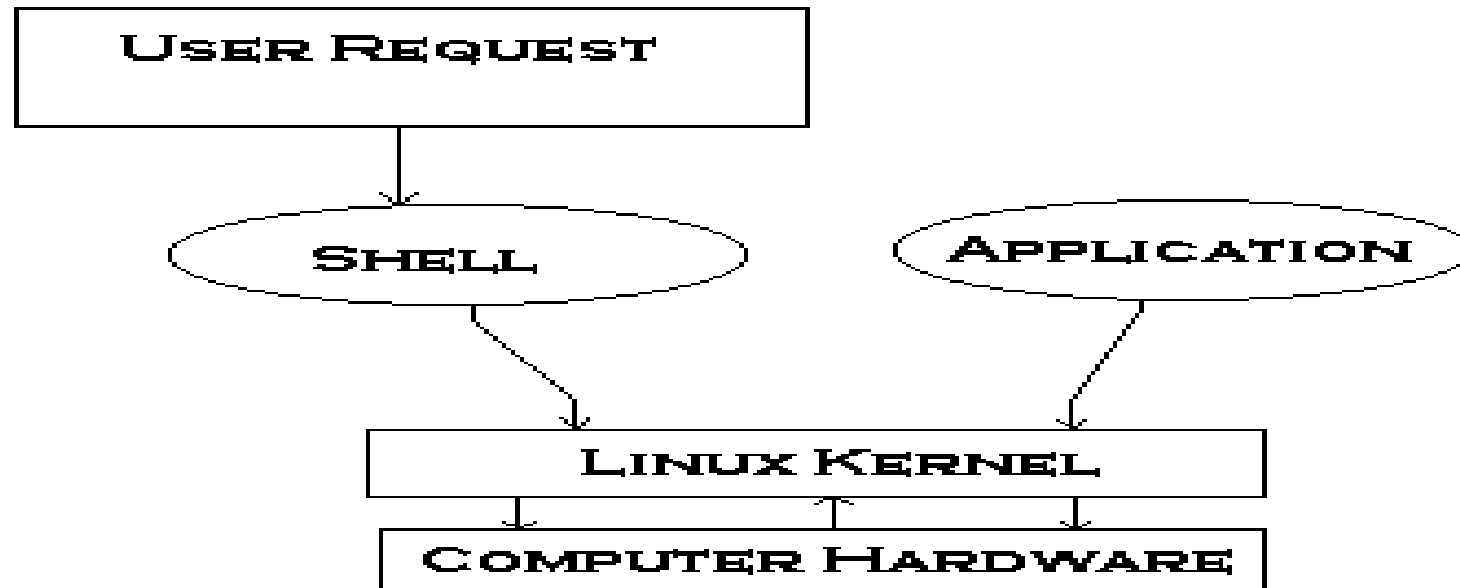


What is shell?

A shell is a program that takes commands typed by the user and calls the operating system to run those commands.

Shell accepts your instruction or commands in English and translates it into computer's native binary language.



Kind of Shells

- Bourne Shell
- C Shell
- Korn Shell
- Bash Shell
- Tcsh Shell

Kind of Shells

1. The C Shell – Denoted as csh.

- It incorporated features such as aliases and command history.
- It includes helpful programming features like built-in arithmetic and C-like expression syntax.
- Command full-path name is `/bin/csh`

Kind of Shells

2. The Bourne Shell – Denoted as sh

- It was written by Steve Bourne at AT&T Bell Labs.
- It is the original UNIX shell.
- It lacks features for interactive use like the ability to recall previous commands. It also lacks built-in arithmetic and logical expression handling.
- It is the default shell for Solaris OS.
- Command full-path name is `/bin/sh` and `/sbin/sh`

Kind of Shells

3. The Korn Shell

- It is denoted as ksh
- It includes features like built-in arithmetic and C-like arrays, functions, and string-manipulation facilities. It is faster than C shell.
- It is compatible with script written for C shell.
- Command full-path name is /bin/ksh

Kind of Shells

4. GNU Bourne-Again Shell –

- Denoted as bash
- It is compatible to the Bourne shell. It includes features from Korn and Bourne shell.
- Command full-path name is `/bin/bash`

Check Shell

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

```
$ cat /etc/shells
```

Changing Your Default Shell

chsh -s <shell path>

COMMAND LINE ARGUMENTS

Create a script

Shell scripts store 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 the shell script.

Setup executable 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>*

Run a script (execute a script)

Now your script is ready with proper executable permission on it. Next, test the script by running it.

- *bash <your-script-name>*
- *sh <your-script-name>*
- *./your-script-name*

Examples

- \$ bash file.sh
- \$ sh file.sh
- \$./bar file.sh

Example

```
$ vi first
```

```
#  
# My first shell script  
#  
clear  
echo "This is my First  
script"
```

```
$ chmod 755 first
```

```
$ ./first
```

Variables in Shell

In Linux (Shell), there are two types of variable:

- **System variables** - Created and maintained by Linux itself. This type of variable defined in CAPITAL LETTERS. i.e environmental variables
- **User defined variables (UDV)** – Created and maintained by user. This type of variable defined in lower letters.

Variables in Shell

A variable is nothing more than a pointer to the actual data. The shell enables you to create, assign, and delete variables.

Defining Variables

```
variable_name=variable_value
```

Variables in Shell

A variable is nothing more than a pointer to the actual data. The shell enables you to create, assign, and delete variables.

Defining Variables

`variable_name=variable_value`

Example:

`VAR1="LPU"`

`VAR2=100`

Accessing Values

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

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

```
NAME="LPU"  
echo $NAME
```

Read-only Variables

mark variables as read-only by using the read-only command. After a variable is marked read-only, its value cannot be changed.

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

```
NAME="Lovely Professional"  
readonly NAME  
NAME="University"
```

Unsetting Variables

Unsetting or deleting a variable directs the shell to remove it from the list of variables it tracks. Once you unset a variable, you cannot access the stored value in the variable.

```
unset variable_name
```

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

```
NAME="LPU"
```

```
unset NAME
```

```
echo $NAME
```

System Environmental Variables

Variable	Description
env	To list all system environment variables
USER	The username
HOME	Default path to the user's home directory
SHELL	Shell being used by the user
UID	User's unique ID
HISTFILE	holds the name and location of your Bash history file
HISTFILESIZE	how many commands can be stored in the .bash_history file
PATH	This variable contains a colon (:)-separated list of directories in which your system looks for executable files.

Read

Used to get input from the user

```
echo "enter your name"  
read name
```

```
echo "your name is $name"
```

Use Command Line Arguments in a Bash Script

Different ways to process the arguments passed to a bash script inside the script:

1. Positional Parameters

- Arguments passed to a script are processed in the same order in which they're sent. The indexing of the arguments starts at one, and the first argument can be accessed inside the script using `$1`.
- Similarly, the second argument can be accessed using `$2`, and so on.
- The positional parameter refers to this representation of the arguments using their position.

Use Command Line Arguments in a Bash Script

1. Positional Parameters

```
echo "Username: $1";  
echo "Age: $2";  
echo "Full Name: $3";
```

Use Command Line Arguments in a Bash Script

Shell Parameters

Parameters	Function
\$1-\$9	Represent positional parameters for arguments one to nine
\${10}-\${n}	Represent positional parameters for arguments after nine
\$0	Represent name of the script
\$*	Represent all the arguments as a single string
@	Same as \$*, but differ when enclosed in ("")
\$#	Represent total number of arguments
\$\$	PID of the script
\$?	Represent last return code

test

On terminal

```
test 5 -gt 6 ; echo $?
```

Shell Relational

Math- ematical Operator in Shell Script	Meaning	Normal Arithmetical/ Mathematical Statements	But in Shell	
			For test statement with if command	For [expr] statement with if command
-eq	is equal to	$5 == 6$	if test 5 -eq 6	if expr [5 -eq 6]
-ne	is not equal to	$5 != 6$	if test 5 -ne 6	if expr [5 -ne 6]
-lt	is less than	$5 < 6$	if test 5 -lt 6	if expr [5 -lt 6]
-le	is less than or equal to	$5 <= 6$	if test 5 -le 6	if expr [5 -le 6]
-gt	is greater than	$5 > 6$	if test 5 -gt 6	if expr [5 -gt 6]
-ge	is greater than or equal to	$5 >= 6$	if test 5 -ge 6	if expr [5 -ge 6]

if condition

Syntax:

if [condition]

then

Statement 1

else

Statement 2

fi

Example

```
$ vim myscript.sh
```

```
read choice
```

```
if [ $choice -gt 0 ]; then
```

```
echo "$choice number is positive"
```

```
else
```

```
echo "$ choice number is negative"
```

```
fi
```

File Test Operators

- d : True if the file exists and is a directory.
- e : True if the file exists.
- f : True if the file exists and is a regular file.
- r : True if the file exists and is readable.
- s : True if the file exists and has a size greater than zero.
- w : True if the file exists and is writable.
- x : True if the file exists and is executable.

Loops in Shell Scripts

Bash supports:

1. for loop.
2. while loop.

Note that in each and every loop:

- First, the variable used in loop condition must be initialized, then execution of the loop begins.
- A test (condition) is made at the beginning of each iteration.
- The body of loop ends with a statement that modifies the value of the test (condition) variable.

for Loop

Syntax:

for { variable name } in { list }

do

execute one for each item in the list

until the list is not finished and

repeat all statement between do and

done

done

Example

```
for i in 1 2 3 4 5
```

```
do
```

```
    echo "Welcome $i times"
```

```
done
```


for Loop

Syntax:

```
for (( expr1; expr2; expr3 ))
```

```
do
```

```
    repeat all statements between
```

```
    do and done until expr2 is
```

```
TRUE
```

```
Done
```

Example

```
for (( i = 0 ; i <= 5; i++ ))
```

```
do
```

```
    echo "Welcome $i times"
```

```
done
```

Nesting of for Loop

```
$ vi nestedfor.sh
```

```
for (( i = 1; i <= 5; i++ ))  
do  
    for (( j = 1 ; j <= 5; j++ ))  
do  
echo -n "$i "  
done
```

while loop

Syntax:

while [condition]

do

command1 command2

command3

done

Example

i=1

while [\$i -le 10]

do

echo "\$n * \$i = `expr \$i * \$n`"

i=`expr \$i + 1`

done

The case Statement

Syntax:

```
case $variable-name in  
    pattern1) command.....;;  
    pattern2) command.....;;  
    pattern N) command.....;;  
  
    *) command ;;  
esac
```

Example

read var

case \$var in

1) echo "One";;

2) echo "Two";;

3) echo "Three";;

4) echo "Four";;

*) echo "Sorry, it is bigger than
Four";;

esac

Array

- Shell supports a different type of variable called an **array variable**. This can hold multiple values at the same time.
- **Defining Array Values**

array—name[index]=value

Examples

Name[0]="ABC"

Name[1]="PQR"

Name[2]="XYZ"

Name[3]="GOD"

- **Accessing Array Values**
- **`${array—name[index]}`**
- You can access all the items in an array in one of the following ways –
- **`${array-name[*]}`**
- **`${array-name[@]}`**

Array Script

- `echo "enter name"`
- `read -a names`
- `echo "names :${names[0]}, ${names[1]}"`

Select command

- Constructs simple menu from word list
- Allows user to enter a number instead of a word
- User enters sequence number corresponding to the word

Syntax:

```
select WORD in LIST
do
    RESPECTIVE-COMMANDS
done
```

Loops until end of input, i.e. ^d (or ^c)

Select example

```
#!/bin/bash
select var in alpha beta gamma
do
```

```
    echo $var
```

```
done
```

Prints: 1) alpha

2) beta

3) gamma

#? 2

beta

#? 4

#? 1

alpha

The until Loop

Purpose:

To execute commands in “command-list” as long as “expression” evaluates to false

Syntax:

```
until [ expression ]  
do  
    command-list  
done
```

Example: Using the until Loop

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

```
COUNTER=20
```

```
until [ $COUNTER -lt 10 ]
```

```
do
```

```
    echo $COUNTER
```

```
    let COUNTER-=1
```

```
done
```

break and continue

The break statement

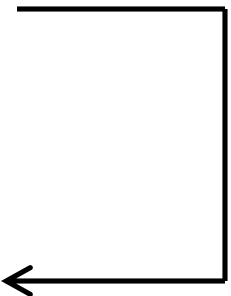
- transfer control to the statement AFTER the done statement
- terminate execution of the loop

The continue statement

- transfer control to the statement TO the done statement
- skip the test statements for the current iteration
- continues execution of the loop

The break command


```
while [ condition ]  
do  
    cmd-1  
    break  
    cmd-n  
done  
echo "done"
```



This iteration is over
and there are no more
iterations

The continue command

```
while [ condition ]  
do  
    cmd-1  
    continue  
    cmd-n  
done  
echo "done"
```

A diagram consisting of a vertical line with an arrowhead pointing left to the closing bracket of the 'while' loop. A horizontal line branches off to the left from the vertical line, indicating a jump back to the start of the loop body.

This iteration is
over; do the next
iteration

Example:

```
for index in 1 2 3 4 5 6 7 8 9 10
do
    if [ $index -le 3 ]; then
        echo "continue"
        continue
    fi
    echo $index
    if [ $index -ge 8 ]; then
        echo "break"
        break
    fi
done
```

Grep: matches a pattern throughout a file.

- Grep <string> <filename>
- Grep <string> <filename> <filename>
- Options
 - -i: ignore case
 - -v: invert match.To select non-matching lines.
 - -c: counting no. of matches

Basic Regular expressions

Symbol	Descriptions
.	matches a single character
^	matches start of string
\$	matches end of string
*	matches up zero or more times the preceding character
\	Represent special characters
()	Groups regular expressions
?	Matches up exactly one character

[a1v]---any one character out of a or 1 or v

[^a1v]---any character except a, 1 and v

[[:alnum:]]---Alphanumeric [a-z A-Z 0-9]

[[:alpha:]]---Alphabetic [a-z A-Z]

[[:blank:]]---Blank characters (spaces or tabs)

[[:digit:]]---Numbers [0-9]

[[:lower:]]---Lowercase letters [a-z]

[[:upper:]]---Uppercase letters [A-Z]

[[:xdigit:]]---Hex digits [0-9 a-f A-F]