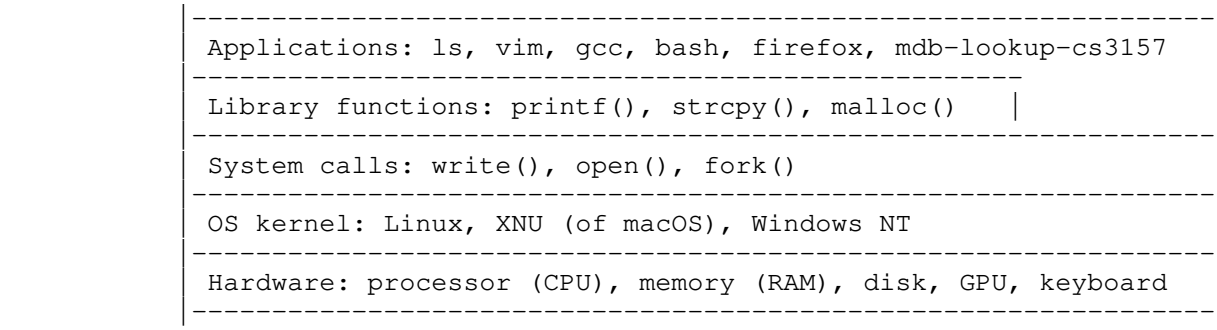
Lecture 15 - UNIX (last updated: 10/31/22)

An operating system (OS) kernel is the software "between" the computer hardware and the other software running on it. The following diagram illustrates their role in modern software system stacks:



With the help of CPU hardware features such as privileged operations and timers, modern OS kernels provide each user program with a virtual environment where they can run as though:

- they have exclusive use of the CPU
- they have access to all memory addresses
- hardware devices are easy to use

Strictly speaking, the kernel is only one part of an OS, which also consists of userspace libraries and programs (e.g., the GNU part of GNU/Linux). Together, these provide the infrastructure upon which modern applications are built.

The UNIX operating system, developed at Bell Labs in the 1970s, established the foundation for most modern OSes. Its derivatives include GNU/Linux, FreeBSD, and macOS, though its design has also greatly influenced Windows. The UNIX-like interface common to these operating systems is codified in the POSIX standard.

In the next few lectures, we will describe some notable features of UNIX. Note that we use the name "UNIX" here to broadly refer to UNIX-like systems, i.e., those that conform to the POSIX standard.

Users

In UNIX, all users have a unique username and user ID (UID). Users also belong to one or more groups, identified by a unique group name and group ID (GID).

You can use the "id" command to find your UID and group membership:

```
$ id
uid=1488(jzh2106) gid=1008(student) groups=1008(student)
```

My username is jzh2106, my UID is 1488, and I am part of the student group.

UNIX systems also have a superuser whose username is root and whose UID is 0. (Note that root should not be confused with sudo ("SuperUser DO"), which is a tool that allows system administrators to easily escalate commands to run with root user privileges.)

File permissions

In UNIX, each file has a user, a group, and a set of permissions (sometimes, the user is called the file's "owner", and the permissions are called the "mode").

You see this metadata using ls -l:

```
    $ ls -l
    -rw------- 1 jzh2106 student  12 Oct 31 01:34 private.txt
    -rw-r--r-- 1 jzh2106 student  17 Oct 31 00:54 public.txt
    -rwxr-x--- 1 jzh2106 student 211 Oct 31 01:03 script.sh
    ^^^^^^^^^^   ^^^^^^^ ^^^^^^^                  ^^^^^^^^^
    |            |       |                        |
    |            |       |                        file name
    |            |       group
    |            user
    permissions
```

Each permission is written here as a 10-character string, which have the following meaning (with spaces added for clarity):

```
    -   r   w   x   r   -   x   -   -   -
    ^   ^^^^^^^^^   ^^^^^^^^^   ^^^^^^^^^
    |   |           |           |
    |   |           |           Other permissions
    |   |           |
    |   |           Group permissions
    |   |
    |   User permissions
    |
    '-' if a file, 'd' if a directory
```

These are sometimes referred to as "owner," "group," and "world" permissions.

Whether a particular user can access a file is determined by their identity and group membership, relative to these permissions. The user permissions only apply when the file owner is trying to access the file; otherwise, the group permissions apply when the user trying to access the file is a member of the file's group; otherwise, the other permissions are used.

For each of the user, group, and other permissions, the three characters (a triad) indicate whether they can be read (r), written (w), and executed (x); a '-' indicates that the owner/group member/other user does not have that permission. So, of the files shown above:

- private.txt can only be read and modified by jzh2106
- public.txt can be be read by anyone, but can only be modified by jzh2106
- script.sh can be read and modified by jzh2106, but can be executed by jzh2106 as well as anyone in the student group

These restrictions do not apply to the superuser, root. The root user can do anything they want.

Permissions triads are often expressed as octal numbers, where the read, write, and execute permissions each correspond to a binary digit with the value 1 when set and 0 otherwise.

So, the files above have the following permissions, in octal notation:

- private.txt: 600
- public.txt: 644
- script.sh: 750

The three octal digits represent the user, group, and other permissions of each file.

You can change the permissions, owner, and group of a file (or directory) using the chmod, chown, chgrp commands, e.g.:

```
$ chmod 755 student.sh    # Make script.sh executable by everyone

$ chown root public.txt   # Give ownership of public.txt to root
```

Note that these command-line tools just provide a convenient interface for the underlying chmod() and chown() system calls.

Be very careful when using these commands/system calls! You can lock yourself out of your own files (or even your entire system) if you incorrectly configure permissions; in that case, you will need help from the root user to reclaim access.

Scripts

Scripts are just human-readable text files with the executable bit set. They contrast executable binaries, which are not human-readable.

Scripts may be written in any interpreted programming language. For example, a shell script just contains a list of shell commands that will be invoked when executed, while a Python script contains Python code.

Scripts should begin with a "shebang" (written #!) to indicate what interpreter should be used to execute the script. For example, for shell scripts, we can use the /bin/sh shell program to interpret the script, by writing the following shebang as the first line:

```
#!/bin/sh
```

Directory permissions

In UNIX, directories have permissions too:

```
$ ls -la
drwxr-xr-x  2 jzh2106 student   5 Oct 31 01:34 .
drwx--x--- 21 jzh2106 student  43 Oct 31 01:34 ..
-rw-------  1 jzh2106 student  12 Oct 31 01:34 private.txt
-rw-r--r--  1 jzh2106 student  17 Oct 31 00:54 public.txt
-rwxr-x---  1 jzh2106 student 211 Oct 31 01:03 script.sh
```

For directories, the permission bits are interpreted a little differently:

- the read bit (r) allows users to list the names of the files in the directory (but not to access those files' metadata)

- the write bit (w), together with the execute bit, allows users to modify entries in that directory, such as creating, deleting, and renaming files; it is meaningless without the execute bit

- the execute bit (x) allows users to access the metadata of a directory entry, provided they already know the name of the file

setuid, setgid, and sticky bits (optional)

UNIX file permissions also include another triad of bits, named the setuid, setgid, and sticky bits. Though they are used less frequently than the user, group, and world permission triads, they provide more fine-grained access control that is very useful for enforcing particular security policies.

The setuid bit allows a user to execute a file as the file owner. For example:

```
$ cd ~j-hui/cs3157-pub/bin

$ ls -l *-cs3157        # only list mdb files related to cs3157
-rwsr-xr-x 1 j-hui j-hui   17416 Oct 18 15:18 mdb-add-cs3157
-rw-r--r-- 1 j-hui student 38200 Oct 31 14:44 mdb-cs3157
-rwxr-xr-x 1 j-hui j-hui     158 Oct 18 15:18 mdb-lookup-cs3157
```

mdb-cs3157 has 644 permissions, so while anyone can read it, only the owner, j-hui, can modify it. Although read permissions are sufficient for mdb-lookup-cs3157, mdb-add-cs3157 needs j-hui's privileges to add entries to the database.

Notice that instead of 'rwx', mdb-add-cs3157's owner permissions are 'rws'; the 's' here indicates that the setuid bit is also set, meaning when other users execute mdb-add-cs3157, they will do so as j-hui, granting them sufficient privileges to modify mdb-cs3157. Since mdb-add-cs3157 is carefully written to only open the mdb-cs3157 file, a malicious user cannot use mdb-add-cs3157's setuid bit to misuse j-hui's privileges.

The setgid bit works similarly for executable files, except it allows the user to run as though they were a member of the file's group. When the setgid is set for directories, it causes files and subdirectories created in that directory to inherit its group ownership, which is convenient for groups of collaborators who are all creating files in that directory.

The sticky bit is only useful for directories; it prevents users with write and execute permissions from modifying or deleting directory contents that they do not own. This bit is primarily useful for shared directories like /tmp, which anyone can add files to; the sticky bit, indicated by the 't' in the directory permissions, prevents users from tampering with each other's files in /tmp:

```
$ ls -d -l /tmp         # -d prevents ls from listing /tmp's contents
drwxrwxrwt 56 root root 28672 Oct 31 16:52 /tmp
```

File creation permissions and umask (optional)

It's very easy to accidentally grant too many permissions to newly created files; for example, in C, files created with fopen() are created with 666 permissions by default. At first glance, this seems wildly insecure: other users would be able to read and write all files created with fopen()!

To prevent lapses in security, UNIX applies a "umask" to newly created files. A umask is a triple of permissions used as a bitmask on the permissions of these files. For example, a umask of 022 ensures that files will be created without group and world write permissions; a umask of 077 ensures that files will start inaccessible by group members and other users.

You can get and set your umask using your shell's umask command (which invokes the umask() system call):

```
$ umask                 # current umask (note: leading 0 is meaningless)
0077

$ touch file1           # create a new file with umask 077

$ umask 022             # set umask to 022

$ touch file2           # create a new file with umask 022

$ umask 000             # set umask to 000 (very insecure!)

$ touch file3           # create a new file with umask 000

$ ls -l                 # show file permissions
-rw------- 1 jzh2106 student 0 Oct 31 17:41 file1
-rw-r--r-- 1 jzh2106 student 0 Oct 31 17:42 file2
-rw-rw-rw- 1 jzh2106 student 0 Oct 31 17:45 file3
```

By default, the touch command tries to create new files with 666 permissions. With a umask 077, the resulting permission is 666 & ~077 = 600; with umask 022, 666 & ~022 = 600; with umask 000, 666 & ~000 = 666.

Note that umask only governs the permissions that a file is created with; the permissions can always be set to something else afterwards, e.g., using chmod.

Everything is a file (optional)

A noteworthy feature of UNIX systems is that "everything is a file": aside from regular files and directories that we expect to reside in a file system, many things not traditionally considered files also appear to user programs as files (i.e., they are referred to using file descriptor numbers), including:

- standard streams (stdin, stdout, stderr)
- pipes (both named pipes created using mkfifo and anonymous pipes created using the pipe() system call)
- processes (discussed next lecture)
- hardware devices (e.g., hard drives, displays, printers, hardware timers)
- kernel options (e.g., low-level networking behavior, logging)

The latter three can be found in the UNIX file hierarchy within the following special directories:

- /proc: contains per-process metadata, statistics, and configuration options
- /dev: contains files representing physical devices
- /sys: exposes a file system-like to configure the kernel at runtime

Because "everything is (just) a file," we can interact with almost all aspects of the system through the same interface we use to read and write regular files.