43. Log-structured File Systems

Operating System: Three Easy Pieces

LFS: Log-structured File System

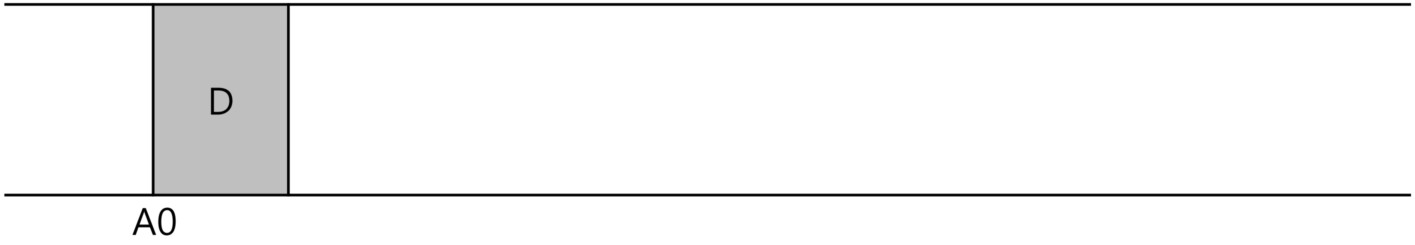
- Proposed by Stanford back in 91

- Motivated by:
  - DRAM Memory sizes where growing.
  - Large (growing) gap between random IO and sequential IO performance (seek times vs bandwidth)
  - Existing File System perform poorly on common workloads.
  - File System were not RAID-aware (small-write problem in RAID-4/5)

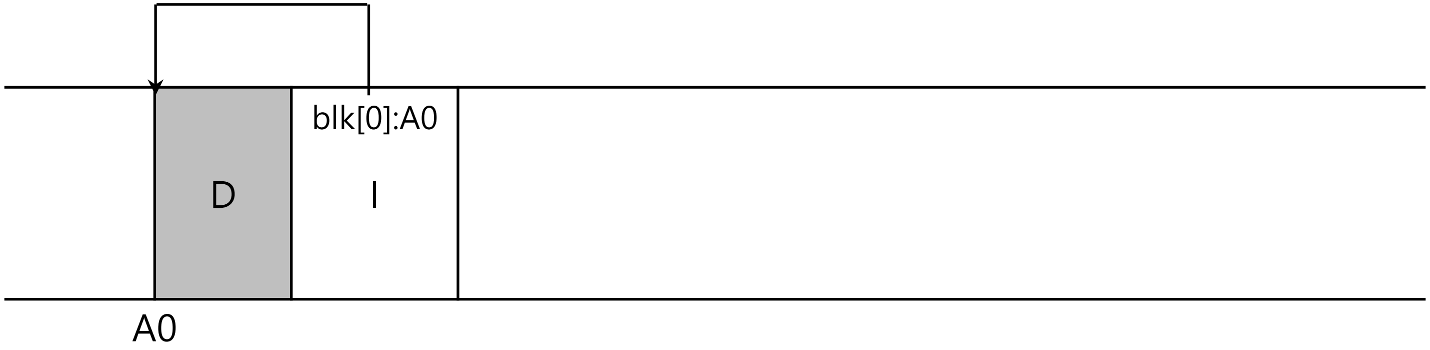
- **Transform disk bandwidth into latency reduction!**

Writing to Disk Sequentially

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

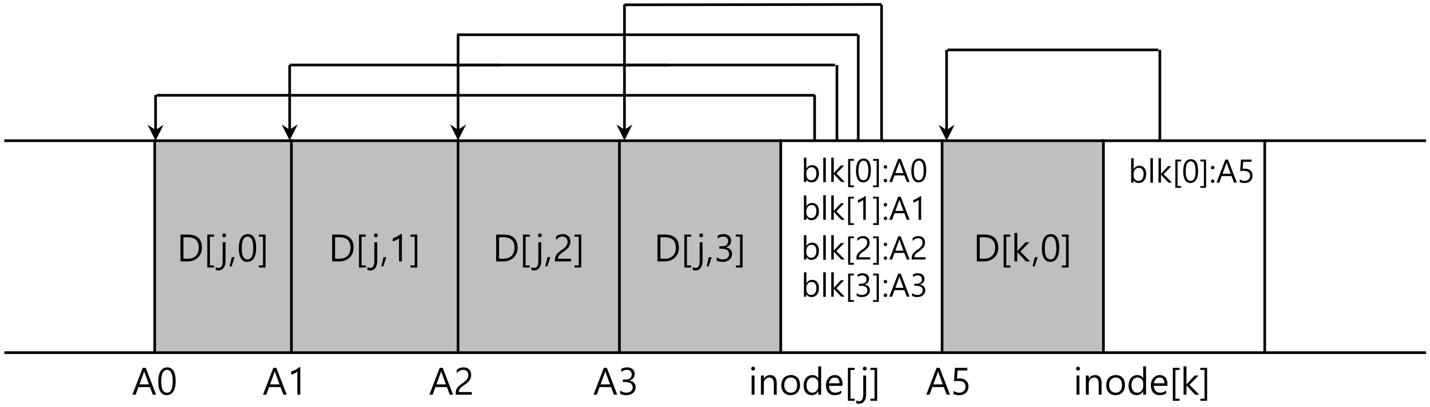


  - metadata needs to be updated too. (Ex. inode)



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 (or the user instruct to do so, i.e., call fsync)



How Much to Buffer?

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

$$T_{write} = T_{position} + \frac{D}{R_{peak}} \quad (43.1)$$

($T_{position}$: positioning time, R_{peak}: disk transfer rate in MB/s)

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

$$R_{effecitve} = \frac{D}{T_{write}} = \frac{D}{T_{position+} \frac{D}{R_{peak}}} \quad (43.2)$$

How Much to Buffer?

- Assume that $R_{effecitve} = F \times R_{peak}$ (F: fraction of peak rate, 0 < F < 1), then

$$R_{effecitve} = \frac{D}{T_{position} + \frac{D}{R_{peak}}} = F \times R_{peak} \ (43.3)$$

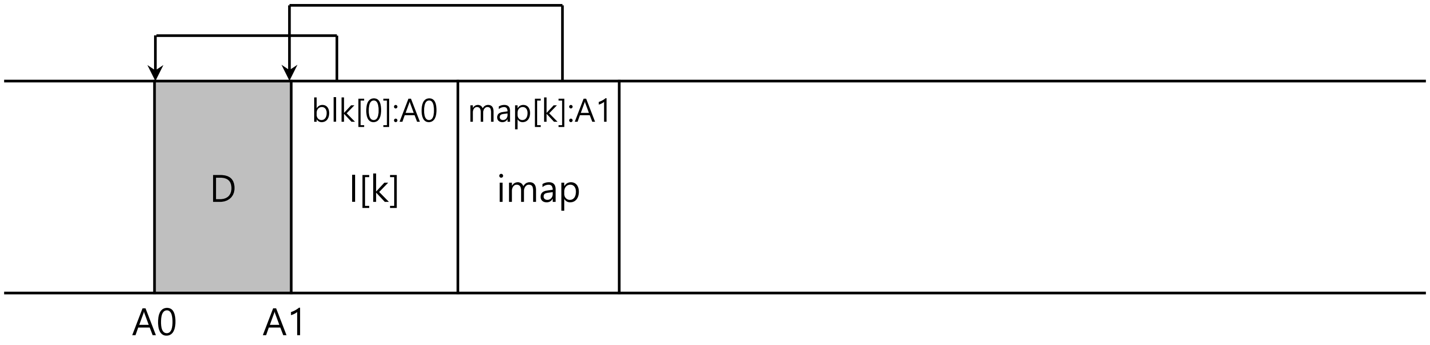
- Solve for D

$$\mathrm{D} = \frac{F}{1-F} \times R_{peak} \times T_{position} \ (43.6)$$

- If we want F to be 0.9 when $T_{position} = 10msec$ and $R_{peak} = 100MB/s$, then $D = 9MB$ by the equation.
  - Segment size should be 9MB at least.

Finding Inode in LFS

- **Problem**: Inodes are scattered throughout the disk! (and the last version keep moving)
- **Solution:** is through indirection "Inode Map" (imap)
- LFS place **the chunks** of the inode map right next to where it is writing all of the other new information

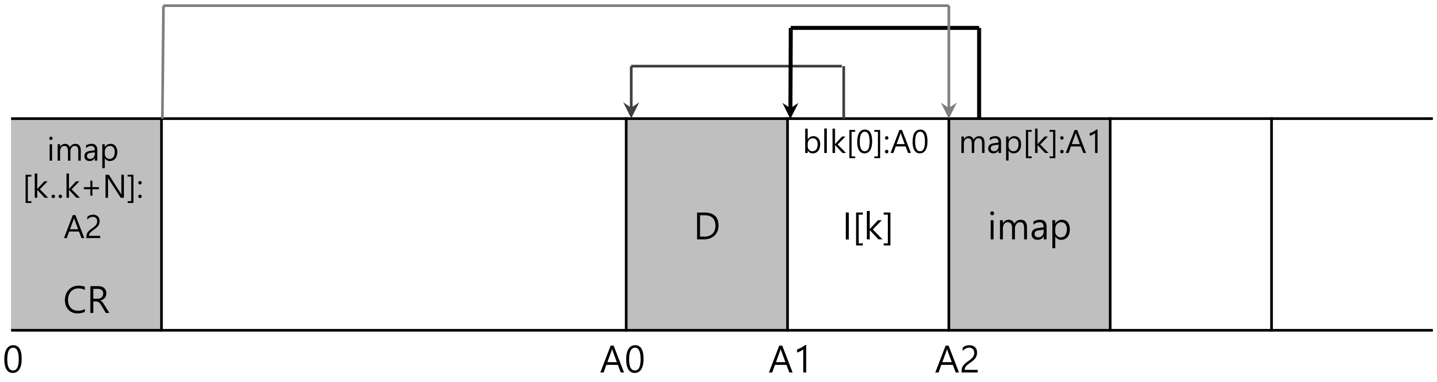


- **Imap** chunks are scattered also across the disk! (close to the inodes)

The Checkpoint Region

- How to find the inode map, spread across the disk?
  - The LFS File system have fixed location on disk to begin a file lookup
- **Checkpoint Region** contains pointers to the latest of the inode map
  - Only updated periodically (ex. Every 30 seconds)

    → performance is not ill-affected

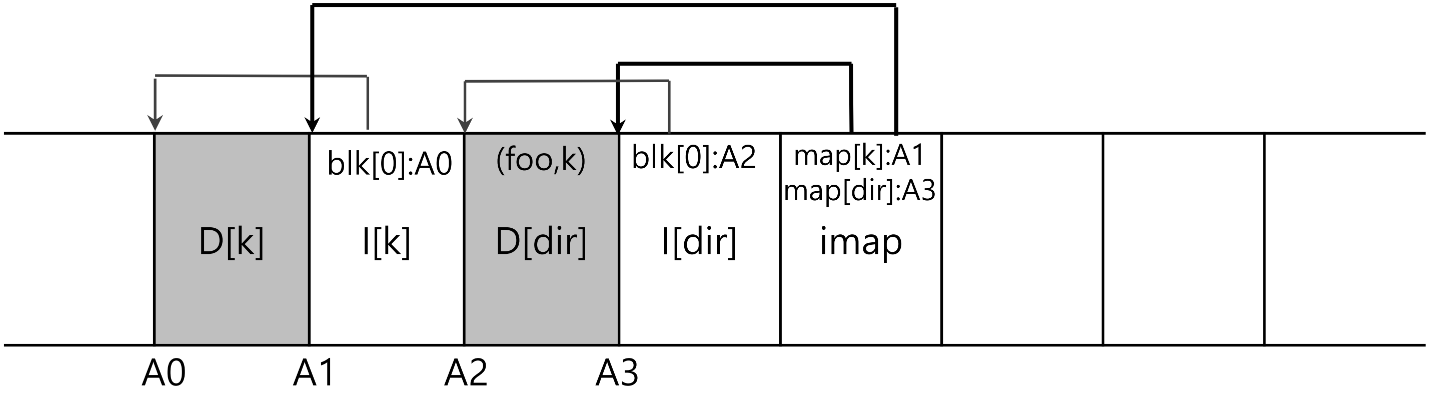


Reading a File from Disk: A Recap

1. Read checkpoint region
2. Read entire inode map and **cache it in memory**
3. Read the most recent inode
4. Read a block from file by using direct or indirect or double-indirect pointers

What About Directories?

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



  - Directory avoids **recursive update problem** using imap (not inodes)
    - When a file change is location, directory won't be updated because inode number of the file doesn’t change (just location)

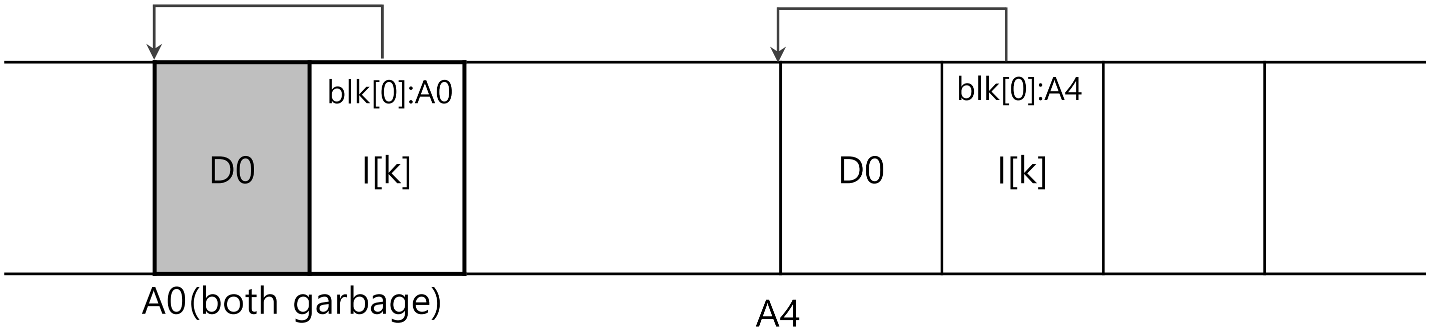
A harder problem: Garbage Collection

- LFS keeps writing newer version of file to new locations.

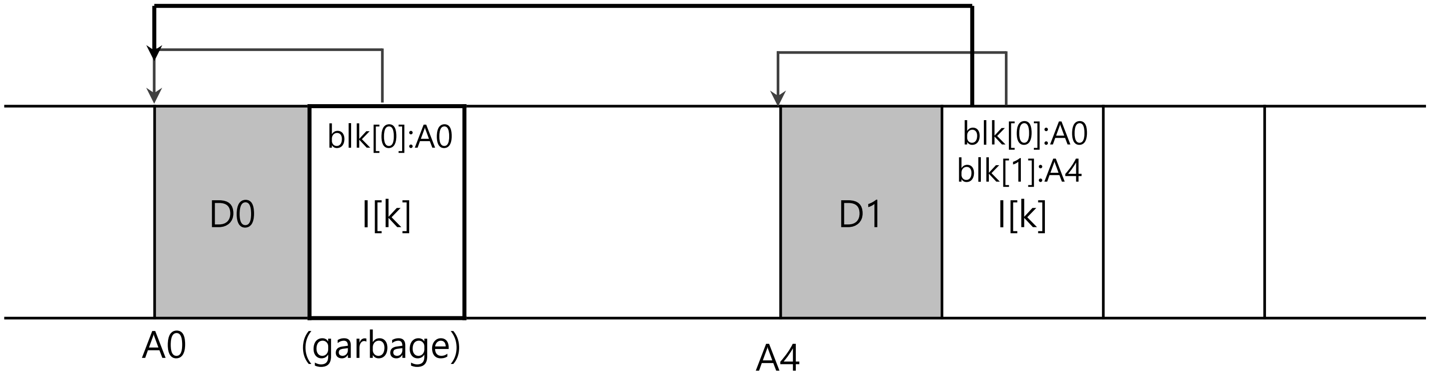
- Thus, LFS leaves the older versions of file structures all over the disk, call as garbage.

Examples: Garbage

- For a file with a singe 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

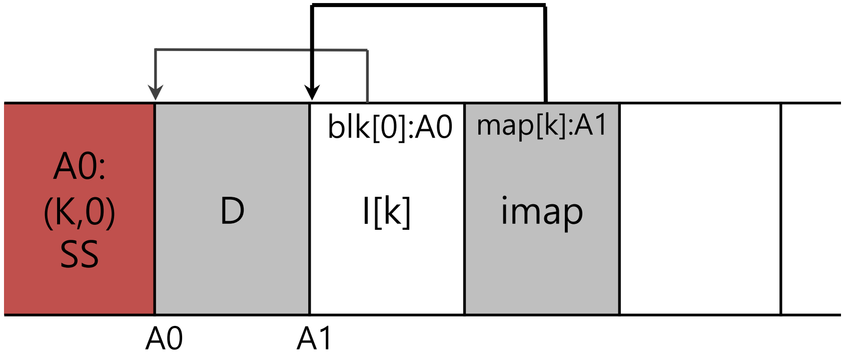
- One possibility: **Versioning file system**
  - keep the older versions around
  - Users can restore old file versions

- LFS approach: **Garbage Collection**
  - Keep only the latest live version and periodically clean old dead versions
  - Segment-by-segment basis
    - Block-by-block basis cleaner eventually make free holes in random location
      → Writes can not be sequential anymore

- Can be **performance critical**
  - In some benchmarks, performance can be terrible (v.gr if garbage collection interferes with the underlying workload)

Mechanism: 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



```
A = Address of D in disk
N = Inode of D
T = offset
(N, T) = SegmentSummary[A];
inode = Read(imap[N]);
if (inode[T] == A)
        // block D is alive
else
        // block D is garbage
```

- **Version number** can be used for efficient liveness determining (version number in inode and SS)

Policy: Which Blocks to Clean, and When?

- When to clean?
  - Periodically
  - During idle time
  - When the disk is full (hysteresis)

- Which blocks to clean?
  - Segregate hot/cold segments
    - Hot segment: frequently over-written

      → more blocks are getting over-written if we wait a long time before cleaning
    - Cold segment: relatively stable

      → May have a few dead blocks, but the other blocks are stable
  - Clean cold segment sooner and hot segment later

Crash Recovery and the Log

- **Log** organization in LFS
  - Checkpoint Region (CR) points to a head and tail segment (in the list of segments to be written)
  - Each segment points to next segment
- LFS can easily recover by simply reading latest valid CR
  - The latest consistent snapshot may be quite old (~30 secs)

- **Guarantee CR correctness**: ensure "atomicity" of CR update
  - Keep two CRs, guarded by **timestamps** (TS) each entry update (imap ptr.)
  - CR update protocol: TS_0→ CR(w) → TS^1
  - CR with consistent TS are recovered.

- **Roll forward** (DB technique) to recover data beyond correct CRs
  - Start from end of the log (pointed by the latest CR with good TS)
  - Read next segments and adopt any valid updates to the file system

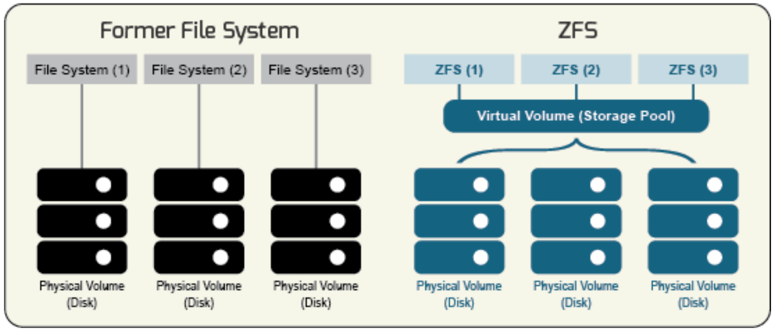
- Garbage collection interference: **wait for CR update**

LFS legacy

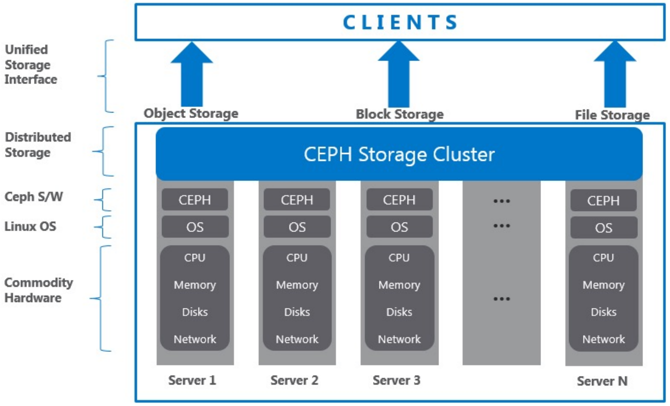
- Initially, garbage collection create some controversy and prevent the idea to success
- Copy-on-write is used by most state-of-the-art FS
  - BTRFS
  - ZFS
  - Some SSD FS: from Performance and Reliability standpoints

Aside: Technological Impact on Storage

- Processing capabilities
  - ZFS, btrfs



- Outscaling
  - CEPH



- This lecture slide set is used in AOS course at University of Cantabria by V.Puente. Was initially developed for Operating System course in Computer Science Dept. at Hanyang University. This lecture slide set is for OSTEP book written by Remzi and Andrea Arpaci-Dusseau (at University of Wisconsin)