CSE 451: Operating Systems
Winter 2004

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 FS's "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 FS itself will have (potentially costly!) consistency problems if a crash occurs between syncs – i-nodes and file blocks can get out of whack
- Solutions:
  - "write-through" the buffer cache (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), or
  - log-structured file system (LFS): we'll talk about this next!

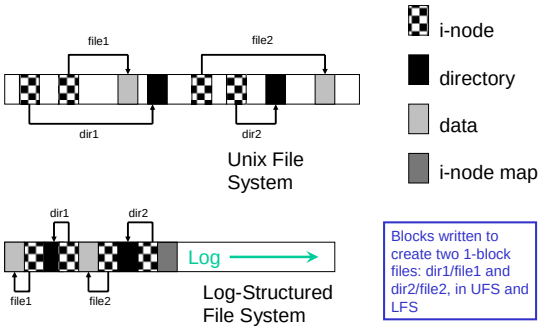
Impact of huge, highly effective read caches

- Most reads are satisfied from the buffer cache
- Thus, from the point of view of the disk, most traffic is write traffic
- So to speed up disk I/O, we need to make writes go faster
- But disk performance is limited ultimately by disk head movement
- With current file systems and *any* of the three alternatives (write-through, write-behind, or NVRAM), adding a block (extending a file) takes several writes (to the file and to the metadata), requiring several disk seeks

LFS: Basic idea

- An alternative is to 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 always append to log!
    - need to recover deleted blocks in old part of log
    - need to fill holes created by recovered blocks
- (Note: Reads are the same as UFS/FFS once you find the i-node – and writes are a ton faster)

Locating data and i-nodes

- LFS uses i-nodes, like FFS
- LFS appends i-nodes to end of log, 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

- Append-only quickly eats up all disk space
  - need to recover deleted blocks
- Solution
  - divide log into (large) segments
  - thread segments 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 a big problem
    - costly overhead, when do you do it?
      - "idleness is not sloth"

Detail: LFS data structures

- i-nodes: as in UNIX, i-nodes contain physical block pointers for files
  - written as part of the segment
- i-node map: a table indicating where each i-node is on the disk
  - i-node map blocks are written as part of the segment; a table in a fixed checkpoint region on disk points to those blocks
- Segment summary: info on every block in a segment
- Segment usage table: info on the amount of "live" data in a block

Detail: LFS read and write

- Every write causes new blocks to be added to the current segment buffer in memory; when that segment is full, it is written to disk
- Reads are no different than in UNIX File System or FFS, once we find the i-node for a file (in LFS, using the i-node map, which is cached in memory)
- Over time, segments in the log become fragmented as we replace old blocks of files with new blocks
- Problem: in steady state, we need to have contiguous free space in which to write

Detail: Cleaning

- The major problem for a LFS is cleaning, i.e., producing contiguous free space on disk
- A cleaner daeman "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

- Basic idea is to handle reads through caching and writes by appending large segments to a log
- Greatly increases disk performance on writes, file creates, deletes, ….
- Reads that are not handled by buffer cache are same performance as normal file system
- Pretty hairy
- Requires cleaning daemon to produce clean space, which takes additional CPU time
- A noble experiment, but is it worth it???

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