25 Naming: Directories

Naming: how do users refer to their files? How does OS find file, given name?

Users need a way of getting back to files that they leave around on disk. One approach is just to have users remember file descriptor indexes (e.g., i-node numbers).

Of course, users want to use text names to refer to files. Special disk structures called *directories* are used to tell what descriptor indices correspond to what names.

Approach #1: have a single directory for the whole disk. Use a special area of disk to hold the directory.

- Directory contains <name, index> pairs.
- If one user uses a name, no-one else can.

Many early computers worked this way.

Approach #2: have a separate directory for each user (TOPS-10 approach). This is still clumsy: names from a user's different projects get confused.

Unix (originally Multics) approach: generalize the directory structure to a tree.

- Directories are stored on disk just like regular files (i.e. file descriptor with 14 pointers, etc.) except file descriptor has special flag bit set. User programs can read directories just like any other file (try it!). Only special system programs may write directories.
- Each directory contains <name, fd index> pairs in no particular order. The file pointed to by the index may be another directory. Hence, get hierarchical tree structure. Names have slashes separating the levels of the tree.
- There is one special directory, called the *root*. This directory has no name, and is the file pointed to by descriptor 2 (descriptors 0 and 1 have other special purposes). Go through lookup example: /a/b/c.

It is possible for more than one directory entry to refer to a single file ("hard links"). UNIX uses reference counts in the file descriptors to keep track of the directory entries, only delete file when last directory entry goes away.

Other things kept in UNIX file descriptors: file size, access times, owner and group id, protection bits.

Directories and file descriptors are separate, and directories are implemented just like files. This simplifies the implementation and management of the structure (can write "normal" programs to manipulate them as files).

Working directory: it is cumbersome constantly to have to specify the full path name for all files.

- In Unix, there is one directory per process, called the current working directory (CWD), which the system remembers.
- When it gets a file name from the process, it assumes that the file is in the working directory. "/" is an escape to allow full path names.
- Many systems allow more than one current directory. For example, check first in A, then in B, then in C. This set of directories is called the *search path* or *search list*. This is convenient when working on large systems with many different programmers in different areas.
- For example, in Unix the shell will automatically check in several places for programs. However, this is built into the shell, not into Unix.
- C-shell implements "~".

`chroot()`: change root (changes the meaning of "/" for a process). Can be used to restrict a process to a subtree of the filesystem name space. Files that are not children of the new root become effectively invisible. Example:
`chroot(‘‘tmp/sandbox’’)` ensures that subsequent calls to
`open(‘‘/foo/bar’’, ...)` are interpreted as
`open(‘‘/tmp/sandbox/foo/bar’’, ...)`.

Symbolic links: a file whose contents are just another file name. Also stored on disk just like regular files, but with a special flag set in descriptor.

Why support both symbolic links and hard links?

- Consider hard links that may point to directories: May get cyclic directory graphs:
  - have to take care to avoid infinite loops during directory searches.
  - potential for unreachable directory subgraphs (cycles of garbage.)
- Unix solution, part 1: users can create hard links to plain files only (superuser can create hard links to directories.)
- Unix solution, part 2: provide symbolic links ("soft links") for convenience.
- Symbolic links may refer to directories and may thus form general graphs (cycles!).
  - since symbolic links are special, can avoid infinite loops by ignoring or limiting the number of symbolic link traversals during directory searches.
  - since symbolic link references are unaccounted, no danger of garbage cycles.
  - symbolic links can refer to directories in other file systems, which may reside on different disks (and even on different computers connected by networks.)
    - can build global namespaces

What if a systems has multiple filesystems (potentially on multiple disks)?

- system boots from primary boot partition on primary disk (configuration parameter)
- in UNIX systems:
  - root directory in boot partition becomes "/"
  - other filesystems are "mounted" into the namespace
  - for instance: (assume /dev/rz1a is boot partition)
    - mount /dev/rz1b /usr1
    - mount /dev/rz2a /usr2
    - mount /dev/rz2b /usr3
- in Microsoft systems:

  - each filesystem is assigned a "drive letter" in range [A-Z]
  - drive letter, followed by ":" is prepended to filename

With a *logical volume manager (LVM)*, multiple block devices or device partitions can be made to appear as one large *virtual* device.

- LVM acts as a layer of indirection between filesystem and physical block devices.
- LVM manages multiple disks, hiding them from the rest of the system. Instead, it presents a large, idealized, contiguous volume (= virtual disk) to the filesystem.
- This allows older filesystems such as FFS/UFS, ext2, etc. to be used across multiple disks/partitions.
- LVM can also provide redundancy, on-the-fly encryption, etc.

ZFS comes with its own LVM:

- system admins can add any number of block devices to a single filesystem
- no need to mount different filesystems into a single namespace