CS262: Advanced Topics in Computer Systems

Fall 2026

Lecture 3: File Systems

Rishabh Iyer

(adapted from slides by Stoica, Ghodsi, and Kubiatowicz)

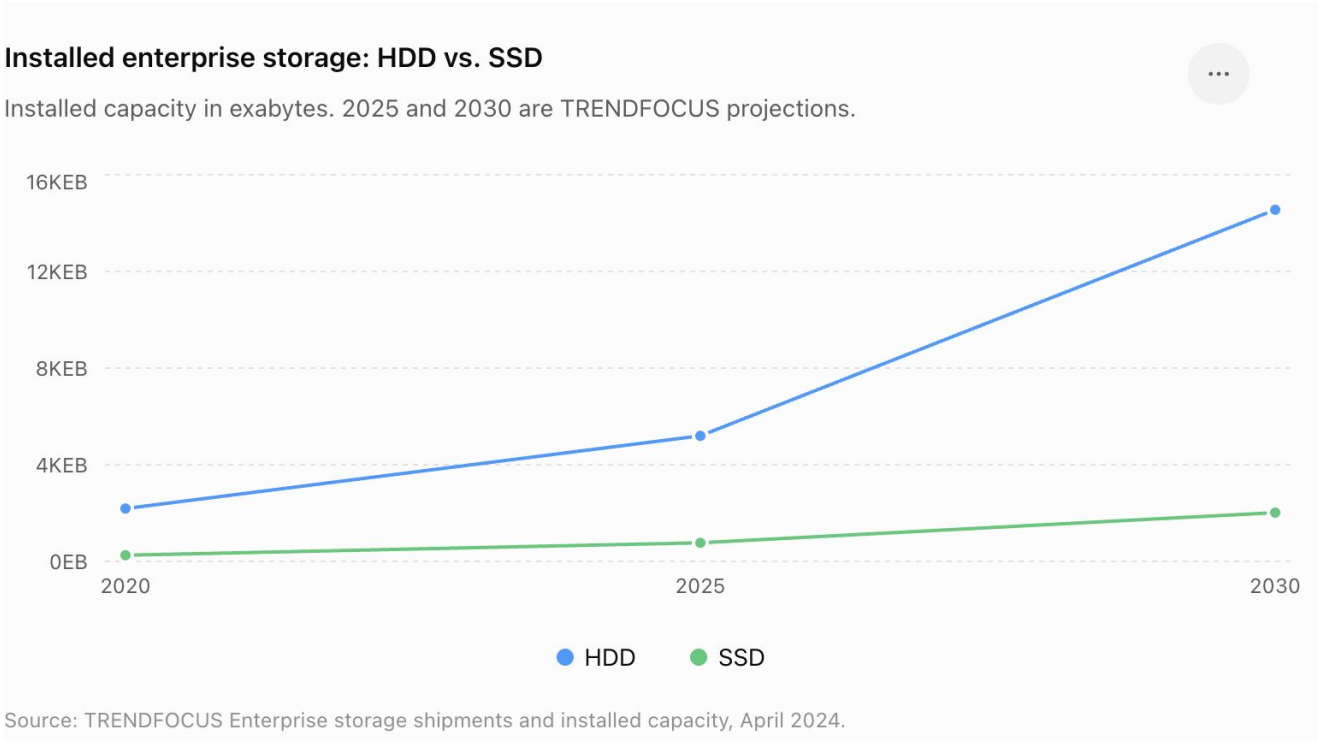
Today's Papers

- **A Fast File System for UNIX**
  - Marshall Kirk McKusick, William N. Joy, Samuel J. Leffler and Robert S. Fabry.
  - ACM Transactions on Computer Systems (TOCS) August 1984.

- **The Design and Implementation of a Log-Structured File System.**
  - Mendel Rosenblum and John Ousterhout,
  - 13th Symposium on Operating Systems Principles (SOSP) 1991.

Both papers were written at Cal!

Why Study File Systems for Hard Disks?



Why Study File Systems for Disks?

HDDs still used to store very large data sets cost effectively

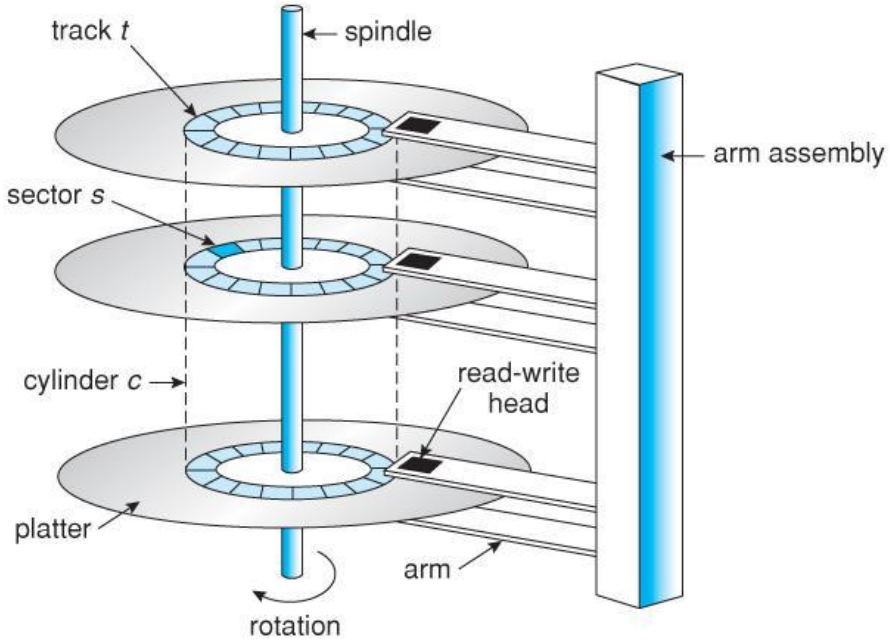
More and more distributed storage systems

- Performance patterns resemble disks.
- Low throughput for random access, high throughput for sequential access

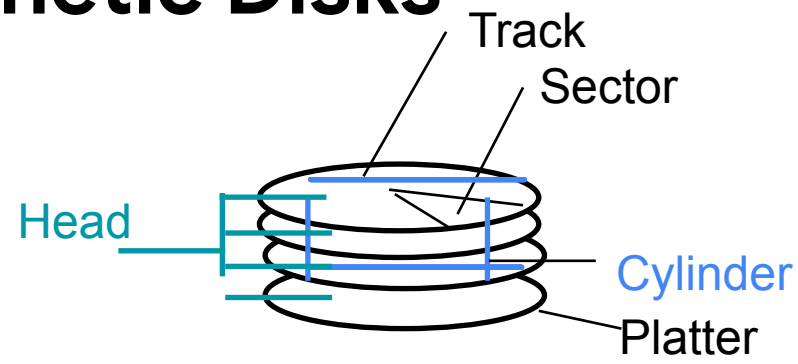
Great examples of system engineering

- Valuable lessons for other application domains

Background: Magnetic Disks



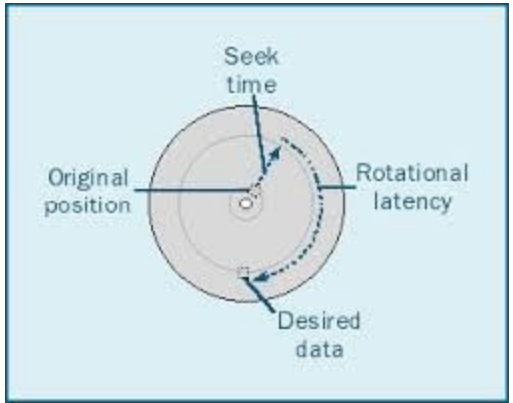
Background: Magnetic Disks



Cylinders: the set of tracks at the same radius across all platter surfaces.

Reading/writing data is a **three-stage** process:

- Seek time: position the head/arm over the proper track
- Rotational latency: wait for desired sector to rotate under r/w head
- Transfer time: transfer a block of bits (sector) under r/w head



Numbers from the 1980s:
Seek time = 30ms
One rotation = 8-16ms
(3600-7200 RPM)

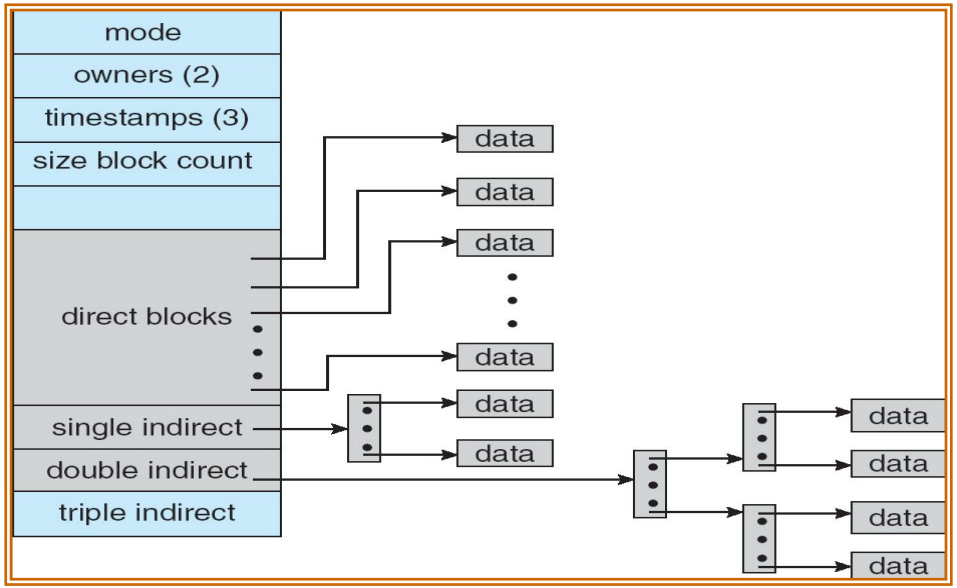
Some Useful Systems Constants

- Main memory latency: 100ns
- Main memory bandwidth: 20GBps per channel (large servers will have 5+ channels)
- HBM latency: ~200–500 ns
- HBM bandwidth: ~1 TB/s per stack; several TB/s per GPU
- SSD read latency: 100us
- SSD read bandwidth: 10 GBps
- Disk read latency: 10ms
- Disk bandwidth: 100MBps

File System Basics

i-node: per-file metadata (unique per file)

- contains: ownership, permissions, timestamps, ~10 data-block pointers
- i-nodes form an array, indexed by "i-number" – so each i-node has a unique i-number



Indirect blocks:

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

A Fast File System For Unix (FFS)

Why Did UNIX Need a New FS?

“The combination of the small block size, limited read-ahead in the system, and many seeks severely limits file system throughput.”

Why did the original FS have small block sizes, limited read-ahead, and incur many seeks?

What throughput was the original FS able to achieve?

Improving FS Performance

What if we just made the block size bigger?

What new challenges would it introduce?

- How does FFS solve those? What are the shortcomings in its solution?

What challenge does making the block size bigger **not** address?

What is the key idea that the designers of the new FS used to fix the second challenge?

Data Layout in FFS

How did the FFS pick which cylinder group to use:

- For a new directory
- New file in existing directory
- Large files?

Once a cylinder group has been chosen, how did it pick the blocks to store consecutive data chunks in? Is this idea still used today?

Why did the FFS need to maintain occupancy at <90%?

Key Results

Achieved two key goals:

- Improved disk bandwidth utilization by 10x.
- Performance stable over time.

Key Results

Achieved two key goals:

- Improved disk bandwidth utilization by 10x.
- Performance stable over time.

Some nuances:

- Where did the bottleneck move?
- The paper mentions two optimizations that it did not implement: block chaining and pre-allocation. What are these optimizations? Why didn't FFS implement them?
- Why was the old FS faster at writes than reads? Is this the case for FFS also?

FFS Takeaways

Three key ideas:

- Large blocks for throughput; fragments for space efficiency
- Cylinder groups + locality-aware allocation
- Parameterize placement for the underlying hardware

Limitations:

- Measurements derived from a single installation
- Didn’t fully anticipate technology trends, specifically CPU and DRAM scaling

Lessons:

- Don’t ignore underlying hardware characteristics. Locality almost always wins.
- Contrasting research approaches: improve what you’ve got vs. design something new (e.g., LFS)

Log-Structured File Systems (LFS)

Motivation

What were the key technological trends that motivated the design of LFS?

- Disk speed vs CPU speed
- DRAM size and the ability to absorb most reads.
- Disk latency (mechanical) vs bandwidth (largely electrical)

What were the key changes in workload characteristics that motivated the design?

- Workloads were dominated by accesses to small files

Why is FFS a Poor Fit?

Lots of little writes.

- Since FFS physically separates different files/directories, this turns into lots of seeks.
- E.g.: 5 seeks for creating a new file. Say you want to `create foo`, you have to seek:
  - new file inode
  - new file data block
  - parent directory data block // add "foo → inode#"
  - new file inode again // recovery ordering
  - parent directory inode // mtime, etc.
- Seeks may be within cylinder group, but they are still seeks.

Writes are synchronous

- Metadata writes are synchronous to simplify crash recovery.

LFS: Key Idea In a Nutshell

Traditional file systems optimize for spatial locality

- Place data based on logical order (e.g., files near directory, consecutive file blocks)

LFS optimizes for temporal locality

- Place data together based on what is modified at the same time.
- Buffer many updates and write them as one large sequential transfer.

Key design implication:

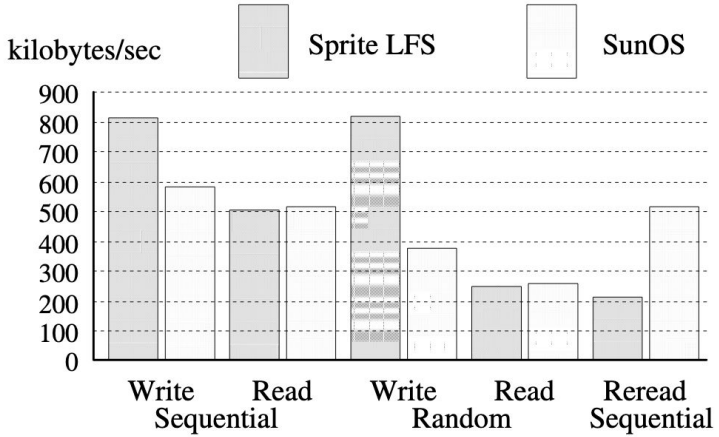
- The log is the *only* structure on disk.
- Completely new design point in the space of file system designs.

Intuiting Performance Tradeoffs

Assume the following benchmark run on both LFS and FFS:

1. Create a 100 MB file with **sequential writes**
2. Read it back **sequentially**
3. Overwrite 100 MB of it with **random writes**
4. Read 100 MB **randomly**
5. Read the whole file **sequentially again** — this is called **“re-read”**

How do you think LFS would perform vs FFS for each phase?



LFS Design Challenges

What are the two main challenges in designing/implementing an LFS?

- Floating inodes.
- How to ensure the disk runs out of space? LFS needs large contiguous blocks and needs to avoid fragmentation.

Floating inodes

When you create a small file (less than a block):

- Write data block to memory log
- Write file inode to memory log
- Write directory block to memory log
- Write directory inode to memory log

No seek for writes, but inodes are now all over the place, i.e., **floating**!

Solving Floating inodes

Need to keep track of current position of inodes. Solution: use an “inode-map”!

- But inode-map could be large (as many entries as there are files in the file system)
- Break inode-map into chunks and cache them
- Write out on the log those chunks that have changed.

But, wait, isn’t the inode-map also now floating?

Solving Floating inode maps

Solution: inode-map-map, i.e., a map of inode map.

- Small enough to be always cached in memory
- Small enough to be written to a fixed position on disk, called the **checkpoint region.**

What are the disadvantages of this design?

- Write amplification: In their production /user6 filesystem, only 0.2% of the live disk contents were inode-map blocks, but inode-map updates consumed 7.8% of log bandwidth.
- Implementation complexity.

Solving Floating inode maps

Solution: inode-map-map, i.e., a map of inode map.

- Small enough to be always cached in memory
- Small enough to be written to a fixed position on disk, called the **checkpoint region.**

What are the disadvantages of this design?

- Write amplification: In their production /user6 filesystem, only 0.2% of the live disk contents were inode-map blocks, but inode-map updates consumed 7.8% of log bandwidth.
- Implementation complexity.

Managing Free Space

Need to compact live data to open up large runs of free space

- Problem: long-lived information gets copied over-and-over

Potential solution:Thread log through free spaces

- Problem: disk fragments, causing I/O to become inefficient again

Their solution: segmented log

- Divide disk into large, fixed-size segments
- Do compaction within a segment; thread between segments
- When writing, use only clean segments (i.e. no live data)
- Occasionally clean segments.
- Maintain segment summary block(s) for each segment

Which Segment to Clean?

What was their initial greedy policy? Why was it reasonable?

What did their simulation results show? What was the root cause?

How does their updated “cost-benefit” policy work? What data structure did they add to ensure that this policy can be implemented efficiently?

LFS Summary

Optimize for temporal locality using batched writes to a log. Log is the only structure on disk.

Designed for write-heavy workloads (assuming DRAM absorbs majority of reads) and small files.

Key tradeoff with FFS:

- LFS pays cleaning overhead; FFS pays fragmentation overhead.
- Depending on workload and configuration, either can win.
- (for more details, refer to the Ousterhout vs Seltzer debates)

Discussion

What do the majority of file systems (e.g., `ext4`) for hard drives use today? Are they closer to FFS or LFS? What ideas did they take from each system?

- For more details, refer to the the optional reading on Journaling File Systems

How would you re-design file systems for SSDs? What is the primary architectural shift and what is the implication?

- For more details, look up the design of F2FS.

The FFS paper claims that the posix interface was not the problem, only the implementation is. Can you think of modern (circa 2008) hardware changes that make the **interface** also a problem?

- For more details, look up the Scalable Commutativity Rule.