CS 537
Lecture 14
Optimized File Systems

Michael Swift

Goals

- Unix FS largely ignorant of locality
  - Puts inodes, data blocks anywhere on disk
- OS allocates LBNs (logical block numbers) to meta-data, file data, and directory data
  - Workload items accessed together should be close in LBN space
  - Leverage **temporal locality** with **spatial locality** on disk
- Implications
  - Large files should be allocated sequentially
  - Files in same directory should be allocated near each other
  - Data should be allocated near its meta-data
- Meta-Data: Where is it stored on disk?
  - Embedded within each directory entry
  - In data structure separate from directory entry
    - Directory entry points to meta-data

More Recent File Systems

- BSD Unix FFS
  - what's at the heart of most UNIX file systems
- LFS
  - a research file system originally from Berkeley

BSD UNIX FFS

- FFS = "Fast File System"
  - original (i.e. 1970's) file system was very simple and straightforwardly implemented
    - but had very poor disk bandwidth utilization
    - why? far too many disk seeks on average
      - From directories to inodes, from inodes to data, and between data blocks
- BSD UNIX folks did a redesign in the mid '80's
  - FFS: improved disk utilization, decreased response time
  - McKusick, Joy, Fabry, and Leffler
  - basic idea is FFS is aware of disk structure
    - I.e., place related things on nearby cylinders to reduce seeks

Review: Inodes and Path Search

- Unix Inodes are NOT directories
  - they describe where on disk the blocks for a file are placed
    - directories are just files, so each directory also has an inode that describes where the blocks for the directory is placed
- Directory entries map file names to inodes
  - to open "/one", use master block to find inode for "/" on disk
    - open "/", look for entry for "one"
    - this gives the disk block number for inode of "one"
  - read the inode for "one" into memory
    - this inode says where the first data block is on disk
    - read that data block into memory to access the data in the file

Data and Inode placement

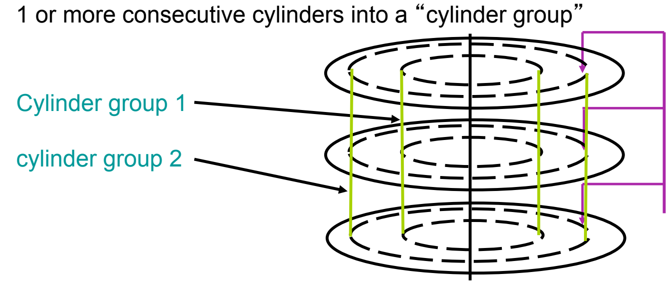
- Original (non-FFS) unix FS had two major problems:
  - 1. data blocks are allocated randomly in aging file systems (using linked list)
    - blocks for the same file allocated sequentially when FS is new
    - as FS "ages" and fills, need to allocate blocks freed up when other files are deleted
      - problem: deleted files are essentially randomly placed
      - so, blocks for new files become scattered across the disk!
  - 2. inodes are allocated far from blocks
    - all inodes at beginning of disk, far from data
    - traversing file name paths, manipulating files, directories requires going back and forth from inodes to data blocks
  - BOTH of these generate many long seeks!

Cylinder groups

- FFS addressed these problems using notion of a cylinder group
  - disk partitioned into groups of cylinders
  - data blocks **from a file** all placed in same cylinder group
  - files in **same directory** placed in same cylinder group
  - **inode for file** in same cylinder group as **file's data**
- Introduces a free space requirement
  - to be able to allocate according to cylinder group, the disk must have free space scattered across all cylinders
    - Need index of free blocks/inodes within a cylinder group
  - in FFS, 10% of the disk is reserved just for this purpose!
    - good insight: keep disk partially free at all times!
    - this is why it may be possible for df to report >100%

Clustering related objects in FFS

- 1 or more consecutive cylinders into a "cylinder group"



  - Key: can access any block in a cylinder without performing a seek. Next fastest place is adjacent cylinder.
  - Tries to put everything related in same cylinder group
  - Tries to put everything not related in different group (?!)

File Buffer Cache (not just for FFS)

- Exploit locality by caching file blocks in memory
  - cache is system wide, shared by all processes
  - even a small (4MB) cache can be very effective
  - many FS's "read-ahead" or "prefetch" into buffer cache
- Caching writes
  - some apps assume data is on disk after write
    - need to "write-through" the buffer cache
  - Or "write-behind": maintain queue of uncommitted blocks, periodically (~30 seconds) flush. Unreliable!
    - Fsync() forces a flush
- Buffer cache issues:
  - competes with VM for physical frames
    - integrated VM/buffer cache?
  - need replacement algorithms here
    - LRU usually

Log-Structured File System (LFS)

- LFS was designed in response to two trends in workload and disk technology:
  - 1. Disk bandwidth scaling significantly (40% a year)
    - but, latency is not
  - 2. Large main memories in machines
    - therefore, large buffer caches
      - absorb large fraction of read requests in caches
    - can use for writes as well
      - coalesce small writes into large writes
- LFS takes advantage of both to increase FS performance
  - Now used extensively in solid-state disks.

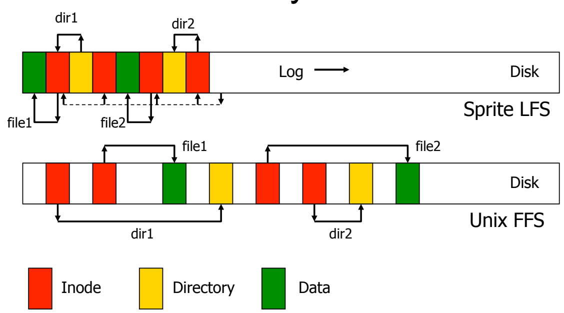
FFS problems that LFS solves

- FFS: placement improved, but can still have many small seeks
  - possibly related files are physically separated
  - inodes separated from files (small seeks or rotations)
  - directory entries separate from inodes
- FFS: metadata required **synchronous writes** for correctness after a crash
  - Example: need to ensure free inode bitmap updated before adding inode to a directory
  - with small files, most writes are to metadata
  - synchronous writes are very slow: cannot use scheduling to improve performance

LFS: The Basic Idea

- Treat the entire disk as a single log for appending
  - collect writes in the disk buffer cache, and write out the entire collection of writes in one large request
    - leverages disk bandwidth with large sequential write
    - no seeks at all! (assuming head at end of log)
  - all info written to disk is appended to log
    - data blocks, attributes, inodes, directories, .etc.
- Sounds simple!
  - but it's really complicated under the covers

LFS Disk Layout Compared to Unix Layout

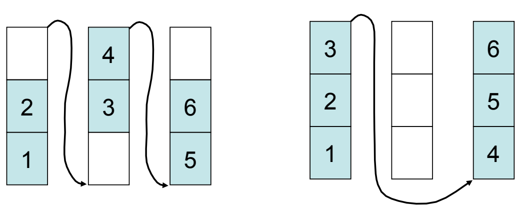


LFS Challenges

- 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 Threaded Segments

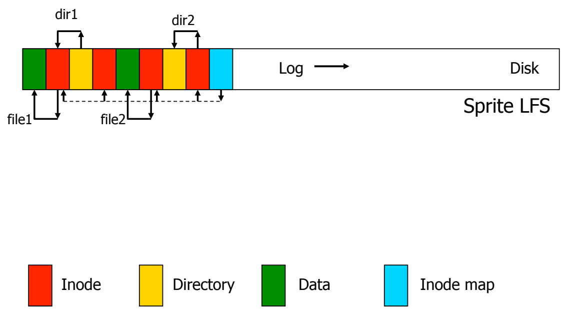
- Sprite LFS uses a hybrid scheme.
  - Disk divided into fixed size segments.
    - Threaded between segments (connected as a list).
    - Compaction within a segment.
  - Segment size chosen so that transfer time is much greater than access time: 512 KB or 1 MB.



LFS: locating data

- FFS uses inodes to locate data blocks
  - inodes preallocated in each cylinder group
  - directories contain locations of inodes
- LFS appends inodes to end of log, just like data
  - makes them hard to find
- Solution:
  - use another level of indirection: inode maps
  - inode maps map file #s to inode location
  - location of inode map blocks are kept in a checkpoint region
  - checkpoint region has a fixed location
  - cache inode maps in memory for performance

Extra Metadata: Inode Map



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"

An Interesting Debate

- Ousterhout vs. Seltzer
  - OS researchers have very "energetic" personalities
    - famous for challenging each others' ideas in public
  - Seltzer published a 1995 paper comparing and contrasting BSD LFS with conventional FFS
    - Ousterhout published a "critique of Seltzer's LFS Measurements", rebutting arguments that LFS performs poorly in some situations
    - 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:
    - *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, including workload and hardware architecture

