CS 318 Principles of Operating Systems

Fall 2018

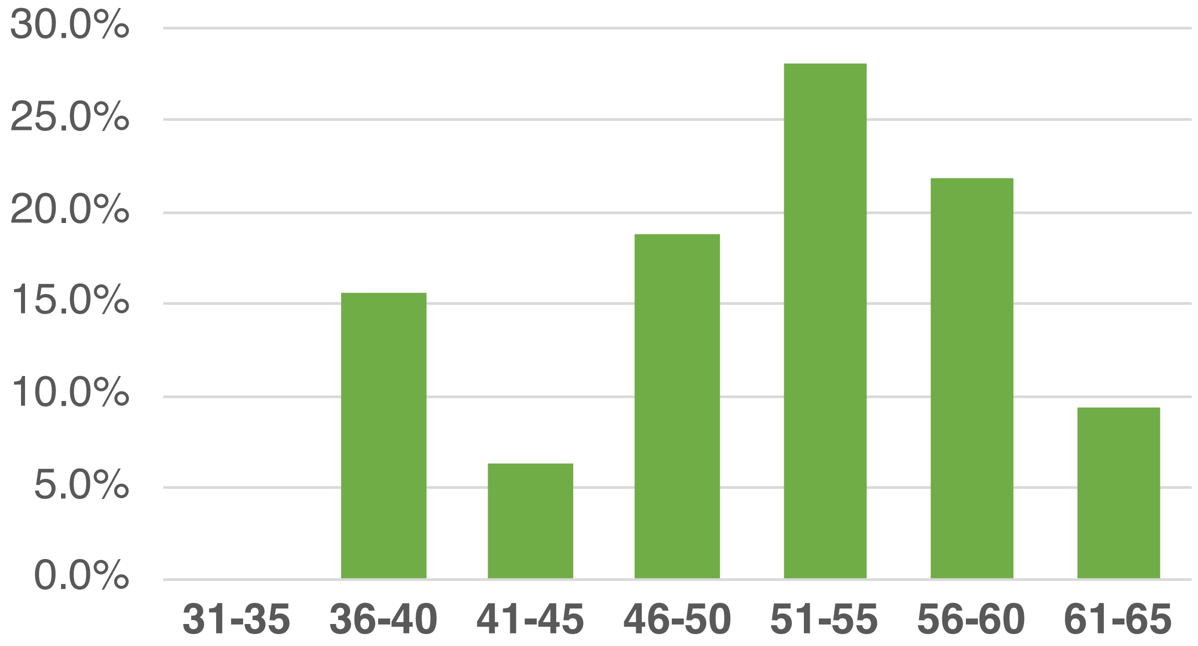
Lecture 15: File Systems

Ryan Huang

JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

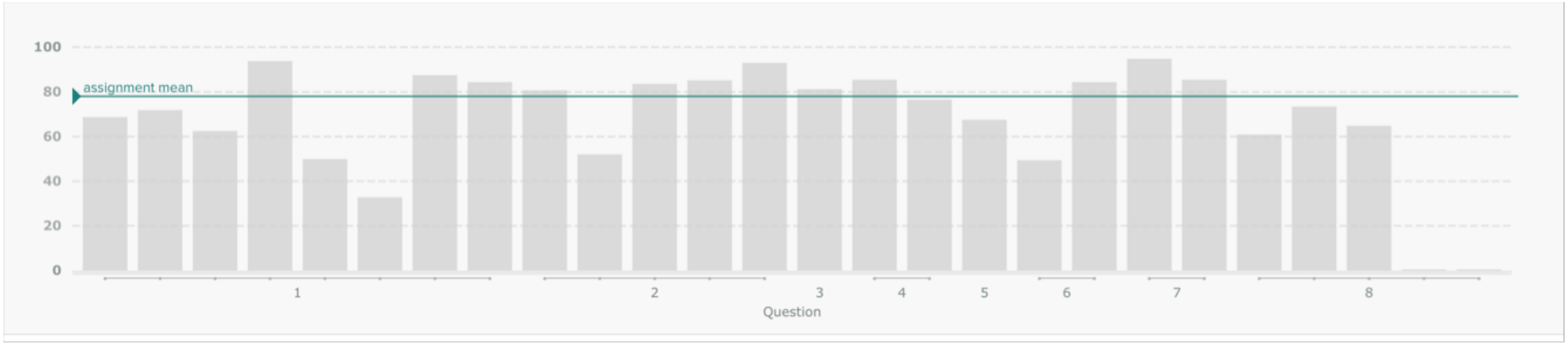
Slides adapted from David Mazières' lecture

Midterm Results

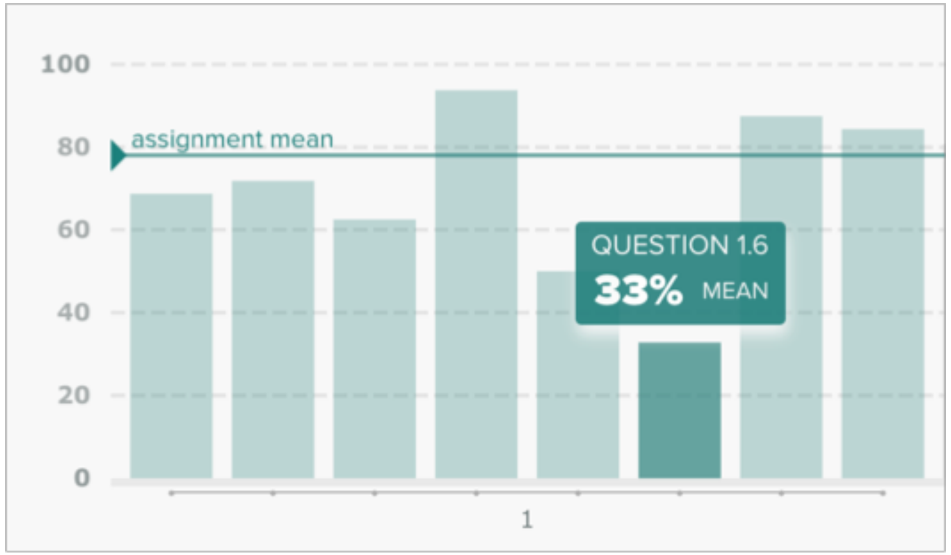


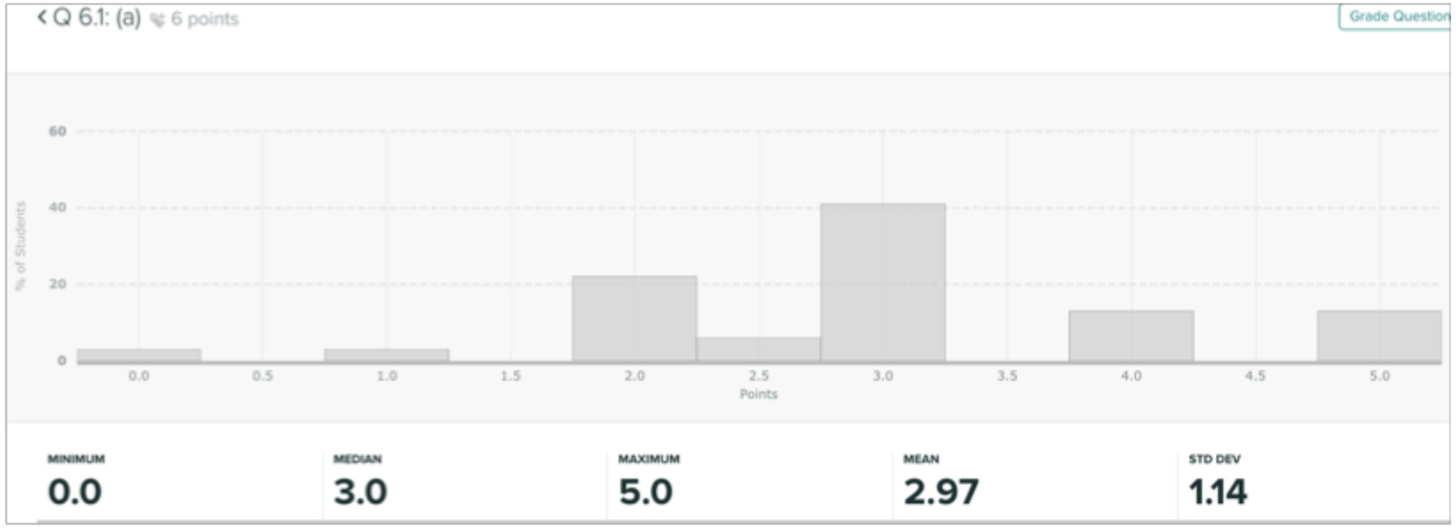
- **Mean: 50.58, Max: 64, STD Dev: 7.72**
  - 318 section: mean 53; 418 section : mean 51.5; 618 section: mean 48.6

Midterm Results



Midterm




< Q 6.1: (a) 6 points

MINIMUM 0.0

MEDIAN 3.0

MAXIMUM 5.0

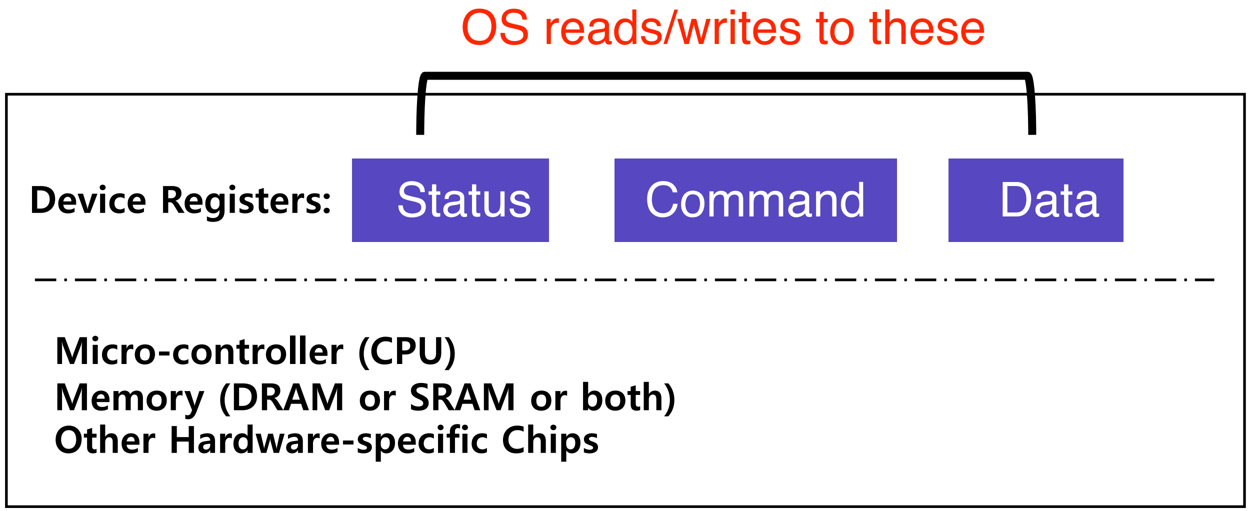
MEAN 2.97

STD DEV 1.14

Midterm Results

- **Scheduling problem perhaps takes the most time**
  - But many get it mostly right (mean is 8/10 points)
- **Synchronization problem is the toughest**
  - No one gets Q5, Q6 completely right
- **Some serious misconception**
  - E.g., syscall makes user-level threads faster (homework question)
  - Checking value of Semaphores
- **Don’t panic if you didn’t do well on midterm**
  - Still a lot of chance to make up, e.g., do Lab 3 well, do the extra credits
  - But do make sure you understand all the questions and answers now

Recap: I/O & Disks


OS reads/writes to these

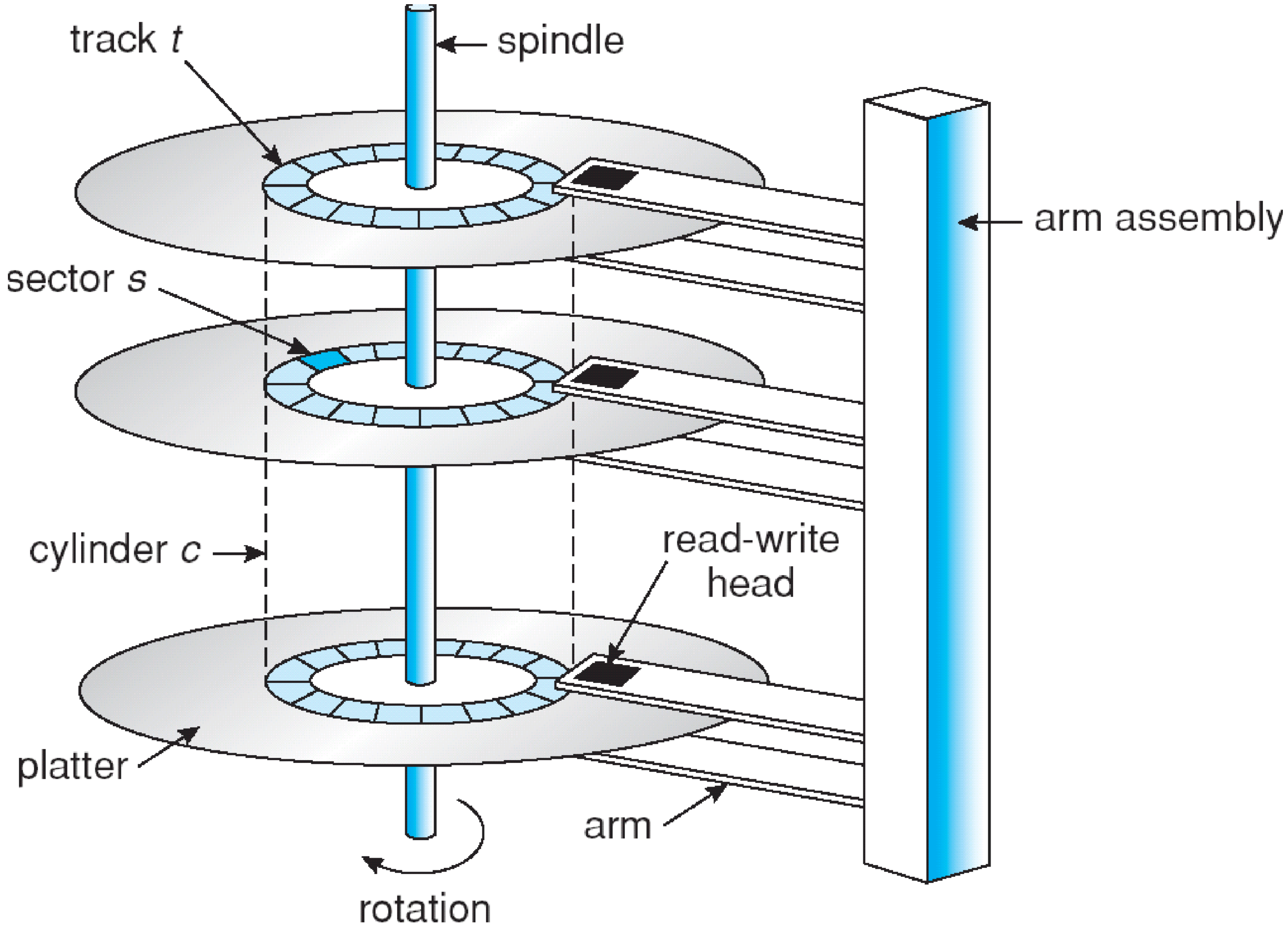
Device Registers: Status Command Data

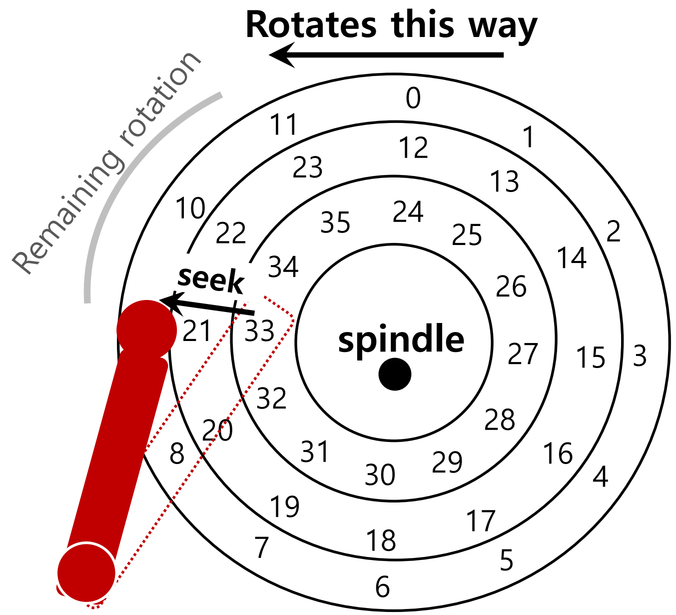
Micro-controller (CPU)
Memory (DRAM or SRAM or both)
Other Hardware-specific Chips

Canonical Device

- **Status checks:** ***polling*** **vs.** ***interrupts***
- **Data:** ***programmed I/O*** **(PIO) vs.** ***direct memory access*** **(DMA)**
- **Control:** ***special instructions*** **vs.** ***memory-mapped I/O***

Recap: I/O & Disks

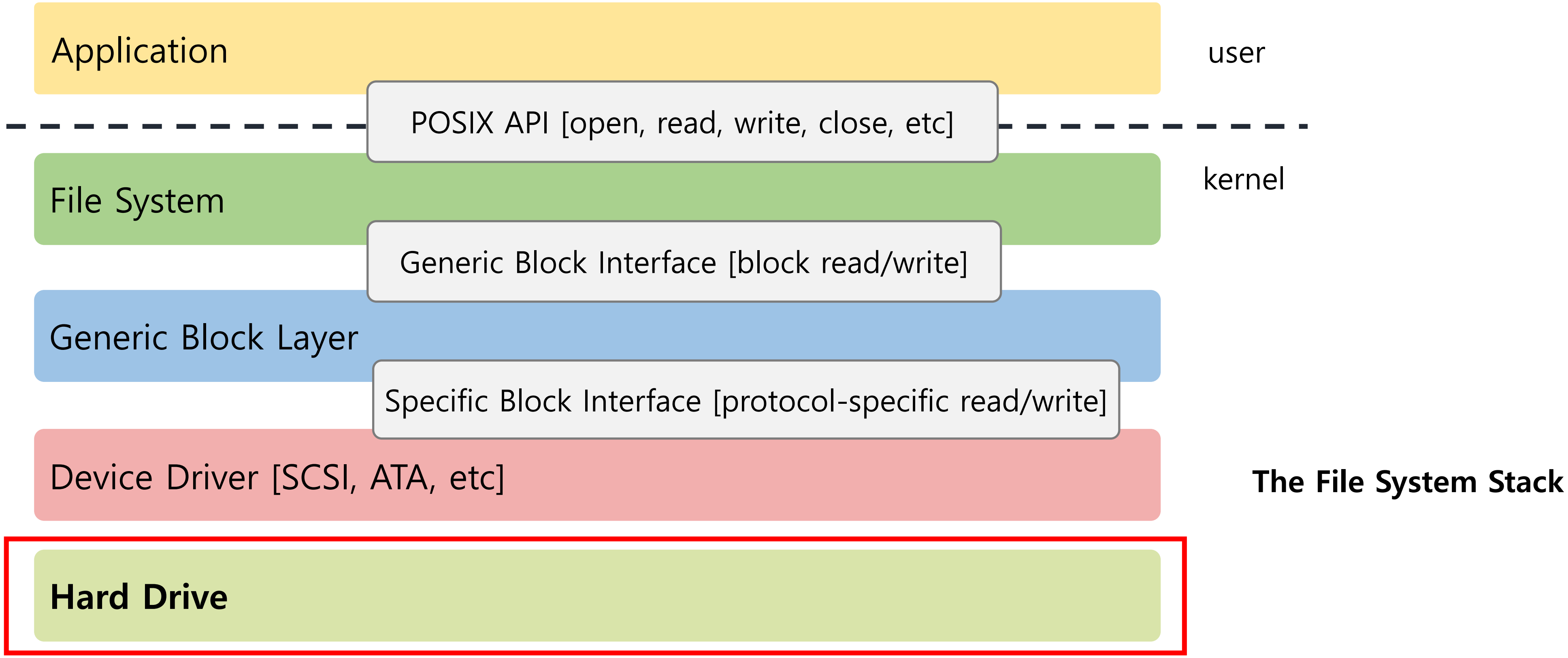




Seek, rotate, transfer

File System Abstraction

- **File system specifics of which disk class it is using.**
  - Ex) It issues **block read** and **write** request to the generic block layer.



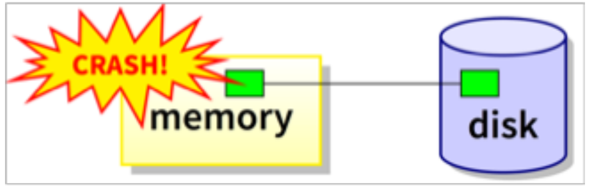
The File System Stack

File System Fun

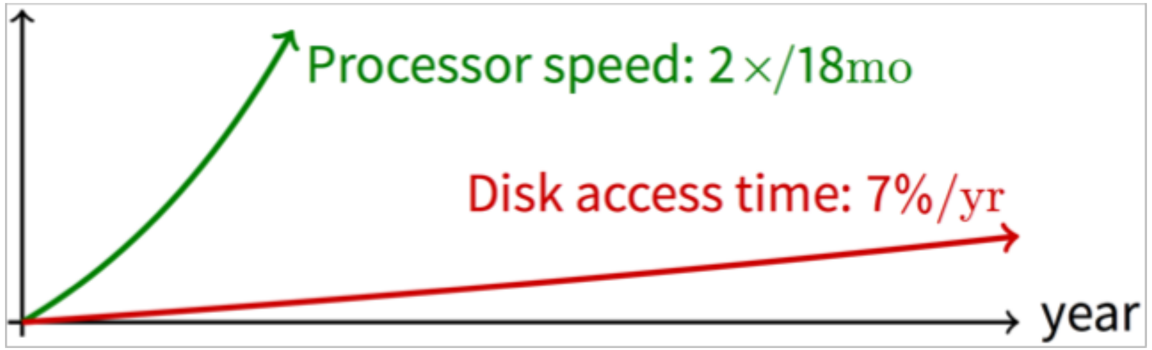
- **File systems: traditionally hardest part of OS**
  - More papers on FSes than any other single topic
- **Main tasks of file system:**
  - Don’t go away (ever)
  - Associate bytes with name (files)
  - Associate names with each other (directories)
  - Can implement file systems on disk, over network, in memory, in non-volatile ram (NVRAM), on tape, w/ paper.
  - We’ll focus on disk and generalize later
- **Today: files, directories, and a bit of performance**

Why disks are different

- **Disk = First state we've seen that doesn't go away**
  - So: Where all important state ultimately resides



- **Slow (milliseconds access vs. nanoseconds for memory)**



- **Huge (100–1,000x bigger than memory)**
  - How to organize large collection of ad hoc information?
  - File System: Hierarchical directories, Metadata, Search

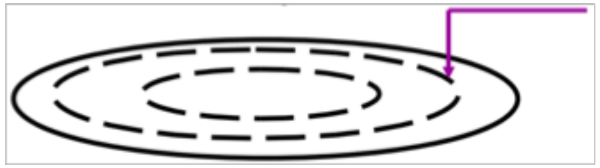
Disk vs. Memory

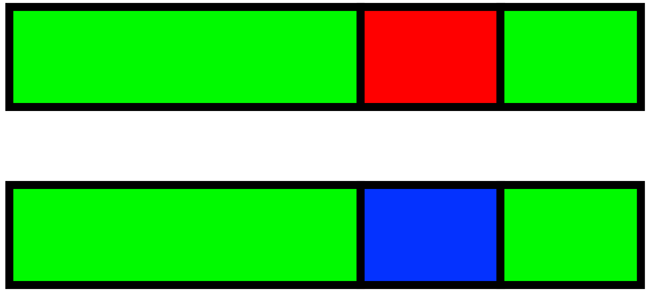
	Disk	**MLC NAND Flash**	**DRAM**
Smallest write	sector	sector	byte
Atomic write	sector	sector	byte/word
Random read	8 ms	3-10 μs	50 ns
Random write	8 ms	9-11 μs*	50 ns
Sequential read	100 MB/s	550–2500 MB/s	> 1 GB/s
Sequential write	100 MB/s	520–1500 MB/s*	> 1 GB/s
Cost	$0.03/GB	$0.35/GB	$6/GiB
Persistence	Non-volatile	Non-volatile	Volatile

*: Flash write performance degrades over time

Disk Review

- **Disk reads/writes in terms of sectors, not bytes**
  - Read/write single sector or adjacent groups





- **How to write a single byte? “Read-modify-write”**
  - Read in sector containing the byte
  - Modify that byte
  - Write entire sector back to disk
  - Key: if cached, don’t need to read in
- **Sector = unit of atomicity.**
  - Sector write done completely, even if crash in middle (disk saves up enough momentum to complete)
- **Larger atomic units have to be synthesized by OS**

Some Useful Trends (1)

- **Disk bandwidth and cost/bit improving exponentially**
  - Similar to CPU speed, memory size, etc.
- **Seek time and rotational delay improving very slowly**
  - Why? require moving physical object (disk arm)
- **Disk accesses a huge system bottleneck & getting worse**
  - Bandwidth increase lets system (pre-)fetch large chunks for about the same cost as small chunk.
  - Trade bandwidth for latency if you can get lots of related stuff.

Some Useful Trends (2)

- **Desktop memory size increasing faster than typical workloads**
  - More and more of workload fits in file cache
  - Disk traffic changes: mostly writes and new data
- **Memory and CPU resources increasing**
  - Use memory and CPU to make better decisions
  - Complex prefetching to support more IO patterns
  - Delay data placement decisions reduce random IO

Files

- **File: named bytes on disk**
  - data with some properties
  - contents, size, owner, last read/write time, protection, etc.
- **A file can also have a type**
  - Understood by the file system
    - Block, character, device, portal, link, etc.
  - Understood by other parts of the OS or runtime libraries
    - Executable, dll, source, object, text, etc.
- **A file's type can be encoded in its name or contents**
  - Windows encodes type in name
    - .com, .exe, .bat, .dll, .jpg, etc.
  - Unix encodes type in contents
    - Magic numbers, initial characters (e.g., #! for shell scripts)

Basic File Operations

Unix

- creat(name)
- open(name, how)
- read(fd, buf, len)
- write(fd, buf, len)
- sync(fd)
- seek(fd, pos)
- close(fd)
- unlink(name)

Windows

- CreateFile(name, CREATE)
- CreateFile(name, OPEN)
- ReadFile(handle, …)
- WriteFile(handle, …)
- FlushFileBuffers(handle, …)
- SetFilePointer(handle, …)
- CloseHandle(handle, …)
- DeleteFile(name)
- CopyFile(name)
- MoveFile(name)

Goal

- **Want: operations to have as few disk accesses as possible & have minimal space overhead (group related things)**

- **What's hard about grouping blocks?**

- **Like page tables, file system metadata constructs mappings**
  - Page table: map virtual page # to physical page #
  - File metadata: map byte offset to disk block address
  - Directory: map name to disk address or file #

File Systems vs. Virtual Memory

- **In both settings, want location transparency**
  - Application shouldn't care about particular disk blocks or physical memory locations
- **In some ways, FS has easier job than than VM:**
  - CPU time to do FS mappings not a big deal (why?) ➔ no TLB
  - Page tables deal with sparse address spaces and random access, files often denser (0 . . . filesize – 1), ~sequentially accessed
- **In some ways, FS's problem is harder:**
  - Each layer of translation = potential disk access
  - Space a huge premium! (But disk is huge?!?!)
    - Cache space never enough; amount of data you can get in one fetch never enough
  - Range very extreme: Many files < 10 KB, some files GB

Some Working Intuitions

- **FS performance dominated by # of disk accesses**
  - Say each access costs ~10 milliseconds
  - Touch the disk 100 times = 1 second
  - Can do a billion ALU ops in same time!
- **Access cost dominated by movement, not transfer:**
  - 1 sector: $5ms + 4ms + 5\mu s\ (\approx 512\ B/(100\ MB/s)) \approx 9ms$
  - 50 sectors: $5ms + 4ms + .25ms = 9.25ms$
  - Can get 50x the data for only ~3% more overhead!
- **Observations that might be helpful:**
  - All blocks in file tend to be used together, sequentially
  - All files in a directory tend to be used together
  - All names in a directory tend to be used together

File Access Methods

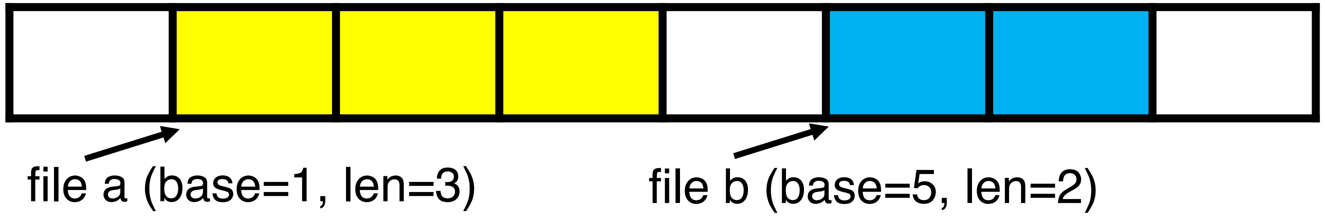
- **Sequential access**
  - read bytes one at a time, in order
  - by far the most common mode
- **Random access**
  - random access given block/byte number
- **Indexed access**
  - file system contains an index to a particular field of each record in a file
  - reads specify a value for that field and the system finds the record via the index
- **Record access**
  - file is array of fixed- or variable-length records
  - read/written sequentially or randomly by record #

Problem: How to Track File's Data

- **Disk management:**
  - Need to keep track of where file contents are on disk
  - Must be able to use this to map byte offset to disk block
  - Structure tracking a file's sectors is called an index node or inode
  - *inodes* must be stored on disk, too
- **Things to keep in mind while designing file structure:**
  - Most files are small
  - Much of the disk is allocated to large files
  - Many of the I/O operations are made to large files
  - Want good sequential and good random access (what do these require?)

Straw Man: Contiguous Allocation

- **“Extent-based”: allocate files like segmented memory**
  - When creating a file, make the user pre-specify its length and allocate all space at once
  - Inode contents: location and size

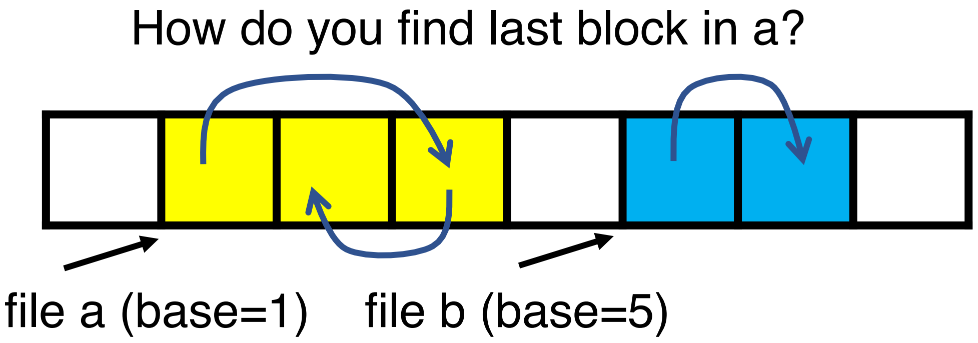


What happens if file c needs 2 sectors?

- **Example: IBM OS/360**
- **Pros?**
  - Simple, fast access, both sequential and random
- **Cons? (Think of corresponding VM scheme)**
  - External fragmentation

Straw Man #2: Linked Files

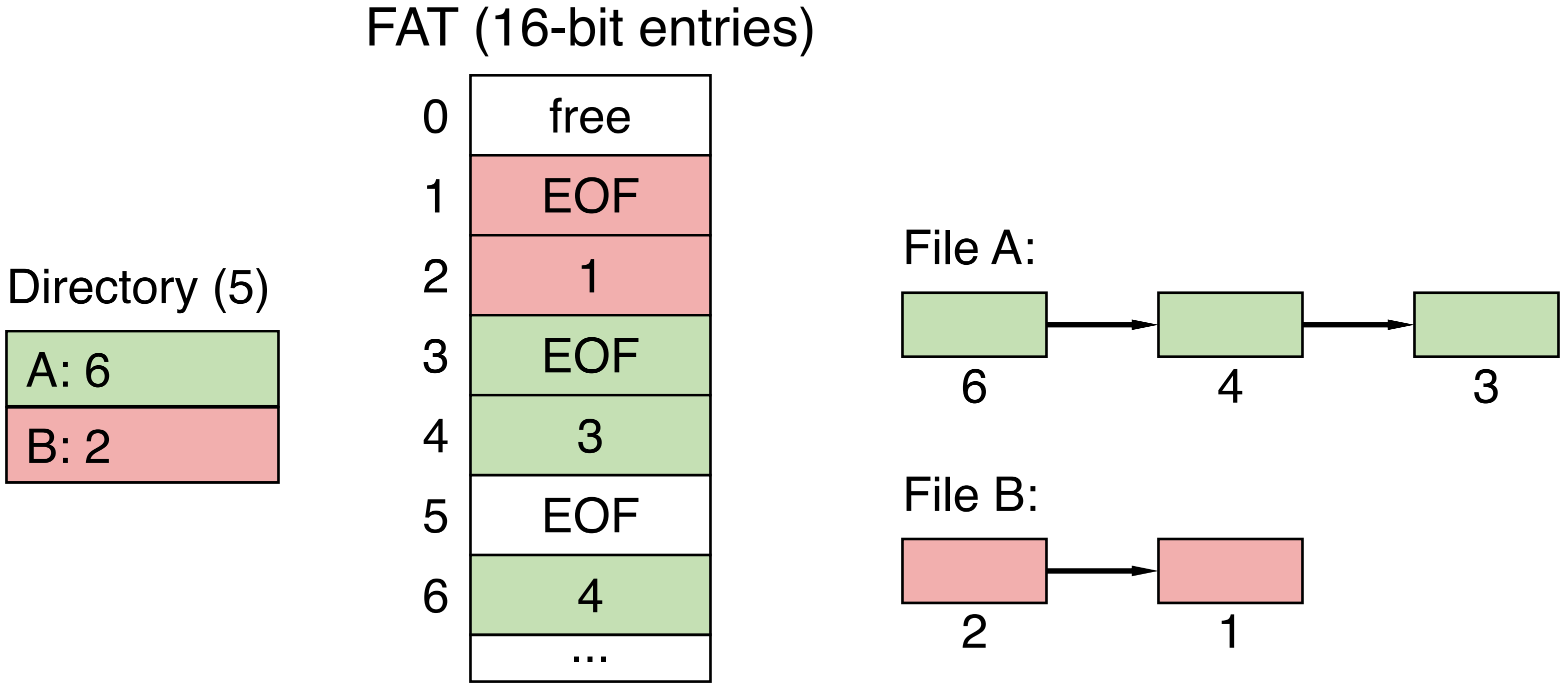
- **Basically a linked list on disk.**
  - Keep a linked list of all free blocks
  - Inode contents: a pointer to file's first block
  - In each block, keep a pointer to the next one



- **Examples (sort-of): Alto, TOPS-10, DOS FAT**
- **Pros?**
  - Easy dynamic growth & sequential access, no fragmentation
- **Cons?**
  - Linked lists on disk a bad idea because of access times
  - Random very slow (e.g., traverse whole file to find last block)
  - Pointers take up room in block, skewing alignment

Example: DOS FS (simplified)

- **Linked files with key optimization: *puts links in fixed-size "file allocation table" (FAT) rather than in the blocks*.**



- **Still do pointer chasing, but can cache entire FAT so can be cheap compared to disk access**

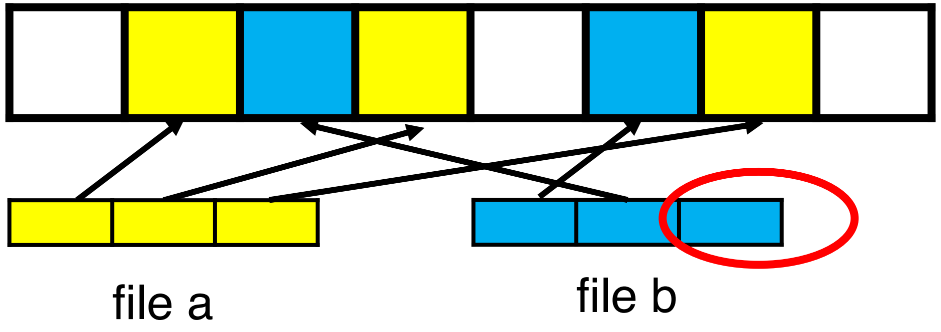
FAT Discussion

- **Entry size = 16 bits (initial FAT16 in MS-DOS 3.0)**
  - What's the maximum size of the FAT? 65,536 entries
  - Given a 512 byte block, what's the maximum size of FS? 32MiB
  - One solution: go to bigger blocks. Pros? Cons?
- **Space overhead of FAT is trivial:**
  - 2 bytes / 512 byte block = ~ 0.4% (Compare to Unix)
- **Reliability: how to protect against errors?**
  - Create duplicate copies of FAT on disk
  - State duplication a very common theme in reliability
- **Bootstrapping: where is root directory?**
  - Fixed location on disk:

FAT	FAT (opt)	Root dir	…

Another Approach: Indexed Files

- **Each file has an array holding all of its block pointers**
  - Just like a page table, so will have similar issues
  - Max file size fixed by array's size (static or dynamic?)
  - Allocate array to hold file's block pointers on file creation
  - Allocate actual blocks on demand using free list



- **Pros?**
  - Both sequential and random access easy
- **Cons?**
  - Mapping table requires large chunk of contiguous space
  - ...Same problem we were trying to solve initially

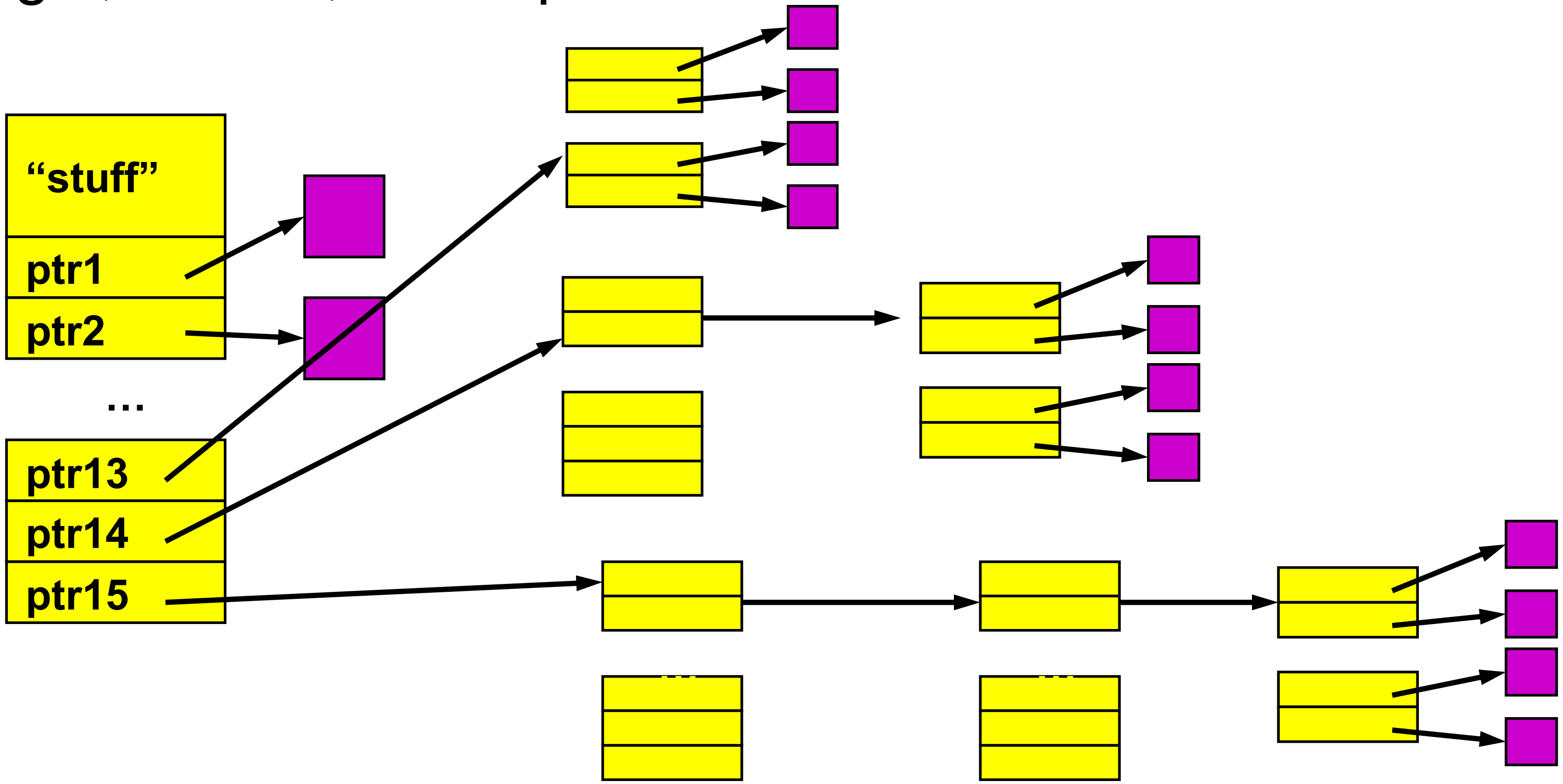
Indexed Files

- **Issues same as in page tables**
  - Large possible file size = lots of unused entries
  - Large actual size? table needs large contiguous disk chunk

- **Solve identically: small regions with index array, this array with another array, ... Downside?**

Multi-level Indexed Files: Unix inodes

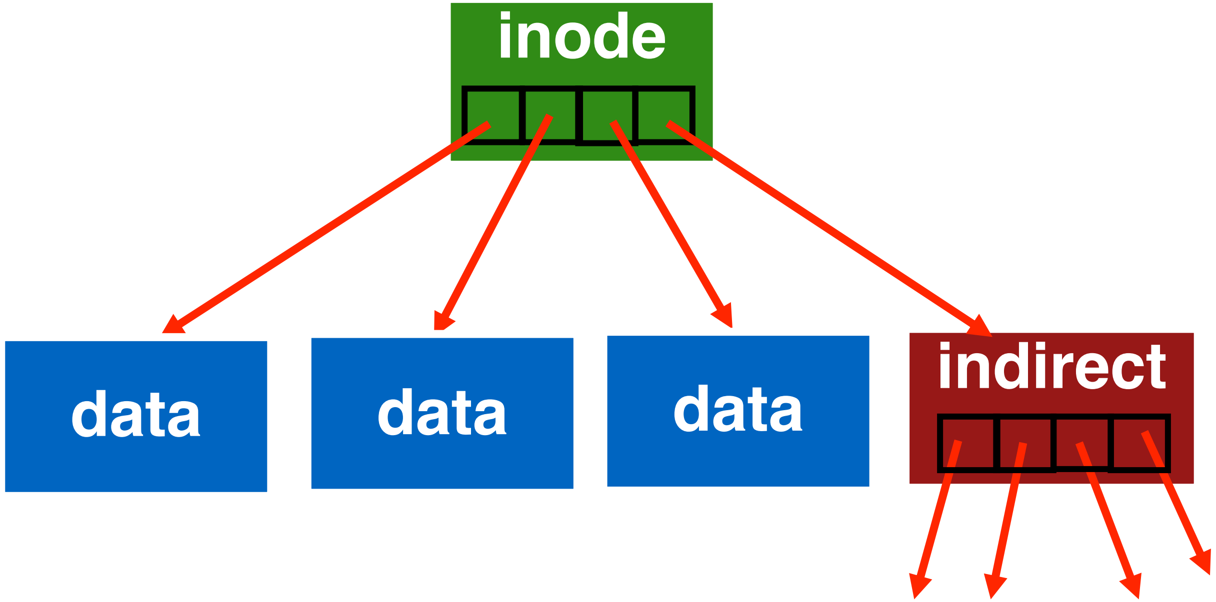
- **inode = 15 block pointers + "stuff"**
  - first 12 are direct blocks: solve problem of first blocks access slow
  - then single, double, and triple indirect block



More About inode

type (file or dir?)
uid (owner)
rwx (permissions)
size (in bytes)
blocks
time (access)
ctime (create)
links_count (# paths)
addrs[N] (N data blocks)

inode

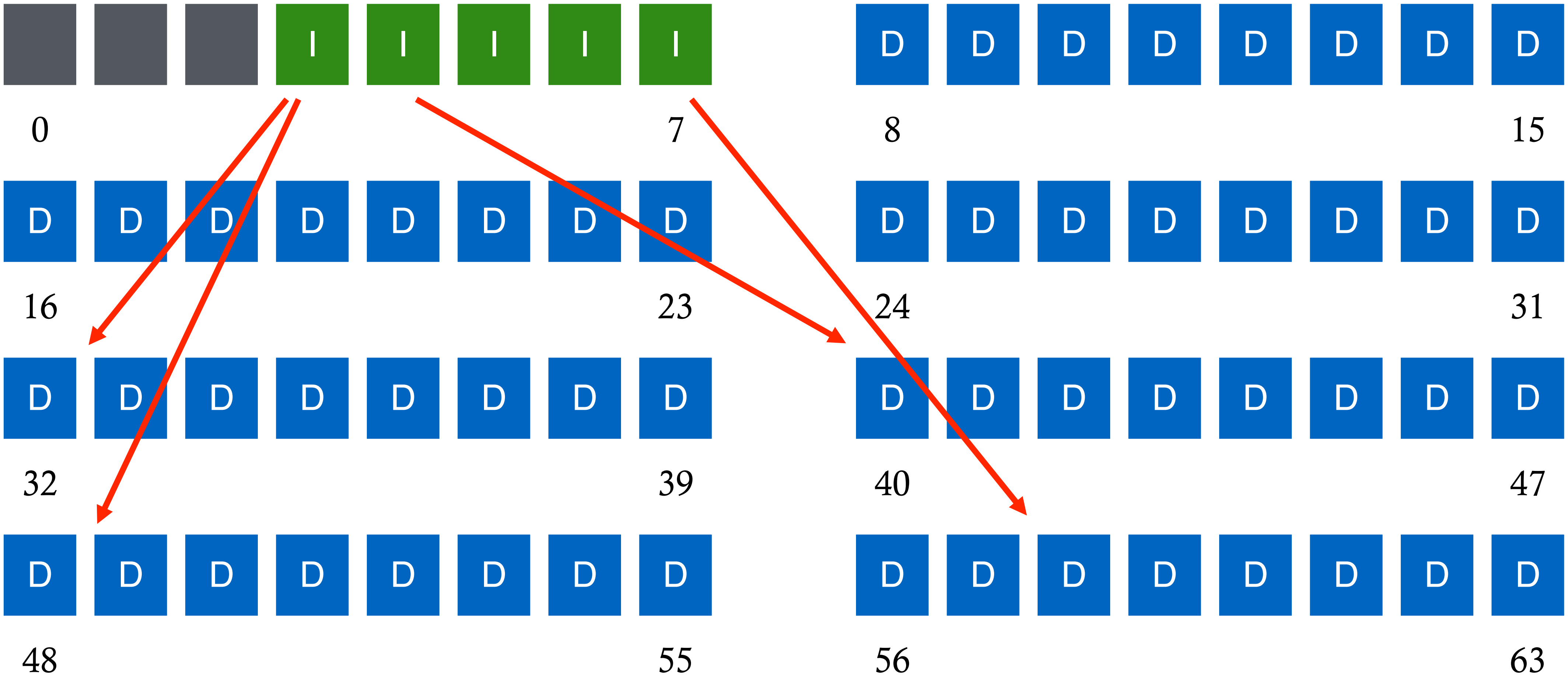


More About inodes

- **inodes are stored in a fixed-size array**
  - Size of array fixed when disk is initialized; can't be changed
  - Lives in known location, originally at one side of disk:

  - The *index* of an inode in the inode array called an i-number
  - Internally, the OS refers to files by *i-number*
  - When file is opened, inode brought in memory
  - Written back when modified and file closed or time elapses

More About inodes



Directories

- **Problem:**
  - "Spend all day generating data, come back the next morning, want to use it." – F. Corbato, on why files/dirs invented
- **Approach 0: Users remember where on disk their files are**
  - E.g., like remembering your social security or bank account #
- **Yuck. People want human digestible names**
  - We use directories to map names to file blocks
- **Directories serve two purposes**
  - For users, they provide a structured way to organize files
  - For FS, they provide a convenient naming interface that allows the separation of logical file organization from physical file placement on the disk

Basic Directory Operations

Unix

- **Directories implemented in files**
  - Use file ops to create dirs
- **C runtime library provides a higher-level abstraction for reading directories**
  - opendir(name)
  - readdir(DIR)
  - seekdir(DIR)
  - closedir(DIR)

Windows

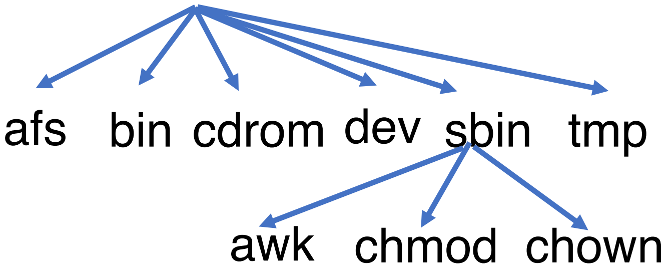
- **Explicit dir operations**
  - CreateDirectory(name)
  - RemoveDirectory(name)
- **Very different method for reading directory entries**
  - FindFirstFile(pattern)
  - FindNextFile()

A Short History of Directories

- **Approach 1: Single directory for entire system**
  - Put directory at known location on disk
  - Directory contains hname, inumberi pairs
  - If one user uses a name, no one else can
  - Many ancient personal computers work this way
- **Approach 2: Single directory for each user**
  - Still clumsy, and ls on 10,000 files is a real pain
- **Approach 3: Hierarchical name spaces**
  - Allow directory to map names to files or other dirs
  - File system forms a tree (or graph, if links allowed)
  - Large name spaces tend to be hierarchical (ip addresses, domain names, scoping in programming languages, etc.)

Hierarchical Directory

- **Used since CTSS (1960s)**
  - Unix picked up and used really nicely
- **Directories stored on disk just like regular files**
  - Special inode type byte set to directory
  - User's can read just like any other file
  - Only special syscalls can write (why?)
  - Inodes at fixed disk location
  - File pointed to by the index may be another directory
  - Makes FS into hierarchical tree
- **Simple, plus speeding up file ops speeds up dir ops!**



<name,inode#>

<afs,1021>
<tmp,1020>
<bin,1022>
<cdrom,4123>
<dev,1001>
<sbin,1011>
…

Naming Magic

- **Bootstrapping: Where do you start looking?**
  - Root directory always inode #2 (0 and 1 historically reserved)

- **Special names:**
  - Root directory: “/”
  - Current directory: “.”
  - Parent directory: “..”

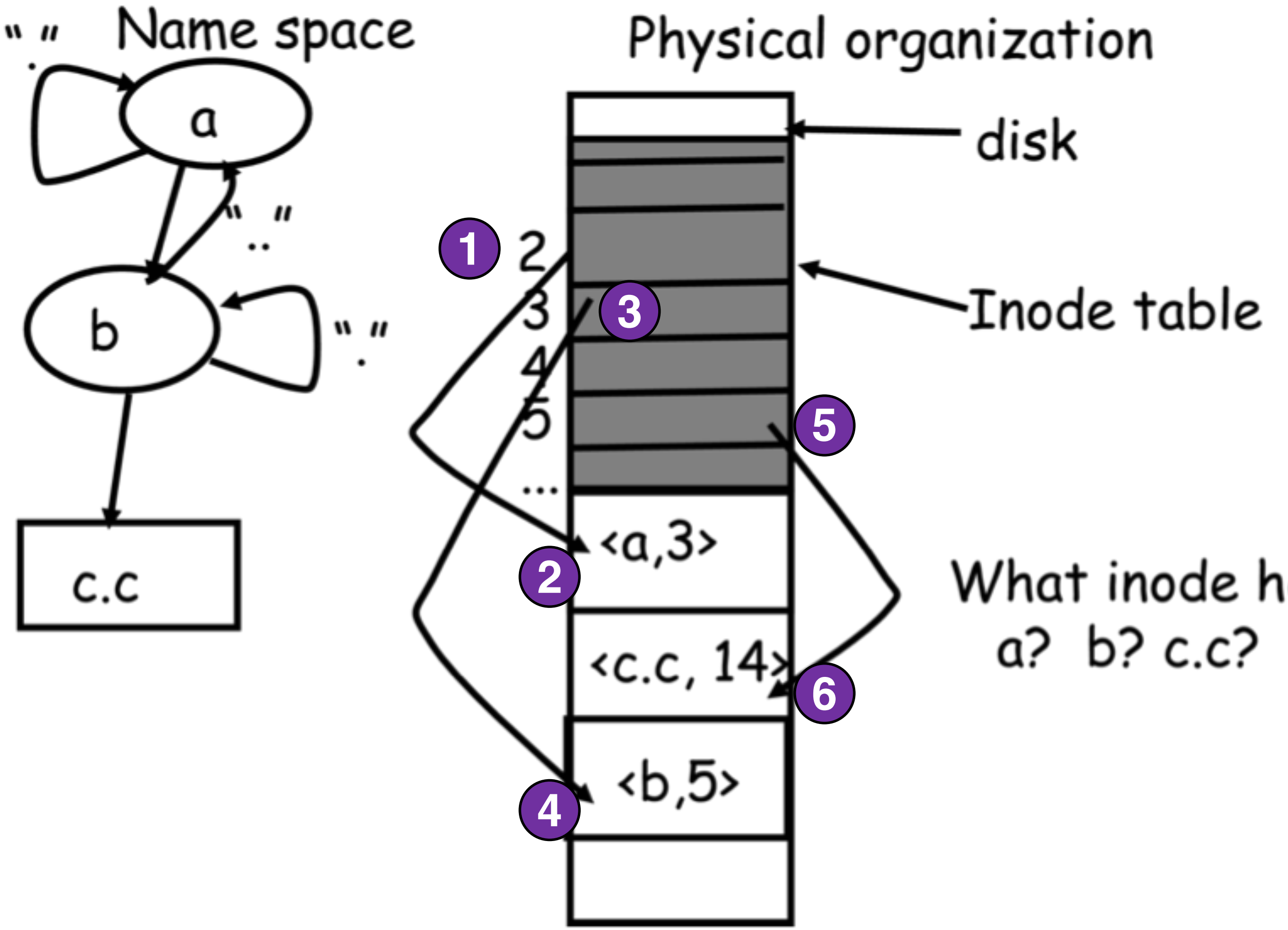
- **Some special names are provided by shell, not FS:**
  - User’s home directory: “~”
  - Globbing: “foo.*” expands to all files starting “foo.”

- **Using the given names, only need two operations to navigate the entire name space:**
  - cd name: move into (change context to) directory name
  - ls: enumerate all names in current directory (context)

Unix inodes and Path Search

- **Unix inodes are not directories**
  - Inodes describe where on the disk the blocks for a file are placed
  - Directories are files, so inodes also describe where the blocks for directories are placed on the disk
- **Directory entries map file names to inodes**
  - To open “/one”, use Master Block to find inode for “/” on disk
  - Open “/”, look for entry for “one”
  - This entry gives the disk block number for the inode for “one”
  - Read the inode for “one” into memory
  - The inode says where first data block is on disk
  - Read that block into memory to access the data in the file

Unix Example: /a/b/c.c



What inode holds file for a? b? c.c?

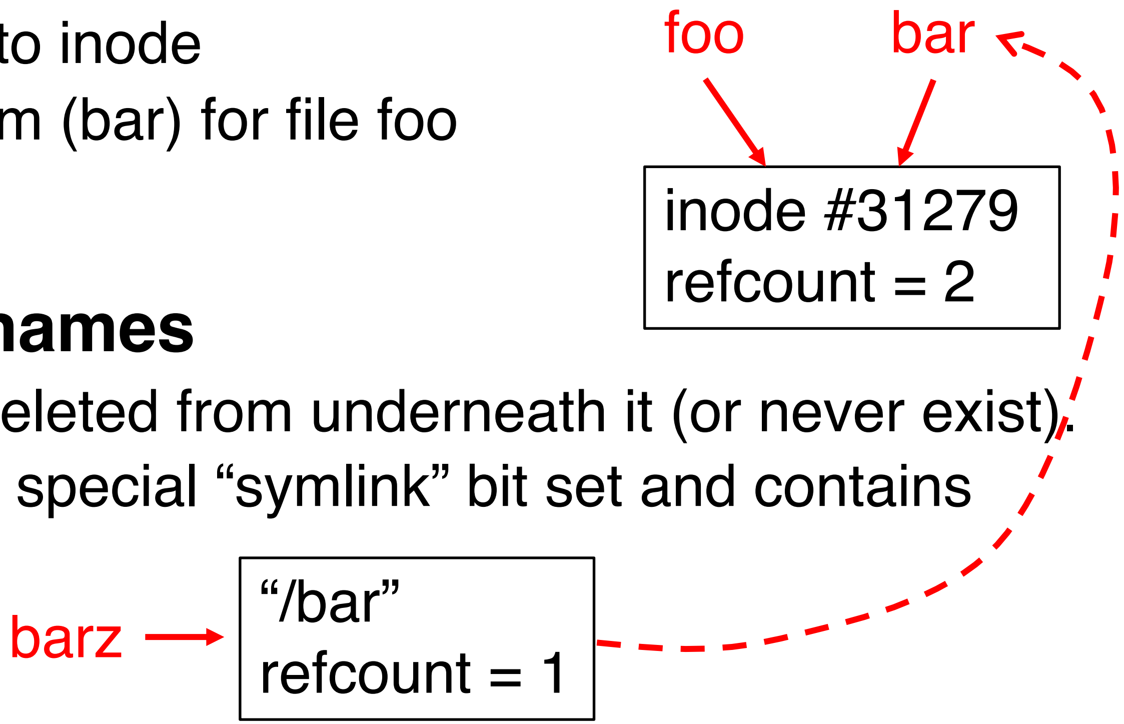
Default Context: Working Directory

- **Cumbersome to constantly specify full path names**
  - In Unix, each process has a “current working directory” (cwd)
  - File names *not* beginning with “/” are assumed to be relative to cwd; otherwise translation happens as before
- **Shells track a default list of active contexts**
  - A “search path” for programs you run
  - Given a search path A:B:C, the shell will check in A, then B, then C
  - Can escape using explicit paths: “./foo”
- **Example of locality**

Hard and Soft Links (synonyms)

- **More than one dir entry can refer to a given file**
  - Unix stores count of pointers ("hard links") to inode
  - To make: "`ln foo bar`" creates a synonym (bar) for file foo
- **Soft/symbolic links = synonyms for names**
  - Point to a file/dir name, but object can be deleted from underneath it (or never exist).
  - Unix implements like directories: inode has special "symlink" bit set and contains name of link target

```
ln –s /bar barz
```



  - When the file system encounters a soft link it automatically translates it (if possible).

File Buffer Cache

- **Applications exhibit significant locality for reading and writing files**

- **Idea: Cache file blocks in memory to capture locality**
  - Called the file buffer cache
  - Cache is system wide, used and shared by all processes
  - Reading from the cache makes a disk perform like memory
  - Even a small cache can be very effective

- **Issues**
  - The file buffer cache competes with VM (tradeoff here)
  - Like VM, it has limited size
  - Need replacement algorithms again (LRU usually used)

Caching Writes

- **On a write, some applications assume that data makes it through the buffer cache and onto the disk**
  - As a result, writes are often slow even with caching
- **OSes typically do write back caching**
  - Maintain a queue of uncommitted blocks
  - Periodically flush the queue to disk (30 second threshold)
  - If blocks changed many times in 30 secs, only need one I/O
  - If blocks deleted before 30 secs (e.g., /tmp), no I/Os needed
- **Unreliable, but practical**
  - On a crash, all writes within last 30 secs are lost
  - Modern OSes do this by default; too slow otherwise
  - System calls (Unix: fsync) enable apps to force data to disk

Read Ahead

- **Many file systems implement "read ahead"**
  - FS predicts that the process will request next block
  - FS goes ahead and requests it from the disk
  - This can happen while the process is computing on previous block
    - Overlap I/O with execution
  - When the process requests block, it will be in cache
  - Compliments the disk cache, which also is doing read ahead
- **For sequentially accessed files can be a big win**
  - Unless blocks for the file are scattered across the disk
  - File systems try to prevent that, though (during allocation)

File Sharing

- **File sharing has been around since timesharing**
  - Easy to do on a single machine
  - PCs, workstations, and networks get us there (mostly)
- **File sharing is important for getting work done**
  - Basis for communication and synchronization
- **Two key issues when sharing files**
  - Semantics of concurrent access
    - What happens when one process reads while another writes?
    - What happens when two processes open a file for writing?
    - What are we going to use to coordinate?
  - Protection

Protection

- **File systems implement a protection system**
  - Who can access a file
  - How they can access it
- **More generally…**
  - Objects are "what", subjects are "who", actions are "how"
- **A protection system dictates whether a given action performed by a given subject on a given object should be allowed**
  - You can read and/or write your files, but others cannot
  - You can read "/etc/motd", but you cannot write it

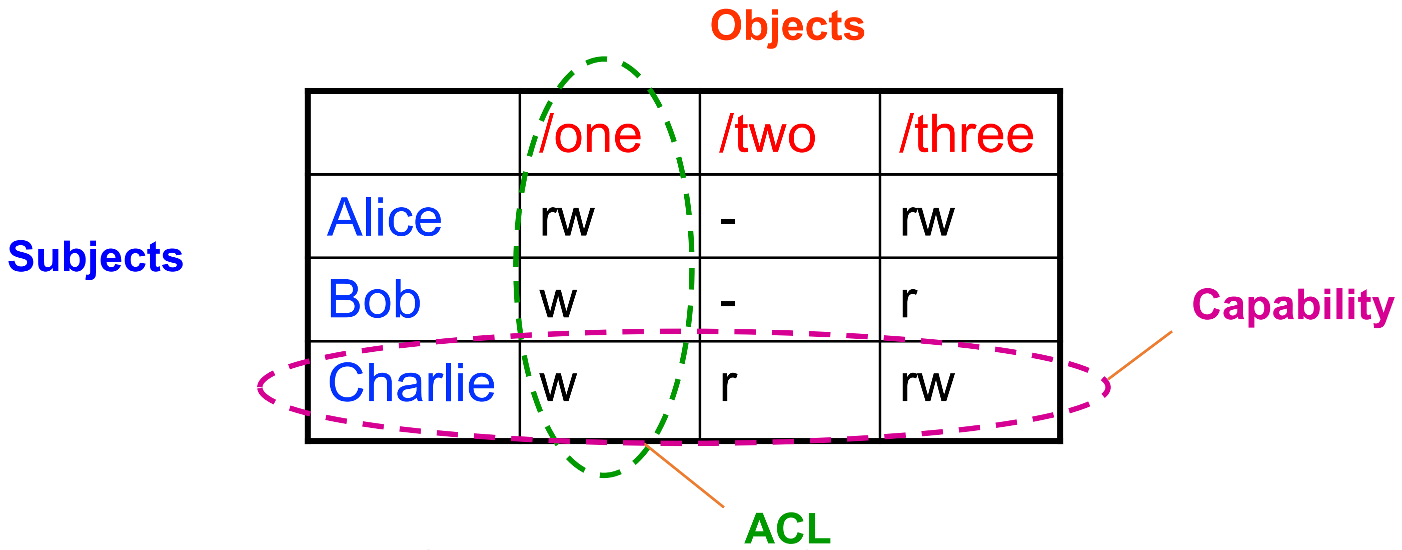
Representing Protection

Access Control Lists (ACL)

- **For each object, maintain a list of subjects and their permitted actions**

Capabilities

- **For each subject, maintain a list of objects and their permitted actions**



Objects

Subjects

	/one	/two	/three
Alice	rw	-	rw
Bob	w	-	r
Charlie	w	r	rw

Capability

ACL

ACLs and Capabilities

- **Approaches differ only in how the table is represented**
  - What approach does Unix use in the FS?
- **Capabilities are easier to transfer**
  - They are like keys, can handoff, does not depend on subject
- **In practice, ACLs are easier to manage**
  - Object-centric, easy to grant, revoke
  - To revoke capabilities, have to keep track of all subjects that have the capability – a challenging problem
- **ACLs have a problem when objects are heavily shared**
  - The ACLs become very large
  - Use groups (e.g., Unix)

Summary

- **Files**
  - Operations, access methods
- **Directories**
  - Operations, using directories to do path searches
- **File System Layouts**
  - Unix inodes
- **File Buffer Cache**
  - Strategies for handling writes
- **Read Ahead**
- **Sharing**
- **Protection**
  - ACLs vs. capabilities

Next Time…

- **Read Chapter 41, 42**