CE 440 Introduction to Operating System

Lecture 16: File System
Fall 2025

Prof. Yigong Hu

BOSTON UNIVERSITY

Slides courtesy of Manuel Egele, Ryan Huang and Baris Kasikci

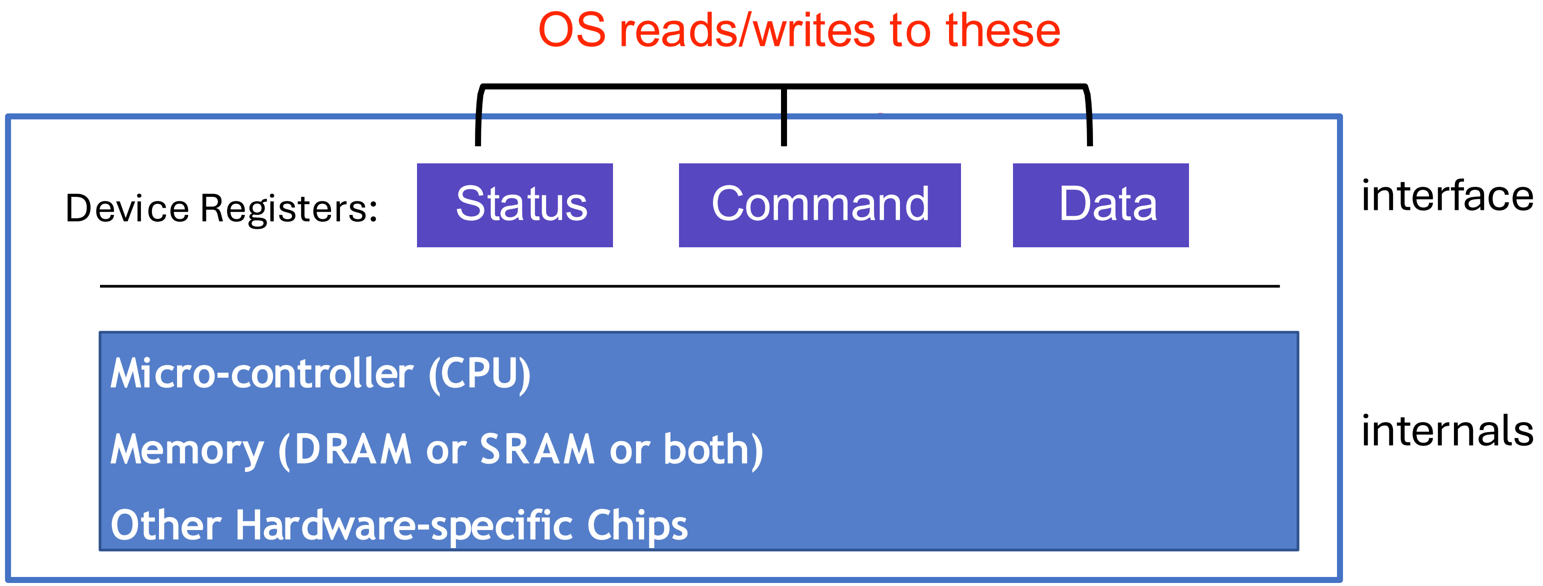
Administrivia

Lab 3a and 3b is out

- **Start the project early**

Homework 4 is out

Recap: I/O & Disks



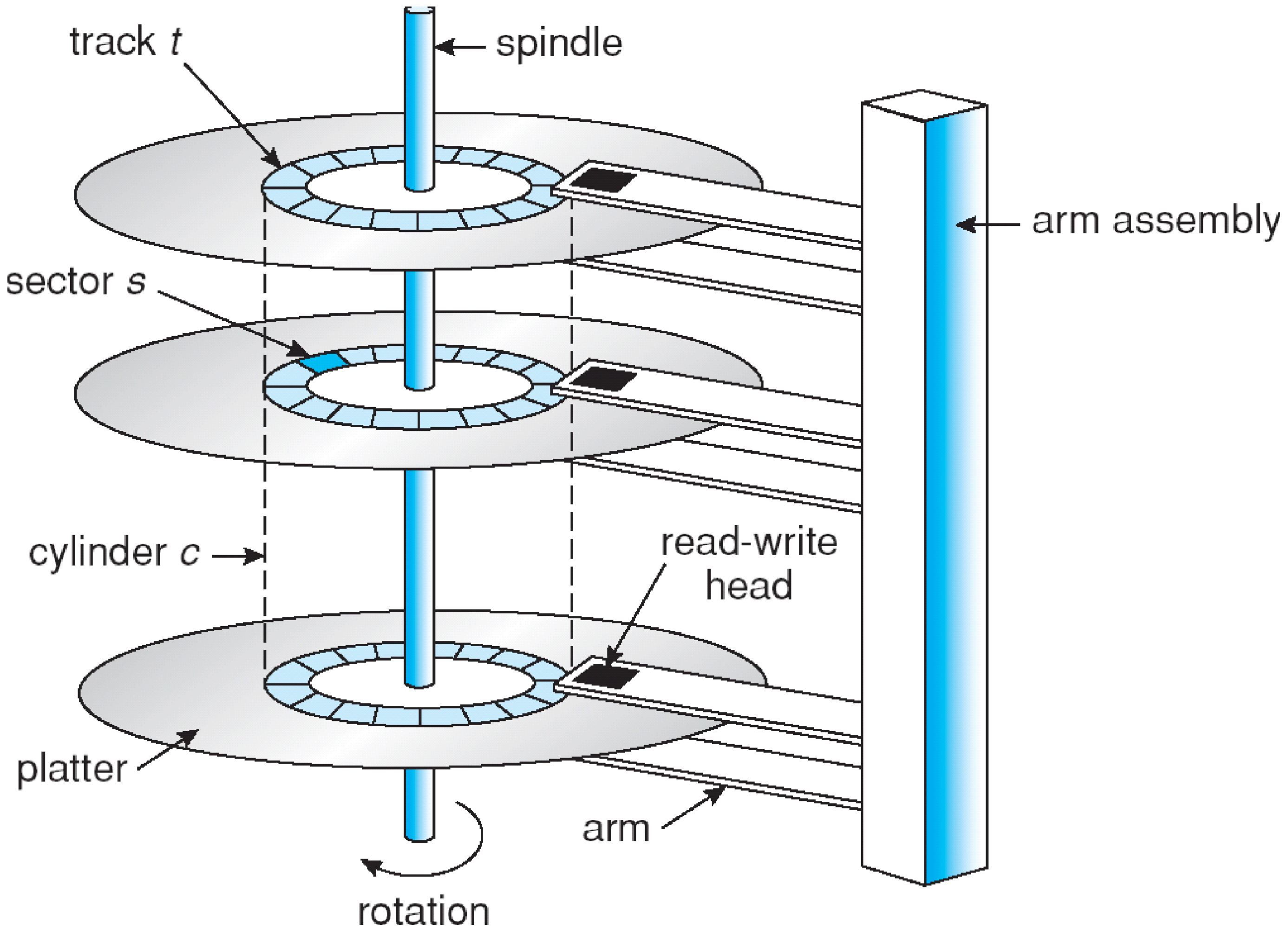
Canonical I/O Device

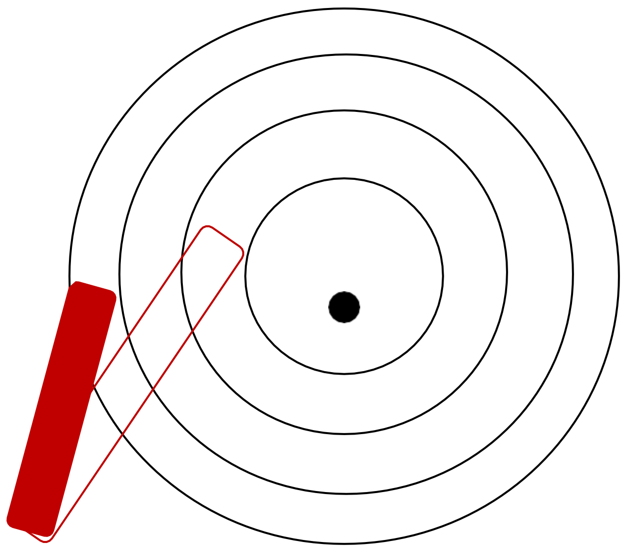
Status checks: ***polling*** **vs.** ***interrupts***

Command: ***special instructions*** **vs.** ***memory-mapped I/O***

Data: ***programmed I/O*** **(PIO) vs.** ***direct memory access*** **(DMA)**

Recap: I/O & Disks





Seek, rotate, transfer

File System ~~Not~~ Fun

File systems: a challenging OS design topic

- More papers on FSes than any other single topic

Main tasks of file system:

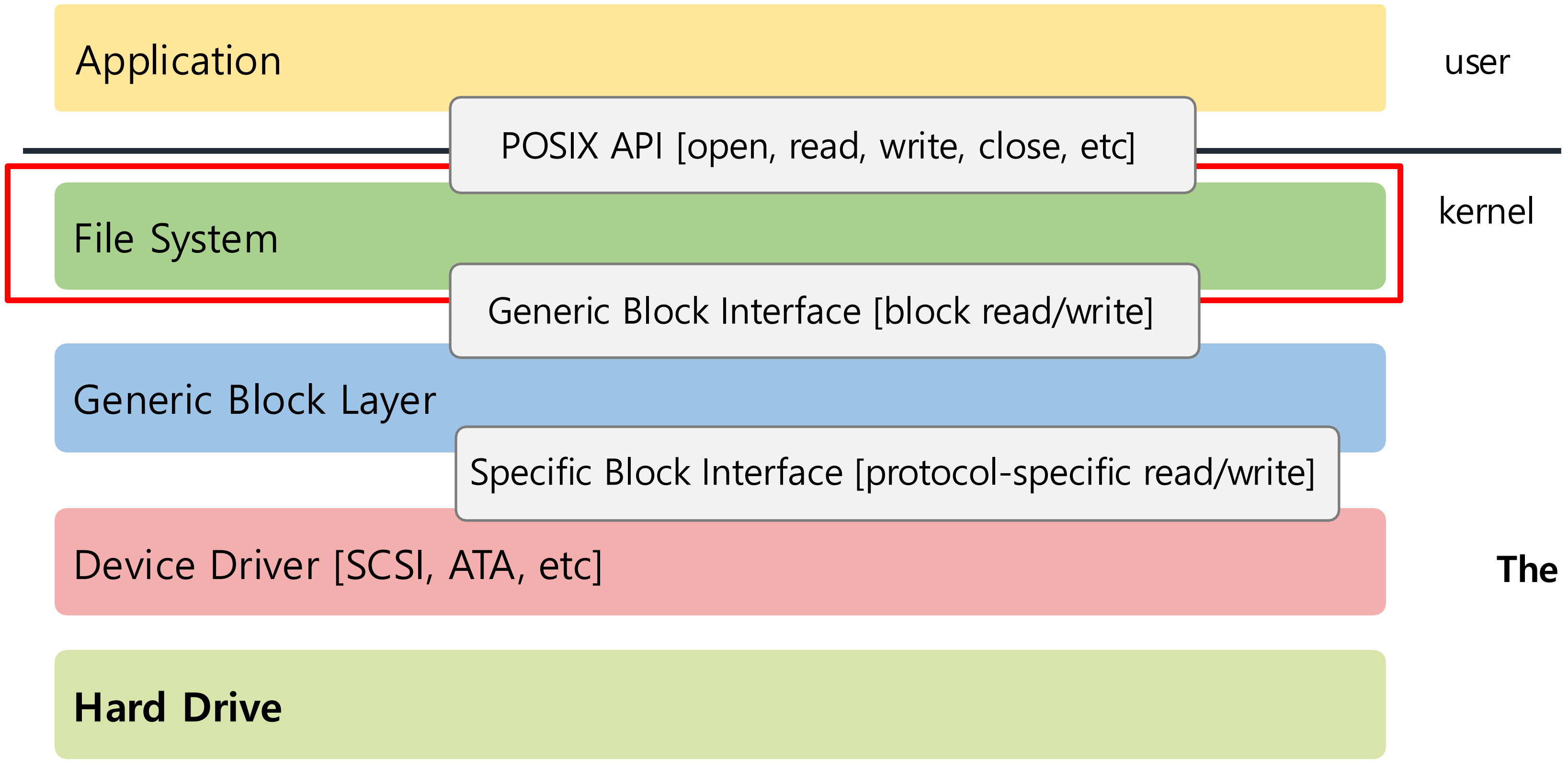
- Don't go away (ever)
- Associate bytes with name (files)
- Associate names with each other (directories)
- Can implement file systems on disk, over network, in memory, in non-volatile
- ram (NVRAM), on tape, w/ paper.
- We'll focus on disk and generalize later

Today: files, directories

File System Abstraction

File system specifics of which disk class it is using.

- It issues **block read** and **write** request to the generic block layer.



The File System Stack

Files

File: named bytes on disk

- Data with some properties
- Contents, size, owner, last read/write time, protection, etc.

How is a file’s data managed by the file system?

- Next lecture’s topic
- Basic idea (in Unix): a struct called an index node or inode
  - Describe where on the disk the blocks for a file are placed
  - Disk stores an array of inodes, inode # is the index in this array

File Types

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)

File Access Methods

FS usually provides different 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
- Record access
  - file is array of fixed- or variable-length records
  - read/written sequentially or randomly by record #
- 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

What file access method does Unix, Windows provide?

Directories

Problem: referencing files

Users remember where on disk their files are (disk sector no.)?...

- E.g., like remembering your social security or bank account #

...People want human digestible names

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

A Short History of Directories

Approach 1: Single directory for entire system

- Put directory at known disk location. 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 running `ls` on 10,000 files is a real pain

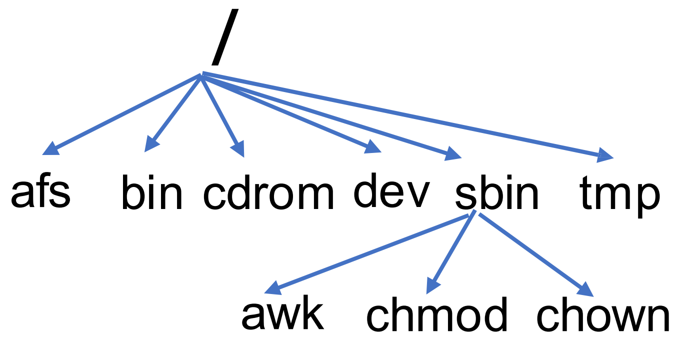
Approach 3: Hierarchical name spaces

- Allow directory to map names to files or other dirs
- File system forms a tree (or graph, if links allowed)

Hierarchical Directory

Used since CTSS (1960s)

- Unix picked up and used really nicely



Large name spaces tend to be hierarchical

- ip addresses, domain names, scoping in programming languages, etc.

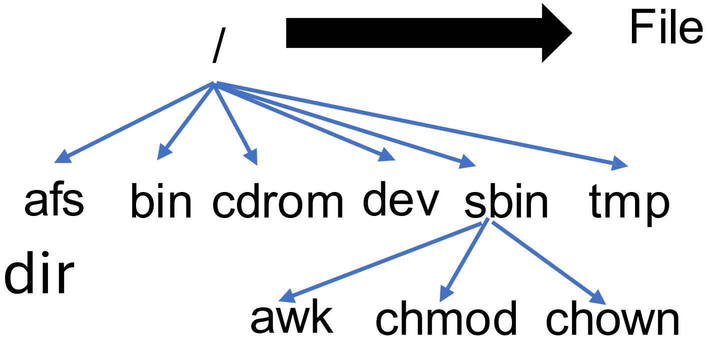
Directory Internals

A directory is a list of entries

- <name, location> tuple, location is typically the *inode #* (more next lecture)
- An inode describes where on the disk the blocks for a file are placed

Directories stored on disk just like regular files

- File type set to directory
- User's can read just like any other file
- Only special syscalls can write (why?)
- File pointed to by the location may be another dir
- Makes FS into hierarchical tree



File content for '/'

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

Simple, plus speeding up file ops speeds up dir ops!

Path Name Translation

Let’s say you want to open “/one/two/three.txt”

What does the file system do?

- Directory entries map file names to location (inode #)
- Open directory “/”: Where? Root directory is always inode #2
- Search for the entry “one”, get location of “one” (in dir entry)
- Open directory “one”, search for “two”, get location of “two”
- Open directory “two”, search for “three”, get location of “three”
- Open file “three”

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)

Basic Directory Operations

Unix

Directories implemented in files

- Use file ops to create dirs.

C library provides a higher-level abstraction for reading directories

- `opendir(name)`
- `readdir(DIR)`
- `seekdir(DIR)`
- `closedir(DIR)`

Windows

Explicit directory operations

- `CreateDirectory(name)`
- `RemoveDirectory(name)`

Very different method for reading directory entries

- `FindFirstFile(pattern)`
- `FindNextFile()`

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

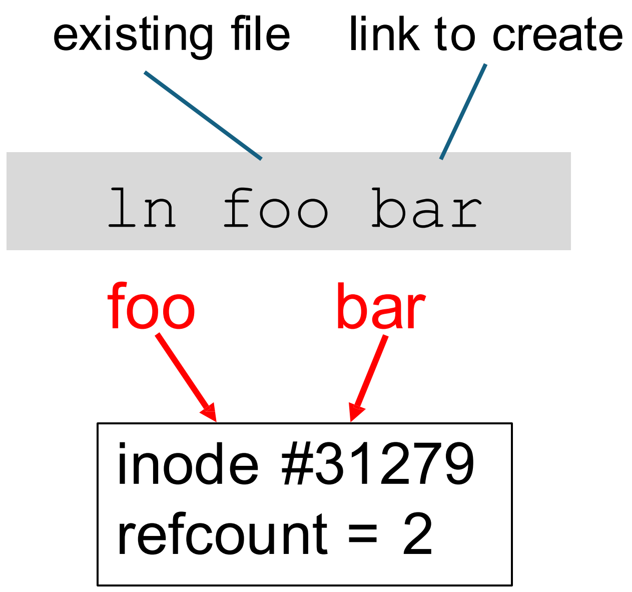
- A "search path" for programs you run
- Given a search path A:B:C, the shell will check in A, then B, then C
- Can escape using explicit paths: "./foo"

Example of locality

Hard Links

More than one dir entry can refer to a given file

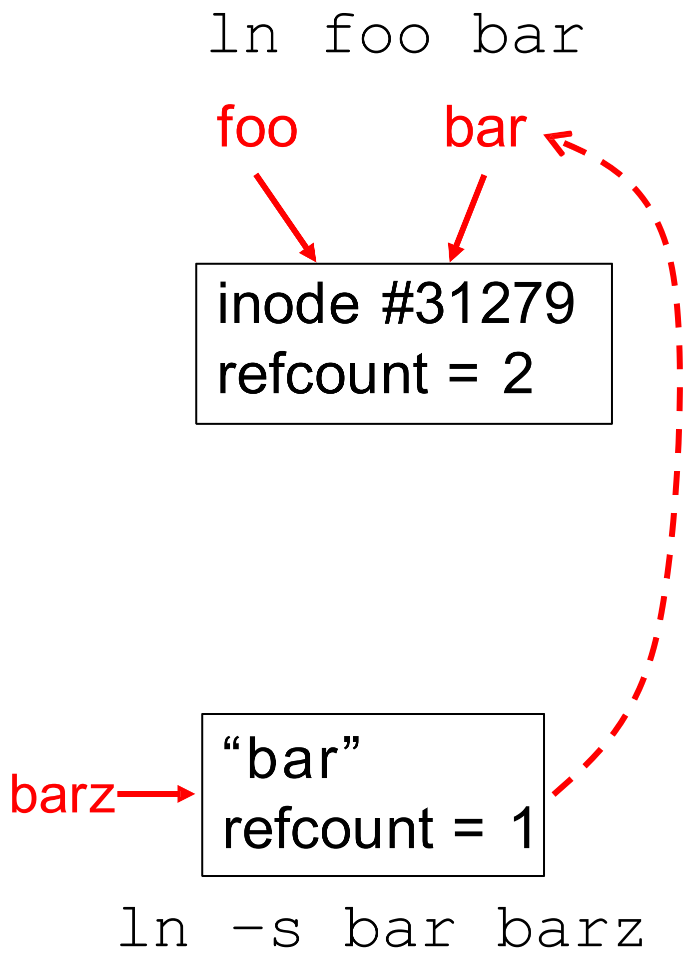
- Hard link creates a synonym for file
- Unix stores count of pointers (“hard links”) to inode
- If one of the links is removed (e.g., rm), the data are still accessible through any other link that remains
- If all links are removed, the space occupied by the data is freed.



Soft Links

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



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

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)

Unix File Protection

What approach does Unix use in the FS?

- Answer: both

ACL: Unix file permissions

Capability: file descriptors

How are they used together?

- Conversion through open()system call

Converted to capability

ACL check, expensive

```
int fd = open("file.txt", O_WRONLY);
if (fd == -1)
  exit(-1);

for (int i = 0; i < 100; i++)
  write(fd, buf + i * 4, 4);
```

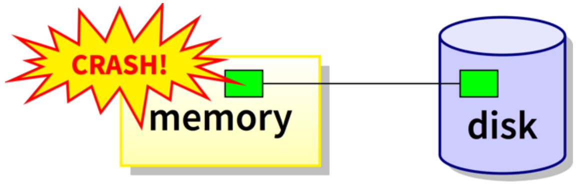
Use capability from then on

Overview

- File System Abstraction

- **File System Implementation**

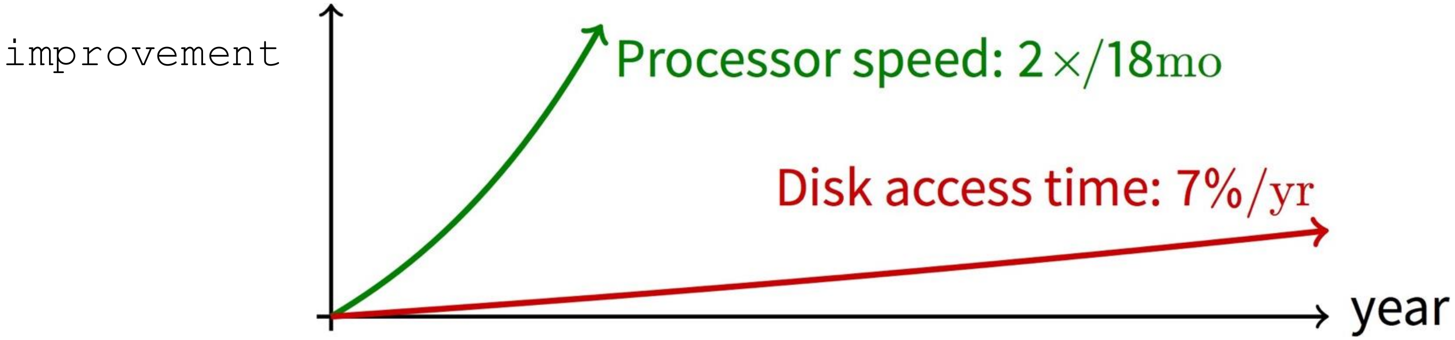
Why Disks Are Different



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

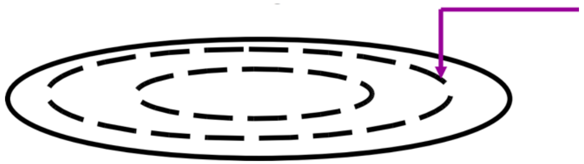
	Disk	MLC NAND Flash	DRAM
Smallest write	sector	sector	byte
Atomic write	sector	sector	byte/word
Random read	8 ms	3-10 μs	50 ns
Random write	8 ms	9-11 μs*	50 ns
Sequential read	100 MB/s	550–2500 MB/s	> 1 GB/s
Sequential write	100 MB/s	520–1500 MB/s*	> 1 GB/s
Cost	$0.03/GB	$0.35/GB	$6/GiB
Persistence	Non-volatile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

*: Flash write performance degrades over time

Disk Review

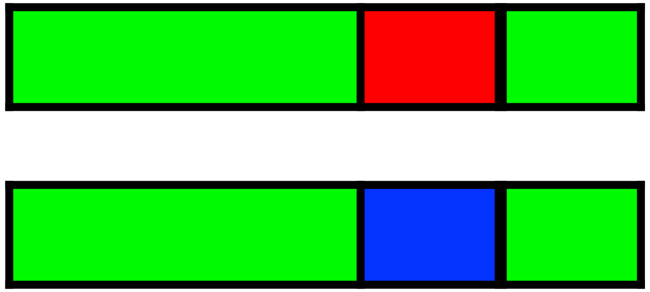
Disk reads/writes in terms of sectors, not bytes

- 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 access is 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

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 VM:

- CPU time to do FS mappings not a big deal (why?) →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 + 5μs (≈ 512 B/(100 MB/s)) ≈ 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

Summary

Files

- Operations, access methods

Directories

- Operations, using directories to do path searches
-

Sharing

Protection

- ACLs vs. capabilities

Next Chapter

Read Chapter 40, 41