CSE 451: Operating Systems

Spring 2022

Module 10

Storage Systems

John Zahorjan

Main Points

- File systems
  - Useful storage abstractions on top of physical devices

Optimize the Common Case

- 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 drives (i.e., easily portable)
- Hardware devices provide
  - Storage that (usually) survives across machine crashes
  - Block level (random) access
  - Large capacity at low cost
    - relative to RAM
  - Relatively slow performance
    - Magnetic disk read takes 10-20M processor instructions

File System as Illusionist: Hide Limitations of Physical Storage

- Persistence of (coherent) data
  - Even if machine is turned off
  - Even if crash happens during an update
  - Even if disk block becomes corrupted
  - Even if flash memory wears out
  - Even if building burns down
  - Even if an earthquake eradicates a large geographic area
- Naming
  - Named data, instead of disk block numbers
  - Hierarchical names, instead of flat names
  - 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
    - Caching
  - Data placement and data structure organization
- Controlled access to shared data

File System Abstraction

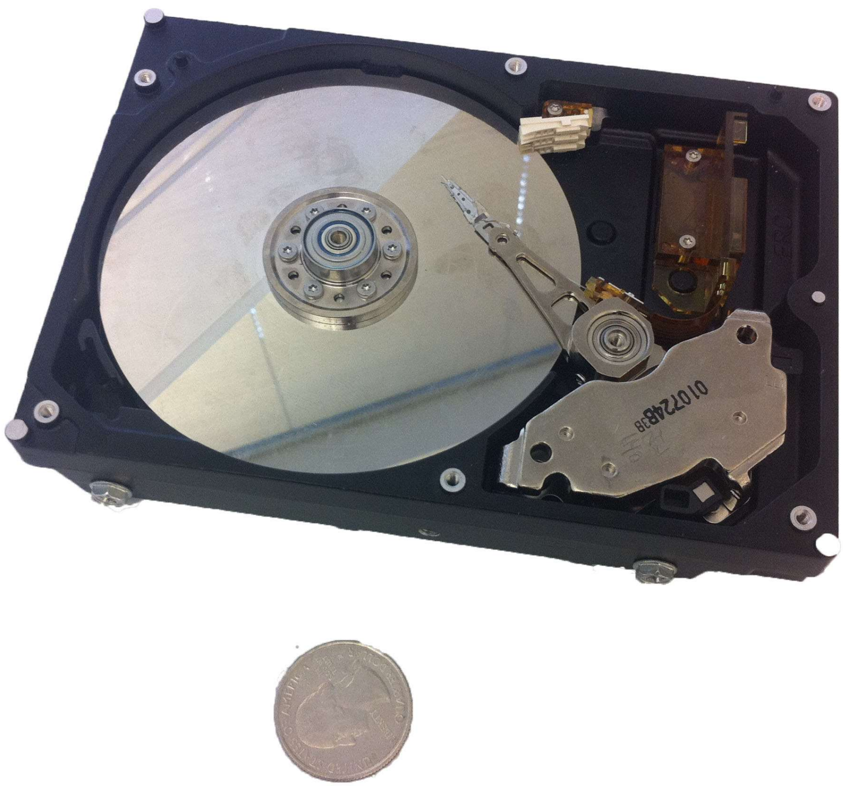
- File system
  - Persistent, named data
  - Hierarchical organization (directories, subdirectories)
  - Access control on data
  - *Not a database server*
  - *Not a web server*
- File
  - Named collection of data
  - Linear sequence of bytes (or a set of sequences)
  - Metadata (e.g., owner, permissions, last modification date, …)
  - 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
  - *(Achieve better than the hardware limit in the average case)*

Storage Devices

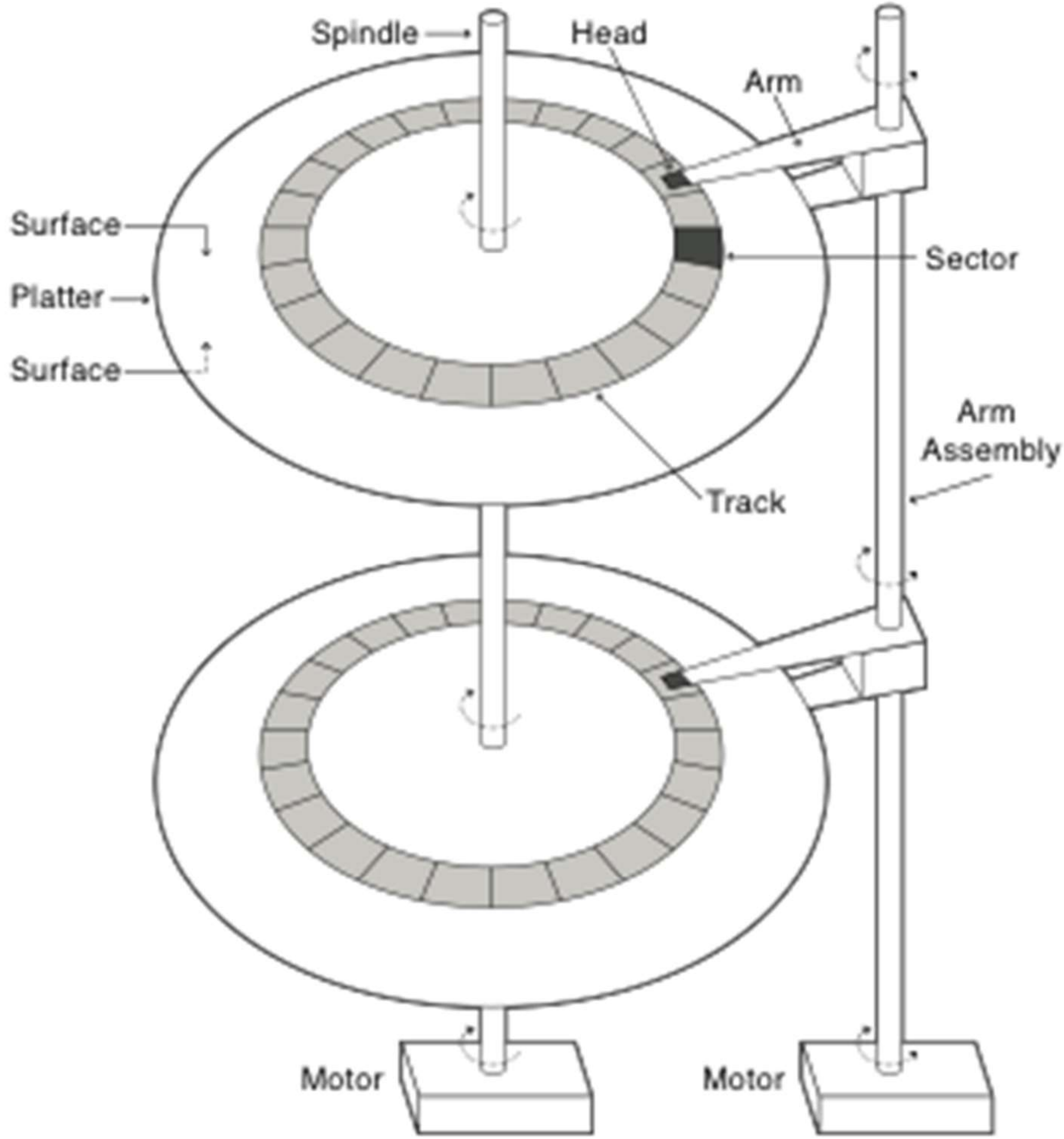
- Magnetic disks
  - Storage that rarely becomes corrupted
  - Large capacity at low cost
  - Block level random access
  - Slow performance for random access
  - Better performance for streaming (sequential on physical device) access

- 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; not as good for random writes

Magnetic (Spinning) Disk



Storage *and firmware*


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

Spinning Disk Tracks

- 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
  - 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”
  - Cheaper/faster disk that mostly works plus mechanism to deal with errors
    - Why make it perfect when I can make it 99% perfect for 80% of the cost and then mask errors in software?
  - 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 performance
- Track skewing
  - Sector numbers offset from one track to the next, to allow for disk head movement for sequential ops

Disk Performance

Latency = time from start of operation to completion of operation

Spinning 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

Spinning Disk Performance: Random / FIFO

- Q: 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

- A: 500 * (10.5 + 4.15 + 0.01)/1000 = 7.3 seconds

Spinning Disk Performance: Sequential

- Q: 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)

Spinning 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
    - SATA native command queueing
- General principle (about scheduling):
  - The more things there are to schedule, the better
    - The more choices available to you, the more likely it tends to be that the best choice is much better than the average choice

Spinning Disk Scheduling

- Goal?
  - Minimize average operation time?
    - Minimize time to complete to current batch of operations
  - Maximize throughput (operations/second)?

- Challenges
  - The time required to satisfy the N^{th} operation depends on the N-1^{st}
  - Why?

- *Note*
  - *Here we take the point of view that the every file block has a fixed location on the device and we want to schedule the current stream of requests given those locations*
  - *Later we consider where the file system should place file blocks on the disk so that "the current stream of requests" is likely to have a much better schedule than a stream of random requests*

Spinning Disk Scheduling

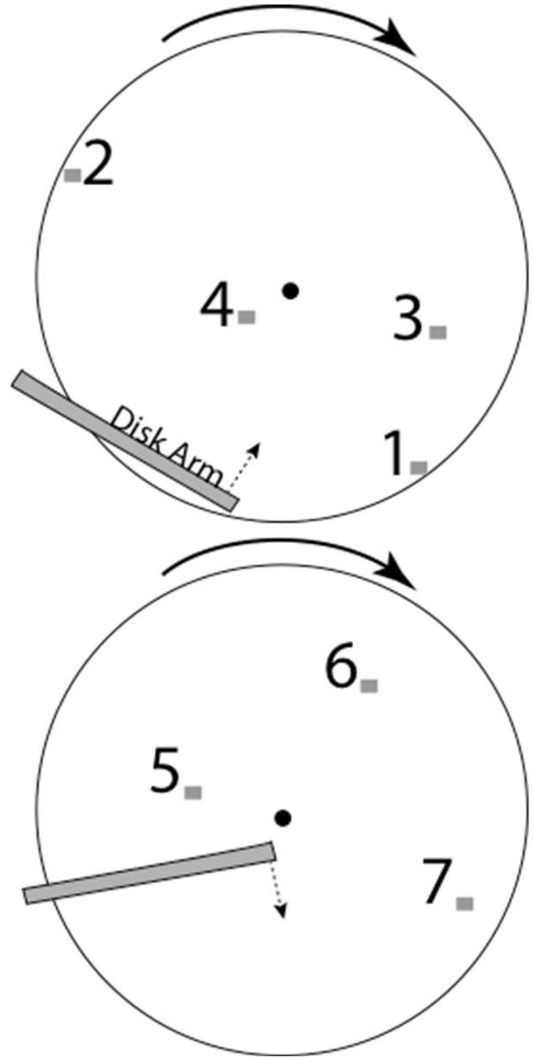
- FIFO
  - Schedule disk operations in order they arrive
  - Downsides?
  - Upside(s)?

Spinning Disk Scheduling

- FIFO
  - Schedule disk operations in order they arrive
  - Downsides?
  - Upside(s)?
- 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?
- *General principal (about scheduling)*
  - *With priorities comes the danger of starvation*

Spinning Disk Scheduling

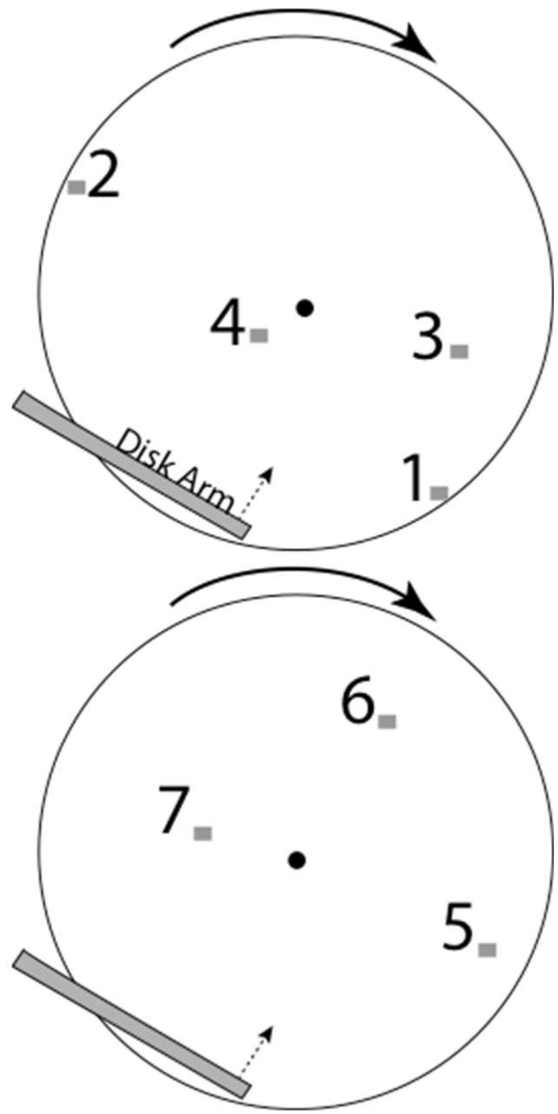
- SCAN: move disk arm in one direction, until all requests satisfied, then reverse direction
- Also called "elevator scheduling"



Con: discriminates against blocks at inner and outer edges

Spinning Disk Scheduling

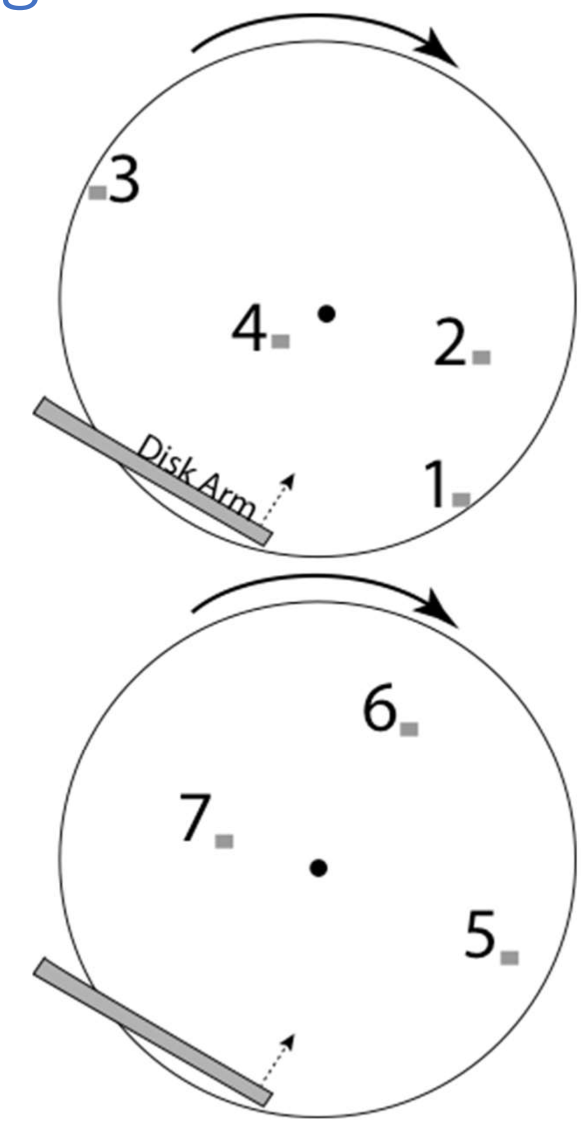
- CSCAN: move disk arm in one direction, until all requests satisfied, then start again from farthest request



Con: long seek in every schedule; considers only seek

Spinning Disk Scheduling

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



Spinning Disk Questions

- How long to complete 500 random disk reads in a well chosen 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*

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

Spinning Disk Questions

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

  Disk capacity / average disk bandwidth

  ~ 10,500 seconds (3 hours)

- *General Implication*
  - *As disk capacities go up, the time required to copy all the data on them goes up*
  - *That makes periodic "backup" – periodically copying all the data from the disk to somewhere that has independent failure semantics – less and less manageable*
  - *Instead, "backup all the time"*
    - *by making multiple online copies*
    - *by using techniques that use multiple disks in a way that allows file recovery even if a single disk fails*

Solid State Disks (SSDs) – Flash Memory

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

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 Memory

(SSD performance is increasing quickly, so distrust the specific values here!)

- Read/write pages (2-4KB)
  - 50-100 usec
- Must erase a page that has already been written to before writing it again
  - **no update in place**
  - must erase first, then write
- Erasure is performed only on large erasure blocks
  - Erasure block: 128 – 512 KB
  - Many pages
- Erasure is slow
  - Several milliseconds
- When you want to update a logical page, “must” move it

Flash Translation Layer

- Support for moving pages is provided by the disk device
  - It's software (ok, firmware), but it runs on the disk, not in the OS on the CPU

- Disk firmware maps logical page # (used by OS) 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

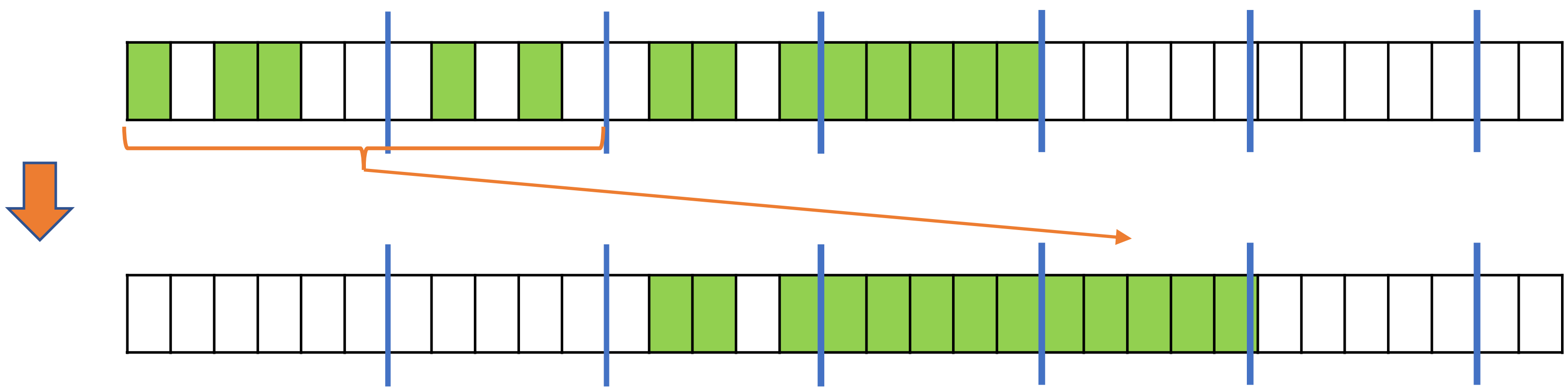
- Transparent to the device user (i.e., the OS)
  - What's great about that?!
  - What's not great about that?!

(Spinning disks map as well)

Flash Translation Layer: Garbage Collection

- Improve performance by garbage collecting pages and cleaning blocks
  - Pack in-use pages into (full) erasure blocks
    - Creates erasure blocks with no in-use pages as a result
  - Pre-clean (erase) those now empty blocks
  - More efficient if pages 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 pages are live?
- To the device, pages are just pages
  - “In use” is a logical idea
- Only the file system knows which pages are in use
  - When a file is deleted, there is no disk operation on its data blocks
- 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

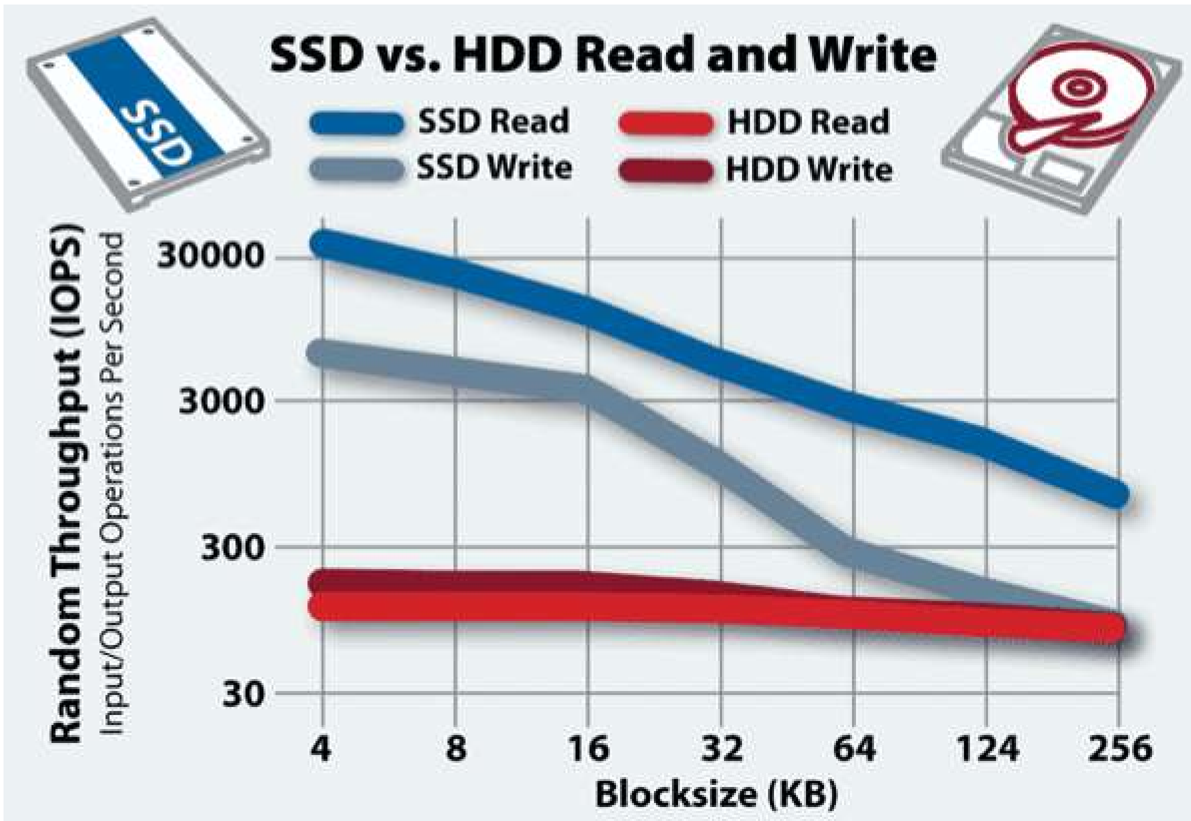
- Each physical page on an SSD can be written only a limited number of times before it becomes unreliable

- Wear-levelling
  - Remap pages to spread wear evenly
  - Unmap pages that no longer work (like sector sparing)
    - including pages that never worked

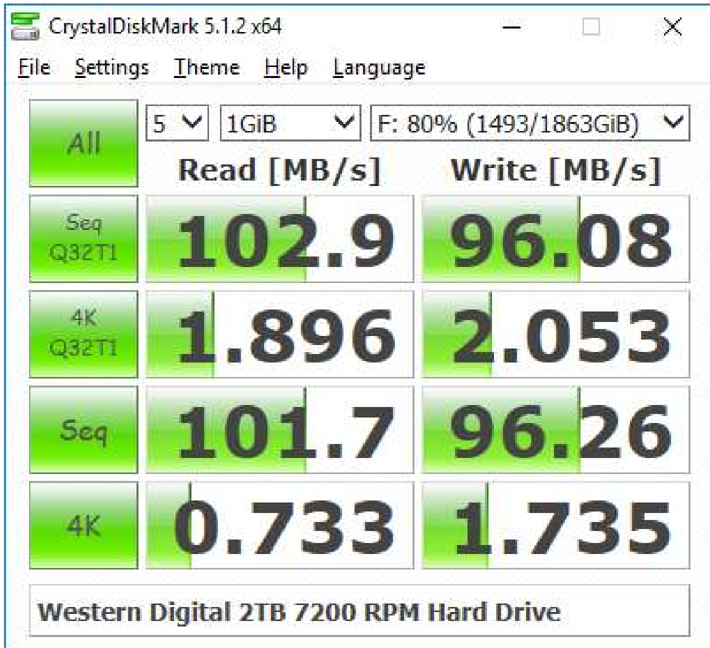
Storage Technology Comparison

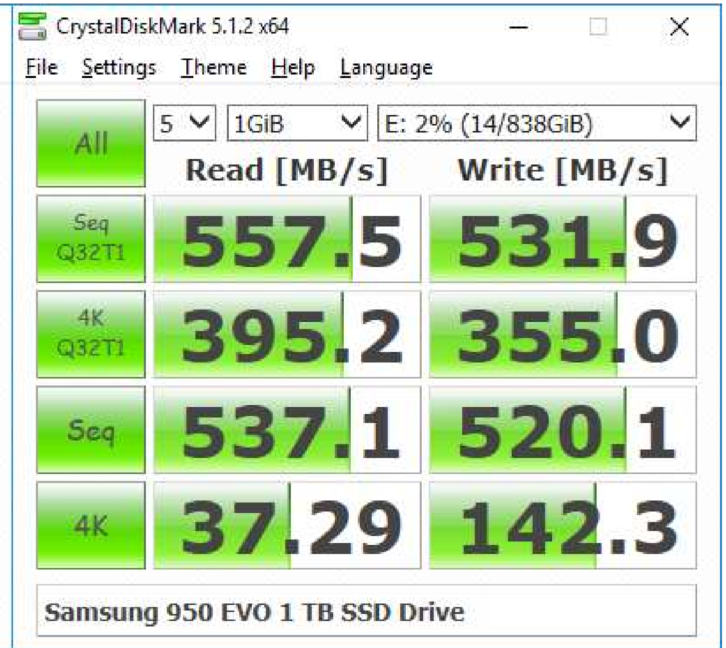
- SDD has interface identical to HDD
  - SATA bus
- NVMe device has internals like SDD but incompatible physical interface
  - faster...

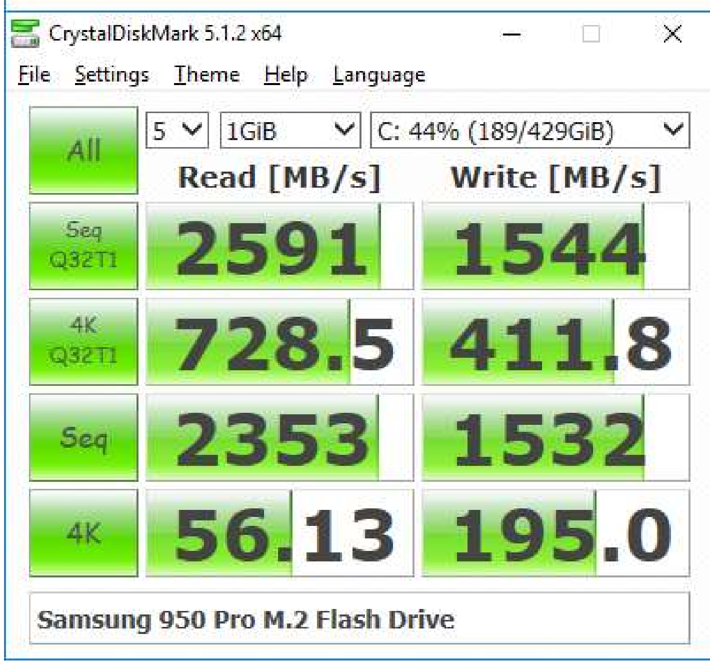
HDD vs. SDD IOPS and Transfer Size



https://www.enterprisestorageforum.com/storage-hardware/ssd-vs-hdd-speed.html







Performance Summary:

Hard Drive: 103 MB/sec read, 96 MB/sec write speed
SSD Drive: 558 MB/sec read, 532 MB/sec write speed
M.2 Drive: 2591 MB/sec read, 1544 MB/sec write speed

In Read Performance:

SSD is 5x Faster than HDD

M.2 is 5x Faster than SSD

M.2 is 25x Faster than HDD

Storage Performance Summary

- Yes, newer is faster
- If you make some component fast enough, some other component becomes the bottleneck
- Large, sequential transfers are advantageous on all device technologies
  - On spinning disks, amortize seek and latency overheads
  - On SSDs, for reasons explained in a moment

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 Decisions and Workloads

- Big blocks or little blocks?
  - Fragmentation
  - Amount of indexing information needed to list blocks in a file
- Which blocks on same track, which on different?
  (Which blocks in a single erasure block?)
- File block indexing structures?
  - Access time vs. space overhead
- Optimize for reading or for writing?

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 IO operations on small files or large ones?
    - Counts IO ops
  - Which accounts for more total I/O bytes: small or large files?
    - Counts bytes transferred

File System Workload

- File access
  - Are most IO operations on small files or large ones?
    - 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 known size at creation
  - Some files start small and grow over time
    - Ex: program stdout, system logs

File System Design

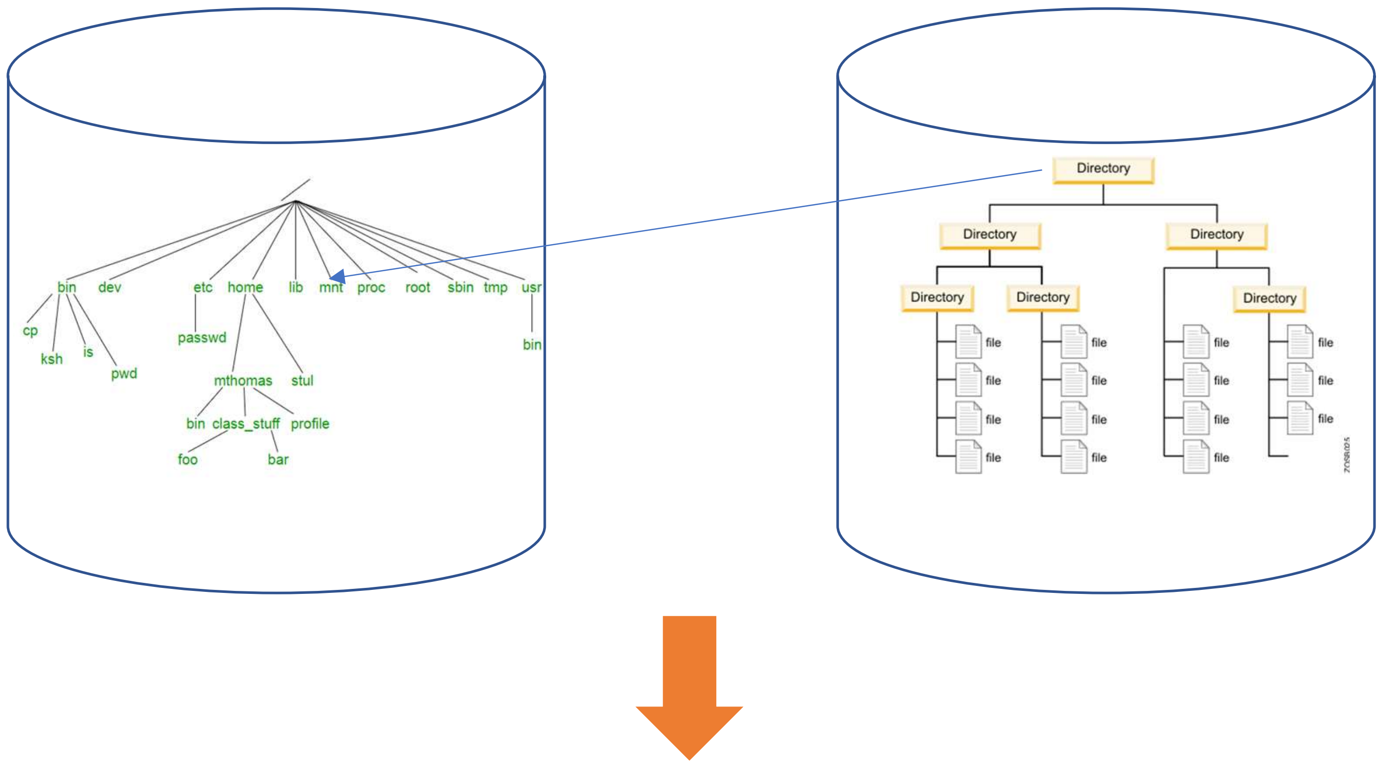
- For small files:
  - Small blocks for storage efficiency
    - minimize internal fragmentation
  - Concurrent ops more efficient than sequential
  - On spinning disk, files used together should be stored together
- For large files:
  - Storage efficient (large blocks)
  - Contiguous allocation for 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

- Path
  - String that uniquely identifies file or directory
  - Ex: /cse/www/education/courses/cse451/20sp
- Links
  - Hard link: link from name to file metadata
  - Soft link: link from one name to an alternate name
- Directory
  - Group of named files, including subdirectories
  - Maps from file name to file metadata location
    - inode number, in xk (and many other systems)
- Mount
  - Mapping from name in one file system to root of another

Mount / “File System”

- Want storage to be self-describing



In memory version of namespace

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)
  - Note: modifications include updated metadata

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?
```

Summary

- Storage systems provide persistent storage
- File systems abstract persistent storage to make it easier to use
- Like all system software, file systems strive to achieve convenient functionality while allowing performance close to what could be achieved implementing on top of raw hardware
- “Optimize the common case” requires that we know what the common case is
- “Optimize the common case” requires that we know relative cost of operations on the hardware device