Intentionally left blank for writing solutions to the provided lab exercises.

2 Experiment 2: Basics of Shell Scripting

2.1 Objective

The primary objectives include creating simple yet functional scripts, grasping scripting syntax elements such as variables, loops, and control structures, and executing these scripts effectively. Moreover, this experiment will emphasize the real-world utility of shell scripting in automating routine tasks, performing file manipulations, and simplifying system administration processes.

2.2 Reference Material[1][3][2]

2.2.1 Shell

In computing, a shell refers to a user interface that allows users to interact with an operating system (such as Linux, Unix, or Windows) to execute commands, run programs, and manage files and directories. It acts as an intermediary between the user and the core functionalities of the operating system.

2.2.2 Shell script

A shell program, also known as a shell script, is like a recipe made up of step-by-step instructions written in a special language that the computer understands. Each instruction in this recipe tells the computer what to do. These scripts are saved as files with a .sh ending and are created using programs like vi. To use them, we have to give permission to the computer to run these script files. We do this by using a command called chmod. Then, in a terminal, we use commands like sh or bash to tell the computer to follow the instructions written in the script file. It's like giving the computer a set of tasks to perform, and it does them one after another.

2.2.3 Shell Variables

Shell variables are placeholders used by a shell (like Bash or PowerShell) to store information or data values. These variables act as containers to hold data temporarily, such as strings of text, numbers, file paths, or configuration settings, for example: $VAR\_NAME = value$.

2.2.4 Types of Variables

Some commonly used shell variables include:

(i) Environment Variables: These are variables that contain information about the environment in which the shell operates, such as user settings, system paths, or configuration preferences.

(ii) User-Defined Variables: These are variables created by users to store custom data or information required for specific tasks or scripts.

2.2.5 Important Environment Variables

- `PATH`: Specifies directories where executable programs are located.
- `HOME`: Represents the current user's home directory.
- `USER`: Displays the username of the current user.
- `SHELL`: Specifies the default shell for the user.
- `PWD`: Indicates the present working directory.
- `LANG`: Determines the language and localization settings.
- `TERM`: Defines the terminal type or emulator being used.
- `EDITOR`: Specifies the default text editor.

LOGNAME , SHLVL , HISTSIZE

2.2.6 Shell Redirections

Shell redirections in Linux/Unix allow users to control input and output streams of commands. Here are common redirection symbols:

- **Standard Input Redirection (<)**: Changes the command's input source to a file.

```
command < input_file.txt
```

- **Standard Output Redirection (>)**: Redirects command output to a file (overwrites existing content).

```
command > output_file.txt
```

- **Appending Output (>>)**: Appends command output to a file.

```
command >> output_file.txt
```

- **Piping Output (|)**: Redirects output of one command as input to another.

```
command1 | command2
```

- **Standard Error Redirection (2> or 2>>)**: Redirects error messages to a file.

```
command 2> error_file.txt
```

2.2.7 Shell Arithmetic using expr and bc Commands

In shell scripting, arithmetic operations can be performed using different commands:

- **expr Command:** Used for integer arithmetic operations within shell scripts. It evaluates and prints the result of expressions.

  Example usage:

  ```
  result=$(expr 5 + 3)    # Adds 5 and 3
  echo "Result: $result" # Output: Result: 8
  ```

  Supported operators in `expr` include addition (+), subtraction (-), multiplication (*), division (/), modulus (

- **bc Command:** Stands for 'Basic Calculator' and supports floating-point arithmetic and advanced mathematical functions.

  Example usage:

  ```
  result=$(echo "5.5 + 3.2" | bc) # Adds 5.5 and 3.2
  echo "Result: $result"          # Output: Result: 8.7
  ```

  `bc` handles floating-point arithmetic and provides functions like sine, cosine, square root, etc., for more complex calculations.

These commands offer basic arithmetic functionalities within shell scripts. `expr` is suitable for simple integer arithmetic, while `bc` provides a broader range of mathematical operations and supports floating-point numbers.

2.2.8 Flow Control in Shell Scripting using if Statements

Shell scripting offers various forms of `if` statements for conditional flow control:

1. **Basic if Statement:**

   ```
   if [ condition ]; then
       # Commands to execute if the condition is true
   fi
   ```

2. **if-else Statement:**

   ```
   if [ condition ]; then
       # Commands to execute if the condition is true
   else
       # Commands to execute if the condition is false
   fi
   ```

3. `if-elif-else` Statement:

```
if [ condition1 ]; then
    # Commands to execute if condition1 is true
elif [ condition2 ]; then
    # Commands to execute if condition2 is true
else
    # Commands to execute if both condition1 and condition2 are false
fi
```

4. Nested `if` Statements:

```
if [ condition1 ]; then
    if [ condition2 ]; then
        # Commands to execute if both condition1 and condition2 are true
    fi
fi
```

5. `if` Statement with Logical Operators:

```
if [ condition1 -a condition2 ]; then
    # Commands to execute if condition1 AND condition2 are true
fi

if [ condition1 -o condition2 ]; then
    # Commands to execute if condition1 OR condition2 is true
fi
```

These variations enable conditional execution of commands based on different conditions within shell scripts, offering flexibility in controlling the flow of the script.

2.2.9 Operators in Shell Scripting

In shell scripting, operators are used to perform various operations on variables, constants, and expressions. Here are the common types of operators:

1. Arithmetic Operators
   - `+`, `-`, `*`, `/`, `%`: Perform addition, subtraction, multiplication, division, and modulus respectively.
2. Relational Operators
   - `-eq`, `-ne`, `-gt`, `-lt`, `-ge`, `-le`: Compare numbers (equal, not equal, greater than, less than, greater than or equal to, less than or equal to) within `test` or `[ ]` brackets.

3. String Operators

- `=`, `!=`, `<`, `>`: Compare strings (equal, not equal, less than, greater than) within `test` or `[ ]` brackets.

4. Logical Operators

- `&&`, `||`, `!`: Perform logical AND, logical OR, and logical NOT operations respectively.

5. Assignment Operators

- `=`, `+=`: Assign values to variables or concatenate strings.

6. Bitwise Operators

- `&`, `|`, `^`, `<<`, `>>`, `~`: Perform bitwise AND, OR, XOR, left shift, right shift, and bitwise NOT operations respectively.

7. File Test Operators

- `-f`, `-d`, `-r`, `-w`, `-x`: Check file properties (existence, directory, readability, writability, executability) within `test` or `[ ]` brackets.

Understanding and utilizing these operators enable the creation of conditions, calculations, string manipulations, and file property checks within shell scripts, enhancing their functionality and flexibility.

2.2.10 Case Structure in Shell Scripting

In shell scripting, the case structure provides a way to perform conditional branching based on the value of a variable or expression. It allows evaluation against multiple patterns and executes commands based on the matching pattern.

The basic syntax of the case structure is as follows:

```
case variable in
    pattern1)
        # Commands to execute if variable matches pattern1
        ;;
    pattern2)
        # Commands to execute if variable matches pattern2
        ;;
    pattern3|pattern4)
        # Commands to execute if variable matches pattern3 or pattern4
        ;;
    *)
        # Default commands to execute if no pattern matches
        ;;
esac
```

Example:

```
fruit="apple"

case $fruit in
    apple)
        echo "It's an apple."
        ;;
    banana|orange)
        echo "It's a banana or an orange."
        ;;
    *)
        echo "It's another fruit."
        ;;
esac
```

This example evaluates the variable `$fruit` against different patterns and executes respective blocks based on the matching pattern.

The `case` structure simplifies code readability when multiple conditions need evaluation against a single variable or expression.

2.2.11 Loops in Shell Scripting

In shell scripting, loops are used to execute a block of code repeatedly based on certain conditions. There are various types of loops available:

1. `for` Loop

The `for` loop iterates through a list of items or values. It is suitable when you have a known set of elements to loop through.

Syntax:

```
for variable in list
do
    # Commands to execute for each iteration
done
```

Example:

```
for i in 1 2 3 4 5
do
    echo "Iteration: $i"
done
```

2. `while` Loop

The `while` loop executes a block of code as long as a specified condition remains true.

Syntax:

```
while [ condition ]
do
    # Commands to execute as long as the condition is true
done
```

Example:

```
count=1
while [ $count -le 5 ]
do
    echo "Count: $count"
    ((count++))
done
```

3. until Loop

The `until` loop executes a block of code until a specified condition becomes true.

Syntax:

```
until [ condition ]
do
    # Commands to execute until the condition becomes true
done
```

Example:

```
count=1
until [ $count -gt 5 ]
do
    echo "Count: $count"
    ((count++))
done
```

4. Nested Loops

You can nest loops within each other to create more complex control structures.

Example:

```
for i in {1..3}
do
    echo "Outer Loop Iteration: $i"
    for j in A B C
    do
        echo "   Inner Loop Iteration: $j"
    done
done
```

These loops in shell scripting provide various ways to iterate through data, execute code repeatedly based on conditions, and control the flow of a script.

2.3 Steps to Prepare and Execute a Shell Script

1. **Create the Shell Script:**
   - Open a text editor or an Integrated Development Environment (IDE) to write the shell script.
   - Write the desired commands and save the file with a `.sh` extension (e.g., `script.sh`).

2. **Set Execution Permissions (if needed):**
   - If the script doesn't have execution permissions, use the `chmod` command to set the execution permission:

     ```
     chmod +x script.sh
     ```

3. **Run the Shell Script:**
   - Open the terminal.
   - Navigate to the directory where the script is saved using the `cd` command.
   - Execute the script using one of the following methods:
     - Using Bash:

       ```
       bash script.sh
       ```

     - Using Shorthand (if the script has execution permissions):

       ```
       ./script.sh
       ```

     - Using Absolute Path:

       ```
       /path/to/your/script.sh
       ```

4. **Verify Output:**
   - After executing the script, verify the output or actions performed by the script in the terminal or through any generated files or changes made by the script.

5. **Debug and Modify (if needed):**
   - If there are errors or the script doesn't behave as expected, edit the script in the text editor, save the changes, and rerun the script.