UNIT-4

Compressing, decompressing and achieving files

- Compressing , decompressing and achieving files: Gzip, gunzip, Tar, Zip and unzip
- Environment variables: Environment variables, Alias, Inline command editing, miscellaneous features, Initialization script.
- Communication commands: Finger, Talk, Mesg, Mailx, Pine, Write, Wall

Unit –4 Compressing, decompressing and achieving files

4.1 Compressing and Uncompressing files:

It is useful to compress files into one file so that they use less disk space and download faster via the Internet.

There is distinction between an archive file and a compressed file. An archive file is a collection of files and directories that are stored in one file. The archive file is not compressed and it uses the same amount of disk space as all the individual files and directories combined.

A compressed file is a collection of files and directories that are stored in one file and stored in a way that uses less disk space than all the individual files and directories combined. You can compress files that you do not use very often or files that you want to save but do not use anymore. You can even create an archive file and then compress it to save disk space.

An archive file is not compressed, but a compressed file can be an archive file.

➢ **Command: TAR**

The compressed file will have the extension of .tar

A ".tar" file is not a compressed files, it is actually a collection of files within a single file uncompressed.

Creating a tar file:

```
tar -cvwf file1.tar myfile.txt
```

The above command would create a file named file1.tar in the directory you currently are in.

Extracting the files from a tar file:

```
tar -xvwf file1.tar
```

In the above example command the system would uncompress (untar) the file1.tar file in the current directory.

➢ **Gzip and Gunzip:**

It is used to Compress or uncompress files.

⇨ ***Compressing using Gzip:***

```
gzip file.txt
```

In the above example command this would compress the file.txt file as file.txt.gz in the current directory.

⇨ ***Uncompressing using Gunzip:***

```
gunzip  file.txt.gz
```

In the above example command it would extract the file.txt from file.txt.gz.

⇨ **Zip and unzip:**

Zip and Unzip are compression and decompression utilities respectively.

Zip puts one or more compressed files into a single "zip file" along with information about the files, including the name, path if requested, date and time last modified, protection, and check information to verify the fidelity of each entry.

```
zip newfile oldfile
```

This commands takes an existing file, here oldfile, and zips it under the name specified in newfile. Remember that the default suffix, if none is specified, will be .zip. This allows specifying suffixes other than ".zip".

To zip up all of the files in your current directory, type:

```
zip mydir *
```

This will create the file "mydir.zip" and put all the files in the current directory in mydir.zip in a compressed form. This will not include any subdirectories in the current directory.

To zip up an entire directory, including subdirectories, the command:

```
zip -r mydir
```

will create the file "mydir.zip" containing all the files and directories in the directory "mydir" that is in the current directory. The "r" option means recursive through the directory structure.

Unzip is the command to decompress these files.

To unzip a file named myfile.zip

```
unzip myfile.zip
```

This will unzip all the files in the current directory that were zipped into the file.

4.2 Environment Variables:

➢ **System Variables:**

These variables as system-level variables, as they are used to configure your entire system, setting values such as the location of executable commands on your system

Some of the system variables are as follows:

PS1 and PS2:

The **PS1** and **PS2** variables contain the primary and secondary prompt symbols, respectively. The primary prompt symbol for the BASH shell is a dollar sign ($).

You can change the prompt symbol by assigning a new set of characters to the PS1 variable. In the next example, the shell prompt is changed to the -> symbol:

```
$ PS1= '->'
-> export PS1
->
```

The **PS2** variable holds the secondary prompt symbol, which is used for commands. The default secondary prompt is >. The added command lines begin with the secondary prompt instead of the primary prompt. You can change the secondary prompt just as easily as the primary prompt, as shown here:

```
$ PS2="@"
```

⇨ **PATH:**

The **PATH** variable contains a series of directory paths separated by colons. Each time a command is executed, the paths listed in the **PATH** variable are searched one by one for that command.

For example, the **ls** command resides on the system in the directory **/bin**. This directory path is one of the directories listed in the **PATH** variable. Each time you execute the ls command, this path is searched and the ls command located.

The system defines and assigns **PATH** an initial set of pathnames. In Linux, the initial pathnames are **/bin** and **/usr/bin**.

The shell can execute any executable file, including programs and scripts you have created. For this reason, the **PATH** variable can also reference your working directory; so if you want to execute one of your own scripts or programs in your working directory, the shell can locate it.

```
$ echo $PATH
/bin:/usr/sbin:
```

You can add any new directory path you want to the **PATH** variable. You can place these new shell script commands in a directory you create and then add that directory to the **PATH** list.

For example:

The user user1 adds a new directory, called mydir, to the **PATH**. In this example, an evaluation of **HOME** is also used to add the user's **home** directory in the new directory's pathname.

```
$ PATH=$PATH:$HOME/mydir:
$ export PATH
$ echo $PATH
/bin:/usr/bin::/home/user1/mydir
```

⇨ **HOME:**

The **HOME** variable contains the pathname of your home directory. Your home directory is determined by the parameter administrator when your account is created.

In the next example, the **echo** command displays the contents of the **HOME** variable:

```
$ echo $HOME
/home/user1
```

The **HOME** variable is often used when you need to specify the absolute pathname of your home directory.

⇨ **LOGNAME:**

This variable is assigned the name of the user.

Both LOGNAME and USER should be set to the username.

Examples:

```
LOGNAME=user1
  To display the username
  echo $LOGNAME
```

➢ **Command line arguments:**

Command line argument is used to get the input from the user, when user runs the shell script. Command line arguments are treated as special variables within the script, the reason I am calling them variables is because they can be changed with the shift command.

The command line arguments are enumerated in the following manner $0, $1, $2, $3, $4, $5, $6, $7, $8 and $9. $0 is special in that it corresponds to the name of the script itself. $1 is the first argument; $2 is the second argument and so on.

gedit myfile.sh

it will open a text editor

add the following code

```
echo $1
echo $2
```

save the file and run your shell script as sh myfile.sh first second *output will be*

```
first
second
```

As well as the command line arguments there are some special builtin variables:

1) **$#**

   $# counts the number of command line arguments. It is useful for controlling if conditions and loop constructs that need to process each parameter.

```
if [ $# ]
then
  echo "command line arguments found"
else
  echo "no command line arguments supplied"
fi
```

2) **$@**

   It expands to all the parameters separated by spaces. It is useful for passing all the parameters to some other function or program.

```
echo  "Enter the word"
read w
for i in $@
do
grep $w $i
done
```

   run this script as sh filename.sh file1 file2

   file1 and file2 should be created before running this code.

   This script will search for a word in multiple files entered through command line arguments.

3) **$$** expands to the process id of the shell innovated to run the script. It is useful for creating unique temporary filenames relative to this instantiation of the script.

4.3 Communication Commands:

⇨ **finger:**

It displays information about the user. It displays more extended information than who.

```
$ finger [username]
```

finger -b -p user1 - Would display the following information about the user user1.

Login name: user1 In real life: User1

On since Jan 01 12:30:16 on pts/1 from domain.user1.com

⇨ **talk:**

This command is used communicate to another user.

talk person [tty]

Talk is a visual program which copies lines from your terminal to that of another user.

⇨ **mesg:**

mesg command is used to give write access to your terminal

mesg [y|n]

mesg manages the access to your terminal by others. It's typically used to allow or disallow other users to write to your terminal

mesg y Allow write access to your terminal.

mesg n Disallow write access to your terminal.

⇨ **mailx:**

It is used for sending and receiving mail through interactive message processing system.

mailx info@computerworld.com - Start a new mail message to be sent to support at Computer World.

⇨ **pine:**

Pine is a command line program for Internet News and Email.

Syntax:

pine [address , address]

address Send mail to address this will open pine directly into the message composer.

⇨ **Write:**

write command in Linux is used to send a message to another user. The write utility allows a user to communicate with other users, by copying lines from one user's terminal to others. When you run the write command, the user you are writing to gets a message of the form:

Message from yourname@yourhost on yourtty at hh:mm ...

Any further lines the user enter will be copied to the specified user's terminal. If the other user wants to reply, they must run write as well. When you are done, type an end-of-file or interrupt character. The other user will see the message 'EOF' indicating that the conversation is over.

Syntax:

write user [tty]

write command without any option: It will print the general syntax of the write. With the help of this, the user will get a generalized idea about how to use this command since there nothing like help option for the write command.

⇨ **Wall:**

wall is a command-line utility that displays a message on the terminals of all logged-in users. The messages can be either typed on the terminal or the contents of a file. wall stands for write all, to send a message only to a specific user use the write command.

Usually, system administrators send messages to announce maintenance and ask users to log out and close all open programs. The messages are shown to all logged-in users with a terminal open. Users using a graphical desktop environment with no terminal open will not see the messages. Each user can control the write access to its terminal with the mesg utility. When the superuser invokes the wall command, all users receive the messages, no matter their mesg settings.

Broadcasting a Message:

The syntax for the wall command is as follows:

wall [OPTIONS] [<FILE>|<MESSAGE>]

If no file is specified wall reads the message from the standard input.

The most straightforward way to broadcast a message is to invoke the wall command with the message as the argument.

☺ ☺ ☺ ☺ ☺