Linux Programming Step Material

UNIT – I

PART – A (Short Answer Questions)

1. **What is an operating system?**
   Ans) The **o**perating **s**ystem (**OS**) is the most important program that runs on a computer. Every general-purpose computer must have an operating system to run other programs and applications. Computer operating systems perform basic tasks, such as recognizing input from the keyboard, sending output to the display screen, keeping track of files and directories on the storage drives, and controlling peripheral devices, such as printers.
   Classification of Operating systems
   - **Multi-user:** Allows two or more users to run programs at the same time. Some operating systems permit hundreds or even thousands of concurrent users.
   - **Multiprocessing :** Supports running a program on more than one CPU.
   - **Multitasking :** Allows more than one program to run concurrently.
   - **Multithreading :** Allows different parts of a single program to run concurrently.
   - **Real time:** Responds to input instantly. General-purpose operating systems, such as DOS and UNIX, are not real-time.

2. Explain the significance of the following commands.

i. ls – l **ii. Cat**

Ans) **i.** The command **ls** supports the **-l** option which would help you to get more information about the listed files −

```
$ls -l
total 1962188
drwxrwxr-x  2 amrood amrood      4096 Dec 25 09:59 uml
-rw-rw-r--  1 amrood amrood      5341 Dec 25 08:38 uml.jpg
drwxr-xr-x  2 amrood amrood      4096 Feb 15  2006 univ
drwxr-xr-x  2 root   root        4096 Dec  9  2007 urlspedia
drwxr-xr-x 11 amrood amrood      4096 May 29  2007 zlib-1.2.3
```

Here is the information about all the listed columns −

- First Column: represents file type and permission given on the file. Below is the description of all type of files.
- Second Column: represents the number of memory blocks taken by the file or directory.
- Third Column: represents owner of the file. This is the Unix user who created this file.
- Fourth Column: represents group of the owner. Every Unix user would have an associated group.
- Fifth Column: represents file size in bytes.
- Sixth Column: represents date and time when this file was created or modified last time.
- Seventh Column: represents file or directory name.

ii. Cat

cat COMMAND:
cat linux command concatenates files and print it on the standard output.

SYNTAX:
The Syntax is
cat [OPTIONS] [FILE]...

OPTIONS:

-A	Show all.
-b	Omits line numbers for blank space in the output.
-e	A $ character will be printed at the end of each line prior to a new line.
-E	Displays a $ (dollar sign) at the end of each line.
-n	Line numbers for all the output lines.
-s	If the output has multiple empty lines it replaces it with one empty line.
-T	Displays the tab characters in the output.
-v	Non-printing characters (with the exception of tabs, new-lines and form-

feeds) are printed visibly.

EXAMPLE:

1. To Create a new file:

cat > file1.txt

This command creates a new file file1.txt. After typing into the file press control+d (^d) simultaneously to end the file.

2. To Append data into the file:

cat >> file1.txt

To append data into the same file use append operator >> to write into the file, else the file will be overwritten (i.e., all of its contents will be erased).

3. To display a file:

cat file1.txt

This command displays the data in the file.

3. Define Shell and kernel.

Ans) SHELL:

The shell acts as an interface between the user and the kernel. When a user logs in, the login program checks the username and password, and then starts another program called the shell. The shell is a command line interpreter (CLI). It interprets the commands the user types in and arranges for them to be carried out.

Types of Different Shells

Name of shell	Command name	Description
C Shell	csh	Similar to the C programming language in syntax
Bash Shell	bash	Bourne Again Shell combines the advantages of the Korn Shell and the C Shell. The default on most Linux distributions.
tcsh	tcsh	Similar to the C Shell

Shell script is a program written in a shell programming language (e.g., bash, csh, ksh or sh) that allows users to issue a single command to execute any combination of commands, including those with options and/or arguments, together with redirection. Shell scripts are well suited for automating simple tasks and creating custom-made filters.

Shell Responsibilities:

- Command line interpreter
- Program Execution
- Meta characters
- Variable and Filename Substitution
- I/O Redirection
- Pipeline Hookup
- Environment Control
- Interpreted Programming Language

Kernel is also called as the heart of the Operating System and the Every Operation is performed by using the Kernel , When the Kernel Receives the Request from the Shell then this will Process the Request and Display the Results on the Screen. The various Types of Operations those are Performed by the Kernel are as followings:-

1) It Controls the State the Process Means it checks whether the Process is running or Process is waiting for the Request of the user.
2) Provides the Memory for the Processes those are Running on the System Means Kernel Runs the Allocation and De-allocation Process , First When we Request for the service then the Kernel will Provides the Memory to the Process and after that he also Release the Memory which is Given to a Process.
3) The Kernel also Maintains a Time table for all the Processes those are Running Means the Kernel also Prepare the Schedule Time means this will Provide the Time to various Process of the CPU and the Kernel also Puts the Waiting and Suspended Jobs into the different Memory Area.
4) When a Kernel determines that the Logical Memory doesn't fit to Store the Programs. Then he uses the Concept of the Physical Memory which Will Stores the Programs into Temporary Manner. Means the Physical Memory of the System can be used as Temporary Memory.

5) Kernel also maintains all the files those are stored into the Computer System and the Kernel Also Stores all the Files into the System as no one can read or Write the Files without any Permission. So that the Kernel System also Provides us the Facility to use the Passwords and also all the Files are Stored into the Particular Manner.

4. What is a filter in UNIX?

Ans) In Unix there are many Filter commands like- cat, awk, grep, head, tail cut etc.

A Filter is a software program that takes an input and produces an output, and it can be used in a stream operation.

E.g. cut -d : -f 2 /etc/passwd | grep abc

We can mix and match multiple filters to create a complex command that can solve a problem.

Awk and Sed are complex filters that provide fully programmable features.

5. Discuss the various backup utilities available in UNIX.

Ans) Backup restore and disk copy with tar :

– Backing up all files in a directory including subdirectories to a tape device (/dev/rmt/0),

tar cvf /dev/rmt/0 *

Viewing a tar backup on a tape

tar tvf /dev/rmt/0

Extracting tar backup from the tape

tar xvf /dev/rmt/0
(Restoration will go to present directory or original backup path depending on
relative or absolute path names used for backup)

Backup restore and disk copy with tar :

Back up all the files in current directory to tape .
find . -depth -print | cpio -ovcB > /dev/rmt/0
cpio expects a list of files and find command provides the list , cpio has
to put these file on some destination and a > sign redirect these files to tape . This can be a file as well .

Viewing cpio files on a tape

cpio -ivtB < /dev/rmt/0

Restoring a cpio backup

cpio -ivcB < /dev/rmt/0

Compress/uncompress files :

You may have to compress the files before or after the backup and it can be done with following commands .

Compressing a file

compress -v file_name
gzip filename

To uncompress a file

uncompress file_name.Z
or
gunzip filename

6. Which is the command used for ordering a file?

Ans) Sort command is helpful to sort/order lines in text files. You can sort the data in text file and display the output on the screen, or redirect it to a file. Based on your requirement, sort provides several command line options for sorting data in a text file.

Sort Command Syntax:

```
$ sort [-options]
```

For example, here is a test file:

```
$ cat test

zzz

sss

qqq

aaa

BBB

ddd

AAA
```

And, here is what you get when sort command is executed on this file without any option. It sorts lines in test file and displays sorted output.

```
$ sort test
```

```
aaa
AAA
BBB
ddd
qqq
sss
zzz
```

7. Explain the following UNIX shell commands with examples.

i. cut   ii. Paste   iii. Egrep   iv. fgrep.

Ans)   i. cut :

cut COMMAND:

cut command is used to cut out selected fields of each line of a file. The cut command uses delimiters to determine where to split fields.

SYNTAX:

The Syntax is

cut [options]

OPTIONS:

-c   Specifies character positions.

-b   Specifies byte positions.

-d flags   Specifies the delimiters and fields.

EXAMPLE:

1. cut -c1-3 text.txt

   Output:

   Thi

Cut the first three letters from the above line.

2. cut -d, -f1,2 text.txt

Output:

This is, an example program

The above command is used to split the fields using delimiter and cut the first two fields.

ii. Paste

paste command is used to paste the content from one file to another file. It is also used to set column format for each line.

SYNTAX:

The Syntax is

paste [options]

OPTIONS:

-s	Paste one file at a time instead of in parallel.
-d	Reuse characters from LIST instead of TABs .

EXAMPLE:

1. paste test.txt>test1.txt

   Paste the content from 'test.txt' file to 'test1.txt' file.

2. ls | paste - - - -

   List all files and directories in four columns for each line.

iii. Egrep

Search for a pattern using extended regular expressions. **egrep** is essentially the same as running **grep** with the **-E** option.
egrep syntax

```
egrep [options] PATTERN [FILE...]
```

Options

Option	Description
-A NUM, **--after-context**=NUM	Print NUM lines of trailing context after matching lines. Places a line containing **--** between contiguous groups of matches.
-a, **--text**	Process a binary file as if it were text; this is equivalent to the **--binary-files=text** option.
-B NUM, **--before-context**=NUM	Print NUM lines of leading context before matching lines. Places a line containing **--** between contiguous groups of matches.
-C NUM, **--context**=NUM	Print NUM lines of output context. Places a line containing **--** between contiguous groups of matches.
-b, **--byte-offset**	Print the byte offset within the input file before each line of output.
--binary-files=TYPE	If the first few bytes of a file indicate that the file contains binary data, assume that the file is of type TYPE. By default, TYPE is binary, and **grep**normally outputs either a one-line message saying that a binary file matches, or no message if there is no match. If TYPE is without-match, **grep** assumes that a binary file does not match; this is equivalent to the **-I** option. If TYPE is text, **grep** processes a binary file as if it were text; this is equivalent to the **-a** option. Warning: **grep --binary-files=text**might output binary garbage, which can have nasty side effects if the output is a terminal and if the terminal driver interprets some of it as commands.

iv. fgrep

fgrep searches for fixed-character strings in a file or files. "Fixed-character" means the string is interpreted literally — metacharacters do not exist, and therefore regular expressions cannot be used.

fgrep is useful when you need to search for strings which contain lots of regular expression metacharacters, such as "**$**", "**^**", etc. By specifying that your search string contains fixed characters, you don't need to escape each of them with a backslash.

If your string contains newlines, each line will be considered an individual fixed-character string to be matched in the search.

Running **fgrep** is the same as running **grep** with the **-F** option.

fgrep syntax

fgrep [-b] [-c] [-h] [-i] [-l] [-n] [-s] [-v] [-x] [-e pattern_list]

[-f pattern-file] [pattern] [file]

Options

-b	Precede each line by the block number on which it was found. This can be useful in locating block numbers by context (first block is 0).
-c	Print only a count of the lines that contain the pattern.
-h	Suppress printing of files when searching multiple files.
-i	Ignore upper/lower case distinction during comparisons.
-l	Print the names of files with matching lines once, separated by new-lines. Does not repeat the names of files when the pattern is found more than once.
-n	Precede each line by its line number in the file (first line is 1).
-s	Work silently, that is, display nothing except error messages. This is useful for checking the error status.

-v	Print all lines except those that contain the pattern.
-x	Print only lines matched entirely.
-e pattern_list	Search for a string in pattern-list (useful when the string begins with a "**-**").
-f pattern-file	Take the list of patterns from pattern-file.
pattern	Specify a pattern to be used during the search for input.
file	A path name of a file to be searched for the patterns. If no file operands are specified, the standard input will be used.

fgrep examples

fgrep "support" myfile.txt

8. Write a shell script to display first n numbers of Fibonacci series.

Ans)

```
clear
 echo "How many number of terms to be generated ?"
 read n
 x=0
 y=1
 i=2
 echo "Fibonacci Series up to $n terms :"
 echo "$x"
 echo "$y"
 while [ $i -lt $n ]
 do
    i=`expr $i + 1 `
    z=`expr $x + $y `
    echo "$z"
    x=$y
    y=$z
 done
```

9. Write short notes on shell script arguments.

Ans) Shell script arguments: positional parameters

- Positional parameters can be used to pass information to a script: directly, with parameters representing command-line arguments, or indirectly, using set and command substitution (see below), letting the parameters represent the output of a command
- Within the script, we refer to the first argument as $1, the second argument as $2, and so on
- $0 stands for the name of the command itself
- $* stands for all arguments from $1 on up
- These numbered arguments are referred to as positional parameters
- $# represents the number of arguments
- set is used to assign values to positional parameters
  Example:
  set `who am i`
  echo $1
  Note: previously assigned values for positional parameters will be erased!!

10. Explain briefly about shells available in Unix.

Ans) In Linux and Unix, a shell refers to a program that is used to interpret the typed commands the user sends to the operating system. The closest analogy in Windows is the DOS Command Prompt. However, unlike in Windows, Linux and Unix computers allow the user to choose what shell they would like to use.

Bourne Shell

The original Bourne shell is named after its developer at Bell Labs, Steve Bourne. It was the first shell used for the Unix operating system, and it has been largely surpassed in functionality by many of the more recent shells. However, all Unix and many Linux versions allow users to switch to the original Bourne Shell, known simply as "sh," if they choose to forgo features such as file name completion and command histories that later shells have added.

C Shell

The C shell, as its name might imply, was designed to allow users to write shell script programs using a syntax very similar to that of the C programming language. It is known as "csh."

TC Shell

TC shell is an expansion upon the C shell. It has all the same features, but adds the ability to use keystrokes from the Emacs word processor program to edit text on the command line. For example, users can press Esc-D to delete the rest of the highlighted word. It is also known as "tcsh."

Korn Shell

Korn Shell was also written by a developer at Bell Labs, David Korn. It attempts to merge the features of the C shell, TC shell and Bourne shell under one package. It also includes the ability for developers to create new shell commands as the need arises.

It is known as "ksh."

Bourne-Again Shell

The Bourne-Again shell is an updated version of the original Bourne shell that was created by the Free Software Foundation for its open source GNU project. For this reason, it is a widely used shell in the open source community.

Its syntax is similar to that used by the Bourne shell, however it incorporates some of the more advanced features found in the C, TC and Korn shells.

Among the added features that Bourne lacked are the ability to complete file names by pressing the TAB key, the ability to remember a history of recent commands and the ability to run multiple programs in the background at once.

11. How to remove duplicate lines from a file using sort?

Ans) The easiest way to remove duplicate files from a text file as by use sort and uniq commands. Consider a following text file:

$ cat file.txt

How can I remove duplicate lines from my text file?

Use sort and uniq commands

Use sort and uniq commands

How can I remove duplicate lines from my text file?

To remove all duplicate lines we first need to pipe the content of the file to sort. Once the content is sorted we use uniq command to remove duplicates:

$ sort file.txt | uniq

How can I remove duplicate lines from my text file?

use sort and uniq commands

12. What is the difference between the text editing provided by ‘ed’ & ‘vi’ text editors.

Ans)

- **Ed:** This is one of the oldest "editor" in the Unix world. It was created by Thomson as a "line editor". I have never really used ed, but, based on what I have read and based on a few brief sessions ... it has powerful regex features for such a seemingly simple editor. NOTE that this is not a full-screen editor like the following two
- **Vi:** This is one of the "stalwarts" of the Unix world. This is a full screen text editor. It also has a line editor companion called ex. Most people proficient with Unix know how to use vi (or a variant) and in fact use it quite regularly. This is a powerful text editor that can help you mangle large text/code files in unbelievable ways. Getting used to it is a bit tough, but once you learn it, you can't go back!

PART – B (Long Answer Questions)

Q. No **Questions**

UNIT – I

1. Explain how Unix operating system provides more security than any other.

Ans)

- User Accounts
- File Permissions
- Data Verification
- Encrypted Storage
- Secure Remote Access with OpenSSH
- Software Management
- Host Integrity Testing
- System Recovery
- Resource Allocation Controls
- Monitoring and Audit Facilities
- The System Firewall
- Application Isolation
- A Note on Viruses and Malware

An introduction to the security facilities of Open Source UNIX-like operating systems, focusing on Linux distributions.

User Accounts

Every UNIX-like system includes a root account, which is the only account that may directly carry out administrative functions. All of the other accounts on the system are unprivileged. This means these accounts have no rights beyond access to files marked with appropriate permissions, and the ability to launch network services.

Network Ports: Only the root account may launch network services that use port numbers lower than 1024. Any account may start network services that use higher port numbers.

Each user should have a single account on the system. Network services may also have their own separate accounts, in order to be able to access those files on the system that they require. Utilities enable authorized users to temporarily obtain root privileges when necessary, so that administrators may manage the system with their own user accounts.

Avoid Logging in as root: You do not need to log in with the root account in order to manage any aspect of your system. Use tools such as su or sudo when you need to carry out an administrative task that requires root privileges.

For convenience, accounts may be members of one or more groups. If a group is assigned access to a resource, then every member of the group automatically has that access.

User Private Groups: On many distributions, each account is automatically made the sole member of a group with the same name as the account. This enables you to easily limit access to particular files or directories, by associating them with a group that will only ever have one member.

The majority of UNIX-like systems use a Pluggable Authentication Modules (PAM) facility to manage access by users. For each login attempt or password change, the relevant service runs the configured PAM modules in sequence. Some modules support authentication sources, such as locally-stored files or LDAP directory services. Administrators may enable other modules that carry out setup tasks during the login process, or check login requests against particular criteria, such as a list of time periods when access is permitted.

File Permissions

Every file and directory on a UNIX-style system is marked with three sets of file permissions that determine how it may be accessed, and by whom:

- The permissions for the owner, the specific account that is responsible for the file
- The permissions for the group that may use the file
- The permissions that apply to all other accounts

Each set may have none or more of the following permissions on the item:

- read
- write
- execute

A user may only run a program file if they belong to a set that has the execute permission. For directories, the executepermission indicates that users in the relevant set may see the files within it, although they may not actually read, write or execute any file unless the permissions of that file permit it. Executable files with the setUID property automatically run with the privileges of the file owner, rather than the account that activates them. Avoid

setting the execute permission or setUID on any file or directory unless you specifically require it.

root Ignores File Permissions: The root account has full access to every file on the system, regardless of the permissions attached to that file.

The majority of files on a UNIX-like system are owned by the root account, and have permissions that restrict or block access from all other accounts. Avoid modifying the permissions on system files and directories.

Historically, user home directories on UNIX-like systems were publicly readable, to facilitate sharing between academic colleagues. Unfortunately, some popular operating systems still make user home directories publicly readable by default.

Access Control Lists: Many, but not all, modern UNIX-like systems include support for a more flexible set of permissions known as Access Control Lists (ACLs). Unfortunately, some common applications are not fully compatible with ACL permissions.

Data Verification

To create a checksum for a file, or to test a file against a checksum, use the sha1sum utility. SHA1 supersedes the older MD5 method, and you should always use SHA1. For more information about about sha1sum, refer to the manual:

man sha1sum

Open Source UNIX-like systems also supply the GNU Privacy Guard (GnuPG) system for encrypting and digitally signing files, such as emails. Many documents refer to GnuPG as gpg, which is the name of the main GnuPG command.

GnuPG Follows the OpenPGP Standard: The files that you sign or encrypt with GnuPG are compatible with other applications that follow the OpenPGP standard.

The Evolution email application automatically supports both signing and encrypting emails with GnuPG. Evolution is the default email application for several of the main Linux distributions, including Fedora, Novell Linux Desktop, Red Hat Enterprise Linux, and Ubuntu.

Encrypted Storage

Create one or more encrypted volumes for your sensitive files. Each volume is a single file which may enclose other files and directories of your choice. To access the contents of the volume, you must provides the correct decryption password to a utility, which then makes the volume available as if it were a directory or drive. The contents of an encrypted volume cannot be read when the volume is not mounted. You may store or copy encrypted volume files as you wish without affecting their security.

In extreme cases, you may decide to encrypt an entire disk partition that holds or caches data, so that none of the contents may be read by unauthorized persons. On Linux you may use either LUKS, CryptoFS, or EncFS to encrypt disks. Unfortunately, many UNIX-like systems do not yet integrate support for disk encryption facilities into their installation and management software, which makes configuration and maintenance more difficult. Disk encryption also reduces performance, and this may not be acceptable for systems that run demanding applications.

Secure Remote Access with OpenSSH

Every common UNIX-like system today includes a version of OpenSSH, an implementation of the SSH standard for secure remote access. An SSH service uses strong encryption by default, and provides the following facilities:

- Remote command-line access
- Remote command execution
- Remote access to graphical software
- File transfers

In addition, the forwarding features of SSH allow you to tunnel connections to other services through SSH. A tunneled service benefits from the same security and data compression features as the built-in facilities of SSH. This enables you to protect almost all communications between any UNIX-like systems, even when the traffic passes over open wireless networks or the public Internet.

SSH software not only encrypts the connection between systems, but also uses a system of keys to provide mutual authentication between each party. Each SSH client utility automatically checks the identity of any remote system that it connects to, by verifying the key. Similarly, users may identify themselves to systems with a key, rather than typing potentially crackable passwords.

Use SSH by Default: SSH potentially offers the most secure method of remote access available today. The standard Open Source desktop environments now also support SSH as a standard method for working with remote files. Only enable access to your systems through other services if you need to do so in order to meet a specific requirement.

Most Linux distributions include the OpenSSH client by default. The OpenSSH service as usually offered as an option, although some distributions also provide it by default. macOS includes both the OpenSSH service, and the client utilities. To access SSH services from Microsoft Windows systems, install PuTTY. You may download PuTTY from the main project Web site:

Software Management

The majority of Linux distributions incorporate software management facilities based on package files and sets of specially prepared Web sites, known as repositories or channels. Package management utilities construct or update working copies of software from these packages, and execute any other setup tasks that packages require. Repositories enable the management tools to automatically provide the correct version of each software product, by downloading the required packages directly from the relevant repository. Most systems also use checksums and digital signature tests to ensure that packages are authentic and correct.

In addition, package management tools can identify outdated versions of software by checking the software installed on a system against the packages in the repositories. This means that you can ensure that all of the supported software on your system does not suffer from a known security vulnerability, simply by running the update routine.

Most desktop systems now automatically alert you when new versions of the installed packages are released to the repositories, and provide options to update your system. Graphical interfaces to their software management tools also enable you to browse and select new software from the available packages.

Use packages from repositories whenever possible, in order to guarantee the provenance of the software on your system, and to ensure that it remains current. If you use software from elsewhere then you will need to verify, install, and update those products yourself. In these cases, download the software directly from the Web site of the manufacturer. Package management tools cannot inventory, check, or maintain any software that was compiled from source code, so you must be particularly careful when you use manually compiled products.

For historical reasons, the main Linux distributions use different package management products. Fedora, Red Hat Enterprise Linux, and related distributions, use the RPM package format, and their software management facility is known as YUM. Debian, Ubuntu, and their derivatives, use the APT management system and the DEB package format. These systems and package formats are largely equivalent.

Host Integrity Testing

To verify that a running system has not been compromised or tampered with, use an integrity testing facility. All host integrity testing software verifies a complete copy of a system by testing each file against a previously made checksum. Solaris and FreeBSD distributions both now include integrity testing utilities for this purpose. You may also use a cross-platform integrity monitoring system, such as Samhain or Osiris. Both Osiris and Samhain support centralized system auditing for multiple systems.

Since system configurations vary, administrators must configure the integrity tester to exclude the particular directories and files that are expected to change on a system, before creating an initial checksum database for that system. Integrity testing can then compare the checksums of each file against the database, and report on any disparity.

System Recovery

You may easily restore program files for all of the software that is included with your distribution with the software management tools. In order to fully recover a system from accident, or deliberate compromise, you must also have access to copies of the data, configuration, and log files. These files require some form of separate backup mechanism.

All effective backup systems provide the ability to restore versions of your files from several earlier points in time. You may discover that the current system is damaged or compromised at any time, and need to revert to previous versions of key files, so keeping only one additional copy of a key file should not be considered an adequate backup.

Duplicates Are Not Archives: File synchronization software and RAID storage make duplicate copies of the current files, and may act as a safeguard against data loss from hardware failures. Unlike backup systems, these measures do not provide access to previous versions of files.

Distributions provide a wide range of backup tools, and leave it to the administrator to configure a suitable backup arrangement for their systems.

Resource Allocation Controls

You may configure several mechanisms to limit the resources that an application or user account may consume. On systems with multiple users, enforce resource limits to ensure that no user may accidental or deliberately cause facilities to fail by using all of the available resources. Since the correct resource allocations vary widely, administrators must configure appropriate limits for the system.

To set resource limits for particular services, edit the systemd configuration file for the service. If you need to limit individual processes, add a ulimit setting to the shell script that launches them. For more information about about ulimit, refer to the manual for the bash shell:

info bash

The PAM login system includes a module to enforce certain resource limits for entire user sessions. The restrictions that this imposes may be circumvented, but they do provide some defence against accidental problems.

You must specifically enable storage quotas on each disk partition, if you require them. Quotas prevent users from overloading the storage and backup facilities, but quota management is often an administrative matter rather than a direct security concern. Configuring storage quotas is beyond the scope of this document.

Monitoring and Audit Facilities

On Linux systems, the syslog and klogd services record activity as it is reported by different parts of the system. The Linux kernel reports to klogd, whilst the other services and facilities

on the system send log messages to a syslog service. Distributions provide several tools for reading and analyzing the system log files.

Several facilities on any UNIX-like system may also email reports and notifications directly to the root account, via the SMTPservice. Edit the aliases file to redirect messages for root to another email address, and you will receive these emails at the specified address.

Automatic Log Summaries: Many distributions automatically send daily reports to the email address for root that summarize the activity logged by syslog and klogd.

To provide a central logging facility for your network, first select one of your systems as the log host. Configure the syslog services on your other systems to forward the information that they receive to the syslog service on the log host. You may then run analyzers on the log host to monitor events across the network.

For detailed real-time monitoring of the systems on your network, install SNMP agents that report to a management service such as Nagios or OpenNMS. SNMP is beyond the scope of this document.

Monitoring Network Appliances: Many network appliances, such as routers, support the syslog and SNMP standards. This enables you to monitor both UNIX systems and other network devices with the same log hosts and SNMP services.

Refer to the man page for basic information about syslogd:

man syslogd

Similarly, for more on klogd, refer to the man page:

man klogd

Both syslog and SNMP rely on software dispatching messages to a central service. If you configure process accounting on a system it maintains records of all the processes that are run on that system. Linux includes some support for process accounting, and distributions supply packages for GNU Accounting Utilities. Refer to the Web page for more information on the GNU Accounting Utilities:

http://www.gnu.org/software/acct/

Fedora and Red Hat Enterprise Linux systems also offer the LAuS (Linux Auditing System) framework. For more information on this, refer to the man pages for auditd:

man auditd

The System Firewall

The netfilter framework included in the Linux kernel restricts incoming and outgoing network connections according to a set of rules that have been defined by the administrator. Several Linux distributions configure firewall rules by default, and offer utilities for

managing simple firewall configurations. You may also manage the firewall rules on any Linux system with the standard iptables and ip6tables command-line utilities, or with third-party utilities such as Firestarter. If you decide to use iptables, remember that it only configures restrictions for IP version 4 connections, and that you will need to use ip6tables to setup rules for IP version 6 as well.

Fedora, Red Hat, and SUSE automatically enable the firewall and supply their own graphical configuration utilities. You must manually configure and enable the firewall on Debian and Ubuntu systems. Current releases of Ubuntu include a command-line utility called ufw for firewall configuration.

Those Linux distributions that enable a firewall by default use a netfilter configuration that blocks connections from other systems. Any attempt by a remote system to access a service on a blocked port simply fails. This means that no other system may connect to an installed service, unless you specifically choose to unblock the relevant port.

Use Only One Means Of Managing Your Firewall: Every firewall utility modifies the current firewall rules on the system. To ensure that your firewall operates correctly, select one method of managing the configuration, and avoid editing the firewall rules by other means.

Application Isolation

The most common UNIX-like operating systems provide several methods of limiting the ability of a program to affect either other running programs, or the host system itself.

- Mandatory Access Control (MAC) supplements the normal UNIX security facilities of a system by enforcing absolute limits that cannot be circumvented by any program or account.
- Virtualization enables you to assign a limited set of hardware resources to a virtual machine, which may be monitored and backed up by separate processes on the host system.
- Linux Container facilities, such as Docker, run processes within a generated filesystem and separate them from the normal processes of the host system
- The chroot utility runs programs within a specified working directory, and prevents them from accessing any other directory on that system.

The administrator may setup guest operating systems in virtual environments for specific tasks, and restrict these guests far more than would be possible for a multi-purpose system. Each specialized system may include far less software, and this also simplifies every administrative task, including MAC configuration. Neither MAC nor virtualization prevent individual applications or services from being compromised, misconfigured or malfunctioning, but may prevent a problem from escalating.

At the simplest level, the SELinux framework can provide MAC facilities, to enforce a policy that defines the access permitted to programs or accounts on the system. SELinux was actually created by the NSA to meet the needs of government agencies handling classified data, and enables administrators to develop extremely detailed and precise security configurations that encompass the entire operating system. Many developers and administrators consider SELinux too high a maintenance burden to implement fully.

Fedora and Red Hat Enterprise Linux systems automatically include a limited SELinux policy that restricts many standard network services, without affecting users or other programs. These distributions also provide some simple management tools for customizing the default policy and troubleshooting SELinux issues, but no tools to assist with developing new policies. Debian provides SELinux, but support is limited.

Ubuntu and SUSE do not enable SELinux by default. Instead, they provide the AppArmor facility. AppArmor configuration is much simpler than SELinux, but it offers more limited capabilities.

Several Open Source solutions exist to run complete operating systems within a virtual environment. By far the most popular are are Xen and KVM. Xen enables you to configure a system to act as a host for multiple virtual environments, all of which are controlled by a single hypervisor. Current Linux distributions on machines that include CPUs with virtualization support may run the simpler and more flexible KVM. The current KVM offers significantly higher performance than the QEMU machine emulator that is it based upon. The original QEMU software operates too slowly for production applications, although it remains useful for testing and development work.

Modern Linux systems include support for containers, and provide tools that enable you to easily use this facility. Docker relies on a background service that manages the containers on the host system, and this can support a large number of containers on a single host. The systemd-nspawn utility that is supplied with systemd runs individual containers without requiring an extra service.

Containers Do Not Isolate Processes: By default, any process within a container still has access to the facilities of the host system, such as networking, even though it does not have access to most of the filesystem. Any process that runs as root in a container can alter the host system.

The older chroot facility is universally available, but was originally designed for development tasks rather than security, and may be circumvented. Developers use this facility for building and testing software in a clean environment. Historically, administrators also used chroot to run potentially unsafe network services such as FTP servers within specially designed environments. Several tools exist to simplify constructing and maintaining chroot environments.

Applications May Escape chroot: Any application that is able to run arbitrary commands can execute code to gain access to the main system. To ensure the security of the chroot

environment, avoid including shells, compilers, or script interpreters within the chroot directory. Any application that runs with root privileges may also escape the restrictions of chroot.

For more information about chroot, refer to the manual:

info chroot

A Note on Viruses and Malware

The security features of UNIX-like systems described above combine to form a strong defense against malware:

- Software is often supplied in the form of packages, rather than programs
- If you download a working program, it cannot run until you choose to mark the files as executable
- By default, applications such as the OpenOffice.org suite and the Evolution email client do not run programs embedded in emails or documents
- Web browsers require you to approve the installation of plug-ins
- Software vulnerabilities can be rapidly closed by vendors supplying updated packages to the repositories

Although a virus could be written for use against current UNIX-like systems, no effective malware is known to exist. It is likely that any future malware would need the consent of a user on the system in order to install itself, significantly reducing the possibility that any such software would be able to spread across networks.

2. Define Shell and its Responsibilities.

Ans) **SHELL:**

The **shell** acts as an interface between the user and the **kernel**. When a user logs in, the login program checks the username and password, and then starts another program called the **shell**. The **shell** is a command line interpreter (CLI). It interprets the commands the user types in and arranges for them to be carried out.

Types of Different Shells

Name of shell	**Command name**	**Description**
C Shell	csh	Similar to the C programming language in syntax
Bash Shell	bash	Bourne Again Shell combines the advantages of the Korn Shell and the C Shell. The default on most

		Linux distributions.
tcsh	tcsh	Similar to the C Shell

Shell script is a program written in a shell programming language (e.g., bash, csh, ksh or sh) that allows users to issue a single command to execute any combination of commands, including those with options and/or arguments, together with redirection. Shell scripts are well suited for automating simple tasks and creating custom-made filters.

Shell Responsibilities:

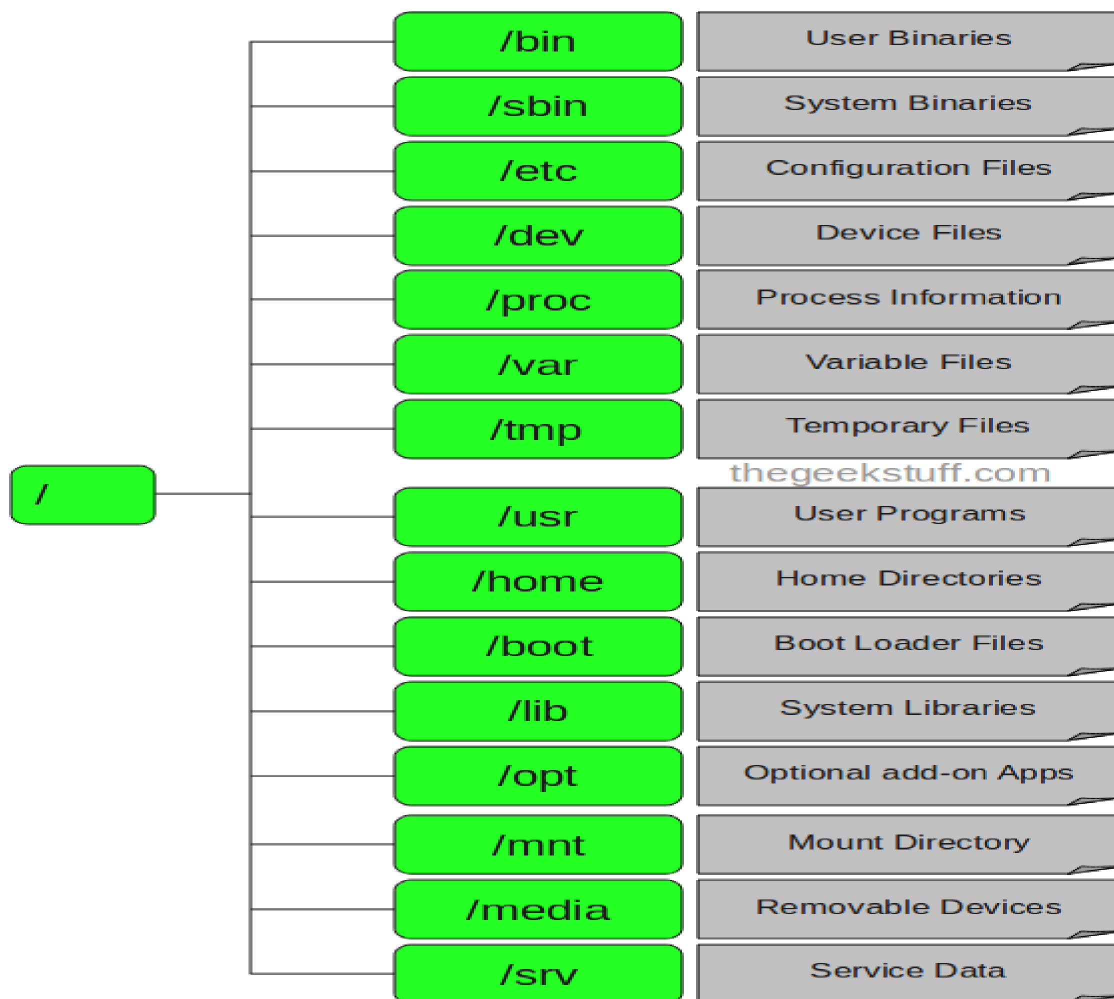
- Command line interpreter
- Program Execution
- Meta characters
- Variable and Filename Substitution
- I/O Redirection
- Pipeline Hookup
- Environment Control
- Interpreted Programming Language

3. Describe the characteristics of Unix File System.

Ans)

LINUX File system

Linux file structure files are grouped according to purpose. Ex: commands, data files, documentation. Parts of a Unix directory tree are listed below. All directories are grouped under the root entry "/". That part of the directory tree is left out of the below diagram.



1. / – Root

- Every single file and directory starts from the root directory.
- Only root user has write privilege under this directory.
- Please note that /root is root user's home directory, which is not same as /.

2. /bin – User Binaries
   - Contains binary executables.
   - Common linux commands you need to use in single-user modes are located under this directory.
   - Commands used by all the users of the system are located here.
   - For example: ps, ls, ping, grep, cp.

3. /sbin – System Binaries
   - Just like /bin, /sbin also contains binary executables.
   - But, the linux commands located under this directory are used typically by system aministrator, for system maintenance purpose.
   - For example: iptables, reboot, fdisk, ifconfig, swapon

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

5. /dev – Device Files
   - Contains device files.
   - These include terminal devices, usb, or any device attached to the system.
   - For example: /dev/tty1, /dev/usbmon0

6. /proc – Process Information
   - Contains information about system process.
   - This is a pseudo filesystem contains information about running process. For example: /proc/{pid} directory contains information about the process with that particular pid.
   - This is a virtual filesystem with text information about system resources. For example: /proc/uptime

7. /var – Variable Files
   - var stands for variable files.
   - Content of the files that are expected to grow can be found under this directory.
   - This includes — system log files (/var/log); packages and database files (/var/lib); emails (/var/mail); print queues (/var/spool); lock files (/var/lock); temp files needed across reboots (/var/tmp);

8. /tmp – Temporary Files
   - Directory that contains temporary files created by system and users.
   - Files under this directory are deleted when system is rebooted.

9. /usr – User Programs
   - 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

10. /home – Home Directories
    - Home directories for all users to store their personal files.
    - For example: /home/john, /home/nikita

11. /boot – Boot Loader Files
    - Contains boot loader related files.
    - Kernel initrd, vmlinux, grub files are located under /boot
    - For example: initrd.img-2.6.32-24-generic, vmlinuz-2.6.32-24-generic

12. /lib – System Libraries
    - Contains library files that supports the binaries located under /bin and /sbin
    - Library filenames are either ld* or lib*.so.*
    - For example: ld-2.11.1.so, libncurses.so.5.7

13. /opt – Optional add-on Applications
    - opt stands for optional.
    - Contains add-on applications from individual vendors.
    - add-on applications should be installed under either /opt/ or /opt/ sub-directory.

14. /mnt – Mount Directory
    - Temporary mount directory where sysadmins can mount filesystems.

15. /media – Removable Media Devices
    - Temporary mount directory for removable devices.
    - For examples, /media/cdrom for CD-ROM; /media/floppy for floppy drives; /media/cdrecorder for CD writer

16. /srv – Service Data
    - srv stands for service.
    - Contains server specific services related data.
    - For example, /srv/cvs contains CVS related data

4. What at are file handling utilities and processing utilities?

Ans)

File handling Utilities: ls,cat,rm,more,mv,cd,cp,touch,wc

Listing Files

To list the files and directories stored in the current directory. Use the following command −

```
$ls
```

Here is the sample output of the above command −

```
$ls


bin        hosts  lib     res.03
ch07       hw1    pub     test_results
ch07.bak   hw2    res.01  users
docs       hw3    res.02  work
```

The command **ls** supports the **-l** option which would help you to get more information about the listed files −

```
$ls -l
total 1962188
drwxrwxr-x  2 amrood amrood      4096 Dec 25 09:59 uml
-rw-rw-r--  1 amrood amrood      5341 Dec 25 08:38 uml.jpg
drwxr-xr-x  2 amrood amrood      4096 Feb 15  2006 univ
drwxr-xr-x  2 root   root        4096 Dec  9  2007 urlspedia
drwxr-xr-x 11 amrood amrood      4096 May 29  2007 zlib-1.2.3
```

Here is the information about all the listed columns −

- First Column: represents file type and permission given on the file. Below is the description of all type of files.

- Second Column: represents the number of memory blocks taken by the file or directory.
- Third Column: represents owner of the file. This is the Unix user who created this file.
- Fourth Column: represents group of the owner. Every Unix user would have an associated group.
- Fifth Column: represents file size in bytes.
- Sixth Column: represents date and time when this file was created or modified last time.
- Seventh Column: represents file or directory name.

In the ls -l listing example, every file line began with a d, -, or l. These characters indicate the type of file that's listed.

Prefix	Description
-	Regular file, such as an ASCII text file, binary executable, or hard link.
b	Block special file. Block input/output device file such as a physical hard drive.
c	Character special file. Raw input/output device file such as a physical hard drive
d	Directory file that contains a listing of other files and directories.
l	Symbolic link file. Links on any regular file.
p	Named pipe. A mechanism for interprocess communications

s	Socket used for interprocess communication.

Meta Characters

Meta characters have special meaning in Unix. For example ***** and **?** are metacharacters. We use ***** to match 0 or more characters, a question mark **?**matches with single character.

For Example −

```
$ls ch*.doc
```

Displays all the files whose name start with ch and ends with .doc −

```
ch01-1.doc   ch010.doc  ch02.doc    ch03-2.doc
ch04-1.doc   ch040.doc  ch05.doc    ch06-2.doc
ch01-2.doc ch02-1.doc c
```

Here ***** works as meta character which matches with any character. If you want to display all the files ending with just **.doc** then you can use following command −

```
$ls *.doc
```

Hidden Files

An invisible file is one whose first character is the dot or period character (.). UNIX programs (including the shell) use most of these files to store configuration information.

Some common examples of hidden files include the files −

- **.profile** − the Bourne shell (sh) initialization script
- **.kshrc** − the Korn shell (ksh) initialization script
- **.cshrc** − the C shell (csh) initialization script
- **.rhosts** − the remote shell configuration file

To list invisible files, specify the -a option to ls −

```
$ ls -a

.       .profile      docs     lib     test_results
..      .rhosts       hosts    pub     users
.emacs    bin          hw1      res.01  work
.exrc    ch07          hw2      res.02
.kshrc    ch07.bak      hw3      res.03
$
```

- Single dot **.** − This represents current directory.
- Double dot **..** − This represents parent directory.

Creating Files

You can use **vi** editor to create ordinary files on any Unix system. You simply need to give following command −

```
$ vi filename
```

Above command would open a file with the given filename. You would need to press key **i** to come into edit mode. Once you are in edit mode you can start writing your content in the file as below −

```
This is unix file....I created it for the first time.....
I'm going to save this content in this file.
```

Once you are done, do the following steps −

- Press key **esc** to come out of edit mode.
- Press two keys **Shift** + **ZZ** together to come out of the file completely.

Now you would have a file created with **filename** in the current directory.

```
$ vi filename
$
```

Editing Files

You can edit an existing file using **vi** editor. We would cover this in detail in a separate tutorial. But in short, you can open existing file as follows −

```
$ vi filename
```

Once file is opened, you can come in edit mode by pressing key **i** and then you can edit file as you like. If you want to move here and there inside a file then first you need to come out of edit mode by pressing key **esc** and then you can use following keys to move inside a file −

- **l** key to move to the right side.
- **h** key to move to the left side.
- **k** key to move up side in the file.
- **j** key to move down side in the file.

So using above keys you can position your cursor where ever you want to edit. Once you are positioned then you can use **i** key to come in edit mode. Edit the file, once you are done press **esc** and finally two keys **Shift** + **ZZ** together to come out of the file completely.

Display Content of a File

You can use **cat** command to see the content of a file. Following is the simple example to see the content of above created file −

```
$ cat filename
This is unix file....I created it for the first time.....
I'm going to save this content in this file.
$
```

You can display line numbers by using **-b** option along with **cat** command as follows −

```
$ cat -b filename
1   This is unix file....I created it for the first time.....
2   I'm going to save this content in this file.
```

```
$
```

Counting Words in a File

You can use the **wc** command to get a count of the total number of lines, words, and characters contained in a file. Following is the simple example to see the information about above created file –

```
$ wc filename
2  19 103 filename
$
```

Here is the detail of all the four columns –

- First Column: represents total number of lines in the file.
- Second Column: represents total number of words in the file.
- Third Column: represents total number of bytes in the file. This is actual size of the file.
- Fourth Column: represents file name.

You can give multiple files at a time to get the information about those file. Here is simple syntax –

```
$ wc filename1 filename2 filename3
```

Copying Files:

To make a copy of a file use the **cp** command. The basic syntax of the command is –

```
$ cp source_file destination_file
```

Following is the example to create a copy of existing file **filename**.

```
$ cp filename copyfile
$
```

Now you would find one more file **copyfile** in your current directory. This file would be exactly same as original file **filename**.

Renaming Files

To change the name of a file use the **mv** command. Its basic syntax is −

```
$ mv old_file new_file
```

Following is the example which would rename existing file **filename** to**newfile**:

```
$ mv filename newfile
$
```

The **mv** command would move existing file completely into new file. So in this case you would fine only **newfile** in your current directory.

Deleting Files

To delete an existing file use the **rm** command. Its basic syntax is −

```
$ rm filename
```

Caution: It may be dangerous to delete a file because it may contain useful information. So be careful while using this command. It is recommended to use**-i** option along with **rm** command.

Following is the example which would completely remove existing file**filename**:

```
$ rm filename
$
```

You can remove multiple files at a tile as follows −

```
$ rm filename1 filename2 filename3
$
```

Standard Unix Streams

Under normal circumstances every Unix program has three streams (files) opened for it when it starts up −

- **stdin** − This is referred to as standard input and associated file descriptor is 0. This is also represented as STDIN. Unix program would read default input from STDIN.
- **stdout** − This is referred to as standard output and associated file descriptor is 1. This is also represented as STDOUT. Unix program would write default output at STDOUT
- **stderr** − This is referred to as standard error and associated file descriptor is 2. This is also represented as STDERR. Unix program would write all the error message at STDERR.

Process Utilities:

operating system tracks processes through a five digit ID number known as the **pid** or process ID . Each process in the system has a unique pid.

Pids eventually repeat because all the possible numbers are used up and the next pid rolls or starts over. At any one time, no two processes with the same pid exist in the system because it is the pid that UNIX uses to track each process.

Starting a Process

When you start a process (run a command), there are two ways you can run it −

- Foreground Processes
- Background Processes

Foreground Processes

By default, every process that you start runs in the foreground. It gets its input from the keyboard and sends its output to the screen.

You can see this happen with the ls command. If I want to list all the files in my current directory, I can use the following command −

```
$ls ch*.doc
```

This would display all the files whose name start with ch and ends with .doc −

```
ch01-1.doc   ch010.doc  ch02.doc    ch03-2.doc
ch04-1.doc   ch040.doc  ch05.doc    ch06-2.doc
ch01-2.doc   ch02-1.doc
```

The process runs in the foreground, the output is directed to my screen, and if the ls command wants any input (which it does not), it waits for it from the keyboard.

While a program is running in foreground and taking much time, we cannot run any other commands (start any other processes) because prompt would not be available until program finishes its processing and comes out.

Background Processes

A background process runs without being connected to your keyboard. If the background process requires any keyboard input, it waits.

The advantage of running a process in the background is that you can run other commands; you do not have to wait until it completes to start another!

The simplest way to start a background process is to add an ampersand (&) at the end of the command.

```
$ls ch*.doc &
```

This would also display all the files whose name start with ch and ends with .doc −

```
ch01-1.doc   ch010.doc  ch02.doc    ch03-2.doc
ch04-1.doc   ch040.doc  ch05.doc    ch06-2.doc
ch01-2.doc   ch02-1.doc
```

Here if the **ls** command wants any input (which it does not), it goes into a stop state until I move it into the foreground and give it the data from the keyboard.

That first line contains information about the background process - the job number and process ID. You need to know the job number to manipulate it between background and foreground.

If you press the Enter key now, you see the following −

```
[1]   +   Done                ls ch*.doc &
$
```

The first line tells you that the **ls** command background process finishes successfully. The second is a prompt for another command.

Listing Running Processes

It is easy to see your own processes by running the **ps** (process status) command as follows −

```
$ps
PID       TTY       TIME        CMD
18358     ttyp3     00:00:00    sh
18361     ttyp3     00:01:31    abiword
18789     ttyp3     00:00:00    ps
```

One of the most commonly used flags for ps is the **-f** (f for full) option, which provides more information as shown in the following example −

```
$ps -f
UID       PID  PPID C STIME    TTY   TIME CMD
amrood   6738 3662 0 10:23:03 pts/6 0:00 first_one
amrood   6739 3662 0 10:22:54 pts/6 0:00 second_one
amrood   3662 3657 0 08:10:53 pts/6 0:00 -ksh
amrood   6892 3662 4 10:51:50 pts/6 0:00 ps -f
```

Here is the description of all the fields displayed by ps -f command −

Column	**Description**

UID	User ID that this process belongs to (the person running it).
PID	Process ID.
PPID	Parent process ID (the ID of the process that started it).
C	CPU utilization of process.
STIME	Process start time.
TTY	Terminal type associated with the process
TIME	CPU time taken by the process.
CMD	The command that started this process.

There are other options which can be used along with **ps** command −

Option	Description
-a	Shows information about all users
-x	Shows information about processes without terminals.
-u	Shows additional information like -f option.
-e	Display extended information.

Stopping Processes

Ending a process can be done in several different ways. Often, from a console-based command, sending a CTRL + C keystroke (the default interrupt character) will exit the command. This works when process is running in foreground mode.

If a process is running in background mode then first you would need to get its Job ID using **ps** command and after that you can use **kill** command to kill the process as follows −

```
$ps -f
UID      PID  PPID C STIME    TTY   TIME CMD
amrood   6738 3662 0 10:23:03 pts/6 0:00 first_one
amrood   6739 3662 0 10:22:54 pts/6 0:00 second_one
amrood   3662 3657 0 08:10:53 pts/6 0:00 -ksh
amrood   6892 3662 4 10:51:50 pts/6 0:00 ps -f
$kill 6738
Terminated
```

Here **kill** command would terminate first_one process. If a process ignores a regular kill command, you can use **kill -9** followed by the process ID as follows −

```
$kill -9 6738
Terminated
```

Parent and Child Processes

Each unix process has two ID numbers assigned to it: Process ID (pid) and Parent process ID (ppid). Each user process in the system has a parent process.

Most of the commands that you run have the shell as their parent. Check ps -f example where this command listed both process ID and parent process ID.

Zombie and Orphan Processes

Normally, when a child process is killed, the parent process is told via a SIGCHLD signal. Then the parent can do some other task or restart a new child as needed. However, sometimes the parent process is killed before its child is killed. In this case, the "parent of all

processes," **init** process, becomes the new PPID (parent process ID). Sometime these processes are called orphan process.

When a process is killed, a ps listing may still show the process with a Z state. This is a zombie, or defunct, process. The process is dead and not being used. These processes are different from orphan processes.They are the processes that has completed execution but still has an entry in the process table.

Daemon Processes

Daemons are system-related background processes that often run with the permissions of root and services requests from other processes.

A daemon process has no controlling terminal. It cannot open /dev/tty. If you do a "ps -ef" and look at the tty field, all daemons will have a ? for the tty.

More clearly, a daemon is just a process that runs in the background, usually waiting for something to happen that it is capable of working with, like a printer daemon is waiting for print commands.

If you have a program which needs to do long processing then its worth to make it a daemon and run it in background.

The top Command

The **top** command is a very useful tool for quickly showing processes sorted by various criteria.

It is an interactive diagnostic tool that updates frequently and shows information about physical and virtual memory, CPU usage, load averages, and your busy processes.

Here is simple syntax to run top command and to see the statistics of CPU utilization by different processes −

```
$top
```

Job ID Versus Process ID

Background and suspended processes are usually manipulated via job number (job ID). This number is different from the process ID and is used because it is shorter.

In addition, a job can consist of multiple processes running in series or at the same time, in parallel, so using the job ID is easier than tracking the individual processes.

5. What are the file permissions for providing security to the files?

Ans) Security related utilities: chmod,chown,

File ownership is an important component of UNIX that provides a secure method for storing files. Every file in UNIX has the following attributes −

- Owner permissions − The owner's permissions determine what actions the owner of the file can perform on the file.
- Group permissions − The group's permissions determine what actions a user, who is a member of the group that a file belongs to, can perform on the file.
- Other (world) permissions − The permissions for others indicate what action all other users can perform on the file.

The Permission Indicators

While using ls -l command it displays various information related to file permission as follows −

```
$ls -l /home/amrood
-rwxr-xr--  1 amrood   users 1024  Nov 2 00:10  myfile
drwxr-xr--- 1 amrood   users 1024  Nov 2 00:10  mydir
```

Here first column represents different access mode ie. permission associated with a file or directory.

The permissions are broken into groups of threes, and each position in the group denotes a specific permission, in this order: read (r), write (w), execute (x) −

- The first three characters (2-4) represent the permissions for the file's owner. For example -rwxr-xr-- represents that owner has read (r), write (w) and execute (x) permission.
- The second group of three characters (5-7) consists of the permissions for the group to which the file belongs. For example -rwxr-xr--represents that group has read (r) and execute (x) permission but no write permission.
- The last group of three characters (8-10) represents the permissions for everyone else. For example -rwxr-xr-- represents that other world has read (r) only permission.

File Access Modes

The permissions of a file are the first line of defense in the security of a Unix system. The basic building blocks of Unix permissions are the read, write, andexecute permissions, which are described below −

1. Read

Grants the capability to read ie. view the contents of the file.

2. Write

Grants the capability to modify, or remove the content of the file.

3. Execute

User with execute permissions can run a file as a program.

Directory Access Modes

Directory access modes are listed and organized in the same manner as any other file. There are a few differences that need to be mentioned:

1. Read

Access to a directory means that the user can read the contents. The user can look at the filenames inside the directory.

2. Write

Access means that the user can add or delete files to the contents of the directory.

3. Execute

Executing a directory doesn't really make a lot of sense so think of this as a traverse permission.

A user must have execute access to the bin directory in order to execute ls or cd command.

Changing Permissions

To change file or directory permissions, you use the chmod (change mode) command. There are two ways to use chmod: symbolic mode and absolute mode.

Using chmod in Symbolic Mode

The easiest way for a beginner to modify file or directory permissions is to use the symbolic mode. With symbolic permissions you can add, delete, or specify the permission set you want by using the operators in the following table.

Chmod operator	Description
+	Adds the designated permission(s) to a file or directory.
-	Removes the designated permission(s) from a file or directory.
=	Sets the designated permission(s).

Here's an example using testfile. Running ls -1 on testfile shows that the file's permissions are as follows −

```
$ls -l testfile
-rwxrwxr--  1 amrood   users 1024  Nov 2 00:10  testfile
```

Then each example chmod command from the preceding table is run on testfile, followed by ls -l so you can see the permission changes −

```
$chmod o+wx testfile
$ls -l testfile
-rwxrwxrwx  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod u-x testfile
$ls -l testfile
-rw-rwxrwx  1 amrood   users 1024  Nov 2 00:10  testfile
$chmod g=rx testfile
$ls -l testfile
-rw-r-xrwx  1 amrood   users 1024  Nov 2 00:10  testfile
```

Here's how you could combine these commands on a single line:

```
$chmod o+wx,u-x,g=rx testfile
$ls -l testfile
-rw-r-xrwx  1 amrood   users 1024  Nov 2 00:10  testfile
```

Using chmod with Absolute Permissions

The second way to modify permissions with the chmod command is to use a number to specify each set of permissions for the file.

Each permission is assigned a value, as the following table shows, and the total of each set of permissions provides a number for that set.

Number	Octal Permission Representation	Ref
0	No permission	---
1	Execute permission	--x
2	Write permission	-w-

3	Execute and write permission: 1 (execute) + 2 (write) = 3	-wx
4	Read permission	r--
5	Read and execute permission: 4 (read) + 1 (execute) = 5	r-x
6	Read and write permission: 4 (read) + 2 (write) = 6	rw-
7	All permissions: 4 (read) + 2 (write) + 1 (execute) = 7	rwx

changing Owners and Groups

While creating an account on Unix, it assigns a owner ID and a group ID to each user. All the permissions mentioned above are also assigned based on Owner and Groups.

Two commands are available to change the owner and the group of files −

- chown − The chown command stands for "change owner" and is used to change the owner of a file.
- chgrp − The chgrp command stands for "change group" and is used to change the group of a file.

Changing Ownership

The chown command changes the ownership of a file. The basic syntax is as follows −

```
$ chown user filelist
```

The value of user can be either the name of a user on the system or the user id (uid) of a user on the system.

Following example −

```
$ chown amrood testfile
```

6. Explain text processing utilities and Network commands?

Ans)

Text Processing Utilities:

Cat:

The **cat** command reads one or more files and prints them to standard output. The operator > can be used to combine multiple files into one. The operator >> can be used to append to an existing file.
The syntax for the **cat** command is:
cat [options] [files]

option:

-e	$ is printed at the end of each line. This option must be used with -v.

Examples:

1. $cat file1 // displays the contents of file1
2. $cat file1 file2 > file3 //concatenates file1 and file2, and writes the results in file3
3. $cat file1 >> file2 //appends a copy of file1 to the end of file2

Tail:

The **tail** command displays the last ten lines of the file.
The syntax for the **tail** command is:
tail [options] [file]

options:

-f	Follow the file as it grows.
-r	Displays the lines in the reverse order.
-n[k]	Displays the file at the nth item from the end of the file.
+n[k]	Displays the file at the nth item from the beginning of the file.

Examples:

1. By default, tail will print the last 10 lines of its input to the standard output. With command line options the number of lines printed and the printing units (lines, blocks or bytes) may be changed. The following example shows the last 20 lines of filename:

$tail -n 20 filename

2. This example shows all lines of filename from the second line onwards:

$tail -n +2 filename

Head:

The head command displays the first ten lines of a file.
The syntax for head command is:

$head [options] <filename>

By default, head will print the first 10 lines of its input to the standard output. The number of lines printed may be changed with a command line option. The following example shows the first 20 lines of filename:

$head -n 20 filename

This displays the first 5 lines of all files starting with foo:

$head -n 5 foo*

Sort : Sorting is the ordering of data in ascending or descending sequence. The sort command orders a file. By default sort reorders lines in ASCII collating sequence_ white space first, then numerals, uppercase letters and finally lowercase letters.

1.Sort by lines: The easiest sort arranges data by lines. Starting at the beginning of the line, it compares the first character in one line with the first character in another line.

Syntax: $ sort filename

2.Sort by fields: in general a field is the smallest unit of data. When a field sort is required we need to define which fields are to be used for the sort.

Syntax: $ sort –t ” “ –k2 filename

Here we use –k option to sort on the specified field. The delimiter option (-t) specifies an alternate delimiter.

3.Reverse order: To order the data from largest to smallest,we specify reverse order(-r).

Syntax: $ sort –t : -k2.3 -r filename

Here it sorting the data based on 3rd character of the second field in descending order.

options:

-tchar	-	Use delimiter char to identify fields
-k n	-	Sorts on nth field
-k m,n	-	Starts sort on mth field and ends sort on nth Field
-k m.n	-	Starts sort on nth column of mth field
-u	-	Removes repeated lines
-n	-	Sorts numerically
-r	-	Reverses sort order

Nl:

Nl Is A Unix Utility For Numbering Lines, Either From A File Or From Standard Input, Reproducing Output On Standard Output.

options:

· s – separator- number and data is separated with separator

· w - width

Eg: $nl filename

$nl –s: file1

$nl –w20 –s: file1

uniq:

This command displays uniq lines of the given files.That is, if successive lines of a file are same then they will be removed. This can be used to remove successive empty lines in a given file.

The syntax of usage of this command is given as:

$uniq[OPTIONS] INPUT [OUTPUT]

options:

-c Prefix lines by their occurrences

-d Only print repeated lines

-D Print all duplicate lines

-u Only print unique lines

Example:

$ uniq testfile (or uniq –u testfile)

$ sort f1 | uniq

Grep: (Global Regular Expression) Searching for a pattern

UNIX has a special family of commands for handling search requirements, and the principal member of this family is the grep command. grep scans its input for a pattern and displays lines conainning the pattern, the linenumbers or filenames where the pattern occurs.

Syntax: grep options pttern filename(s)

Grep searches for pattern in one or more files ,or the standard input if no filename is specified. The first argument is the pattern and the remainning arguments are filenames.

Eg: $ grep "sales" employee

It displays the lines containing the string sales from the file employee.

Grep is also used with multiple filenames.It displays the filenames along with the ouput.

Eg: $ grep director emp1 emp2

$ grep "sales director" emp1 emp2

Here quoting is not necessary.when we use pattern with multiple words then we have to use quoting.

options:

option		significance
-I	-	Ignores case for matching

-v	-	Doesn't display lines matching expression
-n	-	Displays line numbers along with lines
-c	-	Display count of number of occurrences
-l	-	Displays list of filenames only

The fgrep and egrep command:

The fgrep and egrep command are advanced pattern matching command. The fgrep command doesn't use any meta character for its searched pattern. The primary advantage of fgrep is it can also serch two or more than two strings simultaneously. The fgrep command can be used like this:

$fgrep 'good

bad

great' userfile

Here a single quote is used to mark three strings as one argument. Here we are going to search three different strings good, bad, and great. The egrep command is used to search this in a more compact form than fgrep command:

$egrep 'good | bad | great' userfile

The egrep uses an or (|) operator to achieve this. Therefore egrep command is more compact and more versatile than fgrep. Another achievement of egrep is that we can make groups of different patterns. If we use | as operator.

$egrep 'sunil | rohan gavasker ' players

Here sunil is first pattern and everything to the right is considered as second pattern.
If we want to search both sunil gavasker and rohan gavasker use this:

$egrep '(sunil | rohan)' players

$egrep -f pat.lst file1 –this option is for using files instead of directly specifying different patterns.

$fgrep -f pat.lst file1

Cut command:

By using this command, we can extract the required columns or fields from the file.

This command extracts the fields based on either character position or on field delimeters position.

Options:

-c - Used to extracts the fields based on character position or columns.

-f - Used to extracts the required fields based on field delimeters.

(By default field delimeter is **tab**)

-d - Used to define our own delimeters.

Eg:

cut –f1,3 filename

This displays 1st and 3rd words of each line of the given file.Between word to word TAB

should available.

cut –d"**:**" -f1,3 filename

This displays user name ,UID of each legal user of the machine.Herewith –d option

we are specifying that **:** is the field separator between word to word.

The same result is given by the following command cut -d"**:**" -f3,1 /etc/passwd.

cut -d"**:**" -f1-3 filename

This displays 1st word to 3rd word from each line of the given file.

cut -f "**:**" –f3- filename

This displays 3rd word to till last word of each line of the given file.

cut -f “**:**” –f-3 filename

This displays 1st word to till 3rd word of each line of the given file.

cut -c3-5 filename

This displays 3rd character to 5th character of each line in the given file.

Paste command:

This command is used to create new files by gluying together fields or columns

From two or more files.

Syntax: paste filename1 filename2

Consider the following eample

$cat indo.lst

20032

20034

20121

$cat name.lst

H.D Rao

M.G.V Murthy

P.K.Krishna

$paste indo.lst name.lst >info.lst

The result of paste command is

$cat info.lst

20032 H.D Rao

20034 M.G.V Murthy

20121 P.K.Krishna

$paste –d : indo.lst name.lst

This command combines the data in the files by using : symbol in between the fields of two files.

Awk command:

This command made a late entry into the UNIX system in 1977 to augment the tool kit with suitable report formatting capabilities. The awk name is from authors Aho, Weinberger and Kernighan.

Syntax:

Awk options 'selection criteria {action}' file(s)

Examples:

$ awk –F" " '$3 > 100 { print }' file1 (or)

$ awk –F" " '$3 > 100' file1 (or)

$ awk –F" " '$3 > 100 { print $0 }' file1 // displays line in file1 whose 3rd field value is greater than 100

$ awk –F" " '$3 > 100 { print $1,$3 }' file1 //displays 1st and 3rd field in the lines whose 3rd field values is greater than 100

$ awk –F" " ' /mca/ { print }' file1 //displays the lines those contain the data 'mca'

```
$ awk –F" " 'NR==3,NR==6 { print NR,$2,$3}' file1 //displays line number, 2nd and 3rd field in the 3rd and 6th line
```

Printf: for display formatted output

```
$awk –F" " 'NR==3 {

> printf "%3d %20s \n",NR,$1' file1 //displays line number and 1st field value.
```

```
$awk –F" " 'NR==3 {

> printf "%3d %20s \n",NR,$1' file1 > file2 //output is stored in file2
```

Comparison Operators: <, <=, ==, !=, >=, >, ~ - matches a regular expression, !~ - doesn't match a regular expression.

```
$ awk –F" " '$3=="director" || $3=="chairman" { print }' file1
```

Number Processing: +, -, *, / and %

```
$ awk –F" " '$4=="sales" {

> printf "%20s %10d %8.2f \n",$2,$3,$3/11}' file1
```

Variables:

```
$ awk –F" " '$3>100 {

> count = count + 1

> printf "%d \n",count}' file1
```

Supports count++, count += 2 and ++count

Reading the Program form a File:

```
$ cat > sample.awk
$2==100 {print $1}
Press ctrl + d
$ awk –F" " –f sample.awk file1
```

Join command:

Join lines of two files based on a common field. You can join two files based on a common field, that you can specify using field.

Syntax: $ join -t':' -1 N -2 N file1 file2

- -t':' – : is the field separator
- -1 N : Nth field in 1st file
- -2 N : Nth field in 2nd file
- file1 file2 : files that should be joined

In this example, let us combine employee.txt and bonus.txt files using the common employee number field.

```
$ cat employee.txt
100     Emma    Thomas
200     Alex    Jason
300     Madison Randy
400     Sanjay  Gupta
500     Nisha   Singh
```

```
$ cat bonus.txt
```

$5,000 100

$5,500 200

$6,000 300

$7,000 400

$9,500 500

$ join -1 1 -2 2 employee.txt bonus.txt

100 Emma Thomas $5,000

200 Alex Jason $5,500

300 Madison Randy $6,000

400 Sanjay Gupta $7,000

500 Nisha Singh $9,500

Pg command:

pg is a terminal pager program on Unix for viewing text files. It can also be used to page through the output of a command via a pipe. pg uses an interface similar to vi.

Syntax: $pg filename

comm command:

comm - compare two sorted files line by line. The comm command in the Unix family of computer operating systems is a utility that is used to compare two files for common and distinct lines

comm reads two files as input, regarded as lines of text. comm outputs one file, which contains three columns. This functionally is similar to diff.Columns are typically distinguished with the <tab> character

Syntax: comm [OPTION]... FILE1 FILE2

With no options, produce three-column output. Column one contains lines unique to FILE1, column two contains lines unique to FILE2, and column three contains lines common to both files.

Col1- suppress lines unique to FILE1

Col2 - suppress lines unique to FILE2

Col3- suppress lines that appear in both files

eg: $comm-1 file1 file2 //It displays only 2nd and 3rd columns

Cmp command:

cmp is a command line utility for computer systems that use Unix. It compares two files of any type and writes the results to the standard output. By default, cmp is silent if the files are the same; if they differ, the byte and line number at which the first difference occurred is reported.

Syntax : cmp [-c] [-i N] [-l] [-s] [-v] firstfile secondfile

Options:

-c	Output differing bytes as characters.
-l	Write the byte number (decimal) and the differing bytes (octal) for each difference.
-s	Write nothing for differing files; return exit statuses only.
-v	Output version info.

Examples

$cmp file1.txt file2.txt

Compares file1 to file2 and outputs results. Below is example of how these results may look.

$file.txt file2.txt differ: char 1011, line 112

diff command:

This command is used to display file differences. It also tells you which lines in one file have to be changed to make two files identical.

Syntax: $ diff file1 file2

diff uses certain special symbols and instructions to indicate the changes that are required to make two files identical. Each instruction uses an address combined with an action that is applied to the first file.

The instruction

1. **7a8** means appending line after line 7, which become line8 in the second file.
2. **3c3** change line 3 which remains 3 line after the change.
3. **5,7c5,7** changes 3 lines.

tr command: (translate or transliterate)

When executed, the program reads from the standard input and writes to the standard output. It takes asparameters two sets of characters, and replaces occurrences of the characters in the first set with the corresponding elements from the other set.

Syntax: tr options expression1 expression2 < standard input

Eg:1.$tr 'abcd' 'jkmn'

maps 'a' to 'j', 'b' to 'k', 'c' to 'm', and 'd' to 'n'.

Sets of characters may be abbreviated by using character ranges. The previous example could be written:

2.$tr 'a-d' 'jkmn'

3. $ tr ‘a-z’ ‘A-Z’ < file1

Options:

-d -- delete the specified characters

$ tr –d ‘ad’ < file1

-c – delete all the characters except specified character.

$ tr –cd ‘ad’ < file1

Networking Commands

The ping Utility

The ping command sends an echo request to a host available on the network. Using this command you can check if your remote host is responding well or not.

The ping command is useful for the following −

- Tracking and isolating hardware and software problems.
- Determining the status of the network and various foreign hosts.
- Testing, measuring, and managing networks.

Syntax

Following is the simple syntax to use **ping** command −

```
$ping hostname or ip-address
```

Above command would start printing a response after every second. To come out of the command you can terminate it by pressing CNTRL + C keys.

Example

Following is the example to check the availability of a host available on the network −

```
$ping google.com
PING google.com (74.125.67.100) 56(84) bytes of data.
64 bytes from 74.125.67.100: icmp_seq=1 ttl=54 time=39.4 ms
64 bytes from 74.125.67.100: icmp_seq=2 ttl=54 time=39.9 ms
64 bytes from 74.125.67.100: icmp_seq=3 ttl=54 time=39.3 ms
```

```
64 bytes from 74.125.67.100: icmp_seq=4 ttl=54 time=39.1 ms
64 bytes from 74.125.67.100: icmp_seq=5 ttl=54 time=38.8 ms
--- google.com ping statistics ---
22 packets transmitted, 22 received, 0% packet loss, time 21017ms
rtt min/avg/max/mdev = 38.867/39.334/39.900/0.396 ms
$
```

If a host does not exist then it would behave something like this −

```
$ping giiiiiigle.com
ping: unknown host giiiiigle.com
$
```

The ftp Utility

Here ftp stands for **F**ile **T**ransfer **P**rotocol. This utility helps you to upload and download your file from one computer to another computer.

The ftp utility has its own set of UNIX like commands which allow you to perform tasks such as −

- Connect and login to a remote host.
- Navigate directories.
- List directory contents
- Put and get files
- Transfer files as ascii, ebcdic or binary

Syntax

Following is the simple syntax to use **ping** command −

```
$ftp hostname or ip-address
```

Above command would prompt you for login ID and password. Once you are authenticated, you would have access on the home directory of the login account and you would be able to perform various commands.

Few of the useful commands are listed below −

Command	Description
put filename	Upload filename from local machine to remote machine.
get filename	Download filename from remote machine to local machine.
mput file list	Upload more than one files from local machine to remote machine.
mget file list	Download more than one files from remote machine to local machine.
prompt off	Turns prompt off, by default you would be prompted to upload or download movies using mput or mget commands.
prompt on	Turns prompt on.
dir	List all the files available in the current directory of remote machine.
cd dirname	Change directory to dirname on remote machine.
lcd dirname	Change directory to dirname on local machine.
quit	Logout from the current login.

It should be noted that all the files would be downloaded or uploaded to or from current directories. If you want to upload your files in a particular directory then first you change to that directory and then upload required files.

Example

Following is the example to show few commands −

```
$ftp amrood.com
Connected to amrood.com.
220 amrood.com FTP server (Ver 4.9 Thu Sep 2 20:35:07 CDT 2009)
Name (amrood.com:amrood): amrood
331 Password required for amrood.
Password:
230 User amrood logged in.
ftp> dir
200 PORT command successful.
150 Opening data connection for /bin/ls.
total 1464
drwxr-sr-x   3 amrood   group       1024 Mar 11 20:04 Mail
drwxr-sr-x   2 amrood   group       1536 Mar  3 18:07 Misc
drwxr-sr-x   5 amrood   group        512 Dec  7 10:59 OldStuff
drwxr-sr-x   2 amrood   group       1024 Mar 11 15:24 bin
drwxr-sr-x   5 amrood   group       3072 Mar 13 16:10 mpl
-rw-r--r--   1 amrood   group     209671 Mar 15 10:57 myfile.out
drwxr-sr-x   3 amrood   group        512 Jan  5 13:32 public
drwxr-sr-x   3 amrood   group        512 Feb 10 10:17 pvm3
226 Transfer complete.
ftp> cd mpl
250 CWD command successful.
ftp> dir
200 PORT command successful.
150 Opening data connection for /bin/ls.
```

```
total 7320
-rw-r--r--   1 amrood   group      1630 Aug  8 1994  dboard.f
-rw-r-----   1 amrood   group      4340 Jul 17 1994  vttest.c
226 Transfer complete.
ftp> get wave_shift
200 PORT command successful.
150 Opening data connection for wave_shift (525574 bytes).
226 Transfer complete.
528454 bytes received in 1.296 seconds (398.1 Kbytes/s)
ftp> quit
221 Goodbye.
$
```

The telnet Utility

Many times you would be in need to connect to a remote Unix machine and work on that machine remotely. Telnet is a utility that allows a computer user at one site to make a connection, login and then conduct work on a computer at another site.

Once you are login using telnet, you can perform all the activities on your remotely connect machine. Here is example telnet session −

```
C:>telnet amrood.com
Trying...
Connected to amrood.com.
Escape character is '^]'.

login: amrood
amrood's Password:
*****************************************************
*                                    *
*                                    *
*    WELCOME TO AMROOD.COM                     *
```

```
*                                  *
*                                  *
**************************************************

Last unsuccessful login: Fri Mar  3 12:01:09 IST 2009
Last login: Wed Mar  8 18:33:27 IST 2009 on pts/10

  {  do your work }

$ logout
Connection closed.
C:>
```

The finger Utility

The finger command displays information about users on a given host. The host can be either local or remote.

Finger may be disabled on other systems for security reasons.

Following are the simple syntax to use finger command –

Check all the logged in users on local machine as follows –

```
$ finger
Login     Name       Tty      Idle  Login Time   Office
amrood               pts/0          Jun 25 08:03 (62.61.164.115)
```

Get information about a specific user available on local machine –

```
$ finger amrood
Login: amrood                       Name: (null)
Directory: /home/amrood             Shell: /bin/bash
On since Thu Jun 25 08:03 (MST) on pts/0 from 62.61.164.115
No mail.
```

```
No Plan.
```

Check all the logged in users on remote machine as follows −

```
$ finger @avtar.com
Login     Name       Tty      Idle  Login Time   Office
amrood              pts/0         Jun 25 08:03 (62.61.164.115)
```

Get information about a specific user available on remote machine −

```
$ finger amrood@avtar.com
Login: amrood                          Name: (null)
Directory: /home/amrood                Shell: /bin/bash
On since Thu Jun 25 08:03 (MST) on pts/0 from 62.61.164.115
No mail.
No Plan.
```

7. Explain the control structures of shell programming with suitable examples?

Ans) The **if...else** statements

The **case...esac** statement

The if...else statements:

If else statements are useful decision making statements which can be used to select an option from a given set of options.

Unix Shell supports following forms of if..else statement −

- if...fi statement
- if...else...fi statement
- if...elif...else...fi statement

Most of the if statements check relations using relational operators discussed in previous chapter.

The case...esac Statement

You can use multiple if...elif statements to perform a multiway branch. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.

Unix Shell supports **case...esac** statement which handles exactly this situation, and it does so more efficiently than repeated if...elif statements.

There is only one form of case...esac statement which is detailed here −

- case...esac statement

Unix Shell's case...esac is very similar to switch...case statement we have in other programming languages like C or C++ and PERL etc.

Loops are a powerful programming tool that enable you to execute a set of commands repeatedly. In this tutorial, you would examine the following types of loops available to shell programmers −

- The while loop
- The for loop
- The until loop
- The select loop

You would use different loops based on dfferent situation. For example while loop would execute given commands until given condition remains true where as until loop would execute until a given condition becomes true.

Once you have good programming practice you would start using appropriate loop based on situation. Here while and for loops are available in most of the other programming languages like C, C++ and PERL etc.

Nesting Loops

All the loops support nesting concept which means you can put one loop inside another similar or different loops. This nesting can go upto unlimited number of times based on your requirement.

Here is an example of nesting **while** loop and similar way other loops can be nested based on programming requirement −

Nesting while Loops

It is possible to use a while loop as part of the body of another while loop.

Syntax

```
while command1 ; # this is loop1, the outer loop
do
   Statement(s) to be executed if command1 is true

   while command2 ; # this is loop2, the inner loop
   do
      Statement(s) to be executed if command2 is true
   done

   Statement(s) to be executed if command1 is true
done
```

Example

Here is a simple example of loop nesting, let's add another countdown loop inside the loop that you used to count to nine −

```
#!/bin/sh

a=0
while [ "$a" -lt 10 ]    # this is loop1
do
```

```
  b="$a"
  while [ "$b" -ge 0 ]  # this is loop2
  do
    echo -n "$b "
    b=`expr $b - 1`
  done
  echo
  a=`expr $a + 1`
done
```

This will produce following result. It is important to note how **echo -n** works here. Here **-n** option let echo to avoid printing a new line character.

```
0
1 0
2 1 0
3 2 1 0
4 3 2 1 0
5 4 3 2 1 0
6 5 4 3 2 1 0
7 6 5 4 3 2 1 0
8 7 6 5 4 3 2 1 0
9 8 7 6 5 4 3 2 1 0
```

8. Write about here documents?

Ans) A here document is used to redirect input into an interactive shell script or program.

We can run an interactive program within a shell script without user action by supplying the required input for the interactive program, or interactive shell script.

The general form for a here document is −

```
command << delimiter
```

```
document
delimiter
```

Here the shell interprets the << operator as an instruction to read input until it finds a line containing the specified delimiter. All the input lines up to the line containing the delimiter are then fed into the standard input of the command.

The delimiter tells the shell that the here document has completed. Without it, the shell continues to read input forever. The delimiter must be a single word that does not contain spaces or tabs.

Following is the input to the command **wc -l** to count total number of line −

```
$wc -l << EOF
        This is a simple lookup program
        for good (and bad) restaurants
        in Cape Town.
EOF
3
$
```

You can use here document to print multiple lines using your script as follows −

```
#!/bin/sh

cat << EOF
This is a simple lookup program
for good (and bad) restaurants
in Cape Town.
EOF
```

This would produce following result −

```
This is a simple lookup program
for good (and bad) restaurants
in Cape Town.
```

The following script runs a session with the vi text editor and save the input in the file test.txt.

```
#!/bin/sh


filename=test.txt
vi $filename <<EndOfCommands
i
This file was created automatically from
a shell script
^[
ZZ
EndOfCommands
```

If you run this script with vim acting as vi, then you will likely see output like the following –

```
$ sh test.sh
Vim: Warning: Input is not from a terminal
$
```

After running the script, you should see the following added to the file test.txt –

```
$ cat test.txt
This file was created automatically from
a shell script
$
```

9 Describe input redirection and output redirection?

Ans) Output Redirection

The output from a command normally intended for standard output can be easily diverted to a file instead. This capability is known as output redirection:

If the notation > file is appended to any command that normally writes its output to standard output, the output of that command will be written to file instead of your terminal −

Check following **who** command which would redirect complete output of the command in users file.

```
$ who > users
```

Notice that no output appears at the terminal. This is because the output has been redirected from the default standard output device (the terminal) into the specified file. If you would check users file then it would have complete content −

```
$ cat users
oko        tty01   Sep 12 07:30
ai         tty15   Sep 12 13:32
ruth       tty21   Sep 12 10:10
pat        tty24   Sep 12 13:07
steve      tty25   Sep 12 13:03
$
```

If a command has its output redirected to a file and the file already contains some data, that data will be lost. Consider this example −

```
$ echo line 1 > users
$ cat users
line 1
$
```

You can use >> operator to append the output in an existing file as follows −

```
$ echo line 2 >> users
$ cat users
line 1
```

```
line 2
$
```

Input Redirection

Just as the output of a command can be redirected to a file, so can the input of a command be redirected from a file. As the greater-than character > is used for output redirection, the less-than character < is used to redirect the input of a command.

The commands that normally take their input from standard input can have their input redirected from a file in this manner. For example, to count the number of lines in the file users generated above, you can execute the command as follows −

```
$ wc -l users
2 users
$
```

Here it produces output 2 lines. You can count the number of lines in the file by redirecting the standard input of the wc command from the file users −

```
$ wc -l < users
2
$
```

Note that there is a difference in the output produced by the two forms of the wc command. In the first case, the name of the file users is listed with the line count; in the second case, it is not.

In the first case, wc knows that it is reading its input from the file users. In the second case, it only knows that it is reading its input from standard input so it does not display file name.

11 Explain various meta characters in shell with an example script.

Ans) Metacharacters

Unix Shell provides various metacharacters which have special meaning while using them in any Shell Script and causes termination of a word unless quoted.

For example **?** matches with a single charater while listing files in a directory and an ***** would match more than one characters. Here is a list of most of the shell special characters (also called metacharacters) −

```
* ? [ ] ' " \ $ ; & ( ) | ^ < > new-line space tab
```

A character may be quoted (i.e., made to stand for itself) by preceding it with a \.

Example

Following is the example which show how to print a ***** or a **?** −

```
#!/bin/sh

echo Hello; Word
```

This would produce following result −

```
Hello
./test.sh: line 2: Word: command not found

shell returned 127
```

Now let us try using a quoted character −

```
#!/bin/sh

echo Hello\; Word
```

This would produce following result −

```
Hello; Word
```

The $ sign is one of the metacharacters, so it must be quoted to avoid special handling by the shell −

```
#!/bin/sh

echo "I have \$1200"
```

This would produce following result –

```
I have $1200
```

There are following four forms of quotings –

Quoting	Description
Single quote	All special characters between these quotes lose their special meaning.
Double quote	Most special characters between these quotes lose their special meaning with these exceptions: • $ • ` • \$ • \' • \" • \\
Backslash	Any character immediately following the backslash loses its special meaning.
Back Quote	Anything in between back quotes would be treated as a command and would be executed.

The Single Quotes

Consider an echo command that contains many special shell characters –

```
echo <-$1500.**>; (update?) [y|n]
```

Putting a backslash in front of each special character is tedious and makes the line difficult to read −

```
echo \<-\$1500.\*\*\>\; \(update\?\) \[y\|n\]
```

There is an easy way to quote a large group of characters. Put a single quote (') at the beginning and at the end of the string −

```
echo '<-$1500.**>; (update?) [y|n]'
```

Any characters within single quotes are quoted just as if a backslash is in front of each character. So now this echo command displays properly.

If a single quote appears within a string to be output, you should not put the whole string within single quotes instead you whould preceed that using a backslash (\) as follows −

```
echo 'It\'s Shell Programming'
```

The Double Quotes

Try to execute the following shell script. This shell script makes use of single quote −

```
VAR=ZARA
echo '$VAR owes <-$1500.**>; [ as of (`date +%m/%d`) ]'
```

This would produce following result −

```
$VAR owes <-$1500.**>; [ as of (`date +%m/%d`) ]
```

So this is not what you wanted to display. It is obvious that single quotes prevent variable substitution. If you want to substitute variable values and to make invert commas work as expected then you would need to put your commands in double quotes as follows −

```
VAR=ZARA
echo "$VAR owes <-\$1500.**>; [ as of (`date +%m/%d`) ]"
```

Now this would produce following result −

```
ZARA owes <-$1500.**>; [ as of (07/02) ]
```

Double quotes take away the special meaning of all characters except the following −

- $ for parameter substitution.
- Backquotes for command substitution.
- \$ to enable literal dollar signs.
- \` to enable literal backquotes.
- \" to enable embedded double quotes.
- \\ to enable embedded backslashes.
- All other \ characters are literal (not special).

Any characters within single quotes are quoted just as if a backslash is in front of each character. So now this echo command displays properly.

If a single quote appears within a string to be output, you should not put the whole string within single quotes instead you whould preceed that using a backslash (\) as follows −

```
echo 'It\'s Shell Programming'
```

The Back Quotes

Putting any Shell command in between back quotes would execute the command

Syntax:

Here is the simple syntax to put any Shell **command** in between back quotes −

Example

```
var=`command`
```

Example

Following would execute **date** command and produced result would be stored in DATA variable.

```
DATE=`date`

echo "Current Date: $DATE"
```

This would produce following result −

```
Current Date: Thu Jul  2 05:28:45 MST 2009
```

Output Redirection

The output from a command normally intended for standard output can be easily diverted to a file instead. This capability is known as output redirection:

If the notation > file is appended to any command that normally writes its output to standard output, the output of that command will be written to file instead of your terminal −

Check following **who** command which would redirect complete output of the command in users file.

```
$ who > users
```

Notice that no output appears at the terminal. This is because the output has been redirected from the default standard output device (the terminal) into the specified file. If you would check users file then it would have complete content −

```
$ cat users
oko        tty01   Sep 12 07:30
ai         tty15   Sep 12 13:32
ruth       tty21   Sep 12 10:10
pat        tty24   Sep 12 13:07
steve      tty25   Sep 12 13:03
$
```

If a command has its output redirected to a file and the file already contains some data, that data will be lost. Consider this example −

```
$ echo line 1 > users
$ cat users
line 1
$
```

You can use >> operator to append the output in an existing file as follows −

```
$ echo line 2 >> users
$ cat users
line 1
line 2
$
```