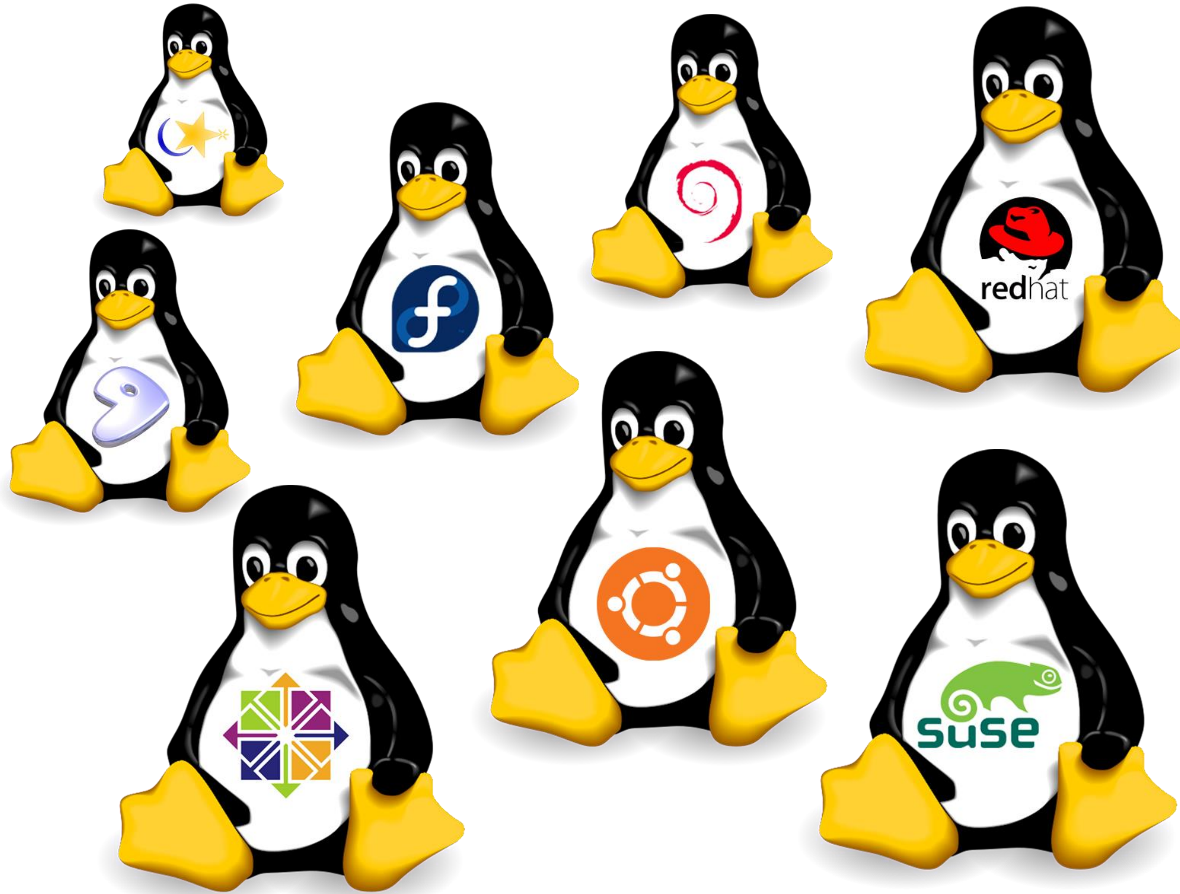
Comp311- Lab Linux

Lab5

Lab5. Shell Usage and Configuration (I)



Instructor :Murad Njoum

Objectives

After completing this lab, the student should be able to:
- Become familiar with **common Linux shells**.
- Recognize and manipulate system and user **defined shell variables.**
- Identify and use **shell functions like command substitution, aliasing, command** line editing and file name completion.

Linux Shells:

The shell contains

- **an interpreter** :It is the main program used by the user to access and use the Linux operating system. When the user enters a command and hits the Return key the shell checks the command and **rejects or accepts it**.

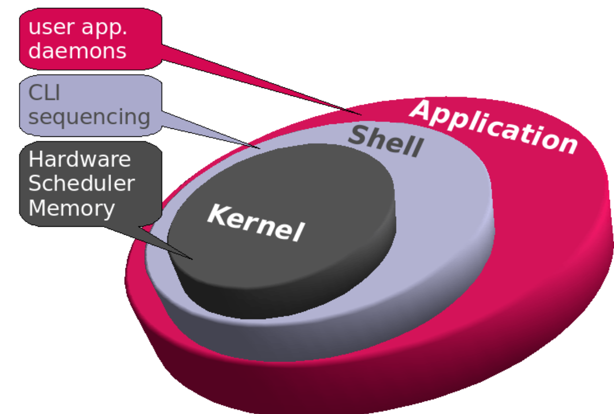
- Accepted commands are then **passed on to the Kernel** part of the OS for execution and the result is displayed by the shell to the user.

You are placed **into a shell** when you

- open a new terminal/console window
- start a shell from a terminal/console window, e.g., by running bash
- open a remote text-based connection as with through the **program ssh**

Like PuTTy, openSSH

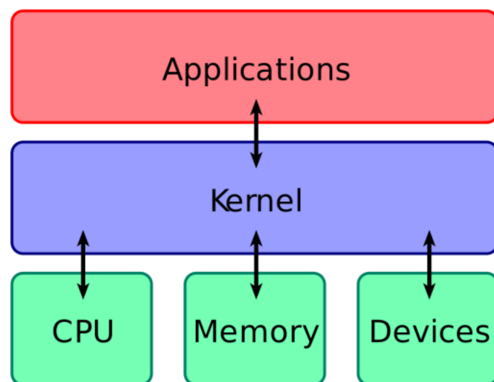
For more info : www.ssh.com

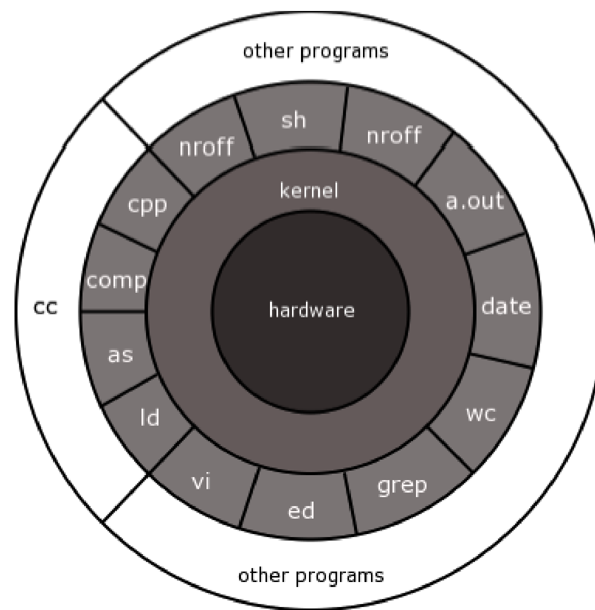


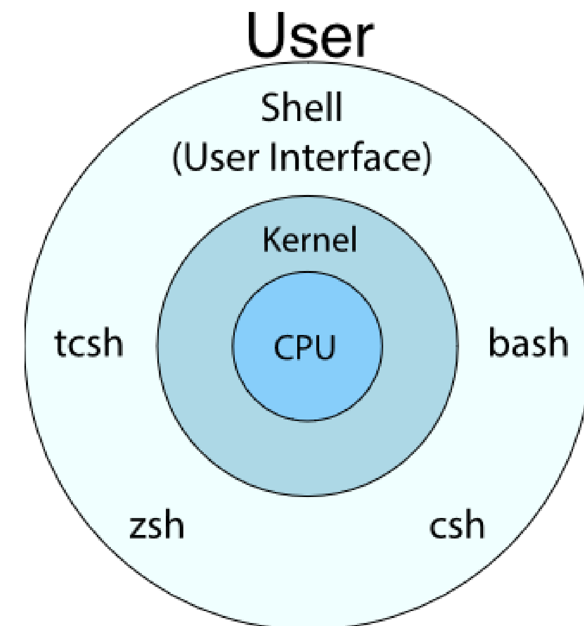
Linux Shells:

Linux provides several different shells

- **sh** (the bourne shell)
- **csh** – incorporated new ideas like history and command line editing
- **tcsh** – updated version of csh
- **bash** – updated version of Bourne shell (bourne again shell)
- **ksh** (korn shell)







Complier Vs Interpreter

Interpreter	**Compiler**
Translates program **one** statement at a time.	Scans the entire program and translates it as a **whole into** machine code.
It takes less **amount of time** to analyze the source code but the overall execution time is slower.	It takes **large amount of time** to analyze the source code but the overall execution time is comparatively faster.
No intermediate object code is generated, hence are **memory efficient**.	Generates intermediate object code which further requires linking, hence requires **more memory**.
Continues translating the program until the first error is met, in which case it stops. Hence **debugging is easy**.	It generates the error message only after scanning the whole program. Hence debugging is **comparatively hard.**
Programming language like Python, Ruby use interpreters.	Programming language like C, C++ use compilers.

echo

- Output instruction, form is
  - echo string
  - Where string is any combination of literal characters, $variables or $(Linux commands) `Linux commands`
- If you forget a $ you get the variable name without its value
  - echo Hello $FIRST LAST
    - outputs Hello followed by the value in FIRST followed by LAST literally because we forgot the $
- Assume FIRST=Fadi, LAST=Zidan
  - echo Hello $FIRST $LAST
    - outputs Hello Fadi Zidan
  - echo “Hello $FIRST $LAST”
    - outputs Hello Fadi Zidan
  - echo ‘Hello $FIRST $LAST’
    - outputs Hello $FIRST $LAST

Shells have many features and functions which allow them to perform their work. The following are some of those features or functionalities:

Variable Substitution:

A shell is a program that has **several variables** that help the shell do its work. Many of those **variables are system defined** variables (**usually written using upper case letters**) and
some may be user defined variables.

Let us consider some of the system defined variables to see how the shell uses them.
PATH variable:

Run the command:
echo PATH
What did you get? PATH __________________
Now run the command:
echo $PATH
What did you get? The full PATH ______________________________________

/home/mnjoum/.local/share/umake/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin

Variable Substitution:

To display the value of a variable you need to precede it with the ($) character.
The **PATH** variable is used by the shell to locate commands for execution. Let us see how the shell is affected by modifying that variable.
Run the following commands:

SAVEPATH=$PATH (saves the value of PATH in variable SAVEPATH)

ls	***Did it work?*** Yes___________________________
PATH=/etc	
ls	***Does it work now?*** NO_____________________***.***

Why? The path for command ls (/bin) was changed, no the interpreter does identified___***.***

Restore the original value for variable PATH.
Command? ***PATH=$ SAVEPATH***___________***.***

Variable Substitution:

Now try the command:
ls Does it work now? ______Yes______.

You can add directories to your PATH. To add the directory /etc to the end of the PATH

use the command:
PATH=$PATH:/etc

Try it and then use the command:
echo $PATH
Was it added as expected? ______Yes______.

PWD and PS1 Variables:

Display the value of the PWD variable.
Command? ___echo $PWD___. Print the current directory working

Change your directory to /etc. Command? ___cd /etc___

What is the value of PWD now? ___echo $PWD Or pwd___.

How do you think the pwd command works? ___cd .. cd.. cd.. , echo $PWD___.

Now run the following command:
PS1="Hello >"
What happened to your prompt? ___Hello>___.

Now run the command:
PS1="$PWD >"
What happened? ______________________________.

PWD and PS1 Variables:

There are several other variables such as HOME, PS2, SHELL, MAIL and so forth. To display the variables in your shell run the command:

set | more

List three more variables other than the ones mentioned above and their values:

1- ________________________________.

2- ________________________________

3- ________________________________

Run the command:

env | more

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______.

Environment Variables

- We don’t generally use our own variables (unless we are shell scripting) but there are useful environment variables, defined by the OS
- To see your environment variables, type **env**
  - **HOSTNAME** – name of your computer
  - **SHELL** – name of the current shell (e.g., /bin/bash)
  - **USER** – your user name
  - **HOME** – your home directory
  - **PWD** – current working directory (this is used by the pwd command)
  - **HISTLIST** – number of commands to be retained in your history list
  - **PS1** – your prompt defined
  - **PATH** – a list of directories **that bash will examine** with every command

Practice:

AGE=21

AGE=$((AGE+1)) // AGE becomes 22

AGE=$AGE+1

this sets AGE to be "21+1" (that is, the characters 2, 1, +, 1)

NAME="$FIRST $LAST"

if $FIRST is Ahmad and $Last is Zaid then NAME is "Ahmad Zaid"

Y=20

X=$((Y/5))

integer division, X is the quotient

Q=$((Y%5))

integer remainder, Q is the remainder

A=$(((X+1)*Y))

added parents to control order of operations

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 OR mkdir –p project/myfiles
touch project/myfiles/firstfile

Now create a new variable called **myprojfiles** as follows:
myprojfiles=$HOME/project/myfiles

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

____.

cp /etc/passwd $myprojfiles

____.

touch good; mv $HOME/good $myprojfiles

____.

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:

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

Try the command:
date **Current date**
What is the result?______________________________________
Now try to command:
echo $(date) **Current date**
What is the result?______________________________________

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?____.

Practice:

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 echo \`ls\`

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

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: **(be careful , No space after=)**

1- alias cls=”clear” ______.

2- alias vi=‘vim’ ______.

3- alias his= “history” ______.

4- 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 *”

Why we use Aliases ?

- You will define aliases to
  - save typing (shorten commands)
  - simplify complex commands
  - eliminate directory paths
  - safety (for instance, forcing the –i option with rm)
  - Typos (if you are commonly making the same mistake, for instance typing sl instead of ls)
    - alias sl=ls
- Defined aliases at the command line prompt
  - but the alias is then only known for this session, close the shell, lose the alias

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.

Try to view the content of .bash_history file? **vi .bash_history**

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 Tab, ESCESC

- Saves you from having to type a full directory or file name
  - Type part of the name<tab>
  - If unique, Bash completes the name
  - If not unique, Bash beeps at you, type <tab><tab> to get a listing of all matches
- **Example**: current directory contains these files
  - forgotten.txt frank.txt fresh.txt
  - functions.txt funny.txt other_stuff.txt
- You type less fo<tab>
  - bash completes with forgotten.txt
- You type less fr<tab>
  - bash beeps at you
- You type less fr<tab><tab>
  - bash lists frank.txt fresh.txt

Try

$ cd /etc

$ less update<TAB>

$ less lib<TAB><TAB>

$less lib<ESC>…<ESC>

Tailoring your bash Shell

- If you enter a variable or alias from the command line, that item is **only defined in the current session**
  - If you type **bash**, you enter a new session
  - If you exit this session, you lose those definitions
  - If you open another window, you do not have those definitions
- It is easier to define these items in a script that is executed at the start of each shell session
- This is where we will define our initial **PATH** variable and **any aliases**
  - We, as users, are free to edit these files to add to or change these definitions and add our own definitions

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.

Save the file and quit.

PATH=$PATH: . :

myproj=$HOME/project/myfiles

Exit the system and then log back in.

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

Thank You for attention !

Published By: Murad Njoum