

LAB CYCLE:4

EXPERIMENT NO: 6

Date:

SHELL SCRIPTING

Aim: a) Shell scripting: study bash syntax, environment variables, variables, control constructs such as if, for and while, aliases and functions, accessing command line arguments passed to shell scripts.

b) Study of startup scripts, login and logout scripts, familiarity with systemd and system 5 init scripts is expected

a) Shell scripting: study bash syntax, environment variables, variables, control constructs such as if, for and while, aliases and functions, accessing command line arguments passed to shell scripts.

Shell scripting refers to the process of writing and executing scripts using a shell, which is a command-line interface or command interpreter on Unix-like operating systems. The most commonly used shell scripting languages are Bash (Bourne Again Shell) and sh (Bourne Shell). Shell scripts can automate tasks, execute commands, and perform various operations on a Unix or Linux system.

Here are some key concepts and features of shell scripting:

1.Script file extension: Shell scripts typically have file extensions like ".sh" to indicate that they are shell scripts.

2.Shebang line: The first line of a shell script starts with a shebang (#) followed by the path to the shell interpreter. For example, #!/bin/bash specifies that the script should be interpreted using Bash.

3.Variables: Shell scripts can define and use variables. Variables are typically assigned values using the assignment operator (=).

For example:

```
name="John"
```

```
age=25
```

4.Command execution: Shell scripts can execute commands using various command-line utilities. Commands can be executed directly or stored in variables for later use. For example:

```
ls
```

```
result=$(ls)
```

5.Control structures: Shell scripting supports control structures like conditionals (if-else statements) and loops (for loops, while loops). These structures allow you to control the flow of execution based on conditions or iterate over a set of values.

6. Input/output redirection: Shell scripts can redirect input and output streams. For example, you can redirect the output of a command to a file using the ">" operator or read input from a file using the "<" operator.

7. Functions: Shell scripts can define functions to encapsulate reusable code blocks. Functions help in modularizing the script and improving code organization.

8. Command-line arguments: Shell scripts can accept command-line arguments. These arguments can be accessed using special variables like \$1, \$2, etc., where \$1 represents the first argument, \$2 represents the second argument, and so on.

9. Environment variables: Shell scripts can access and modify environment variables, which are system-wide variables that hold information about the environment in which the script is running.

10. Error handling: Shell scripts can handle errors using conditional statements and error codes returned by commands. This allows scripts to take appropriate actions based on the success or failure of commands.

Shell scripting is a powerful tool for automation, system administration, and repetitive tasks in Unix-like operating systems. It provides a flexible and efficient way to execute commands, manipulate data, and control the behavior of the shell.

Environment Variables

Environment variables are an essential part of shell scripting as they allow you to store and retrieve information that can be used by various processes and scripts running in the shell. Here are some key points about environment variables in shell scripting:

1. Defining Environment Variables: Environment variables are typically defined using uppercase letters. To set an environment variable, you can use the export command followed by the variable name and its value. For example:

```
export MY_VAR="Hello, World!"
```

2. Accessing Environment Variables: To access the value of an environment variable, you can use the \$ symbol followed by the variable name. For example:

```
echo $MY_VAR
```

3. System-defined Environment Variables: The shell automatically sets some environment variables that contain useful information. Examples include:

\$HOME: The current user's home directory.

\$PATH: A list of directories where the shell looks for executable files.

\$USER: The username of the current user.

\$PWD: The present working directory.

4. Shell Script-defined Environment Variables: You can define your own environment variables within shell scripts. These variables are local to the script by default, but you can export them to make them available to other processes. For example:

```
#Define a local variable
```

```
MY_LOCAL_VAR="Local variable"
```

```
#Export the variable to make it available to other processes
```

```
export MY_LOCAL_VAR
```

5. Using Environment Variables in Scripts: Environment variables can be used within shell scripts just like any other variables. Here's an example:

```
#!/bin/bash
```

```
echo "The value of MY_VAR is: $MY_VAR"
```

6. Unsetting Environment Variables: To remove an environment variable, you can use the `unset` command followed by the variable name. For example:

```
unset MY_VAR
```

7. Persistent Environment Variables: Environment variables defined in a shell session are not persistent and will be lost when the session ends. To make them persistent, you can define them in shell startup files such as `.bashrc` or `.bash_profile` (for Bash) or in the system-wide `/etc/environment` file.

Variables

Variables in shell scripting are used to store and manipulate data within a script. Here are some important points about variables in shell scripting:

1. Variable Declaration: In shell scripting, variables are typically declared without specifying a data type. You can assign a value to a variable using the syntax `variable_name=value`. For example:

```
name="John"
```

```
age=25
```

2. Variable Naming: Variable names in shell scripting can contain letters (a-z, A-Z), digits (0-9), and underscores (`_`). They must start with a letter or an underscore. Variable names are case-sensitive, so `my_var` and `MY_VAR` are considered different variables.

3. Accessing Variable Values: To access the value of a variable, you can use the `$` symbol followed by the variable name. For example:

```
echo $name
```

4. Variable Scope: By default, variables in shell scripts have a global scope and can be accessed from anywhere within the script. If you want to limit the scope of a variable to a specific block of code, you can use functions.

5. Special Variables: Shell scripting provides some predefined variables that have special meanings. Examples include:

`$0`: The name of the script itself.

`$1`, `$2`, etc.: Command-line arguments passed to the script.

`##`: The number of command-line arguments.

`?`: The exit status of the last command.

`$$`: The process ID of the current script.

6. Variable Manipulation: You can perform various operations on variables, such as concatenation, substring extraction, and arithmetic calculations. Some commonly used operators and syntax include:

Concatenation: `result=$var1$var2`

Substring extraction: `substring=${string:position:length}`

Arithmetic calculations: `result=$((num1 + num2))`

7. Quoting Variables: When accessing variable values, it's a good practice to enclose them in double quotes (") to handle cases where the value contains spaces or special characters. For example:

```
echo "$name is $age years old."
```

8. Variable Assignment from Command Output: You can assign the output of a command to a variable using command substitution. The syntax is `variable=$(command)`. For example:

```
date=$(date +%Y-%m-%d)
```

These are some fundamental aspects of variables in shell scripting. They allow you to store and manipulate data, perform calculations, and pass information to and from your script.

Control Structures

In shell scripting, control structures are used to control the flow of execution based on certain conditions or to repeat a set of instructions. The commonly used control structures in shell scripting include:

1. if-else: The if-else statement allows you to perform different actions based on a condition. It has the following syntax:

```
if condition
```

```
then
```

```
    #statements to execute if condition is true
```

```
else
```

```
    #statements to execute if condition is false
```

```
fi
```

2. for loop: The for loop allows you to iterate over a list of values or elements. It has the following syntax:

```
for variable in list
```

```
do
```

```
    #statements to execute for each iteration
```

```
done
```

The variable takes the value of each item in the list on each iteration.

3. while loop: The while loop allows you to repeat a set of statements as long as a condition is true. It has the following syntax:

```
while condition
do
    #statements to execute while condition is true
done
```

The loop continues until the condition evaluates to false.

4. until loop: The until loop is similar to the while loop, but it continues until a condition becomes true. It has the following syntax:

```
until condition
do
    #statements to execute until condition becomes true
done
```

The loop continues until the condition evaluates to true.

5. case statement: The case statement allows you to perform different actions based on the value of a variable. It has the following syntax:

```
case variable in
    pattern1)
        #statements to execute for pattern1
        ;;
    pattern2)
        #statements to execute for pattern2
        ;;
    pattern3)
        #statements to execute for pattern3
        ;;
    *)
        #statements to execute for all other patterns
        ;;
esac
```

The value of variable is compared against each pattern, and the corresponding statements are executed for the matching pattern.

These control structures provide you with the flexibility to conditionally execute code, iterate over a set of values, and make decisions based on different conditions in shell scripting.

Aliasing

In Linux, an alias is a shortcut that references a command. An alias replaces a string that invokes a command in the Linux shell with another user-defined string. Aliases are mostly used to replace long commands, improving efficiency and avoiding potential spelling errors.

The alias command provides a string value that replaces a command name when it is encountered. The alias command lets you create shortcuts for long commands, making them easier to remember and use. It will have the same functionality as if the whole command is run.

How to Create Your Own Linux Commands?

Using the alias command, you'll be able to create your own commands. It's so simple to create your own command.

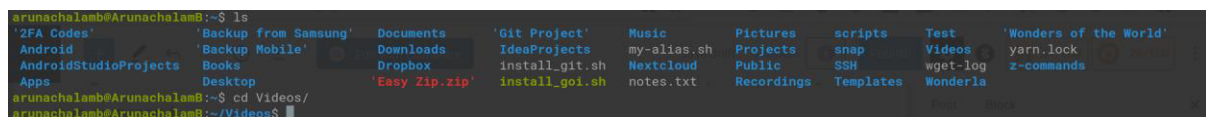
Here's the syntax for the alias command:

```
alias [alias-name[=string]...]
```

Let's look at an example of creating your own command.

Let's assume you want to create a command called `cdv`, and entering the command in the terminal should take you to the Videos directory.

Usually, to navigate to a directory, we use `cd` command. To navigate to Videos we need to use `cd Videos` as shown in the below screenshot:

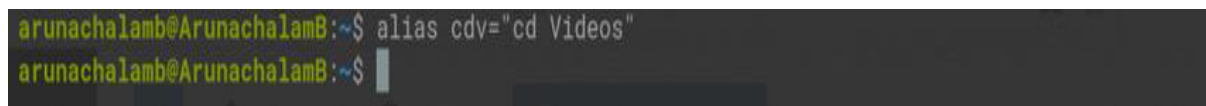


```
arunachalamb@ArunachalamB:~$ ls
'2FA Codes'      'Backup from Samsung'  Documents  'Git Project'  Music      Pictures  scripts  Test      'Wonders of the World'
Android          'Backup Mobile'        Downloads  IdeaProjects  my-alias.sh  Projects  snap     Videos  yarn.lock
AndroidStudioProjects  Books                  Dropbox    install_git.sh  Nextcloud  Public   SSH      wget-log  z-commands
Apps            Desktop                'Easy Zip.zip'  install_goi.sh  notes.txt  Recordings  Templates  Wonderla

arunachalamb@ArunachalamB:~$ cd Videos/
arunachalamb@ArunachalamB:~/Videos$
```

Let's create our command called `cdv` to navigate to the Videos directory. To achieve that, you have to enter the following command in your terminal:

```
alias cdv="cd Videos"
```



```
arunachalamb@ArunachalamB:~$ alias cdv="cd Videos"
arunachalamb@ArunachalamB:~$
```

We have created our command. From the above screenshot, you can see that it does not return anything.

Run the `cdv` command on your terminal to see what happens:



```
arunachalamb@ArunachalamB:~$ cdv
arunachalamb@ArunachalamB:~/Videos$
```

How to View Created Alias Commands

You can view all your alias commands by appending the `-p` flag to the `alias` command like this:

alias -p

```
arunachalamb@Arunachalamb:~$ alias -p
alias a='xdotool key ctrl+shift+t'
alias ad='~/Android/Sdk/emulator/emulator -list-avds'
alias adbr='adb reverse tcp:8081 tcp:8081'
alias alert='notify-send --urgency=low -i "[ $? = 0 ] && echo terminal || echo error"
'\'''"
alias b='cd ..'
alias c='code ./'
alias cdb='cd -'
alias cdv='cd Videos'
```

How to Remove an Alias Command in Linux

Pass your alias name to the `unalias` command as an argument to remove the alias command.

unalias alias_name

```
arunachalamb@Arunachalamb:~$ unalias cdv
arunachalamb@Arunachalamb:~$ alias -p
alias a='xdotool key ctrl+shift+t'
alias ad='~/Android/Sdk/emulator/emulator -list-avds'
alias adbr='adb reverse tcp:8081 tcp:8081'
alias alert='notify-send --urgency=low -i "[ $? = 0 ] && echo terminal || echo error"
'\'''"
alias b='cd ..'
alias c='code ./'
alias cdb='cd -'
alias clear_goi='rm -rf ~/.nvm/versions/node/v14.17.6/bin/goi; rm -rf ~/.nvm/versions/nod
```

How to Remove All Alias Commands in Linux

We have a command to achieve that:

unalias -a

```
arunachalamb@Arunachalamb:~$ alias -p
alias cdd='cd Downloads'
alias cddu='cd Documents'
alias cdv='cd Videos'
arunachalamb@Arunachalamb:~$ unalias -a
arunachalamb@Arunachalamb:~$ alias -p
arunachalamb@Arunachalamb:~$
```

If you create an alias command, it'll be active only for the particular instance of the terminal. It'll not be created permanently, so you won't be able to access it in two different terminal windows unless you run the alias command on both terminals.

b) Study of startup scripts, login and logout scripts, familiarity with systemd and system 5 init scripts is expected

Startup scripts, login scripts, and logout scripts are important components of the initialization process in a Unix-like operating system. They allow administrators to automate tasks and configure system behavior during system startup, user login, and user logout. Familiarity with systemd and System V init scripts is crucial for managing these processes efficiently. Let's explore each of these topics in more detail:

1. Startup Scripts:

Startup scripts are executed during the boot process to initialize the system. In modern Linux distributions that use systemd as the init system, startup scripts are typically managed through systemd unit files. These unit files define services, targets, and other units that control the behavior of the system. The main directory for systemd unit files is `/etc/systemd/system/`.

2. Login Scripts:

Login scripts are executed when a user logs into a system. These scripts provide a way to set environment variables, configure user-specific settings, and perform additional tasks upon login. The specific login script executed depends on the user's shell and the system configuration. For example, in the Bash shell, the login script is usually `~/.bash_profile` or `~/.bash_login` for login shells, and `~/.bashrc` for non-login shells.

3. Logout Scripts:

Logout scripts are executed when a user logs out of a system. These scripts allow for cleanup tasks, such as removing temporary files or logging session statistics. The specific logout script executed depends on the user's shell and the system configuration. In Bash, the logout script is typically `~/.bash_logout`.

4. Systemd:

Systemd is a modern init system and service manager that has replaced the traditional System V init system in many Linux distributions. Systemd provides enhanced features for managing services, including parallel startup, dependency-based service control, and centralized logging. Systemd unit files, such as service unit files (`.service`), target unit files (`.target`), and timer unit files (`.timer`), are used to define and control services.

5. System V Init Scripts:

System V init scripts were the traditional initialization mechanism used in Unix-like systems before the introduction of systemd. Init scripts are stored in `/etc/init.d/` directory and are typically written in shell scripting languages (e.g., Bash). These scripts are responsible for

starting, stopping, and managing system services and can be run manually or automatically during the system startup.

Familiarity with both `systemd` and System V init scripts is important for understanding and managing the initialization process in different Linux distributions. The choice of which init system to use depends on the specific distribution and its configuration. It's beneficial to have knowledge of both to be able to work with various systems effectively.