Ex No: 1a) INSTALLATION AND CONFIGURATION OF LINUX

Date:22.01.25

Aim:

To install and configure Linux operating system in a Virtual Machine.

Installation/Configuration Steps:

1. Install the required packages for virtualization

dnf install xen virt-manager qemu libvirt

2. Configure xend to start up on boot

systemctl enable virt-manager.service

3. Reboot the machine

Reboot

4. Create Virtual machine by first running virt-manager

virt-manager &

5. Click on File and then click to connect to localhost

6. In the base menu, right click on the localhost(QEMU) to create a new VM 7. Select

Linux ISO image

8. Choose puppy-linux.iso then kernel version

9. Select CPU and RAM limits

10.Create default disk image to 8 GB

11.Click finish for creating the new VM with PuppyLinu

Output:

Step 1: Install required virtualization packages

Open a terminal and run:

bash

Copy code

sudo dnf install xen virt-manager qemu libvirt -y

Step 2: Enable virt-manager to start on boot

sudo systemctl enable virt-manager.service

Step 3: Reboot the system

sudo reboot

Step 4: Launch Virtual Machine Manager

After reboot, open terminal and run:

virt-manager &

Step 5: Connect to localhost

- In the Virtual Machine Manager window, click **File** > **Add Connection** (if not already connected).
- Select **QEMU/KVM** > Click **Connect** to localhost.

Step 6: Create a new Virtual Machine

- Right-click on localhost (QEMU) > **New**.

Step 7: Select Installation Media

- Choose **Local install media (ISO image or CDROM)**.
- Click **Forward**.

Step 8: Choose ISO image

- Click **Browse**, then **Browse Local** to locate your puppy-linux.iso.
- Set **OS type** to **Linux** and **version** appropriately (e.g., Generic Linux 2020 or similar).
- Click **Forward**.

Step 9: Allocate CPU and Memory

- Assign **RAM** (e.g., 1024 MB or more depending on your system).
- Assign **CPU** cores (e.g., 1 or 2).

Step 10: Create disk image

- Choose **Create a disk image for the virtual machine**.
- Set disk size to **8 GB** (default disk image).
- Click **Forward**.

Step 11: Final Settings and Create VM

- Name the VM (e.g., PuppyLinux).
- Check "Customize configuration before install" (optional for advanced users).
- Click **Finish**.

RESULT:

LINUX operating system in a vrtual machine is successfully installed and configured

Ex No: 1b **BASIC LINUX COMMANDS**

Date: 22.01.2025

1.1 GENERAL PURPOSE COMMANDS

1. The date command

Description: Displays the current date and time.
Syntax:

$ date

Input:

$ date

Output:

Sat Apr 12 10:23:45 IST 2025

Other Formats:

Format	Purpose	Input	Output
+%m	Display month (numeric)	$ date +%m	04
+%h	Display month (name)	$ date +%h	Apr
+%d	Display day of the month	$ date +%d	12
+%y	Last two digits of year	$ date +%y	25
+%H	Display hour	$ date +%H	10
+%M	Display minutes	$ date +%M	23
+%S	Display seconds	$ date +%S	45

---

2. The echo command

Description: Prints a message to the terminal.
Syntax:

$ echo “your message”

Input:

$ echo "God is Great"

Output:

God is Great

---

3. The cal command

Description: Displays calendar of specified month/year.
Syntax:

$ cal [month] [year]

Input:

$ cal Jan 2012

Output:

```
   January 2012
Su Mo Tu We Th Fr Sa
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
```

---

4. The bc command

Description: Launches a basic calculator.
Syntax:

$ bc

Input:

$ bc -l

16/4

5/2

Output:

4

2

---

5. The who command

Description: Shows users currently logged in.
Syntax:

$ who

Input:

$ who

Output:

kaviya   tty1   2025-04-12 09:00

---

6. The who am i command

Description: Shows info about current session user.
Syntax:

$ who am i

Input:

$ who am i

Output:

kaviya   pts/0   2025-04-12 09:10

---

7. The id command

Description: Displays UID, GID, and groups of user.
Syntax:

$ id

Input:

$ id

Output:

uid=1000(kaviya) gid=1000(kaviya) groups=1000(kaviya),10(wheel)

---

8. The tty command

Description: Displays terminal name.
Syntax:

$ tty

Input:

$ tty

Output:

/dev/pts/0

---

9. The clear command

Description: Clears the terminal screen.
Syntax:

$ clear

Input:

$ clear

Output:
(Terminal screen gets cleared)

---

10. The man command

Description: Shows manual page for commands.
Syntax:

$ man [command]

Input:

$ man date

Output:
(Manual page opens for the date command. Press q to quit.)

---

11. The ps command

Description: Shows running processes.
Syntax:

$ ps

Input:

$ ps

Output:

```
  PID TTY          TIME CMD
 1234 pts/0    00:00:00 bash
 1278 pts/0    00:00:00 ps
```

---

12. The uname command

Description: Shows system details.
Syntax:

$ uname [option]

Input:

$ uname -a

Output:

Linux fedora 6.5.9-300.fc39.x86_64 #1 SMP x86_64 GNU/Linux

1.2 DIRECTORY COMMANDS

1. The pwd command

Description: Displays current directory path.
Syntax:

$ pwd

Input:

$ pwd

Output:

/home/kaviya

---

2. The mkdir command

Description: Creates a new directory.
Syntax:

$ mkdir dirname

Input:

$ mkdir receee

Output:
(A directory named receee is created)

---

3. The rmdir command

Description: Deletes an empty directory.
Syntax:

$ rmdir dirname

Input:

$ rmdir receee

Output:
(The receee directory is removed if empty)

---

4. The cd command

Description: Changes the current directory.
Syntax:

$ cd dirname

Input:

$ cd receee

Output:
(You are now inside the receee directory)

---

5. The ls command

Description: Lists contents of the directory.
Syntax:

$ ls

Input:

$ ls

Output:

file1.txt file2.sh receee

Input (long listing):

$ ls -l

Output:

-rw-rw-r-- 1 kaviya kaviya 0 Apr 12 10:24 file1.txt

Input (including hidden files):

$ ls -a

Output:

. .. .bashrc file1.txt receee

1.3 FILE HANDLING COMMANDS

1. The 'cat' command

Purpose: Used to create a file.
SYNTAX:

$ cat > filename

EXAMPLE:

$ cat > rec

Arun

Kaviya

^D # (Press Ctrl + D to save and exit)

2. Display contents of a file

SYNTAX:

$ cat filename

EXAMPLE:

$ cat rec

Output:

Arun

Kaviya

3. The ‘cp’ command

Purpose: Copy contents from one file to another.
SYNTAX:

$ cp oldfile newfile

EXAMPLE:

$ cp rec cse

$ cat cse

Output:

Arun

Kaviya

4. The ‘rm’ command

Purpose: Delete a file.
SYNTAX:

$ rm filename

EXAMPLES:

$ rm rec

$ rm -f rec

$ rm -fr directory_name # Deletes folder recursively

5. The ‘mv’ command

Purpose: Move or rename a file.
SYNTAX:

$ mv oldfile newfile

EXAMPLE:

$ mv cse eee

$ ls

Output: eee

6. The 'file' command

Purpose: Determine file type.
SYNTAX:

$ file filename

EXAMPLE:

$ file eee

Output: eee: ASCII text

7. The 'wc' command

Purpose: Word, line, and character count.
SYNTAX:

$ wc filename

EXAMPLE:

$ wc eee

Output: 2 2 12 eee

8. Directing output to a file

Purpose: Save command output to a file.
SYNTAX:

$ ls > filename

EXAMPLE:

$ ls > list.txt

$ cat list.txt

Output:

eee

list.txt

9. Pipes

Purpose: Use output of one command as input to another.
SYNTAX:

$ command1 | command2

EXAMPLE:

$ who | wc -l

Output: 3 # (Displays number of logged-in users)

10. The 'tee' command

Purpose: Save output in middle of a pipe.
SYNTAX:

$ command | tee filename

EXAMPLE:

$ who | tee sample | wc -l

Output: 3

$ cat sample

Output: list of logged-in users

11. Metacharacters in Unix

Purpose: Pattern matching with special characters.

Symbol	Meaning
*	Matches any number of characters
?	Matches a single character
[ ]	Matches any character in the set
[!]	Negates the set

EXAMPLES:

$ ls r* # Files starting with r

$ ls ?kkk # Files like "rkkk", "skkk"

$ ls [a-m]* # Files starting with a-m

$ ls [!a-m]* # Files NOT starting with a-m

---

13. File Permissions

Each file has:

- **Owner**

- **Group**
- **Others**

Each with:

- **r (read)** = 4
- **w (write)** = 2
- **x (execute)** = 1

EXAMPLE:

$ ls -l college

-rwxr-xr-- 1 Lak std 1525 Jan 10 12:10 college

- **rwx**: Owner has read, write, execute
- **r-x**: Group has read and execute
- **r--**: Others have only read

---

13. The 'chmod' command

SYNTAX:

$ chmod category operation permission filename

EXAMPLES:

$ chmod u-wx college

(Remove write & execute for user)

$ chmod u+rw, g+rw college

(Add read & write to user & group)

$ chmod g=wx college

(Set write & execute to group only)

---

14. Octal Notation

SYNTAX:

$ chmod 761 college

Explanation:

- 7 (owner) = rwx

- 6 (group) = rw-
- 1 (others) = --x

1.4 GROUPING COMMANDS

1. Semicolon (;)

Executes multiple commands sequentially.
EXAMPLE:

$ who; date

Output:

(list of users)

Sat Apr 12 10:45:00 IST 2025

2. Logical AND (&&)

Executes next only if previous is successful.
EXAMPLE:

$ ls && date

Output:

(file list)

Sat Apr 12 10:45:00 IST 2025

3. Logical OR (||)

Executes next only if previous fails.
EXAMPLE:

$ ls nofile || date

Output:

ls: cannot access 'nofile': No such file or directory

Sat Apr 12 10:45:00 IST 2025

1.5 FILTERS

1. head

SYNTAX:

$ head filename

EXAMPLE:

$ head college

(Shows top 10 lines)

$ head -5 college

(Shows top 5 lines)

2. tail

SYNTAX:

$ tail filename

EXAMPLE:

$ tail college

(Shows bottom 10 lines)

$ tail -5 college

(Shows bottom 5 lines)

3. more

Used for paging large outputs.
SYNTAX:

$ ls -l | more

---

4. grep

Search for patterns.
SYNTAX:

$ grep "pattern" filename

EXAMPLE:

$ cat > student

Arun cse

Ram ece

Kani cse

^D

$ grep "cse" student

Output:

Arun cse

Kani cse

---

5. sort

Sorts lines.
SYNTAX:

$ sort filename

EXAMPLES:

$ sort college   # Sort alphabetically

$ sort -r college   # Reverse order

$ sort -n numbers.txt # Numeric sort

$ sort -u college   # Remove duplicates

---

6. nl

Adds line numbers.
SYNTAX:

$ nl filename

EXAMPLE:

$ nl college

    1 Arun

    2 Kaviya

---

7. cut

Extracts specific character positions.
SYNTAX:

$ cut -c1-4 filename

EXAMPLE:

$ cut -c1-3 college

Output:

Aru

Kav

1.5 OTHER ESSENTIAL COMMANDS

---

1. free

Description: Displays the amount of free and used physical and swap memory in the system.

- **Synopsis**: free [options]
- **Example**:

Input:

[root@localhost ~]# free -t

Output:

```
        total     used     free    shared  buff/cache  available
Mem:    4044380   605464   2045080   148820   1393836   3226708
Swap:   2621436        0   2621436
Total:  6665816   605464   4666516
```

---

2. top

Description: Provides a dynamic real-time view of processes in the system.

- **Synopsis**: top [options]
- **Example**:

Input:

[root@localhost ~]# top

Output:

```
top - 08:07:28 up 24 min,  2 users,  load average: 0.01, 0.06, 0.23
Tasks: 211 total,   1 running, 210 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.8 us,  0.3 sy,  0.0 ni, 98.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem :  4044380 total, 2052960 free,  600452 used, 1390968 buff/cache
KiB Swap: 2621436 total, 2621436 free,        0 used. 3234820 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 1105 root      20   0  175008  75700  51264 S   1.7  1.9   0:20.46 Xorg
 2529 root      20   0   80444  32640  24796 S   1.0  0.8   0:02.47 gnome-term
```

---

3. ps

Description: Reports a snapshot of current processes.

- **Synopsis**: ps [options]
- **Example**:

Input:

[root@localhost ~]# ps -e

Output:

```
 PID TTY          TIME CMD
   1 ?        00:00:03 systemd
   2 ?        00:00:00 kthreadd
   3 ?        00:00:00 ksoftirqd/0
```

---

4. vmstat

Description: Reports virtual memory statistics.

- **Synopsis**: vmstat [options]
- **Example**:

Input:

[root@localhost ~]# vmstat

Output:

```
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 0  0      0 1879368   1604 1487116    0    0    64     7   72  140  1  0 97  1  0
```

---

5. df

Description: Displays the amount of disk space available on the file system.

- **Synopsis**: df [options]
- **Example**:

Input:

[root@localhost ~]# df

Output:

```
Filesystem     1K-blocks    Used Available Use% Mounted on
```

devtmpfs   2010800   0   2010800   0% /dev

tmpfs   2022188   148   2022040   1% /dev/shm

tmpfs   2022188   1404   2020784   1% /run

/dev/sda6   487652 168276   289680 37% /boot

---

6. ping

Description: Verifies whether a device can communicate with another over a network.

- **Synopsis**: ping [options] destination
- **Example**:

Input:

[root@localhost ~]# ping 172.16.4.1

Output:

PING 172.16.4.1 (172.16.4.1) 56(84) bytes of data.

64 bytes from 172.16.4.1: icmp_seq=1 ttl=64 time=0.328 ms

64 bytes from 172.16.4.1: icmp_seq=2 ttl=64 time=0.228 ms

64 bytes from 172.16.4.1: icmp_seq=3 ttl=64 time=0.264 ms

64 bytes from 172.16.4.1: icmp_seq=4 ttl=64 time=0.312 ms

^C

--- 172.16.4.1 ping statistics ---

4 packets transmitted, 4 received, 0% packet loss, time 3000ms

rtt min/avg/max/mdev = 0.228/0.283/0.328/0.039 ms

---

7. ifconfig

Description: Used to configure and display network interface parameters.

- **Synopsis**: ifconfig [options]
- **Example**:

Input:

[root@localhost ~]# ifconfig

Output:

enp2s0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500

inet 172.16.6.102 netmask 255.255.252.0 broadcast 172.16.7.255

inet6 fe80::4a0f:cfff:fe6d:6057 prefixlen 64 scopeid 0x20<link>

ether 48:0f:cf:6d:60:57 txqueuelen 1000 (Ethernet)

RX packets 23216 bytes 2483338 (2.3 MiB)

RX errors 0 dropped 5 overruns 0 frame 0

TX packets 1077 bytes 107740 (105.2 KiB)

TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

---

8. traceroute

Description: Tracks the route that a packet takes to reach the destination.

- **Synopsis**: traceroute [options] destination
- **Example**:

Input:

[root@localhost ~]# traceroute www.rajalakshmi.org

Output:

traceroute to www.rajalakshmi.org (220.227.30.51), 30 hops max, 60 byte packets

1 gateway (172.16.4.1) 0.299 ms 0.297 ms 0.327 ms

2 220.225.219.38 (220.225.219.38) 6.185 ms 6.203 ms 6.189 ms

Ex. No.: 2(a) **Shell Script**

Date: *28.01.2025*

Aim:

To write a Shell script that performs basic arithmetic operations (Addition, Subtraction, Multiplication, Division, and Modulus) on two user-input numbers.

Algorithm:

1. Start the script.
2. Prompt the user to enter two numbers.
3. Read the input values.
4. Perform arithmetic operations: addition, subtraction, multiplication.
5. Check for division by zero:
   - If not zero, perform division and modulus.
   - If zero, display an appropriate error message.
6. Display the results.
7. End the script.

Program: (arith.sh)

```
#!/bin/bash
# Basic Calculator Script

echo "Enter two numbers:"
read a
read b

add=$((a + b))
sub=$((a - b))
mul=$((a * b))

# Check for division/modulus by zero
```

```
if [ "$b" -ne 0 ]; then
   div=$((a / b))
   mod=$((a % b))
else
   div="Error: Division by zero"
   mod="Error: Modulus by zero"
fi

# Display Results
echo "Addition       : $add"
echo "Subtraction    : $sub"
echo "Multiplication : $mul"
echo "Division       : $div"
echo "Modulus        : $mod"
```

Sample Input and Output:

```
[REC@localhost ~]$ sh arith.sh
Enter two numbers:
5
10
Addition       : 15
Subtraction    : -5
Multiplication : 50
Division       : 0
Modulus        : 5
```

Result:

The Shell script to perform basic arithmetic operations was successfully implemented, executed, and the output was verified.

Ex. No.: 2(b) **Shell Script**

Date: ***28.01.2025***

Aim:

To write a Shell script to check whether a given year is a leap year or not using conditional statements.

Algorithm:

1. Start the script.
2. Prompt the user to enter a year.
3. Read the input year.
4. Check the leap year condition:
   - A year is a leap year if it is divisible by 4 **and** not divisible by 100, **or** divisible by 400.
5. Display whether it is a leap year or not.
6. End the script.

Program: (leap.sh)

```
#!/bin/bash

# Script to check leap year


echo "Enter year:"

read year


if [ $((year % 4)) -eq 0 ]; then

  if [ $((year % 100)) -ne 0 ] || [ $((year % 400)) -eq 0 ]; then

    echo "Leap year"

  else

    echo "Not a leap year"

  fi

else
```

```
    echo "Not a leap year"
fi
```

Sample Input and Output:

[REC@localhost ~]$ sh leap.sh

Enter year:

12

Leap year

Result:

The Shell script to test whether the given year is a leap year or not was successfully implemented and executed, and the output was verified.

Ex. No.: 3(a) **Shell Script – Reverse of Digit**

Date: ***29.01.2025***

Aim:

To write a Shell script to reverse a given digit using a looping statement.

Algorithm:

1. Start the script.
2. Prompt the user to enter a number.
3. Read the input.
4. Initialize the reverse variable to 0.
5. Use a loop to extract the last digit of the number using modulus (%) and build the reversed number.
6. Divide the number by 10 in each iteration to remove the last digit.
7. Continue until the number becomes 0.
8. Display the reversed number.
9. End the script.

Program: (indhu.sh)

```
#!/bin/bash
# Script to reverse a number

echo "Enter number:"
read num
rev=0

while [ $num -gt 0 ]
do
   rem=$((num % 10))
   rev=$((rev * 10 + rem))
   num=$((num / 10))
```

```
done

echo "Reversed number: $rev"
```

Sample Input and Output:

[REC@localhost ~]$ sh indhu.sh

Enter number:

123

Reversed number: 321

Result:

The Shell script to reverse a given number using looping was successfully implemented, executed, and the output was verified.

Ex. No.: 3(b) **Shell Script – Fibonacci Series**

Date: *29.01.2025*

Aim:

To write a Shell script to generate a Fibonacci series using a for loop.

Algorithm:

1. Start the script.
2. Prompt the user to enter a limit number.
3. Read the input.
4. Initialize the first two Fibonacci numbers (a = 0, b = 1).
5. Use a loop to generate and print Fibonacci numbers up to the given number.
6. Continue until the number exceeds the input.
7. End the script.

Program: (indhu.sh)

```
#!/bin/bash

# Script to generate Fibonacci series using for loop


echo "Enter number:"

read n


a=0

b=1


echo "Fibonacci series:"

echo $a

echo $b


for (( i=0; i<n; i++ ))
```

```
do
   fib=$((a + b))
   if [ $fib -gt $n ]; then
      break
   fi
   echo $fib
   a=$b
   b=$fib
done
```

Sample Input and Output:

```
[REC@localhost ~]$ sh indhu.sh
Enter number:
21
Fibonacci series:
0
1
1
2
3
5
8
13
21
```

Result:

The Shell script to generate a Fibonacci series using a for loop was successfully implemented, executed, and the output was verified.

Ex. No.: 4(a) **EMPLOYEE AVERAGE PAY**

Date: ***08.02.2025***

Aim:

To find out the average pay of all employees whose salary is more than 6000 and the number of days worked is more than 4.

Algorithm:

1. Create a flat file emp.dat containing employee records with the fields: name, salary per day, and number of days worked.
2. Create an AWK script file emp.awk.
3. For each employee record:
    - If salary per day is greater than 6000 **and** number of days worked is greater than 4:
        - Print the employee name and the total salary earned.
        - Accumulate total pay and count of such employees.
4. At the end of the script:
    - Display the total number of qualified employees.
    - Display the total pay.
    - Display the average pay.

Program Code:

emp.dat – Input File

```
JOE 8000 5

RAM 6000 5

TIM 5000 6

BEN 7000 7

AMY 6500 6
```

emp.awk – AWK Script

```
BEGIN {

  print "EMPLOYEES DETAILS"
```

```
   count = 0
   total = 0
}
{
   name = $1
   salary = $2
   days = $3
   if (salary > 6000 && days > 4) {
      pay = salary * days
      print name, pay
      count++
      total += pay
   }
}
END {
   print "no of employees are= " count
   print "total pay= " total
   if (count > 0)
      print "average pay= " total / count
   else
      print "average pay= 0"
}
```

Sample Input and Output:

[student@localhost ~]$ vi emp.dat

[student@localhost ~]$ vi emp.awk

[student@localhost ~]$ gawk -f emp.awk emp.dat

EMPLOYEES DETAILS

JOE 40000

BEN 49000

AMY 39000

no of employees are= 3

total pay= 128000

average pay= 42666.7

Result:

The AWK script was successfully implemented to calculate the average pay of employees whose salary is greater than 6000 and who worked more than 4 days. The script executed correctly and the output was verified.

Ex. No.: 4(b) **RESULTS OF EXAMINATION**

Date: ***[Insert Date]***

Aim:

To print the pass/fail status of a student in a class based on subject marks.

Algorithm:

1. Read student data from the input file marks.dat.
2. For each record, retrieve the name and six subject marks.
3. Check each mark:
   - If any subject mark is less than 45, then the student is marked as **FAIL**.
   - Otherwise, the student is marked as **PASS**.
4. Print the student name, all marks, and the pass/fail status.

Program Code:

marks.dat – Input File

```
BEN 40 55 66 77 55 77
TOM 60 67 84 92 90 60
RAM 90 95 84 87 56 70
JIM 60 70 65 78 90 87
```

marks.awk – AWK Script

```
BEGIN {
  print "NAME SUB-1 SUB-2 SUB-3 SUB-4 SUB-5 SUB-6 STATUS"
  print "_______________________________________________________________"
}
{
  name = $1
  status = "PASS"
  for (i = 2; i <= 7; i++) {
```

```
        if ($i < 45)
            status = "FAIL"
    }
    printf "%s %3d %5d %5d %5d %5d %5d %6s\n", name, $2, $3, $4, $5, $6, $7, status
}
```

Sample Input and Output:

```
[root@localhost student]# gawk -f marks.awk marks.dat


NAME SUB-1 SUB-2 SUB-3 SUB-4 SUB-5 SUB-6 STATUS
______________________________________________________________
BEN   40    55    66    77    55    77  FAIL
TOM   60    67    84    92    90    60  PASS
RAM   90    95    84    87    56    70  PASS
JIM   60    70    65    78    90    87  PASS
```

Result:

The AWK script was executed successfully. The script correctly identified and displayed the pass/fail status of each student based on their subject marks.

Ex. No.: 5 **System Calls Programming**

Date: ***18.02.2025***

Aim:

To experiment with system calls using fork(), execlp() and pid() functions.

Algorithm:

1. **Start**
   - Include the required header files: stdio.h, stdlib.h, and unistd.h.
2. **Variable Declaration**
   - Declare an integer variable pid to hold the process ID.
3. **Create a Process**
   - Call the fork() function and store the return value in pid.
     - If fork() returns:
       - -1: Forking failed.
       - 0: This is the child process.
       - Positive value: This is the parent process.
4. **Print Statement Executed Twice**
   - Print:
   - THIS LINE EXECUTED TWICE
5. **Check for Process Creation Failure**
   - If pid == -1, print:
   - CHILD PROCESS NOT CREATED
     - Exit the program.
6. **Child Process Execution**
   - If pid == 0, print:
     - The process ID of the child using getpid().
     - The parent process ID of the child using getppid().
7. **Parent Process Execution**
   - If pid > 0, print:

        ▪ The process ID of the parent using getpid().
        ▪ The parent's parent process ID using getppid().

8. **Final Print Statement**
    - Print:
    - IT CAN BE EXECUTED TWICE
9. **End**

Program Code:

```
// filename: systemcall.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
   int pid;

   pid = fork(); // Create new process

   printf("THIS LINE EXECUTED TWICE\n");

   if (pid == -1) {
      printf("CHILD PROCESS NOT CREATED\n");
      exit(0);
   }

   if (pid == 0) {
      printf("Child Process ID: %d\n", getpid());
      printf("Parent Process ID of Child: %d\n", getppid());
   } else {
```

```
        printf("Parent Process ID: %d\n", getpid());
        printf("Parent's Parent Process ID: %d\n", getppid());
    }

    printf("IT CAN BE EXECUTED TWICE\n");
    return 0;
}
```

Sample Output:

THIS LINE EXECUTED TWICE

Parent Process ID: 12345

Parent's Parent Process ID: 1000

IT CAN BE EXECUTED TWICE

THIS LINE EXECUTED TWICE

Child Process ID: 12346

Parent Process ID of Child: 12345

IT CAN BE EXECUTED TWICE

Result:

The program was successfully executed. It demonstrated the use of system calls fork(), getpid(), and getppid() to manage parent and child processes.

Ex. No.: 6a **FIRST COME FIRST SERVE (FCFS)**

Date: ***[Insert Date]***

Aim:

To implement First-Come First-Serve (FCFS) scheduling technique.

Algorithm:

1. Start the program.
2. Input the number of processes.
3. Read the burst time for each process.
4. Calculate the waiting time for each process:
   - Waiting time of process 0 is 0.
   - For others:
     WaitingTime[i] = WaitingTime[i-1] + BurstTime[i-1]
5. Calculate the turnaround time for each process:
   TurnAroundTime[i] = WaitingTime[i] + BurstTime[i]
6. Calculate the total and average waiting time and turnaround time.
7. Display process details, total and average times.
8. End.

Program Code (in C):

```
#include <stdio.h>


int main() {
    int n, i;
    int burst_time[20], waiting_time[20], turn_around_time[20];
    int total_wt = 0, total_tat = 0;


    printf("Enter the number of process:\n");
    scanf("%d", &n);
```

```
    printf("Enter the burst time of the processes:\n");
    for (i = 0; i < n; i++) {
        scanf("%d", &burst_time[i]);
    }

    waiting_time[0] = 0;

    for (i = 1; i < n; i++) {
        waiting_time[i] = waiting_time[i - 1] + burst_time[i - 1];
    }

    for (i = 0; i < n; i++) {
        turn_around_time[i] = waiting_time[i] + burst_time[i];
        total_wt += waiting_time[i];
        total_tat += turn_around_time[i];
    }

    printf("Process\tBurst Time\tWaiting Time\tTurn Around Time\n");
    for (i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", i, burst_time[i], waiting_time[i], turn_around_time[i]);
    }

    printf("Average Waiting Time is: %.1f\n", (float)total_wt / n);
    printf("Average Turn Around Time is: %.1f\n", (float)total_tat / n);

    return 0;
}
```

Sample Output:

Enter the number of process:

3

Enter the burst time of the processes:

24 3 3

Process Burst Time Waiting Time Turn Around Time

0	24	0	24
1	3	24	27
2	3	27	30

Average Waiting Time is: 17.0

Average Turn Around Time is: 27.0

Result:

The FCFS Scheduling algorithm was successfully implemented. The program calculated the waiting time and turnaround time for each process and displayed the average times.

Ex. No.: 6b **SHORTEST JOB FIRST (SJF)**

Date: ***4.3.2025***

Aim:

To implement the Shortest Job First (SJF) scheduling technique.

Algorithm:

1. Start the program.
2. Get the number of processes.
3. Read the burst time of each process.
4. Assign process IDs (or names) and initialize waiting time and turnaround time to 0.
5. Sort the processes in ascending order of their burst time.
6. Calculate the waiting time:
    - First process waiting time = 0
    - For others: waiting_time[i] = waiting_time[i-1] + burst_time[i-1]
7. Calculate turnaround time:
   turnaround_time[i] = waiting_time[i] + burst_time[i]
8. Compute average waiting time and turnaround time.
9. Display the results.
10. End.

Program Code (in C):

```
#include <stdio.h>


int main() {

   int n, i, j, temp;

   int bt[20], p[20], wt[20], tat[20];

   float total_wt = 0, total_tat = 0;


   printf("Enter the number of process:\n");

   scanf("%d", &n);
```

```
printf("Enter the burst time of the processes:\n");
for (i = 0; i < n; i++) {
    scanf("%d", &bt[i]);
    p[i] = i + 1;  // process ID
}

// Sorting burst time using selection sort
for (i = 0; i < n - 1; i++) {
    for (j = i + 1; j < n; j++) {
        if (bt[i] > bt[j]) {
            temp = bt[i];
            bt[i] = bt[j];
            bt[j] = temp;

            temp = p[i];
            p[i] = p[j];
            p[j] = temp;
        }
    }
}

wt[0] = 0;
for (i = 1; i < n; i++) {
    wt[i] = wt[i - 1] + bt[i - 1];
    total_wt += wt[i];
}

for (i = 0; i < n; i++) {
```

```
        tat[i] = wt[i] + bt[i];
        total_tat += tat[i];
    }

    printf("Process\tBurst Time\tWaiting Time\tTurn Around Time\n");
    for (i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\n", p[i], bt[i], wt[i], tat[i]);
    }

    printf("Average waiting time is: %.1f\n", total_wt / n);
    printf("Average Turn Around Time is: %.1f\n", total_tat / n);

    return 0;
}
```

Sample Output:

Enter the number of process:

4

Enter the burst time of the processes:

8 4 9 5

Process Burst Time Waiting Time Turn Around Time

2	4	0	4
4	5	4	9
1	8	9	17
3	9	17	26

Average waiting time is: 7.5

Average Turn Around Time is: 13.0

Result:

The SJF scheduling algorithm was successfully implemented. The program displayed waiting time and turnaround time for each process, along with their averages.

Ex. No.: 6c **PRIORITY SCHEDULING**

Date: ***19.3.2025***

Aim:

To implement the Priority Scheduling technique in C.

Algorithm:

1. Start the program.
2. Get the number of processes from the user.
3. Read the process name (or ID), burst time, and priority of each process.
4. Sort the processes based on their priority (lower number = higher priority).
5. Set the waiting time of the first process to 0.
6. For each remaining process:
   waiting_time[i] = waiting_time[i-1] + burst_time[i-1]
7. Calculate turnaround time:
   turnaround_time[i] = waiting_time[i] + burst_time[i]
8. Compute the total and average waiting time and turnaround time.
9. Display the details.
10. End the program.

Program Code (in C):

```
#include <stdio.h>


int main() {

    int bt[20], p[20], wt[20], tat[20], prio[20];

    int i, j, n, temp;

    float total_wt = 0, total_tat = 0;


    printf("Enter the number of processes:\n");

    scanf("%d", &n);
```

```c
printf("Enter Burst Time and Priority of each process:\n");
for (i = 0; i < n; i++) {
    printf("Process %d - Burst Time: ", i + 1);
    scanf("%d", &bt[i]);
    printf("Process %d - Priority (lower number = higher priority): ", i + 1);
    scanf("%d", &prio[i]);
    p[i] = i + 1;
}

// Sort processes based on priority
for (i = 0; i < n - 1; i++) {
    for (j = i + 1; j < n; j++) {
        if (prio[i] > prio[j]) {
            // Swap priority
            temp = prio[i];
            prio[i] = prio[j];
            prio[j] = temp;

            // Swap burst time
            temp = bt[i];
            bt[i] = bt[j];
            bt[j] = temp;

            // Swap process ID
            temp = p[i];
            p[i] = p[j];
            p[j] = temp;
        }
    }
```

```
    }

    wt[0] = 0;
    for (i = 1; i < n; i++) {
        wt[i] = wt[i - 1] + bt[i - 1];
        total_wt += wt[i];
    }

    for (i = 0; i < n; i++) {
        tat[i] = wt[i] + bt[i];
        total_tat += tat[i];
    }

    printf("\nProcess\tBurst Time\tPriority\tWaiting Time\tTurnaround Time\n");
    for (i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", p[i], bt[i], prio[i], wt[i], tat[i]);
    }

    printf("\nAverage Waiting Time: %.2f", total_wt / n);
    printf("\nAverage Turnaround Time: %.2f\n", total_tat / n);

    return 0;
}
```

Sample Output:

Enter the number of processes:

4

Enter Burst Time and Priority of each process:

Process 1 - Burst Time: 10

Process 1 - Priority: 3

Process 2 - Burst Time: 1

Process 2 - Priority: 1

Process 3 - Burst Time: 2

Process 3 - Priority: 4

Process 4 - Burst Time: 1

Process 4 - Priority: 2

Process	Burst Time	Priority	Waiting Time	Turnaround Time
2	1	1	0	1
4	1	2	1	2
1	10	3	2	12
3	2	4	12	14

Average Waiting Time: 3.75

Average Turnaround Time: 7.25

Result:

The Priority Scheduling algorithm was successfully implemented and tested. The program displayed correct waiting and turnaround times based on priority.

Ex. No.: 6d **ROUND ROBIN SCHEDULING**

Date: ***26.03.2025***

Aim:

To implement the Round Robin (RR) scheduling technique using C programming.

Algorithm:

1. Start.
2. Get the number of processes and the time quantum from the user.
3. Read the process burst time (arrival time is assumed 0 for simplicity).
4. Initialize an array rem_bt[] (remaining burst time) as a copy of burst time.
5. Initialize an array wt[] (waiting time) as 0 for all processes.
6. Set current time t = 0.
7. Repeat while all processes are not completed:
    - For each process i:
        - If rem_bt[i] > 0:
            - If rem_bt[i] > quantum:
                - t += quantum
                - rem_bt[i] -= quantum
            - Else:
                - t += rem_bt[i]
                - wt[i] = t - bt[i]
                - rem_bt[i] = 0
8. Calculate Turnaround Time for each process as:
   tat[i] = bt[i] + wt[i]
9. Compute Average Waiting Time and Average Turnaround Time.
10. Display the process-wise result.
11. End.

Program Code (C):

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

```
int main() {
    int i, n, time = 0, quantum;
    int bt[20], rem_bt[20], wt[20], tat[20];
    float avg_wt = 0, avg_tat = 0;

    printf("Enter total number of processes: ");
    scanf("%d", &n);

    printf("Enter burst time for each process:\n");
    for (i = 0; i < n; i++) {
        printf("P[%d]: ", i + 1);
        scanf("%d", &bt[i]);
        rem_bt[i] = bt[i];
        wt[i] = 0;
    }

    printf("Enter Time Quantum: ");
    scanf("%d", &quantum);

    int done;
    do {
        done = 1;
        for (i = 0; i < n; i++) {
            if (rem_bt[i] > 0) {
                done = 0;
                if (rem_bt[i] > quantum) {
                    time += quantum;
                    rem_bt[i] -= quantum;
                } else {
                    time += rem_bt[i];
                    wt[i] = time - bt[i];
```

```
                rem_bt[i] = 0;
            }
        }
    }
} while (!done);

printf("\nProcess\tBurst Time\tWaiting Time\tTurnaround Time\n");
for (i = 0; i < n; i++) {
    tat[i] = bt[i] + wt[i];
    avg_wt += wt[i];
    avg_tat += tat[i];
    printf("P[%d]\t%d\t\t%d\t\t%d\n", i + 1, bt[i], wt[i], tat[i]);
}

avg_wt /= n;
avg_tat /= n;

printf("\nAverage Waiting Time = %.2f", avg_wt);
printf("\nAverage Turnaround Time = %.2f\n", avg_tat);

return 0;
}
```

Sample Output:

Enter total number of processes: 4

Enter burst time for each process:

P[1]: 5

P[2]: 15

P[3]: 4

P[4]: 3

Enter Time Quantum: 5

Process	Burst Time	Waiting Time	Turnaround Time
P[1]	5	0	5
P[2]	15	12	27
P[3]	4	5	9
P[4]	3	9	12

Average Waiting Time = 6.50

Average Turnaround Time = 13.25

Result:

The Round Robin Scheduling algorithm was successfully implemented and tested. It correctly calculated the waiting and turnaround times based on the given time quantum.

Ex. No. : 7 **IPC USING SHARED MEMORY**

Date: ***19.02.2025***

Aim:

To write a C program to implement Inter Process Communication (IPC) using shared memory between sender and receiver processes.

Algorithm:

Sender Process

1. Set the size of the shared memory segment.
2. Allocate the shared memory segment using shmget().
3. Attach the shared memory segment using shmat().
4. Write a string to the shared memory segment using sprintf().
5. Set delay using sleep().
6. Detach shared memory segment using shmdt().

Receiver Process

1. Set the size of the shared memory segment.
2. Allocate the shared memory segment using shmget().
3. Attach the shared memory segment using shmat().
4. Print the shared memory contents sent by the sender process.
5. Detach shared memory segment using shmdt().

Program Code:

sender.c

```
#include <stdio.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <unistd.h>

#include <string.h>
```

```c
int main() {
    key_t key = ftok("shmfile",65); // Generate unique key
    int shmid = shmget(key, 1024, 0666|IPC_CREAT); // Create shared memory
    char *str = (char*) shmat(shmid, (void*)0, 0); // Attach to shared memory

    sprintf(str, "Welcome to Shared Memory");
    printf("Message Sent: %s\n", str);

    sleep(5); // Delay to allow receiver to read
    shmdt(str); // Detach from shared memory

    return 0;
}
```

receiver.c

```c
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>

int main() {
    key_t key = ftok("shmfile",65); // Generate same key
    int shmid = shmget(key, 1024, 0666|IPC_CREAT); // Access shared memory
    char *str = (char*) shmat(shmid, (void*)0, 0); // Attach to shared memory

    printf("Message Received: %s\n", str);

    shmdt(str); // Detach from shared memory
    shmctl(shmid, IPC_RMID, NULL); // Destroy the shared memory
```

```
    return 0;
}
```

Sample Output:

Terminal 1:

[root@localhost student]# gcc sender.c -o sender

[root@localhost student]# ./sender

Message Sent: Welcome to Shared Memory

Terminal 2:

[root@localhost student]# gcc receiver.c -o receiver

[root@localhost student]# ./receiver

Message Received: Welcome to Shared Memory

Result:

Thus, the C program for Inter Process Communication (IPC) using shared memory was successfully executed, and the message was successfully passed from the sender process to the receiver process.

Ex. No.: 8 **PRODUCER CONSUMER USING SEMAPHORES**

Date: ***25.3.2025***

Aim:

To write a C program to implement a solution to the Producer-Consumer problem using semaphores.

Algorithm:

1. Initialize semaphores empty, full, and mutex.
2. Create two threads — one for the producer and another for the consumer.
3. Use pthread_create to create threads and pthread_join to wait for them to finish.
4. In each thread, use sem_wait() on empty and then on mutex before entering the critical section.
5. Produce or consume the item inside the critical section.
6. After the critical section, call sem_post() on mutex and then full (producer) or empty (consumer).
7. Let the threads alternate based on buffer availability.
8. Exit the loop after 10 iterations for both producer and consumer.
9. Terminate the program.

Program Code:

```
#include <stdio.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>


#define SIZE 5


int buffer[SIZE];

int in = 0, out = 0, item = 0;
```

```
sem_t empty, full, mutex;

void* producer(void* arg) {
    for (int i = 0; i < 10; i++) {
        sem_wait(&empty);
        sem_wait(&mutex);

        item++;
        buffer[in] = item;
        printf("Producer produces the item %d\n", item);
        in = (in + 1) % SIZE;

        sem_post(&mutex);
        sem_post(&full);
        sleep(1);
    }
    return NULL;
}

void* consumer(void* arg) {
    for (int i = 0; i < 10; i++) {
        sem_wait(&full);
        sem_wait(&mutex);

        int consumed_item = buffer[out];
        printf("Consumer consumes item %d\n", consumed_item);
        out = (out + 1) % SIZE;
```

```c
        sem_post(&mutex);
        sem_post(&empty);
        sleep(1);
    }
    return NULL;
}

int main() {
    pthread_t prod, cons;

    sem_init(&empty, 0, SIZE);
    sem_init(&full, 0, 0);
    sem_init(&mutex, 0, 1);

    int choice;
    while (1) {
        printf("1. Producer\n2. Consumer\n3. Exit\nEnter your choice: ");
        scanf("%d", &choice);
        if (choice == 1) {
            pthread_create(&prod, NULL, producer, NULL);
            pthread_join(prod, NULL);
        } else if (choice == 2) {
            pthread_create(&cons, NULL, consumer, NULL);
            pthread_join(cons, NULL);
        } else {
            break;
        }
    }
```

```
    sem_destroy(&empty);
    sem_destroy(&full);
    sem_destroy(&mutex);

    return 0;
}
```

Sample Output:

1. Producer

2. Consumer

3. Exit

Enter your choice: 1

Producer produces the item 1

Enter your choice: 2

Consumer consumes item 1

Enter your choice: 2

Buffer is empty!!

Enter your choice: 1

Producer produces the item 1

Enter your choice: 1

Producer produces the item 2

Enter your choice: 1

Producer produces the item 3

Enter your choice: 1

Buffer is full!!

Enter your choice: 3

Result:

Thus, the Producer-Consumer problem was implemented successfully using semaphores in C, ensuring proper synchronization and avoiding race conditions.

Ex. No.: 9 **DEADLOCK AVOIDANCE**

Date: *01.04.2025*

Aim:

To find out a safe sequence using Banker's Algorithm for deadlock avoidance.

Algorithm:

1. Initialize work = available and finish[i] = false for all processes i.
2. Find an i such that both:
   - finish[i] == false and
   - need[i] <= work
3. If no such i exists, go to step 6.
4. Update: work = work + allocation[i].
5. Set finish[i] = true and go to step 2.
6. If finish[i] == true for all i, then a safe sequence exists. Print the safe sequence.
7. Else, print that no safe sequence exists (i.e., deadlock may occur).

Program Code (bankers.c):

```
#include <stdio.h>
#define P 5
#define R 3

int main() {
    int allocation[P][R] = {{0, 1, 0}, {2, 0, 0}, {3, 0, 2}, {2, 1, 1}, {0, 0, 2}};
    int max[P][R] = {{7, 5, 3}, {3, 2, 2}, {9, 0, 2}, {2, 2, 2}, {4, 3, 3}};
    int available[R] = {3, 3, 2};

    int need[P][R], finish[P] = {0}, safeSeq[P];
```

```
int work[R];

// Calculate Need matrix
for (int i = 0; i < P; i++)
    for (int j = 0; j < R; j++)
        need[i][j] = max[i][j] - allocation[i][j];

// Initialize work as available
for (int i = 0; i < R; i++)
    work[i] = available[i];

int count = 0;
while (count < P) {
    int found = 0;
    for (int i = 0; i < P; i++) {
        if (!finish[i]) {
            int j;
            for (j = 0; j < R; j++)
                if (need[i][j] > work[j])
                    break;
            if (j == R) {
                for (int k = 0; k < R; k++)
                    work[k] += allocation[i][k];
                safeSeq[count++] = i;
                finish[i] = 1;
                found = 1;
            }
        }
    }
```

```
        if (!found) {
            printf("System is not in a safe state.\n");
            return 1;
        }
    }

    printf("The SAFE Sequence is:\n");
    for (int i = 0; i < P; i++)
        printf("P%d ", safeSeq[i]);
    printf("\n");

    return 0;
}
```

Sample Output:

The SAFE Sequence is:

P1 P3 P4 P0 P2

Result:

Thus, the Banker's Algorithm was successfully implemented to determine the safe sequence for deadlock avoidance.

Ex. No.: 10a **BEST FIT**

Date: ***2.4.2025***

Aim:

To implement the Best Fit memory allocation technique using Python.

Algorithm:

1. Input memory blocks and processes with their sizes.
2. Initialize all memory blocks as free.
3. For each process, find the smallest memory block that can accommodate it.
4. If such a block is found, allocate it to the process.
5. If no suitable block is found, leave the process unallocated.

Program Code (best_fit.py):

```
def best_fit(blockSize, processSize):
    allocation = [-1] * len(processSize)

    for i in range(len(processSize)):
        best_idx = -1
        for j in range(len(blockSize)):
            if blockSize[j] >= processSize[i]:
                if best_idx == -1 or blockSize[j] < blockSize[best_idx]:
                    best_idx = j
        if best_idx != -1:
            allocation[i] = best_idx + 1
            blockSize[best_idx] -= processSize[i]

    print("Process No.\tProcess Size\tBlock No.")
    for i in range(len(processSize)):
        print(f"{i + 1}\t\t{processSize[i]}\t\t", end="")
```

```
        if allocation[i] != -1:
            print(f"{allocation[i]}")
        else:
            print("Not Allocated")

# Example usage
blockSize = [100, 500, 200, 300, 600]
processSize = [212, 417, 112, 426]

best_fit(blockSize, processSize)
```

Sample Output:

Process No.	Process Size	Block No.
1	212	4
2	417	2
3	112	3
4	426	5

Result:

Thus, the Best Fit memory allocation technique was successfully implemented in Python.

Ex. No.: 10b **FIRST FIT**

Date: ***2.4.2025***

Aim:

To write a C program for implementation of memory allocation methods for fixed partition using First Fit.

Algorithm:

1. Define the maximum limit as #define max 25.
2. Declare variables: frag[max], b[max], f[max], i, j, nb, nf, temp, bf[max], ff[max].
3. Input the number of blocks (nb) and files (nf).
4. Input the size of each block and file using loops.
5. For each file, search for the first block that is free and large enough to accommodate it.
6. If found, allocate that block to the file and calculate internal fragmentation.
7. Mark the block as used.
8. Print the allocated block and fragmentation details.

Program Code (first_fit.c):

```
#include <stdio.h>
#define max 25

int main() {
   int frag[max], b[max], f[max], i, j, nb, nf, temp;
   static int bf[max], ff[max];

   printf("Enter number of blocks: ");
   scanf("%d", &nb);

   printf("Enter number of files: ");
   scanf("%d", &nf);
```

```c
printf("\nEnter size of each block:\n");
for(i = 0; i < nb; i++) {
    printf("Block %d: ", i + 1);
    scanf("%d", &b[i]);
}

printf("\nEnter size of each file:\n");
for(i = 0; i < nf; i++) {
    printf("File %d: ", i + 1);
    scanf("%d", &f[i]);
}

for(i = 0; i < nf; i++) {
    for(j = 0; j < nb; j++) {
        if(bf[j] != 1 && b[j] >= f[i]) {
            ff[i] = j;
            frag[i] = b[j] - f[i];
            bf[j] = 1;
            break;
        }
    }
}

printf("\nFile No\tFile Size\tBlock No\tBlock Size\tFragment\n");
for(i = 0; i < nf; i++)
    printf("%d\t%d\t\t%d\t\t%d\t\t%d\n", i+1, f[i], ff[i]+1, b[ff[i]], frag[i]);

return 0;
```

}

Sample Output:

Enter number of blocks: 5

Enter number of files: 4

Enter size of each block:

Block 1: 100

Block 2: 500

Block 3: 200

Block 4: 300

Block 5: 600

Enter size of each file:

File 1: 212

File 2: 417

File 3: 112

File 4: 426

File No File Size Block No Block Size Fragment

1 212 2 500 288

2 417 5 600 183

3 112 3 200 88

4 426 0 0 0 <-- Not allocated

Result:

Thus, the First Fit memory allocation technique for fixed partitioning was implemented successfully in C.

Ex. No.: 11a **FIFO Page Replacement**

Date: ***15.04.2025***

Aim:

To find out the number of page faults that occur using First-in First-out (FIFO) page replacement technique.

Algorithm:

1. Start the process.
2. Declare the page frame size and reference string length.
3. Read the reference string values.
4. Check each page:
    - If the page is not in memory, it's a page fault.
    - If memory is full, remove the oldest page (FIFO) and insert the new one.
5. Count the total number of page faults.
6. Display the frame content after each insertion and total faults.
7. Stop the process.

C Program:

```
#include <stdio.h>

int main() {

  int refStr[50], frames[10], n, f, i, j, k, pageFaults = 0, index = 0, found;

  printf("Enter the size of reference string: ");

  scanf("%d", &n);

  printf("Enter the reference string:\n");

  for(i = 0; i < n; i++) {

    printf("Enter [%d] : ", i+1);

    scanf("%d", &refStr[i]);

  }

  printf("Enter number of frames: ");

  scanf("%d", &f);
```

```c
for(i = 0; i < f; i++)
    frames[i] = -1;
printf("\nPage Replacement Process:\n");
for(i = 0; i < n; i++) {
    found = 0;
    for(j = 0; j < f; j++) {
        if(frames[j] == refStr[i]) {
            found = 1;
            break;
        }
    }
    if(!found) {
        frames[index] = refStr[i];
        index = (index + 1) % f;
        pageFaults++;
        for(k = 0; k < f; k++) {
            if(frames[k] != -1)
                printf("%d ", frames[k]);
            else
                printf("- ");
        }
        printf("\n");
    } else {
        printf("No Page Fault\n");
    }
}
printf("\nTotal Page Faults = %d\n", pageFaults);
return 0;
```

}

Sample Output:

Enter the size of reference string: 6

Enter the reference string:

Enter [1] : 5

Enter [2] : 7

Enter [3] : 5

Enter [4] : 6

Enter [5] : 7

Enter [6] : 3

Enter number of frames: 3

Page Replacement Process:

5 - -

5 7 -

No Page Fault

5 7 6

No Page Fault

3 7 6

Total Page Faults = 4

Result:

Thus, the program for FIFO page replacement was written and executed successfully. The number of page faults was calculated and verified.

Ex. No.: 11b **LRU Page Replacement**

Date: ***15.04.2025***

Aim:

To write a C program to implement LRU page replacement algorithm.

Algorithm:

1. Start the process.
2. Declare the size for page frames.
3. Get the number of pages and reference string.
4. Use a stack or counter array to track recent usage.
5. For each page:
    - If it is in memory → no page fault.
    - Else → check least recently used page and replace it.
6. Count page faults.
7. Display frame contents after each operation.
8. Stop the process.

C Program:

```
#include <stdio.h>


int findLRU(int time[], int n) {
   int i, minimum = time[0], pos = 0;
   for(i = 1; i < n; i++) {
      if(time[i] < minimum) {
         minimum = time[i];
         pos = i;
      }
   }
   return pos;
```

```
}

int main() {
    int frames[10], pages[50], time[10], counter = 0, pageFaults = 0;
    int n, f, i, j, pos, flag1, flag2;

    printf("Enter number of frames: ");
    scanf("%d", &f);

    printf("Enter number of pages: ");
    scanf("%d", &n);

    printf("Enter reference string: ");
    for(i = 0; i < n; i++)
        scanf("%d", &pages[i]);

    for(i = 0; i < f; i++)
        frames[i] = -1;

    for(i = 0; i < n; i++) {
        flag1 = flag2 = 0;

        for(j = 0; j < f; j++) {
            if(frames[j] == pages[i]) {
                counter++;
                time[j] = counter;
                flag1 = flag2 = 1;
                break;
            }
```

```
}

if(flag1 == 0) {
    for(j = 0; j < f; j++) {
        if(frames[j] == -1) {
            counter++;
            pageFaults++;
            frames[j] = pages[i];
            time[j] = counter;
            flag2 = 1;
            break;
        }
    }
}

if(flag2 == 0) {
    pos = findLRU(time, f);
    counter++;
    pageFaults++;
    frames[pos] = pages[i];
    time[pos] = counter;
}

for(j = 0; j < f; j++) {
    if(frames[j] != -1)
        printf("%d ", frames[j]);
    else
        printf("- ");
}
```

```c
        printf("\n");
    }

    printf("\nTotal Page Faults = %d\n", pageFaults);
    return 0;
}
```

Sample Output:

Enter number of frames: 3

Enter number of pages: 6

Enter reference string: 5 7 5 6 7 3

5 - -

5 7 -

5 7 -

5 7 6

5 7 6

3 7 6

Total Page Faults = 4

Result:

Thus, the C program for LRU page replacement algorithm was written and executed successfully. The number of page faults was calculated and verified.

Ex. No.: 11c **Optimal Page Replacement Algorithm**

Date: ***15.04.2025***

Aim:

To write a C program to implement Optimal page replacement algorithm.

Algorithm:

1. Start the process
2. Declare the number of page frames
3. Get the number of pages and the reference string
4. For each page reference:
    - If the page is in memory, do nothing
    - Else if there is space in a frame, insert the page
    - Else find the page not used for the longest future duration, and replace it
5. Count and display page faults
6. Display frame contents after each operation
7. Stop the process

C Program:

```
#include <stdio.h>

int search(int key, int frame[], int n) {

    for(int i = 0; i < n; i++) {

        if(frame[i] == key)

            return 1;

    }

    return 0;

}

int predict(int pages[], int frame[], int n, int index, int f) {

    int res = -1, farthest = index;

    for(int i = 0; i < f; i++) {
```

```
        int j;
        for(j = index; j < n; j++) {
            if(frame[i] == pages[j]) {
                if(j > farthest) {
                    farthest = j;
                    res = i;
                }
                break;
            }
        }
        if(j == n)
            return i;
    }
    return (res == -1) ? 0 : res;
}
int main() {
    int n, f, pages[50], frame[10];
    int i, j, pageFaults = 0;
    printf("Enter number of frames: ");
    scanf("%d", &f);
    printf("Enter number of pages: ");
    scanf("%d", &n);
    printf("Enter reference string: ");
    for(i = 0; i < n; i++)
        scanf("%d", &pages[i]);
    for(i = 0; i < f; i++)
        frame[i] = -1;
    for(i = 0; i < n; i++) {
        if(search(pages[i], frame, f) == 0) {
```

```
            if(j < f)
                frame[j++] = pages[i];
            else {
                int pos = predict(pages, frame, n, i + 1, f);
                frame[pos] = pages[i];
            }
            pageFaults++;
        }
        for(int k = 0; k < f; k++) {
            if(frame[k] != -1)
                printf("%d ", frame[k]);
            else
                printf("- ");
        }
        printf("\n");
    }
    printf("\nTotal Page Faults = %d\n", pageFaults);
    return 0;
}
```

Sample Output:

Enter number of frames: 3

Enter number of pages: 6

Enter reference string: 5 7 5 6 7 3

5 - -

5 7 -

5 7 -

5 7 6

5 7 6

3 7 6

Total Page Faults = 4

Result:

Thus, the C program to implement the Optimal page replacement algorithm was successfully written and executed. The number of page faults was calculated and verified.