Improving File Systems

CS 239 — Advanced Operating Systems

April 13, 1995

Outline

- The Log-Based File System
- Virtual File Systems
- Stackable File Systems

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

Therefore, produce something that writes quickly

Like an append-only log

Basic LFS Architecture

Buffer writes, then write them sequentially to disk

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

Converts small synchronous writes to large async writes

Key Issues in Log-based Architecture

1. Retrieving information from the log

2. Managing free space on the disk

Finding Data in the Log

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 With Index

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 most recent version of each inode

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

How Do You Find Inode Maps?

Use a fixed region on the disk

    That always points to most recent inode map blocks

But cache inode maps in main memory

Small enough so few disk access to find inode maps

Reclaiming Space in the Log

Eventually, log reaches end of disk partition

So LFS must reuse disk space

- Like superseded data blocks

Space can be reclaimed in the background

- Or when needed

Goal is to maintain large free extents on disk

Major Alternatives For Writing Log

Threading versus copying

If empty segments are not contiguous, must use threading

  But that leads to fragmentation

Copying avoids fragmentation

  But it is expensive

File Reuse Diagram

Threading for File Log

Compaction for File Log

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

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*

- Identifies what is in each block

Crosscheck blocks with owning inode's block pointers

Written at the end of log write, so overhead is low

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 doing it

LFS cleans a few tens at a time

Till disk drops below threshold value

Empirically, LFS not very sensitive to this

Which Segments to Clean?

Cleaning segments with little dead data gives little benefit

Cleaning segments with lots of dead data gives great benefit

So try to arrange that most segments have good utilization

But cleaner works mostly with the few that don't

Things are actually more complicated than that . . .

Cost-Benefit Analysis

Some segments are hot, some are cold

"Cold" free space is more valuable than "hot" free space

So multiply "age" of the segment by its utilization to get the benefit

And divide by utilization plus one to get the cost

What to Put Where?

Order blocks by age

Write them to segments oldest first

To get 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

Disk Bandwith Utilization

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

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

Logical Locality versus Temporal Locality

In normal file systems, a file's data is kept close together

Logical Locality

In LFS, data written or created at the same time is kept close together

Temporal Locality

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

When Does LFS Work Well?

When there's enough main memory to satisfy most reads from cache

So maybe not in environments with huge files

When files aren't written randomly and read sequentially

File System Extensibility

Any existing file system can be improved

No file system is perfect for all purposes

So the OS should make multiple file systems available

And should allow for future improvements to file systems

Virtual File Systems

Introduced by Sun

Common interface to multiple file system implementations

Goals of VFS

Split FS implementation dependent/independent functionality

Support semantics of important existing file systems

Usable by both servers and clients of remote file systems

Atomicity of operation

Good performance, re-entrant, no centralized resources, “OO” approach

Basic Architecture

Split the existing common Unix file system architecture

- Normal user file-related system calls above the split

- File system dependent implementation details below

Inodes fall below, open() and read() calls above

Architecture Diagram

VFS's and Vnodes

Virtual File Systems

Each VFS is linked into a list of VFS's

First in list is the root VFS

VFS has a pointer to its data

Generic operations used to access VFS's

Vnodes

The per-file data structure made available to applications

Has both public and private data areas

- Public area is static or manipulated only at VFS level

No locking done by the vnode layer

Vnode Example

Further Extensibility Needs

The VFS approach allows adding new file systems

But it isn't as helpful for improving existing file systems

What can be done to add functionality to existing file systems?

Stackable File System Layers

A method to stack one file system on top of another

Requires various changes to the vnode interface and mount commands

Serious performance questions need attention

Example of Stackable Layers

Extending the Interface

New file layers provide new functionality

Possibly requiring new vnode operations

So each layer must be prepared to deal with arbitrary, unknown operations

Bypass vnode operation

Handling a Vnode Operation

A layer can do three things with a vnode operation:

1. Do the operation and return
2. Pass it down to the next lower layer unaltered
3. Do some work and pass it to the next layer

The same choices are available as the result is returned up the stack

Mounting Stackable Layers

Each layer is mounted with a separate command

Essentially pushing the new layer on top of the stack

Mount command identifies—

- The underlying layer
- The new layer
- The mount point for the new layer

What Can You Do With Stackable Layers?

Leverage off existing file system technology, adding—

- Compression
- Encryption
- Recovery of deleted files
- File replication

Performance of Stackable Layers

To be a reasonable solution, per-layer overhead must be low

In UCLA implementation, overhead is apx. 1-2% per layer

  In system time, not elapsed time

Elapsed time overhead apx. .25% per layer

  Highly dependent on application, of course