Introduction to Shell Programming

What is Kernel?

- The kernel is a computer program that is the core of a computer’s operating system, with complete control over everything in the system.
- It manages the following resources of the Linux system –

- File management
- Process management
- I/O management
- Memory management
- Device management etc.

What is Shell?

- A shell is a special user program that provides an interface for the user to use operating system services.
- Shell accepts human-readable commands from users and converts them into something which the kernel can understand.
- It is a command language interpreter that executes commands read from input devices such as keyboards or from files.
- The shell gets started when the user logs in or starts the terminal.

Shell Scripting

- Shells are interactive, which means they accept commands as input from users and execute them.
- As a shell can also take commands as input from file, we can write these commands in a file and can execute them in shell to avoid this repetitive work.
- Shell script consist of set of commands to perform a task.
- All the commands execute sequentially.
- These files are called Shell Scripts or Shell Programs.
- Some task like file manipulation, program execution, user interaction, automation of task etc. can be done

- A shell script comprises the following elements –

- Shell Keywords – if, else, break etc.
- Shell commands – cd, ls, echo, pwd, touch etc.
- Functions
- Control flow – if..then..else, case and shell loops etc.

Shell Types

- There are several shells are available for Linux systems like –

- BASH (Bourne Again Shell) – It is the most widely used shell in Linux systems. It is used as default login shell in Linux systems and in macOS. It can also be installed on Windows OS.
- If you are using a Bourne-type shell, the ***$*** character is the default prompt.
- CSH (C Shell) – The C shell’s syntax and its usage are very similar to the C programming language.
- KSH (Korn Shell) – The Korn Shell was also the base for the POSIX Shell standard specifications etc.
- Each shell does the same job but understands different commands and provides different built-in functions

Why do we need shell scripts?

There are many reasons to write shell scripts:

- To avoid repetitive work and automation
- System admins use shell scripting for routine backups.
- System monitoring
- Adding new functionality to the shell etc.

Some Advantages of shell scripts

- The command and syntax are exactly the same as those directly entered in the command line, so programmers do not need to switch to entirely different syntax
- Writing shell scripts are much quicker
- Quick start
- Interactive debugging etc.

Some Disadvantages of shell scripts

- Prone to costly errors, a single mistake can change the command which might be harmful.
- Slow execution speed
- Design flaws within the language syntax or implementation
- Not well suited for large and complex task
- Provide minimal data structure unlike other scripting languages. etc.

Scripting Format

- A shell script has syntax just like any other programming language.
- Shell scripts are similar to the batch file in MS-DOS. Each shell script is saved with `.sh` file extension e.g., myscript.sh.
- **How to determine Shell ?**
- You can get the name of your shell prompt, with following command :
- *Syntax:*
- echo $SHELL
- The $ sign stands for a shell variable, echo will return the text whatever you typed in.

- The sign **#!** is called she-bang and is written at top of the script. It passes instruction to program ***/bin/sh.***

- To run your script in a certain shell (shell should be supported by your system), start your script with #! followed by the shell name.
- Example:
- ***#!/bin/bash***
- ***echo Hello World***

~/bin

bin directories:

- Program files or commands, also called binary executable files and script files, are kept in various places throughout the system.
- Usually these binary files are stored in bin (short for binary) directories throughout the system.
- If you take a look at the paths are stored in your $PATH environment variable, you will notice that many of these directories end in .../bin.

Your ~/bin directory:

- You may also notice that your path may contain a bin directory that is listed as being in your home directory
- This is where you can store your own compiled programs or scripts that you would like to be able to run from anywhere on the system.

Chmod command in Linux

- In Unix operating systems, the chmod command is used to change the access mode of a file.
- The name is an abbreviation of change mode.
- Which states that every file and directory has a set of permissions that control the permissions like who can read, write or execute the file.
- In this the permissions have three categories: read, write, and execute simultaneously represented by `r`, `w` and `x`.
- These letters combine together to form a specific permission for a group of users.
- The `chmod` command is used to modify this permission so that it can grant or restrict access to directories and files.

- *Syntax:*
- ***chmod [options] [mode] [File_name]***

- ***1***) Symbolic mode
- The following operators can be used with the symbolic mode:

Operators	*Definition*
`+`	Add permissions
`-`	Remove permissions
`=`	Set the permissions to the specified values

- The following letters that can be used in symbolic mode:

Letters	*Definition*
`r`	Read permission
`w`	Write permission
`x`	Execute permission

- The following Reference that are used:

ReferenceClass	
u	Owner
g	Group
o	Others
a	All (owner,groups,others)

- Examples of Using the Symbolic mode:
- Read, write and execute permissions to the file owner:

chmod u+rwx [file_name]

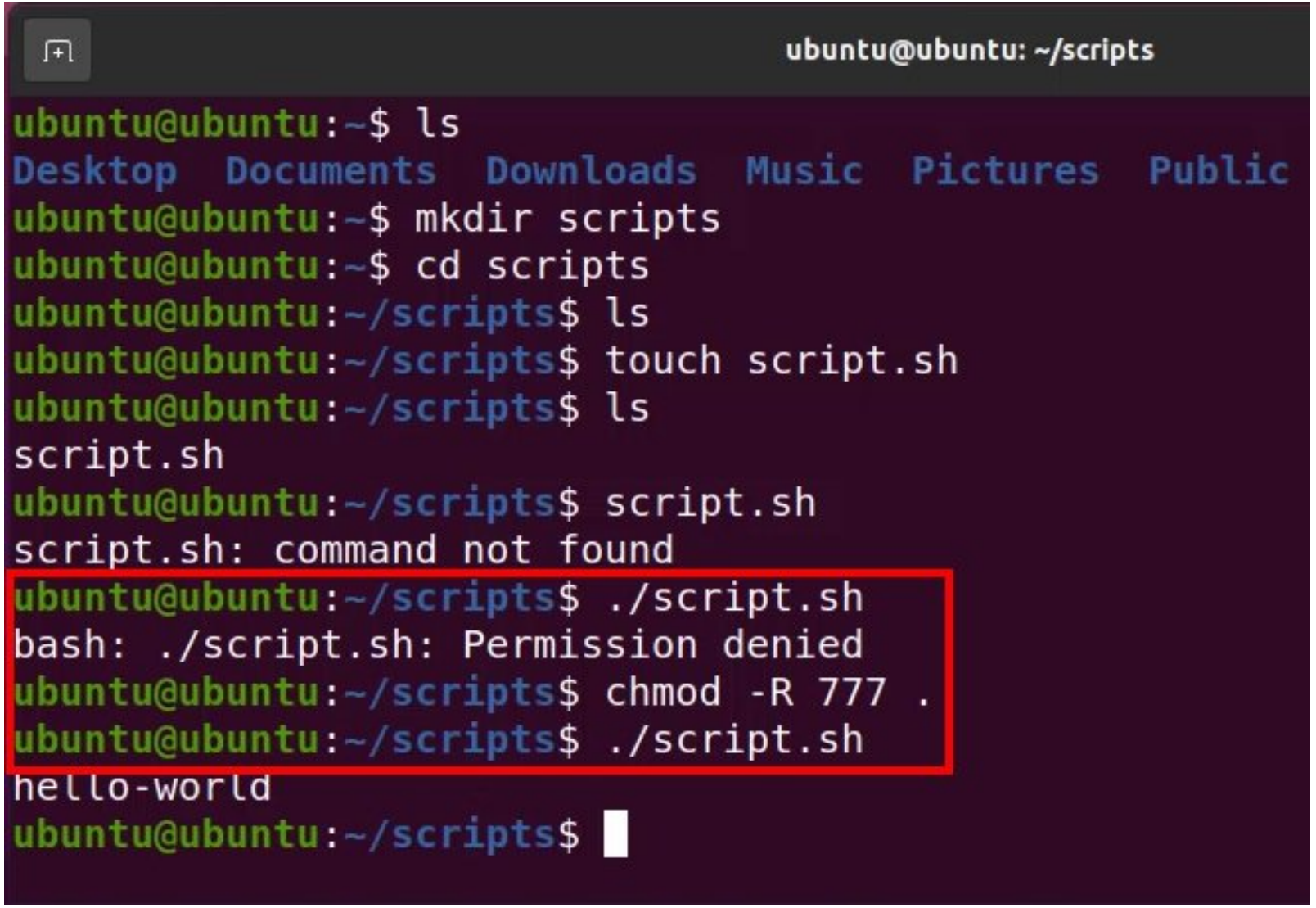
- Remove write permission for the group and others:

chmod go-w [file_name]

- Read and write for Owner, and Read-only for the group and other:

chmod u+rw,go+r [file_name]

How to Create Shell Script in Linux/Unix



Follow these steps to create and run a shell script of your own from your Ubuntu home directory:

- Create a directory to store your shell script
- Create a file in your scripts folder with a .sh extension
- Add an execute permission to the file to avoid permission denied errors during execution
- Create the bash script
  - On the first line of the script, add the shebang (#!) followed by the path to the shell interpreter. In our case, we'll use #!/bin/bash to specify bash as the interpreter.
  - The first line in every bash script is: #!/bin/bash
- Use the ./ (dot slash) notation and run the shell script by name in the Terminal window
- Press CTRL+C to stop the running Linux script if it does not terminate gracefully

- When you run an Linux shell script in an Ubuntu terminal window, you need to prepend a dot slash (./) to the file’s name. If you don’t, Ubuntu will look on the operating system’s PATH for your program.

Shell Scripting Comments

- Any line starting with a hash (#) becomes comment. Comment means, that line will not take part in script execution. It will not show up in the output.

Shell Variables

- 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.

Defining Variables

- Variables are defined as follows −

- variable_name=variable_value
- For example −
- NAME=“xyz"

Accessing Values

- To access the value stored in a variable, prefix its name with the dollar sign ($) −
- #!/bin/sh

NAME=“xyz”

echo $NAME

The above script will produce the following value −

xyz

Read-only Variables:

Shell provides a way to 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="Zara Ali"
readonly NAME
NAME="Qadiri"
```

The above script will generate the following result

```
/bin/sh: NAME: This variable is read only.
```

Different types of Variables

- In shell scripting there are three main types of variables are present. They are –

- Local Variables
- Global Variables or Environment Variables
- Shell Variables or System Variables

- Local Variable
- A local variable is a special type of variable which has its scope only within a specific function or block of code. Local variables can override the same variable name in the larger scope.
- Global Variables
- A global variable is a variable with global scope. It is accessible throughout the program. Global variables are declared outside any block of code or function.
- Shell Variables
- These are special types of variables. They are created and maintained by Linux Shell itself.

Some useful shell variables are –

Variable Name	Description	Usage
BASH_VERSION	Holds the version of this instance of bash.	echo $BASH_VERSION
HOME	Provides a home directory of the current user.	echo $HOME
HOSTNAME	Provides computer name	echo $HOSTNAME
USERNAME	Provides username	echo $USERNAME

Bash read command

- Bash has no built-in function to take the user’s input from the terminal.
- The read command of Bash is used to take the user’s input from the terminal.

Syntax

- read [options] [var1, var2, var3…]

A **no-argument call of *read* fetches a single line from the standard input stream and assigns it to the *REPLY* built-in variable**:

```
#!/bin/bash
#Print the prompt message
echo "Enter your favorite color: "
#Take the input
read
#Print the input value
echo "Your favorite color is $REPLY"
```

```
File  Edit  View  Search  Terminal  Help
fahmida@fahmida:~/Desktop/bash$ bash read1.bash
Enter your favorite color :
Blue
Your favorite color is Blue
fahmida@fahmida:~/Desktop/bash$
```

```
#!/bin/bash
#Print the prompt message
echo "Enter the product name: "
#Take the input with a single variable
read item

#Print the prompt message
echo "Enter the color variations of the product: "
#Take three input values in three variables
read color1 color2 color3

#Print the input value
echo "The product name is $item."
#Print the input values
echo "Available colors are $color1, $color2, and $color3."
```

```
fahmida@fahmida:~/Desktop/bash$ bash read2.bash
Enter the product name:
Shirt
Enter the color variations of the product:
Red Green Blue
The product name is Shirt.
Available colors are Red, Green, and Blue.
fahmida@fahmida:~/Desktop/bash$
```

```
#!/bin/bash

# This script demonstrates the use of variables

# Assign a value to a variable
name="John"

# Access the value of the variable
echo "Hello, $name!"

# Perform arithmetic operations with variables
a=5
b=3
sum=$((a + b))
echo "The sum of $a and $b is $sum"
```

Output:

```
Hello, John!
The sum of 5 and 3 is 8
```

Some options of the read command require an additional parameter to use. The most commonly used options of the read command are mentioned in the following:

Option	Purpose
-d <delimiter>	It is used to take the input until the delimiter value is provided.
-n <number>	It is used to take the input of a particular number of characters from the terminal and stop taking the input earlier based on the delimiter.
-N <number>	It is used to take the input of the particular number of characters from the terminal, ignoring the delimiter.
-p <prompt>	It is used to print the output of the prompt message before taking the input.
-s	It is used to take the input without an echo. This option is mainly used to take the input for the password input.

-a	It is used to take the input for the indexed array.
-t <time>	It is used to set a time limit for taking the input.
-u <file descriptor>	It is used to take the input from the file.
-r	It is used to disable the backslashes.

1. **-p**: Displays a prompt to the user before reading the input.

   For example -

```
read -p 'Enter your name: ' name
```

   Output:

```
Enter your name:
```

   This command prompts the user to enter their name and stores the input in the 'name' variable.

Set Character Limit

The read command offers two options when limiting the number of characters for the user input:

1. Use the `-n` option and provide a number to set the character limit. For example:

```
read -n 3
```

```
kb@phoenixNAP:~$ read -n 3
abckb@phoenixNAP:~$
```

Press **Enter** after one character to end the command before reaching the character limit. Without pressing **Enter**, the command exits automatically after three characters.

2. Use the `-N` option and provide a number to set the character limit while ignoring the delimiter. For example:

```
read -N 3
```

```
kb@phoenixNAP:~$ read -N 3
1
2kb@phoenixNAP:~$
```

Pressing **Enter** does not end the command. However, the keystroke counts as a character.

2. **-t**: Sets a timeout for the read command. If no input is provided within the specified number of seconds, the command ends.

   For example -

```
read -t 5 -p 'Enter your name: ' name
```

   Output:

```
Enter your name:
```

   This command waits for 5 seconds for the user to enter their name. If no input is provided within 5 seconds, the command ends.

3. **-a**: Reads the input into an array instead of a single variable.

For example -

```
read -a names
```

This command reads the input into an array named 'names'.

Arrays

Instead of using individual variables to store a string, add the `-a` option to save the input in an array. For example:

```
read -a array <<< "Hello world!"
```

Retrieve the array elements with:

```
echo ${array[0]}
echo ${array[1]}
```

```
kb@phoenixNAP:~$ read -a array <<< "Hello world!"
kb@phoenixNAP:~$ echo ${array[0]}
Hello
kb@phoenixNAP:~$ echo ${array[1]}
world!
```

Delimiters

The `read` command defines two delimiter types:

1. The delimiter for the `read` command.

By default, pressing **Enter** (newline) ends the command. Add the `-d` tag and provide a different delimiter in quotes to terminate differently.

For example:

```
read -d "-"
```

```
kb@phoenixNAP:~$ read -d "-"
Hello-kb@phoenixNAP:~$ echo $REPLY
Hello
```

Instead of a new line, the new delimiter is a dash (`-`) instead of a new line. The command terminates when reaching the delimiter, disregarding the number of arguments. The response in `$REPLY` or the provided variable stores the user input without the dash (`-`).

Escape Characters and Backslashes

The `read` command allows splitting long inputs into multiple lines using backslashes. For example:

```
read password prompt terminal output
```

```
Hello \
world\
!
```

```
kb@phoenixNAP:~$ read
Hello \
> world\
> !
kb@phoenixNAP:~$ echo $REPLY
Hello world!
```

Command Line Arguments in Shell Script

- Command-line arguments are read in a positional manner, from position $1, $2, ..$n. The pattern $ followed by an integer is a reserved combination to represent the command-line arguments.
- Command line arguments are also known as positional parameters. These arguments are specific with the shell script on terminal during the run time.
- These are also known as special variables provided by the shell.

Special Variable	Variable Details
$1 to $n	$1 is the first arguments, $2 is second argument till $n n'th arguments. From 10'th argument, you must need to inclose them in braces like ${10}, ${11} and so on
$0	The name of script itself
$$	Process id of current shell
$*	Values of all the arguments. All agruments are double quoted
$#	Total number of arguments passed to script
$@	Values of all the arguments
$?	Exit status id of last command
$!	Process id of last command

Positional Parameters

- Command-line arguments are passed in the positional way i.e. in the same way how they are given in the program execution. Let us see with an example.
- *input*

```
echo "The first fruit is: $1"
echo "The second fruit is: $2"
echo "The third fruit is: $3"
```

- Output:

```
./fruit.sh apple pear orange
The first fruit is: apple
The second fruit is: pear
The third fruit is: orange
```

```
echo "The first fruit is: $1"
echo "The second fruit is: $2"
echo "The third fruit is: $3"
echo "All fruits are: $@"
```

```
./fruit.sh apple pear orange
The first fruit is: apple
The second fruit is: pear
The third fruit is: orange
All fruits are: apple pear orange
```

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

Now let’s run this script with the three input parameters:

```
sh userReg-positional-parameter.sh john 25 'John Smith'
```

The output will be:

```
Username : john
Age: 25
Full Name: John Smith
```

Shell

```
1  #!/bin/bash
2
3  ### Print total arguments and their values
4
5  echo "Total Arguments:" $#
6  echo "All Arguments values:" $@
7
8  ### Command arguments can be accessed as
9
10 echo "First->"  $1
11 echo "Second->" $2
```

```
#!/bin/bash

echo "Script name is: $0"
echo "Arg1 is $1"
echo "Arg1 is $2"
echo "Arg1 is $3"
echo "-----------"
echo "All args: $*"
echo "All args count: $#"
```

This shell script is executed with the arguments as shown below:

```
$ bash arg_intro.sh runtime inputs
Script name is: ./arg_intro.sh
Arg1 is runtime
Arg1 is inputs
Arg1 is
-----------
All args: runtime inputs
All args count: 2
```

Now we will use a bash file named **animals.sh** as an example.

```
#!/bin/bash

echo "The animal in the first enclosure is: $1"
echo "The animal in the second enclosure is: $2"
echo "The animal in the third enclosure is: $3"
echo "The total number of animals in the zoo are: $#"
echo "The names of all the animals are: $@"
```

Now we will run this script in the terminal:

```
rahul@tecadmin:~/Documents$ ./animals.sh Lion Rhino Elephant
The animal in the first enclosure is: Lion
The animal in the second enclosure is: Rhino
The animal in the third enclosure is: Elephant
The total number of animals in the zoo are: 3
The names of all the animals are: Lion Rhino Elephant
rahul@tecadmin:~/Documents$
```

Arithmetic Operators

Assume variable **a** holds 10 and variable **b** holds 20 then −

Operator	Example
+ (Addition)	`expr $a + $b` will give 30
- (Subtraction)	`expr $a - $b` will give -10
* (Multiplication)	`expr $a \* $b` will give 200
/ (Division)	`expr $b / $a` will give 2
% (Modulus)	`expr $b % $a` will give 0
= (Assignment)	a = $b would assign value of b into a
== (Equality)	[$a == $b] would return false.
!= (Not Equality)	[$a != $b] would return true.

It is very important to understand that all the conditional expressions should be inside square braces with spaces around them, for example **[$a == $b]** is correct whereas, **[$a==$b]** is incorrect.

```
#!/bin/sh

a=10
b=20

val=`expr $a + $b`
echo "a + b : $val"

val=`expr $a - $b`
echo "a - b : $val"

val=`expr $a \* $b`
echo "a * b : $val"

val=`expr $b / $a`
echo "b / a : $val"

val=`expr $b % $a`
echo "b % a : $val"
```

The above script will produce the following result −

```
a + b : 30
a - b : -10
a * b : 200
b / a : 2
b % a : 0
```

Shell Decision Making

- Unix Shell supports conditional statements which are used to perform different actions based on different conditions. We will now understand two decision-making statements here −

- The **if...else** statement
- The **case...esac** statement

The if...else statements

- If else statements are useful decision-making statements which can be used to select an option from a given set of options.

- Unix Shell supports following forms of if...else statement –

- if...fi statement
- if...else...fi statement
- if...elif...else...fi statement
- nested if-else

- The syntax will be –

if-fi

```
if [ expression ]; then

    statements

fi
```

Example of if-fi

```
Name="Satyajit"
if [ "$Name" = "Satyajit" ]; then
  echo "His name is Satyajit. It is true."
fi
```

Output

```
His name is Satyajit. It is true.
```

if-else-fi

```
if [ expression ]
then
   statement1
else
   statement2
fi
```

Example of if-else-fi

```
Age=17
if [ "$Age" -ge 18 ]; then
    echo "You can vote"
else
    echo "You cannot vote"
fi
```

Output

```
You cannot vote
```

if-elif-else-fi

```
if [ expression1 ]
then
   statement1
   statement2
   .
   .
elif [ expression2 ]
then
   statement3
   statement4
   .
   .
else
   statement5
fi
```

Example of if-elif-else-fi

```
Age=17
if [ "$Age" -ge 18 ]; then
    echo "You can vote"
elif [ "$Age" -eq 17 ]; then
    echo "You can vote after one year"
else
    echo "You cannot vote"
fi
```

Output

```
You can vote after one year
```

nested if-else

```
if [ expression ]
then
   statement1
   if [ expression ]
   then
     statement
   else
     statement
   fi
else
   statement2
fi
```

Example of Nested if-else

```
echo "Enter subject"
read subject

if [ $subject == 'Linux' ]
then
echo "Enter Marks"
read marks
        if [ $marks -ge 30 ]
        then
        echo "You passed"
        else
        echo "You failed"
        fi
else
echo "Wrong Subject"
fi
```

Output 1

```
Enter subject
Linux
Enter Marks
97
You passed
```

Output 2

```
Enter subject
Linux
Enter Marks
29
You failed
```

Output 3

```
Enter subject
DBMS
Wrong Subject
```

Making a file-based decision

- The file-based decision is basically a type of decision-making based on whether the file exists or not.
- A use case of this will be to check for available file permission or to create a file if it is not available etc.
- A minimal structure of such script can be written as –

```
if [ -e gfg.sh ]
then
    echo "file exists"
else
    echo "file does not exist"
fi
```

Example of file-based decision

```
echo "Enter filename"
read filename

if [ -e $filename ]
then
echo "$filename is exits on the directory"
cat $filename

else
    cat > $filename
    echo "File created"
fi
```

Output :

First time:

```
Enter filename
geeks.txt
Hello Geek
File created
```

Second time:

```
Enter filename
geeks.txt
geeks.txt is exits on the directory
Hello Geek
```

String-based Condition

- The string-based condition means in the shell scripting we can take decisions by doing comparisons within strings as well. Here is a descriptive table with all the operators –

Operator	Description
==	Returns true if the strings are equal
!=	Returns true if the strings are not equal
-n	Returns true if the string to be tested is not null
-z	Returns true if the string to be tested is null

```
# ==
if [ 'Geeks' == 'Geeks' ];
then
    echo "same" #output
else
    echo "not same"
fi

# !=
if [ 'Geeks' != 'Apple' ];
then
    echo "not same" #output
else
    echo "same"
fi
```

```
# -n
if [ -n "Geeks" ];
then
    echo "not null" #output
else
    echo "null"
fi

# -z
if [ -z "Geeks" ];
then
    echo "null"
else
    echo "not null" #output
fi
```

Output:

```
same
not same
not null
not null
```

Arithmetic-based Condition

- Arithmetic operators are used for checking the arithmetic-based conditions. Like less than, greater than, equals to, etc. Here is a descriptive perators –

Operator	Description
-eq	Equal
-ge	Greater Than or Equal
-gt	Greater Than
-le	Less Than or Equal
-lt	Less Than
-ne	Not Equal

```
# -eq
if [ 10 -eq 10 ];then
echo "Equal"
fi

# -ge
if [ 10 -ge 9 ];then
echo "Greater or equal"
fi

# -gt
if [ 10 -gt 8 ];then
echo "Greater"
fi
```

```
# -le
if [ 10 -le 12 ];then
echo "Less than or equal"
fi

# -lt
if [ 10 -lt 13 ];then
echo "Less than"
fi

# -ne
if [ 10 -ne 13 ];then
echo "Not Equal"
fi
```

Output:

```
Equal
Greater or equal
Greater
Less than or equal
Less than
Not Equal
```

2. The case-sac statement

- case-sac is basically working the same as switch statement in programming.
- Sometimes if we have to check multiple conditions, then it may get complicated using if statements. At those moments we can use a case-sac statement. The syntax will be –

```
case $var in
   Pattern 1) Statement 1;;
   Pattern n) Statement n;;
esac
```

Example of case-sac statement

```
Name="Satyajit"
case "$Name" in
    #case 1
    "Rajib") echo "Profession : Software Engineer" ;;

    #case 2
    "Vikas") echo "Profession : Web Developer" ;;

    #case 3
    "Satyajit") echo "Profession : Technical Content Writer"
;;
esac
```

Output

```
Profession : Technical Content Writer
```

Shell Boolean Operators:

- Assume variable a holds 10 and variable b holds 20 then –

Operator	Description	Example
!	This is logical negation. This inverts a true condition into false and vice versa.	[! false] is true.
-o	This is logical **OR**. If one of the operands is true, then the condition becomes true.	[$a -lt 20 -o $b -gt 100] is true.
-a	This is logical **AND**. If both the operands are true, then the condition becomes true otherwise false.	[$a -lt 20 -a $b -gt 100] is false.

```
a=10
b=20

if [ $a != $b ]
then
   echo "$a != $b : a is not equal to b"
else
   echo "$a != $b: a is equal to b"
fi

if [ $a -lt 100 -a $b -gt 15 ]
then
   echo "$a -lt 100 -a $b -gt 15 : returns true"
else
   echo "$a -lt 100 -a $b -gt 15 : returns false"
fi
```

```
if [ $a -lt 100 -o $b -gt 100 ]
then
   echo "$a -lt 100 -o $b -gt 100 : returns true"
else
   echo "$a -lt 100 -o $b -gt 100 : returns false"
fi

if [ $a -lt 5 -o $b -gt 100 ]
then
   echo "$a -lt 100 -o $b -gt 100 : returns true"
else
   echo "$a -lt 100 -o $b -gt 100 : returns false"
fi
```

The above script will generate the following result −

```
10 != 20 : a is not equal to b
10 -lt 100 -a 20 -gt 15 : returns true
10 -lt 100 -o 20 -gt 100 : returns true
10 -lt 5 -o 20 -gt 100 : returns false
```

System calls

- System calls provide an interface to the services made available by an operating system.
- The system calls fork(), vfork(), wait(), and execl() are all used to create and manipulate processes.

fork()

- Processes execute the fork() system call to create a new child process.

- The process executing the fork() call is called a parent process. The child process created receives a unique Process Identifier (PID) but retains the parent’s PID as its Parent Process Identifier (PPID).

- The child process has identical data to its parent process. However, both processes have separate address spaces.
- After the creation of the child process, both the parent and child processes execute simultaneously. They execute the next step after the fork() system call.

- Since the parent and child processes have different address spaces, any modifications made to one process will not reflect on the other.

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/types.h>

int main(int argc, char **argv) {
    pid_t pid = fork();
    if (pid==0) {
        printf("This is the Child process and pid is: %d\n",getpid());
        exit(0);
    } else if (pid > 0) {
        printf("This is the Parent process and pid is: %d\n",getpid());
    } else {
        printf("Error while forking\n");
        exit(EXIT_FAILURE);
    }
    return 0;
}
```

vfork()

- vfork() also creates a child process that’s identical to its parent process.
- However, the child process temporarily suspends the parent process until it terminates.
- This is because both processes use the same address space, which contains the stack, stack pointer, and instruction pointer.
- The parent process is always suspended once the child process is created.
- It remains suspended until the child process terminates normally, abnormally, or until it executes the exec system call starting a new process.

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
    pid_t pid = vfork();  //creating the child process

    printf("parent process pid before if...else block: %d\n", getpid());

    if (pid == 0) {  //checking if this is the a child process
        printf("This is the child process and pid is: %d\n\n", getpid());
        exit(0);
    } else if (pid > 0) {  //parent process execution
        printf("This is the parent process and pid is: %d\n", getpid());
    } else {
        printf("Error while forking\n");
        exit(EXIT_FAILURE);
    }
    return 0;
}
```

wait()

- wait() system call suspends execution of current process until a child has exited or until a signal has delivered whose action is to terminate the current process or call signal handler.
- pid_t wait(int * status);
- The wait() system call suspends execution of the current process until one of its children terminates.
- The call wait(&status) is equivalent to:

- waitpid(-1, &status, 0);

- The waitpid() system call suspends execution of the current process until a child specified by pid argument has changed state.
- By default, waitpid() waits only for terminated children, but this behaviour is modifiable via the options argument, as described below.
- Tag Description
- < -1 meaning wait for any child process whose process group ID is equal to the absolute value of pid.
- -1 meaning wait for any child process.
- 0 meaning wait for any child process whose process group ID is equal to that of the calling process.
- > 0 meaning wait for the child whose process ID is equal to the value of pid.`

execl()

- **execl()** family of functions or sys calls replaces current process image with new process image.
- The *execl()* function replaces the current process image with a new process image specified by *path*.
- The new image is constructed from a regular, executable file called the new process image file.
- No return is made because the calling process image is replaced by the new process image.