Log-Structured File Systems

Basic Problem

- Most file systems now have large memory caches (buffers) to hold recently-accessed blocks
  - Most reads are thus satisfied from the buffer cache
- From the point of view of the disk, most traffic is write traffic
  - To speed up disk I/O, we need to make writes go faster
- But disk performance is limited ultimately by disk head movement
- With current file systems, adding a block takes several writes (to the file and to the metadata), requiring several disk seeks

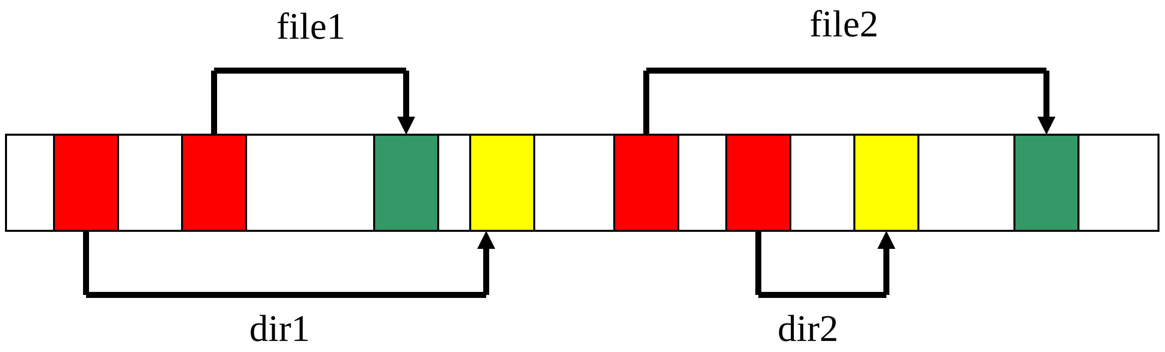
LFS: Basic Idea

- An alternative is to use the disk as a *log*
- A log is a data structure that is written only at the head
- If the disk were managed as a log, there would be effectively no head seeks
- The "file" is always added to sequentially
- New data and metadata (inodes, directories) are accumulated in the buffer cache, then written all at once in large blocks (e.g., segments of .5M or 1M)
- This would greatly increase disk thruput
- How does this really work? How do we read? What does the disk structure look like? etc.?

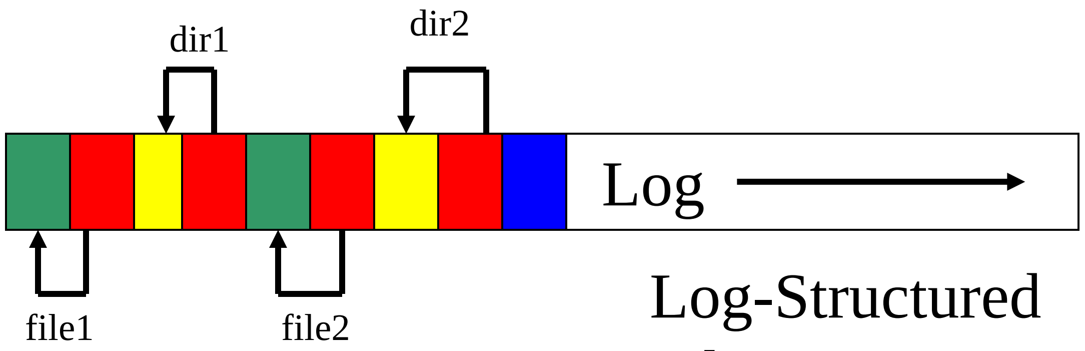
LFS Data Structures

- **Segments: log containing data blocks and metadata**
- **inodes: as in Unix, inodes contain physical block pointers for files**
- **inode map: a table indicating where each inode is on the disk**
  - inode map blocks are written as part of the segment; a table in a fixed checkpoint region on disk points to those blocks
- **segment summary: info on every block in a segment**
- **segment usage table: info on the amount of “live” data in a block**

LFS vs. UFS



Unix File System



Log-Structured File System

🟥 inode

🟨 directory

🟩 data

🟦 inode map

Blocks written to create two 1-block files: dir1/file1 and dir2/file2, in UFS and LFS

LFS: read and write

- Every write causes new blocks to be added to the current segment buffer in memory; when that segment is full, it is written to the disk
- Reads are no different than in Unix File System, once we find the inode for a file (in LFS, using the inode map, which is cached in memory)
- Over time, segments in the log become fragmented as we replace old blocks of files with new block
- Problem: in steady state, we need to have contiguous free space in which to write

LFS Failure Recovery

- Checkpoint and roll-forward
- Recovery is very fast
  - No fsck, no need to check the entire disk
  - Recover the last checkpoint, and see how much data written after the checkpoint you can recover
  - Some data written after a checkpoint may be lost
  - Seconds versus hours

Cleaning

- **The major problem for a LFS is *cleaning*, i.e., producing contiguous free space on disk**
- **A cleaner process "cleans" old segments, i.e., takes several non-full segments and compacts them, creating one full segment, plus free space**
- **The cleaner chooses segments on disk based on:**
  - utilization: how much is to be gained by cleaning them
  - age: how likely is the segment to change soon anyway
- **Cleaner cleans "cold" segments at 75% utilization and "hot" segments at 15% utilization (because it's worth waiting on "hot" segments for blocks to be rewritten by current activity)**

LFS Summary

- **Basic idea is to handle reads through caching and writes by appending large segments to a log**
- **Greatly increases disk performance on writes, file creates, deletes, ….**
- **Reads that are not handled by buffer cache are same performance as normal file system**
- **Requires cleaning demon to produce clean space, which takes additional CPU time**