LINUX COMMANDS

COMMAND FORMAT

A command is an instruction given to the Shell; the Kernel will obey that instruction. Linux provides several commands for its users to easily work with it.

The general format of a command is,

command -options command_arguments

A command is normally entered in a line by typing from the keyboard. Even though the terminal's line width is 80 characters, the command length may exceed to 80 characters.

Commands, options and *command_arguments* must be separated by white spacers) or(tabes) to enable the system to interpret them as words.

Options must be preceded by a minus sign (-) to distinguish them from *command_arguments*. Moreover, options can be combined with only one minus sign.

DIRECTORY ORIENTED COMMANDS

1. **ls**

This command is used to list the content of the specified directory.

General format is,

ls [-options] <directory_name>

where *options* can be,

a Lists all directory entries including the hidden files

l Lists the files in long format (filenames along with file type, file permissions, number of links, owner of the file, file size, file creation/modification time, number of links for a file). The number oflinks for a file refers more than one name for a file, does not mean that there are more copies of that file.
This *ls -l* option displays also the year only when the file was last modified more than a year back. Otherwise, it only displays the date without year.

r Lists the files in the reverse order

t Lists the files sorted by the last modification time

R Recursively lists all the files and sub-directories as well as the files in the sub-directories.

p Puts a slash after each directory
s Displays the number of storage blocks used by a file.
x Lists contents by lines instead of by columns in sorted order
F Marks executable files with * (i.e the file having executable permission to the user) and directories with /

WILD CARD CHARACTERS

'*' represents any number of characters.
'?' represents a single character.
For example,

```
$ ls pgm*
```

This command will list out all the file-names of the current directory, which are starting with "**pgm**". Note that the suffix to **pgm** may be any number of characters.

```
$ ls *s
```

This command will display all the filenames of the current directory, which are ending with "**s**". Note that the prefix to **s** may be any number of characters.

```
$ ls ?gms
```

This command will display four character filenames, which are ending with "**gms**" starting with any of the allowed character. Note that the prefix to **gms** is a single character.

"[]" represents a subset of related filenames. This can be used with range operator "-" to access a set of files. Multiple ranges must be separated by commas.

```
$ ls pgm[1-5]
```

This command will list only the files named, pgm1, pgm2, pgm3, pgm4, pgm5 if they exist in the current directory. Note that the **[1-5]** represents the range from 1 through 5.

2. **mkdir**

This mkdir (make directory) command is used to make (create) new directories.

General format is,

mkdir [-p] <directory_narnel> <directory_narne2>

The option ***-p*** is used to create consequences of directories using a single *mkdir* command.

Examples

```
$ mkdir ibr
```

This command will create '*ibr*' a subdirectory of the current directory.

```
$ mkdir x x/y
```

This command will create *x* as a subdirectory of current working directory, *y* as subdirectory of *x*.

```
$ mkdir ibr/ib/i
```

This command will make a directory *i* as a subdirectory of *ibr/ib*, but the directory structure - *ibr/i* must exist.

```
$ mkdir -p ibr/ib/i
```

Then, for the current directory, a subdirectory named *ibr* is created. Then, for the directory *ibr*, a subdirectory named *ib* is created. After that, the subdirectory *i* is created as a subdirectory of the directory *ib*.

3. **rmdir**

This rmdir (remove directory) command is used to remove (delete) the specified directories.

A directory should be empty before removing it.

General format is,

rmdir [-p] <directory_narnel> <directory_narne2>

The option -p is used to remove consequences of directories using a single *rmdir*command.

Example

```
$ rmdir ibr
```

This command will remove the directory *ibr*, which is the subdirectory of the current directory.

```
$ rmdir ibr/ib/i
```

This command will remove the directory *i* only.

```
$ rmdir -p ibr/ib/i
```

This command will remove the directories *i*, *ib* and *ibr* consequently.

4. **cd**

This cd (change directory) command is used to change the current working directory to a specified directory.

General format is,

cd <directory_name>

Examples

```
$ cd  /home/ibr
```

Then, the directory */home/ibr* becomes as the current working directory.

```
$ cd ..
```

This command lets you bring the parent directory as current directory. Here, . . represents the parent directory.

5. pwd

This pwd (print working directory) command displays the full pathname for the current working directory.

General format is,

pwd

Example

```
$ pwd
/home/bmi
```

Your present working directory is */home/bmi.*

6. **find**

This command recursively examines the specified directory tree to look for files matching some file attributes, and then takes some specified action on those files.

General format is,

find <path_list> <selection_criteria> <action>

It recursively examines all files in the directories specified in *<path_list>* and then matches each file for *<selection_cri teria>* (file attributes). Finally, it takes the specified *<action>* on those selected files.

The *selection_criteria* may be as follows,

`-name <filename>`	Selects the file specified in *<filename>*. If wild-cards are used, then double quote *<filename>*
`-user <username>`	Selects files owned by `<username>`
`-type d`	Selects directories

-size {+n / -n}	Selects files that are greater than/less than "*n*" blocks. (Generally one block is 512 bytes)
-mtime {n / +n / -n}	Selects files that have been modified on exactly *n* days / more than *n* days / less than *n* days
-mmin {n / +n / -n}	Selects files that have been modified on exactly *n* minutes / more than *n* minutes / less than *n* minutes
-atime {n / +n / -n}	Selects files that have been accessed on exactly *n* days / more than *n* days / less than *n* days
-amin {n / +n / -n}	Selects files that have been accessed exactly *n* minutes / more than *n* minutes / less than *n* minutes

The ***action*** may be as follows,

-print	Displays the selected files on the screen
-exec <command>	Executes the specified Linux command ends with {}\;

FILE ORIENTED COMMANDS

1. **cat**

This cat command is used to create and display the contents of the specified files.

General format is,

cat [-options] <filenamel> [<filenarne2> ..].

where *options* can be,

s Suppresses warning about non-existent files
d Lists the sub-directory entries only
b Numbers non-blank output lines
n Numbers all output lines

Examples

```
$ cat a.c
```

This will display the contents of the file *a.c*.

```
$ cat a.c b.c
```

This will display the contents of the files, *a.c* and *b.c*, one by one.

This command can be used with redirection operator (>) to create new files.

General format is,

```
cat > filename
<Type the text>
^d  (press [ctrl+d] at the end)
```

Example

```
$ cat  > x.txt
Hi! This
is
a file.
```

Press [^d]

Then, a file named *x.txt* is created in the current working directory with 3 lines content.

2.cp

This cp (copy) command is used to copy the content of one file into another. If the destination is an existing file, the file is overwritten; if the destination is an existing directory, the file is copied into that directory.

General format is,

cp [-options) <source-file> <destination-file>

where *options* **can be,**

i	**Prompt before overwriting destination files**
p	**Preserve all information, including owner, group, permissions, and timestamps**
R	**Recursively copies files in all subdirectories**

3. rm

This rm (remove) command is used to remove (delete) a file from the specified directory. To remove a file, you must have *write* permission for the directory that contains the file, but you need not have permission on the file itself. If you do not have *write* permission on the file, the system will prompt before removing.

General format is,

rm [-options) <filename>

where *options* can be,

r Deletes all directories including the lower order directories. Recursively deletes entire contents of the specified directory and the directory itself

i Prompts before deleting

f Removes write-protected files also, without prompting

4.mv

This mv (move) command is used to rename the specified files /directories.

General format is,

mv <source> <destination>

Note that to make move, the user must have both write and execute permissions on the <source>.

5. **wc**

This command is used to display the number of lines, words and characters of information stored on the specified file.

General format is,

```
wc [-options] <filename>
```

where *options* can be,

l Displays the number of lines in the file

w Displays the number of words in the file

c Displays the number of characters in the file

6. file

This command lists the general classification of a specified file. It lets you to know if the content of the specified file is *ASCII text, Cprogram text, data, separate executable, empty* or' others.

General format is,

file <filename>

```
[bmi@kousar bmi] $ file test1.txt
test1.txt: ASCII text
[bmi@kousar bmi] $ file test2.txt
test2.txt: ASCII text, with escape sequences
[bmi@kousar bmi] $ file *
Destop:          directory
bmi:             directory
c:               directory
java:            directory
shell:           ASCII text
test1.txt:       ASCII text
test2.txt:       ASCII text, with escape sequences
test3.txt:       ASCII English text
text:            directory
```

Here, '*' indicates the content of the current directory.

7. cmp

This cmp (compare) command is used to compare two files.

General format is,

cmp <filename1> <filename2>

This command reports the first instants of differences between the specified files. That is, the two files are compared byte by byte, and the location of the first mismatch is echoed to the screen.

FILE ACCESS PERMISSIONS

Linux treats everything as files. There are three types of files in Linux as follows,

- Ordinary file
- Directory file
- Special file (Device file)

There are three types of modes for accessing these files as follows,

- Read mode (r)
- Write mode (w)
- Execute mode (x)

Linux separates its users into three groups for security and convenience as follows,

- User (u)
- User group *(g)*
- Others (o)

r	Readable
w	Writeable
x	Executable
-	Denial of permission

If a file has "`-rwxrw-r--`" as file permission, then the file is not a directory and it can be readable, writeable & executable for its *owner* (user), readable & writeable for its *usergroup* and only readable for *others*.

8. chmod

This chmod (change mode) command is used to change the file permissions for an existing file. We can use anyone of the following notations to change file permissions.

- Symbolic notation
- Octal notation

Symbolic mode:

General format is,

```
chmod user_symbols set/deny_symbol access_symbols <filename(s)>
```

where *user_symbols* can be,

u	User
g	User group
o	Others

where *set / deny_ symbol* can be,

+	Assign the permissions
-	Remove the permissions
=	Assign absolute permission

where *access_symbols* can be,

r	Readable
w	Writeable
x	Executable

Examples

```
$ chmod u+x file1
```

This command adds *execute* permission to the user for executing the file – *file1*.

```
$chmod g+x file1
```

This command assigns *execute* permission to *usergroup* to execute the file – *file1* in addition to the existing permissions of that file. Note that the file holds its old permissions other than the changed *execute* permission i.e., the already existing permissions are not removed by default

Octal notation:

This mode uses a three-digit number to change the file permissions. In this number, the first digit represents *user* permissions, the second digit represents *usergroup* permissions and the third digit represents *others* permissions.

General format is,

```
chmod three_digit_number <filename1> [<filename2> ...]
```

Digits and their meanings,

0	No permissions
4	Read
2	Write
1	Execute

We can also sum the numbers for mixing of permissions.

3	Write and Execute
5	Read and Execute
6	Read and Write
7	Read, Write and Execute

9. chown

This chown (change ownership) command is used to change the owner of a specifier file. Only the owner of the file and the Superuser can change the file ownership.

General format is,

chown <new_owner> <filename>

This command changes *<new_owner>* as owner of the file specified in *<filename>*.

10. chgrp

This chgrp (change group) command is used to change the group ownership of a specified file. Only the owner of the file and the Superuser can change the group ownership of the file irrespective of whether the user belongs to the same group or not.

General format is,

chgrp <new_groupname> <filename>

This command makes «*neuigroupname*» as group-owner of the file specified *<filename>*.

HH	Hour (00 - 23)
mm	Minute (00 - 59)

If MMDDHHmm expression is not used, then the current date and time are taken by default.

Example

14. dd

This command copies files and converts them in one format into another.

General format is,

```
dd [options=values]
```

where *options* can be,

`if`	**Input filename**
`of`	**Output filename**
`conv`	**File conversion specification. More than one conversion may be specified by separating them with commas.**

The value for this option may be as follows,

`lcase`	**Converts uppercase letters into lowercase**
`ucase`	**Converts lowercase letters into uppercase**
`ascii`	**Converts the file by translating the character set from EBCDIC to ASCII**
`ebcdic`	**Converts the file by translating the character set from ASCII to EBCDIC**

PROCESS ORIENTED COMMANDS

A process is a job in execution. Since Linux is a multi-user operating system, there might be several programs of several users running in memory. There is a program called ***"Scheduler"*** always running in memory which decides which process should get the CPU time and when. Note that only one process will be executed at a time, because *the system has only one processor (CPU).*

1. ps

This command is used to know which processes are running at our terminal.

General format is,

ps

ps -a: This command lists the processes of all the users who are logged on the system.

ps -t <terminal_name>: This command lists the processes, which are running on the specified terminal -*<terminal_name>*.

ps -u <user_name>: This command lists the processes, which are running for the specified user -*<username>*.

ps -x: This command lists the system processes. Apart from the processes that are generated by user, there are some processes that keep on running all the time. These processes are called *system processes*.

BACKGROUND PROCESSING

Linux provides the facility for background processing. That is, when one process is running in the foreground, another process can be executed in the background. The ampersand (&) symbol placed at the end of a command sends the command for background processing.

Example

$ sort emp.doc&

By the execution of this command, a number is displayed. This is called **PID** *(Process IDentification)* number. In Linux, each and every process has a unique PID to identify the process.The PIDs can range from 0 to 32767.

Then, the command *'sort emp.doc'* will run on background. We can execute another command in foreground (as normal).

2. kill

If you want a command to terminate prematurely, press *[ctrl+c]*. This type of interrupt characters does not affect the background processes, because the background processes are protected by the Shell from these interrupt signals. This *kill* command is used to terminate a background process.

General format is,

kill [-SignalNumber] <PID>

The PID is the process identification number of the process that we want to terminate.

COMMUNICATION ORIENTED COMMANDS

Linux provides the communication facility, from which a user can communicate with the other users. The communication can be *online* or *offline*.

1. .write

This online communication command lets you to write messages on another user's terminal.

General format is,

write <RecipientLoginName>
<message>
^d

General format is,

mesg [y|n]

y Allows ***write*** **access to your terminal.**

n Disallows ***write*** **access to your terminal.**

The *mesg* without argument give the status of the mesg setting.

The "finger" command can be used to know the users who are currently logged on the system and to know which terminals of the users are set to *mesg y* and which are set to *mesg n*. A '*' symbol is placed on those terminals where the *mesg* set to *n*.

2. mail

This command offers *off-line* communication.

General format is,

To send mail,

mail <username>
<message>
^d

The mail program mails the message to the specified user. If the user (recipient) is logged on the system, the message *"you have new mail"* is displayed on the recipient's terminal. However, the user is logged on the system or not, the mail will be kept in the mailbox until the user issues the necessary command to read the mails.

Mail Prompt Commands	Functions
+	Displays the next mail message if exists
-	Displays the previous mail message if exists
<number>	Displays the <number>th mail message if exists
D	Deletes currently viewed mail and displays next mail message if exists
d <number>	Deletes the <number>th mail
s <filename>	Stores the current mail message to the file specified in <filename>
s<number> <filename>	Stores the <number>th mail message to the file specified in <filename>
R	Replies to the sender of the currently viewing mail
r <number>	Replies the <number>th mail to its sender
Q	Quits the mail program

3. wall

Usually, this wall (write all) command is used by the super-user to send a message to all the users who were currently logged on the system.

General format is,

```
wall
<message>
<Press [ctrl + d] at the end>
```

Example

```
$ wall
Meeting at 16:00 hrs.
^d
```

GENERAL PURPOSE COMMANDS

1. date

This command displays the system's date and time.

General format is,

```
date +<format>
```

where *<format>* can be,

%H	Hour – 00 to 23
%I	Hour – 00 to 12
%M	Minute – 00 to 59
%S	Second – 00 to 59
%D	Date – MM/DD/YY

%T	Time – HH:MM:SS
%w	Day of the week
%r	Time in AM / PM
%y	Last two digits of the year

2. who

Since Linux is a multi-user operating system, several users may work on this system. This command is used to display the users who are logged on the system currently.

General format is,

who

3. who am i

This command tells you who you are. (Working on the current terminal)

General format is,

```
who am i
```

Example

```
$ who am i
ibrahim  tty1    Feb    19  10:17
```

4. man

This man (manual) command displays the syntax and detailed usage of the Linux command, which is supplied as argument.

General format is,

```
man <LinuxCommand>
```

Example

```
$ man wc
```

This will display the help details for "*wc*" command.

9. cal

This command will display calendar for the specified month and year.

General format is,

```
cal [<month>] <year>
```

where, month can be ranged from 1 to 12.

Example

```
$ cal 2002
```

This command will display calendar for the year 2002 (for 12 months).

```
$ cal 1 2003
```

This will display the calendar for the month - January of the year 2003.

```
$ cal 2 2003
    February 2003
Su Mo Tu We Th Fr Sa
                   1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28
```

10. split

If a file is very large, then it cannot be edited on an editor. In such situations, we need to split the file in to several small files. For this purpose, this split command is used.

General format is,

```
split -<number> <filename>
```

The -<number> option splits the file specified in *<filename>* into *<number>* lined files. Default for this option is 1000 lines.

The splitted contents are stored in the file names xaa, xab, xac, ... ,xaz, xba, xbb, xbc, ... ,xzz (totally 676 filenames).

Example

```
$ split test.txt
```

If the specified file - *test.txt* contains 5550 lines, then this command creates the files namely *xaa, xab, xac, xad, xae* containing 1000 lines and xaf containg remaining 550 lines.

```
$ split -500 test.txt
```

This command splits the test.txt file into 500 lined files.

11. expr

This command is used to perform arithmetic operations on integers.

The arithmetic operators and their corresponding functions are given below.

+	Addition
-	Subtraction
*	Multiplication
/	Division (Decimal portion will be truncated. Because it perform division operation on integers only. It gives only quotient of the division.)
%	Remainder of division (modulus operator)

A white space must be used on either side of an operator. Note that the multiplication operator (*) has to be escaped to prevent the Shell from interpreting it as the filename metacharacter. This command only works with integers, so, the division yields only integer part.

Examples

```
$ x=5
$ y=2
```

Then,

```
$ expr $x + $y
7
```

12. bc

This command is used to perform arithmetic operations on integers as well as on floats (decimal numbers).

Type the arithmetic expression in a line and press *[Enter]* key. Then the answer will be displayed on the next line. After you have finished your work, press *[Ctrl + d]* keys to end up.

Examples

Add 10 and 20.

```
$ bc
10 + 20    ← arithmetic expression
30         ← result is displayed on the next line
Press ^d to end the work.
```

Divide 8 by 3.

```
$ bc
8 / 3
2                ← Decimal portion is truncated.
```

```
$ bc
a=8
b=3
c=a/b
c
```

```
$ bc
ibase=2     ← Set ibase to binary (2)
101         ← Type the input binary number
5           ← Result in decimal

$ bc
obase=2     ← Set obase to binary
5           ← Type the input decimal number
101         ← Result in binary

$bc
ibase=16    ← Set ibase to hexadecimal
B           ← Type the input hexadecimal number
11          ← Result in decimal

$bc
obase=16    ← Set obase to hexadecimal
11          ← Type the input decimal number
B           ← Result in hexa-decimal
```

PIPES AND FILTERS

PIPE

"*Pipe*" is a mechanism in which the output of one command can be redirected as input to another command.

General format is,

```
command1 | command2
```

The output of *command1* is sent to *command2* as input.

Example

```
$ ls | more
```

The output of command **ls** is sent to the **more** command as input. So, the directory of the current directory is displayed page by page.

REDIRECTION

Linux treats the keyboard as the standard input(value 0) and terminal screen as standard output(value 1) as well as standard error(value 2). However input can be taken from resources other than keyboard and output can be passed to any source other than the terminal screen. Such a process is called 'redirection'.

Redirecting inputs

The < symbol is used to redirect inputs.
Example:
$ cat <abc
Then the file **abc** is taken as input for the command **cat**

Redirecting outputs
The > symbol is used to redirect outputs.
Example:
$ ls > xyz
Then the output of the command ls is stored on the file xyz. We can also use '1>' instead of '>'

Redirecting Error messages
The '2>' symbol is used to redirect error messages.
Example:
$ cat pqr
If there is no file named pqr in the current directory, then the error message is sent to the standard error device(screen).

FILTERS
There are some Linux commands that accept input from standard input or files, perform some manipulation on it, and produces some output to the standard output.

1. sort

This command sorts the contents of a given file based on ASCII values of characters.
General format is,

```
sort [options] <filename>
```

where *options* can be,

`-m <filelist>`	**Merges sorted files specified in <filelist>**
`-o <filename>`	**Stores output in the specified <filename>**
`-r`	**Sorts the content in reverse order (reverse alphabetical order)**
`-u`	**Removes duplicate lines and display sorted content**
`-n`	**Numeric sort**
`-t"char"`	**Uses the specified *"char"* as delimiter to identify fields**
`-c`	**Checks if the file is sorted or not**
`+pos`	**Starts sort after skipping the posth field**
`-pos`	**Stops sorting after the posth field**

2. grep

This grep (global search for regular expression) command is used to search for a specified pattern from a specified file and display those lines containing the pattern.

General format is,

```
grep [-options] pattern <filename>
```

(Quote the pattern if the pattern contains Shell special characters.)

where *options* can be,

b	Ignores spaces, tabs
i	Ignores case distinction for matching (do not differentiate capital and small case letters)
v	Displays only the lines that do not match the specified pattern
c	Displays the total number of occurrences of the pattern in the file
n	Displays the resultant lines along with their line numbers
<number>	Displays the matching lines along with <number> of lines above and below

REGULAR EXPRESSION CHARACTER SET

*	Matches zero or more characters
.	Matches a single character (equalent to ? in Shell)
[r1-r2]	Matches a single character within the ASCII range represented by the characters
[^abcd]	Matches a single character which is not a, b, c or d
^<character>	Matches the lines that are beginning with the character specified in <character>
<character>$	Matches the lines that are ending with the character specified in <character>

3. more

If the information to be displayed on the screen is very long, it scrolls up on the screen fastly. So, the user cannot be able to read it. This *more* command is used to display the output page by page (without scrolling up on the screen fastly). Use *[spacebar]* or *[f]* key to scroll forward one screen, use *[b]* key to scroll backward one screen, use *[q]* key to quit displaying.

General format is,

```
more <filename>
```

Example

```
$ more a.c
```

This command will display the content of the file –a.c page by page.

```
$ ls | more
```

This command will display the directory listing page by page.

6. cut

This command is used to cut the columns / fields of a specified file (Like the head and tail commands cut the lines - rows).

General format is,

```
cut [-options] <filename>
```

where options can be,

`c <columns>`	Cuts the columns specified in *<columns>*. You must separate the *column numbers* by using commas
`f <fields>`	Cuts the fields specified in *<fields>*. You must separate field numbers by using commas

7. paste

This command concatenates the contents of the specified files into a single file vertically. (Like *cut* command separates the columns, this *paste* command merges the columns).

General format is,

```
paste <filename1> <filename2> ...
```

8. tr

This tr (translate) command is used to change the case of alphabets.

General format is,

```
tr <CharacterSet1> <CharacterSet2> <StandardInput>
```

This command translates the first character in the *<CharacterSet1>* into the first character in the *<CharacterSet2>* and this same procedure is continued for remaining characters. Note that this command gets input from ***standard input***, not from a file. But you can use pipe or redirection to use a file as input.

```
[bmi@kousar bmi] $ cat e2.dat
computer
zoology
commerce
botany
computer
[bmi@kousar bmi] $ cat e2.dat | tr "[a-z]" "[A-Z]"
COMPUTER
ZOOLOGY
COMMERCE
BOTANY
COMPUTER
[bmi@kousar bmi] $ tr "c,o" "C,O" < e2.dat
COmputer
zOOlOgy
COmmerCe
bOtany
COmputer
[bmi@kousar bmi] $ tr -d "c,o" < e2.dat
mputer
zlgy
mmerce
btany
mputer
```

The *tr* command used with -d option deletes the characters specified in «*Charactertiet*» from input, does not translate.

VI Editor

INTRODUCTION

Linux offers various types of editors like ex,sed,ed,vi,vim,xvi,nvi,elvis etc., to create and edit your files(data files, program files, text files etc.). The famous one is vi editor(visual full screen editor) created by Bill Joy at the University of California at Berkely.

STARTING VI

This editor can be invoked by typing **vi** at the **$** prompt. If you specify a filename as an argument to vi, then the vi will edit the specified file, if it exists.

```
vi [<filename>]
```

VI MODES

The vi editor works on *three modes* as follows:

INSERT MODE:

- The text should be entered in this mode. And any key press in this mode is treated as text.
- We can enter into this mode from command mode by pressing any of the keys:

  `i, I, a, A, o, O, r, R, s, S.`

COMMAND MODE:

- It is the default mode when we start up vi editor.
- All the commands on vi editor (cursor movement, text manipulation, etc.) should be used in this mode.

- We can enter into this mode from *Insert mode* by pressing the *[Esc]* key, and from *Ex mode* by pressing *[Enter]* key.

EX MODE:

- The ex mode commands (saving files, find, replace, etc.,) can be entered at the last line of the screen in this mode.
- We can enter into this mode from command mode by pressing *[:]* key.

Note that one cannot enter to *ex mode* directly from *input mode* and vice versa.

The following are some of the commands that should be used in *command mode*.

INSERT COMMANDS:

i	Inserts before cursor
I	Inserts at the beginning of the current line (the line at which the cursor is placed)
a	Appends after cursor
A	Appends at the end of the current line
o	Inserts a blank line below the current line
O	Insert a blank line above the current line

DELETE COMMANDS:

x	Deletes a character at the cursor position
<n>x	Deletes specified number (n) of characters from the cursor position
X	Deletes a character before the cursor position
<n>X	Deletes specified number (n) of characters before the cursor position
dw	Deletes from cursor position to end of the current word. It stops at any punctuation (eg. ", .) that appears with the word
dW	Same as "dw"; but ignores any punctuation that appears with the word
db	Deletes from cursor position to beginning of the current word. It stops at any punctuation that appears with the word
dB	Same as "db"; but ignores any punctuation that appears with the word
dd	Deletes current line

REPLACE COMMANDS:

r	Replaces single character at the cursor position
R	Replaces characters until *[Esc]* key is pressed from current cursor position
s	Replaces single character at the cursor position with any number of characters
S	Replaces entire line

CURSOR MOVEMENT COMMANDS:

h or *[back space]*	Moves cursor to the left (left arrow)
l or *[space bar]*	Moves cursor to the right (right arrow)
k	Moves cursor to up (up arrow)
j	Moves cursor down (down arrow)
w	Forwards to first letter of next word; but stops at any punctuation that appears with the word
b	Backwards to first letter of previous word; but stops at any punctuation that appears with the word
e	Moves forward to the end of the current word; but stops at any punctuation that appears with the word
W	Same as w; but ignores punctuation that appears with the word
B	Same as b; but ignores punctuation that appears with the word
E	Same as e; but ignores punctuation that appears with the word
[Enter]	Forwards to beginning of next line
0	Moves to the first location of the current line
^	Moves to the first character of the current line

$	Moves to the last character of the current line
H	Moves to the first character (left end) of top line on the current screen
M	Moves to the first character (left end) of middle line on the current screen
L	Moves to the first character (left end) of lowest line on the current screen
G	Moves to the first character of the last line in the current file
<n>G	Moves to the first character of the specified line (n) in the current file
(	Moves to the first character of the current sentence
)	Moves to the first character of next sentence
{	Moves to the first character of current paragraph
}	Moves to the first character of next paragraph

SEARCH COMMANDS:

/string*[Enter]*	Searches the specified ***string*** forward in the file
?string*[Enter]*	Searches the specified ***string*** backward in the file
n	Finds the next string in the same direction (specified by the above commands)
N	Finds the next string in the opposite direction (specified by the above commands)

YANKING COMMANDS: (COPY & PASTE)

yy (or) Y	Yanks the current line in to the buffer (copy)
nyy (or) nY	Copies the 'n' lines from the current line to the buffer
p	Pastes the yanked text below the current line (below the cursor)
P	Pastes the yanked text above the current line (above the cursor)

REDO COMMAND:

. (period)	Repeats the most recent editing operation performed. Note that it is not applicable to cursor movement commands.

UNDO COMMAND:

u	Undoes the most recent editing operation performed.

For example, assume that you have deleted a line currently, if you use this *undo* command, then the deleted line will be displayed (undone the delete operation).

Forward

SCREEN COMMANDS:

Ctrl F	Scrolls full screen (screen of text - page) forward
Ctrl B	Scrolls full screen backward
Ctrl D	Scrolls half screen forward
Ctrl U	Scrolls half screen backward
Ctrl G	Display the status (*filename, total number of lines in the file, current line number, percentage of file that precedes the cursor*) on the *status line* (at bottom of the screen)

Some of the ***ex*** mode commands are given below. These commands should be used in ex mode prefixed by colon (:)

:w	Saves file without quitting
:w <filename>	Saves the content into a file specified in *<filename>*
:m,n w <filename>	Saves the lines *m* through *n* into the specified file
:.w <filename>	Saves the current line into the specified file
:$w <filename>	Saves the last line into the specified line
:x (or) :wq	Saves file and quits from *vi*
:q!	Quits from *vi* without saving

:sh	Escape to the Linux shell (Temporarily exits from *vi*, by typing *exit* or pressing *[ctrl + d]* on $ prompt, we can enter into that vi session.
:s/s1/s2	Does a single replacement s1 as s2 on text (text of the edited file). Press n for next match on the same direction or Press N for next match on the opposite direction
:s/s1/s2/g	Does the replacement s1 as s2 throughout the text
:n,m s/s1/s2/g	Does the replacement in between the lines n and m
:.s/s1/s2/g	Does the replacement only on the current line
:$s/s1/s2/g	Does the replacement only on the last line
:set number	Displays the line numbers sequentially. If we delete/insert a line, then the line numbers are adjusted automatically
:set nonumber	Removes the line numbers, which are set by *:set number* command
:set showmode	Displays the currently working mode at the last line

:!<command>	Executes the specified command without exiting from *vi (not escaping to Linux Shell)*
:<linenumber>	Moves the cursor to the line specified in *<linenumber>*
:set tabstop=<no>	Sets the tab setting to *<no>* number of spaces. (Default is 8 spaces)
:set ignorecase	Ignores case while searching for patterns
:<n1>co<n2>	Copies the line <n1> into below of the line <n2>
:<n1>,<n2>co<n3>	Copies the lines <n1> through <n2> into below of the line <n3>
:<n1>m<n2>	Moves the line <n1> into the line <n2>
:<n1>,<n2>m<n3>	Moves the lines from <n1> through <n2> into the line <n3>
:<n1>d	Deletes the line <n1>
:<n1>,<n2>d	Deletes the lines <n1> through <n2>

Shell Programming

SHELL SCRIPT

If a sequence of commands is to be used repeatedly, we can assign them permanently in a file. This file is called *Shell script.*

Example

Store the following commands in the file named ***disp.sh***.

```
pwd
date
```

COMMAND GROUPING

We can combine several commands using semi-colon instead of executing one by one.

General format is,

```
commnad1 ; command2 ; ...
```

where ; is the *command separator*.

Examples

```
[bmi@kousar bmi] $ pwd ; date
/home/bmi
Fri Dec 27 14:52:52 IST 2002
```

First the *pwd* command is executed, and then the command *date* is executed.

```
[bmi@kousar bmi] $ (pwd ; date) > result.txt
[bmi@kousar bmi] $ cat result.txt
/home/bmi
Fri Dec 27 14:53:19 IST 2002
```

The above command redirects the outputs of the *pwd & date* command into the file named *result.txt*.

SHELL VARIABLES

The Shell variables are classified into two types as follows:

(i)Build-in Shell variables

(ii) User-defined Shell variables.

Build-in Shell Variables:

These variables are created and maintained by the Linux system. These variables are used to define an environment. So, it is also called as **"environmental variables".**

We can list these system variables (built-in variables) corresponding to a user, using the "set" command.

Some of the *build-in* (system) variables are given below.

System Variables	Meanings
HOME	Contains the path name of home directory (where your login Shell is initially located after logged in)
LOGNAME	Contains user's login name
PATH	Contains the directories in which the Shell will search for files to execute the commands that are given by the users
MAIL	Contains the name of the directory in which electronic mails addressed to the user are placed

User-defined Shell Variables:

These variables are created by the users with the following syntax,

```
<variable_name>=<value>
```

Note that there must not be a space on either of the equal sign.

Rules for Naming Shell Variables:

(*i*) The variables must begin with a letter or underscore character. The remaining characters may be letters, numbers or underscores.

(*ii*) No spaces are allowed on either side of the equal sign.

(*iii*) If the <*value*> contains blank-spaces, then it should be enclosed within double quotes.

Examples

```
$ netpay=14000
```

Then, the value *14000* is assigned to the variable *netpay*.

```
$ name="MOHAMED IBRAHIM"
```

echo

This command is used to display values on the screen.

General format is,

```
echo [<string> $variable_name]
```

The symbol $ prefixed to a variable gives the value of that variable. The *echo* without any argument produces an empty line.

Escape sequences	Meanings
\b	back space
\n	new line
\r	carriage return

\t	tab
\a	alert (beep sound)
\\	back slash
\'	single quote
\"	double quote

Positional Parameters:

Most of the Linux commands accept arguments at the command line.

For example,

```
$cp source target
```

Here, *source* and *target* are command line arguments that are sent for the *'cp'* command. Like this, we can also send arguments (*positional parameters*) to a Shell scripts at the command line while it is executed. The argument values can be utilized in the Shell scripts. If an argument has blank spaces, then double quote it. There are some special Shell variables carrying values regarding positional parameters and command execution. The are given below,

$0	Refers to the name of the command (Shell script name)
$1	Refers to the first argument
$2	Refer to the second argument and so on.
$*	Refers all the arguments (positional parameters)
$#	Refers the total number of arguments
$?	Returns the exit status of the last executed command (If a command is executed successfully without giving any error message, then exit status of that command is *zero*; else the exit status of that command is a *non-zero* number.)
$!	Returns the Process IDentification number (PID) of the last background command (command ended with &)
$$	Returns the Process IDentification number (PID) of the current Shell

Shift

The Bourne Shell allows positional parameters $1 through $9 in the Shell scripts at a time. If the command line contains more than nine arguments, the "shift" command can be used to handle the other parameters. This command is used to shift the positional parameters one step forward. As the result, the second argument ($2) becomes the first ($1), and then $3 becomes$2 and so on. In every shift, the value of the first argument is discarded.

General format is,

```
shift
```

Then, the value of $2 becomes the value of $1, $3 becomes $2 and so on

Examples

```
[bmi@kousar bmi] $ cat script1.sh
for i in $*
do
  echo $i
done

[bmi@kousar bmi] $ ./script1.sh 1 2 3
1
2
3

[bmi@kousar bmi] $ cat script2.sh
for i in $*
do
  echo $1
  shift
done

[bmi@kousar bmi] $ ./script2.sh 1 2 3
1
2
3

[bmi@kousar bmi] $ cat script1.sh
for i in $*
do
   echo $1
   shift
done
```

set

This is also used to assign values to the positional parameters, but explicitly.

General format is,

```
set <list_of_values>
```

Examples

```
$ set I am ibrahim
```

Then, $1 = I ; $2 = am ; $3 = ibrahim

```
$ set date
```

Then, $1 = date ; Note that $2 becomes *null*.

```
$ set `date`
```

If current date is "Sat Dec 28 11:47:21 IST 2002" then the positional parameters are set explicitly as follows:

```
$1 = Sat
$2 = Dec
$3 = 28
$4 = 11:47:21
$5 = IST
$6 = 2002
```

read

This command is used to read a line of text from the standard input device (keyboard) or from a file. And it assigns the read value on the specified variable.

General format is,

```
read <variable>
```

Example

```
$ read name
Mohamed Ibrahim          ← Type here a line of text

$ echo Your name is $name
Your name is Mohamed Ibrahim
```

exit

The exit command ends the current Shell. If it is used in Shell scripts, then it terminates a Shell script prematurely and informs the exit status of the script. If the Shell script was executed successfully before exiting, then the exit status is zero; else the exit status is non-zero.

SHELL META CHARACTERS

The Linux separates certain characters for doing special functions. These characters are called "***Shell meta characters***". Some of them are listed below.

Redirection Characters:

`< or 0<`	Redirect for standard input from a specified file `command < filename`
`> or 1>`	Redirect standard output into the specified file `command > filename`
`2>`	Redirect standard error into the specified file `command 2> filename`
`>>`	Appends standard output into the specified file `command >> filename`
`<`	Takes standard input from the specified file `command < filename`
`|`	Connects standard output of one command (`c1`) into standard input of another command (`c2`) `c1 | c2`

Pattern Matching Characters:

We can use the wild card characters (*, ?) in filenames to represent a group of files.

`?`	Matches any single character in filename. For example, `a?c` represents three character filename(s) starting with 'a', ending with 'c' and the middle may be any character.
`*`	Matches any string of zero or more characters in filename(s) For example, `a*c` represents the filename(s) started with 'a', ending with 'c' and the middle may be any combination of characters of any length.
`[character_list]`	Matches any single specified character (specified in *character_list*) in filenames. For example, `a[bc]d` represents the filename `abc` or `acd`.
`[c1-c2]`	Matches a single character that is within the ASCII range of characters `c1` and `c2`. For example, `a[b-e]k` represents the filenames starting with the letter 'a', ending with the letter 'k', and the middle character is `b`, `c`, `d` or `e`.
`[!character_list]`	Matches any single character that is not specified in *character_list*

Command Terminating Characters:

;	Separates the commands when more than one command is given in a line. `command1 ; command2` Executes the command1 and then executes command2.
&	Like ; character, but does not wait the previous command to complete. `command1 & command2` Unlike like the ; character, the *command2* will not wait for the completion of *command1*.

Comment Character:

#	If an # symbol appears in a line, then the rest of the line will be treated as comment For example, `# This is a comment`

Conditional Execution Characters:

&&	`command1 && command2` Executes *commnad1*, if successful executes *command2*.
||	`command1 || command2` Executes *command1*, if failure executes *command2*.

CONTROL STATEMENTS

Shell provides some control statements to execute the commands based on certain condition

if

(format 1)

```
if (conditional_command)
then
   <commands>
fi
```

if

(format 2)

```
if (conditional_command)
then
   <commands>
else
   <commands>
fi
```

if

(format 3)

```
if (conditional_command)
then
   <commands>
elif (conditional_command)
then
   <commands>
.
.
.
else
   <commands>
fi
```

Test command

This command is used to test the validity of its arguments. And mostly this command is used in *conditional commands*. This test command returns an exit status 0 (true) if the test succeeds and 1 (false) if the test fails.

General format is

```
test expression
```

String comparison:

General format is,

```
test operation
```

where the *operation* can be,

`string1=string2`	**Compares two strings; returns true if both are equal, else returns false**
`string1!=string2`	**Compares two strings; returns true if both are not equal, else returns false**
`-z string`	**Checks whether the specified string is of zero length or not; returns true if the string is *null*, else returns false**
`-n string`	**Checks whether the specified string is of non-zero length or not; returns true if the string is not *null*, else returns false**

Numerical Comparison:

General format is,

```
test number1 operator number2
```

where the *operator* can be,

-eq	Returns true if number1 is equal to number2; else returns false
-ne	Returns true if number1 is not equal to number2; else returns false
-gt	Returns true if number1 is greater than number2; else returns false
-lt	Returns true if the number1 is less than number2; else returns false
-ge	Returns true if the number1 is greater than or equal to number2; else returns false
-le	Returns true if number1 is less than or equal to number2; else returns false

File Checking:

General format is,

```
test operator <filename>
```

where the *operator* can be,

-e	Returns true if the file specified in *<filename>* exists; else returns false
-f	Returns true if the file specified in *<filename>* exists and is a regular file (not a directory); else returns false
-d	Returns true if *<filename>* is a directory, not a file; else returns false
-r	Returns true if the file specified in *<filename>* is readable; else returns false
-w	Returns true if the file specified in *<filename>* is writeable; else returns false
-x	Returns true if the file specified in *<filename>* is executable; else returns false
-s	Returns true if the file specified in *<filename>* exists and its size is greater than zero; else returns false

The expressions (*combination of operands and operators*) can be combined using the following logical operators.

!	NOT operation; It is a unary operator that negates (true to false, false to true) the result of the specified expression
-a	AND operation; Returns true if both the expressions are true, else returns false.
-o	OR operation; Returns false if both the expressions are false, else returns true.

case statement:

This is a multi-branch control statement.

General format is,

```
case value in
pattern1)  <commands> ;;
pattern2) <commands> ;;
.
.
.
patternN) <commands> ;;
esac
```

This statement compares the *value* to the *patterns* from the top to bottom, and performs the commands associated with the matching pattern. The commands for each pattern must be terminated by double semicolon.

Example

```
echo "l. Directory listing          ;  d. date "
echo "p. print working directory    ;  q. quit"
echo "Enter your choice"
read choice


case $choice in
l) ls ;;
d) date ;;
p) pwd ;;
q) exit ;;
*) echo "Invalid choice" ;;
esac
```

ITERATIVE STATEMENTS

These statements are used to execute a sequence of commands repeatedly based on some conditions.

while loop:

General format is,

```
while <conitional_command>
do
  <commands>
done
```

The *<conditional_command>* is executed for each cycle of the loop, if it returns a *zero* exit status (success), then the commands between *do* and *done* are executed. This process continues until the *<conditional_command>* yields a *non-zero* exit status (failure).

Example

until loop

General format is,

```
until <conditional_command>
do
 <commands>
done
```

This loop is similar to the *while* loop except that it continues as long as the *<contional_command>* fails (returns a non-zero exit status).

for loop

General format is,

```
for <variable_name> in <list_of_values>
do
 <commands>
done
```

The *<variable_name>* has a value from *<list_of_values>* in each cycle of the loop. And the loop will be executed until the *<list_of_values>* becomes empty.

Examples

```
(i) for i in 1 2 3 4 5
    do
     echo $i
    done
```

The output of this Shell script will be,

```
1
2
3
4
5
```

break

This command is used to exit the enclosing loop (for, while, until) or case command. The optional parameter n is an integer value that represents the number of levels to break when the loop commands are nested. This method is very convenient to exit a deeply nested looping structure when some type of error has occurred.

General format is,

```
break [n]
```

where n represents the number of levels to break.

continue

This command is used to skip to the top of the next iteration of a looping statement. Any commands within the enclosing loop that follow this *continue* statement are skipped and the execution continues at the top of the loop. If the optional parameter "n" is used, then the specified number of enclosing loop levels are skipped.

General format is,

```
continue [n]
```

INFINITE LOOPS

The loops that we have discussed are executed based on condition. The following loops are executed infinitely. Only ***break*** or ***exit*** commands used within the body of the loop will exit the infinitely.

1)
```
while true
do
  statements
done
```

2)
```
until false
do
  statements
done
```

sleep

This command generally used in Shell scripts to delay the execution.

General format is,

```
sleep <NumberOfSeconds>
```

This command delays for the specified number of seconds.

Shell functions

A Shell function is a group of commands that is referred to by a single name. Shell functions are similar to the Shell scripts except that the Shell functions can be executed directly by the login Shell, but Shell scripts are executed by a sub-shell.

General format is,

```
function_name()
{
  commands
}
```

Example

```
[bmi@kousar bmi] $ disp()
> {
> echo "THE .TXT FILES IN THE CURRENT DIRECTORY ARE."
> ls -l *.txt
> echo "...over."
> }
```

sleep

This command generally used in Shell scripts to delay the execution.
General format is,

```
sleep <NumberOfSeconds>
```

This command delays for the specified number of seconds.

Example

```
[bmi@kousar bmi] $ echo "Hello" ; sleep 60 ; echo "ibrahim"
Hello
ibrahim
```

The string "*ibrahim*" will be displayed after one minute of the display "*Hello*".

SYSTEM ADMINISTRATION

SYSTEM ADMINISTRATOR

- The system administrator is the super user who maintains the entire system.
- The system administrator has a special user login name, named root and special prompt t.
- The System Administrator can create files in or delete any file from any directory.

The super user mode can be acquired by,

- Running in single user mode
- Logging as *root* with giving correct super-user password
- By giving su (substitute user) command from a user's Shell

BOOTING THE SYSTEM

- When the system is switched on at the first time, the system executes the boot loader from the start-up disk.
- This boot loader loads the operating system kernel and then runs the kernel.
- A general-purpose utility named lilo (linux loader) writes this boot loader to the start of the active partition.
- When the system is switched on the next time, the kernel will be started by this *lilo* utility.
- The kernel itself being the first process gets the PID O.
- The kernel executes the *letc*init file to invoke the "init" process.

- This process has the PID 1 and this is responsible for the initiation of all other processes.
- This init process invokes the /etc/inittab file.
- *This inittab* file *informs init* about the status of terminals such as which terminals can be used.
- Then, the *init* invokes the /etc/getty file. This file has the various parameters for communication between the terminal and the computer system.

The Linux system can be set up in a number of modes, called as **run levels** that are controlled by *init*. These run levels and their actions are given below.

Run level	Action
S	Single user mode
5	Multi-user mode
0	Halts the system
4	Starts X windows immediately after booting

For example, the command **`init 0`** halts the system and the command **`init 5`** brings the system in multi-user mode.

Single-user mode: When the system is operating in the single user mode, only the system console will be enabled.

Multi-user mode: It is the default mode and all individual file systems are mounted and also system domains are started.

The following syntax of **telinit** command can be used to invoke the multi-user mode when the system is working in the single user mode.

SHUTTING DOWN THE SYSTEM

- The shutdown utility performs all the necessary tasks to bring down the system safely.

General format of this shutdown utility is

```
shutdown [-options] time
```

where the *options* can be,

h	**Halts the system after shutdown**
r	**Reboots the system after shutdown**
k	**Don't really shutdown, but only warn**

ADDING A USER

The System Administrator can add / remove users to the system manually

A typical format of the 'useradd' command is,

```
useradd <username>
useradd  -u <user-id> -g <group-id>
```

The System Administrator can set an initial password for a user by using the following format of this *passwd* command

```
passwd <username>
```

DELETING A USER

The userdel command is used to delete a user from the System

General format is,

```
userdel <username>
```

This command only removes the user account but the users files remains as they are, not deleted. The system administrator can delete these files separately as required.