File system trace papers

- **The Design and Implementation of a Log-Structured File System. M. Rosenblum, and J.K. Ousterhout. ACM TOCS. Vol. 10, No. 1 (Feb 1992), pp. 26-52**

Log structured file system

- Problem being addressed:
  - CPU is fast, memory is plentiful, disk seek slow
    - memory can catch most of the (repeated) reads
  - Small file writes suffer
- Writes cannot be buffered and delayed indefinitely because of data safety
  - Writing a small file can take five disk operations, read data/index for directory entry and read/write of inode entry (update meta information such are last modified) before writing actual data = 5% utilization
  - Meta data updates are synchronous to ensure consistency on crash
  - Crash recover is linear because we have to go through all entries to ensure consistency (fsck)

lfs

- You collect all writes and write it all at once to disk, paying for a single seek
- Another approach is for file systems to maintain a separate log and data area: all updates are written into the log and are eventually moved into the non-log areas. This paper talks about a system that only contains logs; there are no “other” areas
- They show that for small writes, they are significantly better, for other cases they are similar to ffs

Disk layout

- Super block contains segment information. Segments are large collection of blocks – amortize the seek cost by transfer large amounts of data
- Checkpoint region: inode maps are kept in memory for performance and periodically flushed to the checkpoint log
- Segments: version, offset of each block
- Log: data block locations
- Inode map: inode locations, inode versions
- Segment usage table: bytes free, write times

Locating data for a file

- Inode map gives you inode, which give you data
- Checkpoint: flush everything to disk (data blocks, indirect blocks, inodes, inode map table, segment usage table)
- Current time, pointer to last written segment
- Two check point regions to protect against crashes, use latest checkpoint for recovery (write timestamp last)
- How often should you checkpoint: LFS does every 30 seconds – research topic

Crash recovery

- On crash, need inode map and segment usage table
- Read latest version from checkpoints
- Roll forward to get data written between checkpoints
  - If we find data blocks that belong to a inode, use them
  - Otherwise, file blocks belong to an incomplete write
- Use directory log to recover directory operations (file creates) except when no inode is found for a directory create

Free space management

- Fragmentation is a problem
  - Solution 1: Threading on a segment level avoiding useful data
  - Solution 2: Compact segments using a periodic cleaner
- LFS uses both
- Cleaner:
  - Segment summary tells what files block belong to
  - Check file inode blocks to see if data block is still pointed
  - Inode version numbers can help when inode is recycled
- How often to run cleaner
  - LFS prefers bimodal distribution: empty or full segments

Anamolies

- Hot and cold performs worse
  - Cold segments linger
  - Hot segments frequently cleaned
  - Solution: to clean segment, 1 seg read + write of live data
  - Last segment write in segment table
- LFS is worse than FFS for random writes and sequential reads
  - Real world is better because large files are written and deleted all at once and many files are never rewritten

Current technology

- Log meta data
- Is memory large compare to disk size in modern computers?