**Process Management in Linux**

Process management in Linux involves controlling and managing running processes on the system. Processes are instances of running programs, and understanding how to manage them is crucial for system administration. Here’s an overview of the key commands and concepts in process management

**Key Concepts:**

* **Process**: An instance of a program that is being executed.
* **PID (Process ID)**: A unique identifier for each running process.
* **Parent Process**: A process that creates one or more child processes.
* **Child Process**: A process created by another process (the parent).
* **Foreground Process**: A process that runs in the terminal and blocks the terminal until it completes.
* **Background Process**: A process that runs independently of the terminal and allows you to continue using the terminal.

**Common Commands for Process Management:**

**1. Viewing Processes:**

**-->**ps: Displays a snapshot of currently running processes.

🡪ps aux: Displays the Detailed Info about All running processes.

🡪top: Displays a real-time, dynamic view of system processes. Press **q** to quit top.

🡪htop: A more user-friendly, interactive process viewer (if installed). Press **F10 or q** to quit htop.

**2.Managing processes:**

🡪kill PID: Sends a signal to a process, usually to terminate it. Terminates the process with the specified PID.

🡪kill -9 PID: Forcefully terminates the process.

**🡪**killall: Kills all processes with a given name.

🡪fg: Brings a background process to the foreground. -->fg %1 Brings job number 1 to the foreground.

🡪bg: Resumes a suspended process in the background. --< bg %1 Resumes job number 1 in the background.

🡪pkill: Kills processes by name or other attributes. 🡪 pkill -f process\_name

**3. Monitoring System Performance:**

🡪 free: Displays the amount of free and used memory in the system. 🡪 free -h

🡪df: Displays disk space usage of file systems. 🡪 df -h

🡪uptime: Shows how long the system has been running, along with the system load averages.

**4. Controlling Jobs:**

🡪 &: Run a command in the background. 🡪 command &

🡪jobs: Lists all jobs running in the background or stopped.

**Summary:**

* **ps, top, htop**: Used to view and monitor running processes.
* **kill, killall, pkill**: Used to terminate processes.
* **fg, bg, jobs**: Manage processes running in the foreground and background.
* **free, df, uptime**: Monitor system performance and resource usage.

**Process management Commands**

* **1. Viewing Processes**
* 🡪ps: Display information about active processes.
* 🡪top: Real-time system monitoring of processes.
* 🡪htop: Interactive process viewer (enhanced `top`).
* 🡪pidof: Find the process ID (PID) of a running program.
* 🡪pgrep: Search for processes based on name and other attributes.
* **2. Killing/Stopping Processes**
* 🡪kill: Send a signal to terminate a process by PID.
* 🡪killall: Terminate all processes by name.
* 🡪pkill: Terminate processes based on name or attributes.
* 🡪xkill: Kill a process by clicking on its window (for GUI environments).
* **3. Controlling Process Priority**
* 🡪nice: Launch a program with a specified priority.
* 🡪renice: Change the priority of an existing process.
* 🡪ionice: Set or get the I/O scheduling class and priority of a process.
* **4. Background and Foreground Process Management**
* 🡪&: Run a command in the background.
* 🡪jobs: List background jobs.
* 🡪fg: Bring a background job to the foreground.
* 🡪bg: Resume a suspended job in the background.
* -🡪Ctrl+: Suspend a foreground process.
* 🡪disown: Remove a background job from the shell's job table.
* **5. Monitoring and Logging**
* 🡪watch: Periodically execute a command and display the output.
* 🡪strace Trace system calls and signals.
* 🡪lso: List open files associated with processes.
* 🡪vmstat: Report virtual memory statistics.
* 🡪iostat: Report CPU and I/O statistics.
* 🡪mpstat: Report CPU usage per processor.
* 🡪sar: Collect, report, or save system activity information.
* **6. Process Control and Management**
* 🡪nohu: Run a command immune to hangups, with output to a non-tty.
* 🡪at: Schedule a command to be run once at a later time.
* 🡪cron: Schedule commands to be run periodically.
* 🡪systemctl: Control the systemd system and service manager.
* 🡪service: Start, stop, or restart services.
* 🡪pkexec: Execute commands as another user (like `sudo`, but for graphical
* environments).
* **7. Advanced Process Management**
* 🡪pstree: Display a tree of processes.
* 🡪timeout: Run a command with a time limit.
* 🡪taskset: Set or retrieve a process's CPU affinity.
* 🡪cgroups: Manage control groups (limit, account, and isolate the resource usage of processes).
* 🡪schedtool: Query or alter a process's CPU-scheduling.
* **8. Process Namespaces and Container**
* 🡪nsenter: Enter a Linux namespace.
* 🡪unshare: Run a program with some namespaces unshared from the parent.
* 🡪docker: Manage containers, which are lightweight, isolated processes.
* 🡪podman: A daemonless container engine for managing containers.

These commands, offer a complete set of tools for managing and interacting with processes in Linux. They range from basic tasks like viewing and killing processes to advanced tasks like managing CPU affinity and working with namespaces and containers.

**Scheduling Jobs at Crons and Attachment**

Both cron and at are tools for scheduling jobs on Linux, but they serve different purposes and are used in different scenarios:

* **cron** is used for scheduling recurring tasks at specified intervals (e.g., daily, weekly, monthly).
* **at** is used for scheduling a one-time task to be executed at a specific time in the future.

**Introduction to Shell Scripting and Automation in Linux**

**Shell Scripting** is a Kind of File in which consist of set of Commands, which allow to Automate Task, manages File, Programme Execution, User Interaction and manipulation.

Shell Scripting Commands are Executed Sequentially.

**Basic Scripting Syntax in Linux:**

**🡪echo $0:** To check which shell we are using

**🡪#! /bin/bash:** Normal way to tell Linux through Terminal that the Script should be run in bash shell and also This command should be written in every text editor terminal

🡪**./scriptname:** To run the ScriptCommand**.**

**🡪chmod u+x Scriptname** to change the script mode to executable mode as without Executable mode script can not be run

🡪**.sh**: Extension form of Scripting file.

🡪**mkdir scriptname:** creating a Scripting directory.

🡪**touch scriptname.sh:** creating a file with .sh extension

🡪**vi scriptname.sh:** To edit the Content in a file.

🡪**echo content:** for writing the normal content in a File. echo Hello World

🡪**rm \*:** To remove everything**.**

**Variables in Linux Shell Scripting:**

Variables are used to store data that you can manipulate within your script. In Linux, you don't need to declare the data type of a variable explicitly.

🡪**VARIABLE\_NAME="value":** Define a value to be stored.

Example: Name=Nishat

echo $NAME

for O/P: Nishat

🡪**command=$(commandname):** To store the Command in Variable.

Example: Host=$(Hostname)

echo $Host == displays Hostname when we run the script in terminal

name=$(uname)

echo $name == displays Uname when we run the script in terminal

🡪**read<var\_name\_to\_store\_value>:** To take input from the user.

**echo Content to be taken from user**,

**read name:** Reading user I/P, allowing user to enter i/p

**echo $name**: Displaying the User I/P and storing it in variable

Example: echo what is your name?

read name -- allowing user to enter i/p.

echo $name –Displaying the user i/p.

**Types Of Operators:**

**1. Arithmetic Operators**

🡪 **+** Addition 🡪 result=$((a + b))

🡪**-** Subtraction 🡪 result=$((a - b))

🡪 **\*** Multiplication 🡪 result=$((a \* b))

🡪 **/** Division🡪 result=$((a / b))

🡪**%** Modulus 🡪 result=$((a % b))

🡪**\*\*** Exponentiation 🡪 result=$((a \*\* b))

**2. Relational (Comparison) Operators**

🡪**-eq** Equal to 🡪 if [ "$a" -eq "$b" ]; then

echo "a is equal to b

fi

🡪**-ne** Not equal to 🡪 if [ "$a" -ne "$b" ]; then

echo "a is not equal to b"

fi

🡪**-gt** Greater Than 🡪 if [ "$a" -gt "$b" ]; then

echo "a is greater than b"

fi

🡪**-lt** Lesser then 🡪 if [ "$a" -lt "$b" ]; then

echo "a is less than b"

fi

🡪**-ge** Greater then or Equals to 🡪 if [ "$a" -ge "$b" ]; then

echo "a is greater than or equal to b"

fi

🡪**-le** Lesser then or equal to 🡪 if [ "$a" -le "$b" ]; then

echo "a is less than or equal to b"

fi

**3. Boolean Operators**

Boolean operators are used to perform logical operations.

🡪**&&** Logical AND 🡪 if [ "$a" -gt 0 ] && [ "$b" -gt 0 ]; then

echo "Both a and b are greater than 0"

fi

🡪**||** Logical OR 🡪 if [ "$a" -gt 0 ] || [ "$b" -gt 0 ]; then

echo "Either a or b is greater than 0"

fi

🡪**!** Logical NOT 🡪 if [ ! "$a" -eq "$b" ]; then

echo "a is not equal to b"

fi

**4. String Operators**

String operators are used to compare strings.

🡪**=** Equals To 🡪 if [ "$str1" = "$str2" ]; then

echo "Strings are equal"

fi

🡪**!=**  Not Equals to 🡪 if [ "$str1" != "$str2" ]; then

echo "Strings are not equal"

fi

🡪**z** String is Null (Zero Length) 🡪 if [ -z "$str1" ]; then

echo "String is empty"

fi

🡪**-n**  String is not Null(non zero length)🡪 if [ -n "$str1" ]; then

echo "String is not empty"

fi

**5. File Test Operators**

File test operators are used to test properties of files.

🡪**-e**  Checks if File Exits 🡪 if [ -e "/path/to/file" ]; then

echo "File exists"

fi

🡪**-f** Checks if a file is a regular file 🡪 if [ -f "/path/to/file" ]; then

echo "File is a regular file"

fi

🡪**-d**  Checks if file is directory 🡪 if [ -d "/path/to/directory" ]; then

echo "Directory exists"

fi

🡪**-r** Checks if file is readable 🡪 if [ -r "/path/to/file" ]; then

echo "File is readable"

fi

🡪**-w** Checks if File is writable 🡪 if [ -w "/path/to/file" ]; then

echo "File is writable"

fi

🡪**-x** Checks if file is executable 🡪 if [ -x "/path/to/file" ]; then

echo "File is executable"

fi

**Conditionals in Linux Shell Scripting**

Conditionals are used to make decisions based on different conditions.

**If-Else Statement:**

echo Main Content

echo Taking I/p from User

read i/p –Allowing user to i/p

if [ condition ]

then

echo print statement if condition satisfied

else

echo print statement if condition satisfied

fi

**Example: #!/bin/bash**

# Printing the main content

echo "Main Content"

# Prompting the user for input

echo "Taking Input from User"

read input # Allows user to input a value

# Example condition: Check if the input is equal to "hello"

if [ "$input" == "hello" ]

then

# If the condition is satisfied (input is "hello")

echo "You typed 'hello'. Welcome!"

else

# If the condition is not satisfied

echo "You did not type 'hello'. Try again!"

fi

**Case Scripting:**

In bash scripting, the case statement is used to execute different blocks of code based on the value of a variable or an expression.

**Syntax:**

echo Main Content or choosing a valid option

echo Option 1

echo Option 2

‘

‘

‘

Echo Option n

read Choice from Main Statement

case $Choice in

1)Option1;; shd be a command from Linux

2)Option2;;

` 3) Option3;;

‘

‘

‘

\*) Invalid Option;;

Esac

**Example:1**

#!/bin/bash

echo "Please choose an option:"

echo "1. Display date and time"

echo "2. List files in the current directory"

echo "3. Display the current working directory"

echo "4. Exit"

# Reading user input

read Choice

# Using a case statement to handle different choices

case $choice in

1) echo "Current date and time: $(date)";;

2) echo "Files in the current directory:" ls;;

3) echo "Current working directory: $(pwd)";;

4)echo "Exiting the script." exit 0;;

\*)echo "Invalid option. Please enter a number between 1 and 4.;;

Esac

**Example:2**

#!/bin/bash

# Prompting the user to enter a day of the week

echo "Enter a day of the week (e.g., Monday, Tuesday):"

read day

# Using case statement to determine if it's a weekday or weekend

case $day in " | "Tuesday" | "Wednesday" | "Thursday" | "Friday")

echo "$day is a weekday.";;

"Saturday" | "Sunday")

echo "$day is a weekend.";;

\*)echo "Invalid input. Please enter a valid day of the week.";;

esac

**FOR LOOP:**

In bash scripting, the for loop is used to iterate over a series of items, executing a block of code for each item in the list. It is particularly useful for performing repetitive tasks.

**Syntax:**

for variable in list

do

echo # Commands to execute for each item in the list

done

**Example: 1 Iterating Over a Range of Numbers**

for number in {1..50}

do

echo number in $num

done

**Example:2 Iterating Over a List of Words**

for Word in Dog Cat

do

echo I like $Word

done

**Example:3 for Iterating a file over a Directory**

# A loop that iterates over all files in the current directory

for file in \*

do

echo "File: $file"

done

**Example:4** **Using a for Loop with a Command**

# Looping through the output of a command

for user in $(cat /etc/passwd | cut -d: -f1)

do

echo "User: $user"

done

**Example 5: for Loop with C-Style Syntax**

# A C-style for loop

for ((i = 1; i <= 5; i++))

do

echo "Counter: $i"

done

**WHILE LOOP:**

The while loop in bash scripting is used to repeatedly execute a block of commands as long as a specified condition remains true. It is particularly useful when the number of iterations is not known beforehand.

**Syntax:**

while [ condition ]

do

# Commands to be executed as long as the condition is true

Done

**Example:**

count=0

num=10

while [ $count -le $num ]

do

echo $count

let count++

done

🡪let count++ is used to tell the terminal what has to be done with expression

**Example 2: Reading User Input Until a Condition is Met**

# Initialize the variable

input=""

# Loop until the user enters "quit"

while [ "$input" != "quit" ]

do

echo "Enter something (type 'quit' to exit):"

read input

echo "You entered: $input"

done

echo "Goodbye!"

**Example 3: Infinite Loop**

An infinite loop is a loop that never terminates on its own. This can be useful for certain background tasks or processes that need to run continuously.

#!/bin/bash

# Infinite loop

while true

do

echo "This loop will run forever. Press [CTRL+C] to stop."

sleep 2 # Waits for 2 seconds before repeating

done

**Example 4: Using a while Loop to Read a File Line by Line**

# Using a while loop to read a file line by line

filename="example.txt"

while IFS= read -r line

do

echo "Line: $line"

done < "$filename"

**Creating and Executing Simple Scripts:**

Creating and executing simple scripts in a Linux environment is a great way to automate tasks and learn bash scripting.

**Steps Are:**

🡪Creating a Scripts File

🡪Granting Permission

🡪And Executing a Script

**1. Creating a Simple Bash Script**

**Step 1: Open a Text Editor**

* You can use any text editor, such as nano, vi, or vim, to create a script.

🡪vi script\_name.sh

**Step 2: Write the Script**

* Start the script with a **shebang** (#!), which tells the system which interpreter to use to run the script.
* Add some simple commands to the script.

For example:

#!/bin/bash

# This is a comment

echo "Hello, World!"

echo "Today is $(date)"

echo "Your current working directory is $(pwd)"

**Step 3: Save and Exit**

* If you're using nano, press Ctrl + O to save the file, then enter, and Ctrl + X to exit.
* If you're using vi and vim press esc and :wq To save and exit .

**2. Making the Script Executable**

Before you can execute the script, you need to make it executable.

* Run the following command to change the file permissions to make it executable:

🡪chmod u+x script.sh

**3. Executing the Script**

Once the script is executable, you can run it in several ways:

**Method 1: Execute the Script Directly**

**🡪** **./my\_script.sh**

**Method 2: Execute the Script Using bash**

* Alternatively, you can run the script by passing it as an argument to bash

🡪 **bash my\_script.sh**

**Method 3: Execute the Script Using sh**

* You can also run the script with sh

🡪 **sh my\_script.sh**

**Automating Task with Scripts:**

Automating tasks with scripts is a powerful way to streamline repetitive processes and ensure consistency in your work. In Linux, shell scripting is commonly used for automation.

**1. Creating a Simple Automation Script**

Let’s create a simple script to automate the task of backing up a directory.

**1. Create a Directory and File**

**1.1 Open Terminal**

First, open your terminal.

**1.2 Create a Directory**

Use the mkdir command to create a new directory. For example, to create a directory named projects in your home directory:

🡪**mkdir ~/projects**

~: Represents your home directory.

projects: The name of the new directory.

**1.3 Create a File**

Navigate to the directory and create a file. For example, to create a file named backup.sh in the projects directory

🡪**cd ~/projects**

**touch backup.sh**

cd ~/projects: Change directory to projects.

touch backup.sh: Create an empty file named backup.sh.

**2. Verify the Directory and File**

**2.1 List Directory Contents**

List the contents of the projects directory to verify the file creation:

🡪**ls -l ~/projects** You should see backup.sh listed in the output.

**2.2 Check File Path**

You can check the full path of the backup.sh file using the realpath command:

🡪**realpath ~/projects/backup.sh** This will print the full path of the file.

**3. Example Script Creation**

Now, let’s create a simple script in the backup.sh file. You can use nano, vim, or any text editor.

**3.1 Edit the File with Nano**

Open the backup.sh file with nano:

🡪**nano ~/projects/backup.sh**

**3.2 Add Example Script**

Paste the following example script into backup.sh:

🡪#!/bin/bash

# Define variables

SOURCE\_DIR="$HOME/projects" # Directory to back up

BACKUP\_DIR="$HOME/backup" # Backup storage directory

LOG\_FILE="$HOME/backup/backup.log" # Log file to record backup status

DATE=$(date +"%Y-%m-%d\_%H-%M-%S") # Current date and time

# Create backup directory if it doesn't exist

mkdir -p $BACKUP\_DIR

# Perform the backup

tar -czf $BACKUP\_DIR/backup\_$DATE.tar.gz -C $SOURCE\_DIR .

# Log the result

echo "Backup completed on $DATE" >> $LOG\_FILE

**$HOME**: Represents your home directory.

**tar -czf**: Creates a compressed archive file.

**3.3 Save and Exit**

To save and exit in nano, press Ctrl+X, then Y, and then Enter.

**4. Make the Script Executable**

Set executable permissions for the script:

🡪**chmod +x ~/projects/backup.sh**

**5. Run and Verify**

Run the script to check if it works:

**🡪~/projects/backup.sh**

Verify that the backup file and log file are created in the backup directory:

🡪**ls -l ~/backup**

**Summary**

1. **Create a Directory**: mkdir ~/projects
2. **Create a File**: touch ~/projects/backup.sh
3. **Edit the File**: nano ~/projects/backup.sh
4. **Add Script**: Paste the example script into the file.
5. **Make Executable**: chmm od +x ~/projects/backup.sh
6. **Run the Script**: ~/projects/backup.sh
7. **Verify Output**: Check the backup directory for the backup and log files.

**Environmental Variable and Shell Features:**

Environment variables and shell features are key components in managing your Linux environment. Here’s a quick overview:

**Environment Variables**

**Definition**: Environment variables are dynamic values that affect the behaviour of processes and programs on a system. They are used to configure settings and pass information to programs.

Environmental Variables can be created by any Processes and Shell Process and It can be accessed by other subsequent Processes as well.

Environmental Variable Used to send some data to child process

**Common Environment Variables**:

HOME: User’s home directory.

PATH: List of directories to search for executable files.

USER: Current logged-in user.

SHELL: Path to the current shell (e.g., `/bin/bash`).

PWD: Current working directory.

LANG: Language and locale settings.

**Viewing Environment Variables:**

🡪 **printenv** or **env** to list all environment variables.

🡪**echo $VARIABLE\_NAME** to view a specific variable.

**Creating or Modifying Environment Variables:**

**🡪** export VAR\_NAME="value" Creates or modifies an environment variable and makes it available to sub-processes (child shells). Temporarily

🡪export PATH=$PATH:/new/path Add a directory to the existing PATH variable

🡪Permanently: Add the `export VARIABLE\_NAME=value` line to files like `~/.bashrc`, `~/.bash\_profile`, or `/etc/environment`.

**Assignment Without export:**

This sets the variable for the current shell but does not make it available to sub-processes.

🡪 VAR\_NAME="value"

**Viewing the Value of a Specific Variable:**

echo to display the value of an environment variable

🡪echo $VARNAME

🡪echo $PATH Display the PATH variable

🡪 export PATH=$PATH:/new/path Lists directories where the shell looks for executables. You can modify it by appending new directories.

🡪echo $HOME Points to the current user’s home directory.

🡪echo $SHELL Displays the current user’s default shell

🡪echo $USER The current logged-in user.

🡪 export PS1="[\u@\h \W]$ " PS1: Modifies the shell prompt.

🡪ps -e | grep bash To check the bash shell running and bash ID

**Unsetting/Removing Environment Variables:**

Unset Removes a specific environment variable from the shell's environment

🡪unset VAR\_NAME Unsetting a Variable

🡪 unset $PATH unsetting a path

🡪unset var1 var2 for Unsetting a multiple variable

**Running Commands with Temporary Environment Variables:**

🡪 env VAR\_NAME="value" command Temporarily sets an environment variable for a specific command execution.

🡪 env -u VAR\_NAME command Unsets (removes) a variable for the duration of a single command.

🡪 env -i command Clears all environment variables and runs a command in a clean environment.

**Checking/Viewing Shell-Specific Variables:**

🡪set Lists all shell variables (both environment and local variables).

🡪export -p Lists all exported environment variables.

**Common Shell Files for Environment Variables.**

**🡪**/etc/profile: System-wide configuration file for environment settings.

**🡪**~/.bashrc: User-specific configuration file for bash shells.

**🡪**~/.bash\_profile: Loaded when a login shell is started (for bash).

**Terminator:**

The **command terminator** in Unix/Linux refers to symbols that indicate the end of a command or how multiple commands are executed together in a shell. These terminators are used to separate commands, control their execution flow, and manage how they interact with each other.

**Common Command Terminators**

**1.Semicolon (;)**:

* The semicolon allows you to run multiple commands sequentially, one after the other. Each command is executed independently, meaning if one command fails, the next command will still run.

🡪 command1; command2; command3 🡪 echo "Hello"; ls; pwd

**2.Double Ampersand (&&)**:

* The && operator runs the second command only if the first command is successful (i.e., it returns an exit status of 0). This is useful for chaining commands that depend on each other.

🡪 command1 && command2 🡪 mkdir new\_dir && cd new\_dir

3. **Double Pipe (||)**:

* The || operator runs the second command only if the first command fails (i.e., it returns a non-zero exit status). This is often used for fallback mechanisms or error handling.

🡪 command1 || command2 🡪 mkdir existing\_dir || echo "Directory already exists."

4. **Ampersand (&)**:

* The single & runs a command in the background, allowing you to continue using the terminal while the command executes. The prompt returns immediately, and the command runs asynchronously.

🡪 command & 🡪 sleep 60 &

**5.Pipe (|)**:

* The pipe operator (|) passes the output of one command as the input to another command. This allows you to chain commands together in a way that each command processes data produced by the previous command.

🡪 command1 | command2 🡪 ls -l | grep "txt"

Example Using Multiple Terminators

**🡪mkdir new\_folder && cd new\_folder || echo "Failed to create directory" ; touch file.txt**

**SHELL VARIABLE:**

Shell variables are used to store data that can be referenced and manipulated by the shell or shell scripts. They can hold strings, numbers, or command outputs.

Shell Variables are Created using Shell and it can not be accessed by other processes in the environment.

Here the Commands are used without export key Bcoz variable created by Shell is not accessible to other process

🡪 variable\_name=value 🡪 greeting="Hello, World!"

**1.2. Viewing Variables**

**Description**:  
To see the value of a variable or list all variables, use the following commands:

**Commands**:

🡪echo $variable\_name: Displays the value of the specified variable.

🡪printenv: Lists all environment variables.

🡪set: Lists all shell variables and functions.

Example:

echo $greeting

printenv

set

**1.3. Setting and Exporting Variables**

**Description**:  
Variables can be set and made available to subprocesses by exporting them.

**Commands**:

🡪variable\_name=value: Sets a variable without export so that its not accessed by Child processes.

🡪export variable\_name=value: Sets and exports a variable to be available to child processes.

**1.4. Modifying Variables**

**Description**:  
You can modify the value of a variable by reassigning it.

**Commands**:

🡪variable\_name=new\_value: Reassigns a new value to the variable.

Example: username="JaneDoe"

username="JohnSmith"

**1.5. Unsetting Variables**

**Description**:  
To remove a variable, use the unset command.

**Commands**:

* unset variable\_name: Deletes the variable

🡪 unset username

**Special Shell Variables**

**2.1. Positional Parameters**

**Description**:  
Variables that represent the command-line arguments passed to a script or function.

**Variables**:

* $0: Name of the script.
* $1, $2, …, $N: Command-line arguments.
* $#: Number of arguments.
* $@: All arguments as a list.
* $\*: All arguments as a single string.

Example:

echo "Script name: $0"

echo "First argument: $1"

echo "Number of arguments: $#"

**2.2. Environment Variables**

**Description**:  
Variables that are inherited by processes started by the shell. They provide system-wide information.

**Common Variables**:

* HOME: User's home directory.
* PATH: Directories searched for executables.
* USER: Current logged-in username.
* SHELL: Path to the current shell.

Example:

echo "Home directory: $HOME"

echo "Path: $PATH"

**Viewing Special Variables**

**5.1. Description**  
Special shell variables provide information about the shell's state or processes. Use echo to view them.

**5.2. Commands**

* $$: Process ID of the current shell.
* $?: Exit status of the last command.
* $!: Process ID of the last background command.
* $-: Current shell options.

Example: echo "Current shell PID: $$"

echo "Last command exit status: $?"

echo "Last background command PID: $!"

echo "Shell options: $-"

**Shell Features**

1. **Command History**:

- Use the `history` command to view previously entered commands.

- Use `!n` to execute the nth command from history.

2. **Command Line Editing**:

- Use `Ctrl + A` to move to the beginning of the line.

- Use `Ctrl + E` to move to the end of the line.

- Use `Ctrl + U` to delete from the cursor to the beginning of the line.

- Use `Ctrl + K` to delete from the cursor to the end of the line.

3. **Tab Completion**:

- Press `Tab` to auto-complete file names or commands.

4. **Pipes and Redirection**:

- Use `|` to pipe the output of one command as input to another.

- Use `>` to redirect output to a file.

- Use `<` to redirect input from a file.

5. **Background and Foreground Jobs**:

- Use `&` to run a command in the background.

- Use `fg` to bring a background job to the foreground.

- Use `bg` to resume a suspended job in the background.

- Use `jobs` to list background jobs.

**ALIAS:**

Aliases in Unix/Linux systems are used to create shortcuts for commands or groups of commands. They can simplify complex commands, reduce typing, and customize command behaviors.

We can make custom command using variable as well But alias look simple as in Variable need to use $ sign.

**1.Creating Aliases:**  
To create a simple alias, use the alias command followed by the alias name and the command it represents.

🡪alias commandname=command

**2. Viewing Aliases**

**List All Aliases**  
To see all the aliases currently set in your shell session, use the alias command without arguments.

🡪alias

**3. Removing Aliases**

**Unset an Alias:**  
To remove an alias, use the unalias command followed by the alias name.

🡪unalias alias name 🡪unalias ll

**Once we close the Terminal all the variable we created will be vanished as its for temporary To Make it Permanent we need to use Persistent Aliases. i.e: nano /.bashrc and nano /etc/environment**

**4. Persistent Aliases**

**\*\*./bashrc**

**Adding Aliases to Shell Configuration Files:**To make aliases persistent across shell sessions, add them to your shell’s configuration file (e.g., .bashrc for Bash or .zshrc for Zsh).

Steps:

1. 🡪 nano ~/.bashrc Open the configuration file in a text editor.
2. 🡪 alias ll='ls -l' Add your aliases to the file.
3. 🡪 source ~/.bashrc or source~/.bashrc Save the file and reload it.

**5. Common Alias Examples:**

**Listing Files:**

🡪alias ll='ls -l'

🡪alias la='ls -a'

🡪alias lla='ls -la'

**Navigation:**

🡪alias ..='cd ..'

🡪alias ...='cd ../..'

**System Update:**

**🡪** alias update='sudo apt update && sudo apt upgrade'

**\*\* /etc/environment:** file in Linux is used to set system-wide environment variables. Unlike shell-specific configuration files (like .bashrc or .bash\_profile), which affect a single user's shell session, the /etc/environment file is applied across all users and all processes.

**System-Wide Configuration**: It provides a centralized way to set environment variables that will be available to all users and processes on the system.

**Non-Shell-Specific**: The variables set in this file are loaded for all programs, not just for shell sessions, making it more global in scope.

**How It Works**

* When the system boots up, or when a user logs in, environment variables defined in /etc/environment are loaded.
* These variables are not specific to any shell (e.g., Bash, Zsh) and are available to all applications and users.

**Example of /etc/environment**

Here’s what the contents of an /etc/environment file might look like:

PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"

JAVA\_HOME="/usr/lib/jvm/java-11-openjdk-amd64"

**Common Use Cases**

1. **Setting System-Wide Paths**: If you want all users to have access to a particular binary or application, you can add it to the PATH variable in /etc/environment.
2. **Setting Global Variables**: Variables like JAVA\_HOME, LANG, or LC\_ALL can be defined here so that they are available system-wide.
3. **Persistent Environment Variables**: Environment variables set here persist across sessions, making them available even after system reboots or logouts.

**How to Modify /etc/environment**

1. **Open the file**:

🡪sudo nano /etc/environment

1. **Add or modify the environment variables**:

🡪MY\_VAR="my\_value"

1. **Save and exit the file**.
2. **Apply the changes** (log out and log back in, or source the file):

🡪 source /etc/environment

**Differences from Other Files**

* **/etc/profile and .bashrc**: These files are shell-specific and are typically used to set environment variables for specific users.
* **/etc/environment**: It's not shell-specific and affects the whole system and all users.

In summary, /etc/environment is useful when you need to set environment variables that should be available globally across all users and processes, independent of any specific shell or user configuration.