File and Filesystem

A `filesystem` is a data structure that an OS(Opearting System) used to control how data is stored and retrieved on a storage device, such as like a hard drive, SSD... It organizes data into files and directories(folders) to make it easy for user to access.

A `kernel` is the core component of an OS, acting as an intermediary between the computer's harware and the software applications that run on it. It has the following responsibilities:

- Memory management
- Filesystem management
- Device drivers
- Process management

Tree Structure

- The model popularized by Unix and Linux.
- **Directory** (aka **folder**): a container for files or other directories, known as subdirectories. Directories serve as lookup tables that map *file name components* to actual files, making it easier to organize and navigate the filesystem.
- **Regular files**: the most common file type, representing *a sequence of bytes*. They can contain data, text, or program instructions and can be read from or written to.
- **Special files**: used for device I/O operations(built into the kernel) and are typically found in the `/dev` directory. e.g. `/dev/null`, which is used for discarding any data written to it
- **Metadata**: is information of a file, which includes file permissions, ownership, file type, and pointers to the disk blocks where this file's data is stored
- **Inode number #**: a unique identifier(识别码) associated with a file or directory (not a pointer or a memory address in the traditional sense used in programming). It serves as an index to the inode table *where metadata about the file or directory is stored*.

  > Essentially, the inode # is the filesystem's way of keeping track of files and directories, allowing the O.S to access the file's metadata without referring to its name or path.

- **Symbolic links** (aka **soft links**): special files that point(contain the path) to another file or directory in the filesystem. With the path contained, it can link to any file or directory, including another symlink. When using `ls -l`, you can see symlink with `->` pointing to their **target file**. (Symlinks are resolved recursively until a regular file or directory is reached)

  > The symbolic link's inode # is different from the inode # of the target file it points to.
  > When accessing a file through a symlink, the system uses the path stored in this symlink's inode # to find another inode # of the target file.

- **File Name Lookup**: When accessing(or finding) a file, the filesystem starts at the root directory and follows the path specified, resolving any symbolic links encountered along the way. This process

continues until the final file or directory is accessed.

- **File Name Components and `/` Charactert**: In the POSIX file convention, the `/` character is used as a delimiter(分隔符) in file paths, separating different components of the path. Each component of the path refers to a directory or file name, except for the root directory, which is represented by a single `/`.

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

```
echo abc > /dev/null
# sends the output of echo abc (which would normally print "abc" to the
terminal) to /dev/null, thereby discarding it and displaying nothing
```

TIP: The first flag for a file when using `ls -l` tells you the file type: `d` for directory, `l` for symbolic link, `c` or `b` 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 -lai
...
1234567 -rw-r--r-- 2 yourth staff 4096 Jan 1 12:34 example.txt
...

# 1234567: Inode number of the file

# -rw-r--r--: Permission flags, first '-' means it is a regular file

# 2: Hard link counts. '2' means the file has 2 hard links pointing to it

# yourth: User name

# staff: Group name

# 4096: Size of the file in bytes

# Jan 1 12:34: Timestamp (last modification time)

# example.txt: File name
```

Linux Filesystem

It is structured in a way that resembles a tree(with directories branching out from a root directory). However, due to the presence of **hard links**(aliases for the file) and symbolic links, it's more accurately described as a **Directed Acyclic Graph (DAG)**. This distinction allows for a more flexible structure without creating loops or cycles that could complicate navigation within the filesystem. The iron-clad rule(原则) is that hard links *must not point to directories*.

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

Every directory can link to themselves (`.`) or their parent directory (`..`). These are hidden by default, but you can view these with `ls -a`. Note that the parent of root directory `/` is itself, where `..` points back to itself, illustrating the top of the filesystem hierarchy.

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 #) in the filesystem tree.

You can visualize the filesystem as a graph of nodes representing file objects in memory by `tree /path/to/directory`

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 source_file link_name
```

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. (which means that there can be many hard links to a unique file, BUT every hard link to a unique file shares the **SAME** inode #)

Symbolic Links

Symlinks (Soft link) 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(解析) 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?

> Direct Reference vs Path Reference

A hard link is a direct reference to the file's inode.
A symbolic link contains a path to another file or directory. It's essentially a special file that points to another file or directory by name.

> Same Inode vs Separate Inode

A hard link to a file shares the same inode number as the original file, effectively making it indistinguishable from the original file at the filesystem level.
A symbolic link has its own inode(which is different than the inode of the target file) and its own set of metadata, distinct from the file it points to.

> Reference Count

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

> Deleting and Dangling

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` 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(回收利用) 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 -li
(inode num) (file type AND permissions) (link count) (owner) (last
modification time)
```

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
# denoted by `.`, it searches the current directory
```

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

```
find . -inum 4590273 -exec rm {} ';'
# `-exec`: execute a specified command on each file
# `{}`: placeholder for the current file name found by `find`
# `';'`: command is terminated by a semicolon (;)
```

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
```

Can a symbolic link point to a hard link? Yes.

Can a hard link point to symbolic link? Yes.

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

1. Imagine the following directory structure like this

   ```
   /project
   ├── temp
   │   └── b (Actual file)
   └── c -> b (Symbolic link pointing to "b")
   ```

   In this setup, `/project/c` is a symlink that points to `b`, intending to link to `/project/temp/b`. However, since `c` is in `/project` and `b` is inside `/project/temp`, the symlink `c` is dangling. This means `c` points to a non-existent file from its current location because it expects `b` to be in the same directory (`/project`), not inside `/project/temp`.

2. Making the Symlink Valid

   ```
   mv /project/c /project/temp
   ```

3. Final result

   ```
   /project
   └── temp
       ├── b (Actual file)
       └── c -> b (Symbolic link, now non-dangling)
   ```

ASIDE: `mv` vs `cp` in efficiency

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

```
mv foo.c bar.c # renameing the file
mv foo.c /bar # if bar is a directory, it becomes /bar/foo.c
cp foo.c /bar
```

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**
  `mv` command removes the **directory entry** that maps the file's name to its inode # in the source directory. This operation does not touch the file's data or its inode.

- **Add another entry into a directory**
  `mv` command then creates a new directory entry in the target directory. This new entry maps the file's name (which may be unchanged if you're moving the file or changed if you're renaming it) to the same inode. If the target directory is the same as the source directory (in the case of a rename), this step modifies the existing directory file to include the new name mapping.

`cp` on the other hand is more ***expensive*** because it has to actually iterate over the content of the file.

File Permissions

Basic Permissions(9 bits)

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

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

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

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

```
$ ls -lai
...
1234567 -rw-r-xr-- 2 user group 4096 Jan 1 12:34 example.txt
...

# -rw-r-xr--: Permission flags

# `-`: indicates a regular file

# `rw-`: User(owner) Permissions, file is READABLE and WRITABLE by user

# `r-x`: Group Permissions, file is READABLE and EXECUTABLE by members of
the group

# `r--`: Others Permissions, file is only READABLE by others
```

Special Permissions(12 bits)

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

Set User ID (`s`): When set on an executable file, it allows the file to be executed with the permissions of the file's owner.

Set Group ID (`s`): When set on an executable file, it allows the file to be executed with the permissions of the file's group.

Sticky Bit (`t`): When set on a directory, it restricts file deletion within the directory. Only the file's owner, the directory's owner, or the superuser can delete files.

```
$ ls -lai
...
1234567 -rwsr-xr-t 2 user group 4096 Jan 1 12:34 example.txt
...

# rws: User(owner) permissions with SUID set (s instead of x indicates
that SUID is set and the execute bit is on)

# r-t: Others permissions with the sticky bit set (t instead of x
indicates that the sticky bit is set and the execute bit is on)
```

IMPORTANT: Special Permissions' flags `s` or `t` instead of x indicates that the execute bit `x` is defaultly on.

Modify the Permission using `chmod`

You can change the permission by command `chmod`, which stands for 'changing mode'(aks changing permission).

1. Using `chmod` Symbolically

- `$ chmod [who][+/-][permission flag] file_name` - Give permission for **ugo** to a file (allowing it to be **rwxst**) (change mode(permission))
    - `who` (ugo): user, group, others
    - `permission flag` (rwxst): read, write, execute, setuid/setgid, sticky(restrict deletion)

> `$ chmod u+s g+rw o-x file_name`
>
> `u+s` add s(setuid) permission for the user
>
> `g+rw` add r(read) and w(write) permission for the group
>
> `o-x` remove x(execute) permission for others

2. Using `chmod` Numerically (with an octal value)

- `$ chmod [permission number] file_name` - The **Permission Number** consists of *three digits*, representing *ugo* respectively; The **Size** of each number represents their own *permission*.
    - `0` - no permission
    - **`1` - execute permission**
    - **`2` - write permission**
    - `3` - write and execute permissions (2+1)
    - **`4` - read permission**
    - `5` - read and execute permissions (4+1)
    - `6` - read and write permissions (4+2)
    - `7` - read, write, and execute permissions (4+2+1)

  p.s. You can observe that every additions of permission number are based on `4 2 1`

  > `$ chmod 754 file_name`
  >
  > 7 - User(Owner) is permitted to read (4), write (2), and execute (1).
  >
  > 5 - Group is permitted to read (4) and execute (1).
  >
  > 4 - Others are permitted to read (4) only.

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

The command above attempts to add the setuid bit to `/bin/sh`. The error message indicates that the user does not have such necessary privileges. Only the **superuser(root)** or a user with appropriate capabilities can set or clear the setuid bit on executables.

You can get the privilege by prefixing `sudo` command, which stands for **superuser**.

```
$ sudo chmod u+s /bin/sh
# 'u' stands for user
# '+s' defaultly set the user id, which is same as 'u+s'

$ sudo chmod g+s /bin/sh
# 'g' stands for group
# 'g+s' means set the group id
```

Sensible Permissions

A set of rwx permissions on a file is called "**sensible**" if the user(owner) has all the permissions that the group has, and the group has all the permissions that others have. 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?

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

Total: $4^3=64$ (I don't think it is correct huh, you need ot reconsider it)

Secondary Storage

Secondary storage refers to any data storage device that is not the primary storage or RAM of a computer system.
Unlike primary storage, which is volatile and loses its content when the computer is powered off, secondary storage is non-volatile, meaning it retains data even when the computer is turned off.

Common types of secondary storage include: **HDD(Hard Disk Drive), SSD(Solid-State Drive), USB, CD, Magnetic Tape**, which are used for storing data long-term, backing up data, and transferring data between computers.

Drives vs Wear Out(擦除)

Traditional HDDs use magnetic coatings(磁性涂层) on spinning disks, which are moving parts.
Flash drives like SSDs, use flash memory, which is electronic and has no moving parts, for data storage.
However, flash memory can only endure a limited number of write and erase cycles before the memory cells(存储单元) wear out and then become unreliable.

Total Bytes Written(TBW)

"4 TB drive 200TBW"
4TB capacity: The total amount of data the drive can store.
200TBW: The total amount of data that can be written to the drive over its lifetime.
In this case, it means you can write 200 terabytes of data to the drive before the memory cells begin to wear out significantly.

Linux and Wear Leveling(均匀损耗)

Linux, like other modern operating systems, employs techniques such as wear leveling to extend the lifespan of flash storage.
Wear leveling is a process that aims to distribute write and erase cycles evenly(均衡分配) across the memory cells to prevent any single particular part of the drive from wearing out prematurely(过早损耗).

Data Deletion and Recovery

Due to the process of Wear Leveling, when a file is deleted, the OS typically removes the file's entry from the file system table(current directory), marking the space as available for new data. However, the actual data remains on the drive until it is overwritten.

> **Why OS doesn't directly erase the space by overwritting the new data?**

> If the kernel were to erase the data immediately upon deletion, it would interfere with the wear leveling process, leading to uneven wear and potentially reducing the lifespan of the storage device.

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

```
shred foo
```

`shred` overwrites the file specified as its argument several times. By default, it does this 3 times, but this can be adjusted with the -n option. Each overwrite pass uses different random data, making it difficult to reconstruct the file's original data.

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(缺点) obviously is that it occupies twice the space on drive. The upside(优点) 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.