CS 537
Lecture 14
Unix FS Internals

Michael Swift

The original Unix file system

- Dennis Ritchie and Ken Thompson, Bell Labs, 1969
- "UNIX rose from the ashes of a multi-organizational effort in the early 1960s to develop a dependable timesharing operating system" -- Multics
- Designed for a "workgroup" sharing a single system
- Did its job exceedingly well
  - Although it has been stretched in many directions and made ugly in the process
- A wonderful study in engineering tradeoffs

All Unix disks are divided into five parts …

- Boot block
  - can boot the system by loading from this block
- Superblock
  - specifies boundaries of next 3 areas, and contains head of freelists of inodes and file blocks
- i-node area
  - contains descriptors (i-nodes) for each file on the disk; all i-nodes are the same size; head of freelist is in the superblock
- File contents area
  - fixed-size blocks; head of freelist is in the superblock
- Swap area
  - holds processes that have been swapped out of memory

So …

- You can attach a disk to a dead system …
- Boot it up …
- Find, create, and modify files …
  - because the superblock is at a fixed place, and it tells you where the i-node area and file contents area are
  - by convention, the second i-node is the root directory of the volume

i-node format

- User number
- Group number
- Protection bits
- Times (file last read, file last written, inode last written)
- File code: specifies if the i-node represents a directory, an ordinary user file, or a "special file" (typically an I/O device)
- Size: length of file in bytes
- Block list: locates contents of file (in the file contents area)
  - more on this soon!
- Link count: number of directories referencing this i-node

The flat (i-node) file system

- Each file is known by a number, which is the number of the i-node
  - seriously – 1, 2, 3, etc.!
  - why is it called "flat"?
- Files are created empty, and grow when extended through writes

The tree (directory, hierarchical) file system

- A directory is a flat file of fixed-size entries
- Each entry consists of an i-node number and a file name

i-node number	File name
152	.
18	..
216	my_file
4	another_file
93	oh_my_god
144	a_directory

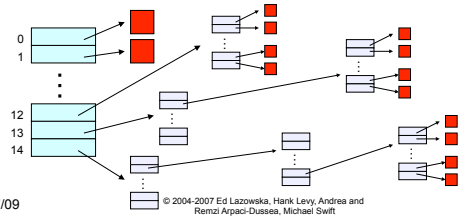
- It's as simple as that!

Using directories

- How do you find files?
  - Read the directory, search for the name you want (checking for wildcards)
- How do you list files (ls)
  - Read directory contents, print name field
- How do you list file attributes (ls -l)
  - Read directory contents, open inodes, print name + attributes
- How do you sort the output (ls -S, ls -t)
  - The FS doesn't do it -- ls does it!

The “block list” portion of the i-node

- Clearly it points to blocks in the file contents area
- Must be able to represent very small and very large files. How?
- Each inode contains 15 block pointers
  - first 12 are direct blocks (i.e., 4KB blocks of file data)
  - then, single, double, and triple indirect indexes



So …

- Only occupies 15 x 4B in the i-node
- Can get to 12 x 4KB = a 48KB file directly
  - (12 direct pointers, blocks in the file contents area are 4KB)
- Can get to 1024 x 4KB = an additional 4MB with a single indirect reference
  - (the 13th pointer in the i-node gets you to a 4KB block in the file contents area that contains 1K 4B pointers to blocks holding file data)
- Can get to 1024 x 1024 x 4KB = an additional 4GB with a double indirect reference
  - (the 14th pointer in the i-node gets you to a 4KB block in the file contents area that contains 1K 4B pointers to 4KB blocks in the file contents area that contian 1K 4B pointers to blocks holding file data)
- Maximum file size is 4TB

File system consistency

- Both i-nodes and file blocks are cached in memory
- The “sync” command forces memory-resident disk information to be written to disk
  - system does a sync every few seconds
- A crash or power failure between sync’s can leave an inconsistent disk
- You could reduce the frequency of problems by reducing caching, but performance would suffer big-time

What could go wrong?

- Creating a file
  - Allocate an inode (remove from list/bitmap)
  - write inode
  - Write file name to directory data
  - write size to directory inode
- What if you crash:
  - after writing inode but before writing the free inode list?
  - after writing the directory inode size before writing directory inode data?

i-check: consistency of the flat file system

- Is each block on exactly one list?
  - create a bit vector with as many entries as there are blocks
  - follow the free list and each i-node block list
  - when a block is encountered, examine its bit
    - If the bit was 0, set it to 1
    - if the bit was already 1
      - if the block is both in a file and on the free list, remove it from the free list and cross your fingers
      - if the block is in two files, call support!
  - if there are any 0's left at the end, put those blocks on the free list

d-check: consistency of the directory file system

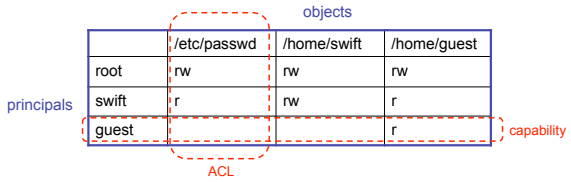
- Do the directories form a tree?
- Does the link count of each file equal the number of directories links to it?
  - I will spare you the details
    - uses a zero-initialized vector of counters, one per i-node
    - walk the tree, then visit every i-node

Protection systems

- FS must implement some kind of protection system
  - to control who can access a file (user)
  - to control how they can access it (e.g., read, write, or exec)
- More generally:
  - generalize files to objects (the "what")
  - generalize users to principals (the "who", user or program)
  - generalize read/write to actions (the "how", or operations)
- A protection system dictates whether a given action performed by a given principal on a given object should be allowed
  - e.g., you can read or write your files, but others cannot
  - e.g., your can read `/etc/motd` but you cannot write to it

Model for representing protection

- Two different ways of thinking about it:
  - access control lists (ACLs)
    - for each object, keep list of principals and principals' allowed actions
    - Like a guest list (check identity of caller on each access)
  - capabilities
    - for each principal, keep list of objects and principal's allowed actions
    - Like a key (something you present to open a door)
- Both can be represented with the following matrix:



ACLs vs. Capabilities

- Capabilities are easy to transfer
  - they are like keys: can hand them off
  - they make sharing easy
- ACLs are easier to manage
  - object-centric, easy to grant and revoke
    - to revoke capability, need to keep track of principals that have it
    - hard to do, given that principals can hand off capabilities
- ACLs grow large when object is heavily shared
  - can simplify by using "groups"
    - put users in groups, put groups in ACLs
    - you are could be in the "cs537-students" group
  - additional benefit
    - change group membership, affects ALL objects that have this group in its ACL

Protection in the Unix FS

- Objects: individual files
- Principals: owner/group/world
- Actions: read/write/execute

- This is pretty simple and rigid, but it has proven to be about what we can handle!

File sharing

- Each user has a "file descriptor table" (or "per-user open file table")
- Each entry in the channel table is a pointer to an entry in the system-wide "open file table"
- Each entry in the open file table contains a file offset (file pointer) and a pointer to an entry in the "memory-resident i-node table"
- If a process opens an already-open file, a new open file table entry is created (with a new file offset), pointing to the same entry in the memory-resident i-node table
- If a process forks, the child gets a copy of the channel table (and thus the same file offset)