12: FFS,LFS and other file systems

Last Modified:
10/25/2002 9:26:22 AM

-1

Building a file system

❑ To build a file system from an array of disk sectors we have to decide things like
   ❍ Must files be allocated contiguously?
   ❍ If not how will be find the pieces?
   ❍ What information is stored about each file in the directory?
   ❍ Where do we put new files that are created?
   ❍ What do we do when files grow or shrink?
   ❍ How do we recover the FS after a crash?

-2

Answers?

❑ We are going to look at two different file systems
   ❍ Fast File System (FFS)
   ❍ Log-Structured File Systems (LFS)

-3

How are they the same?

❑ Both allow files to be broken into multiple pieces
❑ Both use fixed sized blocks (for the most part)
❑ Both use the inode structure we discussed last time

-4

Fast File System

❑ Fast? Well faster than original UNIX file system (1970's)
   ❍ Original system had poor disk bandwidth utilization
   ❍ Remember why that is a problem? Too many seeks
❑ BSD UNIX folks redesigned in mid 1980's
   ❍ Improved disk utilization by breaking files into larger pieces
   ❍ Made FFS aware of disk structure (cylinder groups) and tried to keep related things together
   ❍ Other semi-random improvements like support for long file names etc.

-5

Managing Free Space

❑ Break disk into cylinder groups and then into fixed size pieces called blocks (commonly 4 KB)
❑ Each cylinder group has a certain number of blocks
   ❍ Cylinder group's free list maps which blocks free and which taken
   ❍ Cylinder groups also store a copy of the superblock which contains special bootstrapping information like the location of the root directory (replicated)
   ❍ Cylinder groups also contain a fixed number of I-nodes
   ❍ Rest of blocks used to store file/directory data

-6

Inodes in FFS

❒ In FFS, fixed number of inodes at FS format time
  ○ When create file, pick an inode, will never move (so directory entry need not be updated)
  ○ Can run out of inodes and not be able to create file even though there is free space

Creating a new file

❒ In the pre-FFS UNIX file system
  ○ Free list for the entire disk
  ○ Started out ordered nicely such that if ask for 3 free blocks likely to get 3 together
  ○ Randomized over time as files created and deleted such that pieces of a new file scattered over the disk
  ○ Also when create new file need a new inode too
    • All inodes at beginning of disk, far from the data
  ○ When read through a file likely to be seeks between each block – slow!

FFS

❒ Divide the disk into cylinder groups
  ○ Try to put all blocks of file into same cylinder group
  ○ Inodes in each cylinder group so inodes near their files
  ○ Try to put files in the same directory into the same cylinder group
  ○ Big things forced into new cylinder group
❒ Is this fundamentally a new approach?
  ○ Not really…space within a cylinder group gets treated just like whole disk was
  ○ Space in cylinder group gets fragmented etc
  ○ Basically sort files into bins so reduce the frequent long seeks

Cylinder Groups

❒ To keep things together must know when to keep things apart
  ○ Put large files into a different cylinder group
❒ FFS reserves 10% of the disk as free space
  ○ To be able to sort things into cylinder groups, must have free space in each cylinder group
  ○ 10% free space avoids worst allocation choice as approach full (ex. One block in each cylinder group)

Other FFS Improvements

❒ Small or large blocks?
  ○ Orig UNIX FS had small blocks (1 KB)
  ○ $\frac{1}{4}$ less efficient BW utilization
❒ Larger blocks have problems too
  ○ For files < 4K , results in internal fragmentation
  ○ FFS uses 4K blocks but allows fragments within a block
  ○ Last < 4K of a file can be in fragments
❒ Exactly 4K?
  ○ FFS allows FS to be parameterized to the disk and CPU characteristics
  ○ Another cool example: when laying out logically sequential blocks skip a few blocks in between each to allow for CPU interrupt processing so don't just miss the blocks and force a whole rotation

Update In Place

❒ Both the original UNIX FS and FFS were update-in-place
❒ When block X of a file is written then forever more, reads or writes to block X go to that location until file deleted or truncated
❒ As things get fragmented need "defragmenter" to reorganize things

Another Problem with Update-in-place

Poor crash recovery performance

- Some operations take multiple disk requests so are impossible to do atomically
  - Ex. Write a new file (update directory, remove space from free list, write inode and data blocks, etc.)
- If system crashes (lose power or software failure), there may be file operations in progress
- When system comes back up, may need to find a fix these half done operations
- Where are they?
  - Could be anywhere?
  - How can we restore consistency to the file system?

-13

Fixed order

- Solution: Specify order in which FS ops are done
- Example to add a file
  - Update free list structures to show data block taken
  - Write the data block
  - Update free list structures to show an inode take
  - Write the inode
  - Add entry to the directory
- If crash occurs, on reboot scan disk looking for half done operations
  - Inodes that are marked taken but are not referred to by any directory
  - Data blocks that are maked taken but are not referred to by any inode

-14

Fixed order (con't)

- We've found a half done operation now what?
  - If data blocks not pointed to by any inode then release them
  - If inode not pointed to by any directory link into Lost and Found
- Fsck and similar FS recovery programs do these kinds of checks
  - Problems can be anywhere with update in place so must scan the whole FS!!
- Problems?
  - Recovery takes a long time! (System shutdown uncleanly..checking your FS.. For the next 10 minutes!)
  - Even worse(?) normal operation takes a long time because specific order = many small synchronous writes = slow!

-15

Write-Ahead Logging (Journaling)

- How can we solve problem of recovery in update in place systems?
- Borrow a technique from databases!
  - Logging or journaling
- Before perform a file system operation like create new file or move a file, make a note in the log
- If crash, can simply examine the log to find interrupted operations
  - Don't need to examine the whole disk

-16

Checkpoints

- Periodically write a checkpoint to a well known location
- Checkpoint establishes a consistent point in the file system
- Checkpoint also contains pointer to tail of the log (changes since checkpoint written)
- On recovery start at checkpoint and then "roll forward" through the log
- Checkpoint points to location system will use for first log write after checkpoint, then each log write has pointer to next location to be used
  - Eventually go to next location and find it empty or invalid
- When write a checkpoint can discard earlier portions of the log

-17

Problems with write-ahead logging

- Do writes twice
- Once to log and once to "real" data (still organized like FFS)
- Surprisingly can be more efficient than update-in-place!
  - Batched to log and then replayed to "real" in relaxed order (elevator scheduling on the disk)

-18

Recovery of the file system (not your data)

❒ Write-ahead logging or journaling techniques could be used to protect FS and user data
❒ Normally just used to protect the FS
❒ I look like a consistent FS but your data may be inconsistent
  ○ Even if some of the last files you were modifying are inconsistent still better than FS corrupted (insert bootable device please ☺)

❒ Still, why do we need a "real" data layout why couldn't the log be the FS? Then user data would get same benefits?

-19

Log-Structured File System

❒ Treat the disk as an infinite append only log
  ○ Data blocks, inodes, directories everything written to the log
❒ Batch writes in large units called segments (~ 1 MB)
❒ Garbage collection process called cleaner reclaims holes in the log to regenerate large expanses of free space for log writes
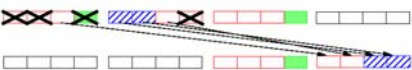
-20

Log Writes and Cleaning

New writes create holes in the log

Cleaner copies live blocks to a new segment

-21

Finding Data

❒ Inodes used to find data blocks
❒ Finding inodes?
  ○ Directories specify location of a file's inode

❒ In an FSS, inodes are preallocated in each cylinder group and a given file's inode never moves (update in place)
❒ In an LFS, inodes written to the log and so they move

-22

Chain Reaction

❒ LFS is not update in place when file block written its location changes
  ○ File location changes => entry in inode (and possibly also indirect blocks) changes => Inode (and indirect blocks) must be rewritten
❒ Parent directory contains location of inode – must directory be rewritten too?
  ○ If so then all directories to root must be rewritten?
❒ No! – introduce another level of indirection
  ○ Directory says inode *number* (rather than location)
  ○ Inode map to map inode number to current location

-23

Inode Map

❒ Inode map maps inode numbers to inode location
  ○ Map kept in a special file the ifile
❒ When a file's inode is written, its parent directory does not change only the ifile does
❒ Caching inode map (ifile) in memory is pretty important for good performance
  ○ How big is this? Approx 2*4bytes(inode number and disk LBA) = 8 bytes for every file/directory in the file system
  ○ Can grow dynamically unlike FFS

-24

Checkpoint

- ❑ Like in Write Ahead Logging, write periodic checkpoints
  - ❍ Kind of like FFS superblocks
- ❑ Checkpoint region has a fixed location
  - ❍ Actually two fixed locations and alternate between them in case die in middle of writing and leave corrupt
  - ❍ Checksums to verify consistent; Timestamps say which is most recent
- ❑ Whats in checkpoint?
  - ❍ Location of inode for ifile and inode number of the root directory
  - ❍ Location of next segment will write log to
  - ❍ Basic FS parameters like segment size, block size, etc

LFS Pros and Cons

- ❑ What is good about this?
  - ❍ Leverage disk BW with large sequential writes
  - ❍ Near perfect write performance
  - ❍ Read performance? Good if read the same way as you write and many reads absorbed by caches
  - ❍ Cleaning can often be done in idle time
  - ❍ Fast efficient crash recovery
  - ❍ User data gets benefits of a log
- ❑ What's bad about this?
  - ❍ Cleaning overhead can be high – especially in the case of random updates to a full disk with little idle time
  - ❍ Reads may not follow write patterns (they may not follow directory structure either though!)
  - ❍ Additional metadata handling (inodes, indirect blocks and ifile rewritten frequently)

Cleaning Costs

- ❑ We are going to focus on talking about the problem of high cleaning costs
- ❑ Often cleaning is not a problem
  - ❍ If there is plenty of idle time (many workloads have this), cleaning costs hidden
  - ❍ Also if locality to writes, then easier to clean
  - ❍ If disk not very full then, segments clean themselves (overwrite everything in old segments before run out of free spaces for new writes)
- ❑ So when is cleaning a problem?
  - ❍ Cleaning expensive when random writes to full disk with no idle time

High Cleaning Costs

Random writes, full disk (little free space), no idle time = Sky-rocketting cleaning costs

For every 4 blocks written, also read 4 segments and write 3 segments!

Copy cleaning vs Hole-plugging

- ❑ Alternate cleaning method?
  - ❍ Hole-plugging = Take one segment break extract the live data and use it to plug holes in other segments
  - ❍ This will work well for full disk, random updates, little idle time!!
- ❑ Hole-plugging avoid problems with copy cleaning but transfers many small blocks which uses the disk less efficiently
- ❑ Could we get the best of both worlds?
  - ❍ First we have to talk about how to quantify the tradeoffs

Write Cost

- ❑ How do we quantify the benefits of large I/Os vs the penalty of copying data?
- ❑ Original LFS paper evaluated efficiency of cleaning algorithms according to the following metric
  - ❍ $(DataWritten_{NewData} + DataRead_{Cleaning} + DataWritten_{Cleaning})/ DataWritten_{NewData}$
  - ❍ Quantifies cleaning overhead in terms of the amount of data transferred while cleaning
  - ❍ What about the impact of large vs small transfers?

Cost of Small Transfers

- ❐ Quantify overhead due to using the disk inefficiently
  - ○ $TransferTime_{Actual}/TransferTime_{Ideal}$
  - ○ Where $TransferTime_{Actual}$ includes seek, rotational delay and transfer time and $TransferTime_{Ideal}$ only includes transfer time
- ❐ By factoring in the cost of small transfers, we see the cost of holeplugging

-31

Overall Write Cost

- ❐ Ratio of actual to ideal costs where
  - ○ Actual includes cost of garbage collection and includes seek/rotational latency for each transfer
  - ○ Ideal includes only cost of original writes to an infinite append only log – no seek/rotational delay and no garbage collection
- ❐ Now we have a metric that lets us compare hole-plugging to copy-cleaning
  - ○ System can use this to choose which one to do!
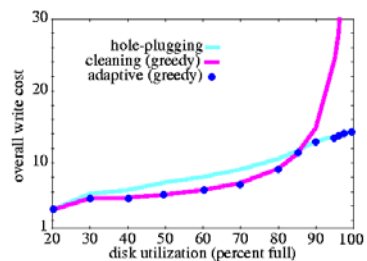 - ○ Adaptive cleaning ☺

-32

Adaptive Cleaning

- ❐ When starting to run out of segments, do garbage collection
- ❐ Look in special file called the segmap that tells you how full each segment is
  - ○ When rewrite a block in a segment, write in segmap file that segment is one block less full
- ❐ Estimate cost to do copy cleaning and cost to do hole-plugging
  - ○ Compute overall write cost by seeing how full segments are
- ❐ Choose the most cost effective method this time
  - ○ Can choose a different one next time ☺
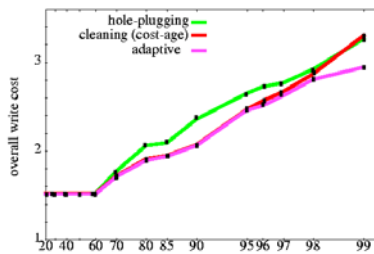
-33

Adaptive Cleaning For Random Update Workload
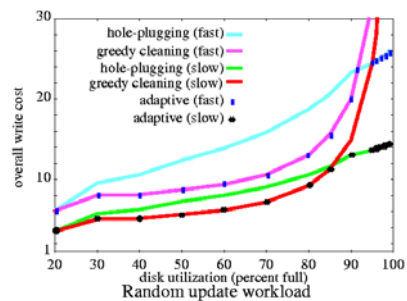

Random update workload
Assume no idle time to clean

-34

Adaptive Cleaning for Normal Usage Trace

Berkeley Auspex trace
Assume no idle time to clean

-35

As Technology Changes

Random update workload

-36

Other factors?

- ❑ How does this layout work for reads?
  - ○ Good if read in the same way you write
  - ○ Well until start reorganizing during cleaning (hole-plugging is worse than copy cleaning here)
  - ○ Special kind of hole-plugging that writes back on top of where it used to be?
- ❑ Accounting for additional metadata handling in the cache?
  - ○ Modifying the write cost metric to account for "churn" in the metadata?
  - ○ Model FFS in this same way

Improving FFS also

- ❑ Extent like performance (McVoy)
- ❑ FFS-realloc (McKusick)
- ❑ FFS-frag and FFS-nochange(Smith)
- ❑ Colocating FFS (Ganger)
- ❑ Soft Updates (Ganger)

Other FS?

- ❑ Update-in-place
  - ○ FAT
  - ○ ext2 (extent based rather than fixed size blocks)
- ❑ Write-ahead Logging (journaling)
  - ○ NTFS
  - ○ ReiserFS (B+ tree indices, optimizations for small files)
  - ○ SGI's XFS (extent based and B+ trees)
  - ○ Ext3 (journaling version of ext2)
  - ○ Veritas VxFS
  - ○ BeOS's BeFS
- ❑ No Update?
  - ○ CD-ROM FS no update and often contiguous allocations (why does that make sense?)

Network/Distributed FS

- ❑ Sun's NFS
- ❑ CMU's AFS and Coda
  - ○ Transarc's (now IBM's) commercial AFS
  - ○ Intermezzo (Linux Coda like system)
- ❑ Netware's NCP
- ❑ SMB

Multiple FS?

- ❑ With all these choices, do we really have to choose just one FS for our OS?
- ❑ If we want to allow multiple FS in the same OS, what would be have to do?
  - ○ Merge them into one directory hierarchy for the user
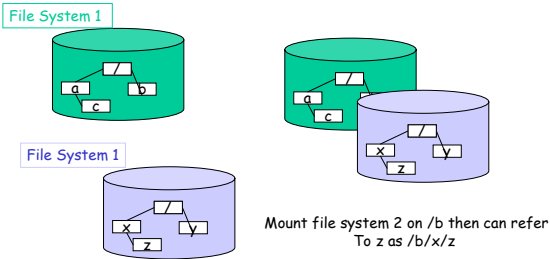 - ○ Make them obey a common interface for the rest of the OS

Mount points

- ❑ Another kind of special file interpreted by the file system is a mount point
- ❑ Contains information about how to access the root of a separate FS tree (device information if local, server information if remote, type of FS, etc.)

Mount Points

Mount file system 2 on /b then can refer
To z as /b/x/z

Common Interface?

❒ Different FS usually need the same "hooks" into the OS
  ○ Some need special things?
❒ Vnode interface
  ○ Proposed in 1986
  ○ Allow multiple FS in the same OS (without ugly case statements everywhere)
  ○ Allow FS to work on multiple OSes? (that's harder)

---

struct vnode

❒ One vnode structure for every opened (in-use) file
❒ Contains:
  ○ Array of pointers to procedures to implement basic operations on files
  ○ Pointer to parent FS
  ○ Pointer to FS that is mounted on top of this file (if any)
  ○ Reference count so know when to release the vnode

Vnode ops

❒ Open, close, create, remove, read, write
❒ Mkdir, rmdir, readdir
  ○ You don't know what that FS's directory format will be
❒ Symlink, Link, readlink (soft/hard links)
❒ Getattr, setattr, access (get/set/check attributes like permissions)
❒ Fsync
❒ Seek
❒ Map, getpage, putpage (memory map a file)
❒ Ioctl (misc I/O control ops)
❒ Rename
❒ ...

---

struct vfs

❒ One vfs structure in the OS for each mounted FS
❒ Contains:
  ○ Array of pointers to procedures that implement basic operations on file systems
  ○ FS type
  ○ Native block size
  ○ Pointer to vnode this FS is mounted on

---

vfsops

❒ Mount: procedure called to mount a FS of this type on a specified vnode
❒ Unmount: procedure to release this FS
❒ Root: return root vnode of this Fs
❒ Statvfs: return research usage status of the FS
❒ Sync: flush all dirty memory buffers to persistent storage managed by this FS
❒ Vget: turn a fileId into a a pointer to vnode for a specific file
❒ Mountroot: mount this FS as the root FS on this host
❒ Swapvp: return vnode of file in this FS to which the OS can swap

Evolving vnode interface?

❒ Kleiman86 => Rosenthal90

Do we need FS interface?

❒ FS Interface
  ○ Giving things file names seems a bit arbitrary
❒ FS hierarchy vs directory search
❒ People like to find information both ways
  ○ I know exactly what I want don't bother looking for me I will get it myself
  ○ Give me everything matching these characteristics

Outtakes