Expt. No.3:-Implement setting of global and local environment variable, shell environment variables.

Environment Variables

The bash shell uses a feature called *environment variables* to store information about the shell session and the working environment. This feature also allows us to store data in memory that can be easily accessed by any program or script running from the shell.
There are two types of environment variables in the bash shell:
■ Global variables
■ Local variables

Global environment variables

Global environment variables are visible from the shell session, and any child processes that the
Shell spawns. Local variables are only available in the shell that creates them. The Linux system sets several global environment variables when we start our bash session. The system environment variables always use all capital letters to differentiate them from normal user environment variables.

To view the global environment variables, we use the printenv command:

```
$ printenv
HOSTNAME=testbox.localdomain
TERM=xterm
SHELL=/bin/bash
HISTSIZE=1000
SSH CLIENT=192.168.1.2 1358 22
OLDPWD=/home/rich/test/test1
SSH TTY=/dev/pts/0
USER=rich
LS COLORS=no=00:fi=00:di=00;34:ln=00;36:pi=40;33:so=00;35:
bd=40;33;01:cd=40;33;01:or=01;05;37;41:mi=01;05;37;41:ex=00;32:
*.cmd=00;32:*.exe=00;32:*.com=00;32:*.btm=00;32:*.bat=00;32:
*.sh=00;32:*.csh=00;32:*.tar=00;31:*.tgz=00;31:*.arj=00;31:
*.taz=00;31:*.lzh=00;31:*.zip=00;31:*.z=00;31:*.Z=00;31:
*.gz=00;31:*.bz2=00;31:*.bz=00;31:*.tz=00;31:*.rpm=00;31:
*.cpio=00;31:*.jpg=00;35:*.gif=00;35:*.bmp=00;35:*.xbm=00;35:
*.xpm=00;35:*.png=00;35:*.tif=00;35:
MAIL=/var/spool/mail/rich
PATH=/usr/kerberos/bin:/usr/lib/ccache:/usr/local/bin:/bin:/usr/bin:
/home/rich/bin
INPUTRC=/etc/inputrc
PWD=/home/rich
LANG=en US.UTF-8
SSH ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass
SHLVL=1
HOME=/home/rich
LOGNAME=rich
CVS RSH=ssh
SSH CONNECTION=192.168.1.2 1358 192.168.1.4 22
LESSOPEN=|/usr/bin/lesspipe.sh %s
G BROKEN FILENAMES=1
=/usr/bin/printenv
$
```

There are lots of global environment variables that get set for the bash shell. Most of them are set by the system during the login process.
To display the value of an individual environment variable, we use the echo command. When referencing an environment variable, we must place a dollar sign before the environment variable name:

```
$ echo $HOME
/home/mhssp
$
```

The global environment variables are also available to child processes running under the current shell session:

```
$ bash
$ echo $HOME
/home/mhssp
$
```

In this example, after starting a new shell using the bash command, we displayed the current value of the HOME environment variable, which the system sets when we log into the main shell.

Local environment variables

Local environment variables, as their name implies, can be seen only in the local process in which they are defined.

Unfortunately there isn't a command that displays only local environment variables. The **set** command displays the entire environment variables set for a specific process. However, this also includes the global environment variables.

Here's the output from a sample set command:

```
$ set
BASH=/bin/bash
BASH ARGC=()
BASH ARGV=()
BASH LINENO=()
BASH SOURCE=()
BASH VERSINFO=([0]="3" [1]="2" [2]="9" [3]="1" [4]="release"
[5]="i686-redhat-linux-gnu")
BASH VERSION='3.2.9(1)-release'
COLORS=/etc/DIR COLORS.xterm
COLUMNS=80
CVS RSH=ssh
DIRSTACK=()
EUID=500
GROUPS=()
G BROKEN FILENAMES=1
HISTFILE=/home/rich/.bash history
HISTFILESIZE=1000
HISTSIZE=1000
HOME=/home/rich
HOSTNAME=testbox.localdomain
HOSTTYPE=i686
IFS=$' \t\n'
INPUTRC=/etc/inputrc
LANG=en US.UTF-8
LESSOPEN='|/usr/bin/lesspipe.sh %s'
LINES=24
LOGNAME=rich
LS COLORS='no=00:fi=00:di=00;34:ln=00;36:pi=40;33:so=00;35:bd=40;33;
01:cd=40;33;01:or=01;05;37;41:mi=01;05;37;41:ex=00;32:*.cmd=00;32:
*.exe=00;32:*.com=00;32:*.btm=00;32:*.bat=00;32:*.sh=00;32:
*.csh=00;32:*.tar=00;31:*.tgz=00;31:*.arj=00;31:*.taz=00;31:
*.lzh=00;31:*.zip=00;31:*.z=00;31:*.Z=00;31:*.gz=00;31:*.bz2=00;31:
*.bz=00;31:*.tz=00;31:*.rpm=00;31:*.cpio=00;31:*.jpg=00;35:
*.gif=00;35:*.bmp=00;35:*.xbm=00;35:*.xpm=00;35:*.png=00;35:
*.tif=00;35:'
MACHTYPE=i686-redhat-linux-gnu
MAIL=/var/spool/mail/rich
MAILCHECK=60
OPTERR=1
OPTIND=1
OSTYPE=linux-gnu
```

```
PATH=/usr/kerberos/bin:/usr/lib/ccache:/usr/local/bin:/bin:/usr/bin:
/home/rich/bin
PIPESTATUS=([0]="0")
PPID=3702
PROMPT COMMAND='echo -ne
"\033]0;${USER}@${HOSTNAME%%.*}:${PWD/#$HOME/~}"; echo -ne "\007"'
PS1='[\u@\h \W]\$ '
PS2='> '
PS4='+ '
PWD=/home/rich
SHELL=/bin/bash
SHELLOPTS=braceexpand:emacs:hashall:histexpand:history:
interactive-comments:monitor
SHLVL=2
SSH ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass
SSH CLIENT='192.168.1.2 1358 22'
SSH CONNECTION='192.168.1.2 1358 192.168.1.4 22'
SSH TTY=/dev/pts/0
TERM=xterm
UID=500
USER=rich
=-H
consoletype=pty
$
```

We notice that all of the global environment variables seen from the **printenv** command appear in the output from the **set** command. However, there are quite a few additional environments variables that now appear. These are the local environment variables.

Setting Environment Variables

We can set our own environment variables directly from the bash shell.

Setting local environment variables

Once we start a bash shell (or spawn a shell script), we are allowed to create local variables that are visible within our shell process. We can assign either a numeric or a string value to an environment variable by assigning the variable to a value using the equal sign:

```
$ test=testing
$ echo $test
testing
$
```

Now any time we need to reference the value of the test environment variable, just reference it by the name $test.

If we need to assign a string value that contains spaces, we shall need to use a single quotation mark to determine the beginning and the end of the string:

```
$ test=testing a long string
-bash: a: command not found
$ test='testing a long string'
$ echo $test
testing a long string
$
```

Without the single quotation marks, the bash shell assumes that the next character is another command to process. For the local environment variable we defined, we used lower-case letters, while the system environment variables we've seen so far have all used upper-case letters.

This is a standard convention in the bash shell. This helps distinguish the personal environment variables from the system environment variables.

It's extremely important that there are no spaces between the environment variable name, the equal sign, and the value. If we put any spaces in the assignment, the bash shell interprets the value as a separate command:

```
$ test2 = test
-bash: test2: command not found
$
```
Once you set a local environment variable, it's available for use anywhere within our shell process. However, if we spawn another shell, it's not available in the child shell:
```
$ bash
$ echo $test
$ exit
exit
$ echo $test
testing a long string
$
```
In this example we started a child shell. As we can see, the **test** environment variable is not available in the child shell (it contains a blank value). After we exited the child shell and returned to the original shell, the local environment variable was still available.
Similarly, if we set a local environment variable in a child process, once we leave the child process the local environment variable is no longer available:
```
$ bash
$ test=testing
$ echo $test
testing
$ exit
exit
$ echo $test
$
```
The test environment variable set in the child shell doesn't exist when we go back to the parent shell.

Setting global environment variables

Global environment variables are visible from any child processes created by the process that sets the global environment variable. The method used to create a global environment variable is to create a local environment variable, then export it to the global environment.
This is done by using the export command:
```
$ echo $test
testing a long string
$ export test
$ bash
$ echo $test
testing a long string
$
```
After exporting the local environment variable test, we started a new shell process and viewed the value of the test environment variable. This time, the environment variable kept its value, as the export command made it global.

Removing Environment Variables

We can also remove an existing environment variable by using the **unset** command:
```
$ echo $test
testing
$ unset test
$ echo $test
$
```
When referencing the environment variable in the unset command, remember not to use the dollar sign.
If you're in a child process and unset a global environment variable, it only applies to the child process. The global environment variable is still available in the parent process:
```
$ test=testing
$ export test
$ bash
$ echo $test
testing
$ unset test

$ echo $test
$ exit
exit
$ echo $test
testing
$
```

In this example we set a local environment variable called test, then exported it to make it a global environment variable. We then started a child shell process and checked to make sure that the global environment variable test was still available. Next, while still in the child shell, we used the unset command to remove the global environment variable test, then exited the child shell.
Now back in the original parent shell, we checked the test environment variable value, and it is still valid.

Shell Environment Variables

There are specific environment variables that the bash shell uses by default to define the system environment. Since the bash shell is a derivative of the original Unix Bourne shell, it also includes environment variables originally defined in that shell.

The environment variables the bash shell provides that are compatible with the original Unix Bourne shell are shown below.

Variable	**Description**
CDPATH	A colon-separated list of directories used as a search path for the cd command.
HOME	The current user's home directory.
IFS	A list of characters that separate fields used by the shell to split text strings.
MAIL	The filename for the current user's mailbox. The bash shell checks this file for new mail.
MAILPATH	A colon-separated list of multiple filenames for the current user's mailbox. The bash shell checks each file in this list for new mail.
OPTARG	The value of the last option argument processed by the getopts command.
OPTIND	The index value of the last option argument processed by the getopts command.
PATH	A colon-separated list of directories where the shell looks for commands.
PS1	The primary shell command line interface prompts string.
PS2	The secondary shell command line interface prompts string.

By far the most important environment variable in this list is the PATH environment variable. When we enter a command in the shell command line interface (CLI), the shell must search the system to find the program. The PATH environment variable defines the directories it searches looking for commands. PATH environment variable looks like this:

```
$ echo $PATH
/usr/mhssp/bin:/usr/lib/ccache:/usr/local/bin:/bin:/usr/bin:/home/rich/bin
$
```

This shows that there are six directories where the shell looks for commands. Each directory in the PATH is separated by a colon. There's nothing at the end of the PATH variable indicating the end of the directory listing. We can add additional directories to the PATH simply by adding another colon, and adding the new directory. The PATH also shows the order in which it looks for commands.

Expt. No.4:-

☐ Create users, groups .Set permissions and ownership.
☐ View the /etc/passwd file and describe its syntax.
☐ View the /etc/shadow file and describe its syntax.
☐ View the /etc/group file and describe its syntax.

There are three types of accounts on a Linux system:

1. **Root account:** This is also called superuser and would have complete and unfettered control of the system. A superuser can run any commands without any restriction. This user should be assumed as a system administrator.
2. **System accounts:** System accounts are those needed for the operation of system-specific components for example mail accounts and the sshd accounts. These accounts are usually needed for some specific function on your system, and any modifications to them could adversely affect the system.
3. **User accounts:** User accounts provide interactive access to the system for users and groups of users. General users are typically assigned to these accounts and usually have limited access to critical system files and directories.

Create users, groups .Set permissions and ownership.

There are three main user administration files:

1. **/etc/passwd:** Keeps user account and password information. This file holds the majority of information about accounts on the Linux system.
2. **/etc/shadow:** Holds the encrypted password of the corresponding account. Not all the system support this file.
3. **/etc/group:** This file contains the group information for each account.

We can check all the above files using **cat** command.

Following are commands available on the majority of Linux systems to create and manage accounts and groups:

Command	Description
Useradd	Adds accounts to the system.
Usermod	Modifies account attributes.
Userdel	Deletes accounts from the system.
Groupadd	Adds groups to the system.
groupmod	Modifies group attributes.
Groupdel	Removes groups from the system.

Creating a Group

We would need to create groups before creating any account otherwise we would have to use existing groups. All the groups are listed in */etc/groups* file.

All the default groups would be system account specific groups and it is not recommended to use them for ordinary accounts. So following is the syntax to create a new group account:

```
groupadd[-g gid[-o]][-r][-f] groupname
```

Here is the detail of the parameters:

Option	Description
-g GID	The numerical value of the group's ID.
-o	This option permits to add group with non-unique GID
-r	This flag instructs groupadd to add a system account
-f	This option causes to just exit with success status if the specified group already exists. With -g, if specified GID already exists, other (unique) GID is chosen

groupname	Actual group name to be created.

If we do not specify any parameter then system would use default values.
Following example would create *mhssp & mhssp1* group with default values:

```
$ groupadd mhssp
$ groupadd mhssp1
```

Modifying a Group:

To modify a group, use the **groupmod command** syntax:

```
$ groupmod-n new_modified_group_name old_group_name
```

To change the mhssp1 group name to mhssp2, type:

```
$ groupmod -n mhssp1 mhssp2
```

Here is how you would change the GID to 678:

```
$ groupmod -g 545 mhssp2
```

Deleting a Group:

To delete an existing group, all we need are the groupdel command and the group name.

```
$ groupdel mhssp2
```

This removes only the group, not any files associated with that group. The files are still accessible by their owners.

Creating an User Account

Following is the syntax to create a user's account:

```
Useradd -d homedir -g groupname –m  -s shell -u userid username
```

Here is the detail of the parameters:

Option	**Description**
-d homedir	Specifies home directory for the account.
-g groupname	Specifies a group account for this account.
-m	Creates the home directory if it doesn't exist.
-s shell	Specifies the default shell for this account.
-u userid	We can specify a user id for this account.
userid	Actual account name to be created

If we do not specify any parameter then system would use default values. The useradd command modifies the /etc/passwd, /etc/shadow, and /etc/group files and creates a home directory.
Following is the example which would create an account *test* setting its home directory to*/home/test* and group as *mhssp*. This user would have Korn Shell assigned to it.

```
$ useradd  -d /home/test  -g mhssp  -s /bin/ksh test
```

Once an account is created we can set its password using the **passwd** command as follows:

```
$ passwd test
Changing password for user test.
New LINUX password:
Retype new LINUX password:
passwd: all authentication tokens updated successfully.
```

Modifying a User Account:

The **usermod** command enables us to make changes to an existing account from the command line. It uses the same arguments as the useradd command, plus the -l argument, which allows us to change the account name.

For example, to change the account name *test* to *test20* and to change home directory accordingly, we need to issue following command:

```
$ usermod -d /home/test20 -m -l test test20
```

Deleting a User Account:

The **userdel** command can be used to delete an existing user. This is a very dangerous command if not used with caution.

There is only one argument or option available for the command: .r, for removing the account's home directory and mail file.

For example, to remove account *test20*, we would need to issue following command:

```
$ userdel -r test20
```

If you want to keep home directory for backup purposes, omit the -r option.

Changing permissions

The chmod command allows us to change the security settings for files and directories. The format of the chmod command is:

chmod *options mode file*

The mode parameter allows us to set the security settings using either **octal** or **symbolic** mode. The octal mode settings are pretty straightforward; just use the standard three-digit octal code we want the file to have:

```
$ chmod 760 newfile
$ ls -l newfile
-rwxrw---- 1 test mhssp 0 Feb 06 9:16  newfile*
$
```

The octal file permissions are automatically applied to the file indicated. The symbolic mode permissions are not so easy to implement.

The format for specifying permission in **symbolic mode** is:

[ugoa...][[+-=][rwxXstugo...]

The first group of characters defines to whom the new permissions apply:

- u for the user
- g for the group
- o for others (everyone else)
- a for all of the above

Next, a symbol is used to indicate whether we want to add the permission to the existing permissions (+), subtract the permission from the existing permission (−), or set the permissions to the value (=).

Finally, the third symbol is the permission used for the setting. We may notice that there are more than the normal rwx values here. The additional settings are:

- X to assign execute permissions only if the object is a directory or if it already had execute permissions
- s to set the UID or GID on execution
- t to save program text
- u to set the permissions to the owner’s permissions
- g to set the permissions to the group’s permissions
- o to set the permissions to the other’s permissions

Using these permissions looks like this:

```
$ chmod o + r newfile
$ ls -l newfile
-rwxrw-r-- 1 test  mhssp 0 Feb 06 10:10  newfile*
$
```

The o + r entry adds the read permission to whatever permissions the everyone security level already had.

```
$ chmod u -x newfile
$ ls -l newfile
-rw-rw-r-- 1 test  mhssp 0 Feb 06 10:10  newfile
$
```

The u -x entry removes the execute permission that the user already had.

Changing ownership

Sometimes we need to change the owner of a file, such as when someone leaves an organization or a developer creates an application that needs to be owned by a system account when it's in production. Linux provides two commands for doing that. The chown command to makes it easy to change the owner of a file, and the chgrp command allows us to change the default group of a file.

The format of the chown command is:

chown options owner[.group] file

We can specify either the login name or the numeric UID for the new owner of the file:

```
# chown ali newfile
# ls -l newfile
-rw-rw-r-- 1 ali  mhssp 0 Feb 06 10:10  newfile *
#
```

The chown command also allows us to change both the user and group of a file:

```
# chown ali .mhssp1 newfile
# ls -l newfile
-rw-rw-r-- 1 ali mhssp1 0 Feb 06 10:10  newfile *
#
```

We can just change the default group for a file:

```
# chown .mhssp newfile
# ls -l newfile
-rw-rw-r-- 1 ali mhssp 0 Feb 06 10:10  newfile *
#
```

If our Linux system uses individual group names that match user login names, we can change both with just one entry:

```
# chown test. newfile
# ls -l newfile
-rw-rw-r-- 1 test test 0 Feb 06 10:10  newfile *
#
```

The chown command uses a few different option parameters. The -R parameter allows us to make changes recursively through subdirectories and files, using a wildcard character. The –h parameter also changes the ownership of any files that are symbolically linked to the file.
Only the root user can change the owner of a file. Any user can change the default group of a file, but the user must be a member of the groups the file is changed from and to.

The **chgrp** command provides an easy way to change just the default group for a file or directory:

```
$ chgrp  shared  newfile
$ ls -l newfile
-rw-rw-r-- 1 test shared 0 Feb 06 10:10  newfile *
$
```

View the /etc/passwd file and describe its syntax

What is Password file?

Passwd file is a text file that contains a list of the system's accounts, giving for each account some useful information like user ID, group ID, home directory, shell, etc. Often, it also contains the encrypted passwords for each account. It should have general read permission (many utilities, like ls use it to map user IDs to user names), but write access only for the superuser (root).
All three files are located in /etc directory and we will see each one this file detailed

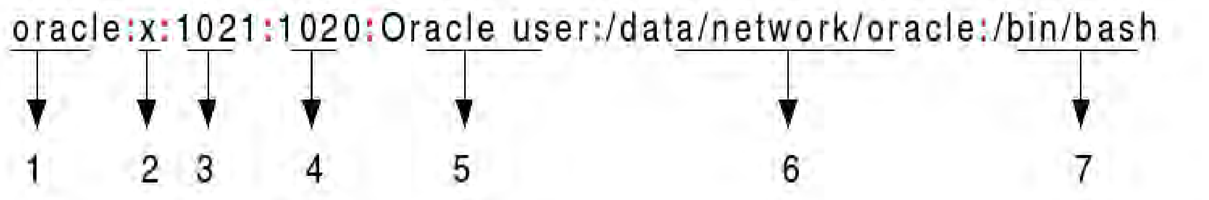
/etc/passwd

/etc/passwd file stores essential information, which is required during login i.e. user account information. There is one entry per line, and each line has the format:

Password file format (Syntax)

account:password:UID:GID:GECOS:directory:shell

The /etc/passwd contains one entry per line for each user (or user account) of the system. Each record contains multiple fields, all fields are separated by a colon (:) symbol. Total seven fields as follows. Generally, passwd file entry looks as follows:


oracle:x:1021:1020:Oracle user:/data/network/oracle:/bin/bash
1
2
3
4
5
6
7

1. **Username**: It is used when user logs in. It should be between 1 and 32 characters in length.
2. **Password**: An x character indicates that encrypted password is stored in /etc/shadow file.
3. **User ID (UID)**: Each user must be assigned a user ID (UID). UID 0 (zero) is reserved for root and UIDs 1-99 are reserved for other predefined accounts. Further UID 100-999 are reserved by system for administrative and system accounts/groups.
4. **Group ID (GID)**: The primary group ID (stored in /etc/group file)
5. **User ID Info (**real name (also known as the GECOS field): The comment field. It allow you to add extra information about the users such as user's full name, phone number etc.
6. **Home directory**: The absolute path to the directory the user will be in when they log in. If this directory does not exists then users directory becomes /
7. **Command/shell (**default shell): The absolute path of a command or shell (/bin/bash). Typically, this is a shell.

View /etc/passwd

/etc/passwd is only used for local users only. To see list of all users, enter:

$ cat /etc/passwd

Write output.

To search for a username called test enter:

```
$ grep test /etc/passwd
```

Write output.

/etc/passwd file permission

The permission on the /etc/passwd file should be read only to users (-rw-r--r--) and the owner must be root:

```
$ ls -l /etc/passwd
```

Output:

```
-rw-r--r-- 1 root root 2659 Sep 17 01:46 /etc/passwd
```

Reading /etc/passwd file

You can read /etc/passwd file using the while loop and IFS separator as follows:

```
#!/bin/bash
# seven fields from /etc/passwd stored in$f1,f2...,$f7
#
While IFS=: read -r f1 f2 f3 f4 f5 f6 f7
do
echo "User $f1 use $f7 shell and stores files in $f6 directory."
Done < /etc/passwd
```

☐ View the /etc/shadow file and describe its syntax.

What is Shadow file?

Shadow file contains the encrypted password information for user's accounts and optional the password aging information.

/etc/shadow

Shadow file format

```
test:Ep6mckrOLChF.:10063:0:99999:7:::
```

If shadow passwords are being used, the `/etc/shadow` file contains users' encrypted passwords and other information about the passwords. Its fields are colon-separated as for `/etc/passwd`, and are as follows:

- username
- encrypted password
- Days since Jan 1, 1970 that password was last changed
- Days before password may be changed
- Days after which password must be changed
- Days before password is to expire that user is warned
- Days after password expires that account is disabled
- Days since Jan 1, 1970 that account is disabled
- A reserved field

View /etc/shadow

```
administrator@Alisir:~$ cat /etc/shadow
```

write output

password is stored in /etc/shadow file

Your encrpted password is not stored in /etc/passwd file. It is stored in /etc/shadow file. Almost, all modern LINUX operating systems use some sort of the shadow password suite, where /etc/passwd has asterisks (*) instead of encrypted passwords, and the encrypted passwords are in /etc/shadow which is readable by the superuser only.

☐ View the /etc/group file and describe its syntax.

What is Group file?

/etc/group is a text file which defines the groups to which users belong under LINUX operating system. Under Linux multiple users can be categorized into groups. Linux file system permissions are organized into three classes, user, group, and others. The use of groups allows additional abilities to be delegated in an organized fashion, such as access to disks, printers, and other peripherals. This method, amongst others, also enables the Superuser to delegate some administrative tasks to normal users.

/etc/group file

It stores group information or defines the user groups i.e. it defines the groups to which users belong. There is one entry per line, and each line has the following format (all fields are separated by a colon (:)

Group file format

group_name:passwd:GID:user_list

```
cdrom:x:24:vivek,student13,raj
_____ _ __ ________________
|     | |   |
|     | |   |
1     2 3   4
```

Where,

1. group_name: It is the name of group. If we run ls -l command, you will see this name printed in the group field.
2. Password: Generally password is not used, hence it is empty/blank. It can store encrypted password or print x. This is useful to implement privileged groups.
3. Group ID (GID): Each user must be assigned a group ID. You can see this number in your /etc/passwd file.
4. Group List: It is a list of user names of users who are members of the group. The user names, must be separated by commas.

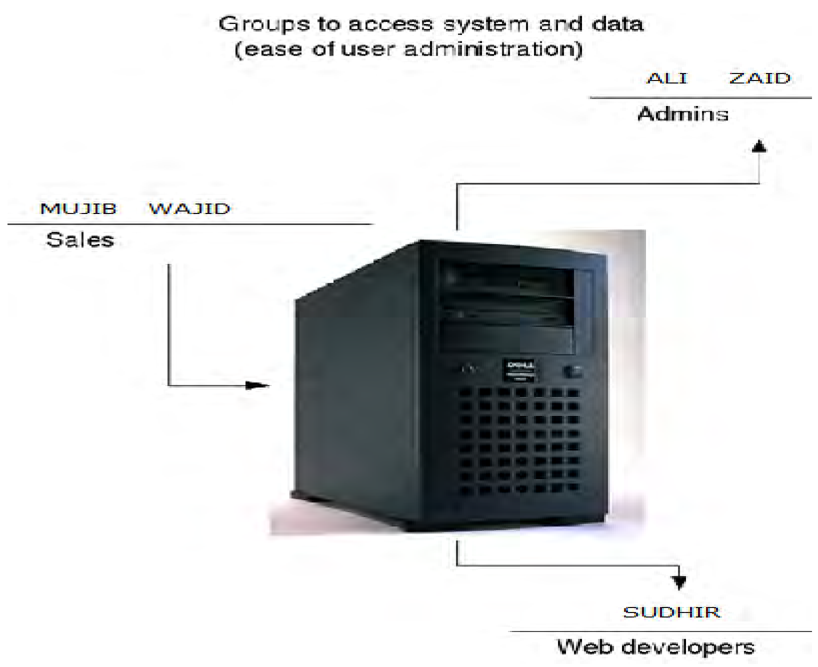
View /etc/group

```
administrator@Alisir:~$ cat /etc/group
```

write output

Users on Linux systems are assigned to one or more groups for the following reasons:

- To share files or other resource with a small number of users
- Ease of user management
- Ease of user monitoring
- Group membership is perfect solution for large Linux (LINUX) installation.
- Group membership gives you or your user special access to files and directories or devices which are permitted to that group


Groups to access system and data
(ease of user administration)
ALI ZAID
Admins
MUJIB WAJID
Sales
SUDHIR
Web developers
(Fig: Understanding groups)

Perform: View Current Groups Settings

Type any one of the following command:

```
$ less /etc/group
```

OR

```
$ more /etc/group
```

Perform: Find Out the Groups a User Is In

Type the following command:

```
$ groups {username}
$ groups
$ groups test
```

Sample outputs:

```
vivek :vivek admdialoutcdromplugdevlpadminnetdev admin sambasharelibvirtd
```

Perform: Print user / group Identity

Use the id command to display information about the given user.

Display only the group ID, enter:

```
$ id -g
$ id -g user
$ id -g test
```

OR

```
$ id –gn test
```

Display only the group ID and the supplementary groups, enter:

```
$ id -G
$ id -G user
$ id -G test
```

OR

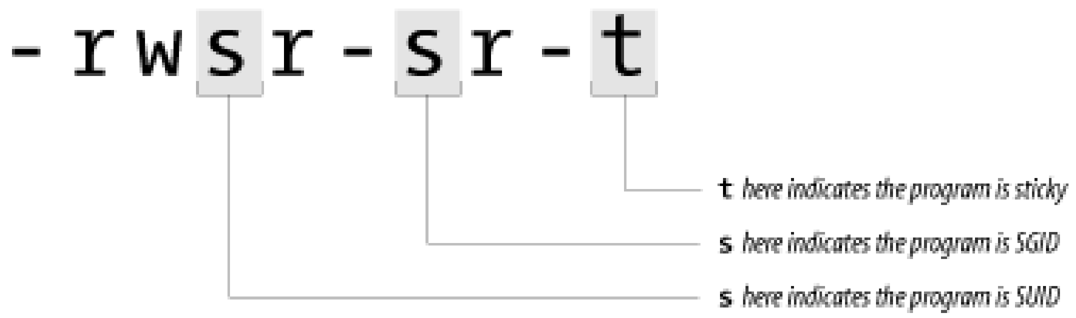
```
$ id –Gn test
```

Experiment No. 5:-Implement setting up and releasing of special permissions (SGID, SUID and sticky bit) and state their effects.

There are three additional bits of information that Linux stores for each file and directory:

Mode	Description
Sticky bit	Used for shared directories to prevent users from renaming or deleting each other's files. The only users who can rename or delete files in directories with the sticky bit set are the file owner, the directory owner, or the super-user (root). The sticky bit is represented by the letter t in the last position of the other permissions display.
SUID	Set user ID, used on executable files to allow the executable to be run as the file owner of the executable rather than as the user logged into the system. SUID can also be used on a directory to change the ownership of files created in or moved to that directory to be owned by the directory owner rather than the user who created it.
SGID	Set group ID, used on executable files to allow the file to be run as if logged into the group (like SUID but uses file group permissions). SGID can also be used on a directory so that every file created in that directory will have the directory group owner rather than the group owner of the user creating the file.

Figure: Additional file permissions


- r w s r - s r - t
t here indicates the program is sticky
s here indicates the program is SGID
s here indicates the program is SUID

The chmod SUID, SGID, and Sticky Bit Octal Values

Binary	Octal	Description
000	0	All bits are cleared.
001	1	The sticky bit is set.
010	2	The SGID bit is set.
011	3	The SGID and sticky bits are set.
100	4	The SUID bit is set.
101	5	The SUID and sticky bits are set.
110	6	The SUID and SGID bits are set.
111	7	All bits are set.

So, to create a shared directory that always sets the directory group for all new files, all you need to do is set the SGID bit for the directory:

```
$ mkdir testdir
$ ls -l
drwxrwxr-x 2 rich rich 4096 Sep 20 23:12 testdir/
$ chgrp shared testdir
$ chmod g+s testdir
$ ls -l
drwxrwsr-x 2 rich shared 4096 Sep 20 23:12 testdir/
$ umask 002
$ cd testdir
$ touch testfile
$ ls -l
total 0
-rw-rw-r-- 1 rich shared 0 Sep 20 23:13 testfile
$
```

The first step is to create a directory that you want to share using the mkdir command. Next, the chgrp command is used to change the default group for the directory to a group that contains the members who need to share files. Finally, the SGID bit is set for the directory, to ensure that any files created in the directory use the shared group name as the default group. For this environment to work properly, all of the group members need to have their umask values set to make files writable by group members. This is why I changed my umask to 002. After all that's done, I can go to the shared directory and create a new file. As expected, the new file uses the default group of the directory, not my user account's default group. Now any user in the shared group can access this file.

The following example displays the SUID permission mode that is set on the passwd command, indicated by the letter s in the last position of the user permission display. Users would like to be able to change their own passwords instead of having to ask the System Administrator to do it for them. Since changing a password involves updating the /etc/passwd file which is owned by root and protected from modification by any other user, the passwd command must be executed as the root user.
The which command will be used to find the full path name for the passwd command, then the attributes of the passwd command will be listed, showing the SUID permission(s).

The SUID Special Permission Mode

```
$ which passwd
/usr/bin/passwd
$ ls -l /usr/bin/passwd
-r-s--x--x    1 root     root        17700 Jun 25  2004
/usr/bin/passwd
```

Here we see not only that the SUID permissions are set up on the passwd command but also that the command is owned by the root user. These two factors tell us that the passwd command will run with the permissions of root regardless of who executes it.

Examples

1. Symbolic way (t, represents sticky bit)

```
$ mkdir public
$ chmod 777 public
$ chmod +t public
$ ls -l
total 4
drwxrwxrwt 2 tclark authors 4096 Sep 14 10:45 public
```

We see that the last character of the permissions string has a t indicating the sticky bit has been set.

2. **Numerical/octal way (1, Sticky Bit value 1)**

```
$ chmod 1777 public
$ ls -l
total 4
drwxrwxrwt 2 tclark authors 4096 Sep 14 10:45 public
```

Now let's say we instead want to make a directory which other users can copy files but which we want the files to instantly become owned by our username and group. This is where the SUID and SGID options come in.

```
$ mkdir drop_box
$ chmod 777 drop_box
$ chmod u+s,g+s drop_box
$ ls -l
total 4
drwsrwsrwx 2 tclark authors 4096 Sep 14 10:55 drop_box
```

Now anyone can move files to this directory but upon creation in drop_box they will become owned by tclark and the group authors. This example also illustrates how you can change multiple levels of permissions with a single command by separating them with a comma. Just like with the other permissions this could have been simplified into one command using the SUID and SGID numeric values (4 and 2 respectively.) Since we are changing both in this case we use 6 as the first value for the chmod command.

```
$ chmod 6777 drop_box
$ ls -l
total 4
drwsrwsrwx 2 oracle users 4096 Sep 14 10:55 drop_box
```

Another Example: Create a folder where people will try to dump files for sharing, but they should not delete the files created by other users.

Sticky Bit can be set in two ways:

3. **Symbolic way (t, represents sticky bit)**
4. **Numerical/octal way (1, Sticky Bit value 1)**

Use chmod command to set Sticky Bit on Folder: **/opt/dump/**

Symbolic way:

```
chmod o+t /opt/dump/
or
chmod +t /opt/dump/
```

We are setting Sticky Bit(+t) to folder /opt/dump by using chmod command.

Numerical way:

```
chmod 1757 /opt/dump/
```

Here in 1757, 1 indicates Sticky Bit set, 7 for full permissions for owner, 5 for read and executes permissions for group, and full permissions for others.

Checking if a folder is set with Sticky Bit or not?

Use ls –l to check if the x in others permissions field is replaced by **t or T**

For example: /opt/dump/ listing before and after Sticky Bit set

```
Before Sticky Bit set:
ls -l
total 8
-rwxr-xrwx 1 xyz xyzgroup 148 Dec 22 03:46 /opt/dump/
```

After Sticky Bit set:

```
ls -l
total 8
-rwxr-xrwt 1 xyz xyzgroup 148 Dec 22 03:46 /opt/dump/
```

Experiment No. 6
Implement I/O Redirection and Pipes.

By using some special notation we can *redirect* the output of many commands to files, devices, and even to the input of other commands.

Standard Output

Most command line programs that display their results do so by sending their results to a facility called ***standard output***. By default, standard output directs its contents to the display.

To redirect standard output to a file, the ">" character is used.

Example:

```
[ali@mhssp]$ ls
```

{Perform and write output}

```
[ali@mhssp]$ ls > file_list.txt
```

{Perform and write output}

```
[ali@mhssp]$ cat file_list.txt
```

{Perform and write output}

In this example, the **ls** command is executed and the results are written in a file named file_list.txt. Since the output of **ls** was redirected to the file, no results appear on the display.
Each time the command above is repeated, file_list.txt is overwritten (from the beginning) with the output of the command **ls**. If we want the new results to be *appended* to the file instead, use ">>".

Example:

```
[ali@mhssp]$ ls >> file_list.txt
```

{Perform and write output}

```
[ali@mhssp]$ cat file_list.txt
```

{Perform and write output}

When the results are appended, the new results are added to the end of the file, thus making the file longer each time the command is repeated. If the file does not exist when we attempt to append the redirected output, the file will be created.

Standard Input

Many commands can accept input from a facility called ***standard input***. By default, standard input gets its contents from the keyboard, but like standard output, it can be redirected.
To redirect standard input from a file instead of the keyboard, the "<" character is used.

Example:

```
[ali@mhssp]$ sort < file_list.txt
```

{Perform and write output}

In the above example we used the **sort** command to process the contents of file_list.txt. The results are output on the display since the standard output is not redirected in this example. We could redirect standard output to another.
Example:

```
[ali@mhssp]$ sort < file_list.txt > sorted_file_list.txt
```

{Perform and write output}

```
[ali@mhssp]$ cat sorted_file_list.txt
```

{Perform and write output}

Pipes

The most useful and powerful thing we can do with I/O redirection is to connect multiple commands together with what are called ***pipes***. With pipes, the standard output of one command is fed into the standard input of another.
Example:

```
[ali@mhssp]$ ls -l | less
```

{Perform and write output}

In this example, the output of the **ls** command is fed into **less**. By using this **"| less"**, we can make any command have scrolling output.

Examples of commands used together with pipes	
Command	**What it does**
`ls -lt	head`	Displays the 10 newest files in the current directory.
`du	sort -nr`	Displays a list of directories and how much space they consume, sorted from the largest to the smallest.
`find . -type f -print	wc -l`	Displays the total number of files in the current working directory and all of its subdirectories.

{Perform and write output}

Filters

One class of programs we can use with pipes is called ***filters***. Filters take standard input and perform an operation upon it and send the results to standard output. In this way, they can be used to process information in powerful ways. Here are some of the common programs that can act as filters:

Common filter commands	
Program	**What it does**
sort	Sorts standard input then outputs the sorted result on standard output.

uniq	Given a sorted stream of data from standard input, it removes duplicate lines of data (i.e., it makes sure that every line is unique).
grep	Examines each line of data it receives from standard input and outputs every line that contains a specified pattern of characters.
fmt	Reads text from standard input, then outputs formatted text on standard output.
pr	Takes text input from standard input and splits the data into pages with page breaks, headers and footers in preparation for printing.
head	Outputs the first few lines of its input. Useful for getting the header of a file.
tail	Outputs the last few lines of its input. Useful for things like getting the most recent entries from a log file.
tr	Translates characters. Can be used to perform tasks such as upper/lowercase conversions or changing line termination characters from one type to another (for example, converting DOS text files into Unix style text files).
sed	Stream editor. Can perform more sophisticated text translations than **tr**.
awk	An entire programming language designed for constructing filters. Extremely powerful.

Performing tasks with pipes

1. **Printing from the command line.** Linux provides a program called lpr that accepts standard input and sends it to the printer. It is often used with pipes and filters.
   Examples:

   ```
   cat report.txt | fmt | pr | lpr
   ```

   ```
   cat dupes.txt | sort | uniq | pr | lpr
   ```

   In the first example, we use **cat** to read the file and output it to standard output, which is piped into the standard input of **fmt. fmt** formats the text into neat paragraphs and outputs it to standard output, which is piped into the standard input of **pr. pr** splits the text neatly into pages and outputs it to standard output, which is piped into the standard input of **lpr. lpr** takes its standard input and sends it to the printer.
   The second example starts with an unsorted list of data with duplicate entries. First, **cat** sends the list into **sort** which sorts it and feeds it into **uniq** which removes any duplicates. Next **pr** and **lpr** are used to paginate and print the list.
2. **Viewing the contents of tar files**
   We can recognize these files by their traditional file extensions, ".tar.gz" or ".tgz". We can use the following command to view the directory of such a file on a Linux system:

   ```
   tar tzvf name_of_file.tar.gz | less
   ```

Experiment No . 7

- ☐ **Write shell script to demonstrate use of conditional and loop control statements.**
- ☐ **Write a shell script that shows effects of quotes on the Output of a variable.**
- ☐ **Write a shell script that looks through all the files in the current directory for the string POSIX and then prints the name of these files to the standard output.**

The shell provides several commands that we can use to control the flow of execution in our program. These include:

- **if**
- **exit**
- **for**
- **while**
- **until**
- **case**
- **break**
- **continue**

THE IF...FI STATEMENT

The **if...fi** statement is the fundamental control statement that allows Shell to make decisions and execute statements conditionally.

Syntax:

```
if [ expression ]
then
   Statement(s) to be executed if expression is true
fi
```

Here Shell *expression* is evaluated. If the resulting value is true, given statement(s) are executed. If *expression* is *false* then no statement would be executed.

There must be spaces between braces and expression. This space is mandatory otherwise we would get syntax error.

If **expression** is a shell command then it would be assumed true if it return 0 after its execution. If it is a Boolean expression then it would be true if it returns true.

Example:

```
#!/bin/sh
a=10
b=20
if [ $a == $b ]
then
   echo "a is equal to b"
fi
if [ $a != $b ]
then
   echo "a is not equal to b"
fi
```

This will produce following result:

```
a is not equal to b
```

test

The **test** command is used most often with the **if** command to perform true/false decisions. The command is unusual in that it has two different syntactic forms:

```
# First form

test expression

# Second form

[ expression ]
```

The **test** command works simply. If the given expression is true, **test** exits with a status of zero; otherwise it exits with a status of 1.

THE IF...ELSE...FI STATEMENT

The **if...else...fi** statement is the next form of control statement that allows Shell to execute statements in more controlled way and making decision between two choices.

Syntax:

```
if [ expression ]
then
   Statement(s) to be executed if expression is true
else
   Statement(s) to be executed if expression is not true
fi
```

Here Shell *expression* is evaluated. If the resulting value is *true*, given statement(s) are executed. If *expression* is *false* then executes all statements up to fi.

Example 1:

If we take above example then it can be written in better way using if…else statement as follows:

```
#!/bin/sh
a=10
b=20
if [ $a == $b ]
then
   echo "a is equal to b"
else
   echo "a is not equal to b"
fi
```

This will produce following result:

```
a is not equal to b
```

Example 2:

```
##!/bin/bash
read -p "Enter a password" pass
if test "$pass" = "jerry"
then
     echo "Password verified."
else
     echo "Access denied."
Fi
```

Example 3:

```
#!/bin/bash
read -p "Enter number : " n
if test $n -ge 0
then
      echo "$n is positive number."
else
      echo "$n number is negative number."
fi
```

THE IF...ELIF...FI STATEMENT

The **if...elif...fi** statement is the one level advance form of control statement that allows Shell to make correct decision out of several conditions.

Syntax:

```
if [ expression 1 ]
then
   Statement(s) to be executed if expression 1 is true
```

```
elif [ expression 2 ]
then
   Statement(s) to be executed if expression 2 is true
elif [ expression 3 ]
then
   Statement(s) to be executed if expression 3 is true
else
   Statement(s) to be executed if no expression is true
fi
```

Here statement(s) are executed based on the true condition, if none of the condition is true then *else* block is executed.

Example:

```
#!/bin/sh
a=10
b=20
if [ $a == $b ]
then
   echo "a is equal to b"
elif [ $a -gt $b ]
then
   echo "a is greater than b"
elif [ $a -lt $b ]
then
   echo "a is less than b"
else
   echo "None of the condition met"
fi
```

This will produce following result:

```
a is less than b
```

THE CASE...ESAC STATEMENT

Shell support **case...esac** statement which handles multiway branch more efficiently than repeated if...elif statements.

Syntax:

```
case word in
  pattern1)
     Statement(s) to be executed if pattern1 matches
     ;;
  pattern2)
     Statement(s) to be executed if pattern2 matches
     ;;
  pattern3)
     Statement(s) to be executed if pattern3 matches
     ;;
esac
```

Here the string word is compared against every pattern until a match is found. The statement(s) following the matching pattern executes. If no matches are found, the case statement exits without performing any action.

There is no maximum number of patterns, but the minimum is one.

When statement(s) part executes, the command ;; indicates that program flow should jump to the end of the entire case statement.

Example:

```
#!/bin/sh

MARKS="50"

case "$MARKS" in
   "70") echo "You are placed in First class distinction."
   ;;
   "60") echo "You are placed in First class."
   ;;
   "50") echo "You are placed in Second class."
   ;;
esac
```

This will produce following result:

```
You are placed in Second class.
```

A good use for a case statement is the evaluation of command line arguments as follows:

```
#!/bin/sh

option="${1}"
case ${option} in
   -f) FILE="${2}"
      echo "File name is $FILE"
      ;;
   -d) DIR="${2}"
      echo "Dir name is $DIR"
      ;;
   *)
      echo "`basename ${0}`:usage: [-f file] | [-d directory]"
      exit 1 # Command to come out of the program with status 1
      ;;
esac
```

Here is a sample run of this program:

```
$./test.sh
test.sh: usage: [ -f filename ] | [ -d directory ]
$ ./test.sh -f index.htm
$ vi test.sh
$ ./test.sh -f index.htm
File name is index.htm
$ ./test.sh -d unix
Dir name is unix
$
```

The while Loop

The while loop enables us to execute a set of commands repeatedly until some condition occurs.

Syntax:

```
while command
do
  Statement(s) to be executed if command is true
done
```

<u>Shell script Using the while loop to display the numbers zero to nine:</u>

```
#!/bin/sh
a=0
while [ $a -lt 10 ]
do
  echo $a
  a=`expr $a + 1`
done
```

output:

```
0
1
2
```

```
3
4
5
6
7
8
9
```

The same program that counts from zero to nine:

```
#!/bin/bash
number=0
while [ $number -lt 10 ]; do
    echo "Number = $number"
    number=$((number + 1))
done
```

Nesting while Loops:

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

Shell Script to Print pattern:

```
#!/bin/sh
a=0
while [ "$a" -lt 10 ]    # this is loop1
do
   b="$a"
   while [ "$b" -ge 0 ]  # this is loop2
   do
      echo -n "$b "
      b=`expr $b - 1`
   done
   echo
   a=`expr $a + 1`
done
```

Output:

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

Here **-n** option let echo to avoid printing a new line character.

The for Loop

The for loop operate on lists of items. It repeats a set of commands for every item in a list.

Syntax 1:

```
for var in word1 word2 ... wordN
do
   Statement(s) to be executed for every word.
done
```

Here *var* is the name of a variable and word1 to wordN are sequences of characters separated by spaces *words. Each time the for loop executes, the value of the variable var is set to the next word in the list of words, word1 to wordN.*

Syntax 2:

```
        for (( expr1; expr2; expr3 ))
        do
             .....
                       ...
             repeat all statements between do and
             done until expr2 is TRUE
        Done
```

Shell Script 1:

Uses for loop to span through the given list of numbers:

```
#!/bin/sh
for var in 0 1 2 3 4 5 6 7 8 9
do
   echo $var
done
```

Output:

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

Shell Script 2:

Following is the example to display all the files starting with **.bash** and available in your home. I'm executing this script from my root:

```
#!/bin/sh
for FILE in $HOME/.bash*
do
   echo $FILE
done
```

This will produce following result:

```
/root/.bash_history
/root/.bash_logout
/root/.bash_profile
/root/.bashrc
```

Nesting of for Loop

Shell Script:

```
$ vi nestedfor.sh
#!/bin/sh

for (( i = 1; i <= 5; i++ ))
do

   for (( j = 1 ; j <= 5; j++ ))
   do
         echo -n "$i "
   done

  echo "" #### print the new line ###

done
```

Run the above script as follows:

```
$ chmod +x nestedfor.sh
$ ./nestefor.sh
1 1 1 1 1
2 2 2 2 2
3 3 3 3 3
4 4 4 4 4
5 5 5 5 5
```

echo "" -print the new line

The until Loop

Syntax:

```
until command
do
   Statement(s) to be executed until command is true
done
```

Here Shell *command* is evaluated. If the resulting value is *false*, given *statements* are executed. If *command* is *true* then no statement would be not executed and program would jump to the next line after done statement.

Shell Script that uses the until loop to display the numbers zero to nine:

```
#!/bin/sh
a=0
until [ ! $a -lt 10 ]
do
   echo $a
   a=`expr $a + 1`
done
```

This will produce following result:

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

```
#!/bin/bash

number=0
until [ $number -ge 10 ]; do
    echo "Number = $number"
    number=$((number + 1))
done
```

Write a shell script that shows effects of quotes on the Output of a variable

```
#!/bin/bash
myname=Mohammad Ali'
echo 'My name is $myname'
echo "My name is $myname"
echo 'The name of this computer is `hostname`'
echo "The name of this computer is `hostname`"
```

Write a shell script that looks through all the files in the current directory for the string POSIX and then prints the name of these files to the standard output.

```
#!/bin/bash
grep -w "POSIX" *
if [ $? -eq 0 ]
then
   echo "String found!"
else
   echo "String NOT found!"
fi
```

Experiment No.8

Write shell script to implement following test commands :

☐For string comparisons.

☐For numeric comparisons.

☐For file comparisons

Shell Scripts For numeric comparisons.

The most common method for using the test command is to perform a comparison of two numeric values. The numeric test conditions can be used to evaluate both numbers and variables.

Example:

```
$ cat test5.sh
#!/bin/bash
# using numeric test comparisons
val1=10
val2=11
if [ $val1 -gt 5 ]
then
echo "The test value $val1 is greater than 5“
fi
if [ $val1 -eq $val2 ]
then
echo "The values are equal“
else
echo "The values are different“
fi
```

```
$ ./test5.sh
The test value 10 is greater than 5
The values are different
$
```

Both of the numeric test conditions evaluated as expected.

Shell Scripts For String comparisons.

The test command also allows us to perform comparisons on string values. Classified into three categories:

- String Equality
- String Order
- String Size

String equality

The equal and not equal conditions are fairly self-explanatory with strings. It’s pretty easy to know when two string values are the same or not:

```
$cat test7.sh
#!/bin/bash
# testing string equality
```

```
testuser=mhssp
if [ $USER = $testuser ]
then
echo "Welcome $testuser"
fi
$ ./test7.sh
Welcome mhssp
$
```

Using the not equals string comparison:

```
$ cat test8.sh
#!/bin/bash
# testing string equality
testuser=baduser
if [ $USER != $testuser ]
then
echo "This isn't $testuser"
else
echo "Welcome $testuser"
fi
$ ./test8.sh
This isn't baduser
$
```

String order

```
$ cat test9.sh
#!/bin/bash
# Using string comparisons
val1=baseball
val2=hockey
if [ $val1 \> $val2 ]
then
echo "$val1 is greater than $val2"
else
echo "$val1 is less than $val2"
fi
$ ./test9.sh
baseball is less than hockey
$
$ cat test9b.sh
#!/bin/bash
# testing string sort order
val1=Testing
val2=testing
if [ $val1 \> $val2 ]
then
echo "$val1 is greater than $val2"
else
echo "$val1 is less than $val2"
fi
$ ./test9b.sh
Testing is less than testing
```

String size

The -n and -z comparisons are handy when trying to evaluate if a variable contains data or not:

```
$ cat test10.sh
```

```
#!/bin/bash
# testing string length
val1=testing
val2=‘ ‘
if [ -n $val1 ]
then
echo "The string ’$val1’ is not empty"
else
echo "The string ’$val1’ is empty"
fi
if [ -z $val2 ]
then
echo "The string ’$val2’ is empty"
else
echo "The string ’$val2’ is not empty"
fi
if [ -z $val3 ]
then
echo "The string ’$val3’ is empty"
else
echo "The string ’$val3’ is not empty"
fi
$ ./test10.sh
The string ’testing’ is not empty
The string ’’ is empty
The string ’’ is empty
$
```

File comparisons

Checking directories

The -d test checks if a specified filename exists as a directory on the system. This is usually a good thing to do if we’re trying to write a file to a directory, or before we try to change to a directory location:

```
$ cat test11.sh
#!/bin/bash
# looking for our directory
if [ -d $HOME ]
then
echo “Our HOME directory exists"
cd $HOME
ls -a
else
echo "There’s a problem with Our HOME directory"
fi
```

Checking for a file

The -e comparison works for both files and directories. To be sure that the object specified is a file, you must use the -f comparison:

```
$ cat test12.sh
#!/bin/bash
# check if a file
if [ -e $HOME ]
then
echo "The object exists, is it a file?"
if [ -f $HOME ]
then
echo "Yes, it’s a file!"
else
echo "No, it’s not a file!"
if [ -f $HOME/.bash history ]
then
echo "But this is a file!"
fi
fi
else
echo "Sorry, the object doesn’t exist"
fi
$ ./test12.sh
The object exists, is it a file?
No, it’s not a file!
But this is a file!
$
```

Cheking for read permission

This is done using the -r comparison:

```
$ cat test13.sh
#!/bin/bash
# testing if you can read a file
pwfile=/etc/shadow
# first, test if the file exists, and is a file
if [ -f $pwfile ]
then
# now test if you can read it
if [ -r $pwfile ]
then
tail $pwfile
else
echo "Sorry, I’m unable to read the $pwfile file"
fi
else
echo "Sorry, the file $file doesn’t exist"
fi
```

```
$ ./test13.sh
Sorry, I’m unable to read the /etc/shadow file
$
```

Checking if you can run a file

The -x comparison is a handy way to determine if we have execute permission for a specific file.

```
$ cat test14.sh
#!/bin/bash
# testing file execution
if [ -x test164.sh]
then
echo "You can run the script:"
./test14.sh
else
echo "Sorry, you are unable to execute the script"
fi
```

```
$ ./test14.sh
You can run the script:
The first attempt failed
The second attempt succeeded
$ chmod u-x test14.sh
$ ./test14.sh
Sorry, you are unable to execute the script
$
```

Experiment No. 9

Write shell script that :

☐Uses command line parameters.

☐Counts number of parameters.

☐Implements shift command.

☐Implements processing option with parameter values.

Command Line Parameters:

$0 is the name of the command

$1 first parameter

$2 second parameter

$3 third parameter etc. etc

$# total number of parameters

$@ all the parameters will be listed

Shell script that uses command line parameters

Shell script of using one command line parameter in a shell script:

```
$ cat test1.sh
#!/bin/bash
# using one command line parameter
factorial=1
for (( number = 1; number <= $1 ; number++ ))
do
factorial=$[ $factorial * $number ]
done
echo The factorial of $1 is $factorial

$ ./test1.sh 5
The factorial of 5 is 120
$
```

More command line parameters: Each parameter must be separated by a space on the command line:

```
$ cat test2.sh
#!/bin/bash
# testing two command line parameters
total=$[ $1 * $2 ]
```

```
echo The first paramerer is $1.
echo The second parameter is $2.
echo The total value is $total.
$ ./test2.sh 2 5
The first paramerer is 2.
The second parameter is 5.
The total value is 10.
$
```

In this example, the command line parameters used were both numerical values. We can also use text strings in the command line:

```
$ cat test3.sh
#!/bin/bash
# testing string parameters
echo Hello $1, glad to meet you.

$ ./test3.sh mhssp
Hello mhssp, glad to meet you.
$
```

If our script needs more than nine command line parameters, we can continue, but the variable names change slightly. After the ninth variable, we must use braces around the variable number, such as ${10}. Here's an example of doing that:

```
$ cat test4.sh
#!/bin/bash
# handling lots of parameters
total=$[ ${10} * ${11} ]
echo The tenth parameter is ${10}
echo The eleventh parameter is ${11}
echo The total is $total

$ ./test4.sh 1 2 3 4 5 6 7 8 9 10 11 12
The tenth parameter is 10
The eleventh parameter is 11
The total is 110
$
```

<u>**Reading the program name**</u>

We can use the $0 parameter to determine the name of the program that the shell started from the command line

```
$ cat test5.sh
#!/bin/bash
# testing the $0 parameter
echo The command entered is: $0

$ ./test5.sh
The command entered is: ./test5
$ /home/rich/test5
The command entered is: /home/rich/test5
$
```

Shell Script to Count number of parameters.

```
$ cat countpara.sh
#!/bin/bash
# getting the number of parameters
echo There were $# parameters supplied.

$ ./countpara.sh
There were 0 parameters supplied.
$ ./countpara.sh1 2 3 4 5
There were 5 parameters supplied.
$ ./countpara.sh 1 2 3 4 5 6 7 8 9 10
There were 10 parameters supplied.
$ ./countpara.sh″Saboo Siddik”
There were 1 parameters supplied.
$
```

□Implementing shift command.

If we don't know how many parameters are available can use shift. We can just operate on the first parameter, shift the parameters over, and then operate on the first parameter again.

```
$ cat test6.sh
#!/bin/bash
# demonstrating the shift command
count=1
while [ -n ″$1″ ]
do
echo ″Parameter #$count = $1″
```

```
count=$[ $count + 1 ]
shift
done

$ ./test6.sh mhssp saboo siddik mumbai
Parameter #1 = mhssp
Parameter #2 = saboo
Parameter #3 = siddik
Parameter #4 = mumbai
$
```

The script performs a while loop, testing the length of the first parameter's value. When the first Parameter's length is zero, the loop ends. After testing the first parameter, the shift command is used to shift all of the parameters one position.
Alternatively, we can perform a multiple location shift by providing a parameter to the shift command. Just provide the number of places we want to shift:

```
$ cat test7.sh
#!/bin/bash
# demonstrating a multi-position shift
echo "The original parameters: $*"
shift 2
echo "Here's the new first parameter: $1"

$ ./test7.sh 1 2 3 4 5
The original parameters: 1 2 3 4 5
Here's the new first parameter: 3
$
```

By using values in the shift command, we can easily skip over parameters you don't need.

☐Implements processing option with parameter values.
Some options require an additional parameter value. In these situations, the command line looks something like this:

```
$ ./testing -a test1 -b -c -d test2
```

Your script must be able to detect when your command line option requires an additional parameter and be able to process it appropriately. Here's an example of how to do that:

```
$ cat test8.sh
#!/bin/bash
# extracting command line options and values
while [ -n "$1" ]
```

```
do
case "$1" in
-a) echo "Found the -a option";;
-b) param="$2"
echo "Found the -b option, with parameter value $param"
shift 2;;
-c) echo "Found the -c option";;
--) shift
break;;
*) echo "$1 is not an option";;
esac
shift
done
count=1
for param in "$@"
do
echo "Parameter #$count: $param"
count=$[ $count + 1 ]
done
```

```
$ ./test8.sh -a -b test1 -d
Found the -a option
Found the -b option, with parameter value test1
-d is not an option
$
```

In this example, the case statement defines three options that it processes. The -b option also requires an additional parameter value. Since the parameter being processed is $1, you know that the additional parameter value is located in $2 (since all of the parameters are shifted after they are processed). Just extract the parameter value from the $2 variable. Of course, since we used two parameter spots for this option, you also need to set the shift command to shift two positions.

```
$ ./test17 -b test1 -a -d
Found the -b option, with parameter value test1
Found the -a option
-d is not an option
$
```

Experiment No. 10

Write shell script:

- ☐ **To implement redirection of Input script.**
- ☐ **For redirecting file descriptors.**
- ☐ **Creating input file descriptor.**

Standard file descriptors

The Linux system handles every object as a file. This includes the input and output process. Linux identifies each file object using a file descriptor. The **file descriptor** is a non-negative integer, which uniquely identifies open files in a session. Each process is allowed to have up to nine open file descriptors at a time. The bash shell reserves the first three file descriptors (0, 1, and 2) for special purposes shown below:

File Descriptor	**Abbreviation**	**Description**
0	**STDIN**	**Standard input**
1	**STDOUT**	**Standard output**
2	**STDERR**	**Standard error**

Shell Script to Implement Redirecting Input in Scripts

The exec command allows you to redirect STDIN to a file on the Linux system:

exec 0< testfile

This command informs the shell that it should retrieve input from the file testfile instead of STDIN. This redirection applies anytime the script requests input.

```
$ cat test1.sh
#!/bin/bash
# redirecting file input
exec 0< testfile
count=1
while read line
do
echo "Line #$count: $line"
count=$[ $count + 1 ]
done

$ ./test1.sh
Line #1: This is the first line.
Line #2: This is the second line.
Line #3: This is the third line.
$
```

Shell Script For redirecting file descriptors.

We can assign an alter- native file descriptor to a standard file descriptor, and vice versa. This means that we can redirect the original location of STDOUT to an alternative file descriptor, then redirect that file descriptor back to STDOUT.

```
$ cat test2.sh
#!/bin/bash
# storing STDOUT, then coming back to it
exec 3>&1
exec 1>test2out
echo "This should store in the output file"
echo "along with this line."
exec 1>&3
echo "Now things should be back to normal"
```

```
$ ./test2.sh
Now things should be back to normal

$ cat test2out
This should store in the output file
along with this line.
$
```

First, the script redirects file descriptor 3 to the current location of file descriptor 1, which is STDOUT. This means that any output sent to file descriptor 3 will go to the monitor.
The second exec command redirects STDOUT to a file. The shell will now redirect any output sent to STDOUT directly to the output file. However, file descriptor 3 still points to the original location of STDOUT, which is the monitor. If we send output data to file descriptor 3 at this point, it'll still go to the monitor, even though STDOUT is redirected.
After sending some output to STDOUT, which points to a file, the script then redirects STDOUT to the current location of file descriptor 3, which is still set to the monitor. This means that now STDOUT is pointing to its original location, the monitor.

Shell Script For Creating input file descriptors

We can redirect input file descriptors exactly the same way as output file descriptors. Save the STDIN file descriptor location to another file descriptor before redirecting it to a file, then when we're done reading the file we can restore STDIN to its original location:

```
$ cat test15
#!/bin/bash
# redirecting input file descriptors
exec 6<&0
exec 0< testfile
count=1
while read line
do
echo "Line #$count: $line"
count=$[ $count + 1 ]
done
exec 0<&6
read -p "Are you done now? " answer
case $answer in
Y|y) echo "Goodbye";;
N|n) echo "Sorry, this is the end.";;
esac

$ ./test15
Line #1: This is the first line.
Line #2: This is the second line.
Line #3: This is the third line.
Are you done now? y
Goodbye
$
```

In this example, file descriptor 6 is used to hold the location for STDIN. The script then redirects STDIN to a file. All of the input for the read command comes from the redirected STDIN, which is now the input file.
When all of the lines have been read, the script returns STDIN to its original location by redirecting it to file descriptor 6. The script tests to make sure that STDIN is back to normal by using another read command, which this time waits for input from the keyboard.

Experiment No. 12

☐Write a shell script using functions. Modify it to handle function with parameters, function returning values.

☐ Write shell script for handling array variables.

☐ Write shell script that uses function returning true or false result.

Creating a function

There are two formats we can use to create functions in bash shell scripts. The first format uses the keyword function, along with the function name we assign to the block of code:

```
function name
{
commands
}
```

The name attribute defines a unique name assigned to the function. Each function we define in our script must be assigned a unique name.

The commands are one or more bash shell commands that make up our function. When we call the function, the bash shell executes each of the commands in the order they appear in the function, just as in a normal script.

The second format for defining a function in a bash shell script more closely follows how functions are defined in other programming languages:

```
name()
{
commands
}
```

The empty parentheses after the function name indicate that we're defining a function. The same naming rules apply in this format as in the original shell script function format.

☐ Write a shell script using functions. Modify it to handle function with parameters, function returning values.

Simple Shell Script using function:

```
#!/bin/bash
# Define function here
Hello ()
{
  echo "Hello World"
}
# Invoke your function
Hello

$./test25.sh
Hello World
$
```

Modify it to handle function with parameters, function returning values.

The parameters would be represented by $1, $2 and so on. Based on the situation we can return any value from our function using the **return** command whose syntax is as follows:

```
return code
```

Shell script is modified where we pass two parameters Saboo and Siddik and then we capture and print these parameters in the function.

```
#!/bin/bash
# Define function here
Hello ()
{
   echo "Hello World $1 $2"
   return 10
}
# Invoke your function
Hello Saboo Siddik
# Capture value returned by last command
ret=$?
echo "Return value is $ret"

$./test25.sh
Hello World Saboo Siddik
Return value is 10
$
```

☐ Write shell script for handling array variables.

```
#!/bin/bash
# adding values in an array
function addarray
{
local sum=0
local newarray
newarray=(`echo "$@"`)
for value in ${newarray[*]}
do
sum=$[ $sum + $value ]
done
echo $sum
}
myarray=(1 2 3 4 5)
echo "The original array is: ${myarray[*]}"
arg1=`echo ${myarray[*]}`
result=`addarray $arg1`
echo "The result is $result"

$ ./test26.sh
The original array is: 1 2 3 4 5
The result is 15
$
```

☐ Write shell script that uses function returning true or false result.

```
#!/bin/bash
isdirectory()
{
  if [ -d "$1" ]
  then
    return 0
  else
    return 1
  fi
}
if isdirectory $1; then echo "is directory"; else echo "its not a directory"; fi

$ ./test27.sh abc
Write output
```

Experiment No. 13

☐Write a shell script which checks disk space and store the value to the variable and display it.

```
#!/bin/bash
# monitor available disk space
space=$(df -k /tmp | tail -1 | awk '{print $4}')
echo "$space"

$./test28.sh
2586456
```

```
#!/bin/bash
# monitor available disk space
space=$(df -k / | tail -1 | awk '{print $4}')
echo "$space"

$./test29.sh
8586963
```

```
$ cat diskmon.sh
#!/bin/bash
# monitor available disk space
SPACE=$(df | sed -n '/ \ / $ / p' | gawk '{print $4}')
echo "$SPACE"
```

```
$./test30.sh
8586963
```

Note: Run any script