

CSE 120

Principles of Operating Systems

Spring 2016

Lecture 12: File Systems

File Systems

- First we'll discuss properties of physical disks
 - ◆ Structure
 - ◆ Performance
 - ◆ Scheduling
- Then we'll discuss how we build file systems on them
 - ◆ Files
 - ◆ Directories
 - ◆ Sharing
 - ◆ Protection
 - ◆ File System Layouts
 - ◆ File Buffer Cache
 - ◆ Read Ahead

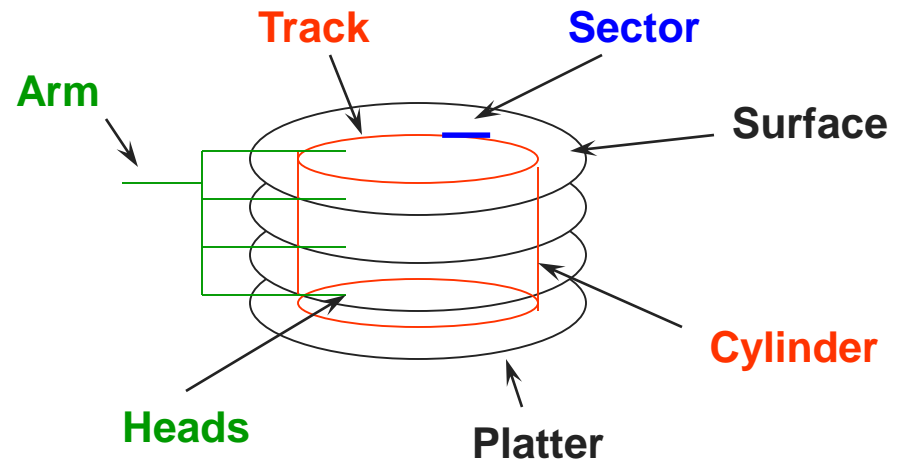
Disks and the OS

- Disks are messy physical devices:
 - ◆ Errors, bad blocks, missed seeks, etc.
- The job of the OS is to hide this mess from higher level software
 - ◆ Low-level device control (initiate a disk read, etc.)
 - ◆ Higher-level abstractions (files, databases, etc.)
- The OS may provide different levels of disk access to different clients
 - ◆ Physical disk (surface, cylinder, sector)
 - ◆ Logical disk (disk block #)
 - ◆ Logical file (file block, record, or byte #)

Physical Disk Structure

- Disk components

- ◆ Platters
- ◆ Surfaces
- ◆ Tracks
- ◆ Sectors
- ◆ Cylinders
- ◆ Arm
- ◆ Heads



Disk Interaction

- Specifying disk requests requires a lot of info:
 - ◆ Cylinder #, surface #, track #, sector #, transfer size...
- Older disks required the OS to specify all of this
 - ◆ The OS needed to know all disk parameters
- Modern disks are more complicated
 - ◆ Not all sectors are the same size, sectors are remapped, etc.
- Current disks provide a higher-level interface (SCSI)
 - ◆ The disk exports its data as a logical array of blocks [0...N]
 - » Disk maps logical blocks to cylinder/surface/track/sector
 - ◆ Only need to specify the logical block # to read/write
 - ◆ But now the disk parameters are hidden from the OS

Modern Disk Specifications

- Seagate Enterprise Performance 3.5" ([server](#))
 - ◆ capacity: 600 GB
 - ◆ rotational speed: 15,000 RPM
 - ◆ sequential read performance: 233 MB/s (outer) – 160 MB/s (inner)
 - ◆ seek time (average): 2.0 ms
- Seagate Barracuda 3.5" ([workstation](#))
 - ◆ capacity: 3000 GB
 - ◆ rotational speed: 7,200 RPM
 - ◆ sequential read performance: 210 MB/s - 156 MB/s (inner)
 - ◆ seek time (average): 8.5 ms
- Seagate Savvio 2.5" ([smaller form factor](#))
 - ◆ capacity: 2000 GB
 - ◆ rotational speed: 7,200 RPM
 - ◆ sequential read performance: 135 MB/s (outer) - ? MB/s (inner)
 - ◆ seek time (average): 11 ms

Disk Performance

- Disk request performance depends upon three steps
 - ◆ Seek – moving the disk arm to the correct cylinder
 - » Depends on how fast disk arm can move (increasing very slowly)
 - ◆ Rotation – waiting for the sector to rotate under the head
 - » Depends on rotation rate of disk (increasing, but slowly)
 - ◆ Transfer – transferring data from surface into disk controller electronics, sending it back to the host
 - » Depends on density (increasing quickly)
- When the OS uses the disk, it tries to minimize the cost of all of these steps
 - ◆ Particularly seeks and rotation

Solid State Disks

- SSDs are a relatively new storage technology
 - ◆ Memory that does not require power to remember state
- No physical moving parts → faster than hard disks
 - ◆ No seek and no rotation overhead
 - ◆ But...more expensive, not as much capacity
- Generally speaking, file systems have remained unchanged when using SSDs
 - ◆ Some optimizations no longer necessary (e.g., layout policies, disk head scheduling), but basically leave FS code as is
 - ◆ Initially, SSDs have the same disk interface (SATA)
 - ◆ Increasingly, SSDs used directly over the I/O bus (PCIe)
 - » Much higher performance

Disk Scheduling

- Because seeks are so expensive (milliseconds!), the OS tries to schedule disk requests that are queued waiting for the disk
 - ◆ FCFS (do nothing)
 - » Reasonable when load is low
 - » Long waiting times for long request queues
 - ◆ SSTF (shortest seek time first)
 - » Minimize arm movement (seek time), maximize request rate
 - » Favors middle blocks
 - ◆ SCAN (elevator)
 - » Service requests in one direction until done, then reverse
 - ◆ C-SCAN
 - » Like SCAN, but only go in one direction (typewriter)

Disk Scheduling (2)

- In general, unless there are request queues, disk scheduling does not have much impact
 - ◆ Important for servers, less so for PCs
- Modern disks often do the disk scheduling themselves
 - ◆ Disks know their layout better than OS, can optimize better
 - ◆ Ignores, undoes any scheduling done by OS

File Systems

- File systems
 - ◆ Implement an abstraction ([files](#)) for secondary storage
 - ◆ Organize files logically ([directories](#))
 - ◆ Permit sharing of data between processes, people, and machines
 - ◆ Protect data from unwanted access (security)

Files

- A file is 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)`

File Access Methods

- Some file systems provide different **access methods** that specify different ways for accessing data in a file
 - ◆ **Sequential access** – read bytes one at a time, in order
 - ◆ **Direct 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 (DBs)
- **What file access method does Unix, Windows provide?**
- Older systems provide the more complicated methods

Directories

- Directories serve two purposes
 - ◆ For users, they provide a structured way to organize files
 - ◆ For the file system, they provide a convenient naming interface that allows the implementation to separate logical file organization from physical file placement on the disk
- Most file systems support multi-level directories
 - ◆ Naming hierarchies (`/`, `/usr`, `/usr/local/`, ...)
- Most file systems support the notion of a current directory
 - ◆ Relative names specified with respect to current directory
 - ◆ Absolute names start from the root of directory tree

Directory Internals

- A directory is a list of entries
 - ◆ <name, location>
 - ◆ Name is just the name of the file or directory
 - ◆ Location depends upon how file is represented on disk
- List is usually unordered (effectively random)
 - ◆ Entries usually sorted by program that reads directory
- Directories typically stored in files
 - ◆ Only need to manage one kind of secondary storage unit

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()

Path Name Translation

- Let's say you want to open `"/one/two/three"`
- What does the file system do?
 - ◆ Open directory `"/` ([well known, can always find](#))
 - ◆ 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"`
- Systems spend a lot of time walking directory paths
 - ◆ This is why open is separate from read/write
 - ◆ OS will cache prefix lookups for performance
 - » `/a/b`, `/a/bb`, `/a/bbb`, etc., all share `"/a"` prefix

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

Subjects

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

ACL

Capability

The diagram shows a table with subjects as rows and objects as columns. A green dashed oval labeled 'ACL' encircles the first column (object '/one'). A pink dashed oval labeled 'Capability' encircles the third row (subject 'Charlie').

-
- Setuid
 - ◆ Show setuid bit
 - Root/sudo/administrator
 - Pic of ls -l

ACLs and Capabilities

- The 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)

File System Layout

How do file systems use the disk to store files?

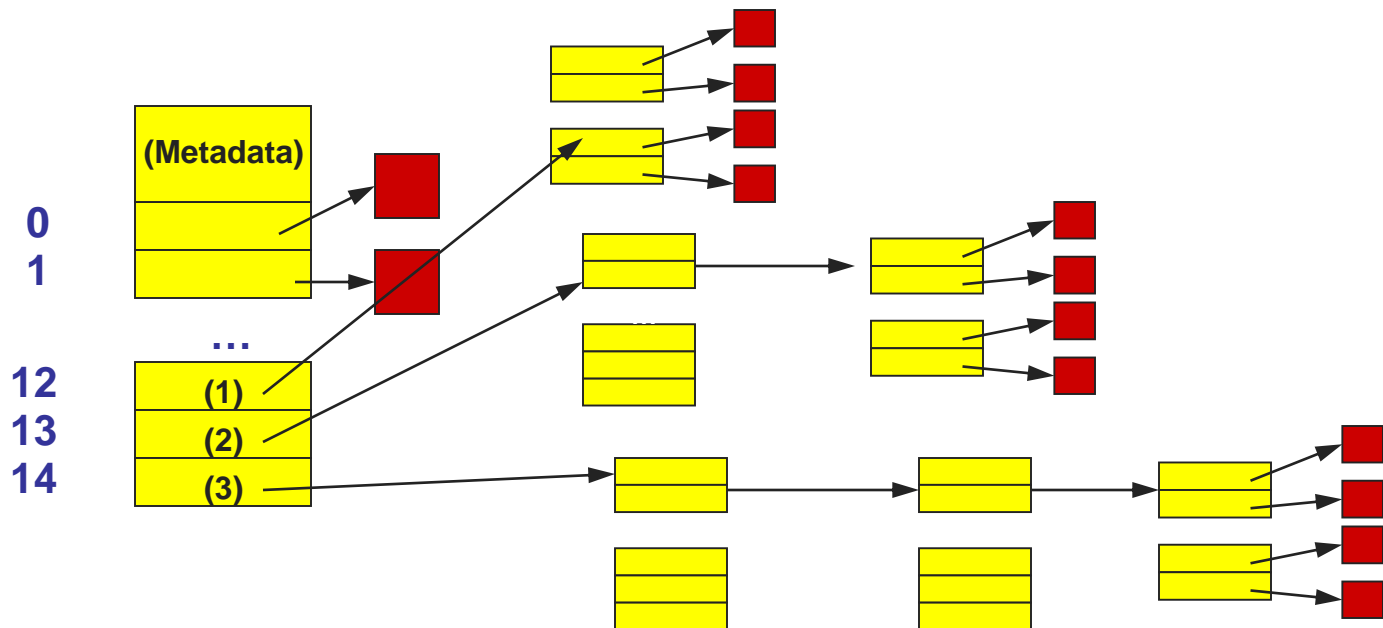
- File systems define a block size (e.g., 4KB)
 - ◆ Disk space is allocated in granularity of blocks
- A “Master Block” determines location of root directory
 - ◆ Always at a well-known disk location
 - ◆ Often replicated across disk for reliability
- A free map determines which blocks are free, allocated
 - ◆ Usually a bitmap, one bit per block on the disk
 - ◆ Also stored on disk, cached in memory for performance
- Remaining disk blocks used to store files (and dirs)
 - ◆ There are many ways to do this

Disk Layout Strategies

- Files span multiple disk blocks
- How do you find all of the blocks for a file?
 1. Contiguous allocation
 - » Like memory
 - » Fast, simplifies directory access
 - » Inflexible, causes fragmentation, needs compaction
 2. Linked structure
 - » Each block points to the next, directory points to the first
 - » Good for sequential access, bad for all others
 3. Indexed structure (indirection, hierarchy)
 - » An “index block” contains pointers to many other blocks
 - » Handles random better, still good for sequential
 - » May need multiple index blocks (linked together)

Unix Inodes

- Unix inodes implement an indexed structure for files
 - ◆ Also store metadata info (protection, timestamps, length, ref count...)
- Each inode contains 15 block pointers
 - ◆ First 12 are direct blocks (e.g., 4 KB blocks)
 - ◆ Then single, double, and triple indirect



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

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)

Summary

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

Next time...

- Read Chapters 11.8, 12.7