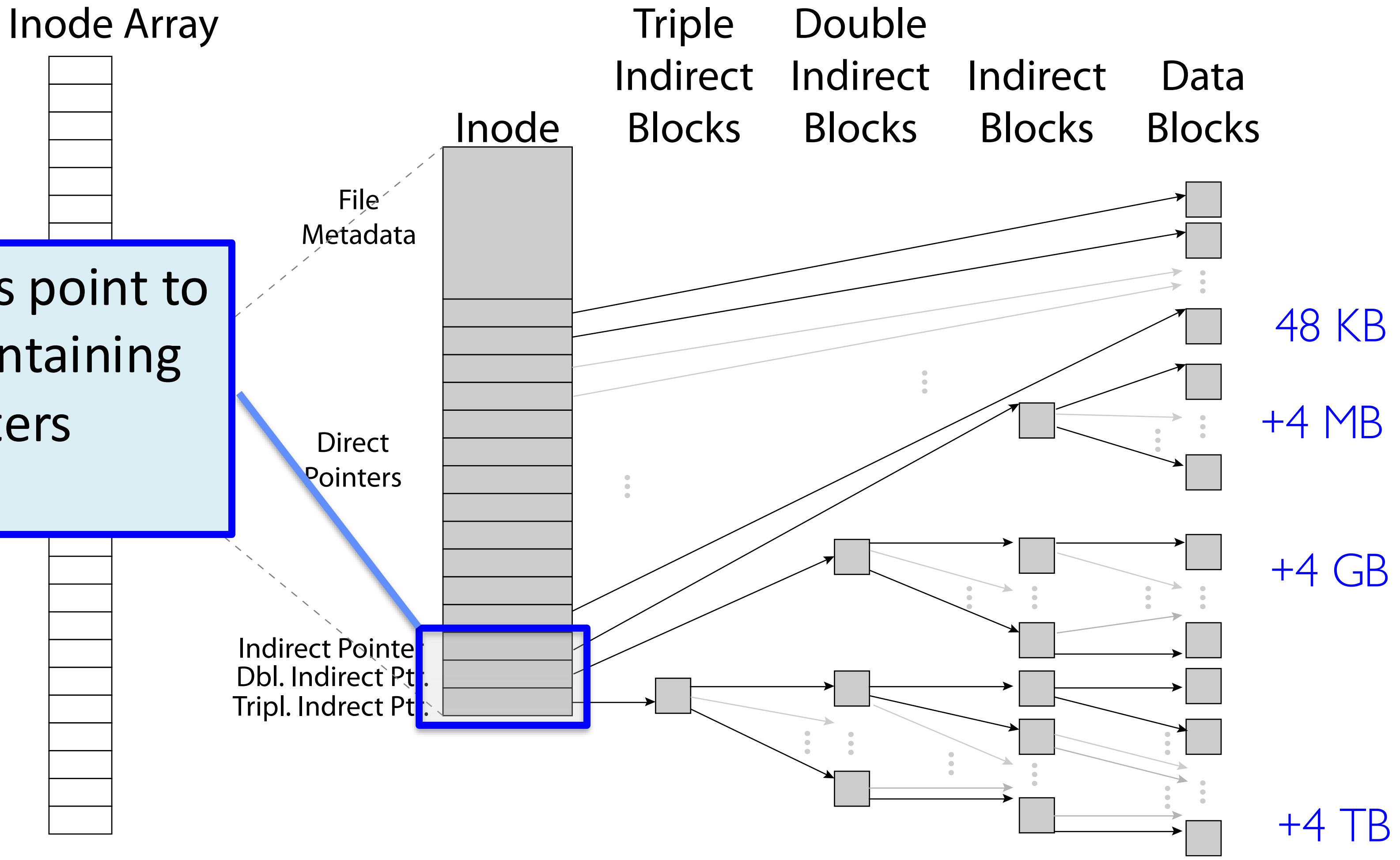


Fig. 2. Histograms of files by size.

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

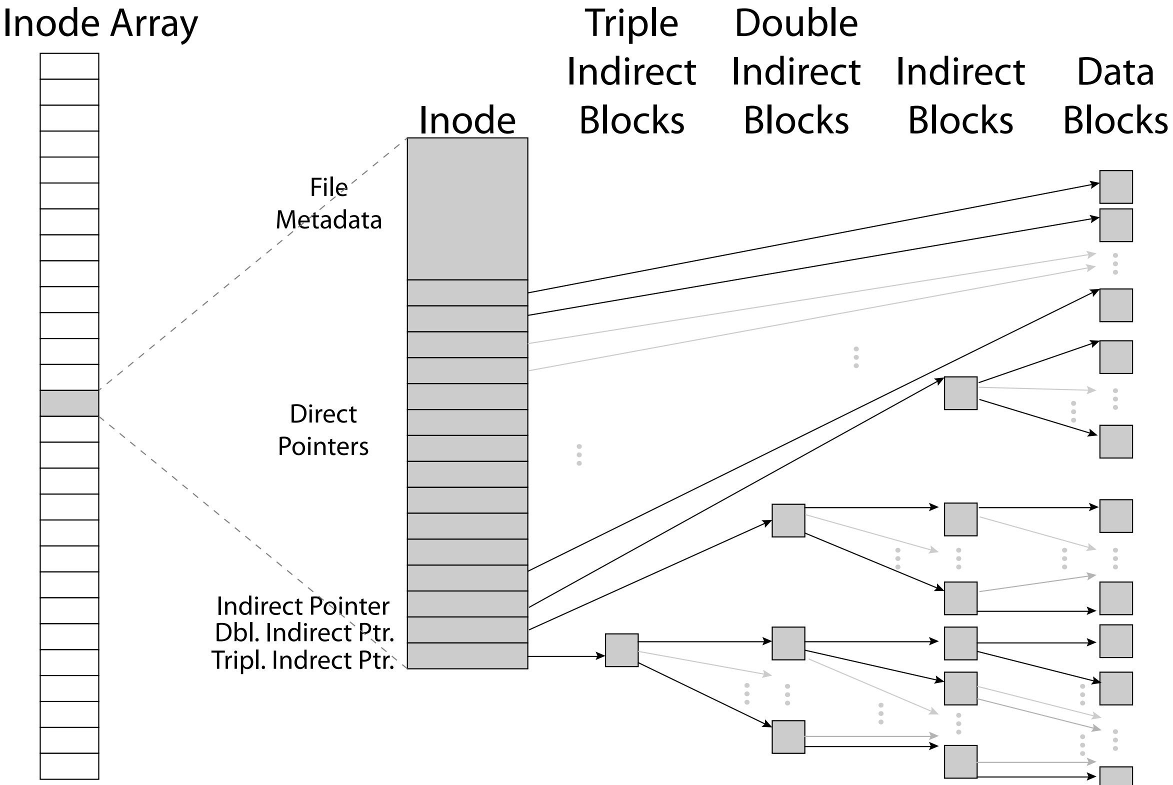
What is the maximum size of a file with double indirect pointers?

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 #1100?
  - » 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

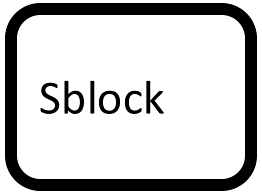
/cs162/matei.txt (60KB)

Each block is 4KB

Inode is 256 Bytes

Putting it together

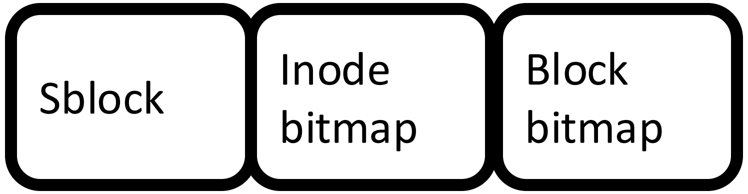
/cs162/matei.txt (60KB)



Sblock

Putting it together

/cs162/matei.txt (60KB)



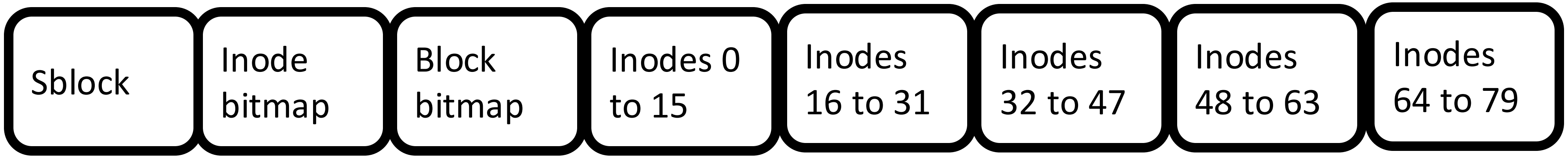
Sblock

Inode bitmap

Block bitmap

Putting it together

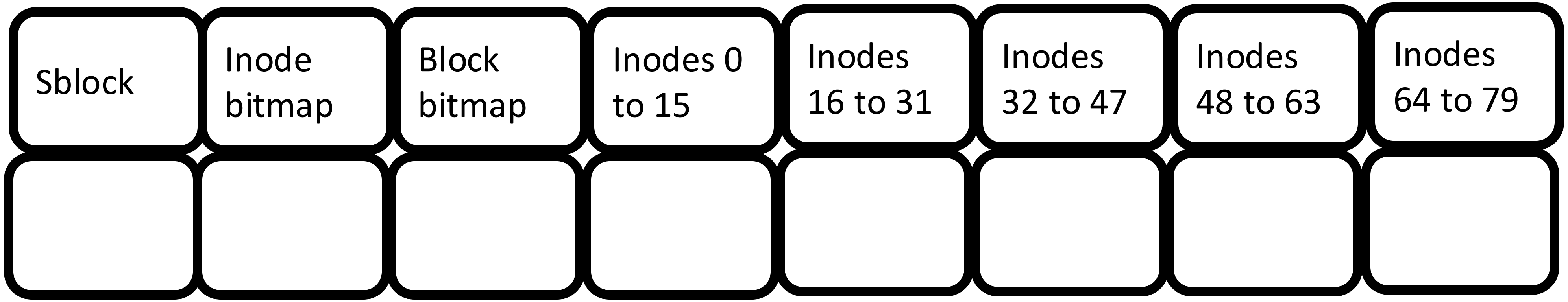
/cs162/matei.txt (60KB)



Sblock	Inode bitmap	Block bitmap	Inodes 0 to 15	Inodes 16 to 31	Inodes 32 to 47	Inodes 48 to 63	Inodes 64 to 79

Putting it together

/cs162/matei.txt (60KB)



Sblock

Inode bitmap

Block bitmap

Inodes 0 to 15

Inodes 16 to 31

Inodes 32 to 47

Inodes 48 to 63

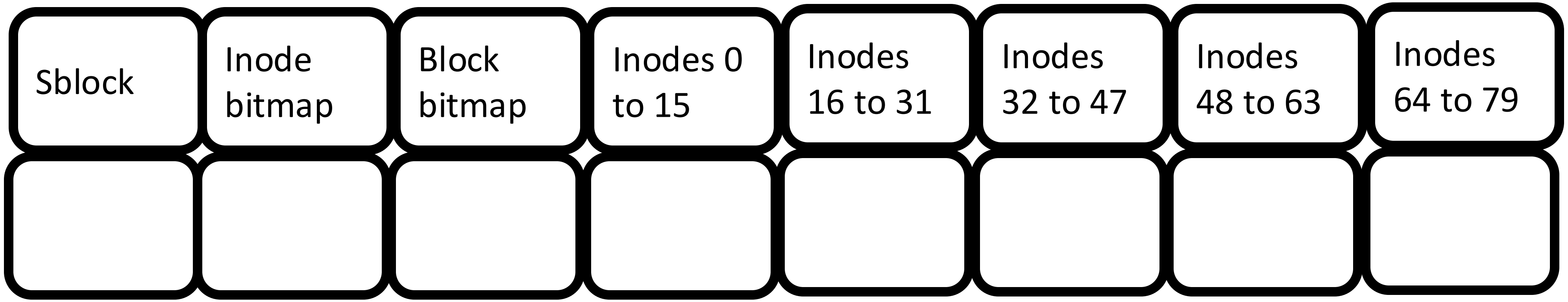
Inodes 64 to 79

Putting it together

/

Allocate inode 0

Create data block



Sblock

Inode bitmap

Block bitmap

Inodes 0 to 15

Inodes 16 to 31

Inodes 32 to 47

Inodes 48 to 63

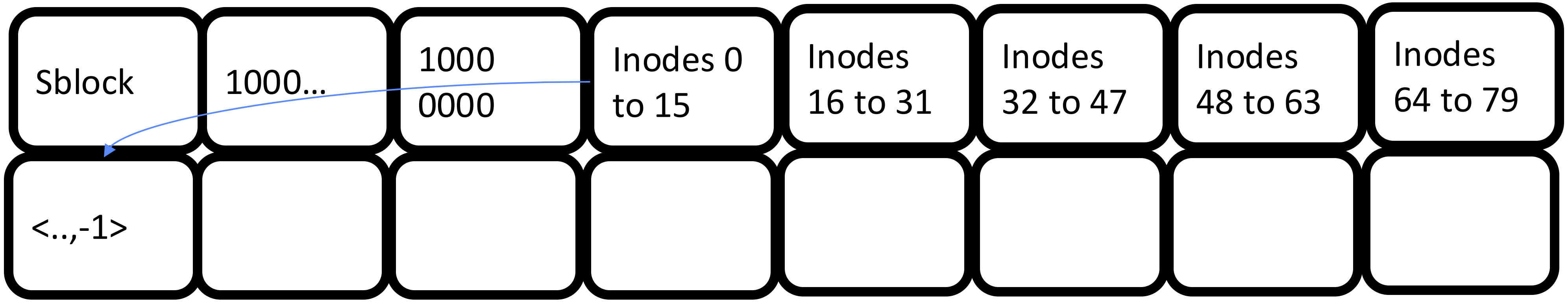
Inodes 64 to 79

Putting it together

/

Allocate inode 0

Create data block



Sblock	1000…	1000 0000	Inodes 0 to 15	Inodes 16 to 31	Inodes 32 to 47	Inodes 48 to 63	Inodes 64 to 79
<..,-1>							

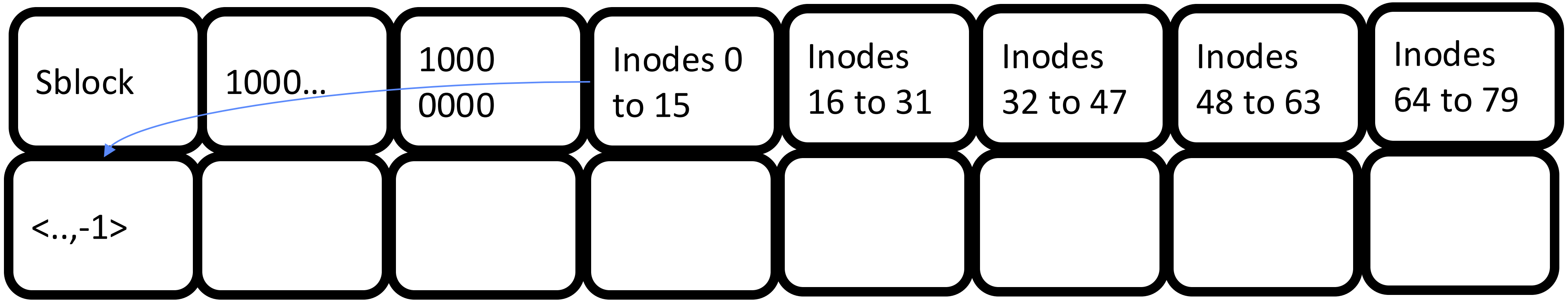
Putting it together

/cs162

Allocate inode 1

Update direntry for /

Create data block



Sblock

1000…

1000
0000

Inodes 0
to 15

Inodes
16 to 31

Inodes
32 to 47

Inodes
48 to 63

Inodes
64 to 79

<..,-1>

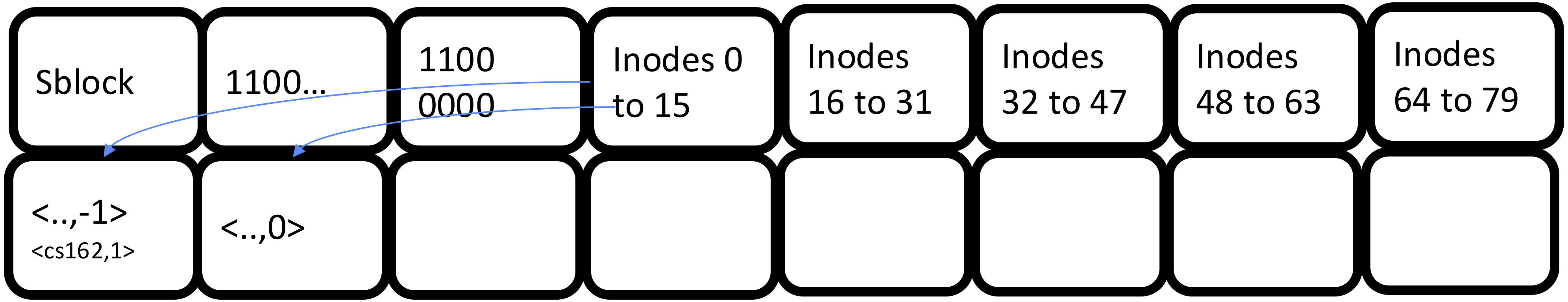
Putting it together

/cs162

Allocate inode 1

Update direntry for /

Create data block



Putting it together

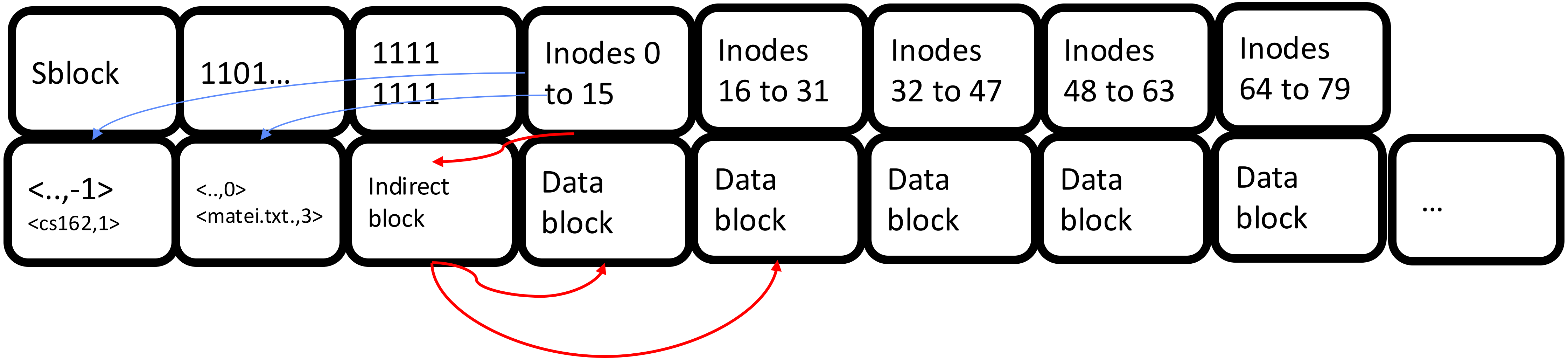
/cs162/matei.txt (60KB)

Allocate inode 3

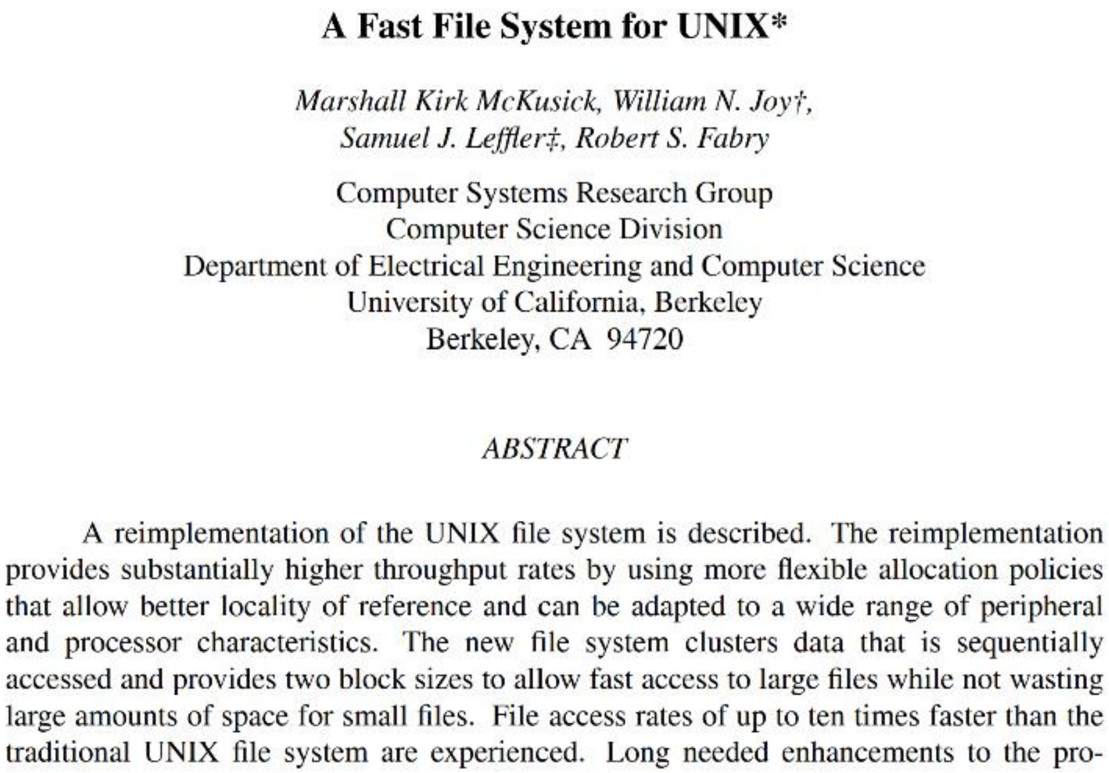
Update dentry

Create indirect block

Create data blocks



Unix File System Improved (Berkeley Fast File System)



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-

Introducing Disk Awareness

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 data is in advance

Size is determined as files are used !!!

– Can write (or read zeros) to expand the file

– Start small and grow, need to make room

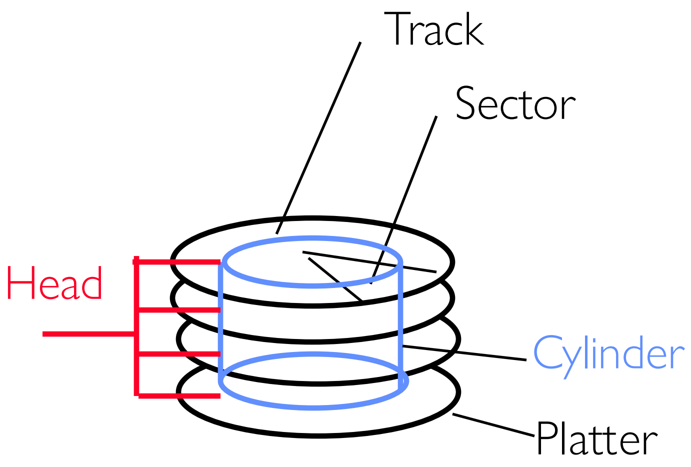
Organized into directories

– 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)

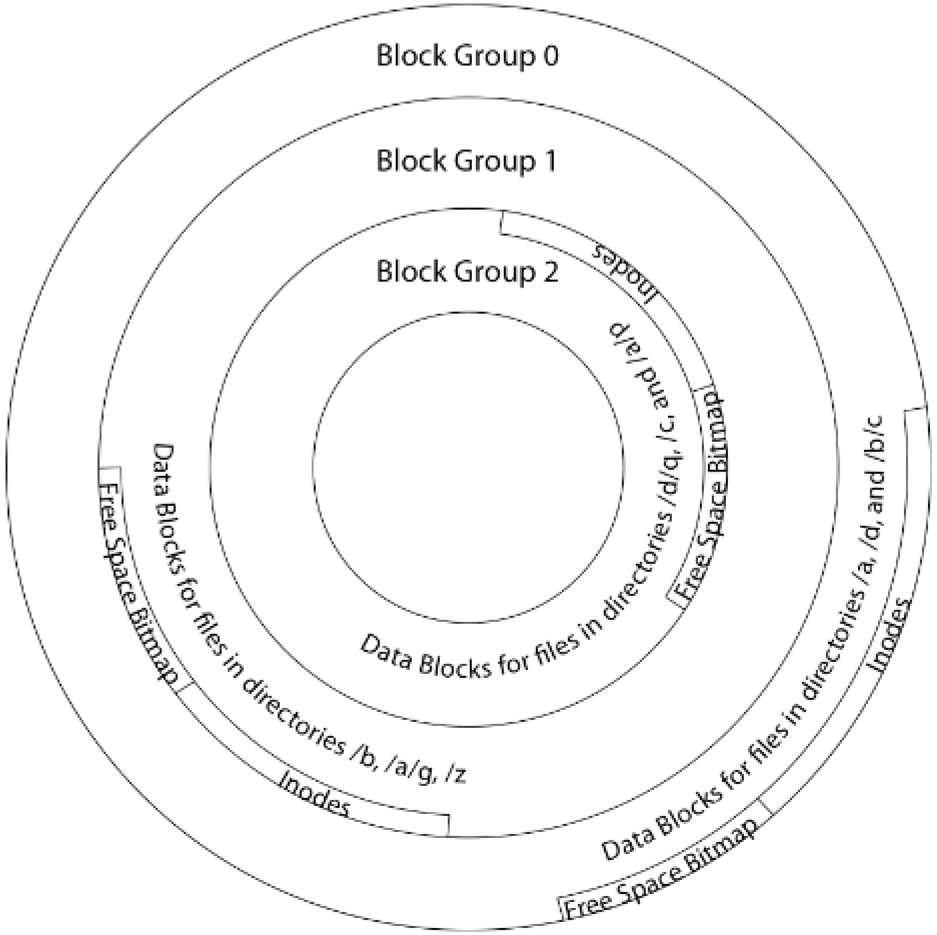
FFS Locality: Block Groups

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 a directory and its files in same block group



Block Group 0
Block Group 1
Block Group 2
Inodes
Data Blocks for files in directories /d/q, /c, and /a/p
Free Space Bitmap
Data Blocks for files in directories /b, /a/g, /z
Free Space Bitmap
Inodes
Data Blocks for files in directories /a, /d, and /b/c
Inodes
Free Space Bitmap

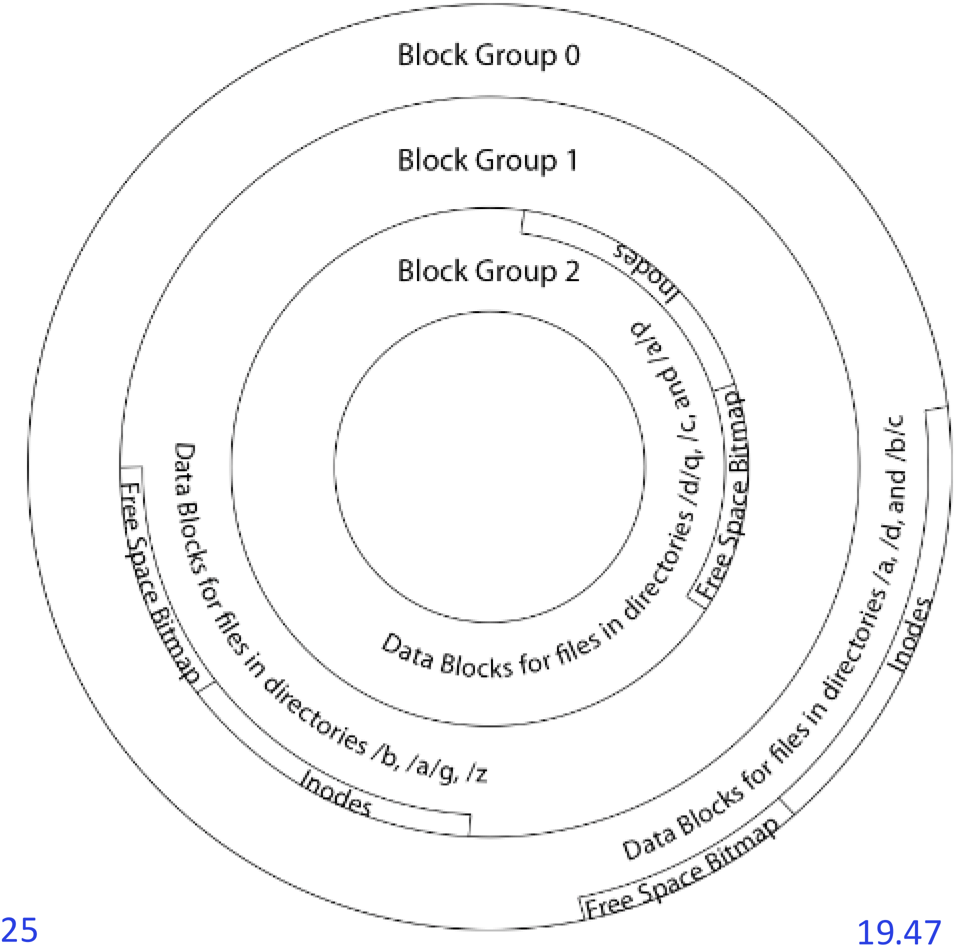
FFS Locality: Block Groups

First-Free allocation of new file blocks

- To expand file, first try successive blocks in bitmap, then choose a 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


Block Group 0
Block Group 1
Block Group 2
Inodes
Data Blocks for files in directories /d/q, /c, and /a/p
Free Space Bitmap
Data Blocks for files in directories /b, /a/g, /z
Inodes
Free Space Bitmap
Data Blocks for files in directories /a, /d, and /b/c
Inodes
Free Space Bitmap

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 by the OS
- » Or by the disk (“track buffers”): many modern 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

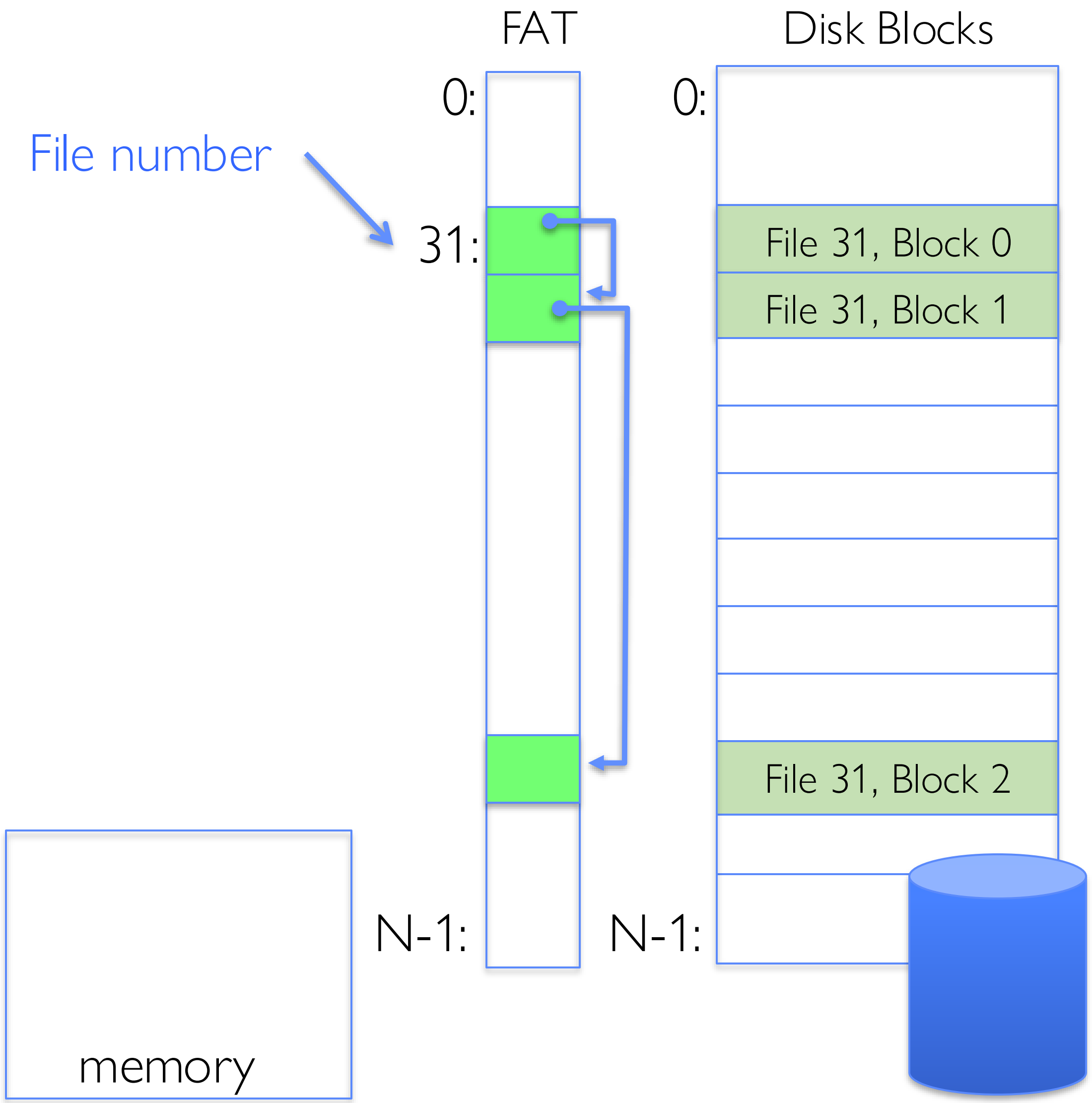
Disk Storage is a collection of Blocks

- 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

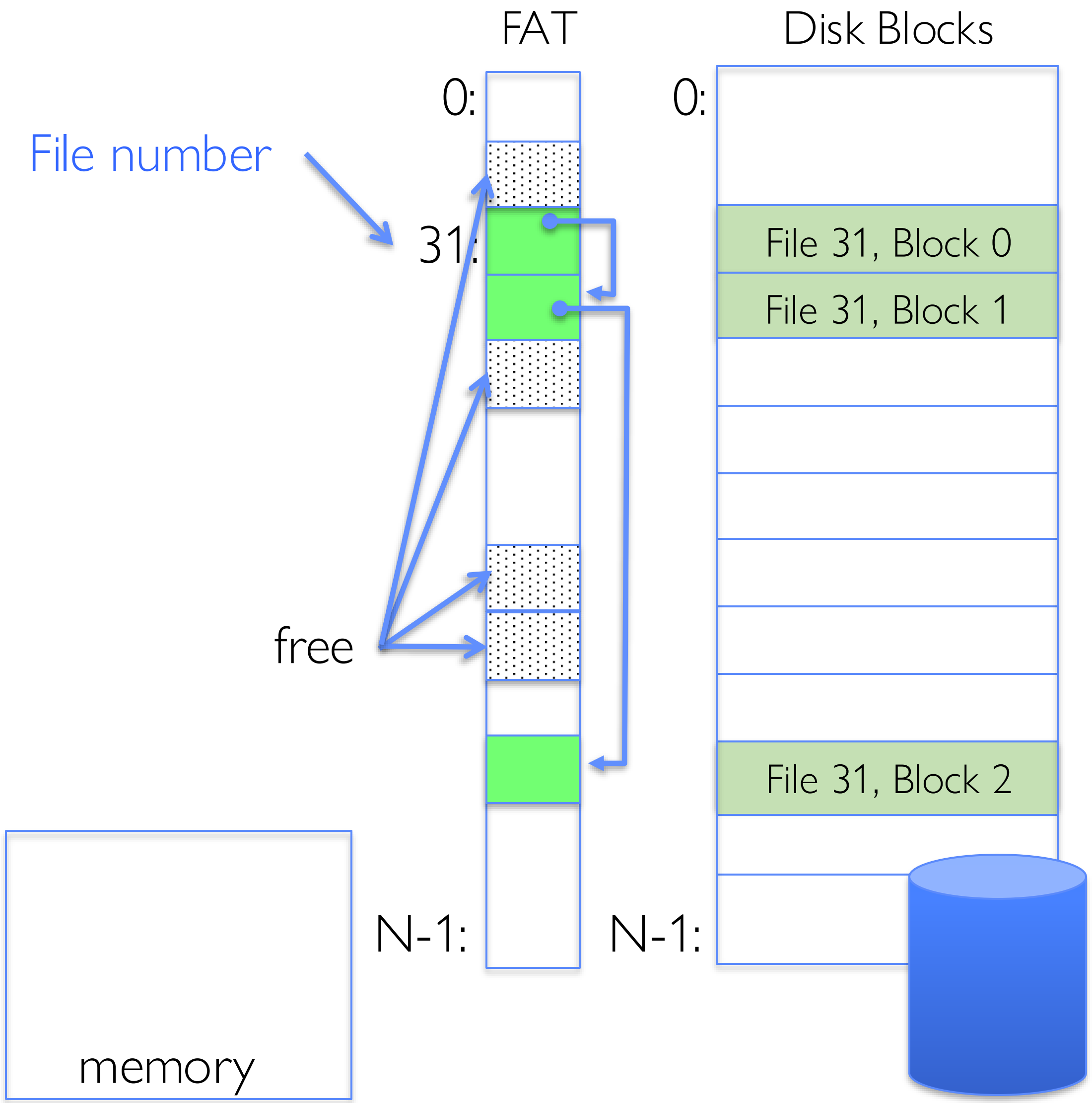
File number is index of root of block list for the file

File offset: block number and offset within block

Follow list to get block number

Unused blocks marked free

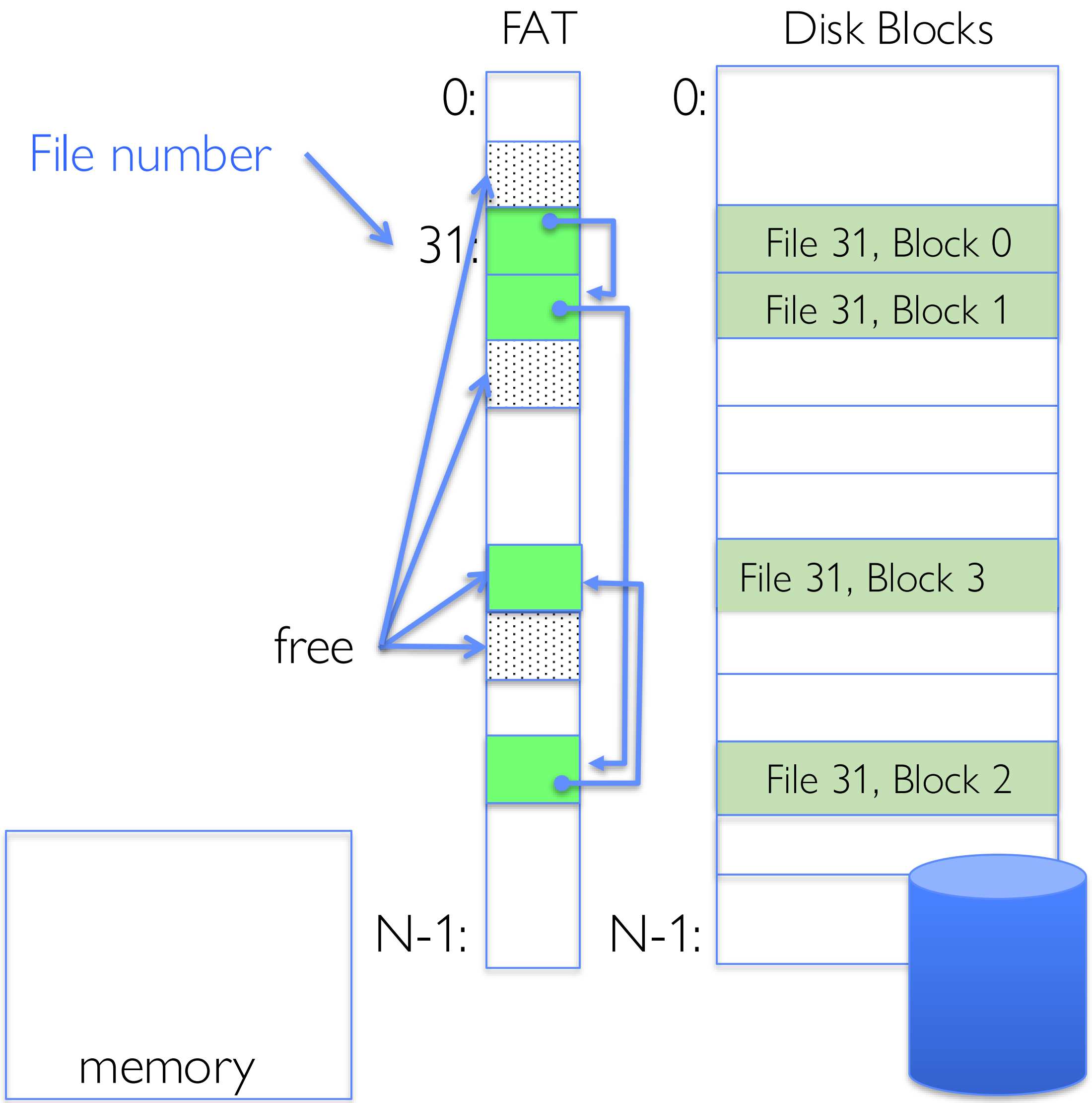
- Could require scan to find
- Or, could use a free list



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

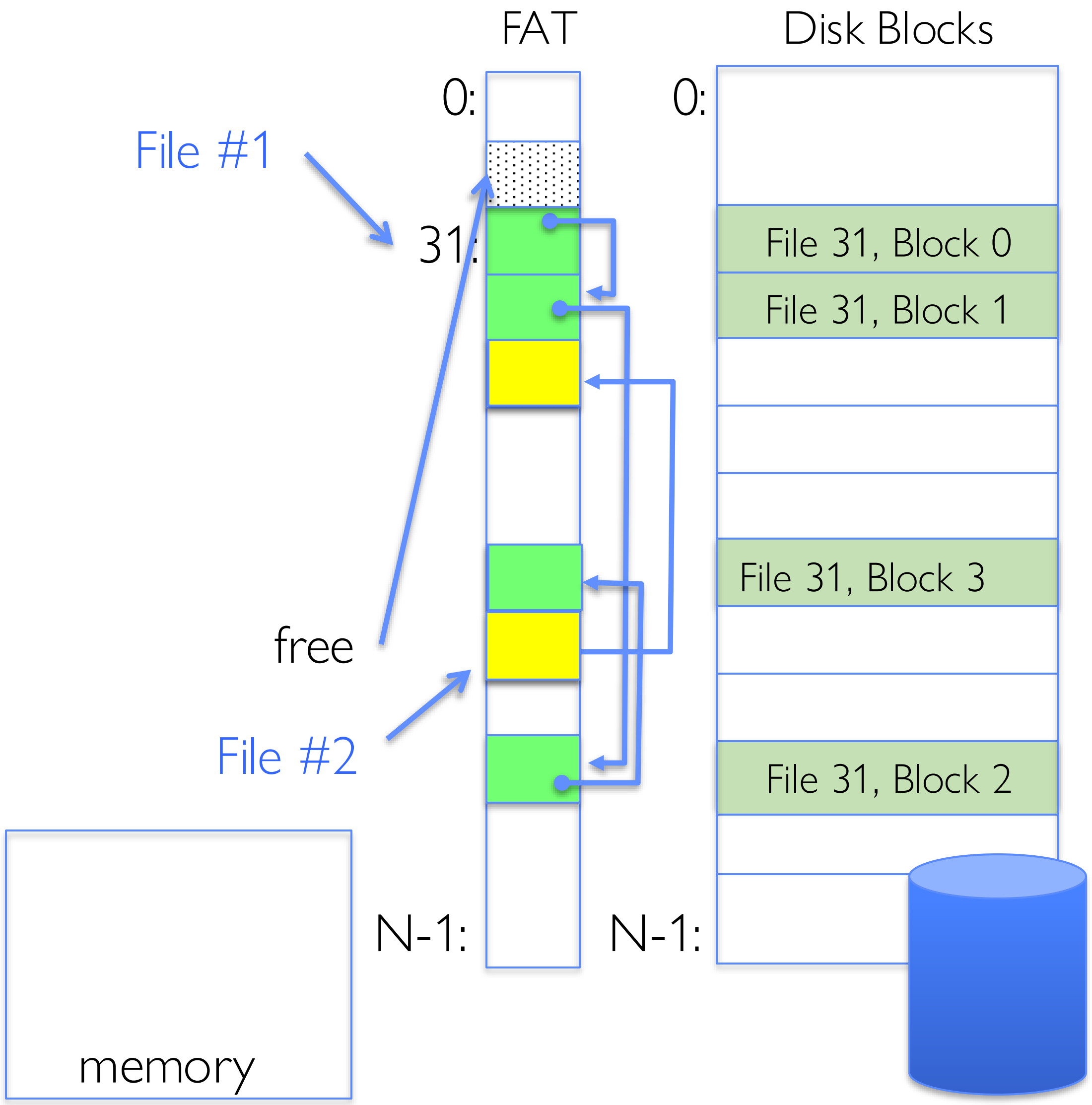
How to format a disk?

- Zero the blocks, mark FAT entries "free"

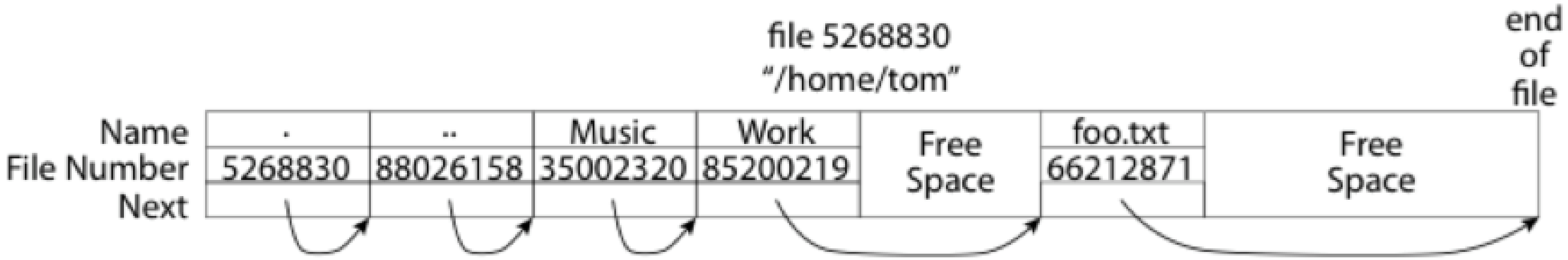
How to quick format a disk?

- Mark FAT entries "free"

Simple: can implement in device firmware



FAT: Directories



Name	.	..	Music	Work	Free Space	foo.txt	Free Space
File Number	5268830	88026158	35002320	85200219		66212871	
Next							

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

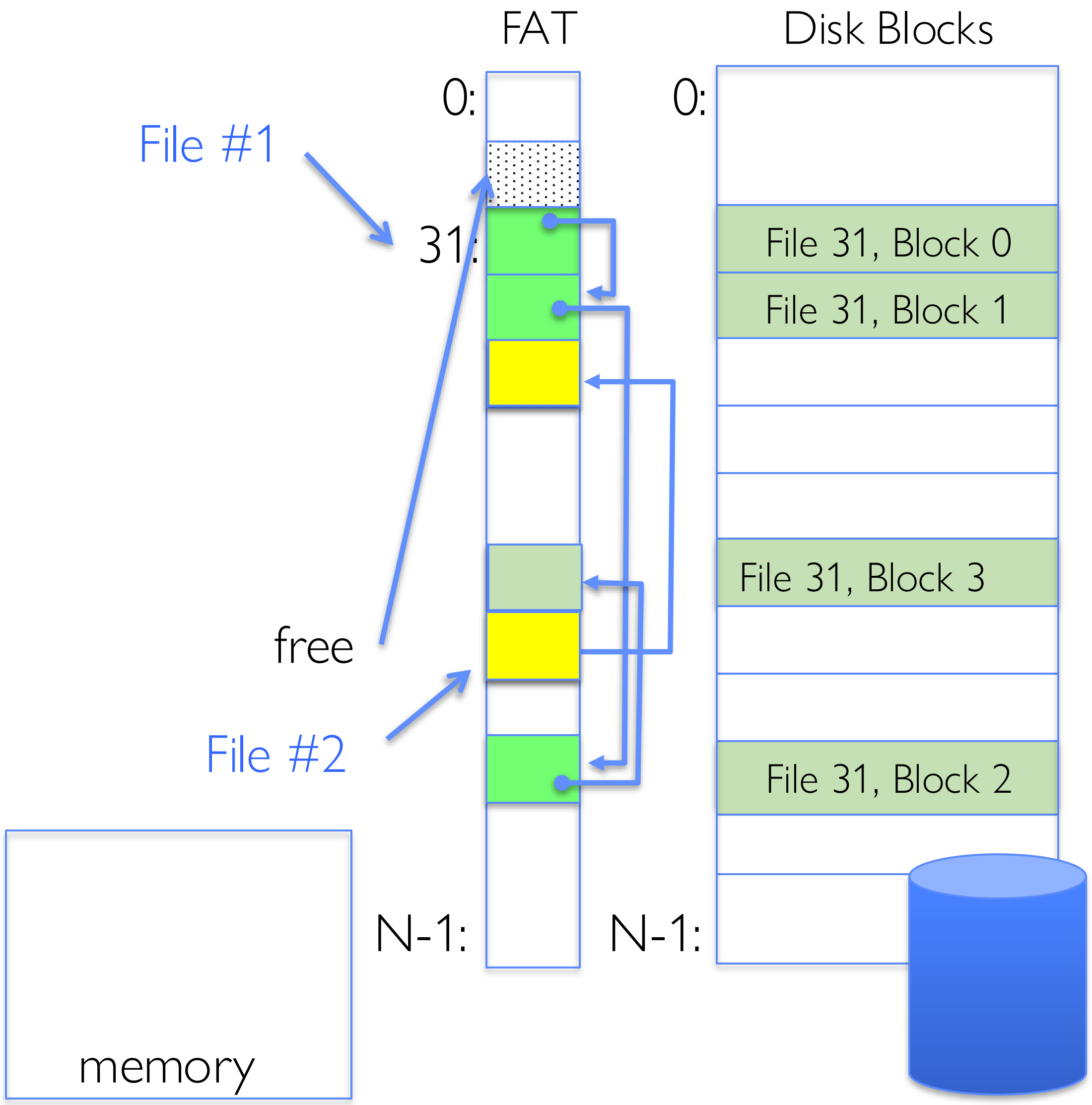
Where do you find root directory ("/")?

- 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 first block?
- Block layout for file?
- Sequential access?
- Random access?
- Fragmentation?
- Small files?
- Big files?



Windows NTFS

New Technology File System (NTFS)

Default on modern Windows systems

Variable length extents

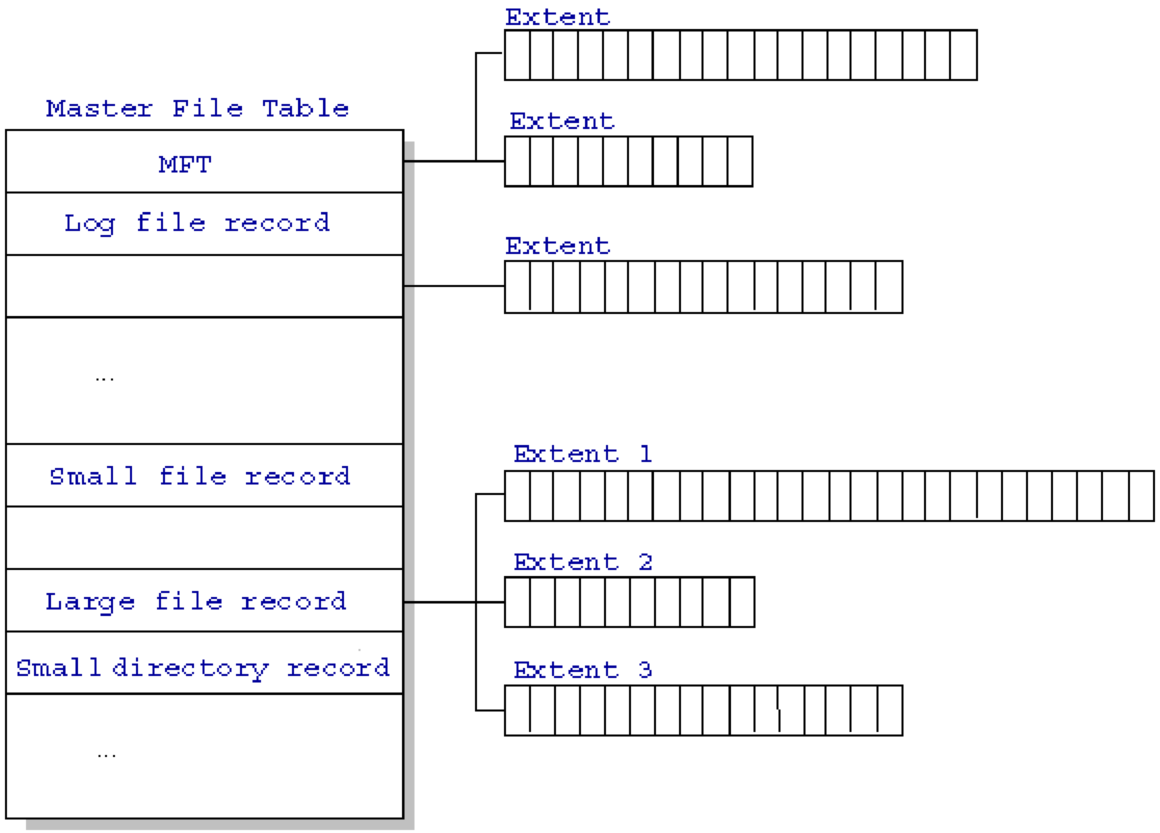
Master File Table

- Everything (almost) is a sequence of <attribute:value>

Each entry in MFT contains metadata and:

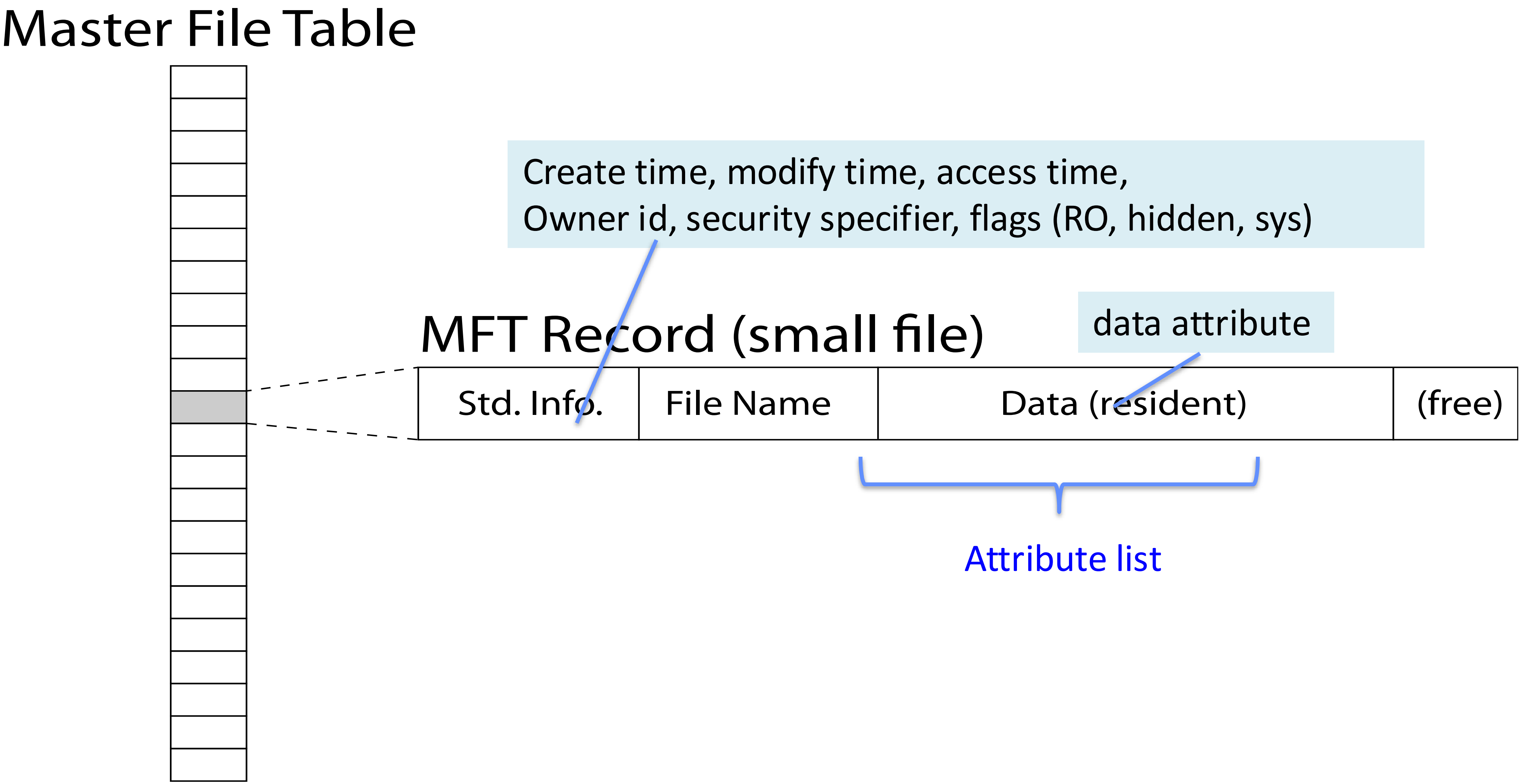
- File’s data directly (for small files)
- A list of *extents* (start block, size) for file’s data
- For big files: pointers to other MFT entries with *more* extent lists

NTFS



http://ntfs.com/ntfs-mft.htm

NTFS Small File: Data stored with Metadata

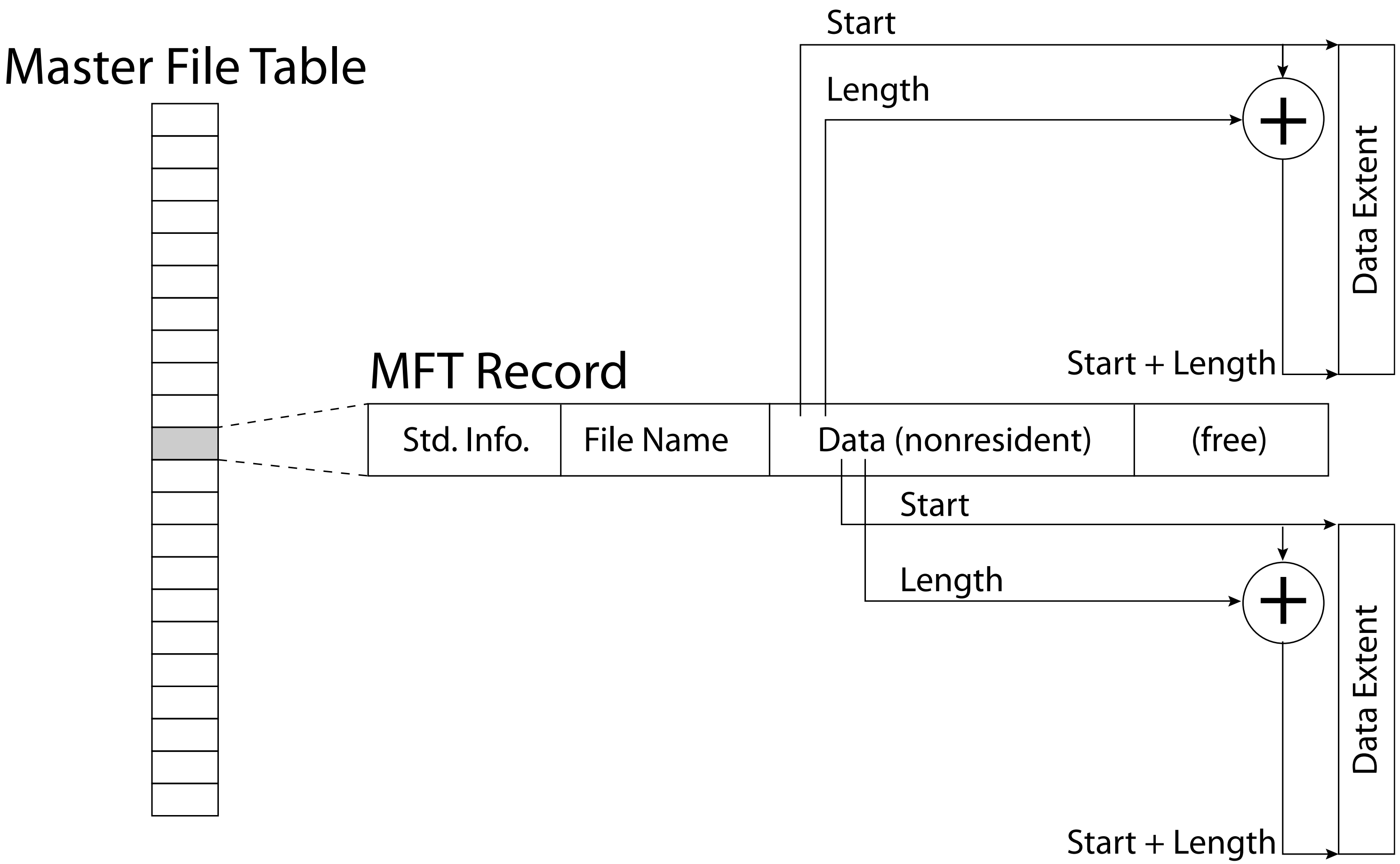


Create time, modify time, access time,
Owner id, security specifier, flags (RO, hidden, sys)

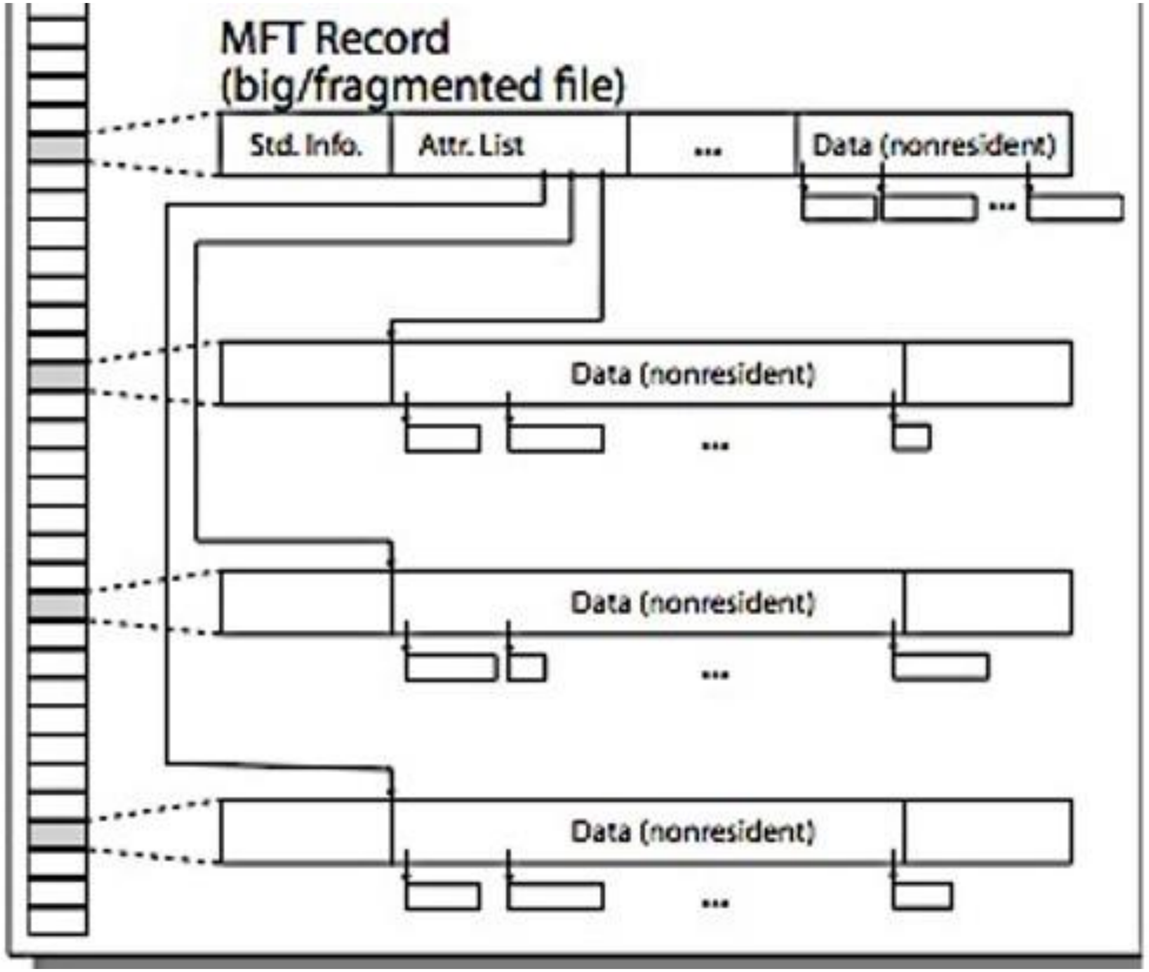
data attribute

Attribute list

NTFS Medium File: Extents for File Data



NTFS Large File: Pointers to Other MFT



NTFS Directories

Directories implemented as B-Trees

File's number identifies its entry in MFT

MFT entry always has a file name attribute

- Human readable name, file number of parent dir

Hard link? Multiple file name attributes in MFT entry