Lecture 11: FFS and LFS

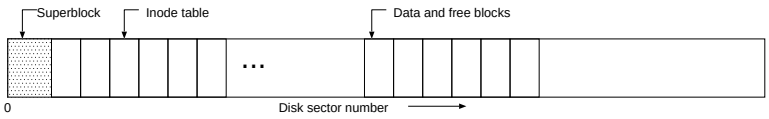
Mothy Roscoe and Joe Hellerstein

November 3, 2005

Review: the old UNIX file system

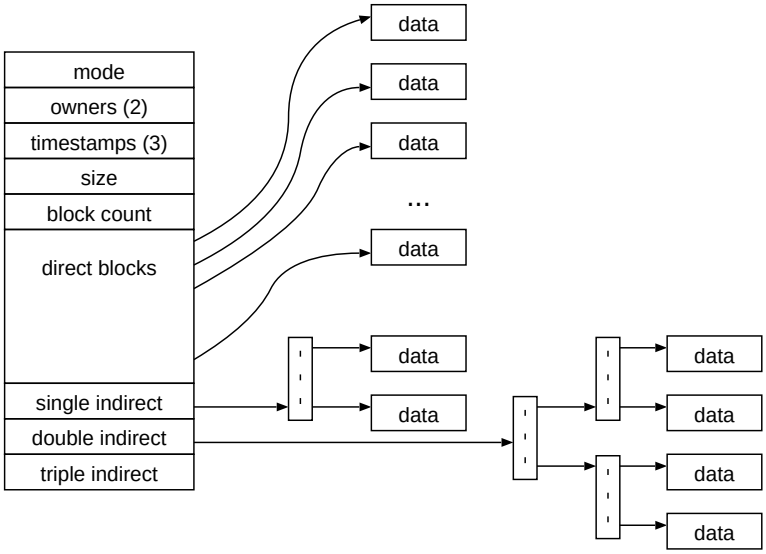
I-node: structure for per-file metadata (unique per file)

- ownership, permissions, timestamps, about 10 data-block pointers
- They form an array, indexed by "ci-number" $\Rightarrow$ each i-node (hence file) has a unique i-number.
- Array stays 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



A Fast File System for Unix

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

Could only achieve about 20 KB/sec/arm; about 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
- Note that i-nodes are now spread across the disk: keeps i-node near file, i-nodes of a directory together
- cylinder groups about 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 (almost a "worst fit" policy).
- note: a newly-formatted file system got 175KB/sec because the 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: within a file, 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' block for block allocation.
- local allocation routines handle specific block requests. Select from a sequence of alternatives 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.

FFS API enhancements

Really a second minipaper - not the main theme of the paper.

- Long file names (now 255 characters instead of 14)
- Advisory file locks (shared or exclusive); process id of holder stored with lock $\Rightarrow$ can reclaim the lock if process is no longer around (why advisory and not mandatory?)
- Symbolic links (contrast to hard links)
- Atomic rename capability (the only atomic read-modify-write operation, before this there was none)
- 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)

FFS Summary

Key features:

- Parameterize the filing system implemention so that it can adapt to the hardware (disk drives, controllers, CPU) it's running on.
- Expose the current set of system tradeoffs to applications if they are interested, but don't force them to adapt.
- Measurement drives the design - this is a heavily performance-oriented system and paper
- Locality is a Good Thing.

Flaws:

- All the measurements were derived from a single installation - unclear that this is hugely representative.

- FFS ignored technology trends, in particular the relative speeds / capacities / costs of RAM and disk (see below!)

A Log-Structured File System

Radically different file system design: compare the incremental approach of FFS (improve what you've got) to design something new.

Technology motivations: the paper contains a close look at industry trends at the time:

- 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.
- Often synchronous: the OS waits for the 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:

- Write any and all data and meta-data into a log with efficient, large, sequential writes.
- Treat the log as the truth (but keep an index on its contents). This is key: in LFS, the log is all there is.
- Rely on a large memory to provide fast read access through caching.
- 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!

Contrast with Journalled Filing Systems (and databases!):

- JFS / RDBMS use the log for *consistency* and (for filing systems) to improve recovery time by removing the need to `fsck`, which looks at all content on the disk.
- LFS uses a log in order to improve *write performance* - recovery actually looks a little slower than a journalled file system.

Two potential problems:

- Log retrieval on cache misses for reads.
- How to get large extents to write the log into, which amounts to preventing fragmentation as the disk fills up.

Log retrieval

- Keep same basic file structure as UNIX (inode, indirect blocks, data), but lay it out very differently.
- Retrieval is just a question of finding a file's inode.
- UNIX inodes kept in one (old FS) or a few big (FFS) arrays, but LFS inodes must float to avoid update-in-place - new inodes are written to the end of the log like everything else.
- Solution: an inode map that tells us where each inode is (as well as 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 and preventing fragmentation

Technical meat of most of the paper!

- 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 (sound familiar?)
  - 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 isn't a free list or bit map (as in FFS), only a list of clean segments - major divide-and-conquer simplicity.

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 $U = 0$, since those are trivial to clean ...
  - $writecost = (totalbytesread\&written)/(newdatawritten) = 2/(1-U)$. (unless U is 0).
  - write cost increases as U increases: $U = .9 \Rightarrow cost = 20$!
  - need a cost of 4 to 10 or less; $\Rightarrow U 0.75\ downto\ 0.45$

How to clean a segment?

- Segment summary block contains map of the segment, listing 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%.

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 achieve 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.)
- $\Rightarrow$ "logical" REDO logging.

An interesting point: LFS's 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 were outpacing disk speeds; implies that I/O is becoming more and more of a bottleneck.
- Main memory is getting bigger and bigger; implies caching can fix read I/O bandwidth (leaving writes as the problem)

- 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?
- Sparked some controversy over performance comparisons with clustered FFS . . .

A Lesson: Be prepared to 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.