Kathmandu University
Department of Computer Science and Engineering
Dhulikhel, Kavre

A Lab Report 4
on

”Understanding File System Permissions”

[Code No: COMP 307]

Operating Systems

Submitted by:

Salina Nakarmi (34)

Submitted to:

Ms. Rabina Shrestha

Department of Computer Science and Engineering

Submission Date: January 26, 2026

Contents

Chapter 1. Introduction

File system permissions are a fundamental security mechanism in Unix-like operating systems, including Linux. They control access to files and directories by defining who can read, write, or execute them. Understanding and properly managing file permissions is crucial for system security, data protection, and multi-user environments.
This lab report explores the practical aspects of viewing, interpreting, and modifying file system permissions using common Linux commands. Through hands-on exercises, we examine permission strings, numeric permission modes, and the chmod command to control file and directory access rights.

1.1 Objectives

The main objectives of this lab are:

- To understand how to view and interpret file system permissions using the `ls` command
- To learn the structure and meaning of permission strings for owners, groups, and others
- To compare permission differences between files and directories
- To practice changing permissions using both symbolic and octal notation with `chmod`
- To understand the security implications of different permission settings

Chapter 2. Theory

2.1 File System Permissions Overview

In Linux and Unix-like systems, every file and directory has associated permissions that determine who can access it and what operations they can perform. These permissions are divided into three categories:

- **Owner (User):** The user who owns the file
- **Group:** Users who are members of the file's group
- **Others:** All other users on the system

2.2 Permission Types

For each category, three types of permissions can be granted:

- **Read (r):** Permission to read the contents of a file or list the contents of a directory
- **Write (w):** Permission to modify a file or create/delete files within a directory
- **Execute (x):** Permission to execute a file as a program or enter a directory

2.3 Permission String Format

When using `ls -l`, permissions are displayed as a 10-character string. For example: `-rw-r--r--`

- **Character 1:** File type (- for regular file, d for directory, l for symbolic link)
- **Characters 2-4:** Owner permissions (rwx)
- **Characters 5-7:** Group permissions (rwx)
- **Characters 8-10:** Others permissions (rwx)

2.4 Octal Notation

Permissions can also be represented numerically using octal (base-8) notation:

- Read (r) = 4
- Write (w) = 2
- Execute (x) = 1

Permissions are calculated by adding these values. For example:

- rwx = 4 + 2 + 1 = 7
- rw- = 4 + 2 + 0 = 6
- r– = 4 + 0 + 0 = 4
- — = 0

A complete permission set like `-rw-r--r--` translates to 644 in octal notation.

2.5 Directory Permissions

Directory permissions work differently than file permissions:

- **Read (r):** Allows listing directory contents
- **Write (w):** Allows creating, deleting, or renaming files within the directory
- **Execute (x):** Allows entering the directory (required to access files within it)

The execute permission on directories (`x`) is essential—without it, users cannot access the directory even if they have read permissions.

Chapter 3. Procedure and Results

3.1 Viewing and Interpreting Permissions

3.1.1 Exercise 1: Creating and Examining a File

A file called `notes.txt` was created in the home directory using the `touch` command, and its permissions were examined.
Commands executed:

```
touch note.txt
ls -l note.txt
```

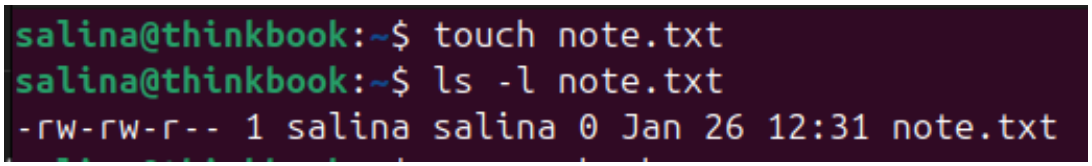

Figure 3.1: Creating note.txt and viewing its default permissions

Observations:
The default permission string displayed was `-rw-rw-r--`, which breaks down as:

- **-**: Regular file (not a directory or link)
- **rw-**: Owner (salina) has read and write permissions
- **rw-**: Group (salina) has read and write permissions
- **r–**: Others have read-only permission

This default permission (664 in octal) allows both the owner and group members to modify the file while restricting other users to read-only access. The file is owned by user ”salina” and belongs to group ”salina”.

3.1.2 Exercise 2: Creating and Examining a Directory

A directory called `project` was created and compared with the file permissions.
Commands executed:

```
mkdir project
ls -ld project
```

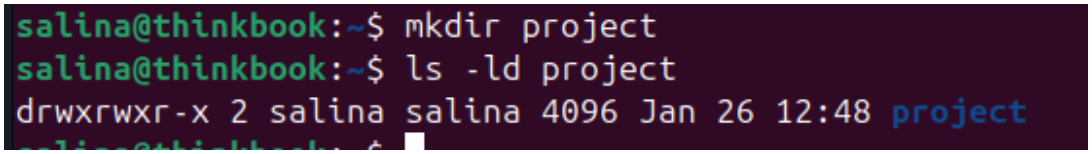

Figure 3.2: Creating project directory and comparing permissions with note.txt

Observations:
The directory displayed permissions of `drwxrwxr-x`, which breaks down as:

- **d**: Directory type indicator
- **rwx**: Owner(salina) can read, write, and execute (enter) the directory
- **rwx**: Group(salina) can read, write, and execute (enter) the directory
- **r-x**: Others can read and execute but not write

Key Difference: Directories show `rwx` for execute permissions, which is essential for navigating into the directory. Without the execute bit, users cannot enter the directory even if they have read permissions to list its contents.

3.2 Using chmod (Change Permissions)

3.2.1 Exercise 3: Changing File Permissions with Symbolic Notation

Permissions on `notes.txt` were modified using symbolic notation to give only the owner full permissions while removing all permissions for group and others.
Commands executed:

```
chmod u+rwx,g-rw,o-rwx note.txt
ls -l note.txt
```

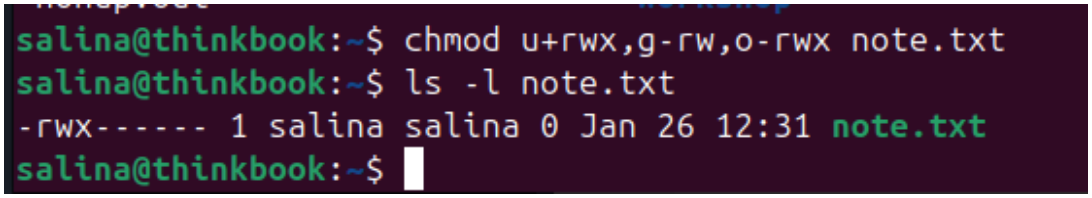

Figure 3.3: Changing permissions on notes.txt using symbolic notation

Result: The permission string changed to `-rwx------`, indicating:

- Owner: read, write, and execute (rwx)
- Group: no permissions (—)
- Others: no permissions (—)

This corresponds to octal notation 700.

3.2.2 Exercise 3b: Using Octal Notation

The same result was achieved using octal notation with the `chmod 700` command.
Commands executed:

```
chmod 700 notes.txt
ls -l notes.txt
```

Explanation of 700:

- **7** (owner): 4(r) + 2(w) + 1(x) = 7 (rwx)
- **0** (group): 0 (—)
- **0** (others): 0 (—)

Octal notation is more concise and efficient for setting permissions compared to symbolic notation, especially when setting complete permission sets.

3.2.3 Exercise 4: Changing Directory Permissions

The project directory permissions were modified to allow the group read and execute access while removing all permissions for others.
Commands executed:

```
chmod 750 project
ls -ld project
```

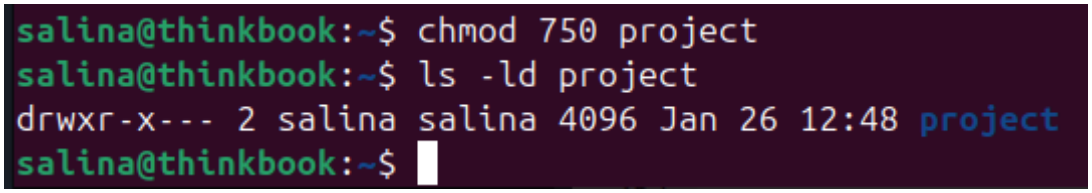

Figure 3.4: Changing permissions on project directory

Result: The permission string became `drwxr-x---`, meaning:

- Owner: full access (rwx) = 7
- Group: read and execute (r-x) = 5
- Others: no access (—) = 0

This setting allows group members to navigate into the directory and view its contents but prevents them from creating or deleting files. Others are completely restricted from accessing the directory.

Chapter 4. Short Answer Questions

4.1 Question 1: Why is it not recommended to use chmod 777 on important files or directories?

Using `chmod 777` grants full permissions (read, write, and execute) to everyone—the owner, group, and all other users on the system. This creates several serious security risks:

- **Unauthorized Modification:** Any user can modify or delete important files, leading to potential data loss or corruption
- **Security Vulnerabilities:** Malicious users or compromised accounts can alter system files or inject malicious code
- **Accidental Damage:** Even well-intentioned users might accidentally modify or delete critical files
- **Compliance Issues:** Many security standards and regulations prohibit such permissive settings
- **Malware Execution:** If applied to directories, malware could be placed and executed by any user

The principle of least privilege dictates that users should only have the minimum permissions necessary to perform their tasks. For important files, typically only the owner (or specific group members) should have write access, while others should have read-only access or no access at all.

4.2 Question 2: When would changing the group of a file (with chgrp) be more appropriate than changing its owner (with chown)?

Changing the group ownership with `chgrp` is more appropriate in several scenarios:
Collaborative Projects: When multiple users need to work on shared files, changing the group allows all team members (who are part of that group) to access and modify files without transferring individual ownership. This maintains accountability while enabling collaboration.
Department or Team Resources: In organizational settings, files often belong to departments rather than individuals. For example, documentation for the engineering team should be owned by the ”engineering” group so all members can contribute.
Preserving Original Ownership: When you want to maintain the record of who created a file (the owner) but need to grant access to additional users, changing the group is more appropriate than transferring ownership.
Security and Permissions Management: Groups provide a more flexible and scalable way to manage permissions. Instead of using `chown` to transfer ownership each time someone new needs access, you can add users to the appropriate group.

Administrative Control: In many systems, only root can change file ownership with `chown`, but regular users can often change groups (if they belong to both the current and target group). This makes `chgrp` more practical for everyday permission management.
Example Scenario: If a developer leaves a project but their code needs to be maintained by the team, using `chgrp` to assign the files to the development team's group is better than changing ownership. This preserves the creation history while ensuring continuity of access.

Chapter 5. Conclusion

This lab provided comprehensive hands-on experience with Linux file system permissions, demonstrating both theoretical concepts and practical applications. Through the exercises, we learned how to view, interpret, and modify permissions using both symbolic and octal notation.
Key Learnings:

- Permission strings provide a clear visual representation of access rights for owner, group, and others
- Directory permissions function differently than file permissions, with the execute bit being essential for directory access
- The `chmod` command offers flexibility through both symbolic notation (u+rwx) and octal notation (700)
- Octal notation is more efficient for setting complete permission sets in a single command
- Proper permission management is crucial for system security and preventing unauthorized access

Practical Implications:
Understanding file permissions is essential for maintaining secure systems, especially in multi-user environments. The principle of least privilege should guide permission settings—granting only the minimum necessary access to accomplish tasks. Overly permissive settings like 777 should be avoided on important files and directories to prevent security vulnerabilities.
The ability to use both symbolic and octal notation provides flexibility in different scenarios. Symbolic notation is intuitive for making incremental changes, while octal notation excels at setting complete permission configurations quickly.
Future Applications:
These skills form the foundation for more advanced topics in system administration, including special permissions (setuid, setgid, sticky bit), access control lists (ACLs), and security hardening practices. Proper permission management is a critical component of system security, backup procedures, and collaborative development environments.