**01 Starting off with Sha-Bang!!!!**

### What is Linux Shell?

Linux shell is a user interface present between user and kernel.

It is used for executing commands and communication with Linux OS. Linux shell is basically a program used by users for executing commands. It accepts human-readable commands as input and converts them into kernel understandable language.

The shell gets started when you log in or open a console (terminal).

The shell is not part of system kernel, but uses the system kernel to execute programs, create files etc.

Diagram

Description automatically generated

The shell is a command line interpreter and is the interface between the user and the kernel. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.

Linux machine boots up, it executes the shell scripts in /etc/rc.d to restore the system configuration and set up services

**Shell characteristics and details**

* It is collection of Shell commands and programming constructs, written in a single file
* Shell executes the script by spawning another (child) shell:
  + that is when we run one shell program, it runs in a child shell area
* Scripts are interpreted; not compiled:
  + interpreted: each line will be processed by the system during run time. If any syntax error exists, that line will be skipped with error
  + compiled: in compiled oriented language, code will first be checked for syntax correctness. if any syntax error then compiler will catch that during the pre-processing time instead of checking during run time

### When not to use shell scripts

List of commands

* Resource-intensive tasks, especially where speed is a factor (sorting, hashing, recursion [2] ...)
* Procedures involving heavy-duty math operations, especially floating point arithmetic, arbitrary
* precision calculations, or complex numbers (use C++ or FORTRAN instead)
* Cross-platform portability required (use C or Java instead)
* Complex applications, where structured programming is a necessity (type-checking of variables, Function prototypes, etc.

### Types of Shell?

* Bourne again shell [BASH]
* Bourne shell [SH]
* Korn shell [KSH]
* C shell [CSH]

### Shell Prompt

There are various ways to get shell access:

* Terminal - Linux desktop provide a GUI based login system. Once logged in you can gain access to a shell by running X Terminal (XTerm), Gnome Terminal (GTerm), or KDE Terminal (KTerm) application.
* Connect via secure shell (SSH) - You will get a shell prompt as soon as you log in into remote server or workstation.
* Use the console - A few Linux system also provides a text-based login system. Generally you get a shell prompt as soon as you log in to the system.

### BASH Shell

A *shell program* is typically an executable **binary** that takes commands that you type and (once you hit return), translates those commands into (ultimately) system calls to the Operating System API.  
  
**Each shell script consists of**

* Shell keywords such as if..else, do..while.
* Shell commands such as pwd, test, echo, continue, type.
* Linux binary commands such as w, who, free etc..
* Text processing utilities such as grep, awk, cut.
* Functions - add frequent actions together via functions. For example, /etc/init.d/functions file contains functions to be used by most or all system shell scripts in the /etc/init.d directory.
* Control flow statments such as if..then..else or shell loops to preform repeated actions.

It is the shell that expands wildcard characters, such as \* or ?, thus saving you laborious typing

**Shell Commands**

The bash shell comes with two types of commands:

* Internal commands (builtins) - part of the shell itself, i.e. built into the shell.
* External commands - separate binaries stored in /sbin, /usr/sbin, /usr/bin, /bin, or /usr/local/bin directories.

**Bash and Command Types**

The bash shell understands the following types of commands:

* Aliases such as ll
* Keywords such as if
* Functions (user defined functions such as genpasswd)
* Built in such as pwd
* Files such as /bin/date

### Type Command

The type command can be used find out a command type.

The type command can be used to find out if a command is built in or an external binary file.

$ type -a ls # ls is /bin/ls

$ type -a history # history is a shell built in

$ type -a echo # echo is a shell built in && echo is /bin/echo

### Bash Login

User can login locally into the console when in runlevel # 3 or graphically when in runlevel # 5 (the level numbers may differ depending on the distribution). In both cases you need to provide username and password.

Bash uses the following initialization and start-up files:

1. /etc/profile - The systemwide initialization file, executed for login shells.

2. /etc/bash.bashrc - The systemwide per-interactive-shell startup file. This is a non-standard file which may not exist on your distribution. Even if it exists, it will not be sourced unless it is done explicitly in another start-up file.

3. /etc/bash.logout - The systemwide login shell cleanup file, executed when a login shell exits.

4. $HOME/.bash\_profile - The personal initialization file, executed for login shells.

5. $HOME/.bashrc - The individual per-interactive-shell startup file.

6. $HOME/.bash\_logout - The individual login shell cleanup file

7. $HOME/.inputrc - Individual readline initialization file.

**Login Shell**

Login shells are first shell started when you log in to the system. Login shells set environment which is exported to non-login shells. Login shell calls the following when a user logs in:

* Login shell executes /etc/profile
* /etc/profile executes all scripts in /etc/profile.d/
* Then executes users ~/.bash\_profile
* ~/.bash\_profile executes users ~/.bashrc
* ~/.bashrc executes /etc/bashrc

**Non Login Shell**

Non Login Shell is the shell, which is started by the login shell. For example, A shell which you started from another shell or started by a program etc.

A non login shell executes the following script to set the shell environment.

* Non login shell first executes ~/.bashrc
* Then ~/.bashrc executes /etc/bashrc
* /etc/bashrc calls the scripts in /etc/profile.d

### which command

You can also use the which command to display the full path of (shell) commands:

$ which commandname

$ which bash # /bin/bash

### Shebang!!

#!/usr/bin/sh OR #!/bin/sh Declares a Bourne shell

#!/usr/bin/ksh OR #!/bin/ksh Declares a Korn shell

#!/usr/bin/csh OR #!/bin/csh Declares a C shell

#!/usr/bin/bash OR #!/bin/bash Declares a Bourne-Again shell

* #!/bin/sh
* #!/bin/bash
* #!/usr/bin/perl
* #!/usr/bin/tcl
* #! /bin/sed -f
* #!/bin/awk –f

The #! Syntax is used in scripts to indicate an interpreter for execution under Linux Operating System.

**#!**  Is a 2 byte magic number that designates a file type, in this case it denote executable shell script

**/bin/bash/** is a path name. This is the path to the program that interprets the commands in the script, whether it be a shell, a programming language, or a utility.

If you do not specify an interpreter line, the default is usually the /bin/sh. But, it is recommended that you set

#!/bin/bash line.

### Hello World!!

*$ vim helloWorld.sh*

*#!/bin/bash*

echo "Hello, World!"

echo "Knowledge is power." wq!

$ chmod 755 helloWorld.sh

### Invoking the Script

* **sh <script name>** (Not recommended is using **sh <scriptname**, since this effectively disables reading from stdin within the script)
* **bash <script name>**
* **./ <script\_name>**
* **/home/venkat/<script name>**

Create script folder & place all script there & use that path as global path variable. So you can invoke script anywhere.

* /etc/profile (for all users)
* ~/.bash\_profile (for current user)
* ~/.bash\_login (for current user)
* ~/.profile (for current user)

Sudo vim /etc/profile

export SCRIPT\_HOME="/home/ec2-user/Shell\_programing/"

export PATH=$SCRIPT\_HOME:$PATH

source /etc/profile

### Sourcing the script

Its another way to execute the script

$ source example.sh or . example.sh

* When you *execute* the script you are opening a *new* shell, type the commands in the new shell, copy the output back to your current shell, then close the new shell. Any changes to environment will take effect only in the new shell and will be lost once the new shell is closed.
* When you *source* the script you are typing the commands in your *current* shell. Any changes to the environment will take effect and stay in your current shell.
* Sourcing is common way to import variable assignments or functions.

**Bash Startup**

.bash\_profile is read when bash is invoked as a login shell

.bashrc is executed everytime when a new shell is started.

Settings Env variable are typically done in .bash\_profile not in .bashrc as it will grow with each nested shell invocation.

Exported varible are copied into new process so set it in .bash\_profile

Certain things are not exported example aliases and functions have to available in every shell. Set it in .bashrc