COMP 3000, Lecture #18 (November 18, 2004)
Notes by Mark Wong

Anil's
~~Mark's~~ FAQ

A Filesystem solution to what problem?

There are two types of memory:
1. small amounts of fast, volatile storage (RAM and L1, L2, L3 cache)
2. large amounts of slow, persistent storage (hard disks, but also CD-ROMs, DVDs, etc.)
   virtual memory does not care if this storage is persistent or not
   need a nice way to organize large amounts of storage

A filesystem is a solution for the second type, (persistent storage) of memory.

How do you organize a filesystem?

Filesystems allow organization amongst a set of files

What we can use
– can set up a table with fields and records (like a spreadsheet)
– can also setup a relational database (sets of connected tables)

What we typically use
- key/value mappings of filename to contents of file
- Windows systems have multiple trees for each drive "c:\", "a:\"
- UNIX systems have a single tree for everything ("/") with drives as sub-trees ("cdrom")

Why do we have separate commands to prepare and clean up file access?
Or, why not just say read(filename, offset, # bytes) instead of:
- open(filename)
- read / write / seek
- close

We do this for security and performance. The open file command can do name resolution (key to data mapping), caching, and security checks at once when a file is opened so that read / write / seek do not have to do them on each call.

What is a namespace?

A namespace is just a set of names in which every name is unique. (Thanks, Dictionary.com!)

Do filenames form a namespace?

Yes! (To be precise, *pathnames* form a namespace – after all, you can have multiple files with the same name so long as they are present in different directories.)

Why do we have multiple namespaces? Why not just use the filename namespace for everything?

Some systems, such as Plan 9, do use the filename namespace for everything (network sockets, pipes, processes, etc.). UNIX systems do this to a large degree, but not completely.

In the past, people created separate namespaces for reasons of efficiency: files had a minimum size and were slow to reference, so it made sense to make new namespaces within individual

files. Hence, Windows has the registry: a set of key/value pairs, where registry keys form a namespace. Maybe in the future the registry will just become a directory tree full of lots of small files, but I'm not holding my breath.

New filesystems such as reiserfs aim to make filesystems good at storing both large and small values (file contents) – but such systems are hardly widespread.

What other namespaces are there?

- Windows Registry keys
- Uniform Resource Locators (URLs, e.g. web page addresses). Note that on a given web server, an http request may map to a file, or it may map to a database query or something else entirely. Thus, the URL namespace is not strictly hierarchical.
- In UNIX the "/sys" and “/proc” directories form namespaces that maps named kernel data structures to their contents. Note how these namespaces are mapped into the larger filesystem namespace (“/”).

Why do I see those damn tree structured filesystems everywhere?

– Trees are a good way of representing hierarchies – and humans like organizing things in hierarchies
– Trees can be easily manipulated – merged, split, and rearranged.
– A tree structure is elegant for access control, because permissions from a given node in a tree can propagate to its children.
– In contrast, a database structure is not so elegant for access control. Take Oracle for example. It is a database structured filesystem. It can be so complex to setup permissions that people don't bother (e.g. the initial installation of Banner at Carleton).

What are the different types of ways of linking in filesystems?

- Hard Link
  Supported by UNIX operating systems
  Links share the same data, however each link has its own filename for the data.

- Symbolic Link
  Reference to another file using a path
  think Windows shortcuts
  symbolic links can be “broken” by moving/deleting the link's target. (Hard links aren't affected by moves or deletions.)

How do hard links work?

To make hard links work we need another data structure and another namespace: inodes and inode numebers
– An inode number is a unique name for an inode in a given UNIX filesystem (each filesystem defines its own inode number namespace)
– Every regular file is actually a hard link: a directory entry that maps a name to an inode #
– Because hard links refer to inode #'s, they can only refer to inodes on the same filesystem (e.g. same physical device/partition).
– In contrast, symbolic links can span multiple devices. Thus, if your home directory in /home is in a separate partition from that holding /usr/bin, you can't make a hard link from /usr/bin/more to /home/soma/more – but you can always make a symbolic link.
– Hard links can be created with “ln”; symbolic links with “ln -s”.
– “rm” just removes hard links in UNIX. An inode is automatically deleted when there are no remaining hard links to it.

What are some of the UNIX filesystem block types?

Note that filesystem blocks not have to match device block sizes (*eg: device block of 512bytes and filesystem block of 4k*)

UNIX filesystem bock types
1) Superblock
- this is the root of the filesystem tree (not the pathname tree, the tree that holds references to all filesystem blocks)
- this tells the filesystem where everything else is (*iNodes, Data*)
- UNIX filesystems have multiple copies of the superblock in standard locations to help with disaster recovery (sysadmins must manually tell the system to look at alternative superblocks, though)

2) iNode (inode)
can either be statically or dynamically allocated. (*static allocation is typical*)
the superblock specifies a range of addresses where iNodes reside
static allocation
there are a set number of iNodes that are statically allocated when creating the filesystem
if you run out of iNodes, you can't create any files
this is done for simplicity for the superblock. (*knows exactly the range of iNodes*)

3) Data
File contents

How do you connect inodes to data blocks?

- we do not put data directly after the iNode because of fragmentation issues
- the ISO9660 format for cdroms uses iNodes with a single pointer to data in consecutive blocks
this strategy is ideal only for storing data which is read only
if data changed, we'd get horrible fragmentation
- can have a linked list of blocks where end of blocks points to next block
- can have a wide tree structure to minimize number of drive seeks per file access. (normal case)

What is contained in an inode?

Standard fields in an inode
- inode number (or, may simply be implied by location of inode on disk)
- metadata (*data about the file*) – DOES NOT contain the filename!
- direct pointers to data blocks *(for small files with few data blocks)*
- indirect single (*points to one block which points to a bunch of other blocks*)
- indirect double (*points to a block which is a two level tree if single indirect fills up*)
- indirect triple (*points to a block which is a three level tree if double indirect fills up*)

Some filesystems stuff small files directly into the inode.

Extent-based filesystems try to arrange data blocks into contiguous block ranges to reduce number of disk seeks. (Remember, disks are really slow at random access vs. streaming reads and writes!)

What structure does the inode metadata take?

Contents of iNode metadata
- user ID
- group ID
- #links (*number of hard links referring to the inode when the number of links hits zero, the inode is deleted and the referred data blocks are recycled*)

- timestamps
  - data last access
  - data last modified
  - inode last modified
- permissions

What is the structure of UNIX permissions?

Unix permissions can be applied to files and directories.

Structure of UNIX permissions
- read / write / execute bits, divided into three groupings: user / group / other
- there are also a few special bits
  - u+s, g+s on a executable file: will run as another user/group (allows regular users to do things as root by running certain programs, e.g. passwd)

each user / group / other have three bits which represents RWX (*read/write/execute*)

Directories have somewhat odd permissions:
- Write access to a directory means a file (a name->inode mapping) can be created or deleted
- Read access allows the directory's contents to be listed
- Execute lets you retrieve the referenced inode if you already know the name
- So, if you have execute but not read access to a directory, you can access files for which you know the name, but you can't list the contents of the directory.
- You can't do anything with directories that you have read but not execute permissions
- u+s, g+s on a directory ensures that new files will be created with the same user ID/group ID as that of the directory.
- +t on a directory means that files created by one user cannot be deleted by another user (normally used on temporary directories that are writable by everyone)