**Operating System**

**LAB-3**

**Name \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Roll No \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Date \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_­\_**

**Marks Obtained \_\_\_\_\_\_\_\_\_\_\_\_**

**Signature\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

UNIX/LINUX Shell programming

Variable, Special Variable.

**Objective: Understand and implement Variable, Special Variable in Linux 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.

The following examples are vUniversityd variable names −

\_UNIVERSITY

TOKEN\_A

VAR\_1

VAR\_2

Following are the examples of invUniversityd 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.

## **Defining Variables**

Variables are defined as follows −

variable\_name=variable\_value

For example:

NAME="IU\_stuudent"

Above example defines the variable NAME and assigns it the value "IU\_Student". Variables of this type are called scalar variables. A scalar variable can hold only one value at a time.

The shell enables you to store any value you want in a variable. For example −

VAR1="IU\_Student"

VAR2=100

## **Accessing Values**

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

For example, following script would access the value of defined variable NAME and would print it on STDOUT −

#!/bin/sh

NAME="IU\_Student"

echo $NAME

This would produce following value −

IU\_Student

## **Read-only Variables**

The shell provides a way to mark variables as read-only by using the readonly command. After a variable is marked read-only, its value cannot be changed.

For example, following script would give error while trying to change the value of NAME −

#!/bin/sh

NAME="IU\_Student"

readonly NAME

NAME="New\_student"

This would produce following result −

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

## **Unsetting Variables**

Unsetting or deleting a variable tells the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you would not be able to access stored value in the variable.

Following is the syntax to unset a defined variable using the **unset** command −

unset variable\_name

Above command would unset the value of a defined variable. Here is a simple example −

#!/bin/sh

NAME="IU\_Student"

unset NAME

echo $NAME

Above example would not print anything. You cannot use the unset command to**unset** variables that are marked **readonly**.

## **Variable Types**

When a shell is running, three main types of variables are present −

* **Local Variables** − A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at command prompt.
* **Environment Variables** − An environment variable is a variable that is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually a shell script defines only those environment variables that are needed by the programs that it runs.
* **Shell Variables** − A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.
* The following table shows a number of special variables that you can use in your shell scripts

|  |  |
| --- | --- |
| **Variable** | **Description** |
| **$0** | The filename of the current script. |
| **$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 "Second Parameter : $2"

echo "Quoted Values: $@"

echo "Quoted Values: $\*"

echo "Total Number of Parameters : $#"

Here is a sample run for the above script −

$./test.sh IU Student

FileName:./test.sh

FirstParameter: IU

SecondParameter:Student

QuotedValues:IU Student

QuotedValues:IU Student

TotalNumber of Parameters:2

.

## **Special Parameters $\* 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 −

#!/bin/sh

for TOKEN in $\*

do

echo $TOKEN

done

There is one sample run for the above script −

$./test.sh IU Student17YearsOld

IU

Student

17

Years

Old

**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 −

$./test.sh IU Student

FileName:./test.sh

FirstParameter:IU

SecondParameter:Student

QuotedValues:IU Student

QuotedValues:IU Student

TotalNumber of Parameters:2

$echo $?

0

$

**Task: Use following variables to write your CV in Linux bash script.**

|  |  |  |
| --- | --- | --- |
| **Variable** | **Script** | **Output** |
| **$0** |  |  |
| **$n** |  |  |
| **$#** |  |  |
| **$\*** |  |  |
| **$@** |  |  |
| **$?** |  |  |
| **$$** |  |  |
| **$!** |  |  |