Ps -S -ef

which command

بتحكي وين موجود

*set : local + environoment variable

*env: environoment variables only

3- EUID.

Run the command:

```
env | more
```

→ variable جواها

Is the output the same as the set command or different? different.

What is the difference between set and env? (hint: Check the man pages).

Set : variable are set in local shell (bash), but env set variable as global for shells.

*User-Defined Variables

Users can define their own shell variables to simplify their work or store values for later use.

Under your home directory (*cd*) create the following structure:

```
mkdir project
mkdir project/myfiles
touch project/myfiles/firstfile
```

Now create a new variable called **myprojfiles** as follows:

```
myprojfiles=$HOME/project/myfiles
```

a shortcut for the path

Now you can use the new variable to manipulate your project directory. Try the following commands and write what each does:

```
vi $myprojfiles/firstfile
```

opens the file to edit.

```
cp /etc/passwd $myprojfiles
```

~~mov~~ copies files from /etc/passwd to the directory $myprojfiles

```
touch good; mv $HOME/good $myprojfiles
```

makes a file in the home directory named good, then moves it from home to the directory we make

To summarize, a shell checks a command for any variables (words starting with $) and substitutes them with their values before executing the command. E.g. in the command *echo $PWD* the shell first substitutes the variable **PWD** for its value and then executes the echo command on that value.

Command Substitution

تبديل

Another important shell function is command substitution where the shell substitutes commands with their results before executing the main command.

* command doesn't print when you don't type ( )

Try the command:

```
        date
```

What is the result? current date.

Now try to command:

```
        echo   $(date)
```

* ***What is the result?*** current date (command sustitution).

The result of both commands is the same, but for different reasons. In the first case, command *date* is executed and the result of the command is displayed. In the second case, the shell first substitutes the result of the command *date* (which is indicated using the ***$(command)*** notation) and then executes the main command echo on that result. Thus, the output of the *date* command is used as an argument for command ***echo***.

Command substitution is very useful for saving command outputs in variables for later use.

Run the command:

```
        grep yourusername  /etc/passwd | cut -d:  -f5 | cut  -d_  -f1
```

What is the result? Dana ~~u1201112:x:1201112:66:Dana-Abdallah-Ismail~~ Atesh

To get that result again you need to run the same command each time. You can save the result of that command in a variable for later use using command substitution as follows:

```
        firstname=$(grep yourusername  /etc/passwd | cut  -d:  -f5  | cut  -d_  -f1)
```

Now you can use the variable **firstname** whenever you need it. This is especially useful in shell scripts. You can run the following command for example:

```
        echo  how are you doing  $firstname?
```

The notation ***$(command)*** is common to many shells, but not all. The **csh** shell does not use that notation. There is another older notation which is understood by most if not all shells. Instead of ***$(command)*** the notation used is ***`command`*** (The single quote used here is the one below the ESC key on the keyboard).

Try the new notation to get your last name and save it in a variable called ***lastname.***

Command: lastname=`grep dana /etc/passwd|cut -d: -f6 | cut -d/ -f3'

Aliasing

Another function of the shell is aliasing which is basically used to give new simple names to complicated or long commands. For example:

```
        alias  dir="ls  -ali"
        dir
```

The new alias **dir** will now behave exactly as ***"ls –ali"*** when executed.

To display the aliases you already have on your system, run the command:

```
alias
```

List three aliases that you have and their values: (no space after =)

1- ~~alias~~ alias ds = "clear".

2- alias vi = 'vim'.

3- alias his = "history".

alias sl = "ls"

To cancel an alias, use the ***unalias*** command. For example:

unalias dir (cancels the ***dir*** alias)

Always be careful of aliases that have the same names as commonly used commands. An alias such as the following may be very dangerous. Do NOT try it.

alias ls="rm -f *"

Command Line Editing

The commands you enter on the command line are stored by the shell in a history file called ***.bash_history*** under the bash shell. To use or modify commands you executed earlier you can use the arrow keys. The up and down keys are used to get commands and the left and right arrows are used to move and modify a command if needed.

Rename (use the mv command) the file .bash_history to .save_history.

Command: mv .bash_history .save history.

Exit from the system and log back in.

Check the commands stored in .bash_history. What did you find? Why?

mv .bash_history .save history

.bash_history file doesn't exist (renamed with new file name, .save_history)

What can you do to restore all your previous commands?

mv .save_history .bash_history.

File Name Completion

Another useful shell function is file name completion where the shell completes long file names for you when you type commands as follows:
Suppose I have a file called : abcdefghijklmnopqrstuvwxyz and I need to copy it.
All I have to do is type:

cp abcESCESC newfilename

If there are no other files starting with abc then the shell will complete the long name for me. If there are other files that start with abc then the shell will display them and I need to specify the first different character and then press ESCESC for the shell to complete the name.

Making changes permanent

Many of the changes mentioned above such as creating new variables, changing existing variables, or creating aliases will disappear after exiting and logging back into the system. To make those changes permanent, they need to be added to your environment file (**.bash_profile**). Be very careful when modifying this file and always copy it first before making modifications.

Copy the file .bash_profile to file .save_bash_profile

Command: cp. bash-profile .save-profile .

Add the following to the end of your .bash_profile file:

1- Add the . (current directory) to your PATH variable
2- Add a variable called myproj with the name of a project directory under your home directory.

PATH = $PATH:.:
myproj = $HOME/Project /myfiles

Save the file and quit.
Exit the system and then log back in.

Check to see if the changes still exist on the system. Do they? Yes .

Lab6. Shell Usage and Configuration (II)

Objectives

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

- Understand and use shell input, output, and error redirection.
- Use pipes to join several Linux commands into single powerful commands.

I/O Redirection

Commands (and programs) usually receive input and then produce output and error. By default the input is usually received from the keyboard and the output and error are usually both directed to the screen. Linux shells allow us to change those defaults and redirect input, output, and errors.

Input Redirection

To understand input redirection, let us first use the ***mail*** command. The mail command is the default command used to send and receive mail on most UNIX based systems. To send email to another user simply use the command:

mail username (username@system if on another system)

You can try sending yourself an email by typing:

```
mail yourusername
subject:hello
This is my mail message
Goodbye
.
cc:
```

As you can see the mail asks you for a subject (title of message) and you end the mail by typing a dot (.) by itself on a line and then pressing enter.

To read your email, you can simply type:

mail

You will get the & sign. Type ? for help on how to use (read/delete/save/reply/forward/…) the mail program. To quit just type **q** and Enter.

The input for the mail command was received from the keyboard (default). You can redirect the input such that it is received from a file. To do that use the (<) character as follows:

Create a file called message and type the following two lines inside:

```
This is my message file
Goodbye
```

Then save and quit

Now run the following command:

```
mail -s hello yourusername < message
```

The input in this case was redirected to come from file *message* instead of the keyboard.

Another example is the *tr* (translate) command. This is a useful command used to change input characters and may be used to encrypt characters.

Run the command

```
tr "a-z" "A-Z"
how are you
```

The result is "HOW ARE YOU". As you can see the input was received from the keyboard. You may redirect the input to come from the file message you created earlier as follows:

```
tr "a-z" "A-Z" < message
```

What was the output?

makes the letters inside the file message all

Upper case

You can append the redirected input using the here text (<<). Run the following command:

```
tr "a-z" "A-Z" <<!
➢ hello
➢ how are you
➢ hope well
➢ bye
➢ !
➢
```

What did you get as output?

```
HELLO
HOW ARE YOU
HOPE WELL
BYE
```

Output Redirection

The output of commands is sent to the screen by default. You may redirect the output by using the (>) character. Run the command:

```
ls –al
```

The output will be shown on the screen.
Now run the command:

```
ls –al > lsfile
```

No output will be displayed on the screen. View the file **lsfile** using the *more* command. It should contain the output of the "*ls -al*" command.

Using the (>) character will create a new file or overwrite an existing file.
To append the output to a file, you can use the (>>) character as follows:

```
ls  -al >> lsfile
who  >> lsfile
echo hi  >> lsfile
```

One of the main Linux philosophies is that everything is treated as a file including hardware devices. To interact with hardware devices, Linux interacts with device files which represent those hardware devices. This means that if we are able to redirect input or output from/to files then we actually do the same with devices. We can try this with device files that represent our terminals (screens).

Open two terminals (if using telnet then do two telnet connections).
Run the command:

```
who
```

Record the *pts* numbers (you should have two, one form each terminal).
Assume the terminal you are working on has *pts/4* and the other terminal has *pts/5* (you need to use your numbers when testing).
Type the following command:

```
echo hello
```

This will display the word hello on your current terminal (i.e. pts/4) which is the default.
Now type the following command:

```
echo  hello  >  /dev/pts/5
```

What happened? Explain.

prints the string hello on the terminal that
has number 5

Error Redirection

Command output is sometimes mixed up with command errors since they are both sent to the screen by default. Run the following command:

```
        cp
```

What did you get displayed?

missing file operand.

Is that output or error? error.

Now run the command:

```
        cp   >   cpfile
```

What happened? puts the output of cp in cpfile.

Since the same message got displayed on the screen and was not sent to file ***cpfile*** then it must not be output. It is error.

To understand how to redirect errors, we should learn about file descriptors. There are three file descriptors used by programs to specify input, output, and error.

Standard input has file descriptor 0
Standard output has file descriptor 1
Standard error has file descriptor 2

There is no need to use the file descriptors 0 and 1 when redirecting input and output respectively since they use two different characters namely < and >.
To redirect error we need to use the (>) character so to distinguish it from redirecting error, we must specify the file descriptor before the > character as follows:

```
        cp   2>   cpfile
```

What happened now? the error ~~is the~~ was sent to cpfile.

Check the contents of file cpfile. What did you find?

missing file operand.

Redirecting output and error to different places may be very useful especially when dealing with commands that produce both at the same time. Try the following command:

```
        find   /   -name   passwd   -print
```

What did you get? Was that output or error? both.

Now run the command as follows:

```
        find   /   -name   passwd   -print   2>   errors
```

What did you get now? output only.

Check file errors content.
Now run the command as follows:

```
        find  /  -name   passwd   -print   >   qutput     2>    error
```

What happened? empty but put output here put error here.

Check both files output and error.

To append errors use (2>>).

Pipes

One of the main Linux philosophies is to have commands where each does one thing very well. For example, the ls command has so many options to display file information in so many different ways. Another philosophy that complements that is the ability to join different commands together in a chain to produce more powerful commands. This is usually done using pipes.

Run the following command:

```
cat /etc/passwd | grep yourusername | cut -d: -f5 | cut -d_ -f1
```

What did you get? The first name of ____________.

This command is made up of four different commands joined together using pipes (|). Pipes usually work with commands we call **filters**. They take input and filter it to produce a certain output. They usually do not change the original input source. This is how the above command works:

"*cat /etc/passwd*" produces the passwd file (many lines) as output. The **passwd** file is passed as input to the "*grep yourusername*" command which in turn filters that into a single line that contains your username. This line is then passed to the command "*cut -d: -f5*" which filters it to one -field (field five) (-f5) based on dividing fields by delimiter : (-d:). This output is then passed as input to the next cut command "*cut -d_ -f1*" which filters it to get the first field (your first name) by cutting based on delimiter underscore (-d_). The output (your first name) is then displayed on the screen.

What command would you use to get your group number from /etc/passwd:

sat /etc/passwd | grep username | cut -d: f4 .

What command would you use to get your login time from the who command?
(Hint: use the tr command with the squeeze option)

who | tr -s " " | cut -d" " -f4 .

What command would you use to get the default group name for any given user? command sustitution

grep $(grep username /etc/passwd | cut -d: -f4) /etc/group

group number

group name

~~cat~~
cat: shows content at once
more: ,, ,, in pages

Try the following command:

```
find / -name passwd -print | more
```

What happened? Why is the result of the command not filtered by more?

it prints out the errors and the outputs

How can we fix this? output and error redirection

find / -name passwd 2> file1 > file2

putting the output and the errrr in different files.

or
2>&1
بتوجه الايرور برن
مع الوت بوت

or
put the error in
find / -name passwd 2> dev/null
محرقة

find: search and locate the list of files and directories based on conditions you speify for filee that match the argument.

* how to add a user in linux?
sudo useradd -d /home/ahmad -g 66 ahmad

* add a new group
vi /etc/group
exp: teachers:x:55

* change shell to a user
chsh -s bin/bash user
exp

*which → where the commands come from

Lab7. Job and Process Management

Objectives

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

- Manage several jobs running in the background.
- Understand how processes are created using the fork and exec steps.
- Control the priority of newly created processes using the nice command.
- Identify and use signals for manipulating processes.

Job Control

Sometimes we need to execute more than one job on the same terminal, but we are forced to wait until one command is done executing and getting the shell prompt back before we can execute the next command. This is especially a problem if the one of the jobs we are executing takes a long time such as a backup job. To get around this, Linux allows us to run several jobs at the same time in the background. This is called job control.

To be able to understand job control, we need to create and use a command that will take a long time. To do this, we do the following steps:

1- Create a new file called forever using vi as follows:

```
vi forever
        while  true
        do
        echo  running  >  myfile
        done
:wq
```

This is basically a script file with an infinite loop.

2- Now we have to make sure that our PATH variable includes the current directory (.). This step is important for the shell to locate our newly created command forever. This is done as follows:

حتى لما ننادي forever ما يعطيني خطأ path

```
PATH=$PATH:.
```

3- The third step is adding the ~~execute (x) permission~~ to the command to make it executable. This is done by adding x to all parts of the mode as follows:

```
chmod   +x   forever
```

Now we have a command called *forever* that runs for a long time and that can be used to understand job control.

To run a job in the background, we follow the command with an ampersand (&). In our case we are going to run three forever jobs in the background as follows:

helps you run the command in the background and continue working

```
forever&
[1][2000]
```

```
    forever&
    [2][2500]
    forever&
    [3][2503]
```

Each time we run a job in the background the system displays two numbers.. The first is the job id number and the second is the job process number. These numbers are important to be able to reference the job later on for manipulation.

We can display our background job by using the command:

```
    jobs
```

This will display an output similar to the following:

```
[1]     Running     forever
[2] -   Running     forever
[3] +   Running     forever
```

The number is the job id number. The plus and minus signs reference the last and the one before last jobs. The status of all jobs is running. The last column is the name of the command used to create the job.

We can manipulate the jobs in several ways, as follows:
To get a job back to the foreground we use the *fg* (foreground) command followed by the job id number. E.g. to get job 2 to the foreground, we run the command:

```
    fg   %2
```

use ctrl +z
بتضلها ع ال loop*
ctrl + c
بتسكّر

This brings the job to the foreground. To send the job to the background, we press *ctrl-z*. The job is moved back to the background.
Run the following command:

```
    jobs
```

What do you notice different about job # 2?

it's stopped running.

To resume a stopped or suspended job, we use the *bg* (background) command followed by the job id number. To resume job 2 (change its status to running) we use the command:

```
    bg   %2
```

Run the command:

```
    jobs
```

What is the status of job # 2 now? it's running.

To terminate a job we use the ***kill*** command followed by the job id number. E.g. to kill job 3 we issue the following command:

stop working ←

```
    kill   %3
```

If we type the command: ***jobs*** quickly enough we will see the status of job 3 changing to Terminated and if we check again it will disappear.

Do the following:
kill all remaining jobs such that none are in the background.
Write the sequence of commands needed to have the following output displayed when the command "jobs" is issued:

```
[1]   Stopped      forever
[2]-  Terminated   forever
[3]+  Running      forever
```

Commands:

fg %1
ctrl + z

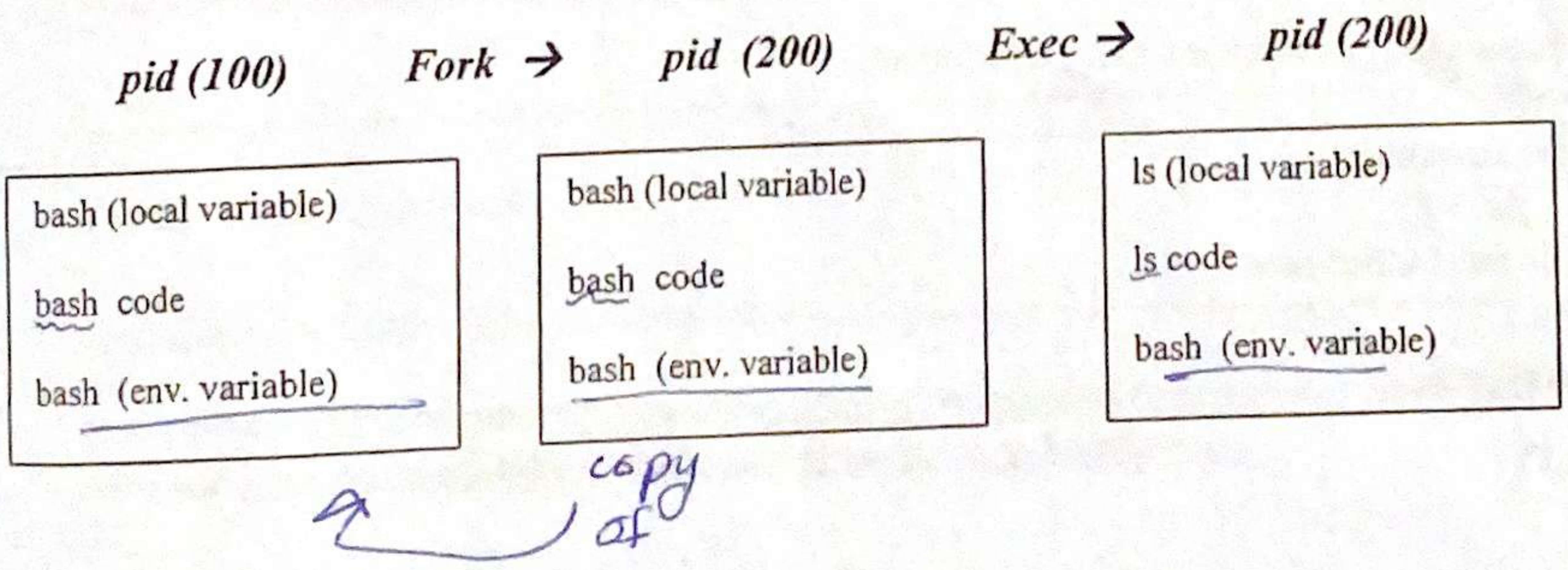
~~fg %2~~
~~ctrl~~ kill %2
fg %3
ctrl + z
bg %3

Process Control

A process is simply a program in execution. Each command we run results in one or more processes. There are several processes running in the background that allow us to use the system and provide us with different services. Interacting and manipulating processes is called process control.

When a command is run, a duplicate copy of the parent process is created using the fork function. This copy is similar to the original except for its process id number (pid). After that the system executes the command using the exec step which basically loads the new command on top of the copy created as follows:

When we run the command ls under the bash shell, a copy of bash is created and which is replaced by ls.



pid (100) *Fork →* *pid (200)* *Exec →* *pid (200)*

bash (local variable)
bash code
bash (env. variable)

bash (local variable)
bash code
bash (env. variable)

ls (local variable)
ls code
bash (env. variable)

copy of

* الامر بنتهي لما ال prompt ترجع

* Zombie pocess

PID
PPID

Notice that the environment variables are passed from the parent process (**bash**) to the child process (**ls**).

Let us now run through to see the some details on what happens above.
To view process information, we can use the ps (process status) command. To see our running processes we use ps with the –f option as follows:

ps -f

Describe the output?

-bash , ps -f ,
Terminalted

Let us create two variables called var1 and var2 respectively.

var1=first
var2=second

When new variables are created they are defined as local variables. To change a variable from local to environment, we export it (use the export command). Let us make var2 an environment variable as follows:

export var2

The ***set*** command is used to display both local and environment variables. The command ***env*** is used to check the environment variables only. Let us check for **var1** and **var2** in our main process (bash shell):
Run the command:

set | grep var

Which of the two variables (var1 and var2) do you see in the output? Why?

both var1, var2

Now run the command:

env | grep var

Which do you see now? Why?

novor one or var2

opens a new shell

Now run a child processes (ksh) as follows:

ksh

Run the command:

ps -f

What is the output now?

father
-bash
Ksh

Notice the numbers pid (process id) and ppid (parent process id). Those should tell you that bash is the parent process and ksh is the child process.
You are now in the child process. Let us check for the variables **var1** and **var2** in the child process (ksh).
Run the command:

```
set | grep var
```

Which of the two variables (var1 and var2) do you see in the output? Why?

~~both 2 var 1 & var 2~~ ~~both not mark~~ only var 2 b/c. it's exported

Now run the command:

```
env | grep var
```

Which do you see now? Why?

only var 2 .

This shows that only environment variables are passed from parent processes to child processes.

As shown above any created process goes through the **fork** and **exec** steps explained above. We can use the *exec* command to skip the **fork** step and just do the exec step and see what happens, as follows:
Run the command:

```
ps –f
```

You should have three processes (bash, ksh, and ps –f). ps –f does not exist anymore. Now register the pid number for the ksh process. Now instead of running the "ps –f" command as before, run is as follows:

```
exec ps –f
```

do full format listing

What processes do you see now? What happened to ksh (hint: note the pid number for the ps –f process)

-ksh disappears from terminal , doesn't make a copy (fork)

& puts it on the original. the child comes over the parent

What would you expect to happen if you run the command "exec ps –f" again?

____.

Try it. What happened?

gets out of the terminal .

This shows that processes do go through both the fork and the exec steps, otherwise a new child process will take over its parent process and destroy it.

-max +19 } ⇒ normal user can decrese priority
min -20 } ⇒ " " can't increase "

Priority: 0 — 139 in linux system
0 - 99 for reel time, 100-139 for users

nice
-20 — 19
high lowest
0 is defult

Pri = nice +20

Nice command

Users may decrease the priority of their processes (especially those that take a long time and are not of high priority such as backups) to allow other users to run their processes at a higher priority. When they do that, they are nice and to do that they use the nice command. The only user that can both decrease and increase the priority of his/her processes is the root (system administrator). Let us see how the nice command is used.

sudo

Run the command:

```
ps -l
```

Note the two new columns displayed namely:

PRI (which refers to the priority of the process)
NI (which refers to the nice value of the process)

Now run the above command as follows:

```
nice -6 ps -l
```

long format

Notice what happened to the **PRI** and **NI** values for process "**ps -l**". They increased. Increasing the priority number actually makes the priority for that process less.

Now try to run the command:

nice --8 ps -l *(--8 = two dashes then 8)*

What happened? Why?

permision denid, you can't increase your priorty if you want to (use sudo).

Signals

Users can control their processes through sending signals using the "kill" command. There are many signals that may be sent to a process. To get a list you may use the following command:

```
man 7 signal
```

There are three interesting signals that stand out. Those are namely **TERM** (also called SIGTERM) which has the number 15, **HUP** (also called SIGHUP) which has the number 1, and **KILL** (also called SIGKILL) which has the number 9. The default signal is the TERM signal.

The TERM signal tries to terminate signals cleanly and may be blocked by processes such as shells. The HUP signal is used to restart a process to have it upload any changes in its configuration files. The KILL signal is used to kill a process uncleanly and cannot be blocked. Let us try the TERM and KILL signals:

Run the command we created in the beginning of this lab (***forever***) in the background and note the process id number given (let us assume it is **1234**). Check to see that the process is running in the background (use the ***jobs*** command).

Try the following command:

kill 1234 (use the number shown for your process)

Now recheck if the process is running with the ***jobs*** command. What did you find?

____.

Now repeat the same steps (i.e. create the *forever* job in the background and check its PID (we are assuming its 1234, but it could be anything)).

For ***each time*** you create the *forever* job try killing it with one of the following commands:

clean

kill -15 1234 (specify the correct PID, we are assuming its 1234)
kill -TERM 1234
kill -SIGTERM 1234

What did you notice about each of the three commands above?

If you use -15 ~~g~~ to kill your bash it may not work if it. was the terminal but if you use -9 or -1 with it it will kill it , -15 is enough to kill processes

Open two terminals (if you are using telnet then open two telnet connections)
Use the *ps* command to determine the process id number of the terminal you are not using, as follows:

ps –f

What is the pid number for the bash process running on the pts number different from the pts number that your ps –f process is running. That is the pid you need. Now try running the following command:

kill pidofbash (or kill -15 pidbash)

What happened? Why?

The other terminal got terimenated.

Now try the following kill command:

not clean

Kill -9 pidofbash (-9 is equivalent to –KILL or -SIGKILL)

Now what happened?

it will still close the terminal.
other

Lab8. Text Processing Tools and Regular Expressions

Objectives

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

- Identify and use filters as valuable text processing tools.
- Use simple regular expressions to make text processing more efficient.

Text Processing using Filters

In the pipes lab, we mentioned a group of commands called filters. These are basically commands that take some input and then filter it to produce the requested output without changing the original source of input. In this lab we will practice how to use some filters as useful tools for text processing.

The filters we will use are (***use the manual pages (man command) to get more information on those filters and their available options***):

head and tail: used to display lines from the beginning or end of a given input respectively.
cat: used to view or concatenate files.
grep: used to extract certain rows (lines) from a given input. We will concentrate on the options -i, -l (EL), -v.
cut: used to extract certain columns from a given input. We will use the options -d, -f, and –c.
tr: translates (changes) a given input to a specified output
wc: used to count lines, words, or characters in a given input.
sort: used for sorting a given input. We will present the options -i, -o, -u, -n, -k, and –t.
sed: used for stream editing (changing parts of an input to a specified output)

Create the following file called ***students*** using the ***vi*** command and then save and quit:

```
ah6:506:Ahmad_Hamdan
sh5:345:Suha_HAMDAN
rd7:427:Ribhi_ahmad
hr4:234:hamdan_ribhi
ad6:386:Arwa_Ahmad
ad5:285:ahmadi_Ahmad
```

Each line of the file ***students*** contain three fields: ***student user name*** (e.g. ah6), ***student id number*** (e.g. 506), and ***student full name*** (first and last names separated by an underscore e.g. Ahmad_Hamdan), respectively.

Let us now try our filters on the file *students*. Write down what each of the following commands does. Make sure you understand why each command behaves that way.

head -2 students

displays the top 2 students in the file (First two lines).

tail -3 students

displays the last 3 lines in the file students.

What command would you use to get the fourth line only from file students (hint: mix head and tail with pipes):

head -4 students | grep -i hamdan_ribhi.

cat students

displays the content of file student.

grep ahmad students

displays the students whose name is exactly "ahmad" case sensetive

Join both cat and grep with pipes to get the same result as the previous grep command:

cat students | grep ahmad.

grep -i Ahmad students

displays all names containing ahmad ignoring the case.

grep -l Ribhi *

only names of files containing "Ribhi" are written st90/P only files in current directory are searched

grep -v Ribhi students

invert ~~search~~ match , selected lines don't contain the word "Ribhi"

invert match

grep -iv hamdan students

invert match , outputs lines not matching any of the patterns.

→ names are listed per file searched.

cut -d: -f2 students

outputs 2nd column of all.

What command would you use to get the last names for all users in file students:

cut -d: -f3 students |cut -d_ -f2 | tr "_" " ".

What command would you use to get the first names of all users with last name hamdan (all cases):

grep -i hamdan students |cut -d: -f3 | cut -d_ -f1.

cut -c2,3 students

-c specifies character positions to cut each line.

What command would you use to get the middle digit in the id numbers for all users with last name hamdan:

grep -i ~~students~~ hamdan student] cut -d: f2 | cut -c2.

tr "a-z" "A-Z" < students (Describe output)

outputs the content of students in uppercase.

What command would you use to get the first names (all in lower case) of all users that have the word ahmad (all cases) as part of their full name:

grep -i ahmad students | tr "A-Z" "a-z" |cut -d: -f3 |cut -d_ -f1.

wc -l students lines

shows number of lines in students (6 students).

head -1 students | cut -d: -f3 | cut -d_ -f2 | wc -c

eof

shows number of characters in 2nd name of first student (t)).

What command would you use to count the number of files in your home directory? Hint: use the ls and wc commands:

ls -1 | wc -l.

sort students (Describe output)

sorts students based on the ASCII value of the first character, if equals it looks at the 2nd character.

sort -o result students (What happened?)

outputs sorted content of students into file "result", instead of i/o

field separator

sort -k2 -t: -n students (Describe output)

sorts numirically based on the 2nd file with fields separated by :.

What command would you use to list all the first names of users in file students sorted based on lower case letters and without repetition (hint: check the -f and -u options for sort):

convert all lowercase into uppercase

uniquekeys

sort -fu students|cut d: -f3 |cut -d- -f1.

sed 's/ahmad/damha/' students

switches every first ahmad to damha.

What is different when we run the same command with the i (ignore case) option, as follows:

sed 's/ahmad/damha/i' students

same as above but ignores uppercase sinstivity.

What is different when we run the same command with the g (global) option, as follows:

sed 's/ahmad/damha/ig' students

same as above but changes all accurance of Ahmad.

Regular Expressions

Some of the filters mentioned above such as *grep* and *sed* may use what we call regular expressions to be more powerful and precise. To get more information about the power and extent of regular expressions, you can read the man pages using the command:

man regex

We will just give a very basic introduction (a simple taste) to how regex may be used with some filters. The following are some common regular expressions:

pattern$: applied to a pattern if it is at the end of a given line.
^pattern: applied to a pattern if it is at the beginning of a given line.
[abc]: means a or b or c
[^abc]: means all characters except a, b, or c.

Let us try some commands with *regex*. Write down what each command does:

grep -i 'hamdan$' students

prints all ~~names~~ student that ends with hamdan ignoring case.

cut -d: -f3 students | grep -i '^ahmad'

prints all names that ~~one~~ start s with Ahmad.

cut -d: -f3 students | cut -d_ -f1 | grep -i '^ahmad$'

samee as above but prints the first name only.

cut -d: -f1 students | grep a[dh][^6]

prints all names that have both ah without th 6.

cut -d: -f3 students | sed 's/^ahmad/sameer/ig'

cuts name ending w/ ahmad ignore case.

sed 's/ahmad$/Sameer/i' students

students ending with ahmad will be changed to samer

Using what you learned above, write the commands that are needed to extract and display the following information from the /etc/passwd file:

Display the first names of all users whose last names end with the letter 'n' or 'm':

grep ^[nm] /etc/passwd

Display the last names of all the users sorted by their user id numbers (ascending order):

sort -k3 -t: -n /etc/passwd |

List the login names for all users with the bash as their default shell.

____

Display the default shell used by user root.

____

Display the number of files in directory /etc that end with the word .conf

ls -al /etc/ s | grep .conf$ | wc -l

Lab9. Shell Scripts (I) –Introduction

Objectives

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

- Create and execute simple shell scripts.
- Use positional parameters and shifting to pass command line arguments to scripts.

Introduction to Shell Scripts

One of the most powerful tools in Linux is the ability to group several commands into scripts in order to automate manual tasks. Scripts are also used as configuration and setup files in different areas of the Linux File System. Let us start by writing and executing our first simple script. To create and setup a script you need to do the following:

1- Using the vi editor open a new file and write your script as follows:

```
vi  myfirst
        echo this is my first Linux script
        echo I like it
        echo bye
:wq  (save and quit)
```

2- You now need to add the x (execute) permission to your script to run it. This is done only the first time you create your script. If you try to run your script and get the error "permission denied" then the reason would most likely be that you did not do this step. To add the x permission run the following command:

```
chmod  +x   myfirst
```

3- You must make sure that you have the following line added to the end of your environment setup file (.bash_profile):

```
PATH=$PATH:.
```

or ./myfirst

To make this step take effect you either exit the system and log back in or you run the following command:

```
.    .bash_profile
```

This adds the current directory (.) to your search path which would make the shell search for your script in the current directory. If you try to run your script and get the error "permission denied" then the reason would most likely be that you are missing this step.

This third step is only done once for all your future script to run without trouble.

Now you are ready to run your first script by typing its name on the command line as follows:

```
myfirst
```

What was the result of running the script?

____.