

EXPERIMENT NO:6

DATE:

SHELL SCRIPT

Shell scripting is a way to automate tasks using a command-line shell, such as the Bash shell on Linux or macOS. It involves writing scripts or programs that contain a series of commands and instructions, which are executed in sequence by the shell. Shell scripts can be used to perform various tasks, ranging from simple file manipulation and system administration tasks to complex data processing and automation. They are particularly useful for repetitive tasks or tasks that require a sequence of commands to be executed. Here are some key aspects of shell scripting:

Shell Selection: Different operating systems use different shells, such as Bash, C shell (csh), Korn shell (ksh), and others. The choice of shell depends on the operating system and personal preference. Bash is one of the most commonly used shells and is available on many platforms.

Script File: Shell scripts are typically stored in plain text files with a .sh extension. Before executing a shell script, it needs to have execute permissions set using the chmod command.

Shebang: 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 executed using the Bash shell.

Comments: Shell scripts can include comments to provide explanations or documentation.

Comments begin with the # character and are ignored by the shell.

Variables: Shell scripts use variables to store and manipulate data. Variables are assigned values using the = operator, and their values can be accessed using the variable name preceded by the \$ symbol. For example, name="John" assigns the value "John" to the variable name.

Command Execution: Shell scripts can execute various commands, such as running programs, manipulating files and directories, performing arithmetic operations, and more. Commands are executed by simply writing them in the script, one command per line.

Control Structures: Shell scripts support control structures like conditionals (if-else), loops (for, while), and case statements. These structures allow you to control the flow of execution based on certain conditions.

Input and Output: Shell scripts can read input from the user or from files, and they can produce output to the console or redirect it to files.

Functions: Shell scripts can define functions to encapsulate a series of commands. Functions can be called multiple times within a script or from other scripts.

Error Handling: Shell scripts can handle errors by checking the return codes of commands and taking appropriate actions based on the results. The special variables `?` and `$?` are commonly used for error checking.

Shell scripting is a vast topic, and there are many resources available online to learn more about it, including tutorials, guides, and examples.

- A UNIX shell program interprets user commands which are either directly entered by the user or which can be read from a file called a shell program.
- `cat /etc/shells =>` shells supported by a system
- `/bin/sh`
- `/bin/dash`
- `/bin/bash`
- `/bin/rbash`

Bash Syntax:

BASH = Bourne Again SHell

Bash is a shell written as a free replacement to the standard Bourne Shell (`/bin/sh`) originally written by Steve Bourne for UNIX systems. It has all of the features of the original Bourne Shell, plus additions that make it easier to program with and use from the command line.

Since it is Free Software, it has been adopted as the default shell on most Linux systems.

File Structure

- `#!/bin/sh`

- All the scripts would have the .sh extension.
- Before you add anything else to your script, you need to alert the system that a shell script is being started. This is done using the shebang construct.
- *# symbol is called a hash, and the ! symbol is called a bang.*
- A Shell provides you with an interface to the Unix system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output. Execute Script
- To execute a program available in the current directory, use ./program_name

```
#!/bin/bash

# Author : Zara Ali
# Copyright (c) Tutorialspoint.com
# Script follows here:
pwd
ls
```

Save the above content and make the script executable –

```
$chmod +x test.sh
```

When file is created using touch it has only read & write permission, no execution permission

- The following script uses the read command which takes the input from the keyboard and assigns it as the value of the variable PERSON and finally prints it on STDOUT.

```
#!/bin/sh

# Author : Zara Ali
# Copyright (c) Tutorialspoint.com
# Script follows here:

echo "What is your name?"
read PERSON
echo "Hello, $PERSON"
```

```
$. /test.sh
What is your name?
Zara Ali
Hello, Zara Ali
$
```

- # comments

Variables

- Container stores data
- Character string which assigns a value
- Value -> number, text, filename, device, or other type of data.
- The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (_).

a. Defining Variables

- Variables are defined as follows –

variable_name=variable_value

Eg: NAME="Zara Ali" b

b .Accessing Values

- To access the value stored in a variable, prefix its name with the dollar sign (\$)

```
#!/bin/sh
```

```
NAME="Zara Ali" echo
```

```
$NAME
```

The above script will produce the following value –

```
Zara Ali
```

c. Special Variables

- These variables are reserved for specific functions. For example, the \$ character represents the process ID number, or PID, of the current shell –

```
$echo $$
```

- The above command writes the PID of the current shell –

Environment Variables

Environment variables are global variables that are available to all processes running in a Bash session. They hold information about the environment and configuration. Some commonly used environment variables are:

PATH: Specifies the directories to search for executable programs.

HOME: Represents the home directory of the current user.

USER: Stores the username of the current user.

PWD: Holds the current working directory.

SHELL: Stores the path of the current shell executable.

You can access the value of an environment variable using the \$ symbol followed by the variable name, e.g., \$PATH.

Control Constructors

If Statement: The if statement allows conditional execution of commands based on a specific condition. The basic syntax is as follows:

```
if condition then
```

```
# commands to execute if the condition is true else
```

```
# commands to execute if the condition is false
```

```
fi
```

For Loop: The for loop is used to iterate over a list of values and perform a set of commands repeatedly. The basic syntax is as follows:

```
for variable in value1 value2 value3 do # commands to execute
```

```
Done
```

While Loop: The while loop allows you to execute a block of commands repeatedly as long as a condition is true. The basic syntax is as follows:

```
while condition do
```

```
# commands to execute
```

```
Done
```

Aliases: Aliases are a way to create shortcuts or alternate names for commands or command sequences. They can be defined in a shell script or a configuration file like .bashrc. Here's an example

```
#!/bin/bash

# Alias example
alias ll='ls -l'
alias c='clear'

ll # Executes 'ls -l'
c  # Executes 'clear'
```

Functions

Functions allow to group a set of commands together and reuse them within a script. The basic syntax to define a function is as follows:

```
function_name() {
    # commands to execute
}

# Invoking the function
function_name
```

```
#!/bin/bash

# Function example
greet() {
    echo "Hello, $1!"
}

greet "John" # Outputs "Hello, John!"
```

Accessing command line arguments passed to shell scripts.

To access command-line arguments passed to a shell script, you can use special variables called positional parameters. These variables are automatically set by the shell and provide access to the arguments passed to the script.

Here's how you can work with command-line arguments in a shell script:

```
#!/bin/bash

# Accessing command-line arguments
echo "The script name is: $0"
echo "The first argument is: $1"
echo "The second argument is: $2"
echo "All arguments are: $@"
echo "The number of arguments is: $#"
```

In a shell script, the command-line arguments are represented as follows:

\$0 represents the name of the script itself.

\$1, \$2, \$3, and so on, represent the first, second, third, and subsequent arguments, respectively.

Here's an example to illustrate how to access command-line arguments in a shell script:

```
$ ./example.sh argument1 argument2
```

The output will be:

```
Script name: ./example.sh
First argument: argument1
Second argument: argument2
All arguments: argument1 argument2
Number of arguments: 2
```

Study of startup scripts, login and logout scripts.

Startup Scripts: Startup scripts are executed when a system boots up or when a user logs in. They initialize the system, configure services, and perform various tasks required for the system to operate correctly. In Linux, startup scripts are typically located in the /etc/init.d/ directory or defined as unit files in systemd. For Linux startup scripts, you can use bash or non-bash file. To use a non-bash file, designate the interpreter by adding a #! to the top of the file.

- For example, to use a Python 3 startup script, add #! /usr/bin/python3 to the top of the file.
- Specify a startup script by using one of these procedures :

1. Copies the startup script to the VM
2. Sets run permissions on the startup script
3. Runs the startup script as the root user when the VM boot

Login and Logout Scripts: Login scripts and logout scripts are executed when a user logs into or logs out of a system. These scripts can be used to set environment variables, define aliases, execute specific commands, or perform other custom actions during login or logout.

Login Script: In Linux, login scripts are often referred to as "shell initialization scripts" and include files such as .bashrc or .profile in the user's home directory. Login scripts are often stored in the user's home directory with filenames like

✦ “.bash_profile or bash_login “ for the bash shell.

✦ They can be created or modified using a text editor such as ‘vi’ or ‘nano’.

✦ Login scripts are executed only once during the login process.

```
bash
#!/bin/bash

# Commands to set up the environment
command1
command2
...

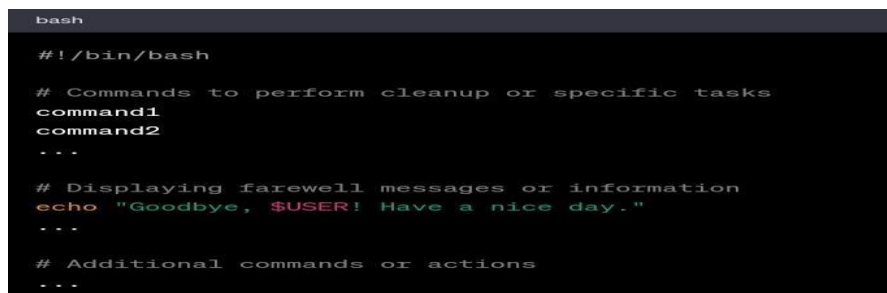
# Displaying messages or information
echo "Welcome, $USER!"
echo "Today's date is $(date)"
...

# Additional commands or actions
...
```


- † The shebang `#!/bin/bash` specifies that the script should be executed using the Bash shell.
- † The script starts with the `#!/bin/bash` shebang line followed by any necessary commands to set up the environment, such as exporting environment variables or modifying shell settings.
- † Commands to display messages or information to the user can be included using the `'echo'` command.
- † Additional commands or actions specific to the login process can be added as needed.

Logout Script: Like login scripts, they can be created or modified using a text editor such as `'vi'` or `'nano'`.

- † Logout scripts are executed only once during the logout process.
- † Examples of actions performed by logout scripts include displaying a farewell message, removing temporary files, terminating background processes, or saving session-specific data.
- † Logout scripts provide an opportunity to perform any necessary cleanup actions before the user's session ends.
- † They can enhance security by automatically logging out of connected services or ensuring that sensitive data is properly handled.

A screenshot of a terminal window showing a Bash script. The script starts with a shebang line `#!/bin/bash`. It then contains a comment `# Commands to perform cleanup or specific tasks` followed by `command1`, `command2`, and an ellipsis. Next is a comment `# Displaying farewell messages or information` followed by `echo "Goodbye, $USER! Have a nice day."` and an ellipsis. Finally, there is a comment `# Additional commands or actions` followed by an ellipsis.

```
bash
#!/bin/bash

# Commands to perform cleanup or specific tasks
command1
command2
...

# Displaying farewell messages or information
echo "Goodbye, $USER! Have a nice day."
...

# Additional commands or actions
...
```

- † The shebang `'#!/bin/bash'` specifies that the script should be executed using the Bash shell.
- † The script starts with the `'#!/bin/bash'` shebang line followed by any necessary commands to perform cleanup or specific tasks before the user's session ends.

‡ Commands to display farewell messages or information to the user can be included using the 'echo' command.

‡ Additional commands or actions specific to the logout process can be added as needed.

- **echo:** Displays farewell messages or information.

Example: `echo "Goodbye, $USER! Have a nice day."`

- **rm:** Removes files or directories.

Example: `rm -rf /tmp/temporary_files`

- **command:** Executes a command or script.

Example: `command_to_execute`

- **trap:** Sets up a signal handler to run specific commands on logout .

- Example: `trap "cleanup_function" EXIT`

Systemd: Systemd is a modern init system and service manager used in many Linux distributions. It replaces the traditional System V init system and provides advanced features for managing services and controlling the boot process. Systemd organizes services into units, which can be started, stopped, and managed individually. Unit files, typically stored in the `/etc/systemd/system/` directory, define the behavior and dependencies of these units.

Systemd is the parent of all processes :Systemd is the parent of all processes on the system, it is executed by the kernel and is responsible for starting all other processes; it is the parent of all processes whose natural parents have died and it is responsible for reaping those when they die.

Service Management: Systemd manages system services, including starting, stopping, restarting, and enabling or disabling them. Each service is represented by a unit file, typically stored in the directory, with the extension.

Dependency Management: Systemd allows specifying dependencies between services, ensuring that necessary services are started before dependent services. This simplifies the management of complex service relationships.

System Initialization: Systemd is responsible for the boot process and system initialization. It starts essential system services and processes in parallel to speed up the boot time.

Journaling: Systemd includes a logging system called the journal. It captures and stores system logs, including kernel logs and messages from services. Logs can be accessed and queried using the command.

Target Units: Systemd introduces the concept of target units, which represent system states or runlevels. Targets group multiple services together, and switching between targets allows transitioning the system into different states (e.g., multi-user, graphical).

Control and Monitoring: Systemd provides various commands to control and monitor system services, such as starting or stopping services (`systemctl start`), checking service status (`systemctl status`), enabling or disabling services at boot (`systemctl enable`), etc.

Systemd Units: Besides service units, systemd supports other types of units, such as socket units for managing network sockets, device units for device management, mount units for managing file systems, and timer units for defining periodic or scheduled tasks.

- `systemctl start` : Starts a specified unit, such as a service, socket, or target.
- `systemctl stop` : Stops a specified unit.
- `systemctl restart` : Restarts a specified unit.
- `systemctl enable` : Enables a unit to start automatically at boot.
- `systemctl disable` : Disables a unit from starting automatically at boot.
- `systemctl status` : Displays the status and information of a unit.
- `systemctl list-units`: Lists all active units and their statuses.
- `systemctl list-unit-files`: Lists all available unit files and their state

Configurations:

- **Unit Files:** Unit files define the behavior of a systemd unit. They are typically stored in `/etc/systemd/system/` or `/usr/lib/systemd/system/`. Unit files have various

sections, such as [Unit], [Service], [Socket], [Timer], etc., where you can define properties and dependencies.

- **Targets:** Targets represent system states or runlevels. They are defined in target unit files (ending with .target) and specify which units should be active in that target state. For example, multi-user.target represents the multi-user text-based mode, while graphical.target represents the graphical mode.
- **Dependencies:** Dependencies between units are defined using directives such as Requires, Wants, Before, After, etc., within unit files. They ensure that necessary services or resources are started or stopped in the correct order.
- **Environment Variables:** You can define environment variables specific to a unit by using the Environment directive in the [Service] section of a unit file.
- **Resource Limits:** Systemd allows setting resource limits for units, such as CPU, memory, and I/O limits, using directives like CPUShares, MemoryLimit, IOWeight, etc., in the [Service] section of a unit file.
- **Logging:** Systemd captures and manages system logs using the journal. You can configure logging behavior, such as log storage size, rotation policies, and filtering, in the /etc/systemd/journald.conf file.

Init Scripts: In the context of Linux, init scripts refer to the scripts used by the init system (e.g., System V init or systemd) to control the boot process and manage services. These scripts are responsible for starting, stopping, restarting, and managing various system services and daemons. Init scripts are typically located in the /etc/init.d/ directory and follow a specific naming convention, such as service_name start, service_name stop, etc.

Familiarity with system and system 5 init scripts

Familiarity with 5 Init Scripts: While the specific init scripts used can vary depending on the Linux distribution and configuration, here are five commonly encountered init scripts:

- a. `apache2`: Controls the Apache HTTP server.
- b. `sshd`: Manages the SSH server, responsible for secure remote access.
- c. `mysql`: Handles the MySQL database server.
- d. `nginx`: Controls the Nginx web server.
- e. `cron`: Manages scheduled tasks using the cron daemon.

System V supported 3 major interfaces:

- Shared memory interfaces, namely `shmget()`, `shmat()`, `shmdt()`, `shmctl()`, allow to create and allocate, manage, control & delete memory interfaces.
- Message queue interfaces, namely `msgget()`, `msgsnd()`, `msgrcv()`, `msgctl()`, allows to create, send, receive and control message queues.
- Semaphore interfaces, namely `semget()`, `semop()`, `semctl()`, Allows to create, perform and control semaphores.

All System V init scripts are stored in `/etc/rc.d/init.d/` or `/etc/init.d` directory. These scripts are used to control system startup and shutdown. Usually you will find scripts to start a web server or networking. For example

```
# /etc/init.d/httpd start
```

Or

```
# /etc/init.d/network restart
```

The main purpose of `init` is to start and stop essential processes on the system. Sys V does that with scripts. All scripts are run sequentially, they must terminate before the next one is run.

e.g. while waiting for one script, run all other non-dependent ones. The pros of using this implementation of init, is that it's relatively easy to solve dependencies. however performance isn't great because usually one thing is starting or stopping at a time. When using Sys V, the state of the machine is defined by runlevels which are set from 0 to 6.

0:Shutdown

1: Single User Mode

2: Multiuser mode without networking

3: Multiuser mode with networking

4: Unused

5: Multiuser mode with networking and GUI

6: Reboot

When your system starts up, it looks to see what runlevel you are in and executes scripts located inside that runlevel configuration. The scripts are located in /etc/rc.d/rc[runlevel number].d/ or /etc/init.d. All init-compatible scripts conform to a standard of recognized arguments. You can run each rc.d script with “start”, “stop”, “restart”, or “reload”.Ex: /etc/rc2.d/S19mysql start. Most init scripts are stored in /etc/init.d/, and symlinked where needed. When booting up, all scripts are run with “start” argument, etc. Scripts start with either an S or a K. If it starts with S, call “start” on it. If it starts with K, call “stop” on it. Starting runlevel could mean starting certain things and stopping others. For the most part, starting up will have mostly ”S” scripts, while shutdown will have mostly ”K” scripts. Numbers matter, e.g. S19mysql – service is started in numerical order. S18 will start first. This is more or less how dependencies are taken care of

RESULT: Familiarized with Shell Script.