Linux Shell Scripting

Part-1

I. Linux

Linux is an open source operating system that manages system's hardware and resources (CPU, memory and storage). An operating system connects the software and hardware of the system to make them function. It handles the communication between the physical resources and the software. Linux came in mid-1990s and it has been widely used today in smartphones, cars, devices, appliances, etc. Linux operating system mainly comprises of three components:

- **Kernel:** Kernel is the core component of Linux. It communicates with the system's hardware while managing resources of the system. Kernel is the lowest level of the OS and handles memory, process and file management (CPU, RAM, I/O).
- **System User Space:** This works as an administrative layer for system level tasks such as installation and configuration. This is a system library which can access kernel modules. It consists of init system, shell, command line, daemons (background services), processes that run in the background, desktop environment (users interact with desktop environment such as GNOME), user interface which is responsible of displaying the graphics on the screen (Xserver).
- **Applications:** Software using which user can perform tasks. Linux OS provides a lot of apps in its store to install. For example, Ubuntu software center provides apps which can be installed with a click!

Command line can be used to access the system directly.

I.1. Why do we use Linux?

Linux is a free, open source operating system. It is considered as the most reliable and secure platform. Linux can be used, run, modified and redistributed by any user. Anyone can contribute

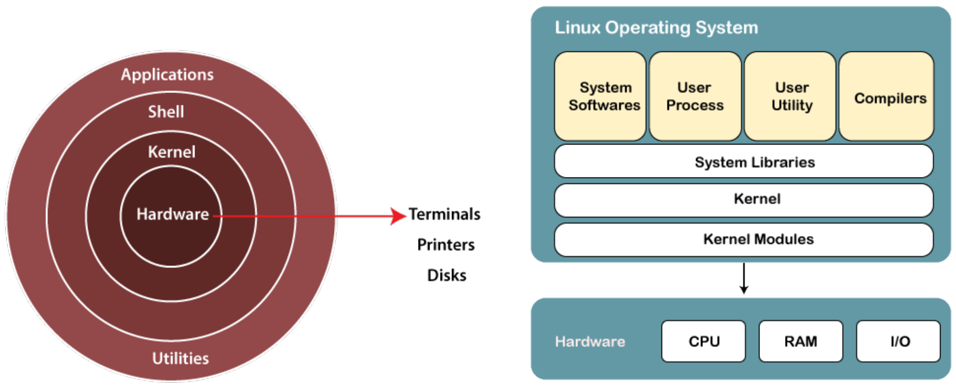
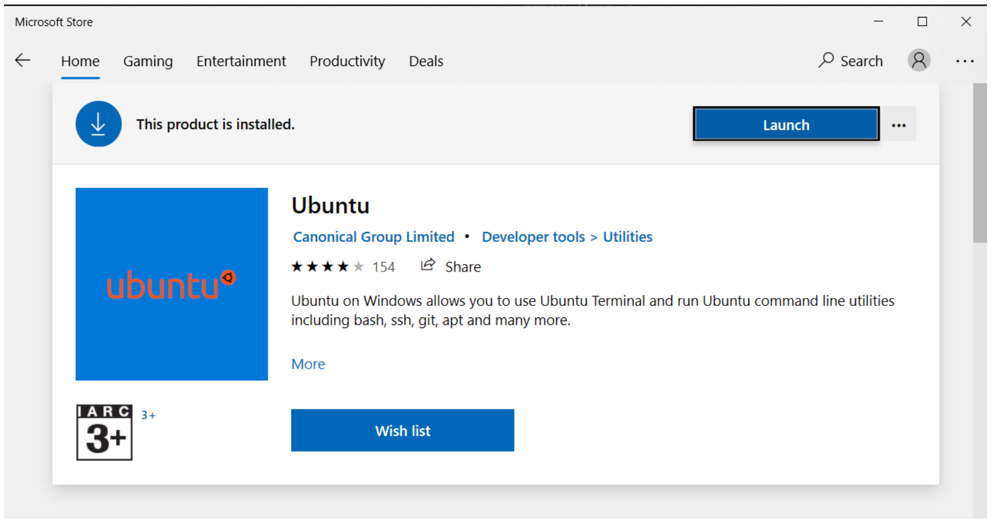


Figure 1: Architecture of Linux OS.



This product is installed.

Ubuntu on Windows allows you to use Ubuntu Terminal and run Ubuntu command line utilities including bash, ssh, git, apt and many more.

Figure 2: Ubuntu app on Windows.

to its source code and redistribute it.

I.2. Installation!

Linux:

1. Check shell type.
2. You can start working if it is bash.
3. Otherwise, install bash using `sudo apt install bash`.

Mac:

1. mac comes with zsh shell type.
2. Install bash via Homebrew.

Windows:

1. Access Windows Microsoft Store.
2. Search for Ubuntu App.
3. Click on Install and we end up getting a SHELL Terminal.

II. Shell

Shell is a linux command line interpreter which provides an interface between the user and the kernel. Shell allows user to execute commands. It translates the command and send it to the system. There are different types of shells such as Bourne (Sh), Bourne again (Bash), Korn (Ksh), C-style (Csh, Tcsh) shell, etc. A shell can differ from the other shell in terms of syntax. We are going to talk about bash shell throughout the course. To check which shell our current system has, do the following:

```
echo $SHELL
```

Here, `SHELL` is a variable. To see the path to the parent directory of the system:

```
echo $HOME
```

Note: Shell scripting is case sensitive.

II.1. Bash Scripting

Multiple commands can be executed to perfom a specific task using semi-colon (;) in between the commands.

II.2. Basic Usage of Shell

2. Basic Commands

1. `echo` - It prints the text to the terminal.

   ```
   echo "Hello Linux!"
   ```

2. `date` - It displays the current date and time. Options: `-u` (to print the date and time in UTC), `--date="string"` (output based on the given string).

   ```
   date
   date -u
   date --date="tomorrow"
   ```

3. `man` - Displays the user manual of any command we pass as an argument. For example, `man echo` will display the complete manual of command `echo`.

2. Navigate in the Shell

1. `pwd` - It prints path of the current directory.
2. `cd` - It is used to move from one directory to another.

   ```
   cd Documents      #move to the Documents.
   cd ..             #move to the parent of the current directory.
   cd                #move to the home/ parent directory of the system.
   ```

3. `ls` - List all the files in the current directory. Options: `-l` (to list all the files with details such as permissions, size, etc.)
4. `which` - It locates the command and prints its path.

   ```
   which python
   which ls
   which date
   ```

II.3. Command type

`type` command displays the type of the command which is provided during the execution of type command. For example, we can see the type of the commonly used `ls` and `echo` command.

```
type cp        #prints "cp is /bin/cp"
type echo      #prints "echo is a shell builtin"
type while      #prints "while is a shell keyword"
type -t cp        #prints "file"
```

There are five types of commands:

- Alias
- Function
- Shell built-in
- Keyword
- File

Aliases are shortcuts which exist in the memory. We can make use of aliases in case of long commands. Aliases (with options) can also be created and saved in `~/.bashrc` file in the following way:

```
alias mylinux='ls -la'
```

II.4. Files and Directories: Connecting Input-Output

We will see commands to manage files and directories in a linux based system. We will demonstrate how to create and manipulate files and also directories which follow a tree-like structure in the system.

1. `mkdir` - Creates a new directory.

```
mkdir dirname
```

2. `rmdir` - Removes the directory.

   ```
   rmdir dirname
   ```

3. `touch` - Creates simple empty file(s).

   ```
   touch file1
   touch file1 /home/files/file2 file3
   ```

4. `rm` - It is used to delete a file.

   ```
   rm filename
   ```

5. `cat` - This command is used to display the file contents and can copy contents of one file to another file. We can also create a file and enter the contents with `cat` command. Following are some of the examples:

   ```
   cat > demo.txt      #enter contents in a file (ctrl+d to save)
   cat demo.txt        #displays file contents
   cat demo1.txt>demo2.txt #copy contents of demo1 to demo2
   ```

   If we want to append the content at the end of the file, we can do it in the following way:

   ```
    echo "new text at the end">>demo.txt
   ```

6. `sort` - This sorts the lines of a file. Option `-r` will sort the file in a reverse order. If a file `sample.txt` looks like this:

   ```
   abcd
   aaaa
   bcd
   aab
   11
   213
   132
   ```

Command `sort sample.txt` will output:

```
11
132
213
aaaa
aab
abcd
bcd
```

7. `cp` - It is used to create a copy of a file from source to destination. We can either provide a destination or a filename with destination so that a copy of a file is created with a new filename.

   ```
   cp source/filename destination/
   ```

8. `mv` - move a file from source to destination. It will remove file from the source location and create a new file in the destination directory with the same name and contents. This command can also be used to rename a file.

   ```
   mv source/filename destination/
   ```

9. `head` - This displays first ten lines of the file.

   ```
   head filename
   ```

10. `tail` - This displays the last ten lines of the file.

    ```
    tail filename
    ```

11. `tac` - Reverse of `cat` command. It displays the contents of file in reverse order.

    ```
    tac filename
    ```

12. `find` - This command looks for a file in a particular location provided with the command. Find command can also be used to search for a pattern. For example, if we want to list all the pdf files in the current directory, we can use find command.

```
find /sample_directory -name filename
find . -name "*.pdf"
```

13. `locate` - It is similar to `find` command but it takes only one argument i.e., filename and searches for the file in all possible locations. It is faster than the `find` command.

    ```
    locate filename
    ```

14. `grep` - `grep` stands for globally search for regular expression and print out. It searches for a regular expression in a file or multiple files and prints all possible outputs.

    ```
    grep "ab" sample.txt
    ```

II.5. Editor: Create and Execute Scripts

We can use editors like `vim`, `vi`, `nano` or `gedit` to create and write shell scripts. Shell script is saved with `.sh` as an extension. To create a script, write:

```
vi hello.sh
```

Write `#!/bin/bash` in the first line of the script. It tells the OS to invoke the bash shell to execute the commands present in the script. `#!` is referred to as a shebang.

```
#!/bin/bash
echo "Hello Linux!"
```

`:wq!` to save and exit the `vi` or `vim` editor. To run the script, write the following command in the terminal:

```
./hello.sh
```

Important: Before executing the script, let's understand the permissions of a file:

- Read permission: The user can read or check the file contents.
- Write permission: The user can edit the file.
- Execute permission: The user can execute the file.

We need to give execute permission to the script in order to execute it. To make script executable:

```
chmod +x hello.sh
```

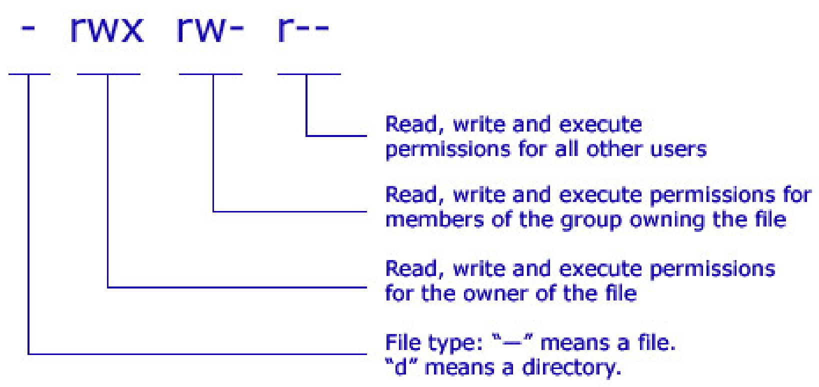

- rwx rw- r--
Read, write and execute permissions for all other users
Read, write and execute permissions for members of the group owning the file
Read, write and execute permissions for the owner of the file
File type: "—" means a file. "d" means a directory.

Figure 3: File permissions in shell.

II.6. Script with Arguments

We can run a script with arguments to be passed to the command line. The argument will be processed in the script as a variable based on its position during execution. For example:

```
#!/bin/bash
echo "Hello $1"
```

If we execute the script as `./hello.sh Ubuntu`, the output will be `Hello Ubuntu`.
Note: Make script executable once you create it using `chmod` command.

- `$0` - 0th argument i.e., name of the script itself.
- `$1` - Positional argument present at the first position.
- `$*` or `$@` - Read all the arguments.
- `$#` - Count the number of arguments passed.

II.7. Importance of Quotes

- Without arguments: Single and double quotes work same in case of text without any variables or arguments. For example, `echo "Hello Linux!"` and `echo ‘Hello Linux!’` will give the same output i.e., `Hello Linux!`.
- With arguments: Double quotes allow `$` to store the value. So, both the quotes will have different outputs in case of arguments. For example, the script is as follows:

  ```
  #!/bin/bash
  echo "Hello $1"
  echo ‘Hello $1’
  echo "Hello \$1"
  ```

Command `./hello.sh Linux!` for the above script will output:

```
Hello Linux!
Hello $1
Hello $1
```

II.8. Variables

In bash scripts, we can declare and make use of variables. A variable stores a value which can be used throughout the script once it is declared. There are two types of variables which can be used in the script:

1. **User-defined Variables:** We can declare variables using equal sign without spaces in between. The variable is considered as a command by shell in the presence of spaces in between and leads to the error. For example:

    ```
    number=40     #correct
    number = 40     #error
    number =40     #error
    number= 40     #error
    ```

    Arrays are another type of user-defined variable. An array index starts with zero. We can declare an array in the following way:

    ```
    #!/bin/bash
    cities=(Hyderabad Mumbai Delhi Bangalore)
    echo ${cities[1]}   #prints first element i.e., Mumbai
    ```

    Note: We will see "Arrays" later in detail.

2. **Environment Variables:** Environment variables are variables which are already defined by shell and are part of environment. These variables are exported to the processes by shell. All the environment variables are written in capital letters. One can list all the environment variables with their values using `printenv` command in the terminal. Here are few examples of environment variables:

    ```
    echo $BASH_VERSION    #prints bash version
    echo $HOME    #current user's parent/home directory
    echo $USER    #current user
    echo $PATH    #list of all directories that shell consults to run a command.
    ```

II.9. Scope of Variable

Usually, a variable can be used anywhere in the script once it is declared. But, can we use a variable declared in one script in the second script? The answer is no. A variable's scope is limited to the script in which it is created. However, a variable can be exported to another script using `export` command. For example,

- Here is the first script:

  ```
  #!/bin/bash
  #num1.sh
  number=40
  export number
  ./num2.sh
  ```

- Second script:

  ```
  #!/bin/bash
  #num2.sh
  echo $number
  ```

  The second script prints 40 as the output.

Note: The important point to note here is that the second script doesn't change the original variable and only makes a copy of it.

II.10. Functions

A function in a bash script is a block of code consists of a set of commands that can be called numerous times. Here is an example of a bash script with functions:

```
#!/bin/bash
func(){
        echo "This is a demo function"
}
line_count(){
        wc -l $1 | awk '{print $1}'
}
func
num=$(line_count $1)
echo "File $1 has $num lines."
```

The output will be

```
This is a demo function
File hello.txt has 3 lines.
```

II.11. Interactive Scripts: Usage of *echo* and *read*

Parameters are generally passed as arguments. We can make scripts interactive that prompt for the information in the middle of the execution of the script. `echo` and `read` can be used for interactive scripts.

```
#!/bin/bash
echo "Which country do you live in?"
read
echo "Hello $REPLY"
```