ADVANCED OPERATING SYSTEMS

FFS

LFS

A Fast File System (1984)

- What kind of paper is this?
  - New big idea?
  - Measurement paper?
  - Experiences/lessons learnt paper?
  - A system description?
  - Performance study?
  - Refute-conventional wisdom?
  - Survey paper?

A Fast File System (1984)

- Original Unix file system – simple, elegant, slow.
  - Only achieve 20Kb/sec (2% of disk maximum)
- What were the problems with the original design?

Problems

- Blocks too small – 512 bytes
  - file index too large, transfer rate low.
- Consecutive file blocks not close together
  - Poor sequential access performance
- i-nodes far from data blocks
  - Poor file data access performance
- i-nodes of a directory not close together
  - Poor “ls” command performance
- No read-ahead

Block size too small

- Why not just make larger?
- What did they wind up doing?

Block size too small

- Most Unix file systems composed of many small files
- Increasing block size, increases file bandwidth but wastes space
- Choose 4K or 8K byte blocks
- Allow large blocks to be chopped into small ones
  - Called fragments
  - Used for little files and pieces at the end of files
  - Limit number of fragments per block to 2,4, or 8

Freelist

- Freelist leads to random allocation
- How was this addressed?

Random allocation problem

- Switch from freelist to bit map of free blocks
  - Easier to find contiguous free blocks
  - Bitmap: 01110000000111111110101111

Cylinder groups

- Divided disk into cylinder groups containing
  - Superblock
  - i-nodes
  - Bitmap of free blocks
  - Usage summary info
- Why introduce cylinder groups?

Cylinder groups

- Divided disk into cylinder groups containing
  - Superblock
  - i-nodes
  - Bitmap of free blocks
  - Usage summary info
- Why introduce cylinder groups? Used to:
  - Keep i-nodes near their data blcoks
  - keep i-nodes of a directory together
- Cylinder groups act like lots of little Unix file systems

Key to high performance: locality

- Two techniques for achieving locality
  - 1. don't let the disk fill up
    - Always find free space nearby
  - 2. paradox: spread unrelated things far apart
    - Room for related things to be placed together

Application of locality in BSD allocator

- Keep files in a directory in same cylinder group
  - Locality of i-nodes in a directory
  - Locality of files in a directory
- Spread out directories among the cylinder groups
  - Make room for locality within a directory
- Allocate runs of blocks within a cylinder group
  - locality of blocks in a file
- Switch to a different cylinder group after 48K
  - Prevent one file from filling a cylinder group

Layout policies: Global and local

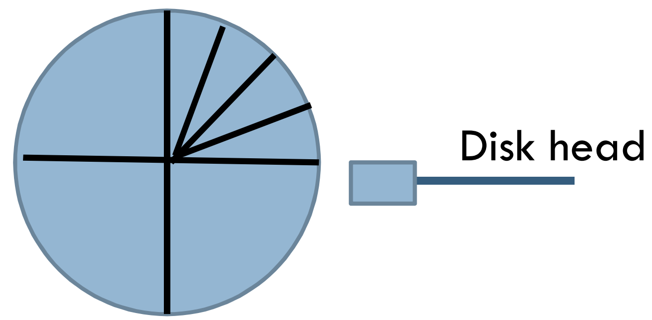
- Global
  - Allocate files and directories to cylinder group
  - Pick "optimal" next block for allocation
- Local
  - Handles request for specific blocks
  - If available, use it
  - If not free, check a sequence of alternatives

Alternative for local placement

- 1. Rotationally optimal block
- 2. Next block rotationally close on same cylinder
- 3. A block within cylinder group
- 4. Rehash cylinder group # to choose another cylinder group
- 5. Exhaustive search

Rotationally optimal placement

- Skip-sector allocation based on CPU speed and device characteristics



Performance improvements

- Able to get 20-40% of disk bandwidth on large files
- 10x-20x original Unix file system
- Better small file performance
- 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)
  - flock()
  - Process id of holder stored with lock; can reclaim the lock if process is no longer around
- Symbolic links
  - Inter-file system links
  - Links to directories

Enhancements made to system interface (cont'd)

- Atomic rename capability
  - rm name; ln name tmpName; rm tmpName

    versus
  - Rename tmpName name;
- Disk quotas

Problems for FFS

- Crash recovery
  - Synchronous writes of metadata
    - Scavenger program fsck
- Small file performance
- Ties to disk geometry
  - Head switch times
  - Disk arrays (RAIDs)

Three key features of paper

- Parametrize FS implementation for the hw it's running on
- Measurement-driven design decisions
- Locality "wins"

A major flaw

- Measurements derived from a single installation
- Ignored technology trends
  - A lesson for the future: don't ignore underlying hw characteristics
- Contrasting research approaches:
  - improve what you've got vs. design something new

The design and implementation of a Log-Structured File System (1991)

- What kind of paper is this?
  - New big idea?
  - Measurement paper?
  - Experiences/lessons learnt paper?
  - A system description?
  - Performance study?
  - Refute-conventional wisdom?
  - Survey paper?

Hardware trend

- What is the key hardware trend motivating this paper?

Hardware trend

- What is the key hardware trend motivating this paper?
  - Problem: CPU speeds increasing faster than I/O speeds
  - Result: program will all be I/O bound
  - Solution: decouple performance of programs from I/O
    - Take advantage of file data caches
    - Reduce number of synchronous operations
  - Just increase size of the file cache?

Just increase size of the file cache?

- Improves read performance
- Doesn't improve write performance much
- Why?

Just increase size of the file cache?

- Improves read performance
- Doesn't improve write performance much
- Why?
  - Data must still be written to disk for safety
    - Sooner is safer
    - Later reduces traffic (overwrites and deletes)
    - Metadata is a problem
- Assumption of paper: use caches for reads, but need another technique for writes

Approach

- Write data to disk sequentially in a log
  - Eliminate almost all seeks, hence write perf increases
- Write data in large pieces
  - Reduce number of accesses

Problem

- Problem achieving large sequential areas (extents)
  - Unix workload is mostly access to small files
  - Many of these are temporary files
  - Lots of metadata and seek overhead
  - 5 writes required to create a file
    - What are these?
  - LFS concentrates on improving small file performance

5 seeks to create a new file

- Create i-node for file
- Add file data
- Update directory (add new directory entry)
- Finalize i-node for file
- Update i-node for directory (modification time)

Two key challenges

- How to locate data in a log without a complete scan?
- How to find large extends of free space?

Two key challenges

- How to locate data in a log without a complete scan?
  - Soln: write index (i-nodes) to log as well
- How to find large extends of free space?
  - Soln: cleaning process & usage information
  - compromise between extents and blocks: fixed-size but large segments

Segment writes

- A segment write contains
  - Data for multiple files
  - Their i-node information
  - Directory data and i-node changes
  - Inode map (where to find the i-nodes for files)

How do you retrieve something from the log?

- Just find the inode, how?

How do you retrieve something from the log?

- Inode map
  - Given inumber of file, gives location of inode on disk
  - Cached in main memory to avoid disk accesses
  - Divided into blocks that are written to segments
  - Checkpoint region gives location of all inode map blocks
    - Periodically we will checkpoint whole log so recovery doesn't take too long

Checkpoints

- What is a checkpoint?
  - A position in the log at which all the FS structures are consistent and complete
- To perform a checkpoint:
  - Write out data, inodes, inode map blocks and segment usage table (described later) to log
  - In fixed checkpoint region write
    - The address of inode map blocks and segment usage table blocks
    - A pointer to last segment written
    - Current timestamp

Checkpoints (cont'd)

- Alternate between 2 fixed checkpoint regions for safety
- "Roll forward" to recover data after last checkpoint
  - How does it work?
  - What do you lose if you don't do roll forward?
  - Roll forward never implemented successfully

Maintaining free space

- This is the hardest part
  - What to do when log wraps around on disk?
  - Some data earlier in log is no longer valid (no longer "live"
  - We must take advantage of this to continue logging
- Choices
  - Threading: leave "live" data on disk and place new data in the now dead areas; Downside?
  - Copy live data, coalescing it, to new place in log; Downside?

LFS Solution: "segment cleaning"

- Divide disk into fixed-sized but large segments
- Copy new data to free segments (threading at segment level)
- To free up segments, copy live data from several segments to one new segment (packs live data together)
- Avoids copying segments with lots of live data

Problems

- How to identify live blocks in a segment?
- How to identify file and offset of each live block?
- How to update that file's inode with new location of live blocks?

Solution

- Segment summary block written at end of each segment
  - Identifies each piece of info in segment (file number/offset, etc.)
  - Summary block written after every partial segment write
  - Used to detect block liveness (inode still points to this block?)
  - Version number optimizes this for deleted/truncated files; How?
- Note: there is no free list or bitmap in LFS

Segment cleaning questions

- When?
- How many?
- Which segments?
- How should live data be sorted when written out?

Write cost

- Measure of how busy disk is per byte of new data written
  - Includes segment cleaning overhead
  - Ignore rotational latency – look just at # of bytes
  - 1.0 is perfect
  - 10 means only 1/10 of disk time is spent writing new data
- Write cost = (read segs + write segs +write new)/write new

Write cost (cont'd)

- If utilization (live data) of segments is u & we read N segs:
  - Read N segs
  - Write N*u old data
  - Leaves space for N*(1-u) of new data
  - Write cost=[N + N*u + N*(1-u)] / N*(1-u)
  - If u = 0, then no need to read segment and the write cost is 1
  - Note that u is not overall disk utilization!

Segment cleaning costs simulated

- Key to good performance: bimodal distribution of segments
  - Must be easy to find low-utilization segments
  - Therefore other segments should be very high in utilization

Simulation of greedy policy (cleaner chooses least-filled seg)

- Uniform: each file equal likelihood of being overwritten
  - No reorganization of data when written out
- Hot & cold: simulate locality of access
  - 10% of the files have 90% chance of being overwritten
  - Live blocks sorted by age when written
  - Attempt to provide a bimodal distribution of hot & cold data
- Surprising result: locality and “better” grouping make things worse!

Why?

- Each segment chosen as it passes cleaning threshold
- Cold segments take longer to cross threshold
- Many cold segments hover at this point, pinning down lots of free space
- Solution: treat hot and cold data differently
  - Treat cold segments as more valuable (once cleaned, they free stuff up for longer)
  - Hot segments will continue to fragment, so wait longer to clean them
- Value of a segment's free space depends on the stability of data in the segment

Problem: requires predicting future of segment usage

- Solution:
  - Approximate future usage by current age of data in segment
  - Choose highest ration of benefit to cost of cleaning
  - Benefit:
    - Amount of free space released: 1-u
    - Amount of time that space will stay free: use age of youngest block in segment
  - Cost of cleaning: 1 (for segment) plus u (amount still live)
  - Ratio: free space generated * age of data / cost=

    $= (1\text{-}u)*\text{age} / (1+u)$

Simulation

- Cost-benefit policy
  - Cleans cold segments at 75% utilization
  - Waits till 15% utilization for hot segments
  - Most segments cleaned are hot (90% of writes to hot segments)
  - Up to 50% better than greedy policy
  - Better than FFS cost for up to 80% utilization

Performance measurements

- Microbenchmarks:
  - Small file performance:
    - Much faster than SunOS at file creation and deletion
  - Large file performance:
    - Better than FFS on sequential and random writes
    - About the same on sequential and random reads
    - Much worse on sequential re-reads of randomly written data; Why?
- Cleaning overheads seem to be lower than the simulations

Locality

- Traditional file system
  - Logical locality on disk
  - Extra overhead for writes to maintain this locality
- LFS
  - Temporal locality
  - Temporal locality may match logical locality
  - When it doesn't LFS read performance will be worse

Key lesson

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

Problems with the paper

- No roll forward – what cost of complexity/performance?
- Performance measurements didn't include cleaning
- Read traffic not modeled
- 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?

Controversy

- Lots of controversy ensued
  - Debate includes performance issues on _real_ workloads
  - John Ousterhout @ Berkeley vs Margo Seltzer @ Harvard

Bottom line

- Very hard to come up with definitive benchmarks proving that one system is better than another
  - Can always find a scenario where one system design outperforms another
  - Difficult to extrapolate and make definitive conclusions based on benchmark tests