File System Performance and Extensibility
CS239
Advanced Operating Systems
April 23, 1997

Outline

- File system performance
  - Serverless file systems (xFS)
- File system extensibility
- Non-disk file systems

Serverless File Systems

Take the Zebra idea and make a more distributed version of it

xFS, developed at Berkeley

Attacks all of Zebra's bottleneck problems

Also, unlike Zebra, not all participants are members of a single stripe group

Key xFS Concepts

- Metadata managers
- Stripe groups for data storage
- Cooperative caching
- Distributed cleaning processes

How Do I Locate a File in xFS?

I've got a file name, but where is it?

Assuming its not locally cached

File's directory converts name to a unique index number

Consult the metadata manager to find out where file with that index number is stored - the *manager map*

The Manager Map

Data structure that allows translation of index numbers to file managers

Not necessarily file locations

Kept by each metadata manager

Globally replicated data structure

Simply says what machine manages the file

What Does the Manager Do?

Manager keeps two types of information

1). imap information

2). caching information

If some other site has the file in its cache, tell requester to go to that site

Always use cache before disk

Even if cache is remote

What If No One Caches the Block?

Metadata manager for this file then must consult its *imap*

Imap tells which disks store the data block

Files are striped across disks stored on multiple machines

Typically single block is on one disk

Stripe Groups

Set of disks that cooperatively store data in RAID fashion

xFS uses single parity disk

Alternative to striping all data across all disks

Like Zebra, LFS methods used to batch small writes across a stripe group

Cooperative Caching

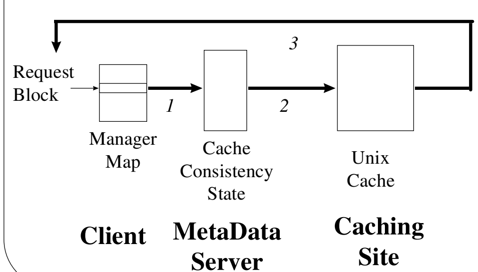
Each site's cache can service requests from all other sites

Working from assumption that network access is quicker than disk access

Metadata managers used to keep track of where data is cached

So remote cache access takes 3 network hops

Getting a Block From a Remote Cache



Providing Cache Consistency

Per-block token consistency

To write a block, client requests token from metadata server

Metadata server retrieves token from whoever has it

And invalidates other caches

Writing site keeps token

Which Sites Should Manage Which Files?

Could randomly assign equal number of file index groups to each site

Better if the site using a file also manages it

In particular, if most frequent writer manages it

xFS uses heuristic of locating management at client creating file

Can reduce network traffic by ~ 40%

Cleaning Up

File data (and metadata) is stored in log structures spread across machines

A distributed cleaning method is required

Each machine stores info on its usage of stripe groups

Each cleans up its own mess

Basic Performance Results

Early results from incomplete system

Can provide around 10 times the bandwidth of file data as single NFS server

Even better on creating small files

Doesn't compare xFS to multimachine servers

Where Did xFS Look For Performance Improvements?

- Parallel use of disks and file servers
  - Improve overall delivered bandwidth
- Small write performance improved through log structure
- Distributed file system services to lessen bottlenecks

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

Approaches to File System Extensibility

- Modify an existing file system
- Virtual file systems
- Layered and stackable file system layers

Modifying Existing File Systems

Make the changes you want to an already operating file system

+ Reuses code

– But changes everyone’s file system

– Requires access to source code

– Hard to distribute

Virtual File Systems

Permit a single OS installation to run multiple file systems

Using the same high-level interface to each

OS keeps track of which files are instantiated by which file system

Introduced by Sun

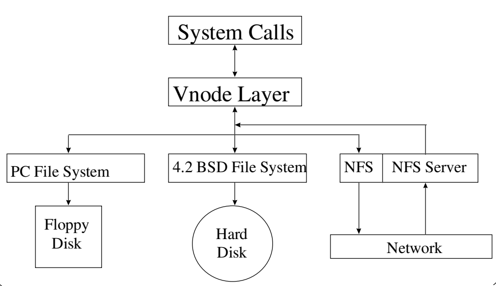
Goals of Virtual File Systems

- Split FS implementation-dependent and -independent functionality
- Support semantics of important existing file systems
- Usable by both clients and servers of remote file systems
- Atomicity of operation
- Good performance, re-entrant, no centralized resources, “OO” approach

Basic VFS 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

VFS Architecture Block Diagram



Virtual File Systems

Each VFS is linked into an OS-maintained list of VFS’s

First in list is the root VFS

Each VFS has a pointer to its data

Which describes how to find its files

Generic operations used to access VFS’s

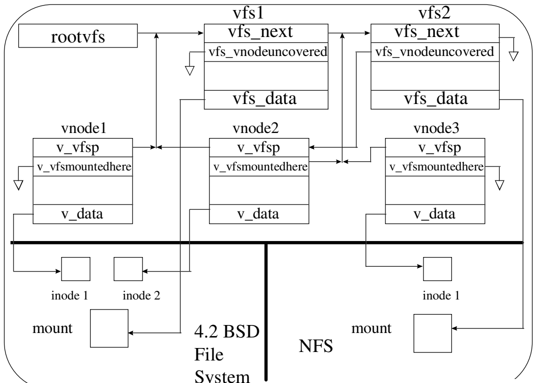
Vnodes

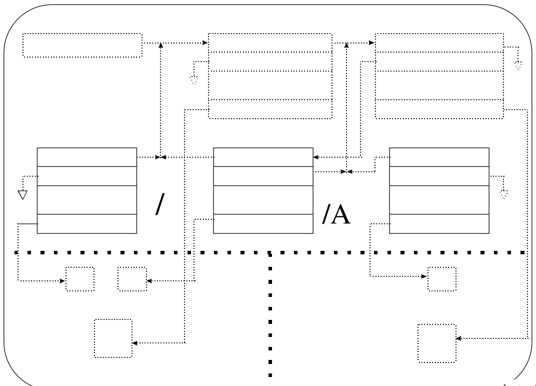
The per-file data structure made available to applications

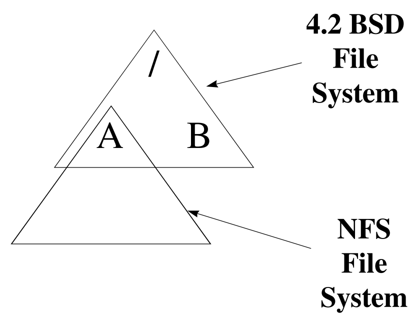
Has both public and private data areas

- Public area is static or maintained only at VFS level

No locking done by the vnode layer







Does the VFS Model Give Sufficient Extensibility?

The VFS approach allows us to add 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?

Layered and Stackable File System Layers

Increase functionality of file systems by permitting some form of composition

- One file system calls another, giving advantages of both

Requires strong common interfaces, for full generality

Layered File Systems

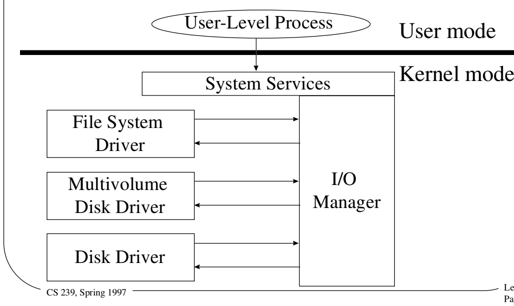
Windows NT provides one example of layered file systems

File systems in NT are the same as device drivers

Device drivers can call other device drivers

Using the same interface

Windows NT Layered Drivers Example



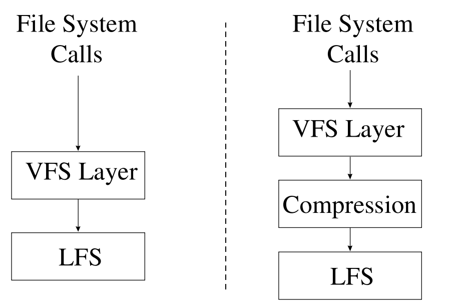
Another Approach - UCLA Stackable Layers

More explicitly built to handle file system extensibility

Layered drivers in Windows NT <u>allow</u> extensibility

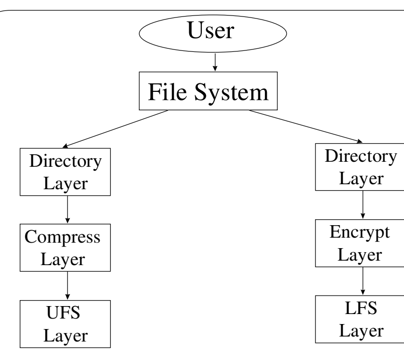
Stackable layers <u>supports</u> extensibility

Stackable Layers Example



How Do You Create a Stackable Layer?

- Write just the code that the new functionality requires
- Pass all other operations to lower levels (*bypass* operations)
- Reconfigure the system so the new layer is on top



What Changes Does Stackable Layers Require?

- Changes to vnode interface
  - For full value, must allow expansion to the interface
- Changes to mount commands
- Serious attention to performance issues

Extending the Interface

New file layers provide new functionality

Possibly requiring new vnode operations

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 layer
3. Do some work, then pass it down

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 new layer on stack

Can be performed at any normal mount time

Not just on system build or boot

What Can You Do With Stackable Layers?

Leverage off existing file system technology, adding

- Compression
- Encryption
- Object oriented operations
- File replication

All without altering any existing code

Performance of Stackable Layers

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

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

In system time, not elapsed time

Elapsed time overhead ~.25% per layer

Highly application dependent, of course

File Systems Using Other Storage Devices

All file systems discussed so far have been disk-based

The physics of disks has a strong effect on the design of the file systems

Different devices with different properties lead to different file systems

Other Types of File Systems

- In-core file systems
- Flash memory file systems
- Network/distributed file systems
  - already discussed

Fitting Various File Systems Into the OS

Something like VFS is very handy

Otherwise, need multiple file access interfaces for different file systems

With VFS, interface is the same and storage method is transparent

Stackable layers makes it even easier

Simply replace the lowest layer

In-Core File Systems

Store files in main memory, instead of disk

+ Fast access and high bandwidth
+ Usually simple to implement

– Hard to make persistent

– Often of limited size

– May compete with other memory needs

Where Are In-Core File Systems Useful?

- When brain-dead OS can't use all main memory for other purposes
- For temporary files
- For files requiring very high throughput

In-Core File System Architectures

- Dedicated memory architectures
- Pageable in-core file system architectures

Dedicated Memory Architectures

Set aside some segment of physical memory to hold the file system
- Usable only by the file system

Either it's small, or the file system must handle swapping to disk

RAM disks are typical examples

Pageable Architectures

Set aside some segment of virtual memory to hold the file system
- Share physical memory system

Can be much larger and simpler

More efficient use of resources

UNIX `/tmp` file systems are typical examples

A Typical Pageable Memory File System

Designed for Unix systems
- At UC Berkeley

Meant to service needs of programs requiring high throughput for temporary files

E.g., compilers

Basic Architecture of Pageable Memory File System

Uses VFS interface

Inherits most of code from standard disk FS
- Including caching code

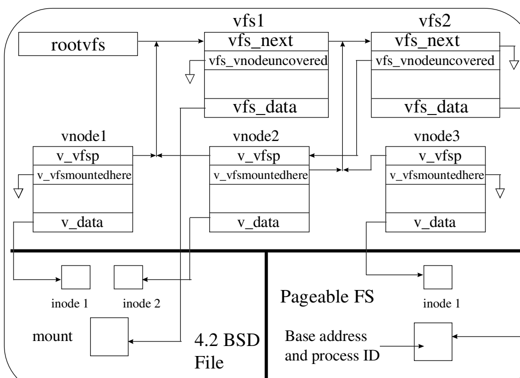
Major differences are calls that mount, access "disk," and unmount

Mounting in Pageable FS

VFS sets up a mount table entry
- Partly FS-independent
- Partly FS-specific

FS-specific part stores base address of memory used for this file system

Also stores process ID of `newfs` process



Mounting in Pageable FS, Continued

Finally, sets up initial empty list of I/O requests for this file system

When FS-specific data set up, instead of exiting, `newfs` sleeps

Sleep waits on requests to access physical “disk”

Accessing the “Disk” in the Pageable Memory FS

Read and write requests not satisfied by file system cache are sent to a “strategy” process

Strategy process for disk FS knows how to access physical disk

Strategy process for memory FS puts request on FS I/O queue and wakes sleeping mount process

When the Sleeper Awakes . . .

Mount process examines its queue of I/O requests

If non-empty, deals with first by determining offset into its memory space

- Then copies to the kernel buffer associated with the call

Unmounting in Pageable Memory File System

Normal file system unmounting code called, first

Unmounting code specific to memory FS flushes I/O queue

How Well Does This Perform?

Not as well as you might think

- Around 2 times disk based FS

Why?

How Well Does This Perform?

Not as well as you might think

- Around 2 times disk based FS

Why?

Because any access requires two memory copies

1. From FS area to kernel buffer
2. From kernel buffer to user space

Fixable if VM can swap buffers around

Other Reasons Performance Isn't Better

Disk file system makes substantial use of caching

Which is already just as fast

But speedup for file creation/deletion is faster

Since that requires multiple trips to disk

Flash Memory File Systems

- What is flash memory?
- Why is it useful for file systems?
- A sample design of a flash memory file system

Flash Memory

A form of solid-state memory similar to ROM

Holds data without power supply

Reads are fast

Can be written once, more slowly

Can be erased, but very slowly

Limited number of erase cycles before degradation

Writing In Flash Memory

If writing to empty location, just write

If writing to previously written location, erase it, then write

Typically, flash memories allow erasure only of an entire sector

Can read (sometimes write) other sectors during an erase

Typical Flash Memory Characteristics

Read cycle	80-150 ns
Write cycle	10 µs/byte
Erase cycle Cycle limit Sector size	1 s/block 100,000 times 64 Kbytes
Power Consumption	30-50 mA active 20-100 mA standby
Price	~ $15/Mbyte

Advantages and Disadvantages of Flash Memory

+ Small, and light
+ Uses less power than disk
+ Read time comparable to main memory access time
+ No rotation/seek complexities
+ No moving parts (shock resistant)
– Expensive (disk costs 15 cents/Mbyte)
– Erase cycle very slow
– Limited number of erase cycles

Flash Memory File System Architectures

- One basic decision to make
  - Is flash memory disk-like?
  - Or memory-like?

Should flash memory be treated as a separate device, or as a special part of addressible memory?

Hitachi Flash Memory File System

Treats flash memory as device

As opposed to directly addressable memory

Basic architecture similar to log file system

Basic Flash Memory FS Architecture

- Writes are appended to tail of sequential data structure
- Translation tables to find blocks later
- Cleaning process to repair fragmentation
- This architecture does no wear-leveling

Flash Memory Banks and Segments

- Architecture divides entire flash memory into banks (8, in current implementation)
- Banks are subdivided into segments
  - 8 segments per bank, currently
- 256Kbytes per segment
- 16Mbyte total capacity

Writing Data in Flash Memory File System

One bank is currently active

New data is written to block in active bank

When this bank is full, move on to bank with most free segments

Various data structures maintain illusion of “contiguous” memory

Cleaning Up Data

Cleaning is done on a segment basis

When a segment is to be cleaned, its entire bank is put on a cleaning list

No more writes to bank till cleaning is done

Segments chosen in manner similar to LFS

Cleaning a Segment

- Copy live data to another segment
- Erase entire segment
  - segment is erasure granularity
- Return bank to active bank list

Performance of the Prototype Flash Memory System

No seek time, so sequential/random access should be equally fast

Around 650-700 Kbytes per second

Read performance goes at this speed

Write performance slowed by cleaning

How much depends on how full the file system is

Also, writing is simply slower in flash

More Flash Memory File System Performance Data

On Andrew Benchmark, performs comparably to pageable memory FS

Even when flash memory nearly full

This benchmark does lots of reads, few writes

Allowing flash file system to perform lots of cleaning without delaying writes