Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION)

BASIC & ADVANCED LINUX COMMANDS

CHAPTER 1: INTRODUCTION TO LINUX COMMANDS

Linux commands are the foundation of **system administration, scripting, and automation** in Linux-based environments. Whether you are a beginner or an experienced user, knowing **basic and advanced Linux commands** is essential for managing files, processes, and system configurations efficiently. Linux offers a powerful **command-line interface (CLI)** where users can execute commands to perform various tasks such as **file manipulation, searching, text processing, system monitoring, and archiving**.

The **Linux shell** (Bash, Zsh, Fish) interprets commands and executes them in the system. Unlike graphical user interfaces (GUIs), the CLI allows for **automation, batch processing, and fine-grained control** over the system. Learning Linux commands not only improves efficiency but also provides deeper insight into the **inner workings of the operating system**.

This study material covers both **basic and advanced Linux commands**, including **file handling, searching, text processing, and archiving tools** such as ls, cat, grep, find, awk, sed, and tar. Through detailed examples, exercises, and a real-world case study, this chapter will help users master these commands and apply them effectively in **system administration, development, and troubleshooting**.

CHAPTER 2: BASIC LINUX COMMANDS

Basic Linux commands are essential for **navigating the file system, managing directories, and handling files**. These commands allow users to interact with the system efficiently and form the foundation for more complex operations.

File and Directory Management Commands

1. Listing Files and Directories (ls)

The ls command displays the contents of a directory. It provides various options to filter and format the output.

Syntax:

ls [options] [directory]

Common Options:

- ls – Lists files in the current directory.
- ls -l – Displays files in a long format with permissions, owner, and timestamps.
- ls -a – Shows hidden files (files starting with .).
- ls -lh – Human-readable format with file sizes.

Example:

ls -l /home/user/Documents

This command lists all files in the /home/user/Documents directory along with details such as size and permissions.

2. Displaying File Contents (cat)

The cat command is used to view the contents of a file, combine multiple files, and create new files.

Syntax:

cat [filename]

Common Uses:

- cat file.txt – Displays the contents of file.txt.
- cat file1.txt file2.txt > merged.txt – Combines two files into one.

Example:

cat /etc/passwd

This command displays system user account details stored in /etc/passwd.

Chapter 3: Searching & Filtering Commands

Searching and filtering commands allow users to **locate files, search for patterns, and process large data efficiently**.

1. Searching for Files (find)

The find command searches for files and directories based on various attributes such as name, type, size, and modification time.

Syntax:

find [directory] [options] [search term]

Common Options:

- find /home -name "*.txt" – Searches for all .txt files in /home.
- find /var/log -size +10M – Finds files larger than 10MB in /var/log.

Example:

find / -type f -name "config.conf"

This command searches for a file named config.conf across the entire system.

---

2. Searching Inside Files (grep)

The grep command is used to search for specific patterns within files. It is a powerful tool for text processing and log analysis.

Syntax:

grep [options] "pattern" [filename]

Common Uses:

- grep "error" /var/log/syslog – Finds lines containing "error" in system logs.
- grep -i "warning" logfile.log – Case-insensitive search for "warning".

Example:

grep -r "password" /etc/

This searches for the term "password" in all files within /etc/.

Chapter 4: Advanced Text Processing Commands

Text processing commands help manipulate, format, and extract data from files.

1. Stream Editor (sed)

The sed command is used to **search, replace, delete, and modify text** in files.

Syntax:

sed 's/old-text/new-text/g' filename

Common Uses:

- sed 's/Linux/UNIX/g' file.txt – Replaces "Linux" with "UNIX" in file.txt.
- sed -i 's/error/failure/g' log.txt – Modifies the file in place.

Example:

echo "Hello Linux" | sed 's/Linux/World/'

This outputs Hello World after replacing "Linux" with "World".

---

2. Pattern Scanning & Processing (awk)

The awk command processes structured text data, such as logs and CSV files.

Syntax:

awk '{ print $1 }' filename

Common Uses:

- awk '{print $1, $3}' employees.csv – Prints the first and third column of a CSV file.

- awk '/error/ {print $0}' syslog – Prints all lines containing "error".

Example:

df -h | awk '{print $1, $5}'

This extracts the disk partition name and usage percentage.

CHAPTER 5: FILE ARCHIVING AND COMPRESSION COMMANDS

1. Archiving Files (tar)

The tar command is used to create **compressed archives** for backup and distribution.

Syntax:

tar [options] archive_name.tar file_or_directory

Common Uses:

- tar -cvf backup.tar /home/user/ – Creates an archive of /home/user/.
- tar -xvf backup.tar – Extracts an archive.

Example:

tar -czvf logs.tar.gz /var/log/

This creates a compressed .tar.gz archive of /var/log/.

CHAPTER 6: CASE STUDY – AUTOMATING LOG ANALYSIS WITH LINUX COMMANDS

Scenario:

A system administrator needs to analyze **web server logs** to find failed login attempts.

Solution:

1. Use grep to find failed login attempts:
2. grep "failed" /var/log/auth.log
3. Extract usernames from logs using awk:
4. grep "failed" /var/log/auth.log | awk '{print $10}'
5. Store results in an archive:
6. grep "failed" /var/log/auth.log | awk '{print $10}' > failed_attempts.txt
7. tar -czvf failed_attempts.tar.gz failed_attempts.txt

Impact:

This automation **saves time and improves security monitoring** by identifying unauthorized access attempts quickly.

Chapter 7: Exercise

1. **List three ways to use the ls command with examples.**
2. **Write a command to find all .log files larger than 5MB in /var/log/.**
3. **Use grep to extract lines containing "error" from /var/log/syslog.**

4. **Write an awk command to extract usernames from /etc/passwd.**
5. **Create a tar archive of your home directory and extract it.**

CONCLUSION

Mastering **basic and advanced Linux commands** enhances **productivity, troubleshooting skills, and system administration capabilities**. Whether managing files, processing text, or automating tasks, Linux commands provide unmatched flexibility and power.

USER & GROUP MANAGEMENT

CHAPTER 1: INTRODUCTION TO USER AND GROUP MANAGEMENT IN LINUX

User and group management is one of the most critical aspects of Linux system administration. Linux is a **multi-user operating system**, meaning multiple users can access and operate the system simultaneously. To maintain **security, access control, and system integrity**, Linux provides robust **user and group management tools**.

Every user in Linux has a **unique user ID (UID)** and belongs to at least one group, which helps define **permissions and privileges**. Similarly, groups allow administrators to **assign collective permissions** to multiple users, ensuring efficient access management.

User and group management is essential for:

- **System security** – Restricting unauthorized access.
- **Resource allocation** – Assigning disk space and system privileges.
- **Collaboration** – Allowing specific user groups to share resources.
- **Process control** – Managing background jobs and system operations efficiently.

This chapter explores user and group management in Linux, covering **user creation, deletion, modification, group management, and permission settings**. We will also discuss **real-world applications, examples, case studies, and exercises** to enhance understanding.

CHAPTER 2: UNDERSTANDING USERS IN LINUX

Linux classifies users into three main categories:

1. **Root User (Superuser)**
   - The most privileged user (UID 0).
   - Has unrestricted access to all system files and commands.
   - Can create, modify, and delete any user or file.
2. **Regular Users**
   - Created by system administrators.
   - Have restricted access to certain files and directories.
   - Can execute system commands with **sudo** (if granted permission).
3. **System Users**
   - Created automatically during system installation.
   - Used for system services (e.g., www-data for web servers).
   - Usually have **non-login shells** (/usr/sbin/nologin).

Example: Viewing Users on a System

To list all users in Linux, check the /etc/passwd file:

cat /etc/passwd

Each line represents a user and contains fields like **username, UID, home directory, and shell**.

Chapter 3: Creating and Managing Users

1. Creating a New User (useradd)

The useradd command creates a new user account.

Syntax:

sudo useradd [options] username

Common options:

- -m – Creates a home directory (/home/username).
- -s – Sets the login shell (/bin/bash).
- -G – Assigns the user to a group.

Example:

sudo useradd -m -s /bin/bash -G developers john

This command creates a user **john**, assigns them a **home directory**, a **bash shell**, and adds them to the developers group.

---

2. Setting User Passwords (passwd)

After creating a user, assign a password:

sudo passwd john

The system prompts for a password, securing the new account.

---

3. Modifying User Accounts (usermod)

To modify user properties like username, shell, or groups, use:

sudo usermod -l newname oldname # Rename user

sudo usermod -s /bin/zsh john   # Change shell to Zsh

sudo usermod -aG sudo john   # Add user to sudoers

---

4. Deleting a User (userdel)

To remove a user:

sudo userdel john

To remove the user along with their home directory:

sudo userdel -r john

CHAPTER 4: MANAGING GROUPS IN LINUX

Groups in Linux allow administrators to assign permissions **collectively** instead of managing individual users.

1. Creating a Group (groupadd)

To create a new group:

sudo groupadd developers

2. Adding a User to a Group (usermod -aG)

To add an existing user to a group:

sudo usermod -aG developers john

The -aG flag ensures the user is **added without removing existing groups**.

3. Viewing Group Membership (groups)

To check which groups a user belongs to:

groups john

4. Removing a User from a Group (gpasswd -d)

To remove a user from a group:

sudo gpasswd -d john developers

5. Deleting a Group (groupdel)

To remove a group:

sudo groupdel developers

CHAPTER 5: USER & GROUP PERMISSIONS

Permissions control **who can read, write, or execute files** in Linux. They are assigned at three levels:

1. **Owner (User)**
2. **Group**
3. **Others (Everyone else)**

1. Checking File Permissions (ls -l)

ls -l file.txt

Output example:

-rw-r--r-- 1 john developers 1234 Jan 1 10:00 file.txt

Here:

- rw- (Owner: **read/write**)
- r-- (Group: **read-only**)
- r-- (Others: **read-only**)

2. Changing File Permissions (chmod)

chmod 755 script.sh

This grants:

- **Owner** full access (rwx).
- **Group & Others** only read/execute (r-x).

3. Changing File Ownership (chown)

sudo chown john:developers file.txt

This assigns **john** as the owner and **developers** as the group.

Chapter 6: Case Study – Managing Users in an IT Organization

Scenario:

An IT company requires:

- A **developer team** with access to /dev-projects.
- A **sysadmin team** with full access to /etc/admin.

Solution:

1. **Create Groups**:

2. sudo groupadd developers
3. sudo groupadd sysadmins
4. **Create Users & Assign Groups**:
5. sudo useradd -m -G developers alice
6. sudo useradd -m -G sysadmins bob
7. **Set Folder Permissions**:
8. sudo chown :developers /dev-projects
9. sudo chmod 770 /dev-projects

Outcome:

- **Developers** can access **/dev-projects** but not **/etc/admin**.
- **Sysadmins** have full control over **/etc/admin**.
- **Security & Collaboration are properly managed**.

CHAPTER 7: EXERCISE

1. **Create a new user named testuser with a home directory and Bash shell.**
2. **Change the password for testuser.**
3. **Create a group named research and add testuser to it.**
4. **Change ownership of /var/logs to research group.**
5. **Set permissions on /var/logs so that only group members can modify files.**

CONCLUSION

User and group management is a **core Linux administration skill**. Proper user and group configurations ensure **security, collaboration, and system efficiency**. By mastering **user creation, group management, permissions, and ownership**, users can effectively **control access to resources** while maintaining system integrity.

FILE PERMISSIONS AND OWNERSHIP

CHAPTER 1: INTRODUCTION TO FILE PERMISSIONS AND OWNERSHIP IN LINUX

In Linux, **file permissions and ownership** play a crucial role in **controlling access to files and directories**. Every file and directory in Linux is associated with a **user (owner), a group, and permission settings** that define who can **read, write, or execute** them. This ensures system security and proper access control among multiple users.

Since Linux is a **multi-user operating system**, managing file access is essential to prevent **unauthorized modifications and security breaches**. Linux permissions work at three levels:

1. **User (Owner):** The person who created the file.
2. **Group:** A collection of users with shared access.
3. **Others:** All other users on the system.

Each file and directory in Linux follows a **permission model**, represented in a combination of **letters (symbolic mode) or numbers (octal mode)**.

By understanding file permissions and ownership, system administrators can **enforce security policies, protect sensitive data, and optimize collaboration** in a Linux environment.

CHAPTER 2: UNDERSTANDING FILE PERMISSIONS

1. Viewing File Permissions (ls -l)

To check file permissions, use:

ls -l file.txt

Example output:

-rw-r--r-- 1 john developers 1234 Jan 1 10:00 file.txt

This structure represents:

- **-** (File Type: - for file, d for directory).
- **rw-** (Owner: **read, write**).
- **r--** (Group: **read-only**).
- **r--** (Others: **read-only**).

Each permission is represented in **three groups of three characters**:

- **r (Read)** → View file contents.
- **w (Write)** → Modify the file.
- **x (Execute)** → Run the file as a program/script.

For directories:

- **r** → Allows listing contents (ls).
- **w** → Allows creating/deleting files.
- **x** → Allows entering (cd) the directory.

---

2. Changing File Permissions (chmod)

The chmod command modifies file permissions.

Symbolic Mode:

Modify permissions using u (user), g (group), o (others), and a (all).

chmod u+x script.sh # Add execute permission to the owner

chmod g-w report.txt # Remove write permission from group

chmod o+r document.txt # Grant read access to others

Octal Mode (Numeric Representation)

Each permission is assigned a number:

- **r = 4, w = 2, x = 1**.

Example:

chmod 755 script.sh

Explanation:

- **7 (Owner = rwx)**
- **5 (Group = r-x)**
- **5 (Others = r-x)**

Common permissions:

Octal	Permission	Description
777	rwxrwxrwx	Full access to all users (insecure).
755	rwxr-xr-x	Owner has full access, others can read/execute.
644	rw-r--r--	Owner can read/write, others can only read.
600	rw-------	Only the owner can read/write (secure files).

CHAPTER 3: UNDERSTANDING FILE OWNERSHIP

1. Viewing File Ownership (ls -l)

ls -l example.txt

Output example:

-rw-r--r-- 1 alice developers 1234 Jan 1 10:00 example.txt

- **Owner:** alice
- **Group:** developers

---

2. Changing File Ownership (chown)

The chown command changes file ownership.

sudo chown bob file.txt # Change owner to bob

sudo chown bob:staff file.txt # Change owner and group

---

3. Changing Group Ownership (chgrp)

To change only the group:

sudo chgrp developers file.txt

Chapter 4: Special Permissions (SUID, SGID, Sticky Bit)

Beyond standard permissions, Linux offers **special permissions** for **security and controlled execution**.

1. SUID (Set User ID)

- When set, a file runs with the permissions of the **owner**, not the user executing it.

- Commonly used for **privileged commands** like passwd.

Example:

chmod u+s /bin/passwd

ls -l /bin/passwd

Output:

-rwsr-xr-x 1 root root 541K Jan 1 12:00 /bin/passwd

The s indicates SUID is set.

---

2. SGID (Set Group ID)

- Ensures files created within a directory **inherit the group ownership**.
- Used for **shared project directories**.

Example:

chmod g+s /shared

ls -ld /shared

Output:

drwxr-sr-x 2 alice developers 4096 Jan 1 12:00 /shared

The s under the group permissions indicates SGID is set.

---

3. Sticky Bit

- Prevents **unauthorized file deletion** in shared directories.

- Used in **/tmp** to prevent users from deleting others' files.

Example:

chmod +t /public

ls -ld /public

Output:

drwxrwxrwt 2 root root 4096 Jan 1 12:00 /public

The t at the end indicates the sticky bit is set.

CHAPTER 5: CASE STUDY – SECURING A MULTI-USER LINUX SYSTEM

Scenario:

A university IT administrator needs to **secure student and faculty data** on a shared Linux server.

Solution:

1. **Create separate groups:**
2. sudo groupadd students
3. sudo groupadd faculty
4. **Assign users to groups:**
5. sudo usermod -aG students alice
6. sudo usermod -aG faculty bob
7. **Set permissions for student work directory:**
8. sudo chown :students /home/students

9. sudo chmod 770 /home/students
10. **Enable sticky bit for /public to prevent deletion of files by others:**
11. sudo chmod +t /public

Outcome:

- Students and faculty have **isolated workspaces**.
- The **public directory is secured** from accidental deletion.
- Unauthorized access is **prevented through permissions and ownership rules**.

CHAPTER 6: EXERCISE

1. **List three different Linux file permissions and explain their impact.**
2. **Create a file named test.txt, change its permissions to 644, and verify them.**
3. **Modify ownership of test.txt to a user named john.**
4. **Set the SUID permission on /usr/bin/passwd and explain its effect.**
5. **Create a shared directory for the developers group and enable SGID.**

CONCLUSION

Understanding **file permissions and ownership** is crucial for **system security, collaboration, and data integrity**. By correctly configuring **standard and special permissions**, administrators can ensure that only authorized users have access to critical files while maintaining a **secure Linux environment**.

Disk Management and Partitioning

Chapter 1: Introduction to Disk Management and Partitioning in Linux

Disk management and partitioning are fundamental aspects of **Linux system administration**. Efficient disk partitioning ensures that storage is optimized for **performance, security, and system organization**. Linux provides several tools and commands for managing disks, creating partitions, formatting file systems, and mounting storage devices.

Disk partitioning is the process of dividing a **physical storage device (HDD, SSD, USB)** into separate sections, each functioning as an independent unit. Partitions allow users to **separate operating systems, data storage, and swap space**, ensuring better resource allocation and system efficiency.

Proper disk management is essential for:

- **Efficient storage utilization** – Organizing disk space for different purposes.
- **Data security and isolation** – Protecting important data in separate partitions.
- **Performance optimization** – Preventing disk fragmentation and enhancing speed.
- **Multi-OS environments** – Running multiple operating systems on a single machine.

This chapter will cover **disk identification, partitioning, formatting, mounting, and disk monitoring tools**, along with practical exercises and a real-world case study.

CHAPTER 2: UNDERSTANDING DISK STRUCTURE AND PARTITION TYPES

1. Disk and Partition Structure in Linux

Linux treats all storage devices as files located in **/dev/**. Common disk devices include:

- **/dev/sda** – First SATA disk.
- **/dev/nvmeon1** – NVMe SSD storage.
- **/dev/mmcblko** – Flash storage (SD cards).

Each disk is divided into **partitions**, labeled as:

- **/dev/sda1** – First partition on /dev/sda.
- **/dev/sdb2** – Second partition on /dev/sdb.

2. Partition Table Types

There are two primary partitioning schemes:

Partition Type	**Description**
MBR (Master Boot Record)	Supports up to **4 primary partitions**, limited to **2TB disk size**.
GPT (GUID Partition Table)	Supports **128 partitions** and **disks larger than 2TB**, required for UEFI booting.

Example: Checking Partition Table Type

sudo fdisk -l /dev/sda

This command lists the partition scheme (MBR/GPT) and details of the disk layout.

CHAPTER 3: CREATING AND MANAGING PARTITIONS

Partitioning is essential for organizing disk space. Linux offers several tools to manage partitions:

- **fdisk** – For MBR-based partitioning.
- **parted** – For GPT-based partitioning.
- **lsblk** – Displays block devices and partitions.

1. Listing Available Disks (lsblk, fdisk)

To list all storage devices:

lsblk

Output example:

```
NAME   MAJ:MIN RM  SIZE  RO TYPE MOUNTPOINT
sda      8:0    0  500G  0  disk
├─sda1   8:1    0  100G  0  part /
├─sda2   8:2    0  200G  0  part /home
└─sda3   8:3    0  200G  0  part /data
```

This shows **disk partitions, their sizes, and mount points**.

---

2. Creating a New Partition Using fdisk (MBR Disks)

sudo fdisk /dev/sda

Steps:

1. Type **n** (new partition).
2. Choose **Primary (p) or Extended (e)**.
3. Specify partition size (e.g., **+50G** for 50GB).
4. Press **w** to write changes and exit.

To apply the changes:

sudo partprobe /dev/sda

---

3. Creating a New Partition Using parted (GPT Disks)

sudo parted /dev/sda

Steps:

1. Create a **GPT partition table**:
2. mklabel gpt
3. Create a new partition:
4. mkpart primary ext4 1MiB 100GiB
5. Exit and apply changes:
6. quit

Chapter 4: Formatting and Mounting Partitions

After creating a partition, it needs to be **formatted** with a file system before use.

1. Formatting a Partition (mkfs)

To format a partition with the **ext4** file system:

sudo mkfs.ext4 /dev/sda1

For **XFS** (used in enterprise systems):

sudo mkfs.xfs /dev/sda2

---

2. Mounting a Partition (mount)

To use a partition, it must be **mounted** to a directory.

sudo mount /dev/sda1 /mnt/data

To make this permanent, add an entry in **/etc/fstab**:

echo "/dev/sda1 /mnt/data ext4 defaults 0 2" | sudo tee -a /etc/fstab

---

3. Unmounting a Partition (umount)

To safely remove a mounted partition:

sudo umount /mnt/data

Chapter 5: Monitoring and Managing Disks

Linux provides multiple commands to check **disk usage, health, and performance**.

1. Checking Disk Usage (df)

df -h

Output example:

Filesystem Size Used Avail Use% Mounted on

/dev/sda1 100G 20G 80G 20% /

---

2. Checking Inode Usage (df -i)

df -i

This shows the number of **inodes (used/free)**, important for servers handling many small files.

---

3. Checking Disk Health (smartctl)

To check HDD/SSD health:

sudo smartctl -a /dev/sda

This provides **disk temperature, bad sectors, and overall health status**.

---

4. Checking Disk I/O Performance (iostat)

sudo iostat -dx

This helps monitor disk **read/write speed and latency**.

Chapter 6: Case Study – Setting Up Disk Partitions for a Web Server

Scenario:

A company is setting up a **Linux-based web server** that requires **optimized disk partitioning**.

Solution:

The system administrator follows these steps:

1. **Create a partition scheme:**
   - / (Root) → 50GB
   - /home → 200GB (for user files)
   - /var → 100GB (for logs & databases)
   - swap → 8GB (for virtual memory)
2. **Partitioning the disk (parted)**
3. sudo parted /dev/sdb mklabel gpt
4. sudo parted /dev/sdb mkpart primary ext4 1MiB 50GiB
5. **Format and mount partitions:**
6. sudo mkfs.ext4 /dev/sdb1
7. sudo mount /dev/sdb1 /var
8. **Automate mounting (/etc/fstab)**
9. echo "/dev/sdb1 /var ext4 defaults 0 2" | sudo tee -a /etc/fstab

Outcome:

- The web server now has **optimized partitions** for stability.
- Log files and databases are stored separately for **better performance**.

- The system is **easier to maintain and scale**.

CHAPTER 7: EXERCISE

1. **List all partitions and disk usage on your system.**
2. **Create a new partition on /dev/sdb with 20GB size using fdisk.**
3. **Format the partition with ext4 and mount it at /mnt/storage.**
4. **Check the health of /dev/sda using smartctl.**
5. **Configure /etc/fstab to mount /dev/sdb1 at boot.**

CONCLUSION

Mastering **disk management and partitioning** helps **optimize storage, enhance performance, and ensure data security** in Linux. By learning **partitioning, formatting, mounting, and monitoring tools**, administrators can efficiently **manage storage in servers, desktops, and cloud environments**.

SHELL SCRIPTING (BASH SCRIPTING BASICS)

CHAPTER 1: INTRODUCTION TO SHELL SCRIPTING

What is Shell Scripting?

Shell scripting is the practice of writing a sequence of **commands in a script file** that is executed by the **Linux shell**. The **Bash shell (Bourne Again Shell)** is the most commonly used shell in Linux systems, allowing users to automate repetitive tasks, configure system settings, and create complex workflows.

A **shell script** is simply a text file containing a series of commands that the shell interprets and executes. Instead of manually typing commands one by one, a shell script **automates processes** and **enhances efficiency** in Linux system administration and development.

Why Learn Shell Scripting?

- **Automation** – Reduces repetitive tasks (e.g., backups, log analysis).
- **System Administration** – Automates user management, software updates, and process monitoring.
- **Application Development** – Helps in setting up build environments, testing, and deployment.
- **Custom Workflows** – Enables users to create personalized scripts for routine operations.

This chapter will cover the **basics of Bash scripting**, including **variables, loops, conditionals, functions, and debugging techniques**, with examples and a real-world case study.

CHAPTER 2: CREATING AND RUNNING A SHELL SCRIPT

1. Creating a Shell Script

A shell script is a text file with a **.sh** extension. The first line must specify the **interpreter** (Bash) using a **shebang (#!)**:

Example:

```
#!/bin/bash

echo "Hello, Welcome to Bash Scripting!"
```

2. Making the Script Executable

After creating the script, it needs **execution permissions**:

```
chmod +x script.sh
```

To execute the script:

```
./script.sh
```

3. Running a Script Without Execution Permission

You can also execute a script using Bash directly:

```
bash script.sh
```

This method does not require the chmod +x step.

CHAPTER 3: VARIABLES IN BASH SCRIPTING

1. Declaring and Using Variables

Variables store data **without specifying a data type**.

Syntax:

variable_name="value"

echo $variable_name

Example:

#!/bin/bash

name="Alice"

echo "Hello, $name!"

2. Reading User Input (read)

To take user input in a script:

#!/bin/bash

echo "Enter your name: "

read user_name

echo "Hello, $user_name!"

3. Using Command Substitution

Bash allows storing **command outputs** in variables:

date_today=$(date)

echo "Today's date is: $date_today"

This script stores the **current date** in a variable and prints it.

Chapter 4: Conditional Statements in Bash

Conditional statements allow scripts to execute commands **based on conditions**.

1. If-Else Statement

```
#!/bin/bash

echo "Enter a number: "

read num


if [ $num -gt 10 ]; then

   echo "The number is greater than 10."

else

   echo "The number is 10 or less."

fi
```

- -gt → Greater than
- -lt → Less than
- -eq → Equal to

2. Case Statement (Switch Alternative)

```
#!/bin/bash

echo "Enter a fruit name:"

read fruit


case $fruit in

   "apple") echo "Apples are red." ;;

   "banana") echo "Bananas are yellow." ;;
```

```
  "grape") echo "Grapes are purple." ;;
  *) echo "Unknown fruit." ;;
esac
```

This script **matches user input with predefined cases** and executes the respective action.

CHAPTER 5: LOOPS IN BASH SCRIPTING

Loops allow scripts to **repeat commands** for a defined number of times or until a condition is met.

1. For Loop

```
#!/bin/bash
for i in {1..5}; do
  echo "Iteration: $i"
done
```

This loop runs **5 times**, printing iteration numbers.

2. While Loop

```
#!/bin/bash
count=1
while [ $count -le 5 ]; do
  echo "Count: $count"
  ((count++))
done
```

This loop continues until count exceeds 5.

CHAPTER 6: FUNCTIONS IN BASH SCRIPTING

Functions **organize code** and allow reusability in scripts.

1. Defining and Calling a Function

```
#!/bin/bash

greet() {

  echo "Hello, $1!"

}

greet "Alice"

greet "Bob"
```

- $1 represents the **first argument** passed to the function.
- This script **greets multiple users dynamically**.

2. Returning Values from Functions

```
#!/bin/bash

sum() {

  echo $(($1 + $2))

}

result=$(sum 5 10)

echo "Sum: $result"
```

Functions can perform calculations and **return results**.

Chapter 7: Debugging and Logging in Shell Scripts

1. Debugging with set -x

Enable debugging mode to trace script execution:

#!/bin/bash

set -x

echo "Debugging Mode On"

To disable debugging:

set +x

2. Redirecting Output to Logs

./script.sh > output.log 2>&1

This captures **both standard output and errors** in output.log.

Chapter 8: Case Study – Automating System Updates with Bash Script

Scenario:

A system administrator needs to automate the **software update process** for a Linux server.

Solution:

1. Create a script (update_system.sh):
2. #!/bin/bash

3. echo "Updating system..."
4. sudo apt update && sudo apt upgrade -y
5. echo "System update completed!"
6. Make it executable:
7. chmod +x update_system.sh
8. Schedule it using **cron** (automatic execution):
9. crontab -e

Add this line to run the script daily at midnight:

0 0 * * * /path/to/update_system.sh

Outcome:

- The system is **automatically updated** without manual intervention.
- Security patches and software updates are applied on time.
- The script **saves admin time and improves system security**.

CHAPTER 9: EXERCISE

1. **Write a Bash script to check disk usage (df -h) and save the output to a log file.**
2. **Create a script that asks the user for a name and prints "Hello, [name]!".**
3. **Write a loop that prints numbers from 1 to 10.**

4. **Modify the system update script to also clean up unused packages (sudo apt autoremove).**
5. **Use a function to calculate and print the factorial of a given number.**

CONCLUSION

Mastering **Bash scripting** enhances **automation, system administration, and process optimization** in Linux environments. By learning **variables, conditionals, loops, functions, and debugging techniques**, users can create powerful scripts for **automating tasks, managing servers, and improving workflow efficiency**.

CONDITIONAL STATEMENTS & LOOPING IN BASH SCRIPTING

CHAPTER 1: INTRODUCTION TO CONDITIONAL STATEMENTS AND LOOPING IN BASH

Conditional statements and loops are essential components of **Bash scripting** that allow scripts to make decisions and execute **repetitive tasks automatically**. These constructs **enhance automation, optimize system operations, and reduce manual intervention** in Linux environments.

- **Conditional Statements** (if-else, case) help **execute different commands** based on conditions.
- **Loops** (for, while, until) allow executing a **set of commands multiple times** until a specified condition is met.

Why Use Conditionals and Loops?

- **Automate decision-making processes** (e.g., checking disk usage and sending alerts).
- **Optimize repetitive tasks** (e.g., renaming multiple files in a directory).
- **Improve system efficiency** (e.g., continuously monitor system health).

This chapter covers **if-else statements, case statements, for loops, while loops, and until loops**, with **detailed examples, case studies, and exercises** to reinforce learning.

CHAPTER 2: UNDERSTANDING CONDITIONAL STATEMENTS IN BASH

Conditional statements allow scripts to **execute different commands** based on specific conditions.

1. If-Else Statement

The if statement is used to check a condition and execute commands accordingly.

Syntax:

```
if [ condition ]; then
  # Commands if condition is true
else
  # Commands if condition is false
fi
```

Example: Checking If a File Exists

```
#!/bin/bash
echo "Enter file name:"
read filename

if [ -f "$filename" ]; then
   echo "The file $filename exists."
else
   echo "The file $filename does not exist."
fi
```

- -f checks if the given file exists.

- The script **asks for a filename and verifies its existence**.

---

2. Nested If-Else Statements

You can nest if statements for multiple conditions.

Example: Checking User Privileges

```
#!/bin/bash

if [ "$USER" == "root" ]; then

  echo "You are running as root."

  if [ -w "/etc/passwd" ]; then

    echo "You have permission to modify system files."

  else

    echo "You do not have write access."

  fi

else

  echo "You must run this script as root!"

fi
```

- Checks if the script is **executed as root**.
- If root, it verifies if the user has **write permissions** for system files.

---

3. Using Else-If (elif) for Multiple Conditions

Use elif to **check multiple conditions** in an if statement.

Example: Checking Disk Space Usage

```
#!/bin/bash
used_space=$(df -h / | awk 'NR==2 {print $5}' | sed 's/%//')

if [ $used_space -lt 50 ]; then
    echo "Disk usage is under control."
elif [ $used_space -lt 80 ]; then
    echo "Warning: Disk usage is moderate ($used_space%)."
else
    echo "Critical: Disk usage is high ($used_space%)!"
fi
```

- The script **retrieves disk usage** and warns the user if usage is high.

Chapter 3: Case Statement (Alternative to If-Else)

The case statement simplifies multi-condition scenarios, making scripts **easier to read** than nested if statements.

Syntax:

```
case variable in
  pattern1) command1 ;;
  pattern2) command2 ;;
```

```
  *) default_command ;;
esac
```

Example: Simple User Menu

```
#!/bin/bash
echo "Select an option: (start, stop, restart)"
read action

case $action in
  start) echo "Starting the service..." ;;
  stop) echo "Stopping the service..." ;;
  restart) echo "Restarting the service..." ;;
  *) echo "Invalid option!" ;;
esac
```

- The script presents **a menu for starting, stopping, or restarting a service**.
- The *) case handles **invalid inputs**.

CHAPTER 4: LOOPING IN BASH

Loops execute **a set of commands repeatedly** based on conditions.

1. For Loop

The for loop runs commands for **each item in a list or range**.

Example: Iterating Over a List

```
#!/bin/bash
for name in Alice Bob Charlie; do
   echo "Hello, $name!"
done
```

- The script **greets each name in the list**.

Example: Looping Through Files in a Directory

```
#!/bin/bash
for file in *.txt; do
   echo "Processing $file..."
done
```

- Lists all .txt files and processes them one by one.

---

2. While Loop

The while loop executes **until the condition becomes false**.

Example: Countdown Timer

```
#!/bin/bash
count=5
while [ $count -gt 0 ]; do
   echo "Countdown: $count"
   ((count--))
```

```
  sleep 1
done
echo "Time's up!"
```

- The script **counts down from 5** and waits **1 second between iterations**.

---

3. Until Loop

The until loop runs **until a condition becomes true**.

Example: Waiting for a File to Appear

```
#!/bin/bash
until [ -f /tmp/signal.txt ]; do
  echo "Waiting for signal file..."
  sleep 5
done
echo "Signal file detected!"
```

- The script **waits until a specific file appears**, checking every 5 seconds.

Chapter 5: Combining Conditional Statements and Loops

By combining conditionals and loops, we can create **dynamic, interactive, and automated scripts**.

Example: Monitoring System Load

```
#!/bin/bash
while true; do
  load=$(uptime | awk '{print $10}' | sed 's/,//')
  if (( $(echo "$load > 1.5" | bc -l) )); then
    echo "Warning: High system load ($load)"
  fi
  sleep 10
done
```

- This script **monitors system load** and prints a warning if it **exceeds 1.5**.

CHAPTER 6: CASE STUDY – AUTOMATING USER MANAGEMENT WITH BASH

Scenario:

A system administrator needs to **create multiple users automatically** and **assign them default passwords**.

Solution:

1. **Create a file (users.txt) with usernames:**
2. alice
3. bob
4. charlie
5. **Write a script to create users and set passwords:**

6. #!/bin/bash
7. while read user; do
8. sudo useradd -m "$user"
9. echo "$user:password123" | sudo chpasswd
10. echo "User $user created successfully!"
11. done < users.txt
12. **Execute the script:**
13. ./create_users.sh

Outcome:

- **Multiple users are created in seconds**, saving admin time.
- The script ensures **consistency and security**.

Chapter 7: Exercise

1. **Write a script that checks if a directory exists; if not, it should create one.**
2. **Modify the disk space check script to delete old log files if usage exceeds 80%.**
3. **Create a script using a case statement that provides different system info (uptime, disk usage, memory usage) based on user input.**
4. **Write a loop that renames all .log files in a directory to .bak.**
5. **Develop a script that continuously monitors a process (apache2) and restarts it if it stops running.**

CONCLUSION

Understanding **conditional statements and loops** in Bash scripting allows users to create **automated, interactive, and efficient** scripts. These tools enable system administrators to **handle complex workflows**, **automate repetitive tasks**, and **enhance system monitoring**.

Process Management (Foreground/Background Processes, Kill, Nice, Jobs)

Chapter 1: Introduction to Process Management in Linux

In Linux, every command executed by a user or the system is a **process**. Managing processes efficiently is crucial for **performance, stability, and resource allocation**. Linux provides various tools to **monitor, control, prioritize, and terminate processes**.

Process management includes:

- **Foreground and Background Processes** – Running tasks interactively or in the background.
- **Process Control (kill, pkill, killall)** – Terminating unresponsive or unwanted processes.
- **Process Prioritization (nice, renice)** – Adjusting process priority to allocate system resources efficiently.
- **Job Control (jobs, fg, bg, disown)** – Managing running processes within the shell.

Mastering these concepts allows system administrators and users to **optimize system performance, troubleshoot issues, and manage resource allocation effectively**.

Chapter 2: Understanding Linux Processes

1. What is a Process?

A process is an instance of a running program. Linux assigns a **unique Process ID (PID)** to each process, allowing users to **monitor, manage, and control** them efficiently.

Linux processes are categorized into:

1. **Foreground Processes** – Executed interactively in the terminal.
2. **Background Processes** – Run in the background, allowing users to continue working in the terminal.
3. **Daemon Processes** – System services that run in the background (e.g., cron, sshd).
4. **Zombie Processes** – Completed processes that still have an entry in the process table.

2. Listing Running Processes (ps, top, htop)

To view active processes:

ps aux

This displays process details such as **PID, user, CPU/memory usage, and command**.

To continuously monitor processes:

top

For an interactive interface:

htop

(htop provides an enhanced process monitoring experience with scrolling and sorting).

CHAPTER 3: FOREGROUND AND BACKGROUND PROCESSES

1. Running a Process in the Foreground

By default, a command runs in the **foreground**, occupying the terminal until it completes.

Example:

ping google.com

This command will **keep running** until you manually stop it (CTRL + C).

---

2. Running a Process in the Background (&)

Appending & at the end of a command runs it in the **background**, freeing the terminal.

Example:

ping google.com > output.txt &

This allows the process to run **while you continue working** in the same terminal.

To view **background processes**:

jobs

To move a process to the **foreground**:

fg %1

(%1 represents the **job number**, seen in the jobs output).

To send a foreground process to the **background**:

1. Press CTRL + Z (pauses the process).
2. Use:
3. bg %1

This resumes the process in the **background**.

CHAPTER 4: KILLING PROCESSES (KILL, PKILL, KILLALL)

Sometimes, processes become **unresponsive** or consume excessive resources, requiring termination.

1. Killing a Process by PID (kill)

Find the process ID (PID) using:

ps aux | grep process_name

Then terminate it:

kill PID

To forcefully terminate:

kill -9 PID

(-9 sends the **SIGKILL** signal, immediately stopping the process).

---

2. Killing a Process by Name (pkill, killall)

To terminate a process by name:

pkill process_name

To kill **all instances** of a process:

killall process_name

(killall is useful when multiple processes of the same name are running).

Example: Killing a Stuck Application

pkill firefox

This **terminates all running instances of Firefox**.

CHAPTER 5: PROCESS PRIORITIZATION (NICE & RENICE)

In Linux, each process has a **priority level**, which determines how CPU time is allocated.

1. Checking Process Priority (ps -l)

ps -eo pid,ppid,cmd,pri,nice

This displays **priority (PRI) and nice value (NI)**.

Nice Value (NI)	Priority (PRI)	Effect
-20	Highest priority	Process gets more CPU time
0	Default priority	Normal scheduling
+19	Lowest priority	Process gets less CPU time

---

2. Running a Process with a Custom Priority (nice)

To start a process with a **lower priority**:

nice -n 10 long_running_task.sh

To give a process **higher priority**, use:

sudo nice -n -5 backup_script.sh

(-5 increases priority, requires sudo).

---

3. Changing the Priority of a Running Process (renice)

To adjust the priority of an **existing process**:

renice -10 -p PID

(PID is the process ID).

Example: Lowering the Priority of a CPU-Intensive Task

renice 15 -p 1234

This ensures other tasks **get more CPU time** than the given process.

Chapter 6: Job Control in Linux (jobs, fg, bg, disown)

1. Viewing Background Jobs (jobs)

To list all background jobs:

jobs -l

Example output:

[1]+ Running backup.sh &

[2]- Stopped download.sh

- [1]+ → Job number 1, running in the background.
- [2]- → Job number 2, stopped (paused).

---

2. Bringing a Job to the Foreground (fg)

To resume job **1** in the foreground:

fg %1

---

3. Moving a Foreground Job to the Background (CTRL + Z → bg)

1. **Pause the process** with CTRL + Z.
2. Resume it in the **background**:
3. bg %1

---

4. Removing a Job from the Shell (disown)

To **detach** a process so it continues running **even after logging out**:

disown -h %1

(%1 refers to the job number).

Chapter 7: Case Study – Optimizing Server Performance with Process Management

Scenario:

A system administrator manages a **web server** where a background script (log_backup.sh) frequently consumes **high CPU usage**, affecting performance.

Solution:

1. **Check process priority and CPU usage:**

2. ps aux --sort=-%cpu | head -5
3. **Lower priority using renice:**
4. sudo renice 15 -p 12345
5. **Move process to background:**
6. bg %1
7. **Ensure it runs even after logout (disown)**:
8. disown -h %1

Outcome:

- The backup script now runs with **low priority**, reducing CPU load.
- The administrator **continues working** without interruptions.
- The server remains **responsive for critical tasks**.

CHAPTER 8: EXERCISE

1. **Run a long-running process in the background and bring it back to the foreground.**
2. **Find the PID of the Firefox process and terminate it using kill.**
3. **Start a process with nice at priority 10 and adjust it to -5 using renice.**
4. **Use disown to detach a job so it continues running after logout.**

5. **Write a script to monitor the top 5 CPU-consuming processes every 5 seconds.**

CONCLUSION

Process management is **vital for system performance and stability**. By mastering **foreground/background processes, job control, process prioritization, and termination techniques**, users can **optimize workflows, troubleshoot performance issues, and efficiently manage system resources**.

Environment Variables & Configuration Files

Chapter 1: Introduction to Environment Variables and Configuration Files

What Are Environment Variables?

Environment variables are **dynamic values** stored in the system that **affect the behavior of running processes**. They help in configuring the **shell, applications, and user preferences** by passing information between the operating system and executing programs.

Common uses of environment variables:

- **Configuring system-wide settings** (e.g., PATH, HOME).
- **Defining user preferences** (e.g., HISTSIZE, EDITOR).
- **Passing settings to applications** (e.g., JAVA_HOME, PYTHONPATH).

What Are Configuration Files?

Configuration files (config files) store **persistent settings** for applications, services, and user environments. They are typically stored in:

- **System-wide configuration:** /etc/ (affects all users).
- **User-specific configuration:** ~/.config/ or ~/.bashrc.

Understanding **environment variables and configuration files** is essential for **system administrators, developers, and Linux users** to efficiently **customize, optimize, and manage system behavior**.

CHAPTER 2: UNDERSTANDING ENVIRONMENT VARIABLES

1. Types of Environment Variables

There are two main types of environment variables:

Type	Description
System Variables	Set by the system and available to all users (PATH, HOME).
User-defined Variables	Created by users for personal configurations.

2. Viewing Environment Variables (env, printenv)

To list all active environment variables:

env

To check a **specific variable**:

echo $HOME

printenv USER

---

3. Setting Environment Variables (export)

To create a **temporary environment variable**:

export MY_VAR="Hello World"

echo $MY_VAR

This variable exists **only in the current session**.

To make it **permanent**, add it to **~/.bashrc (for user) or /etc/environment (for system-wide)**.

Example: Adding to ~/.bashrc

echo 'export MY_VAR="Hello World"' >> ~/.bashrc

source ~/.bashrc

---

4. Removing Environment Variables (unset)

To delete a variable:

unset MY_VAR

Chapter 3: Important Environment Variables in Linux

Variable	Description
PATH	Directories where executable programs are located.
HOME	User's home directory.
SHELL	The default shell (/bin/bash, /bin/zsh).
USER	The currently logged-in user.
PWD	The current working directory.
EDITOR	Default text editor (vim, nano).
LANG	Language settings.

---

1. Modifying the PATH Variable

The PATH variable determines where the system searches for executable files.

To add a new directory (/opt/myapp/bin) to PATH:

export PATH=$PATH:/opt/myapp/bin

To make this change **permanent**, add it to ~/.bashrc or ~/.profile:

echo 'export PATH=$PATH:/opt/myapp/bin' >> ~/.bashrc

source ~/.bashrc

---

2. Changing the Default Text Editor

By default, Linux may use **vi** as the system editor. To change it to nano:

export EDITOR=nano

To make this permanent:

echo 'export EDITOR=nano' >> ~/.bashrc

Chapter 4: Configuration Files in Linux

1. System-Wide Configuration Files (/etc/)

System-wide configuration files are stored in **/etc/** and apply to all users.

File	Description
/etc/passwd	Stores user account information.
/etc/group	Stores user groups.
/etc/fstab	Contains disk mount points.

File	Description
/etc/environment	Defines system-wide environment variables.
/etc/profile	Runs startup scripts for all users.

Example: Adding a System-Wide Variable (/etc/environment)

To add a variable that applies to all users:

echo 'GLOBAL_VAR="System Wide Variable"' | sudo tee -a /etc/environment

source /etc/environment

echo $GLOBAL_VAR

---

2. User-Specific Configuration Files (~/.bashrc, ~/.profile)

Each user has configuration files in their **home directory (~)** to personalize the environment.

File	Description
~/.bashrc	Runs when a new terminal session starts.
~/.profile	Runs at login, loads environment variables.
~/.bash_logout	Runs when logging out.
~/.config/	Stores application-specific settings.

Example: Customizing ~/.bashrc to Show a Welcome Message

Edit ~/.bashrc:

echo 'echo "Welcome, $USER!"' >> ~/.bashrc source ~/.bashrc

Now, every time the user **opens a terminal**, they will see:

Welcome, username!

---

3. Configuration Files for Applications

Many applications store their settings in configuration files:

Application	Configuration File
Bash Shell	~/.bashrc, ~/.profile
SSH	/etc/ssh/sshd_config
Apache Web Server	/etc/apache2/apache2.conf
MySQL	/etc/mysql/my.cnf

Example: Changing the **default SSH port** in /etc/ssh/sshd_config:

sudo nano /etc/ssh/sshd_config

Change: Port 22 → Port 2222

sudo systemctl restart sshd

CHAPTER 5: CASE STUDY – AUTOMATING ENVIRONMENT SETUP FOR DEVELOPERS

Scenario:

A **development team** needs a consistent environment with:

- Custom **Python path (PYTHONPATH)**.

- Custom **alias for git commands**.
- Default **text editor set to Vim**.

Solution:

The admin adds the following to **/etc/profile** (for all users):

export PYTHONPATH=/usr/local/lib/python3.10

export EDITOR=vim

alias gs='git status'

alias gp='git push'

After applying changes (source /etc/profile), every developer now has:

- **Correct Python paths set up**.
- **Shortcuts for Git operations**.
- **Vim as the default editor**.

Outcome:

- Developers **work in a consistent environment**, reducing setup time.
- The **system configuration is standardized**, minimizing issues.

Chapter 6: Exercise

1. **List all current environment variables using env.**
2. **Create a custom environment variable (MY_PROJECT=LinuxProject) and make it persistent.**

3. **Modify PATH to include /opt/scripts/ and test if a script inside that directory runs without specifying the full path.**

4. **Configure nano as the default editor in ~/.bashrc and verify the change.**

5. **Edit /etc/environment to add a system-wide variable (SERVER_ENV=Production). Restart the system and check if it persists.**

CONCLUSION

Understanding **environment variables and configuration files** is crucial for **customizing Linux systems, automating workflows, and optimizing user environments**. Whether modifying **PATH variables, setting application defaults, or creating system-wide configurations**, these concepts **enhance productivity and system efficiency**.

Assignment Solution: Automating User Data Backup with a Shell Script

Objective

The goal of this assignment is to **create a shell script** that automates the backup of user data, ensuring important files are **saved regularly** to prevent data loss. The script will:

- **Backup user files from the home directory (/home/username).**
- **Compress the backup using tar.**
- **Store the backup in a designated directory (/backup).**
- **Optionally, schedule the script using cron for automatic execution.**

Step 1: Understanding the Script Components

The backup script will:

- Use **tar** to archive files.
- Use **date formatting** to create unique backup filenames.
- Save backups in the /backup/ directory.
- Keep **logs of backup operations**.
- Optionally **delete old backups** after a certain period.

Step 2: Writing the Shell Script

1. Create the Script File

Open a terminal and create a new script file:

```
nano backup_script.sh
```

2. Add the Script Code

Copy and paste the following script:

```
#!/bin/bash


# Backup Configuration

BACKUP_SOURCE="/home/$USER"   # Directory to back up

BACKUP_DEST="/backup"       # Backup storage directory

LOG_FILE="/backup/backup.log" # Log file location

TIMESTAMP=$(date +"%Y-%m-%d_%H-%M-%S")

BACKUP_FILE="$BACKUP_DEST/user_backup_$TIMESTAMP.tar.gz"


# Ensure backup directory exists

if [ ! -d "$BACKUP_DEST" ]; then

  mkdir -p "$BACKUP_DEST"

  echo "Backup directory created: $BACKUP_DEST"

fi
```

```
# Perform backup using tar

echo "Starting backup at $(date)" >> $LOG_FILE

tar -czf "$BACKUP_FILE" "$BACKUP_SOURCE" 2>> $LOG_FILE


# Verify backup success

if [ $? -eq 0 ]; then

   echo "Backup successful: $BACKUP_FILE" >> $LOG_FILE

else

   echo "Backup failed!" >> $LOG_FILE

   exit 1

fi


# Delete backups older than 7 days

find "$BACKUP_DEST" -type f -name "*.tar.gz" -mtime +7 -exec rm {} \;

echo "Old backups deleted" >> $LOG_FILE


# Completion message

echo "Backup completed successfully at $(date)" >> $LOG_FILE
```

Step 3: Making the Script Executable

Grant **execute permission** to the script:

chmod +x backup_script.sh

Step 4: Running the Script Manually

Execute the script to perform a test backup:

./backup_script.sh

After execution, check:

- **Backup file**:
- ls /backup/
- **Log file**:
- cat /backup/backup.log

Step 5: Automating the Backup Using Cron Job

To run the script **daily at midnight**, add it to the cron scheduler:

crontab -e

Add the following line at the end of the file:

0 0 * * * /path/to/backup_script.sh

This will automatically execute the script **every day at midnight**.

To list scheduled cron jobs:

crontab -l

STEP 6: VERIFYING AUTOMATION

1. Run cron logs to confirm execution:
2. grep CRON /var/log/syslog
3. Check if a new backup file is created daily:
4. ls -lh /backup/

CONCLUSION

This automated backup script:
✓ **Secures user data** by creating daily backups.
✓ **Uses timestamps** to prevent overwriting backups.
✓ **Deletes old backups** to save space.
✓ **Logs all operations** for troubleshooting.
✓ **Runs automatically** using cron.

Assignment Solution: Managing User Permissions and Creating a New User Group

Objective

The objective of this assignment is to **manage user permissions and create a new user group in Linux**. The task involves:

1. **Creating a new user group.**
2. **Adding users to the group.**
3. **Setting correct file permissions for the group.**
4. **Testing user access and permissions.**

By the end of this guide, you will have **a well-organized user group with restricted access to specific resources**.

Step 1: Creating a New User Group

A group allows multiple users to share the same set of permissions, simplifying access management.

1. Create a New Group

sudo groupadd developers

This creates a new group called **developers**.

2. Verify Group Creation

cat /etc/group | grep developers

This checks if the group has been successfully added to the system.

Step 2: Adding Users to the Group

Once the group is created, we need to **add users to it**.

1. Add Existing Users to the Group

sudo usermod -aG developers alice

sudo usermod -aG developers bob

This adds **alice** and **bob** to the **developers** group.

2. Verify User Membership

groups alice

This will list all groups the user **alice** belongs to, including **developers**.

To check **all users in a group**, use:

getent group developers

Step 3: Setting Group Permissions on a Shared Directory

Now that users are part of the **developers** group, we need to **assign group permissions** to a shared directory (/dev_projects).

1. Create the Shared Directory

sudo mkdir /dev_projects

2. Change Ownership to the Group

sudo chown :developers /dev_projects

This assigns the **developers** group as the owner of the directory.

3. Set Correct Group Permissions

sudo chmod 770 /dev_projects

- 7 (rwx) → Group members **can read, write, and execute** files.
- 0 (---) → Others **have no access**.

4. Verify Permissions

ls -ld /dev_projects

Expected output:

drwxrwx--- 2 root developers 4096 Jan 1 10:00 /dev_projects

This confirms that **only developers can access the folder**.

STEP 4: TESTING USER ACCESS

1. Switch to a Developer User

su - alice

cd /dev_projects

touch testfile.txt

If successful, **alice** can create files in /dev_projects.

2. Test Access for a User Outside the Group

su - guest

cd /dev_projects

Expected output:

Permission denied

This confirms that users **outside the developers group cannot access the directory**.

Step 5: Removing a User from the Group

If a user no longer needs access, remove them from the group:

sudo gpasswd -d alice developers

To verify removal:

groups alice

Step 6: Deleting the Group (Optional)

If the group is no longer needed, **remove it**:

sudo groupdel developers

Make sure no users or files depend on it before deleting.

Conclusion

This step-by-step guide has demonstrated how to:

✓ **Create and manage user groups**.

✓ **Add and remove users from groups**.

✓ **Set appropriate file permissions for shared directories**.

✓ **Test access control using permissions**.