Name : **Arman** SAP ID : **52124567** Assignment : **Linux & Shell Scripting**

Q· What is Linux?

Just like Windows, iOS, and Mac OS, Linux is an operating system. In fact, one of the most popular platforms on the planet, Android, is powered by the Linux operating system. An operating system is software that manages all of the hardware resources associated with your desktop or laptop. To put it simply, the operating system manages the communication between your software and your hardware. Without the operating system (OS), the software wouldn't function.

Q · How Linux and UNIX are different and similar?

Features	Linux	Unix
Basic Definition	Linux is an open-source operating system. This OS is supported on several computer platforms and includes multiple software features that handle computer resources, and allow you to do tasks.	Unix is a powerful and multitasking operating system that behaves like a bridge between the user and the computer.
Launched by	This operating system was launched by Linus Torvalds at the University of Helsinki in 1991.	This operating system was launched in 1960 and released by AT&T Bell Labs.
OS family	It belongs to the Unix-like family.	It belongs to the Unix family.
Available in	It is available in multiple languages.	It is available in English.

Kernel Type	It is monolithic.	It can be microkernel, monolithic, and hybrid.
Written in	C and other programming languages.	C and assembly language.
File system support	It supports more file systems than Unix.	It also supports less than Linux.
Usage	It is used in several systems like desktop, smartphones, mainframes and servers.	Unix is majorly used on workstations and servers.
Examples	Some examples of Linux are: Fedora, Debian, Red Hat, Ubuntu, Android, etc.	Some examples of unix are IBM AIX, Darwin, Solaris, HP-UX, macOS X, etc.
Security	Linux provides higher security.	Unix is also highly secured.
Price	Linux is free and its corporate support is available at a price.	Unix is not totally free. There are some Unix versions that are free, other than that UNIX is expensive.

Q · Explain briefly about its history?

NIX development was started in 1969 at Bell Laboratories in New Jersey. Bell Laboratories was (1964–1968) involved on the development of a multi-user, time-sharing operating system called Multics (Multiplexed Information and Computing System). Multics was a failure. In early 1969, Bell Labs withdrew from the Multics project.

Bell Labs researchers who had worked on Multics (Ken Thompson, Dennis Ritchie, Douglas McIlroy, Joseph Ossanna, and others) still wanted to develop an operating system for their own and Bell Labs' programming, job control, and resource usage needs.

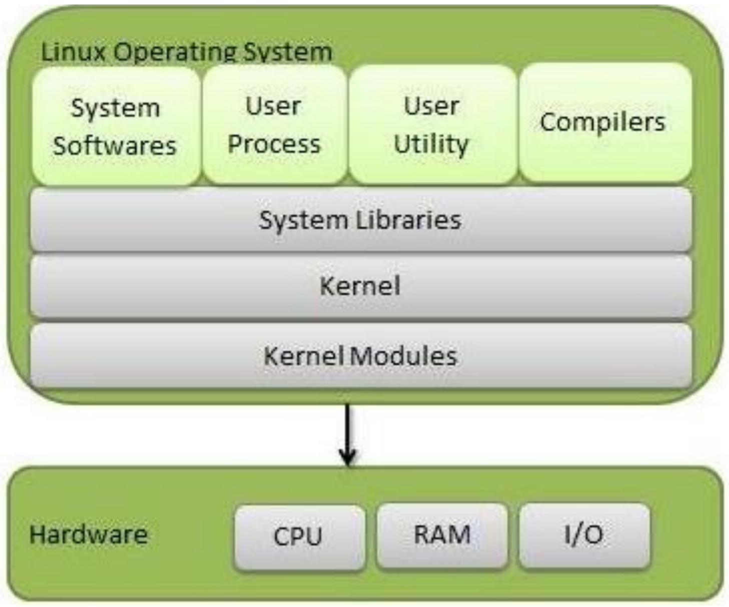
When Multics was withdrawn Ken Thompson and Dennis Ritchie needed to rewrite an operating system in order to play space travel on another smaller machine (a DEC PDP-7 [Programmed Data Processor 4K memory for user programs). The result was a system called UNICS (UNiplexed Information and Computing Service) which was an 'emasculated Multics'.

The first version of Unix was written in the low-level PDP-7 assembler language. Later, a language called TMG was developed for the PDP-7 by R. M. McClure. Using TMG to develop a FORTRAN compiler, Ken Thompson instead ended up developing a compiler for a new high-level language he called B, based on the earlier BCPL language developed by Martin Richard. When the PDP-11 computer arrived at Bell Labs, Dennis Ritchie built on B to create a new language called C. Unix components were later rewritten in C, and finally with the kernel itself in 1973.

Q. What are the basic components of Linux architecture

A computer's operating system interface to the hardware is referred to as a software application. A number of software applications are run on operating systems to manage hardware resources on a computer.

The diagram illustrates the structure of the Linux system, according to the layers concept.



The Linux architecture is largely composed of elements such as the Kernel, System Library, Hardware layer, System, and Shell functions.

Kernel: The kernel is one of the fundamental parts of an operating system. It is responsible for each of the primary duties of the Linux OS. Each of the major procedures of Linux is coordinated with hardware directly. The kernel is in charge of creating an appropriate abstraction for concealing trivial hardware or application strategies. The following kernel varieties are mentioned:

1. Monolithic Kernel
2. Micro kernels
3. Exo kernels
4. Hybrid kernels

System Libraries: A set of library functions may be specified as these functions. These functions are implemented by the operating system and do not require code access rights on the kernel modules.

System Utility Programs: A system utility program performs specific and individual jobs.

Hardware layer: The hardware layer of Linux is made up of several peripheral devices such as a CPU, HDD, and RAM.

Shell: Different operating systems are classified as graphical shells and command-line shells. A graphical shell is an interface between the kernel and the user. It provides kernel services, and it runs kernel operations. There are two types of graphical shells, which differ in appearance. These operating systems are divided into two categories, which are the graphical shells and command-line shells.

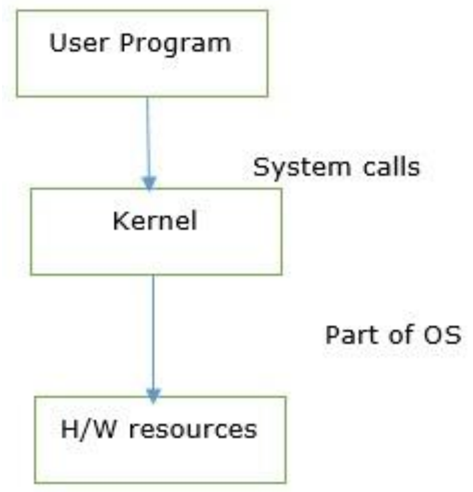
The graphical line shells allow for graphical user interfaces, while the command line shells enable for command line interfaces. As a result, both of these shells operate. However, graphical user interfaces performed using the graphical line shells are faster than those using the command line shells.

· Core of the Linux operating system is called?

Kernel. The core of the Linux system, the kernel handles network access, schedules processes or applications, manages basic peripheral devices and oversees all file system services. The Linux kernel is the software that interfaces directly with the computer hardware.

Q · Discuss the process management system calls in Linux

System call provides an interface between user program and operating system. The structure of system call is as follows –



When the user wants to give an instruction to the OS then it will do it through system calls. Or a user program can access the kernel which is a part of the OS through system calls.

It is a programmatic way in which a computer program requests a service from the kernel of the operating system.

Types of system calls

The different system calls are as follows –

- System calls for Process management
- System calls for File management
- System calls for Directory management

Now let us discuss process management system calls.

System calls for Process management

A system is used to create a new process or a duplicate process called a fork.

The duplicate process consists of all data in the file description and registers common. The original process is also called the parent process and the duplicate is called the child process.

The fork call returns a value, which is zero in the child and equal to the child's PID (Process Identifier) in the parent. The system calls like exit would request the services for terminating a process.

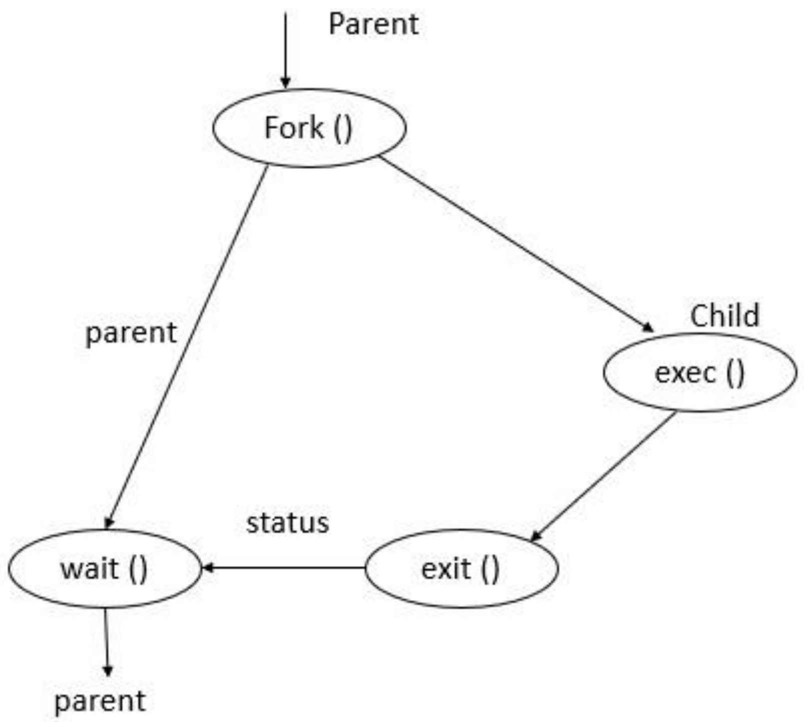
Loading of programs or changing of the original image with duplicate needs execution of exec. Pid would help to distinguish between child and parent processes.

Example

Process management system calls in Linux.

- **fork** − For creating a duplicate process from the parent process.
- **wait** − Processes are supposed to wait for other processes to complete their work.
- **exec** − Loads the selected program into the memory.
- **exit** − Terminates the process.

The pictorial representation of process management system calls is as follows −



fork() − A parent process always uses a fork for creating a new child process. The child process is generally called a copy of the parent. After execution of fork, both parent and child execute the same program in separate processes.

exec() − This function is used to replace the program executed by a process. The child sometimes may use exec after a fork for replacing the process memory space with a new program executable making the child execute a different program than the parent.

exit() − This function is used to terminate the process.

wait() − The parent uses a wait function to suspend execution till a child terminates. Using wait the parent can obtain the exit status of a terminated child.

Q · What is the use of LILO or GRUB

GNU GRand Unified Bootloader (GRUB) is a boot loader created by the ***GNU project***. GRUB allows the user to choose from a list of operating systems to load, allowing numerous operating systems to run on the same machine. GRUB is the default boot loader in most modern Linux distributions. GRUB may be customized dynamically since it permits changes to the configuration during boot. Users are given a simple command line interface through which they can dynamically input new boot configurations. GRUB provides several user-friendly characteristics, such as high portability, independence from geometry translation, support for many executable formats, and support for many file systems, including most UNIX systems, ***NTFS, VFAT,*** and ***LBA (Logical Block Address)*** mode. Most Linux distributions that use GRUB give a customized boot menu by leveraging its support for numerous GUIs (Graphical User Interfaces). GRUB2 is currently replacing GRUB, and GRUB has been renamed GRUB Legacy.

LILO is a Linux boot loader found in Linux-based devices that have been one of the most widely used and oldest boot loaders. The development of LILO went via many stages. The LILO was updated or changed by three developers. ***Werner Almesberger*** worked on LILO from ***1992 to 1995, John Coffman*** worked from ***1997 to 2007***, and ***Joachim Wiedorn*** has been with the project since **2000**. LILO has been simplified and made easier to use due to these three developers.

It only supports a single OS, which is Linux OS. It has been the default boot loader of Linux OS based devices for several years after gaining popularity from loading.

Compared to GRUB, it's an outdated boot loader and lacks a graphical user interface menu option.

Although GRUB is now present in most OS systems, LILO and ELILO are still incredibly popular in the modern days. LILO software has been good and dependable, keeping the propriety and operating system effectively..

Q · Explain the Linux directory structure.

The Linux directory structure is like a tree. The base of the Linux file system hierarchy begins at the root. Directories branch off the root, but everything starts at root.

The directory separator in Linux is the forward slash (/). When talking about directories and speaking directory paths, “forward slash” is abbreviated to “slash.” Often the root of the file system is referred to as “slash” since the full path to it is /. If you hear someone say “look in slash” or “that file is in slash,” they are referring to the root directory.

The /bin directory is where you will find binary or executable files. Programs are written in source code which is human readable text. Source code is then compiled into machine readable binaries. They are called binaries because machine code is a series of zeros and ones. The import thing to know is that commands, programs, and applications that you can use are sometimes located in /bin.

Configuration files live in the /etc directory. Configuration files control how the operating system or applications behave. For example, there is a file in /etc that tells the operating system whether to boot into a text mode or a graphical mode.

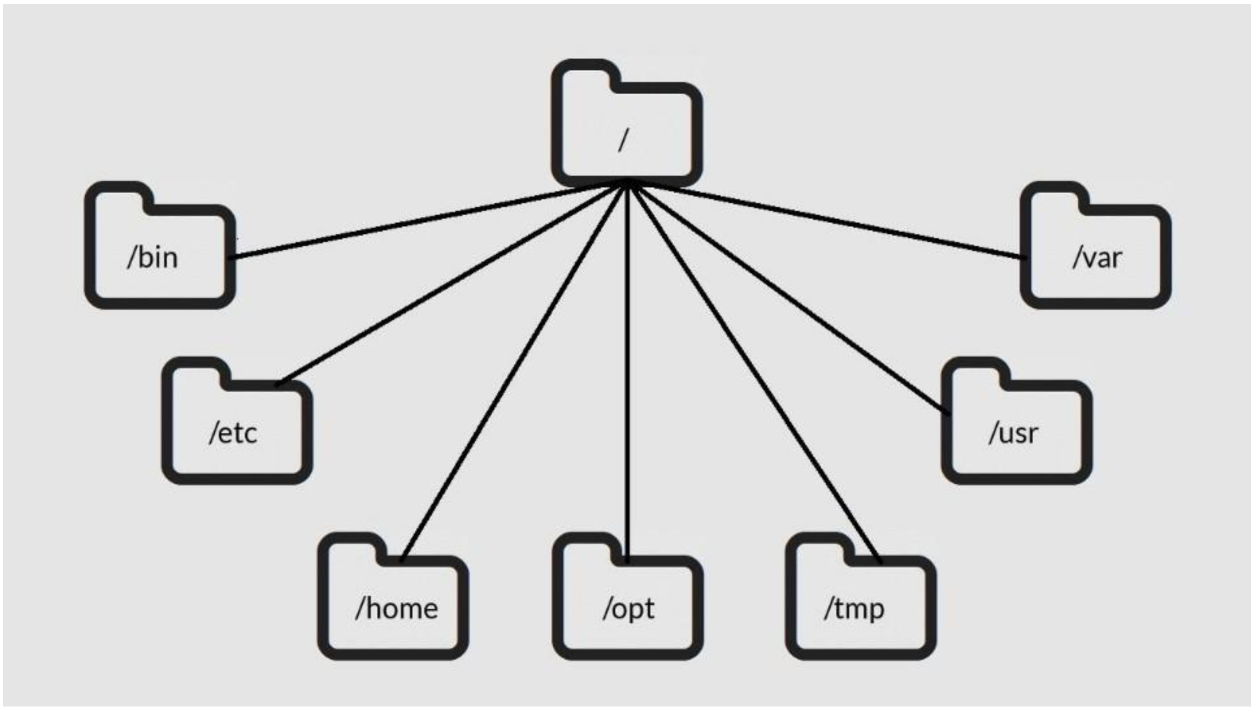
User home directories are located in /home. If your account name is “pat” your home directory will be /home/pat. Linux systems can and often do have multiple user accounts. Home directories allow each user to separate their data from the other users on the system. The pat directory is knows as a subdirectory. A subdirectory is simply a directory that resides inside another directory.

The /opt directory houses optional or third party software. Software that is not bundled with the operating system will often been installed in /opt. For example, the Google Earth application is not part of the standard Linux operating system and gets installed in the /opt/google/earth directory.

Temporary space is allocated in /tmp. Most Linux distributions clear the contents of /tmp at boot time. Be aware that if you put files in /tmp and the Linux system reboots, your files will more than likely be gone. The /tmp directory is a great place to store temporary files, but do not put anything in /tmp that you want to keep long term.

The /usr directory is called "user." You will find user related binary programs and executables in the /usr/bin directory.

Variable data such as log files reside in /var. Specifically, the /var/log directory contains logs generated by the operating system and other applications.



Q · Explain permissioning in Linux

File permissions are core to the security model used by Linux systems. They determine who can access files and directories on a system and how. This article provides an overview of Linux file permissions, how they work, and how to change them.

All the three owners (user owner, group, others) in the Linux system have three types of permissions defined. Nine characters denotes the three types of permissions.

1. **Read (r) :** The read permission allows you to open and read the content of a file. But you can't do any editing or modification in the file.

2. **Write (w) :** The write permission allows you to edit, remove or rename a file. For instance, if a file is present in a directory, and write permission is set on the file but not on the directory, then you can edit the content of the file but can't remove, or rename it.
3. **Execute (x):** In Unix type system, you can't run or execute a program unless execute permission is set.But in Windows, there is no such permission available.

Q · Explain the differences between cron and anacron

Cron

- Used to execute scheduled commands
- Assumes the system is continuously running.
- If system is not running during the time the jobs is scheduled, it will not run.
- Can schedule jobs down to the precise minute
- Universally available on all Linux systems
- Cron is a daemon

Anacron

- Used to execute commands periodically
- Suitable for systems that are often powered down when not in use (Laptops, workstations, etc..)
- Jobs will run if it hasn't been executed in the set amount of time.
- Minimum time frame is 1 day
- Anacron is not a daemon and relies on other methods to run

Q. Basics of GNU Utility Commands

Command	Usage
`ls`	Lists the content of a directory
`alias`	Define or display aliases
`unalias`	Remove `alias` definitions
`pwd`	Prints the working directory
`cd`	Changes directory
`cp`	Copies files and directories
`rm`	Remove files and directories
`mv`	Moves (renames) files and directories
`mkdir`	Creates directories
`man`	Displays manual page of other commands
`touch`	Creates empty files
`chmod`	Changes file permissions
`./`	Runs an executable
`exit`	Exits the current shell session

Command	Usage
sudo	Executes commands as superuser
shutdown	Shutdowns your machine
htop	Displays processes and resources information
unzip	Extracts compressed ZIP files
apt , yum , pacman	Package managers
echo	Displays lines of text
cat	Prints file contents
ps	Reports shell processes status
kill	Terminates programs
ping	Tests network connectivity
vim	Efficient text editing
history	Shows a list of previous commands
passwd	Changes user password
which	Returns the full binary path of a program
shred	Overwrites a file to hide its contents

Command	Usage
`less`	Inspects files interactively
`tail`	Displays last lines of a file
`head`	Displays first lines of a file
`grep`	Prints lines that match patterns
`whoami`	Outputs username
`whatis`	Shows single-line descriptions
`wc`	Word count files
`uname`	Displays OS information
`neofetch`	Displays OS and hardware information
`find`	Searches for files that follow a pattern
`wget`	Retrieves files from the internet

Q· Which Linux commands can be used to view directory content

The **ls** command displays all information in alphabetic order by file name. If the command is executed by a user with root authority, it uses the **-A** flag by default, listing all entries except dot (.) and dot dot (..). To show all entries for files, including those that begin with a dot (.), use the **ls -a** command.

Q· When to use the cd command.

CD (Change Directory). This command enables you to change the current directory or, in other words, to navigate to another folder from your PC.

Q· What is the use of cat command

Cat(concatenate) command is very frequently used in Linux. It reads data from the file and gives their content as output. It helps us to create, view, concatenate files. So let us see some frequently used cat commands.

1) To view a single file
Command:

```
$cat filename
```

Output

It will show content of given filename

2) To view multiple files
Command:

```
$cat file1 file2
```

Output

This will show the content of file1 and file2.

3) To view contents of a file preceding with line numbers.
Command:

```
$cat -n filename
```

Output

It will show content with line number

example:- cat-n mytext.txt

1)This is mytext

2)A unique array

4) Create a file
Command:

```
$ cat > newfile
```

Output

Will create a file named newfile

5) Copy the contents of one file to another file.
Command:

```
$cat [filename-whose-contents-is-to-be-copied] > [destination-filename]
```

Output

The content will be copied in destination file

6) Cat command can suppress repeated empty lines in output
Command:

```
$cat -s geeks.txt
```

Output

Will suppress repeated empty lines in output

7) Cat command can append the contents of one file to the end of another file.
Command:

```
$cat file1 >> file2
```

Output

Will append the contents of one file to the end of another file

8) Cat command can display content in reverse order using tac command.
Command:

```
 $tac filename
```

Output

Will display content in reverse order

9) Cat command can highlight the end of line.
Command:

```
$cat -E "filename"
```

Output

Will highlight the end of line

10) If you want to use the -v, -E and -T option together, then instead of writing -vET in the command, you can just use the -A command line option.
Command

```
$cat -A  "filename"
```

11) Cat command to open dashed files.
Command:

```
$cat -- "-dashfile"
```

Output

Will display the content of -dashfile

12) Cat command if the file has a lot of content and can’t fit in the terminal.
Command:

```
$cat "filename" | more
```

Output

Will show that much content, which could fit in terminal and will ask to show more.

13) Cat command to merge the contents of multiple files.
Command:

```
$cat "filename1" "filename2" "filename3" > "merged_filename"
```

Output

Will merge the contents of file in respective order and will insert that content in "merged_filename".

14) Cat command to display the content of all text files in the folder.
Command:

```
$cat *.txt
```

Output

Will show the content of all text files present in the folder.

15) Cat command to write in an already existing file.
Command :

```
$cat >> mytext.txt
```

The newly added text.

Output

Will append the text "The newly added text." to the end of the file.

Q· Difference between tail and head

he command line is a powerful tool for managing and manipulating files and directories in Linux. Two essential commands for working with text files are the “**head**” and “**tail**” **commands**. These commands allow users to display the beginning or end of a file, respectively. In this article, we will explore the usage and options of the head and tail commands in Linux.

head command

The **head** command is used to view the first few lines of a file. By default, it will display the first **10 lines** of a file, but this number can be changed with the '-n' option. The syntax for the **head** command is as follows −

```
$ head [options] [file(s)]
```

head command options

The head command has several options that can be used to customize its output. Some of the most used options are −

- **-n** − The -n option is used to specify the number of lines to display. For example, to view the first 20 lines of a file named "example.txt", the command would be:

```
$ head -n 20 example.txt
```

- **-q** − The -q option is used to suppress header printing when multiple files are used.
- **-v** − The -v option is used to always print headers when using multiple files.

Uses of head command

The **head** command can be used in various situations, such as −

Viewing the beginning of a large file

When working with large files, it can be useful to view the beginning of the file to get an idea of its contents without having to open it in an editor or viewer.

Comparing the beginning of multiple files

You can also use the main command to compare the beginning of multiple files.

```
$ head -n 20 file1.txt file2.txt
```

tail Command

The **tail** command is used to display the last few lines of a file. Like the head command, tail will display the last 10 lines of a file by default, but this number can be changed with the **-n** option. The syntax of the tail command is as follows −

```
$ tail [options] [file(s)]
```

tail command options

The **tail** command also has several options that can be used to customize its output. Some of the most used options are −

-n − The -n option is used to specify the number of lines to display. For example, to view the last 20 lines of a file named "example.txt", the command would be:

```
$ tail -n 20 example.txt
```

-f − The '-f' option is used to keep the file open and continue displaying output as the file grows. This option is useful when working with log files.

```
$ tail -f example.log
```

-F − The '-F' option is similar to the '-f' option, but it also controls file **truncation**.

Usage of tail command

The tail command can be used in various situations, such as −

See the end of a large file

When working with large files, it can be useful to view the end of the file to get an idea of its contents without having to open it in an editor or viewer.

Monitoring log files

The tail -f option is commonly used to monitor log files in real time. This is useful for troubleshooting and analyzing the behavior of a system or application.

Comparing the end of multiple files

You can also use the **tail** command to compare the ends of multiple files.

```
$ tail -n 20 file1.txt file2.txt
```

Advanced usage

The **head** and **tail** commands can also be combined with other command line utilities such as **grep**, **sed**, **awk**, etc. to perform more complex tasks, such as −

- Extracting a specific section of a file based on a pattern
- Extracting a specific column from a **CSV** file
- Extracting a specific line from a file

Q· Explain PIPE's' in Linux

A pipe is a form of redirection (transfer of standard output to some other destination) that is used in Linux and other Unix-like operating systems to send the output of one command/program/process to another command/program/process for further processing. The Unix/Linux systems allow stdout of a command to be connected to stdin of another command. You can make it do so by using the pipe character '|'. Pipe is used to combine two or more commands, and in this, the output of one command acts as input to another command, and this command's output may act as input to the next command and so on. It can also be visualized as a temporary connection between two or more commands/ programs/ processes. The command line programs that do the further processing are referred to as filters.

This direct connection between commands/ programs/ processes allows them to operate simultaneously and permits data to be transferred between them continuously rather than having to pass it through temporary text files or through the display screen.

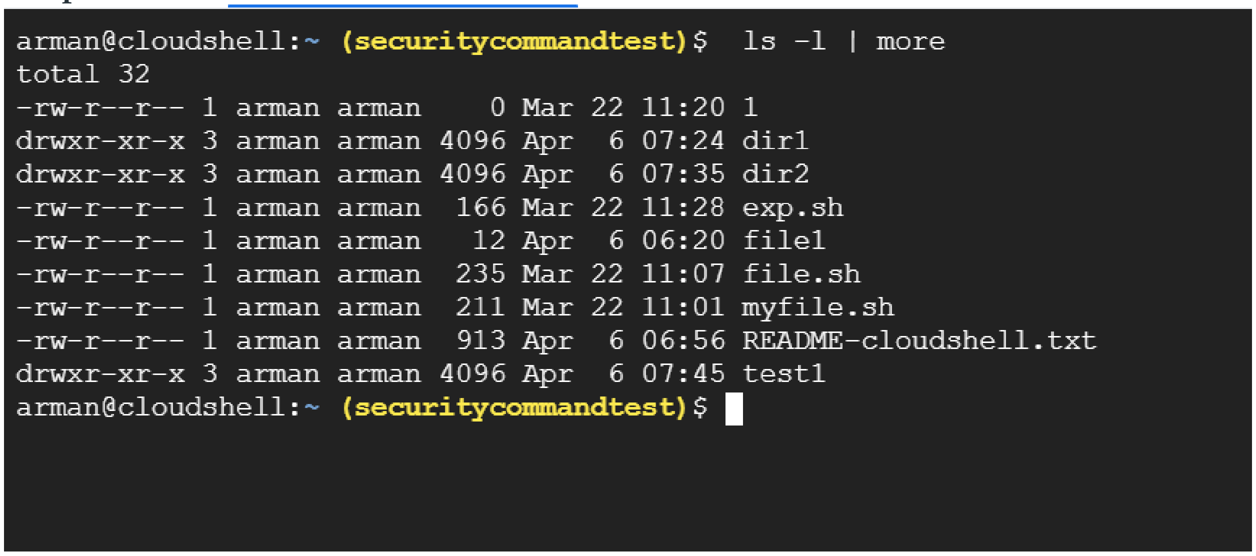
Syntax :

command_1 | command_2 | command_3 | .... | command_N

Example :
1. Listing all files and directories and give it as input to more command.

$ ls -l | more

Output :



```
arman@cloudshell:~ (securitycommandtest)$  ls -l | more
total 32
-rw-r--r-- 1 arman arman    0 Mar 22 11:20 1
drwxr-xr-x 3 arman arman 4096 Apr  6 07:24 dir1
drwxr-xr-x 3 arman arman 4096 Apr  6 07:35 dir2
-rw-r--r-- 1 arman arman  166 Mar 22 11:28 exp.sh
-rw-r--r-- 1 arman arman   12 Apr  6 06:20 file1
-rw-r--r-- 1 arman arman  235 Mar 22 11:07 file.sh
-rw-r--r-- 1 arman arman  211 Mar 22 11:01 myfile.sh
-rw-r--r-- 1 arman arman  913 Apr  6 06:56 README-cloudshell.txt
drwxr-xr-x 3 arman arman 4096 Apr  6 07:45 test1
arman@cloudshell:~ (securitycommandtest)$
```

· What does following folders contain

1. /etc 2. /var 3. /sbin 4. /usr 5. /opt

/etc : Host-specific system-wide configuration files.

- Contains configuration files required by all programs.
- This also contains startup and shutdown shell scripts used to start/stop individual programs.
- Example: /etc/resolv.conf, /etc/logrotate.conf.

/var contains variable data files. This includes spool directories and files, administrative and logging data, and transient and temporary files.

/sbin : Essential system binaries, e.g., fsck, init, route.

- Just like /bin, /sbin also contains binary executables.
- The linux commands located under this directory are used typically by system administrator, for system maintenance purpose.
- Example: iptables, reboot, fdisk, ifconfig, swapon

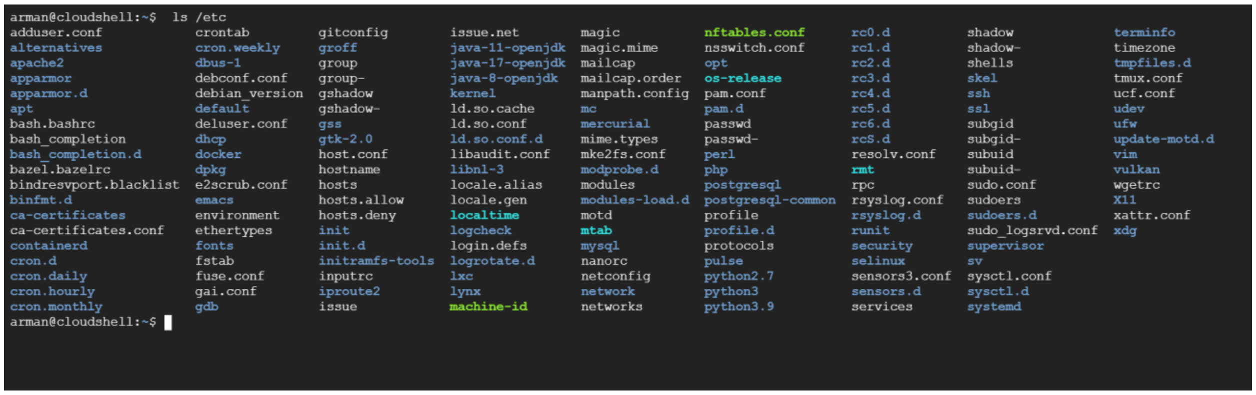
/usr : Secondary hierarchy for read-only user data; contains the majority of (multi-)user utilities and applications.

- Contains binaries, libraries, documentation, and source-code for second level programs.
- /usr/bin contains binary files for user programs. If you can't find a user binary under /bin, look under /usr/bin. For example: at, awk, cc, less, scp
- /usr/sbin contains binary files for system administrators. If you can't find a system binary under /sbin, look under /usr/sbin. For example: atd, cron, sshd, useradd, userdel
- /usr/lib contains libraries for /usr/bin and /usr/sbin
- /usr/local contains users programs that you install from source. For example, when you install apache from source, it goes under /usr/local/apache2
- /usr/src holds the Linux kernel sources, header-files and documentation

/opt : Optional application software packages.

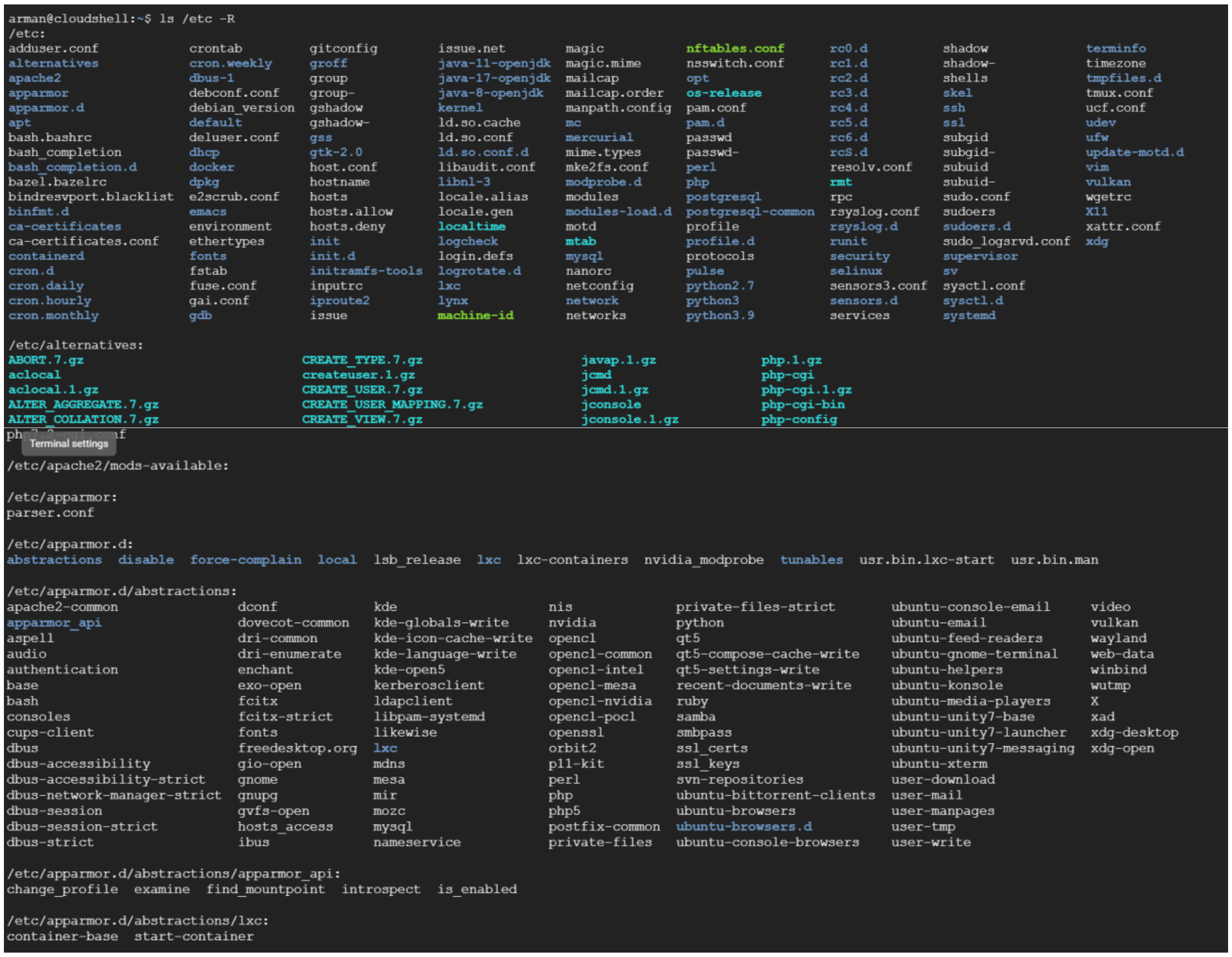
- Contains add-on applications from individual vendors.
- Add-on applications should be installed under either /opt/ or /opt/ sub-directory.

Q· List /etc folder using linux command



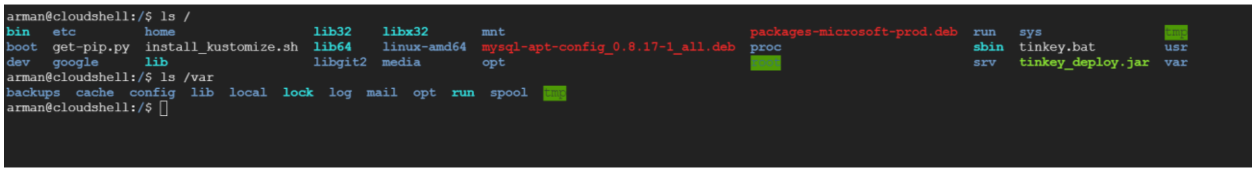
```
arman@cloudshell:~$  ls /etc
adduser.conf            crontab          gitconfig         issue.net        magic           nftables.conf      rc0.d          shadow             terminfo
alternatives            cron.weekly      groff             java-11-openjdk  magic.mime      nsswitch.conf      rc1.d          shadow-            timezone
apache2                 dbus-1           group             java-17-openjdk  mailcap         opt                rc2.d          shells             tmpfiles.d
apparmor                debconf.conf     group-            java-8-openjdk   mailcap.order   os-release         rc3.d          skel               tmux.conf
apparmor.d              debian_version   gshadow           kernel           manpath.config  pam.conf           rc4.d          ssh                ucf.conf
apt                     default          gshadow-          ld.so.cache      mc              pam.d              rc5.d          ssl                udev
bash.bashrc             deluser.conf     gss               ld.so.conf       mercurial       passwd             rc6.d          subgid             ufw
bash_completion         dhcp             gtk-2.0           ld.so.conf.d     mime.types      passwd-            rcS.d          subgid-            update-motd.d
bash_completion.d       docker           host.conf         libaudit.conf    mke2fs.conf     perl               resolv.conf    subuid             vim
bazel.bazelrc           dpkg             hostname          libnl-3          modprobe.d      php                rmt            subuid-            vulkan
bindresvport.blacklist  e2scrub.conf     hosts             locale.alias     modules         postgresql         rpc            sudo.conf          wgetrc
binfmt.d                emacs            hosts.allow       locale.gen       modules-load.d  postgresql-common  rsyslog.conf   sudoers            X11
ca-certificates         environment      hosts.deny        localtime        motd            profile            rsyslog.d      sudoers.d          xattr.conf
ca-certificates.conf    ethertypes       init              logcheck         mtab            profile.d          runit          sudo_logsrvd.conf  xdg
containerd              fonts            init.d            login.defs       mysql           protocols          security       supervisor
cron.d                  fstab            initramfs-tools   logrotate.d      nanorc          pulse              selinux        sv
cron.daily              fuse.conf        inputrc           lxc              netconfig       python2.7          sensors3.conf  sysctl.conf
cron.hourly             gai.conf         iproute2          lynx             network         python3            sensors.d      sysctl.d
cron.monthly            gdb              issue             machine-id       networks        python3.9          services       systemd
arman@cloudshell:~$ █
```

Q· List /etc folder recursive using linux command



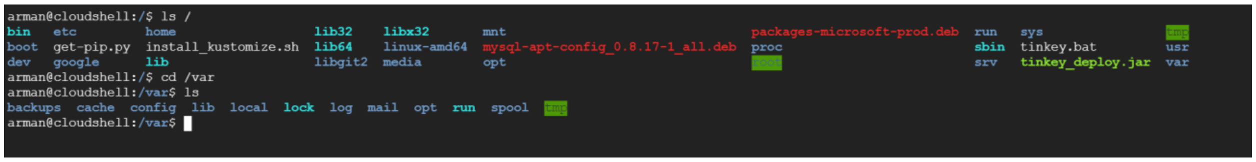
```
arman@cloudshell:~$ ls /etc -R
/etc:
adduser.conf             crontab          gitconfig        issue.net         magic            nftables.conf       rc0.d           shadow             terminfo
alternatives             cron.weekly      groff            java-11-openjdk   magic.mime       nsswitch.conf       rc1.d           shadow-            timezone
apache2                  dbus-1           group            java-17-openjdk   mailcap          opt                 rc2.d           shells             tmpfiles.d
apparmor                 debconf.conf     group-           java-8-openjdk    mailcap.order    os-release          rc3.d           skel               tmux.conf
apparmor.d               debian_version   gshadow          kernel            manpath.config   pam.conf            rc4.d           ssh                ucf.conf
apt                      default          gshadow-         ld.so.cache       mc               pam.d               rc5.d           ssl                udev
bash.bashrc              deluser.conf     gss              ld.so.conf        mercurial        passwd              rc6.d           subgid             ufw
bash_completion          dhcp             gtk-2.0          ld.so.conf.d      mime.types       passwd-             rcS.d           subgid-            update-motd.d
bash_completion.d        docker           host.conf        libaudit.conf     mke2fs.conf      perl                resolv.conf     subuid             vim
bazel.bazelrc            dpkg             hostname         libnl-3           modprobe.d       php                 rmt             subuid-            vulkan
bindresvport.blacklist   e2scrub.conf     hosts            locale.alias      modules          postgresql          rpc             sudo.conf          wgetrc
binfmt.d                 emacs            hosts.allow      locale.gen        modules-load.d   postgresql-common   rsyslog.conf    sudoers            X11
ca-certificates          environment      hosts.deny       localtime         motd             profile             rsyslog.d       sudoers.d          xattr.conf
ca-certificates.conf     ethertypes       init             logcheck          mtab             profile.d           runit           sudo_logsrvd.conf  xdg
containerd               fonts            init.d           login.defs        mysql            protocols           security        supervisor
cron.d                   fstab            initramfs-tools  logrotate.d       nanorc           pulse               selinux         sv
cron.daily               fuse.conf        inputrc          lxc               netconfig        python2.7           sensors3.conf   sysctl.conf
cron.hourly              gai.conf         iproute2         lynx              network          python3             sensors.d       sysctl.d
cron.monthly             gdb              issue            machine-id        networks         python3.9           services        systemd

/etc/alternatives:
ABORT.7.gz                              CREATE_TYPE.7.gz                     javap.1.gz             php.1.gz
aclocal                                 createuser.1.gz                      jcmd                   php-cgi
aclocal.1.gz                            CREATE_USER.7.gz                     jcmd.1.gz              php-cgi.1.gz
ALTER_AGGREGATE.7.gz                    CREATE_USER_MAPPING.7.gz             jconsole               php-cgi-bin
ALTER_COLLATION.7.gz                    CREATE_VIEW.7.gz                     jconsole.1.gz          php-config
php7.2-cgi.conf

/etc/apache2/mods-available:

/etc/apparmor:
parser.conf

/etc/apparmor.d:
abstractions  disable  force-complain  local  lsb_release  lxc  lxc-containers  nvidia_modprobe  tunables  usr.bin.lxc-start  usr.bin.man

/etc/apparmor.d/abstractions:
apache2-common                 dconf             kde                   nis              private-files-strict        ubuntu-console-email       video
apparmor_api                   dovecot-common    kde-globals-write     nvidia           python                      ubuntu-email               vulkan
aspell                         dri-common        kde-icon-cache-write  opencl           qt5                         ubuntu-feed-readers        wayland
audio                          dri-enumerate     kde-language-write    opencl-common    qt5-compose-cache-write     ubuntu-gnome-terminal      web-data
authentication                 enchant           kde-open5             opencl-intel     qt5-settings-write          ubuntu-helpers             winbind
base                           exo-open          kerberosclient        opencl-mesa      recent-documents-write      ubuntu-konsole             wutmp
bash                           fcitx             ldapclient            opencl-nvidia    ruby                        ubuntu-media-players       X
consoles                       fcitx-strict      libpam-systemd        opencl-pocl      samba                       ubuntu-unity7-base         xad
cups-client                    fonts             likewise              openssl          smbpass                     ubuntu-unity7-launcher     xdg-desktop
dbus                           freedesktop.org   lxc                   orbit2           ssl_certs                   ubuntu-unity7-messaging    xdg-open
dbus-accessibility             gio-open          mdns                  p11-kit          ssl_keys                    ubuntu-xterm
dbus-accessibility-strict      gnome             mesa                  perl             svn-repositories            user-download
dbus-network-manager-strict    gnupg             mir                   php              ubuntu-bittorrent-clients   user-mail
dbus-session                   gvfs-open         mozc                  php5             ubuntu-browsers             user-manpages
dbus-session-strict            hosts_access      mysql                 postfix-common   ubuntu-browsers.d           user-tmp
dbus-strict                    ibus              nameservice           private-files    ubuntu-console-browsers     user-write

/etc/apparmor.d/abstractions/apparmor_api:
change_profile  examine  find_mountpoint  introspect  is_enabled

/etc/apparmor.d/abstractions/lxc:
container-base  start-container
```

Q· List all the folders in /var using linux command



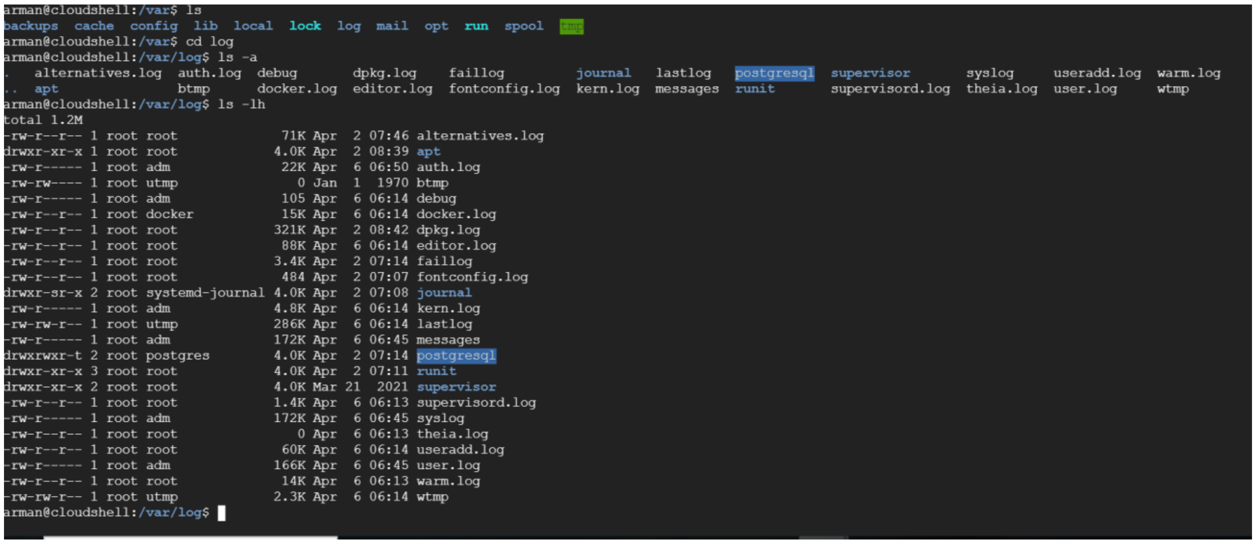
```
arman@cloudshell:/$ ls /
bin   etc         home                  lib32    libx32        mnt                                packages-microsoft-prod.deb  run   sys                tmp
boot  get-pip.py  install_kustomize.sh  lib64    linux-amd64   mysql-apt-config_0.8.17-1_all.deb  proc                         sbin  tinkey.bat         usr
dev   google      lib                   libgit2  media         opt                                root                         srv   tinkey_deploy.jar  var
arman@cloudshell:/$ ls /var
backups  cache  config  lib  local  lock  log  mail  opt  run  spool  tmp
arman@cloudshell:/$ 
```

OR



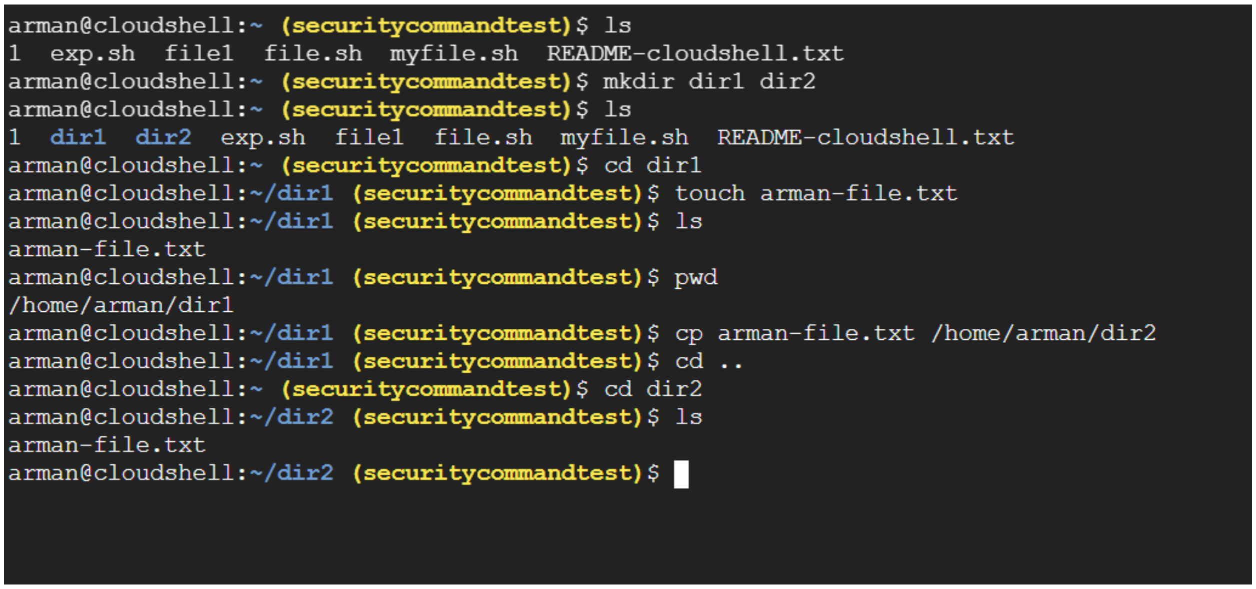
```
arman@cloudshell:/$ ls /
bin   etc         home                  lib32    libx32        mnt                                packages-microsoft-prod.deb  run   sys                tmp
boot  get-pip.py  install_kustomize.sh  lib64    linux-amd64   mysql-apt-config_0.8.17-1_all.deb  proc                         sbin  tinkey.bat         usr
dev   google      lib                   libgit2  media         opt                                root                         srv   tinkey_deploy.jar  var
arman@cloudshell:/$ cd /var
arman@cloudshell:/var$ ls
backups  cache  config  lib  local  lock  log  mail  opt  run  spool  tmp
arman@cloudshell:/var$ 
```

Q· List only files in /var/log folder using linux command



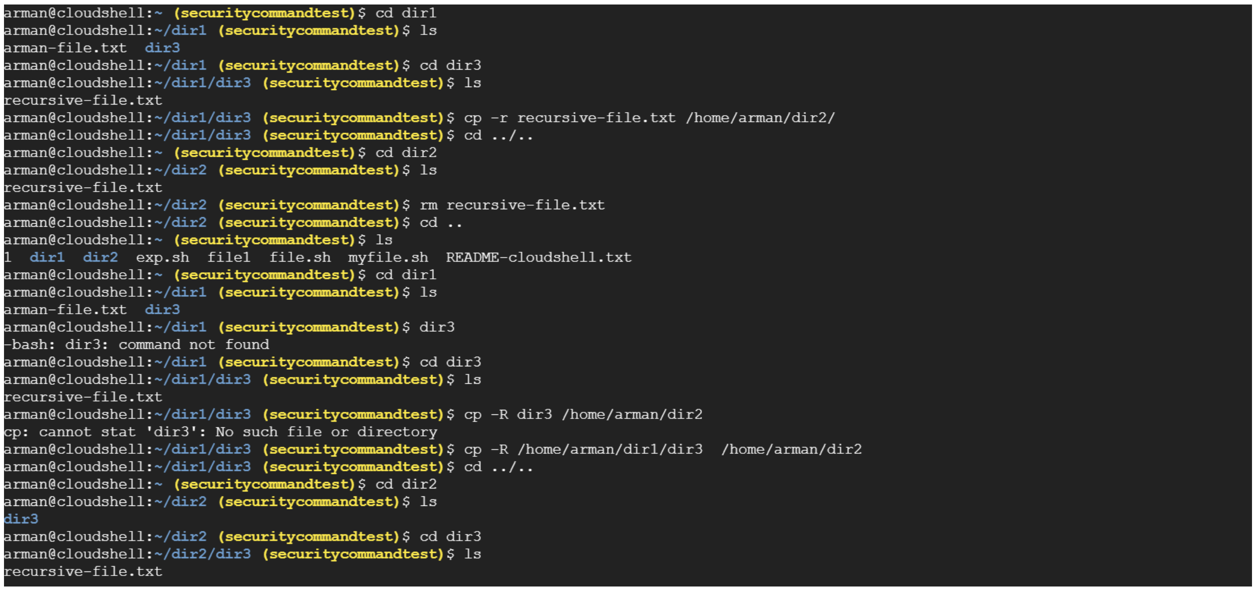
```
arman@cloudshell:/var$ ls
backups  cache  config  lib  local  lock  log  mail  opt  run  spool  tmp
arman@cloudshell:/var$ cd log
arman@cloudshell:/var/log$ ls -a
.   alternatives.log  auth.log  debug       dpkg.log    faillog         journal   lastlog   postgresql  supervisor       syslog     useradd.log  warm.log
..  apt               btmp      docker.log  editor.log  fontconfig.log  kern.log  messages  runit       supervisord.log  theia.log  user.log     wtmp
arman@cloudshell:/var/log$ ls -lh
total 1.2M
-rw-r--r-- 1 root root            71K Apr  2 07:46 alternatives.log
drwxr-xr-x 1 root root           4.0K Apr  2 08:39 apt
-rw-r----- 1 root adm             22K Apr  6 06:50 auth.log
-rw-rw---- 1 root utmp              0 Jan  1  1970 btmp
-rw-r----- 1 root adm             105 Apr  6 06:14 debug
-rw-r--r-- 1 root docker          15K Apr  6 06:14 docker.log
-rw-r--r-- 1 root root           321K Apr  2 08:42 dpkg.log
-rw-r--r-- 1 root root            88K Apr  6 06:14 editor.log
-rw-r--r-- 1 root root           3.4K Apr  2 07:14 faillog
-rw-r--r-- 1 root root            484 Apr  2 07:07 fontconfig.log
drwxr-sr-x 2 root systemd-journal 4.0K Apr  2 07:08 journal
-rw-r----- 1 root adm            4.8K Apr  6 06:14 kern.log
-rw-rw-r-- 1 root utmp           286K Apr  6 06:14 lastlog
-rw-r----- 1 root adm            172K Apr  6 06:45 messages
drwxrwxr-t 2 root postgres       4.0K Apr  2 07:14 postgresql
drwxr-xr-x 3 root root           4.0K Apr  2 07:11 runit
drwxr-xr-x 2 root root           4.0K Mar 21  2021 supervisor
-rw-r--r-- 1 root root           1.4K Apr  6 06:13 supervisord.log
-rw-r----- 1 root adm            172K Apr  6 06:45 syslog
-rw-r--r-- 1 root root              0 Apr  6 06:13 theia.log
-rw-r--r-- 1 root root            60K Apr  6 06:14 useradd.log
-rw-r----- 1 root adm            166K Apr  6 06:45 user.log
-rw-r--r-- 1 root root            14K Apr  6 06:13 warm.log
-rw-rw-r-- 1 root utmp           2.3K Apr  6 06:14 wtmp
arman@cloudshell:/var/log$
```

Q· Copy files from one folder to another using linux command



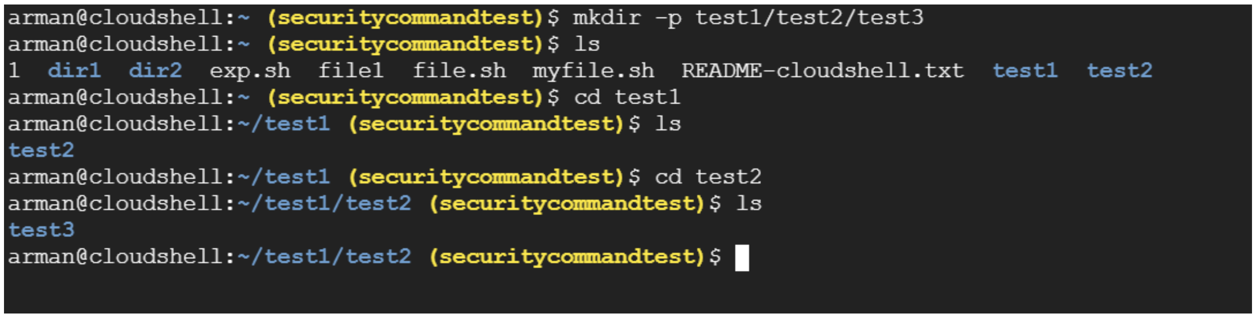
```
arman@cloudshell:~ (securitycommandtest)$ ls
1  exp.sh  file1  file.sh  myfile.sh  README-cloudshell.txt
arman@cloudshell:~ (securitycommandtest)$ mkdir dir1 dir2
arman@cloudshell:~ (securitycommandtest)$ ls
1  dir1  dir2  exp.sh  file1  file.sh  myfile.sh  README-cloudshell.txt
arman@cloudshell:~ (securitycommandtest)$ cd dir1
arman@cloudshell:~/dir1 (securitycommandtest)$ touch arman-file.txt
arman@cloudshell:~/dir1 (securitycommandtest)$ ls
arman-file.txt
arman@cloudshell:~/dir1 (securitycommandtest)$ pwd
/home/arman/dir1
arman@cloudshell:~/dir1 (securitycommandtest)$ cp arman-file.txt /home/arman/dir2
arman@cloudshell:~/dir1 (securitycommandtest)$ cd ..
arman@cloudshell:~ (securitycommandtest)$ cd dir2
arman@cloudshell:~/dir2 (securitycommandtest)$ ls
arman-file.txt
arman@cloudshell:~/dir2 (securitycommandtest)$
```

Q· Copy recursively one folder to another folder.



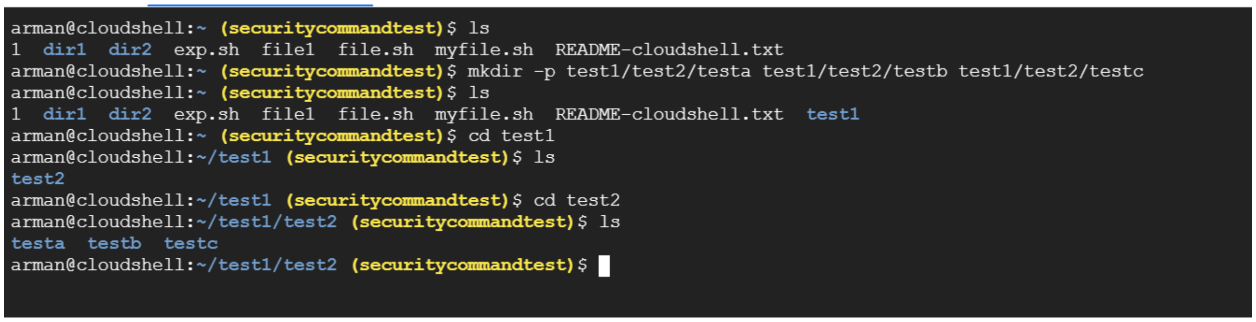
```
arman@cloudshell:~ (securitycommandtest)$ cd dir1
arman@cloudshell:~/dir1 (securitycommandtest)$ ls
arman-file.txt  dir3
arman@cloudshell:~/dir1 (securitycommandtest)$ cd dir3
arman@cloudshell:~/dir1/dir3 (securitycommandtest)$ ls
recursive-file.txt
arman@cloudshell:~/dir1/dir3 (securitycommandtest)$ cp -r recursive-file.txt /home/arman/dir2/
arman@cloudshell:~/dir1/dir3 (securitycommandtest)$ cd ../..
arman@cloudshell:~ (securitycommandtest)$ cd dir2
arman@cloudshell:~/dir2 (securitycommandtest)$ ls
recursive-file.txt
arman@cloudshell:~/dir2 (securitycommandtest)$ rm recursive-file.txt
arman@cloudshell:~/dir2 (securitycommandtest)$ cd ..
arman@cloudshell:~ (securitycommandtest)$ ls
1  dir1  dir2  exp.sh  file1  file.sh  myfile.sh  README-cloudshell.txt
arman@cloudshell:~ (securitycommandtest)$ cd dir1
arman@cloudshell:~/dir1 (securitycommandtest)$ ls
arman-file.txt  dir3
arman@cloudshell:~/dir1 (securitycommandtest)$ dir3
-bash: dir3: command not found
arman@cloudshell:~/dir1 (securitycommandtest)$ cd dir3
arman@cloudshell:~/dir1/dir3 (securitycommandtest)$ ls
recursive-file.txt
arman@cloudshell:~/dir1/dir3 (securitycommandtest)$ cp -R dir3 /home/arman/dir2
cp: cannot stat 'dir3': No such file or directory
arman@cloudshell:~/dir1/dir3 (securitycommandtest)$ cp -R /home/arman/dir1/dir3  /home/arman/dir2
arman@cloudshell:~/dir1/dir3 (securitycommandtest)$ cd ../..
arman@cloudshell:~ (securitycommandtest)$ cd dir2
arman@cloudshell:~/dir2 (securitycommandtest)$ ls
dir3
arman@cloudshell:~/dir2 (securitycommandtest)$ cd dir3
arman@cloudshell:~/dir2/dir3 (securitycommandtest)$ ls
recursive-file.txt
```

Q· Create folder test1/test2/test3 using mkdir command



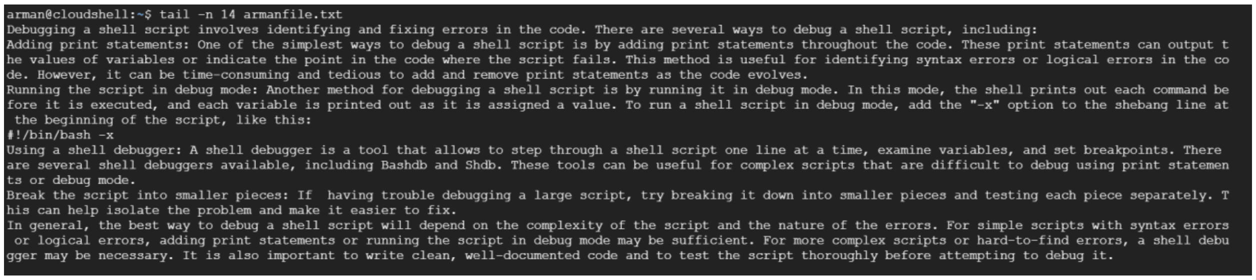
```
arman@cloudshell:~ (securitycommandtest)$ mkdir -p test1/test2/test3
arman@cloudshell:~ (securitycommandtest)$ ls
1  dir1  dir2  exp.sh  file1  file.sh  myfile.sh  README-cloudshell.txt  test1  test2
arman@cloudshell:~ (securitycommandtest)$ cd test1
arman@cloudshell:~/test1 (securitycommandtest)$ ls
test2
arman@cloudshell:~/test1 (securitycommandtest)$ cd test2
arman@cloudshell:~/test1/test2 (securitycommandtest)$ ls
test3
arman@cloudshell:~/test1/test2 (securitycommandtest)$
```

Q. Create folders test1/test2/testa, test1/test2/testb and test1/test2/testc using mkdir command



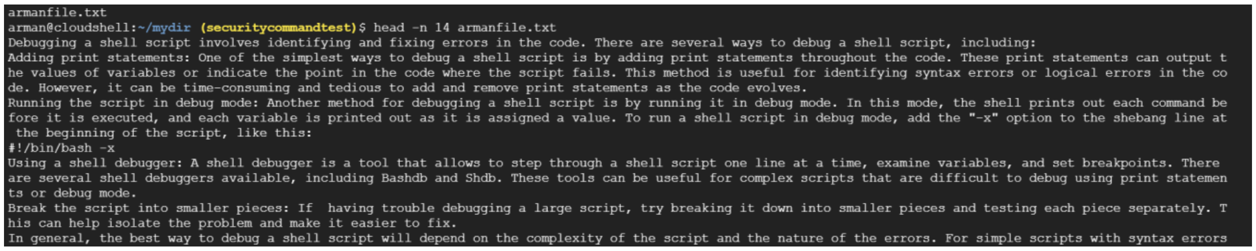
```
arman@cloudshell:~ (securitycommandtest)$ ls
1  dir1  dir2  exp.sh  file1  file.sh  myfile.sh  README-cloudshell.txt
arman@cloudshell:~ (securitycommandtest)$ mkdir -p test1/test2/testa test1/test2/testb test1/test2/testc
arman@cloudshell:~ (securitycommandtest)$ ls
1  dir1  dir2  exp.sh  file1  file.sh  myfile.sh  README-cloudshell.txt  test1
arman@cloudshell:~ (securitycommandtest)$ cd test1
arman@cloudshell:~/test1 (securitycommandtest)$ ls
test2
arman@cloudshell:~/test1 (securitycommandtest)$ cd test2
arman@cloudshell:~/test1/test2 (securitycommandtest)$ ls
testa  testb  testc
arman@cloudshell:~/test1/test2 (securitycommandtest)$
```

Q. View last 14 lines of a text file



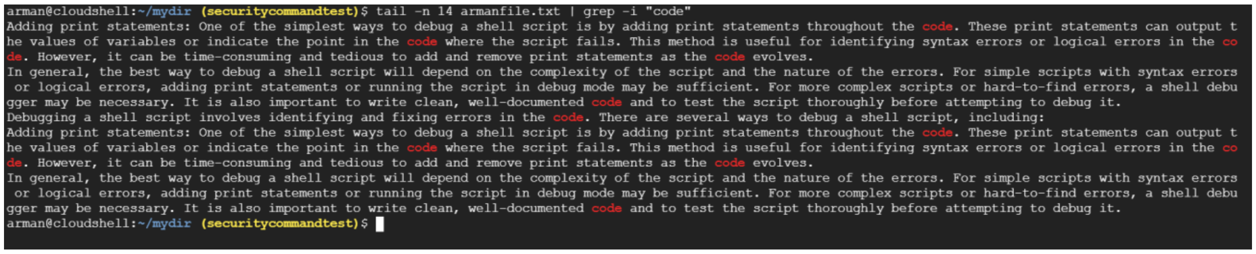
```
arman@cloudshell:~$ tail -n 14 armanfile.txt
Debugging a shell script involves identifying and fixing errors in the code. There are several ways to debug a shell script, including:
Adding print statements: One of the simplest ways to debug a shell script is by adding print statements throughout the code. These print statements can output t
he values of variables or indicate the point in the code where the script fails. This method is useful for identifying syntax errors or logical errors in the co
de. However, it can be time-consuming and tedious to add and remove print statements as the code evolves.
Running the script in debug mode: Another method for debugging a shell script is by running it in debug mode. In this mode, the shell prints out each command be
fore it is executed, and each variable is printed out as it is assigned a value. To run a shell script in debug mode, add the "-x" option to the shebang line at
 the beginning of the script, like this:
#!/bin/bash -x
Using a shell debugger: A shell debugger is a tool that allows to step through a shell script one line at a time, examine variables, and set breakpoints. There
are several shell debuggers available, including Bashdb and Shdb. These tools can be useful for complex scripts that are difficult to debug using print statemen
ts or debug mode.
Break the script into smaller pieces: If  having trouble debugging a large script, try breaking it down into smaller pieces and testing each piece separately. T
his can help isolate the problem and make it easier to fix.
In general, the best way to debug a shell script will depend on the complexity of the script and the nature of the errors. For simple scripts with syntax errors
 or logical errors, adding print statements or running the script in debug mode may be sufficient. For more complex scripts or hard-to-find errors, a shell debu
gger may be necessary. It is also important to write clean, well-documented code and to test the script thoroughly before attempting to debug it.
```

Q. View first 14 lines of a text file



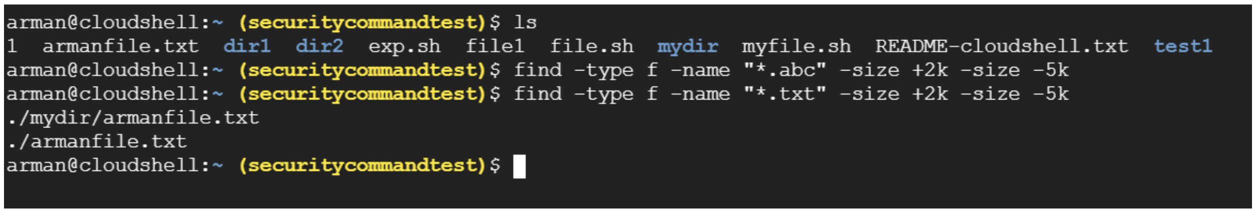
```
armanfile.txt
arman@cloudshell:~/mydir (securitycommandtest)$ head -n 14 armanfile.txt
Debugging a shell script involves identifying and fixing errors in the code. There are several ways to debug a shell script, including:
Adding print statements: One of the simplest ways to debug a shell script is by adding print statements throughout the code. These print statements can output t
he values of variables or indicate the point in the code where the script fails. This method is useful for identifying syntax errors or logical errors in the co
de. However, it can be time-consuming and tedious to add and remove print statements as the code evolves.
Running the script in debug mode: Another method for debugging a shell script is by running it in debug mode. In this mode, the shell prints out each command be
fore it is executed, and each variable is printed out as it is assigned a value. To run a shell script in debug mode, add the "-x" option to the shebang line at
 the beginning of the script, like this:
#!/bin/bash -x
Using a shell debugger: A shell debugger is a tool that allows to step through a shell script one line at a time, examine variables, and set breakpoints. There
are several shell debuggers available, including Bashdb and Shdb. These tools can be useful for complex scripts that are difficult to debug using print statemen
ts or debug mode.
Break the script into smaller pieces: If  having trouble debugging a large script, try breaking it down into smaller pieces and testing each piece separately. T
his can help isolate the problem and make it easier to fix.
In general, the best way to debug a shell script will depend on the complexity of the script and the nature of the errors. For simple scripts with syntax errors
```

Q. View last 14 lines of a text file and search for a word in it.



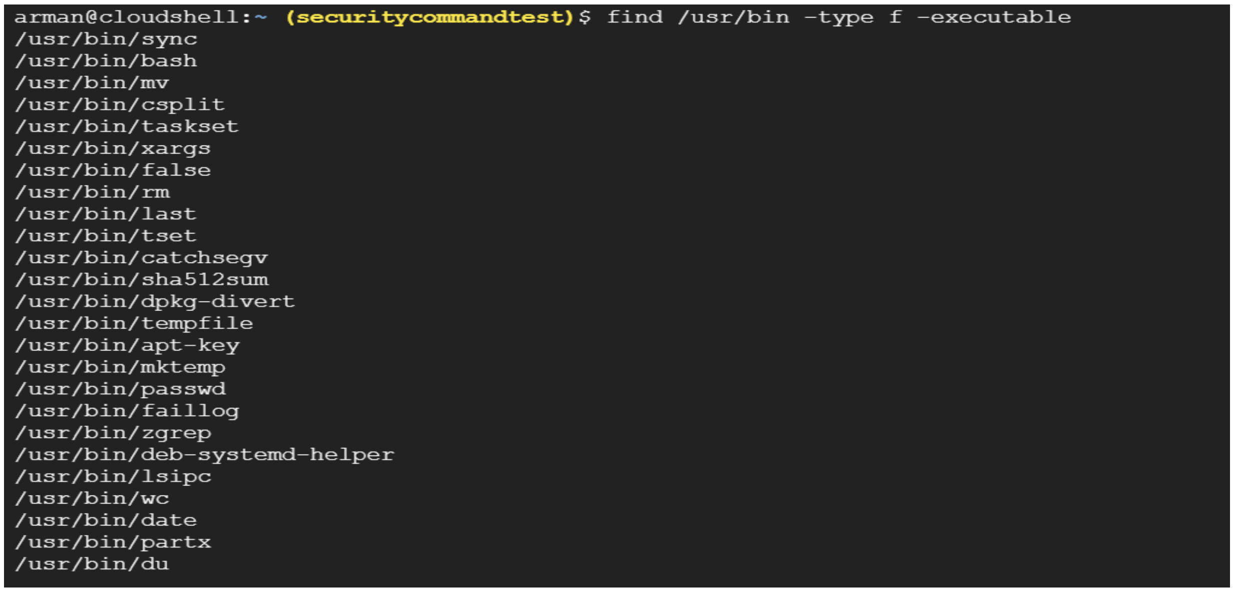
```
arman@cloudshell:~/mydir (securitycommandtest)$ tail -n 14 armanfile.txt | grep -i "code"
Adding print statements: One of the simplest ways to debug a shell script is by adding print statements throughout the code. These print statements can output t
he values of variables or indicate the point in the code where the script fails. This method is useful for identifying syntax errors or logical errors in the co
de. However, it can be time-consuming and tedious to add and remove print statements as the code evolves.
In general, the best way to debug a shell script will depend on the complexity of the script and the nature of the errors. For simple scripts with syntax errors
 or logical errors, adding print statements or running the script in debug mode may be sufficient. For more complex scripts or hard-to-find errors, a shell debu
gger may be necessary. It is also important to write clean, well-documented code and to test the script thoroughly before attempting to debug it.
Debugging a shell script involves identifying and fixing errors in the code. There are several ways to debug a shell script, including:
Adding print statements: One of the simplest ways to debug a shell script is by adding print statements throughout the code. These print statements can output t
he values of variables or indicate the point in the code where the script fails. This method is useful for identifying syntax errors or logical errors in the co
de. However, it can be time-consuming and tedious to add and remove print statements as the code evolves.
In general, the best way to debug a shell script will depend on the complexity of the script and the nature of the errors. For simple scripts with syntax errors
 or logical errors, adding print statements or running the script in debug mode may be sufficient. For more complex scripts or hard-to-find errors, a shell debu
gger may be necessary. It is also important to write clean, well-documented code and to test the script thoroughly before attempting to debug it.
arman@cloudshell:~/mydir (securitycommandtest)$
```

Q. Find files with abc extension who have size between 2k to 5k



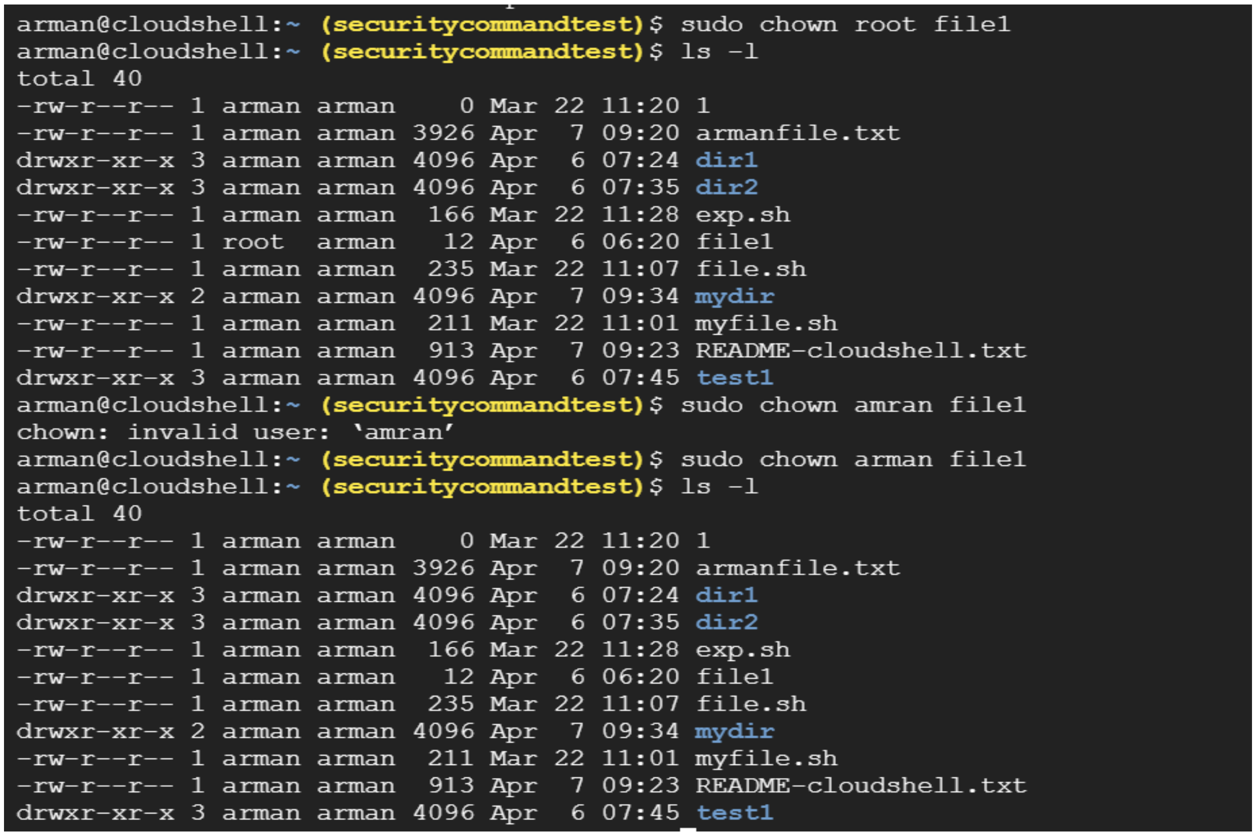
```
arman@cloudshell:~ (securitycommandtest)$ ls
1  armanfile.txt  dir1  dir2  exp.sh  file1  file.sh  mydir  myfile.sh  README-cloudshell.txt  test1
arman@cloudshell:~ (securitycommandtest)$ find -type f -name "*.abc" -size +2k -size -5k
arman@cloudshell:~ (securitycommandtest)$ find -type f -name "*.txt" -size +2k -size -5k
./mydir/armanfile.txt
./armanfile.txt
arman@cloudshell:~ (securitycommandtest)$
```

Q. Find all executable files in /usr folder



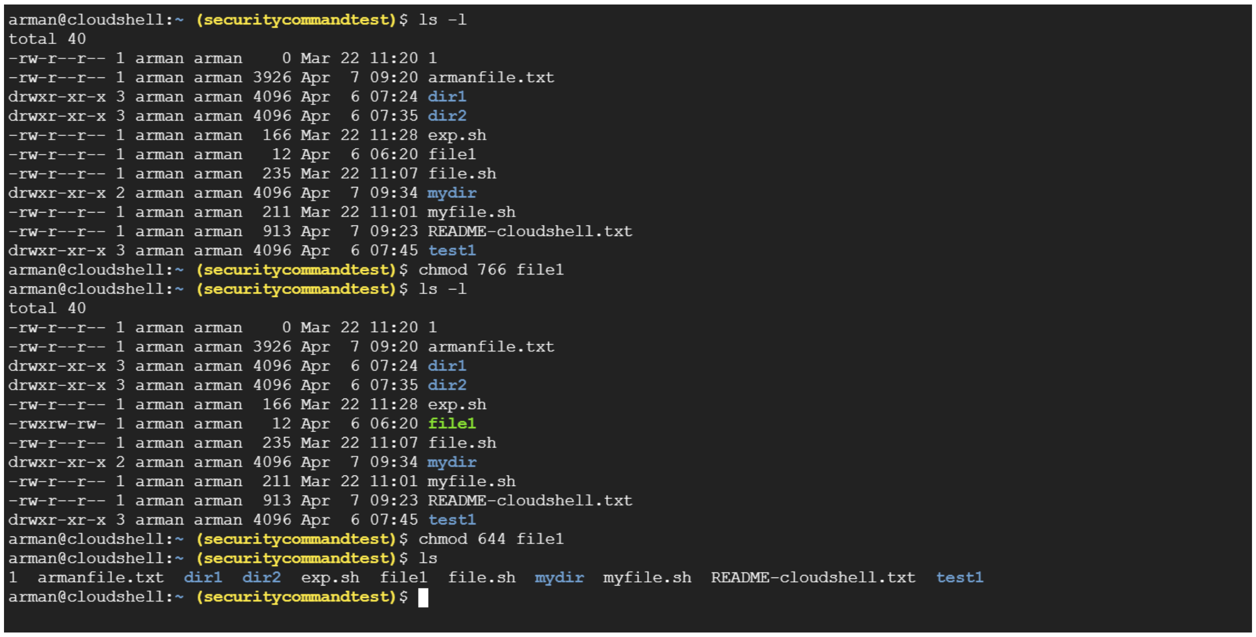
```
arman@cloudshell:~ (securitycommandtest)$ find /usr/bin -type f -executable
/usr/bin/sync
/usr/bin/bash
/usr/bin/mv
/usr/bin/csplit
/usr/bin/taskset
/usr/bin/xargs
/usr/bin/false
/usr/bin/rm
/usr/bin/last
/usr/bin/tset
/usr/bin/catchsegv
/usr/bin/sha512sum
/usr/bin/dpkg-divert
/usr/bin/tempfile
/usr/bin/apt-key
/usr/bin/mktemp
/usr/bin/passwd
/usr/bin/faillog
/usr/bin/zgrep
/usr/bin/deb-systemd-helper
/usr/bin/lsipc
/usr/bin/wc
/usr/bin/date
/usr/bin/partx
/usr/bin/du
```

Q. Change ownership of a file to root and back



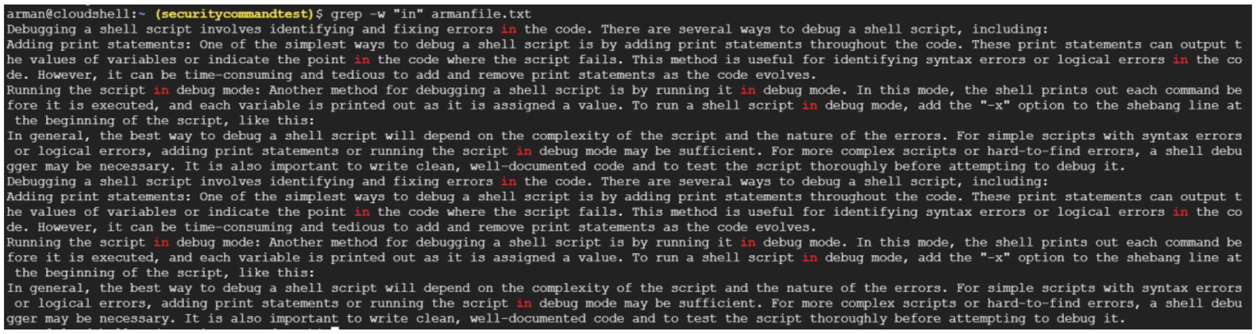
```
arman@cloudshell:~ (securitycommandtest)$ sudo chown root file1
arman@cloudshell:~ (securitycommandtest)$ ls -l
total 40
-rw-r--r-- 1 arman arman    0 Mar 22 11:20 1
-rw-r--r-- 1 arman arman 3926 Apr  7 09:20 armanfile.txt
drwxr-xr-x 3 arman arman 4096 Apr  6 07:24 dir1
drwxr-xr-x 3 arman arman 4096 Apr  6 07:35 dir2
-rw-r--r-- 1 arman arman  166 Mar 22 11:28 exp.sh
-rw-r--r-- 1 root  arman   12 Apr  6 06:20 file1
-rw-r--r-- 1 arman arman  235 Mar 22 11:07 file.sh
drwxr-xr-x 2 arman arman 4096 Apr  7 09:34 mydir
-rw-r--r-- 1 arman arman  211 Mar 22 11:01 myfile.sh
-rw-r--r-- 1 arman arman  913 Apr  7 09:23 README-cloudshell.txt
drwxr-xr-x 3 arman arman 4096 Apr  6 07:45 test1
arman@cloudshell:~ (securitycommandtest)$ sudo chown amran file1
chown: invalid user: 'amran'
arman@cloudshell:~ (securitycommandtest)$ sudo chown arman file1
arman@cloudshell:~ (securitycommandtest)$ ls -l
total 40
-rw-r--r-- 1 arman arman    0 Mar 22 11:20 1
-rw-r--r-- 1 arman arman 3926 Apr  7 09:20 armanfile.txt
drwxr-xr-x 3 arman arman 4096 Apr  6 07:24 dir1
drwxr-xr-x 3 arman arman 4096 Apr  6 07:35 dir2
-rw-r--r-- 1 arman arman  166 Mar 22 11:28 exp.sh
-rw-r--r-- 1 arman arman   12 Apr  6 06:20 file1
-rw-r--r-- 1 arman arman  235 Mar 22 11:07 file.sh
drwxr-xr-x 2 arman arman 4096 Apr  7 09:34 mydir
-rw-r--r-- 1 arman arman  211 Mar 22 11:01 myfile.sh
-rw-r--r-- 1 arman arman  913 Apr  7 09:23 README-cloudshell.txt
drwxr-xr-x 3 arman arman 4096 Apr  6 07:45 test1
```

Q. Change file permission to 766 and then change it back to original



```
arman@cloudshell:~ (securitycommandtest)$ ls -l
total 40
-rw-r--r-- 1 arman arman    0 Mar 22 11:20 1
-rw-r--r-- 1 arman arman 3926 Apr  7 09:20 armanfile.txt
drwxr-xr-x 3 arman arman 4096 Apr  6 07:24 dir1
drwxr-xr-x 3 arman arman 4096 Apr  6 07:35 dir2
-rw-r--r-- 1 arman arman  166 Mar 22 11:28 exp.sh
-rw-r--r-- 1 arman arman   12 Apr  6 06:20 file1
-rw-r--r-- 1 arman arman  235 Mar 22 11:07 file.sh
drwxr-xr-x 2 arman arman 4096 Apr  7 09:34 mydir
-rw-r--r-- 1 arman arman  211 Mar 22 11:01 myfile.sh
-rw-r--r-- 1 arman arman  913 Apr  7 09:23 README-cloudshell.txt
drwxr-xr-x 3 arman arman 4096 Apr  6 07:45 test1
arman@cloudshell:~ (securitycommandtest)$ chmod 766 file1
arman@cloudshell:~ (securitycommandtest)$ ls -l
total 40
-rw-r--r-- 1 arman arman    0 Mar 22 11:20 1
-rw-r--r-- 1 arman arman 3926 Apr  7 09:20 armanfile.txt
drwxr-xr-x 3 arman arman 4096 Apr  6 07:24 dir1
drwxr-xr-x 3 arman arman 4096 Apr  6 07:35 dir2
-rw-r--r-- 1 arman arman  166 Mar 22 11:28 exp.sh
-rwxrw-rw- 1 arman arman   12 Apr  6 06:20 file1
-rw-r--r-- 1 arman arman  235 Mar 22 11:07 file.sh
drwxr-xr-x 2 arman arman 4096 Apr  7 09:34 mydir
-rw-r--r-- 1 arman arman  211 Mar 22 11:01 myfile.sh
-rw-r--r-- 1 arman arman  913 Apr  7 09:23 README-cloudshell.txt
drwxr-xr-x 3 arman arman 4096 Apr  6 07:45 test1
arman@cloudshell:~ (securitycommandtest)$ chmod 644 file1
arman@cloudshell:~ (securitycommandtest)$ ls
1  armanfile.txt  dir1  dir2  exp.sh  file1  file.sh  mydir  myfile.sh  README-cloudshell.txt  test1
arman@cloudshell:~ (securitycommandtest)$
```

Q. Find all lines where "in" word is present in a file



```
arman@cloudshell:~ (securitycommandtest)$ grep -w "in" armanfile.txt
Debugging a shell script involves identifying and fixing errors in the code. There are several ways to debug a shell script, including:
Adding print statements: One of the simplest ways to debug a shell script is by adding print statements throughout the code. These print statements can output t
he values of variables or indicate the point in the code where the script fails. This method is useful for identifying syntax errors or logical errors in the co
de. However, it can be time-consuming and tedious to add and remove print statements as the code evolves.
Running the script in debug mode: Another method for debugging a shell script is by running it in debug mode. In this mode, the shell prints out each command be
fore it is executed, and each variable is printed out as it is assigned a value. To run a shell script in debug mode, add the "-x" option to the shebang line at
 the beginning of the script, like this:
In general, the best way to debug a shell script will depend on the complexity of the script and the nature of the errors. For simple scripts with syntax errors
 or logical errors, adding print statements or running the script in debug mode may be sufficient. For more complex scripts or hard-to-find errors, a shell debu
gger may be necessary. It is also important to write clean, well-documented code and to test the script thoroughly before attempting to debug it.
Debugging a shell script involves identifying and fixing errors in the code. There are several ways to debug a shell script, including:
Adding print statements: One of the simplest ways to debug a shell script is by adding print statements throughout the code. These print statements can output t
he values of variables or indicate the point in the code where the script fails. This method is useful for identifying syntax errors or logical errors in the co
de. However, it can be time-consuming and tedious to add and remove print statements as the code evolves.
Running the script in debug mode: Another method for debugging a shell script is by running it in debug mode. In this mode, the shell prints out each command be
fore it is executed, and each variable is printed out as it is assigned a value. To run a shell script in debug mode, add the "-x" option to the shebang line at
 the beginning of the script, like this:
In general, the best way to debug a shell script will depend on the complexity of the script and the nature of the errors. For simple scripts with syntax errors
 or logical errors, adding print statements or running the script in debug mode may be sufficient. For more complex scripts or hard-to-find errors, a shell debu
gger may be necessary. It is also important to write clean, well-documented code and to test the script thoroughly before attempting to debug it.
```

Q. Name different commands that are available to check the disk usage.

Two related commands that every system administrator runs frequently are **df** and **du**. While **du** reports files' and directories' disk usage, **df** reports how much disk space your file system is using. The **df** command displays the amount of disk space available on the file system with each file name's argument.

Q. Explain ways to create shortcuts in Linux.

There are two ways to create shortcut in Linux:

1. Hard Links: They are the low-level links. It links more than one filename with the same Inode and it represents the physical location of a file. When hard link is created for a file, it directly points to the Inode of the original file in the disk space, which means no new Inode is created. Directories are not created using hard links and they cannot cross file system boundaries. When the source file is removed or moved, then hard links are not affected.

2. Soft Links: They are very common. It represents a virtual or abstract location of the file. It is just like the shortcuts created in Windows. A soft link doesn't contain any information or content of the linked file, instead it has a pointer to the location of the linked file. In other words, a new file is created with new Inode, having a pointer to the Inode location of the original file. It is used to create link between directories and can cross file system boundaries. When the source file is removed or moved, then soft links are not updated.

Q. How to check whether a link is a hard one or a soft link?

For hard link, use the ls -i command to view the Inode number. Files that are hard-linked together share the same Inode number.

For soft link, use the ls –l command. The output will clearly indicate that the folder is a symbolic link and it will also list the folder where it points to.

Q. How Hard link and Soft Link are different?

Comparison Parameters	**Hard link**	**Soft link**
Inode number*	Files that are hard linked take the same inode number.	Files that are soft linked take a different inode number.
Directories	Hard links are not allowed for directories. (Only a superuser* can do it)	Soft links can be used for linking directories.
File system	It cannot be used across file systems.	It can be used across file systems.
Data	Data present in the original file will still be available in the hard links.	Soft links only point to the file name, it does not retain data of the file.
Original file's deletion	If the original file is removed, the link will still work as it accesses the data the original was having access to.	If the original file is removed, the link will not work as it doesn't access the original file's data.
Speed	Hard links are comparatively faster.	Soft links are comparatively slower.

Q. Can we identify hard link file?

Yes, by using the ls -i command to view the Inode number. Files that are hard-linked together share the same Inode number.

Q. What happens to hark linked file, when original file is deleted?

If the original file is deleted, the data still exists under the hard linked file.

Q. What happens to soft linked file, when original file is deleted?

If the original file is deleted, the soft link is broken.

Q. What are sed and awk commands used for?

awk command is used for pattern matching as well as for text processing. It searches one or more files to see if they contain lines that matches with the specified patterns and then perform the associated actions.

sed command is used for editing a 'stream' of text. It can read input from a text file or from piped input, and process the input in one pass.

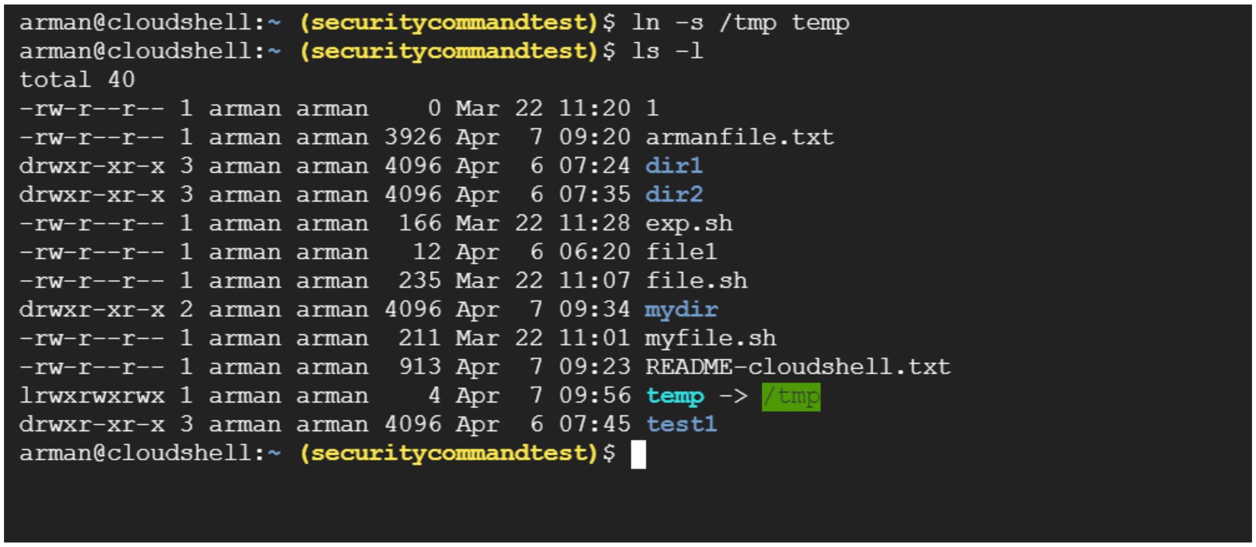
Q. How to use pipe command?

The vertical bar " | " between the two commands is called a pipe. It is used to pipe, or transfer, the standard output from the command on its left into the standard input of the command on its right.

Q. Name the alternative command for echo.

printf command

Q. Create a shortcut for /tmp folder.



```
arman@cloudshell:~ (securitycommandtest)$ ln -s /tmp temp
arman@cloudshell:~ (securitycommandtest)$ ls -l
total 40
-rw-r--r-- 1 arman arman    0 Mar 22 11:20 1
-rw-r--r-- 1 arman arman 3926 Apr  7 09:20 armanfile.txt
drwxr-xr-x 3 arman arman 4096 Apr  6 07:24 dir1
drwxr-xr-x 3 arman arman 4096 Apr  6 07:35 dir2
-rw-r--r-- 1 arman arman  166 Mar 22 11:28 exp.sh
-rw-r--r-- 1 arman arman   12 Apr  6 06:20 file1
-rw-r--r-- 1 arman arman  235 Mar 22 11:07 file.sh
drwxr-xr-x 2 arman arman 4096 Apr  7 09:34 mydir
-rw-r--r-- 1 arman arman  211 Mar 22 11:01 myfile.sh
-rw-r--r-- 1 arman arman  913 Apr  7 09:23 README-cloudshell.txt
lrwxrwxrwx 1 arman arman    4 Apr  7 09:56 temp -> /tmp
drwxr-xr-x 3 arman arman 4096 Apr  6 07:45 test1
arman@cloudshell:~ (securitycommandtest)$
```

Q. What does alias command do? Explain in details.

In Linux, an alias is a shortcut that references a command. An alias replaces a string that invokes a command in the Linux shell with another user-defined string.

Aliases are mostly used to replace long commands, improving efficiency and avoiding potential spelling errors. Aliases can also replace commands with additional options, making them easier to use.

The alias command uses the following syntax:

alias [option] [name]='[value]'

The different elements of the alias command syntax are:

- **alias**: Invokes the alias command.
- **[option]**: Allows the command to list all current aliases.
- **[name]**: Defines the new shortcut that references a command. A name is a user-defined string, excluding special characters and 'alias' and 'unalias', which cannot be used as names.

- **[value]**: Specifies the command the alias references. Commands can also include options, arguments, and variables. A value can also be a path to a script you want to execute.

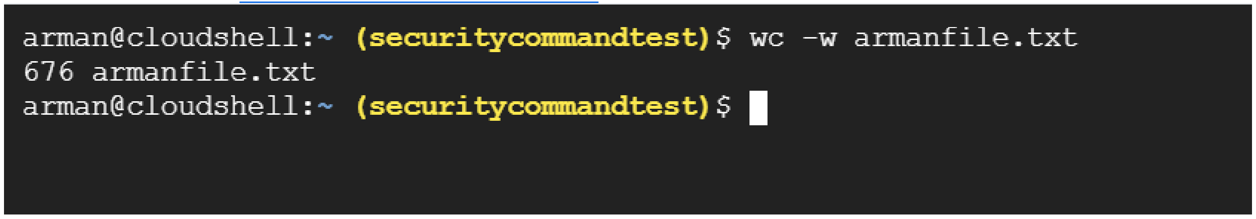
Q. Give the file's name where shell programmes are kept.

/etc folder

Q. What does . (dot) indicate at the beginning of the file name also explain its usecase?

A dot at the beginning of a filename hides the file in common file managers and for common shell programs. The ls command does not display them unless the -a or -A flags (ls -a or ls -A) are used.

Q. Write a command sequence to count the words in a given file.



```
arman@cloudshell:~ (securitycommandtest)$ wc -w armanfile.txt
676 armanfile.txt
arman@cloudshell:~ (securitycommandtest)$
```

Q. What command can you use to most effectively monitor a log file that is constantly updating?

tail –f command

Q. What are the four key elements of each Linux file system?

superblock, inode, data block, directory block, and indirection block.

Q. What is a kernel?

The kernel is the main component of a Linux operating system (OS) and is the core interface between a computer's hardware and its processes. It communicates between the 2, managing resources as efficiently as possible.

The kernel is so named because like a seed inside a hard shell it exists within the OS and controls all the major functions of the hardware, whether it's a phone, laptop, server, or any other kind of computer.

Q. How do you use functions in a shell script?

A function is a block of code that carries out a certain activity. A function can be called and reused. In shell scripting, functions are analogous to other programming languages' subroutines, procedures, and functions.

Syntax:

```
function_name () {
  list of commands
}
```

Our function's name is "function_name", and we'll call it by that name throughout our scripts. The name of the function must be preceded by parentheses and a series of instructions surrounded by braces.

Q. How do you use variables in a shell script?

We can define variables by running the command variable_name="value " and access it using $ sign before it.

For example:

```
NAME="Arman Malik"
echo $NAME
```

The above script will produce the following value:

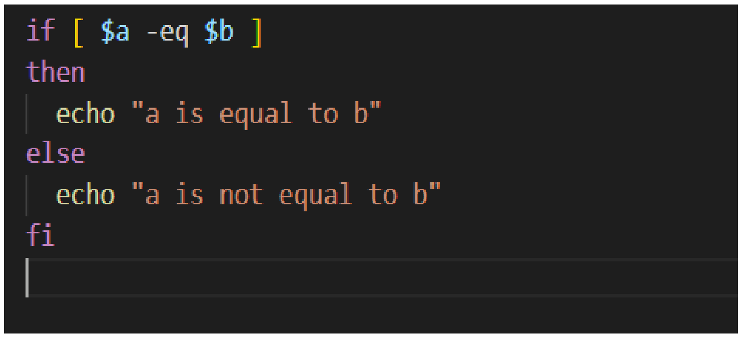
```
Arman Malik
```

Basics of GNU Utility Commands-

Q. What are some common shell script constructs, such as loops and conditional statements, and how do you use them?

A. There are several common shell script constructs in Linux, including loops and conditional statements. Here's a brief overview of some of these constructs and how they are used:

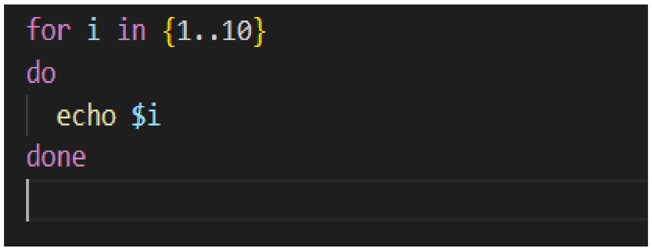
Conditional statements: Conditional statements are used to test whether a certain condition is true or false, and then execute different commands depending on the result. The most commonly used conditional statements in shell scripting are "if" and "case". Here's an example of an "if" statement in a shell script:



```
if [ $a -eq $b ]
then
  echo "a is equal to b"
else
  echo "a is not equal to b"
fi
```

In this example, the script checks whether the variables "a" and "b" are equal, and then prints out a message depending on the result.

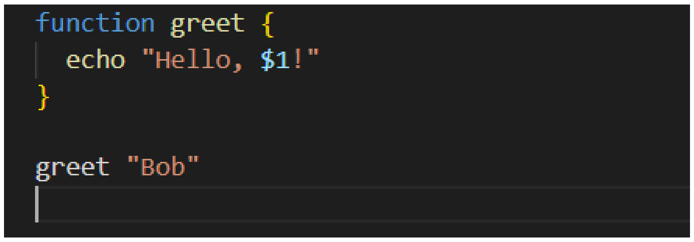
Loops: Loops are used to execute a set of commands repeatedly, either a specific number of times or until a certain condition is met. The most used loop statements in shell scripting are "for" and "while". Here's an example of a "for" loop in a shell script:



```
for i in {1..10}
do
  echo $i
done
```

In this example, the script prints out the numbers 1 through 10, one at a time.

Functions: Functions are used to group a set of commands together, which can then be called from within a script. Here's an example of a function in a shell script:



```
function greet {
  echo "Hello, $1!"
}

greet "Bob"
```

In this example, the script defines a function called "greet" which takes one argument and prints out a greeting with that argument.

These are just a few examples of the many shell script constructs available in Linux. By using these constructs, you can create powerful scripts to automate tasks and simplify your work on the command line.

Q. How do you use regular expressions in a shell script?

A. Regular expressions are a powerful tool for pattern matching and text manipulation in Linux shell scripting. You can use regular expressions in your shell scripts by using a command or function that supports regular expressions, such as grep, sed, or awk.

1. grep: grep is a command-line tool that searches for patterns in text files. You can use regular expressions with grep by using the -E option, which enables extended regular expressions.

2. sed: sed is a command-line tool that can perform text substitutions and other text transformations. You can use regular expressions with sed by using the "s" command, which stands for "substitute".

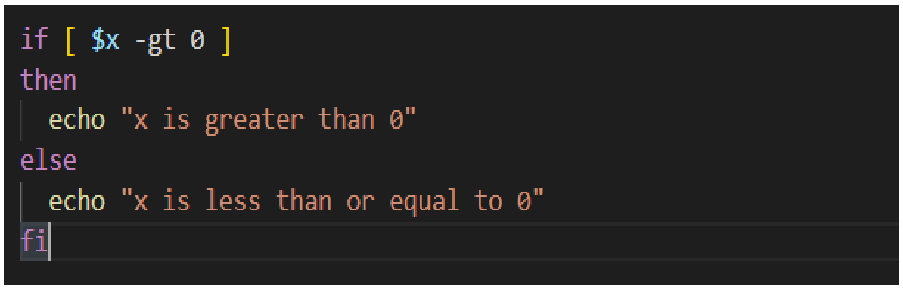
3. awk: awk is a command-line tool that can perform text processing and data analysis. You can use regular expressions with awk by using the match() function, which returns the position of the first occurrence of a regular expression in a string.

By mastering regular expressions, you can perform complex text processing and pattern matching tasks in your shell scripts with ease.

Q. How do you use conditional statements (e.g., if, case) in a shell script?

A. Conditional statements are a fundamental part of shell scripting in Linux, and they allow you to perform different actions based on specific conditions. There are several types of conditional statements that you can use in shell scripts, including the "if" statement, the "case" statement, and the "test" command. Here's a brief overview of how to use each of these conditional statements in a shell script:

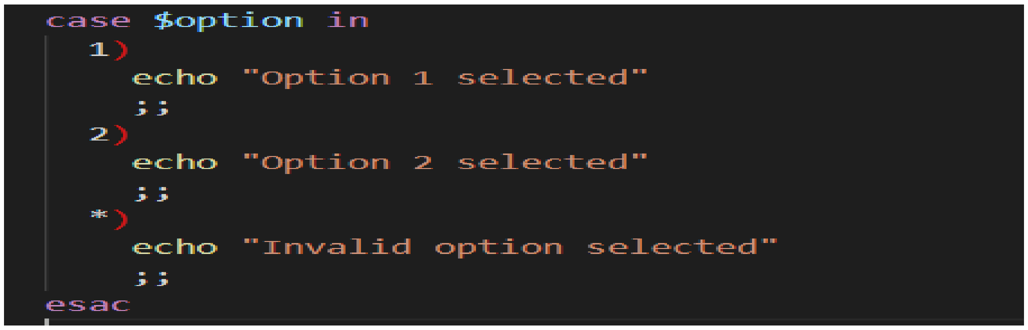
1. if statement: The "if" statement is used to test a condition and execute one set of commands if the condition is true, and another set of commands if the condition is false. Here's an example of an "if" statement in a shell script:



```
if [ $x -gt 0 ]
then
  echo "x is greater than 0"
else
  echo "x is less than or equal to 0"
fi
```

In this example, the script tests whether the variable "x" is greater than zero, and prints out a message depending on the result.

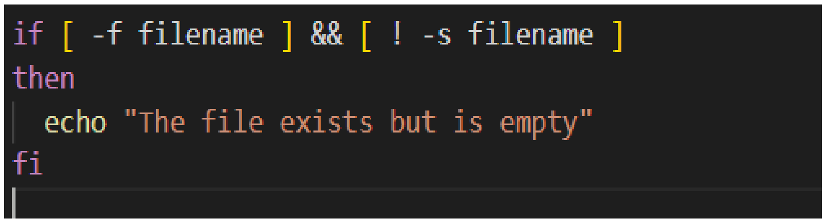
3. **case statement**: The "case" statement is used to match a variable against a set of patterns, and execute a set of commands for the first pattern that matches. Here's an example of a "case" statement in a shell script:



```
case $option in
    1)
      echo "Option 1 selected"
      ;;
    2)
      echo "Option 2 selected"
      ;;
    *)
      echo "Invalid option selected"
      ;;
esac
```

In this example, the script tests the variable "option" against the patterns "1" and "2", and executes a set of commands for the first pattern that matches. If none of the patterns match, the script executes the commands in the default case.

3. test command: The "test" command is used to test a condition and return a Boolean value (0 for true, 1 for false). You can use the "test" command in combination with conditional statements to perform more complex tests. Here's an example:



```
if [ -f filename ] && [ ! -s filename ]
then
  echo "The file exists but is empty"
fi
```

In this example, the script tests whether the file "filename" exists and is not empty, and prints out a message if the condition is true.

These are just a few examples of how to use conditional statements in Linux shell scripting. By mastering conditional statements, you can create powerful shell scripts that can perform complex tasks and automate your work on the command line.

Q. How do you handle errors and exceptions in a shell script?

A. Handling errors and exceptions is an important part of writing robust shell scripts in Linux. Error handling helps to ensure that your script runs smoothly, and can prevent unexpected behaviour or crashes. Here are some ways to handle errors and exceptions in a shell script:

1. Use exit codes: Most commands in Linux return an exit code, which is a numeric value that indicates whether the command completed successfully or encountered an error. In a shell script, you can use the "exit" command to set the exit code for your script, and use conditional statements to check the exit code and take appropriate actions.

2. Use the "set -e" option: The "set -e" option tells the shell to exit immediately if any command in the script returns a non-zero exit code. This can be useful for catching errors early and preventing your script from running in an unexpected state.

3.Use error messages and logging: You can use error messages and logging to provide more information about errors and exceptions that occur in your script. For example, you can use the "echo" command to print out error messages, or redirect output to a log file for later analysis.

Q. How do you work with files and directories in a shell script?

A. Here are some examples of how to work with files and directories in a shell script:

Creating and deleting directories: You can use the "mkdir" command to create a new directory, and the "rmdir" or "rm" command to delete a directory. For example:

```
mkdir mydir

rmdir mydir
```

In this example, the script creates a new directory called "mydir", and then deletes it using the "rmdir" command. Note that the "rmdir" command can only delete empty directories. If you want to delete a directory and its contents, use the "rm" command with the "-r" (recursive) option.

Creating and deleting files: You can use the "touch" command to create a new file, and the "rm" command to delete a file. For example:

```
touch myfile.txt

rm myfile.txt
```

In this example, the script creates a new file called "myfile.txt" using the "touch" command, and then deletes it using the "rm" command.

Listing files and directories: You can use the "ls" command to list the files and directories in a directory. For example:

```
ls /home/user/documents/
```

In this example, the script lists the files and directories in the "/home/user/documents/" directory.

Renaming files and directories: You can use the "mv" command to rename a file or directory. For example:

```
mv oldname.txt newname.txt
```

In this example, the script renames the file "oldname.txt" to "newname.txt" using the "mv" command.

Copying files and directories: You can use the "cp" command to copy a file or directory. For example:

```
cp myfile.txt /home/user/documents/
```

In this example, the script copies the file "myfile.txt" to the "/home/user/documents/" directory using the "cp" command.

Q. What are some common pitfalls when writing shell scripts?

A. There are several common pitfalls to watch out for when writing shell scripts in Linux. Here are some of the most common pitfalls and how to avoid them:

Not quoting variables: When using variables in shell scripts, it's important to put them in quotes to avoid issues with spaces and special characters.

Not checking for errors: It's important to check for errors when running commands in a shell script, and to handle them appropriately. One way to check for errors is to use the "set -e" option, which causes the script to exit immediately if any command returns a non-zero exit code.

Not using portable syntax: Some shell scripts may run fine on one system but not on another, due to differences in shell syntax or environment variables. To ensure that your script is portable across different systems, you should use portable syntax and avoid relying on system-specific features. For example, you should avoid using the "echo -n" option, which is not supported by all shells, and use the "printf" command instead.

Not cleaning up after the script: Shell scripts may create temporary files or directories that should be cleaned up after the script has finished running. To avoid leaving these files behind, you should use the "trap" command to catch signals and perform clean-up actions.

By watching out for these common pitfalls and following best practices for shell scripting, you can create more robust and reliable scripts that can help you automate your work more efficiently.

Q. How do you pass arguments to a shell script and how do you access them within the script?

A. You can pass arguments to a shell script by specifying them when you run the script from the command line. The arguments are passed to the script as positional parameters, which can be accessed within the script using special variables.

Here's an example of a simple script that takes two arguments and uses them to print a message:

```
#!/bin/bash

echo "Hello $1 and $2"
```

In this script, the first argument is accessed using the "$1" variable, and the second argument is accessed using the "$2" variable. To run this script with two arguments, you would type:

```
$ ./myscript.sh Alice Bob
```

This would produce the output:

```
Hello Alice and Bob
```

In addition to the "$1", "$2", etc. variables, you can also access all of the arguments passed to the script using the "$@" variable. This variable represents an array of all the positional parameters, and can be used to loop over all the arguments or pass them to another command.

Q. What does the. (dot) indicate at the beginning of a file name and how should it be listed?

A. The dot (.) at the beginning of a file name in Linux indicates that the file is a hidden file. Hidden files are not displayed by default when you use the "ls" command, unless you use the "-a" option to show all files, including hidden ones.

For example, if you have a hidden file called ".myconfig" in your home directory, you can list it using the following command:

```
$ ls -a ~ | grep '^\.'
```

In this command, the "-a" option is used to show all files in the home directory, including hidden ones. The output is then piped to the "grep" command to filter out all files that don't start with a dot. The "^" symbol in the regular expression indicates the beginning of the line, so the command matches only files that start with a dot.

It's important to note that hidden files are not truly hidden or protected in any way - they are simply files with a dot at the beginning of their name. You can still view, edit, and delete hidden files just like any other file, as long as you know their name and location.

Q. What is the alternative command available to echo and what does it do?

A. In Linux, the "echo" command is used to print messages and values to the terminal. An alternative command to "echo" is the "**printf**" command, which is more powerful and flexible than "echo".

The "printf" command is used to format and print text and variables to the terminal. It allows you to specify the format of the output using special formatting codes, such as "%s" for strings and "%d" for integers.

Overall, the "printf" command provides a more flexible and powerful way to format and print text and variables to the terminal than the "echo" command.

Q. Which command needs to be used to know how long the system has been running?

A. In Linux, you can use the "uptime" command to know how long the system has been running. The "uptime" command provides information about the current time, how long the system has been running, how many users are currently logged in, and the system load average for the past 1, 5, and 15 minutes.

To run the "uptime" command, simply open a terminal and type "**uptime**" followed by Enter.

Overall, the "uptime" command is a quick and easy way to check how long the system has been running and get an idea of its current status.

Q. How to find all the available shells in your system?

A. In Linux, you can use the "cat" command to view the contents of the "/etc/shells" file to find all the available shells on your system. This file contains a list of all shells installed on the system, including their paths.

To view the contents of the "/etc/shells" file, open a terminal and type the following command:

```
$ cat /etc/shells
```

The output of the "cat" command will list all the available shells on your system, with each shell on a separate line.

Q. What are the various commands that may be used to check the disc usage?

A. In Linux, there are several commands that you can use to check disk usage. Here are some of the most commonly used commands:

df: The "df" command shows the amount of disk space used and available on mounted filesystems. It can also display information about the file system type, the total size of the file system, and the percentage of disk space used.

```
$ df -h
```

du: The "du" command displays the disk usage of files and directories. It can be used to check the size of individual files or directories, and can display the size of each file or directory in human-readable format.

```
$ du -h /path/to/directory
```

ls: The "ls" command lists files and directories in a directory. By using the "-l" option, it can display information about file sizes and disk usage.

```
$ ls -lh
```

ncdu: The "ncdu" command is a text-based disk usage analyzer that allows you to navigate and view disk usage information in a directory tree. It provides a graphical representation of the disk usage and allows you to interactively explore and manage disk space usage.

```
$ ncdu /path/to/directory
```

These commands can be used to get a better understanding of the disk usage on your Linux system, and to identify any areas where disk space might be running low.

Q. Explain in brief about awk command with an example.

A. Awk is a powerful text processing command in Linux that allows you to manipulate and analyze data files. Awk is especially useful for working with structured data files, such as those containing delimited fields or fixed-width columns.

Awk works by scanning through a file or stream of input data, one line at a time. Each line is split into fields based on a specified delimiter (by default, whitespace is used as the delimiter). Awk then performs various operations on these fields, such as pattern matching, filtering, and data transformation. Here's an example:

```
awk '{if (match($0, /pattern/)) print $0}' filename
```

In this example, the script searches for lines in the file "filename" that contain the regular expression "pattern" and prints out those lines.

Overall, the Awk command provides a powerful and flexible way to process and manipulate text data in Linux. It is especially useful for working with structured data files and can be combined with other commands like grep, sed, and cut to create more complex data processing pipelines.

Q. What are the three different security provisions provided by UNIX for a file or data?

A. **User-based security**: In UNIX, each file or directory is associated with a user and a group. User-based security is implemented by setting permissions for each file or directory based on the user who owns it, the group it belongs to, and other users who are not the owner or a member of the group. The "chmod" command is used to set permissions for files and directories, and the "chown" command is used to change the ownership of files and directories.

Role-based security: In UNIX, role-based security is implemented by creating user groups with specific privileges and assigning users to these groups based on their roles or responsibilities. This is often used in enterprise environments where different users have different levels of access to files or systems. Role-based security can be implemented using the "usermod" and "groupmod" commands.

Access control lists (ACLs): Access control lists provide a more fine-grained way to control access to files or data than traditional UNIX permissions. ACLs allow you to specify permissions for specific users or groups on a per-file or per-directory basis, rather than just for the owner or group. ACLs can be set and modified using the "setfacl" and "getfacl" commands.

Together, these security provisions provide a robust and flexible way to protect files and data in UNIX systems. By setting appropriate permissions, assigning users to groups, and using ACLs where necessary, system administrators can control who has access to sensitive data and ensure that it is kept secure.

Q. Explain in brief about sed command with an example.

A. Sed (short for Stream Editor) is a command-line utility that is commonly used in Unix-like operating systems, including Linux. It is a powerful tool for processing and transforming text data from files or pipelines.

Sed works by reading one line of input at a time, applying a set of editing commands to it, and then printing the modified line to standard output. The editing commands can be used to perform various tasks, such as:

Replacing or deleting text,

Inserting or appending text,

Filtering lines based on a pattern match,

Transforming text based on regular expressions,

Performing more complex operations with scripts.

Here is an example of how to use the sed command to replace a string in a file:

$ sed 's/old-string/new-string/g' input-file > output-file

In this command:

"s/old-string/new-string/g" is the substitution command, which replaces all occurrences of the "old-string" with the "new-string".

"input-file" is the name of the file to process.

"> output-file" redirects the output of the sed command to a new file called "output-file".

Sed is a versatile and powerful tool that can be combined with other commands like grep, awk, and cut to create more complex data processing pipelines.

Q. Why is bash a weakly typed language?

A. Bash is considered a weakly typed language because it does not require variable declaration or explicitly specifying variable types. In other words, a variable in Bash can hold any data type without needing to define its type beforehand.

This flexibility can be convenient for scripting and rapid prototyping, but it also means that Bash scripts may be more error-prone and harder to debug if variables are misused or assigned incorrect values.

In contrast, strongly typed languages like Java or C++ require explicit declaration of variable types and will throw errors if variables are assigned values of the wrong type. While this can be more verbose and may require more effort upfront, it can also prevent certain types of programming errors and improve code reliability in the long run.

Q. What function does a pipe operator serve? How can you run several commands in one line?

A. The pipe operator, represented by the vertical bar symbol |, is used in Linux and Unix shells to connect the output of one command to the input of another command. This allows you to create more complex and powerful command-line tools by combining multiple simple commands into a single pipeline.

For example, suppose you have a text file named data.txt that contains a list of numbers, one per line, and you want to find the average of these numbers. You could use the cat command to display the contents of the file, and then pipe the output to the awk command to perform the calculation:

cat data.txt | awk '{ sum += $1 } END { print sum / NR }'

In this command, the cat command reads the contents of the file and prints them to standard output, which is then fed into the awk command via the pipe operator. The awk command performs the calculation by adding up all the numbers in the first column (sum += $1) and then printing the average at the end of the input stream (END { print sum / NR }).

To run several commands in one line, you can use a semicolon ; to separate the commands. For example, to create a new directory, change into it, and then create a new file inside it, you could use the following command:

mkdir mydir; cd mydir; touch myfile.txt

This command will execute each of the three commands in order, regardless of whether any of them produce output or errors. You can also use the && operator to execute commands sequentially only if the previous command succeeds, or the || operator to execute commands sequentially only if the previous command fails.

Q. Differentiate between grep and egrep.

A. grep and egrep are both commands used in Unix and Linux environments to search for specific patterns or regular expressions in a given text file or output. The main difference between them is in the type of regular expressions they support.

grep supports basic regular expressions (BREs), which means that it uses a simpler syntax to match patterns. In BREs, special characters like |, *, +, and (need to be escaped with a backslash \ to be treated as literals.

egrep (or grep -E) supports extended regular expressions (EREs), which offer a more powerful syntax for matching patterns. In EREs, many special characters do not need to be escaped, and additional features like alternation and quantifiers are available.

In general, egrep is more versatile and easier to use for complex pattern matching, while grep is simpler and faster for basic searches.

Q. List 10 commonly used Linux commands.

A. Here are 10 commonly used Linux commands:

ls - List the contents of a directory

cd - Change the current working directory

mkdir - Create a new directory

rm - Remove a file or directory

cp - Copy a file or directory

mv - Move or rename a file or directory

grep - Search for a pattern in a file or output

awk - Process and manipulate text data

chmod - Change the permissions of a file or directory

sudo - Execute a command with superuser privileges

These commands are essential for navigating the Linux command line and performing basic file and system operations.

Q. Explain the lifespan of a variable inside a shell script?

A. The lifespan of a variable inside a shell script depends on its scope, which is determined by where and how the variable is declared.

If a variable is declared outside of any function or loop, it has a global scope and can be accessed from any part of the script. Its lifespan lasts until the script terminates or the variable is unset.

If a variable is declared inside a function, it has a local scope and can only be accessed within that function. Its lifespan lasts as long as the function is running and the variable is in scope. Once the function ends, the variable is destroyed and its value is no longer available.

Additionally, variables can also have a lifespan within a loop, where they are created and destroyed with each iteration of the loop.

It's important to note that variables in shell scripts are not strongly typed, meaning their data type can change depending on the value assigned to them. This can affect their lifespan and scope if the variable is reassigned with a different type of value.

Q. How to make variables as immutable?

A. In Bash, it is possible to make variables immutable (i.e., read-only) by using the readonly command. This prevents the variable from being modified or reassigned later in the script.

To make a variable immutable, use the following syntax:

readonly variable_name

Immutable variables are useful in situations where you want to ensure that a certain value remains constant throughout the script execution, and prevent accidental or intentional changes to it.

Q. What is Shell?

A shell is a command-line interface that provides a way for users to interact with the operating system. It interprets commands entered by the user and executes them.

Q. Why is a shell script needed?

A shell script is a program that consists of a series of commands written in a scripting language. It is needed to automate repetitive tasks, perform system administration tasks, and perform complex operations that cannot be done manually.

Q. Write some advantages of shell scripting.

Shell scripts are easy to write and can be created quickly.

They are portable and can be run on any system that has a compatible shell installed.

They can automate repetitive tasks, saving time and reducing the risk of errors.

They can be used for system administration tasks, such as backups, monitoring, and user management.

They can be used to perform complex operations that cannot be done manually.

Write some limitations of shell scripting.

Shell scripts can be slower than compiled programs.

They can be less secure than compiled programs.

They may be less reliable than compiled programs because of their reliance on external commands.

They may be limited in their ability to perform complex operations.

Q. Name the file in which shell programs are stored.

Shell programs are stored in shell scripts, which are plain text files with a .sh extension.

Q. Name different types of shells available.

Some common types of shells include:

Bourne shell (sh)

C shell (csh)

Korn shell (ksh)

Bourne-Again shell (bash)

Z shell (zsh)

Q. Write the difference between Bourne Shell and C Shell.

The Bourne shell (sh) and C shell (csh) are two different types of shells. The Bourne shell is a simple shell that is best suited for running shell scripts. It has a simple syntax and is easy to learn. The C shell, on the other hand, is a more complex shell that is best suited for interactive use. It has a more complex syntax and offers more features than the Bourne shell.

Q. What do you mean by Shell variable?

A shell variable is a variable that is used to store a value in a shell script. It can be used to hold data that can be accessed and modified by the script.

Q. What are different types of variables mostly used in shell scripting?

The different types of variables mostly used in shell scripting are:

Local variables

Environment variables

Shell variables

Q. Explain the term positional parameters.

Positional parameters are the arguments that are passed to a shell script or function when it is called. They are referenced using the $ symbol followed by a number, starting with 1 for the first argument.

Q. What are control instructions?

Control instructions are commands that control the flow of execution in a shell script. They include conditional statements, loops, and functions.

Q. How many types of control instructions are available in a shell?

There are three types of control instructions available in a shell:

Conditional statements (if/elif/else)

Loops (for/while/until)

Functions

Q. What do you mean by GUI Scripting?

GUI scripting is the process of automating user interactions with a graphical user interface (GUI) using a scripting language. It is used to automate repetitive tasks that involve interacting with the GUI.

Q. What is the shebang line in shell scripting?

The shebang line is the first line in a shell script that starts with #! followed by the path to the shell interpreter that will be used to run the script.

Q. What is a file system? Write down the four core components of a Linux file system.

A file system is a way of organizing and storing files on a computer or storage device. It provides a hierarchical structure for files and directories, and manages access to files and storage space. The four core components of a Linux file system are:

Boot block: contains the boot loader program and other boot related files.

Superblock: contains metadata about the file system, including size, location of inodes, and location of free blocks.

Inode: contains metadata about a file, such as its permissions, owner, group, and location on disk.

Data block: contains the actual contents of files and directories.

Q. What is a shell script and why is it used?

A shell script is a script written in a shell language, such as Bash, that can be executed in a command-line interface. It is used to automate repetitive tasks or to execute a series of commands in a specific sequence. Shell scripts can be used for system administration, application deployment, and data processing, among other purposes.

Q. How do you create a shell script?

To create a shell script, open a text editor and start writing the commands or scripts you want to automate. Save the file with a .sh extension, which indicates that it is a shell script. Then, make the script executable by changing its file permissions with the chmod command.

Q. What is the #! (shebang) used for in a shell script?

The #! (shebang) is a special character sequence that is used to specify the interpreter for a shell script. It appears at the beginning of the script and is followed by the path to the shell interpreter that will execute the script. For example, #!/bin/bash specifies that the Bash shell should be used to interpret the script.

Q. How do you include comments in a shell script?

Comments in a shell script can be included using the # character at the beginning of a line. Everything after the # character is considered a comment and is ignored by the shell interpreter. Comments can be used to provide documentation, describe the purpose of the script, or to disable certain commands temporarily.

Q. How do you run a shell script?

To run a shell script, open a terminal window and navigate to the directory where the script is located. Then, enter the command ./scriptname.sh, replacing scriptname.sh with the name of your script. Alternatively, you can run the script by specifying the full path to the script.

Q. How do you perform input and output redirection in a shell script?

Input and output redirection can be performed in a shell script using the < and > characters, respectively. For example, to redirect the output of a command to a file, you can use the command > filename.txt. To redirect the input of a command from a file, you can use the command < filename.txt.

Q. What are some common shell script commands?

Some common shell script commands include echo, which displays messages on the screen; cd, which changes the current directory; ls, which lists the contents of a directory; and grep, which searches for text within a file or output.

Q. What are some common shell environments?

Some common shell environments include Bash, which is the default shell in many Linux distributions; Zsh, which is a more powerful shell with advanced features; and Fish, which is designed to be more user-friendly than other shells.

Q. How do you use command line arguments and options in a shell script?

Command line arguments are parameters passed to a script when it is invoked. They allow the user to provide inputs to the script without modifying the script itself. In a shell script, you can access the command line arguments using the variables $1, $2, $3, etc., where $1 represents the first argument, $2 represents the second argument, and so on. You can also use the special variable $0 to access the name of the script itself.#!/bin/bash

```
echo "The first argument is: $1"

echo "The second argument is: $2"

echo "The third argument is: $3"

echo "The script name is: $0"
```

Q. When should shell programming/scripting not be used?

Shell programming is not suitable for complex tasks or programs that require a lot of computational power. It is best suited for automating small tasks or for administrative tasks. In addition, if a task requires high-level programming, it is better to use other programming languages such as Python, C, or Java.

Q. What are the default permissions of a file when it is created?

When a file is created in Linux, it is assigned default permissions of 666. This means that the owner, group, and others have read and write access to the file.

Q. How can a variable be made to be unchangeable?

In shell scripting, you can make a variable read-only using the readonly command.

Q. What are some of Shell Scripting's drawbacks?

Some of the drawbacks of shell scripting include:

Limited support for data structures

Difficulty in handling errors and exceptions

Limited portability between different shell environments

Limited support for multi-threading and parallel processing

Lack of support for object-oriented programming

Q. Explain the way you can connect to a database server.

You can connect to a database server from a shell script using command-line utilities such as mysql, psql, or sqlite3. These utilities allow you to execute SQL commands and retrieve data from the database.

To connect to a database server, you need to provide the server address, port number, username, and password. Here is an example of connecting to a MySQL database server:

```
#!/bin/bash

# Connect to MySQL database server
mysql -h localhost -P 3306 -u myusername -p mypassword mydatabase
```

Q. What is the importance of $#?

A. In Bash, $# is a special parameter that represents the number of positional parameters passed to a shell script or function.

Positional parameters are the arguments passed to a script or function when it is invoked, and they are referred to using the special variables $1, $2, $3, and so on. $# provides a way to determine the total number of positional parameters passed, regardless of their values.

The importance of $# lies in its ability to enable the script or function to handle variable number of arguments dynamically. For example, if a script is designed to take a variable number of arguments, it can use $# to check the number of arguments passed and adjust its behaviour accordingly.

Q. Write the command that is used to execute a shell file.

A. To execute a shell script in Linux, you can use the following command:

```
./script.sh
```

where script.sh is the name of the shell script you want to execute. The ./ notation specifies that the script is located in the current directory. If the script is located in a different directory, you can specify the full path to the script instead of ./.

Q. Name the command that can be used to find out the number of arguments passed to the script?

A. In Bash, the number of arguments passed to a shell script can be determined using the $# special parameter. The value of $# represents the number of positional parameters passed to the script.

For example, if your script is called myscript.sh, you can check the number of arguments passed to it like this:

```
#!/bin/bash

echo "The number of arguments is: $#"
```

Q. How long does a variable in a shell script last?

A. The lifespan of a variable in a shell script is determined by its scope, which is defined by where the variable is declared and initialized.

If a variable is declared and initialized within a function or a block of code, its scope is limited to that function or block. The variable is destroyed when the function or block exits.

If a variable is declared and initialized outside of any function or block, its scope is global. The variable is accessible from anywhere in the script, and its value persists until the script exits.

In addition to scope, the lifespan of a variable can also be affected by its type. For example, environment variables are inherited by child processes, so their lifespan extends beyond the lifespan of the script that created them.

It's important to note that shell scripts do not have garbage collection, so variables that are not explicitly unset or re-assigned can potentially consume memory until the script terminates. To avoid memory leaks, it's best practice to unset variables when they are no longer needed.

Overall, the lifespan of a variable in a shell script depends on its scope and type, and is determined by the script's execution flow.

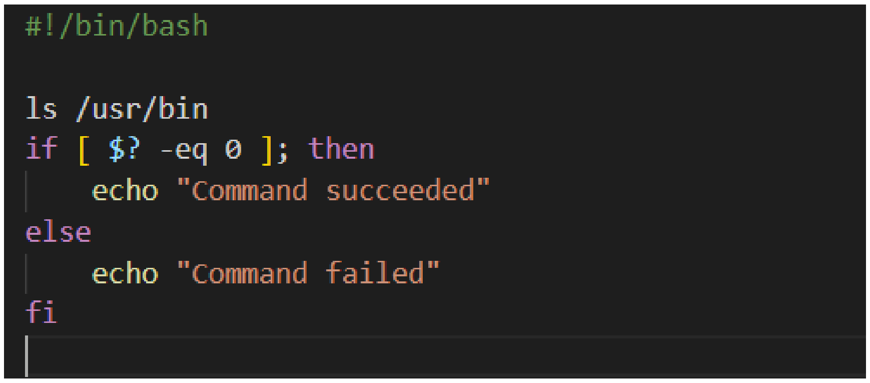
Q. What is the use of the "$?" command?

A. In Linux and Unix-like systems, the $? command is a special shell variable that stores the exit status of the last executed command or script.

After running a command or script, the value of $? will be set to a numeric value indicating the result of the command. A value of 0 indicates success, while any non-zero value indicates an error or failure of some kind. The specific values returned can vary depending on the command or script being executed.

The $? variable is often used in scripts to test the success or failure of a command or script. For example, you can use it to control the flow of your script, such as stopping the script if a critical command fails, or running a different command if the previous one was successful.

Here's an example of using $? to check the success or failure of a command in a shell script:



```bash
#!/bin/bash

ls /usr/bin
if [ $? -eq 0 ]; then
    echo "Command succeeded"
else
    echo "Command failed"
fi
```

In this example, the ls command is executed to list the files in the /usr/bin directory. The $? variable is then checked to see if the command succeeded ($? -eq 0) or failed ($? -ne 0), and a message is printed accordingly.

Q. What is the best way to run a script in the background?

A. To run a script in the background in Linux or Unix-like systems, you can use the & operator at the end of the command or script. This will start the command or script in the background and return control to the shell prompt immediately, allowing you to continue using the terminal.

This will start the script in the background and return the prompt to you. You can continue to use the terminal while the script runs in the background.

Q. What do you mean by crontab and name two files of crontab command.

A. Crontab is a command in Linux and Unix-based operating systems that allows users to schedule commands or scripts to run automatically at specified intervals or times. The name cron comes from the Greek word "chronos", which means "time".

The crontab command is used to create, view, and edit the cron jobs that are scheduled to run on the system. It uses a special format for specifying the time and date when the command should be executed.

Two files associated with the crontab command are:

/etc/crontab: This file is the system-wide crontab file that is used by the cron daemon to schedule system-level tasks. It is usually edited by system administrators and contains entries for running commands or scripts as specific users at specific times.

~/crontab: This file is the user-specific crontab file that is used by each individual user to schedule tasks that run under their own user account. Users can create, edit, and delete their own crontab files using the crontab command.

Both of these files use the same format for specifying the time and date when commands or scripts should be executed, but they have different locations and permissions.

Q. Which command can be used to take the backup.

A. In Linux, there are several commands that can be used to take backups, depending on the type of backup you want to take and the files or directories you want to include in the backup. Here are a few commonly used backup commands:

tar: This command is used to create an archive file from one or more files or directories. It can also be used to compress the archive file to save space. For example, to create a backup of the /home/user1 directory, you could use the following command:

```
tar -cvzf /backup/user1_backup.tar.gz /home/user1
```

This command will create a compressed tar archive (user1_backup.tar.gz) of the /home/user1 directory and save it in the /backup directory.

rsync: This command is used to synchronize files and directories between different locations. It can be used to create incremental backups that only copy the changes made to files since the last backup. For example, to create a backup of the /home/user1 directory using rsync, you could use the following command:

rsync -avz /home/user1 /backup/user1_backup

This command will create a copy of the /home/user1 directory in the /backup/user1_backup directory.

dd: This command is used to create a bit-by-bit image of a disk or partition. It can be used to create a backup of an entire disk or to clone a disk to another disk. For example, to create a backup of the /dev/sda disk, you could use the following command:

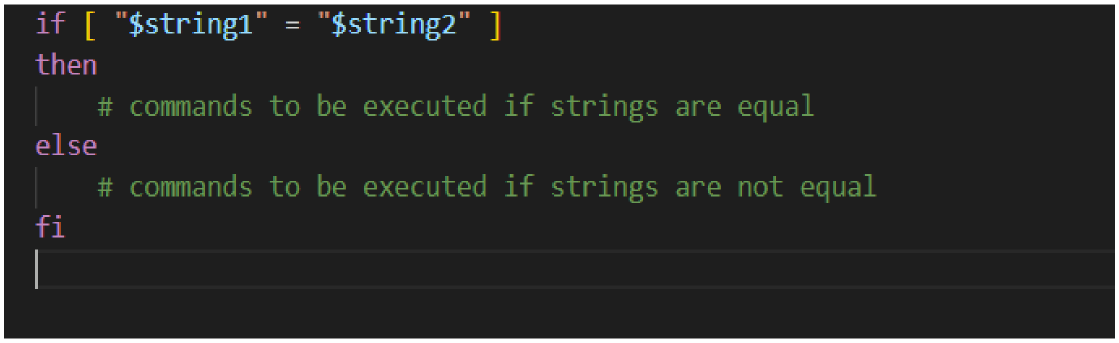
dd if=/dev/sda of=/backup/sda_backup.img

This command will create an image (sda_backup.img) of the /dev/sda disk and save it in the /backup directory.

These are just a few examples of backup commands that can be used in Linux. There are many other commands and tools available for taking backups, each with its own features and options.

Q. Name the command that is used to compare the strings in a shell script.

A. The test or [ ] command can be used to compare strings in a shell script. The syntax for string comparison is:



```
if [ "$string1" = "$string2" ]
then
    # commands to be executed if strings are equal
else
    # commands to be executed if strings are not equal
fi
```

Here, = is used to check if the two strings are equal. Other comparison operators such as !=, <, >, <=, and >= can also be used as per the requirement.

Q. Write the difference between $* and $@

The $* expression starts from one and expands to the positional parameters. If the expression is not used with double quotes, each positional parameter expands to a separate word and when used within the double quotes, the value of each parameter expands to a single word.

The $@ expression starts from one and expands to the positional parameters. If the expression is used within double quotes, each parameter expands to a separate word.

Q. List all supported types of loops in bash.

There are three types of loops supported in Bash:

For loop: A "for" loop is used to iterate over a sequence of values, such as an array or a range of numbers. The syntax for a "for" loop in Bash is as follows:

```
 for variable in list
  do
  # commands to be executed
done
```

Here, "variable" is a variable that takes on the values in "list" one at a time, and the commands within the loop are executed for each value.

While loop: A "while" loop is used to repeatedly execute a set of commands as long as a certain condition is true. The syntax for a "while" loop in Bash is as follows:

```
 while [ condition ]
 do
  # commands to be executed
 done
```

Here, "condition" is an expression that evaluates to true or false, and the commands within the loop are executed as long as the condition is true.

Until loop: An "until" loop is similar to a "while" loop, but it executes the commands within the loop as long as the condition is false. The syntax for an "until" loop in Bash is as follows:

```
 until [ condition ]
 do
  # commands to be executed
done
```

Here, "condition" is an expression that evaluates to true or false, and the commands within the loop are executed until the condition becomes true.

These loops are powerful constructs that allow Bash scripts to perform repeated actions efficiently and effectively.

Q. What does interactive and non-interactive shells means?

An interactive shell is defined as the shell that simply takes commands as input and acknowledges the output to the user. Interactive scripts couldn’t run in background as they required input from user.
For example-: a bash shell is an interactive shell.

As the name implies, a non-interactive shell is a type of shell that doesn’t interact with the user. A non-interactive shell is probably run from an automated process so it can’t assume it can request input or that someone will see the output.
For example-: Scripts like Init and startup

Q. How pass and access arguments are passed in a script?

The arguments are passed to the script as command line arguments when the script as command line arguments when the script is invoked.
For example-: if the script is named “abcscript.sh” and it requires two arguments, like ./abcscript.sh arg1 arg2

These arguments can be accessed using the variable ‘$’ symbol followed by the integer to access the arguments passed like ‘$1’, ‘$2’, ‘$3’ and so on where ‘$1’ refers to the first argument, ‘$2’ refers to second argument and so on. If the script requires more than nine arguments, we can use braces to enclose the variable number. For example -: to access 10th argument, ‘${10}’.

Q. s" permission bit in a file means?

The “s” permission bit has a different meaning depending on permission fields of the file. When the “s” permission bit is set in the user permission field it indicates setuid bit is set. This means that when a user executes the files the process will run with the permission of the files owner.
When the “s” permission bit is set in the group permission field it indicates the setgid bit is set. This means that when a user executes the files the process will run with the permission of the files group.
If the “s” permission bit is set in the other permission field it is treated as a lowercase “s”.

Q. Detail all the debug processes in the shell script and explain the best way.

Debugging a shell script involves identifying and fixing errors in the code. There are several ways to debug a shell script, including:

Adding print statements: One of the simplest ways to debug a shell script is by adding print statements throughout the code. These print statements can output the values of variables or indicate the point in the code where the script fails. This method is useful for identifying syntax errors or logical errors in the code. However, it can be time-consuming and tedious to add and remove print statements as the code evolves.

Running the script in debug mode: Another method for debugging a shell script is by running it in debug mode. In this mode, the shell prints out each command before it is executed, and each variable is printed out as it is assigned a value. To run a shell script in debug mode, add the "-x" option to the shebang line at the beginning of the script, like this:

```
#!/bin/bash -x
```

Using a shell debugger: A shell debugger is a tool that allows to step through a shell script one line at a time, examine variables, and set breakpoints. There are several shell debuggers available, including Bashdb and Shdb. These tools can be useful for complex scripts that are difficult to debug using print statements or debug mode.

Break the script into smaller pieces: If having trouble debugging a large script, try breaking it down into smaller pieces and testing each piece separately. This can help isolate the problem and make it easier to fix.

In general, the best way to debug a shell script will depend on the complexity of the script and the nature of the errors. For simple scripts with syntax errors or logical errors, adding print statements or running the script in debug mode may be sufficient. For more complex scripts or hard-to-find errors, a shell debugger may be necessary. It is also important to write clean, well-documented code and to test the script thoroughly before attempting to debug it.

Q. What are metacharacters?

In shell scripting, metacharacters are characters that have a special meaning to the shell and are used to perform various operations. Some common metacharacters in shell scripting include:

1. * (asterisk): Matches any number of characters, including none.

2. ‘?' (question mark): Matches any single character.

3. '|" (pipe): Used for piping the output of one command as input to another command.

4. ">' (greater than): Used for redirecting the output of a command to a file or device.

5. <` (less than): Used for redirecting input from a file or device to a command.

6. '& (ampersand): Used to run a command in the background.

7. ';' (semicolon): Used to separate multiple commands on a single line.

when using metacharacters in shell scripting, they must be escaped or quoted appropriately to ensure that they are interpreted correctly by the shell.

Q. Write the difference between “=” and “==”.

In shell scripting, "=" and "==" are used for different purposes:

1. "" is used for variable assignment. It assigns the value on the right-hand side of the operator to the variable on the left-hand side. For example

name="John"

This assigns the string "John" to the variable

"name"

2, "==" is used for string comparison. It checks whether the strings on both sides of the operator are equal, For example:

if [“$name” == “John”]

then

echo "The name is John"

This compares the value of the variable "name" with the string "John" and prints the message "The name is John" if they are equal.

Q. Name the command that is used to display the shell's environment variable.

The "env" command is used to display the shell's environment variables. The output of the "env" command shows a list of all environment variables that are currently set in the shell, along with their values. The syntax for the "env" command is as follows:

env

We can also use the "printenv" command to display the values of individual environment variables. For example, to display the value of the "PATH" environment variable, use the following command: printenv PATH

Q. How do you use shell scripts to manage and manipulate files and directories?

Shell scripts are a powerful tool for managing and manipulating files and directories in a Unix-like operating system. Here are some common ways to use shell scripts for file and directory management:

1. Moving and renaming files and directories: The `mv ` command can be used to move and rename files and directories. For example, the following command moves a file named `file.txt ` from the current directory to a directory named `newdir and renames it to 'newfile.txt`:

```
mv file.txt newdir/newfile.txt
```

2, Creating and deleting flles and directories: The 'touch' command can be used to create an empty file, while the mkdir command can be used to create a new directory. The `rm` command can be used to delete files and directories. To delete a directory and all of its contents, you can use the '-r`.

3 Listing files and directories: The 'ls' command can be used to list the files and directories in the current directory. The '1s` command also has several options that can be used to customize its output.

4. Searching for files: The find' command can be used to search for files in a directory tree. For example, the following command finds the files in the current directory and its subdirectories that have the .txt extension:

```
find . -name "*.txt"
```

5. Modifying file permissions: The ` chmod` command can be used to modify the permissions of a file or directory.

These are just a few examples of how shell scripts can be used for file and directory management.

Q. How will you find total shells available in your system?

We can use the ‘cat' command to display the contents of the /etc/shells' file, which contains a list of all the shells available on our system. Each shell is listed on a separate line .

To display the contents of the ‘/etc/shells ` file, entering the following command : cat /etc/shells will display the contents that available in our system.

Q. **Explain how you will open a read-only file in Unix/Shell.**

To open a read-onty file In Unix or shell, we can use any of the following commands:

‘cat filename ': This command will display the contents of the file on the terminal. Since it is read-only, you cannot modify the file.

2 `less filename `: This command will open the file in a read-only mode and display its contents on the terminal. You can navigate through the file using the arrow keys, but you cannot modify it.

3. more filename `: This command is similar to the less ` command and will display the contents of the file on the terminal, However, it will not allow you to navigate through the file using arrow keys.

4. "head filename`; This command will display the first 10 lines of the file on the terminal.

5. 'tail filename'; This command will display the last 10 lines of the file on the terminal.

Q. Name the command that one should use to know how long the system has been running.

The command that can be used to know how long the system has been running is "uptime". When we enter this command in the terminal or command prompt, it will display the current time, how long the system has been running, and the number of users currently logged in.

Q. Write the difference between $$ and $!?

$$ is a special variable that holds the process ID (PID) of the current shell. This can be useful for various purposes, such as creating temporary files with unique names.

$! is a special variable that holds the PID of the last background process started by the shell. This can be useful for monitoring the status of background processes, or for waiting for them to finish before continuing with other commands.

$$ holds the PID of the current shell, while $! holds the PID of the last background process started by the shell.

Q. Write difference between grep and find command.

grep is a command-line utility that is used to search for patterns in files or text streams. It is commonly used to search for specific strings or regular expressions within files or output generated by other commands. The basic syntax of grep

```
grep pattern [file ...]
```

where pattern is the string or regular expression to search for, and file is the name of the file

find is a command-line utility that is used to search for files and directories in a file system hierarchy. It is commonly used to locate files based on various criteria such as name, size, modification time, and ownership. The basic syntax of find is

```
find [path ...] [expression]
```

where path is the starting directory or directories to search in, and expression is the search criteria that specifies which files or directories to include in the search.

grep is used to search for patterns within files or text streams, while find is used to search for files and directories based on various criteria within a file system hierarchy.

Q. How can we create a function in shell script?

In shell scripting, you can create a function by defining it using the following syntax:

```
function_name() {

  # function body

}
```

where, function_name: This is the name given to function. It should be a descriptive name that reflects the purpose of the function.

(): This is where we specify any arguments that our function takes. If function doesn't take any arguments, we can leave this empty.

{}: This is where we define the body of our function. We can include any commands or code that we want our function to execute.

Here's an example function that takes two arguments and prints them to the console:

```
print_arguments() {
  echo "Argument 1: $1"
  echo "Argument 2: $2"
}
```

To call this function and pass it two arguments, one can use the following syntax:

```
print_arguments "Hello" "world"
```

We can define as many functions as need in our shell script, and we can call them from anywhere in our code. Functions can help make our code more modular and easier to maintain.

Q. How do you optimize the performance of a shell script?

Optimizing the performance of a shell script can involve several techniques, depending on the specific requirements and constraints of the script. Here are some general tips that can help improve the performance of a shell script:

Check the loops in the script. Time consumed by repetitive operations adds up quickly. If at all possible, remove time-consuming operations from within loops.

Avoiding unnecessary commands, particularly in a pipe.

Use built-in commands: Built-in commands are typically faster than external commands because they don't require launching a separate process. For example, use "echo" instead of "cat" to print the contents of a file.

Try to minimize file I/O. Bash is not particularly efficient at handling files, so using more appropriate tools for this within the script.

Q. How do you use shell scripts to manipulate and manage processes, such as by killing, starting, or stopping them?

Shell scripts can be used to manipulate and manage processes in a variety of ways, such as by killing, starting, or stopping them. Here are some examples of how to do this:

1. Killing a process: The "kill" command is used to terminate a process. To kill a process with a specific PID (Process ID), use the following command:

```
kill PID
```

2. Starting a process: To start a process from a shell script, use the command that we would normally use to start the process. For example, to start a new instance of the Apache web server, use the following command:

```
/etc/init.d/apache2 start
```

3. Stopping a process: The command used to stop a process will depend on the process in question. For example, to stop the Apache web server, we use the following command:

```
/etc/init.d/apache2 stop
```

If we want to restart a process instead of stopping it, use the "restart" command instead of "stop".

4. Checking the status of a process: The "ps" command can be used to check the status of a process. For example, to check the status of the Apache web server, use the following command

```
ps -ef | grep apache
```

This command will show a list of all processes that contain the word "apache" in their name.

5. Managing processes with variables: We can use variables to store the PID of a process and manipulate it later. For example, to store the PID of the Apache web server in a variable called "apache_pid", use the following command:

```
apache_pid=$(ps -ef | grep apache | awk '{print $2}')
```

We can then use the "kill" command to terminate the process using the variable:

```
kill $apache_pid
```

These are some examples of how to use shell scripts to manipulate and manage processes. The specific commands and techniques used will depend on the process in question and the specific requirements of the script

Q. How do you use shell scripts to interact with APIs or other external services?

Curl is commonly referred to as a non-interactive web browser for the Linux terminal. It is a tool to transfer data to or from a server, with access to a huge variety of protocols, including HTTP, FTP, SFTP, SCP, IMAP, POP3, LDAP, SMB, SMTP, and many more. It's a useful tool for the average sysadmin, whether we use it as a quick way to download a file needed from the Internet, or to script automated updates. Curl is also an important tool for testing remote APIs. If a service we rely on or provide is unresponsive, we can use the curl command to test it.

Q. How you will check if a file exists on the file system?

In Bash, we can use a 'test command' to check whether a file exists and determine the type of a file or its equivalent shorthand [command.

Syntax :

test [expression]

Here, in expression, we write parameter and file name. There are some parameters that can be used in the expression. Some are listed below-:

–f: It returns True if the file exists as a common (regular) file.

-d: it returns True if directory exists.

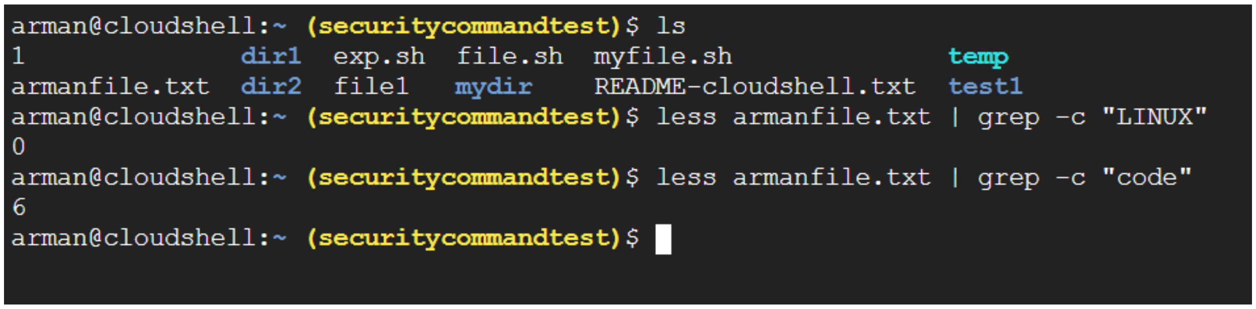
-e: It returns True if any type of file exists.

Q. What is the difference between `[[ $string == "efg*" ]]` and `[[ $string == efg* ]]`

In the first example, [[$string == "efg*"]], the double quotes around "efg*" indicate that the string should be treated literally, and the * character should be interpreted as a literal asterisk. This means that the expression will only evaluate to true if the value of $string is exactly "efg*".

In the second example, [[$string == efg*]], the double quotes are not present, which means that the * character will be interpreted as a wildcard. This means that the expression will evaluate to true if the value of $string starts with "efg" followed by any number of characters.

Q. Find the number of lines in a file that contains the letter "LINUX."



```
arman@cloudshell:~ (securitycommandtest)$ ls
1              dir1  exp.sh  file.sh  myfile.sh              temp
armanfile.txt  dir2  file1   mydir    README-cloudshell.txt  test1
arman@cloudshell:~ (securitycommandtest)$ less armanfile.txt | grep -c "LINUX"
0
arman@cloudshell:~ (securitycommandtest)$ less armanfile.txt | grep -c "code"
6
arman@cloudshell:~ (securitycommandtest)$
```

Q. Explain in details Zombie Processes in shell scripting?

A zombie process is a process whose execution is completed but it still has an entry in the process table. Zombie processes usually occur for child processes, as the parent process still needs to read its child's exit status. Once this is done the zombie process is eliminated from the process table. This is known as reaping the zombie process. Zombie processes are generally harmless and do not consume any system resources. However, they can accumulate over time and consume memory resources in the process table, which can lead to performance degradation.

Zombie processes can be identified using the ps command, which shows the process status of running processes. To remove zombie processes, the parent process should use the wait() system call to collect the exit status of the child process. In shell scripting, it's good practice to clean up zombie processes to prevent system resource depletion and improve system performance.