2

Linux Commands

Objectives:

After the completion of this chapter, you should know,

- How to interact with the system
- Directory oriented commands
- File oriented commands
- Process oriented commands
- Communication oriented commands
- Miscellaneous 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. The command simply overflows to the next line, though it is still in a single logical line.

Commands, *options* and *command_arguments* must be separated by white space(s) or tab(s) 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.

For example, you can use the command, "`wc -l -w -c a.c`" as "`wc -lwc a.c`".

The command along with its *options*, *command_arguments* is entered in one line. This line is referred as "**command line**". A command line usually ends with a new-line character. The command line completes only after the user has hit the *[Enter]* key. The \ symbol placed at the end of a line continues the command to the next line, ignoring the hit of *[Enter]* key.

Several commands may be written in a single command line. They must be separated by semicolon (;).

For example, `$ date ; who`

The important Linux commands are grouped according to their functions and explained as follows.

- Directory Oriented Commands
- File Oriented Commands

- Process Oriented Commands
- Communication Oriented Commands
- General Purpose Commands
- Pipes and Filters

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 of links 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 /

<directory_name> specifies a name of the directory whose contents are to be displayed. If the *<directory_name>* is not specified, then the contents of the current directory are displayed.

Examples

```
[bmi@kousar bmi] $ ls
bmi     desktop    maxsizefile.sh    test1.txt    text
c       java       polindrome.sh     test2.txt

[bmi@kousar bmi] $ ls - r
text    test1.txt    maxisizefile.sh    Desktop      bmi
test2.txt            polindrome.sh      java         c

[bmi$kousar bmi] $ ls - 1
total 36
drwxrwxr-x 4 bmi  bmi                  4096  Jan  10    15:36 bmi
drwxrwxr-x 2 bmi  bmi                  4096  Jan  10    12:11 c
drwxr-xr-x 2 bmi  bmi                  4096  Dec  11     2002 Desktop
drwxrwr-x  3 bmi  bmi                  4096  Jan  13    10:03 java
```

```
-rwxrw-r-- 1 bmi  bmi                214  Jan  10    15:24 maxsizefile.sh
-rwxrw-r-- 1 bmi  bmi                382  Jan  10    14:41 polindrome.sh
-rw-rw-r-- 1 bmi  bmi                 75  Jan  13    14:35 test1.txt
-rw-rw-r-- 1 bmi  bmi                397  Jan  13    14:36 test2.txt
drwxrwxr-x 2 bmi  bmi               4096  Jan  10    12:11 text
```

```
[bmi@kousar bmi] $ ls -t
Desktop       test1.txt    bmi              polindrome.sh text
text2.txt     java         maxsizefile.sh   c
```

```
[bmi@kousar bmi] $ ls -a
.             .bash_history   c                polindrome.sh       text
..            .bash_logout    Desktop          .polindrome.sh.swp  .ty.swp
.a2.swp       .bash_profile   jave             .screenrc           .Xauthority
.a.swo        .bashrc         .kde             test1.txt
.a.swp        bmi             maxsizefile.sh   test2.txt
```

```
[bmi@kousar bmi] $ ls -s
total 36
4 bmi     4 Desktop      4 maxsizefile.sh     4 test1.txt     4 text
4 c       4 java         4 polindrome.sh      4 test2.txt
```

```
[bmi@kousar bmi] $ ls -p
bmi/      Desktop        maxsizefile.sh       test1.txt       text/
c/        java/          polindrome.sh        test2.txt
```

```
[bmi@kousar bmi] $ ls -F
bmi/      Desktop        maxsizefile.sh*      test1.txt       text/
c/        java/          polindrome.sh*       test2.txt
```

```
[bmi@kousar bmi] $ ls -lt
total 36
drwxr-xr-x     2 bmi     bmi          4096  Dec  11    2002  Desktop
-rw-rw-r--     1 bmi     bmi           397  Jan  13   14:36  test2.txt
-rw-rw-r--     1 bmi     bmi            75  Jan  13   14:35  test1.txt
drwxrwxr-x     3 bmi     bmi          4096  Jan  10   10:03  java
drwxrwxr-x     4 bmi     bmi          4096  Jan  10   15:36  bmi
-rwxrw-r--     1 bmi     bmi           214  Jan  10   15.24  maxsizefile.sh
-rwxrw-r--     1 bmi     bmi           382  Jan  10   14:41  polindrome.sh
drwxrwxr-x     2 bmi     bmi          4096  Jan  10   12:11  c
drwxrwxr-x     2 bmi     bmi          4096  Jan  10   12:11  text
```

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.

Examples

```
[bmi@kousar bmi] $ ls test?.txt
text1.txt     text2.txt     test3.txt
[bmi@kousar bmi] $ ls test[1-2].txt
text1.txt     text2.txt
```

2. mkdir

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

General format is,

```
mkdir [-p] <directory_name1> <directory_name2>
```

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_name1> <directory_name2>
```

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_criteria>* (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 {}\;

If *<path_list>* and *<action>* are not specified, then *current directory* and *-print* are taken respectively as default arguments.

Examples

```
$ find /home/ibrahim -name "*.java" -print
```

This command will recursively displays all *.java* files that are stored in the directory `/home/ibrahim` including all its sub-directories.

```
$ find /home/ibrahim -mtime 5 -print
```

If the current date is 20-02-2003, then this command will display the files that have been modified on 15-02-2003.

```
$ find /home/ibrahim -mtime +5 -print
```

If the current date is 20-02-2003, then this command will display the files that have been modified before 15-02-2003.

```
$ find /home/ibrahim -mtime -5 -print
```

If the current date is 20-02-2003, then this command will display the files that have been modified after 15-02-2003.

Making changes on a file means "*modifying*". Opening / Modifying a file means "*accessing*".

7. du

This du (**disk usage**) command reports the disk spaces that are consumed by the files in a specified directory, including all its sub-directories.

General format is,

```
du [-options] [<directory_name>]
```

With no arguments, 'du' reports the disk space for the current directory. Normally the disk space is printed in units of 1024 bytes, but this can be overridden.

where *options* can be,

a Displays counts for all files, not just directories

b Displays sizes in bytes

c Displays output along with grand total of all arguments

k Displays the sizes in KiloBytes

m Displays the sizes in MegaBytes

Examples

```
[bmi@kousar bmi] $ du
12              ./.kde/Autostart
16              ./.kde
24              ./Desktop
16              ./c
12              ./text
4               ./bmi/c
4               ./bmi/text
152             ./bmi
4               ./java/ss
16              ./java
380             .
[bmi@kousar bmi] $ du /home/bmi/text
12              /home/bmi/text
[bmi@kousar bmi] $ du -a /home/bmi/text
4               /home/bmi/text/x.txt
4               /home/bmi/text/y.txt
12              /home/bmi/text
```

8. df

This df (disk free) command reports the available free space on the mounted file systems (disks).

General format is,

```
df [-options]
```

where *options* can be,

l Shows local file systems only

k Displays the sizes in KiloBytes

m Displays the sizes in MegaBytes

i Reports free, used, and percentage of used i-nodes.

This command reports the free spaces in blocks. Generally, one block is 512 bytes. The *i-nodes* entry indicates that up to the specified number of files can be created on the file system.

Examples

```
[bmi@kousar bmi] $ df
Filesystem      1k-blocks       Used     Aailable    Use%   Mounted on
/dev/hda8         1612808      63632      1467248      5%   /
/dev/hda6           23302       3489        18610     16%   /boot
```

```
/dev/hda9          1035660        7100        975952      1%   /home
/dev/hda11         1778840      742916        945560     44%   /usr
/dev/hda10         1517920       17028       1423784      2%   /var
[bmi@kousar bmi] $ df
Filesystem           Inodes      IUsed       IFree   IUse%   Mounted on
/dev/hda8            205088      17204      187884      9%   /
/dev/hda6              6024         25        5999      1%   /boot
/dev/hda9            131616       1571      130045      2%   /home
/dev/hda11           226240      45867      180373     21%   /usr
/dev/hda10           193152        501      192651      1%   /var
```

FILE ORIENTED COMMANDS

1. cat

This cat (**cat**enated -concatenate) command is used to display the contents of the specified file(s).

General format is,

```
cat [-options] <filename1> [<filename2> …]
```

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

Example

```
$ cp a.c b.c
```

The content of *a.c* is copied in to *b.c*.

3. rm

This rm (**rem**ove) 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

Examples

```
$ rm a.c
```

This command deletes the file – *a.c* from the current directory. (But current directory will still exist with other files.)

```
$ rm -f/usr/ibrahim
```

This command deletes all the files and subdirectories of the specified directory */usr/ibrahim*. Note that the directory '*ibrahim*' also will be deleted.

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>.

Example

```
$mv a.c b.c
```

Then, the file *a.c* is renamed to *b.c*.

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

Examples

```
$ wc a.c
```

It displays the number of lines, words and characters in the file – *a.c.*

```
$ wc -l a.c
```

It displays the number of lines in the file – *a.c.*

6. ln

This ln (**link**) command is used to establish an additional filename to a specified file. It doesn't mean that creating more copies of the specified file.

General format is,

```
ln <filename> <additional_filename>
```

where, *<filename>* is the name of the file for which *<additional_filename>* is to be established. The additional file names can be located on any directory. Thus, Linux allows a file to have more than one name, and yet maintain a single copy in the disk. But, changes to one of these files are also reflected to the others. If you delete one filename using *rm* command, then the other link-names will still exist.

Examples

```
[bmi@kousar bmi] $ ls -1 test1.txt
-rw-rw-r--       1 bmi          bmi           75 Jan 13 14:35 test1.txt
[bmi@kousar bmi] $ ln test1.txt test2.txt
[bmi@kousar bmi] $ ls -1 test*.txt
-rw-rw-r--       2 bmi          bmi           75 Jan 13 14:35 test1.txt
-rw-rw-r--       2 bmi          bmi           75 Jan 13 14:35 test2.txt
```

Note that the number of links for *test1.txt* and *test2.txt* are converted to *2*.

```
[bmi@kousar bmi] $ rm test1.txt
[bmi@kousar bmi] $ ls -1 test*.txt
-rw-rw-r--       1 bmi          bmi           75 Jan 13 14:35 test2.txt
```

7. 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, C program text, data, separate executable, empty* or others.

General format is,

```
file <filename>
```

Example

```
[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.

8. 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.

Examples

```
[bmi@kousar bmi] $ cat file1.txt
I am ibrahim,
What is your name?
[bmi@kousar bmi] $ cat file2.txt
I am ibrahim,
What are you doing?
[bmi@kousar bmi] $ cmp file1.txt file2.txt
file1.txt file2.txt differ: char 20, line 2
```

The file1.txt differs from file2.txt at 20th character, which occurs at the 2nd line.

9. comm

This comm (**common**) command uses two sorted files as arguments and reports what is common. It compares each line of the first file with its corresponding line in the second file. The output of this command is in three columns as follows,

column1	Contains lines unique to filename1
column2	Contains lines unique to filename2
column3	Contains lines common for both filenam1 and filename2

General format is,

```
comm [-options] <filename1> <filename2>
```

where *options* can be,

1 Suppresses listing of column1

2 Suppresses listing of column2

3 Suppresses listing of column3

Examples

```
[bmi@kousar bmi] $ cat file1.txt
I am ibrahim,
What is your name?
[bmi@kousar bmi] $ cat file2.txt
I am ibrahim,
What are you doing?
[bmi@kousar bmi] $ comm file1.txt file2.txt
                I am ibrahim,
          What are you doing?
What is your name?
```

This command displays the lines that are unique to *a.c* in first *column 1*, the lines that are unique to *b.c* in *column 2* and the lines that are common for *a.c* and *b.c* in *column 3*.

```
[bmi@kousar bmi] $ -23 file.txt file2.txt
What is your name?
```

This command displays only the lines unique to the file -*file1.txt*, other columns (*column1* and *column2*) are suppressed.

```
[bmi@kousar bmi] $ comm -1 file1.txt file2.txt
          I am ibrahim,
What are you doing?
```

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)

The ordinary files consist of a stream of data that are stored on some magnetic media. A directory does not contain any data, but keeps track of an account of all the files and sub-directories that it contains. Linux treats even physical devices as files. Such files are called special files.

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

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

The user can access a file with the above modes only if he/she has the corresponding permission. If he/she has only read permission on a file, then he/she cannot write and execute that file.

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

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

The system administrator assigns necessary file permissions to the above users.

We can know the file permissions of the files using "`ls -l`" command. This command will list the directory contents in long format including file permissions. The file permission is displayed in 10 characters width as follows,

BIT POSITION:	1	234	567	8910
MEANING:	file or directory (if d, then directory if –, then file)	rwx permission to users	rwx permission to usergroup	rwx permission to others

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*.

10. chmod

This chmod (**change mode**) command is used to change the file permissions for an existing file. We can use any one 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.

```
$chmod ugo-rwx file2
```

This command removes *read*, *write* and *execute* permissions from the *user*, *usergroup* and *others* for the file – *file2*. So, the file – *file2* will not *read, written and executed* by *user, usergroup and others*.

More than one permission can also be set with a single *chmod* command. The multiple file permission expressions must be delimited by commas (,).

```
$ chmod u+r,g-x,o+rw file1
```

This command assigns *read* permission to *users* (`u+r`), removes *execute* permission from *group* (`g-x`) and sets *read* & *write* permissions to *others* (`o+rw`).

The following command assigns *read* permission and removes *write & execute* permissions to/from *user, user group* and *others* on *file1*.

```
$ chmod ugo+r,ugo-wx file1
```

The above operation can also be achieved by using the following command.

```
$chmod ugo=r file1
```

Unlike the + or – operators, the absolute assignment (=) assigns only those permissions that are specified along with it, and removes other permissions.

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

Example

```
$ chmod 740 file1
```

Then, the *user* can *read, write* and *execute* the file – *file1*, because **7** is set as first di The *usergroup* can *only read* the file – *file1*, because **4** is set as second digit. The *other* can *read*, write and execute the file – *file1*, because **0** is set as third digit.

11. chown

This chown (**change ownership**) command is used to change the owner of a specif 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>*.

Example

```
[root@kousar bmi] # ls -l file1.txt
-rw-rw-r--       1 bmi          bmi          33 Jan 13 16:05 file1.txt
[root@kousar bmi] # chown ibrahim file1.txt
[root@kousar bmi] $ ls -l file1.txt
-rw-rw-r--       1 ibrahim      bmi          33 Jan 13 16:05 file1.txt
```

12. chgrp

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

General format is,

```
chgrp <new_groupname> <filename>
```

This command makes *<new_groupname>* as group-owner of the file specified i *<filename>*. Note that the group-owner of the file is also the *group* to which the file-owne belongs (no changes in default setting).

Example

```
[root@kousar bmi] # ls -l file1.txt
-rw-rw-r--       1 bmi          bmi          33 Jun 6 14:57 file1.txt
[root@kousar bmi] # chgrp mca file1.txt
[root@kousar bmi] $ ls -l file1.txt
-rw-rw-r--       1 bmi          mca          33 Jun 6 14:57 file1.txt
```

13. touch

This command is used to change the last modification and access time of a specified file in to the specified time.

General format is,

```
touch MMDDHHmm <filename1> [<filename2> ...]
```

where,

MM	**Month** (01 -12)
DD	**Day** (01 - 31)

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

```
[bmi@kousar bmi] $ ls -l text1.txt
-rw-rw-r--       1 bmi          bmi            102 Jan 2 10:56 test1.txt
[bmi@kousar bmi] $ date
Thu Jan 13 16:55:28 IST 2000
[bmi@kousar bmi] $ touch test1.txt

[bmi@kousar bmi] $ ls -l test1.txt
-rw-rw-r--       1 bmi          bmi            102 Jan 13 16:55 test1.txt
```

The above *touch* command changed the last modified time and last access time of the file – *file1.txt* to current date and time.

```
[bmi@kousar bmi] $ touch 01100925 test1.txt
[bmi@kousar bmi] $ ls -l test1.txt
-rw-rw-r--       1 bmi          bmi            102 Jan 10 09:25 test1.txt
```

The following command changes the last modified and access time of all the files in the current directory into the current date and time. Here, '*' represents all the files in the current directory.

```
[bmi@kousar bmi] $ touch *
```

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

Examples

```
[bmi@kousar bmi] $ cat small.txt
This is a file named SMALL.TXT
```

```
[bmi@kousar bmi] $ dd if="small.txt" of="capital.txt" conv=ucase
0+1 records in
0+1 records out
[bmi@kousar bmi] $ cat capital.txt
THIS IS A FILE NAMED SMALL.TXT
[bmi@kousar bmi] $ dd if="capital.txt" of="lower.txt" conv=lcase
0+1 records in
0+1 records out
[bmi@kousar bmi] $ cat lower.txt
this is a file named small.txt
```

15. expand

This command converts all the tabs present in the specified file into blank spaces a displays the result on the screen.

General format is,

```
expand [-i] <filename>
```

where the `-i` option converts only the initial tab into blank spaces.

Example

```
[bmi@kousar bmi] $ cat test.txt
This      is      a       file    named     test.txt
[bmi@kousar bmi] $ expand test.txt
This      is      a       file    named     test.txt
[bmi@kousar bmi] $ cat test.txt > test1.txt
[bmi@kousar bmi] $ ls -l test.txt test1.txt
-rw-rw-r--      1 bmi          bmi           49 Jan 21 09:36 test1.txt
-rw-rw-r--      1 bmi          bmi           30 Jan 21 09:35 test.txt
```

Note that the expanded file-size (*test1.txt*) is greater than the original file-size (*test.tx* because all the tabs (5) in the *test.txt* file are converted into blank spaces.

16. nl

This command numbers all non-blank lines in the specified text file and displays t same on the screen.

General format is,

```
nl <filename>
```

Example

```
[bmi@kousar bmi] $ cat test.txt
This is first line.

This is second line.


This is third line.
```

```
[bmi@kousar bmi] $ nl test.txt
     1    This is first line.

     2    This is second line.


     3    This is third line.
```

17. tac

This command reverses a file, so that the last line becomes the first line.

General format is,

```
tac <file_name>
```

Example

```
[bmi@kousar bmi] $ cat test.txt
This is the first line.
This is the second line.
This is the third line.

[bmi@kousar bmi] $ tac test.txt
This is the third line.
This is the second line.
This is the first line.
```

18. tail

This command displays the end of the specified file.

General format is,

```
tail ±n <filename>
```

If "**+n**" option is used, then the command will display from the n^{th} line to the end of the specified file. If "**-n**" option is used, then the last "n" lines of the specified file are displayed. If no option is specified, then the last 10 lines are displayed.

Examples

```
[bmi@kousar bmi] $ tail +4 employee.dat
1001        ismail        botany        bot        28-03-1965
1004        kadar         computer      cs         23-05-1988

[bmi@kousar bmi] $ tail -3 employee.dat
1003        abdulla       commerce      com        25-11-1985
1001        ismail        botany        bot        28-03-1965
1004        kadar         computer      cs         23-05-1988

[bmi@kousar bmi] $ ls -l | tail
-rwxrwxrwx     1 bmi      bmi            11 Dec 14 09:40 pgm1.sh
-rw-rw-r--     1 bmi      bmi           579 Dec 18 12:34 s1
-rw-rw-r--     1 bmi      bmi           282 Dec 19 12:41 ss
-rw-rw-r--     1 bmi      bmi         21153 Dec 19 09:21 t
-rw-rw-r--     1 bmi      bmi        211530 Dec 19 09:24 t1
-rw-rw-r--     1 bmi      bmi            31 Dec 16 09:23 test1.txt
```

```
-rw-rw-r--    1 bmi      bmi           31 Dec 14 09:39 test2.txt
-rw-rw-r--    1 bmi      bmi         1484 Dec 16 15:42 typescript
-rw-rw-r--    1 bmi      bmi           93 Dec 16 16:14 userlist.txt
-rw-rw-r--    1 bmi      bmi           46 Dec 19 12:18 www
```

This command displays the last 10 directory listing of the current directory.

19. head

This command displays the top of the specified file.

General format is,

```
head [-n] <filename>
```

If **-n** option is specified, then the first *n* lines of the file are displayed. Default value for this option is 10.

Examples

```
[bmi@kousar bmi] $ head -3 employee.dat
1005        yasin          computer       cs          15-08-1978
1002        abdulla        zoology        zoo         22-07-1956
1003        abdulla        commerce       com         25-11-1985
[bmi@kousar bmi] $ ls -1 | head
total 428
-rw-rw-r--    1 bmi      bmi         34520 Dec 24 10:28 a
-rw-rw-r--    1 bmi      bmi         16481 Dec 24 10:34 a1
-rw-rw-r--    1 bmi      bmi         19941 Dec 24 10:34 a2
drwxrwxr-x    2 bmi      bmi          4096 Dec 16 09:23 cpp
-rwxr----     1 bmi      bmi            21 Dec 16 09:54 d
drwxr-xr-x    2 bmi      bmi          4096 Dec 11 11:08 Desktop
-r--r--r--    1 bmi      bmi            21 Dec 11 09:54 dq
-rw-rw-r--    1 bmi      bmi           213 Dec 18 12:29 e
-rw-rw-r--    1 bmi      bmi            25 Dec 19 12:07 e1.dat
```

This command displays the first 10 directory listings of the current directory.

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)*.

Some of the process-oriented commands are given below.

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*.

Examples

```
$ ps
$ ps -a
$ ps -t tty3d
```

This displays the processes, which are running on the terminal - *tty3d*.

```
$ ps -u ibrahim
```

This displays the processes, which are running for the user - *ibrahim*.

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. Use *ps* command to know the PIDs of the current processes.

By default, this *kill* command uses the signal number 15 to terminate a process. But, some programs like login shell simply ignore this signal of interruption, and continue execution normally. In this case, you can use the signal number 9 (often referred as *sure kill*).

Examples

```
$kill 120
```

This command terminate the process who has the PID 120.

```
$kill -9 130
$kill -9 0
```

This command kills all the processes including the login Shell. (The Kernel itself being the first process gets the PID 0. The above command kills the Kernel, so all the processes are killed.)

3. nohup

If a user wants a process that he has executed not to die even when he logged-out from the system, you can use this nohup (**no hangup**) command. Note that normally all the processes of a user are terminated if he logs out.

General format is,

```
nohup <command>&
```

Example

```
$ nohup sort emp.doc&
1116
```

Here, 1116 is the PID of the process. Then, the command *sort emp.doc* will be executed even if you logged out from the system. The output of this command will be stored in a file named *nohup.out*.

4. at

This command is used to execute the specified Linux commands at future time.

General format is,

```
at <time>
<commands>
^d
```

(Press *[ctrl + d]* at the end)

Here, <*time*> specifies the time at which the specified <*commands*> are to be executed.

Example

```
$ at 12:00
echo "LUNCH BREAK"
^d
```

at offers the keywords – ***now, noon, midnight, today*** and ***tomorrow*** and they convey special meanings.

```
$ at noon                          ← 12:00
echo "LUNCH BREAK"
^d
```

at also offers the keywords ***hours, days, weeks, months*** and ***years***, can be used with + operator as shown in the following examples.

```
$ at 12:00 + 1 day                 ← at 12:00 tomorrow
$ at 13:00 Jan 20, 2003 + 2 days   ← at 1 pm January 20, 2004
```

The **atq** command is used to list the jobs submitted by you on at queue. This command lists the *job number, scheduled date of execution.*

General format is,

```
atq
```

The **atrm** command is used to remove a job from *at* queue.

General format is,

```
atrm <jobnumber>
```

Example

```
$ atq
7   2002-12-16 12:00 a bmi
8   2002-12-17 12:00 a bmi
9   2003-01-22 13:00 a bmi

$ atrm 8

$ atq
7   2002-12-16 12:00 a bmi
9   2003-01-22 13:00 a bmi
```

5. batch

This command is used to execute the specified commands when the system load permits (when CPU becomes nearly free).

General format is,

```
batch
<commands>
^d
```

Any job scheduled with batch also goes to the *at* queue, you can list or delete them through *atq* and *atrm* respectively.

Example

```
$ batch
sort a.c
sort b.c
^d
```

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***. In online communication, the user, to whom the message is to be sent (recipient), must be logged on the system. In offline communication, the recipient need not be logged on the system.

Some of the communication-oriented commands are given below:

1. write

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

General format is,

```
write <RecipientLoginName>
<message>
^d
```

Example

```
$ wirte ibrahim
Hello ibrahim!
How are you?
^d
```

On the execution of this command, the specified message is displayed on the terminal of the user - *ibrahim*.

If the recipient does not want to allow these messages (sent by other users) on his terminal, then he can use "**mesg n**" command. If he wants to revoke this option and want to allow any one to communicate with him, then he can use "**mesg y**" command.

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.

To check mails, give the ***mail*** command without arguments. This command will list all the incoming mails received since the latest usage of the mail command. A & symbol is displayed at the bottom. This is called **mail prompt**. Here we can issue several mail prompt commands (referred as internal commands).

Some commands which can be given in mail prompt are given below,

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
```

The specified message "*Meeting at 16:00 hrs.*" will be displayed on everyone's terminal with a beep sound like ***write***'s message, but ignoring *mesg* settings.

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

Examples

```
$ date
Mon Dec 16 15:13:10 IST 2002

$ date +%H
15

$ date +%I
03

$ date +%M
13

$ date +%S
10

$ date +%D
12/16/02

$ date +%T
15:13:10

$ date +%w
1          ← 0=Sunday, 1=Monday, 2=Tuesday, ...

$ date +%r
03:13:10 PM

$ date +%y
02
```

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
```

Example

```
$ who
ibrahim     tty1      Feb      19    10:17
sheik       tty3      Feb      19    10:20
mukash      tty8      Feb      19    11:02
```

The first column of the output represents the user names. The second column represents the corresponding terminal names and the remaining columns represents the time at which the users are logged on.

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.

Almost all of the commands offer **`--help`** option that displays a short listing of all the options.

```
$ wc --help
```

5. 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
```

You can also use **clear** command for this purpose.

Example

```
$ tput clear
```

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 meta-character. This command only works with integers, so, the division yields only integer part.

Examples

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

Then,

```
$ expr $x + $y
7

$ expr $x - $y
3

$ expr $x \* $y          ← Escape the multiplication operator
10
```

```
$ expr $x / $y
2                ← Note that decimal portion is truncated.

$ expr $x % $y
1                ← Remainder part of division.

$ expr $x + 10
15
```

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
2
```

By default, *bc* performs truncated division (integer division). If you do not want to truncate any value (requiring float division), then you have to set scale to the number of digits of precision before the operation.

```
$ bc
scale=1
8/3
2.6
scale=2
8/3
2.66

^d
```

Note that the result (2.66) is not rounded off; the actual result is 2.6666...

3 Pipes and Filters

Objectives:

After the completion of this chapter, you should know,

- Uses of Pipe
- Redirection procedure
- Commands used for filtering data and output.
- The workingness of *sed* and *gawk* utilities

INTRODUCTION

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 the command '***ls***' is sent to the '***more***' command as input. So, the directory listing of the current directory is displayed page by page (with pause).

The following two-command group displays the total number of users who are currently loged on the system.

```
$ who > userlist.txt
$ wc -l userlist.txt
```

Note that unnecessarily a file named *userlist.txt* is created. If we use pipe mechanism, there is no need to create such temporary file.

```
$ who | wc -l
```

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 sources other than the 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 < file1.txt
```

Then the file – *file1.txt* is taken as input for the command –*cat*.

Redirecting outputs:

The ">" symbol is used to redirect outputs.

Example

```
$ ls > list.doc
```

Then, the output of the command '***ls***' is stored on the file '***list.doc***'. We can also use '1>' instead of '>'.

Redirecting Error messages:

The '2>' symbol is used to redirect error messages.

Example

```
$ cat list1.doc
```

If there is no file named '*list1.doc*' in the current directory, then the error message is sent to the standard error device (usually screen). We can redirect this error message using '2>' symbol.

```
$ cat list1.doc 2> errormes.txt
```

Then, the error message, if generated, will be stored on the file - *errormes.txt*.

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. Since these commands perform some filtering operations on data, they are appropriately called as "***filters***". These filters are used to display the contents of a file in sorted order, extract the lines of a specified file that contains a specific pattern, etc.

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
+pos.n	Starts sort after the nth column of the (pos+1)th field
-pos.n	Stops sort on the nth column of the (pos+1)th field

Example

```
$ cat empname.txt
yasin
abdulla
ibrahim
abdulla
ismail

$ sort empname.txt
abdulla
abdulla
ibrahim
ismail
kadar
yasin
```

This command displays sorted contents of the file *-empname.txt* on the screen.

```
$ sort -r empname.txt
yasin
kadar
ismail
ibrahim
abdulla
abdulla
```

This command displays the sorted content, sorted in reverse alphabetical order, of the file *–empname.txt*.

```
$ sort empname.txt -o result.txt

$ cat result.txt
abdulla
abdulla
ibrahim
ismail
kadar
yasin
```

This command stores the sorted contents of "*empname.txt*" in to "*result.txt*".

```
$ cat empname1.txt
yasin
abdulla
ibrahim
```

```
$ cat empname2.txt
ismail
kadar
abdulla

$ sort empname1.txt > e1.txt
$ sort empname2.txt > e2.txt

$ cat e1.txt
abdulla
ibrahim
yasin

$ cat e2.txt
abdulla
ismail
kadar

$ sort -m e1.txt e2.txt
abdulla
abdulla
ibrahim
ismail
kadar
yasin
```

This command merges the sorted contents of the files *-e1.txt* and *e2.txt*. It is alway preferable to sort files separately before merging them.

```
$ sort empname.txt
abdulla
abdulla
ibrahim
ismail
kadar
yasin

$ sort -u empname.txt
abdulla
ibrahim
ismail
kadar
yasin
```

The sort with **-u** option removes duplicate lines from output.

Generally, sorting takes place in the order of numerals, uppercase letters and then lower case letters. As the rule, the digits, alphabets and other special characters are converted to their ASCII (*American Standard Code for Information Interchange*) value. Then, this sort arranges its input according to their ASCII value. Since sorting is based on ASCII values of the characters, the line "*20*" is less than the line "*3*"(for example). So, the result will be unexpectable. The sort with **-n** option will arrange its input according to numerical values.

```
$ cat numbers.txt
5
10
7
20

$ sort numbers.txt
10
20
5
7

$ sort -n numbers.txt
5
7
10
20
```

The *sort* with -c option is used to check whether the specified file has been sorted or not. If the file is sorted, no message will be displayed, otherwise it will indicate the unsortedness.

```
$ sort -c numbers.txt
sort: numbers.txt:2: disorder: 10

$ sort numbers.txt > n.txt
$ sort -c n.txt
$
```

If a file contains records (written in a single line) containing fields normally separated by a blank space or a tab space, and if we want to sort the file based on any particular field, then +pos & -pos options are used.

```
[bmi@fayas bmi] $ cat employee.dat
1005        yasin          computer       cs        15-08-1978
1002        abdulla        zoology        zoo       22-07-1956
1006        ibrahim        computer       cs        18-09-1987
1003        abdulla        commerce       com       25-11-1985
1001        ismail         botany         bot       28-03-1965
1004        kadar          computer       cs        23-05-1988
```

(Records are separated by a tab space)

```
[bmi@fayas bmi] $ sort employee.dat
1001        ismail        botany        bot        28-03-1965
1002        abdulla       zoology       zoo        22-07-1956
1003        abdulla       commerce      com        25-11-1985
1004        kadar         computer      cs         23-05-1988
1005        yasin         computer      cs         15-08-1978
1006        ibrahim       computer      cs         18-09-1987
[bmi@fayas bmi] $ sort +1 employee.dat
1003        abdulla       commerce      com        25-11-1985
1002        abdulla       zoology       zoo        22-07-1956
1006        ibrahim       computer      cs         18-09-1987
1001        ismail        botany        bot        28-03-1965
1004        kadar         computer      cs         23-05-1988
1005        yasin         computer      cs         15-08-1978
```

The argument **+1** indicates that sorting should start after skipping the first field. So, the above sort command does sorting after skipping the first field (employee number field). If you want to start from third field, use **+2** as argument.

```
[bmi@fayas bmi] $ sort +2 employee.dat
1001        ismail        botany        bot        28-03-1965
1003        abdulla       commerce      com        25-11-1985
1005        yasin         computer      cs         15-08-1978
1006        ibrahim       computer      cs         18-09-1987
1004        kadar         computer      cs         23-05-1988
1002        abdulla       zoology       zoo        22-07-1956
```

The argument +1 indicates that sorting should start after skipping the first field. And the argument -2 indicates that sorting should **stop** after the second field. The following command sorts the *employee.dat* according to empname (second field). Note that this output differs from the output of "***sort +1 employee.dat***".

```
[bmi@fayas bmi] $ sort +1 -2 employee.dat
1002        abdulla       zoology       zoo        22-07-1956
1003        abdulla       commerce      com        25-11-1985
1006        ibrahim       computer      cs         18-09-1987
1001        ismail        botany        bot        28-03-1965
1004        kadar         computer      cs         23-05-1988
1005        yasin         computer      cs         15-08-1978
```

The default field separator for sort command is a blank space or a tab space. You can specify any other character as field separator by using **-t** option.

```
[bmi@fayas bmi] $ cat employee.dat
1005|yasin|computer|cs|15-08-1978
1002|abdulla|zoology|zoo|22-07-1956
1006|ibrahim|computer|cs|18-09-1987
1003|abdulla|commerce|com|25-11-1985
1001|ismail|botany|bot|28-03-1965
1004|kadar|computer|cs|23-05-1988
```

```
[bmi@fayas bmi] $ sort employee.dat
1001|ismail|botany|bot|28-03-1965
1002|abdulla|zoology|zoo|22-07-1956
1003|abdulla|commerce|com|25-11-1985
1004|kadar|computer|cs|23-05-1988
1005|yasin|computer|cs|15-08-1978
1006|ibrahim|computer|cs|18-09-1987
[bmi@fayas bmi] $ sort -t"|" +1 -2 employee.dat
1002|abdulla|zoology|zoo|22-07-1956
1003|abdulla|commerce|com|25-11-1985
1006|ibrahim|computer|cs|18-09-1987
1001|ismail|botany|bot|28-03-1965
1004|kadar|computer|cs|23-05-1988
1005|yasin|computer|cs|15-08-1978
```

The above command treats the character "|" as field separator.

If you want to sort fourth field (dept. code) as primary key and second field (empname) as secondary key, use the following *sort* command.

```
[bmi@fayas bmi] $ cat employee.dat
1005        yasin          computer        cs        15-08-1978
1002        abdulla        zoology         zoo       22-07-1956
1006        ibrahim        computer        cs        18-09-1987
1003        abdulla        commerce        com       25-11-1985
1001        ismail         botany          bot       28-03-1965
1004        kadar          computer        cs        23-05-1988
[bmi@fayas bmi] $ sort +3 -4 +1 employee.dat
1001        ismail         botany          bot       28-03-1965
1003        abdulla        commerce        com       25-11-1985
1006        ibrahim        computer        cs        18-09-1987
1004        kadar          computer        cs        23-05-1988
1005        yasin          computer        cs        15-08-1978
1002        abdulla        zoology         zoo       22-07-1956
```

The argument +3 indicates that the sort should start after the third field and should stop after the fourth field (i.e., exactly fourth field). The third argument +1 indicates that resumption of sort starts after the first field.

You can also specify the character position within a specified field to be the beginning and ending positions of sort. The following command will sort the file *employee.dat* according to last four digits of the fifth field (year -after the fourth field, in which after the sixth character).

```
[bmi@fayas bmi] $ sort +4.6 employee.dat
1002        abdulla        zoology         zoo       22-07-1956
1001        ismail         botany          bot       28-03-1965
1005        yasin          computer        cs        15-08-1978
1003        abdulla        commerce        com       25-11-1985
1006        ibrahim        computer        cs        18-09-1987
1004        kadar          computer        cs        23-05-1988
```

The following command will sort the file *employee.dat* according to first two digits of the fifth field (date).

```
[bmi@fayas bmi] $ sort +4.0 -4.3 employee.dat
1005        yasin          computer        cs        15-08-1978
1006        ibrahim        computer        cs        18-09-1987
1002        abdulla        zoology         zoo       22-07-1956
1004        kadar          computer        cs        23-05-1988
1003        abdulla        commerce        com       25-11-1985
1001        ismail         botany          bot       28-03-1965
```

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

Example

```
[bmi@kousar bmi] $ cat employee.dat
1005        yasin          computer        cs        15-08-1978
1002        abdulla        zoology         zoo       22-07-1956
1006        ibrahim        computer        cs        18-09-1987
1003        abdulla        commerce        com       25-11-1985
1001        ismail         botany          bot       28-03-1965
1004        kadar          computer        cs        23-05-1988
[bmi@kousar bmi] $ grep "cs" employee.dat
1005        yasin          computer        cs        15-08-1978
1006        ibrahim        computer        cs        18-09-1987
1004        kadar          computer        cs        23-05-1988
[bmi@kousar bmi] $ grep -1 "ibrahim" employee.dat
1002        abdulla        zoology         zoo       22-07-1956
1006        ibrahim        computer        cs        18-09-1987
1003        abdulla        commerce        com       25-11-1985
```

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>

```
[bmi@kousar bmi] $ grep "com*" employee.dat
1005        yasin          computer        cs        15-08-1978
1006        ibrahim        computer        cs        18-09-1987
1003        abdulla        commerce        com       25-11-1985
1004        kadar          computer        cs        23-05-1988
[bmi@kousar bmi] $ grep "8$" employee.dat
1005        yasin          computer        cs        15-08-1978
1004        kadar          computer        cs        23-05-1988
[bmi@kousar bmi] $ ls -l | grep "^d"
drwxrwxr-x   2 bmi          bmi             4096 Dec 16 09:23 cpp
drwxr-xr-x   2 bmi          bmi             4096 Dec 11 11:08 Desktop
```

This command displays only directory entries. Note that directory entries are starting with "d".

egrep

The egrep (**extended global search for regular expression**) command offers additional features than grep. Multiple patterns can be searched by using pipe symbol (|).

Example

```
[bmi@kousar bmi] $ grep "ibrahim|smail" employee.dat
                                                    ← No Result
[bmi@kousar bmi] $ egrep "ibrahim|smail" employee.dat
1006        ibrahim        computer        cs        18-09-1987
1001        ismail         botany          bot       28-03-1965
```

fgrep

This fgrep (**fixed grep**) command works like grep, but it does not accept *regular expressions* unlike *grep*.

Examples

```
[bmi@kousar bmi] $ fgrep "cs" employee.dat
1005        yasin          computer        cs        15-08-1978
1006        ibrahim        computer        cs        18-09-1987
1004        kadar          computer        cs        23-05-1988
[bmi@kousar bmi] $ fgrep "c*" employee.dat
[bmi@kousar bmi] $
```

3. uniq

This uniq (**unique**) command is used to handle duplicate lines in a file. If this command is used without any option, it displays the lines by eliminating duplicate lines.

General format is,

```
uniq [-option] <filename>
```

where *options* can be,

u Displays only the non-repeated lines

d Displays only the duplicated lines

c Displays each line by eliminating duplicate lines, and prefixing the number of times it occurs.

Examples

```
[bmi@kousar bmi] $ cat employee.dat
1005        yasin          computer       cs        15-08-1978
1002        abdulla        zoology        zoo       22-07-1956
1002        abdulla        zoology        zoo       22-07-1956
1006        ibrahim        computer       cs        18-09-1987
1006        ibrahim        computer       cs        18-09-1987
1006        ibrahim        computer       cs        18-09-1987
1003        abdulla        commerce       com       25-11-1985
1001        ismail         botany         bot       28-03-1965
1004        kadar          computer       cs        23-05-1988
[bmi@kousar bmi] $ uniq employee.dat
1005        yasin          computer       cs        15-08-1978
1002        abdulla        zoology        zoo       22-07-1956
1006        ibrahim        computer       cs        18-09-1987
1003        abdulla        commerce       com       25-11-1985
1001        ismail         botany         bot       28-03-1965
1004        kadar          computer       cs        23-05-1988
[bmi@kousar bmi] $ uniq -u employee.dat
1005        yasin          computer       cs        15-08-1978
1003        abdulla        commerce       com       25-11-1985
1001        ismail         botany         bot       28-03-1965
1004        kadar          computer       cs        23-05-1988
[bmi@kousar bmi] $ uniq -d employee.dat
1002        abdulla        zoology        zoo       22-07-1956
1006        ibrahim        computer       cs        18-09-1987
[bmi@kousar bmi] $ uniq -c employee.dat
   1    1005        yasin          computer       cs        15-08-1978
   2    1002        abdulla        zoology        zoo       22-07-1956
   3    1006        ibrahim        computer       cs        18-09-1987
   1    1003        abdulla        commerce       com       25-11-1985
   1    1001        ismail         botany         bot       28-03-1965
   1    1004        kadar          computer       cs        23-05-1988
```

4. 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.

You can also use more specialized command -**less**, instead of *more*. This *less* command has the same syntax and functions of *more*.

```
$ less a.c
```

```
$ ls | less
```

Note: For this paging purpose, you can also use key combinations. The *[ctrl+s]* is used to stop scrolling of the screen output. The *[ctrl+q]* is used to resume scrolling of the screen output. But it is not an ideal method.

5. pr

This command displays the contents of the specified file adding with suitable headers and footers. This command can be used with *lpr* command for neat hard copies. The Header part consists of the last modification date and time along with file-name and page number.

General format is,

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

where *options* can be,

`-l <number>`	It changes the page size to specified <*number*> of lines (By default, the page size is 66 lines)
`-<number>`	Prepares the output in <*number*> columns
`-n`	Numbers lines
`-t`	Turns off the heading at the top of the page

Examples

```
[bmi@kousar bmi] $ cat employee.dat
1005          yasin           computer        cs          15-08-1978
1002          abdulla         zoology         zoo         22-07-1956
1003          abdulla         commerce        com         25-11-1985
1001          ismail          botany          bot         28-03-1965
1004          kadar           computer        cs          23-05-1988
[bmi@kousar bmi] $ pr employee.dat
2002-12-19 10:13              employee.dat                  Page     1

1005          yasin           computer        cs          15-08-1978
1002          abdulla         zoology         zoo         22-07-1956
1003          abdulla         commerce        com         25-11-1985
1001          ismail          botany          bot         28-03-1965
1004          kadar           computer        cs          23-05-1988
```

—Note that remaining lines are empty—

```
[bmi@kousar bmi] $ pr -n employee.dat
2002-12-19 10:13                employee.dat                Page      1

1     1005        yasin         computer      cs          15-08-1978
2     1002        abdulla       zoology       zoo         22-07-1956
3     1003        abdulla       commerce      com         25-11-1985
4     1001        ismail        botany        bot         28-03-1965
5     1004        kadar         computer      cs          23-05-1988
```

—Note that remaining lines are empty—

```
[bmi@kousar bmi] $ pr -2 employee.dat
2002-12-19 10:13                employee.dat                Page      1

1005        yasin         computer      cs          15-08-1978
1002        abdulla       zoology       zoo         22-07-1956
1003        abdulla       commerce      com         25-11-1985
```

—Note that remaining lines are empty—

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

Examples

```
[bmi@kousar bmi] $ cat employee.dat
1005        yasin         computer      cs          15-08-1978
1002        abdulla       zoology       zoo         22-07-1956
1003        abdulla       commerce      com         25-11-1985
1001        ismail        botany        bot         28-03-1965
1004        kadar         computer      cs          23-05-1988
[bmi@kousar bmi] $ cut -f 3 employee.dat
computer
zoology
commerce
botany
computer
[bmi@kousar bmi] $ cut -f 3-5 employee.dat
computer        cs          15-08-1978
zoology         zoo         22-07-1956
commerce        com         25-11-1985
botany          bot         28-03-1965
computer        cs          23-05-1988
```

```
[bmi@kousar bmi] $ cut -f 1-3,5 employee.dat
1005          yasin           computer        15-08-1978
1002          abdulla         zoology         22-07-1956
1003          abdulla         commerce        25-11-1985
1001          ismail          botany          28-03-1965
1004          kadar           computer        23-05-1988
[bmi@kousar bmi] $ cat e.dat
1005|yasin|computer|cs|15-08-1978
1002|abdulla|zoology|zoo|22-07-1956
1003|abdulla|commerce|com|25-11-1985
1001|ismail|botany|bot|28-03-1965
1004|kadar|computer|cs|23-05-1988
[bmi@kousar bmi] $ cut -d"|" -f 3-4 e.dat
computer|cs
zoology|zoo
commerce|com
botany|bot
computer|cs
[bmi@kousar bmi] $ cut -c 1-4 e.dat
1005
1002
1003
1001
1004
[bmi@kousar bmi] $ cut -c 6-10,15,17 e.dat
yasinpt
abdulol
abdulom
ismaitn
kadarpt
```

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> ...
```

Examples

```
[bmi@kousar bmi] $ cat e1.dat
1005
1002
1003
1001
1004
```

```
[bmi@kousar bmi] $ cat e2.dat
computer
zoology
commerce
botany
computer
[bmi@kousar bmi] $ paste e1.dat e2.dat
1005          computer
1002          zoology
1003          commerce
1001          botany
1004          computer
```

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.

Examples

```
[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
```

4

VI Editor

Objectives:

After the completion of this chapter, you should know,

- How to work with vi editor
- The three modes of vi editor
- Insert, delete, replace text, cursor movement, search, yanking, redo and undo, search commands, etc.

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 Berkeley.

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>]
```

A status line at the bottom of the screen (25th line) shows the filename, current line and character position in the edited file.

```
vi +<linenumber> <filename>
```

- Edits the file specified in *<filename>* and places the cursor on the *<linenumber>*th line.

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
<n>dd	Deletes specified number of lines (n) from the current line
d*[Enter]*	Deletes current line and the following line
d0	Deletes all the characters from the beginning of the current line to previous character of cursor position
D	Deletes all the characters from the current character to the end of the current line
d/<pattern>/*[Enter]*	Deletes all characters but before to specified ***pattern*** occurs

d)	Deletes all the characters from current cursor position to end of the current sentence
d(	Deletes all the characters from current cursor position to the beginning of the current sentence
d}	Deletes all the characters from the current cursor position to the end of the current paragraph
d{	Deletes all the characters before the cursor position to beginning of the current paragraph

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) **n**Y	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>

SUMMARY

Vi works on three different modes.

The text should be inserted in the insert mode.

All the commands should be entered in command mode.

One can enter into the command mode from the Insert mode by pressing the [Esc] key, and from Ex mode by pressing [Enter] key.

The ex mode commands should be entered prefixed by colon (:).

SELF-ASSESSMENT QUESTIONS

1. Write a note on different modes of *vi* editor.
2. Explain ex mode commands.
3. How do you insert and delete text in *vi* editor?
4. List and illustrate cursor movement commands.

5

Shell Programming

Objectives:

After the completion of this chapter, you should know,

- What is a Shell script?
- Shell variables
- Exporting Shell variables
- Escape mechanisms
- Shell Meta characters
- Control statements
- Iterative statements
- Uses of *sleep* and *basename* commands

INTRODUCTION

The shell is a Linux system mechanism for the communication between the users and the system. And it is a command interpreter that interprets the user commands and conveys them to the kernel, which executes them. There are many Shells available in Linux including ***Bash*** (Bourne again shell) Shell, ***Bourne*** Shell, ***C*** Shell (tcsh), ***Korn*** Shell (public domain korn shell – pdksh). Among them, the most popular one is *bash*, the super set of the *Bourne Shell*. This bash Shell is compatible with Bourne. The prompt of this bash Shell is **$** symbol. And we have already seen several commands, which can be used in this Shell. Let us discuss the features of this Shell in detail.

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***. These scripts (files) act like batch files in *Disk Operating System (DOS)*. Note that the name of the Shell script must not be a Linux command name.

The Shell script can be executed in two methods. One method is using the **sh** command. Another one is to grant executable permission to the file using ***chmod*** command and then type the *filename* at the $ prompt.

General format of ***sh*** command is,

```
sh [-v] <script_filename>
```

The **-v** option causes the Shell to echo (display) each command before it is executed.

Example

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

```
pwd
date
```

Linux does not require file extension. A file extension ***.sh*** may be used for user's convenience.

To execute this script (file) using *sh* command,

```
[bmi@kousar bmi] $ sh disp.sh
/home/bmi
Fri Dec 27 10:10:43 IST 2002
```

(Results of the commands - *pwd* and *date* – on the *disp.sh* are displayed.

The *sh* command creates a *child shell process*. This new Shell reads the commands from the specified file – *disp.sh*, executes them and returns the control to the parent Shell.

```
[bmi@kousar bmi] $ sh -v disp.sh
pwd
/home/bmi
date
Fri Dec 27 10:09:22 IST 2002
```

To execute the script by giving execute permission, use the following commands.

```
[bmi@kousar bmi] $ chmod u+x disp.sh
[bmi@kousar bmi] $ .\disp.sh
/home/bmi
Fri Dec 27 10:11:05 IST 2002
```

This chmod command gives executable permission to user. Then *disp.sh* becomes executable and acts as a command. To execute a command that is stored in the current directory, we must include a **.** (**.** represents current directory) in the **PATH** Shell variable. Because the Shell first searches the working directory for executing a command, only when a dot (**.**) is specified at the start of the PATH variable. But it is not advisable to include a dot in the beginning of the PATH variable, because it may execute a ***Trojon Horse*** program (a program that has the same name as a standard Linux command that causes severe consequences). So, if a program has to be run in the working directory, it can be performed by preceding the command name with **./**, for instance ./disp.sh

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**". An environment is an area in memory where we can place definitions of Shell variables so that they are accessible from the programs. We can list these system variables (built-in variables) corresponding to a user, using the "**set**" command.

Example

```
[bmi@kousar bmi] $ set
BASH=/bin/bash
BASH_ENV=/home/bmi/.bashrc
BASH_VERSINFO=([0]="2" 1]="04" [2]="21" [3]="1"
BASH_VERSION='2.04.21(1,-release'
COLORS=/etc/DIR_COLOUR
COLUMNS=97
DIRSTACK=()
EUID=502
GROUPS=()
HISTFILE=/home/bmi/.bash_history
HISTFILESIZE=1000
HISTSIZE=1000
HOME=/home/bmi
HOSTNAME=kousar
HISTTYPE=i386
IFS='
'
INPUTRC=/etc/inputrc
KDEDIR=/usr
LANG=en_US
LESSOPEN='|/usr/bin/lesspipe.sh %s'
LINES=24
LOGNAME=bmi
...
...
...
```

Left side of the "=" sign is the *system variable*, right sides of the "=" sign is its corresponding value. In Linux, the system variables are specified by upperecases for convenience.

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
IFS	A list of characters that are used to separate words in a command line including *[space], [tab], [newline]* character (IFS - Internal Field Separators)
EXINIT	Initialisation instructions for the *vi* and *ex* editors
PS1	Prompt String 1, normally PS1 is "**$**"
PS2	Prompt String 2, it is used when Linux thinks that you have started a new line without hitting *[Enter]* key, normally PS2 is "**>**".
TERM	Determines the terminal type being used

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.

Example

```
[bmi@kousar bmi] $ echo Hi!
Hi!

[bmi@kousar bmi] $ echo MOHAMED IBRAHIM
MOHAMED IBRAHIM

[bmi@kousar bmi] $ netpay=14000
[bmi@kousar bmi] $ echo $netpay
14000

[bmi@kousar bmi] $ echo name
name

[bmi@kousar bmi] $ echo $name
                                    ← null value occurs

[bmi@kousar bmi] $ name="MOHAMED IBRAHIM"
[bmi@kousar bmi] $ echo $name
MOHAMED IBRAHIM

[bmi@kousar bmi] $ echo Hi! $name, your salary is $netpay
Hi! MOHAMED IBRAHIM, your salary is 14000
```

Whenever the Shell finds continuous spaces and tabs in the command line, it compresses them to a single space. That's why, when you issue the following echo command, you find the output to be compressed.

```
[bmi@kousar bmi] $ echo MOHAMED      IBRAHIM
MOHAMED IBRAHIM
```

To preserve the spaces, you have to place the string within quotes.

```
[bmi@kousar bmi] $ echo "MOHAMED      IBRAHIM"
MOHAMED      IBRAHIM
```

The echo command will generate a carriage return by default. You can use the echo command with **-n** option to avoid such automatic generation of the carriage return.

```
[bmi@kousar bmi] $ echo "Dear" ; echo "ibrahim"
Dear
ibrahim

[bmi@kousar bmi] $ echo -n "Dear" ; echo "ibrahim"
Dear ibrahim
```

The **-e** option of the echo command enables the interpretation of the following character sequences (escape sequences) in the argument string.

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

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

Example

```
[bmi@kousar bmi] $ echo -e "\t Your name is \n IBRAHIM"
        Your name is
IBRAHIM
```

export

When a new Shell (child Shell) is created, the newly created Shell is ignorant of the parent Shell's variables. That is, the variables created in one Shell are local to it. These environment variables can also be exported to the child Shells using this *export* command. This makes the new Shell becomes aware of the old Shell's variables.

General format is,

```
export <shell_variable>
```

Example

```
[bmi@kousar bmi] $ name=ibrahim
[bmi@kousar bmi] $ echo $name
ibrahim
[bmi@kousar bmi] $ sh          ← Creates a new Shell
sh-2.04$ echo $name
                               ← null value is defined. The value 'ibrahim' is not available.
sh-2.04$ exit                  ← exit from child Shell and return to old Shell
exit
[bmi@kousar bmi] $ echo $name
ibrahim
```

Now, let us use the *export* command.

```
[bmi@kousar bmi] $ name=ibrahim
[bmi@kousar bmi] $ echo $name
ibrahim
[bmi@kousar bmi] $ export name
[bmi@kousar bmi] $ sh
sh-2.04$ echo $name
ibrahim            ← The value of old Shell's variable is available in this child Shell.
sh-2.04$ name=mohamed          ← Assign new value
sh-2.04$ echo $name
mohamed
sh-2.04$ exit
exit
```

Double quotes (" ")

The double quotes turn off the special meaning of all the enclosed characters (including #, *, ?, &, |, :, (,), [,], ^) except $, ` and \.

Example

```
[bmi@kousar bmi] $ name=ibrahim
[bmi@kousar bmi] $ echo "Your name is $name"
Your name is ibrahim
```

Back Slash (\)

The back slash turns off the special meaning of a single character that immediately follows it.

Examples

```
[bmi@kousar bmi] $ name=ibrahim
[bmi@kousar bmi] $ echo "Your name is $name"
Your name is ibrahim

[bmi@kousar bmi] $ echo "Your name is \$name"
Your name is $name                    ← The symbol \ turns off the meaning of $

[bmi@kousar bmi] $ echo `date`
Sat Dec 28 10:46:05 IST 2002

[bmi@kousar bmi] $ echo \`date\`
`date`
```

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. But the *bash* Shell has no limit on the number of positional parameters. However, the using of shift command is the best method to access positional parameters.

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
[bmi@kousar bmi] $ echo $?
0
[bmi@kousar bmi] $ cat s1.sh
cat: s1.sh: No such file or directory
[bmi@kousar bmi] $ echo $?
1
```

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
```

Note that the `` `date` `` will produce the current date as its output, Sat Dec 28 11:47:21 IST 2002. So, the positional parameters have their values based on this output.

A Shell script example:

```
set `who am i`
user=$1
terminal=$2
set `date`
echo "Dear $user, now time is $4"
echo "please logout from $terminal"
```

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 conditions

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
```

If the specified *<conditional_command>* is executed successfully (exit status is 0), then the commands in between then and else (*if block*) will be executed. Otherwise, if the specified *<conditional_command>* is not executed successfully (exit status is nonzero), then the commands in between else and fi (*else block*) will be executed.

Consequences of *if..else* statements may be written using a *if..elif* statement.

Example

```
if grep printf a.c > /dev/null
then
  echo "The pattern -printf is found in the file -a.c."
else
  echo "The pattern -printf is not found in the file -a.c."
fi
```

Here, if the command *–grep printf a..c* is successfully executed (the pattern *printf* is found in *a.c* – exit status is 0) then the message *"The pattern -printf is found in the file –a.c"* is displayed. Otherwise if the command is failure (the pattern *printf* is not found in *a.c* – exit status is nonzero) then the message *"The pattern –printf is not found in the file –a.c"* is displayed.

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

Example

```
if test $# -ne 2
  echo Usage: $0 <filename1> <filename2>
  echo This script requires two arguments
fi
```

Note that $# refers the number command line arguments.

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

Example

```
if test -r a.c
then
  echo The file -a.c exists and it is readable
fi
```

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.

Examples

```
(i)  if test -r a.c -a -w a.c
     then
       echo "a.c is both readable and writeable"
     fi
(ii) if test ! -r a.c
     then
       echo "a.c is not readable"
     fi
```

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

```
[bmi@kousar bmi] $ cat e1.dat
1001 ibrahim
1002 yasin

[bmi@kousar bmi] $ cat e2.dat
1003 kadar
1001 ibrahim

[bmi@kousar bmi] $ cat e3.dat
1004 abdulla
1005 kumar

[bmi@kousar bmi] $ cat e4.dat
1001 ibrahim
1006 ashok

[bmi@kousar bmi] $ cat whiledemo.sh
pattern=$1
shift
while grep $pattern $1 > /dev/null
do
  echo "The pattern is found in the file -$1"
  shift
done
echo "The pattern is not found in the file -$1"
echo "So, the loop ends without examining remaining arguments"
```

```
[bmi@kousar bmi] $ chmod u+x whiledemo.sh
[bmi@kousar bmi] $ ./whiledemo.sh ibrahim e1.dat e2.dat e3.dat e4.dat
The pattern is found in the file -e1.dat
The pattern is found in the file -e2.dat
The pattern is not found in the file -e3.dat
So, the loop ends without examining remaining arguments
```

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).

Example

```
until who|grep ibrahim > /dev/null
do
 sleep 5
done
echo "The user -ibrahim is logged on."
```

This Shell script will delay the process until the user *-ibrahim* logs on i.e. the loop will be executed until the user *-ibrahim* logs on. The '**sleep**' command is used to wait the process at the specified number of seconds mentioned as its argument. (`sleep` - do nothing)

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
```

```
(ii) for i in $*
     do
      cat $i
     done
```

If we pass `e1.dat`, `e2.dat`, and `e3.dat` as arguments to the above Shell script, then the contents of these files are displayed.

```
(iii) for i in *
      do
       echo FILE NAME: $i
       cat $i
      done
```

This Shell script will display the contents of all the files in the current directory.

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]
```

Example

```
[bmi@kousar bmi] $ cat continuedemo.sh
while true
do
  echo "Enter a word"
  read word
  if test -z "$word"
  then
    echo "Null string"
    continue
  fi
echo "Given word is $word"
break
done

[bmi@kousar bmi] $ chmod u+x continuedemo.sh
[bmi@kousar bmi] $ ./continuedemo.sh
```

```
Enter a word
ibrahim
Given word is ibrahim

[bmi@kousar bmi] $ ./continuedemo.sh
Enter a word
                                    ← Press [Enter] key i.e. null value
Null string
Enter a word
                                    ← Press [Enter] key i.e. null value
Null string
Enter a word
mohamed
Given word is mohamed
```

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
```

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."
> }

[bmi@kousar bmi] $ disp
THE .TXT FILES IN THE CURRENT DIRECTORY ARE,
-rw-rw-r--    1 bmi      bmi            31 Jan  2  09:39  capital.txt
-rw-rw-r--    1 bmi      bmi             0 Dec 31  13:00  e1.txt
-rw-rw-r--    1 bmi      ibrahmi        43 Mar 18   2002  file1.txt
-r--------    1 bmi      bmi            34 Dec 16   2002  file2.txt
```

```
-rw-rw-rw-    1 bmi      bmi           67 Dec 16   2002 letter.txt
-rw-rw-r--    1 bmi      bmi           31 Jan  2 09:39 lower.txt
-rw-rw-r--    1 bmi      bmi        14920 Dec 16   2002 ipr.txt
-rw-rw-r--    1 bmi      bmi           39 Dec 27   2002 result.txt
-rw-rw-r--    1 bmi      bmi           31 Jan  2 09:38 small.txt
-rw-rw-r--    1 bmi      bmi           31 Dec 16   2002 test1.txt
-rw-rw-r--    1 bmi      bmi           31 Dec 14   2002 test2.txt
-rw-rw-r--    1 bmi      bmi           93 Dec 16   2002 userlist.txt
...over.
```

On typing the name of the function - ***disp*** at the $ prompt, the output of the commands specified in the Shell script are executed.

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*".

```
[bmi@kousar bmi] $ cat delayeg
echo "Hello"
sleep 60
echo "ibrahim"
[bmi@kousar bmi] $ sh delayeg
Hello
ibrahim
```

basename

This command extracts the "*base*" file-name from an absolute path-name.

General format is,

```
basename <absolute_pathname>
```

Example

```
[bmi@kousar bmi] $ basename /home/bmi/employee.dat
employee.dat
```

When this command is used with a string (as second argument), it strips off that string from the first argument.

```
[bmi@kousar bmi] $ basename employee.dat ee.dat
employ
```

You know that when moving multiple files using ***mv*** command, the last argument must be a directory. Linux doesn't accept the following sequence.

$mv *.txt *.doc i.e. trying to convert all *.txt* extensions into *.doc*.

7

System Administration

Objectives:

After the completion of this chapter, you should know,

- The functions of the System Administrator
- Operations on booting the system
- Adding and removing users and user-groups
- Managing devices
- Mounting the file system
- Compression and Backup utilities
- Remote system accessing using *telnet* and *ftp*

INTRODUCTION

The system administration, maintained by System Administrator and he is also called as Super user, includes the management of the entire system such as installing and removing terminals and workstations, installing printers and backup activities, adding and removing new users and user group, providing security, managing disks and file systems and monitoring the network.

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 **#**. The System Administrator can create files in or delete any file from any directory. He can also read from, write to or execute any file.

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** (**s**ubstitute **u**ser) command from a user's Shell

When the *su* command was used to become the System Administrator, the user can return to the normal mode by terminating the Shell by pressing *[Ctrl+d]* or by giving the *exit* command.

Most of the commands that are used as a System Administrator are stored in the **/sbin** and **/usr/sbin** directories. The commands can be executed by giving their full path names or

by including these `/sbin` and `/usr/sbin` directories in the PATH when logged as System Administrator.

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 **0**. The kernel executes the **`/etc/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. And also decides what programs are to be run to perform various startup and maintenance functions.

Then, the *init* invokes the **`/etc/getty`** file. This file has the various parameters for communication between the terminal and the computer system. By using these parameters, the login prompt is displayed.

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.

```
telinit 5
```

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

The *time* specifies the absolute or relative time that when the *shutdown* command starts its function.

Example

```
# shutdown -h 16:30
```

This command halts the system at 4:30 PM. (Absolute time)

```
# shutdown -r +40
```

This command reboots the system after 40 minutes of issuing this command. (Relative time)

The **halt** command can be used to halt the system. The **reboot** command reboots the system. Note that *halt* will do the function of *-h* option of *shutdown* command and the *reboot* does the function of *-r* option of the *shutdown* command.

Always do proper shutdown. Because, Linux stores data in memory buffers that are periodically written into the disk to speed up disk access. If there is no proper shutdown, the contents of the disk buffers are lost. The shutdown command forces the buffers to be written into the disk.

This sync (synchronize) command can be used to flush the file-system buffers i.e., it writes the information stored in the buffers (main memory) into the appropriate files on the disk. Generally, to improve the system performance, the kernel does not write the information to the disk immediately; but it keeps the data in main memory and writes them into the disk when it desires. This *sync* command forces the kernel to write any data buffered in memory out to disk. And this command can be used to save the data from system crashes.

ADDING A USER

The System Administrator can add / remove users to the system manually, or he can also use tools such as *useradd*, to do the same job which avoids risks.

A typical format of the 'useradd' command is,

```
useradd <username>
useradd   -u <user-id> -g <group-id>
          -c <comment>  -d <user's_home_directory>
          -s <default_lonin_Shell>  <username>
```

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

```
passwd <username>
```

If the SA uses *adduser* command for creating a user, then he must set an initial password for the user using the above format of the *passwd* command, or he must delete the password entry of the user in */etc/passwd* file. However, the logged user can change this password using the *passwd* command without any arguments.

This *useradd* command inserts the necessary entries in the files - */etc/passwd*, */etc/shadow* (if shadow password system is used) and */etc/group*.

All the new user's informations are stored in the /etc/passwd file as a single line in the following format:

```
Username:Password:User_ID:Group_ID:Comment:Home_directory:User's_Shell
```

Username : Maximum eight characters-length name of the user

Password	: The system places 'x' in this field if the *shadow* password system is used, otherwise the user's encrypted password is stored in this field.
User ID	: A number that identifies the user
Group ID	: A number that identifies the group that the user belongs
Comment	: Comments about user's original name, designation, address etc.
Home Directory	: It is the directory that the system will place the corresponding user after the successful login
User's Shell	: This specifies the Shell type for the user. If /bin/tcsh is specified, then the System will provide C Shell for the user after the successful login. If no Shell type is specified, then /bin/bash (Bash Shell) is taken by default.

A typical entry in the */etc/passwd* file is shown below.

```
bmi:x:100:25:IBRAHIM:/home/kgr:/bin/bash
```

For every line in */etc/passwd* file, there is a corresponding entry in */etc/shadow* (if the System uses shadow password) and in */etc/group*.

The entry in the */etc/shadow* includes user_name, encrypted password, the number of days that the user-account is valid.

```
User's_name:Encrypted_password:...
```

Then, the user's name is inserted at the corresponding group that is stored in the */etc/group* file. A typical line in the */etc/group* file has the following format:

```
Group_name:Password:Group_ID:Users_belonging_to_this_group
```

Example

```
[root@kousar /root]# adduser kgr
[root@kousar /root]# passwd kgr
Changing password for user kgr
New UNIX password:
Retype new UNIX password:
password: all authentication tokens updated successfully
```

(Note that the password characters are not echoed.)

Then, a user account for the user - *kgr* is created and his home directory is */home/kgr*, login Shell is bash. All these informations are default excluding *username* and *password*.

By changing the username in the */etc/passwd*, the System Administrator can make a user account inaccessible temporarily. The System Administrator must change the changed username into the original name in the */etc/passwd* file in order to bring that user account accessible.

DELETING A USER

The `userdel` command is used to delete a user from the System, which removes all the entries pertaining to the specified user from the three files - */etc/passwd*, */etc/shadow*, and */etc/group*.

General format is,

```
userdel <username>
```

This command only removes the user account, but the user's files remains as they are, not deleted. The System Administrator can delete these files separately if required.

The System Administrator can also create the necessary groups and add the users to these groups, instead of leaving to the *adduser* command.

The `groupadd` command is used to add a *group* to the System, which has the following syntax.

```
groupadd [-g <group_id>] <group_name>
```

The *<group_id>* is a non-negative numeric value, which refers the group by number. The default for <group_id> is to use the smallest numeric value greater than 500 and greater than every other group. The values between 0 and 499 are typically reserved for System accounts.

The `groupdel` command is used to delete an existing group. But all the users who belong to that group must be removed before using of this command.

General format of this command is,

```
groupdel <group_name>
```

Example

```
# mkdir /home/kumar
# groupadd -g 550 mca
# useradd -u 1050 -g 550 -c KUMAR -d /home/kumar -s /bin/bash kumar
# passwd kumar
```

(Set initial password for kumar)

Then, the user named *kumar* is created, who belongs to the group *mca*.

MANAGINE DEVICES

Linux treats all the hardware devices as files. These special files are stored in the directory `/dev`.

These devices can be grouped into two types: **Character devices** and **Block devices**.

A character device is a device, which uses serial line for data transfer. We can read / write a sequence of characters from / to those devices. The *tape drives* and *printers* are examples for character devices. Nowadays, the character devices are removed by Block devices that provide random access. Through which, we can read / write blocks of data and it allows access to all parts of the devices equally. Hence, these devices are also called as ***random access devices.***

The content of the */dev* directory is listed below by using the `ls -l` command.

```
[root@kousar bmi]# ls -1 /dev
total 180
crw--------   1 root     root     10,    10  Mar 24    2001 adbmouse
crw-r---r--   1 root     root     10,   175  Mar 24    2001 agpgart
crw--------   1 root     root     10,     4  Mar 24    2001 amigamouse
crw--------   1 root     root     10,     7  Mar 24    2001 amigamousel
crw--------   1 root     root     10,   134  Mar 12   11:10 apm_bios
crw--------   1 root     root     10,     5  Mar 24    2001 atarimouse
crw--------   1 root     root     10,     3  Mar 24    2001 atibm
crw--------   1 root     root     10,     3  Mar 24    2001 atimouse
```