Log Structured File System

- a type of copy-on-write file system
- goal: optimize for write performance on disk
  - why not reads? assume cache can serve most reads
- isn’t FFS also designed for good disk performance?
  (in place file system)
- block groups, place related data in the same block group.
- how does this impact seek time & rotational time?
  - nearby tracks, smaller arm movement
  - still several ms structures (inodes, bitmaps, data blocks)
  - still live on noncontiguous disk blocks, each write incurs its own rotational delay (wait for desired sector to spin under the disk head)
How does LFS optimize for write performance?

- Reduce seek & rotational time
  - almost only write large sequential chunks (1 seek + 1 rotational delay)

- Core structure of the file system is a sequential log

- All updates are buffered in memory until it fills up a segment (several MBs)

- When a segment is ready, append to the log

  all modified blocks are written into the log, source of truth = latest version in the log

- Avoid the recursive update problem w/ a level of indirection

  inode map: inode # → block #

  (sharded into many pieces, each piece track a disjoint range of inode mappings)
Inode map: keeps inode location on disk
- normally completely cached in memory
  - easy for reads, inode map -> inode -> data block
- but how do we find all pieces of inode map upon start up?
  - tracked by the **Checkpoint Region** (CR)

* the only structure that lives in a fixed location

<table>
<thead>
<tr>
<th>CR</th>
<th>Log</th>
</tr>
</thead>
</table>

  - written periodically (30s), not upon each segment write! (might be stale)

  - tracks a consistent snapshot of a fs state
  - stores head & tail segments (range of log entries that makes up the fs state)
  - stores location of inode map pieces

→ writing to LFS: log is a circular log, just append to the tail segment, no need to track bitmap
Crash Recovery

- on boot, reads from the latest valid checkpoint region
- how's CR updated?
  - can span over multiple blocks
  - matching timestamp = valid/complete CR
- wait... isn't CR updated infrequently?
- if we only reserve 1 loc for CR
  - then we can detect invalid CR
  - but would also lose a consistent CR due to overwrite!
- reserve space for 2 CRs!
  - overwrite the invalid or older CR for every update

CR tracked region

valid

can keep applying segments post the CR tail
to roll forward the system
Garbage Collection

→ Segment

Some blocks are live & some are garbage

Each has a segment summary (sometimes multiple...)
Tracking each data block's inode # & offset

→ Compact live blocks

Within multiple segments, write into a new segment!