CSE 451: Operating Systems
Spring 2005

Module 16
Berkeley Log-Structured File System

Ed Lazowska
lazowska@cs.washington.edu
Allen Center 570

More on caching (applies both to FS and FFS)

- Cache (often called *buffer cache*) is just part of system memory
- It's system-wide, shared by all processes
- Need a replacement algorithm
  - LRU usually
- Even a small (4MB) cache can be very effective
- Today's huge memories => bigger caches => even higher hit ratios
- Many file systems "read-ahead" into the cache, increasing effectiveness even further

Caching writes, vs. reads

- Some applications assume data is on disk after a write (seems fair enough!)
- And the file system itself will have (potentially costly!) consistency problems if a crash occurs between syncs – i-nodes and file blocks can get out of whack
- Approaches:
  - "write-through" the buffer cache (synchronous – slow), or
  - "write-behind": maintain queue of uncommitted blocks, periodically flush (unreliable – this is the sync solution), or
  - NVRAM: write into battery-backed RAM (expensive) and then later to disk

So, you can make things better, but …

- As caches get big, most reads will be satisfied from the cache
- No matter how you cache write operations, though, they are *eventually* going to have to get back to disk
- Thus, most disk traffic will be write traffic
- If you eventually put blocks (i-nodes, file content blocks) back where they came from on the disk, then even if you schedule disk writes cleverly, there's still going to be a lot of head movement (which dominates disk performance) – so you simply won't be utilizing the disk effectively

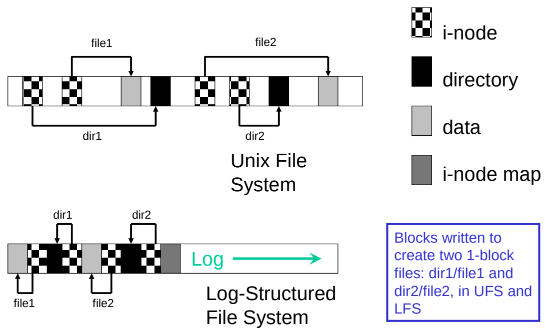
LFS inspiration

- Suppose, instead, what you wrote to disk was a log of changes made to files
  - log includes modified data blocks and modified metadata blocks
  - buffer a huge block ("segment") in memory – 512K or 1M
  - when full, write it to disk in one efficient contiguous transfer
    - right away, you've decreased seeks by a factor of 1M/4K = 250
- So the disk contains a single big long log of changes, consisting of threaded segments

LFS basic approach

- Use the disk as a *log*
- A log is a data structure that is written only at one end
- If the disk were managed as a log, there would be effectively no seeks
- The "file" is always added to sequentially
- New data and metadata (i-nodes, directories) are accumulated in the buffer cache, then written all at once in large blocks (e.g., segments of .5M or 1M)
- This would greatly increase disk write throughput

LFS vs. UNIX File System or FFS



LFS challenges

- Locating data written in the log
  - FFS places files in a well-known location, LFS writes data "at the end of the log"
- Even locating i-nodes!
  - in LFS, i-nodes too go in the log!
- Managing free space on the disk
  - disk is finite, and therefore log must be finite
  - so cannot just keep appending to log, ad infinitum!
    - need to recover deleted blocks in old part of log
    - need to fill holes created by recovered blocks
- (Note: Reads are the same as FS/FFS once you find the i-node – and writes are a ton faster)

Locating data and i-nodes

- LFS uses i-nodes to locate data, just like FS/FFS
- LFS appends i-nodes to end of log, just like data
  - makes them hard to find
- Solution
  - use another level of indirection: "i-node maps"
    - i-node maps map file #s (i-node #s) to i-node location
  - location of i-node map blocks are kept in a checkpoint region
    - checkpoint region has a fixed location
  - cache i-node maps in memory for performance

Free space management

- Reads are no different than in UNIX File System or FFS, once we find the i-node for a file
  - using the i-node map, which is cached in memory, find the i-node, which gets you to the blocks
- Every write causes new blocks to be added to the current "segment buffer" in memory
  - when segment is full, it is written to disk
- Over time, segments in the log become fragmented as we replace old blocks of files with new blocks
  - we can "garbage collect" segments with little "live" data and recover the disk space

Segment cleaning

- Log is divided into (large) segments
- Segments are "threaded" on disk (linked list)
  - segments can be anywhere
- Reclaim space by cleaning segments
  - read segment
  - copy live data to end of log
  - now have free segment you can reuse!
- Cleaning is an issue
  - costly overhead, when do you do it?

Detail: Cleaning

- The major problem for a LFS is cleaning, i.e., producing contiguous free space on disk
- A cleaner daemon "cleans" old segments, i.e., takes several non-full segments and compacts them, creating one full segment, plus free space
- The cleaner chooses segments on disk based on:
  - utilization: how much is to be gained by cleaning them
  - age: how likely is the segment to change soon anyway

LFS summary

- As caches get big, most reads will be satisfied from the cache
- No matter how you cache write operations, though, they are eventually going to have to get back to disk
- Thus, most disk traffic will be write traffic
- If you eventually put blocks (i-nodes, file content blocks) back where they came from, then even if you schedule disk writes cleverly, there's still going to be a lot of head movement (which dominates disk performance)

- Suppose, instead, what you wrote to disk was a log of changes made to files
  - log includes modified data blocks and modified metadata blocks
  - buffer a huge block ("segment") in memory – 512K or 1M
  - when full, write it to disk in one efficient contiguous transfer
    - right away, you've decreased seeks by a factor of 1M/4K = 250
- So the disk is just one big long log, consisting of threaded segments

- What happens when a crash occurs?
  - you lose some work
  - but the log that's on disk represents a consistent view of the file system at some instant in time
- Suppose you have to read a file?
  - once you find its current i-node, you're fine
  - i-node maps provide a level of indirection that makes this possible
    - details aren't that important

- How do you prevent overflowing the disk (because the log just keeps on growing)?
  - segment cleaner coalesces the active blocks from multiple old log segments into a new log segment, freeing the old log segments for re-use
    - Again, the details aren't that important

Tradeoffs

- LFS wins, relative to FFS
  - metadata-heavy workloads
    - small file writes
    - deletes

    (metadata requires an additional write, and FFS does this synchronously)
- LFS loses, relative to FFS
  - many files are partially over-written in random order
    - file gets splayed throughout the log

LFS history

- Designed by Mendel Rosenblum and his advisor John Ousterhout at Berkeley in 1991
  - Rosenblum went on to become a Stanford professor and to co-found VMware, so even if this wasn't his finest hour, he's OK
- Ex-Berkeley student Margo Seltzer (faculty at Harvard) published a 1995 paper comparing and contrasting LFS with conventional FFS, and claiming poor LFS performance in some realistic circumstances
- Ousterhout published a "Critique of Seltzer's LFS Measurements," rebutting her arguments
- Seltzer published "A Response to Ousterhout's Critique of LFS Measurements," rebutting the rebuttal
- Ousterhout published "A Response to Seltzer's Response," rebutting the rebuttal of the rebuttal

- Moral of the story
  - If you're going to do OS research, you need a thick skin
  - *Very* difficult to predict how a FS will be used
    - So it's hard to generate reasonable benchmarks, let alone a reasonable FS design
  - *Very* difficult to measure a FS in practice
    - depends on a HUGE number of parameters, involving both workload and hardware architecture