Architectures for Improved File System Performance

CS239
Advanced Operating Systems
Spring 1996

Outline

- Introduction
- The Log-Based File System
- Disk Array Based File Systems
  - Also an architecture for reliability

Introduction

- Why do we need to work on file system performance?

1. It's not that good yet
2. Patterns of access change
3. File servers, distributed file systems, and parallel file systems have different performance characteristics

What Can the OS Do?

- Improve locality on disk
- Maximize size of data transfers
- Minimize amount of disk access
  - Cache disk blocks
  - Buffer writes
- Fetch from multiple disks in parallel

Improving Locality on Disk

- Make sure next disk block you need is close to the last one you got
- File layout is important here
- Ordering of accesses in controller helps

– Effect - less seek time & rotational delay

Maximize Size of Data Transfer

- Transfer big blocks or multiple blocks in one read
- Tradeoff against wasted disk space
- Readahead is one good method here

– Effect - Increase effective disk bandwidth and reduce number of disk I/Os

Minimize Amount of Disk Access

- Avoid disk accesses when possible
- Use caching to hold file blocks in memory
- Buffer writes

– Effect - decreases latency by removing the relatively slow disk from the path

Fetch From Multiple Disks in Parallel

- Use more than one physical disk per system
- Cleverly lay out data so requests are somehow "spread" across disks

– Effect - higher delivered bandwidth

The Log-Based File System

Good, large buffer caches can catch almost all reads

But most writes have to go to disk

So file system performance can be limited by writes

So, produce FS that writes quickly

Like an append-only log

Basic LFS Architecture

Buffer writes, send them sequentially to disk

- Data blocks
- Attributes
- Directories
- And almost everything else

Converts small sync writes to large async writes

A Simple Log Disk Structure

File A Block 7	File Z Block 1	File M Block 202	File A Block 3	File F Block 1	File A Block 7	File L Block 26	File L Block 25

Head of Log →

Key Issues in Log-Based Architecture

1. Retrieving information from the log

   No matter how well you cache, sooner or later you have to read

2. Managing free space on the disk

   You need contiguous space to write - in the long run, how do you get more?

Finding Data in the Log

File A	File Z	File M	File A	File F	File A	File L	File L
Block 7	Block 1	Block 202	Block 3	Block 1	Block 7	Block 26	Block 25

Give me block 25 of file L
Or,
Give me block 1 of file F

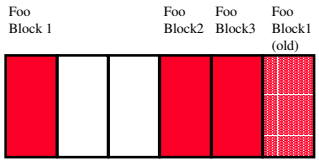
Retrieving Information From the Log

Must avoid sequential scans of disk to read files

Solution - store index structures in log

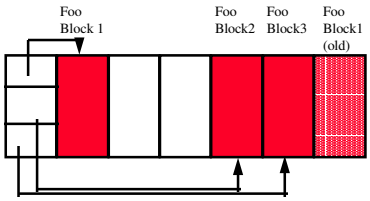
Index is essentially the most recent version of the inode

Finding Data in the Log



How do you find all blocks of file Foo?

Finding Data in the Log With an Inode



How Do You Find a File's Inode?

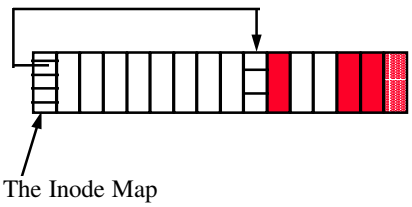
You could search sequentially

LFS optimizes by writing *inode maps* to the log

Inode map points to the most recent version of each inode

A file system's inodes cover multiple blocks of inode map

How Do You Find the Inode?



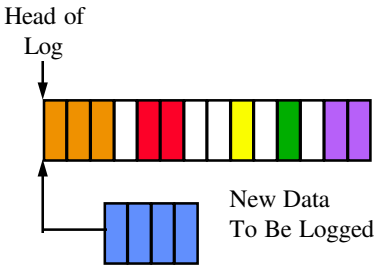
How Do You Find Inode Maps?

- Use a fixed region on the disk
  - That always points to the most recent inode map blocks
- But cache inode maps in main memory
- Small enough that few disk accesses required to find inode maps

Reclaiming Space in the Log

- Eventually, the log reaches the end of the disk partition
- So LFS must reuse disk space
  - Like superseded data blocks
- Space can be reclaimed in background
  - Or when needed
- Goal is to maintain large free extents on disk

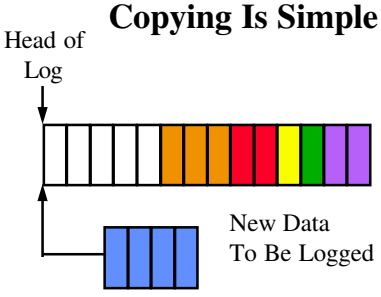
Example of Need for Reuse



Major Alternatives For Reusing Log

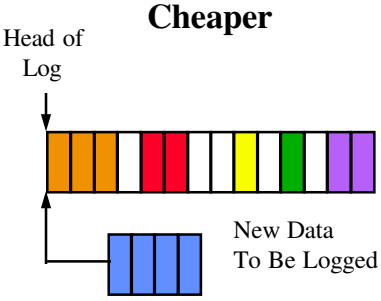
- Threading vs. copying
- If empty segments are not contiguous, must use threading
  - But that leads to fragmentation
- Copying avoids fragmentation
  - But it is expensive

Copying Is Simple

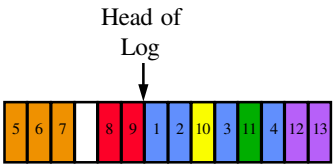


But required many writes to disk

Threading Is Complex, But Cheaper

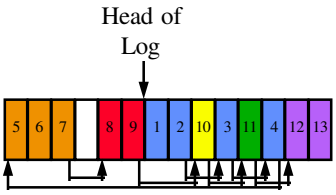


Threading the Log



Eventually, you want to compact

Threaded Log Showing Pointers



LFS Management Policy

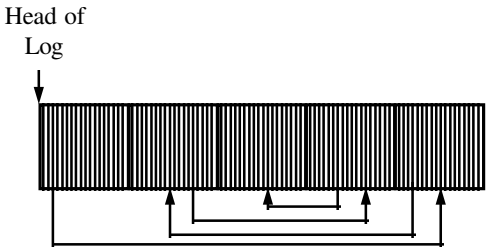
Combination of copying and threading

Copy to free large fixed-size *segments*

Thread free segments together

Try to collect long-lived data permanently into segments

A Threaded, Segmented Log



Cleaning a Segment

1. Read several segments into memory
2. Identify the live blocks
3. Write live data back

Hopefully into a smaller number of segments

Identifying Live Blocks

Clearly not feasible to track down live blocks of all files

Instead, each segment maintains a segment summary block

Identifying what is in each block

Crosscheck blocks with owning inode's block pointers

Written at end of log write, for low overhead

Segment Cleaning Policies

What are the important questions?

1. When do you clean segments?
2. How many segments to clean?
3. Which segments to clean?
4. How to group blocks in their new segments?

When To Clean

- Periodically
- Continuously
- During off-hours
- When disk is nearly full
- On demand

LFS uses a threshold system

How Many Segments To Clean

The more cleaned at once, the better the reorganization of the disk

But the higher the cost of cleaning

LFS cleans a few tens at a time

Till disk drops below threshold value

Empirically, LFS not very sensitive to this factor

Which Segments to Clean?

Cleaning segments with little dead data gives little benefit

Cleaning segments with lots of dead data gives great benefit

Want to arrange that most segments have good utilization

But cleaner works mostly with a few that don't

Actually a complicated question

Hot and Cold Segments

Some segments are hot, some are cold

Cold segments don't get chosen for cleaning

But "cold" free space is more valuable than "hot" free space

Since cold blocks tend to stay cold

Choosing Segments Through Cost-Benefit Analysis

u = utilization

A = age

Benefit to cost = u*A/(u+1)

Clean cold segments with some space, hot segments with a lot of space

What to Put Where?

Given a set of live blocks and some cleaned segments, which goes where?

– Order blocks by age

– Write them to segments oldest first

Goal is very cold, highly utilized segments

Performance of LFS

On modified Andrew benchmark, 20% faster than FFS

But this benchmark is CPU-intensive

Micro-benchmarks provide better performance vs. FFS

Except for sequentially reading a randomly written file

LFS Disk Bandwidth Utilization

LFS can create and delete 8 times as many files per second as FFS

LFS can read 1 1/2 times as many small files

Why does LFS perform better?

Logical Locality vs. Temporal Locality

Normal file systems keep a file's data blocks close together

Logical locality

LFS keeps data written at the same time close together

Temporal locality

When temporal locality == logical locality, systems perform the same

When Does LFS Work Well?

- When the cache is big
  - Big enough so most reads go to the cache
  - So maybe not in environments with huge files
- When files aren't written randomly and read sequentially

Increased File System Performance From Parallelism

One disk can only deliver data at its maximum rate

So to get more data faster, get it from multiple disks simultaneously

Saving on rotational latency and seek time

Simple Disk Parallelism

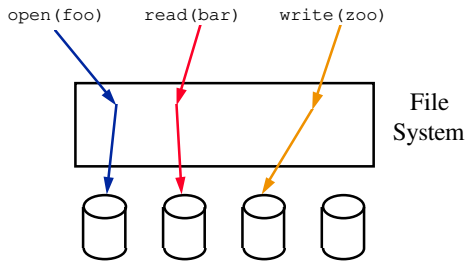
If you have multiple disks on the system, you automatically get some benefit

But only if you generate accesses to all of the disks

And in a parallelizable fashion

Not often the case

Example of Simple Disk Parallelism



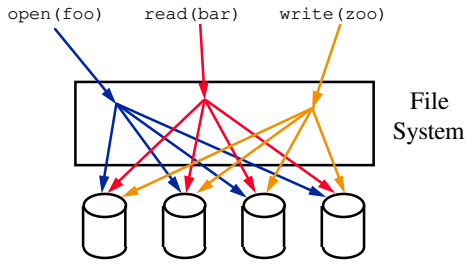
Improving Disk Access Parallelism

Instead of satisfying each access from one disk, use multiple disks for each access

Store part of each data block on several disks

So fetching any data block involves multiple disks

Improved Disk Parallelism Example



Data Striping

Transparently distributing data over multiple disks

Benefits -

- Increases disk parallelism
- Faster response for big requests

Major parameters are number of disks and size of data interleaf

Fine-Grained Vs. Course-Grained Data Interleaving

- Fine grain data interleaving
  +High data rate for all requests
  –But only one request per disk array
  –Lots of time spent positioning
- Course grain data interleaving
  +Large requests access many disks
  +Many small requests handled at once
  –Small I/O requests access few disks

Reliability of Disk Arrays

Without disk arrays, failure of one disk among N loses 1/Nth of the data

With disk arrays (fine grained across all N disks), failure of one disk loses all data

N disks 1/Nth as reliable as one disk

Adding Reliability To Disk Arrays

- Buy more reliable disks
- Build redundancy into the disk array
  - Multiple levels of disk array redundancy possible
  - Most organizations can prevent any data loss from single disk failure

Basic Reliability Mechanisms

- Duplicate data
- Parity for error detection
- Error Correcting Codes for detection and correction

Duplicating Data

- Store each data item more than once
  - Rarely stored more than twice
- Must be careful that the second copy is really separate
  - Not on same disk
  - Not using same controller

Parity Methods

- Can use parity to detect multiple errors
  - But typically used to detect single error
- If hardware errors are self-identifying, parity can also correct errors
- When data is written, parity must be written too

Error-Correcting Codes

- Based on Hamming codes, mostly
- Not only detect error, but identify which bit is wrong
- Superfluous if hardware signals which part has failed
  - But it doesn't always do that

RAID Architectures

Redundant **A**rrays of **I**ndependent **D**isks

Basic architectures for organizing disks into arrays

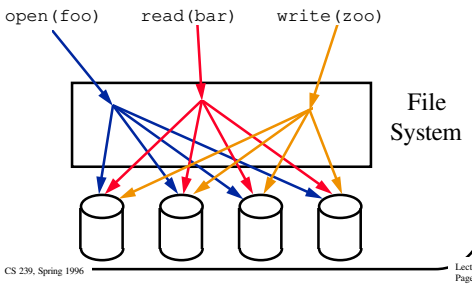
Assuming independent control of each disk

Standard classification scheme divides architectures into levels

Non-Redundant Disk Arrays (RAID Level 0)

- No redundancy at all
- So, what we just talked about
- Any failure causes data loss
- Strictly speaking, not redundant at all

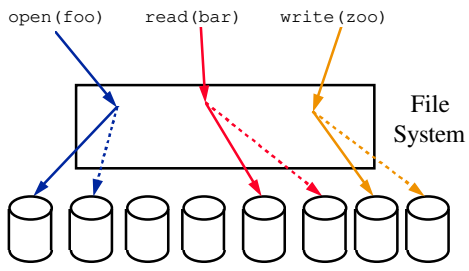
Non-Redundant Disk Array Diagram



Mirrored Disks (RAID Level 1)

- Each disk has second disk that mirrors its contents
  - – Writes go to both disks
  - – No data striping
- \+ Reliability is doubled
- \+ Read access faster
- – Write access slower
- – Expensive and inefficient

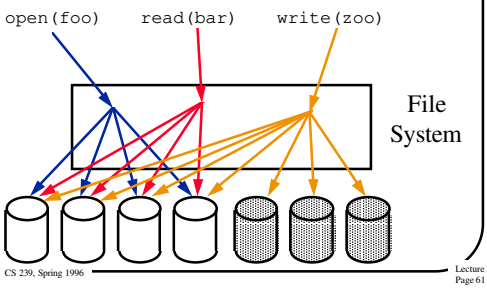
Mirrored Disk Diagram



Memory-Style ECC (RAID Level 2)

- Some disks in array are used to hold error-correcting codes
- E.g, 4 data disks require 3 ECC disks
- \+ More efficient than mirroring
- \+ Can correct, not just detect, errors
- – Still fairly inefficient

Memory-Style EEC Diagram



Bit-Interleaved Parity (RAID Level 3)

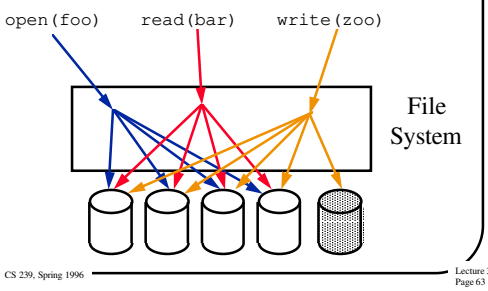
- Each disk stores one bit of each data block
- One disk in array stores parity for other disks

\+ More efficient than Levels 2 and 3

– Parity disk doesn't add bandwidth

– Can't correct errors

Bit-Interleaved RAID Diagram



Block-Interleaved Parity (RAID Level 4)

- Like bit-interleaved, but data is interleaved in blocks of arbitrary size
  - Size is called *striping unit*
  - Small read requests use 1 disk

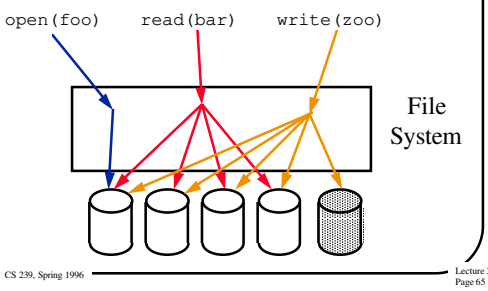
\+ More efficient data access than level 3

\+ Satisfies many small requests at once

– Parity disk can be a bottleneck

– Small writes require 4 I/Os

Block-Interleaved Parity Diagram



Block-Interleaved Distributed-Parity (RAID Level 5)

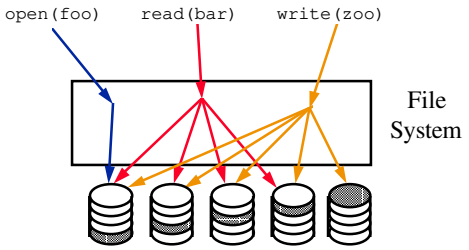
- Spread the parity out over all disks

\+ Eliminates parity disk bottleneck

\+ All disks contribute read bandwidth

– Still requires 4 I/Os for small writes

Block-Interleaved Distributed-Parity Diagram



P+Q Redundancy (RAID Level 6)

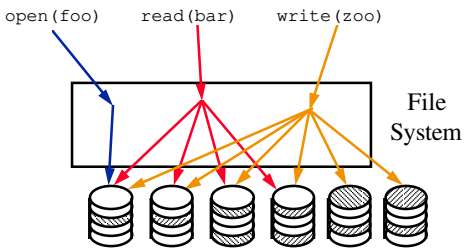
- Protects from up to two disk failures
  - Using Reed-Solomon codes
  - Similar organization to block-interleaved distributed-parity

+ Provides higher level of protection

– Small writes require 6 I/Os

– Somewhat less efficient than some of the other schemes

P+Q Redundancy Diagram



Comparing RAID Schemes

- Three primary metrics
  - Reliability
  - Performance
  - Cost
- But multiple ways of measuring each
- And multiple choices in trading them off

So Which RAID Do You Choose?

- Depends on what you're doing
- Different levels have different properties

Equivalence of Various RAID Levels

Some RAID levels are subsets of others

E.g., RAID level 1 is similar to RAID level 5 with parity group of 2

RAID level 3 also maps to level 5

RAID Level 5 is better than RAID levels 2 and 4 in almost all ways

So the problem is choosing right parameters for RAID level 5

RAID - Hardware vs. Software

RAID architectures can be built in either hardware or software

Some forms of hardware offer greater reliability than software can

Most RAIDs today built in hardware, with software support