Advanced File Systems Issues

CS239
Advanced Operating Systems
Spring 1996

Outline

- File system basics
- Making file systems faster
  - Case study - Unix Fast File System
  - Case study - Log Based File System
  - Architectures for improved performance
- Making file systems more reliable
- Making file systems do more
- Using other forms of persistent storage

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

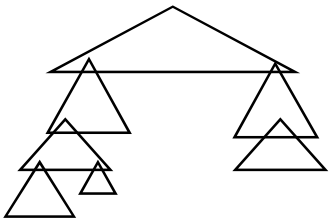
File Systems and File Hierarchies

Typically, a single OS supports multiple file systems

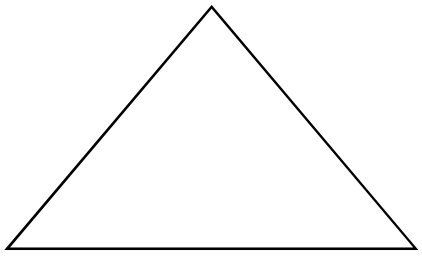
But all file systems are typically bound into a single namespace

Often hierarchical

A Hierarchy of File Systems



Why Not a Single File System?



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 systems

Overall Hierarchical Organizations

- Constrained
- Unconstrained

Constrained Organizations

Independent file systems only located at particular places

Usually at highest level in the hierarchy

E.g., DOS/Windows and Macintosh

Advantage - simplicity, simple user model

Disadvantage - lack of flexibility

Unconstrained Organizations

Independent file systems can be put anywhere in the hierarchy

Unix file systems are examples

Advantages - generality, invisible to user

Disadvantages - complexity, not always what user expects

These organizations require *mounting*

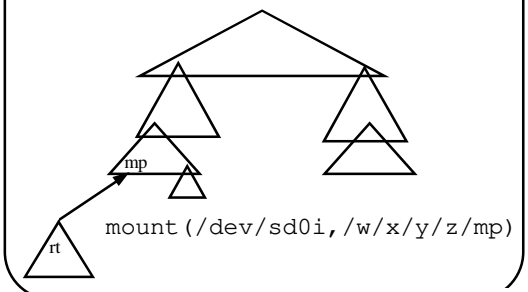
Mounting File Systems

Each FS is a tree structure with a single root

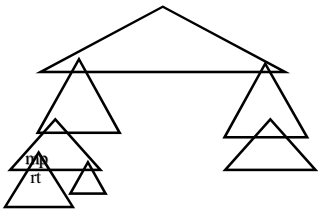
Its root is spliced into the overall tree

- Typically on top of another file
  - The *mount point*
- Complexities in traversing mount points

Mounting Example



After the Mount



What Do I See Before and After the Mount?

Before mounting, if you issue

```
ls /w/x/y/z/mt
```

you see the contents of /w/x/y/z/mt

After mounting, if you issue

```
ls /w/x/y/z/mt
```

you see the contents of rt

What Is a File?

A collection of data, plus

A collection of meta-data

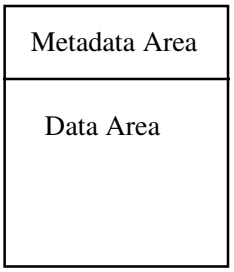
- Often called *attributes*

Usually in persistent storage

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

- Other systems have similar data structures

Layout of an Inode



File Attributes

Typical attributes include:

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

Typically stored in special fixed-size area

Extended Attributes

Some systems store more information with attributes

- E.g., the Macintosh OS
- Sometimes user-defined attributes

Some such data can be very large

- In such cases, treat attributes similar to file data

Storing File Data

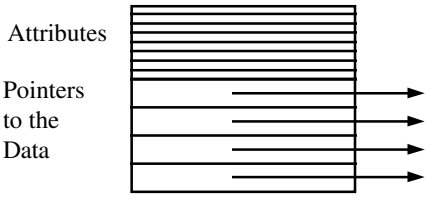
Where do you store the data?

Next to the attributes, or elsewhere?

Usually elsewhere

- Data is not of single size
- Data is changeable, including its size
- Storing elsewhere allows more flexibility in handling device

More Detailed Inode Diagram



Directories

A directory is a special type of file

Instead of normal data, it contains "pointers" to other files

Directories are hooked together to create the hierarchical namespace

Directory Diagram

File1	715
File2	393
File3	26
File4	1266
File5	7098
File6	690

First field specifies file name
Second field is the pointer to the file
In Unix, an inode number

Links

An alternate way to get to a file

Instead of through its standard name

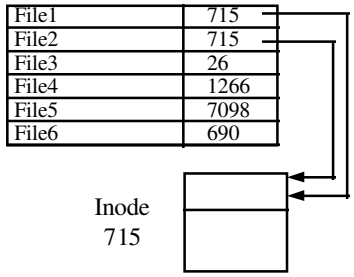
Very useful to effectively provide multiple different names for the same file

Hard Links Vs. Symbolic Links

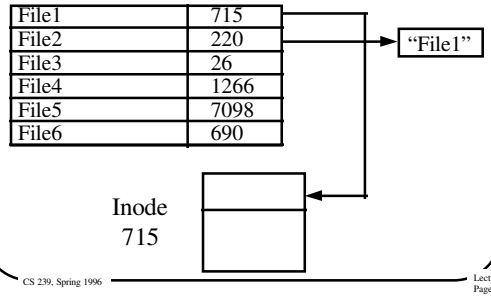
A *hard link* is simply a second name that points to the same file

A *symbolic link* is a special file that directs file name translation to take another path

Hard Link Diagram



Symbolic Link Diagram



Implications of Hard Links

- Multiple indistinguishable pathnames for the same file
- Need to keep link count with file for garbage collection
- "Remove" sometimes only removes a name
- Rather odd and unexpected semantics

Implications of Symbolic Links

- If file at the other end of the link is removed, dangling link
- But only one true pathname per file
- Effectively, just a mechanism to re-direct pathname translation
- Less system complications

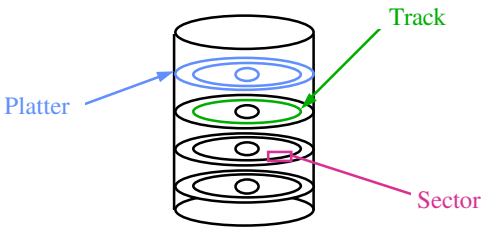
A Brief Description of Disk Hardware

Hard disk drives consist of one or more rotating *platters*

Each platter is divided into circular *tracks*

Each track is divided into *sectors*

A Disk Drive Diagram



Cylinders and Heads

Disks can have more than one platter

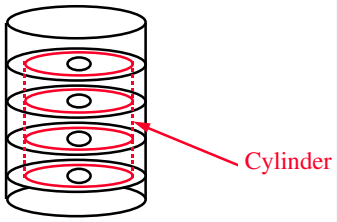
In which case, corresponding tracks on different platters are a *cylinder*

Typically, one *head* per platter

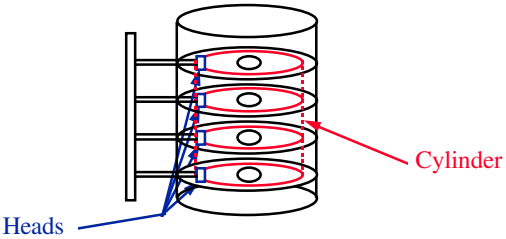
All heads usually move together

Usually, only one head can be used at once

Disk Drive With Cylinder



Disk Drive With Cylinder and Heads



Hard Disk Reliability

- Mean time to failure for a modern disk is on order of 10's to 100's of years
- But if you have 500 disks in your installation, several fail every year
- Backups are required
  - But are not always sufficient

Laying Out Files on Disks

Consider a long sequential file

And a disk divided into sectors

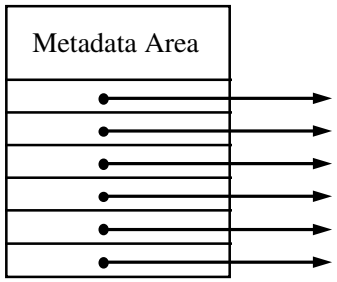
In turn divided into 1K blocks

Where should you put the bytes of the file?

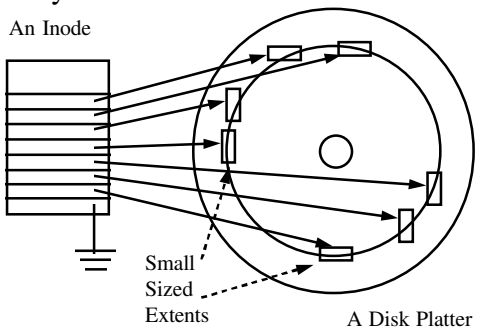
File Layout Methods

1. Put attributes somewhere in one piece
   Unless FS supports large, variable sized attributes
2. Put data in contiguous area
3. Divide file data into many small, fixed-sized extents
4. Divide file data into a few large, variable-sized extents

Layout of an Inode With Extents



Layout of Files With Small Extents

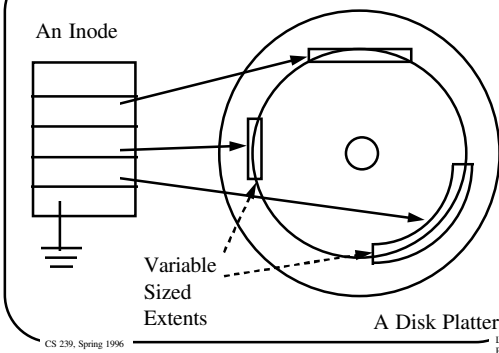


Small-Sized Extent Method - Pros and Cons

+ Little fragmentation
+ Easier allocation methods
– High overhead for certain operations
– Potential for poor usage of disk bandwidth

Used in Unix systems

Layout of Files With Variable Extents



Variable-Sized Extent Method - Pros and Cons

+ Low overhead for common operations
+ Easier to effectively deliver raw disk bandwidth
– Potentially serious fragmentation
– Harder to manage free storage and predict file needs

Used in IBM's OS/MVS

Inodes and File Size

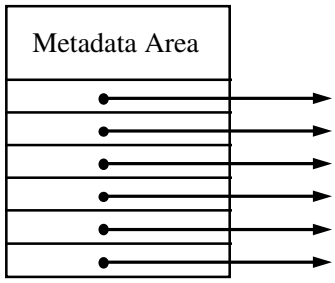
Inodes are typically fixed size

So there are a fixed number of extent pointers

Which limits the size of a file

Especially with fixed extents

Layout of an Inode With Extents



Indirect Blocks in Inodes

A method of making files potentially bigger

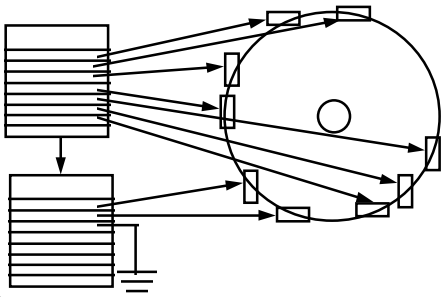
Use last pointer in inode to point to another block of pointers

Typically another inode

Can be done again for even bigger files

Unix originally had triple indirection

Inode With Indirect Block



Pros and Cons of Indirect Block

+ Allows larger files

Even with fixed size extents

– Some blocks require extra disk access to find

blocks may require two or three extra accesses if doubly or triply indirect

File System Performance Issues

Disk-based file system performance limited by -

- Disk seek
- Rotational latency
- Disk access time
- Disk bandwidth

File System Performance Parameters

- Read latency
- Write latency
- Bandwidth delivered

Basic File System Performance Improvement Methods

- Increase locality
  To minimize seek latency
  Improves latency times
- Improve data layout
  To make larger transfers possible
  Improves bandwidth (and sometimes latency)
- Satisfy requests from main memory

Typical Disk Overheads

9-15 msec seek time

Apx. 8 msec rotational delay

Apx. .25 msec to transfer 1000 byte block

Apx. 4.5 Mbyte per second BW

So 17-23 msec to get 1000 bytes at random from disk

Poor Bandwidth Usage Example

Using disk figures from previous slide

Getting to start of sector takes ~ 20 msec

If sector 1K long, ~ .25 msec to read

Total time to read 1000 bytes - ~21 msec

Effective BW is only 50,000 bytes/sec

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

How Are Disks Improving?

- Density measurements improve 10-25% per year
- Capacity improves 25% per year
- Transfer rate improves 20% per year
- Seek time improves 8% per year
- Rotational latency improves 5-8% per year

All slower than processor speed increases

The Disk/Processor Gap

Since processor speeds double every two to three years

And disk seek times double every ten years

Processors are waiting longer and longer for data from disk

Important for OS to cover this gap

What Can the OS Do?

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

Minimizing Disk Access

- Cache disk blocks in main memory
- Read ahead
- Buffer writes

Caching

OS controls I/O buffer cache

Typically with LRU methods

Generally used for all I/O, not just disk

What size should the buffer cache be?

Varies, but 10% of total system memory is common

Buffer Cache Design Factors

- 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?

- Design for holding small files
- Read-ahead is good
  - When access is sequential
- Dynamically changing the size of the cache might help

Read-Ahead

OS attempts to anticipate disk needs of programs

By reading blocks they are likely to request later

During time disk would otherwise be idle

Pros and Cons of Read-Ahead

+ Very good for sequential access of large files
  + E.g., executables
+ Allows immediate satisfaction of disk requests
– Reduces number of LRU blocks in cache
– Extra OS complexity

Buffering Writes

Buffer writes so that they needn't be performed at once

Reducing latency on writes

But buffered writes are asynchronous

Potential cache consistency and crash problems

Some systems make certain critical writes synchronously

Should We Buffer Writes?

- If files don't live long, buffered writes might be good
  - But danger of losing data in face of crashes
  - And most short-lived files are also short in length
  - 1/4 of all bytes deleted/overwritten in 30 sec.

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

Increase size of extent

But wasteful if many files are small

Fetching From Multiple Disks in Parallel

- Multiprogramming can cause some of this automatically
- Use of disk arrays can parallelize even a single process' access
  - At the cost of extra complexity

A Case Study - The Unix Fast File System

Originally, Unix's file system used 512 byte blocks (later 1024 bytes)

Buffered by kernel

Written synchronously to disk

Poor throughput - 4% of raw disk BW

Not smart about locating data on the disk

The Unix Fast File System fixed these (and other) problems

Important Changes in Unix 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

At minimum

Block fragments addressable for very short files

Variable sized blocks

On per-file system basis

Only doubly indirect blocks used

Effects of Block Size Changes in FFS

- Better throughput for medium-to-large files
- Avoids some fragmentation
- Tunable for special purpose file systems
- More complexity in file location algorithms

Duplication of the Superblock

What is a superblock?

A structure describing the entire file system

Corruption of the superblock is thus a major disaster

Duplication makes that less likely

But synchronization required

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 information

Where To Put a New File?

FFS tries to keep files in the same directory together on disk

I.e., 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

What is "empty"?

Many free inodes

Few directories already there

Avoids clustering all files in one cylinder group

The Importance of Free Space

File systems must not run too close to disk capacity

No room for new files

Layout policies ineffective when too few free blocks

Typically, FFS needs 10% of the total blocks free to perform well

New Features in FFS

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

Performance of FFS

4 to 15 times the bandwidth of old Unix file system

Depending on size of disk blocks

Performance on original file system limited by CPU speed

Due to memory-to-memory buffer copies

Unix FFS is Not the Ultimate Solution

Based on technology of the early 80s

And file usage patterns of those times

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