

Lecture 20: Reliable, High Performance Storage

CSC 469H1F / 2208 H1F
Fall 2007
Angela Demke Brown



11/22/2007

1

Review

- We've looked at fault tolerance via server replication
 - Continue operating with up to f failures
 - Recovery requires restoring state on failed server
 - With replicated state machines, can rebuild state from non-faulty replicas *unless another fault occurs*
 - May require each server to maintain local state on *stable storage* to allow recovery to consistent global state following catastrophic failure



11/22/2007

CSC469

Overview

- Now we're going to look at some example file and storage systems designed for high performance and reliability
 - BSD Unix Fast File System (FFS) (review)
 - Log-structured File System (LFS)
 - Soft Updates
 - Redundant Array of Inexpensive Disks (RAID)



11/22/2007

CSC469

Basic Disk & File System properties

- Disks provide large-scale non-volatile storage
- Access time determined by:
 - Seek time
 - Rotational latency
 - Transfer time
- Good performance depends on
 - Reducing seeks
 - Creating large transfers
- File systems provide logical organization and management of data on disk



11/22/2007

CSC469

Unix Inodes: Indirection & Independence

File size grows dynamically, allocations are independent
 Problem: hard to achieve closeness and amortization

CSC469 11/22/2007

Original Unix File System

- Recall FS sees storage as linear array of blocks
 - Each block has a *logical block number (LBN)*

Superblock

Default usage of LBN space

- Simple, straightforward implementation
 - Easy to implement and understand
 - Poor bandwidth utilization (lots of seeking)
 - Poor reliability (lose superblock, lose everything)

CSC469 11/22/2007

Cylinder Groups

- BSD FFS addressed placement problems using the notion of a **cylinder group** (aka *allocation groups* in lots of modern FS's)
 - Data blocks in same file allocated in same cylinder group
 - Files in same directory allocated in same cylinder group
 - Inodes for files allocated in same cylinder group as file data blocks
 - Superblock *replicated* to improve reliability

Cylinder group organization

CSC469 11/22/2007


Log-structured File System

- The Log-structured File System (LFS) was designed in response to two trends in workload and technology:
 - Disk bandwidth is scaling significantly (40% a year)
 - Latency is not
 - Large main memories in machines
 - Large buffer caches
 - Absorb large fraction of read requests
 - Can use for writes as well
 - Coalesce small writes into large writes
- LFS takes advantage of both of these to increase FS performance
 - Rosenblum and Ousterhout (Berkeley, '91)

CSC469 11/22/2007


LFS vs. FFS

- LFS addresses some problems with FFS
 - Placement is improved, but still have many small seeks
 - Possibly related files are physically separated
 - Inodes separated from files (small seeks)
 - Directory entries separate from inodes
 - Metadata requires synchronous writes
 - With small files, most writes are to metadata (synchronous)
 - Synchronous writes very slow



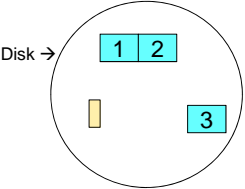
LFS Approach

- Treat the disk as a single log for appending
 - Collect writes in disk cache, write out entire collection in one large disk request
 - Leverages disk bandwidth
 - No seeks (assuming read/write head is at end of log)
 - All info written to disk is appended to log
 - Data blocks, attributes, inodes, directories, etc.
- Simple, eh?
 - Alas, only in abstract

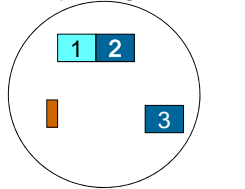






Example: Update file in FFS

Before...




After updating blocks 2 & 3



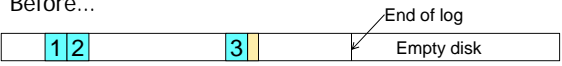
 File inode before write  File data before write	 File inode after write  File data after write
---	---

- Blocks (data & metadata are updated in place

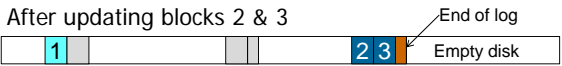







Example: Update file in LFS


Before...



After updating blocks 2 & 3




 File inode before write  File data before write  File inode after write  File data after write  Obsolete blocks	<ul style="list-style-type: none"> • Modified blocks (data & metadata) are written to new location at end of log
---	---



LFS Challenges


- LFS has two challenges it must address for it to be practical
 1. Locating data written to the log
 - FFS places files in a location, LFS writes data "at the end"
 2. Managing free space on the disk
 - Disk is finite, so log is finite, cannot always append
 - Need to recover deleted blocks in old parts of log



CSC469 11/22/2007

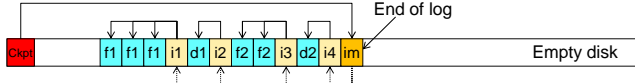
LFS: Locating Data

- FFS uses inodes to locate data blocks
 - Inodes pre-allocated in each cylinder group
 - Directories contain locations of inodes
- LFS appends inodes to end of the log just like data
 - Makes them hard to find
- Approach
 - Use another level of indirection: *Inode maps*
 - *Inode maps* map file #s to inode location
 - Inode map blocks are written to log like any other block
 - Location of inode map blocks kept in *checkpoint region*
 - Checkpoint region has a fixed location
 - Cache inode maps in memory for performance




CSC469 11/22/2007

LFS Layout



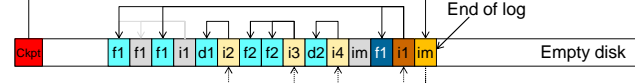
- ix File inode before write
- fx File data before write
- im Inode map block
- Checkpoint region (fixed location)

- Suppose we just created directory d1 with file f1, and directory d2 with file f2.




CSC469 11/22/2007

LFS Layout after update to file 1



- ix File inode before write
- fx File data before write
- im Inode map block
- Checkpoint region (fixed location)
- ix File inode after write
- fx File data after write

- New inode map generated with each write to disk



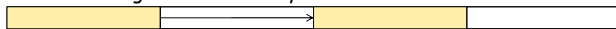
CSC469 11/22/2007

LFS: Free Space Management

- LFS append-only quickly runs out of disk space
 - Need to recover deleted blocks
- Approach:
 - Fragment log into segments, write segment-at-once



- Thread active 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
- High cleaning overhead is a big problem

CSC469



11/22/2007

LFS: Crash Recovery

- Traditional Unix FS: changes can be made anywhere
 - Must scan entire disk to restore metadata consistency after crash (fsck)
- LFS: most recent changes are at end of log
 - Much faster/easier to achieve a consistent state
- LFS borrows 2 database techniques for recovery after a crash:
 - *Checkpoints* define a consistent file system state
 - *Roll-forward* recovers information written since the last checkpoint

CSC469



11/22/2007

LFS Checkpoints

- Special, fixed location on disk containing:
 - Addresses of all inode map blocks
 - Segment usage table (for cleaning)
 - Address of last segment written
 - Timestamp
- Crash can occur while writing the checkpoint
 - Keep two checkpoint regions
 - Write timestamp out last
 - On recovery, use region with most recent timestamp

CSC469



11/22/2007

LFS Roll-Forward

- Can recover data written since last checkpoint
- Examine segments written since segment identified by checkpoint region
 - Segments contain special summary blocks and *directory operation logs* that record changes to file system metadata
 - Operation log entry is guaranteed to appear before modified directory or inode blocks
 - If directory or inode is inconsistent, operation log entry can be used to correct it
- Note ordering constraints on metadata updates

CSC469

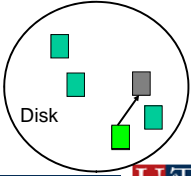



11/22/2007

Metadata update problem

- File system metadata
 - inodes, directory blocks, allocation bitmaps)
- Interdependencies must be handled carefully during disk updates
 - So that file system is consistent after crash


Cache
(main
memory)



CSC469

11/22/2007

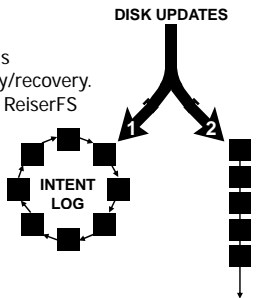
Solutions to Metadata Update

- Synchronous writes
 - Each update is completed at disk before next one is allowed
 - FFS
 - SLOW!
- Logged writes
 - Journaling file systems
- Ordered writes
 - Soft Updates


CSC469

11/22/2007

Write-ahead logging

Benefit of logging is strictly for reliability/recovery. Used by ext3, JFS, ReiserFS




This approach journals metadata updates to an intent log (1) before modifying the file itself (2).

CSC469

11/22/2007

Soft Updates

- Delayed metadata writes
 - No waiting for disk
- Protect consistency of disks by controlling order of writes
 - use write-back caching for all (non-fsync) updates
 - make sure updates propagate to disk in the correct order
 - works great, but only goes as far as update ordering can go

CSC469

11/22/2007

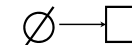
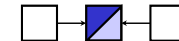
Update ordering as a tool

- Just what it sounds like...
- Good for single-direction dependencies
 - just do one before the other
- Problem: doesn't work for bidirectional dependencies
 - which, unfortunately, is most of them
- Solution: some can be converted to single-dir
 - because some directions are more important than others ;)
 - clean-up must be done after system failures



Basic Update Ordering Rules

- Purpose: integrity of metadata pointers
 - in face of unpredictable system failures
 - similar to rules of programming with pointers
- Resource Allocation
 - initialize resource before setting pointer
- Resource De-allocation
 - nullify previous pointer before reuse
- Resource "Movement"
 - set new pointer before nullifying old one
- Notice that something always left dangling
 - ... assuming a badly-timed crash



CSC469



11/22/2007

FS crash recovery

- Need to deal with all of the dangling stuff
 - worse: need to find it all first
- Traditional recovery examines entire contents
 - walk directory hierarchy and each file's block list
 - identify unclaimed resources and incorrect counts
 - rebuild free space/inode bitmaps
- Post-crash time to mount
 - Traditional: 5 to 7 minutes (several years ago)
 - grows with FS size and #files
- Soft Updates requires no post-crash recovery
 - Only inconsistencies are "leaked" blocks
 - Can find free blocks in background while using file system



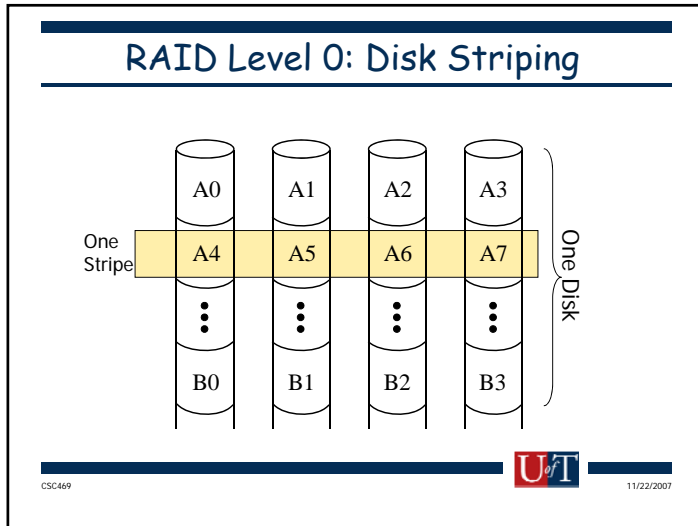
RAID

- Redundant Array of Inexpensive Disks (RAID)
 - A storage system, not a file system
 - Patterson, Katz, and Gibson (Berkeley, '88)
- Idea: Use many disks in parallel to increase storage bandwidth, improve reliability
 - Files are striped across disks
 - Each stripe portion is read/written in parallel
 - Bandwidth increases with more disks

CSC469



11/22/2007



Disk striping details

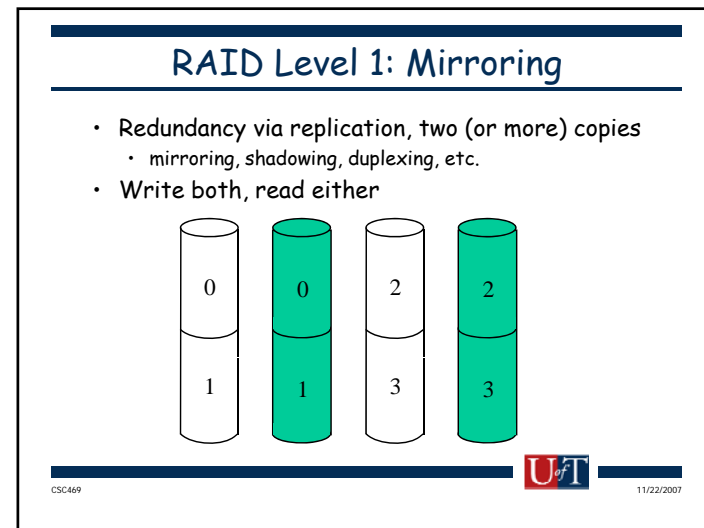
- How disk striping works
 - Break up total space into fixed-size **stripe units**
 - Distribute the stripe units among disks in round-robin fashion
 - Compute location of block #B as follows
 - $disk\# = B \% N$ ($\% = \text{modulo}$, $N = \# \text{ of disks}$)
 - $LBN\# = B / N$ (computes the LBN on given disk)
- Key design decision: picking the stripe unit size
 - too big: no parallel transfers and imperfect load balancing
 - too small: small transfers span stripe unit boundaries
 - also, should be a multiple of block size to assist alignment
- No redundancy

CSC469 11/22/2007

RAID 0 Challenges

- Small files (small writes less than a full stripe)
 - Need to read entire stripe, update with small write, then write entire stripe out to disks
- Reliability
 - More disks increases the chance of media failure (MTBF)
- Turn reliability problem into a feature
 - Use one disk to store parity data
 - XOR of all data blocks in stripe
 - Can recover any data block from all others + parity block
 - Hence "redundant" in name
 - Introduces overhead, but, hey, disks are "inexpensive"

CSC469 11/22/2007



Raid Level 2&3: Parity disks

- Both:
 - very small stripe unit (single byte or word)
 - All writes update parity disk(s)
 - Can correct single-bit errors
- Level 2:
 - #parity disks = $\log_2(\#data\ disks)$
 - overkill
- Level 3:
 - One extra disk

The diagram shows five vertical cylinders representing disks. The first four cylinders are labeled A, B, C, and D from top to bottom. The fifth cylinder is labeled Ap, Bp, Cp, and Dp from top to bottom. The Ap, Bp, Cp, and Dp sections of the fifth cylinder are highlighted in green.

11/22/2007

RAID Levels 4-6

- Levels 4-6 introduce *independence*:
 - Each disk may be writing different data
- Level 4 is similar to Level 3
- Level 5 removes parity disk bottleneck →
 - Distributes parity bits across all data disks
- Level 6 tolerates 2 errors

The diagram shows five vertical cylinders representing disks. The first four cylinders are labeled A, B, C, and D from top to bottom. The fifth cylinder is labeled Ap, Bp, Cp, and Dp from top to bottom. The Bp, Cp, and Dp sections of the fifth cylinder are highlighted in green.

11/22/2007