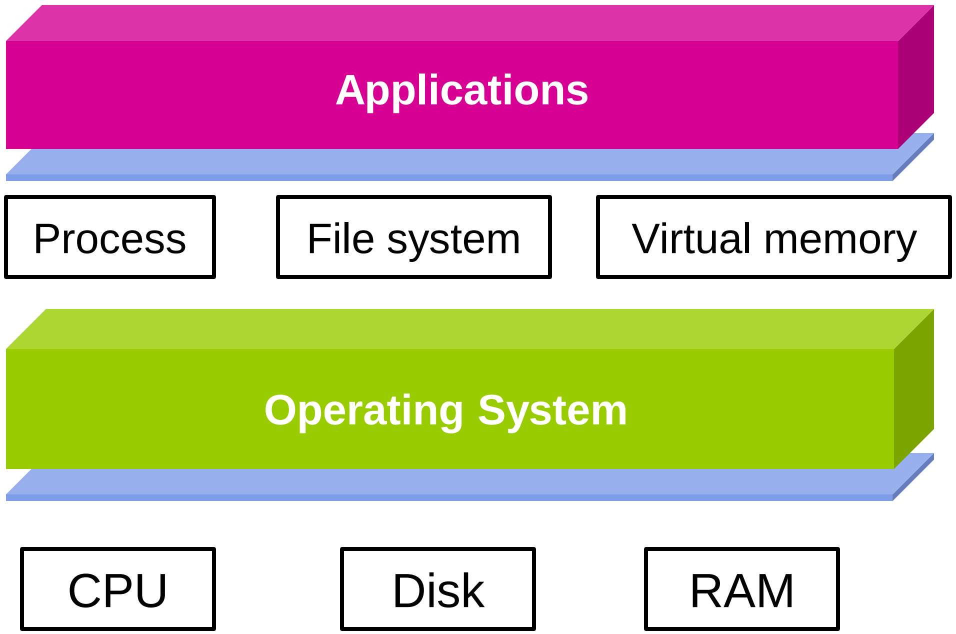
# CS 202: Advanced Operating Systems

## File Systems

# OS Abstractions



# What's Inside A Disk Drive?

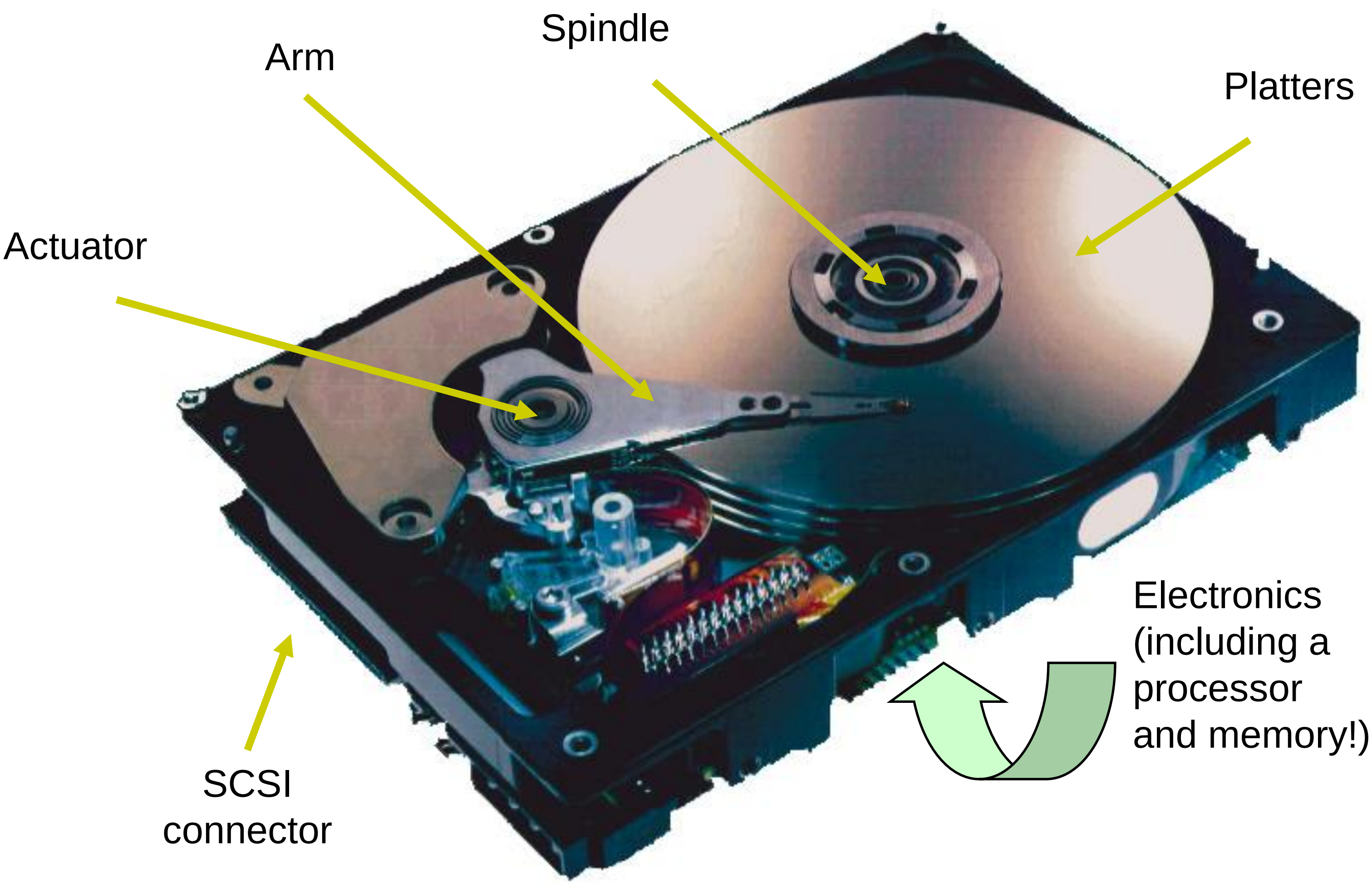
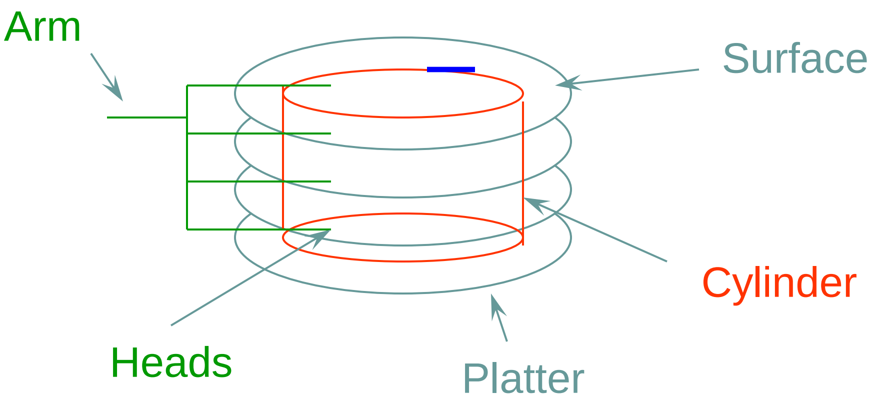


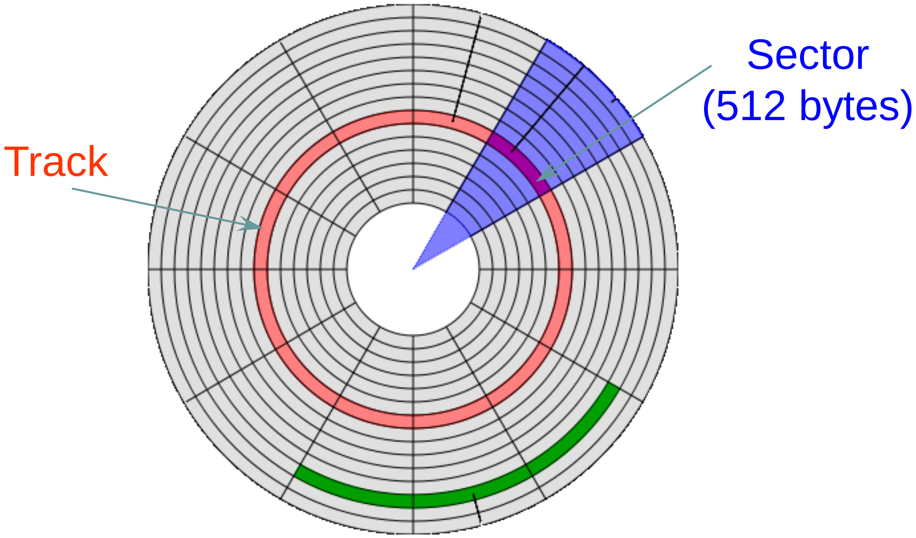

*Image courtesy of Seagate Technology*

# Physical Disk Structure

- Disk components
  - Platters
  - Surfaces
  - Tracks
  - Sectors
  - Cylinders
  - Arm
  - Heads

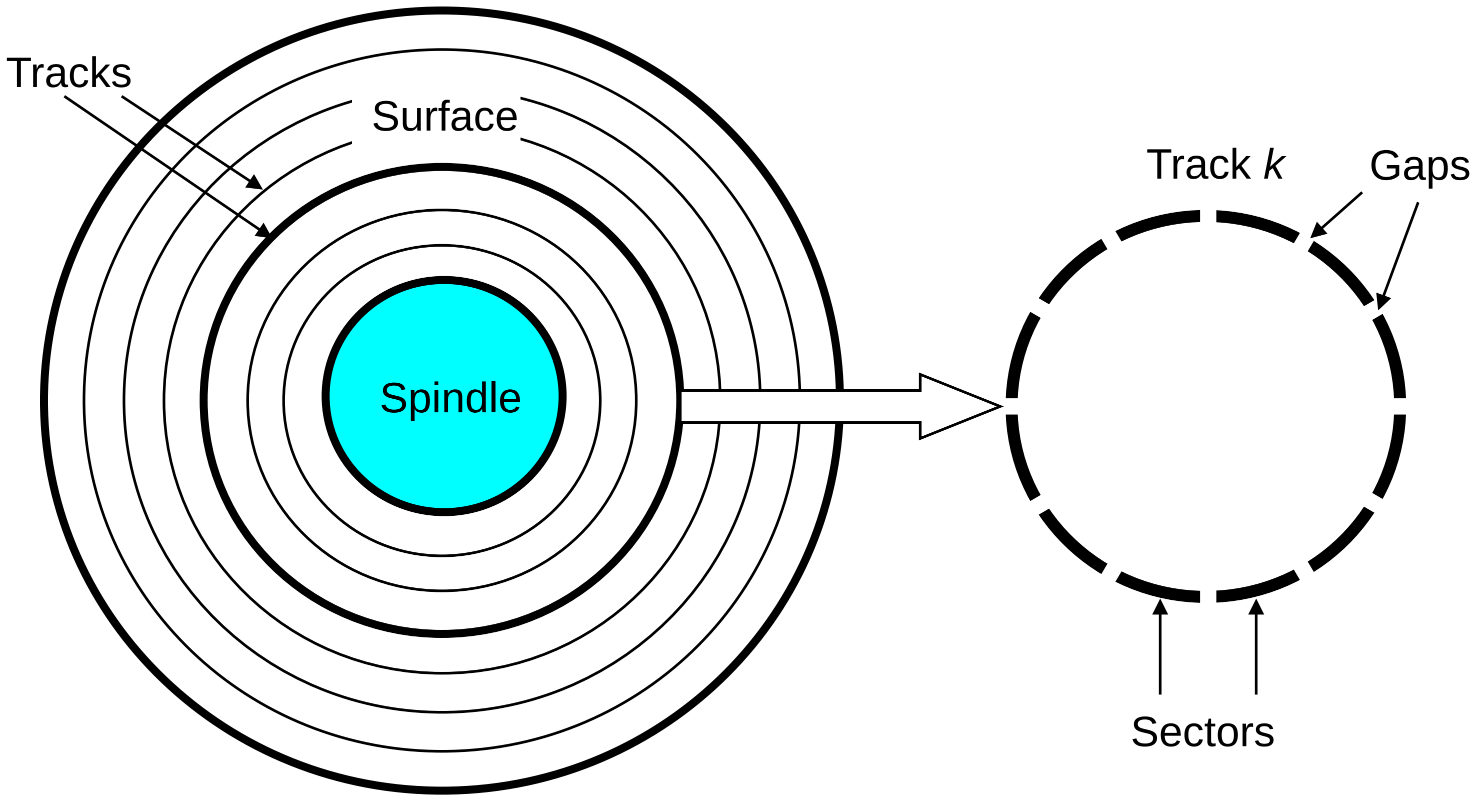






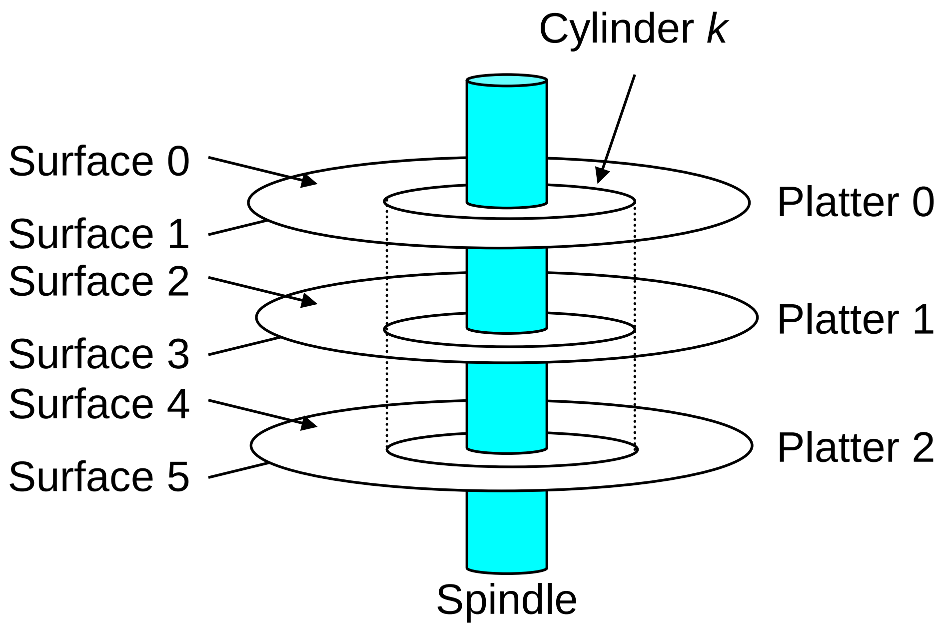
# Disk Geometry

- Disks consist of platters, each with two surfaces.
- Each surface consists of concentric rings called tracks.
- Each track consists of sectors separated by gaps.



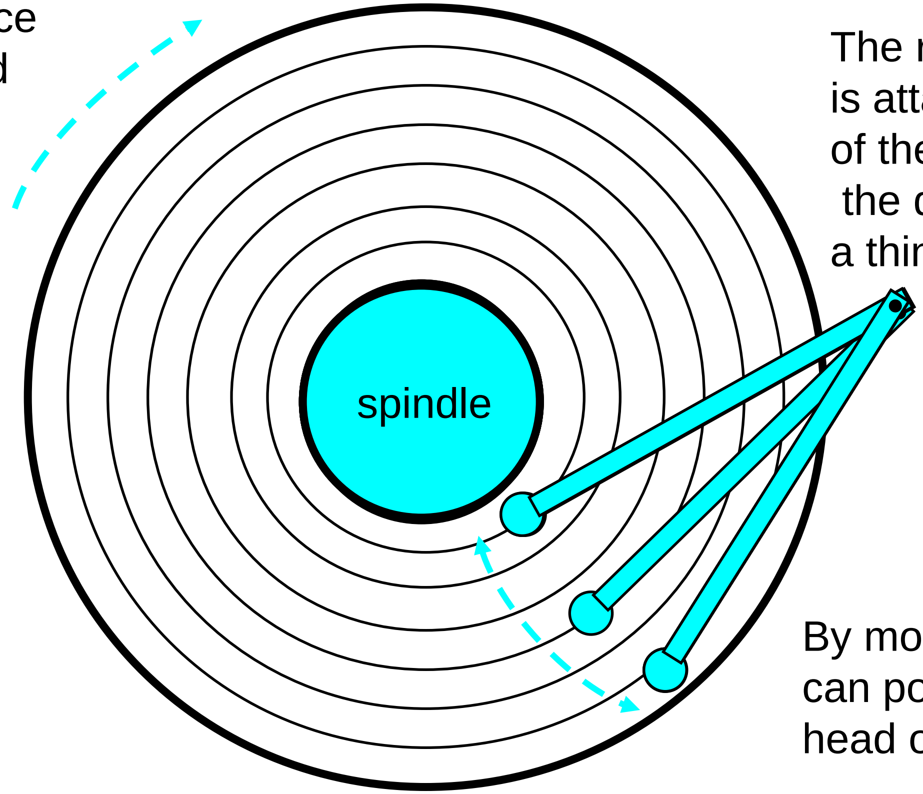
# Disk Geometry (Muliple-Platter View)

- Aligned tracks form a cylinder.



# Disk Operation (Single-Platter View)

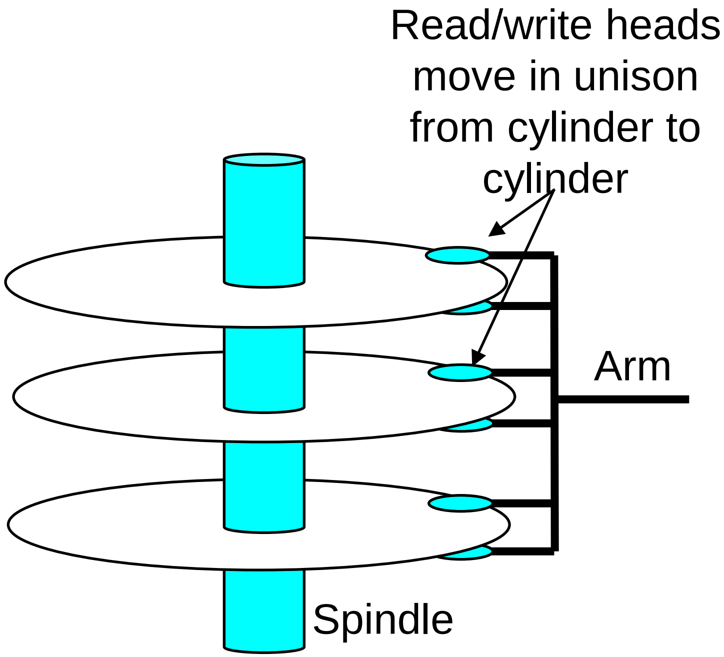
The disk surface spins at a fixed rotational rate




The read/write *head* is attached to the end of the *arm* and flies over the disk surface on a thin cushion of air.

By moving radially, the arm can position the read/write head over any track.

# Disk Operation (Multi-Platter View)



# Disk Access Time

- Average time to access some target sector approximated by :
  - Taccess = Tavg seek + Tavg rotation + Tavg transfer
- Seek time (Tavg seek)
  - Time to position heads over cylinder containing target sector.
  - Typical Tavg seek is 3—9 ms
- Rotational latency (Tavg rotation)
  - Time waiting for first bit of target sector to pass under r/w head.
  - Tavg rotation = 1/2 x 1/RPMs x 60 sec/1 min
  - Typical Tavg rotation = 7200 RPMs
- Transfer time (Tavg transfer)
  - Time to read the bits in the target sector.
  - Tavg transfer = 1/RPM x 1/(avg # sectors/track) x 60 secs/1 min.

# Disk Access Time Example

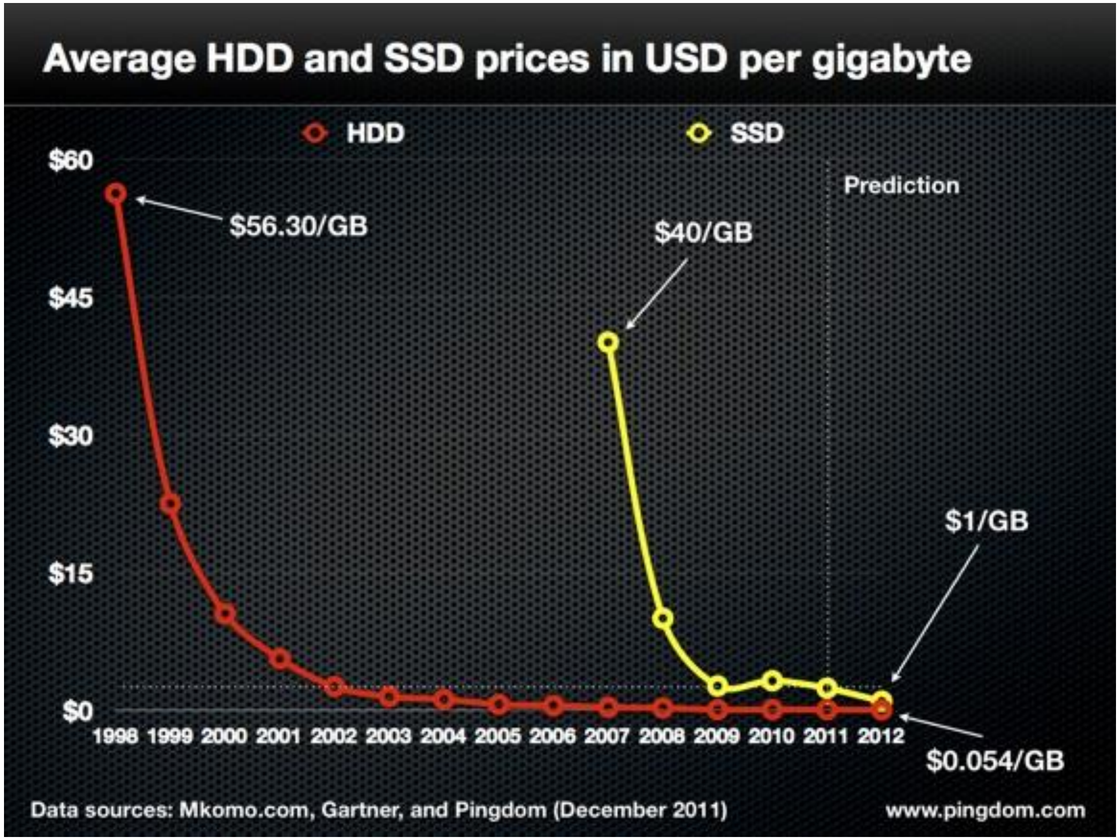
- Given:
  - Rotational rate = 7,200 RPM
  - Average seek time = 9 ms.
  - Avg # sectors/track = 400.
- Derived:
  - Tavg rotation = 1/2 x (60 secs/7200 RPM) x 1000 ms/sec = 4 ms.
  - Tavg transfer = 60/7200 RPM x 1/400 secs/track x 1000 ms/sec = 0.02 ms
  - Taccess = 9 ms + 4 ms + 0.02 ms
- Important points:
  - Access time dominated by seek time and rotational latency.
  - First bit in a sector is the most expensive, the rest are free.
  - SRAM access time is about 4 ns/doubleword, DRAM about 60 ns
    - Disk is about 40,000 times slower than SRAM,
    - 2,500 times slower than DRAM.

# Recent: Seagate Enterprise

- 24TB!
- 7 (3.5”) platters, 2 heads each
- 7200 RPM, 4ms seek latency
- 286/279 MB/sec read/write transfer rates
- 512MB cache
- $479

# Contrarian View

- FFS doesn't matter in 2012!




- What about Journaling? Is it still relevant?

# Samsung PM1643A MZILT30THALA-00007 30.72TB SSD SAS 12GBPS

Home / Solid State Drives / SAS 12GBPS SSD / 30.72TB-2.5Inch /

PM1643A Samsung MZILT30THALA-00007 30.72TB SSD SAS 12GBPS. New Sealed in Box (NIB) 3 Years Warranty

Mfg Part #: **MZILT30THALA-00007**

FREE DELIVERY


~~$8,170.00~~

**$6,240.00**

You save: $1,930.00 (24%)

Ask a question

**Quantity:** 1

ADD TO CART

QUOTE

# Storage Performance & Price

| | Bandwidth (sequential R/W) | Cost/GB | Size |
|---|---|---|---|
| HDD | 50-100 MB/s | $0.05-0.1/GB | 2-4 TB |
| SSD[1] | 200-500 MB/s (SATA)<br>6 GB/s (PCI) | $1.5-5/GB | 200GB-1TB |
| DRAM | 10-16 GB/s | $5-10/GB | 64GB-256GB |

[1] http://www.fastestssd.com/featured/ssd-rankings-the-fastest-solid-state-drives/

BW: SSD up to x10 than HDD, DRAM > x10 than SSD
Price: HDD x30 less than SSD, SSD x4 less than DRAM

# File system abstractions

- How do users/user programs interact with the file system?
  - Files
  - Directories
  - Links
  - Protection/sharing model

- Accessed and manipulated by a virtual file system set of system calls

- File system implementation:
  - How to map these abstractions to the storage devices
  - Alternatively, how to implement those system calls

# File system basics

- Virtual file system abstracts away concrete file system implementation
  - Isolates applications from details of the file system

- Linux vfs interface includes:
  - creat(name)
  - open(name, how)
  - read(fd, buf, len)
  - write(fd, buf, len)
  - sync(fd)
  - seek(fd, pos)
  - close(fd)
  - unlink(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
- 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

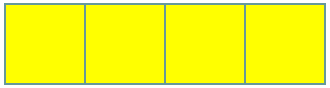
# Directory Internals

- 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

# Disk Layout Strategies

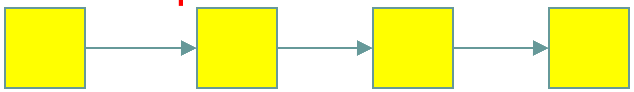
- Files span multiple disk blocks
- How do you find all of the blocks for a file?

1. Contiguous allocation



   - Like memory
   - Fast, simplifies directory access
   - Inflexible, causes fragmentation, needs compaction

2. Linked structure

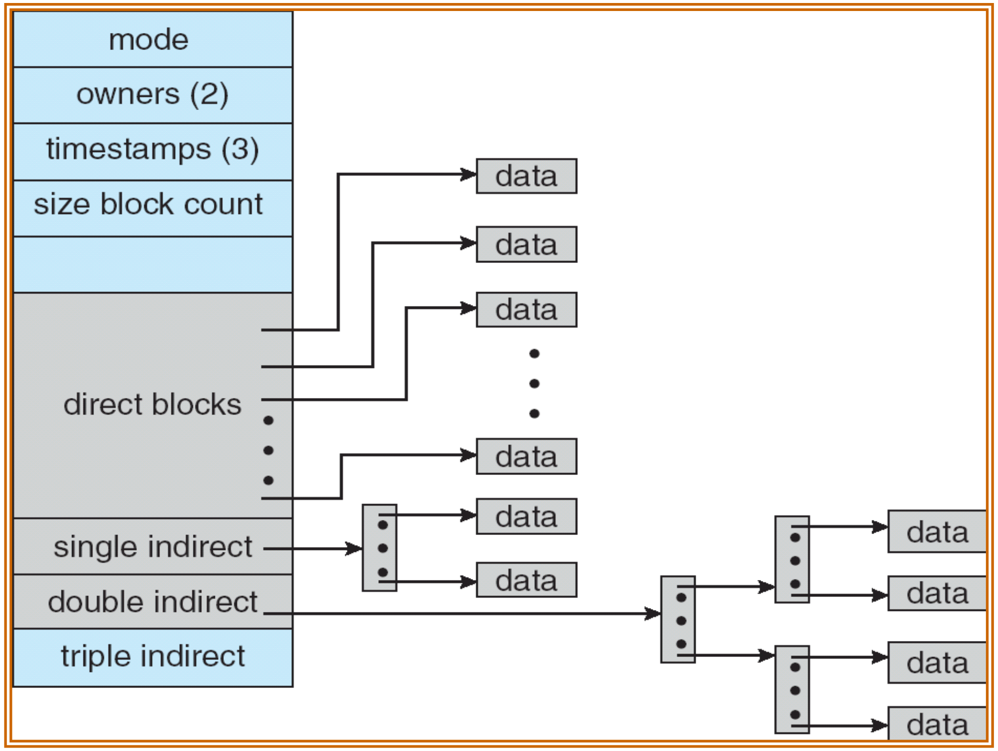


   - Each block points to the next, directory points to the first
   - Bad for random access patterns

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)

# Zooming in on i-node

- i-node: structure for per-file metadata (unique per file)
  - contains: ownership, permissions, timestamps, about 10 data-block pointers
  - i-nodes form an array, indexed by "i-number" – so each i-node has a unique i-number
  - Array is explicit for FFS, implicit for LFS (its i-node map is cache of i-nodes indexed by i-number)

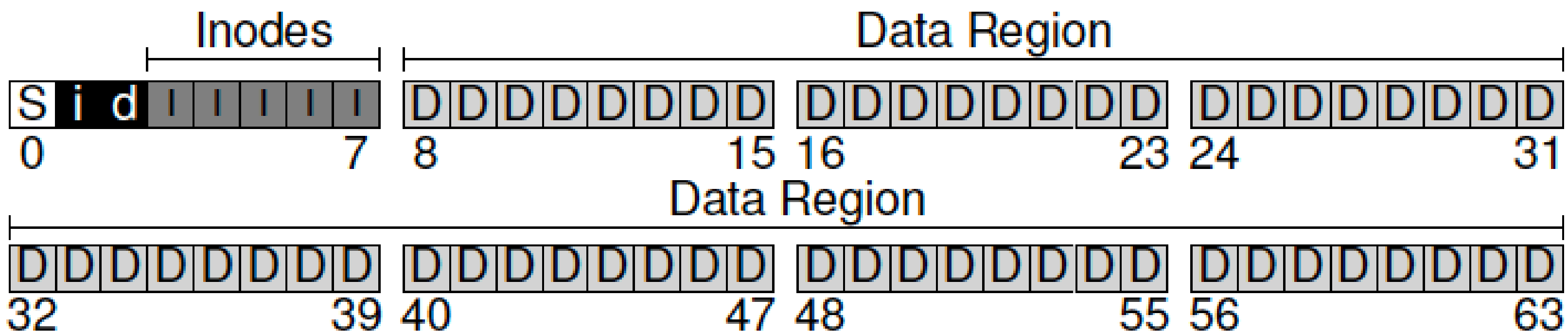


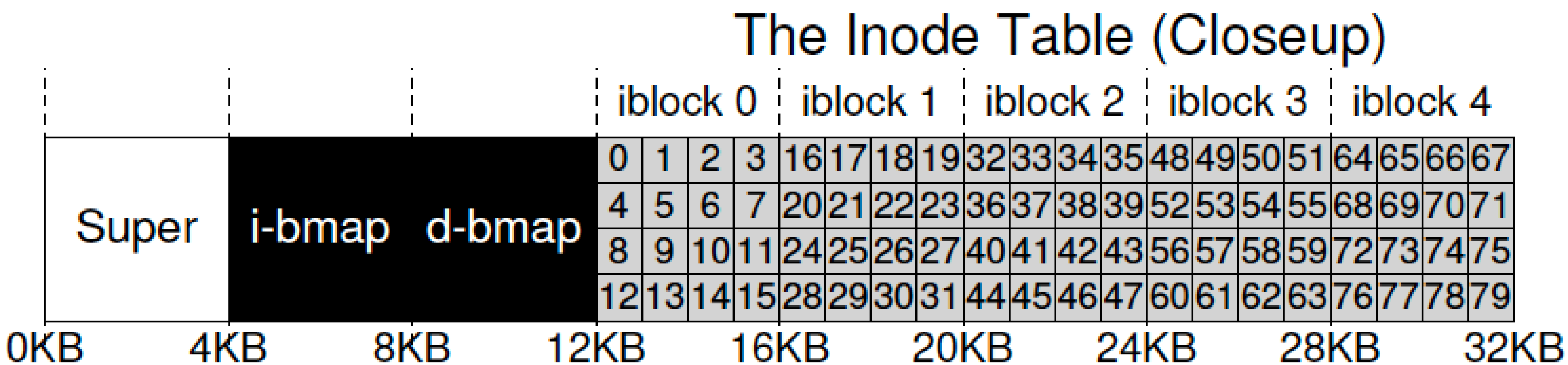

- Indirect blocks:
  - i-node only holds a small number of data block pointers (direct pointers)
  - For larger files, i-node points to an indirect block containing 1024 4-byte entries in a 4K block
  - Each indirect block entry points to a data block
  - Can have multiple levels of indirect blocks for even larger files

# Unix Inodes and Path Search

- Unix Inodes are not directories
- Inodes describe where on 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
- This is why we have *open* in addition to *read* and *write*

# A naïve implementation



# Directory Organization

| inum | reclen | strlen | name |
|---|---|---|---|
| 5 | 12 | 2 | . |
| 2 | 12 | 3 | .. |
| 12 | 12 | 4 | foo |
| 13 | 12 | 4 | bar |
| 24 | 36 | 28 | foobar_is_a_pretty_longname |

# File Read Timeline

| | data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data | bar data [0] | bar data [1] | bar data [2] |
|---|---|---|---|---|---|---|---|---|---|---|
| open(bar) | | | **read** | | | | | | | |
| | | | | | | **read** | | | | |
| | | | | **read** | | | | | | |
| | | | | | | | **read** | | | |
| | | | | | **read** | | | | | |
| read() | | | | | **read** | | | | | |
| | | | | | | | | **read** | | |
| | | | | | **write** | | | | | |
| read() | | | | | **read** | | | | | |
| | | | | | | | | | **read** | |
| | | | | | **write** | | | | | |
| read() | | | | | **read** | | | | | |
| | | | | | | | | | | **read** |
| | | | | | **write** | | | | | |

# File Write Timeline

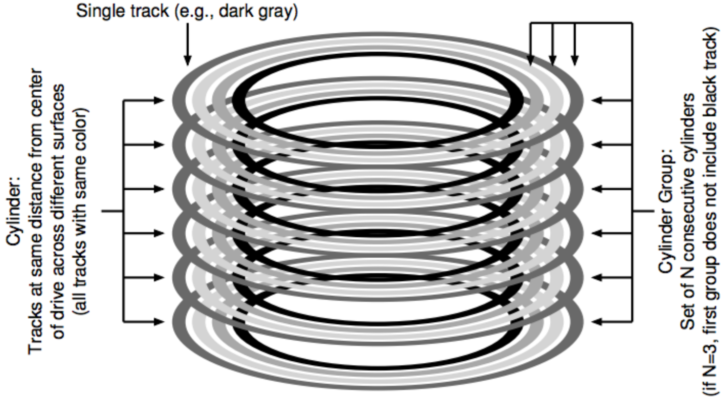
| | data bitmap | inode bitmap | root inode | foo inode | bar inode | root data | foo data | bar data [0] | bar data [1] | bar data [2] |
|---|---|---|---|---|---|---|---|---|---|---|
| create (/foo/bar) | | | read | | | | | | | |
| | | | | | | read | | | | |
| | | | | read | | | | | | |
| | | | | | | | read | | | |
| | | read | | | | | | | | |
| | | write | | | | | | | | |
| | | | | | | | write | | | |
| | | | | | read | | | | | |
| | | | | | write | | | | | |
| | | | | write | | | | | | |
| write() | | | | | read | | | | | |
| | read | | | | | | | | | |
| | write | | | | | | | | | |
| | | | | | | | | write | | |
| | | | | | write | | | | | |
| write() | | | | | read | | | | | |
| | read | | | | | | | | | |
| | write | | | | | | | | | |
| | | | | | | | | | write | |
| | | | | | write | | | | | |
| write() | | | | | read | | | | | |
| | read | | | | | | | | | |
| | write | | | | | | | | | |
| | | | | | | | | | | write |
| | | | | | write | | | | | |

# What's wrong with original unix FS?

- Original UNIX FS was simple and elegant, but slow
- Could only achieve about 20 KB/sec/arm; ~2% of 1982 disk bandwidth

- Problems:
  - Blocks too small
    - 512 bytes (matched sector size)
  - Consecutive blocks of files not close together
    - Yields random placement for mature file systems
  - i-nodes far from data
    - All i-nodes at the beginning of the disk, all data after that
  - i-nodes of directory not close together
  - no read-ahead
    - Useful when sequentially reading large sections of a file

# FFS Changes -- Locality is important


- Aspects of new file system:
  - 4096 or 8192 byte block size (why not larger?)
  - large blocks and small fragments
  - disk divided into cylinder groups
  - each contains superblock, i-nodes, bitmap of free blocks, usage summary info
  - Note that i-nodes are now spread across the disk:
    - Keep i-node near file, i-nodes of a directory together (shared fate)
  - Cylinder groups ~ 16 cylinders, or 7.5 MB
  - Cylinder headers spread around so not all on one platter

# FFS Locality Techniques

- Goals
  - Keep directory within a cylinder group, spread out different directories
  - Allocate runs of blocks within a cylinder group, every once in a while switch to a new cylinder group (jump at 1MB)
- Layout policy: global and local
  - Global policy allocates files & directories to cylinder groups – picks “optimal” next block for block allocation
  - Local allocation routines handle specific block requests – select from a sequence of alternative if need to

# FFS Results

- 20-40% of disk bandwidth for large reads/writes
- 10-20x original UNIX speeds
- Size: 3800 lines of code vs. 2700 in old system
- 10% of total disk space unusable (except at 50% performance price)
- Could have done more; later versions do
- Watershed moment for OS designers– File system matters

# FFS Summary

- 3 key features:
  - Parameterize FS implementation for the hardware it's running on
  - Measurement-driven design decisions
  - Locality "wins"
- Major flaws:
  - Measurements derived from a single installation
  - Ignored technology trends

- A lesson for the future: don't ignore underlying hardware characteristics

- Contrasting research approaches: improve what you've got vs. design something new

# File operations still expensive

- How many operations (seeks) to create a new file?
  - New file, needs a new inode
  - But at least a block of data too
  - Check and update the inode and data bitmap (eventually have to be written to disk)
  - Not done yet – need to add it to the directory (update the directory inode and the directory data block – may need to split if its full)…
  - Whew!! How does all of this even work?

- So what is the advantage?
  - Not removing any operations
  - Seeks are just shorter…