What is Physical Machine or Server?

A physical machine or physical server refers to a tangible, hardware-based computing system that exists physically, as opposed to existing virtually or in the cloud. It is a standalone piece of hardware with its own physical components, such as a motherboard, central processing unit (CPU), memory (RAM), storage devices, network interfaces, and other hardware components.

Here are some key characteristics of a physical machine or physical server:

- **Independent Hardware:** A physical machine is a self-contained unit with its own dedicated hardware resources. It is not shared with other virtual machines or servers.
- **Operating System (OS):** Physical machines typically run a full-fledged operating system (OS) directly on the hardware. Common server operating systems include Linux distributions (e.g., Ubuntu, CentOS) or Windows Server.
- **Direct Access to Hardware Resources:** Applications and the operating system on a physical machine have direct access to the underlying hardware resources, providing optimal performance without the virtualization overhead.
- **Single Tenancy:** A physical server is single-tenancy, meaning that the entire machine is dedicated to a specific set of tasks or applications. This is in contrast to virtualization, where multiple virtual machines can run on the same physical server.
- **Deployment Data Centers or On-Premises:** Physical servers can be deployed in on-premises data centers or co-location facilities, providing organizations with control over the physical infrastructure.
- **Higher Resource Capacity:** Physical servers typically have higher resource capacity compared to individual virtual machines on the same hardware, making them suitable for resource-intensive applications.
- **Maintenance and Upgrades:** Physical servers require manual maintenance, upgrades, and replacements of hardware components. This may involve downtime for the services running on the server.

While physical machines have been the traditional form of server infrastructure, the advent of virtualization and cloud computing has introduced alternative deployment models. Virtual machines (VMs) allow multiple virtual servers to run on a single physical machine, providing greater flexibility and resource efficiency. In contrast, physical machines offer advantages in terms of raw performance, predictable resource allocation, and control over the hardware environment. The choice between physical and virtual infrastructure depends on specific use cases, performance requirements, and organizational preferences.

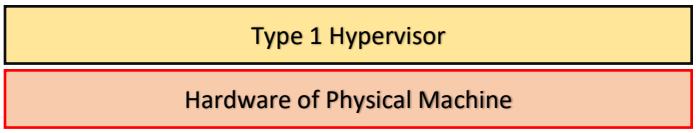
What is Hypervisor and its role and responsibilities?

A hypervisor, also known as a Virtual Machine Monitor (VMM), is a software or firmware layer that allows multiple operating systems (OS) to share a single physical hardware host. It facilitates the creation and management of virtual machines (VMs) by abstracting the underlying hardware resources and providing each VM with a virtualized environment.

The primary role of a hypervisor is to enable virtualization, allowing multiple operating systems to run concurrently on a single physical machine.

There are two main types of hypervisors:

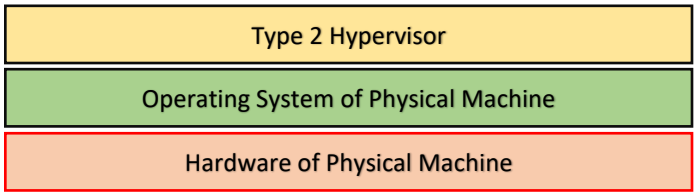
- **Type 1 Hypervisor (Bare-Metal Hypervisor):**
  - **Direct Interaction with Hardware:** A Type 1 hypervisor runs directly on the physical hardware without the need for a host operating system. It has direct access to the underlying hardware resources.



- **Efficiency and Performance:** Since there is no intermediate host OS, Type 1 hypervisors often provide better performance and resource utilization compared to Type 2 hypervisors.
- **Examples:**
  - VMware vSphere/ESXi
  - Microsoft Hyper-V (when installed as standalone without Windows OS)

Type 2 Hypervisor (Hosted Hypervisor):

- **Runs top of Host Operating System:** A Type 2 hypervisor runs on top of a host operating system in your physical machine. It relies on the host OS to manage hardware resources and provides virtualization capabilities to guest operating systems.



- **Ease of Use Setup:** Type 2 hypervisors are easier to install and use, making them suitable for development and testing environments.
- **Examples:**
  - VMware Workstation
  - Oracle VirtualBox
  - XEN

Roles and Responsibilities of hypervisors:

- **Virtualization of Hardware:** The hypervisor abstracts physical hardware resources (CPU, memory, storage, network) and presents them to VMs as virtualized resources.
- **Isolation of Virtual Machine:** Each VM operates in its own isolated environment, preventing interference and conflicts between VMs. This isolation enhances security and stability.
- **Resources Allocation and Scheduling:** The hypervisor manages the allocation of physical resources to VMs, ensuring fair distribution and efficient utilization. It also handles resource scheduling to avoid contention.
- **Virtual Machine Creation and Configuration:** The hypervisor allows the creation, configuration, and deployment of virtual machines. Users can define VM characteristics such as CPU, memory, and disk space.
- **Snapshot and Cloning:** Hypervisors often support features like snapshotting, which allows the capture and restoration of a VM's state, and cloning, which enables the rapid creation of identical VM copies.
- **Live Migration:** Some hypervisors support live migration, allowing VMs to be moved from one physical host to another without downtime. This is useful for load balancing and maintenance.
- **Security Features:** Hypervisors may include security features such as secure boot, encrypted VMs, and isolation mechanisms to enhance the overall security of virtualized environments.
- **Monitoring and Reporting:** Hypervisors provide tools and interfaces for monitoring the performance and health of virtualized environments. This includes metrics related to resource usage, VM activity, and more.

Hypervisors play a crucial role in data centers, cloud computing environments, and virtualization platforms by enabling the efficient and flexible use of hardware resources. They facilitate the creation and management of VMs, allowing organizations to run multiple operating systems and applications on a single physical server.

What is Virtualization?

Virtualization is a technology that allows multiple virtual instances of operating systems (OS), applications, or resources to run on a single physical machine. The primary goal of virtualization is to maximize resource utilization, increase flexibility, and improve efficiency in computing environments. Virtualization is basically creating a lot of virtual machines or virtual servers top of physical machines.

Key components and concepts of virtualization include:

- **Hypervisor (Virtual Machine Monitor - VMM):** The hypervisor is a software or firmware layer that enables the creation and management of virtual machines (VMs) on a physical host. It abstracts and virtualizes the underlying hardware, allowing multiple VMs to coexist on the same physical machine.
- **Virtual Machines (VMs):** VMs are instances of virtualized operating systems and applications that run on a host machine. Each VM operates as if it were a standalone physical machine, with its own virtualized hardware resources, including CPU, memory, storage, and network interfaces.
- **Host Machine:** The physical hardware that runs the hypervisor and hosts the virtual machines. It can be a server, desktop, or other computing device.
- **Resource Pooling:** Virtualization allows the pooling of physical resources such as CPU, memory, and storage. These resources are then dynamically allocated to VMs based on demand.
- **Isolation:** VMs are isolated from each other, providing security and preventing interference. Failures or issues in one VM typically do not affect others.
- **Snapshot and Cloning:** Virtualization platforms often support features like snapshotting, which captures the current state of a VM for backup or recovery, and cloning, which allows the rapid creation of identical VM copies.
- **Live Migration:** Some virtualization systems support live migration, enabling the movement of VMs from one physical host to another without disrupting services. This is useful for load balancing and maintenance.
- **Flexible Scaling:** Virtualization provides the ability to scale resources up or down as needed. VMs can be easily provisioned or decommissioned, allowing for efficient resource allocation.

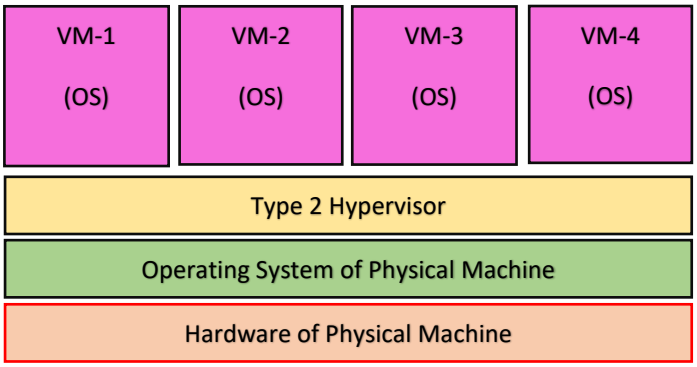
Types of virtualization include:

- **Server Virtualization:** Multiple virtual servers run on a single physical server, enabling better resource utilization and management.
- **Desktop Virtualization:** Virtual desktops run on a central server, allowing users to access their desktop environments from different devices.
- **Network Virtualization:** Virtualization of network resources, enabling the creation of virtual networks and network services.
- **Storage Virtualization:** Abstraction of physical storage resources to create a virtualized storage pool that can be dynamically allocated.

Virtualization is a fundamental technology in modern computing, enabling the consolidation of workloads, improved hardware utilization, and increased agility in managing IT infrastructure. It plays a crucial role in data centers, cloud computing, and various other computing environments.

What is Virtual Machine (VMs)?

A Virtual Machine (VM) is a software emulation of a physical computer that runs an operating system and applications. It allows multiple instances of operating systems to coexist and run on a single physical machine. Each virtual machine operates independently and is isolated from the others, providing a way to consolidate and efficiently utilize hardware resources.



Importance of Virtual Machines (VMs):

- **Resource Utilization:** VMs allow for better utilization of physical hardware by running multiple operating systems on a single server. This leads to improved resource efficiency and reduced hardware costs.
- **Isolation:** VMs provide isolation between different operating systems and applications. Failures or issues in one VM do not impact others, enhancing security and stability. The VMs are tightly separated, because they have individual operating system.
- **Flexibility and Scalability:** VMs can be easily provisioned or decommissioned, providing flexibility in scaling resources based on demand. This agility is crucial in dynamic computing environments.
- **Consolidation:** Virtualization enables the consolidation of workloads, allowing organizations to run multiple applications on a single physical server without interference.
- **Snapshot and Cloning:** VMs support snapshotting, allowing the capture of a VM's current state for backup or recovery purposes. Cloning allows the rapid creation of identical VM copies.
- **Testing and Development:** VMs are widely used in testing and development environments, providing developers with isolated environments to test software or deploy applications.
- **Legacy Application Support:** VMs can run older or legacy applications that might not be compatible with the host operating system or hardware.

Roles and Responsibilities of Virtual Machines (VMs):

- **Execution of Operating System:** VMs run complete operating systems just like physical machines. They execute applications and services within their own virtualized environment.
- **Resources Allocation and Managements:** VMs receive virtualized resources from the hypervisor, including CPU, memory, storage, and network interfaces. The hypervisor manages the allocation of these resources based on demand.
- **Isolation and Security:** VMs provide isolation between different instances, ensuring that each VM operates independently. This isolation enhances security and prevents interference between VMs.
- **Application Hosting:** VMs host applications and services, allowing organizations to deploy multiple applications on a single physical server without conflicts.

- **Dynamic Scaling:** VMs support dynamic scaling, allowing for the adjustment of allocated resources to meet changing workloads.
- **Snapshotting and Scaling:** VMs support features such as snapshotting and cloning, providing mechanisms for backup, recovery, and rapid deployment.
- **Compatibility and Probability:** VMs encapsulate an entire operating system and its dependencies, making them portable across different environments. This enhances compatibility and simplifies deployment.
- **Integration and Cloud Services:** VMs are integral to cloud computing services, where they provide the foundation for deploying and managing virtualized resources.

Virtual machines play a critical role in modern IT infrastructure, offering flexibility, efficiency, and scalability. They have become a cornerstone technology in data centers, cloud computing environments, and various enterprise computing scenarios.

What is difference if I connect the "VM from MobaXterm from my local machine" and "AWS CLI?"

It seems there might be a slight confusion in your question. Connecting to a virtual machine (VM) using MobaXtrem and using the AWS Command Line Interface (AWS CLI) are two different things, and they serve different purposes.

Connecting to a VM using MobaXterm:

- MobaXterm is a terminal software for Windows that provides a variety of tools, including an integrated SSH client.
- When you connect to a VM using MobaXterm, you typically use SSH (Secure Shell) to access the command line of the remote machine.
- This is useful when you have a virtual machine or a remote server, and you want to execute commands, transfer files, or perform other administrative tasks.

Connecting to a VM using AWS CLI:

- AWS CLI is a command-line interface for interacting with AWS services. It allows you to manage and automate AWS resources directly from your local machine's terminal or command prompt.
- When you use AWS CLI, you are not connecting to a specific VM but rather interacting with AWS services such as EC2 (Elastic Compute Cloud), S3 (Simple Storage Service), etc.
- AWS CLI commands are used to manage AWS resources, deploy applications, and perform various tasks related to your AWS account.

What is Shell?

A shell is a command-line interface (CLI) or a command interpreter that provides a user interface for interacting with an operating system or a software application. It allows users to execute commands, run scripts, and perform various tasks by typing text-based commands.

Key characteristics of a shell include:

- **Command Interpretation:** The shell interprets user commands and translates them into instructions that the operating system or software can understand and execute.
- **Scripting:** Shells often support scripting languages, allowing users to write scripts (sequences of commands) for automation and customization of tasks.
- **User Interface:** Shells provide a text-based interface where users interact with the system by typing commands. This interface is commonly referred to as the command line or terminal.
- **Environment Variables:** Shells manage environment variables, which are variables that store information about the environment in which commands are executed.
- **Redirection and Pipelines:** Shells allow users to redirect the input and output of commands, as well as create pipelines to pass the output of one command as input to another.
- **Job Control:** Shells provide features for managing processes and background jobs, such as putting jobs in the background, bringing them to the foreground, and managing their execution.
- **Customization:** Users can customize their shell environment by configuring settings, defining aliases, and creating shell scripts to automate repetitive tasks.

Shell Scripting:

Shell scripting refers to the **creation** and **execution** of a **sequence of commands** or a script using a shell, which is a command-line interpreter. Shell scripting is a powerful way to automate tasks, execute commands in a specific sequence, and perform various operations within a command-line environment. Shell scripts are typically written using scripting languages supported by shells, such as Bash, Zsh, or PowerShell.

Shell scripting is commonly used in various scenarios, including **system administration**, **DevOps**, **software deployment**, and **automation of routine tasks**. It provides a flexible and efficient way to work with command-line interfaces and automate complex workflows.

Why Shell Scripting used by a DevOps Engineers:

Shell scripting is widely used by DevOps engineers for several reasons, as it plays a crucial role in automating and streamlining various tasks within the DevOps workflow.

Here are some key reasons why shell scripting is essential for DevOps engineers:

- **Automation of Routine Task:** DevOps involves managing and automating repetitive tasks in the software development and deployment lifecycle. Shell scripts enable automation of routine tasks such as code builds, deployments, and configuration management.
- **Infrastructure as Code (IaC):** Shell scripts are often used to write Infrastructure as Code (IaC) scripts that define and provision infrastructure resources. Tools like Terraform, Ansible, and Chef Leverage shell scripts to automate the provisioning and configuration of infrastructure.
- **Configuration Management:** DevOps engineers use shell scripts to configure and manage software applications, servers, and other components of the IT infrastructure. Shell scripts define the desired state of the system and ensure consistency across environments.

- **Environment Setup and Teardown:** Shell scripts are used to set up development and testing environments quickly. DevOps engineers can create scripts to install dependencies, configure services, and prepare the environment for application development and testing.
- **Log Analysis and Monitoring:** Shell scripts assist in automating log analysis and monitoring tasks. DevOps engineers can create scripts to parse log files, extract relevant information, and trigger alerts based on predefined conditions.
- **Task Orchestration:** Shell scripts provide a way to orchestrate complex workflows by executing a sequence of tasks in a specified order. This is particularly useful in coordinating activities across different stages of the DevOps pipeline.
- **Cross-Platform Compatibility:** Shell scripts are inherently portable and can be executed across various operating systems. This flexibility is beneficial in heterogeneous environments where different platforms need to be managed consistently.
- **Resource Management:** Shell scripts are used for resource management tasks, including starting and stopping services, managing user accounts, and monitoring system resources. This ensures efficient utilization of resources in the IT infrastructure.
- **Integration with Other Tools:** DevOps tools and platforms often provide command-line interfaces that can be scripted using shell scripts. This allows DevOps engineers to integrate different tools seamlessly.
- **Customization and Extensibility:** DevOps processes often require customization to suit specific organizational needs. Shell scripts provide a flexible and extensible way to tailor automation workflows according to unique requirements.

Commands in Linux for Shell Scripting:

echo:

- **Description:** Prints text to the terminal.
- **Uses/Importance:** Used for displaying messages and variables in scripts.
- **Example:**

```bash
echo "Hello, World!"
```

cd:

- **Description:** Changes the current working directory.
- **Uses/Importance:** Useful for navigating directories in scripts.
- **Example:**

```bash
cd /path/to/directory
```

ls:

- **Description:** Lists files and directories in the current directory.
- **Uses/Importance:** Used to gather information about files in scripts.
- **Example:**

```bash
ls -l
```

pwd:
- **Description:** Prints the current working directory.
- **Uses/Importance:** Useful for obtaining the current directory path in scripts.
- **Example:**

cp:
- **Description:** Copies files or directories.
- **Uses/Importance:** Used for creating backups or duplicating files in scripts.
- **Example:**

mv:
- **Description:** Moves or renames files or directories.
- **Uses/Importance:** Useful for organizing files and directories in scripts.
- **Example:**

rm:
- **Description:** Removes files or directories.
- **Uses/Importance:** Important for cleaning up unnecessary files in scripts.
- **Example:**

touch:
- **Description:** Creates an empty file or updates the timestamp of an existing file.
- **Uses/Importance:** Useful for creating placeholder files in scripts.
- **Example:**

mkdir:
- **Description:** Creates a new directory.
- **Uses/Importance:** Essential for organizing files and creating directory structures in scripts.
- **Example:**

cat:
- **Description:** Concatenates and displays the content of files.
- **Uses/Importance:** Useful for displaying file contents or combining files in scripts.
- **Example:**

grep:
- **Description:** Searches for a pattern in files.
- **Uses/Importance:** Used for text searching and pattern matching in scripts.
- **Example:**

```bash
grep "pattern" file
```

sed:
- **Description:** Stream editor for filtering and transforming text.
- **Uses/Importance:** Useful for text manipulation and substitution in scripts.
- **Example:**

```bash
sed 's/old_string/new_string/' file
```

awk:

- **Description:** Pattern scanning and processing language.
- **Uses/Importance:** Powerful for text and data extraction in scripts.
- **Example:**

```bash
awk '{print $1}' file
```

chmod:

- **Description:** Changes file permissions.
  - **File permission:** Owner permission, group permission, other permission.
    - For, read (r) → 4
    - For, write (w)→ 2
    - For, execute (e)→ 1
- **Uses/Importance:** Important for securing and controlling access to files in scripts.
- **Example:**
  - chmod <owner_per> <group_per > <other_per > <file_name>
  - chmod 777 file_name.txt → give read, write, and execute permission to all the user.

curl:

- **Description: curl** is a command-line tool and library for transferring data with URLs. **It is used to make HTTP requests and receive responses**. **curl** supports various protocols, including HTTP, HTTPS, FTP, FTPS, LDAP, and more.
- **Uses/Importance:**
  - **curl** is essential for interacting with web services, APIs, and other resources from the command line.
  - It is commonly used in scripts, automation, and testing to perform HTTP operations like GET, POST, PUT, DELETE, etc.
  - **curl** can be used to download files from the internet directly to the local system.
  - It is useful for debugging and troubleshooting network-related issues.

chown:

- **Description:** Changes file owner and group.
- **Uses/Importance:** Used for modifying file ownership in scripts.
- **Example:**

```bash
chown user:group file
```

ps:

- **Description:** Displays information about active processes.
- **Uses/Importance:** Useful for process management and monitoring in scripts.
- **Example:**

```bash
ps aux
```

kill:

- **Description:** Sends a signal to terminate a process.
- **Uses/Importance:** Used for stopping or killing processes in scripts.
- **Example:**

```bash
kill -9 PID
```

tar:

- **Description:** Archives files and directories into a single file.
- **Uses/Importance:** Essential for compressing and decompressing files in scripts.
- **Example:**

```bash
tar -czvf archive.tar.gz directory/
```

wget:

- **Description:** Downloads files from the internet.
- **Uses/Importance:** Useful for fetching resources in scripts.
- **Example:**

```bash
wget https://example.com/file
```

date:

- **Description:** Displays or sets the system date and time.
- **Uses/Importance:** Used for timestamping and date-related operations in scripts.
- **Example:**

```bash
current_date=$(date +"%Y-%m-%d")
```

Difference between “vim” and “touch” command:

vim and **touch** are two distinct commands with different purposes in a Unix-like operating system.

Here are the main differences between them:

vim:

- **Purpose:** vim is a text editor that allows users to create, edit, and manipulate text files. It provides a full-featured interface with various commands for text editing and navigation.

- **Usages:** To open or create a file using **vim**, you typically provide the filename as an argument. If the file exists, **vim** opens it for editing; otherwise, it creates a new file with the specified name.

```bash
vim filename
```

- **Example:** To open or create a file named "example.txt" with **vim**, you would use:

```bash
vim example.txt
```

- **Note: vim** has a learning curve and is more suitable for users who are comfortable with command-line text editors.

touch:

- **Purpose: touch** is a command used to update the access and modification timestamps of a file or, if the file doesn't exist, to create an empty file with the specified name.

- **Usages:** The basic usage of **touch** involves providing the filename as an argument. If the file exists, **touch** updates its timestamps; if the file doesn't exist, **touch** creates an empty file with the specified name.

```bash
touch filename
```

- **Example:** To create an empty file named "example.txt" or update its timestamps, you would use:

```bash
touch example.txt
```

- **Note: touch** is a simple command primarily used for timestamp manipulation and file creation.

In summary, while both **vim** and **touch** can be used to create files, **vim** is a powerful text editor with advanced editing capabilities, and **touch** is a lightweight command for updating timestamps and creating empty files. The choice between them depends on whether you need to edit the file's content (use **vim**) or simply create or update timestamps (use **touch**).

What is “#!/bin/bash” or “#!/bin/sh”

#!/bin/bash is called a **shebang** or hashbang. It is a special line at the beginning of a script or shell program in Unix-like operating systems. The shebang indicates the path to the interpreter that should be used to execute the script.

In the case of “#!/bin/bash”:

- **“#!”**, This symbol is called the shebang, hashbang, or pound-bang.
- **“/bin/bash”,** It specifies the path to the Bash interpreter (**bash**), which is a widely used shell in Unix-like systems.

When a script is executed, the operating system uses the specified interpreter (in this case, Bash) to interpret and execute the commands in the script.

For example, if you have a Bash script named “myscript.sh” with the following content:

```bash
#!/bin/bash

echo "Hello, world!"
```

When you run **"./myscript.sh"** from the command line, the operating system reads the shebang line, understands that it should use Bash to interpret the script, and executes the script using the Bash interpreter.

This shebang line is essential for proper execution of the script, as it ensures that the script is interpreted by the correct shell or interpreter. It allows you to write scripts in different scripting languages and specify the interpreter to use for each script.

What is "bash", "sh", "ksh", "dash":

"Bash," "sh," "ksh," and "dash" are all different Unix/Linux shells, each with its own characteristics and features:

- **bash (Bourne Again Shell):**
  - **Description:** Bash is a widely used shell that is an enhanced version of the original Bourne Shell (**sh**). It provides advanced features and improvements over the basic shell, including command history, tab completion, and more.
  - **Usages:** Bash is the default shell for many Unix-like operating systems and is the standard shell on Linux systems.
  - **Example:**

```bash
#!/bin/bash

echo "Hello, world!"
```

- **sh (Sh Shell):**
  - **Description:** The Bourne Shell (**sh**) is one of the earliest UNIX shells. It is a simple and lightweight shell with limited features compared to modern shells like Bash.
  - **Usages:** The **sh** shell is still available on many systems, and some scripts use **sh** for portability. However, on some systems, **sh** is symlinked to another shell, like Bash or Dash.
  - **Example:**

```bash
#!/bin/sh
echo "Hello, world!"
```

- **ksh (KornShell):**
  - **Description:** KornShell **ksh** is a shell that combines features from the Bourne Shell, C Shell, and Bash. It is known for its powerful scripting capabilities and interactive features.

- **Usages: ksh** is less common than Bash but is used in various environments. There are different versions, including the original KornShell (ksh88) and the improved KornShell (ksh93).

- **Example:**

```bash
#!/bin/ksh
echo "Hello, world!"
```

dash:

- **Description:** Dash is a lightweight POSIX-compliant shell designed for efficiency and speed. It aims to be a minimal shell, and it is often used as **/bin/sh** on some systems for better script execution performance.

- **Usages:** Dash is the default **/bin/sh** on some Linux distributions, including Debian. It is designed to be a faster alternative for executing scripts.

- **Example:**

```bash
#!/bin/dash
echo "Hello, world!"
```

What is difference between when you execute a shell script file like "sh file_name.sh" and "./file_name.sh":
Both are used to run the script file (.sh), but for

- **"sh file.sh":** this command does not require to give any permission (owner, group, others) to the file.
- **"./file.sh":** Before executing the script file using this command, you need to give file permission (owner, group, others).
  - chmod 777 file.sh → give the read, write and execute permission to all the user.

grep and pipe command:

grep command:

- **Definition:** is a command-line utility in UNIX and Unix-like operating systems that is used to **search** for a **specific pattern** or **regular expression** within a file or a stream of text. The name "grep" stands for "Global Regular Expression Print."

- **Usage:**

```bash
grep [options] pattern [file...]
```

- **Example:**

```bash
grep "search_term" file.txt
```

- **Functionality:**

- **grep** searches for a specified pattern (regular expression) in one or more files.
- It prints lines that match the pattern.
- It is often used in combination with other commands and utilities to filter and process text data

Pipe (|) command:

- **Definition:** The Pipe (|) command is used to combine the output of one command as the input for another command in Unix-like operating systems.
- **Usages:**

```bash
command1 | command2
```

- **Example:**

```bash
cat file.txt | grep "search_term"
```

- **Functionality:**
  - The output of **command1** is used as the input for **command2**.
  - It allows the chaining of commands, enabling the creation of complex data processing pipelines.
  - It facilitates the composition of smaller, specialized commands to achieve more complex tasks.

Common Usages:

Combining **grep** with a pipe is a powerful way to search for specific patterns in the output of other commands, filter data, and extract relevant information.

```bash
ps aux | grep "process_name"
```

In this example, **ps aux** lists all running processes, and the output is then passed through a pipe to **grep** to filter and display only the lines containing the specified process name.

Difference between grep and awk command:

awk command:

- **Purpose: awk** is a more versatile and general-purpose text processing tool. It is used for processing and analyzing structured text data by defining patterns and actions.
- **Usage:**

```bash
awk 'pattern { action }' file
```

- **Example:**

```bash
awk '/search_term/ { print $1 }' file.txt
```

- **Functionality:**
  - Processes input data based on patterns and actions defined in a script.
  - Splits input lines into fields and provides easy access to individual fields.
  - Allows for more complex text processing and data manipulation.
  - Suitable for generating reports, performing calculations, and more.

Key Difference:

- **Pattern Matching:**
  - **grep:** is specialized in simple pattern matching and line printing. It is designed for more complex pattern-based processing, allowing the definition of actions based on patterns.
  - **awk:** is designed for more complex pattern-based processing, allowing the definition of actions based on patterns.
- **Field Processing:**
  - **grep:** typically works with entire lines of text.
  - **awk:** excels at working with fields within lines, allowing easy extraction and manipulation of specific columns of data.
- **Pattern Matching:**
  - **grep:** Use **grep** when the main goal is to search and filter lines based on patterns.
  - **awk:** Use **awk** when more sophisticated text processing, data extraction, or reporting is needed.
- **Use Cases:**
  - **grep:** is more flexible and powerful for handling structured data and implementing custom processing logic.
  - **awk:** is simpler and more focused on basic pattern matching.
- **Flexibility:**
  - **grep:** is more flexible and powerful for handling structured data and implementing custom processing logic.
  - **awk:** is simpler and more focused on basic pattern matching.

What is “set –x”, “set –e”, “set –o pipefail”

These are special settings or options that can be used with the **set** command in Unix-like shells to modify the behavior of shell scripts.

Here's a brief explanation of each:

set –x:

- **Purpose:** Enables debugging mode, which prints each command and its arguments to the standard error output (stderr) before executing them.
- **Effect:** When the script is executed, each line of the script (along with its expanded form) is printed to stderr before being executed. Useful for tracing and debugging script execution.

set –e:

- **Purpose:** Exits the script immediately if any command it runs exits with a non-zero status (i.e., if any command fails).

- **Effect:** If any command within the script fails, the script will terminate immediately, preventing subsequent commands from being executed. This is useful for ensuring that errors are caught and the script doesn't continue in an unpredictable state.

set –o pipefail:

- **Purpose:** Causes a pipeline (commands separated by pipes **|**) to return the exit status of the last (rightmost) command that has a non-zero exit status.
- **Effect:** By default, a pipeline returns the exit status of the last command in the pipeline, which may mask errors in earlier commands. **set –o pipefail** ensures that the overall exit status reflects the failure of any command in the pipeline.

Example:

```bash
#!/bin/bash
set -e  # Exit on error
set -x  # Enable debugging
set -o pipefail  # Capture failures in pipeline
```

What is difference between "curl" and "wget" command:

curl and **wget** are both command-line tools used for downloading files from the internet, but they have some differences in terms of functionality and usage:

"curl" command:

- **curl** command is to show the information (some condition also apply) not download the files.
- **Example:**
  - **curl <http_file_name>**
    - curl https://raw.githubusercontent.com/dibyendubiswas1998/NewsShort/main/logs/trainings_logs.txt (get all logs related info)
  - **filtering the error from log file:**
    - curl https://raw.githubusercontent.com/dibyendubiswas1998/NewsShort/main/logs/trainings_logs.txt | grep "Error"

"wget" command:

- **wget** command helps to download the files.
- **Example:**
  - Here you first download the files and then filter the logs by applying grep command
    - wget https://raw.githubusercontent.com/dibyendubiswas1998/NewsShort/main/logs/trainings_logs.txt (file name is: training_logs.txt)
    - cat training_logs.txt | grep "error"

"find" command:

The **find** command in Unix-like operating systems is a powerful and versatile tool used for searching and locating files and directories based on various criteria. It is particularly useful for finding files that match specific patterns or conditions within a directory hierarchy. Here is a basic overview of the **find** command:

Basic Syntax:

```bash
find [start_directory] [options] [expression]
```

sudo find / -name pam → here find the all pam file.

Key Components:

- **start_directory:** Specifies the starting directory for the search. If not provided, the search starts from the current directory.
- **options:** Control various aspects of the search, such as depth, file types, and more.
- **expression:** Defines the conditions that files or directories must meet to be considered a match.

Common Examples:

- **Find the File by Name:** Searches for files with the specified name.

```bash
find /path/to/search -name "filename"
```

- **Find the File by Type:** Searches for regular files.

```bash
find /path/to/search -type f
```

- **Find Directories:** Searches for directories.

```bash
find /path/to/search -type d
```

- **Find Files Modified Within the Last N Days:** Searches for files modified within the last N days.

```bash
find /path/to/search -mtime -N
```

- **Find Files by Size:** Searches for files larger than 10 megabytes.

```bash
find /path/to/search -size +10M
```

- **Find and Execute a Command on Each Result:** Searches for **“.txt”** files and executes the **“ls -l”** command on each result.

```bash
find /path/to/search -name "*.txt" -exec ls -l {} \;
```

- **Find and Delete File:** Searches for files matching the pattern and deletes them.

```bash
find /path/to/search -name "tempfile*" -delete
```

- **Find and Print Results with Path:** Searches for **“.log”** files and prints their paths.

```bash
find /path/to/search -name "*.log" -print
```

Interview Question:

- List some of the commonly used shell commands?
- Write a simple shell script to list all processes
- Write a script to print only errors from a remote log
- Write a shell script to print numbers divided by 3 & 5 and not 15
- Write a script to print number of "S" in Mississippi
- How will you debug the shell script?
- What is crontab in Linux? Can you provide an example of usage?
- How to open a read-only file?
- What is a difference between soft and hard link?
- What is a difference between break and continue statements?
- What are some disadvantages of Shell scripting?
- What a different types of loops and when to use?
- Is bash dynamic or statically typed and why?
- Explain about a network troubleshooting utility?
- How will you sort list on names in a file?
- How will you manage logs of a system that generate huge log files every day?