Files and Filesystems

Tree Structure

- The model popularized by Unix and Linux.
- **Directory**: file that maps *file name* ***components*** to files; they are literally just look-up tables.
- **Regular files**: byte sequences (can be read from or written to, unlike directories).
- **Special files**: built into the kernel e.g. `/dev/null`.
- **Symbolic links** (aka **soft links**): single files that contain a single string that is interpreted as a file name (could be any file or directory, including another symbolic link). When using `ls -l`, you can see symbolic links with `->` pointing to their **target**.
- Every file name look-up follows the pattern: starting at the root, consult the directory table, resolving any symbolic links along the way.
- A **file name component** is a nonempty sequence of non-slash characters. The `/` character is *very special* in the POSIX file convention because they are interpreted as part of a path in the tree filesystem.

> `/dev/null` is useful for "reading nothing" or discarding output:
>
> ```
> echo abc > /dev/null
> ```

TIP: The first flag for a file when using `ls -l` tells you the file type: `d` for directory, `l` for symbolic link, `c` for special files like `/dev/null`, `-` for regular files.

File names are not recursive by design. If you want to search for files that match the pattern like `/usr/*/emacs`, then use the `find` command:

```
find /usr/ -name emacs
```

Every file in the POSIX standard has a unique **inode number** associated with it. You can see them with the `-i` flag in the `ls` command:

```
$ ls -idl /
2 dr-xr-xr-x. 21 root root 4096 Dec 17  2021 /
# 2 is the inode num
```

Linux Filesystems

It's not *actually* a tree, but it still does not have loops - it's a **directed acyclic graph (DAG)**. They achieved this with the notion of **hard links**, which are like aliases for the same file in memory. The iron-clad rule is that hard links *must not point to directories*.

```
$ ls -al /usr/bin
total 373832
sdafhausegulaskhglkas .
dsafhsagkjsajhgjasjga ..
sdafkjdhsajsadhgklsda '['
```

Every directory includes links to themselves (`.`) and their parent directory (`..`). These are hidden by default, but you can view these with `ls -a`. Note that the parent of the root directory `/` is itself, `/`.

File names *starting* with slash are **absolute paths**, paths that start at the root. File names that do not start with a slash are interpreted as **relative paths**, paths starting from the **current working directory**.

Remember that the file system is just data structures that are mostly on *secondary storage*, which are *persistent*, but much MUCH slower than primary storage aka RAM. Just like RAM, filesystems can have "pointers" too which reference locations on the hard drive. Thus, data structures in the filesystem have to be very intelligently designed in order to be efficient.

Data Structure Representations

A directory is really just a *mapping* of the file names components to the inodes (via inode numbers) in the filesystem tree.

You can visualize the filesystem as a graph of nodes representing file objects in memory but the file name components along the *edges* of the graph.

The filesystem is not a tree, but there are some limitations enforced:

1. You cannot have two different parents with the same directory. The "tree-like" structure holds for directories.
2. However, you can have multiple links to the same non-file. Files are often identified with names, but actually names are just **paths TO** the file. You cannot in general look at a file in a filesystem and determine what it's "name" is because it could have multiple of them. Names are really just useful aliases that we as users assign to the actual file object.

Hard Links

Creating a **hard link** - "create a file named b that's the same as the file already named a":

```
ln a b
```

This is like a *mutable reference* in C++. If you modify one, you modify the other because they're the *same file in memory* and even share the same inode number:

```
$ ln a b
$ echo "foo" > b
$ cat b
foo
```

```
$ cat a
foo
```

Hard links work because a directory is simply a *mapping* of file name components to files. The file names are different, and there's no rule saying different keys can't map to the same value, so everything is consistent with what the definition of a directory.

Soft Links

Soft link (symbolic links) on the other hand are actual separate data structures that have content (the string that is interpreted as the path to their target). A hard link only contributes to the directory size (expands the mapping by one entry). The underlying file remains unchanged.

Symbolic links can also point to nowhere. They can be **dangling**. When using such a pointer, the OS will try to resolve the existing path that's saved as the content of its file, but if that file no longer exists, then you get the error:

```
linkname: No such file or directory
```

A symbolic link is always interpreted *when you use it*, NOT when you create it. If a dangling symlink is pointing to a non-existent `foo`, but then you create a new file `foo`, the symlink works again.

What is the main difference between a hard link and a symbolic link?

A hard link increments the file object's reference count. A symbolic link does not increment the file's reference count.

Deleting all hard links to a file (and ceasing all operations that use the file) deletes the file. Deleting symbolic links do not affect the underlying file. A symbolic link can become dangling if the underlying file is deleted.

Destroying a File

You can use the "remove" command:

```
rm file
```

However, this is actually an operation on the directory, not the file in memory itself. What `rm` is doing is modifying the current directory so that `file` no longer *maps* to the file object it did. The contents of the file object still exist in memory. So as a continuation from above:

```
rm bar
ls -ali
# foo still shows up
```

So how does the filesystem know when to reclaim the memory?

Recursively searching every directory for any remaining hard links would be too slow.

Instead, the filesystem maintains a reference count called a **link count**. Associated with each file is a number that counts the number of hard links to the file i.e the number of entities that map to this file.

IMPORTANT: Symlinks DO NOT contribute to the link count. Link counts ONLY count hard links.

You can use this to list the link count to each file, next to the permission flags:

```
ls -li
```

HOWEVER: Memory cannot be reclaimed until all processes using/accessing the file are done with it.

Operations like `rm` simply decrement the link count. **If it's 0,** there's still one step left: the OS must wait until every process accessing the file exits or closes. **Then** the OS can reclaim the memory.

Likewise, operations like `ln` *increment* the link count, for both the original file and the new hard link.

ALSO: Even if the link count is 0 and no processes are working, the data is still sitting in storage. It could be overwritten naturally by new files, but it is not guaranteed, and you must use low level techniques to either irreversibly remove the content or recover it.

ASIDE: Some Filesystem Commands

The `ls -li` flag outputs each entry in the format:

```
$ ls -l
(inode num) (file type AND permissions) (link count) (owner)
```

Listing all the filesystems and how much space is available in each of them:

```
df -k
```

`find` supports searching by inode number:

```
find . -inum 4590237 -print
```

Remove every file with that inode number found within the current directory:

```
find . -inum 4590273 -exec rm {} ';'
```

Soft Link Edge Cases

Can you have symbolic links to directories?

```
$ ls -li /bin
... /bin -> usr/bin
```

Yes. A symlink can even be resolved in the middle of a file name (in which case, it better be a directory).

Can a symlink point to another symlink? Yes.

Can a symlink point to itself? Yes. It can even point to a symlink that points back to itself. They aren't *dangling* lists, but if you try resolving the path, you enter an infinite loop.

The kernel has a guard against this. If you attempt this, you get the error:

```
filename: Too many levels of symbolic links
```

Symbolic links to hard links? Yes.

Hard links to symbolic links? Yes.

Give an example of how renaming a dangling symbolic link can transform it into a non-dangling symbolic link.

Soft links can be linked to a relative path. Suppose `c` has content `b`, but `b` only exists as `temp/b` relative to the current working directory. We can use `mv c temp` to make the relative path `b` now work since `c` is now in the same directory as `b`.

Secondary Storage

Flash drives wear out.

"4 TB drive 200TBW" means it has 4TB capacity and you are guaranteed that you can overwrite the same block of memory with 200TB worth of data. After that, that part of the drive wears out.

Linux solves this problem by moving the file around. The user thinks they're writing to the same area, but the OS is actually writing to different parts of the drive.

The implication is that if you remove the file, there are actually parts of the file still scattered around the drive. Attempting a clean wipe of the drive is more involved, and there is also software dedicated to recovering deleted date from these data remnants.

ASIDE: Overwriting the content of a file with random junk:

```
shred foo
```

The `mv` Command

You can use the "move" command to rename and/or move a file.

```
mv foo.c bar.c
mov foo.c bar  # if bar's a directory, it becomes bar/foo.c
```

This is a very *cheap* operation because it actually just modifies the mapping of the directory instead of the files themselves. What this does at the low level is:

- Remove one directory entry
- Add another entry in a directory, possibly the same one or another directory

`cp` on the other hand is more expensive because it has to actually iterate over the content of the file. Although, we learn later that that might not always be the case with the **copy-on-write** technique.

File Permissions

When you run something like `ls -l` you see that the first column has a sequence of characters represented like:

```
rw-r--r--
```

The permissions bits are just a 9-bit number, which stores 3 groups of octal numbers representing the `rwx` permission bits for the `ugo` (owner, group, other) of the file. The flags displayed with `ls -l` have ten bits, with the leading bit representing the **file type**:

Bit	File Type
-	regular
d	directory
l	symbolic link
c	char special file
...	...

Technically, the 9 bits are actually 12 bits because they are encoded in a way to allow for the special flag:

```
$ ls -lai /bin/sudo
... -rws-r-xr-x 1 root root ... /bin/sudo
```

The `x` flag of the octal number for the owner category is an `s`, short for **superuser**. This means that this command is *trusted* by the OS. And no, you cannot just:

```
$ chmod +s /bin/sh
chmod: changing permissions of 'bin/sh': Operation not permitted
```

Sensible Permissions

A set of rwx permissions on a file is called "**sensible**" if the owner has al the permissions of the group and the group has all the permission of others. For example:

- 551 (r-x|r-x|--x) is sensible. The owner's permissions are a *superset* of those of the group.
- 467 (r--|rw-|rwx) is not. The group has `w` permission while the owner doesn't.

Non-sensible permissions don't make sense because the *owner* is considered to be the most closely related to the file, so they should have the most access.

How many distinct sensible permissions are there?

Solution:

Consider r, w, x permissions separately. For each of then, there are 4 ways to gst sensible permissions for u, g, and o (in binary, 000, 100, 110, 111).

Total: $4^3 = 64$

So How Do You *Actually* Update a File?

Options:

1. Write directly to a file `F`. But if some other program reads the file at the same time, problems could arise. You want to be able to update files (and databases) **atomically**.
2. Write to a temporary file `F#` and then `mv F# F`. The downside obviously is that it occupies twice the space on drive. The upside is that because `mv` is **atomic**, any other processes attempting to use the file at the same time will either get the old file or the new file, not some intermediate state.