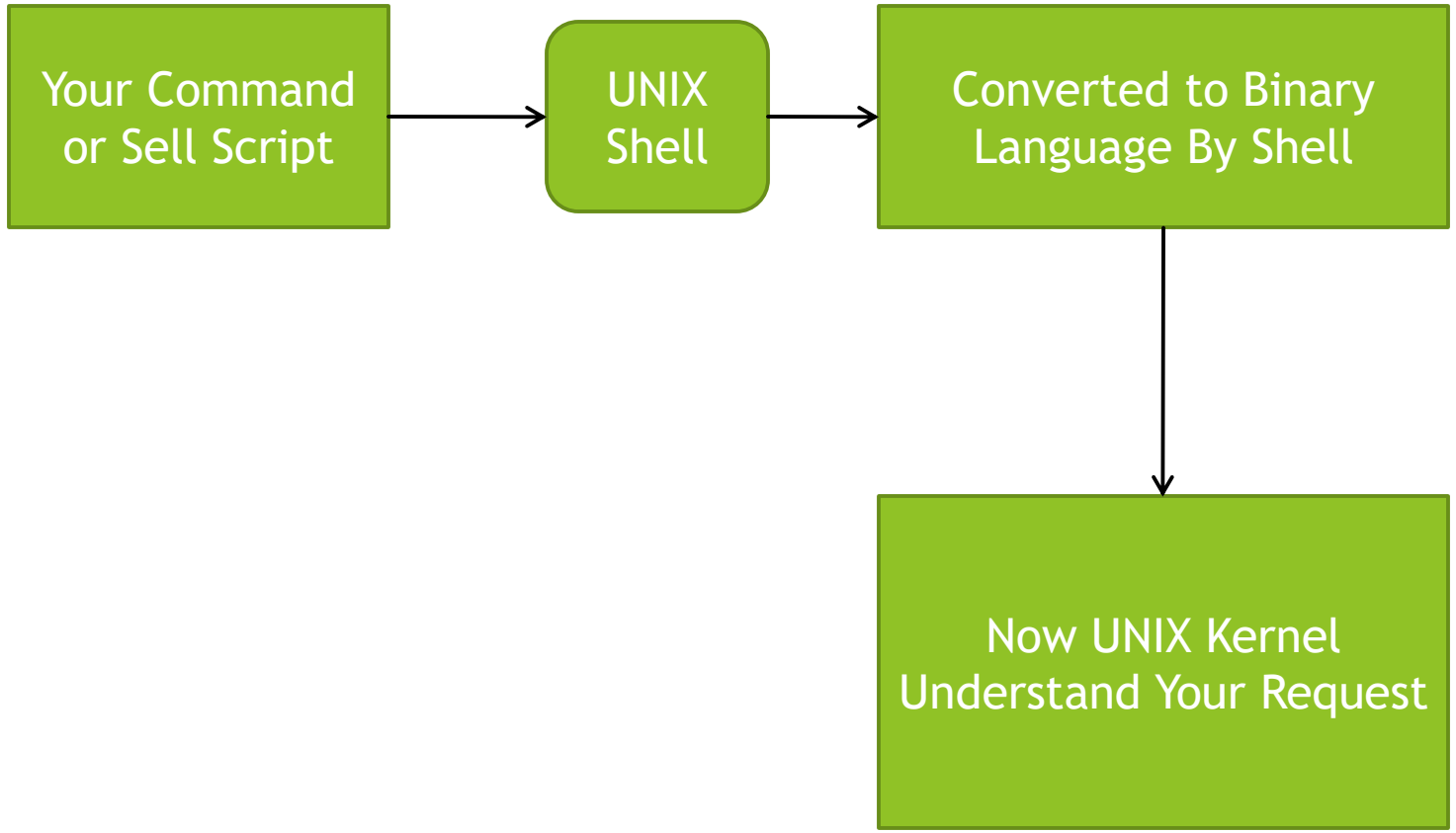
Shell Programming

Amit Patel

INTRODUCTION

- A *shell is a program that takes commands typed by the user and calls the operating system to run those commands.*
- A *shell is a program that acts as the interface between you and the Unix system, allowing you to enter* commands for the operating system to execute.
- Shell accepts your instruction or commands in English and translate it into computers native binary language.
- There are following type of shell.

Bourne Shell	C Shell	Korn Shell	Bash Shell	Tcsh Shell


Your Command or Sell Script
UNIX Shell
Converted to Binary Language By Shell
Now UNIX Kernel Understand Your Request

Changing Your Shell

- To find all available shells in your system type following command:

  $ cat /etc/shells

- The basic Syntax :

  chsh *username new_default_shell*

- The administrator can change your default shell.

Why Use Shell Script?

- You can use shell scripts to automate administrative tasks.
- Encapsulate complex configuration details.
- Get at the full power of the operating system.
- The ability to combine commands allows you to create new commands
- Adding value to your operating system.

Shell – Programming Language

- There are two ways of writing shell programs.
  1. You can type a sequence of commands and allow the shell to execute them interactively.
  2. You can store those commands in a file that you can then invoke as a program(shell script).

Shell – Programming Language

- *Shell script is a series of command(s) stored in a plain text file.*
- *A shell script is similar to a batch file in MS-DOS, but is much more powerful.*
- A good shell script will have comments, preceded by a pound sign, #, describing the steps.
- There are conditional tests, such as value A is greater than value B, loops allowing us to go through massive amounts of data, files to read and store data, and variables to read and store data, and the script may include functions.
- Shell scripts is interpreted. This means they are not compiled.

Why to Write Shell Script?

- Shell script can take input from user, file and output them on screen.
- Useful to create our own commands.
- Save lots of time.
- To automate some task of day today life.
- System Administration part can be also automated.

How to Write Shell Script?

- Create a script using vi editor with .sh or .bsh extension.
- Setup executable permission.
- Run the script.

Step 1: Creating a Script

- As discussed earlier shell scripts stored in plain text file, generally one command per line.

```
$ vi myscript.sh
```

- Make sure you use .bash or .sh file extension for each script. This ensures easy identification of shell script.

Step 2: Set Execute Permission

- Once script is created, you need to setup executable permission on a script.

  Why?

  - Without executable permission, running a script is almost impossible.
  - Besides executable permission, script must have a read permission.
- Syntax to setup executable permission:
  - $ chmod +x <your-script-name>
  - $ chmod 755 <your-script-name>

Step 3: Run Script

- Now your script is ready with proper executable permission on it. Next, test script by running it.
  - bash your-script-name
  - sh your-script-name
  - ./your-script-name
- Examples
  - $ bash bar
  - $ sh bar
  - $ ./bar

Example Script

- we create a test.sh script. Note all the scripts would have .sh extension. : $ vi test.sh
- Before we 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.
- For example: #!/bin/sh
- This tells the system that the commands that follow are to be executed by the Bourne shell.
- It's called a shebang because the # symbol is called a hash, and the ! symbol is called a bang.
- To create a script containing these commands, you put the shebang line first and then add the commands –

```
#!/bin/bash
pwd
ls
```

Example Script

```
$ vi first
```

```
#!/bin/sh
# My first shell script
#
clear
echo “This is my First script”
```

```
$ chmod 755 first

$ ./first
```

Variable in Shell

- In Shell, there are two types of variable:
  - **System variables** - Created and maintained by Linux itself. This type of variable defined in CAPITAL LETTERS.
  - **User defined variables (UDV)** - Created and maintained by user. This type of variable defined in lower letters.

Variable in Shell

- A variable is a character string to which we assign a value.
- The value assigned could be a number, text, filename, device, or any other type of data.
- A variable is nothing more than a pointer to the actual data.
- The shell enables you to create, assign, and delete variables.

Variable Names

- The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (_).
- By convention, Unix Shell variables would have their names in UPPERCASE.

Variable in Shell

- The following examples are valid variable names

  _ALI , TOKEN_A, VAR_1, VAR_2

- Following are the examples of invalid variable names

  2_VAR, -VARIABLE, VAR1-VAR2, VAR_A!

- The reason you cannot use other characters such as !,*, or - is that these characters have a special meaning for the shell.

User Defined Variable

- To define UDV use following syntax:
  - variable name=value
  - Example: $ no=10
- **Rules for Naming variable name**

1. Variable name must begin with Alphanumeric character or underscore character (_), followed by one or more Alphanumeric character.
2. Don't put spaces on either side of the equal sign when assigning value to variable.
3. Variables are case-sensitive.
4. You can define NULL variable
5. Do not use **?,*** etc, to name your variable names.

Print OR Access UDV

- To print or access UDV use following syntax :
  - $variablename.
- Examples:
  - $vech=Bus
  - $ n=10
  - $ echo $vech
  - $ echo $n

Print OR Access UDV

- Don’t try
  - **$ echo vech**
  - it will print vech instead its value 'Bus‘.
  - **$ echo n**
  - it will print n instead its value '10‘.
- You must *use $ followed by variable name.*

Special Variables

- We can not use certain non-alphanumeric characters in your variable names. This is because those characters are used in the names of special Unix variables.
- Special variables are reserved for specific functions.
- For example: $ character represents the process ID number, or PID, of the current shell:

```
$echo $$
```

- Above command would write PID of the current shell – 29949
- There are following types of special variable:

$0	The filename of the current script.
$n	These variables correspond to the arguments with which a script was invoked. Here n is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on).
$#	The number of arguments supplied to a script.
$*	All the arguments are double quoted. If a script receives two arguments, $* is equivalent to $1 $2.
$@	All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2.
$?	The exit status of the last command executed.
$$	The process number of the current shell. For shell scripts, this is the process ID under which they are executing.
$!	The process number of the last background command.

Command-line Arguments

- The command-line arguments $1, $2, $3,...$9 are positional parameters, with $0 pointing to the actual command, program, shell script, or function and $1, $2, $3, ...$9 as the arguments to the command.
- Following script uses various special variables related to command line –

```
#!/bin/sh

echo "File Name: $0"

echo "First Parameter : $1"

echo "First Parameter : $2"

echo "Quoted Values: $@"

echo "Quoted Values: $*"

echo "Total Number of Parameters : $#"
```

Command-line Arguments

- Here is a sample run for the above script −

```
$./test.sh OM SAI

File Name : ./test.sh

First Parameter : OM

Second Parameter : SAI

Quoted Values: OM SAI

Quoted Values: OM SAI

Total Number of Parameters : 2
```

$* and $@

- There are special parameters that allow accessing all of the command-line arguments at once.
- $* and $@ both will act the same unless they are enclosed in double quotes, "".
- Both the parameter specifies all command-line arguments but

  The "$*" special parameter takes the entire list as one argument with spaces between

  And

  The "$@" special parameter takes the entire list and separates it into separate arguments.
- We can write the shell script shown below to process an unknown number of command-line arguments with either the $* or $@ special parameters −

$* and $@

```
#!/bin/sh
for i in "$*"
do
      echo $i ;
done
echo "for loop completed of \$*";
for i in "$@"
do
      echo $i;
done
echo "for loop completed of \$@";
```

$* and $@

- There is one sample run for the above script –

$./test.sh "First Argument" "Second Argument“

Output of this as follows:

First Argument Second Argument
for loop completed of $*
First Argument
Second Argument
For loop completed of $@

Output shows that

- $* is considering all arguments as one string. And whole string assigns to one variable only.
- But $@ is considering quoted string as a single argument and displaying there two arguments as it is.

Exit Status

- The $? variable represents the exit status of the previous command.
- Exit status is a numerical value returned by every command upon its completion.
- As a rule, most commands return an exit status of 0 if they were successful, and 1 if they were unsuccessful.
- Some commands return additional exit statuses for particular reasons.
- For example, some commands differentiate between kinds of errors and will return various exit values depending on the specific type of failure.
- Following is the example of successful command –

Exit Status

$./test.sh OM SAI

File Name : ./test.sh

First Parameter : OM

Second Parameter : SAI

Quoted Values: OM SAI

Quoted Values: OM SAI

Total Number of Parameters : 2

$echo $?

0