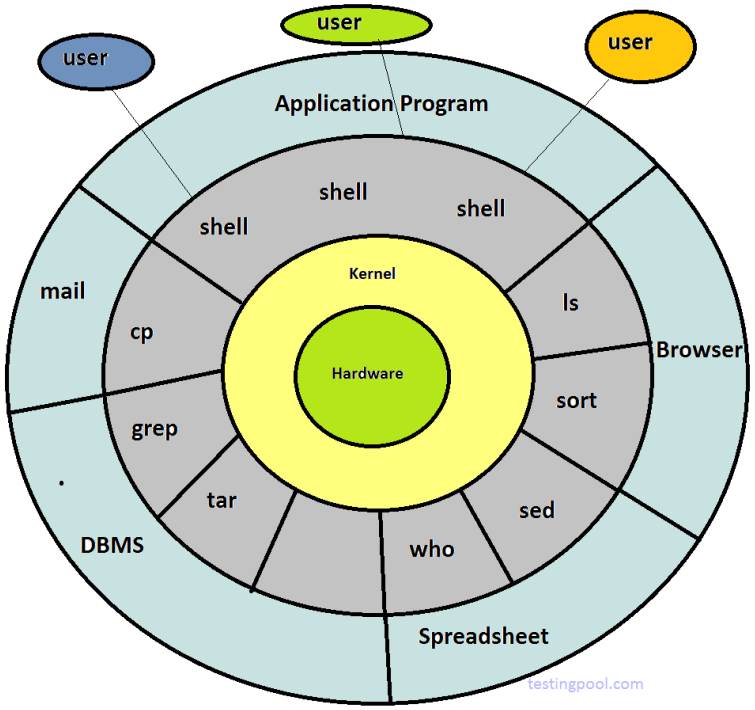
**Unix Operating System Structure:**




1. **Writing Shell Scripts:**

Basics of Shell Scripting: Learn the syntax and basic commands of shell scripting (e.g., #!/bin/bash, echo, sh).

Executing Scripts: Understand how to run a shell script using sh script_name.sh or ./script_name.sh.

**Basic Commands:**

**ls:** List. Lists files and directories in the current directory.

**cd:** Change Directory. Navigates to a different directory.

**mkdir:** Make Directory. Creates a new directory.

**rmdir:** Remove Directory. Deletes an empty directory.

**touch:** Creates a new empty file.

**cp:** Copy. Copies files or directories.

**mv:** Move. Moves or renames files or directories.

**rm:** Remove. Deletes files.

**cat:** Concatenate. Displays the content of a file.

**less:** Views file content one page at a time.

**head:** Displays the first few lines of a file.

**tail:** Displays the last few lines of a file.

**chmod:** Change Mode. Changes the file permissions.

**chown:** Change Owner. Changes the file owner.

**man:** Manual. Shows the manual for a command.

**wc:** Counts lines, words, and characters in a file.

**date:** Display the current date and time.

## 2. Conditional Statements:

Conditional statements in shell scripts allow you to make decisions based on certain conditions. Here are some examples of how to use conditional statements in shell scripts.

### Logical Operators

Logical operators such as && (AND) and || (OR) can be used within conditions.

**Using if Statements:** Learn how to use if, elif, and else for conditional execution in shell scripts.

### If-Else Statement

The basic syntax for an if-else statement is:

```
if [ condition ]
then
   # commands to execute if condition is true
else
   # commands to execute if condition is false
fi
```

### If-Elif-Else Statement

The if-elif-else statement allows you to check multiple conditions.

```
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 none of the conditions are true
fi
```

**Case Statement**

The case statement is used for pattern matching.

```
case expression in
    pattern1)
        # commands to execute if pattern1 matches
        ;;
    pattern2)
        # commands to execute if pattern2 matches
        ;;
    *)
        # commands to execute if no pattern matches
        ;;
esac
```

- **Check if a number is even or odd**

```
#!/bin/bash
# Read the number from the user
echo "Enter a number:"
read number
# Check if the number is even or odd
if [ $((number % 2)) -eq 0 ]; then
    echo "The number $number is even."
else
    echo "The number $number is odd."
fi
```

- **Compare three numbers and find the largest**

```
#!/bin/bash
# Read three numbers from the user
echo "Enter the first number:"
read num1
echo "Enter the second number:"
read num2
echo "Enter the third number:"
read num3
# Compare the numbers and find the largest
if [ "$num1" -ge "$num2" ] && [ "$num1" -ge "$num3" ]; then
   largest=$num1
elif [ "$num2" -ge "$num1" ] && [ "$num2" -ge "$num3" ]; then
   largest=$num2
else
   largest=$num3
fi
# Print the largest number
echo "The largest number is: $largest"
```

**Practice:**

- Write a shell script code to check a year is leap year or not
- Check if a number is within a range.
- Determine the grade based on marks
- Display the day of the week( Using switch case)
- Perform basic arithmetic operations (Using Switch case)
- Check a year is leap year or not
- Determine if a number is positive, negative, or zero:

3. **Looping Constructs:**

For Loops: Understand how to use for loops to iterate over files or a list of items.

While Loops: Understand how to use while loops for continuous execution until a condition is met.

- **Factorial of a given number:**

```
#!/bin/bash
# Read the number from the user
echo "Enter a number:"
read number
# Initialize the factorial variable
factorial=1
# Calculate the factorial using a loop
for (( i=1; i<=number; i++ ))
do
   factorial=$((factorial * i))
done
# Print the factorial
echo "The factorial of $number is: $factorial"
```

- **Sum of digit of a given number:**

```
#!/bin/bash
# Read the number from the user
echo "Enter a number:"
read number
# Initialize the sum variable
sum=0
# Extract digits and calculate the sum
while [ $number -gt 0 ]
do
   digit=$(( number % 10 ))  # Get the last digit
   sum=$(( sum + digit ))    # Add the digit to the sum
   number=$(( number / 10 )) # Remove the last digit
done
echo "The sum of the digits is: $sum"           # Print the sum of the digits
```

**1.** Write a shell script to check a given number is Krishnamurthy (145=1!+4!+5!) or not.

**2.** Series f(x,n)=1+x 2 /2!+x 4 /4!+….+x 2*n /(2*n)!

**Functions on shell script:**

To declare a function, simply use the following syntax −

```
function_name () {
   list of commands
}
```

**Pass Parameters to a Function**

You can define a function that will accept parameters while calling the function. These parameters would be represented by **$1**, **$2** and so on.

- **calculate power of given two number in shell script using function:**

```
#!/bin/bash
# Function to calculate power of a number
power() {
    base=$1
    exp=$2
    result=1
    for (( i=1; i<=exp; i++ ))
    do
        result=$((result * base))
    done
    echo $result
}

# Main script
echo "Enter the base number:"
read base
echo "Enter the exponent number:"
read exponent
```

```
# Call the function to calculate the power and store the result
result=$(power $base $exponent)
# Print the result
echo "$base raised to the power of $exponent is: $result"
```

**Practice for function:**

1. Calculate Factorial.
2. Calculate Series Sum

   1+ x^1 + x^2 + x^3 + …+ x^n
3. Calculate nCr Value.

**Command Line Arguments:**

Command-line arguments are parameters that are passed to a script while executing them in the bash shell. They are also known as positional parameters in Linux.

Passing Arguments: sh filename.sh arg1 arg2 arg3 ..…

sh filename.sh arg1 arg2 arg3 ..…

Bash saves these variables numerically ($1, $2, $3, … $n)

The special character **$#** stores the total number of arguments. We also have **$@** and **$*** as wildcard characters which are used to denote all the arguments.

**IFS (Internal Field Separator)**

In Bash, the **"IFS"** variable controls how fields in a string are separated. IFS defaults to a space, tab, and newline character, which means that, by default, fields in a string are separated by any combination of these characters. However, the IFS value can be changed. This helps us to handle various field delimited strings.

**Example:**

```
echo "Enter the date of birth(dd/mm/yyyy) "
```

```
read dob
IFS="/"
set $dob
dd=$1
mm=$2
yy=$3
echo “$dd $mm $yy”
```

**I/P : 20/10/2000 O/P : 20 10 2000**

## date command

```
#$ date
Fri Jul 26 11:58:11 AM IST 2024
d=`date +”%d”`
o/p : 26
```

See the remaining options of date command using **man** command.

### File Handling in Shell Scripts:

Reading Files: Know how to read from a file line by line using a loop (while IFS=" " read -r ...).

Writing Files: Understand basic file operations like redirection (> and >>).

- **Checking if a File Exists:**

```
#!/bin/bash
file="myfile.txt"
if [ -f "$file" ]; then
    echo "$file exists."
else
    echo "$file does not exist."
fi
```

- **Reading and Processing Lines from a File:**

```
#!/bin/bash
file="data.txt"
while IFS= read -r line
do
    echo "Processing line: $line"
    # Perform operations on $line
done < "$file"
```

- **Counting Lines, Words, and Characters in a File:**

```
#!/bin/bash
file="data.txt"
lines=$(wc -l < "$file")
words=$(wc -w < "$file")
characters=$(wc -m < "$file")
echo "File: $file"
echo "Lines: $lines"
echo "Words: $words"
echo "Characters: $characters"
```

- **To copy lines from one file and paste them line by line into another file using a shell script.**

```
#!/bin/bash
# Input file
input_file="input.txt"
# Output file
output_file="output.txt"
# Check if input file exists
if [ ! -f "$input_file" ]; then
    echo "Input file $input_file not found."
    exit 1
```

```
fi

# Clear or create output file
> "$output_file"


# Read input file line by line and copy to output file
while IFS= read -r line
do
    echo "$line" >> "$output_file"
done < "$input_file"
echo "Lines copied from $input_file to $output_file."
```

**Function:**

A function is a reusable block of code. Often, we put repeated code in a function and call that function from various places.

**Calling function**

In Shell calling function is exactly same as calling any other command. For instance, if your function name is my_func, then it can be executed as follows:

```
$ my_func
```

If any function accepts arguments, then those can be provided from command line as follows:

```
$ my_func arg1 arg2 arg3
```

**Defining function**

We can use below syntax to define function:

```
 function function_name {
        Body of function
}
```

Body of the function can contain any valid command, loop constraint, other function or script.

**Example:**

```
#!/bin/sh
myfunc()
{
  echo "I was called as : $@"
  x=2
}
### Main script starts here
echo "Script was called with $@"
x=1
echo "x is $x"
myfunc 1 2 3
echo "x is $x"
```

- **Factorial:**

```
#!/bin/bash

# Function to calculate factorial

factorial() {

    fact=1

    for (( i=1; i<=$1; i++ )); do

        fact=$((fact * i))

    done

    echo $fact

}

# Read number from user

read -p "Enter a number: " num

# Calculate factorial
```

```
result=$(factorial $num)

# Display the result

echo "Factorial of $num is $result"
```

- **Fibonacci Series:**

```
#!/bin/bash
# Function to generate Fibonacci sequence up to n terms
fibonacci() {
    a=0

    b=1

    for (( i=0; i<$1; i++ )); do

        echo -n "$a "

        fn=$((a + b))

        a=$b

        b=$fn

    done

    echo

}


# Read number of terms from user

read -p "Enter the number of terms: " num

# the Fibonacci sequence

echo "Fibonacci sequence up to $num terms:"

fibonacci $num
```

**Practice:**

1. Prime Number Check
2. Greatest Common Divisor (GCD)
3. Palindrome Check
4. Sum of Digits

**File Handling:**

**x=`head -n 3 t.txt | tail -n 1 | cut -d " " -f 1`**

Here's what each part of the command does:

1. **head -n 3 t.txt**: This command outputs the first three lines of t.txt.
2. **tail -n 1**: This command takes the output of the previous command (the first three lines) and extracts the last line of that output, which is the third line of t.txt.
3. **cut -d " " -f 1**: This command takes the third line and cuts it into fields using a space as the delimiter, then extracts the first field.
4. **x=...``: This assigns the output of the entire command to the variable x.

**Explanation:**

- **head -n 3 t.txt**: Shows the first 3 lines of t.txt.
- **tail -n 1**: Gets the last line from the output of head, which is the 3rd line of t.txt.
- **cut -d " " -f 1**: Splits the line by spaces and takes the first part.
- **Backticks (...)**: Executes the enclosed command and returns the output.

**l=`wc -l < t.txt`**

**Explanation:**

- wc -l: Counts the number of lines.
- < t.txt: Redirects the content of t.txt to the wc -l command.
- Backticks (...): Execute the enclosed command and capture the output into the variable l.

This way, the variable l will contain the line count of t.txt.

**out=`sh gcd.sh $x $y`**

**Explanation:**

1. **sh gcd.sh $x $y**: This runs the shell script gcd.sh with two arguments, $x and $y.
2. **Backticks (...)**: These are used to execute the command inside them and capture its output.
3. **out=**: Assigns the captured output to the variable out.

**ls > t.txt**

**Explanation:**

- **ls**: Lists the contents of the current directory.
- >: Redirects the output of the ls command to a file.
- **t.txt**: The file where the output is stored. If t.txt already exists, its content will be overwritten. If it doesn't exist, it will be created.