**ASSIGNMENT-2**

**Q.1. Introduction of internal and external commands.**

**->**

Internal Commands

Internal commands are built into the command interpreter(like cmd.exe in Windows or the shell in Unix/Linux).

These commands do not require any separate files to run; they are directly loaded into memory when the command interpreter starts.

Examples:

* Windows: cd (change directory), dir (list directory contents), copy, echo.
* Unix/Linux: cd, echo, pwd (print working directory), exit.

External Commands

External commands are not built into the command interpreter. Instead, they are separate executable files stored on disk.

When an external command is invoked, the command interpreter locates the corresponding file on disk and loads it into memory to execute it.

Examples:

* Windows: ping, ipconfig, format, xcopy.
* Unix/Linux: ls (list directory contents), grep (search text), cp (copy files), ps (process status).

Q.2. Feeding output of one command to another command by pipelining.

->

Pipelining in command-line interfaces (such as in Unix/Linux or Windows PowerShell) allows the output of one command to be directly used as the input for another command. This is done using the pipe symbol **|**

A pipe takes the standard output (stdout) of one command and passes it as the standard input (stdin) to the next command.

Syntax: command1 | command2 | command3

Example in Unix/Linux

* Using ‘ls’ and ‘grep’:

*ls -l | grep ".txt*"

ls -l lists the files in the current directory in long format.

grep ".txt" filters the list to show only files with a .txt extension.

* Using ps and grep:

*ps aux | grep "python"*

ps aux displays all running processes.

grep "python" filters the output to show only processes related to Python.

* We can chain multiple commands together using pipes. For example, to find the number of .log files in a directory:

*ls -l | grep ".log" | wc -l*

ls -l lists the files.

grep ".log" filters for .log files.

wc -l counts the number of lines (i.e., the number of .log files).

Q.3. expr, locating command.

->

**expr**

The expr command in Linux is used for evaluating expressions. It can perform basic arithmetic, string operations, and logical comparisons. However, expr is quite limited and typically used in shell scripts for simple calculations and string manipulations

Arithmetic Expressions

Addition: *expr 5+5*

Subtraction: *expr 10-5*

Multiplication: *expr 10 \\* 5*

Division: *expr 10 / 5*

Modulus: expr 10 % 5

String Operations

String Length: *expr length “helloworld”*

Substring Extraction: *expr substr “helloworld” 1 3 (Extract the first 3 characters)*

String Comparision: *expr "hello" = "hello" (Output: 1)*

Logical Comparision

Greater Than: *expr 10 \> 5*

Less Than: *expr 5 \< 10*

Equal: *expr 5 = 5*

**locate**

The locate command is a utility to quickly find files and directories on our system. It searches through a pre-built database, allowing for fast lookups, but does not immediately reflect newly created files unless the database is updated.

* To find all occurrences of a file named example.txt

*locate example.txt*

* To update the locate database (requires root privileges):

*sudo updated*

Q.4. echo command.

->

The echo command in Linux is used to display a line of text or a variable value in the terminal. It's one of the most basic and commonly used commands in shell scripting.

* To print a simple text message:

echo "Hello, World!”

* First, assign a value to a variable and then display it:

name="John Doe "

echo "Hello, $name!"

* To print multiple lines with a single echo command, use the -e option and escape sequences like \n for a new line:

echo -e "Hello,\nWorld!"

* By default, echo adds a newline at the end of the output. To prevent this, use the -n option:

echo -n "Hello, World!"

* New Line (\n):

echo -e "Hello,\nWorld!"

* Tab (\t):

echo -e "Hello,\tWorld!"

* Redirecting Output to a File**:** We can use echo to write text to a file:

echo "This is a line of text." > myfile.txt

* To append text to an existing file:

echo "This is another line of text." >> myfile.txt

* If you want to display text with special characters like \n without interpreting them, you can use single quotes:

echo 'Hello,\nWorld!'

Q.5. Using . and ..

->

The “**.**” represents the current directory you are in. It's often used to reference files or execute scripts in the current directory.

Example:

**Running a Script in the Current Directory:** If we have a script named myscript.sh in the current directory, we

can run it using:

*./myscript.sh*

The ./ tells the shell to look for myscript.sh in the current directory and execute it.

The“**..**” represents the parent directory, which is one level up from the current directory.

Example:

If you want to access a file named file.txt in the parent directory:

*cat ../file.txt*

Q.6. Ways for signing off from linux.

->

Q.7. Ping, Man and help command.

->

The ping command is used to check the network connectivity betweenour computer and another host (such as a website or server). It sends ICMP Echo Request packets to the target host and waits for a response, helping us determine if the host is reachable and how long it takes to send and receive data.

*ping <hostname or IP address>*

Example:

ping google.com

Ping with a Specific Number of Requests:

ping -c 4 google.com

Ping with a Timeout:

ping -w 5 google.com

The man (manual) command is used to display the user manual of any command that we can run on the terminal. It provides detailed information about the command, including its options, usage, and examples.

*man <command>*

Get Help on the ls Command:

man ls

This will display the manual page for the ls command, showing all the options and usage details.

**Sections in Man Pages:** Man pages are divided into sections, such as:

* **User Commands** (general commands)
* **System Calls** (functions provided by the kernel)
* **Library Calls** (functions within program libraries)
* **Special Files** (usually found in /dev)

We can specify a section to search within:

man 2 mkdir

The help command provides help information for built-in shell commands. It’s useful for understanding the syntax and options of commands that are part of the shell itself, rather than standalone programs.

*help <shell-builtin>*

Examples:

Get Help on the cd Command:

help cd

List All Shell Built-ins:

Help

Q.8. Combining the commands.

->

1. **Sequential Execution (;)**

We can run multiple commands one after the other, regardless of whether the previous command succeeded or failed. Each command is executed in order, and the next command starts only after the previous one has finished.

Example:

mkdir new\_folder; cd new\_folder; touch file.txt

2. **Conditional Execution with AND (&&)**

Commands are executed only if the previous command was successful (i.e., it exited with a status code of 0). If a command fails, the subsequent commands are not executed.

Example:

mkdir new\_folder && cd new\_folder && touch file.txt

Here, the cd command will only execute if mkdir new\_folder was successful, and touch file.txt will only run if the directory change was successful.

3. **Conditional Execution with OR (||)**

Commands are executed only if the previous command failed (i.e., it exited with a non-zero status code). This is useful for handling errors.

Example:

mkdir new\_folder || echo "Failed to create directory"

If the mkdir command fails, the message "Failed to create directory" will be printed.

4. **Combining AND and OR (&& and ||)**

We can combine both && and || to create complex conditional logic in a single line.

mkdir new\_folder && echo "Directory created" || echo "Failed to create directory"

This command attempts to create new\_folder. If successful, it prints "Directory created". If it fails, it prints "Failed to create directory".

5. **Pipes (|)**

The output of one command is passed as the input to another command using a pipe. This is very powerful for processing data in stages.

Example:

ls -l | grep ".txt"

This lists all files in long format and then filters the list to show only those with .txt in their names.

6. **Subshells (() and {})**

We can group commands together to be executed in a subshell or as a sequence.

Example with () for a ‘Subshell’:

(cd /tmp && ls)

This changes to the /tmp directory and lists its contents, but the current directory remains unchanged after the subshell finishes.

Example with {} for ‘Grouping’:

{ cd /tmp; ls; } > output.txt

This changes to the /tmp directory, lists its contents, and redirects the output to output.txt. The braces ensure that both commands are executed in the same shell instance.

7. **Background Execution (&)**

Run a command in the background, allowing you to continue using the terminal while the command runs.

Example:

sleep 10 &

echo "This runs immediately"

The sleep 10 command will run in the background, allowing the echo command to run immediately.

8. **Command Substitution ($())**

Use the output of one command as an argument in another command.

Example:

echo "Today is $(date)"

This prints "Today is " followed by the current date.

9. **Redirection (>, >>, <)**

Redirect the output of a command to a file, or use a file as input.

Examples:

1. Redirect Output to a File:

ls > file\_list.txt

Saves the output of ls to file\_list.txt.

1. Append Output to a File:

echo "New Line" >> file\_list.txt

Appends "New Line" to file\_list.txt.

1. Use a File as Input:

sort < unsorted.txt > sorted.txt

Sorts the contents of unsorted.txt and writes the result to sorted.txt.

10. **Chaining with xargs**

Use xargs to build and execute commands from standard input

Example:

find . -name "\*.txt" | xargs grep "search\_term"

This finds all .txt files in the current directory and then searches for "search\_term" within those files

Q.9. File permissions and changing the access rights (chmod)

->

Q.10. vi editor and its basics: write a small paragraph using vi editor





