CSE 451: Operating Systems
Autumn 2009

Module 17
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

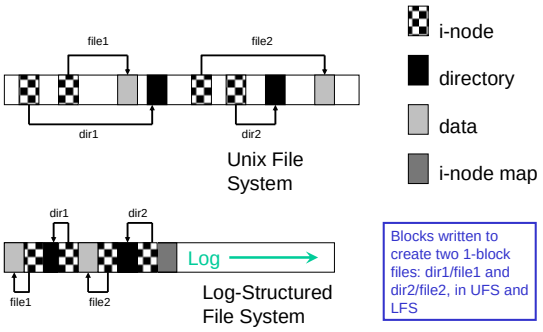
LFS inspiration

- 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
- Sounds simple – but really complicated under the covers

LFS vs. UNIX File System or FFS



LFS Challenges

- There are two main challenges with LFS:
  - 1. locating data written in the log
    - FFS places files in a well-known location, LFS writes data "at the end of the log"
  - 2. managing free space on the disk
    - disk is finite, and therefore log must be finite
    - cannot always append to log!
      - need to recover deleted blocks in old part of log
      - need to fill holes created by recovered blocks

LFS: locating data

- FFS uses i-nodes to locate data blocks
  - i-nodes pre-allocated in each cylinder group
  - directories contain locations of i-nodes
- 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 i-node #s to i-node location
  - so how do you find the i-node map?
    - after all, changes to it must be appended to the log
    - 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

LFS: free space management

- LFS: append-only quickly eats up all disk space
  - need to recover deleted blocks
- Solution:
  - fragment log into segments
  - thread segments on disk
    - 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"

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 vs. JFS
  - JFS is "robust" like LFS, but data must eventually be written back "where it came from" so disk bandwidth is still an issue