Experiment [1]: [Linux OS Environment Setup]

Name: Omkareshwar Chaubey, Roll No.: 590025556, Date: 2025-09-17

AIM:

- To install and configure different Linux operating system environments on a Windows machine. We will use two distinct technologies: **Windows Subsystem for Linux (WSL)** for a lightweight command-line environment and **Oracle VirtualBox** for a full graphical virtual machine.

Requirements:

- A Windows 10/11 PC.
- Administrator access and **hardware virtualization enabled in the BIOS/UEFI**.
- An internet connection.

Theory:

- This experiment is designed to provide hands-on experience with two primary methods of running Linux on a Windows host.This is ideal for developers and system administrators who require a Linux command-line without the overhead of a full virtual machine.
- **Oracle VirtualBox**, on the other hand, is a traditional Type 2 hypervisor. It creates a complete, virtualized computer system on which a guest operating system (like Ubuntu or Linux Mint) can be installed. This method provides a fully isolated environment, complete with a graphical user interface (GUI).

Procedure & Observations

Part 1: Installing and Configuring WSL (Ubuntu)

Exercise 1: [Installing WSL on Windows]

- **Task Statement:** Enable the required Windows features and install the Ubuntu Linux distribution using a single command.
- **Explanation:** This demonstrates how the modern `wsl --install` command simplifies the entire setup process, automating what previously required multiple manual steps.
- **Command(s):**

```
wsl --install -d ubuntu
```

- **Observation:** The command automatically enabled the "Virtual Machine Platform" and "Windows Subsystem for Linux" optional components. It then proceeded to download the Ubuntu distribution. The system requested a reboot to complete the final stage of the installation.

Exercise 2: [Configuring the Ubuntu Distribution]

- **Task Statement:** After the initial installation and reboot, configure the Ubuntu environment by creating a new user account.
- **Explanation:** This step is crucial for security and user management within the Linux environment. The new UNIX username and password created are separate from the Windows user account.
- **Observation:** Upon reboot, a terminal window opened automatically. It prompted for a "New UNIX username" and a password. After entering the credentials, the setup was complete and the command-line interface became available.

Exercise 3: [Verifying WSL Installation]

- **Task Statement:** Confirm that the WSL installation is successful and the Ubuntu distribution is ready for use.
- **Explanation:** This command provides a simple way to list all installed WSL distributions, showing their names, versions, and current state.
- **Command(s):**

```
wsl -l -v
```

- **Output:**

```
  NAME      STATE     VERSION
* Ubuntu    Running   2
```

- **Observation:** The output confirmed that Ubuntu was correctly installed and was currently in a "Running" state, with version 2 (indicating it is running on the WSL 2 architecture).

Part 2: Installing VirtualBox and a Linux VM (Linux Mint)

Exercise 4: [Installing Oracle VirtualBox]

- **Task Statement:** Download and install the Oracle VirtualBox hypervisor on the Windows host machine.
- **Procedure:** The VirtualBox installer was downloaded from the official website. Permission to install device software for network interfaces was granted.
- **Observation:** The VirtualBox application was successfully installed on the Windows system, along with the necessary drivers to support virtual machines.

Exercise 5: [Creating a Virtual Machine]

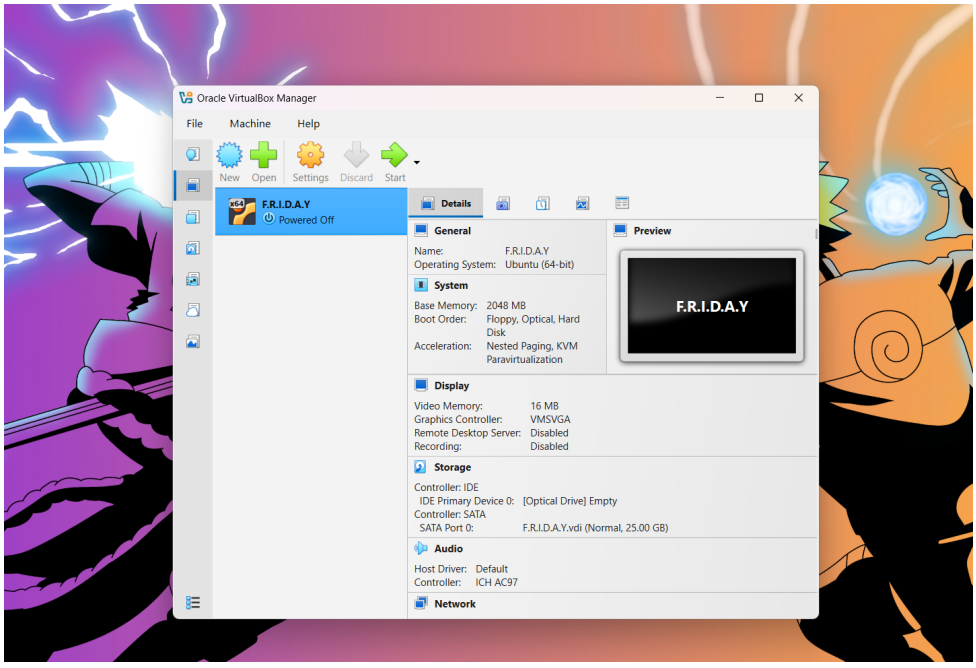
- **Task Statement:** Create a new virtual machine to host the Linux Mint operating system.
- **Procedure:** In the VirtualBox Manager, a new VM was created. The name was set to "Linux Mint", and a downloaded `.iso` file was selected as the installation medium. Hardware resources were configured with **4096 MB RAM** and **2 CPUs**. A new dynamically allocated virtual hard disk of **25 GB** was created.
- **Observation:** The VM was configured with the specified resources, creating a virtualized hardware environment ready to receive an operating system.

Exercise 6: [Installing Linux Mint]

- **Task Statement:** Install the Linux Mint OS on the virtual machine.
- **Procedure:** The newly created VM was started, which booted directly from the Linux Mint `.iso`.
- **Observation:** The installation proceeded without issues, partitioning the virtual disk and copying the OS files. The process was identical to a standard installation on a physical computer.

Result

- The experiment was successfully completed by setting up two distinct Linux environments. **Windows Subsystem for Linux** and a complete **virtual machine** with Linux Mint. This project demonstrated proficiency in using both a compatibility layer and a full hypervisor to meet different virtualization needs.



Experiment [2]: [Linux file systems permissions and essential commands]

Name: Omkareshwar Chaubey Roll.: 590025556 Date: 2025-17-05

AIM:

- [To Learn linux file systems permissions and essential commands]

Requirements:

- [Any Linux Distro, any kind of text editor (vs code, vim, notepad, nano, etc,]

Theory:

- [Basic Linux file systems permissions and essential commands]

Procedure & Observations

TASK 1: [Directory Navigation]

Task Statement:

- [Create a directory called test_project in your home directory, then create subdirectories docs, scripts, and data inside it. Navigate to the scripts directory and display your current path.]

Explanation:

- [Use mkdir to create the wanted directory we can use cd to navigate and use pwd to show current path]

Command(s):

''' mkdir test_project cd test_project mkdir docs scripts data cd scripts pwd '''

Output:

TASK 2: [FIle Creation and Content]

Task Statement:

- [Create three files in the docs directory: readme.txt, notes.txt, and todo.txt. Add the text "Project documentation" to readme.txt and "Important notes" to notes.txt. Display the contents of both files.]

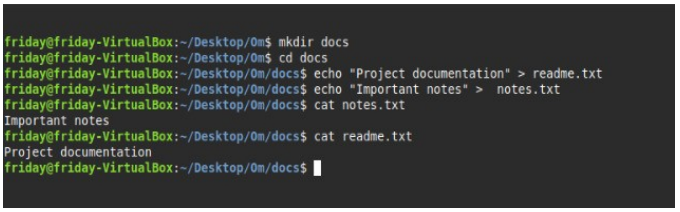
Explanation:

- [We can use touch to create empty files and using echo "text" > file.txt to add content to a file and using cat to display file contents]

Command(s):

```
cd docs touch readme.txt notes.txt todo.txt echo "Project documentation" > readme.txt echo "Important notes" > notes.txt cat notes.txt cat readme.txt
```

Output:



TASK 3: [FIle Operations]

Task Statement:

- [Copy readme.txt to the data directory and rename the copy to project_info.txt. Then move todo.txt from docs to scripts directory.]

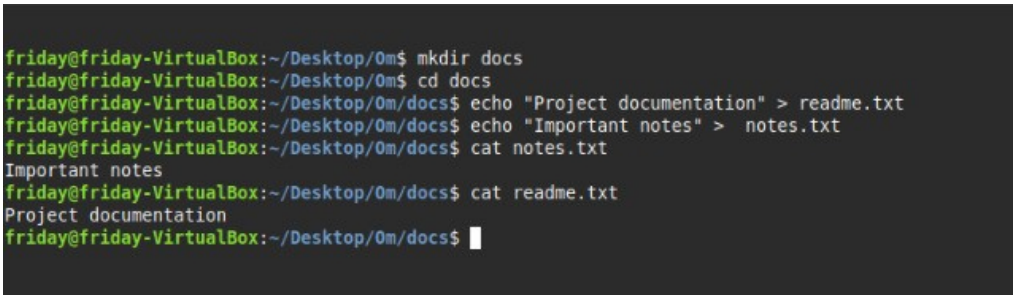
Explanation:

- [- We can use the cp source destination to copy files and using the mv oldname newname to rename files also using the same command mv file directory/ to move files to another directory we can also combine copy and rename: cp file.txt newdir/newname.txt]

Command(s):

```
cp readme.txt data/project_info.txt
```

Output:



TASK 4: [FIle Permissions]

Task Statement:

- [Create a shell script file called backup.sh in the scripts directory. Add the content #!/bin/bash and echo "Backup complete" to it. Make the file executable only for the owner.]

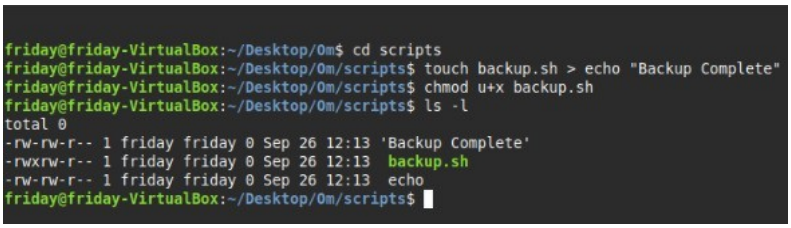
Explanation:

- [Using chmod u+x filename we can make the file executable for user only using ls -l to check for permissions also script files typically need executable permission to run]

Command(s):

```
cd scripts touch backup.sh > echo "Backup complete" chmod u+x backup.sh
```

Output:



TASK 5: [FIle Viewing]

Task Statement:

- [Create a file called numbers.txt with numbers 1 to 20 (each on a new line). Display only the first 5 lines, then only the last 3 lines, then search for lines containing the number "1".]

Explanation:

- [I can quickly generate a list of numbers by running seq 1 20 > numbers.txt. To check the first few numbers, I use head -n 5 to see the first 5 lines, and tail -n 3 to see the last 3 lines. If I want to find all numbers containing a “1”, I can use grep "1". Alternatively, I could create the list manually by using multiple echo commands.]

Command(s):

```
seq 1 20 > numbers.txt head -n 5 tail -n 3 grep "1"
```

Output:

TASK 6: [Text Editing]

Task Statement:

- [Using nano, create a file called config.txt with the following content:

Database=localhost Port=5432 Username=admin

Save the file and then display its contents.]

Explanation:

- [I open a file in Nano using nano filename.txt and type my content normally. Once I'm done, I press Ctrl+O to save the file and Ctrl+X to exit Nano. After that, I use cat to check the contents and make sure everything was saved correctly.]

Command(s):

```
vim config.txt cat config.txt
```

Alternatively

```
nano config.txt cat config.txt
```

Output:

TASK 7: [System Information]

Task Statement:

- [Create a file called system_info.txt that contains: your username, current date, your current directory, and disk usage information in human-readable format.]

Explanation:

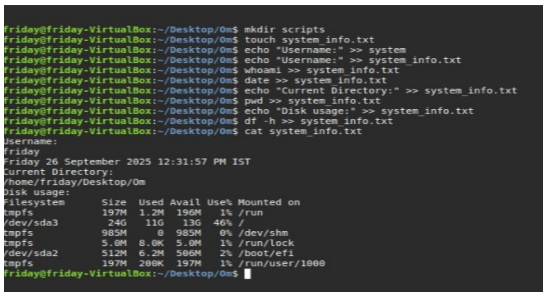
- [I can use whoami to check my username, date to see the current date, and pwd to know my current directory. To check disk usage, I use df -h. I can save the output of any command to a file by using redirection like command >>

filename.txt. If I want to add labels, I use echo like this: echo "Username:" >> file.txt.]

Command(s):

```
cd scripts touch system_info.txt echo "Username:" >>
system_info.txt whoami >> system_info.txt echo "Date:" >>
system_info.txt date >> system_info.txt echo "Current Directory:"
>> system_info.txt pwd >> system_info.txt echo "Disk Usage:" >>
system_info.txt df -h >> system_info.txt
```

Output:



TASK 8: [File Organisation]

Task Statement:

- [In your test_project directory, create a backup folder. Copy all .txt files from all subdirectories into this backup folder. Then list all files in the backup folder with detailed information.]

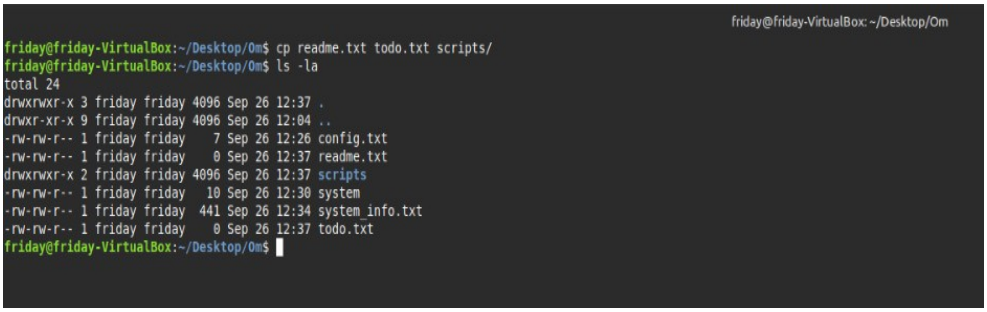
Explanation:

- [I can use find . -name "*.txt" to locate all .txt files. Alternatively, I can navigate to each directory and copy files manually. To copy multiple files at once, I use cp file1.txt file2.txt destination/. If I want detailed information about the files, I use ls -la. The wildcard *.txt helps me match all files that end with .txt.]

Command(s):

```
cp test_project/data/project_info.txt test_project/docs/notes.txt
test_project/docs/readme.txt test_project/docs/todo.txt
test_project/scripts/config.txt test_project/scripts/numbers.txt
test_project/scripts/system_info.txt test_project/scripts/
todo.txt backup/
```

Output:



TASK 9: [Process and History]

Task Statement:

- [Display your command history and count how many commands you've executed. Then show the top 10 most recent commands.]

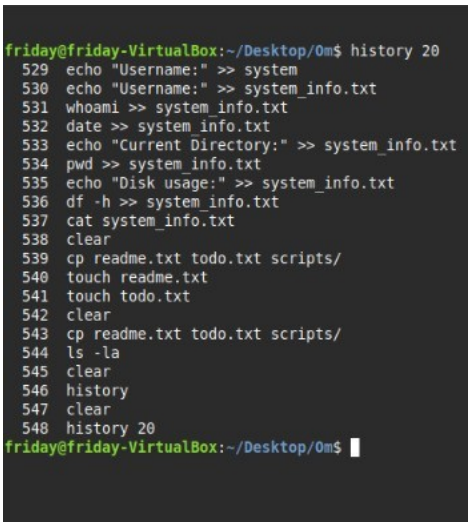
Explanation:

- [I can use history to see all the commands I’ve typed. To count the total number of commands, I use history | wc -l. If I want to view just the last 10 commands, I can use history 10 or history | tail -10. The wc -l command simply counts the number of lines in the output.]

Command(s):

```
history 10
```

Output:



TASK 10: [Comprehensive Cleanup]

Task Statement:

- [Set the permissions of your backup.sh script to be readable, writable, and executable by owner, readable and executable by group, and readable by others. Then create a summary file that lists the total number of files and directories in your entire test_project.]

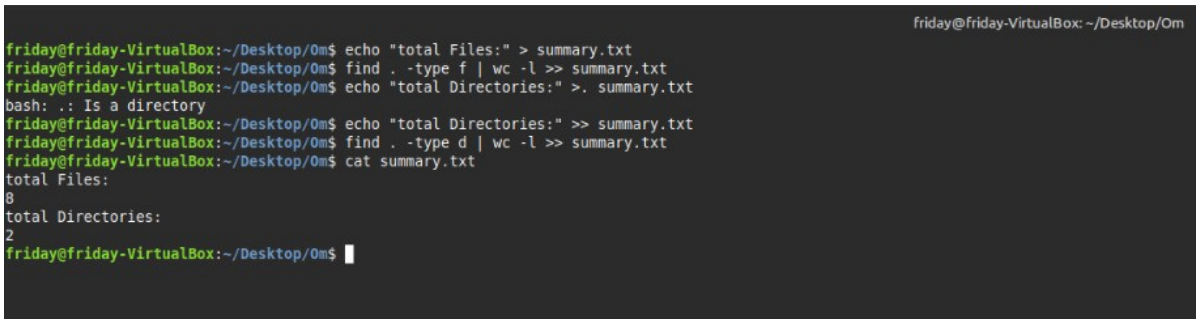
Explanation:

- [I can set permissions for backup.sh using chmod 754 backup.sh to give rwxr-xr-- permissions. Alternatively, I can use chmod u=rwx,g=rx,o=r backup.sh. To count all files, I use find . -type f | wc -l, and to count directories, I use find . -type d | wc -l. If I want to see the full directory structure recursively, I use ls -R. I can also combine multiple commands with && or save the outputs to a summary file for later reference.]

Command(s):

```
chmod 754 backup.sh echo "Total files:" > summary.txt find .
-type f | wc -l >> summary.txt echo "Total directories:" >>
summary.txt find . -type d | wc -l >> summary.txt
```

Output:



Experiment 3: Linux File Manipulation and System Manipulation I

Name: Omkareshwar Chaubey Roll No.: 590025556 Date: 2025-09-23

Aim:

- To practice Linux file manipulation commands like `touch`, `cp`, `mv`, `rm`, `cat`, `less`, `head`, `tail`.
- To explore file permissions and ownership with `ls -l`, `chmod`, `chown`, and `chgrp`.
- To search and filter files using `find` and `grep`.
- To understand archiving and compression with `tar`, `gzip`, and `gunzip`.
- To create and manage links (`ln`) for both hard and symbolic links.

Requirements

- A Linux machine with bash shell (Ubuntu/Fedora/other).
- User privileges to create, modify, and delete files and directories.
- Access to system utilities like `tar`, `gzip`, `grep`, and `find`.

Theory

Linux file management involves creating, copying, moving, removing, and viewing files. File permissions and ownership ensure secure access control. Searching and filtering tools like `grep` and `find` help locate information efficiently. Archiving with `tar` and compression with `gzip` reduce storage usage and simplify file transfer. Links (`ln`) allow multiple references to the same file data (hard links) or path references (symbolic links).

Procedure & Observations

Exercise 1: Creating and Managing Files

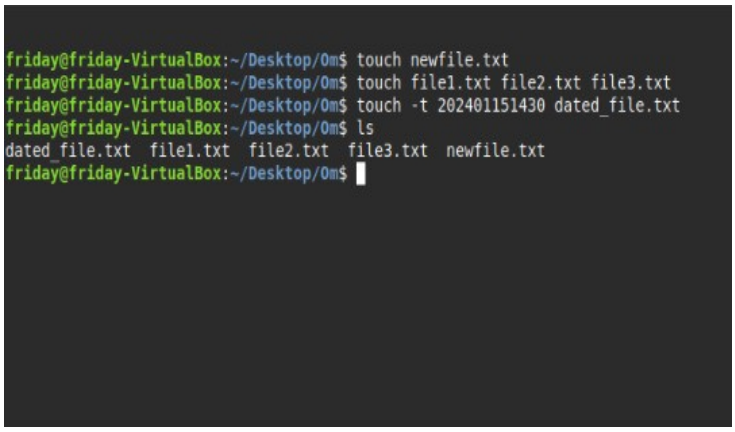
Task Statement:

Create files and manage timestamps using `touch`.

Command(s):

```
touch newfile.txt
touch file1.txt file2.txt file3.txt
touch -t 202401151430 dated_file.txt
```

Output:



Exercise 2: Copying, Moving, and Deleting Files

Task Statement:

Use `cp`, `mv`, and `rm` to copy, rename, move, and delete files and directories.

Command(s):

```
cp document.txt backup_document.txt
mv oldname.txt newname.txt
rm unwanted_file.txt
rm -r old_directory/
```

Output:

```
friday@friday-VirtualBox:~/Desktop/Om$ cp file1 file2
cp: -r not specified; omitting directory 'file1'
friday@friday-VirtualBox:~/Desktop/Om$ cp -r file1 file2
friday@friday-VirtualBox:~/Desktop/Om$ cp myfile.txt -/documents/
cp: invalid option -- '/'
Try 'cp --help' for more information.
friday@friday-VirtualBox:~/Desktop/Om$ touch myfile.txt documents
friday@friday-VirtualBox:~/Desktop/Om$ cp myfile.txt documents
friday@friday-VirtualBox:~/Desktop/Om$ cp -v myfile.txt documents
'myfile.txt' -> 'documents'
friday@friday-VirtualBox:~/Desktop/Om$ touch original.txt
friday@friday-VirtualBox:~/Desktop/Om$ cp original.txt backup_original.txt
friday@friday-VirtualBox:~/Desktop/Om$ cp file2 backup_file.txt
cp: -r not specified; omitting directory 'file2'
friday@friday-VirtualBox:~/Desktop/Om$ cp -r file2 backup_file.txt
friday@friday-VirtualBox:~/Desktop/Om$
```

Exercise 3: Viewing File Contents

Task Statement:

Display file contents using `cat`, `less`, `head`, and `tail`.

Command(s):

```
cat filename.txt
less /var/log/syslog
head -n 5 filename.txt
tail -n 20 filename.txt
tail -f /var/log/syslog
```

Output:

Exercise 4: File Permissions and Ownership

Task Statement:

Explore file permissions and ownership with `ls -l`, `chmod`, `chown`, and `chgrp`.

Command(s):

```
ls -l
chmod 755 script.sh
chmod u+x script.sh
sudo chown newuser:newgroup file.txt
chgrp developers project.txt
```

Output:

Exercise 5: File Searching with `find`

Task Statement:

Search files by name, type, size, and permissions using `find`.

Command(s):

```
find /home -name "*.txt"
find /home -type f -size +100M
find /etc -name "*conf*"
find /tmp -type f -empty -delete
```

Output:

```
friday@friday-VirtualBox:~/Desktop/linux$ bash act1.sh
What is your age
17
You are not eligible to vote!
friday@friday-VirtualBox:~/Desktop/linux$ bash act1.sh
What is your age
20
You are eligible to vote!
friday@friday-VirtualBox:~/Desktop/linux$
```

Exercise 6: Pattern Searching with `grep`

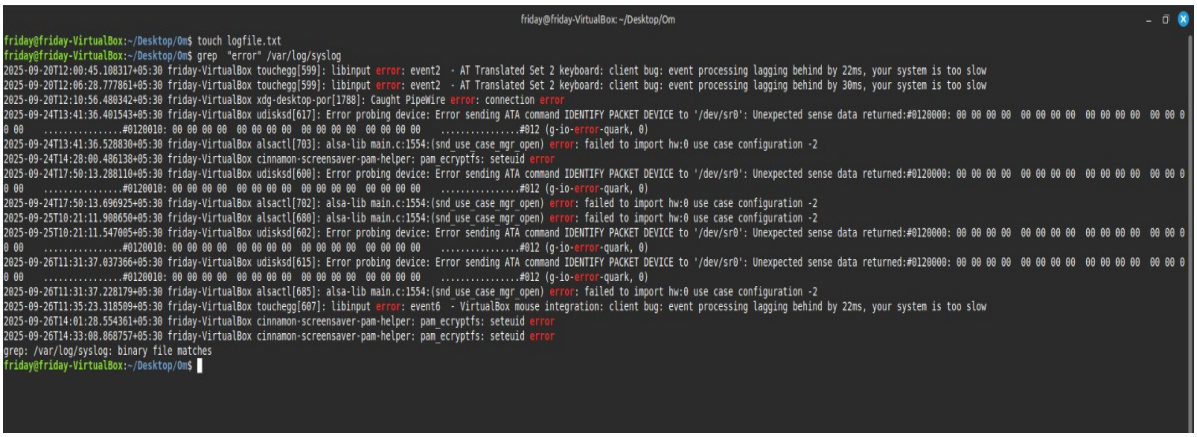
Task Statement:

Search for patterns in files using `grep`.

Command(s):

```
grep "error" /var/log/syslog
grep -i "Error" logfile.txt
grep -r "function" ~/code/
grep -n "TODO" *.txt
```

Output:



Exercise 7: Archiving and Compression

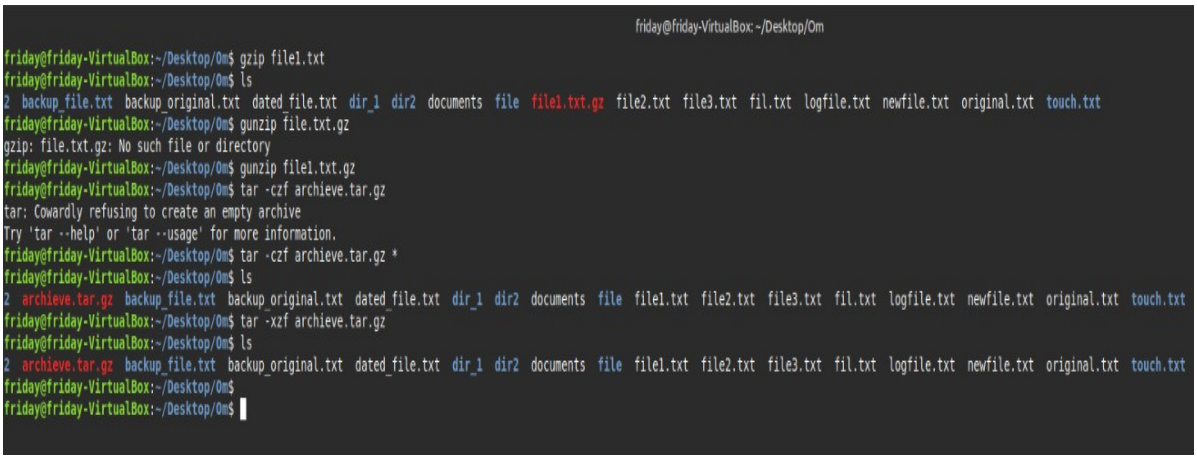
Task Statement:

Create and extract archives using `tar`, compress and decompress with `gzip`/`gunzip`.

Command(s):

```
tar -czf backup.tar.gz /home/user/documents
tar -xzf backup.tar.gz -C /restore/
gzip largefile.txt
gunzip largefile.txt.gz
```

Output:



Exercise 8: Creating Links

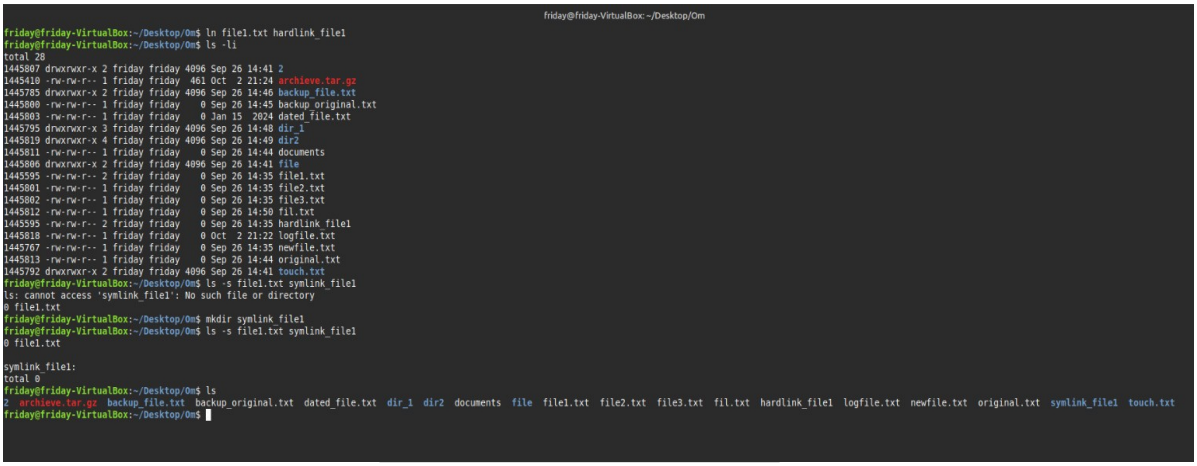
Task Statement:

Create and test hard and symbolic links using `ln`.

Command(s):

```
echo "Hello" > original.txt
ln original.txt hardlink.txt
ln -s original.txt symlink.txt
ls -li original.txt hardlink.txt symlink.txt
```

Output:



Result

- Successfully created, copied, moved, and deleted files.
- Practiced viewing file contents and monitoring logs.
- Explored file permissions and ownership management.
- Used `find` and `grep` to locate and filter data.
- Created archives and compressed files.
- Demonstrated both hard and symbolic links.

Challenges Faced & Learning Outcomes

- Challenge 1: Accidentally deleted files with `rm` without `-i`. Learned to use `rm -i` for safety.
- Challenge 2: Remembering numeric vs symbolic permissions in `chmod`. Fixed through repeated practice.

Learning:

- Gained practical skills with file manipulation and permission commands.
- Learned how to efficiently search files and patterns in Linux.
- Understood how to archive and compress files for better storage management.
- Understood differences between hard and symbolic links.

Conclusion

This experiment provided hands-on experience with core Linux file management, permissions, searching, archiving, and linking. These are foundational skills for effective Linux system administration and daily usage.

Experiment [4]: [Bash Scripting]

Name:Omkareshwar Chaubey, Roll No.: 59002556, Date: 2025-09-04

AIM:

- [To Learn Basics of Bash Scripting.]

Requirements:

- [Any Linux Distro, any kind of text editor (vs code, vim, notepad, nano, etc)]

Theory:

- [Learning the basics of bash scripting.]

Procedure & Observations

Exercise 1: [Hello World Script]

Task Statement:

- [Basic Usage of Shell Scripts]

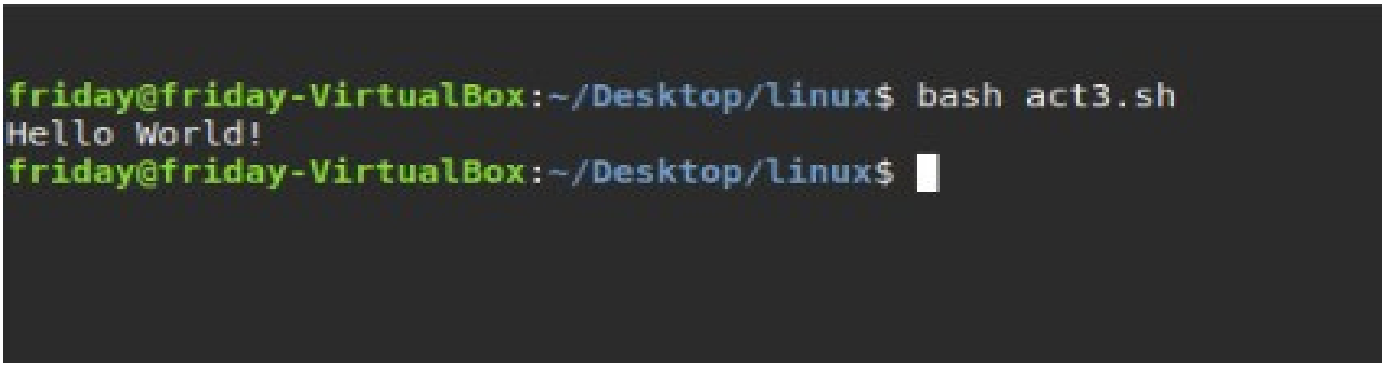
Explanation:

- [Writing Begginer level Shell Scripts]

Command(s):

```
#!/bin/bash echo "Hello, World!"
```

Output:



Exercise 2: [Personalized Greeting Script]

Task Statement:

- [Basic Shell Script to callout user defined function.]

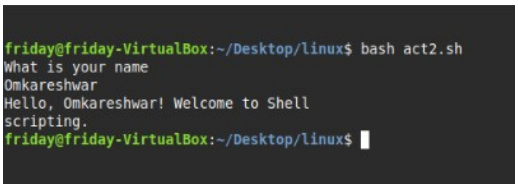
Explanation:

- [This Shell script will take input from user and store it in a variable and then call the variable which will output the stored value.]

Command(s):

```
#!/bin/bash echo "What is your name?" read name echo "Hello, $name! Welcome to Shell Scripting."
```

Output:



Exercise 3: [Arithmetic Operations in Shell Scripting]

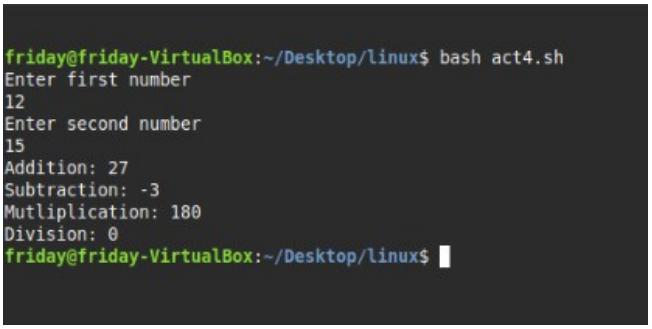
Task Statement:

- [Using Basic Arithmetic Operations in Shell Scripts]

Command(s):

```
#!/bin/bash echo "Enter first number: " read num1 echo "Enter second number: " read num2 echo "Addition: $((num1 + num2))" echo "Subtraction: $((num1 - num2))" echo "Multiplication: $((num1 * num2))" echo "Division: $((num1 / num2))"
```

Output:



Exercise 4:

- [Voting Eligibility]

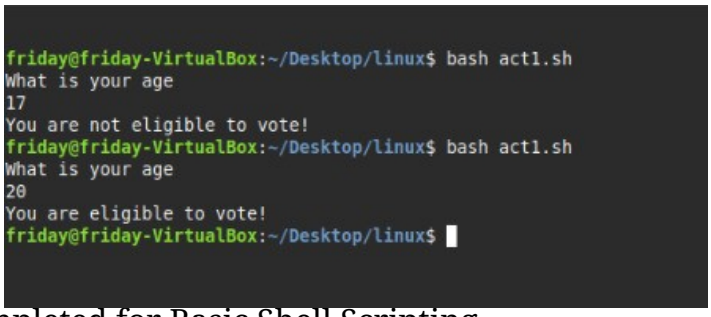
Task Statement:

- [Using Conditionals in Shell script]

Command(s):

```
#!/bin/bash echo "What is your age?" read age if [ $age -ge 18 ]; then echo "You are eligible to vote!" else echo "You are not eligible to vote!" fi
```

Output:



Result

- The Exercises were successfully completed for Basic Shell Scripting

Experiment [5]: [Shell Programming]

Name: Omkareshwar Chaubey Roll.: 590025556 Date: 2025-10-05 AIM:

- [To Learn Basic Conditional Statements in Bash Scripting]

Requirements:

- [Any Linux Distro, any kind of text editor (vs code, vim, notepad, nano, etc)]

Theory:

- [Basic usage of conditions and arrays in bash scripting.]

Procedure & Observations

Exercise 1: [Prime Number Check]

Task Statement:

- [To check if the number given by the user is a prime number or not.]

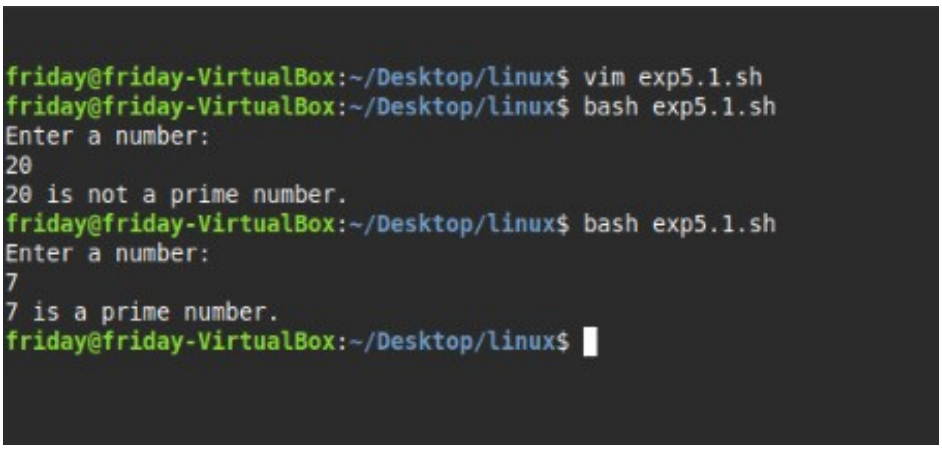
Explanation:

- [using if else loop wap to check if the number is a prime number or not.]

Command(s):

```
#!/bin/bash echo "Enter a number: " read num flag=0 for ((i=2; i<=num/2; i++)) do if [ $((num % i)) -eq 0 ] then flag=1 break fi done if [ $flag -eq 0 ] then echo "$num is a prime number." else echo "$num is not a prime number." fi
```

Output:



Exercise 2: [Sum of Digits]

Task Statement:

- [Take input from user and give the sum of two digits.]

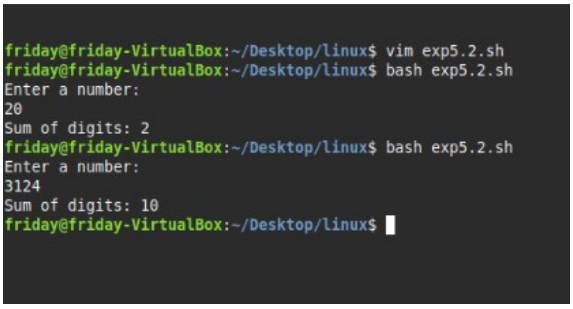
Explanation:

- [This script will take input from user and will give the following output.]

Command(s):

```
#!/bin/bash echo "Enter a number: " read num sum=0 while [ $num
-gt 0 ] do digit=$((num % 10)) sum=$((sum + digit)) num=$((num /
10)) done echo "Sum of digits: $sum"
```

Output:



Exercise 3: [Armstrong Numbers]

Task Statement:

- [Take input user and give the sum of Armstrong number of n digits is a number equal to the sum of its digits raised to the power n. Example: 153 = 1^3^ + 5^3^ + 3^3^]

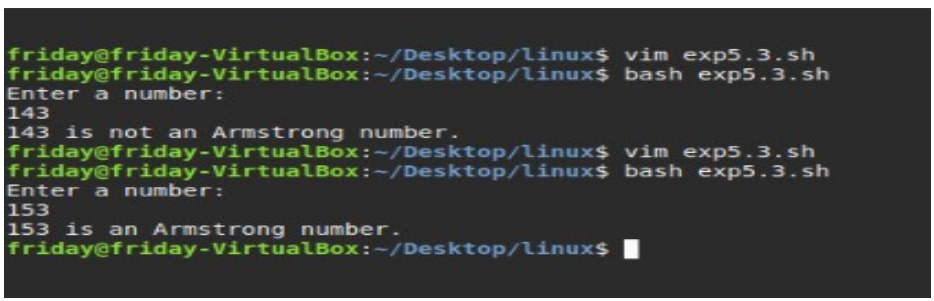
Explanation:

- [This script will tell if the number entered by the user is an armstrong number or not.]

Command(s):

```
#!/bin/bash echo "Enter a number: " read num temp=$num n=${#num}
# number of digits sum=0 while [ $temp -gt 0 ] do digit=$((temp %
10)) sum=$((sum + digit**n)) temp=$((temp / 10)) done if [ $sum
-eq $num ] then echo "$num is an Armstrong number." else echo
"$num is not an Armstrong number." fi
```

Output:



Result:

- The Exercises were successfully completed for Basic Shell Scripting.

Experiment 6: Shell Loops

Name:Omkareshwar Chaubey Roll No.: 590025556 Date: 2025-09-23

Aim:

- To understand and implement shell loops (`for`, `while`, `until`) in Bash.
- To practice loop control constructs (`break`, `continue`) and loop-based file processing.

Requirements

- A Linux system with bash shell.
- A text editor (nano, vim) and permission to create and execute shell scripts.

Theory

Loops allow repeated execution of commands until a condition is met. Common loop constructs in Bash include `for` (iterate over items), `while` (repeat while condition true), and `until` (repeat until condition becomes true). Loop control statements like `break` and `continue` change the flow inside loops. Loops are essential for automating repetitive tasks such as processing multiple files, generating sequences, and collecting user input.

Procedure & Observations

Exercise 1: Simple `for` loop

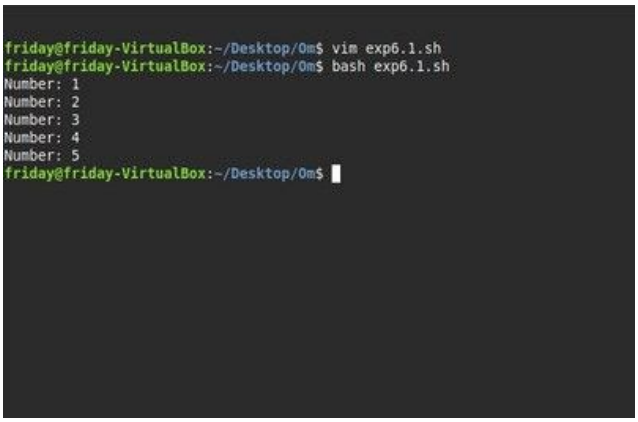
Task Statement:

Write a `for` loop that prints numbers 1 to 5.

Command(s):

```
for i in 1 2 3 4 5; do
  echo "Number: $i"
done
```

Output:



Exercise 2: `for` loop over files

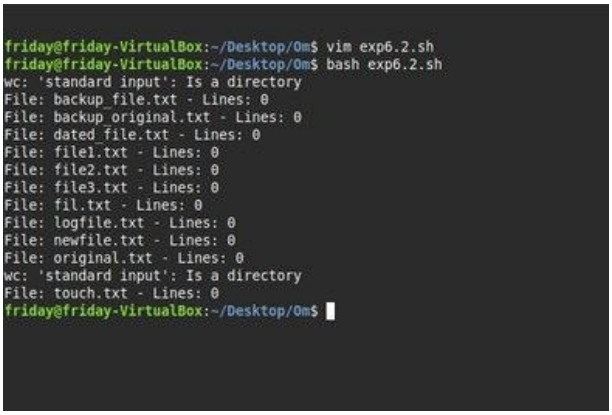
Task Statement:

Process all `.txt` files in a directory and count lines in each.

Command(s):

```
for f in *.txt; do
  echo "File: $f - Lines: $(wc -l < "$f")"
done
```

Output:



Exercise 3: C-style `for` loop

Task Statement:

Use arithmetic C-style loop for numeric iteration.

Command(s):

```
for ((i=0;i<5;i++)); do
  echo "i=$i"
done
```

Output:

Exercise 4: `while` loop and reading input

Task Statement:

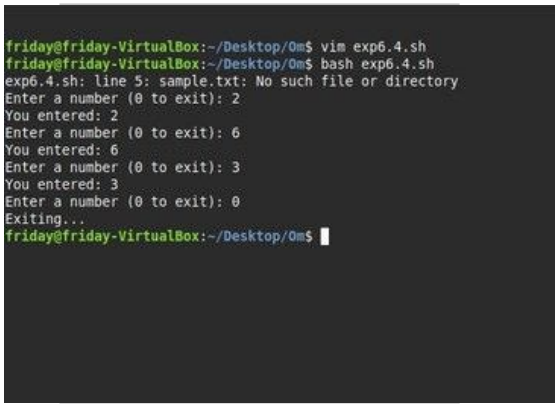
Write a `while` loop that reads lines from a file or from user input.

Command(s):

```
# Read from file
while read -r line; do
  echo "Line: $line"
done < sample.txt

# Read from user with exit condition
while true; do
  read -p "Enter a number (0 to exit): " n
  if [[ $n -eq 0 ]]; then
    echo "Exiting..."; break
  fi
  echo "You entered: $n"
done
```

Output:



Exercise 5: `until` loop

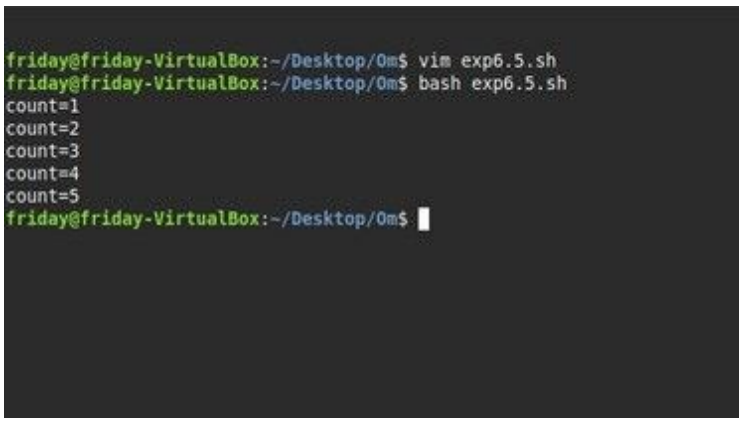
Task Statement:

Use an `until` loop to run until a condition becomes true.

Command(s):

```
count=1
until [ $count -gt 5 ]; do
  echo "count=$count"
  ((count++))
done
```

Output:



Exercise 6: `break` and `continue`

Task Statement:

Demonstrate `break` and `continue` inside a loop.

Command(s):

```
for i in {1..10}; do
  if [[ $i -eq 5 ]]; then
    echo "Reached 5, breaking"; break
  fi
  if (( i % 2 == 0 )); then
    echo "Skipping even $i"; continue
  fi
  echo "Processing $i"
done
```

Output:

Exercise 7: Nested loops

Task Statement:

Create nested loops to generate a multiplication table.

Command(s):

```
for i in {1..3}; do
  for j in {1..3}; do
    echo -n "$((i*j)) "
  done
  echo
done
```

Output:

Result

- Implemented `for`, `while`, and `until` loops and used loop control statements.
- Practiced reading input, processing files, and nested iteration.

Challenges Faced & Learning Outcomes

- Challenge 1: Handling spaces and special characters when iterating filenames — learned to use quotes and `read -r`.
- Challenge 2: Remembering arithmetic syntax in Bash — used `(( ))` and `expr` where needed.

Learning:

- Loops are powerful for automation in shell scripting. Correct quoting and use of control constructs prevent common bugs.

Conclusion

The lab demonstrated practical loop constructs in Bash for automating repetitive tasks and processing data efficiently.

Experiment 7: Shell Programming, Process and Scheduling

Name: Omkareshwar Chaubey Roll No.: 590025556 Date: 2025-09-23

Aim:

- To write shell scripts that demonstrate process management.
- To understand how to schedule processes using `cron` and `at`.
- To monitor running processes and practice job control commands.

Requirements

- A Linux machine with bash shell.
- Access to process management commands (`ps`, `top`, `kill`, `jobs`, `fg`, `bg`).
- Access to scheduling utilities (`cron`, `at`).

Theory

Every program running in Linux is a process identified by a unique process ID (PID). Shell programming allows automation of tasks including spawning and controlling processes. Process management commands like `ps`, `top`, `kill`, `jobs`, `bg`, and `fg` let users monitor and control execution. Scheduling utilities such as `cron` (repeated tasks) and `at` (one-time tasks) allow tasks to run automatically at defined times. Combining scripting with scheduling is a core system administration skill.

Procedure & Observations

Exercise 1: Writing a basic shell script

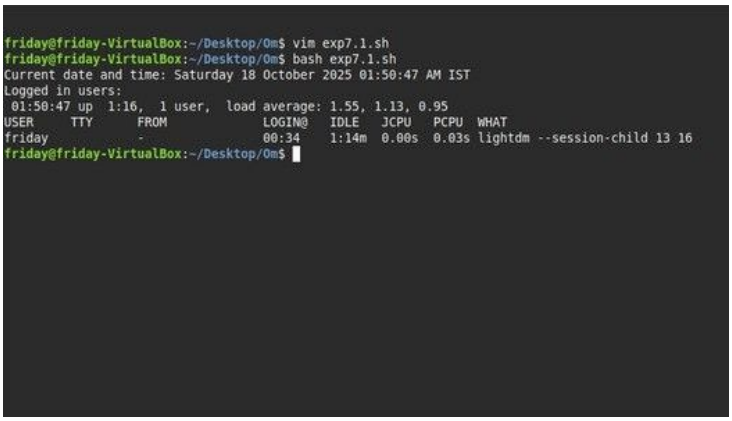
Task Statement:

Create a shell script that prints the current date, time, and the list of logged-in users.

Command(s):

```
#!/bin/bash
echo "Current date and time: $(date)"
echo "Logged in users:"
w
```

Output:



Exercise 2: Background and foreground processes

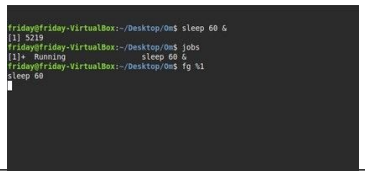
Task Statement:

Run a process in background and bring it to the foreground.

Command(s):

```
sleep 60 &
jobs
fg %1
```

Output:



Exercise 3: Killing a process

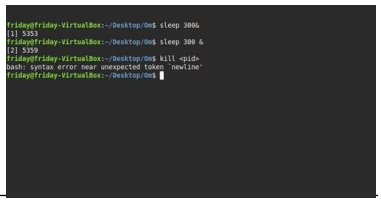
Task Statement:

Start a process and terminate it using `kill`.

Command(s):

```
sleep 300 &
ps aux | grep sleep
kill <pid>
```

Output:



Exercise 4: Monitoring processes

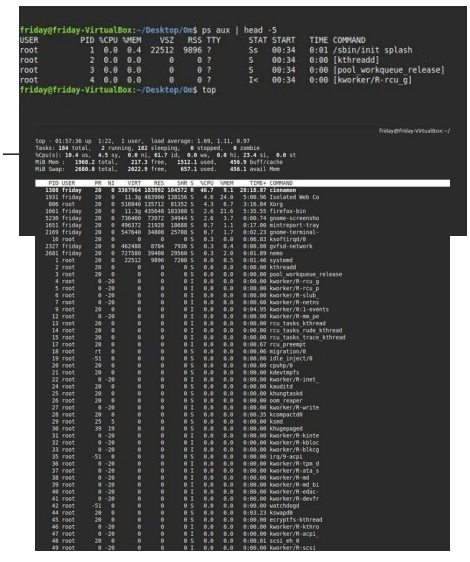
Task Statement:

Use `ps` and `top` to monitor processes.

Command(s):

```
ps aux | head -5
top
```

Output:



Exercise 5: Using `cron` for scheduling

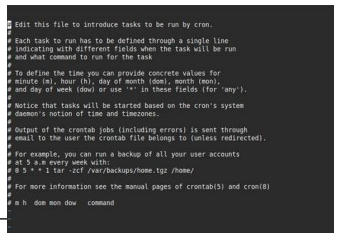
Task Statement:

Schedule a script to run every day at 7:00 AM using `cron`.

Command(s):

```
crontab -e
# Add the following line
0 7 * * * /home/user/myscript.sh
```

Output:



Exercise 6: Using `at` for one-time scheduling

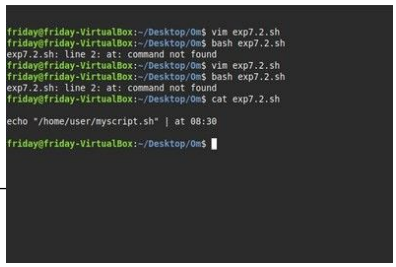
Task Statement:

Schedule a script to run once at a specified time using `at`.

Command(s):

```
echo "/home/user/myscript.sh" | at 08:30
atq
```

Output:



Result

- Learned to create and run shell scripts.
- Managed processes using background, foreground, and kill commands.
- Monitored processes with `ps` and `top`.
- Scheduled recurring tasks with `cron` and one-time tasks with `at`.

Challenges Faced & Learning Outcomes

- Challenge 1: Remembering the `crontab` time format. Solved by using online crontab generators and practice.
- Challenge 2: Ensuring `atd` service is running for `at` command. Fixed by starting the service with `systemctl start atd`.

Learning:

- Gained hands-on knowledge of process creation and termination.
- Learned job control and scheduling using `cron` and `at`.

Conclusion

This experiment provided practical experience with shell scripting, process management, and scheduling. These are critical skills for system administrators to automate and control Linux environments effectively.

Experiment 8: Shell Programming (Continued)

Name: Omkareshwar Chaubey Roll No.: 590025556 Date: 2025-09-23

Aim:

- To extend shell programming concepts by using conditional statements, advanced scripting constructs, and command-line arguments.
- To practice writing scripts that perform decision-making and parameter handling.

Requirements

- A Linux system with bash shell.
- Text editor and permission to create/execute shell scripts.

Theory

Conditional execution in shell scripts allows branching logic using `if`, `elif`, `else`, and `case` statements. Scripts can accept command-line arguments using `$1`, `$2`, … and `$@` for all arguments. Control flow constructs combined with user input and arguments allow dynamic and reusable scripts.

Procedure & Observations

Exercise 1: Using `if-else`

Task Statement:

Write a script to check whether a given number is positive, negative, or zero.

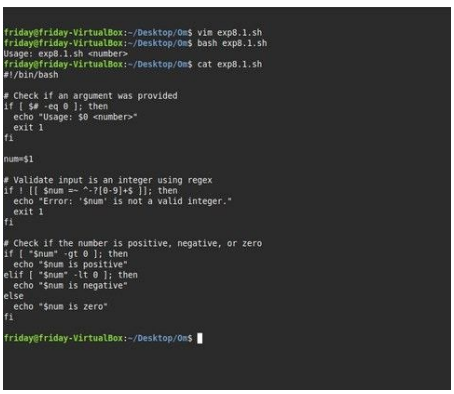
Explanation:

We used an `if-elif-else` construct to compare the number against 0.

Command(s):

```
#!/bin/bash
num=$1
if [ $num -gt 0 ]; then
  echo "$num is positive"
elif [ $num -lt 0 ]; then
  echo "$num is negative"
else
  echo "$num is zero"
fi
```

Output:



Exercise 2: Using `case`

Task Statement:

Write a script that takes a character as input and classifies it as vowel, consonant, digit, or special character.

Explanation:

The `case` statement provides pattern matching for multiple options.

Command(s):

```
#!/bin/bash
ch=$1
case $ch in
  [aeiouAEIOU]) echo "$ch is a vowel" ;;
  [bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ]) echo "$ch is a consonant" ;;
  [0-9]) echo "$ch is a digit" ;;
  *) echo "$ch is a special character" ;;
esac
```

Output:

Exercise 3: Command-line arguments

Task Statement:

Write a script that accepts filename(s) as arguments and prints the number of lines in each file.

Explanation:

Command-line arguments are accessed using `$@`. Looping through each argument allows file-wise operations.

Command(s):

```
#!/bin/bash
for file in "$@"; do
  if [ -f "$file" ]; then
    echo "$file: $(wc -l < "$file") lines"
  else
  echo "$file not found"
  fi
  done
```

Output:

```
friday@friday-VirtualBox:~/Desktop/Om$ vim exp8.3.sh
friday@friday-VirtualBox:~/Desktop/Om$ bash exp8.3.sh
friday@friday-VirtualBox:~/Desktop/Om$ cat exp8.3.sh
#!/bin/bash
for file in "$@"; do
  if [ -f "$file" ]; then
    echo "$file: $(wc -l < "$file") lines"
  else
    echo "$file not found"
  fi
done
friday@friday-VirtualBox:~/Desktop/Om$
```

Exercise 4: Nested conditionals

Task Statement:

Write a script to check if a year is a leap year.

Explanation:

A leap year is divisible by 4, but if divisible by 100 it must also be divisible by 400.

Command(s):

```
#!/bin/bash
year=$1
if (( year % 400 == 0 )); then
  echo "$year is a leap year"
elif (( year % 100 == 0 )); then
  echo "$year is not a leap year"
elif (( year % 4 == 0 )); then
  echo "$year is a leap year"
else
  echo "$year is not a leap year"
fi
```

Output:

```
friday@friday-VirtualBox:~/Desktop/Om$ vim exp8.4.sh
friday@friday-VirtualBox:~/Desktop/Om$ bash exp8.4.sh
Usage: exp8.4.sh <year>
friday@friday-VirtualBox:~/Desktop/Om$ cat exp8.4.sh
#!/bin/bash

# Check if an argument is provided
if [ $# -eq 0 ]; then
  echo "Usage: $0 <year>"
  exit 1
fi

year=$1

# Validate that the input is a positive integer
if ! [[ "$year" =~ ^[0-9]+$ ]]; then
  echo "Error: '$year' is not a valid year."
  exit 1
fi

# Leap year logic
if (( year % 400 == 0 )); then
  echo "$year is a leap year"
elif (( year % 100 == 0 )); then
  echo "$year is not a leap year"
elif (( year % 4 == 0 )); then
  echo "$year is a leap year"
else
  echo "$year is not a leap year"
fi
friday@friday-VirtualBox:~/Desktop/Om$
```

Result

- Implemented conditional statements (`if-else`, `case`) in shell scripts.
- Practiced handling command-line arguments and nested conditions.
- Wrote reusable and flexible shell scripts.

Challenges Faced & Learning Outcomes

- Challenge 1: Forgetting to quote variables in conditions — resolved by using `"$var"` to avoid word splitting.

- Challenge 2: Pattern matching in `case` — practiced with multiple examples.

Learning:

- Learned practical use of branching and decision-making in shell scripting.
- Understood command-line argument handling for automation.

Conclusion

This experiment extended shell programming by introducing decision-making and parameter handling. The scripts demonstrate the flexibility of shell programming for different use cases.