SYSTEM PROGRAMMING AND SYSTEM ADMINISTRATION

SEM-6TH

SECTION-D(NOTES)

SHELL:

A **Shell** provides you with an interface to the Unix system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.

Shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of a shell, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions.

Shell Prompt

The prompt, **$**, which is called the **command prompt**, is issued by the shell. While the prompt is displayed, you can type a command.

Shell reads your input after you press **Enter**. It determines the command you want executed by looking at the first word of your input. A word is an unbroken set of characters. Spaces and tabs separate words.

Following is a simple example of the **date** command, which displays the current date and time −

```
$date
Thu Jun 25 08:30:19 MST 2009
```

You can customize your command prompt using the environment variable PS1 explained in the Environment tutorial.

Shell Types

In Unix, there are two major types of shells −

- **Bourne shell** − If you are using a Bourne-type shell, the **$** character is the default prompt.
- **C shell** − If you are using a C-type shell, the % character is the default prompt.

The Bourne Shell has the following subcategories −

- Bourne shell (sh)
- Korn shell (ksh)
- Bourne Again shell (bash)
- POSIX shell (sh)

The different C-type shells follow −

- C shell (csh)
- TENEX/TOPS C shell (tcsh)

The original Unix shell was written in the mid-1970s by Stephen R. Bourne while he was at the AT&T Bell Labs in New Jersey.

Bourne shell was the first shell to appear on Unix systems, thus it is referred to as "the shell".

Bourne shell is usually installed as **/bin/sh** on most versions of Unix. For this reason, it is the shell of choice for writing scripts that can be used on different versions of Unix

Shell Scripts

The basic concept of a shell script is a list of commands, which are listed in the order of execution. A good shell script will have comments, preceded by **#** sign, describing the steps.

There are conditional tests, such as value A is greater than value B, loops allowing us to go through massive amounts of data, files to read and store data, and variables to read and store data, and the script may include functions.

Shell Comments

```
#!/bin/bash

# Author : Zara Ali
# Copyright (c) Tutorialspoint.com
# Script follows here:
pwd
ls
```

A variable is nothing more than a pointer to the actual data. The shell enables you to create, assign, and delete variables.

Variable Names

The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (_).

By convention, Unix shell variables will have their names in UPPERCASE.

Defining Variables

Variables are defined as follows −

```
variable_name=variable_value
```

Accessing Values

To access the value stored in a variable, prefix its name with the dollar sign (**$**) −

For example, the following script will access the value of defined variable NAME and print it on STDOUT −

```
#!/bin/sh

NAME="Zara Ali"
echo $NAME
```

Variable Types

When a shell is running, three main types of variables are present −

- **Local Variables** − A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at the command prompt.
- **Environment Variables** − An environment variable is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually, a shell script defines only those environment variables that are needed by the programs that it runs.
- **Shell Variables** − A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.

Command-Line Arguments

The command-line arguments $1, $2, $3, ...$9 are positional parameters, with $0 pointing to the actual command, program, shell script, or function and $1, $2, $3, ...$9 as the arguments to the command.

Following script uses various special variables related to the command line −

```
#!/bin/sh

echo "File Name: $0"
echo "First Parameter : $1"
echo "Second Parameter : $2"
echo "Quoted Values: $@"
echo "Quoted Values: $*"
echo "Total Number of Parameters : $#"
```

Special Parameters $* and $@ (WILDCARDS):

There are special parameters that allow accessing all the command-line arguments at once. **$*** and **$@** both will act the same unless they are enclosed in double quotes, **""**.

Both the parameters specify the command-line arguments. However, the "$*" special parameter takes the entire list as one argument with spaces between and the "$@" special parameter takes the entire list and separates it into separate arguments.

We can write the shell script as shown below to process an unknown number of command line arguments with either the $* or $@ special parameters −

```
#!/bin/sh

for TOKEN in $*
do
  echo $TOKEN
done
```

SHELL PROGRAMMING CONSTRUCTS:

The following types of loops available to shell programmers −

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

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
```

The shell performs substitution when it encounters an expression that contains one or more special characters.

Example

Here, the printing value of the variable is substituted by its value. Same time, **"\n"** is substituted by a new line −

```
#!/bin/sh

a=10
echo -e "Value of a is $a \n"
```

Variable Substitution

Variable substitution enables the shell programmer to manipulate the value of a variable based on its state.

```
#!/bin/sh

echo ${var:-"Variable is not set"}
echo "1 - Value of var is ${var}"

echo ${var:="Variable is not set"}
echo "2 - Value of var is ${var}"

unset var
echo ${var:+"This is default value"}
echo "3 - Value of var is $var"

var="Prefix"
echo ${var:+"This is default value"}
echo "4 - Value of var is $var"

echo ${var:?"Print this message"}
echo "5 - Value of var is ${var}"
```

Upon execution, you will receive the following result −

```
Variable is not set
1 - Value of var is
Variable is not set
2 - Value of var is Variable is not set

3 - Value of var is
This is default value
4 - Value of var is Prefix
Prefix
5 - Value of var is Prefix
```

SHELL CONSTRUCTS:

The looping **constructs**, for and while, allow a **program** to iterate through groups of commands in a loop. The conditional control commands, if and case, execute a group of commands only if a particular set of conditions is met. The break command allows a **program** to exit unconditionally from a loop. following two statements that are used to control shell loops−

- The **break** statement
- The **continue** statement

The infinite Loop

All the loops have a limited life and they come out once the condition is false or true depending on the

loop.

A loop may continue forever if the required condition is not met. A loop that executes forever without terminating executes for an infinite number of times. For this reason, such loops are called infinite loops.

```
#!/bin/sh

a=10

until [ $a -lt 10 ]
do
   echo $a
   a=expr $a + 1`
done
```

The break Statement

The **break** statement is used to terminate the execution of the entire loop, after completing the execution of all of the lines of code up to the break statement. It then steps down to the code following the end of the loop.

Syntax

The following **break** statement is used to come out of a loop −

```
break
```

The break command can also be used to exit from a nested loop using this format −

```
break n
```

Here **n** specifies the **n**th enclosing loop to the exit from.

ADVANTAGES OF SHELL PROGRAMMING

A. Easy to use.
B. Quick start and interactive debugging.
C. Time saving.
D. System admin task automation.
E. Shell scripts can execute without any additional effort on nearly any modern UNIX / Linux / BSD / Mac OS X operating system as they are written an interpreted language.

DISADVANTAGES OF SHELL PROGRAMMING

A. Compatibility problem between different platforms.
B. Slow execution speed.

Advanced Shell Scripting

The special operator `&&` and `||` can be used to execute functions in sequence

```
grep '^harry:' /etc/passwd || useradd harry
```

The `||` means to only execute the second command if the first command returns an error. In the above case, `grep` will return an exit code of 1 if `harry` is not in the `/etc/passwd` file, causing `useradd` to be executed.

Special Parameters: `$?`, `$*`,...

An ordinary variable can be expanded with `$`*VARNAME*. Commonly used variables like `PATH` and special variables like `PWD` and `RANDOM`

$*

Expands to the positional parameters (i.e., the command-line arguments passed to the shell script, with **$1** being the first argument, **$2** the second etc.), starting from one. When the expansion occurs within double quotes, it expands to a single word with the value of each parameter separated by the first character of the **IFS** special variable. That is, "**$***" is equivalent to "**$1***c***$2***c***...**", where *c* is the first character of the value of the **IFS** variable. If **IFS** is unset, the parameters are separated by spaces. If **IFS** is null, the parameters are joined without intervening separators.

$@

Expands to the positional parameters, starting from one. When the expansion occurs within double quotes, each parameter expands to a separate word. That is, "**$@**" is equivalent to "**$1**" "**$2**" ... When there are no positional parameters, "**$@**" and **$@** expand to nothing

$#

Expands to the number of positional parameters in decimal (i.e. the number of command-line arguments).

$?

Expands to the status of the most recently executed foreground pipeline.

$-

Expands to the current option flags as specified upon invocation, by the **set** builtin command, or those set by the shell itself (such as the **-i** option).

$$

Expands to the process ID of the shell. In a () subshell, it expands to the process ID of the current shell, not the subshell.

$!

Expands to the process ID of the most recently executed background (asynchronous) command.

SYSTEM ADMINISTRATION:

Systems administration is the installation and maintenance of the UNIX computer system. The system administrator will need to maintain the software and hardware for the system. This includes hardware configuration, software installation, reconfiguration of the kernel, networking, and anything else that's required to make the system work and keep it running in a satisfactory manner. To do this the system administrator can assume superuser, or root, privileges to perform many tasks not normally available to the average user of the system.

A **system administrator**, or **sysadmin**, is a person who is responsible for the upkeep, configuration, and reliable operation of computer systems; especially multi-user computers, such as servers.

The system administrator seeks to ensure that the uptime, performance, resources, and security of the computers he or she manages meet the needs of the users, without exceeding the budget.

BOOTING PROCESS:

1. BIOS

- BIOS stands for Basic Input/Output System
- Performs some system integrity checks
- Searches, loads, and executes the boot loader program.
- It looks for boot loader in floppy, cd-rom, or hard drive. You can press a key (typically F12 of F2, but it depends on your system) during the BIOS startup to change the boot sequence.
- Once the boot loader program is detected and loaded into the memory, BIOS gives the control to it.
- So, in simple terms BIOS loads and executes the MBR boot loader.

2. MBR

- MBR stands for Master Boot Record.
- It is located in the 1st sector of the bootable disk. Typically /dev/hda, or /dev/sda
- MBR is less than 512 bytes in size. This has three components 1) primary boot loader info in 1st 446 bytes 2) partition table info in next 64 bytes 3) mbr validation check in last 2 bytes.
- It contains information about GRUB (or LILO in old systems).
- So, in simple terms MBR loads and executes the GRUB boot loader.

3. GRUB

- GRUB stands for Grand Unified Bootloader.
- If you have multiple kernel images installed on your system, you can choose which one to be executed.
- GRUB displays a splash screen, waits for few seconds, if you don’t enter anything, it loads the default kernel image as specified in the grub configuration file.
- GRUB has the knowledge of the filesystem (the older Linux loader LILO didn’t understand filesystem).

- Grub configuration file is /boot/grub/grub.conf (/etc/grub.conf is a link to this).

4. Kernel

- Mounts the root file system as specified in the "root=" in grub.conf
- Kernel executes the /sbin/init program
- Since init was the 1st program to be executed by Linux Kernel, it has the process id (PID) of 1. Do a 'ps -ef | grep init' and check the pid.
- initrd stands for Initial RAM Disk.
- initrd is used by kernel as temporary root file system until kernel is booted and the real root file system is mounted. It also contains necessary drivers compiled inside, which helps it to access the hard drive partitions, and other hardware.

5. Init

- Looks at the /etc/inittab file to decide the Linux run level.
- Following are the available run levels
    - 0 – halt
    - 1 – Single user mode
    - 2 – Multiuser, without NFS
    - 3 – Full multiuser mode
    - 4 – unused
    - 5 – X11
    - 6 – reboot
- Init identifies the default initlevel from /etc/inittab and uses that to load all appropriate program.
- Execute 'grep initdefault /etc/inittab' on your system to identify the default run level
- If you want to get into trouble, you can set the default run level to 0 or 6. Since you know what 0 and 6 means, probably you might not do that.
- Typically you would set the default run level to either 3 or 5.

6. Runlevel programs

- When the Linux system is booting up, you might see various services getting started. For example, it might say "starting sendmail …. OK". Those are the runlevel programs, executed from the run level directory as defined by your run level.
- Depending on your default init level setting, the system will execute the programs from one of the following directories.
    - Run level 0 – /etc/rc.d/rc0.d/
    - Run level 1 – /etc/rc.d/rc1.d/
    - Run level 2 – /etc/rc.d/rc2.d/
    - Run level 3 – /etc/rc.d/rc3.d/
    - Run level 4 – /etc/rc.d/rc4.d/

- Run level 5 – /etc/rc.d/rc5.d/
- Run level 6 – /etc/rc.d/rc6.d/

BACKUP AND RESORATION:

Backup media has generally fallen into two categories. One category is tape media and the other is removal cartridge. Tape media is generally cheaper is cost and supports larger sizes; however, tape media does not easily support random access to information. Removal cartridge drives, whether optical or magnetic, do support random access to information; however they have lower capacity and a much higher cost per megabyte than tape media. Finally, optical drives generally retain information for a longer time period than tapes and may be used if a permanent archival is needed (side note: Nothing is permanent, if that is a requirement, you must consider ways to make additional backup copies and periodically check the media).
Many backup options out there:

tar

tape archive program, easy to use and transportable. has limit on file name size, won't backup special files, does not follow symbolic links, doesn't support multiple volumes.

advantage is that tar is supported everywhere. Also useful for copying directories

tar -cf . | (cd /user2/directory/jack; tar -xBf -)
tar cvfb /dev/tape 20 .
tar xvf /dev/tape .

cpio

Copy in/out archive program, rarely used except on older System V type machines. Must be given a list of file names to archive. Use find to do this:

find . -print | cpio -ocBV > /dev/tape

reloading a archive from tape
cpio -icdv < /dev/tape

dump

dump (or rdump) reads the raw file system and copies the data blocks out to tape. dump is program that is generally used by most groups. One feature of dump is that dump (via rdump) can work across a network and dump a file system to a remote machine's tape drive. Dump cannot be used to back up just a specific directory. Thus for backup of a directory tree tar or cpio is better suited.

Restoring Files

It always arises to require a restore. To restore an entire file system you must remake the file system (mkfs, mount), then cd to that file system and run the command restore -r. If you have multiple levels of the file system restore will restore the files in reverse order of when they were dumped.

To restore just one file, use the restore command in interactive mode. This is done with the command **restore -i**. In interactive mode, you use the basic unix commands to navigate on the restore tape and then use the sub-command **add filespec** to add that file to the list to be extracted. When complete, use the command **extract** to have restore load in the files from tape. Add a directory to the list also causes all of the files within that directory to be added.

restore - restore files or file systems from backups made with dump
restore -C [**-cdHklMvVy**] [**-b** *blocksize*] [**-D** *filesystem*] [**-f** *file*] [**-F** *script*] [**-L** *limit*] [**-s** *fileno*] [**-T** *directory*]
restore -i [**-acdhHklmMNouvVy**] [**-A** *file*] [**-b** *blocksize*] [**-f** *file*] [**-F** *script*] [**-Q** *file*] [**-s** *fileno*] [**-T** *directory*]

restore -P *file* [**-acdhHklmMNuvVy**] [**-b** *blocksize*] [**-f** *file*] [**-F** *script*] [**-s** *fileno*] [**-T** *directory*] [**-X** *filelist*] [*file ...*]

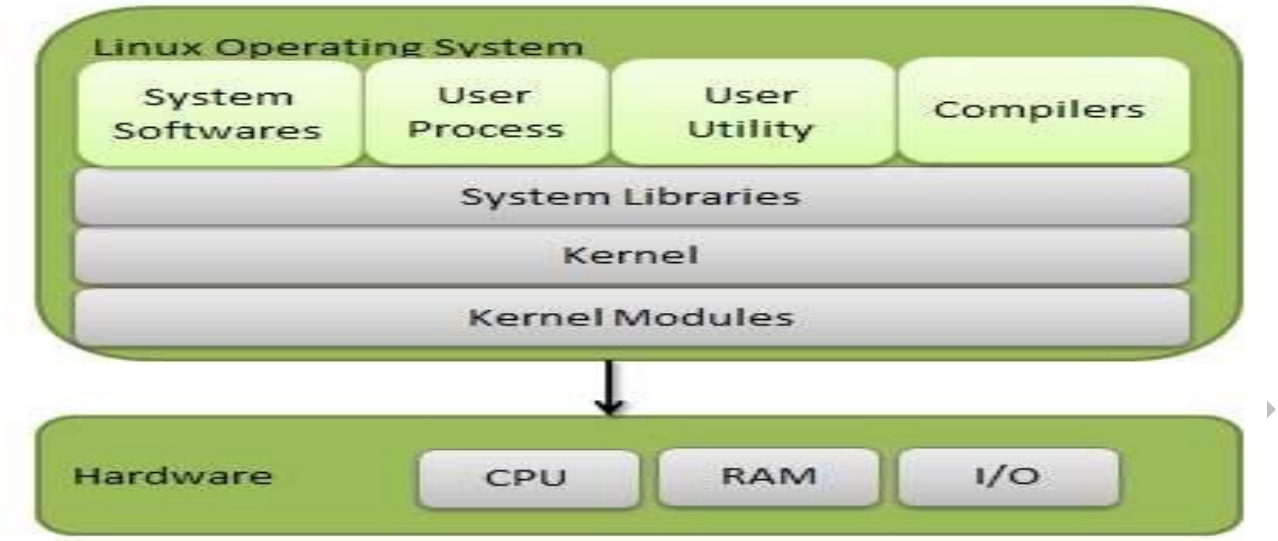
LINUX OPERATING SYSTEM:

Linux is one of popular version of UNIX **operating System**. It is open source as its source code is freely available. It is free to use. ... Its functionality list is quite similar to that of UNIX.
Linux is one of popular version of UNIX operating System. It is open source as its source code is freely available. It is free to use. Linux was designed considering UNIX compatibility. Its functionality list is quite similar to that of UNIX.

Components of Linux System

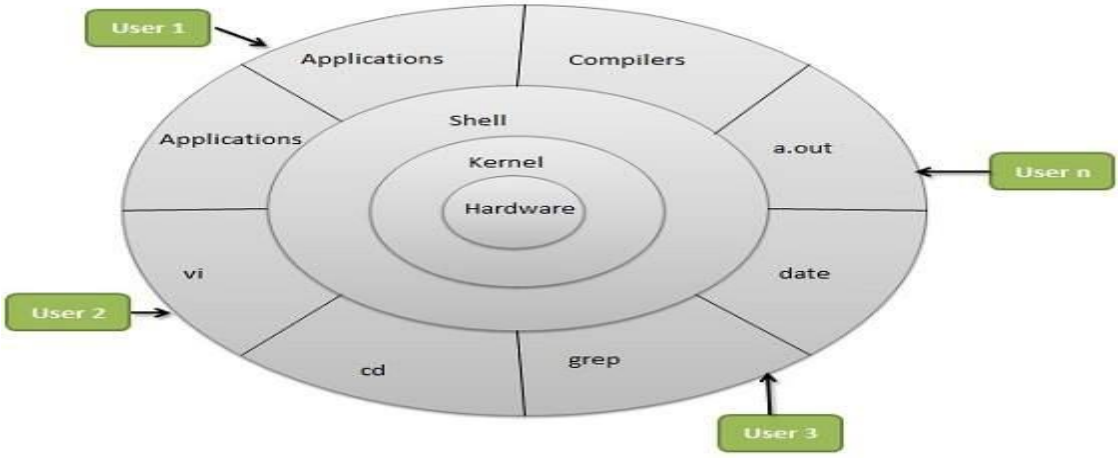
Linux Operating System has primarily three components

- **Kernel** − Kernel is the core part of Linux. It is responsible for all major activities of this operating system. It consists of various modules and it interacts directly with the underlying hardware. Kernel provides the required abstraction to hide low level hardware details to system or application programs.
- **System Library** − System libraries are special functions or programs using which application programs or system utilities accesses Kernel's features. These libraries implement most of the functionalities of the operating system and do not requires kernel module's code access rights.
- **System Utility** − System Utility programs are responsible to do specialized, individual level tasks.



Architecture

The following illustration shows the architecture of a Linux system −



Basic Features

Following are some of the important features of Linux Operating System.

- **Portable** − Portability means software can works on different types of hardware in same way. Linux kernel and application programs supports their installation on any kind of hardware platform.
- **Open Source** − Linux source code is freely available and it is community based development project. Multiple teams work in collaboration to enhance the capability of Linux operating system and it is continuously evolving.
- **Multi-User** − Linux is a multiuser system means multiple users can access system resources

like memory/ ram/ application programs at same time.

- **Multiprogramming** − Linux is a multiprogramming system means multiple applications can run at same time.
- **Hierarchical File System** − Linux provides a standard file structure in which system files/ user files are arranged.
- **Shell** − Linux provides a special interpreter program which can be used to execute commands of the operating system. It can be used to do various types of operations, call application programs. etc.
- **Security** − Linux provides user security using authentication features like password protection/ controlled access to specific files/ encryption of data.