Module 1

1. Linux Operating System (OS)

Definition:
Linux is an open-source, UNIX-like operating system that is widely used for a variety of purposes, from servers and desktop systems to embedded devices. It provides multitasking, multi-user capabilities, and networked environment support.

Key Components of Linux:

- **Kernel**:
  The core of the operating system. The kernel manages hardware, system resources, and communication between hardware and software. The kernel handles:
  - **Memory Management**: Allocates and frees memory as required by processes.
  - **Process Management**: Schedules processes, manages CPU time, and controls multitasking.
  - **File System Management**: Provides access to files and directories.
  - **Device Drivers**: Interfaces between hardware and the operating system.
- **Shell**:
  The command-line interface that allows users to interact with the kernel. There are different types of shells like:
  - **Bash (Bourne Again Shell)**: The default shell for most Linux distributions.
  - **Zsh, Csh**: Other shells with different syntax and capabilities.
  - The shell interprets user commands and translates them into instructions for the kernel.
- **File System**:
  Linux follows a hierarchical file system structure where everything is treated as a file, including hardware devices. Key features of the Linux file system include:
  - **Root Directory (`/`)**: The top level of the file system hierarchy.
  - **Home Directory**: Each user has a home directory to store personal files (`/home/user`).
  - **Inodes**: Data structures that store metadata about files (permissions, owner, size, and location).

Linux supports various file systems like **ext4, ext3**, and **XFS**.

2. Linux Features

Open Source:

- **Free and Customizable**:
  Linux's source code is freely available for anyone to modify, distribute, or use as they wish. This has led to the development of a wide variety of Linux distributions (Ubuntu, Fedora, CentOS, etc.), each with specific use cases.

Multitasking:

- **Multiple Processes**:
  Linux supports the execution of multiple processes at the same time. The kernel efficiently schedules processes, allocating CPU time and memory resources.

Multi-User Support:

- **User and Group Management**:
  Linux allows multiple users to log in and use system resources simultaneously. Each user has their own permissions, preventing unauthorized access to files and processes.

Security:

- **File Permissions**:
  Linux uses a powerful permission system that controls access to files based on three categories: owner, group, and others. Permissions include:
  - **Read (r)**: Allows viewing the file's contents.
  - **Write (w)**: Allows modifying the file.
  - **Execute (x)**: Allows executing the file if it's a script or program.
- **Process Isolation**:
  Each process runs in its own memory space, isolated from other processes. This prevents unauthorized access between processes, enhancing system security.
- **SSH (Secure Shell)**:
  Linux supports secure remote connections using SSH, which is essential for administering systems over a network.

3. Linux Utilities

Linux comes with a variety of utilities for managing files, processes, and network configurations. Here are some important utilities:

File Handling Utilities:

- **cp**: Copies files from one location to another.
- **mv**: Moves or renames files.
- **rm**: Deletes files.
- **mkdir**: Creates new directories.

Process Management Utilities:

- **ps**: Displays a list of currently running processes.
- **top**: Shows real-time information about system processes and resource usage.
- **kill**: Sends a signal to terminate a process.

Networking Utilities:

- **ifconfig**: Configures or displays network interfaces.
- **ping**: Tests network connectivity to a remote host.
- **netstat**: Displays network connections and statistics.

Backup Utilities:

- **tar**: Archives multiple files into a single file (tarball) for backup.

- **rsync**: Synchronizes files and directories across systems, often used for backups.

4. Linux File Permissions

In Linux, file permissions are crucial for system security. Each file has three sets of permissions for the **owner**, **group**, and **others**:

- **r** (read), **w** (write), **x** (execute).
- Permissions are set using the `chmod` command.
    - Example: `chmod 755 file` – sets **rwx** for the owner and **r-x** for others.

Octal Notation:

- Linux permissions can also be represented using numeric values (octal notation):
    - **r = 4**, **w = 2**, **x = 1**.
    - Example: **755** means the owner has full permissions (7 = rwx), and the group and others have read and execute permissions (5 = r-x).

5. Importance of Linux in Cybersecurity

Since Linux is widely used in servers and networking environments, it is also a key part of cybersecurity. Its robust permission model, support for encrypted connections (SSH), and extensive logging tools make it ideal for secure environments. Cybersecurity specialists often use Linux to perform tasks like:

- **System hardening**: Reducing the attack surface by disabling unused services and setting strict permissions.
- **Intrusion detection**: Using tools like **Wireshark** or **Nmap** for network traffic analysis.
- **Firewall configuration**: Using **iptables** to manage inbound and outbound traffic.

6. Key Components of Linux Architecture

1. **Kernel**:
    - **Role**: The kernel is the core of the Linux OS, responsible for managing all critical operations like memory management, process scheduling, and device drivers.
    - **Functions**:
        - **Process Management**: Controls process creation, termination, and multitasking.
        - **Memory Management**: Allocates and deallocates memory to processes, ensuring efficient use of system resources.
        - **File System Management**: Handles the way files are stored and accessed on disks.
        - **Device Drivers**: Acts as an intermediary between hardware and the OS.
2. **System Libraries**:
    - These are special functions or programs that application programs use to access kernel features. For example, `libc` is a standard library for C language programs.
3. **Shell**:
    - The **shell** is the interface between the user and the kernel. It interprets user commands and communicates with the kernel to execute them.

- **Types**: There are several types of shells, such as **Bash**, **C shell**, and **Korn shell**, each providing different features.

4. **File System**:
    - Linux uses a hierarchical file system structure, with `/` as the root directory. Everything in Linux (files, directories, devices) is organized under this tree.
    - **Inodes**: Linux tracks files using **inodes**, which store metadata like file size, permissions, and pointers to data blocks.
5. **User Space**:
    - This is the space where user applications and processes run. These interact with the kernel through system calls.
6. **System Calls**:
    - System calls provide an interface between user programs and the operating system. For example, `open()`, `read()`, and `write()` are system calls for file handling.

1. File Handling Commands:

- `cp`: Copy files.
    - `-r`: Recursive copy (for directories).
    - `-i`: Prompt before overwriting.
    - `-u`: Copy only when the source is newer or the destination is missing.
    - `-v`: Verbose output.
    - Example: `cp -r folder1 folder2`
- `mv`: Move or rename files.
    - `-i`: Prompt before overwriting.
    - `-f`: Force move without prompt.
    - `-u`: Move only if the source is newer.
    - `-v`: Verbose output.
    - Example: `mv -i file1 /path/to/destination/`
- `rm`: Remove files.
    - `-r`: Remove directories recursively.
    - `-i`: Prompt before each removal.
    - `-f`: Force removal (ignore nonexistent files).
    - `-d`: Remove an empty directory.
    - Example: `rm -rf dir1`
- `chmod`: Change file permissions.
    - `-R`: Change permissions recursively.
    - `u`, `g`, `o`, `a`: Set permissions for user, group, others, or all.
    - Example: `chmod u+x file.sh` or `chmod 755 file.sh`
- `chown`: Change file ownership.
    - `-R`: Apply ownership change recursively.

    - Example: `chown -R user:group /path/to/directory/`
- `ln`: Create links to files.
    - `-s`: Create symbolic (soft) links.
    - Example: `ln -s /path/to/file link_name`

2. Text Processing Commands:

- `grep`: Search patterns in files.
    - `-i`: Case-insensitive search.
    - `-r`: Recursive search.
    - `-v`: Invert match (show lines that do not match).
    - `-n`: Show line numbers.
    - `-c`: Count matching lines.
    - Example: `grep -i "pattern" file.txt`
- `awk`: Text scanning and processing.
    - `'{print $1}'`: Print first column.
    - `-F`: Specify input field separator.
    - Example: `awk -F: '{print $1}' /etc/passwd`
- `sed`: Stream editor for filtering and transforming text.
    - `'s/old/new/g'`: Replace all occurrences of "old" with "new".
    - `-i`: Edit files in place.
    - Example: `sed -i 's/error/fix/g' file.txt`

3. Process Management Commands:

- `ps`: Display process information.
    - `-e`: Show all processes.
    - `-f`: Full-format listing.
    - Example: `ps -ef`
- `top`: Display dynamic process information.
    - `k`: Kill a process.
    - `q`: Quit.
    - Example: `top` (interactive)
- `kill`: Terminate processes.
    - `-9`: Force kill.
    - Example: `kill -9 1234`

4. Networking Commands:

- `ifconfig`: Network interface configuration.
    - `up`, `down`: Activate or deactivate an interface.
    - Example: `ifconfig eth0 up`
- `ping`: Test network connectivity.

    - `-c N`: Send N packets.
    - Example: `ping -c 4 google.com`
- `netstat`: Show network connections.
    - `-t`: Display TCP connections.
    - `-u`: Display UDP connections.
    - Example: `netstat -tu`

5. Redirection and Pipes:

- `>`, `>>`: Redirect output to a file (overwrite or append).
- `<`, `<<`: Redirect input from a file.
- Example: `ls > output.txt` (overwrite), `ls >> output.txt` (append)
- **Pipe (`|`)**: Send output of one command to another.
- Example: `ls -l | grep "pattern"`

6. Backup and Synchronization:

- `rsync`: Synchronize files and directories.
    - `-r`: Recursive.
    - `-a`: Archive mode (preserve permissions, timestamps, etc.).
    - `-v`: Verbose output.
    - Example: `rsync -av /source /destination`

Module 2:

1. Need for Shells

- The **shell** is an interface between the user and the operating system. It interprets user commands, executes them in the kernel, and displays the output.
- **Types of Shells**:
  - **Bourne shell (sh)**: One of the earliest Unix shells.
  - **Bourne Again shell (bash)**: An improved version of the Bourne shell and the most common in Linux.
  - **C shell (csh)**: Known for its C-like syntax, useful for programming.
  - **Korn shell (ksh)**: Combines features of the C and Bourne shells.
  - **Z shell (zsh)**: Includes features from bash, ksh, and tcsh, with additional functionalities .

---

2. Redirection

- **Redirection** allows us to change the standard input/output/error streams of commands.
  - **(0) Standard input (`stdin`)**: Default input stream (e.g., keyboard, file).
  - **(1) Standard output (`stdout`)**: Default output stream (e.g., terminal).
  - **(2) Standard error (`stderr`)**: Default error output stream.
- **Redirection Operators**:
  - `>`: Redirects output to a file (overwrites the file).
    - Example: `ls > output.txt` - redirects output of `ls` to `output.txt`.
  - `>>`: Appends output to a file.
    - Example: `echo "Hello" >> output.txt` - appends "Hello" to `output.txt`.
  - `<`: Redirects input from a file.
    - Example: `cat < file.txt` - reads input from `file.txt`.
  - `2>`: Redirects error output to a file.
    - Example: `command 2> error.txt` - redirects errors to `error.txt`.
  - `&>`: Redirects both stdout and stderr to a file.
    - Example: `command &> log.txt` .

---

3. Pipes

- A **pipe (`|`)** allows the output of one command to be the input of another command. This avoids the need for intermediate files.
  - Example: `ls -l | grep "pattern"` - lists files and then filters the results for the pattern.
  - Another example: `cat file.txt | wc -l` - counts the number of lines in a file by passing the output of `cat` to `wc -l` .

---

4. File Operations

- **File Handling Commands**:
  - `touch`: Creates an empty file or updates the timestamp of an existing file.
    - Example: `touch newfile.txt`.
  - `cp`: Copies files from one location to another.
    - Example: `cp source.txt destination.txt`.
  - `mv`: Moves or renames files.
    - Example: `mv file1.txt file2.txt`.
  - `rm`: Removes files.
    - Example: `rm file1.txt`.
  - `mkdir`: Creates directories.
    - Example: `mkdir new_dir`.
  - `rmdir`: Removes empty directories.
    - Example: `rmdir old_dir` .

---

5. Variables

- **Variables** are used to store values in shell scripts. They hold data like numbers, strings, or the result of commands.
  - **Defining a Variable**:
    - Syntax: `VAR_NAME=value` (no spaces around `=`).
    - Example: `name="Kunal"`.
  - **Using a Variable**: Access the value using `$` before the variable name.
    - Example: `echo $name` - prints the value of `name`.
  - **Types of Variables**:
    - **Local Variables**: Declared in a script and accessible only within that script.
    - **Environment Variables**: Predefined variables that affect the system (e.g., `PATH`, `HOME`).
    - **Special Variables**: Positional parameters like `$1`, `$2` for script arguments, `$0` for the script name .

---

6. Quotes

- In shell scripting, different types of quotes are used to handle variable expansion and special characters.
  - **Single quotes (`'`)**: Preserve the literal value of the characters inside them.
    - Example: `'This is $name'` will print `This is $name` without expanding the variable.
  - **Double quotes (`"`)**: Allow variable expansion and special character interpretation.
    - Example: `"This is $name"` will expand to `This is Kunal` if `name="Kunal"`.
  - **Backticks (`` ` ``)**: Used for command substitution (evaluating a command and using its output).

- Example: `result=date`` will store the current date in `result` .

---

7. Command Substitution

- **Command substitution** allows the output of a command to be substituted into another command or variable.
  - Syntax: `$(command)` or `` `command` ``.
  - Example: `today=$(date)` - stores the current date in the `today` variable.
  - Another example: `echo $(ls)` - prints the result of `ls` command .

---

8. Job Control

- **Job Control** refers to managing multiple tasks in a shell session. It involves suspending, resuming, or killing processes.
  - **Foreground Jobs**: Jobs that run actively in the terminal.
  - **Background Jobs**: Jobs that run in the background, allowing you to continue using the terminal.
    - Command `&`: Adds `&` to run a command in the background.
      - Example: `sleep 100 &`.
  - **Controlling Jobs**:
    - `fg`: Brings a background job to the foreground.
    - `bg`: Sends a suspended job to the background.
    - `jobs`: Lists all jobs running in the current shell.
    - `kill`: Terminates a job or process by its job number or PID.
      - Example: `kill %1` - kills job 1 .

Advanced Shell Scripting Topics

1. Control Structures

Control structures allow you to create more complex scripts by adding logic.

a. Conditional Statements

- **If-Else Statements**:

```
bash

#!/bin/bash
read -p "Enter a number: " num
if [ $num -gt 10 ]; then
    echo "$num is greater than 10"
else
    echo "$num is less than or equal to 10"
fi
```

- **Using `case` Statement**:

```
bash

#!/bin/bash
```

```
read -p "Enter a day (1-7): " day
case $day in
    1) echo "Monday" ;;
    2) echo "Tuesday" ;;
    3) echo "Wednesday" ;;
    4) echo "Thursday" ;;
    5) echo "Friday" ;;
    6) echo "Saturday" ;;
    7) echo "Sunday" ;;
    *) echo "Invalid day" ;;
esac
```

b. Loops

- **For Loop**:

```
bash

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

- **While Loop**:

```
bash

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

Practice Exercise:

1. Write a script using a loop that calculates the factorial of a number entered by the user.

---

2. Functions

Functions allow you to group commands into reusable blocks.

Defining and Calling Functions:

```
bash

#!/bin/bash
function greet {
    echo "Hello, $1!"
}

greet "Kunal"
```

Return Values:

```
bash

#!/bin/bash
function add {
    return $(($1 + $2))
}

add 5 10
```

```
echo "The sum is: $?"
```

Practice Exercise:

1. Write a script that defines a function to check if a number is even or odd.

---

3. Arrays

Arrays are useful for storing lists of items.

Creating and Accessing Arrays:

```
bash

#!/bin/bash
fruits=("Apple" "Banana" "Cherry")
echo "First fruit: ${fruits[0]}"
echo "All fruits: ${fruits[@]}"
```

Looping Through Arrays:

```
bash

#!/bin/bash
for fruit in "${fruits[@]}"; do
    echo "Fruit: $fruit"
done
```

Practice Exercise:

1. Write a script that takes a list of names as arguments and stores them in an array, then prints each name.

---

4. File Handling in Scripts

Enhance your file manipulation skills by incorporating file checks and operations.

Checking if a File Exists:

```
bash

#!/bin/bash
if [ -f "testfile.txt" ]; then
    echo "File exists."
else
    echo "File does not exist."
fi
```

Reading from a File:

```
bash

#!/bin/bash
while IFS= read -r line; do
    echo "$line"
done < "testfile.txt"
```

Appending to a File:

```
bash

echo "New Line" >> testfile.txt
```

Practice Exercise:

1. Create a script that reads a file and counts the number of lines, words, and characters.

---

5. Signal Handling

Handling signals in scripts allows you to manage interruptions gracefully.

Using Traps:

```
bash

#!/bin/bash
trap "echo 'You pressed Ctrl+C!'; exit" SIGINT

while true; do
    echo "Running... (Press Ctrl+C to stop)"
    sleep 1
done
```

Practice Exercise:

1. Write a script that listens for `SIGTERM` and performs cleanup actions before exiting.

---

6. Command Line Arguments

Learn to handle user input through command line arguments.

Accessing Arguments:

```
bash

#!/bin/bash
echo "Script name: $0"
echo "First argument: $1"
echo "Second argument: $2"
echo "All arguments: $@"
```

Number of Arguments:

```
bash

echo "Number of arguments: $#"
```

Practice Exercise:

1. Write a script that takes a file name as an argument and prints its contents. If the file does not exist, print an error message.

---

7. Logging and Debugging

Incorporate logging to track script execution and help debug issues.

Logging to a File:

```
bash

exec > >(tee -i logfile.txt) 2>&1  # Redirects stdout and stderr to logfile.txt
echo "Script started."
```

Using `set -x` for Debugging:

```
bash

#!/bin/bash
set -x  # Enable debugging
echo "This will show command execution"
set +x  # Disable debugging
```

Practice Exercise:

1. Write a script that logs the output of a command to a file and includes a timestamp.