THE DESIGN & IMPLEMENTATION OF A LOG-STRUCTURED FILE SYSTEM

- Mendel Rosenblum & John K. Ousterhout.
- Published in the ACM Symposium on operating Systems Principles (SOSP '91)
- and ACM Transactions on Computer Systems (TOCS '92)

★ Was Mendel Rosenblum's Ph.D. Thesis at UC-Berkeley

★ Won the ACM Doctoral Dissertation Award.

- First things first - why are we studying a paper from over 30 years ago?

  - Why not? It's a classic! An example of elegant systems design motivated by technology trends of the time.

  - To see an example of great quality research - the one that can have <u>impact</u>.

  - You'll see the same ideas reused over 25 years later, in the design of file systems for SSDs.

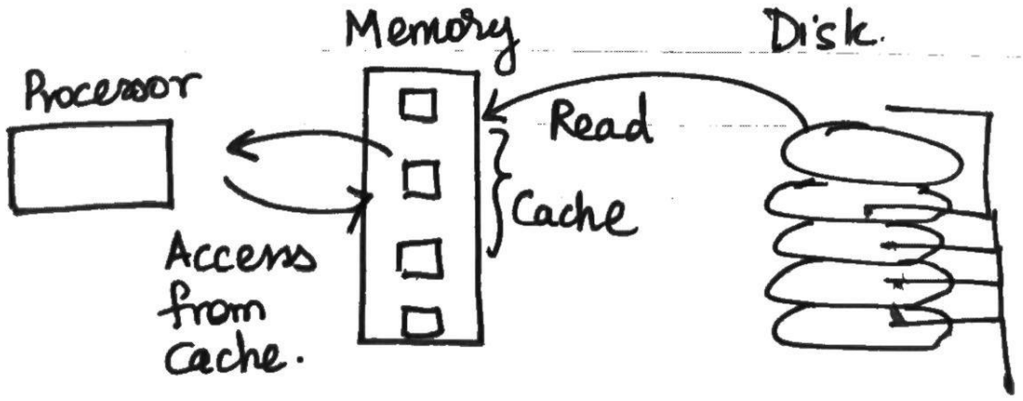
- So, why a new file system? Why doesn't FFS suffice?
  * FFS still left room for performance improvement
  * FFS only addresses the problem from a single perspective - placement of data on disk, and not a holistic perspective.

Let us look at the technology trend in the late 1980s and early 1990s:-

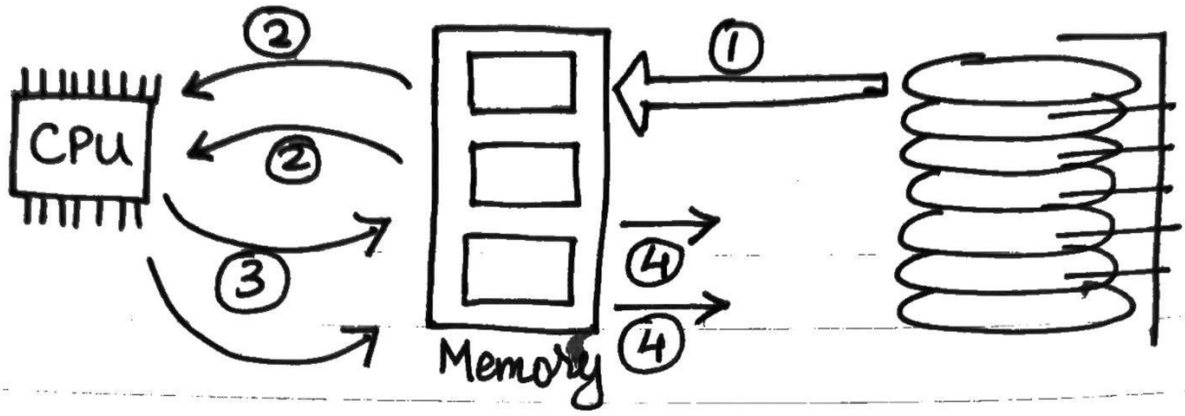
* Disks were getting fast (i.e. more bandwidth) but seek & rotational delays still large.
* Memory was getting cheaper, and therefore more abundant. Bandwidth also increasing.
* Processors speed increasing exponentially and capable of running very fast

⇒ More often than not, disk I/O is bottleneck.

Abundant memory ⇒ can use memory as a cache.



Now think what happens if you have a large memory-based disk cache:



- On first access of a piece of data from disk ①, fetch a large amount of "related" data from disk and cache it in memory.

- All subsequent accesses to disk can be quenched from memory alone. No disk access (or only when new data not in ~~disk~~ cache is accessed).

- So reads largely serviced from memory.

- But what about writes?
  - Can still write to memory cache. ③
  - But these writes will still need to be persisted to disk periodically. ④
  - If you look at disk traffic, it will be dominated by write traffic.
  - Reads once in a while, but writes mostly (flushing large chunks of dirty data from cache to disk)

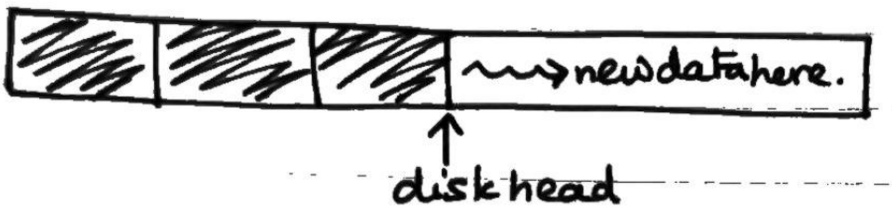
- When you have disk traffic largely dominated by writes what is the bottleneck? Think from FFS perspective:
  - Most of the time will go seeking and rotating to the location of the data on disk.
  - Becomes main bottleneck.

- How to avoid? Change the file system layout on disk to avoid seeks & rotational delays almost completely.

> Idea! Make the file system an append-only log. All writes to the end of the log. Voila! No more seeks/rotational delay on a new write.

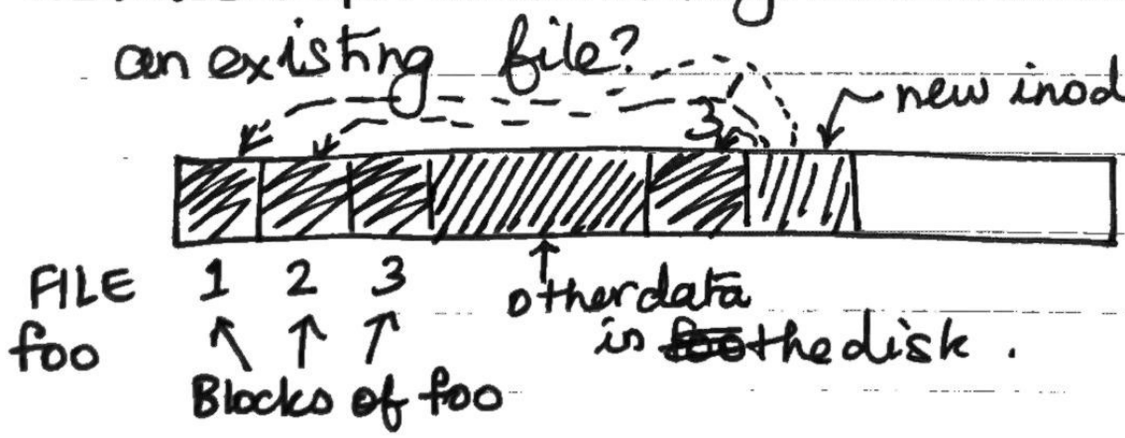
- Workloads (of the early 1990s) + FS characteristics.
  - Office & engineering workloads.
  - Lots of random disk accesses on small files
  - File creation/deletion time dominated by directory & inode updates
  - I-nodes far from data ⇒ < 5% of disk bandwidth actually used for data xfer.
  - All inode writes are synchronous

Key idea: All modifications to disk written in a sequential log-like fashion.



- Adv: converts many small random writes into a large sequential write on disk.
- Memory cache is simply a write buffer
- Fast recovery on crash. No need to search full disk, new data written found at end of log!
  ⇒ things like fsck can work much faster.

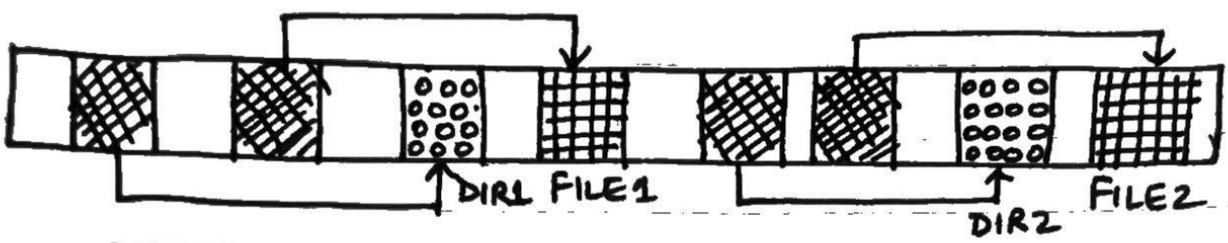
So, LFS takes the log idea to the limit
⇒ EVERYTHING in the log, including data & inodes.
⇒ CHALLENGE: what if you access & modify an existing file?

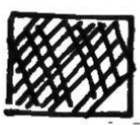


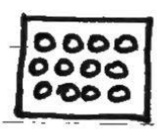
Say, you modify data in block 3.
Add a new entry to log! 3'
How to index it? Write new inode!

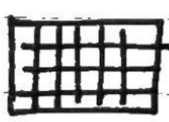
So, how does disk end up looking?

On FFS:-

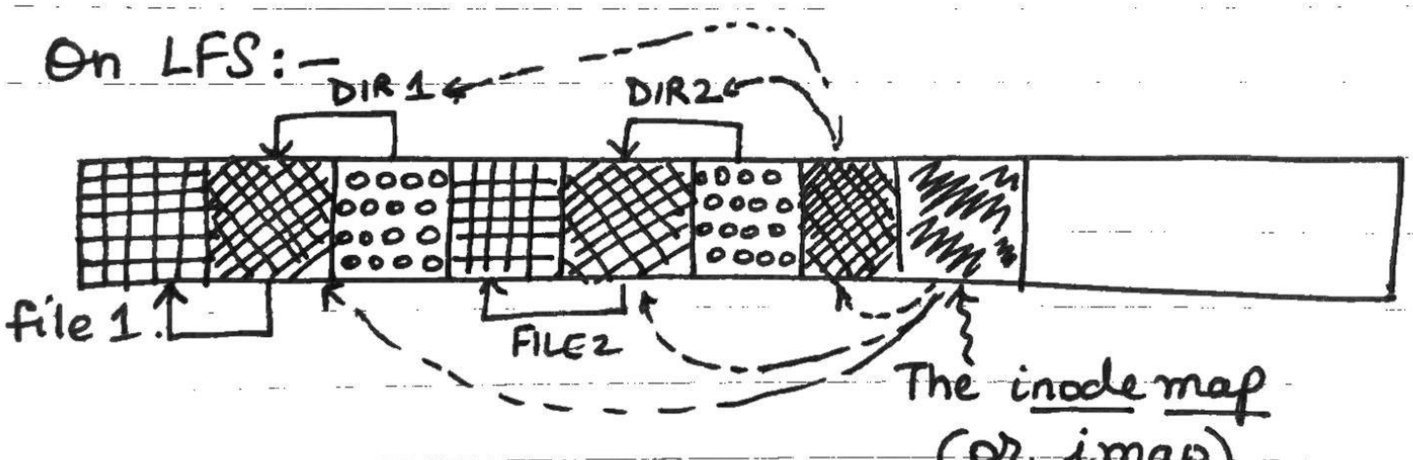


 → Inodes.

 → Directories.

 → Data blocks of files.

On LFS:-



The imap is the location of all active inode blocks. imaps themselves stored in log. written out each time a file block is modified. Active blocks cached in main memory.

Checkpoint regions contain addresses of imap blocks at checkpoint time.

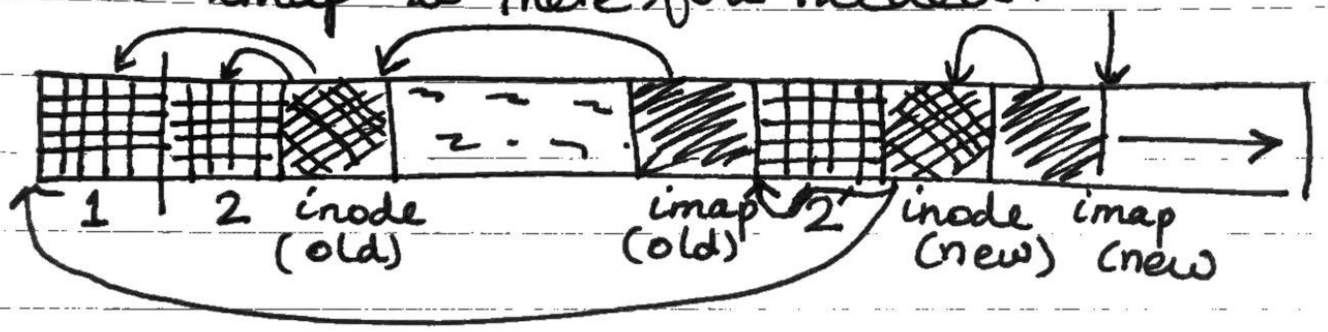
Updating a file. Example.

Say your write modifies data in block 2 of the file.

→ new block 2' is generated.

→ new inode is now needed

→ new imap is therefore needed.



→ The old inode & olad imap are garbage

→ Must garbage collect from time to time else disk gets filled with garbage.

→ How to do?

MANAGE FREE SPACE USING SEGMENTS.

Let's think how to manage free space.
Really, only two options:

(1) Threading the log.

(2) Copying.

What LFS needs: large free extents on disk on which to grow the log.
→ But log keeps growing, garbage left in log, so disk can get full.
→ Then, no large extents will remain!

Threading = "Thread the log" around old blocks.

Copying = "copy all old, active data in a compact form from time to time.

Look at Figure 2 in paper for an illustration.

What LFS does:
→ Divide disk into fixed, large-sized extents called segments
→ Each segment written to in "log-like" fashion.
→ Segments can be copied to other segments from time to time, compacting along the way by discarding garbage.
→ This segment can then be reused.
→ Segments linked up by threaded structure.

→ Segment operations therefore can use up full disk bandwidth!

Garbage collection = Segment cleaning.

1. Copy a full segment to memory
2. identify live data (how?)
3. write out a buffer with all live data
4. flush this out to disk in a new segment.

A Segment Summary block helps identify easily which blocks of a segment are live, which block belongs to which file, etc.

When to clean segments?

→ How many segments to clean?
→ Which segments to clean?
→ When cleaning how to group live blocks?

} POLICY DECISIONS

Quantified by WRITE COST

→ average time disk is busy per new byte written, including cleaning overheads.

$$= \frac{\text{Total bytes read \& written}}{\text{new data written}}.$$

So, lower write cost is better. Lowest = 1.0.

Depends very much on the utilization (% of live data) in segments cleaned.

Steady State: cleaner must produce one empty segment / new segment written.
How to produce empty segments?
→ Read N old segments, compact & write out.

$$\text{write cost} = \frac{\text{read segs} + \text{write live} + \text{write new}}{\text{write new}}$$

$$\frac{N + N*u + N(1-u)}{N(1-u)} = \frac{2}{1-u}$$

So, write cost grows as u (disk utilization) increases.

Please see paper for thorough discussion on various segment cleaning policies, quantified by write cost - greedy vs. Hot & cold!
→ STUDY THIS!

CRASHES & RECOVERY

- Only need to inspect end of log!
- Correct the inconsistencies.
- Checkpoints + Roll-forward.

CHECKPOINT = position at log when file system is consistent.
LFS writes out to a fixed location on disk called checkpoint region

Checkpoint region has addresses of all blocks in inode map, and pointer to last segment written.

Just restoring from checkpoint restores the FS to a consistent state. But roll-forward does more, by trying to recover recently-written data (i.e, after last checkpoint).

⇒ Look at segment summary block
⇒ Any new inodes? Fetch data blocks they point to.

Performance metrics:

→ Sequential Read & write performance.
→ Random Read & write performance.