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 processes 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 processes.
  - `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 processes 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 processes 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 processes -- 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 processes -- "); 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 processes --        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 processes are completed.

ALGORITHM:

Step 1: Start the process

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

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 processes -- ");
        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 processes – 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 processes. The waiting time of each process is obtained by summing up the burst times of all the previous processes.

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 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 processes 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 processes --- ");
        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 processes -- 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 processes 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 processes (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.
```