CSE 451: Operating Systems
Winter 2005

FFS and LFS

Steve Gribble

File System Implementations

- We've looked at disks and file systems generically
  - now it's time to bridge the gap by talking about specific file system implementations
- We'll focus on two:
  - BSD Unix FFS
    - what's at the heart of most UNIX file systems
  - LFS
    - a research file system originally from Berkeley

FFS

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
- 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

Data and Inode placement

- Original (non-FFS) unix FS had two major problems:
  - 1. data blocks are allocated randomly in aging file systems
    - 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
  - 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%

Other FFS innovations

- I lied: original UNIX FS had 1KB blocks, not 4KB
  - even more seeking == less bandwidth
  - small max file size (function of block size)
- FFS fixes by using a larger block (4KB)
  - allows for very large files (4TB)
  - but, introduces internal fragmentation
    - on average, each file wastes 2KB!
    - worse, in practice, average file size is only about 1KB!
  - fix: introduce “fragments”
    - 1KB pieces of a block
- Old FS was unaware of disk parameters
  - FFS: parameterize FS according to disk and CPU characteristics
    - e.g.: account for CPU interrupt and processing time to layout sequential blocks
      - skip according to rotational rate and CPU latency!

LFS

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
  - Rosenblum and Ousterhout (Berkeley, '91)
    - note: Rosenblum went on to become Stanford prof, and to co-found VMware, inc!

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 bandwith 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

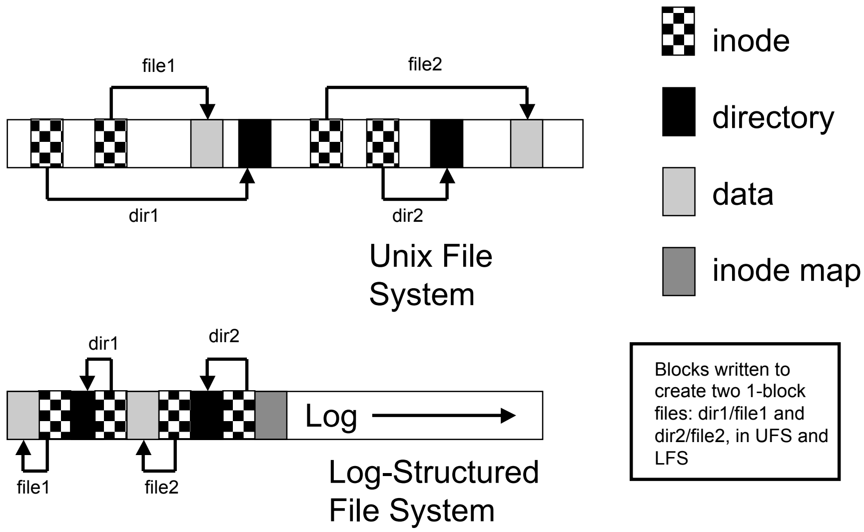
LFS Challenges

- 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 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

LFS vs. FFS



LFS: reads and writes

- Every write causes new blocks to be added to the current “segment buffer” in memory
  - when segment is full, it is written to the disk
- Reads are no different than in FFS
  - find the inode (using inode map in LFS), then use inode to find the file blocks
- Over time, though, segments become “fragmented” as we replace old blocks of a file with new blocks
  - need to get rid of this fragmentation so we have contiguous free space to write

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”
    - which segments do you clean? turns out to be really tricky.