System Programming - Linux
Week 3 - Working with Scripts

Dr. Obbo Aggrey

April 3, 2025

1 Introduction

These lecture notes provide an overview of shell scripting in Linux. It provides the reader with the knowledge necessary to automate tasks, manage files, and optimize workflow using the powerful command-line interface. It delves into key concepts such as shells, variables, and the running of shell scripts. By the conclusion of these notes, the reader will have the ability to create reliable work with scripts, boosting his productivity and getting ready for further application of scripting knowledge.

Lecture Outcomes

- Define shells and describe the different types of shells
- Define Variables and describe the different types of variables
- Run Shell scripts on Linux

2 What is shell

A shell serves as a command-line interpreter that offers a user interface to access services of the operating system. It takes commands from users or scripts and carries them out. Notable examples include Bash (Bourne Again SHell), sh (Bourne Shell), zsh (Z Shell), ksh (Korn Shell), and csh (C Shell)[1][2].

Linux shells are essential interfaces that connect users with the operating system's kernel as illustrated in Figure 1 below. They decode commands provided by the user, converting them into operations comprehensible by the kernel. This functionality allows users to engage with the system, handle files, run applications, and automate processes. The utilities include ls, cd, vi etc. While the different users can choose from a number of shells that can be Bourne shell, C shell or Z shells. This presentation will discuss the key concepts of Linux shells, concentrating on the frequently utilized Bash shell, its characteristics, and introductory scripting

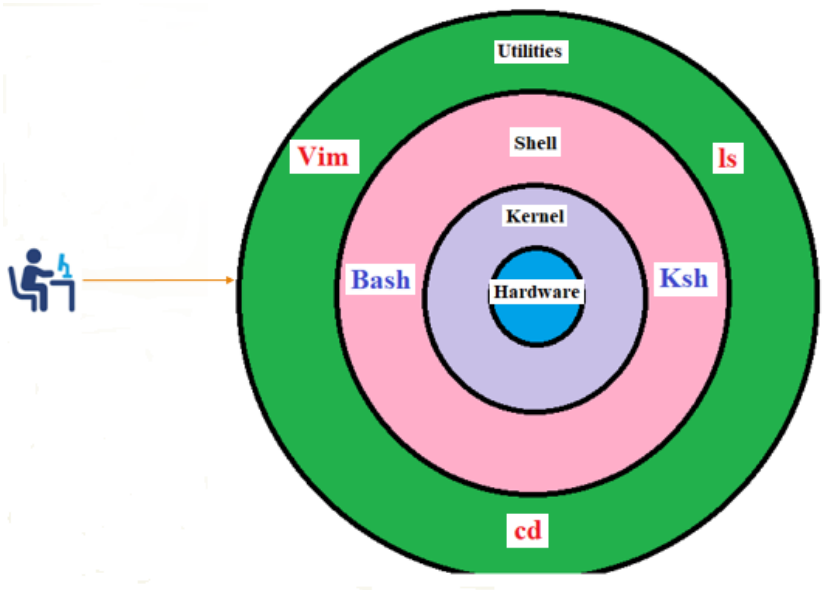

Figure 1: How the shell operates

2.1 Types of Linux Shells

Linux presents a wide array of shells, each possessing distinct features and advantages. The types range from the widely used Bash to the robust Zsh and the traditional sh. These shells influence the way users engage with the operating system. Grasping their distinctions is crucial for selecting the appropriate tool for your command-line requirements.

2.1.1 Bash (Bourne Again SHell)

This is the standard shell for many Linux distributions. It is a highly capable and flexible shell that maintains compatibility with older Bourne shell scripts. Its advantages include; It offers comprehensive scripting features, such as loops, conditionals, and functions. It also has excellent command-line editing and history functionality and is widely available and supported by a strong community. In addition, it provides good compatibility with older versions. Although it is powerful, its syntax can be more verbose than that of some modern shells.

2.1.2 Bourne Shell (sh)

This is the original Unix shell. It's a simpler shell, often used for system scripts where portability is crucial. It is extremely portable across Unix-like systems, lightweight and efficient, and good for basic system-level scripts. However, it lacks many of the advanced features found in Bash and other shells, like

command-line editing and advanced scripting constructs.

2.1.3 Z Shell(zsh)

Z Shell is a more advanced version of the Bourne shell that includes numerous enhancements compared to Bash. Its robust tab completion and spelling correction features make it a good option for customization and theming. It has improved capabilities for scripting and support for plugins. Its limitation is that its comprehensive features may result in a more challenging learning curve compared to Bash-derived shells.

2.1.4 Korn Shell (ksh)

A shell created by David Korn at Bell Labs that merges characteristics of both the Bourne and C shells. Its strength is in the fact of its enhanced scripting capabilities when compared to sh. It has support for job management and the ability to create command aliases. It is ideal for users who are familiar with traditional Unix environments. Its drawback, however, is that it is not widely used as Bash or Zsh, which may result in reduced community assistance and fewer available resources.

2.1.5 C Shell(csh))

C Shell is a shell with a C-like syntax. tcsh is an improved version of csh. It has C-like syntax, which can be familiar to C programmers, job control, and command history. The tcsh has improved command line completion. However, its limitations are scripting which is often considered less intuitive and powerful than in Bourne-derived shells. It is generally not recommended for scripting.

Essentially, Bash serves as the reliable tool for typical Linux tasks, whereas zsh provides advanced functionalities for more experienced users. Sh continues to be important for scripts at the system level, and ksh holds a specialized user base. C shells are rarely employed for scripting purposes.

3 What is a Linux Variables

In Linux, a variable can be described as a designated storage space that contains a value. This value may consist of text, a number, or different types of data.

Shell scripting in Linux empowers users to automate tasks and manipulate data through the command-line interface. Variables are fundamental building blocks in shell scripts, acting as containers for storing and retrieving information. They allow you to dynamically manage data, making your scripts more flexible and powerful. This introduction will explore the various types of variables in Linux shell scripting, providing examples and insights into their usage. Understanding these variable types is crucial for writing effective and efficient shell scripts [3][4].

3.1 Local Variables

Local variables in Linux scripts are limited to the execution context of the script. They are created and utilized internally, allowing for isolation from other scripts or the system environment. Applications for these variables include saving temporary information, controlling loop indices, and retaining specific functions values.

3.2 Environment variables

Environment variables in Linux are dynamic, named values that affect how running processes behave. Unlike local variables, they can be accessed system-wide or by child processes of the shell in which they were created. They hold important system information, such as PATH (which indicates where to look for executables), HOME (the user's home directory), and LANG (which specifies language preferences). Their applications range from system configuration and software execution to personalizing user settings. Additionally, they are utilized to supply configuration data to applications. For instance, a web server might employ an environment variable to retrieve the database connection string.

3.3 Shell Variables

The term "shell variable" is frequently used synonymously with "local variable." Nevertheless, some make a distinction by referring to shell variables as those specific to the interactive shell rather than those found within a script. These variables are created in the shell and remain in existence until the session concludes. They modify the behavior of the shell, such as PS1 (prompt), or hold temporary data for interactive use.

4 Implementing Shell Scripts

Shell scripting serves as a robust means for automating tasks and optimizing workflows in Linux. This session will offer hands-on examples and insights to help you begin.

Implementing a script involves majorly three major fundamental steps; writing the script using a text editor, making the script executable, and running the script

4.1 Writing the script

A shell script is a text file that consists of a series of commands. The initial line, known as the shebang (#!binbash), indicates which interpreter to use. Using the bash shell. below is a simple shell script

#!binbash
echo "Hello, World!"
ls

ls -l

The script has to be saved as a .sh file. And the commands are followed from top to bottom. The script is expected to print "Hello world", provide a listing of files and directories, and provide the files and directories in list format.

4.2 Making a Script Executable

Use chmod +x script_name.sh to grant execute permissions.

Example:

chmod +x hello.sh. or chmod 777 script_name.sh

4.3 Running a Script

Execute the script using ./script_name.sh.

Example:

./hello.sh.

Working with variables

#!binbash

Naming and assigning variables

my_variable="Hello, World!"

my_other_variable=123

Accessing and printing value of the variables

echo "$my_variable" echo "Value of another_variable: $another_variable"

Unsetting a variable

unset my_variable

Attempting to access an unset variable (results in empty output)

echo "$my_variable"

Reassigning a variable

my_other_variable="New Value"

echo "Reassigned another_variable: $my_other_variable"

Unsetting all variables

unset my_other_variable

echo $my_other_variable

Variables are identified using a combination of letters, numbers, and underscores, and they must begin with a letter or an underscore. Assignment is carried out using the equals sign (=), without any spaces. They can be accessed using the syntax $variable_name. The command unset variable_name will eliminate a variable, leaving its value empty. To reassign a variable, simply assign a new value to it. This example illustrates the fundamentals of declaring, assigning, accessing, unsetting, and reassigning variables. However read-only variables cannot be unset

Incorporating special variables

#!binbash

Demonstrating the use of wildcard ''
echo "Files starting with 'test':"
ls test*
# Demonstrating the use of special character '$' for variable substitution
name="Linux User"
echo "Hello, $name!"
Demonstrating the use of > for output redirection
echo "This goes to a file" > output.txt
cat output.txt

The asterisk (*) is used to identify files that start with "test" While the dollar sign ($) retrieves the value of a variable, resulting in the output "Hello, Linux User!". The greater-than symbol (>) redirects the output from the echo command into a file called "output.txt", which is subsequently displayed using cat. These illustrations demonstrate some of the essential and fundamental applications of special characters in shell scripts.

Working with Command line Arguments

Command-line arguments enable users to provide information to Linux shell scripts upon execution.

Represented by positional parameters such as 1,2, etc., they enhance the flexibility and re usability of scripts. For example, a script may accept a filename as $1 for processing. This dynamic input removes the necessity to alter the script's code for each execution. $0 holds the script's name. They empower scripts to adjust to different user requirements, making them effective tools for automation and data manipulation.

Example

#!binbash
echo "Script name: $0"
echo "Argument 1: $1"
echo "Argument 2: $2"
echo "Argument 3: $3"
echo "Number of arguments: $#"
echo "All arguments: $"

As shown in the simple script above, $0 is the script's name, $1-$3 are the first three arguments, $# is the argument count, and $* is all arguments as a single string.

List of References

1. Blum, R. (2008). Linux command line and shell scripting bible (Vol. 481). John Wiley & Sons.

2. Newham, C. (2005). Learning the bash shell: Unix shell programming. " O'Reilly Media, Inc.".

3. Albing, C., Vossen, J. P., & Newham, C. (2007). bash Cookbook: Solutions and Examples for bash Users. " O'Reilly Media, Inc.".

4. Parker, S. (2011). Shell scripting: expert recipes for Linux, Bash, and more. John Wiley & Sons.