210 Systems Programming
The Linux File System, System Administration: Process and User Management

Fall 2025

Week 5

Overview

- Reminder: File permissions
- File system organization
  - inode
  - Symbolic and Hard links
- Becoming a super user
- Querying and manipulating processes

Reminder: Permission Syntax

- Three permission types: read, write, execute (rwx) for three classes: owner, group, others (ugo) $\Rightarrow$ 9 permission flags:
  - $r_u w_u x_u r_g w_g x_g r_o w_o x_o$
- Examples:
  - user has rwx, group has r, others have r $\Rightarrow$ 111 100 100 or `rwxr--r--`
  - user has rw, group has rw, others have none $\Rightarrow$ 110 110 000 or `rw-rw----`

Reminder: Changing permissions with `chmod`

- Can use the *symbolic* or the *numeric* mode
  - The format of a symbolic mode is `[ugoa...][[-+=][perms...]...]`.
  - Examples:
    - `chmod +x foo`
    - `chmod u+rwx,g+x,o-rwx foo`
  - The numeric mode use octal digits (0-7) to represent the 3-bit permissions for each ugo category.
  - Examples:
    - `chmod 754 foo`
    - `chmod 600 foo`
- Only the owner of the file/directory or a superuser can change the permissions of that file/directory.

Changing the owner and the group of a file/directory

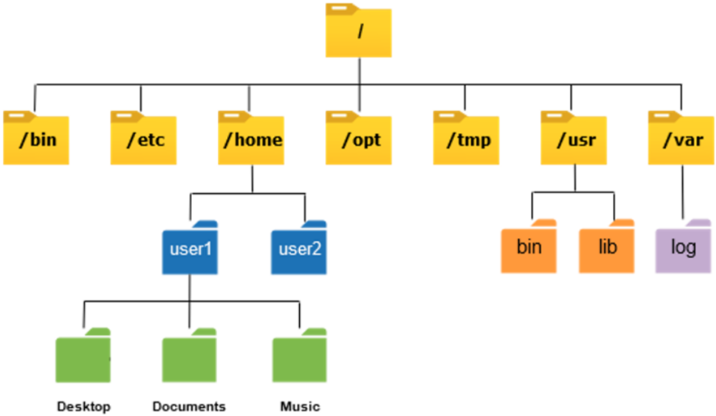
- Only a privileged user can do this. (will learn how to elevate to a *superuser* in the next lecture)

`chown [OPTION]... [OWNER][:[GROUP]] FILE...`

- **Ch**ange **own**er and group to `OWNER` and `GROUP` for each `FILE`
- Can use the `-R` option to recursively change the ownership info for the entire content of a directory.

Reminder: Linux Directory Structure

- Files are organized as a set of nested directories



Let's inspect a directory's contents

```
-rw-rw-r--  1 tolgacan tolgacan      936 Aug 31  2022 log.cpp
-rwxrwxr-x  1 tolgacan tolgacan    17320 Aug 29  2022 myProgram
-rw-rw-r--  1 tolgacan tolgacan       90 Aug 23  2023 myProgram_backup.cpp
-rw-rw-r--  1 tolgacan tolgacan       90 Aug 29  2022 myProgram.cpp
-rw-rw-r--  1 tolgacan tolgacan        0 Jul  9 13:04 newFile
-rw-rw-r--  1 tolgacan tolgacan      981 Aug 31  2022 orig_log.cpp
drwxrwxr-x  6 tolgacan tolgacan       11 Sep  9  2022 project0-setup-tolgacan
drwxrwxr-x  7 tolgacan tolgacan       17 Sep 23  2022 project1-stack-tolgacan
```

- Consider the file `newFile` in the above output
  - Where is the file name stored?
    - Its size is 0 bytes, so the file name cannot be stored within the file itself.
  - Similarly, where do we store modification date, owner, group, etc.?
    - This additional information about the file is called its *metadata*.
    - In the Linux file system, the metadata is stored separate from the content of the file.

The Unix/Linux File System

A well-formed file system is composed of three main components:

The file	inode	Directory structure
The actual data *blocks* containing the files primary data	A structure assigned to each file to store its metadata	A table to store file names and inode numbers

Let's look at each of these components in detail and learn how they are all connected.

The File

- A file is a collection of data blocks, on a storage device, containing the data of interest
- For example, if we create a file named `foo` with the following command:
  - `$ echo "Hello World!" > foo`
- The file `foo` is a group of 13 bytes (including the end of line character)
- These blocks do not store the file name `foo`
- This raises a new question:
  - How do we know where these blocks are?

inode

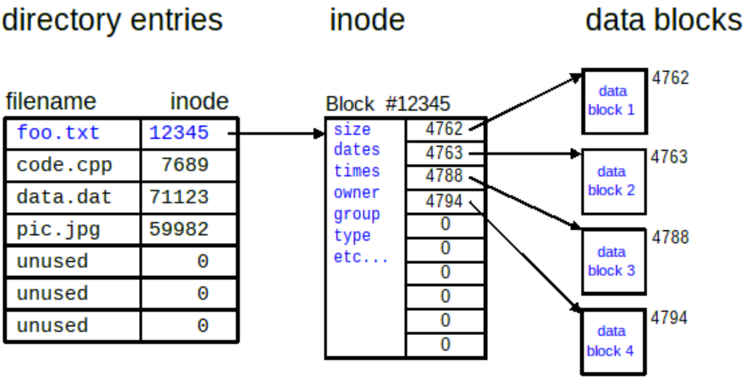
- Each file has an inode to store its metadata and locate the file's contents
  - A regular file's inode contains a pointer to its data blocks
  - A directory's inode contains a pointer to its *directory structure*
- inodes are stored in a table, which is allocated when the filesystem is first created
  - Which means there are a fixed number of inodes in a file system. How is this number determined?
    - In Linux `ext4`, a default ratio of 1 inode per 16KBs of total capacity is used.
    - Could this be a problem?
- Use the `ls` option `-i` to get the inode number information of files/directories.
- Use `df -i` to get usage information on inodes in a file system.

Directory Structure

- Two pieces of metadata are not stored in the inode:
  - Filename and inode number
- The directory structure represents the contents of each directory as a table of the contained file/directory names and their corresponding inode numbers
  - To inspect a directory (instead of listing its contents), use `ls -d`

Filename	inode #
a.txt	33534535
b.txt	34545221
dir1	45455455

Three main components together:



From: `https://azrael.digipen.edu/~mmead/www/Courses/CS180/FileSystems-1.html`

Linking: Symbolic Links

- Symbolic links:
  - Same as shortcuts on Windows.
  - They are small (independent and separate) files that contain the *path* information of a target file.
  - When listed inside a directory, the `l` flag at the very first column of a long format listing identifies them as *links*.
  - Also, the target file path is listed along with the symbolic link's file name.
  - Deleting the target file does not remove the symbolic links pointing to that file
  - Create symbolic links using: `ln -s [target file] [link name]`
  - Symbolic links are also called *soft* links

Linking: Hard Links

- Hard links:
  - They are independent entries in the same or different directory structures that point to the **same inode**. In other words, they are aliases to the same single file.
  - You are already familiar with aliases to a directory name: `.`, `..`, and `dirname`,
  - The system keeps track of number of hard links to a file, so that the data blocks are deleted from the disk only after the last remaining link to that inode is deleted.
  - Create hard links using: `ln [target file] [hard link name]`
  - Experiment with hard links and symbolic links and inspect their inode numbers with `ls -i`

The Superuser

- Administers the OS.
  - Is not affected by file permissions.
    - Can move, delete any file that belongs to anyone.
  - Installs, updates applications.
  - Monitors system, network log files.
- Used to be called the *root* user.
- Today, instead of logging as the root user each time, regular users can be given *superuser* privileges by listing them in a special file named: `sudoers`.

Becoming the *superuser*

- While its use is not advised, on some systems you can log into root with `su`.
- This command can be very dangerous, as you will retain the superuser privileges until you exit the *root* shell with the `exit` command.
- Most distributions (i.e. Ubuntu) disable logging into root with `su` altogether.
- Instead, using the `sudo` command to execute **a** command **as** the superuser is recommended.

sudo

`sudo [OPTIONS] [CMD]`

Super **u**ser **do**. Run *CMD* as *root*.

- Allows execution of commands with root's privileges, but does not log you in as root.
- Can also be used to execute commands as a different user with the `-u USER` option.

- sudo is generally safe, but we should always be cautious using it.
- Only users in the `/etc/sudoers` may run this command.
- **Do not try it on isengard**. It will be considered as an attack by the system admins.

Installing Programs

- Generally, package managers are used to install programs in an operating systems.
  - On Ubuntu, Debian, and Kali Linux, **apt** is the main package manager.
    - Use, `sudo apt-get update` to update the repository information.
    - Use, `sudo apt-get install package_name` to install a package.
  - On Arch-based distributions, including Arch and Manjaro, **pacman** is used as the package manager.

See: `https://en.wikipedia.org/wiki/List_of_software_package_management_systems` for more information.

User management

- `who`
  - Prints information about all users who are currently logged in
  - The command `w` is similar and prints more information
- `id [USERNAME]`
  - Prints the user and group information for the specified [USERNAME]
- Users can be added using the `useradd` command and can be removed using the `userdel` command
  - Must have superuser privileges

What is a *process*?

- A *process* is a *running* copy of a program along with its allocated resources such as, memory, file handles, network sockets, etc.
- There can be multiple different processes of a program.
  - Example: vim is a program installed on *isengard*. When different users edit their own files simultaneously, there will be multiple and separate *vim* processes with their own process ids, memory footprints, etc.
- Processes are dynamic, i.e., there are various events in their lifetimes:
  - Created, Paused, Run, Killed, Completed

How to get a list of current processes?

- Use the `ps` command to get a list of processes.
  - `ps -ef` will list all the processes in full-format.
  - Check `man ps` for different options
  - Demo on *isengard*

Other tools for process monitoring

- `top` and `htop` are two interactive programs for monitoring processes and system resources such as CPU and memory.
- Try them out on isengard (or on your own Linux)

Foreground and background processes

- When you run a program in your bash shell, by default, it is run as a *foreground* process, meaning that your interaction with the shell is suspended until that program finishes it execution.
- Most programs finish in an instant, so this is not a big deal. However, imagine you have a program that runs for 1-2 hours or maybe for days.
  - You would want to run it as a *background* process to keep it running in the background why you get back to your shell prompt instantly.
  - Use `<program> &` to run a program as a background process.

Sending pause/kill signals to a foreground process

- A running foreground process can be terminated by sending it a SIGINT signal with Ctrl-C.
  - We will learn more about signals when we talk about processes in Module 3.
- A running foreground process can be paused by sending it a SIGTSTP signal with Ctrl-Z.
  - The most recently paused processed can resume execution in the foreground with a `fg` command and in the background with a `bg` command.

Using the `kill` command to terminate/pause processes

- If you want to terminate or pause a process you own that is running in the background, you can use the `kill` command, to send it SIGTERM or SIGTSTP signals. Examples:
  - `kill -TSTP pid` to pause (sends SIGTSTP signal)
  - `kill -CONT pid` to resume execution
  - `kill pid` to terminate (sends SIGTERM signal)
- Alternatively, you can use the `pkill` command to send signals processes based on their name instead of their process ids.

Why is there a *parent* process?

- When you list the processes using `ps -ef`, you see that each process has a PPID (Parent Process ID) listed along with other information.
- What does this tell you about how processes are created?

Process creation

- We will talk about this in more detail in Module 3; but, shortly, each process is created by a parent process.
    - The parent process uses the `fork()` system call to create a clone of itself.
    - The clone is identical to the current process except its process id and parent process id
    - The new process then loads and executes the desired program

`fork()` example

- When `fork()` is successful, it returns two different outputs to the child and the parent process.
  - Child PID is returned to the parent
  - 0 is returned to the child
- When `fork()` fails, no child process is created and -1 is returned to the parent.

fork() example

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
        printf("hello world (pid:%d)\n", (int) getpid());
        int rc = fork();

        if (rc < 0) {
                fprintf(stderr, "fork failed\n");
                exit(1);
        } else if (rc == 0) { // child (new process)
                printf("hello, I am child of %d (my pid:%d)\n",(int) getppid(), (int) getpid());
        } else { // parent goes down this path (original process)
                printf("hello, I am parent of %d (my pid:%d)\n",rc, (int) getpid());
        }
        return 0;
}
```