LFS: Log-Structured Filesystem

Motivation:

- RAM is cheap, so most OSes have a large disk cache in RAM
- This large cache absorbs most disk reads
- But disk writes must go through to disk eventually
- Consequence: the disk will see mostly writes

LFS exploits the cache-driven shift in read/write ratio by eliminating seeks on many writes. It writes data in big, consecutive chunks.

LFS — Uses of logs

In general, why are logs used? Recall from Chapter 8:

- **Stability**: as backup copy for primary storage
- **Archiving**: to maintain complete record of every operation
- **Recovery**: to go back to a consistent state on crashes

LFS adds:

- **Performance**: to make write operations sequential, and hence faster

LFS — Overview

Assume (for now) that disk size is infinite. To create a file:

- Write to the end of the log, like a tape
- Write the **inode map** to the log
- One big sequential write $\Rightarrow$ just one seek.

In contrast, to update a file in Unix FFS:

- Create file data
- Update inode table and parent directory
- Lots of random-access updates $\Rightarrow$ lots of seeks!

LFS — Free Space Management

Disk size is not infinite! We need to periodically clean up blocks that have been deleted, or superseded by new data.

- Divide disk into “segments”
- Periodically perform **segment cleaning** (stop-and-copy garbage collection):
  - Read a few segments into memory
  - Identify live data (data which have not been deleted or superseded)
  - Write live data back to disk in a new place, in fewer segments
- To speed cleaning, maintain a **segment summary block** in each segment, so we can quickly identify the files the segment contains, and find segment utilization and age

LFS — Cleaning Policies

- **When to execute?** When only a few tens of segments are free
- **How long to execute?** Until 50–100 segments are free
- **Which segments to clean?** Benefit/cost ratio:

$$\frac{\text{free space generated} \times \text{age of data}}{\text{cost}} = \frac{(1-u) \times \text{age}}{1+u}$$

- **How to group blocks?** Sort by age — hope that blocks written near the same time will be read near the same time

LFS — Recovery

- Periodically write a **checkpoint** which contains:
  - Pointers to blocks in the inode map and segment usage table
  - Point to the last segment written (end of the log)
  - Checkpoint time (last)

  Two checkpoint regions on disk. Switch between them to make checkpointing atomic.
- To recover, read checkpoint and **roll forward**, i.e., replay any changes that occurred after the checkpoint. Update inode maps accordingly
- How to prevent directories from having inconsistent state? Use **directory operation log** to replay directory operations

Replication and Consistency

We often have to keep multiple copies of data at different sites within a system. For example:

- L1 and L2 processor caches: replicating RAM data onchip
- Disk caches: replicating disk data in memory
- Network caches: replicating networked data on the local machine
- RAID: replicating disk data on other disks

Why?

- **Performance**. Cheap, big memory and networks are usually slow, so we use caches
- **Reliability**. Storage devices can fail
- **Connectivity**. You can't always be connected to a network

Replication: Challenges

Maintaining consistent replicas is tough! Why?

- Data change over time
- Hard to keep of track who has the "correct" copy of data
- Hard to manage synchronizing between replicas

. . .especially if availability is intermittent (as in Coda) or updates are transactional (as in databases)

Ways to Provide Replication

- **Replicated state machines**: every server receives the same inputs and handles the inputs in the same way. This is tough because:
  - Inputs must be exactly the same, and in exactly the same order
  - Data replicas can drift apart
  - State machines need to be absolutely identical
- **Single state machine**: a **master** server periodically sends a copy of the entire data set to every **slave**
  sends changed data)
  - Often: partition the database into small regions (e.g., files, tables, or rows) which can be updated independently
  - Assign different masters for different partitions

Other Replication Techniques

- **Voting**: ask multiple servers, and the most common answer wins
- **Backup**: keep an extra copy to be swapped in if the original copy becomes unavailable
- **Incremental backup**: only back up *changes*, not the whole data set.

Coda

Motivation:

Typical network filesystems like NFS and AFS (we use the latter at MIT) don't handle the case where a remote server has failed or is unavailable.

Coda uses caching (replication) to improve availability, and, in particular, to support **disconnected operation**.

Coda — Design Overview

- Use **callbacks** for cache coherence
- Hoard critical data to improve disconnected operation (cf. typical caching, based entirely on previous access pattern)
- On reconnection, synchronize state with server, notifying user if there is a consistency problem

Key principles:

- Keep functionality on clients, not servers. Similar in spirit to NFS's stateless approach, although we need *some* state on the server to handle callbacks
- Optimistic replica control

Coda — Optimistic vs. Pessimistic Replica Control

- **Pessimistic**: Before changing a file, make sure every other client knows that you're writing it
  - Advantage: prevents conflicts
- **Optimistic**: Let anyone write anytime, and try to resolve conflicts when they occur
  - Advantage: can write even when disconnected

Coda uses optimistic replica control, since allowing disconnected operation is an important goal.

Coda — Hoarding

Which files should be hoarded on the local machine?

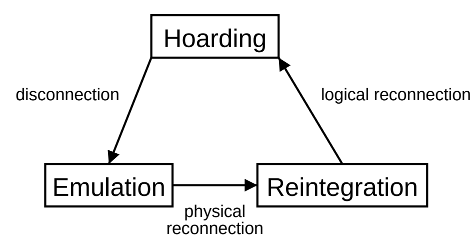
- Any file currently or recently in use (dynamic priorities)
- The **hoard profile** contains a list of files to hoard, and their priorities
- When connected, do a **hoard walk** every 10 minutes to re-establish **equlibrium** by replacing low-priority files with higher-priority ones from the server

Coda — Caching and Callbacks

Coda uses **callbacks** to maintain cache coherence.

- Client loads whole file into local cache on `open` (if it's not there already)
- Client keeps server informed as to which files are in its cache
- Server contract: "I will tell you immediately when your copy of a file is no longer valid, i.e., on **callback break**"

Coda — States



- **Hoarding**: The usual "connected" state. Maintains the cache, trying to keep files in the **hoard profile** cached
- **Emulating**: The usual "disconnected" state. Serves files from the cache, does security checks locally, and logs changes for reintegration
- **Reintegration**: Synchronizes with the server, bailing out if there are any conflicts.

Coda — Reintegration

What does the client do on reconnection?

- Lock all changed files on server; replay changes; unlock changed files
- If someone else has modified the file (the **storeid** has changed on the server since when the client originally read it), abort the *entire* reintegration and write out a `tar` file containing the whole log
- Handle directories specially — don't abort when a directory-file conflict occurs, just merge changes. We can do this since we know directory-file semantics