Unix commands are a set of commands that are used to interact with the Unix operating system. Unix is a powerful, multi-user, multi-tasking operating system that was developed in the 1960s by Bell Labs. Unix commands are entered at the command prompt in a terminal window, and they allow users to perform a wide variety of tasks, such as managing files and directories, running processes, managing user accounts, and configuring network settings. Unix is now one of the most commonly used Operating systems used for various purposes such as Personal use, Servers, Smartphones, and many more. It was developed in the 1970’s at AT& T Labs by two famous personalities Dennis M. Ritchie and Ken Thompson.

You’ll be surprised to know that the most popular programming language C came into existence to write the Unix Operating System.

Linux is Unix-Like operating system.

The most important part of the Linux is Linux Kernel which was first released in the early 90s by Linus Torvalds. There are several Linux distros available (most are open-source and free to download and use) such as Ubuntu, Debian, Fedora, Kali, Mint, Gentoo, Arch and much more.

Unix and the C were found by AT&T and distributed to government and academic institutions, which led to both being ported to a wider variety of machine families than any other operating system. The main focus that was brought by the developers in this operating system was the Kernel . Unix was considered to be the heart of the operating System. The system Structure of Unix OS are as follows:

UNIX is a family of multitasking, multiuser computer operating systems developed in the mid 1960s at Bell Labs. It was originally developed for mini computers and has since been ported to various hardware platforms. UNIX has a reputation for stability, security, and scalability, making it a popular choice for enterprise-level computing.

For those preparing for exams like GATE , a thorough understanding of operating systems, including Unix, is essential. Our GATE course provides an in-depth exploration of Unix, covering its history, structure, and key concepts that are crucial for the exam

The basic design philosophy of UNIX is to provide simple, powerful tools that can be combined to perform complex tasks. It features a command-line interface that allows users to interact with the system through a series of commands, rather than through a graphical user interface (GUI).

Some of the key features of UNIX include:

Multiuser support: UNIX allows multiple users to simultaneously access the same system and share resources.

Multitasking: UNIX is capable of running multiple processes at the same time.

Shell scripting: UNIX provides a powerful scripting language that allows users to automate tasks.

Security: UNIX has a robust security model that includes file permissions, user accounts, and network security features.

Portability: UNIX can run on a wide variety of hardware platforms, from small embedded systems to large mainframe computers.

Communication: UNIX supports communication methods using the write command, mail command, etc.

Process Tracking: UNIX maintains a record of the jobs that the user creates. This function improves system performance by monitoring CPU usage. It also allows you to keep track of how much disk space each user uses, and the use that information to regulate disk space.

Today, UNIX is widely used in enterprise-level computing, scientific research, and web servers. Many modern operating systems, including Linux and macOS, are based on UNIX or its variants.

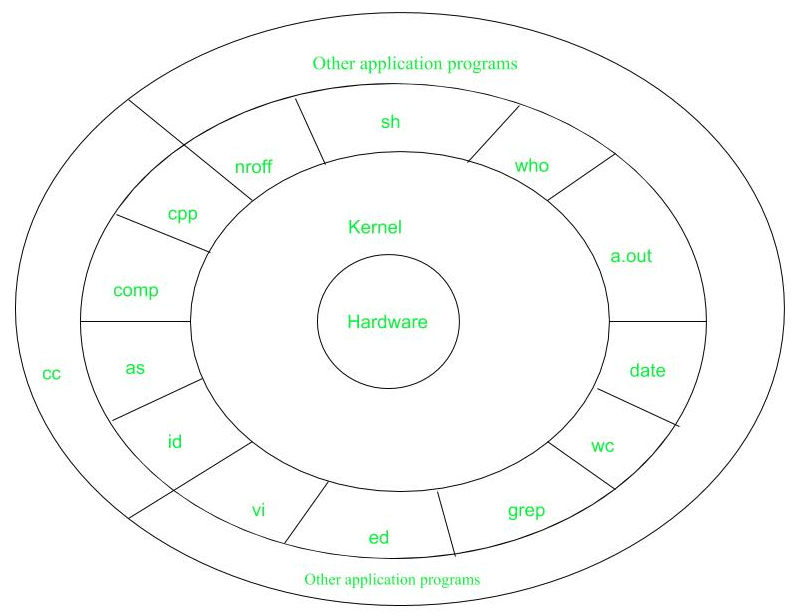


Figure – system structure

Layer-1: Hardware: It consists of all hardware related information.

Layer-2: Kernel: This is the core of the Operating System. It is a software that acts as the interface between the hardware and the software. Most of the tasks like memory management, file management, network management, process management, etc., are done by the kernel.

Layer-3: Shell commands: This is the interface between the user and the kernel. Shell is the utility that processes your requests. When you type in a command at the terminal, the shell interprets the command and calls the program that you want. There are various commands like cp, mv, cat, grep, id, wc, nroff, a.out and more.

Layer-4: Application Layer: It is the outermost layer that executes the given external applications.

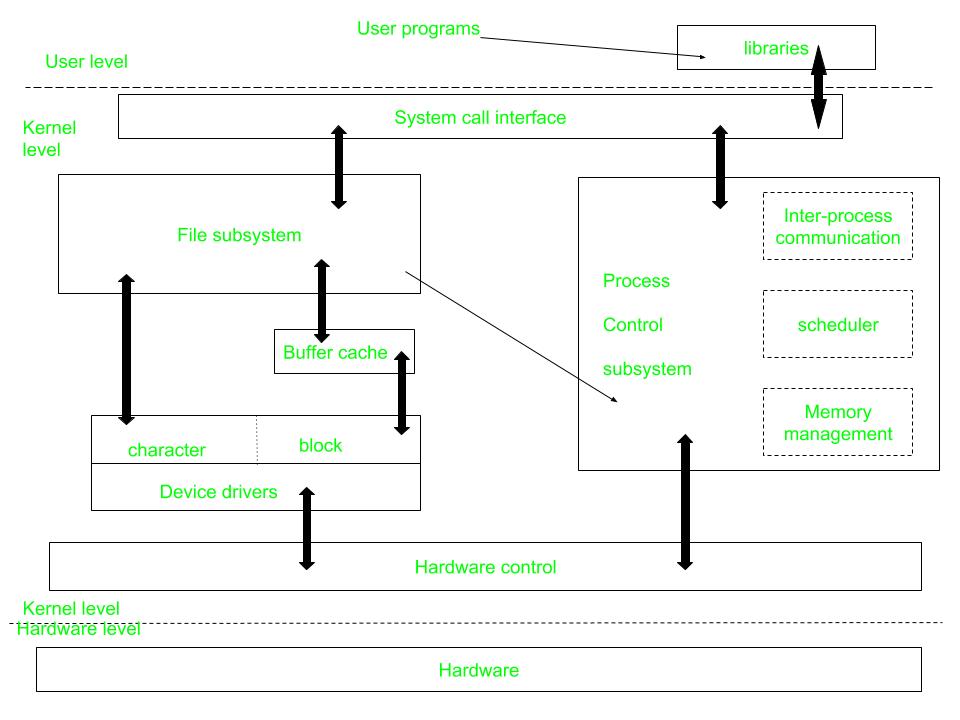


Figure – kernel and its block diagram

This diagram shows three levels: user, kernel, and hardware.

The system call and library interface represent the border between user programs and the kernel. System calls look like ordinary function calls in C programs. Assembly language programs may invoke system calls directly without a system call library. The libraries are linked with the programs at compile time.

The set of system calls into those that interact with the file subsystem and some system calls interact with the process control subsystem. The file subsystem manages files, allocating file space, administering free space, controlling access to files, and retrieving data for users.

Processes interact with the file subsystem via a specific set of system calls, such as open (to open a file for reading or writing), close, read, write, stat (query the attributes of a file), chown (change the record of who owns the file), and chmod (change the access permissions of a file).

The file subsystem accesses file data using a buffering mechanism that regulates data flow between the kernel and secondary storage devices. The buffering mechanism interacts with block I/O device drivers to initiate data transfer to and from the kernel.

Device drivers are the kernel modules that control the operator of peripheral devices. The file subsystem also interacts directly with “raw” I/O device drivers without the intervention of the buffering mechanism. Finally, the hardware control is responsible for handling interrupts and for communicating with the machine. Devices such as disks or terminals may interrupt the CPU while a process is executing. If so, the kernel may resume execution of the interrupted process after servicing the interrupt.

Interrupts are not serviced by special processes but by special functions in the kernel, called in the context of the currently running process.

**Difference between Unix and Linux**

Linux is essentially a clone of Unix. But, basic differences are shown below:

| **Linux** | **Unix** |
| --- | --- |
| The source code of Linux is freely available to its users | The source code of Unix is not freely available general public |
| It has graphical user interface along with command line interface | It only has command line interface |
| Linux OS is portable, flexible, and can be executed in different hard drives | Unix OS is not portable |
| Different versions of Linux OS are Ubuntu, Linux Mint, RedHat Enterprise Linux, Solaris, etc. | Different version of Unix are AIS, HP-UX, BSD, Iris, etc. |
| The file systems supported by Linux are as follows: xfs, ramfs, vfat, cramfsm, ext3, ext4, ext2, ext1, ufs, autofs, devpts, ntfs | The file systems supported by Unix are as follows: zfs, js, hfx, gps, xfs, vxfs |
| Linux is an open-source operating system that was first released in 1991 by Linus Torvalds. | Unix is a proprietary operating system that was originally developed by AT&T Bell Labs in the mid 1960s. |
| The Linux kernel is monolithic, meaning that all of its services are provided by a single kernel. | The Unix kernel is modular, meaning that it is made up of a collection of independent modules that can be loaded and unloaded dynamically. |
| Linux has much broader hardware support than Unix. | Unix was originally designed to run on large, expensive mainframe computers, while Linux was designed to run on commodity hardware like PCs and servers. |
| User Interface of Linux is Graphical or text-based. | User Interface of unix is text-based. |
| Command Line Interface of Linux is Bash, Zsh, Tcsh. | Command Line Interface of unix is Bourne, Korn, C, Zsh. |

Advantages of UNIX:

Stability: UNIX is known for its stability and reliability. It can run for long periods of time without requiring a reboot, which makes it ideal for critical systems that need to run continuously.

Security: UNIX has a robust security model that includes file permissions, user accounts, and network security features. This makes it a popular choice for systems that require high levels of security.

Scalability: UNIX can be scaled up to handle large workloads and can be used on a variety of hardware platforms.

Flexibility: UNIX is highly customizable and can be configured to suit a wide range of needs. It can be used for everything from simple desktop systems to complex server environments.

Command-line interface: UNIX’s command-line interface allows for powerful and efficient interaction with the system.

Disadvantages of UNIX:

Complexity: UNIX can be complex and difficult to learn for users who are used to graphical user interfaces (GUIs).

Cost: Some UNIX systems can be expensive, especially when compared to open-source alternatives like Linux.

Lack of standardization: There are many different versions of UNIX, which can make it difficult to ensure compatibility between different systems.

Limited software availability: Some specialized software may not be available for UNIX systems.

Steep learning curve: UNIX requires a certain level of technical knowledge and expertise, which can make it challenging for novice users.

--------------------------------------------------------------------------------

Now coming to the Basic and most usable commands of Linux/Unix part. (Please note that all the linux/unix commands are run in the terminal of a linux system.Terminal is like command prompt as that of in Windows OS)

Linux/Unix commands are case-sensitive i.e Hello is different from hello.

Basic Unix commands:

Table of Content:-

File System Navigation Unix Command

File Manipulation Unix Command

Process Management Unix Command

Text Processing Unix Command

Network Communication Unix Command

Text Editors in Unix

**File System Navigation Unix Command**

| **Command** | **Description** | **Example** |
| --- | --- | --- |
| **cd** | Changes the current working directory. | cd Documents |
| **ls** | Lists files and directories in the current directory. | ls |
| **pwd** | Prints the current working directory. | pwd |
| **mkdir** | Creates a new directory. | mkdir new\_folder |
| **rmdir** | Removes an empty directory. | rmdir empty\_folder |
| **mv** | Moves files or directories. | mv file1.txt Documents/ |

**File Manipulation Unix Command**

| **Command** | **Description** | **Example** |
| --- | --- | --- |
| **touch** | Creates an empty file or updates the access and modification times. | touch new\_file.txt |
| **cp** | Copies files or directories. | cp file1.txt file2.txt |
| **mv** | Moves files or directories. | mv file1.txt Documents |
| **rm** | Remove files or directories. | rm old\_file.txt |
| **chmod** | Changes the permissions of a file or directory. | chmod 644 file.txt |
| **chown** | Changes the owner and group of a file or directory. | chown user:group file.txt |
| **ln** | Creates links between files. | ln -s target\_file symlink |
| **cat** | Concatenates files and displays their contents. | cat file1.txt file2.txt |
| **head** | Displays the first few lines of a file. | head file.txt |
| **tail** | Displays the last few lines of a file. | tail file.txt |
| **more** | Displays the contents of a file page by page. | more file.txt |
| **less** | Displays the contents of a file with advanced navigation features. | less file.txt |
| **diff** | Compares files line by line. | diff file1.txt file2.txt |
| **patch** | Applies a diff file to update a target file. | patch file.txt < changes.diff |

**Process Management Unix Command**

| **Command** | **Description** | **Example** |
| --- | --- | --- |
| **ps** | Displays information about active processes, including their status and IDs. | ps aux |
| **top** | Displays a dynamic real-time view of system processes and their resource usage. | top |
| **kill** | Terminates processes using their process IDs (PIDs). | kill <pid> |
| **pkill** | Sends signals to processes based on name or other attributes. | pkill -9 firefox |
| **killall** | Terminates processes by name. | killall -9 firefox |
| **renice** | Changes the priority of running processes. | renice -n 10 <pid> |
| **nice** | Runs a command with modified scheduling priority. | nice -n 10 command |
| **pstree** | Displays running processes as a tree. | pstree |
| **pgrep** | Searches for processes by name or other attributes. | pgrep firefox |
| **jobs** | Lists active jobs and their status in the current shell session. | jobs |
| **bg** | Puts a job in the background. | bg <job\_id> |
| **fg** | Brings a background job to the foreground. | fg <job\_id> |
| **nohup** | Runs a command immune to hangups, with output to a specified file. | nohup command & |
| **disown** | Removes jobs from the shell’s job table, allowing them to run independently. | disown <job\_id> |

**Text Processing Unix Command**

| **Command** | **Description** | **Example** |
| --- | --- | --- |
| **grep** | Searches for patterns in text files. | grep "error" logfile.txt |
| **sed** | Processes and transforms text streams. | sed 's/old\_string/new\_string/g' file.txt |
| **awk** | Processes and analyzes text files using a pattern scanning and processing language. | awk '{print $1, $3}' data.csv |

**Network Communication Unix Command**

| **Command** | **Description** | **Example** |
| --- | --- | --- |
| **ping** | Tests connectivity with another host using ICMP echo requests. | ping google.com |
| **traceroute** | Traces the route that packets take to reach a destination. | traceroute google.com |
| **nslookup** | Queries DNS servers for domain name resolution and IP address information. | nslookup google.com |
| **dig** | Performs DNS queries, providing detailed information about DNS records. | dig google.com |
| **host** | Performs DNS lookups, displaying domain name to IP address resolution. | host google.com |
| **whois** | Retrieves information about domain registration and ownership. | whois google.com |
| **ssh** | Provides secure remote access to a system. | ssh username@hostname |
| **scp** | Securely copies files between hosts over a network. | scp file.txt username@hostname:/path/ |
| **ftp** | Transfers files between hosts using the File Transfer Protocol (FTP). | ftp hostname |
| **telnet** | Establishes interactive text-based communication with a remote host. | telnet hostname |
| **netstat** | Displays network connections, routing tables, interface statistics, masquerade connections, and multicast memberships. | netstat -tuln |
| **ifconfig** | Displays or configures network interfaces and their settings. | ifconfig |
| **iwconfig** | Configures wireless network interfaces. | iwconfig wlan0 |
| **route** | Displays or modifies the IP routing table. | route -n |
| **arp** | Displays or modifies the Address Resolution Protocol (ARP) cache. | arp -a |
| **ss** | Displays socket statistics. | ss -tuln |
| **hostname** | Displays or sets the system’s hostname. | hostname |
| **mtr** | Combines the functionality of ping and traceroute, providing detailed network diagnostic information. | mtr google.com |

**System Administration Unix Command**

| **Command** | **Description** | **Example** |
| --- | --- | --- |
| **df** | Displays disk space usage. | df -h |
| **du** | Displays disk usage of files and directories. | du -sh /path/to/directory |
| **crontab -e** | Manages cron jobs, which are scheduled tasks that run at predefined times or intervals. | crontab -e |

**Text Editors in Unix**

| **Text Editor** | **Description** | **Example** |
| --- | --- | --- |
| **Vi / Vim** | Vi (Vim) is a highly configurable, powerful, and feature-rich text editor based on the original Vi editor. Vim offers modes for both command-line operations and text editing. | Open a file with Vim: vim filename Exit Vim editor: Press Esc, then type :wq and press Enter |
| **Emacs** | Emacs is a versatile text editor with extensive customization capabilities and support for various programming languages. | Open a file with Emacs: emacs filename Save and exit Emacs: Press Ctrl + X, then Ctrl + S and Ctrl + X, then Ctrl + C to exit |
| **Nano** | Nano is a simple and user-friendly text editor designed for ease of use and accessibility. | Open a file with Nano: nano filename Save and exit Nano: Press Ctrl + O, then Ctrl + X |
| **Ed** | Ed is a standard Unix text editor that operates in line-oriented mode, making it suitable for batch processing and automation tasks. | Open a file with Ed: ed filename Exit Ed editor: Type q and press Enter |
| **Jed** | Jed is a lightweight yet powerful text editor that provides an intuitive interface and support for various programming languages. | Open a file with Jed: jed filename Save and exit Jed: Press Alt + X, then type exit and press Enter |

Unix Commands – FAQs

**What is Unix and how does it differ from other operating systems?**

This question aims to clarify the unique features and characteristics of Unix compared to other operating systems like Windows or macOS.

**Who developed Unix and what is its significance in the history of computing?**

Users might want to know about the origins of Unix, its developers, and its role in shaping the modern computing landscape.

**What are some popular Unix-like operating systems and how do they relate to Unix?**

This question seeks to understand the relationship between Unix and Unix-like systems such as Linux, and the various distributions available for different purposes.

**What are the essential Unix commands and how are they used?**

Users may seek clarification on the basic Unix commands listed in the article and how they can be applied in practical scenarios.

**Conclusion**

In conclusion, Unix commands serve as a fundamental toolkit for navigating and managing the Unix operating system, which has evolved from its inception in the 1960s to become one of the most widely used OS platforms across various domains including personal computing, servers, and mobile devices. From its origins at Bell Labs with developers Dennis M. Ritchie and Ken Thompson to the birth of the C programming language and the subsequent emergence of Unix-like systems such as Linux, the Unix ecosystem has significantly shaped the computing landscape. Understanding basic Unix commands is essential for users to efficiently manipulate files, manage processes, configure networks, and perform system administration tasks, thereby empowering them to leverage the full potential of Unix-based systems for diverse computing needs.

**Introduction**

**Definition of Bash scripting**

A bash script is a file containing a sequence of commands that are executed by the bash program line by line. It allows you to perform a series of actions, such as navigating to a specific directory, creating a folder, and launching a process using the command line.

By saving these commands in a script, you can repeat the same sequence of steps multiple times and execute them by running the script.

**Advantages of Bash scripting**

Bash scripting is a powerful and versatile tool for automating system administration tasks, managing system resources, and performing other routine tasks in Unix/Linux systems. Some advantages of shell scripting are:

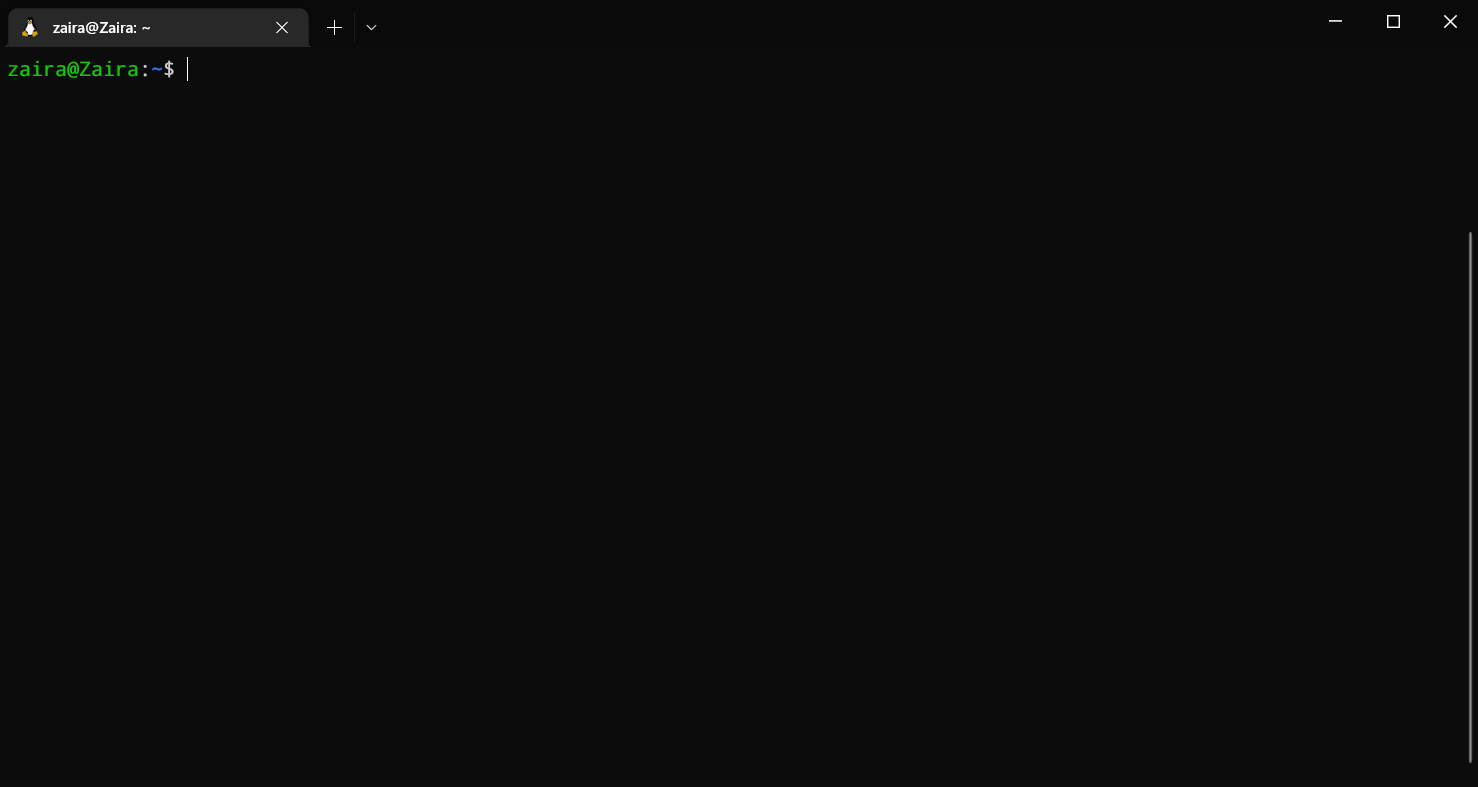
* **Automation**: Shell scripts allow you to automate repetitive tasks and processes, saving time and reducing the risk of errors that can occur with manual execution.
* **Portability**: Shell scripts can be run on various platforms and operating systems, including Unix, Linux, macOS, and even Windows through the use of emulators or virtual machines.
* **Flexibility**: Shell scripts are highly customizable and can be easily modified to suit specific requirements. They can also be combined with other programming languages or utilities to create more powerful scripts.
* **Accessibility**: Shell scripts are easy to write and don't require any special tools or software. They can be edited using any text editor, and most operating systems have a built-in shell interpreter.
* **Integration**: Shell scripts can be integrated with other tools and applications, such as databases, web servers, and cloud services, allowing for more complex automation and system management tasks.
* **Debugging**: Shell scripts are easy to debug, and most shells have built-in debugging and error-reporting tools that can help identify and fix issues quickly.

**Overview of Bash shell and command line interface**

The terms "shell" and "bash" are used interchangeably. But there is a subtle difference between the two.

The term "shell" refers to a program that provides a command-line interface for interacting with an operating system. Bash (Bourne-Again SHell) is one of the most commonly used Unix/Linux shells and is the default shell in many Linux distributions.

A shell or command-line interface looks like this:

The shell accepts commands from the user and displays the output

In the above output, root@vilas is the shell prompt. When a shell is used interactively, it displays a $ when it is waiting for a command from the user.

If the shell is running as root (a user with administrative rights), the prompt is changed to #. The superuser shell prompt looks like this:

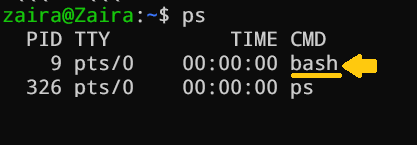
[root@host ~]#

Although Bash is a type of shell, there are other shells available as well, such as Korn shell (ksh), C shell (csh), and Z shell (zsh). Each shell has its own syntax and set of features, but they all share the common purpose of providing a command-line interface for interacting with the operating system.

You can determine your shell type using the ps command:

ps

Here is the output for me:

Checking the shell type. I'm using bash shell

In summary, while "shell" is a broad term that refers to any program that provides a command-line interface, "Bash" is a specific type of shell that is widely used in Unix/Linux systems.

Note: In this tutorial, we will be using the "bash" shell.

**How to Get Started with Bash Scripting**

**Running Bash commands from the command line**

As mentioned earlier, the shell prompt looks something like this:

[username@host ~]$

You can enter any command after the $ sign and see the output on the terminal.

Generally, commands follow this syntax:

command [OPTIONS] arguments

Let's discuss a few basic bash commands and see their outputs. Make sure to follow along :)

* date: Displays the current date

root@vilas:~/shell-tutorial$ date

Tue Mar 14 13:08:57 PKT 2023

* pwd: Displays the present working directory.

root@vilas:~/shell-tutorial$ pwd

/home/zaira/shell-tutorial

* ls: Lists the contents of the current directory.

root@vilas:~/shell-tutorial$ ls

check\_plaindrome.sh count\_odd.sh env log temp

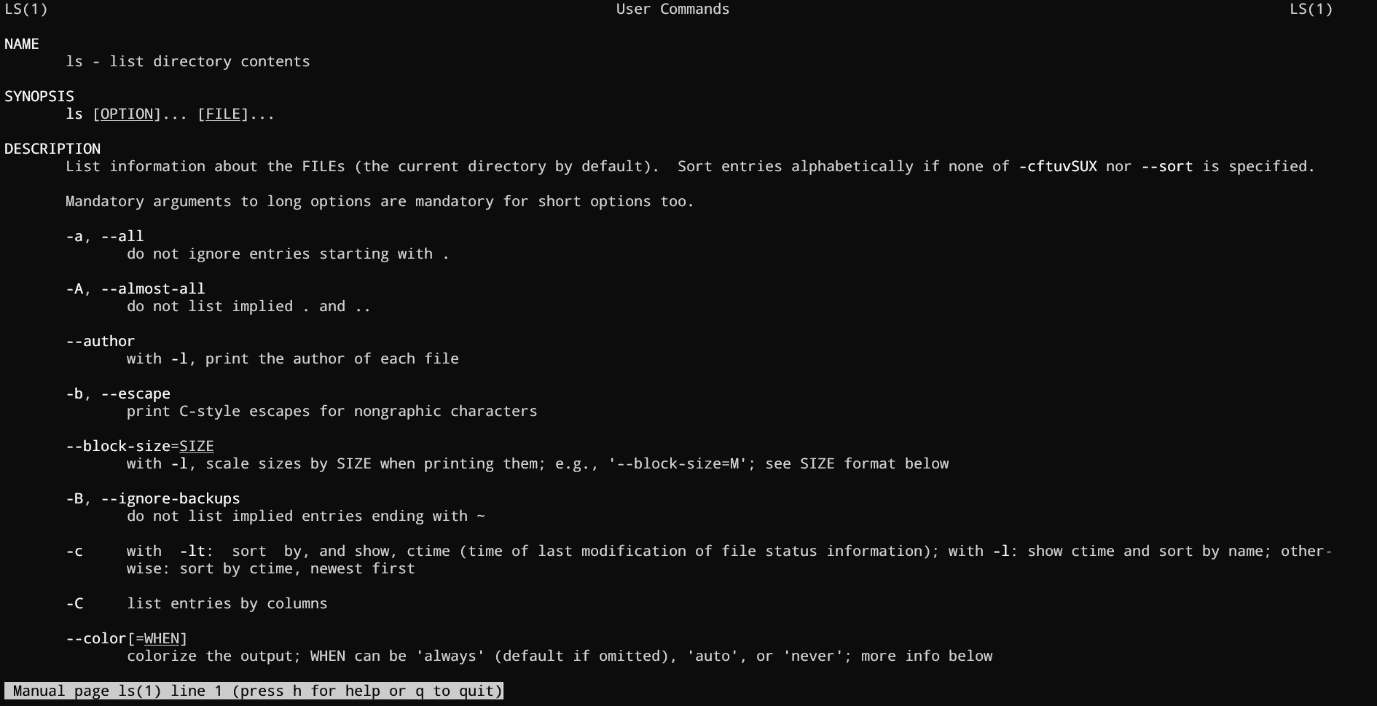
* echo: Prints a string of text, or value of a variable to the terminal.

root@vilas:~/shell-tutorial$ echo "Hello bash"

Hello bash

You can always refer to a commands manual with the man command.

For example, the manual for ls looks something like this:

You can see options for a command in detail using man

**How to Create and Execute Bash scripts**

**Script naming conventions**

By naming convention, bash scripts end with .sh. However, bash scripts can run perfectly fine without the sh extension.

**Adding the Shebang**

Bash scripts start with a shebang. Shebang is a combination of bash # and bang ! followed by the bash shell path. This is the first line of the script. Shebang tells the shell to execute it via bash shell. Shebang is simply an absolute path to the bash interpreter.

Below is an example of the shebang statement.

**#!/bin/bash**

You can find your bash shell path (which may vary from the above) using the command:

which bash

**Creating our first bash script**

Our first script prompts the user to enter a path. In return, its contents will be listed.

Create a file named run\_all.sh using the vi command. You can use any editor of your choice.

vi run\_all.sh

Add the following commands in your file and save it:

Explain

**#!/bin/bash**

echo "Today is " `date`

echo -e "\nenter the path to directory"

read the\_path

echo -e "\n you path has the following files and folders: "

ls $the\_path

Script to print contents of a user supplied directory

Let's take a deeper look at the script line by line. I am displaying the same script again, but this time with line numbers.

Explain

1 #!/bin/bash

2 echo "Today is " `date`

3

4 echo -e "\nenter the path to directory"

5 read the\_path

6

7 echo -e "\n you path has the following files and folders: "

8 ls $the\_path

* Line #1: The shebang (#!/bin/bash) points toward the bash shell path.
* Line #2: The echo command is displaying the current date and time on the terminal. Note that the date is in backticks.
* Line #4: We want the user to enter a valid path.
* Line #5: The read command reads the input and stores it in the variable the\_path.
* line #8: The ls command takes the variable with the stored path and displays the current files and folders.

**Executing the bash script**

To make the script executable, assign execution rights to your user using this command:

chmod u+x run\_all.sh

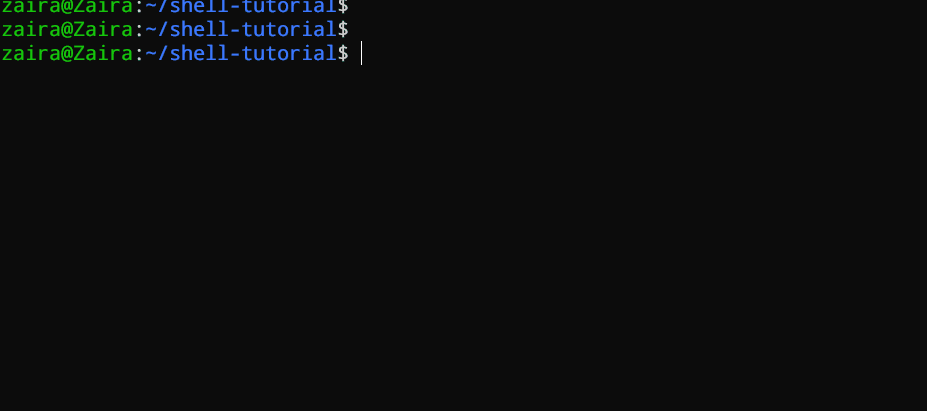
Here,

* chmod modifies the ownership of a file for the current user :u.
* +x adds the execution rights to the current user. This means that the user who is the owner can now run the script.
* run\_all.sh is the file we wish to run.

You can run the script using any of the mentioned methods:

* sh run\_all.sh
* bash run\_all.sh
* ./run\_all.sh

Let's see it running in action 🚀



**Bash Scripting Basics**

**Comments in bash scripting**

Comments start with a # in bash scripting. This means that any line that begins with a # is a comment and will be ignored by the interpreter.

Comments are very helpful in documenting the code, and it is a good practice to add them to help others understand the code.

These are examples of comments:

# This is an example comment

# Both of these lines will be ignored by the interpreter

**Variables and data types in Bash**

Variables let you store data. You can use variables to read, access, and manipulate data throughout your script.

There are no data types in Bash. In Bash, a variable is capable of storing numeric values, individual characters, or strings of characters.

In Bash, you can use and set the variable values in the following ways:

1. Assign the value directly:

country=CountryValue

2.  Assign the value based on the output obtained from a program or command, using command substitution. Note that $ is required to access an existing variable's value.

same\_country=$country

This assigns the value of countryto the new variable same\_country

To access the variable value, append $ to the variable name.

Explain

root@vilas:~$ country=CountryValue

root@vilas:~$ echo $country

CountryValue

root@vilas:~$ new\_country=$country

root@vilas:~$ echo $new\_country

CountryValue

Assigning and printing variable values

**Variable naming conventions**

In Bash scripting, the following are the variable naming conventions:

1. Variable names should start with a letter or an underscore (\_).
2. Variable names can contain letters, numbers, and underscores (\_).
3. Variable names are case-sensitive.
4. Variable names should not contain spaces or special characters.
5. Use descriptive names that reflect the purpose of the variable.
6. Avoid using reserved keywords, such as if, then, else, fi, and so on as variable names.

Here are some examples of valid variable names in Bash:

Explain

name

count

\_var

myVar

MY\_VAR

And here are some examples of invalid variable names:

2ndvar (variable name starts with a number)

my var (variable name contains a space)

my-var (variable name contains a hyphen)

Following these naming conventions helps make Bash scripts more readable and easier to maintain.

**Input and output in Bash scripts**

**Gathering input**

In this section, we'll discuss some methods to provide input to our scripts.

1. Reading the user input and storing it in a variable

We can read the user input using the read command.

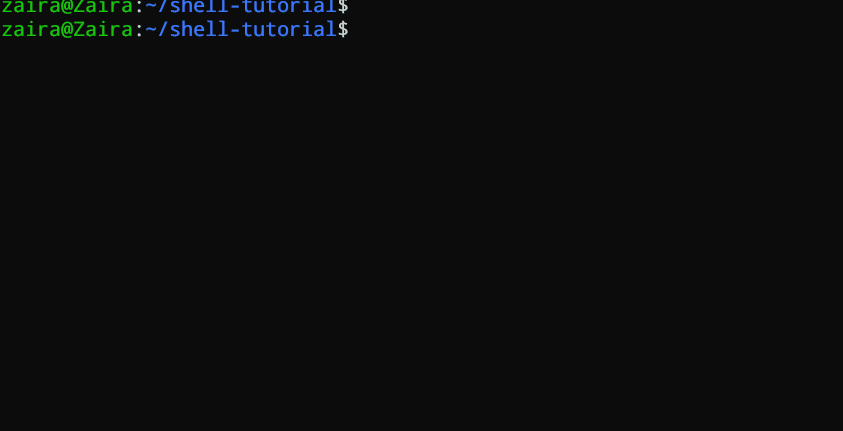
Explain

**#!/bin/bash**

echo "What's your name?"

read entered\_name

echo -e "\nWelcome to bash tutorial" $entered\_name



2.  Reading from a file

This code reads each line from a file named input.txt and prints it to the terminal. We'll study while loops later in this article.

Explain

while read line

do

echo $line

done < input.txt

3.  Command line arguments

In a bash script or function, $1 denotes the initial argument passed, $2 denotes the second argument passed, and so forth.

This script takes a name as a command-line argument and prints a personalized greeting.

echo "Hello, $1!"

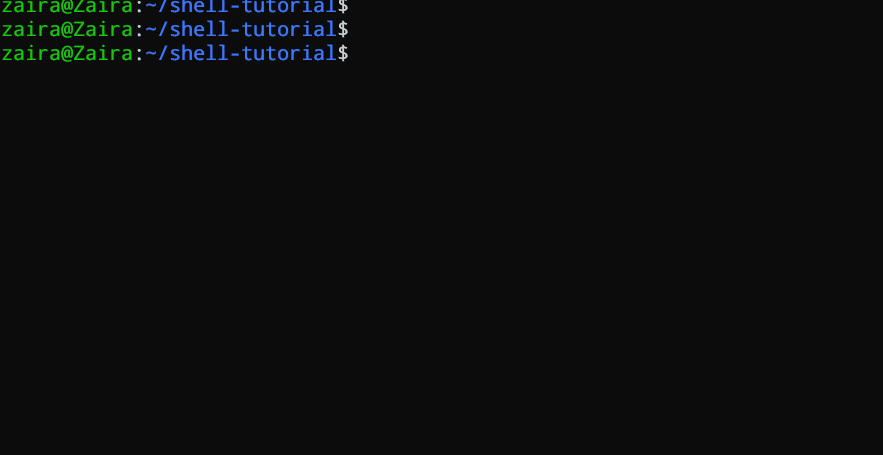
We have supplied Zaira as our argument to the script.

**#!/bin/bash**

echo "Hello, $1!"

The code for the script: greeting.sh

**Output:**



**Displaying output**

Here we'll discuss some methods to receive output from the scripts.

1. Printing to the terminal:

echo "Hello, World!"

This prints the text "Hello, World!" to the terminal.

2.  Writing to a file:

echo "This is some text." > output.txt

This writes the text "This is some text." to a file named output.txt. Note that the >operator overwrites a file if it already has some content.

3.  Appending to a file:

echo "More text." >> output.txt

This appends the text "More text." to the end of the file output.txt.

4.  Redirecting output:

ls > files.txt

This lists the files in the current directory and writes the output to a file named files.txt. You can redirect output of any command to a file this way.

**Basic Bash commands (echo, read, etc.)**

Here is a list of some of the most commonly used bash commands:

1. cd: Change the directory to a different location.
2. ls: List the contents of the current directory.
3. mkdir: Create a new directory.
4. touch: Create a new file.
5. rm: Remove a file or directory.
6. cp: Copy a file or directory.
7. mv: Move or rename a file or directory.
8. echo: Print text to the terminal.
9. cat: Concatenate and print the contents of a file.
10. grep: Search for a pattern in a file.
11. chmod: Change the permissions of a file or directory.
12. sudo: Run a command with administrative privileges.
13. df: Display the amount of disk space available.
14. history: Show a list of previously executed commands.
15. ps: Display information about running processes.

**Conditional statements (if/else)**

Expressions that produce a boolean result, either true or false, are called conditions. There are several ways to evaluate conditions, including if, if-else, if-elif-else, and nested conditionals.

**Syntax**:

Explain

if [[ condition ]];

then

statement

elif [[ condition ]]; then

statement

else

do this by default

fi

Syntax of bash conditional statements

We can use logical operators such as AND -a and OR -o to make comparisons that have more significance.

if [ $a -gt 60 -a $b -lt 100 ]

This statement checks if both conditions are true: a is greater than 60 AND b is less than 100.

Let's see an example of a Bash script that uses if, if-else, and if-elif-else statements to determine if a user-inputted number is positive, negative, or zero:

Explain

**#!/bin/bash**

echo "Please enter a number: "

read num

if [ $num -gt 0 ]; then

echo "$num is positive"

elif [ $num -lt 0 ]; then

echo "$num is negative"

else

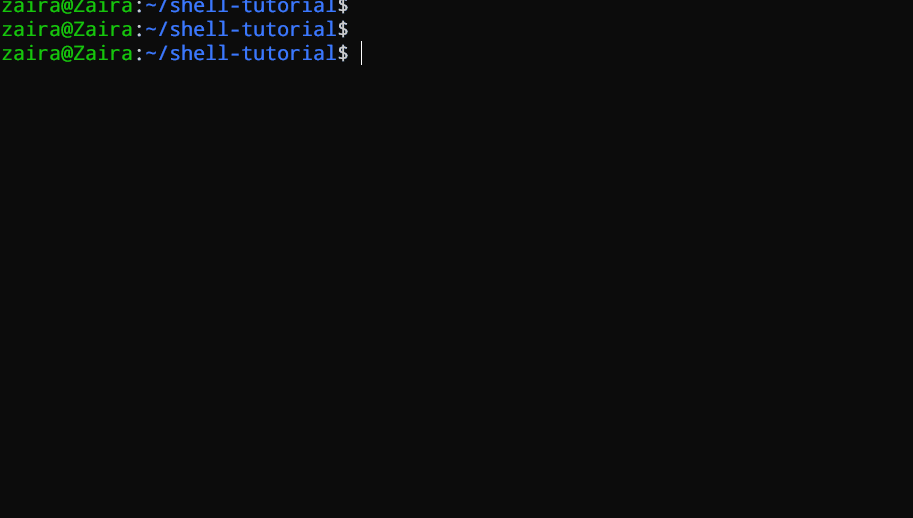
echo "$num is zero"

fi

Script to determine if a number is positive, negative, or zero

The script first prompts the user to enter a number. Then, it uses an if statement to check if the number is greater than 0. If it is, the script outputs that the number is positive. If the number is not greater than 0, the script moves on to the next statement, which is an if-elif statement. Here, the script checks if the number is less than 0. If it is, the script outputs that the number is negative. Finally, if the number is neither greater than 0 nor less than 0, the script uses an else statement to output that the number is zero.

Seeing it in action 🚀



**Looping and Branching in Bash**

**While loop**

While loops check for a condition and loop until the condition remains true. We need to provide a counter statement that increments the counter to control loop execution.

In the example below, (( i += 1 )) is the counter statement that increments the value of i. The loop will run exactly 10 times.

Explain

**#!/bin/bash**

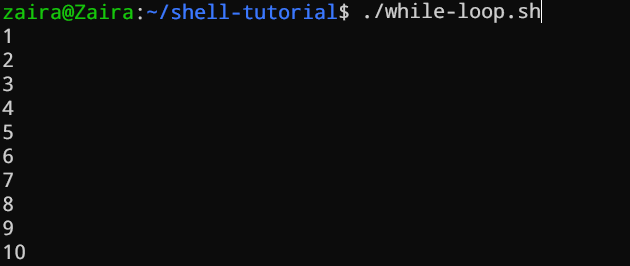
i=1

while [[ $i -le 10 ]] ; do

echo "$i"

(( i += 1 ))

done

While loop that iterates 10 times.

**For loop**

The for loop, just like the while loop, allows you to execute statements a specific number of times. Each loop differs in its syntax and usage.

In the example below, the loop will iterate 5 times.

Explain

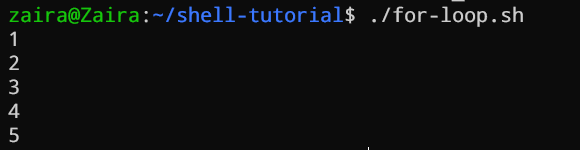
**#!/bin/bash**

for i in {1..5}

do

echo $i

done

For loop that iterates 5 times.

**Case statements**

In Bash, case statements are used to compare a given value against a list of patterns and execute a block of code based on the first pattern that matches. The syntax for a case statement in Bash is as follows:

Explain

case expression in

pattern1)

# code to execute if expression matches pattern1

;;

pattern2)

# code to execute if expression matches pattern2

;;

pattern3)

# code to execute if expression matches pattern3

;;

\*)

# code to execute if none of the above patterns match expression

;;

esac

Case statements syntax

Here, "expression" is the value that we want to compare, and "pattern1", "pattern2", "pattern3", and so on are the patterns that we want to compare it against.

The double semicolon ";;" separates each block of code to execute for each pattern. The asterisk "\*" represents the default case, which executes if none of the specified patterns match the expression.

Let's see an example.

Explain

fruit="apple"

case $fruit in

"apple")

echo "This is a red fruit."

;;

"banana")

echo "This is a yellow fruit."

;;

"orange")

echo "This is an orange fruit."

;;

\*)

echo "Unknown fruit."

;;

esac

Example of case statement

In this example, since the value of "fruit" is "apple", the first pattern matches, and the block of code that echoes "This is a red fruit." is executed. If the value of "fruit" were instead "banana", the second pattern would match and the block of code that echoes "This is a yellow fruit." would execute, and so on. If the value of "fruit" does not match any of the specified patterns, the default case is executed, which echoes "Unknown fruit."