Lecture Notes, 9/4

Log-Structured FS
HP AutoRAID

I. Log-Structured File System

Radically different file system design.

Technology motivations:

- CPUs outpacing disks: I/O becoming more-and-more of a bottleneck.
- Big memories: file caches work well, making most disk traffic writes.

Problems with current file systems:

- Lots of little writes.
- Synchronous: wait for disk in too many places. (This makes it hard to win much from RAIDs, too little concurrency.)

Basic idea of LFS:

- Log all data and meta-data with efficient, large, sequential writes.
- Treat the log as the truth (but keep an index on its contents).
- Rely on a large memory to provide fast access through caching.

Two potential problems:

- Log retrieval on cache misses.
- Wrap-around: what happens when end of disk is reached?
  - No longer any big, empty runs available.
  - How to prevent fragmentation?

Log retrieval:

- Keep same basic file structure as UNIX (inode, indirect blocks, data).
- Retrieval is just a question of finding a file's inode.
- UNIX inodes kept in one or a few big arrays, LFS inodes must float to avoid update-in-place.

- Solution: an *inode map* that tells where each inode is. (Also keeps other stuff: version number, last access time, free/allocated.)
- Inode map gets written to log like everything else.
- Map of inode map gets written in special checkpoint location on disk; used in crash recovery.

Disk wrap-around:

- Compact live information to open up large runs of free space. Problem: long-lived information gets copied over-and-over.
- Thread log through free spaces. Problem: disk will get fragmented, so that I/O becomes inefficient again.
- Solution: *segmented log*.
  - Divide disk into large, fixed-size segments.
  - Do compaction within a segment; thread between segments.
  - When writing, use only clean segments (i.e. no live data).
  - Occasionally *clean* segments: read in several, write out live data in compacted form, leaving some fragments free.
  - Try to collect long-lived information into segments that never need to be cleaned.

Which segments to clean?

- Keep estimate of free space in each segment to help find segments with lowest utilization.
- If utilization of segments being cleaned is U:
  - write cost = (total bytes read & written)/(new data written) = 2/(1-U). (unless U is 0).
  - write cost increases as U increases: U = .9 => cost = 20!
  - need a cost of less than 4 to 10; => U of less than .75 to .45.

Simulation of LFS cleaning:

- Initial model: uniform random distribution of references; greedy algorithm for segment-to-clean selection.
- Why does the simulation do better than the formula? Because of variance in segment utilizations.
- Added locality (i.e. 90% of references go to 10% of data) and things got worse!
- First solution: write out cleaned data ordered by age to obtain hot and cold segments.
  - What prog. language feature does this remind you of? Generational GC.
  - Only helped a little.
- Problem: even cold segments eventually have to reach the cleaning point, but they drift

down slowly. tying up lots of free space. *Do you believe that's true?*

- Solution: it's worth paying more to clean cold segments because you get to keep the free space longer.
- New selection function: MAX(T*(1-U)/(1+U)).
  - Resulted in the desired bi-modal utilization function.
  - LFS stays below write cost of 4 up to a disk utilization of 80%.

Crash recovery:

- Unix must read entire disk to reconstruct meta data.
- LFS reads checkpoint and rolls forward through log from checkpoint state.
- Result: recovery time measured in seconds instead of minutes to hours.

An interesting point: LFS' efficiency isn't derived from knowing the details of disk geometry; implies it can survive changing disk technologies (such variable number of sectors/track) better.

Key features of paper:

- CPUs outpacing disk speeds; implies that I/O is becoming more-and-more of a bottleneck.
- Write FS information to a log and treat the log as the truth; rely on in-memory caching to obtain speed.
- Hard problem: finding/creating long runs of disk space to (sequentially) write log records to. Solution: clean live data from segments, picking segments to clean based on a cost/benefit function.

Some flaws:

- Assumes that files get written in their entirety; else would get intra-file fragmentation in LFS.
- If small files "get bigger" then how would LFS compare to UNIX?

A Lesson: Rethink your basic assumptions about what's primary and what's secondary in a design. In this case, they made the log become the truth instead of just a recovery aid.

II. HP Auto RAID

Goals: automate the efficient replication of data in a RAID

- RAIDs are hard to setup and optimize
- Mix fast mirroring (2 copies) with slower, more space-efficient parity disks
- Automate the migration between these two levels

3 levels of RAID:

Lecture 3

- Mirroring (simple, fast, but requires 2x storage)
- Parity disk (RAID level 3)
- Rotating parity disk (RAID level 5)

Each kind of replication has a narrow range of workloads for which it is best...

- Mistake ⇒ 1) poor performance, 2) changing layout is expensive and error prone
- Also difficult to add storage: new disk ⇒ change layout and rearrange data...

(another problem: spare disks are wasted)

Key idea: mirror active data (hot), RAID 5 for cold data

- Assumes only part of data in active use at one time
- Working set changes slowly (to allow migration)

Where to deploy:

- sys-admin: make a human move around the files.... BAD. painful and error prone
- File system: best choice, but hard to implement/deploy; can't work with existing systems
- Smart array controller: (magic disk) block-level device interface. Easy to deploy because there is a well-defined abstraction

Features:

- Block Map: level of indirection so that blocks can be moved around among the disks
- Mirroring of active blocks
- RAID 5 for inactive blocks or large sequential writes (why?)
- Start out fully mirrored, then move to 10% mirrored as disks fill
- Promote/demote in 64K chunks (8-16 blocks)
- Hot swap disks, etc. (A hot swap is just a controlled failure.)
- Add storage easily (goes into the mirror pool)
- No need for an active hot spare (per se); just keep enough working space around
- Log-structured RAID 5 writes. (Why is this the right thing? Nice big streams, no need to read old parity for partial writes)

Issues:

- When to demote? When there is too much mirrored storage (>10%)
- Demotion leaves a hole (64KB). What happens to it? Moved to free list and reused
- Demoted RBs are written to the RAID5 log, one write for data, a second for parity
- Why log RAID5 better than update in place? Update of data requires reading all the old data to recalculate parity. Log throws ignores old data (which becomes garbage) and

writes only new data/parity stripes.

- When to promote? When a RAID5 block is written... Just write it to mirrored and the old version becomes garbage.
- How big should an RB be? Bigger ⇒ finer-grain migration, smaller ⇒ less mapping information, bigger ⇒ fewer seeks
- How do you find where an RB is? Convert addresses to (LUN, offset) and then lookup RB in a table from this pair. Map size = Number of RBs and must be proportional to size of total storage.
- Controller uses cache for reads
- Controller uses NVRAM for fast commit, then moves data to disks. What is NVRAM is full? Block until NVRAM blocks flushed to disk, then write to NVRAM.
- Disks writes normally go to two disks (since newly written data is "hot"). Must wait for both to complete (why?). Does the host have to wait for both? No, just for NVRAM.
- What happens in the background? 1) compaction, 2) migration, 3) balancing.
- Compaction: clean the RAID5 and plug holes in the mirrored disks. Do mirrored disks get cleaned? Yes, when a PEG is needed for RAID5; i.e., pick a disks with lots of holes and move its used RBs to other disks. Resulting empty PEG is now usable by RAID5.
- What if there aren't enough holes? Write the excess RBs to RAID5, then reclaim the PEG.
- Migration: which RBs to demote? Least-recently-written
- Balancing: make sure data evenly spread across the disks. (Most important when you add a new disk)

Bad cases? One is thrashing when the working set is bigger than the mirrored storage

Performance:

- consistently better than regular RAID, comparable to plain disks (but worse)
- They couldn't get RAIDs to work well...

Other things:

- "shortest seek" -- pick the disk (of 2) whose head is closet to the block
- When idle, plugs holes in RAID5 rather than append to log (easier because all RBs are the same size!) Why not all the time? Requires reading the rest of the stripe and recalculating parity
- Very important that the behavior is dynamic: makes it robust across workloads, across technology changes, and across the addition of new disks. Greatly simplifies management of the disk system

Key features of paper:

- RAIDs difficult to use well
- Mix mirroring and RAID5 automatically
- Hide magic behind simple SCSI interface