BSCS03: Operating Systems

Rakesh Yadav
Associate Professor
Department of Computer Science
Shivaji College, University of Delhi
rakeshyadav@shivaji.du.ac.in

UNIT VI

- **Shell scripting:**
- Shell variables
- Parameter passing
- Conditional statements
- Iterative statements
- Writing and executing shell scripts
- Utility programs

What is Shell?

- The shell provides you with an interface to the UNIX system.
- It gathers input from you and executes programs based on that input.
- When a program finishes executing, it displays that program's output.
- A shell is an environment in which we can run our commands, programs, and shell scripts.
- Shell Prompt:
- The prompt, $, which is called command prompt, is issued by the shell.
- While the prompt is displayed, you can type a command.
- The shell reads your input after you press Enter.
- It determines the command you want executed by looking at the first word of your input.
- A word is an unbroken set of characters.
- Spaces and tabs separate words.

Shell Types:

- In UNIX there are two major types of shells:
- 1. The Bourne shell. If you are using a Bourne-type shell, the default prompt is the $ character.
- 2. The C shell. If you are using a C-type shell, the default prompt is the % character.
- There are again various subcategories for Bourne Shell which are listed as follows:
- Bourne shell (sh)
- Korn shell (ksh)
- Bourne Again shell (bash)
- POSIX shell (sh)

- The different C-type shells follow:
- C shell (csh)
- TENEX/TOPS C shell (tcsh)
- The original UNIX shell was written in the mid-1970s by Stephen R. Bourne while he was at AT&T Bell Labs in New Jersey.
- The Bourne shell was the first shell to appear on UNIX systems, thus it is referred to as "the shell".
- The Bourne shell is usually installed as /bin/sh on most versions of UNIX.
- It is the shell of choice for writing scripts to use on several different versions of UNIX.

Shell Scripts:

- The basic concept of a shell script is a list of commands, which are listed in the order of execution.
- 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 and functions are both interpreted. This means they are not compiled.

Example Script:

- we create a test.sh script.
- Note all the scripts would have **.sh** extension.
- Before you 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

Shell Comments

- #!/bin/bash
- # Author : Rakesh Yadav
- # Copyright (c) rakeshyadav@Shivaji.du.ac.in
- # Script follows here:
- pwd
- ls

- Now you save the above content and make this script executable as follows:
- $chmod +x test.sh
- Now you have your shell script ready to be executed as follows:
- $./test.sh
- **Note:** To execute your any program available in current directory you would execute using **./program_name**

Second Method

- To create shell script: vi filename.sh
- Press Esc key then
- Write : then
- :wq for save a script
- Run the script file : sh filename.sh
- If you want to change/update the script : go to command prompt $ type vi filename.sh press enter.

Extended Shell Scripts:

- Shell scripts have several required constructs that tell the shell environment what to do and when to do it.
- The shell is, after all, a real programming language, complete with variables, control structures, and so forth.
- No matter how complicated a script gets, however, it is still just a list of commands executed sequentially.

- echo "What is your name?"
- read a
- echo "Hello, $a"
- *Here script use the **read** command which takes the input from the keyboard and assigns it as the value of the variable a and finally prints it on STDOUT.*

Unix-Using 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.
- 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 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

Defining Variables:

- Variables are defined as follows:
- variable_name=variable_value
- NAME=“Jannat"
- Here the variable NAME and assigns it the value “Jannat".
- 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=“Jannat"
- 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:
- NAME=“Jannat"
- echo $NAME

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:
- NAME=“Jannat"
- readonly NAME
- NAME="Qadiri"
- 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=“jannat"
- unset NAME
- echo $NAME
- Here 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.

Unix-Special Variables

- Using certain non-alphanumeric characters in your variable names.
- This is because those characters are used in the names of special Unix variables.
- These variables are reserved for specific functions.
- For example, the $ character represents the process ID number, or PID, of the current shell:
- $echo $$
- This command would write PID of the current shell: 602

The 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 : $#"

- $ sh commandlineagrument.sh
- File Name: commandlineagrument.sh
- First Parameter :
- First Parameter :
- Quoted Values:
- Quoted Values:
- Total Number of Parameters : 0

- Here is a sample run for the above script:
- $./test.sh Zara Ali
- File Name : ./test.sh
- First Parameter : Zara
- First Parameter : Ali
- Quoted Values: Zara Ali
- Quoted Values: Zara Ali
- Total Number of Parameters : 2

Unix -Basic Operators

- There are various operators supported by each shell (Bourne).
- There are following operators which we are going to discuss:
- Arithmetic Operators.
- Relational Operators.
- Boolean Operators.
- String Operators.
- File Test Operators.
- The Bourne shell didn't originally have any mechanism to perform simple arithmetic but it uses external programs, either **awk** or the must simpler program **expr**.

The command expr

- Here is simple example to add two numbers:
- #!/bin/sh
- val=`expr 2 + 2`
- echo "Total value : $val“
- There are following points to note down:
- There must be spaces between operators and expressions for example 2+2 is not correct, where as it should be written as 2 + 2.
- Complete expression should be enclosed between ``, called inverted commas.
- The command expr which is capable of evaluating an arithmetic expression

Arithmetic Operators:

- + Addition - Adds values on either side of the operator `expr $a + $b`
- - Subtraction - Subtracts right hand operand from left hand operand `expr $a - $b`
- * Multiplication - Multiplies values on either side of the operator `expr $a \* $b`
- / Division - Divides left hand operand by right hand operand `expr $b / $a`
- % Modulus - Divides left hand operand by right hand operand and returns remainder `expr $b % $a` will give 0
- = Assignment - Assign right operand in left operand a=$b would assign value of b into a
- == Equality - Compares two numbers, if both are same then returns true. [$a == $b] would return false.
- != Not Equality - Compares two numbers, if both are different then returns true. [$a != $b] would return true.
- It is very important to note here that all the conditional expressions would be put inside square braces with one spaces around them, for example [$a == $b] is correct where as [$a==$b] is incorrect.

- There are following points to note down:
- There must be spaces between operators and expressions for example 2+2 is not correct, where as it should be written as 2 + 2.
- Complete expression should be enclosed between ``, called inverted commas.
- We should use \ on the * symbol for multiplication.

Relational Operators:

- Bourne Shell supports following relational operators which are specific to numeric values.
- These operators would not work for string values unless their value is numeric.
- For example, following operators would work to check a relation between 10 and 20 as well as in between "10" and "20" but not in between "ten" and "twenty"

Relational Operators

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

Operator	Description	Example
-eq	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a -eq $b] is not true.
-ne	Checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true.	[$a -ne $b] is true.
-gt	Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true.	[$a -gt $b] is not true.
-lt	Checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true.	[$a -lt $b] is true.
-ge	Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -ge $b] is not true.
-le	Checks if the value of left operand is less than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -le $b] is true.

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

Boolean Operators

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.

String Operators

Assume variable **a** holds "abc" and variable **b** holds "efg" then

Operator	Description	Example
=	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a = $b] is not true.
!=	Checks if the value of two operands are equal or not; if values are not equal then the condition becomes true.	[$a != $b] is true.
-z	Checks if the given string operand size is zero; if it is zero length, then it returns true.	[-z $a] is not true.
-n	Checks if the given string operand size is non-zero; if it is nonzero length, then it returns true.	[-n $a] is not false.
str	Checks if **str** is not the empty string; if it is empty, then it returns false.	[$a] is not false.

File Test Operators

- We have a few operators that can be used to test various properties associated with a Unix file.
- Assume a variable **file** holds an existing file name "test" the size of which is 100 bytes and has **read**, **write** and **execute** permission on

File Test Operators

Operator	Description	Example
-b file	Checks if file is a block special file; if yes, then the condition becomes true.	[-b $file] is false.
-c file	Checks if file is a character special file; if yes, then the condition becomes true.	[-c $file] is false.
-d file	Checks if file is a directory; if yes, then the condition becomes true.	[-d $file] is not true.
-f file	Checks if file is an ordinary file as opposed to a directory or special file; if yes, then the condition becomes true.	[-f $file] is true.
-g file	Checks if file has its set group ID (SGID) bit set; if yes, then the condition becomes true.	[-g $file] is false.
-k file	Checks if file has its sticky bit set; if yes, then the condition becomes true.	[-k $file] is false.
-p file	Checks if file is a named pipe; if yes, then the condition becomes true.	[-p $file] is false.
-t file	Checks if file descriptor is open and associated with a terminal; if yes, then the condition becomes true.	[-t $file] is false.

File Test Operators

Operator	Description	Example
-u file	Checks if file has its Set User ID (SUID) bit set; if yes, then the condition becomes true.	[-u $file] is false.
-r file	Checks if file is readable; if yes, then the condition becomes true.	[-r $file] is true.
-w file	Checks if file is writable; if yes, then the condition becomes true.	[-w $file] is true.
-x file	Checks if file is executable; if yes, then the condition becomes true.	[-x $file] is true.
-s file	Checks if file has size greater than 0; if yes, then condition becomes true.	[-s $file] is true.
-e file	Checks if file exists; is true even if file is a directory but exists.	[-e $file] is true.

Unix –Decision Making

- Unix Shell supports conditional statements which are used to perform different actions based on different conditions.
- There are two types of decision-making statements
- 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

The case...esac Statement

- You can use multiple **if...elif** statements to perform a multiway branch. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.
- Unix Shell supports **case...esac** statement which handles exactly this situation, and it does so more efficiently than repeated **if...elif** statements.
- There is only one form of **case...esac** statement
- case...esac statement
- The **case...esac** statement in the Unix shell is very similar to the **switch...case** statement we have in other programming languages like **C** or **C++** and **PERL**, etc.

if...fi statement

- a=10
- b=20
- if [$a == $b]
- then
- echo "a is equal to b"
- fi
- if [$a != $b]
- then
- echo "a is not equal to b"
- fi

if...else...fi statement

- a=10
- b=20
- if [$a == $b]
- then
- echo "a is equal to b"
- else
- echo "a is not equal to b"
- fi

if...elif...else...fi statement

- The **if...elif...fi** statement is the one level advance form of control statement that allows Shell to make correct decision out of several conditions.
- if [expression 1]
- then
- Statement(s) to be executed if expression 1 is true
- elif [expression 2]
- then
- Statement(s) to be executed if expression 2 is true
- elif [expression 3]
- then
- Statement(s) to be executed if expression 3 is true
- else
- Statement(s) to be executed if no expression is true
- fi

Example:

- a=10
- b=20
- if [$a == $b]
- then
- echo "a is equal to b"
- elif [$a -gt $b]
- then
- echo "a is greater than b"
- elif [$a -lt $b]
- then
- echo "a is less than b"
- else
- echo "None of the condition met"
- fi

The case...esac Statement:

- case word in
- pattern1)
- Statement(s) to be executed if pattern1 matches
- ;;
- pattern2)
- Statement(s) to be executed if pattern2 matches
- ;;
- pattern3)
- Statement(s) to be executed if pattern3 matches
- ;;
- esac

Example

- FRUIT="kiwi"
- case "$FRUIT" in
- "apple") echo "Apple pie is quite tasty."
- ;;
- "banana") echo "I like banana nut bread."
- ;;
- "kiwi") echo "New Zealand is famous for kiwi."
- ;;
- esac

Write a shell script to print days of a week

- **echo "enter a number"**
  read n
  case $n in
  1) echo "Sunday" ;;
  2) echo "Monday" ;;
  3) echo "Tuesday" ;;
  4) echo "Wednesday" ;;
  5) echo "Thursday" ;;
  6) echo "Friday" ;;
  7) echo "Saturday" ;;
  ***) echo "enter value between 1 to 7" ;;**
  esac
- *) acts as default and it is executed if no match is found.

Unix –Shell Loops

- Loops are a powerful programming tool that enable you to execute a set of commands repeatedly.
- There are total 4 looping statements which can be used in bash programming
  1) while statement
  2) for statement
  3) until statement
  4) select statement
- To alter the flow of loop statements, two commands are used they are,
- break
- continue

The while loop

- The while loop enables you to execute a set of commands repeatedly until some condition occurs.
- It is usually used when you need to manipulate the value of a variable repeatedly.
- Syntax:-
  - while command
  - do
  - Statement(s) to be executed if command is true
  - done

The while loop to display the numbers zero to nine:

- a=0
- while [$a -lt 10]
- do
- echo $a
- a=`expr $a + 1`
- done

The for loop

- Syntax:
  - for var in word1 word2 ... Word n / we can also give range { 1..100}
  - do
  - Statement(s) to be executed for every word.
  - done
- for var in 0 1 2 3 4 5 6 7 8 9
- do
- echo $var
- done

Example (vi for.sh)

- i=1
- for day in Mon Tue Wed Thu Fri
- do
- echo "Weekday $((i++)) : $day"
- done

Implementing for loop with break statement

```
for a in 1 2 3 4 5 6 7 8 9 10
do
    # if a is equal to 5 break the loop
    if [ $a == 5 ]
    then
        break
    fi
    # Print the value
    echo "Iteration no $a"
done
```

Implementing for loop with continue statement

```
for a in 1 2 3 4 5 6 7 8 9 10
do
    # if a = 5 then continue the loop and
    # don't move to line 8
    if [ $a == 5 ]
    then
        continue
    fi
    echo "Iteration no $a"
done
```

The until loop

- The until loop is executed as many as times the condition/command evaluates to false.
- The loop terminates when the condition/command becomes true.
- Syntax:-
  - until command
  - do

    Statement to be executed until command is true
  - done

Implementing until loop

- a=0
- until [$a -gt 10]
- do
- echo $a
- a=`expr $a + 1`
- done

The select loop

- The *select* loop provides an easy way to create a numbered menu from which users can select options.
- It is useful when you need to ask the user to choose one or more items from a list of choices.
- Syntax:-
  - select var in word1 word2 ... wordN
  - do
  - Statement(s) to be executed for every word.
  - done

example to let the user select a drink of choice:

- select DRINK in tea cofee water juice appe all none
- do
- case $DRINK in
- tea|cofee|water|all)
- echo "Go to canteen"
- ;;
- juice|appe)
- echo "Available at home"
- ;;
- none)
- break
- ;;
- *) echo "ERROR: Invalid selection"
- ;;
- esac
- done

We can change the prompt displayed by the select loop by altering the variable PS3 as:- PS3="Please make a selection => " ; export PS3

Unix –Using Arrays

- A shell variable is capable enough to hold a single value.
- This type of variables are called scalar variables.
- Shell supports a different type of variable called an array variable that can hold multiple values at the same time.
- Arrays provide a method of grouping a set of variables.
- Instead of creating a new name for each variable that is required, you can use a single array variable that stores all the other variables.

Defining Array Values:

- The difference between an array variable and a scalar variable
- Say that you are trying to represent the names of various students as a set of variables.
- Each of the individual variables is a scalar variable as follows:
- NAME01=“Atul"
- NAME02=“Vrinda"
- NAME03=“Rahul"
- NAME04=“Kirti Yadav"
- NAME05=“Gaurav"

- We can use a single array to store all the above mentioned names. Following is the simplest method of creating an array variable is to assign a value to one of its indices. This is expressed as follows:
- array_name[index]=value
- NAME[0]=“Atul"
- NAME[1]=“Vrinda"
- NAME[2]=“Rahul"
- NAME[3]=“Kirti Yadav"
- NAME[4]=“Gaurav"

Accessing Array Values:

- After you have set any array variable, you access it as follows:
- ${array_name[index]}
- NAME[0]=“Gaurav"
- NAME[1]=“Tushar Kalra"
- NAME[2]=“Prabhat Berwal"
- NAME[3]=“Ankit Joshi"
- NAME[4]=“Jannat"
- echo "First Index: ${NAME[0]}"
- echo "Second Index: ${NAME[1]}"

- We can access all the items in an array in one of the following ways:
- ${array_name[*]}
- ${array_name[@]}

Unix –Shell Functions

- Functions enable you to break down the overall functionality of a script into smaller, logical subsections, which can then be called upon to perform their individual task when it is needed.
- Using functions to perform repetitive tasks is an excellent way to create code reuse. Code reuse is an important part of modern object-oriented programming principles.
- Shell functions are similar to subroutines, procedures, and functions in other programming languages.

Creating Functions:

- To declare a function:
- function_name () {
- list of commands
- }
- The name of your function is function_name, and that's what you will use to call it from elsewhere in your scripts.
- The function name must be followed by parentheses, which are followed by a list of commands enclosed within braces.

Example:

- # Define your function here
- Hello () {
- echo "Hello World"
- }
- # Invoke your function
- Hello

Pass Parameters to a Function:

- We can define a function which would accept parameters while calling those function.
- These parameters would be represented by $1, $2 and so on.
- Following is an example where we pass two parameters *Sachin* and *Tendulkar* and then we capture and print these parameters in the function.
- # Define your function here
- Hello () {
- echo "Hello World $1 $2"
- }
- # Invoke your function
- Hello Sachin Tendulkar

Returning Values from Functions:

- If you execute an exit command from inside a function, its effect is not only to terminate execution of the function but also of the shell program that called the function.
- If you instead want to just terminate execution of the function, then there is way to come out of a defined function.
- Based on the situation you can return any value from your function using the **return** command whose syntax is as follows:
- return code
- Here *code* can be anything you choose here, but obviously you should choose something that is meaningful or useful in the context of your script as a whole.

Example:

- # Define your function here
- Hello () {
- echo "Hello World $1 $2"
- return 10
- }
- # Invoke your function
- Hello Sachin Tendulkar
- # Capture value returnd by last command
- ret=$?
- echo "Return value is $ret"

Nested Functions:

- One of the more interesting features of functions is that they can call themselves as well as call other functions.
- A function that calls itself is known as a *recursive function*.
- # Calling one function from another
- number_one () {
- echo "This is the first function speaking..."
- number_two
- }
- number_two () {
- echo "This is now the second function speaking..."
- }
- # Calling function one.
- number_one

4. Write a shell script to check if the number entered at the command line is prime or not.

- **ALGORITHM**
- Step 1 – Loop from 2 to n/2, i as loop variable.
- Step 2 – **if number** is divisible, print “The **number** is **not prime**” and flag = 1;
- Step 3 – **if** flag != 1, then print “The **number** is **prime**”.
- Step 4 – Exit.

References

- Unix: Concepts and Applications, Sumitabha Das, TMH, 4th Edition, 2009.
- https://www.geeksforgeeks.org.
- https://www.tutorialspoint.com.