Storage Systems

Module 8

Main Points

- File systems
  - Useful abstractions on top of physical devices
- Storage hardware characteristics
  - Disks and flash memory
- File system usage patterns

File Systems

- Abstraction on top of persistent storage
  - Magnetic (spinning) disk
  - SSD (Solid State Disk)
  - Flash memory (e.g., USB thumb drive)
- Devices provide
  - Storage that (usually) survives across machine crashes
  - Block level (random) access
  - Large capacity at low cost
  - Relatively slow performance
    - Magnetic disk read takes 10-20M processor instructions

File System as Illusionist: Hide Limitations of Physical Storage

- Persistence of data stored in file system
  - Even if crash happens during an update
  - Even if disk block becomes corrupted
  - Even if flash memory wears out
- Naming
  - Named data instead of disk block numbers
  - Directories instead of flat storage
  - Byte addressable data even though devices are block-oriented
- Performance
  - The fastest IO op is the one you don’t have to do
    - Cached data
  - Data placement and data structure organization
- Controlled access to shared data

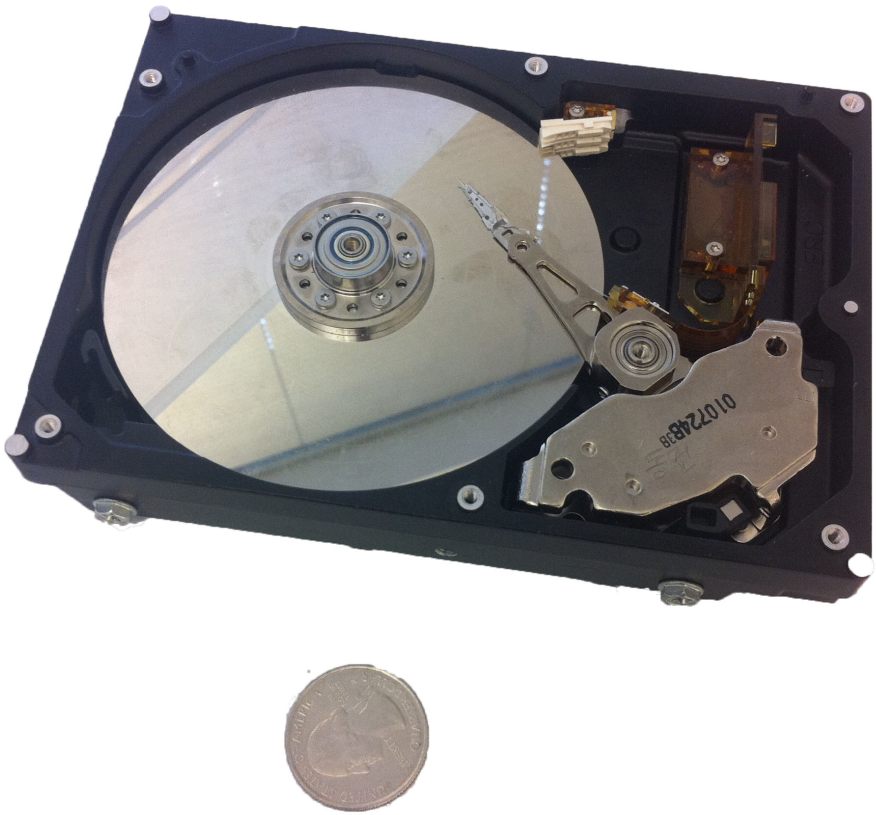
File System Abstraction

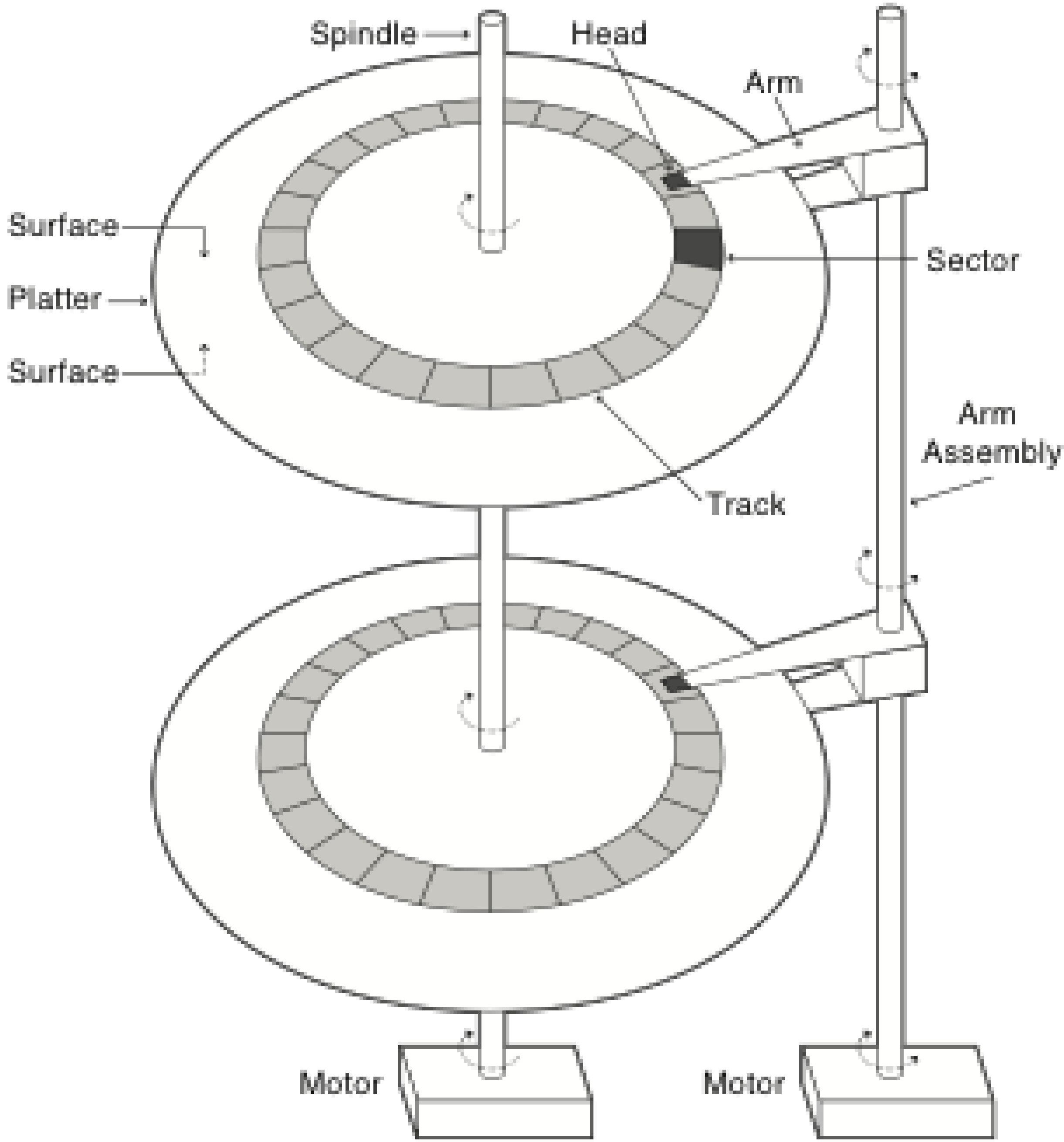
- File system
  - Persistent, named data
  - Hierarchical organization (directories, subdirectories)
  - Access control on data
- File
  - named collection of data
  - Linear sequence of bytes (or a set of sequences)
  - Read/write interface or memory mapped
- Crash and storage error tolerance
  - Operating system crashes (and disk errors) leave file system in a valid state
  - Some individual files may not be so lucky...
- Performance
  - Achieve close to the hardware limit in the average case

Storage Devices

- Magnetic disks
  - Storage that rarely becomes corrupted
  - Large capacity at low cost
  - Block level random access
  - Slow performance for random access
  - Better performance for streaming (sequential on physical device) access
- Solid state disk
  - Storage that rarely becomes corrupted
  - Capacity at intermediate cost (3x disk)
  - Lower power consumption (especially when idle)
  - Block level random access
  - Much better performance than spinning drives
    - Good performance for reads; worse for random writes

Magnetic Disk




Spindle
Head
Arm
Surface
Platter
Surface
Sector
Arm
Assembly
Track
Motor
Motor

Disk Tracks

- ~ 1 micron wide
  - Wavelength of light is ~ 0.5 micron
  - Resolution of human eye: 50 microns
  - 100K tracks on a typical 2.5" disk
- Separated by unused guard regions
  - Reduces likelihood neighboring tracks are corrupted during writes (still a small non-zero chance)
- Track length varies across disk
  - Outside: More sectors per track, higher bandwidth
  - Disk is organized into regions of tracks with same # of sectors/track
  - Only outer half of radius is used
    - Most of the disk area in the outer regions of the disk

Sectors

Sectors contain sophisticated error correcting codes
- Disk head magnet has a field wider than track
- Hide corruptions due to neighboring track writes

- Sector sparing
  - Remap bad sectors transparently to spare sectors on the same surface
- Slip sparing
  - Remap all sectors (when there is a bad sector) to preserve sequential behavior
- Track skewing
  - Sector numbers offset from one track to the next, to allow for disk head movement for sequential ops

Disk Performance

Disk Latency =

- Seek Time + Rotation Time + Transfer Time
- Seek Time: time to move disk arm over track (1-20ms)
  - Fine-grained position adjustment necessary for head to "settle"
  - Head switch time ~ track switch time (on modern disks)
- Rotation Time: time to wait for disk to rotate under disk head
  - Disk rotation: 4 – 15ms (depending on speed/price of disk)
  - "On average", need to wait only half a rotation
- Transfer Time: time to transfer data onto/off the disk
  - Disk head transfer rate: 50-100MB/s (5-10 usec/sector)
  - Host transfer rate dependent on I/O connector (USB, SATA, …)

Seagate Barracuda 2.5” Disk (2019)

Capacity	1TB
Bytes per Sector (logical/physical)	512/4096
Interface	SATA 6Gb/s
Data Transfer Rate	Up to 160 MB/sec
Cache	128 MB
Rotation speed	7200 RPM
Nonrecoverable read errors per bits read, Max	1 per 10E14
Startup current (+5V, A)	1.0
R/W Power, Average (W)	1.9/1.7
Idle Power, Average (W)	0.7

Question

- How long to complete 500 random disk reads, in FIFO order?
  - Seek: average (assumed) 10.5 msec
  - Rotation: average 4.15 msec
  - Transfer: 5-10 usec
- 500 * (10.5 + 4.15 + 0.01)/1000 = 7.3 seconds

Question

- How long to complete 500 sequential disk reads?
  - Seek Time: 10.5 ms (to reach first sector)
  - Rotation Time: 4.15 ms (to reach first sector)
  - Transfer Time:

    (500 sectors) * (512 bytes / sector) / (128MB/sec) = 2ms

Total: 10.5 + 4.15 + 2 = 16.7 ms

Might need an extra head or track switch (+1ms)

Track buffer may allow some sectors to be read off disk out of order (-2ms)

Disk Scheduling

- What does "disk scheduling" mean?
  - The order in which disk I/O requests are served
- Why does it matter?
  - Seek and latency depend on location of I/O op data relative to R/W head
- How much can it matter?
  - See the previous slides!
- Who does it?
  - Could be OS
  - Could be the device itself
    - Command queueing

Disk Scheduling

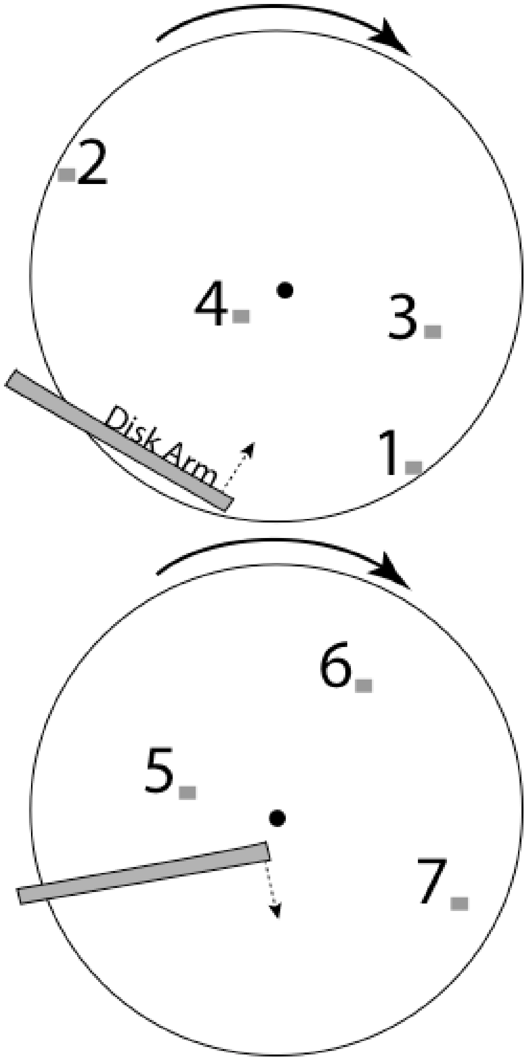
- FIFO
  - Schedule disk operations in order they arrive
  - Downsides?

Disk Scheduling

- SSTF (Shortest seek time first)
  - Not optimal!
    - (That it's not optimal might seem counter-intuitive if we had done CPU scheduling already, but we postponed that to get to disks, because of the project)
    - Suppose one request toward outer edge and a "ladder" of requests toward inner request with each next one always closer than the outer edge request
  - Besides not being optimal, other downsides?

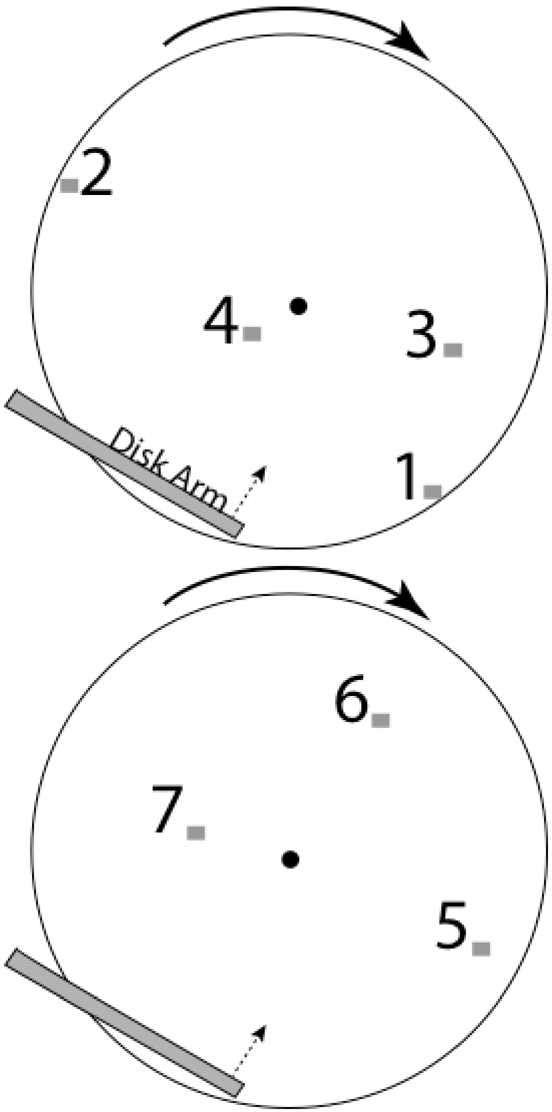
Disk Scheduling

- SCAN: move disk arm in one direction, until all requests satisfied, then reverse direction
- Also called “elevator scheduling”



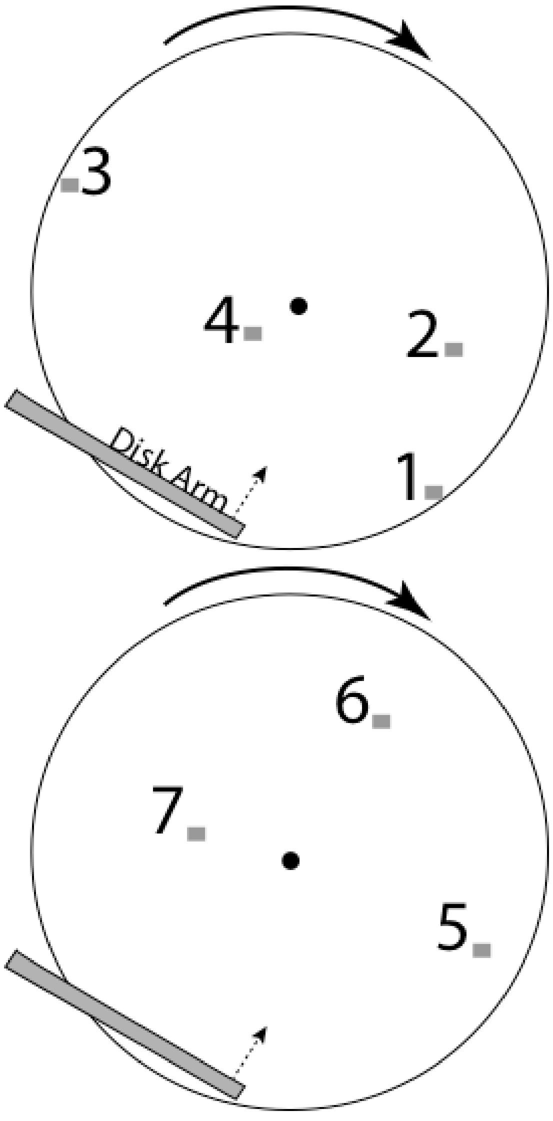
Disk Scheduling

- CSCAN: move disk arm in one direction, until all requests satisfied, then start again from farthest request



Disk Scheduling

- R-CSCAN: CSCAN but take into account that short track switch is < rotational delay



Question

- How long to complete 500 random disk reads, in any order?
  - Disk seek: 1ms (most will be short)
  - Rotation: 4.15ms
  - Transfer: 5-10usec
- Total: 500 * (1 + 4.15 + 0.01) = 2.2 seconds
  - Would be a bit shorter with R-CSCAN
  - vs. 7.3 seconds if FIFO order

Question

- Why would reads be random?
- How could you try to reduce the likelihood that they were random?

Question

- How long to read all of the bytes off of a disk?
  - Disk capacity: 1TB
  - Disk bandwidth: 54-128MB/s
- Transfer time =

  Disk capacity / average disk bandwidth

  ~ 10,500 seconds (3 hours)

SSDs – Flash Memory

- No moving parts!
  - No seek time, no latency time, no limitation on transfer rate due to limited rotation time
    - (That last one was a bit misleading. Why?)
- More "penalty-free random access" than spinning disks
- Less "penalty-free random access" than main memory

Flash Memory

- Write/read page (2-4KB)
  - 50-100 usec
- But…, writes must be to “clean” cells
  - no update in place
  - Large block erasure required before write
  - Erasure block: 128 – 512 KB
  - Erasure time: Several milliseconds
  - (SSD performance is increasing quickly, so distrust the specific values here!)

Seagate Firecuda M.2 Disk (2019)

Capacity	1TB
Interface	PCIe Gen4 x4, NVMe 1.3
NAND Flash Memory	3D TLC
Sequential Read (Max), 128KB	5000 MB/s
Sequential Write (Max), 128KB	4400 MB/s
Random Read (Max, QD32)	760,000 IOPS
Random Write (Max, QD32)	700,000 IOPS
Active Power, Average	5.6 W
Idle Power, Average	15 mW
Lower Power mode	2 mW
Total Bytes Written (before failure)	1800 TB

Flash Translation Layer

- Flash device firmware maps logical page # to a physical location
  - The device presents a name space, page numbers, for the OS to use, but they are not physical addresses on the device
- Transparent to the device user (i.e., the OS)

- *(Spinning disks map as well)*

Flash Translation Layer: Garbage Collection

- Improve performance by garbage collecting pages and cleaning blocks
  - Pack in-use pages into an erasure block
    - Creates erasure blocks with no in-use pages
  - Pre-clean those now empty blocks
  - More efficient if blocks stored at same time are deleted at same time (e.g., keep blocks of a file together)

- Who's doing this, the disk or the OS?

File System – SSD

- How does SSD device know which blocks are live?
  - To the device, blocks are just blocks
  - Only the file system knows which blocks are in use
  - But the device is doing erasures, and must understand which blocks are live to do so efficiently
- TRIM command
  - File system tells device when blocks are no longer in use

Flash Translation Layer: Wear Leveling

- Each physical page on an SSD can be written only a limited number of times before it becomes unrealiabe
- Wear-levelling
  - Remap pages to spread wear evenly
  - Unmap pages that no longer work (like sector sparing)
    - including pages that never worked

File System Workloads

- A file system decides how to use disk storage to maintain information about:
  - file contents (data)
  - file names (and other meta-data)
  - directories
- One goal of the file system is performance
  - Remember “optimize the common case”
- What is the common case?
  - If we knew, it might help us design an efficient file system

File System Workload

- File sizes (static measure)
  - Are most files small or large?
  - Which accounts for more total storage: small or large files?

File System Workload

- File sizes
  - Are most files small or large?
    - SMALL
  - Which accounts for more total storage: small or large files?
    - LARGE

File System Workload

- File access (dynamic measure)
  - Are most accesses to small or large files?
  - Which accounts for more total I/O bytes: small or large files?

File System Workload

- File access
  - Are most accesses to small or large files?
    - SMALL
  - Which accounts for more total I/O bytes: small or large files?
    - LARGE

File System Workload

- How are files used?
  - Most files are read/written sequentially
  - Some files are read/written randomly
    - Ex: database files, swap files
  - Some files have a pre-defined size at creation
  - Some files start small and grow over time
    - Ex: program stdout, system logs

File System Design

- For small files:
  - Small blocks for storage efficiency
    - minimize internal fragmentation
  - Concurrent ops more efficient than sequential
  - On spinning disk, files used together should be stored together
- For large files:
  - Storage efficient (large blocks)
  - Contiguous allocation for sequential access
  - Efficient lookup for random access
    - E.g., don’t use a linked list of blocks on disk!
- May not know at file creation
  - Whether file will end up small or large
  - Whether file is persistent or temporary
  - Whether file will be used sequentially or randomly

File System Abstraction

- Directory
  - Group of named files or subdirectories
  - Mapping from file name to file metadata location
- Path
  - String that uniquely identifies file or directory
  - Ex: /cse/www/education/courses/cse451/19au
- Links
  - Hard link: link from name to file data
  - Soft link: link from one name to an alternate name
- Mount
  - Mapping from name in one file system to root of another

UNIX File System API

- create, link, unlink, createdir, rmdir
  - Create file, link to file, remove link
  - Create directory, remove directory
- open, close, read, write, seek
  - Open/close a file for reading/writing
  - Seek resets current position
- fsync
  - File modifications can be cached in memory
  - fsync forces modifications to disk (like a memory barrier)

File System Interface

- UNIX file open is a Swiss Army knife:
  - Open the file, return file descriptor
  - Options:
    - if file doesn’t exist, return an error
    - If file doesn’t exist, create file and open it
    - If file does exist, return an error
    - If file does exist, open file
    - If file exists but isn’t empty, nix it then open
    - If file exists but isn’t empty, return an error
    - …

Interface Design Question

- Why not provide separate syscalls for open/create/exists?
  - Would be more modular!

```
if (!exists(name))
    create(name);   // can create fail?
fd = open(name);   // does the file exist?
```