Lecture Notes

UNIX Fast File System
Log-Structured File System
Analysis and Evolution of Journaling File Systems

I. Background

i-node: structure for per-file metadata (unique per file)

- contains: ownership, permissions, timestamps, about 10 data-block pointers
- They form an array, indexed by "i-number". So each i-node has a unique i-number.
- Array is explicit for FFS, implicit for LFS (its i-node map is cache of i-nodes indexed by i-number)

indirect blocks:

- i-node only holds a small number of data block pointers
- for larger files, i-node points to an indirect block (holds 1024 entries for 4-byte entries in a 4K block), which in turn points to the data blocks.
- Can have multiple levels of indirect blocks for even larger files

II. A Fast File System for UNIX

Original UNIX file system was simple and elegant, but slow.

Could only achieve about 20 KB/sec/arm; ~2% of 1982 disk bandwidth

Problems:

- blocks too small
- consecutive blocks of files not close together (random placement for mature file system)
- i-nodes far from data (all i-nodes at the beginning of the disk, all data after that)
- i-nodes of directory not close together
- no read-ahead

Aspects of new file system:

- 4096 or 8192 byte block size (why not larger?)
- large blocks and small fragments
- disk divided into cylinder groups
- each contains superblock, i-nodes, bitmap of free blocks, usage summary info

FFS/LFS

- Note that i-nodes are now spread across the disk: keeps i-node near file, i-nodes of a directory together
- cylinder groups ~ 16 cylinders, or 7.5 MB
- cylinder headers spread around so not all on one platter

Two techniques for locality:

- don't let disk fill up in any one area
- paradox: to achieve locality, must spread unrelated things far apart
- note: new file system got 175KB/sec because free list contained sequential blocks (it did generate locality), but an old system has randomly ordered blocks and only got 30 KB/sec

Specific application of these techniques:

- goal: keep directory within a cylinder group, spread out different directories
- goal: allocate runs of blocks within a cylinder group, every once in a while switch to a new cylinder group (jump at 1MB).
- layout policy: global and local
- global policy allocates files & directories to cylinder groups. Picks "optimal" next block for block allocation.
- local allocation routines handle specific block requests. Select from a sequence of alternative if need to.

Results:

- 20-40% of disk bandwidth for large reads/writes.
- 10-20x original UNIX speeds.
- Size: 3800 lines of code vs. 2700 in old system.
- 10% of total disk space unusable (except at 50% perf. price)

Could have done more; later versions do.

Enhancements made to system interface: (really a second mini-paper)

- long file names (14 -> 255)
- advisory file locks (shared or exclusive); process id of holder stored with lock => can reclaim the lock if process is no longer around
- symbolic links (contrast to hard links)
- atomic rename capability (the only atomic read-modify-write operation, before this there was none)

FFS/LFS

- disk quotas
- Could probably have gotten copy-on-write to work to avoid copying data from user->kernel. (would need to copies only for parts that are not page aligned)
- Overallocation would save time; return unused allocation later. Advantages: 1) less overhead for allocation, 2) more likely to get sequential blocks

3 key features of paper:

- parameterize FS implementation for the hardware it's running on.
- measurement-driven design decisions
- locality "wins"

A major flaws:

- measurements derived from a single installation.
- ignored technology trends

A lesson for the future: don't ignore underlying hardware characteristics.

Contrasting research approaches: improve what you've got vs. design something new.

III. 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.)
- 5 seeks to create a new file: (rough order) file i-node (create), file data, directory entry, file i-node (finalize), directory i-node (modification time).

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.

FFS/LFS

- Data layout on disk has “temporal locality” (good for writing), rather than “logical locality” (good for reading). Why is this a better? Because caching helps reads but not writes!

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.
  - Note there is not free list or bit map (as in FFS), only a list of clean segments.

Which segments to clean?

- Keep estimate of free space in each segment to help find segments with lowest

utilization.

- Always start by looking for segment with utilization=0, since those are trivial to clean...
- 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.

How to clean a segment?

- Segment summary block contains map of the segment. Must list every i-node and file block. For file blocks you need {i-number, block #}
- To clean an i-node: just check to see if it is the current version (from i-node map). If not, skip it; if so, write to head of log and update i-node map.
- To clean a file block, must figure out it if is still live. First check the UID, which only tells you if this file is current (UID only changes when is deleted or has length zero). Note that UID does not change every time the file is modified (since you would have to update the UIDs of all of its blocks). Next you have to walk through the i-node and any indirect blocks to get to the data block pointer for this block number. If it points to this block, then move the block to the head of the log.

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.
- Better way to think about this: don't clean segments that have a high d-free/dt (first derivative of utilization). If you ignore them, they clean themselves! LFS uses age as an approximation of d-free/dt, because the latter is hard to track directly.
- 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%.

FFS/LFS

Checkpoints:

- Just an optimization to roll forward. Reduces recovery time.
- Checkpoint contains: pointers to i-node map and segment usage table, current segment, timestamp, checksum (?)
- Before writing a checkpoint make sure to flush i-node map and segment usage table.
- Uses “version vector” approach: write checkpoints to alternating locations with timestamps and checksums. On recovery, use the latest (valid) one.

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.
- Directory operation log == log *intent* to acheive atomicity, then redo during recovery, (undo for new files with no data, since you can’t redo it)

Directory operation log:

- Example of “intent + action”: write the intent as a “directory operation log”, then write the actual operations (create, link, unlink, rename)
- This makes them atomic
- On recovery, if you see the operation log entry, then you can REDO the operation to complete it. (For new file create with no data, you UNDO it instead.)
- => “logical” REDO logging

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?

FFS/LFS

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.

IV. Analysis and Evolution of Journaling File Systems

Journaling file systems:

- Write-ahead logging: commit data by writing it to log, synchronously and sequentially
- Unlike LFS, then later moved data to its normal (FFS-like) location; this write is called *checkpointing* and like segment cleaning, it makes room in the (circular) journal
- Better for random writes, slightly worse for big sequential writes
- All reads go the the fixed location blocks, not the journal, which is only read for crash recovery and checkpointing
- Much better than FFS (fsck) for crash recovery (covered below) becuase it is much faster
- Ext3 filesystem is the main one in Linux; RieserFS was becoming popular

Three modes:

- **writeback** mode: journal only metadata, write back data and metadata independently
  metadata may thus have dangling references after a crash (if metadata written before the data with a crash in between)
- **ordered** mode: journal only metadata, but always write data blocks before their referring metadata is journaled. This mode generally makes the most sense and is used by Windows NTFS and IBM's JFS.
- **data journaling** mode: write both data and metadata to the journal
  Huge increase in journal traffic; plus have to write most blocks twice, once to the journal and once for checkpointing (why not all?)

Crash recovery:

- Load superblock to find the tail/head of the log
- Scan log to detect whole commited transactions (they have a commit record)
- Replay log entries to bring in-memory data structures up to date
  This is called "redo logging" and entries must be "idempotent"
- Playback is oldest to newest; tail of the log is the place where checkpointing stopped
- How to find the head of the log?

Some fine points:

- Can group transactions together: fewer syncs and fewer writes, since hot metadata may changes several times within one transaction
- Need to write a commit record, so that you can tell that all of the compound transaction made it to disk
- ext3 logs whole metadata blocks (physical logging); JFS and NTFS log logical records instead, which means less journal traffic
- head of line blocking: compound transactions can link together concurrent streams (e.g.

from different apps) and hinder asynchronous apps performance (Figure 6). This is like having no left turn lane and waiting on the car in front of you to turn left, when you just want to go straight.

- Distinguish between ordering of writes and durability/persistence: careful ordering means that after a crash the file system can be recovered to a consistent ***past*** state. But that state could be far in the past in the case of JFS. 30 seconds behind is more typical for ext3. If you really want something to be durable you must flush the log synchronously.

Semantic Block-level Analysis (SBA):

- Nice idea: interpose special disk driver between the file system and the real disk driver
- Pros: simple, captures ALL disk traffic, can use with a black-box filesystem (no source code needed and can even use via VMWare for another OS), can be more insightful than just a performance benchmark
- Cons: must have some understanding of the disk layout, which differs for each filesystem, requires a great deal of inference; really only useful for writes
- To use well, drive filesystem with smart applications that test certain features of the filesystem (to make the inference easier)

Semantic trace playback (STP):

- Uses two kinds of interpositionL 1) SBA driver that produces a trace, and 2) user-level library that fits between the app and the real filesystem
- User-level library traces dirty blocks and app calls to fsync
- Playback: given the two traces, STP generates a timed set of commands to the raw disk device. This sequence can be timed to understand performance implications.
- Claim: faster to modify the trace than to modify the filesystem and simpler and less error-prone than building a simulator
- Limited to simple FS changes
- Best example usage: showing that dynamically switching between ordered mode and data journaling mode actually gets the best overall performance. (Use data journaling for random writes.)