CSE 451: Operating Systems
Winter 2013

File Systems

Gary Kimura

File systems

- The concept of a file system is simple
  - the implementation of the abstraction for secondary storage
    - abstraction = files
  - logical organization of files into directories
    - the directory hierarchy
  - sharing of data between processes, people and machines
    - access control, consistency, …
- The discussion on file systems often center around two concepts
  - There is the on-disk structure (i.e., how is the data persistently stored on secondary storage)
  - There is the software component that manages the storage and communicates with the user to store and retrieve data (hopefully without any loss of information)

Files

- A file is a collection of data with some properties
  - contents, size, owner, last read/write time, protection …
- Files may also have types
  - understood by file system
    - device, directory, symbolic link
  - understood by other parts of OS or by runtime libraries
    - executable, dll, source code, object code, text file, …
- Type can be encoded in the file's name or contents
  - windows encodes type in name (and contents)
    - .com, .exe, .bat, .dll, .jpg, .mov, .mp3, …
  - old Mac OS stored the name of the creating program along with the file
  - unix does both as well
    - in content via magic numbers or initial characters (e.g., #!)

Basic operations

Unix

- create(name)
- open(name, mode)
- read(fd, buf, len)
- write(fd, buf, len)
- sync(fd)
- seek(fd, pos)
- close(fd)
- unlink(name)
- rename(old, new)

NT

- CreateFile(name, CREATE)
- CreateFile(name, OPEN)
- ReadFile(handle, …)
- WriteFile(handle, …)
- FlushFileBuffers(handle, …)
- SetFilePointer(handle, …)
- CloseHandle(handle, …)
- DeleteFile(name)
- CopyFile(name)
- MoveFile(name)

File access methods

- Some file systems provide different access methods that specify ways the application will access data
  - sequential access
    - read bytes one at a time, in order
  - direct access
    - random access given a block/byte #
  - record access
    - file is array of fixed- or variable-sized records
  - indexed access
    - FS contains an index to a particular field of each record in a file
    - apps can find a file based on value in that record (similar to DB)
- Why do we care about distinguishing sequential from direct access?
  - what might the FS do differently in these cases?

Directories

- Directories provide:
  - a way for users to organize their files
  - a convenient file name space for both users and FS's
- Most file systems support multi-level directories
  - naming hierarchies (c:\, c:\DocumentsAndSettings, c:\DocumentsAndSettings\MarkZ, …)
- Most file systems support the notion of current directory
  - absolute names: fully-qualified starting from root of FS

    `C:\> cd c:\Windows\System32`
  - relative names: specified with respect to current directory

    `C:\> c:\Windows\System32` (absolute)

    `C:\Windows\System32> cd Drivers`

    (relative, equivalent to cd `c:\Windows\System32\Drivers`)

Directory internals

- A directory is typically just a file that happens to contain special metadata
  - directory = list of (name of file, file attributes)
  - attributes include such things as:
    - size, protection, location on disk, creation time, access time, …
  - the directory list can be unordered (effectively random)
    - when you type “ls” or “dir /on” , the command sorts the results for you.
    - some file systems organize the directory file as a BTree, giving a “natural” ordering
      - What case to use for sort?
      - What about international issues?

Path name translation

- Let's say you want to open "C:\one\two\three"

```
success = CreateFile("c:\\one\\two\\three", …);
```

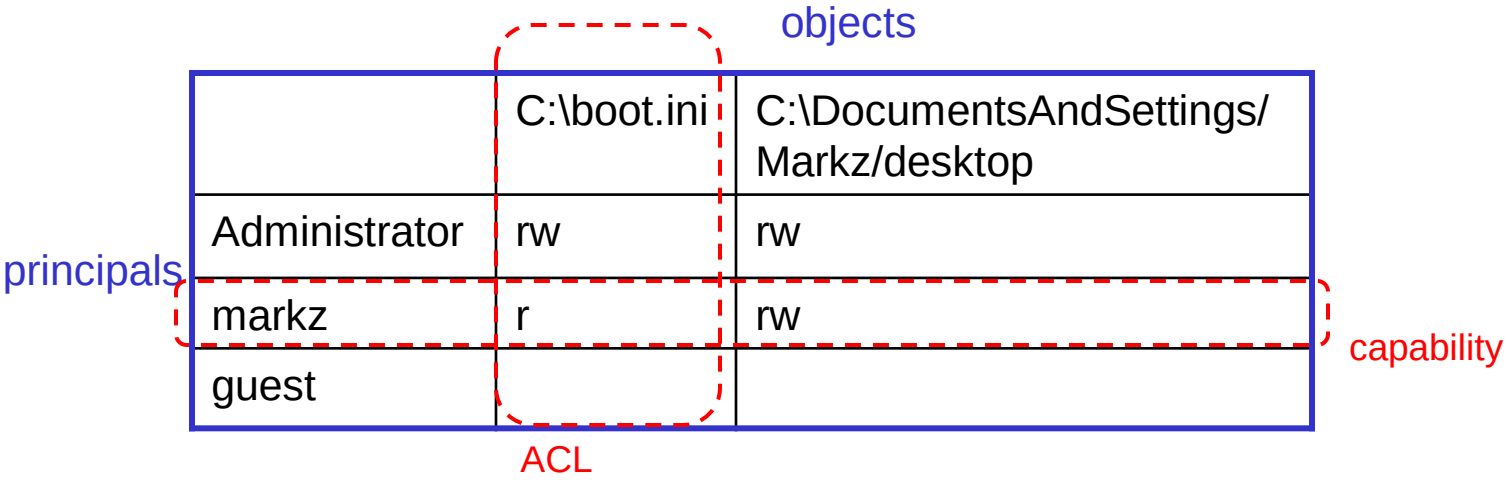
- What goes on inside the file system?
  - open directory "c:\" (well known, can always find)
  - search the directory for "one", get location of "one"
  - open directory "one", search for "two", get location of "two"
  - open directory "two", search for "three", get loc. of "three"
  - open file "three"
  - (of course, permissions are checked at each step)
- FS spends lots of time walking down directory paths
  - this is one reason why open is separate from read/write (session state)
  - FS will cache prefix lookups to enhance performance
    - C:\Windows, C:\Windows\System32, C:\Windows\System32\Drivers all share the "C:\Windows" prefix

File protection

- 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 delete)
- 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 `C:\Windows\System32\ntoskrnl.exe` 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
  - capabilities
    - for each principal, keep list of objects and principal's allowed actions
- 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
  - additional benefit
    - change group membership, affects ALL objects that have this group in its ACL

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 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
  - superblock also contains i-node number of root directory

The flat (i-node) file system

- Each file is known by a number, which is the number of the i-node
  - seriously – 0, 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

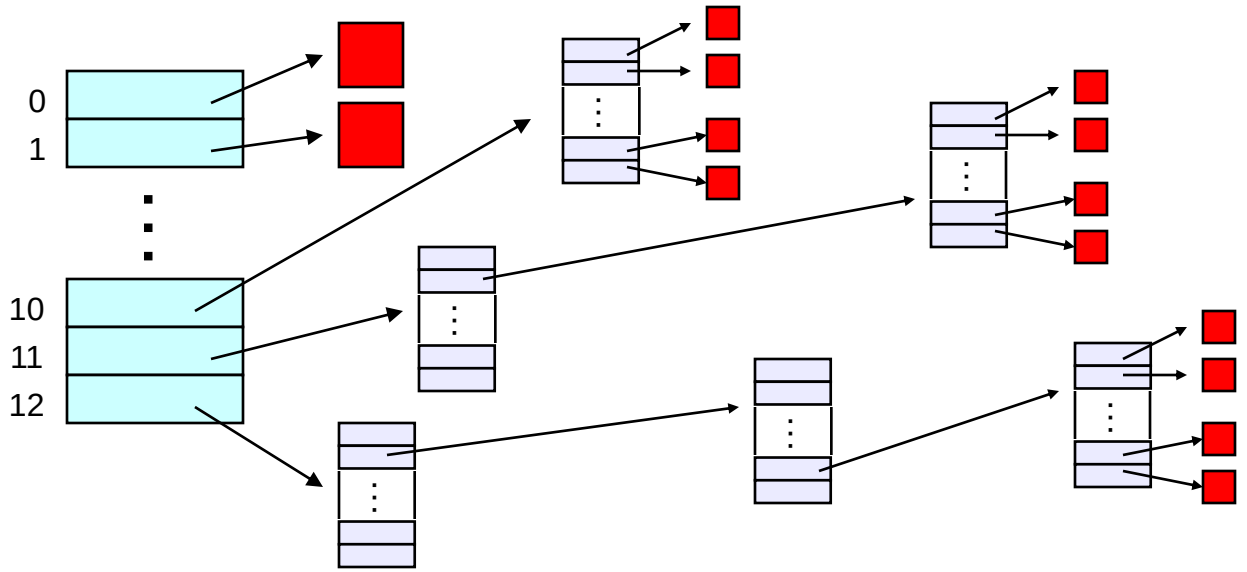
- A directory is a flat file of fixed-size entries
- Each entry consists of an i-node number and a file name

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!

The “block list” portion of the i-node (Unix Version 7)

- Points to blocks in the file contents area
- Must be able to represent very small and very large files. How?
- Each inode contains 13 block pointers
  - first 10 are “direct pointers” (pointers to 512B blocks of file data)
  - then, single, double, and triple indirect pointers



Protection

- Objects: individual files
- Principals: owner/groups/everyone
- Actions: read/write/execute

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

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 or via write-through, but performance would suffer big-time

Consistency of the Flat file system

- Is each block accounted for?
  - Belongs to precisely one file or is on free list
  - What to do if in multiple files?
- Mark-and-sweep garbage collection of disk space
  - Start with bitmap (one bit per block) of zeros
  - For every inode, walk allocation tree setting bits
  - Walk free list setting bits
  - Bits that are one along the way?
  - Bits that are zero at the end?

Consistency of the directory structure

- Verify that directories form a tree
- Start with vector of counters, one per inode, set to zero
- Perform tree walk of directories, adjusting counters on every name reference
- At end, counters must equal link count
  - What do you do when they don’t?

Journaling File Systems

- Became popular ~2002, but date to early 80's
- There are several options that differ in their details
  - Ntfs (Windows), Ext3 (Linux), ReiserFS (Linux), XFS (Irix), JFS (Solaris)
- Basic idea
  - update metadata, or all data, *transactionally*
    - *"all or nothing"*
    - *Failure atomicity*
  - if a crash occurs, you may lose a bit of work, but the disk will be in a consistent state
    - more precisely, you will be able to quickly get it to a consistent state by using the transaction log/journal – rather than scanning every disk block and checking sanity conditions

Why are journaling file systems so popular?

- In any file system buffering is necessary for performance
- But suppose a crash occurs during a file creation:
    1. Allocate a free inode
    2. Point directory entry at the new inode
- In general, after a crash the disk data structures may be in an inconsistent state
  - metadata updated but data not
  - data updated but metadata not
  - either or both partially updated
- fsck (i-check, d-check) are *very* slow
  - must touch every block
  - worse as disks get larger!

Where is the Data?

- In the file systems we have seen already, the data is in two places:
  - On disk
  - In in-memory caches
- The caches are crucial to performance, but also the source of the potential “corruption on crash” problem
- The basic idea of the solution:
  - Always leave “home copy” of data in a consistent state
  - Make updates persistent by writing them to a sequential (chronological) journal partition/file
  - At your leisure, push the updates (in order) to the home copies and reclaim the journal space
  - Or, make sure log is written before updates

Undo/Redo log

- Log: an append-only file containing log records
  - <start t>
    - transaction t has begun
  - <t,x,v>
    - transaction t has updated block x and its new value is v
      - Can log block “diffs” instead of full blocks
      - Can log *operations* instead of data (operations must be idempotent and undoable)
  - <commit t>
    - transaction t has committed – updates will survive a crash
- Committing involves writing the records – the home data needn’t be updated at this time

If a crash occurs

- Open the log and parse
  - <start> <commit> => committed transactions
  - <start> no <commit> => uncommitted transactions
- Redo committed transactions
  - Re-execute updates from all committed transactions
  - Aside: note that update (write) is *idempotent*: can be done any positive number of times with the same result.
- Undo uncommitted transactions
  - Undo updates from all uncommitted transactions
  - Write “compensating log records” to avoid work in case we crash during the undo phase

Managing the Log Space

- A cleaner thread walks the log in order, updating the home locations (on disk, not the cache!) of updates in each transaction
  - Note that idempotence is important here – may crash while cleaning is going on
- Once a transaction has been reflected to the home blocks, it can be deleted from the log

Impact on performance

- The log is a big contiguous write
  - very efficient, but it IS another I/O
- And you do fewer scattered synchronous writes
  - very costly in terms of performance
- So journaling file systems can actually improve performance (but not in a busy system!)
- As well as making recovery very efficient

Want to know more?

- CSE 444! This is a direct ripoff of database system techniques
  - But it is *not* what Microsoft Windows Longhorn (aka Vista) was supposed to be before they backed off – "the file system is a database"
  - Nor is it a "log-structured file system" – that's a file system in which there is nothing but a log ("the log is the file system")

- "New-Value Logging in the Echo Replicated File System", Andy Hisgen, Andrew Birrell, Charles Jerian, Timothy Mann, Garret Swart
  - http://citeseer.ist.psu.edu/hisgen93newvalue.html