

CSE 120

Principles of Operating Systems

Fall 2023

Lecture 12: File Systems

Geoffrey M. Voelker

File Systems

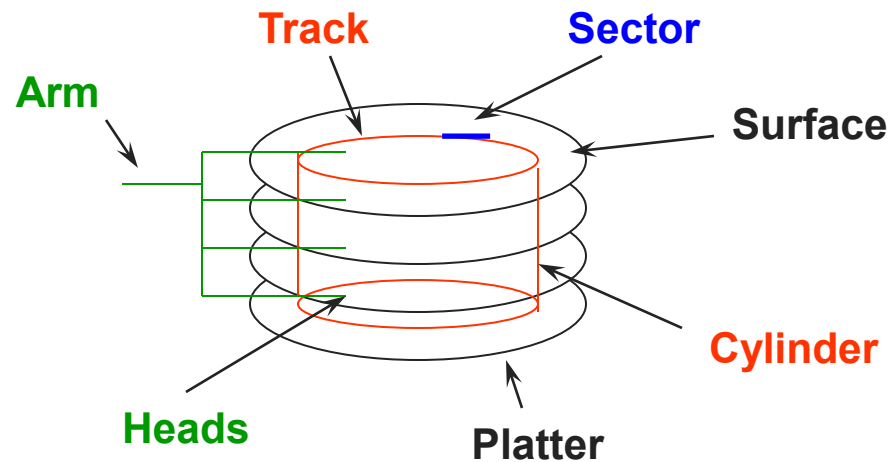
- First we'll discuss properties of physical disks
 - ◆ Structure
 - ◆ Performance
- Then how file systems support users and programs
 - ◆ Files + Directories
 - ◆ Sharing
 - ◆ File buffer cache
- Then how file systems are implemented
 - ◆ File system data structures and layouts
 - ◆ Name resolution
- End with protection

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
 - ♦ The disk exports its data as a logical array of blocks [0...N]
 - » Disk maps logical blocks to cylinder/surface/track/sector
 - » Block size can be configured via low-level formatting
 - ♦ Only need to specify the logical block # to read/write
 - ♦ But now the disk parameters are hidden from the OS

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 now standard in personal devices
 - ◆ 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 M.2/3)
 - » Much higher performance

Non-Volatile Memory (NVM)

- Under development are new technologies that provide non-volatile memory
 - ◆ Phase change (PCM), spin-torque transfer (STTM), resistive
- Performance close to DRAM...but persistent!
- Byte-addressable
 - ◆ SSD is in units of a page
- Unlike SSDs, NVM will have a dramatic effect on both operating systems and applications
 - ◆ See Steve Swanson's and Jishen Zhao's research!
- Intel/Micron Optane first commercially available product
 - ◆ Although recently cancelled...

File Systems

- File systems
 - ◆ Implement an abstraction (**files**) for secondary storage
 - ◆ Organize files logically (**directories**)
 - ◆ Permit sharing of data among processes, people, and machines
 - ◆ Protect data from unwanted access (**protection**)
- OSes abstract file systems behind an interface
 - ◆ Unix: **virtual file system (VFS)**
 - ◆ Windows: **installable file system (IFS)**
- Interface defines set of methods and data types
 - ◆ OS implemented to use any file system through this interface
 - ◆ Linux (**ext3, ext4, xfs, btrfs**), Windows (**FAT16, FAT32, NTFS**)
 - ◆ Another example of level-of-indirection in systems design

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)`
- `rename(old,new)`

Windows

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

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
- File systems support multi-level directories
 - ◆ Naming hierarchies (`/`, `/usr`, `/usr/local/`, ...)
- OSes support the notion of a current directory
 - ◆ Relative names specified with respect to current directory
 - ◆ Absolute names start from the root of directory tree
 - ◆ Maintained on a per-process basis

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
 - ◆ Try `ls -U /bin`
- 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 (v1)

- 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
 - ◆ 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

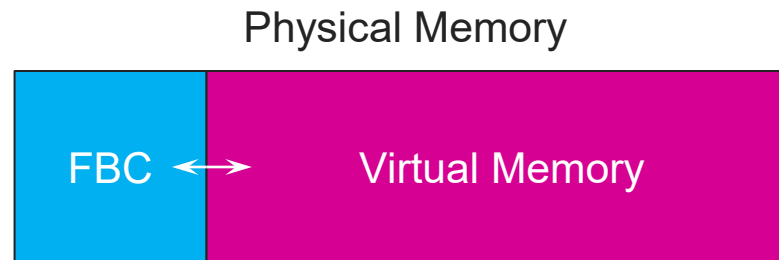
- File sharing has been around since timesharing
 - ◆ Easy to do on a single machine
 - ◆ Networks give us remote sharing (mostly)
- File sharing is important for getting work done
 - ◆ Basis for communication and synchronization
- Two key issues when sharing files
 - ◆ Semantics of concurrent access
 - » 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

File Buffer Cache

- Applications exhibit significant locality for reading and writing files → exploit this locality for performance
- 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: More physical memory for file cache, less for VM**
 - ◆ Like VM, it has limited size
 - ◆ Need replacement algorithm (**form of LRU usually used**)

Physical Memory Split

- VM and buffer cache dynamically compete over time



```
top - 10:55:28 up 55 days, 21:52, 3 users, load average: 2.05, 1.67, 0.95
Tasks: 213 total, 1 running, 135 sleeping, 1 stopped, 0 zombie
%Cpu(s): 24.9 us, 2.2 sy, 0.0 ni, 72.7 id, 0.1 wa, 0.0 hi, 0.1 si, 0.0
KiB Mem: 98967440 total, 86684056 free, 1098808 used, 11184572 buff/cache
KiB Swap: 2097148 total, 2097148 free, 0 used, 96879288 avail Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
8558	ubachira	20	0	28.067g	44056	17960	S	16.6	0.0	0:00.50	java
4436	voelker	20	0	33532	3704	3164	R	0.7	0.0	0:00.31	top
9	root	20	0	0	0	0	I	0.3	0.0	40:02.62	rcu_sch+
279	root	0	-20	0	0	0	I	0.3	0.0	7:44.40	kworker+
30948	root	20	0	584532	22712	10016	S	0.3	0.0	62:47.92	fail2ba+
1	root	20	0	225532	9640	7036	S	0.0	0.0	1:40.59	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:01.07	kthreadd
4	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	kworker+
6	root	20	0	0	0	0	I	0.0	0.0	0:05.14	kworker+
7	root	0	-20	0	0	0	I	0.0	0.0	0:00.00	mm_perc+
8	root	20	0	0	0	0	S	0.0	0.0	0:05.45	ksoftir+
10	root	20	0	0	0	0	I	0.0	0.0	0:00.00	rcu_bh
11	root	rt	0	0	0	0	S	0.0	0.0	0:00.31	migrati+

Caching Writes

- On a write, some applications assume that data makes it through the buffer cache and onto the disk
 - ◆ Writes become 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

- File systems implement a “read ahead” optimization
 - ◆ FS predicts that the process will request next block
 - ◆ FS goes ahead and requests it from the disk
 - ◆ 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)

Next time...

- Chapter 40