Log-Structured File Systems

CSC 343, Operating Systems

Topics covered in this lecture

- Log-structured file systems

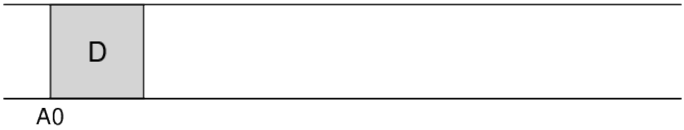
This slide deck covers chapters 43 in OSTEP.

LFS: Log-structured File System

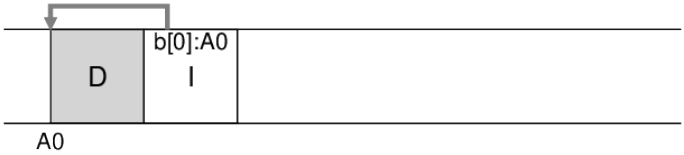
- Motivation
  - Memory sizes were growing
  - Large gap between random IO and sequential IO performance
  - Existing file systems perform poorly on common workloads
  - File systems were not RAID-aware

Writing to Disk Sequentially

- How do we transform all updates to file system state into a series of sequential writes to disk?
  - Data update

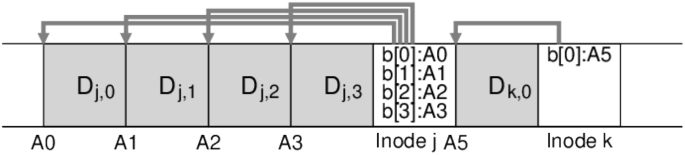


  - Metadata needs to be updated too



Writing to Disk Sequentially and Effectively

- Writing single blocks sequentially does not guarantee efficient writes
  - After writing into A0, next write to A1 will be delayed by disk rotation
- Write buffering for effectiveness
  - Keeps track of updates in memory buffer (also called segment)
  - Writes them to disk all at once, when it has sufficient number of updates.



How Much to Buffer?

- Each write to disk has fixed overhead of positioning
  - Time to write out D MB

$$T_{write} = T_{position} + \frac{D}{R_{peak}}$$

  where $T_{position}$ is positioning time and R_{peak} is disk transfer rate.

- To amortize the cost, how much should LFS buffer before writing?
  - Effective rate of writing can be denoted as

$$R_{effective} = \frac{D}{T_{write}} = \frac{D}{T_{position} + \frac{D}{R_{peak}}}$$

How Much to Buffer?

- Assume that $R_{effective} = F \times R_{peak}$, where F is a fraction of peak rate, $0 < F < 1$, then

$$R_{effective} = \frac{D}{T_{position} + \frac{D}{R_{peak}}} = F \times R_{peak}$$

- Solve for D

$$D = \frac{F}{1 - F} \times R_{peak} \times T_{position}$$

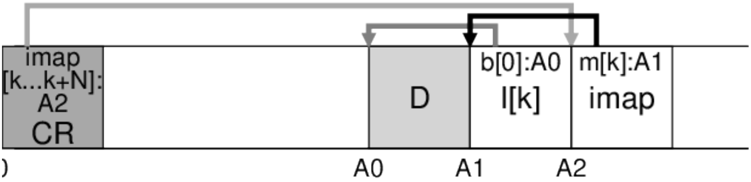
- If we want F to be 0.9 when $T_{position} = 10ms$ and $R_{peak} = 100\frac{MB}{s}$, then $D = 9MB$
  - That is, the segment size should be at least 9MB.

Finding Inodes in LFS

- Inodes are scattered throughout the disk.
- Solution is through indirection "Inode Map" (imap)
- LFS places the chunks of the inode map right next to where it is writing all of the other new information

The Checkpoint Region

- How to find the inode map when it is spread across the disk?
  - The LFS has a fixed location on disk to begin a file lookup
- The Checkpoint Region contains pointers to the latest of the inode map
  - Only updated periodically (for example, every 30 seconds)

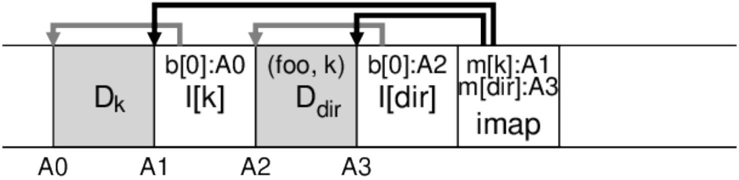


Reading a File from Disk

- Read the checkpoint region
- Read the entire inode map and cache it in memory
- Read the most recent inode
- Read a block from the file using direct or indirect pointers

What About Directories?

- Directory structure of LFS is basically identical to classic UNIX file systems
  - A directory is a file which data blocks consist of directory information

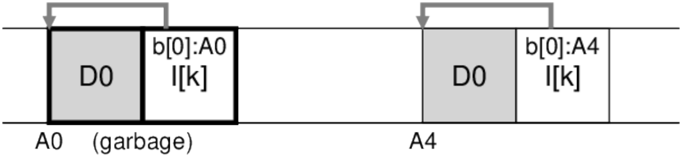


Garbage Collection

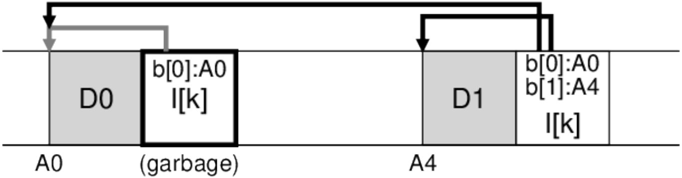
- LFS keeps writing newer version of file to new locations
- LFS leaves the older versions of file structures all over the disk, we call this garbage.

Examples: Garbage

- For a file with a single data block
  - Overwrite the data block: both old data block and inode become garbage



  - Append a block to that original file k: old inode becomes garbage



Handling Older Versions of Inodes and Data Blocks

- One possibility: versioning file system
  - keep the older versions around
  - users can restore old file versions
- LFS approach: garbage collection
  - keep only the latest live version and periodically clean old dead versions
  - segment-by-segment basis
    - a block-by-block basis collector eventually makes free holes in random locations so writes cannot be sequential

Determining Block Liveness

- Segment summary block (SS)
  - Located in each segment
  - Inode number and offset for each data block are recorded
- Determining Liveness
  - The block is live if the latest inode indicates the block

  - Version number can be used to determine liveness efficiently

Which Blocks to Clean and When?

- When to clean
  - Periodically
  - During idle time
  - When the disk is full
- Which blocks to clean
  - Segregate hot/cold segments
    - Hot segment: frequently overwritten
    - Cold segment: relatively stable
  - Collect cold segments sooner and hot segments later

Crash Recovery and the Log

- Log organization in LFS
  - CR points to a head and tail segment
  - Each segment points to next segment
- LFS can easily recover by simply reading the latest valid CR
  - The latest consistent snapshot may be quite old
- Ensuring the atomicity of the CR
  - Keep two CRs
  - CR update protocol: timestamp $\rightarrow$ CR $\rightarrow$ timestamp
- Roll forward
  - Start from end of log (pointed to by the latest CR)
  - Read next segments and adopt any valid updates to the file system