Lecture 16 - UNIX

(last updated: 2023-11-12)

Introduction and history

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:

```
|----------------------------------------------------------------------|
| Applications: ls, vim, gcc, bash, chrome, firefox, mdb-lookup-cs3157 |
|-----------------------------------------------------------+          |
| Library functions: printf(), strcpy(), malloc(), fwrite() |          |
|----------------------------------------------------------------------|
| System calls: write(), fork(), execl()                               |
|----------------------------------------------------------------------|
| OS kernel: Linux, XNU (of macOS), Windows NT                         |
|----------------------------------------------------------------------|
| Hardware: processor (CPU), memory (RAM), disk, GPU, keyboard         |
|----------------------------------------------------------------------|
```

With the help of CPU hardware features such as privileged operations and periodic timer interrupts, 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 exclusive use of large linear memory address space
- hardware devices from different vendors respond to a uniform set of commands

The kernel is only one part of an OS, which also consists of libraries and various application programs. Windows and macOS come bundled with many application programs. Strictly speaking, "Linux" is the kernel of the GNU/Linux Operating System, where the GNU part refers to the rest -- libraries and application programs -- that are actually a much bigger part of a usable operating system.

In 1970, Ken Thompson and Dennis Ritchie at AT&T Bell Labs developed the UNIX operating system, along with the C programming language for the purpose of writing UNIX. Since then, many descendants of UNIX have come and gone from AT&T, and other companies and universities who licensed UNIX and developed their own versions. They include AT&T's System V, UC Berkeley's BSD and its many descendants, Sun's Solaris, IBM's AIX, HP's HP-UX, and Microsoft's Xenix.

Five decades later, there are three descendants of UNIX that are still going strong. GNU/Linux, which combines the Linux kernel created by Linus Torvalds in 1991 with an extensive collection of free software from GNU Project, is the dominant OS for servers today. macOS, the OS for Apple's Mac computers, is the second most widely used desktop OS after Microsoft Windows. Lastly, FreeBSD is the most widely used BSD descendant. While its popularity is tiny compared to Linux or macOS, FreeBSD is significant because it forms the basis of other OSes. First and foremost, FreeBSD is a part of macOS and the reason why macOS is considered to be a UNIX system. FreeBSD also forms the basis of the OSes for some prominent hardware devices including Internet routers and game consoles. Even Microsoft used FreeBSD's networking code in its Windows OS.

The UNIX-like interface common to these operating systems is codified in a standard called "POSIX", which stands for Portable Operating System Interface. We use the name "UNIX" to broadly refer to UNIX-like systems, i.e., those that conform to the POSIX standard.

Users and groups

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(abc1234) gid=1008(student) groups=1008(student)
```

The output shows that the username is abc1234, UID is 1488, and the user is part of the student group.

UNIX systems also have a superuser whose username is root and whose UID is 0.

File permissions

Each file is associated with a user, a group, and a set of permissions. The file’s associated user is also called the file’s "owner", the group is called the file’s "group owner", and the permissions are called the file’s "mode".

You can see this metadata using ls -l:

```
    $ ls -l
    -rw------- 1 abc1234 student  12 Oct 31 01:34 private.txt
    -rw-r--r-- 1 abc1234 student  17 Oct 31 00:54 public.txt
    -rwxr-x--- 1 abc1234 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 abc1234

- public.txt can be be read by anyone, but can only be modified by abc1234

- script.sh can be read, modified, executed by abc1234; can be read and executed by anyone in the student group; but others have no access

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    # Change script.sh's mode to 755

    $ chown root public.txt   # Change public.txt's owner to root
```

We passed a mode in octal notation to the chmod command above. We can also pass modes in symbolic notation:

```
    $ chmod go-x student.sh
```

"go-x" tells chmod to turn off the execute permission for group and other. If the permission was rwxr-xr-x (755) to begin with, it will become rwxr--r-- (744). The following command will turn the x permission back on only for group:

```
    $ chmod g+x student.sh    # The mode will become rwxr-xr-- (754)
```

And the following will turn on the x permission for "all" (i.e., user, group, and other):

```
    $ chmod a+x student.sh    # The mode will become rwxr-xr-x (755)
```

Be careful when using these commands! 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.

Directory permissions

In UNIX, directories have permissions too:

```
    $ ls -la
    drwxr-xr-x  2 abc1234 student   5 Oct 31 01:34 .
    drwx--x--- 21 abc1234 student  43 Oct 31 01:34 ..
    -rw-------  1 abc1234 student  12 Oct 31 01:34 private.txt
    -rw-r--r--  1 abc1234 student  17 Oct 31 00:54 public.txt
    -rwxr-x---  1 abc1234 student 211 Oct 31 01:03 script.sh
```

For directories, the permissions are interpreted a little differently:

- The read permission (r) allows the corresponding users to "read" the content of the directory -- i.e., you can run "ls". (But "ls -l" requires the execute permission.)

- The write permission (w), together with the execute permission, allows the corresponding users to modify the content of the directory -- i.e., you can create, delete, or name files. The write permission for directory is meaningless without the execute permission.

- The execute permission (x) is the prerequisite for full access to the items within the directory -- displaying metadata with "ls -l" and modifying the content of the directory, combined with r and w, respectively.

  The execute permission is also needed to enter or get past a directory. For example, in order to run "cd a/b/c/d", one needs the execute permissions for a, b, c, and d.

Shell scripts

A shell script is a text file that contains a list of commands for a shell program to execute one after another. If you want to execute "ls" three times, for example, you can create a file "ls3.sh" that contains the following lines:

```
    #!/bin/bash
    ls
    ls
    ls
```

You can run a shell script by passing it as an argument to the shell program:

```
    $ /bin/bash ls3.sh
```

You will see the ls command run three times. The first line is treated as a comment because it starts with '#'. (The line is actually a bit more special than that; more on this below.)

Most shells not only let you run individual commands, but also offers its own programming language. For example, you can write ls3.sh this way:

```
    #!/bin/bash
    for i in {1..3}
    do
        ls
    done
```

UNIX provides another way to run a shell script. You can run the script itself by typing "./ls3.sh" instead of "/bin/bash ls3.sh". When the kernel is given a program file to execute (through the execv()/execl() system calls as we studied), it will look at the first two bytes of the program file to determine the format of the executable. If the first two bytes are '#' and '!', it is recognized as the "interpreter directive". In that case, UNIX will execute the program that the interpreter directive specifies instead -- "/bin/bash" in this case -- and pass the original script file (ls3.sh) as a command line argument. The end result is the same as running "/bin/bash ls3.sh". The interpreter directive is commonly referred to as the "shebang" line.

Before this can happen, the shell script must be made executable using the chmod command:

```
    chmod +x ls3.sh
```

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 /home/jae/cs3157-pub/bin

    $ ls -l *-cs3157
    -rwsr-xr-x 1 jae jae   17416 Oct 18 15:18 mdb-add-cs3157
    -rw-r--r-- 1 jae jae   38200 Oct 31 14:44 mdb-cs3157
    -rwxr-xr-x 1 jae jae     158 Oct 18 15:18 mdb-lookup-cs3157
```

mdb-cs3157 has 644 permissions, so while anyone can read it, only the owner, jae, can modify it. Recall that mdb-cs3157 was the class database for lab4. Everyone in class was able to add entries to this class-wide database by running mdb-add-cs3157. Normally, when a user abc1234 runs a program, say mdb-lookup-cs3157, the program carries the identity of the user who is running it, and the UID of the user running the program will be subject to the permission checks. So when the user abc1234 runs mdb-lookup-cs3157 to read mdb-cs3157, the user is able to see the content of the database because mdb-cs3157 allows read access to everyone. But if abc1234 were to open mdb-cs3157 using vim, for example, abc1234 won't be able to modify the database file because mdb-cs3157 allows write access only to its owner, jae.

How, then, was every student in class able to add records into mdb-cs3157 using mdb-add-cs3157? 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 jae, 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 jae'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:

```
    $ umask 077             # set umask to 077

    $ 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
    -rw------- 1 abc1234 student 0 Oct 31 17:41 file1
    -rw-r--r-- 1 abc1234 student 0 Oct 31 17:42 file2
    -rw-rw-rw- 1 abc1234 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.