K.S.R.M. COLLEGE OF ENGINEERING
(UGC - Autonomous)
Approved by AICTE, New Delhi & Affiliated to JNTUA, Ananthapuramu
Accredited by NAAC with A+ Grade & B.Tech. (EEE, ECE, CSE, CE and ME) Programs by NBA

Department of CSE & Allied Branches

(2305451) OPERATING SYSTEMS LAB

B.Tech. IV Semester (R23UG)

(Common to CSE & Allied Branches)

Lab Manual

Academic Year: _____

Name : _____

Roll. Number : _____

Sem & Branch : _____

K.S.R.M. COLLEGE OF ENGINEERING

(UGC-AUTONOMOUS)

Kadapa, Andhra Pradesh, India– 516 005

Approved by AICTE, New Delhi & Affiliated to JNTUA, Ananthapuramu.

An ISO 14001:2004 & 9001: 2015 Certified Institution

DEPARTMENT OF AI&ML

****

INSTITUTE VISION:

To evolve as center of repute for providing quality academic programs amalgamated with creative learning and research excellence to produce graduates with leadership qualities, ethical and human values to serve the nation.

INSTITUTE MISSION:

M1: To provide high quality education with enriched curriculum blended with impactful teaching-learning practices.

M2: To promote research, entrepreneurship and innovation through industry collaborations.

M3: To produce highly competent professional leaders for contributing to Socio-economic development of region and the nation.

DEPARTMENT VISION:

To become a renowned department by offering industry need education with upgrading technology, nurturing collaborative culture & modelling the students in global professions with ethical and leadership sprit.

DEPARTMENT MISSION:

- To produce globally competent and qualified professionals in the areas of AI & ML
- To impart knowledge in cutting edge Artificial Intelligence technologies in par with industrial standards.
- To encourage students to engage in life-long learning by creating awareness of the contemporary developments in Artificial Intelligence and Machine Learning.

PROGRAMME EDUCATIONAL OBJECTIVES (PEOs):

PEO1 - Technical & Employability skills: To prepare the students to meet industrial needs and embed the knowledge so that the students become efficient in the design and development of required software and create solutions for automation of real time scenarios throughout their career.

PEO2 - Leadership Quality: To instil confidence and train the students in enhancing soft skills, leadership abilities to understand the ethical and social responsibilities in their professional lives as successful entrepreneurs

PEO3 - Problem Solving: To attain technical knowledge, using modern tools on the latest technologies and professionally advanced, the result makes the student to solve complex technical issues.

PEO4 - Professional Ethics: To prepare the students work in multidisciplinary teams on problems whose solutions lead to significant societal benefit.

PROGRAMME OUTCOMES (POs):

PO1 - Engineering Knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2 - Problem Analysis: Identify, formulate, review research literature, and analyse complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

PO3 - Design/Development of solutions: Design solutions for complex engineering problems and design system components or pro that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

PO4 - Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5 - Modern tool usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.

PO6 - The engineer and society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.

PO7 - Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

PO8 - Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of engineering practice.

PO9 - Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10 - Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

PO11 - Project management and finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12 - Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAMME SPECIFIC OUTCOMES (PSOs):

PSO1 - Develop an in-depth knowledge and skill set in human cognition, Artificial Intelligence, Machine Learning and Data engineering for designing intelligent systems to address modern computing challenges.

PSO2 - Evaluate, analyse and synthesize solutions for real time problems in Artificial Intelligence and Machine Learning domain to conduct research in a wider theoretical and practical context.

PSO3 - Do innovative system design with analytical knowledge by developing modern tools and techniques.

K.S.R.M. COLLEGE OF ENGINEERING
(AUTONOMOUS)

DEPARTMENT OF AI&ML

GENERAL LABORATORY INSTRUCTIONS

1. Students are advised to come to the laboratory at least 5 minutes before (to starting time), those who come after 5 minutes will not be allowed into the lab.
2. Plan your task properly much before to the commencement, come prepared to the lab with the program / experiment details.
3. Student should enter into the laboratory with:
    1. Laboratory observation notes with all the details (Problem statement, Aim, Algorithm, Procedure, Program, Expected Output, etc.,) filled in for the lab session.
    2. Laboratory Record updated up to the last session experiments and other utensils (if any) needed in the lab.
    3. Proper dress code and identity card.
4. Sign in the laboratory login register, write the **TIME-IN** and **System number**, and occupy the computer system allotted to you by the faculty.
5. Execute your task in the laboratory, and record the results / output in the lab observation note book, and get certified by the concerned faculty.
6. All the students should be polite and cooperative with the laboratory staff, must maintain the discipline and decency in the laboratory.
7. Computer labs are established with sophisticated and high end branded systems, which should be utilized properly.
8. Students / Faculty must keep their mobile phones in **SWITCHED OFF** mode during the lab sessions. Misuse of the equipment, misbehaviours with the staff and systems etc., will attract severe punishment.
9. Students must take the permission of the faculty in case of any urgency to go out; if anybody found loitering outside the lab / class without permission during working hours will be treated seriously and punished appropriately.
10. Students should **LOG OFF/ SHUT DOWN** the computer system before he/she leaves the lab after completing the task (experiment) in all aspects. He/she must ensure the system / seat is kept properly.

Head of the Department

OPERATING SYSTEMS LAB (Professional Core)								
Course Code	**Category**	**Hours/Week**			**Credits**	**Maximum Marks**		
2305451	**Engineering**	**L**	**T**	**P**	**C**	**Continuous Internal Assessment**	**Sem.-End Exam**	**Total**
		0	**0**	**3**	**1.5**	**30**	**70**	**100**

Pre-Requisites: Nil

Course Objectives:

- **CEO3.** Provide insights into system calls, file systems, semaphores,
- **CEO1.** Develop and debug CPU Scheduling algorithms, page replacement algorithms.
- **CEO2.** Implement Bankers Algorithms to Avoid the Dead Lock

Course Outcomes: On successful completion of this course, the students will be able to

- **CO1.** Trace different CPU Scheduling algorithms (L2).
- **CO2.** Implement Bankers Algorithms to Avoid the Dead Lock (L3).
- **CO3.** Evaluate Page replacement algorithms (L5).
- **CO4.** Illustrate the file organization techniques (L4).
- **CO5.** Illustrate Inter process Communication (L4)

List of Exercises/List of Experiments

1. Practicing of Basic UNIX Commands.
2. Simulate UNIX commands like cp, ls, grep, etc.,
3. Simulate the following CPU scheduling algorithms

   a) FCFS b) SJF c) Priority d) Round Robin
4. Write a program to illustrate concurrent execution of threads using pthreads library.
5. Write a program to solve producer-consumer problem using Semaphores.
6. Implement the following memory allocation methods for fixed partition

   a) First fit b) Worst fit c) Best fit
7. Simulate the following page replacement algorithms

   a) FIFO b) LRU c) LFU
8. Simulate Paging Technique of memory management.
9. Implement Bankers Algorithm for Dead Lock avoidance and prevention
10. Simulate the following file allocation strategies

    a) Sequential b) Indexed c) Linked

REFERENCE BOOKS/LABORATORY MANUALS

1. Operating System Concepts, Silberschatz A, Galvin P B, Gagne G, 10th Edition, Wiley,2018.
2. Modern Operating Systems, Tanenbaum A S, 4th Edition, Pearson, 2016
3. Operating Systems -Internals and Design Principles, Stallings W, 9th edition, Pearson,2018
4. Operating Systems: A Concept Based Approach, D.M Dhamdhere, 3rd Edition,McGraw- Hill, 2013

INDEX

S.No.	Name of the experiment		Page No.	Faculty Signature
1				
2				
3				
4				
5				
6				
7				

8				
9				

Exercise 1	**Practicing of Basic UNIX Commands**	**Date:**

Practicing basic UNIX commands is essential for anyone working with Linux or UNIX-based systems. Below are some essential commands categorized for better understanding:

---

1. File and Directory Management

- `pwd` – Print the current working directory.
  - `pwd`
- `ls` – List files and directories in the current location.
  - `ls -l  # List in long format`
  - `ls -a  # Show hidden files`
- `cd` – Change directory.
  - `cd /path/to/directory`
- `mkdir` – Create a new directory.
  - `mkdir new_folder`
- `rmdir` – Remove an empty directory.
  - `rmdir folder_name`
- `rm` – Remove files or directories.
  - `rm file.txt  # Remove file`
  - `rm -r folder_name  # Remove directory and contents`
- `cp` – Copy files or directories.
  - `cp file1.txt file2.txt`
  - `cp -r dir1 dir2  # Copy directory`
- `mv` – Move or rename files.
  - `mv oldname.txt newname.txt  # Rename file`
  - `mv file.txt /new/location/  # Move file`

---

2. File Viewing and Editing

- `cat` – View file contents.
  - `cat file.txt`
- `less` – View file page by page.
  - `less file.txt`
- `head` – Display first 10 lines of a file.

- `head file.txt`

- `tail` – Display last 10 lines of a file.
  - `tail file.txt`
- `nano`, `vim`, `vi` – Edit files.
  - `nano file.txt`
  - `vim file.txt`

---

3. File Permissions and Ownership

- `ls -l` – Check file permissions.
- `chmod` – Change file permissions.

```
chmod 755 file.sh  # Read/write/execute for owner, read/execute for others
chmod u+x file.sh  # Give execute permission to the owner
```

- `chown` – Change file ownership.
  - `chown user:group file.txt`

---

4. Process Management

- `ps` – Display running pro.
  - `ps aux`
- `top` – Show real-time system resource usage.
- `kill` – Terminate a process by PID.
  - `kill 1234  # Replace 1234 with the PID`
- `killall` – Kill all pro by name.
  - `killall firefox`
- `htop` – Interactive process manager (if installed).
  - `htop`

---

5. Disk and Storage Management

- `df` – Show disk usage.
  - `df -h`
- `du` – Show directory size.
  - `du -sh /path/to/directory`

---

6. Networking Commands

- `ping` – Check network connectivity.
  - `ping google.com`
- `ifconfig` or `ip` – Display network information.
  - `ifconfig`
  - `ip addr show`
- `netstat` – Display active network connections.
- `wget` – Download files from the internet.
  - `wget http://example.com/file.zip`
- `curl` – Transfer data from a URL.
  - `curl http://example.com`

---

7. Searching and Finding Files

- `find` – Search for files in directories.
  - `find /home -name "*.txt"`
- `grep` – Search within files.
  - `grep "word" file.txt`
  - `grep -r "word" /path/`
- `locate` – Quickly find a file.
  - `locate file.txt`

---

8. User and System Management

- `whoami` – Show current logged-in user.
- `who` – Show all users currently logged in.
- `id` – Show user ID and group.
- `passwd` – Change user password.
  - `passwd`
- `uptime` – Show system uptime.
- `date` – Display current date and time.
  - `date`
- `history` – Show command history.

---

9. Archiving and Compression

- `tar` – Archive and extract files.
  - `tar -cvf archive.tar directory/`
  - `tar -xvf archive.tar`
- `zip` / `unzip` – Compress and extract files.
  - `zip -r archive.zip folder/`
  - `unzip archive.zip`

---

10. Scripting & Automation

- Create and run a simple shell script.
  - `nano script.sh`

  Add the following:

```
#!/bin/bash
echo "Hello, UNIX!"
```

  Make it executable and run:

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

How to Practice?

1. **Use a Virtual Machine** – Install Ubuntu or CentOS in VirtualBox.
2. **Practice on WSL (Windows Subsystem for Linux)** – Enable Linux on Windows.
3. **Try Online UNIX Terminals** – Websites like JSLinux allow online practice.
4. **Write Scripts** – Automate simple tasks like file management.

You can practice UNIX commands online using these web-based UNIX/Linux terminals:

1. JS Linux

🔗 https://bellard.org/jslinux/

- A simple in-browser Linux emulator.
- No registration required.

2. Webminal – Online Linux Terminal

🔗 https://www.webminal.org/

- Best for beginners to practice commands.
- Requires free registration.

3. JDOODLE – Online Bash Compiler

🔗 https://www.jdoodle.com/linux-bash-online-shell/

- Run Bash scripts online.
- Supports file handling.

4. TutorialsPoint Linux Terminal

🔗 https://www.tutorialspoint.com/terminal/online-linux-terminal.php

- Browser-based Linux terminal.
- Supports command execution and basic scripting.

5. CoCalc (Collaborative Linux Terminal)

🔗 https://cocalc.com/

- Great for team collaboration.
- Requires an account.

Exercise 2	Simulate UNIX commands like cp, ls, grep, etc.,	Date:

I can simulate UNIX commands like `ls`, `cp`, `grep`, etc., in a controlled environment. Below is an interactive simulation where you can specify a command, and I'll show the expected output.

Simulating UNIX Commands

Assume the following directory structure:

```
/home/user/
├── documents/
│   ├── report.txt
│   ├── notes.txt
├── scripts/
│   ├── backup.sh
│   ├── test.sh
├── pictures/
│   ├── photo1.jpg
│   ├── photo2.jpg
└── log.txt
```

1. Listing Files (`ls`)

```
A)      $ ls
        documents  log.txt  pictures  scripts

B)      $ ls -l
        drwxr-xr-x  2 user user 4096 Feb 06 10:00 documents
        -rw-r—r --  1 user user 1234 Feb 06 10:05 log.txt
        drwxr-xr-x  2 user user 4096 Feb 06 10:00 pictures
        drwxr-xr-x  2 user user 4096 Feb 06 10:00 scripts
```

---

2. Copying a File (`cp`)

```
$ cp log.txt documents/copy_log.txt
$ ls documents
copy_log.txt  notes.txt  report.txt
```

---

3. Moving/Renaming a File (`mv`)

```
$ mv log.txt backup_log.txt
$ ls
backup_log.txt  documents  pictures  scripts
```

---

4. Searching for a Word in a File (`grep`)

```
$ grep "error" log.txt
[ERROR] Disk space is low.
[ERROR] Failed to load configuration.
```

---

5. Finding a File (`find`)

```
$ find . -name "*.sh"
./scripts/backup.sh
./scripts/test.sh
```

---

6. Displaying First/Last 3 Lines (`head`, `tail`)

```
$ head -3 log.txt
[INFO] System started.
[INFO] User logged in.
[WARNING] High memory usage detected.

$ tail -3 log.txt
[INFO] Shutting down.
[WARNING] Network latency detected.
[ERROR] Connection lost.
```

---

7. Checking Disk Usage (`df`, `du`)

```
$ df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/sda1       50G   20G   30G  40% /

$ du -sh documents
4.0K    documents
```

Exercise 3	**Simulate the following CPU scheduling algorithms:** a) **FCFS b) SJF c) Priority d) Round Robin**	**Date:**

i. FIRST COME FIRST SERVE:

AIM: To write a C program to simulate the CPU scheduling algorithm First Come FirstServe (FCFS)

DESCRIPTION:

To calculate the average waiting time using the FCFS algorithm first the waiting time of the first process is kept zero and the waiting time of the second process is the burst time of the first process and the waiting time of the third process is the sum of the burst times of the first and the second process and so on. After calculating all the waiting times the average waiting time is calculated as the average of all the waiting times. FCFS mainly says first come first serve the algorithm which came first will be served first.

ALGORITHM:

Step 1: Start the process

Step 2: Accept the number of pro in the ready Queue

Step 3: For each process in the ready Q, assign the process name and the burst time

Step 4: Set the waiting of the first process as _0'and its burst time as its turnaround time

Step 5: for each process in the Ready Q calculate

a). Waiting time (n) = waiting time (n-1) + Burst time (n-1)

b) Turnaround time (n)= waiting time(n)+Burst time(n)

Step 6: Calculate

a) Average waiting time = Total waiting Time / Number of process

b) Average Turnaround time = Total Turnaround Time / Number of process

Step 7: Stop the process

SOURCE CODE:

```
#include<stdio.h>
#include<conio.h>
main()
{
int bt[20], wt[20], tat[20], i, n;
 float wtavg, tatavg;
clrscr();
printf("\nEnter the number of processes -- ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
    printf("\nEnter Burst Time for Process %d -- ", i);
    scanf("%d", &bt[i]);
}
wt[0] = wtavg = 0;
tat[0] = tatavg = bt[0];
for(i=1;i<n;i++)
{
    wt[i] = wt[i-1] +bt[i-1];
    tat[i] = tat[i-1] +bt[i];
    wtavg = wtavg + wt[i];
    tatavg = tatavg + tat[i];
}
printf("\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
for(i=0;i<n;i++)
        printf("\n\t P%d \t\t %d \t\t %d \t\t %d", i, bt[i], wt[i], tat[i]);
        printf("\nAverage Waiting Time -- %f", wtavg/n);
printf("\nAverage Turnaround Time -- %f", tatavg/n);
getch();
}
```

INPUT

Enter the number of pro -- 3
Enter Burst Time for Process 0 -- 24
Enter Burst Time for Process 1 -- 3
Enter Burst Time for Process 2 -- 3

OUTPUT

PROCESS	BURST TIME	WAITING TIME	TURNAROUND TIME
P0	24	0	24
P1	3	24	27
P2	3	27	30

Average Waiting Time-- 17.000000
Average Turnaround Time -- 27.000000

i. SHORTEST JOB FIRST:

AIM: To write a program to stimulate the CPU scheduling algorithm Shortest Job First (Non-Preemption)

DESCRIPTION:

To calculate the average waiting time in the shortest job first algorithm the sorting of the process based on their burst time in ascending order then calculate the waiting time of each process as the sum of the bursting times of all the process previous or before to that process.

ALGORITHM:

Step 1: Start the process
Step 2: Accept the number of processes in the ready Queue
Step 3: For each process in the ready Q, assign the process id and accept the CPU burst time
Step 4: Start the Ready Q according the shortest Burst time by sorting according to lowest to highest burst time.
Step 5: Set the waiting time of the first process as ‗0' and its turnaround time as its burst time.
Step 6: Sort the processes names based on their Burt time
Step 7: For each process in the ready queue, calculate
a) Waiting time(n)= waiting time (n-1) + Burst time (n-1)
b) Turnaround time (n)= waiting time(n)+Burst time(n)
Step 8: Calculate
c) Average waiting time = Total waiting Time / Number of process
d) Average Turnaround time = Total Turnaround Time / Number of process
Step 9: Stop the process

SOURCE CODE :

```
#include<stdio.h>
#include<conio.h>
main()
{
    int p[20], bt[20], wt[20], tat[20], i, k, n, temp;
    float wtavg, tatavg;
    clrscr();
    printf("\nEnter the number of pro -- "); scanf("%d", &n);
    for(i=0;i<n;i++)
    {
            p[i]=i;
            printf("Enter Burst Time for Process %d -- ", i);
            scanf("%d", &bt[i]);
    }
    for(i=0;i<n;i++)
    for(k=i+1;k<n;k++)
    if(bt[i]>bt[k])
    {
            temp=bt[i];
            bt[i]=bt[k];
            bt[k]=temp;
            temp=p[i];
            p[i]=p[k];
            p[k]=temp;
    }
    wt[0] = wtavg = 0;
    tat[0] = tatavg = bt[0];
    for(i=1;i<n;i++)
    {
    wt[i] = wt[i-1] +bt[i-1];
    tat[i] = tat[i-1] +bt[i];
```

```
wtavg = wtavg + wt[i];
tatavg = tatavg + tat[i];
}
printf("\n\t PROCESS \tBURST TIME \t WAITING TIME\t TURNAROUND TIME\n");
for(i=0;i<n;i++)
printf("\n\t P%d \t\t %d \t\t %d \t\t %d", p[i], bt[i], wt[i], tat[i]);
printf("\nAverage Waiting Time -- %f", wtavg/n);
printf("\nAverage Turnaround Time -- %f", tatavg/n);
getch();
}
```

INPUT

```
Enter the number of pro --          4
Enter Burst Time for Process 0 --   6
Enter Burst Time for Process 1 --   8
Enter Burst Time for Process 2 --   7
Enter Burst Time for Process 3 --   3
```

OUTPUT

PROCESS	BURST TIME	WAITING TIME	TURNAROUND TIME
P3	3	0	3
P0	6	3	9
P2	7	9	16
P1	8	16	24

```
Average Waiting Time --     7.000000
Average Turnaround Time --  13.000000
```

iii) ROUND ROBIN:

AIM: To simulate the CPU scheduling algorithm round-robin.

DESCRIPTION:

To aim is to calculate the average waiting time. There will be a time slice, each process should be executed within that time-slice and if not it will go to the waiting state so first check whether the burst time is less than the time-slice. If it is less than it assign the waiting time to the sum of the total times. If it is greater than the burst-time then subtract the time slot from the actual burst time and increment it by time-slot and the loop continues until all the pro are completed.

ALGORITHM:

Step 1: Start the process

Step 2: Accept the number of pro in the ready Queue and time quantum (or) time slice

Step 3: For each process in the ready Q, assign the process id and accept the CPU bursttime

Step 4: Calculate the no. of time slices for each process where No. of timeslice for process (n) = burst time process (n)/time slice

Step 5: If the burst time is less than the time slice then the no. of time slices =1.

Step 6: Consider the ready queue is a circular Q, calculate

a) Waiting time for process (n) = waiting time of process(n-1) + burst time ofprocess (n-1) + the time difference in getting the CPU from process (n-1)

b) Turnaround time for process(n) = waiting time of process(n) + burst time ofprocess(n) + the time difference in getting CPU from process(n).

Step 7: Calculate

a) Average waiting time = Total waiting Time / Number of process

b) Average Turnaround time = Total Turnaround Time / Number of process

Step8: Stop the process

SOURCE CODE

```
#include<stdio.h>
main()
{
        int i,j,n,bu[10],wa[10],tat[10],t,ct[10],max;
        float awt=0,att=0,temp=0;
        clrscr();
        printf("Enter the no of pro -- ");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("\nEnter Burst Time for process %d -- ", i+1);
                scanf("%d",&bu[i]);
                ct[i]=bu[i];
        }
        printf("\nEnter the size of time slice -- ");
        scanf("%d",&t);
        max=bu[0];
        for(i=1;i<n;i++)
        if(max<bu[i])
        max=bu[i];
        for(j=0;j<(max/t)+1;j++)
        for(i=0;i<n;i++)
        if(bu[i]!=0)
        if(bu[i]<=t)
        {
                tat[i]=temp+bu[i];
                temp=temp+bu[i];
                bu[i]=0;
        }
```

```
        else
        {
                bu[i]=bu[i]-t;
                temp=temp+t;
        }
        for(i=0;i<n;i++)
        {
                wa[i]=tat[i]-ct[i];
                att+=tat[i]; awt+=wa[i];
        }
printf("\nThe Average Turnaround time is -- %f",att/n);
printf("\nThe Average Waiting time is -- %f ",awt/n);
printf("\n\tPROCESS\t BURST TIME \t WAITING TIME\tTURN AROUND TIME\n");
for(i=0;i<n;i++)
printf("\t%d \t %d \t\t %d \t\t %d \n",i+1,ct[i],wa[i],tat[i]);
getch();
}
```

INPUT:

Enter the no of pro – 3

Enter Burst Time for process 1 – 24 Enter Burst Time for process 2 -- 3 Enter Burst Time for process 3 – 3 Enter the size of time slice – 3

OUTPUT:

PROCESS	BURST TIME	WAITING TIME	TURNAROUNDTIME
1	24	6	30
2	3	4	7
3	3	7	10

The Average Turnaround time is – 15.666667

The Average Waiting time is 5.666667

iv) PRIORITY:

AIM: To write a c program to simulate the CPU scheduling priority algorithm.

DESCRIPTION:

To calculate the average waiting time in the priority algorithm, sort the burst times according to their priorities and then calculate the average waiting time of the pro. The waiting time of each process is obtained by summing up the burst times of all the previous pro.

ALGORITHM:

Step 1: Start the process

Step 2: Accept the number of pro in the ready Queue

Step 3: For each process in the ready Q, assign the process id and accept the CPU bursttime

Step 4: Sort the ready queue according to the priority number.

Step 5: Set the waiting of the first process as _0' and its burst time as its turnaround timeStep 6: Arrange the pro based on process priority

Step 7: For each process in the Ready Q calculate

Step 8: for each process in the Ready Q calculate

a) Waiting time(n)= waiting time (n-1) + Burst time (n-1)

b) Turnaround time (n)= waiting time(n)+Burst time(n)

Step 9: Calculate

a) Average waiting time = Total waiting Time / Number of process

b) Average Turnaround time = Total Turnaround Time / Number of process

Print the results in an order.

Step10: Stop

SOURCE CODE:

```
#include<stdio.h>
main()
{
        int p[20],bt[20],pri[20], wt[20],tat[20],i, k, n, temp;
        float wtavg, tatavg;
        clrscr();
        printf("Enter the number of pro --- ");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                p[i] = i;
                printf("Enter the Burst Time & Priority of Process %d --- ",i);
                scanf("%d %d",&bt[i], &pri[i]);
        }
        for(i=0;i<n;i++)
        for(k=i+1;k<n;k++)
        if(pri[i] > pri[k])
        {
                temp=p[i];
                p[i]=p[k];
                p[k]=temp;
                temp=bt[i];
                bt[i]=bt[k];
                bt[k]=temp;
                temp=pri[i];
                pri[i]=pri[k];
                pri[k]=temp;
        }
        wtavg = wt[0] = 0;
        tatavg = tat[0] = bt[0];
        for(i=1;i<n;i++)
```

```
        {
                wt[i] = wt[i-1] + bt[i-1];
                tat[i] = tat[i-1] + bt[i];
                wtavg = wtavg + wt[i];
                tatavg = tatavg + tat[i];
        }
    printf("\nPROCESS\t\tPRIORITY\tBURST TIME\tWAITING TIME\tTURNAROUND TIME");
    for(i=0;i<n;i++)
    printf("\n%d \t\t %d \t\t %d \t\t %d \t\t %d ",p[i],pri[i],bt[i],wt[i],tat[i]); p
    rintf("\nAverage Waiting Time is --- %f",wtavg/n);
    printf("\nAverage Turnaround Time is --- %f",tatavg/n);
    getch();
    }
```

INPUT

```
Enter the number of pro -- 5
Enter the Burst Time & Priority of Process 0 --- 10        3
Enter the Burst Time & Priority of Process 1 --- 1         1
Enter the Burst Time & Priority of Process 2 --- 2         4
Enter the Burst Time & Priority of Process 3 --- 1         5
Enter the Burst Time & Priority of Process 4 --- 5         2
```

OUTPUT

PROCESS	PRIORITY	BURST TIME	WAITIN G TIME	TURNARO UND TIME
1	1	1	0	1
4	2	5	1	6
0	3	10	6	16
2	4	2	16	18
3	5	1	18	19

```
Average Waiting Time is ---              8.200000
Average Turnaround Time is ---------- 12.000000
```

VIVA QUESTIONS

1) Define the following
   a) Turnaround time b) Waiting time c) Burst time d) Arrival time
2) What is meant by process scheduling?
3) What are the various states of process?
4) What is the difference between preemptive and non-preemptive scheduling
5) What is meant by time slice?
6) What is round robin scheduling?

Exercise 4	Write a program to illustrate concurrent execution of threads using pthreads library.	Date:

Aim: Write a program to illustrate concurrent execution of threads using pthreads library.

Description:

The provided C program demonstrates concurrent execution of multiple threads using the POSIX Threads (pthread) library. The main objective of the program is to show how multiple threads can execute simultaneously, sharing CPU time and performing tasks concurrently.

- Threads: Lightweight pro that share the same memory space. Multiple threads can run concurrently within a single process.
- Concurrency: Multiple threads execute overlapping in time, which increases the efficiency and responsiveness of applications.
- pthread_create(): Creates a new thread.
- pthread_join(): Makes the main thread wait for the completion of other threads, ensuring orderly execution.
- sleep(): Simulates work by pausing the thread, allowing other threads to run and showcasing concurrent behavior.

Algorithm:

Main Program:

```
START
 SET NUM_THREADS to desired number of threads (e.g., 3)
 DECLARE threads array and thread_ids array

 FOR i FROM 0 TO NUM_THREADS - 1 DO
  ASSIGN thread_ids[i] = i + 1
  CALL pthread_create() to start thread_function with thread_ids[i]
  IF pthread_create() fails THEN
   PRINT "Failed to create thread"
   EXIT program with error
  END IF
 END FOR
```

```
FOR i FROM 0 TO NUM_THREADS - 1 DO
   CALL pthread_join() to wait for threads[i] to finish
   IF pthread_join() fails THEN
     PRINT "Failed to join thread"
     EXIT program with error
   END IF
  END FOR

  PRINT "Main thread: All threads have completed execution."
END
```

Thread function:

```
thread_function(thread_id):
  PRINT "Thread [thread_id]: Starting execution."

  FOR i FROM 1 TO 5 DO
    PRINT "Thread [thread_id]: Working... (i)"
    CALL sleep(1)  // Simulate work
  END FOR

  PRINT "Thread [thread_id]: Finishing execution."
  EXIT thread
```

SOURCE CODE:

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

// Function to be executed by each thread
void* thread_function(void* arg)
{
    int i;
    int thread_num = *((int*)arg);
    printf("Thread %d: Starting execution.\n", thread_num);

    // Simulate some work using sleep
    for (i = 0; i < 5; i++)
   {
        printf("Thread %d: Working... (%d)\n", thread_num, i + 1);
        sleep(1);  // Sleep for 1 second
    }

    printf("Thread %d: Finishing execution.\n", thread_num);
    pthread_exit(NULL);
}

int main()
{
    int i;
    const int NUM_THREADS = 3;
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];
```

```
    // Create threads
    for (i = 0; i < NUM_THREADS; i++)
{
        thread_ids[i] = i + 1;
        if (pthread_create(&threads[i], NULL, thread_function, (void*)&thread_ids[i]) != 0)
      {
            perror("Failed to create thread");
            return 1;
        }
    }

    // Wait for all threads to finish
    for (i = 0; i < NUM_THREADS; i++)
   {
        if (pthread_join(threads[i], NULL) != 0)
       {
            perror("Failed to join thread");
            return 1;
        }
    }

    printf("Main thread: All threads have completed execution.\n");
    return 0;
}
```

INPUT & OUTPUT:

```
Thread 3: Starting execution.
Thread 3: Working... (1)
Thread 1: Starting execution.
Thread 1: Working... (1)
Thread 2: Starting execution.
Thread 2: Working... (1)
Thread 2: Working... (2)
Thread 1: Working... (2)
Thread 3: Working... (2)
Thread 3: Working... (3)
Thread 2: Working... (3)
Thread 1: Working... (3)
Thread 1: Working... (4)
Thread 2: Working... (4)
Thread 3: Working... (4)
Thread 3: Working... (5)
Thread 2: Working... (5)
Thread 1: Working... (5)
Thread 3: Finishing execution.
Thread 2: Finishing execution.
Thread 1: Finishing execution.
Main thread: All threads have completed execution.
```

Exercise 5	Write a C program to simulate producer-consumer problem using semaphores	Date:

Aim: Write a C program to simulate producer-consumer problem using semaphores

Description:

The Producer-Consumer problem is a classic synchronization problem in computer science, where two pro (or threads), the producer and the consumer, share a common bounded buffer. The problem focuses on ensuring that:

1. The producer should not add items to the buffer when it is full.
2. The consumer should not remove items from the buffer when it is empty.
3. Access to the shared buffer should be mutually exclusive (only one thread should access it at a time to avoid race conditions).

The solution typically involves:

- Semaphores to keep track of empty and full buffer slots.
- A mutex to ensure only one thread accesses the buffer at a time.
- A circular buffer approach to efficiently use the buffer space.

Algorithm:

Producer Algorithm:

Repeat for each item to be produced:

1. Produce an item (generate or fetch data).
2. Wait for an empty slot (sem_wait(&empty)):
    - If empty is 0, the producer will wait until the consumer consumes an item.
3. Acquire mutex (pthread_mutex_lock(&mutex)) to enter the critical section.
4. Insert the item into the buffer at the in index.
5. Update in index:
    - in = (in + 1) % BUFFER_SIZE (circular increment).
6. Release mutex (pthread_mutex_unlock(&mutex)) after adding the item.
7. Signal full (sem_post(&full)):
    - Increments full, indicating there is a new item available for the consumer.
8. Sleep for a while to simulate production time (optional).

Consumer Algorithm:

Repeat for each item to be consumed:

1. Wait for a full slot (sem_wait(&full)):
    - If full is 0, the consumer waits until the producer adds an item.
2. Acquire mutex (pthread_mutex_lock(&mutex)) to enter the critical section.
3. Remove the item from the buffer at the out index.
4. Update out index:
    - out = (out + 1) % BUFFER_SIZE (circular increment).
5. Release mutex (pthread_mutex_unlock(&mutex)) after removing the item.
6. Signal empty (sem_post(&empty)):
    - Increments empty, indicating there is now space for the producer to add more items.
7. Consume the item (process or use the data).
8. Sleep for a while to simulate consumption time (optional).

SOURCE CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define BUFFER_SIZE 5  // Size of the buffer
int buffer[BUFFER_SIZE];  // Shared buffer
int in = 0;             // Index for the next produced item
int out = 0;            // Index for the next consumed item

// Semaphores
sem_t empty;  // Counts empty buffer slots
sem_t full;   // Counts filled buffer slots
pthread_mutex_t mutex;  // Mutex for critical section
```

```
// Producer function
void* producer(void* arg)
{
    int i,item;
    for (i = 0; i < 10; i++)
   {
        item = rand() % 100;  // Produce an item

        sem_wait(&empty);           // Wait if buffer is full
        pthread_mutex_lock(&mutex);  // Enter critical section

        buffer[in] = item;
        printf("Producer produced: %d at index %d\n", item, in);
        in = (in + 1) % BUFFER_SIZE;

        pthread_mutex_unlock(&mutex);  // Exit critical section
        sem_post(&full);            // Signal that buffer has new item
        sleep(1);  // Simulate production time
    }
    pthread_exit(NULL);
}

// Consumer function
void* consumer(void* arg)
{
    int i,item;
    for (i = 0; i < 10; i++)
  {
        sem_wait(&full);            // Wait if buffer is empty
        pthread_mutex_lock(&mutex);   // Enter critical section
```

```
        item = buffer[out];
        printf("Consumer consumed: %d from index %d\n", item, out);
        out = (out + 1) % BUFFER_SIZE;

        pthread_mutex_unlock(&mutex);  // Exit critical section
        sem_post(&empty);              // Signal that buffer has empty slot
        sleep(2);  // Simulate consumption time
    }
    pthread_exit(NULL);
}
int main()
{
    pthread_t producer_thread, consumer_thread;
    // Initialize semaphores and mutex
    sem_init(&empty, 0, BUFFER_SIZE);  // Initially, buffer is empty
    sem_init(&full, 0, 0);             // Initially, buffer is empty (no full slots)
    pthread_mutex_init(&mutex, NULL);

    // Create producer and consumer threads
    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);

    // Wait for threads to complete
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);

    // Cleanup
    sem_destroy(&empty);
    sem_destroy(&full);
    pthread_mutex_destroy(&mutex);
    printf("Main thread: Producer and Consumer have finished execution.\n");
    return 0;
}
```

INPUT & OUTPUT:

```
Producer produced: 41 at index 0
Consumer consumed: 41 from index 0
Producer produced: 67 at index 1
Consumer consumed: 67 from index 1
Producer produced: 34 at index 2
Producer produced: 0 at index 3
Consumer consumed: 34 from index 2
Producer produced: 69 at index 4
Producer produced: 24 at index 0
Consumer consumed: 0 from index 3
Producer produced: 78 at index 1
Producer produced: 58 at index 2
Consumer consumed: 69 from index 4
Producer produced: 62 at index 3
Producer produced: 64 at index 4
Consumer consumed: 24 from index 0
Consumer consumed: 78 from index 1
Consumer consumed: 58 from index 2
Consumer consumed: 62 from index 3
Consumer consumed: 64 from index 4
Main thread: Producer and Consumer have finished execution.
```

Exercise 6	Implement the following memory allocation methods for fixed partition a) First fit b) Worst fit c) Best fit	Date:

FIRST FIT:

Aim: Implementation of First Fit memory allocation method for fixed partition

Description:

This program implements the First Fit memory allocation strategy for fixed par. In this approach, each process is assigned to the first available partition that is large enough to accommodate it. The memory par are fixed, meaning their sizes do not change, but their available space reduces when a process is allocated.

How It Works:

1. The user inputs the number of memory par and their respective sizes.
2. The user then inputs the number of pro along with their memory requirements.
3. The First Fit algorithm traverses the list of par for each process:
    - It finds the first partition that is large enough to accommodate the process.
    - If found, the process is allocated, and the remaining space in that partition is reduced.
    - If no suitable partition is found, the process remains unallocated.
4. The program displays the allocation results, indicating which process is assigned to which partition or if it remains unallocated.

Algorithm:

1. Start
2. Input the number of memory par and their sizes.
3. Input the number of pro and their sizes.
4. Initialize all pro as unallocated.
5. For each process:
    a. Traverse the par from the beginning.
    b. If a partition has enough space, allocate the process and reduce the partition size.
    c. Move to the next process.
6. Print the allocation results.
7. End

Program:

```
#include <stdio.h>
#define MAX_PAR 10
#define MAX_PRO 10
/*
 * First Fit Memory Allocation for Fixed Par
 * This program implements the First Fit memory allocation strategy for fixed par.
 * It assigns pro to the first available partition that can accommodate them.
 */
// Function to implement First Fit memory allocation
void firstfit(int par[], int numpar, int pro[], int numpro)
{
   int allocation[MAX_PRO];
    // Initialize allocation array (no allocation initially)
   for (int i = 0; i < numpro; i++)
  {
        allocation[i] = -1;
        for (int j = 0; j < numpar; j++)
        {
              if (par[j] >= pro[i])
              {
                    allocation[i] = j;
                    par[j] -= pro[i]; // Reduce partition size after allocation
                    break; // Move to the next process
              }
        }
  }

   // Print allocation result
   printf("\nProcess No.\tProcess Size\tPartition Allocated\n");
   for (int i = 0; i < numpro; i++)
  {
         printf("%d\t\t%d\t\t", i + 1, pro[i]);
        if (allocation[i] != -1)
        printf("%d\n", allocation[i] + 1);
```

```
        else
            printf("Not Allocated\n");
    }
}

int main()
{
        int par[MAX_PAR], pro[MAX_PRO];
         int numpar, numpro;
         // Input number of par and their sizes
         printf("Enter the number of memory partitions: ");
         scanf("%d", &numpar);
         printf("Enter the size of each partition: \n");
        for (int i = 0; i < numpar; i++)
        {
                printf("Partition %d: ", i + 1);
                scanf("%d", &par[i]);
        }
        // Input number of pro and their sizes
        printf("Enter the number of processes: ");
         scanf("%d", &numpro);
        printf("Enter the size of each process: \n");
        for (int i = 0; i < numpro; i++)
        {
                printf("Process %d: ", i + 1);
                scanf("%d", &pro[i]);
        }

    // Call First Fit algorithm
    firstFit(par, numpar, pro, numpro);
    return 0;
}
```

INPUT & OUTPUT:

Enter the number of memory partitions: 3

Enter the size of each partition:

Partition 1: 200

Partition 2: 300

Partition 3: 150

Enter the number of processes: 3

Enter the size of each process:

Process 1: 100

Process 2: 200

Process 3: 160

Process No.	Process Size	Partition Allocated
1	100	1
2	200	2
3	160	Not Allocated

WORST FIT:

Aim: Implementation of Worst Fit memory allocation method for fixed partition

Description:

The Worst Fit memory allocation strategy assigns incoming processes to the largest available partition that can accommodate them. This approach aims to leave larger remaining free spaces, potentially allowing future processes to fit more easily.

How It Works:

1. The user inputs the number of fixed memory partitions and their sizes.
2. The user then inputs the number of processes along with their memory requirements.
3. The Worst Fit algorithm is applied:
    - Each process searches for the largest partition that is big enough to hold it.
    - If found, the process is allocated to this partition, and the remaining space in that partition is reduced.
    - If no suitable partition is found, the process remains unallocated.
4. The program displays the allocation results, showing which processes have been allocated and which remain unallocated.

This strategy is useful in scenarios where large memory chunks should be preserved for future allocations. However, it may lead to internal fragmentation, where large partitions are split inefficiently, leading to wasted memory.

Algorithm:

1. Start
2. Input the number of memory partitions and their sizes.
3. Input the number of processes and their sizes.
4. Initialize all processes as unallocated.
5. For each process:
    a. Traverse all partitions and find the largest partition that can accommodate the process.
    b. If found, allocate the process to the partition and reduce its available space.
    c. If no suitable partition is found, leave the process unallocated.
6. Print the allocation results.
7. End

Program:

```
#include <stdio.h>
#define MAX_PARTITIONS 10
#define MAX_PROCESSES 10
/*
 * Worst Fit Memory Allocation for Fixed Partitions
 * This program implements the Worst Fit memory allocation strategy for fixed partitions.
 * It assigns processes to the largest available partition that can accommodate them.
 */
// Function to implement Worst Fit memory allocation
void worstfit(int par[], int numpar, int pro[], int numpro)
{
        int i, j, allocation[MAX_PROCESSES];
        // Initialize allocation array (no allocation initially)
        for (i = 0; i < numpro; i++)
        {
                allocation[i] = -1;
                int worstIndex = -1;
                // Find the worst (largest) partition that fits the process
                for (j = 0; j < numpar; j++)
                {
                        if (par[j] >= pro[i])
                        {
                                if (worstIndex == -1 || par[j] > par[worstIndex])
                                {
                                        worstIndex = j;
                                }
                        }
                }
        }

        // Allocate process to the worst partition found
        if (worstIndex != -1)
        {
                allocation[i] = worstIndex;
                par[worstIndex] -= pro[i]; // Reduce partition size after allocation
```

```
        }
    }

    // Print allocation result
    printf("\nProcess No.\tProcess Size\tPartition Allocated\n");
    for (i = 0; i < numpro; i++)
    {
        printf("%d\t\t%d\t\t", i + 1, pro[i]);
        if (allocation[i] != -1)
            printf("%d\n", allocation[i] + 1);
        else
            printf("Not Allocated\n");
    }
}

int main()
{
        int par[MAX_PARTITIONS], pro[MAX_PROCESSES];
        int i, numpar, numpro;
        // Input number of par and their sizes
         printf("Enter the number of memory partitions: ");
        scanf("%d", &numpar);
        printf("Enter the size of each partition: \n");
        for (i = 0; i < numpar; i++)
        {
                printf("Partition %d: ", i + 1);
                scanf("%d", &par[i]);
        }
        // Input number of processes and their sizes
        printf("Enter the number of processes: ");
        scanf("%d", &numpro);
         printf("Enter the size of each process: \n");
         for (i = 0; i < numpro; i++)
         {
                printf("Process %d: ", i + 1);
```

```
            scanf("%d", &pro[i]);
        }

    // Call Worst Fit algorithm
    worstfit(par, numpar, pro, numpro);
    return 0;
}
```

INPUT & OUTPUT:

```
Enter the number of memory partitions: 3
Enter the size of each partition:
Partition 1: 100
Partition 2: 300
Partition 3: 150
Enter the number of processes: 3
Enter the size of each process:
Process 1: 200
Process 2: 300
Process 3: 160
```

Process No.	Process Size	Partition Allocated
1	200	2
2	300	Not Allocated
3	160	Not Allocated

BEST FIT:

Aim: Implementation of Best Fit memory allocation method for fixed partition

Description:

The **Best Fit** memory allocation strategy assigns each incoming process to the **smallest available partition** that can accommodate it. This approach minimizes wasted space by ensuring that processes use the most efficient memory block possible.

How It Works:

1. The user inputs the number of **fixed memory partitions** and their sizes.
2. The user then inputs the number of **processes** and their memory requirements.
3. The **Best Fit** algorithm is applied:
    - Each process searches for the **smallest** partition that is large enough to hold it.
    - If found, the process is allocated to this partition, and the remaining space in that partition is reduced.
    - If no suitable partition is found, the process remains unallocated.
4. The program displays the allocation results, showing which processes have been allocated and which remain unallocated.

Advantages:

- Reduces **internal fragmentation** by using the most efficiently sized partition.
- Helps preserve larger memory partitions for bigger processes.

Disadvantages:

- Can lead to **external fragmentation** over time as small gaps may be left between allocated partitions.
- Requires more computational overhead to search for the best possible partition.

Algorithm:

1. Start
2. Input the number of memory partitions and their sizes.
3. Input the number of processes and their sizes.
4. Initialize all processes as unallocated.
5. For each process:
    a. Traverse all partitions and find the smallest partition that can accommodate the process.
    b. If found, allocate the process to the partition and reduce its available space.
    c. If no suitable partition is found, leave the process unallocated.
6. Print the allocation results.
7. End

Program:

```
#include <stdio.h>
#define MAX_PARTITIONS 10
#define MAX_PROCESSES 10
/*
 * Best Fit Memory Allocation for Fixed Partitions
 * This program implements the Best Fit memory allocation strategy for fixed partitions.
 * It assigns processes to the smallest available partition that can accommodate them,
 * minimizing wasted memory space.
 */
// Function to implement Best Fit memory allocation
void bestfit(int par[], int numpar, int pro[], int numpro)
{
        int i, j, allocation[MAX_PROCESSES];
        // Initialize allocation array (no allocation initially)
        for (i = 0; i < numpro; i++)
        {
                allocation[i] = -1;
                int bestIndex = -1;
               // Find the best (smallest suitable) partition that fits the process
                for (j = 0; j < numpar; j++)
               {
                        if (par[j] >= pro[i])
                        {
                                if (bestIndex == -1 || par[j] < par[bestIndex])
                                {
                                        bestIndex = j;
                                }
                        }
                }
        }

        // Allocate process to the best partition found
        if (bestIndex != -1)
       {
            allocation[i] = bestIndex;
```

```
            par[bestIndex] -= pro[i]; // Reduce partition size after allocation
        }
    }

    // Print allocation result
    printf("\nProcess No.\tProcess Size\tPartition Allocated\n");
    for (i = 0; i < numpro; i++)
    {
        printf("%d\t\t%d\t\t", i + 1, pro[i]);
        if (allocation[i] != -1)
        printf("%d\n", allocation[i] + 1);
        else
        printf("Not Allocated\n");
    }
}

int main()
{
    int par[MAX_PARTITIONS], pro[MAX_PROCESSES];
    int i, numpar, numpro;
    // Input number of partitions and their sizes
    printf("Enter the number of memory partitions: ");
    scanf("%d", &numpar);
    printf("Enter the size of each partition: \n");
    for (i = 0; i < numpar; i++)
    {
        printf("Partition %d: ", i + 1);
        scanf("%d", &par[i]);
    }

    // Input number of processes and their sizes
    printf("Enter the number of processes: ");
    scanf("%d", &numpro);
    printf("Enter the size of each process: \n");
```

```
    for (i = 0; i < numpro; i++)
     {
         printf("Process %d: ", i + 1);
         scanf("%d", &pro[i]);
       }

       // Call Best Fit algorithm
       bestfit(par, numpar, pro, numpro);
       return 0;
    }
```

INPUT & OUTPUT:

```
    Enter the number of memory partitions: 3
    Enter the size of each partition:
    Partition 1: 100
    Partition 2: 300
    Partition 3: 150
    Enter the number of processes: 3
    Enter the size of each process:
    Process 1: 200
    Process 2: 300
    Process 3: 50

    Process No.    Process Size    Partition Allocated
    1              200             2
    2              300             Not Allocated
    3              50              1
```

Exercise 7	Implement Bankers Algorithm for Dead Lock avoidance and prevention	Date:

Aim: Implement Bankers Algorithm for Dead Lock avoidance and prevention

Description:

Deadlock is a situation where in two or more competing actions are waiting f or the other to finish, and thus neither ever does. When a new process enters a system, it must declare the maximum number of instances of each resource type it needed. This number may exceed the total number of resources in the system. When the user requests a set of resources, the system must determine whether the allocation of each resource will leave the system in safe state. If it will the resources are allocation; otherwise the process must wait until some other process release the resources.

Algorithm:

Safety Algorithm:

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1. Let Work and Finish be vectors of length ‘m’ and ‘n’ respectively.
   Initialize: Work = Available
   Finish[i] = false; for i=1, 2, 3, 4….n
2. Find an i such that both
   2.1 Finish[i] = false
   2.2 Needi <= Work
   if no such i exists goto step (4)
3. Work = Work + Allocation[i]
   Finish[i] = true
   goto step (2)
4. if Finish [i] = true for all i
   then the system is in a safe state

Resource-Request Algorithm:

Let $Request_i$ be the request array for process P_i. $Request_i$ [j] = k means process P_i wants k instances of resource type Rj. When a request for resources is made by process P_i, the following actions are taken:

1. If $Request_i <= Need_i$

Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.

2. If $Request_i <= Available$

Goto step (3); otherwise, Pi must wait, since the resources are not available.

3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:

$Available = Available – Request_i$

$Allocation_i = Allocation_i + Request_i$

$Need_i = Need_i – Request_i$

Program:

```
#include <stdio.h>
#define MAX_PROCESSES 5
#define MAX_RESOURCES 3
int available[MAX_RESOURCES];
int maximum[MAX_PROCESSES][MAX_RESOURCES];
int allocation[MAX_PROCESSES][MAX_RESOURCES];
int need[MAX_PROCESSES][MAX_RESOURCES];

// Function to check if the system is in a safe state
int isSafe(int processes[], int work[], int finish[])
{
   int safeSequence[MAX_PROCESSES];
   int i, p , r, count = 0;
   while (count < MAX_PROCESSES)
  {
        int found = 0;
        for (p = 0; p < MAX_PROCESSES; p++)
        {
              if (!finish[p])
```

```
                {
                        int flag = 1;
                        for (r = 0; r < MAX_RESOURCES; r++)
                        {
                                if (need[p][r] > work[r])
                                {
                                        flag = 0;
                                        break;
                                }
                        }
                        if (flag)
                        {
                                for (r = 0; r < MAX_RESOURCES; r++)
                                work[r] += allocation[p][r];
                                safeSequence[count++] = p;
                                finish[p] = 1;
                                found = 1;
                        }
                }
            }
         if (!found)
        {
                printf("The system is not in a safe state.\n");
                return 0;
        }
      }

    printf("The system is in a safe state. Safe sequence is: ");
    for (i = 0; i < MAX_PROCESSES; i++)
    printf("%d ", safeSequence[i]);
    printf("\n");
    return 1;
}
```

```
// Main function
int main()
{
        int processes[MAX_PROCESSES] = {0, 1, 2, 3, 4};
         int i, j;
        printf("Enter available resources: ");
        for (i = 0; i < MAX_RESOURCES; i++)
        scanf("%d", &available[i]);
        printf("Enter maximum resource matrix: \n");
         for (i = 0; i < MAX_PROCESSES; i++)
                for (j = 0; j < MAX_RESOURCES; j++)
                        scanf("%d", &maximum[i][j]);

        printf("Enter allocation matrix: \n");
        for (i = 0; i < MAX_PROCESSES; i++)
                 for (j = 0; j < MAX_RESOURCES; j++)
                        scanf("%d", &allocation[i][j]);

        // Calculate need matrix
         for (i = 0; i < MAX_PROCESSES; i++)
                for (j = 0; j < MAX_RESOURCES; j++)
                        need[i][j] = maximum[i][j] - allocation[i][j];

        int finish[MAX_PROCESSES] = {0};
        int work[MAX_RESOURCES];

        for (i = 0; i < MAX_RESOURCES; i++)
        work[i] = available[i];

        // Checking for safe state
        isSafe(processes, work, finish);
        return 0;
}
```

Input:

Enter available resources: 3 3 2

Enter maximum resource matrix:

7 5 3

3 2 2

9 0 2

2 2 2

4 3 3

Enter allocation matrix:

0 1 0

2 0 0

3 0 2

2 1 1

0 0 2

Output:

The system is in a safe state. Safe sequence is: 1 3 4 0 2

Exercise 8	Simulate the following page replacement algorithms a) FIFO b) LRU c) LFU	Date:

i) Aim: Implement FIFO page replacement algorithm

Description:

The FIFO Page Replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. There is not strictly necessary to record the time when a page is brought in. By creating a FIFO queue to hold all pages in memory and by replacing the page at the head of the queue. When a page is brought into memory, insert it at the tail of the queue.

Algorithm:

1. Start the process
2. Declare the size with respect to page length
3. Check the need of replacement from the page to memory
4. Check the need of replacement from old page to new page in memory
5. Format queue to hold all pages
6. Insert the page require memory into the queue
7. Check for bad replacement and page fault
8. Get the number of processes to be inserted
9. Display the values
10. Stop the process

Program:

```
#include<stdio.h>
main()
{
int i, j, k, f, pf=0, count=0, rs[25], m[10], n;
printf("\n Enter the length of reference string -- ");
scanf("%d",&n);
printf("\n Enter the reference string -- ");
for(i=0;i<n;i++)
scanf("%d",&rs[i]);
printf("\n Enter no. of frames -- ");
```

```
scanf("%d",&f);
for(i=0;i<f;i++)
m[i]=-1;
printf("\n The Page Replacement Process is -- \n");
for(i=0;i<n;i++)
{
        for(k=0;k<f;k++)
        {
                if(m[k]==rs[i])
                break;
        }
        if(k==f)
        {
                m[count++]=rs[i];
                pf++;
        }
        for(j=0;j<f;j++)
        printf("\t%d",m[j]);
        if(k==f)
        printf("\tPF No. %d",pf);
        printf("\n");
        if(count==f)
        count=0;
}
printf("\n The number of Page Faults using FIFO are %d",pf);
getch();
}
```

Input:

Enter the length of reference string -- 10

Enter the reference string -- 1

3
7
6
4
0
5
3
4
8

Enter no. of frames -- 3

Output:

The Page Replacement Process is --

1	-1	-1	PF No. 1
1	3	-1	PF No. 2
1	3	7	PF No. 3
6	3	7	PF No. 4
6	4	7	PF No. 5
6	4	0	PF No. 6
5	4	0	PF No. 7
5	3	0	PF No. 8
5	3	4	PF No. 9
8	3	4	PF No. 10

The number of Page Faults using FIFO are 10

ii) Aim: Implement LRU page replacement algorithm

Description:

The Least Recently Used replacement policy chooses to replace the page which has not been referenced for the longest time. This policy assumes the recent past will approximate the immediate future. The operating system keeps track of when each page was referenced by recording the time of reference or by maintaining a stack of references.

Algorithm:

1. Start the process
2. Declare the size
3. Get the number of pages to be inserted
4. Get the value
5. Declare counter and stack
6. Select the least recently used page by counter value
7. Stack them according the selection.
8. Display the values
9. Stop the process

Program:

```
#include<stdio.h>
main()
{
    int i, j , k, min, rs[25], m[10], count[10], flag[25], n, f, pf=0, next=1;
    printf("Enter the length of reference string -- ");
    scanf("%d",&n);
    printf("Enter the reference string -- ");
    for(i=0;i<n;i++)
    {
        scanf("%d",&rs[i]);
        flag[i]=0;
    }
    printf("Enter the number of frames -- ");
    scanf("%d",&f);
```

```
for(i=0;i<f;i++)
{
        count[i]=0;
        m[i]=-1;
}
printf("\nThe Page Replacement process is -- \n");
for(i=0;i<n;i++)
{
        for(j=0;j<f;j++)
        {
                if(m[j]==rs[i])
                {
                        flag[i]=1;
                        count[j]=next;
                        next++;
                }
        }
        if(flag[i]==0)
        {
                if(i<f)
                {
                        m[i]=rs[i];
                        count[i]=next;
                        next++;
                }
                else
                {
                        min=0;
                        for(j=1;j<f;j++)
                        if(count[min] > count[j])
                        min=j;
                        m[min]=rs[i];
                        count[min]=next;
                        next++;
                }
```

```
                    pf++;
            }
            for(j=0;j<f;j++)
            printf("%d\t", m[j]);
            if(flag[i]==0)
            printf("PF No. -- %d" , pf);
            printf("\n");
    }
    printf("\nThe number of page faults using LRU are %d",pf);
    getch();
    }
```

Input:

Enter the length of reference string -- 10

Enter the reference string -- 7

5
6
2
3
2
1
0
1
1

Enter the number of frames -- 3

Output:

The Page Replacement process is --

7	-1	-1	PF No. -- 1
7	5	-1	PF No. -- 2
7	5	6	PF No. -- 3
2	5	6	PF No. -- 4
2	3	6	PF No. -- 5
2	3	6	
2	3	1	PF No. -- 6

2 0 1 PF No. -- 7

2 0 1

2 0 1

The number of page faults using LRU are 7

iii) Aim: Implement LFU page replacement algorithm

Description:

The **Least Frequently Used (LFU)** page replacement algorithm is a caching technique that removes the page with the **lowest frequency of use** when a new page needs to be loaded into memory. It is designed to predict that pages accessed less frequently in the past are less likely to be accessed in the future.

Algorithm:

Step1: Start the program

Step2: Declare the required variables and initialize it.

Step3; Get the frame size and reference string from the user

Step4: Keep track of entered data elements

Step5: Accommodate a new element look for the element that is not to be used in frequently replace.

Step6: Count the number of page fault and display the value

Step7: Terminate the program

Program:

```
#include<stdio.h>
main()
{
    int rs[50], i, j, k, m, f, cntr[20], a[20], min, pf=0;
    printf("\nEnter number of page references -- ");
    scanf("%d",&m);
    printf("\nEnter the reference string -- ");
    for(i=0;i<m;i++)
    scanf("%d",&rs[i]);
    printf("\nEnter the available no. of frames -- ");
    scanf("%d",&f);
    for(i=0;i<f;i++)
    {
        cntr[i]=0;
        a[i]=-1;
    }
```

```
printf("\nThe Page Replacement Process is – \n");
for(i=0;i<m;i++)
{
        for(j=0;j<f;j++)
                if(rs[i]==a[j])
                {
                        cntr[j]++;
                        break;
                }
                if(j==f)
                {
                        min = 0;
                        for(k=1;k<f;k++)
                        if(cntr[k]<cntr[min])
                        min=k;
                        a[min]=rs[i];
                        cntr[min]=1;
                        pf++;
                }
        printf("\n");
        for(j=0;j<f;j++)
        printf("\t%d",a[j]);
        if(j==f)
        printf("\tPF No. %d",pf);
}
printf("\n\n Total number of page faults -- %d",pf);
getch();
}
```

Input:

Enter number of page references -- 10

Enter the reference string -- 1

2

3

1

3

4

5

2

4

2

Enter the available no. of frames -- 3

Output:

The Page Replacement Process is --

1	-1	-1	PF No. 1
1	2	-1	PF No. 2
1	2	3	PF No. 3
1	2	3	PF No. 3
1	2	3	PF No. 3
1	4	3	PF No. 4
1	5	3	PF No. 5
1	2	3	PF No. 6
1	4	3	PF No. 7
1	2	3	PF No. 8

Total number of page faults – 8

Exercise 9	Simulate Paging Technique of memory management	Date:

Aim: Simulate Paging Technique of memory management

Description: In Operating Systems, Paging is a storage mechanism used to retrieve processes from the secondary storage into the main memory in the form of pages. The main idea behind the paging is to divide each process in the form of pages. The main memory will also be divided in the form of frames. One page of the process is to be stored in one of the frames of the memory. The pages can be stored at the different locations of the memory but the priority is always to find the contiguous frames or holes. Pages of the process are brought into the main memory only when they are required otherwise, they reside in the secondary storage. Different operating system defines different frame sizes. The sizes of each frame must be equal. Considering the fact that the pages are mapped to the frames in Paging, page size needs to be as same as frame size.

Algorithm:

Step 1: Read all the necessary input from the keyboard.

Step 2: Pages - Logical memory is broken into fixed - sized blocks.

Step 3: Frames – Physical memory is broken into fixed – sized blocks.

Step 4: Calculate the physical address using the following Physical address =

(Frame number * Frame size) + offset

Step 5: Display the physical address.

Step 6: Stop the process.

Program:

```
#include <stdio.h>
#define MAX_PAGES 100
#define MAX_FRAMES 50

void paging(int page_table[], int p, int page_size)
{
        int la, pageno, offset, pa;
        printf("\nEnter a logical address (negative to exit): ");
```

```
        while (1)
        {
                scanf("%d", &la);
                 if (la < 0)
                break;
                pageno = la / page_size;
                offset = la % page_size;
               if (pageno >= p)
                {
                        printf("Invalid logical address! Page number out of bounds.\n");
                 }
                else
                {
                        int frameno = page_table[pageno];
                        pa = frameno * page_size + offset;
                         printf("Logical Address %d -> Physical Address %d\n", la, pa);
                }
                printf("\nEnter another logical address (negative to exit): ");
        }
}

int main()
{
   int i, p, f, page_size;
   int page_table[MAX_PAGES];
   // Input the number of pages, frames, and page size
   printf("Enter the number of pages: ");
   scanf("%d", &p);
   printf("Enter the number of frames: ");
   scanf("%d", &f);
   printf("Enter the page size (in bytes): ");
   scanf("%d", &page_size);
```

```
    if (p > MAX_PAGES || f > MAX_FRAMES)
    {
        printf("Error: Exceeding maximum limits.\n");
        return 1;
    }

    // Input the page table (mapping pages to frames)
    printf("Enter the page table (frame number for each page):\n");
    for (i = 0; i < p; i++)
    {
        printf("Page %d -> Frame: ", i);
        scanf("%d", &page_table[i]);
    }

    // Simulate logical to physical address translation
    paging(page_table, p, page_size);
    return 0;
}
```

Input:

```
Enter the number of pages: 4
Enter the number of frames: 4
Enter the page size (in bytes): 8
Enter the page table (frame number for each page):
Page 0 -> Frame: 4
Page 1 -> Frame: 2
Page 2 -> Frame: 3
Page 3 -> Frame: 1
```

Output:

```
Enter another logical address (negative to exit): 3
Logical Address 3 -> Physical Address 35

Enter another logical address (negative to exit): 12
Logical Address 12 -> Physical Address 20

Enter a logical address (negative to exit): 1024
Invalid logical address! Page number out of bounds.

Enter another logical address (negative to exit): -1
```

Exercise 10	Simulate the following file allocation strategies a) Sequential b) Indexed c) Linked	Date:

a) Sequential

AIM: To write a C program for implementing sequential file allocation method

DESCRIPTION:

The most common form of file structure is the sequential file in this type of file, a fixed format is used for records. All records (of the system) have the same length, consisting of the same number of fixed length fields in a particular order because the length and position of each field are known, only the values of fields need to be stored, the field name and length for each field are attributes of the file structure.

ALGORITHM:

Step 1: Start the program.

Step 2: Get the number of files.

Step 3: Get the memory requirement of each file.

Step 4: Allocate the required locations to each in sequential order a). Randomly select a location from availablelocation s1= random(100);

a) Check whether the required locations are free from the selected location.

```
if(b[s1].flag==0)
{
        for(j=s1;j<s1+p[i];j++)
        {
                if((b[j].flag)==0) count++;
        }
        if(count==p[i]) break;
}
```

b) Allocate and set flag=1 to the allocated locations.

```
 for(s=s1;s<(s1+p[i]);s++)
{
        k[i][j]=s; j=j+1; b[s].bno=s; b[s].flag=1;
}
```

Step 5: Print the results file no, length, Blocks allocated.

Step 6: Stop the program

SOURCE CODE:

```
#include<stdio.h>
main()
{
    int f[50],i,st,j,len,c,k;
    for(i=0;i<50;i++)
    f[i]=0;
    X:
        printf("\n Enter the starting block & length of file"); scanf("%d%d",&st,&len);
        for(j=st;j<(st+len);j++) if(f[j]==0)
        {
            f[j]=1;
            printf("\n%d->%d",j,f[j]);
        }
        else
        {
            printf("Block already allocated"); break;
        }
        if(j==(st+len))
        printf("\n the file is allocated to disk");
        printf("\n if u want to enter more files?(y-1/n-0)"); scanf("%d",&c);
        if(c==1) goto X; else exit();
        getch();
}
```

OUTPUT:

Enter the starting block & length of file 4 10 4->1

5->1

6->1

7->1

8->1

9->1

10->1

11->1

12->1

13->1

The file is allocated to disk.

INDEXED:

AIM: To implement allocation method using chained method

DESCRIPTION:

In the chained method file allocation table contains a field which points to starting block of memory. From it for each bloc a pointer is kept to next successive block. Hence, there is no external fragmentation.

ALGORITHM:

Step 1: Start the program.

Step 2: Get the number of files.

Step 3: Get the memory requirement of each file.

Step 4: Allocate the required locations by selecting a location randomly q= random(100);

a) Check whether the selected location is free .

b) If the location is free allocate and set flag=1 to the allocated locations.

```
q=random(100);
{
        if(b[q].flag==0)
        b[q].flag=1;
        b[q].fno=j;
        r[i][j]=q;
```

Step 5: Print the results file no, length, Blocks allocated.

Step 6: Stop the program

SOURCE CODE:

```
#include<stdio.h>
int f[50],i,k,j,inde[50],n,c,count=0,p;
main()
{
    for(i=0;i<50;i++)
    f[i]=0;
x: printf("enter index block\t");
scanf("%d",&p);
if(f[p]==0)
{
    f[p]=1;
    printf("enter no of files on index\t");
    scanf("%d",&n);
```

```
}
else
{
        printf("Block already allocated\n");
        goto x;
}
for(i=0;i<n;i++)
scanf("%d",&inde[i]);
for(i=0;i<n;i++)
if(f[inde[i]]==1)
{
        printf("Block already allocated");
        goto x;
}
for(j=0;j<n;j++)
f[inde[j]]=1;
printf("\n allocated");
printf("\n file indexed");
 for(k=0;k<n;k++)
printf("\n %d->%d:%d",p,inde[k],f[inde[k]]);
printf(" Enter 1 to enter more files and 0 to exit\t");
scanf("%d",&c);
if(c==1)
goto    x;
else exit();
getch();
}
```

OUTPUT:

enter index block 9 Enter no of files on index 3 1 2 3

Allocated File indexed 9->1:1

9->2;1

9->3:1 enter 1 to enter more files and 0 to exit

LINKED:

AIM: To implement linked file allocation technique.

DESCRIPTION:

In the chained method file allocation table contains a field which points to starting block of memory. From it for each bloc a pointer is kept to next successive block. Hence, there is no external fragmentation

ALGORTHIM:

Step 1: Start the program.

Step 2: Get the number of files.

Step 3: Get the memory requirement of each file.

Step 4: Allocate the required locations by selecting a location randomly q= random(100);

a) Check whether the selected location is free .

b) If the location is free allocate and set flag=1 to the allocated locations.

While allocating next location address to attach it to previous location

```
for(i=0;i<n;i++)
{
        for(j=0;j<s[i];j++)
    {
        q=random(100);
                if(b[q].flag==0)
        b[q].flag=1;
        b[q].fno=j;
        r[i][j]=q;
       if(j>0)
       {
        }
       }
        p=r[i][j-1];
        b[p].next=q;}
```

Step 5: Print the results file no, length ,Blocksallocated.

Step 6: Stop the program

SOURCE CODE :

```
#include<stdio.h>
main()
{
int f[50],p,i,j,k,a,st,len,n,c;
clrscr();
for(i=0;i<50;i++)
 f[i]=0;
printf("Enter how many blocks that are already allocated");
scanf("%d",&p);
printf("\nEnter the blocks no.s that are already allocated");
for(i=0;i<p;i++)
{
        scanf("%d",&a);
        f[a]=1;
}
X:
printf("Enter the startingindex block & length");
scanf("%d%d",&st,&len);
 k=len;
for(j=st;j<(k+st);j++)
{
        if(f[j]==0)
        {
                f[j]=1;
                printf("\n%d->%d",j,f[j]);
        }
        else
        {
                printf("\n %d->file is already allocated",j);
                k++;
        }
}
printf("\n If u want to enter one more file? (yes-1/no-0)");
scanf("%d",&c);
```

```
if(c==1)
goto    X;
else
exit();
getch( );
}
```

OUTPUT:

Enter how many blocks that are already allocated 3

Enter the blocks no.s that are already allocated 4 7

Enter the starting index block & length 3 7 9

3->1

4->1 file is already allocated 5->1

6->1

7->1 file is already allocated 8->1

9->1file is already allocated 10->1

11->1

12->1