Log-structured File Systems

CMPU 334 – Operating Systems
Jason Waterman

Motivation for a New Type of File System

- Memory sizes are growing
  - Disk reads are mostly serviced by larger disk caches
  - File system performance is largely determined by write performance
- Large gap between random I/O and sequential I/O performance
  - Transfer bandwidth has increased a great deal over the years
  - Seek and rotational delays have decreased slowly
- Existing file system perform poorly on common workloads
  - FFS performs a large number of writes to create a new file
  - Blocks in the same block group still have many short seeks and rotational delays

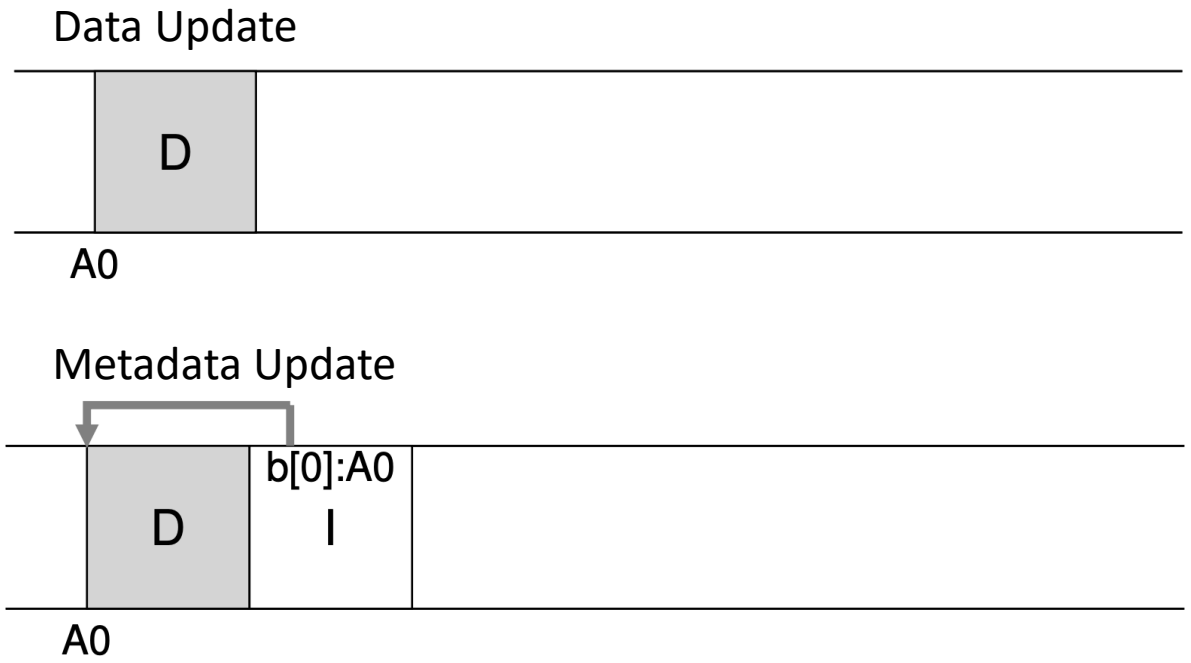
Log-structured File System

- Goals for the new file system
  - Have the file system focus on write performance
    - Try to make use of the sequential bandwidth of the disk
  - Perform well on common workloads
    - Writing out data as well as frequent metadata structures

- Log-structured file system basic idea:
  - When writing to disk, buffer all updates including metadata into a **segment**
  - When the segment is full, write to disk in one long sequential transfer
    - To an unused part of the disk, always writing to a free location
    - LFS never overwrites existing data
    - Because the segments are large, the disk is used efficiently

Writing to the Disk Sequentially

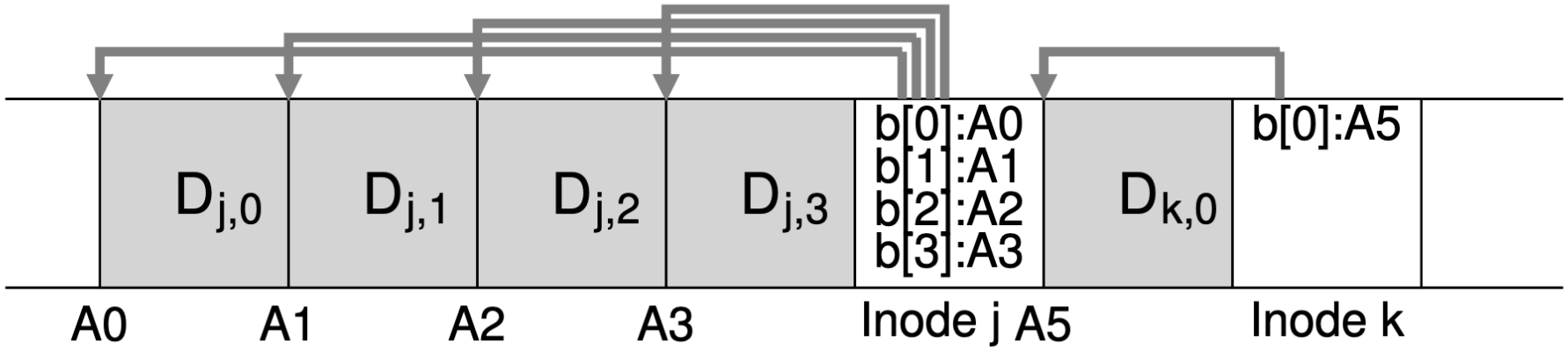
- How do we transform all updates to file-system state into a series of sequntial writes to disk?
  - Example: writing a data block D at disk address A0 to a file



- Write all updates (including metadata) sequentially is the key idea of LSF

Writing to Disk Sequentially and Effectively

- Writing single blocks sequentially does not guarantee efficient writes
  - After writing into A0, the next write to A1 might be delayed by disk rotation
  - Must issue one large write to achieve good write performance
- Write buffering for effectiveness
  - Keep track of updates in a **memory buffer** (also called **segment**)
  - Write them to disk all at once, when there are a large number of updates (a few MB)



How Much to Buffer?

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

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

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

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

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

How Much to Buffer?

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

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

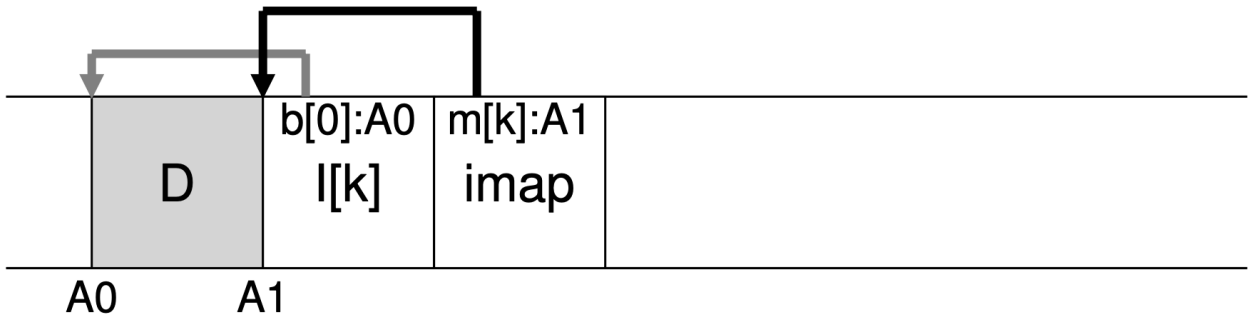
- Solve for D

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

- 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 at least 9MB

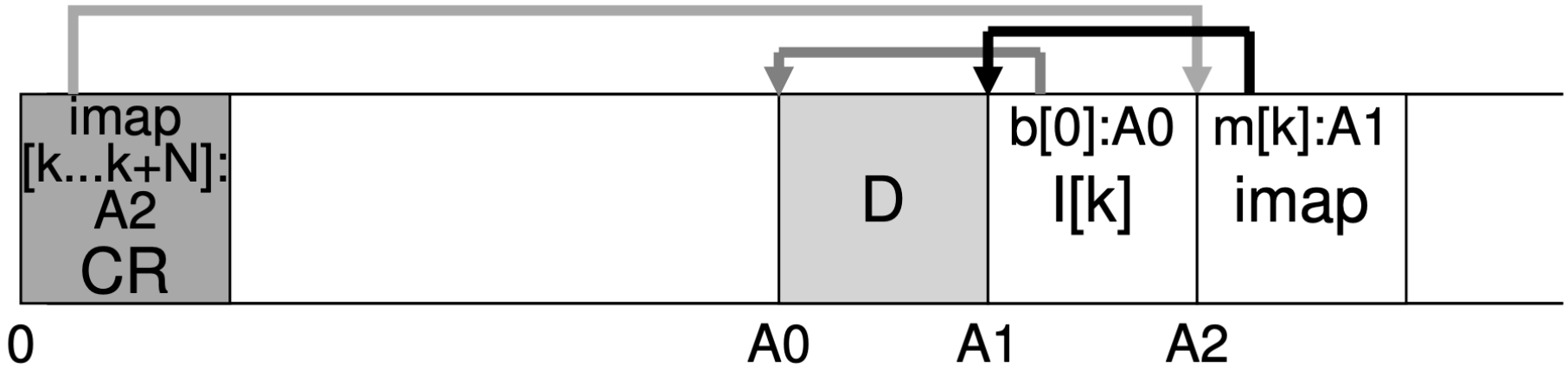
Finding Inodes in LFS

- Inodes are now scattered throughout the disk
- Add a layer of indirection: the **Inode Map** (imap)
  - Mapping of inode numbers to disk addresses
- Where should the imap go?
  - For performance, LFS places chunks of the inode map next to where it writes the other new information
  - So, the imap is also scattered throughout the disk!
    - How do you locate the correct imap?



The Checkpoint Region

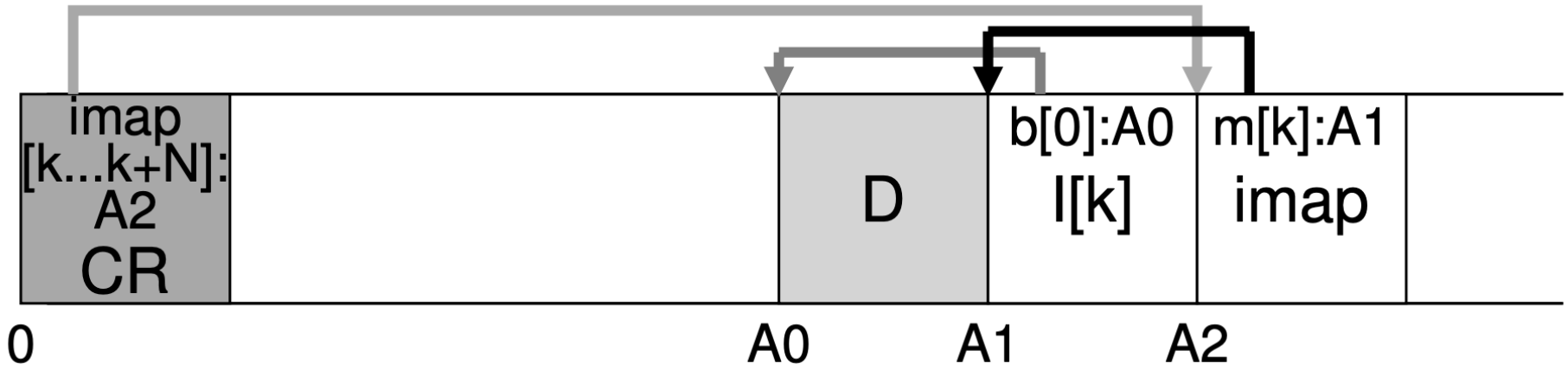
- How to find the inode map, which is spread across the disk?
  - LFS has a fixed location on disk to begin a file lookup
- **Checkpoint Region (CR)** contains pointers to the latest pieces of the inode map
  - Normally cached in memory
  - Only updated periodically (e.g., every 30 seconds)
    - Limited impact on performance



How to Read a File from Disk in LFS

Assume we have nothing in memory to begin with

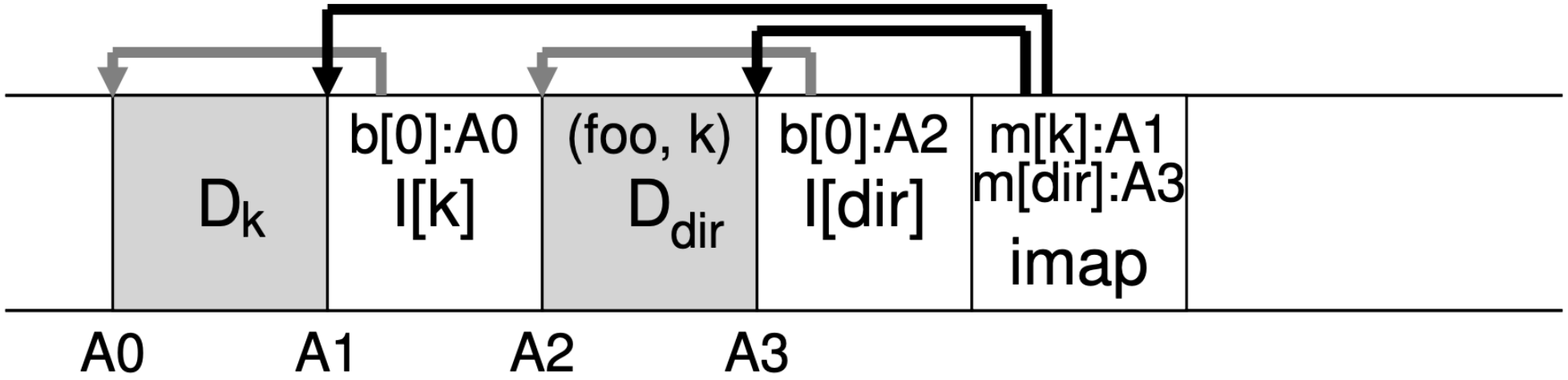
1. Read checkpoint region
2. Read entire inode map (imap) and **cache it in memory**
3. Read the most recent inode from the address given in inode map
4. Read a block from file by using direct or indirect or doubly-indirect pointers



What About Directories?

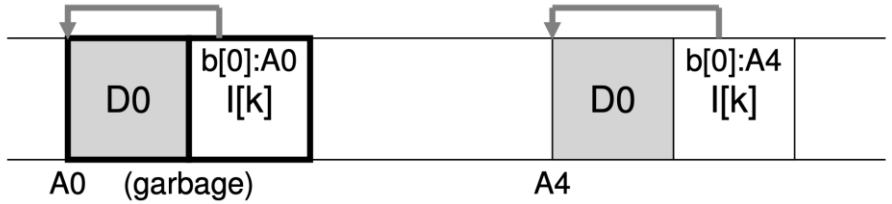
- Directory structure of LFS is like the classic UNIX file system
  - Directory is a file which data blocks consisting of directory information
  - Just another file as far as the file system is concerned

- Example: creating a file `foo` (inode k) in a directory (inode dir)

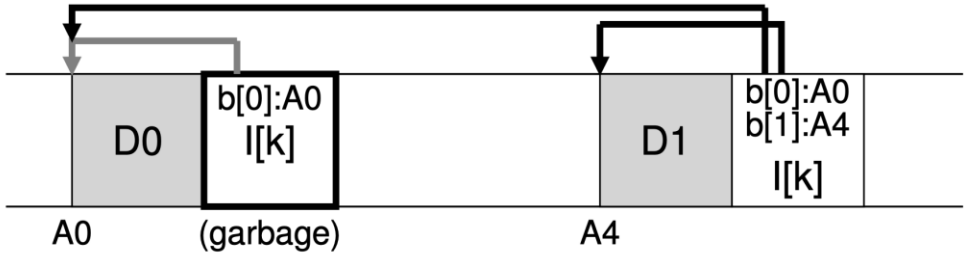


Data updates

- LFS writes newer versions of file to new locations
- Example: updating a file with a singe data block
  - Write the data block with new data, create updated inode, update inode map (not shown)
  - Both old data block and inode become garbage

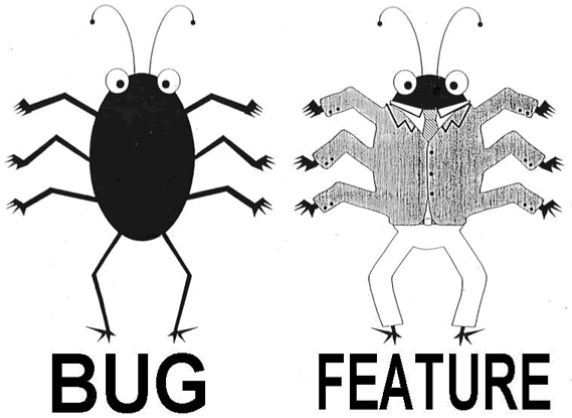


- Example: appending a block to a file (inode k)
  - Old inode becomes garbage



Handling older versions of inodes and data blocks

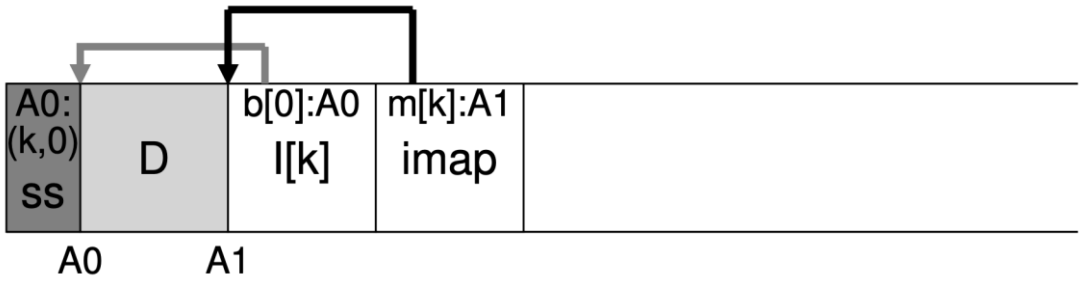
- One possibility: a **Versioning file system**
  - Turn a flaw into a feature
  - Keep the older file versions around
  - Users can restore old files as needed



- LFS approach: **Garbage Collection**
  - Keep only the latest live version and periodically clean old dead versions
  - Clean data on a segment-by-segment basis
    - A block-based cleaner would make free holes in random locations
      - We write data one entire segment at a time
      - Writes would not be sequential anymore
  - Looks at which blocks are live in a segment
    - Write those live blocks out to a new segment
    - After copying those live blocks, the segment is free to be reused

Determining Block Liveness (mechanism)

- **Segment Summary block (SS)**
  - Added information located at the start of each segment
  - Inode number and inode data block offset (which block in the file) for each data block are recorded here
- Determining Liveness
  - For each block in the segment:
    - Look up inode address in the imap
    - In the inode, get the data block address using the inode offset
    - See if the data block address matches the address in the SS
      - If so, the data is still live



Which Blocks to Clean, and When? (policy)

- When to clean?
  - Periodically
  - During idle time
  - When the disk is full
- Which blocks to clean?
  - LFS Segregates into 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

- LFS organizes writes in a **log**
  - Checkpoint Region (CR) points to a head and tail segment
  - Each segment points to next segment to be written
- Crashes can occur in writes to the CR and to the segment
- To ensuring atomicity of CR update
  - Keep two CRs, one at either end of the disk
  - Write to them alternately
  - CR update protocol: timestamp → CR → timestamp (same timestamp)
- Upon failure, LFS can easily recover by simply reading latest valid CR
  - However, the latest consistent snapshot may be quite old
- Try to rebuild missing segments since last CR update by Roll forward
  - Get any updates since the last snapshot of the CR
  - Start from end of the log (pointed by the latest CR)
  - Read next segments and adopt any valid updates to the file system

LFS Flame Wars of the 90s

- In two papers (one in 1993, one in 1995) Seltzer compared FFS to a BSD port of Sprite's LFS, finding that:
  - LFS is great for workloads with:
    - Frequent small writes
    - Read patterns that are amenable to hitting in the buffer cache
    - Enough idle time for cleaning to run without hurting foreground tasks
  - LFS is not great when:
    - The disk is full (because the cleaner must read many segments just to find a little free space)
    - Writes are too random (because dead space will be spread evenly throughout the segments, forcing the cleaner to read many segments to free space)
- Ousterhout (who wrote Sprite LFS) claimed that:
  - BSD LFS was poorly implemented and had performance bugs
  - The benchmarks used to evaluate BSD LFS were unfair (e.g., the compilation benchmark was CPU bound and doesn't provide much insight into file system behavior; the transaction processing workload contains a pathological number of random writes)
  - FFS fragmentation can hurt performance just as much as LFS cleaning

File Systems Summary

- The original Unix file system (and vsfs) was a simple design but slow
  - Metadata was not placed near the data, requiring many disk seeks
  - Only provided 2-4% of the sequential disk bandwidth
    - Got worse over time due to external fragmentation
    - Small block size (512 bytes) also hurt performance

- FFS improved performance by being disk aware
  - Related files and directories are stored in the same block/cylinder group
  - Block size increased from 512 bytes to 4 KB
  - Used ”skip sectors” to minimize rotational latency
  - Used `fsck` for failure recovery

File Systems Summary

- Journaling file systems (e.g., ext3) use write-ahead logging to make crash recovery faster
  - Write metadata in the journal before actually updating on-disk structures
  - Perform redo logging of physical blocks on recovery
  - Much faster than multiple passes of the entire disk (`fsck`)

- LFS turns the entire file system into a log
  - Assumes that a large buffer cache will handle most reads
  - Making writes fast is the most important thing
    - Turns all writes (random or sequential, large or small) into large sequential writes
    - No writes in-place, updates are written in a new segment
  - Fast recovery
  - Needs garbage collection, which may reduce the benefits of all sequential writes