Advanced File Systems Issues

CS 239 — Advanced Operating Systems

April 6, 1995

Outline

- File system basics
- Improving file system performance
- Case study—Unix Fast File System
- Log structured file systems
- Extensibility and file systems

File System Basics

A file system is an organized collection of files

An operating system may support multiple file systems

- Instances of the same type
- Different types of file systems

File Systems and File Hierarchies

Typically, a single OS supports multiple file systems

All file systems typically bound into a single namespace

- Often hierarchically

Advantages of Independent File Systems

- Easier support for multiple hardware devices
- More control over disk usage
- Fault isolation
- Quicker to run consistency checks
- Support for multiple types of file system

Overall Hierarchical Organizations

Constrained—Independent file systems only located at particular places

Typically at the highest level in the hierarchy

DOS/Windows and Macintosh are examples

Unconstrained—Independent file systems can be put anywhere in hierarchy

Unix file systems are examples

File Hierarchy Composed of File Systems

Mounting

Each file system is a tree structure with a single root

Its root is spliced into the overall tree

- Typically on top of another file
- The file system is *mounted* on a *mount point*
- Special complexities in traversing mount points

What is a file?

A collection of data, plus

A collection of meta-data

Often called *attributes*

Usually in persistent storage

In Unix, a file is represented by an *inode*

Typical File Diagram

Attributes

Typical attributes include:

- File length
- File ownership
- File type
- Access permissions

Extended Attributes

Some systems store more information with attributes

E.g., the Macintosh OS

Some of it can be very large

In such cases, attributes need to be treated similar to file data

File With Data Pointers

Laying Out Files on Disks

Consider a long sequential file

And a disk divided into sectors

In turn divided into 1K blocks

Where should you put the bytes of the file?

File Layout Methods

Put attributes somewhere in one piece

Unless FS supports large, variable sized attributes

Divide file data into many small, fixed-size extents

Divide file data into few large, variable-size extents

Small-Sized Extent Method

Small-Sized Extent Method—Pros and Cons

Advantages—

- Little fragmentation
- Easier allocation methods

Disadvantages—

- High overhead for certain operations
- Potential for poor usage of disk bandwidth

Used in Unix systems

Large-Sized Extent Method

Large-Sized Extent Method—Pros and Cons

Advantages—

- Low overhead for common operations
- Easier to effectively deliver raw disk bandwidth

Disadvantages—

- Potentially serious fragmentation problems
- Harder to manage free storage and predict file needs

Used in IBM's OS/MVS

File System Performance Issues

File system performance limited by—

- Disk seek, rotational latency, and access time
- Disk bandwidth

Basic Methods to Improve File System Performance

- Increase locality

  To minimize seek latency

- Improve data layout

  To make larger transfers possible

Typical Disk Overheads

9-15 msec seek time

Apx. 8 msec rotational delay

Apx. .5 msec to transfer 1000 byte block off disk

Apx. 2.5 Mbyte per second bandwidth

So 18-24 msec to get 1000 bytes at random from a disk

Prefetching, caching, ordering seeks, can improve time

Poor Bandwidth Usage Example

Using disk figures above

Getting to the start of a sector takes 20 msec

If the sector was 1K long, it takes apx. .5 msec to read

Total time to read 1000 bytes—apx. 21 msec

Effective bandwidth is only 50,000 bytes/second

Unix Fast File System utilizes only 5-10% of disk bandwidth

What Can the OS Do?

Improve locality on disk

Maximize size of data transfers

Cache disk blocks

Improving Locality

Try to put related blocks close together on disk

Tricky when new data is written

Also tricky when file system is nearly full

Maximizing Data Transfers

Again, keep blocks contiguous

Both for reading and writing

Caching

OS controls I/O buffer cache

Typically with LRU methods

Generally used for all I/O, not just disk

What size?

Varies, but 10% of total system memory is common

Asynchronous vs. Synchronous Writes

If writes are buffered, they occur asynchronously

Potential cache consistency problems

Potential problems in the face of crashes

Some systems make certain critical writes synchronously

Parameters for Designing the Buffer Cache

- Most files are short (but long files can be very long)

- User access is bursty

- File accesses are largely sequential

  70-90% of all accesses

- Most files aren't open long

  75% for less than 1/4 second

- File lifetimes are short

  65-80% live less than 30 seconds

What Does This Imply About Buffer Caches?

Readahead is good (when access is sequential)

If files don't live long, buffering writes might be good

- But danger of losing data in case of a crash
- And most short-lived files are also short in length
  - 1/4 of all bytes deleted/overwritten in 30 seconds

Dynamically changing the size of the cache may help

A Case Study—The Unix Fast File System

Originally, Unix's file system used 512 byte blocks

- Eventually changed to 1024 byte blocks
- Buffered by kernel and synchronously written to disk

Poor throughput—4% of raw disk bandwidth

Not smart about locating data on the disk

Unix Fast File System fixed these (and other) problems

Important Changes in Fast File System

- Block size changes

- Duplication of superblock for reliability

- Changes in disk layout strategy

- New features

Blocks in FFS

Block size increased to 4096 bytes (minimum)

Block fragments addressable for very short files

Variable sized blocks (on per-file system basis)

Only doubly indirect blocks used

Duplication of Superblock

What is a superblock?

A structure describing the entire file system

Corruption of the superblock is thus a major disaster

Must ensure common hardware failures don't corrupt all copies

Changes in Disk Layout Strategy

Where to put the blocks of a new file?

Preferably close together

Preferably close to related files

But must not cluster everything together

Cylinder Groups

A *cylinder group* is a set of consecutive disk cylinders

FFS file systems are divided into cylinder groups

Each cylinder group has copy of

- superblock
- set of inodes
- bookkeeping info

Where to Put a New File?

FFS tries to keep files in the same directory together

In the same cylinder group

Within cylinder group, tries to keep “contiguous”

But must not let a cylinder group fill up

Where to Put a New Directory?

Put new directory in relatively empty cylinder group

Many free inodes, few directories already there

To avoid clustering of all files in one cylinder group

The Importance of Free Space

File systems must not run too close to capacity

Layout policies ineffective when too few free blocks

Typically, FFS needs 10% of the total blocks free

New Features

- Long file names
- File locking
- Symbolic links
- Atomic rename system call
- Usage quotas

Performance of FFS

Four to 15 times the bandwidth of old Unix file system

Depending on size of disk blocks

Performance on original system limited by CPU speed

- Due to memory-to-memory buffer copies

Unix FFS is Not the Ultimate Solution

Based on technology of the day

And file usage patterns of the day

In modern systems, only achieves 5-10% of raw bandwidth