【OS】Day46

Class	Operating System: Three Easy Pieces
Date	@March 7, 2022

【Ch39】Interlude: Files and Directories

39.15 Symbolic Links

There is one other type of link that is really useful, which is called symbolic link or sometimes soft link.

To create such a link, we can use the same program `ln`, but with the `-s` flag. Here is the example

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
```

The symbolic links are actually quite different from hard links. The first difference is that a symbolic link is actually a file itself, of a different type.

A symbolic link is a third type the file system knows about. A stat on the symlink reveals all:

```
prompt> stat file
... regular file ...
prompt> stat file2
... symbolic link ...
```

Running `ls` also reveals this fact. The first character shown in the left-most column is a `-` for regular files, a `d` for directories, and an `l` for soft links.

We can also see the size of the symbolic link(4 bytes in this case) and what the link points to(the file named `file`).

```
prompt> ls -al
drwxr-x---  2 remzi remzi   29 May  3 19:10 ./
drwxr-x--- 27 remzi remzi 4096 May  3 15:14 ../
-rw-r-----  1 remzi remzi    6 May  3 19:10 file
lrwxrwxrwx  1 remzi remzi    4 May  3 19:10 file2 -> file
```

The reason that `file2` is 4 bytes is because the way a symbolic link is formed is by holding the pathname of the linked-to file as the data of the link file.

Finally, because of the way symbolic links are created, they leave the possibility for what is known as a dangling reference:

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> rm file
prompt> cat file2
cat: file2: No such file or directory
```

Removing the original file named file cause the link to point to a pathname that no longer exist.

39.16 Permission Bits and Access Control Lists

Because files are commonly shared among different users, a more comprehensive set of mechanisms for enabling various degrees of sharing are usually present within file systems.

The first form of such mechanisms is the classic UNIX permission bits. To see permissions for a file foo.txt, we can type

```
prompt> ls -l foo.txt
-rw-r--r-- 1 remzi wheel  0 Aug 24 16:29 foo.txt
```

The permission bits(`rw-r--r--`) determines, for each regular file, directory, and other entities, exactly who can access it and how.

The permissions consist of three groupings: what the owner of the file can do to it, what someone in a group can do to the file, and what anyone(sometimes refer to as other) can do.

This permission bits show that the file is readable and writable by the owner(`rw-`), and only readable by members of the group `wheel` and also by anyone else in the system(`r--` followed by `r--`)

The owner of the file can readily change these permissions by using the `chmod` command(to change the file mode).

```
prompt> chmod 600 foo.txt
```

This command enables the readable bit(4) and writable bit(2) for the owner, but set the group and other permission bits to 0 and 0, respectively, thus setting the permissions to `rw-------`

The execute bit is particularly interesting. For regular files, its presence determines whether a program can be run or not.

For example, if we have a simple shell script called `hello.csh`, we may wish to run it by typing:

```
prompt> ./hello.csh
hello, from shell world.
```

However, if we don't set the execute bit properly for this file, the following happens:

```
prompt> chmod 600 hello.csh
prompt> ./hello.csh
./hello.csh: Permission denied
```

39.17 Making and Mounting a File System

To make a file system, most file systems provide a tool, usually referred to as `mkfs` that performs exactly this task.

However, once such a file system is created, it needs to be made accessible within the uniform file-system tree. This task is achieved via the mount program. What mount does is take an existing directory as a target mount point and essentially paste a new file system onto the directory tree at that point.

Summary

ASIDE: KEY FILE SYSTEM TERMS

- A **file** is an array of bytes which can be created, read, written, and deleted. It has a low-level name (i.e., a number) that refers to it uniquely. The low-level name is often called an **i-number**.
- A **directory** is a collection of tuples, each of which contains a human-readable name and low-level name to which it maps. Each entry refers either to another directory or to a file. Each directory also has a low-level name (i-number) itself. A directory always has two special entries: the `.` entry, which refers to itself, and the `..` entry, which refers to its parent.
- A **directory tree** or **directory hierarchy** organizes all files and directories into a large tree, starting at the **root**.
- To access a file, a process must use a system call (usually, `open()`) to request permission from the operating system. If permission is granted, the OS returns a **file descriptor**, which can then be used for read or write access, as permissions and intent allow.
- Each file descriptor is a private, per-process entity, which refers to an entry in the **open file table**. The entry therein tracks which file this access refers to, the **current offset** of the file (i.e., which part of the file the next read or write will access), and other relevant information.
- Calls to `read()` and `write()` naturally update the current offset; otherwise, processes can use `lseek()` to change its value, enabling random access to different parts of the file.
- To force updates to persistent media, a process must use `fsync()` or related calls. However, doing so correctly while maintaining high performance is challenging [P+14], so think carefully when doing so.
- To have multiple human-readable names in the file system refer to the same underlying file, use **hard links** or **symbolic links**. Each is useful in different circumstances, so consider their strengths and weaknesses before usage. And remember, deleting a file is just performing that one last `unlink()` of it from the directory hierarchy.
- Most file systems have mechanisms to enable and disable sharing. A rudimentary form of such controls are provided by **permissions bits**; more sophisticated **access control lists** allow for more precise control over exactly who can access and manipulate information.