

Operating Systems Lab

Experiment 01: Basic Linux Commands

1. ls – It lists the contents of the directory.

```
localhost:~# ls
bench.py      hello.c      hello.js      readme.txt
```

2. cat – It displays the contents of the files

```
localhost:~# cat hello.c
#include <stdio.h>
int main(void)
{
    printf("hello world\n");
    return 0;
}
```

3. pwd – It prints the current working directory

```
localhost:~# pwd
/root
```

4. whoami – It displays the current user name

```
localhost:~# whoami
root
```

5. clear – It clears the terminal screen.

```
localhost:~# [REDACTED]
```

6. mkdir – It creates a new directory

```
localhost:~# mkdir Y_kataria
localhost:~# ls
Y_kataria  bench.py  hello.c  hello.js
```

7. cd – It changes the current working directory

```
localhost:~# cd Y_kataria
localhost:~/Y_kataria# pwd
/root/Y_kataria
```

8. cd .. – It moves back to the parent directory.

```
localhost:~/Y_kataria# cd ..
localhost:~# pwd
/root
```

9. rmdir – It deletes the directory.

```
localhost:~# rmdir Y_kataria
localhost:~# ls
bench.py  hello.c  hello.js  readme.txt
```

10. cat > hello.txt – It allows us to create a file named hello.txt and allows to write content in it. Press Ctrl + D to save and exit.

```
localhost:~# cat > hello.txt
This is my 1st OS lab practical.
localhost:~# ls
bench.py  hello.c  hello.js  hello.txt  readme.txt
localhost:~# cat hello.txt
This is my 1st OS lab practical.
```

11. cat >> hello.txt – It appends the content to the hello.txt file.

```
localhost:~# cat >> hello.txt
OS stands for Operating Systems...
localhost:~# ls
bench.py    hello.c    hello.js    hello.txt    readme.txt
localhost:~# cat hello.txt
This is my 1st OS lab practical.
OS stands for Operating Systems...
```

12. cp hello.txt bye.txt – It copies hello.txt file to a new file named bye.txt. Here, hello.txt acts as source file and bye.txt as a destination file.

```
localhost:~# cp hello.txt bye.txt
localhost:~# ls
bench.py    bye.txt    hello.c    hello.js    hello.txt    readme.txt
localhost:~# cat bye.txt
This is my 1st OS lab practical.
OS stands for Operating Systems...
```

13. mv bye.txt file.txt – It renames or moves bye.txt file to file.txt file. Here, bye.txt is source file and file.txt is destination file.

```
localhost:~# mv bye.txt file.txt
localhost:~# ls
bench.py    file.txt    hello.c    hello.js    hello.txt
localhost:~# cat file.txt
This is my 1st OS Lab practical
OS stands for Operating Systems...
```

14. rm file.txt – It deletes the file ‘file.txt’.

```
localhost:~# rm file.txt
localhost:~# ls
bench.py    hello.c    hello.js    hello.txt    readme.txt
localhost:~#
```

Experiment 02

Aim: Create an Animal directory, now create 2 more directories in Animal namely Mammal and Reptile. In Mammal directory create 2 files cow.txt and lizard.txt with 2 lines about cow and lizard respectively. Now, move lizard.txt from mammal to reptile directory.

Step 1: Create a directory Animal

mkdir Animal

```
localhost:~# mkdir Animal
localhost:~# ls
Animal      bench.py    hello.c
```

Step 2: In Animal, create two directories mammals and reptiles

cd Animal

mkdir mammals reptiles

```
localhost:~# cd Animal
localhost:~/Animal# mkdir mammals reptiles
localhost:~/Animal# ls
mammals  reptiles
```

Step 3: In mammals add a file cow.txt with 2 lines on cow written in the file.

cd mammals

cat > cow.txt

```
localhost:~/Animal# cd mammals
localhost:~/Animal/mammals# cat > cow.txt
Cows are a source of milk.
Cows are herbivorous animals.
```

Step 4: Again, in mammals add a file lizard.txt with 2 lines on lizard written in the file.

cat > lizard.txt

```
localhost:~/Animal/mammals# cat > lizard.txt
Lizards are cold-blooded creatures.
Lizards have long tails.
```

Step 5: Now, move lizard.txt file to reptiles from mammals.

mv lizard.txt ./reptiles/lizard.txt

```
localhost:~/Animal/mammals# mv lizard.txt ..../reptiles/lizard.txt
localhost:~/Animal/mammals# ls
cow.txt
localhost:~/Animal/mammals# cd ..
localhost:~/Animal# cd reptiles
localhost:~/Animal/reptiles# ls
lizard.txt
```

Experiment 03

Aim: Create a directory Vehicle. In Vehicle create three directories with names Fourwheelers, Threewheelers, Twowheelers.

- In Fourwheelers create directories Bus, Car, Truck.
- In Threewheelers create directories Auto.
- In Twowheelers create directories Cycle and Scooty.

In the Car directory, create files: Carbrand.txt, Busbrand.txt, Truckbrand.txt, Autocolor.txt, Cyclebrand.txt. Write 2 lines in each file. Move each file to their respective directories. Also, delete the Cycle directory finally.

Step 1: Create a directory named Vehicle.

mkdir Vehicle

ls

```
localhost:~# mkdir vehicle
localhost:~# ls
bench.py    hello.c    hello.js    readme.txt  vehicle
```

Step 2: In vehicle directory create subdirectories - Fourwheelers, Threewheelers, Twowheelers.

cd vehicle

mkdir Fourwheelers Threewheelers Twowheelers

ls

```
localhost:~# cd vehicle
localhost:~/vehicle# mkdir Fourwheelers Threewheelers Twowheelers
localhost:~/vehicle# ls
Fourwheelers  Threewheelers  Twowheelers
```

Step 3: Inside the Fourwheelers directory, create the following subdirectories – Bus, Car, Truck.

cd Fourwheelers

mkdir Bus Car Truck

ls

```
localhost:~/vehicle# cd Fourwheelers
localhost:~/vehicle/Fourwheelers# mkdir Bus Car Truck
localhost:~/vehicle/Fourwheelers# ls
Bus      Car      Truck
```

Step 4: Inside the Threewheelers directory, create a subdirectory named Auto

cd Threewheelers

mkdir Auto

ls

```
localhost:~/vehicle# cd Threewheelers
localhost:~/vehicle/Threewheelers# mkdir Auto
localhost:~/vehicle/Threewheelers# ls
Auto
```

Step 5: Inside the Twowheelers directory, create the following subdirectories – Cycle, Scooty.

```
cd Twowheelers  
mkdir Cycle Scooty  
ls
```

```
localhost:~/vehicle# cd Twowheelers  
localhost:~/vehicle/Twowheelers# mkdir Cycle Scooty  
localhost:~/vehicle/Twowheelers# ls  
Cycle  Scooty
```

Step 6: In the Car directory, create the following files - Carbrand.txt, Busbrand.txt, Truckbrand.txt, Autocolor.txt, Cyclebrand.txt.

```
cd Fourwheelers  
cd car  
cat > carbrand.txt  
cat > busbrand.txt  
cat > truckbrand.txt  
cat > autocolour.txt  
cat > cyclebrand.txt  
ls
```

```
localhost:~/vehicle/Fourwheelers# cd Car  
localhost:~/vehicle/Fourwheelers/Car# cat>carbrand.txt  
Car brands are Toyota, Mercedes, BMW.  
Sports car brands are Porschce and Ferrari.  
localhost:~/vehicle/Fourwheelers/Car# cat>busbrand.txt  
Bus brands consists of Volvo, Force.  
Volvo is a luxurious bus.  
localhost:~/vehicle/Fourwheelers/Car# cat>truckbrand.txt  
Truck brand consists of Bharat benz and Tata.  
localhost:~/vehicle/Fourwheelers/Car# cat>autocolour.txt  
Autos comes in many colors Green-Yellow, Pink and Black.  
Green-Yellow is most commonly seen on roads.  
localhost:~/vehicle/Fourwheelers/Car# cat>cyclebrand.txt  
Cycle brands are as follows:  
Avon,Atlas and Firefox.  
localhost:~/vehicle/Fourwheelers/Car# ls  
autocolour.txt  busbrand.txt  carbrand.txt  cyclebrand.txt  truckbrand.txt
```

Step 7: Now move the files from Car directory to their respective directories.

```
mv busbrand.txt ..//Bus//busbrand.txt  
mv truckbrand.txt ..//Truck/truckbrand.txt  
ls
```

```
localhost:~/vehicle/Fourwheelers/Car# mv busbrand.txt ..//Bus/busbrand.txt  
localhost:~/vehicle/Fourwheelers/Car# mv truckbrand.txt ..//Truck/truckbrand.txt  
localhost:~/vehicle/Fourwheelers/Car# ls  
autocolour.txt  carbrand.txt  cyclebrand.txt
```

```
mv autocolour.txt ..///Auto
ls
mv cyclebrand.txt ..///Cycle
ls
```

```
localhost:~/vehicle/Fourwheelers/Car# mv autocolour.txt ..///Auto
localhost:~/vehicle/Fourwheelers/Car# ls
carbrand.txt    cyclebrand.txt
localhost:~/vehicle/Fourwheelers/Car# mv cyclebrand.txt ..///Cycle
localhost:~/vehicle/Fourwheelers/Car# ls
carbrand.txt
```

Step 8: Delete the Cycle Directory

```
cd Twowheelers
rmdir Cycle
ls
```

```
localhost:~/vehicle/Fourwheelers/Car# cd ..
localhost:~/vehicle/Fourwheelers# cd ..
localhost:~/vehicle# cd Twowheelers
localhost:~/vehicle/Twowheelers# rmdir Cycle
localhost:~/vehicle/Twowheelers# ls
Scooty
```

Experiment 04

Aim: Learn use of chmod command and vi text editor.

The “chmod” command modifies the read, write, and execute permissions of specified files. The octal digits used for assigning permissions are as follows:

Octal Digit	Permissions	Symbolic Display
7	read, write, execute	rwx
6	read, write	rw-
5	read, execute	r-x
4	read	r--
3	write, execute	-wx
2	write	-w-
1	execute	--x
0	no permissions	---

- Owner is denoted by ‘u’.
- Group is denoted by ‘g’.
- Others is denoted by ‘o’.

Examples

1. Using octal notation:
 - chmod 711 test.sh
2. Using symbolic notation:
 - chmod u+rwx test.sh
 - chmod go+--x test.sh

The “vi text editor” is a powerful text editor available in Unix and Linux systems. It is widely used for editing configuration files and scripts.

Basic Modes in vi

1. Insert Mode (For writing text)
 - Press i to enter insert mode.
 - Start typing the content.
2. Command Mode (Default mode)
 - Used for navigation and executing commands.
 - Press Esc to return to this mode.
3. Last Line Mode (For saving and exiting)
 - Press Esc, then type : to enter last line mode.
 - :wq → Save and exit.

Program Case 1: Creating hello.sh file in a directory with your name and giving permission to user only using chmod command.

```
ls  
mkdir Akshat  
cd Akshat  
cat > hello.sh  
#!/bin/bash  
echo "Hello World"  
chmod u+rwx hello.sh  
./hello.sh
```

```
localhost:~# ls  
bench.py    hello.c    hello.js    readme.txt  
localhost:~# mkdir Akshat  
localhost:~# cd Akshat  
localhost:~/Akshat# cat > hello.sh  
#!/bin/bash  
echo "Hello World"  
localhost:~/Akshat# chmod u+rwx hello.sh  
localhost:~/Akshat# ./hello.sh  
Hello World  
localhost:~/Akshat#
```

Program Case 2: Creating vi text editor file namely test.sh

```
vi test.sh  
i (for start inserting text)  
#!/bin/bash  
var1="Hello"  
var2="Akshat"  
echo $var1 $var2  
esc (for escaping insert mode and come back to command mode)  
:wq(for save and exit)  
chmod u+rwx test.sh  
./test.sh
```

```
localhost:~# vi test.sh  
#!/bin/bash  
var1="Hello"  
var2="Akshat"  
echo $var1 $var2  
~  
~  
~  
~  
~  
localhost:~# chmod u+rwx test.sh  
localhost:~# ./test.sh  
Hello Akshat  
localhost:~#
```

Experiment 05

Aim: If – Else statements in bash

Bash scripting allows conditional execution using if-else statements. These statements enable decision-making within a script, executing different commands based on whether a condition evaluates to be true or false.

Basic Syntax of If-Else Statement

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

Examples of If-Else Statements

1. Simple If Statement:

This script checks if 1 is equal to 1 and prints a message.

```
localhost:~/Akshat# cat > testif.sh
#!/bin/bash
if [ 1 -eq 1 ];
then
    echo "1 is equal to 1"
fi
localhost:~/Akshat# bash testif.sh
1 is equal to 1
```

2. If-Else Statement:

This script compares two numbers.

```
localhost:~/Akshat# cat > testifelse.sh
#!/bin/bash
if [ 1 -eq 1 ];
then
    echo "1 is equal to 1"
else
    echo "The numbers are not equal"
fi
localhost:~/Akshat# bash testifelse.sh
1 is equal to 1
```

3. File Existence Check:

This script checks if a file named `data.txt` exists.

```
localhost:~/Akshat# cat > testifelse1.sh
#!/bin/bash
if [ -f data.txt ]; then
    echo "File exists"
else
    touch data.txt
    echo "New file created"
fi
localhost:~/Akshat# bash testifelse1.sh
File exists
```

4. String Comparison:

This script compares two string variables.

```
localhost:~/Akshat# cat>stringcompare.sh
#!/bin/bash
user="admin"
administrator="admin"

if [ "$user" = "$administrator" ]; then
    echo "The strings match"
else
    echo "The strings do not match"
fi
localhost:~/Akshat# bash stringcompare.sh
The strings match
```

5. User Authentication Check:

This script prompts the user for a name and verifies if they are an administrator.

```
localhost:~/Akshat# cat>ifelse5.sh
#!/bin/bash
auth_user="admin"
read -p "What's your name? " user

if [[ "$user" == "$auth_user" ]]; then
    echo "You are an administrator"
elif [[ -z "$user" ]]; then
    echo "Please enter a username"
    read user
    echo "Hello $user, Greetings!"
else
    echo "You are a standard user"
fi
localhost:~/Akshat# bash ifelse5.sh
What's your name? Akshat
You are a standard user
```

Experiment 06

Aim: To study the usage of For loop in shell.

- Using a for loop in shell scripting can be handy for iterating through lists of items or performing operations on files.
- In shell scripting, for loops typically follow this syntax:

```
for item in list
do
    # commands to be executed for each item
done
```

Program Case 1: Echo Basic Manage

```
vi forloop1.sh
#!/bin/bash
$SERVERS="s1 s2 s3"
for S in $SERVERS; do
    echo "updating pkg on: $S"
done
~
~
chmod u+rwx forloop1.sh
./forloop1.sh
```

```
localhost:~# vi forloop1.sh
#!/bin/bash
$SERVERS="s1 s2 s3"
for S in $SERVERS; do
    echo "updating pkg on: $S"
done
~
~
localhost:~# chmod u+rwx forloop1.sh
localhost:~# ./forloop1.sh
updating pkg on: s1
updating pkg on: s2
updating pkg on: s3
```

Program Case 2: Iterating through range of Numbers

```
vi for2.sh
#!/bin/bash
for value in {1..5}
do
    echo "number: $value"
done
~
~
chmod u+rwx for2.sh
./for2.sh
```

```
localhost:~# vi for2.sh
#!/bin/bash
for value in {1..5}
do
    echo "number: $value"
done
~
~

localhost:~# chmod u+rwx for2.sh
localhost:~# ./for2.sh
number: 1
number: 2
number: 3
number: 4
number: 5
```

Program Case 3: Iterating through multiple files

```
vi forloop3.sh
#!/bin/bash
for file in /root/*
do
    chmod 755 "$file"
    echo "update permission for: $file"
done
~
chmod u+rwx forloop3.sh
./forloop3.sh
```

```
localhost:~# vi forloop3.sh
#!/bin/bash
for file in /root/*
do
    chmod 755 "$file"
    echo "update permission for: $file"
done
~
~

localhost:~# chmod u+rwx forloop3.sh
localhost:~# ./forloop3.sh
update permission for: /root/bench.py
update permission for: /root/forloop3.sh
update permission for: /root/hello.c
update permission for: /root/hello.js
update permission for: /root/readme.txt
```

Program Case 4: Creating an Infinite Loop

```
vi forloop4.sh
#!/bin/bash
for (( ; ; ))
do
    echo "This is infinite loop"
    echo "Use Ctrl+C to stop it"
done
~
~

Chmod u+rwx forloop4.sh
./forloop4.sh
```

```
localhost:~# vi forloop4.sh
#!/bin/bash
for (( ; ; ))
do
    echo "This is infinite loop"
    echo "Use Ctrl+C to stop it"
done
~
~
localhost:~# chmod u+rwx forloop4.sh
localhost:~# ./forloop4.sh
This is infinite loop
Use Ctrl+C to stop it
This is infinite loop
Use Ctrl+C to stop it
This is infinite loop
Use Ctrl+C to stop it
This is infinite loop
.
^C
localhost:~#
```

Program Case 5: Implementing a Nested for loop

```
vi forloop5.sh
#!/bin/bash
for serverd in A B C; do
    for app in apache dp; do
        echo "$serverd can run $app LAMP package"
    done
done
~
~
Chmod 711 forloop5.sh
./forloop5.sh
```

```
localhost:~# vi forloop5.sh
#!/bin/bash
for serverd in A B C; do
    for app in apache dp; do
        echo "$serverd can run $app LAMP package"
    done
done
~
~
localhost:~# chmod 711 forloop5.sh
localhost:~# ./forloop5.sh
A can run apache LAMP package
A can run dp LAMP package
B can run apache LAMP package
B can run dp LAMP package
C can run apache LAMP package
C can run dp LAMP package
```

Program Case 6: Array utilization in for loop

```
vi forloop6.sh
#!/bin/bash
apps=("apache" "mysql" "php")
for app in "${apps[@]}"
do
    echo "The application name is $app"
done
~
~
chmod 711 forloop6.sh
./forloop6.sh
```

```
localhost:~# vi forloop6.sh
#!/bin/bash
apps=("apache" "mysql" "php")
for app in "${apps[@]}"
do
    echo "The application name is $app"
done
~
~

localhost:~# chmod 711 forloop6.sh
localhost:~# ./forloop6.sh
The application name is apache
The application name is mysql
The application name is php
```

Program Case 7: Using break statement in for loop

```
vi forloop7.sh
#!/bin/bash
for file in ~/.* ; do
    if [[ "$file" == "./data.txt" ]]
    then
        echo "$file is available"
        break
    fi
done
~
chmod 711 forloop7.sh
./forloop7.sh
```

```
localhost:~# vi forloop7.sh
#!/bin/bash
for file in ~/.* ; do
    if [[ "$file" == "./data.txt" ]]
    then
        echo "$file is available"
        break
    fi
done
~

localhost:~# chmod 711 forloop7.sh
localhost:~# ./forloop7.sh
```

Program Case 8: Use of command substitution

```
vi forloop8.sh
#!/bin/bash
for log in $(cat ~/testfile)
do
    echo "Log entry: $log"
done
~
~
chmod 711 forloop8.sh
./forloop8.sh
```

```
localhost:~# vi forloop8.sh
#!/bin/bash

for log in $(cat ~/testfile)
do
    echo "Log entry:..$log"
done
~
~
~

localhost:~# chmod 711 forloop8.sh
localhost:~# ./forloop8.sh
```

Experiment 07

Aim: Building C project using Makefile

Introduction

- When working on large projects with multiple source files, compiling each file manually becomes inefficient.
- This is where **Makefile** comes in, it automates the compilation process, reducing repetitive tasks and improving efficiency.

Components of the Example Project

The given example demonstrates building a simple project using a Makefile with four key files:

1. hellomake.c

- This is the main file that calls the function myPrintHelloMake().

```
localhost:~# mkdir C_Project
localhost:~# cd C_Project
localhost:~/C_Project# vi hellomake.c
#include <hellomake.h>
int main() {
    myPrintHelloMake();
    return 0;
}
~
```

- It includes hellomake.h, which contains the function prototype.
- The main() function calls myPrintHelloMake().

2. hellofunc.c

- This file defines the function myPrintHelloMake().
- It includes the standard input-output library (stdio.h) to use printf().
- It also includes hellomake.h to ensure function consistency across multiple files.
- The function prints "Hello makefile!".

```
localhost:~/C_Project# vi hellofunc.c
#include <stdio.h>
#include <hellomake.h>
void myPrintHelloMake() {
    printf("Hello makefile!\n");
    return;
}
~
```

3. hellomake.h

- This is the header file that declares the function myPrintHelloMake().
- It ensures the function is recognized in all source files that include this header.

```
localhost:~/C_Project# vi hellomake.h
void myPrintHelloMake();
~
```

4. Makefile

- A **Makefile** is used to automate the compilation process. The example Makefile contains:

```
localhost:~/C_Project# vi makefile
CC = clang
CFLAG = -I.
DEPS = hellomake.h

%.o: %.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAG)

hellomake: hellomake.o hellofunc.o
    $(CC) -o hellomake hellomake.o hellofunc.o
~
```

Final Execution:

```
localhost:~/C_Project# make
clang -c -o hellomake.o hellomake.c -I.
clang -c -o hellofunc.o hellofunc.c -I.
clang -o hellomake hellomake.o hellofunc.o
localhost:~/C_Project# ./hellomake
Hello makefile!
localhost:~/C_Project#
```

Experiment 08

Aim: Basic programs in shell scripting

1. WAP to check whether no. entered is even or odd

```
localhost:~# vi evenodd.sh
#!/bin/bash
read -p "Enter a number: " num

if [ $((num % 2)) -eq 0 ]
then
    echo "Your number is even"
else
    echo "Your number is odd"
fi

~
~

localhost:~# chmod 711 evenodd.sh
localhost:~# ./evenodd.sh
Enter a number: 12
Your number is even
```

2. WAP to print the factorial of a number

```
localhost:~# vi factorial.sh
#!/bin/bash
read -p "Enter a number: " num

factorial=1

for ((i = 1; i <= num; i++))
do
    factorial=$((factorial * i))
done

echo "Factorial: $factorial"
~

localhost:~# chmod 711 factorial.sh
localhost:~# ./factorial.sh
Enter a number: 5
Factorial: 120
```

3. WAP to create directories through vi text editor

```
localhost:~# mkdir yash
localhost:~# cd yash
localhost:~/yash# vi directories.sh
#!/bin/bash
mkdir -p {Maths, English}/{Notes, Examresults}
~
~
localhost:~/yash# chmod +x directories.sh
localhost:~/yash# ./directories.sh
localhost:~/yash# ls -R
.:
English}      Examresults}    directories.sh {Maths,
./English}:
{Notes,
./English}/{Notes,:
./Examresults}:
./{Maths,:
```

4. WAP to read a file into a variable

```
localhost:~/yash# cat > mysaplefile.txt
My name is Yash Kataria. I study in MRIIRS.
localhost:~/yash# vi readfile.sh
~
#!/bin/bash
myvalue=$(cat mysamplefile.txt)
echo "$myvalue"
~
localhost:~/yash# chmod 711 readfile.sh
localhost:~/yash# ./readfile.sh
My name is Yash Kataria. I study in MRIIRS.
```

5. WAP to read a file line by line

```
localhost:~/yash# cat > car.txt
Maruti Suzuki swift dezire
MG Hector
Mahindra Thar roxx
localhost:~/yash# vi printfile.sh
#!/bin/bash
myfile="car.txt"
i=1
while read lines; do
    echo "$i: $lines"
    i=$((i+1))
done < "$myfile"
~
localhost:~/yash# chmod u+rwx printfile.sh
localhost:~/yash# ./printfile.sh
1: Maruti Suzuki swift dezire
2: MG Hector
3: Mahindra Thar roxx
```

6. WAP to display system information

```
localhost:~/yash# vi system.sh
#!/bin/bash
echo "Date"
date
echo "Uptime"
uptime
echo "Memory usage"
free -m
echo "Network usage"
ip a
~

localhost:~/yash# chmod 711 system.sh
localhost:~/yash# ./system.sh
Date
Mon Mar 17 06:38:05 UTC 2025
Uptime
 06:38:05 up 13 min,  load average: 0.00, 0.00, 0.00
Memory usage
              total        used        free      shared  buff/cache   available
Mem:          119           5         113           0           1          111
Swap:            0           0           0
Network usage
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNOWN qlen 1000
    link/ether 02:bc:60:45:c5:f3 brd ff:ff:ff:ff:ff:ff
```

7. WAP to find and replace text in a string

```
localhost:~/yash# vi findreplace.sh
#!/bin/bash
first="I drive BMW and Volvo"
second="Audi"
echo "${first/$second}"
~

localhost:~/yash# chmod u+rwx findreplace.sh
localhost:~/yash# ./findreplace.sh
I drive Audi and Volvo
```

Experiment 09

Aim: To study and implement the **read()**, **write()**, and **fork()** system calls in Unix/Linux operating systems.

System Calls in Unix/Linux

A system call is a direct interface between a user program and the operating system kernel. It allows programs to request services such as file I/O, process control, and inter-process communication.

In this experiment, we focus on the following three fundamental system calls:

1. **write()** – For low-level output operations.
2. **read()** – For low-level input operations.
3. **fork()** – For process creation.

write() System Call

The write() system call is used to **output data** to a file descriptor, such as the standard output (screen).

```
localhost:~# vi writesc.c
#include <stdio.h>
#include <unistd.h>

int main() {
    int count;
    count = write(1, "hello\n", 6);
    printf("Total bytes written: %d\n", count);
    return 0;
}

localhost:~# gcc writesc.c -o writesc
localhost:~# ./writesc
hello
Total bytes written: 6
```

read() System Call

The read() system call is used to **read data** from an input file descriptor, such as the keyboard (standard input).

```
localhost:~# vi readsc.c
#include <stdio.h>
#include <unistd.h>

int main() {
    int nread;
    char buff[20];

    // Read 10 bytes from standard input
    nread = read(0, buff, 10);

    // Write the read bytes to standard output
    write(1, buff, 10);

    return 0;
}
```

```
localhost:~# gcc readsc.c -o readsc
localhost:~# ./readsc
123456789
123456789
```

fork() System Call

The fork() system call is used to **create a new child process** from the parent process.

```
localhost:~# vi forksc.c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    pid_t p;

    printf("Before fork\n");

    p = fork(); // Create child process

    if (p == 0) {
        printf("I am child having id %d\n", getpid());
        printf("My parent id is %d\n", getppid());
    }
    else {
        printf("My child id is %d\n", p);
        printf("I am parent having id %d\n", getpid());
    }

    printf("Common statement\n"); // Executes in both processes
    return 0;
}

localhost:~# gcc forksc.c -o forksc
localhost:~# ./forksc
Before fork
My child id is 89
I am parent having id 88
Common statement
I am child having id 89
My parent id is 88
Common statement
```

Experiment 10

Aim: To create an Orphan process and a Zombie Process.

- **An orphan process** is a process whose parent has terminated before it finishes its execution.
- **A zombie process** is a process that has completed execution but still has an entry in the process table.

Program to create an orphan process

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

int main() {
    pid_t p;
    p = fork();

    if (p == 0) {
        // Child process
        sleep(5);
        printf("I am child having PID: %d\n", getpid());
        printf("My parent PID is: %d\n", getppid());
    } else {
        // Parent process
        printf("My child PID is: %d\n", p);
    }

    return 0;
}
```

```
localhost:~# vi orphan.c
// orphan.c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    pid_t p;
    p = fork();

    if (p == 0) {
        // Child process
        sleep(5);
        printf("I am child having PID: %d\n", getpid());
        printf("My parent PID is: %d\n", getppid());
    } else {
        // Parent process
        printf("I am parent having PID: %d\n", getpid());
        printf("My child PID is: %d\n", p);
    }

    return 0;
}

localhost:~# gcc -o orphan orphan.c
localhost:~# ./orphan
I am parent having PID: 81
My child PID is: 82
localhost:~# I am child having PID: 82
My parent PID is: 1
```

Program to create a zombie process

```
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t p;
    p = fork();

    if (p == 0) {
        // Child process
        printf("Child having ID: %d\n", getpid());
    } else {
        // Parent process
        printf("Parent having ID: %d\n", getpid());
        sleep(15); // Run `ps` command during this time to observe zombie
    }

    return 0;
}
```

```
localhost:~# vi zombie.c
#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t p;
    p = fork();

    if (p == 0) {
        // Child process
        printf("Child having ID: %d\n", getpid());
    } else {
        // Parent process
        printf("Parent having ID: %d\n", getpid());
        sleep(15); // Run `ps` command during this time to observe zombie
    }

    return 0;
}
localhost:~# gcc -o zombie zombie.c
localhost:~# ./zombie &
localhost:~# Parent having ID: 89
Child having ID: 90

localhost:~# ps -elf | grep zombie
  92 root      0:00 grep zombie
[1]+  Done                      ./zombie
```

Experiment 11

Aim: Write a program to create a thread in Linux OS.

- A **thread** is the smallest unit of execution within a process. Threads allow a program to perform multiple tasks concurrently, sharing the same memory space.
- In Linux, threads can be created using the **POSIX Threads (pthreads)** library.
- Each thread runs independently and can perform different parts of a program simultaneously.
- Multithreading is especially useful for improving the performance of applications that require parallel processing.

Sample Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

void* thread_function(void* arg) {
    int i;
    printf("Inside thread\n");
    for (i = 0; i < 5; i++) {
        printf("i: %d\n", i);
        sleep(1);
    }
    return NULL;
}

int main() {
    pthread_t a_thread; // Thread declaration
    int j;

    pthread_create(&a_thread, NULL, thread_function, NULL);
    pthread_join(a_thread, NULL); // Process waits for thread to finish

    printf("Inside main program\n");
    for (j = 20; j < 25; j++) {
        printf("j: %d\n", j);
        sleep(1);
    }
    return 0;
}
```

Sample Screenshots:

Creating a vi file named threads.c

```
localhost:~/Akshat_kaushik# vi threads.c
```

Writing the C program in threads.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

void *thread_function(void *arg); // Function prototype
int i, j;

void *thread_function(void *arg)
{
    printf("Inside thread\n");
    for (i = 0; i < 5; i++)
    {
        printf("i: %d\n", i);
        sleep(1);
    }
    return NULL;
}

int main()
{
    pthread_t a_thread; // Thread declaration

    pthread_create(&a_thread, NULL, thread_function, NULL);
    pthread_join(a_thread, NULL); // Process waits for thread to finish

    printf("Inside main program\n");
    for (j = 20; j < 25; j++)
    {
        printf("j: %d\n", j);
        sleep(1);
    }

    return 0;
}
```

Compilation and Output execution:

```
localhost:~/Akshat_kaushik# gcc threads.c -o threads -pthread
localhost:~/Akshat_kaushik# ./threads
Inside thread
i: 0
i: 1
i: 2
i: 3
i: 4
Inside main program
j: 20
j: 21
j: 22
j: 23
j: 24
localhost:~/Akshat kaushik# █
```

Experiment 12

Aim: Write a program for Process Synchronization using mutex lock.

- **Mutex (Mutual Exclusion)** is a lock used to ensure that only one thread accesses the critical section at a time.

Why is Synchronization Needed?

- When multiple threads access a shared variable (shared) concurrently:
 - They may read/write incorrect or stale data.
 - This leads to unpredictable results, called a race condition.

Sample Code:

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
void *fun1();
void *fun2();
int shared = 1;           // Shared variable
pthread_mutex_t l;        // Mutex lock
int main() {
    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, fun1, NULL);
    pthread_create(&thread2, NULL, fun2, NULL);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    printf("Final value of shared is: %d\n", shared);
    return 0;
}
void *fun1() {
    int x;
    printf("Thread1 trying to acquire lock\n");
    pthread_mutex_lock(&l); // Thread acquires lock
    printf("Thread1 acquired lock\n");
    x = shared;
    printf("Thread1 reads the shared variable as: %d\n", x);
    x++;
    printf("Local updation by thread1: %d\n", x);
    sleep(1); // Thread1 is preempted by Thread2
    shared = x; // Thread1 updates the value of shared
    printf("Value of shared variable updated by Thread1 is: %d\n", shared);
    pthread_mutex_unlock(&l);
    printf("Thread1 released the lock\n");
    return NULL;
}
void *fun2() {
    int y;
    printf("Thread2 trying to acquire lock\n");
    pthread_mutex_lock(&l); // Thread acquires lock
    printf("Thread2 acquired lock\n");
    y = shared;
    printf("Thread2 reads the shared variable as: %d\n", y);
```

```

y++;
printf("Local updation by thread2: %d\n", y);
sleep(1); // Thread2 is preempted by Thread1
shared = y; // Thread2 updates the value of shared
printf("Value of shared variable updated by Thread2 is: %d\n", shared);
pthread_mutex_unlock(&l);
printf("Thread2 released the lock\n");
return NULL;
}

```

Sample Screenshots:

Creating a vi file named mutex_lock.c

```
localhost:~/AK_12# vi mutex_lock.c
```

Write the C program in mutex_lock.c

```

printf("Value of shared variable updated by Thread1 is: %d\n", shared);

pthread_mutex_unlock(&l);
printf("Thread1 released the lock\n");

return NULL;
}
void *fun2() {
    int y;
    printf("Thread2 trying to acquire lock\n");

    pthread_mutex_lock(&l); // Thread acquires lock
    printf("Thread2 acquired lock\n");

    y = shared;
    printf("Thread2 reads the shared variable as: %d\n", y);

    y++;
    printf("Local updation by thread2: %d\n", y);

    sleep(1); // Thread2 is preempted by Thread1

    shared = y; // Thread2 updates the value of shared
    printf("Value of shared variable updated by Thread2 is: %d\n", shared);

    pthread_mutex_unlock(&l);
    printf("Thread2 released the lock\n");

    return NULL;
}

```

Compilation & Output execution:

```
localhost:~/AK_12# gcc mutex_lock.c -o mutex_lock -lpthread
localhost:~/AK_12# ./mutex_lock
Thread2 trying to acquire lock
Thread2 acquired lock
Thread2 reads the shared variable as: 1
Local updation by thread2: 2
Thread1 trying to acquire lock
Value of shared variable updated by Thread2 is: 2
Thread1 acquired lock
Thread1 reads the shared variable as: 2
Local updation by thread1: 3
Thread2 released the lock
Value of shared variable updated by Thread1 is: 3
Thread1 released the lock
Final value of shared is: 3
```