As you can see the echo command is used to print output messages from your script to the output device (screen or file).

Let us now see how to create another script that has both input and output as follows:

```
vi  greetings
        echo What is your name
        read  name
        echo hello  $name
:wq

chmod  +x  greetings

greetings
```

What do you think is the purpose of the read command?

To take input from the user.

Notice that when you read a value in your variable you just put the name of the variable (e..g name) while when you print the value, you need to put the $ sign at the beginning (e.g. $name).

By default, shell scripts treat all variables as strings.

Now let us write our own script for deleting a file:

```
vi  delete
        echo Enter file name:
        read  filename
        rm  $filename
        echo  File $filename has been deleted
:wq
```

*Now run your script. Did you forget to add the (x) permission?*_____.

Now it is your turn to write a complete script and run it.
Write a script called copy that asks the user to enter a source filename and a destination filename and then copies the source to the destination. Your script should work as follows:

```
copy
Enter source file name:
one
Enter destination file name:
two
File one is copied to file two
```

Your script:

```
vi copy
echo enter source file name:
read one
echo enter destination file name:
read two
cp one two
echo file copied succesfuly
:wq
```

Try to run your copy script. Did it work? Yes.

You probably realize that programs like delete and copy would behave more like similar commands if they took their input from the command line instead of asking the user to enter those after running the program. To do this we need to use positional parameters. Let us write a simple script to understand how those are used:

```
vi params
        echo $1
        echo $3 $2
        echo $#
        echo $0
        echo $5
        echo $*
:wq
```

Now run the script params as follows:

```
params  one two 3 four 5 6 bye
```

What was the output?

```
one
3 two
7
./params
5
one two 3 four 56 bye
```

Now what do you think are the values of each of the following variables:

$1: prints the first input (positional)

$3: " " 3rd " (position 3)

$*: prints all input

$#: displays the number of inputs

$0: prints the $0 file name

Now that you understand how positional parameters work, ***rewrite both the delete and copy scripts above to run as follows:***

delete thefile
thefile has been deleted

Answer: vi delete

```
echo Enter file name:
rm $1
echo $1 has been deleted.
:wq
```

copy file1 file2
File file1 has been copied to file2

Answer: vi copy

```
cp $1 $2
echo file one is copied to file 2
    → or echo $1 is copied to $2
:wq
```

Now try your new delete and copy scripts? Did they work? yes.

Notice that since you already had the *x* permission on the previous scripts (***delete*** and ***copy***), you did not have to do that step again in order to run them.

Practice:
Write a script called whoisuser that takes the login name of a user as a parameter and then uses the /etc/passwd file to get and print the full name of that user as follows:

whoisuser u1122334
u1122334 = Ahmad Hamdan

hint: use variable and command substitution.
Answer:

```
echo $1 = $(grep + /etc/passwd |cut -d: -f5|tr '_' ' ')

or
username = $ ((grep '$1 /etc/passwd)|cut -d: -f5)
   echo $ username
```

Shifting parameters

To shift script command line parameters to the left, we use the shift command as follows:

shift number of shifts (e.g. shift 2 for 2 shifts)
shift (no number shifts one)

To understand how shift works, let us rewrite and run the params script above as follows:

```
vi params
        echo $1
        shift 2
        echo $2 $3
        echo $#
        shift
        echo $0
        shift 3
        echo $1
        echo $*
:wq
```

output :

one
5 4
9
./params
seven
seven 89 ten bye

Now run the script as follows and notice the effects of shifting:

params one two three 4 5 6 seven 8 9 ten bye

Which parameter is not effected by the shift command? params $0 .

Comments

You can add comments to your scripts by using the # sign followed by the comment anywhere in your script. Lines that start with (#) are interpreted as comments except in one case where shells have (#!) followed by the name of a shell as the first line of a script. In that case that line is interpreted as the name of the shell to be used for executing that script.

Example:
If your script starts with the line:
#!/bin/bash
Then the script is meant to be executed using the **/bin/bash** shell.

Check out the following system scripts:

more /etc/rc.sysinit
more /etc/rc.local

What is the first line in those files (scripts)?

_____

What is the difference between the first line and the few lines that come after it?

_____.

Lab10. Shell Scripts (II)- Programming
(Selection Constructs)

Objectives

After completing this lab, the student should be able to:

- Include programming selection constructs in shell scripts.
- Use the if/else statement to manipulate integer and string values as well as file properties.
- Apply the case statement programming construct for efficient selections as well as creating menus.

Script Selection Constructs

In the previous lab, you have noticed that in our scripts we made several assumptions that files and user names already existed and that we have permissions to remove, copy, or view files and that the correct number of command line arguments where given to our scripts. This is not always the case. Our scripts should be able to check for values and properties before executing what is required. To do this, we need to use selection statements (the If and Case statements).

Unix commands return a value (success = zero and failure or error = non-zero) to the shell. This value is stored in the variable (?) as follows:

Run the command:

ls –al

Now run the command:

echo $?

What result did you get? 0 ***Why?*** it was a success.

Now run the command:

cp

followed by the command:

echo $?

What result did you get? 1 ***Why?*** it wasn't done correctly.

The value returned by Linux commands may be checked in scripts using the if/else structure.

Write the following script:

```
vi checkcommand
        if $1 > out 2> err
        then
                echo Command $1 succeeded
        else
```

end of if

echo Command $1 failed ⇒ anything else is false

(fi)

:wq

Now run the script as follows:

checkcommand date

What result did you get? succeeded ***Why?*** it returned 0.

Now run the command:

checkcommand mv

What result did you get? failed ***Why?*** it returned as sth else than 0.

This is one way to use the if/else structure. Still, many scripts do not check commands, but rather check for variable values, file properties, and number of arguments. To do that we need to use one of two syntaxes:

if test condition (e.g. if test $# -eq 2)

or

if [condition] (e.g. if [$# -eq 2])

The general syntax for the if/else statement is as follows:

```
if condition
then
        statements
elif condition
then
        statements
else
        statements
fi
```

To compare integer values, we use the following relational operators:

-lt (less than), -gt (greater than) -eq (equal)
-le (less than or equal) -ge (greater than or equal), -ne (not equal).

Let us rewrite the delete script we wrote in the previous lab to check for the correct number of arguments as follows:

```
vi delete
        if [ $# -eq 1 ]
        then
         rm $1
         echo $1 is deleted
         exit 0                  # This line returns 0 from the script (success)
else
```

```
        echo  Usage: delete filename
        exit  1
    fi
:wq
```

Now try the above script as follows:

delete myfile (assuming myfile exists and is a regular file)

Then run the command:

echo $?

Did it work? Yes.

What is the value of variable (?) ? 0

Now try it as follows:

delete

Then run the command:

echo $?

What happened? delete didn't work ***Why?*** b/c we didn't give it a file

What is the value of variable (?) ? 1

To check file values we use the following operators:

-f filename (to check if file exists and is of type file)

-d filename (to check if directory exists and is of type directory)

-x,-r,-w (to check if a user has execute, read, or write permissions on a file)

Let us rewrite our delete script to include those:

```
vi delete
    if [ $# -ne 1 ]
    then
        echo  Usage: delete filename
        exit  1
    else
        if [ -f $1 ]      # $1 exists and is a file name
        then
            rm $1
            echo  File  $1 is deleted
            exit  0
elif [ -d $1 ]
then
        rm -r  $1    # $1 exists and is a directory
        echo Directory  $1 is deleted
        exit 0
else
        echo  $1: No such file or directory
```

```
        exit 2
    fi
fi
:wq
```

Now create a file and a directory using the following commands:

touch myfile; mkdir mydir

No try the updated delete script in the following ways:

delete

What happened? usage: delete filename.

delete myfile (myfile exists and is a file)

What happened? the file gets deleted.

delete mydir (mydir exists and is a directory)

What happened? the directory is deleted.

delete wrong (wrong does not exist)

What happened? it prints that it doesn't exist.

Now rewrite the copy script to act as follows:

copy
Usage: copy src dest
copy myfile newfile
File myfile is copied to file newfile
copy mydir newdir
Directory mydir is copied to newdir
copy wrong good
wrong: No such file or directory

```
#! /bin/bash
if [ $# -ne 2 ]
then
  echo
```

Try the new copy script and make sure it works as above?

Did it work correctly? yes.

Sometimes our scripts need to check string values. To do that we need to use the following operators:

= (equal), != (not equal) -n (none null string) -z (zero string (null))

Let us try some of those. let us write a script to check the value of the name entered by the user:

vi checkname

```
if [ $# -ne 1 ]
then
        echo Usage: checkname name
        exit 1
else
        if [ "$1" = "ahmad" ]
        then
                echo $1: Hello
                exit 0
        else
                echo $1: Goodbye
                exit 0
        fi
fi
:wq
```

try it as follows:

checkname ahmad

What happened? ahmad: Hello.

checkname suha

What happened? suha: goodbye.

checkname

What happened? Usage: checkname name.

use -z ***Write a script called checkusername which works as follows:***

```
checkusername
        No names were entered
checkusername   u1112233
        u1112233 = Ahmad Hamdan
checkusername   u11
        u11  = No such user name
checkusername   bash
        bash = No such user name
```

Script:

```
if [ -z "$1" ]
then
 echo No names were entered
 exit 1
fi
var = $(grep ^$1 /etc/passwd |cut -d: -f1)
if [ "$var" = "$1" ]
then
 name = $(grep $1 /etc/passwd| cut -d: -f5 | tr "_" " ")
  echo $1 = $name
  exit 0
else
  echo $1 = no such user name
  exit 2
fi
```

Now try it with similar cases to those written above.
What happened? _____.

Case Statement

We can also use a case statement (similar to switch in c) to check for values. The syntax is as follows:

```
case   value   in
pattern1)  statements
                ;;      # ;;  is the break statement
pattern2)  statements
                ;;
*)  statements          #   * stands for anything which is the default case
esac
```

The patterns may be strings or parts of strings. Those can include the * wild card, the (|) OR operator, as well as ranges (e.g [0-9] or [a-f]) as follows:

s* | S* | good)
means any pattern that starts with s or S or the word good.

[A-Z]*[0-5])
means any pattern with any size that starts with a capital letter and ends with a number between 0 and 5

[a-z][0-9][0-9][0-9] | [0-9][A-Z][A-Z][A-Z][a-f])
means the accepted pattern must consist of exactly four characters the first is a small letter and the next three are numbers or the pattern must be exactly five characters with the first being a number followed by three capital letters and then one small letter between a and f.

Write a script that uses case statement with patterns similar to the above.
Did they work? _____.

Case statements are usually used for handling menus and menu options. Let us try a simple example that uses a menu to call different scripts (modular programming):

Create three different scripts called ***script1***, ***script2***, and ***script3*** respectively. In each script put one line to display which script you're in (e.g in script1 put the line "echo this is script 1").
Now create a script called ***mainscript*** that displays the following menu:

Please select your choice (1-4):
1 - Run script1
2- Run script2
3- Run script3
4- Exit main script

Using a case statement, have your script run the suitable script (1,2, or 3) or exit based on the user's selection.

vi mainscript

```
#! /bin/bash
echo "Please select your choice (1-4):
1-Run script1
2-Run script2
3- Run script3
4-exit "
read choice
case $choicein
1) ./script1
;;
2) ./script2
;;
3) ./script3
;;
4) exit
esac
```

Now try ***mainscript***. ***Did it work?*** Yes.

Lab11. Shell Scripts (III)- Programming
(Looping Constructs)

Objectives

After completing this lab, the student should be able to:
- Include programming looping constructs in shell scripts.
- Understand and use the while, until, and for loops constructs.
- Learn how to make for loops more efficient by using command outputs as lists.

Shell Script Loops

In order to create useful scripts that can automate real jobs, we need to learn how to include loops in those scripts. There are different loop constructs that may be used in shell scripts which include:

while loops
until loops
for loops

Each has its own useful features that make it useful in certain situations.

While Loop

Let us first start with the while loop. The structure of the while loop is as follows:

```
while   condition
do
        statement(s)
done
```

example:

```
vi listarguments
        while  [ $# -ne  0 ]
        do
                echo  $1
                shift
        done
:wq
```

Run the above script as follows:

```
listarguments  a  hello  7  x
```

Check the output.

Note: Rules that apply to conditions used in selection statements are exactly the same as those that apply to conditions in loop statements.

After making sure you understand the above example do the following:

*bash is space senstive

Rewrite the delete script we wrote in the last lab such that it works as follows:

delete file1 wrong dir1 file2

File file1 is deleted
wrong: No such file or directory
Directory dir1 is deleted
File file2 is deleted

Answer:

vi delete

```
#! /bin/bash
if [ $# -eq 0 ]
then
    echo Usage: delete filename
    exit 1
else
while [ $# -ne 0 ]
do
    if [ -f $1 ]
    then
        rm $1
        echo file $1 is deleted
    elif [ -d $1 ]
    then
        rm -r $1
        echo directory $1 is deleted
    else
        echo no such file or directory
shift fi
done
fi
```

:wq

Now try it with existing file and directory names as arguments. ***Does it work?*** yes .

Sometimes the loop will stop executing based on the user input, as follows:

```
vi findahmad
        echo Enter name
        read name
        while [ "$name" != "ahmad" ]
        do
                echo $name: wrong name. Try again.
                echo Enter name
                read name
        done
:wq
```

Now modify the ***checkusername*** script from the previous lab such that it is called ***checkusernames*** instead and works as follows:

checkusernames

Enter user name to check or word "enough" to stop
u1112345
Enter user name to check or word "enough" to stop
u11
Enter user name to check or word "enough" to stop
u1123456
Enter user name to check or word "enough" to stop
enough

u1112345 = Salem Hamdi
u11 = No such user name
u1123456 = Sabah Khaled

Answer:

vi checkusernames

```
declare -a array
i=-1
echo Enter username to check or word "enough" to stop
read name
while ["$name" != "enough"]
    do
    i=$((i+1))
    array[$i]="$name"
    echo Enter username to check or word "enough" to stop
    read name
    done
```

:wq

Break and Continue Statements

The programmer can use ***break*** and ***continue*** statements inside shell script loops which mean the same as they do in the C language:
break - exit the loop immediately.
continue – stop running the current cycle but go back and check the condition.

In addition they can use ***break*** and ***continue*** followed by a number to specify how many loop levels they want them to work for. For example:
break 2
Will exit out of two nested loops if they exist.

until loop

The until loop is similar to the while loop, but stops when the condition becomes true.

```
until  false
do
        statements
done
```

Modify the above two programs such that they use the until construct instead of the while construct and try them out. Did they work? _____.

For loop

In shell scripts, the for loop is very powerful and useful. The general structure of the for loop is as follows:

```
for  item  in  list of items
do
        statement(s)
done
```

What makes a for loop powerful is the different ways a list of items may be specified.
Let us start with a simple example:

```
vi names
        for  name in ahmad hamdan subha khaled
        do
                echo $name
        done
:wq
```

Run the script names. It should display the names given in the list.

Now change the first line in script names to the following:

*for name in $** (remember that $* holds all the arguments as a list)
and run the modified script as follows:

```
names  ahmad subha khaled
```

What happened? _____.

Rewrite the delete script we wrote at the beginning of this lab such that it uses a for loop instead of a while loop. Did it work? _____.

The best feature about the for loop is that we can treat the output of a command as a list of items as follows:

```
vi lines
    for line in $(cat /etc/passwd)
    do
        echo $line
    done
```

Using a for loop, write a script called *comp311* that lists the full names of all the users that are registered in the comp311 course.

Answer:

```
for line in $(cat /etc/passwd)
do
    groupId= $(echo $line | cut -d: -f4)

for i in $(grep comp311 /etc/passwd|cut -d: -f5)
  do
  echo $i
  done
```

Now rewrite the script *comp311* such that it will display only the names of the users that are currently logged in to the system. (hint: use the output of the who command)

Answer:

```
for login in $(who |tr -s ' ' |cut -d' ' -f1)
  do
  name = $(grep $login /etc/passwd |cut -d: -f5)
  echo $name
  done
```

The for loop can also be applied to a directory of files as follows:

```
vi myfiles
for file in *
do
        echo $file
done
```

Write a script called ***filetypes*** that uses a for loop to type the name and type (file, dir, or unknown) for each file in a given directory as follows:
Assume that I use the script in the following way:

filetypes /etc

then the script should display the names of all the files under directory /etc and the type of each of those files:

Answer:

```
for file in $(ls $1)
do
    echo $file
done
```

```
#! /bin/bash
for file in $1/*
do
 if [ -f $file ]
 then
 echo $file: is file type
 elif [ -d $file ]
  then
  echo $file: is directory type
 else
  echo $file: is unknown type
 fi
 done
```

The which command displays the directory in the PATH that contains the command. Try it as follows:

which ls

What is the result? /bin/ls .

Write a script called ***mywhich*** that simulates the which command. You are not allowed to use the which command in your script. (***hint: use the for loop and the sed command***).

Answer:

```
#!/bin/bash
for list in $(echo $PATH | sed "s/:/ /g")
    do
    if [ -f $list/$1 ]
    1 then
      echo $list/$1
      exit 0
    fi
    done
    echo $1 = No such command
```

Lab12. Security and Networking Concepts

Objectives

After completing this lab, the student should be able to:

- Understand through example the importance and usage of set user id (suid) and set group id (permissions) in Linux.
- Set and modify suid and sgid values on Linux files.
- Identify and learn some Linux networking tool basics.

Suid and Sgid

Linux systems are very secure and have multiple levels of security that takes volumes to discuss. We have already talked about a part of one of those security levels which is file security where we explained the permissions (mode) and how they are used to control who can access and use files and directories. The permissions we talked about were the read (r), write (w), and execute (x). In this lab we will present a less obvious, but very powerful permission called the setuid (set user id) and setgid (set group id) permission usually referenced with an (s) permission.

Set User Id (suid) Permission

To understand how the suid permission is used let's take an example based on the passwd command which we use to change our passwords.

run the command

which passwd

This gives you the absolute path name of the passwd command (usually /usr/sbin/passwd).

now run the command ***ls –al*** on that file as follows:

ls -al /usr/bin/passwd (or whatever the which command produced)

Notice the permissions on that file.

What are they? _____.

The (s) on the user part of the mode is the suid. This (s) is very important and without it a user will not be able to change his/her password.

When a command such as passwd is executed, a process is created as explained in lab 6. That process has many properties. Four of those are:

real uid (real user id)
real gid (real group id)
effective uid (effective user id)
effective gid (effective group id)

The real uid and gid are the same as the username and group of the user executing that command. The effective uid is the same as the real uid and the effective gid is the same as the real gid except when there is an (s) permission. In this case the effective uid will be same as the owner of the file and the effective gid will be same as the group name on the file.

The process resulting from running the passwd command has an effective uid as root (owner of the file passwd) which is why this command is able to open and modify files (e.g. /etc/passwd and /etc/shadow files) which the user running the command is not allowed to.
This gives great flexibility by giving regular users the ability to access files through running commands which they cannot access normally.

Set Group id (sgid) Permission

Let us now see how the same idea is applied to the sgid.

run the command:

which write

what are the permissions on the write command? _____.

What is the name of the group name on the write command ? _____.

Using the ***who*** command find the pts file for your neighbor. Now run the ls –al command on that pts file in the dev directory as follows:

Assume the pts file is 5 then you run:

ls -al /dev/pts/5

what is the group on that file and what permission does the group have?

_____

Now to see how that helps try to write a message directly to your neighbor's terminal as follows:

echo hello > /dev/pts/5

What happened? _____.

Now using the ***write*** command write the same message to your neighbor's terminal as follows:

write u1112233
hello
ctrl-d

What happened? _____.

In both cases, it was you (same user with same permissions) that was trying to write a message to the other user's terminal. ***Why did it not work when you tried to do it directly while it worked using the write command? (hint the (s) permission on the write command).***

Adding (s) Permission

To add the s permission to your files, use the chmod command with four digits instead of three as before, for example:
create a file called newfile (touch newfile).

chmod 2777 newfile

What permissions are now on file newfile? _____.

chmod 4777 newfile

What permissions are now on file newfile? _____

chmod 6777 newfile

What permissions are now on file newfile? _____

As you can see adding an even digit (2 or 4 or 6) will put (s) on group, user, or both respectively.

What command would you use to set the permissions on newfile to:

1. *r_s_wxrwx* _____
2. *r_xrwsr___* _____
3. *rwSrwsr___* _____

How do you get a capital s (S) and a small s (s)?

_____.

Networking

As users we are not allowed to modify network setups, but we can view some information on how networks are configured on Linux.

Run the command:

/sbin/ifconfig

What is the ip address (inet) of your machine?

_____.

What is the MAC (HWaddr) address of your machine?

_____.

What is the netmask used by your machine?

_____.

Run the command

/sbin/route

What is the default gateway?

_____.

Run the command:

/bin/netstat -n | grep 23

This will give information about the telnet connections made to/from the system.
List the quad (Socket Connection) for your telnet connection:

_____.

Use the ftp tool to copy files from windows to Linux and vice versa.
Show your work to the instructor.