College code: 6F
PALLAVI ENGINEERING COLLEGE
(UGC AUTONOMOUS)
Accredited by NBA and NAAC with 'A' grade, Approved by AICTE, New Delhi & Affiliated to JNTUH-Hyderabad
Certified by ISO 9001 : 2015 | ISO 14001 : 2015 | ISO 50001 : 2018
Kuntloor(V), Adbullapurmet(M), Near Hayathanagar, R.R. Dist. Hyd - 501505, (T.S.) India

Department of Computer Science and Engineering (AI&ML)

OPERATING SYSTEM
LAB MANUAL

Regulation
PR24

Class: II B.Tech I Semester

Prepared by

M.SREE PAVANI

Assistant Professor

LAB FACULTY **HOD** **PRINCIPAL**

VISION OF THE INSTITUTE

- To emerge as a global leader in imparting quality technical education emphasizing ethical values for the betterment of the society.

MISSION OF THE INSTITUTE

- To create an excellent teaching learning environment and inculcate the aptitude for research.
- To establish centers of excellence through collaborative initiatives.
- To empower the student community by developing creativity and innovation.

Proposed Vision and Mission of the Department

VISION OF THE DEPARTMENT

- To become a leading centre of excellence in Artificial Intelligence and Machine Learning by fostering innovation, research, and collaboration in diverse areas of computer science. We aim to address global challenges and emerging societal needs through advanced education, cutting-edge technologies, and impactful solutions in AI and ML.

MISSION OF THE DEPARTMENT

- To equip students with the knowledge and skills to solve complex, real-world problems in multidisciplinary fields using AI and ML technologies.
- To foster strong domain expertise and research capabilities, enabling students to pursue challenging careers and advanced education in AI and ML.
- To provide students with a strong sense of ethics, professionalism, and a desire for lifelong learning, enabling them to make significant contributions to both the field and society.

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

The Computer Science and Engineering – Data Science graduate will:

PEO	Statements
PEO1	Graduates will be prepared for a successful career in Computer Science discipline and related industry to meet the needs of the nation and leading industries and also to excel in postgraduate programs.
PEO2	Graduates will continue to learn and apply the acquired knowledge to solve Engineering problems and appreciation of the arts, humanities and social sciences.
PEO3	Graduates will have good and broad scientific and engineering knowledgebase so as to comprehend, analyze, design and create novel products and solutions for real-time applications.
PEO4	Graduates will understand professional and ethical responsibility, develop leadership, utilize membership opportunities, and develop effective communication skills, teamwork skills, multidisciplinary approach and life-long learning required for a successful professional career.

PROGRAM SPECIFIC OUTCOMES (PSOs)

The Computer Science and Engineering – Data Science graduate will be able to:

PSOs	Statements
PSO1	Expertise in different aspects and appropriate models of Data Science and use large data sets to cater for the growing demand for data scientists and engineers in industry.
PSO2	Apply the principles and techniques of database design, administration, and implementation to enhance data collection capabilities and decision-support systems.

Program outcomes:

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

2. Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

3. 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.

4. 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.

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

6. 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.

7. 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.

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

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

10. 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 andreceive clear instructions.

11. **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.

12. **Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest.

Course Objectives:

- To provide an understanding of the design aspects of operating system concepts through simulation
- Introduce basic Unix commands, system call interface for process management, interprocess communication and I/O in Unix

Course Outcomes:

CO Number	CO Statement
C217.1	Simulate and implement operating system concepts such as scheduling, deadlock management, file management and memory management.
C217.2	Able to implement C programs using Unix system calls
C217.3	Implement producer-Consumer Problem.
C217.4	Implementing Page replacement and disk scheduling techniques.
C217.5	Use Different system calls for writing application program.

JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY HYDERABAD

II Year B.Tech. CSE. II – Sem **L T P C**

Course Code: CS406PC **0 0 3 2**

OPERATING SYSTEMS LAB SYLLABUS

Course Objectives:

- To provide an understanding of the design aspects of operating system concepts through simulation.
- Introduce basic Unix commands, system call interface for process management, interprocess communication and I/O in Unix

Course Outcomes:

- Simulate and implement operating system concepts such as scheduling, deadlock management, file management and memory management.
- Able to implement C programs using Unix system calls.

LIST OF EXPERIMENTS:

1. Write C programs to simulate the following CPU Scheduling algorithms
   a) FCFS b)SJF c)Round Robin d)priority
2. Write programs using the I/O system calls of UNIX/LINUX operating system (open, read, write, close, fcntl, seek, stat, opendir, readdir)
3. Write a C program to simulate Bankers Algorithm for Deadlock Avoidance and Prevention.
4. Write a C program to implement the Producer – Consumer problem using semaphores using UNIX/LINUX system calls.
5. Write C programs to illustrate the following IPC mechanisms
   a) Pipes b) FIFOs c)Message Queues d) Shared Memory
6. Write C programs to simulate the following memory management techniques
   a) Paging b) Segmentation

TEXT BOOKS:

1. Operating System Principles- Abraham Silberchatz, Peter B. Galvin, Greg Gagne 7th Edition, John Wiley
2. Advanced programming in the Unix environment, W.R.Stevens, *Pearson*education.

REFERENCE BOOKS:

1. Operating Systems – Internals and Design Principles, William Stallings, Fifth Edition–2005, Pearson Education/PHI
2. Operating System - A Design Approach-Crowley,TMH.
3. Modern Operating Systems, Andrew S Tanenbaum, 2nd edition, Pearson/PHI
4. UNIX Programming Environment, Kernighan and Pike, PHI/Pearson Education
5. UNIX Internals: The New Frontiers, U.Vahalia, Pearson Education

LAB PROGRAMS

Program No.	Program Aim	Page No
1	Write C programs to simulate the following CPU Scheduling algorithms a) FCFS b) SJF c) Round Robin d) priority	
2	Write programs using the I/O system calls of UNIX/LINUX operating system (open, read, write, close, fcntl, seek, stat, opendir, readdir)	
3	Write a C program to simulate Bankers Algorithm for Deadlock Avoidance and Prevention.	
4	Write a C program to implement the Producer – Consumer problem using semaphores using UNIX/LINUX system calls.	
5	Write C programs to illustrate the following IPC mechanisms a) Pipes b) FIFOs c) Message Queues d) Shared Memory	
6	Write C programs to simulate the following memory management techniques a) Paging b) Segmentation	

EXPERIMENT 1

OBJECTIVE

Write C programs to simulate the following CPU Scheduling algorithms

a) FCFS b) SJF c) Round Robin d) Priority

DESCRIPTION

FCFS CPU SCHEDULING ALGORITHM: For FCFS scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. The scheduling is performed on the basis of arrival time of the processes irrespective of their other parameters. Each process will be executed according to its arrival time. Calculate the waiting time and turnaround time of each of the processes accordingly.

SJF CPU SCHEDULING ALGORITHM: For SJF scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times. Arrange all the jobs in order with respect to their burst times. There may be two jobs in queue with the same execution time, and then FCFS approach is to be performed. Each process will be executed according to the length of its burst time. Then calculate the waiting time and turnaround time of each of the processes accordingly.

ROUND ROBIN CPU SCHEDULING ALGORITHM: For round robin scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times, and the size of the time slice. Time slices are assigned to each process in equal portions and in circular order, handling all processes execution. This allows every process to get an equal chance. Calculate the waiting time and turnaround time of each of the processes accordingly.

PRIORITY CPU SCHEDULING ALGORITHM: For priority scheduling algorithm, read the number of processes/jobs in the system, their CPU burst times, and the priorities. Arrange all the jobs in order with respect to their priorities. There may be two jobs in queue with the same priority, and then FCFS approach is to be performed. Each process will be executed according to its priority. Calculate the waiting time and turnaround time of each of the processes accordingly.

a) FCFS

ALGORITHM:

Step 1: Input the processes along with their processid(pid),arrival time(at) and burst time (bt).

Step 2: Sort the processes according to their arrival time.

Step 3: Find the completion time for all other processes i.e.

for process i ->ct[i] = ct[i-1] + bt[i].

Step 4: Find waiting time (wt) for all processes.

Step 5: As first process that comes need not to wait so waiting time for process 1 will be 0 i.e. wt[0] = 0.

Step 6: Find waiting time for all other processes i.e. for process i ->

wt[i] = bt[i-1] + wt[i-1] .

Step 7: Find turnaround time = waiting_time + burst_time for all processes.

Step 8: Find average waiting time = total_waiting_time / no_of_processes.

Step 9: Similarly, find average turnaround time = total_turn_around_time / no_of_processes.

SOURCE CODE:

```
#include<stdio.h>
int main()
{
int pid[10]={0},bt[10]={0},at[10]={0},tat[10]={0},wt[10]={0},ct[10]={0};
int n,sum=0,temp,temp1,i,j,k,temp2;
float totalTAT=0,totalWT=0;
printf("Enter number of processes      ");
scanf("%d",&n);
printf("Enter the processes details\n\n");
for(i=0;i<n;i++)
        {
                printf("Enter processid");
                scanf("%d",&pid[i]);
                printf("Arrival time of process[%d] ",i+1);
                scanf("%d",&at[i]);
                printf("Burst time of process[%d]      ",i+1);
                scanf("%d",&bt[i]);
                printf("\n");
        }
for (i=0; i<n-1; i++)
{
for (j=0; j<n-i-1; j++)
        {
        if (at[j]>at[j+1])
                {
                // sorting the arrival times
                temp = at[j];
                at[j] = at[j+1];
                at[j+1] = temp;
```

```
                // sorting the burst times
                temp1 = bt[j];
                bt[j] = bt[j+1];
                bt[j+1] = temp1;
                // sorting the process numbers
                temp2=pid[j];
                pid[j]=pid[j+1];
                pid[j+1]=temp2;
                }
        }
}
        //calculate completion time of processes

        for(j=0;j<n;j++)
        {
                sum+=bt[j];
                ct[j]+=sum;
        }
        //calculate turnaround time and waiting times

        for(k=0;k<n;k++)
        {
                tat[k]=ct[k]-at[k];
                totalTAT+=tat[k];
        }

        wt[0]=0;
        for(k=0;k<n;k++)
        {
                wt[k]=0;
```

```
            for(j=0;j<k;j++)
        wt[k]+=bt[j];
            totalWT+=wt[k];
        }

        printf("Solution: \n\n");
        printf("P#\t AT\t BT\t CT\t TAT\t WT\t\n\n");
        for(i=0;i<n;i++)
        {
            printf("P%d\t %d\t %d\t %d\t %d\t %d\n",pid[i],at[i],bt[i],ct[i],tat[i],wt[i]);
        }
        printf("\n\nAverage Turnaround Time = %f\n",totalTAT/n);
        printf("\nAverage Waiting Time = %f\n\n",totalWT/n);
        return 0;
}
```

OUTPUT:

Enter number of processes 5

Enter the processes details

Enter processid1

Arrival time of process[1] 2

Burst time of process[1] 3

Enter processid2

Arrival time of process[2] 5

Burst time of process[2] 4

Enter processid3

Arrival time of process[3] 1

Burst time of process[3] 5

Enter processid4

Arrival time of process[4] 3

Burst time of process[4] 3

Enter processid5

Arrival time of process[5] 4

Burst time of process[5] 2

Solution:

P#	AT	BT	CT	TAT	WT
P3	1	5	5	4	0
P1	2	3	8	6	5
P4	3	3	11	8	8
P5	4	2	13	9	11
P2	5	4	17	12	13

Average Turnaround Time = 7.800000

Average Waiting Time = 7.400000

b) SJF NON-PREEMPTIVE

ALGORITHM:

Step 1: Input the processes along with their processnumber(p)and burst time (bt).

Step 2: Sort the processes according to their burst times.

Step 3: Find the completion time for all other processes i.e.

for process i -> ct[i] = ct[i-1] + bt[i].

Step 4: Find waiting time (wt) for all processes.

Step 5: As first process that comes need not to wait so waiting time for process 1 will be 0 i.e. wt[0] = 0.

Step 6: Find waiting time for all other processes i.e. for process i ->

wt[i] = bt[i-1] + wt[i-1] .

Step 7: Find turnaround time = waiting_time + burst_time for all processes.

Step 8: Find average waiting time = total_waiting_time / no_of_processes.

Step 9: Similarly, find average turnaround time = total_turn_around_time / no_of_processes.

SOURCE CODE:

```
#include<stdio.h>
#include<conio.h>
#define max 30
void main()
{
inti,j,n,t,p[max],bt[max],wt[max],tat[max];
floatawt=0,atat=0;
printf("Enter the number of processes\n");
scanf("%d",&n);
//Enter the processes according to their arrival times
for(i=0;i<n;i++)
{
        printf("Enter the process number\n");
        scanf("%d",&p[i]);
        printf("Enter the burst time of the process\n");
        scanf("%d",&bt[i]);
}
//Apply the bubble sort technique to sort the processes according to their burst times
for(i=0;i<n;i++)
{
        for(j=0;j<n-i-1;j++)
        {
                if(bt[j]>bt[j+1])
                {
                // Sort according to the burst times
                t=bt[j];
                bt[j]=bt[j+1];
                bt[j+1]=t;
                //Sorting Process Numbers
```

```
                t=p[j];
                p[j]=p[j+1];
                p[j+1]=t;
                }
        }
}
printf("Process\t Burst Time\t Waiting Time\t Turn Around Time\n");
for(i=0;i<n;i++)
{
        wt[i]=0;
        tat[i]=0;
        for(j=0;j<i;j++)
                wt[i]=wt[i]+bt[j];
        tat[i]=wt[i]+bt[i];
        Total_wt=Total_wt +wt[i];
        Total_tat=Total_tat+tat[i];
        printf("%d\t %d\t\t %d\t\t %d\n",p[i],bt[i],wt[i],tat[i]);
}
awt=(float)Total_wt /n;
atat=(float)Total_tat /n;
printf("The average waiting time = %f\n",awt);
printf("The average turn aroud time = %f\n",atat);
getch();
}
```

OUTPUT:

Enter the number of processes

4

Enter the process number

1

Enter the burst time of the process

2

Enter the process number

2

Enter the burst time of the process

8

Enter the process number

3

Enter the burst time of the process

1

Enter the process number

4

Enter the burst time of the process

4

Process	Burst Time	Waiting Time	Turn Around Time
3	1	0	1
1	2	1	3
4	4	3	7
2	8	7	15

The average waiting time = 2.750000

The average turn around time = 6.500000

c) ROUND ROBIN

ALGORITHM:

Step 1: Input the processes along with their burst time (bt).

Step 2: Input the time quantum (or) time slice

Step 3: Create an array rem_bt[] to keep track of remaining burst time of processes. This array is

initially a copy of bt[] (burst times array)

Step 4:Create another array wt[] to store waiting times of processes. Initialize this array as 0.

Step 5:Initialize time : t = 0

Step 6: Keep traversing the all processes while all processes are not done. Do following for

i'th process if it is not done yet.

a- If rem_bt[i] > quantum

(i) t = t + quantum

(ii) bt_rem[i] -= quantum;

b- Else // Last cycle for this process

(i) t = t + bt_rem[i];

(ii) wt[i] = t - bt[i]

(ii) bt_rem[i] = 0; // This process is over

Step 8: Find average waiting time = total_waiting_time / no_of_processes.

Step 9: Similarly, find average turnaround time = total_turn_around_time / no_of_processes.

SOURCE CODE:

```
#include<stdio.h>
int main()
{
int i,n,count=0,time_quantum,t,at[10],bt[10],rem_bt[10],wt[10],tat[10],flag=0;
floattotal_wt=0 , total_tat=0
printf("Enter Total Process:\t ");
scanf("%d",&n);
for(i=0;i<n;i++)
{
        printf("Enter Burst Time for Process %d :",i+1);
        scanf("%d",&bt[i]);
}
printf("Enter Time Quantum:\t");
scanf("%d",&time_quantum);
for (i = 0 ; i < n ; i++)
      rem_bt[i] = bt[i];
t = 0; // Current time
// Keep traversing processes in round robin manner until all of them are not done.
while (1)
{
flag=1;
// Traverse all processes one by one repeatedly
for (i = 0 ; i < n; i++)
{
        // If burst time of a process is greater than 0 then only need to process further
        if (rem_bt[i] > 0)
        {
                flag=0; // There is a pending process
                if (rem_bt[i] > time_quantum)
```

```
                {
        // Increase the value of t i.e. shows how much time a process has been processed
        t += time_quantum;
                // Decrease the burst_time of current process by quantum
                rem_bt[i] -= time_quantum;
                }
                // If burst time is smaller than or equal to quantum. Last cycle for this process
                else
                {
        // Increase the value of t i.e. shows how much time a process has been processed
                t = t + rem_bt[i];
                 // Waiting time is current time minus time used by this process
                wt[i] = t - bt[i];
                // As the process gets fully executed make its remaining burst time = 0
                 rem_bt[i] = 0;
                }
        }
}
if (flag==1)
        break;
}
for (i = 0; i < n ; i++)
        tat[i] = bt[i] + wt[i];
printf("\n Process BT\t WT\t TAT \n");
for(i=0;i<n;i++)
        printf("\n %d \t %d \t %d \t %d \t",i+1,bt[i],wt[i],tat[i]);
for (i = 0; i < n ; i++)
{
        total_wt= total_wt+wt[i];
        total_tat= total_tat+tat[i];
```

```
}
printf("\nAverage waiting time = %f",        total_wt/n);
printf ("\nAverage turn around time = %f",total_tat/n);
}
```

OUTPUT:

Enter Total Process: 4

Enter Burst Time for Process 1 :4

Enter Burst Time for Process 2 :1

Enter Burst Time for Process 3 :8

Enter Burst Time for Process 4 :1

Enter Time Quantum: 1

Process	BT	WT	TAT
1 4	5	9	
2	1	1	2
3	8	6	14
4	1	3	4

Average waiting time = 3.750000

Average turn around time = 7.250000

d) PRIORITY NON-PREEMPTIVE

ALGORITHM:

Step 1: Input the processes along with their processnumber(p),burst time (bt) and priority(pr)

Step 2: Sort the processes according to their priorities.

Step 3: Find the completion time for all other processes i.e.

for process i -> ct[i] = ct[i-1] + bt[i].

Step 4: Find waiting time (wt) for all processes.

Step 5: As first process that comes need not to wait so waiting time for process 1 will be 0 i.e. wt[0] = 0.

Step 6: Find waiting time for all other processes i.e. for process i ->

wt[i] = bt[i-1] + wt[i-1] .

Step 7: Find turnaround time = waiting_time + burst_time for all processes.

Step 8: Find average waiting time = total_waiting_time / no_of_processes.

Step 9: Similarly, find average turnaround time = total_turn_around_time / no_of_processes.

SOURCE CODE:

```
#include<stdio.h>
#define max 30
void main()
{
int i,j,n,t,p[max],bt[max],pr[max],wt[max],tat[max],Total_wt=0,Total_tat=0;
float awt=0,atat=0;
printf("Enter the number of processes\n");
scanf("%d",&n);
//Enter the processes according to their arrival times
for(i=0;i<n;i++)
{
        printf("Enter the process number\n");
        scanf("%d",&p[i]);
        printf("Enter the burst time of the process\n");
        scanf("%d",&bt[i]);
        printf("Enter the priority of the process\n");
        scanf("%d",&pr[i]);
}
//Apply the bubble sort technique to sort the processes according to their priorities times
for(i=0;i<n;i++)
{
        for(j=0;j<n-i-1;j++)
        {
                if(pr[j]>pr[j+1])
                {
                // Sort according to priorities
                t=pr[j];
                pr[j]=pr[j+1];
                pr[j+1]=t;
```

```
                // Sorting burst times
                t=bt[j];
                bt[j]=bt[j+1];
                bt[j+1]=t;
                // Sorting Process numbers
                t=p[j];
                p[j]=p[j+1];
                p[j+1]=t;
                }
        }
}
printf("Process\t Burst Time\t Priority\tWaiting Time\t Turn Around Time\n");
for(i=0;i<n;i++)
{
        wt[i]=0;
        tat[i]=0;
        for(j=0;j<i;j++)
                wt[i]=wt[i]+bt[j];
        tat[i]=wt[i]+bt[i];
        Total_wt=Total_wt+wt[i];
        Total_tat=Total_tat+tat[i];
        printf("P%d\t %d\t\t%d\t\t %d\t\t %d\n",p[i],bt[i],pr[i],wt[i],tat[i]);
}
awt=(float)Total_wt/n;
atat=(float)Total_tat/n;
printf("The average waiting time = %f\n",awt);
printf("The average turn aroud time = %f\n",atat);
}
```

OUTPUT:

Enter the number of processes

4

Enter the process number

1

Enter the burst time of the process

21

Enter the priority of the process

2

Enter the process number

2

Enter the burst time of the process

3

Enter the priority of the process

1

Enter the process number

3

Enter the burst time of the process

6

Enter the priority of the process

4

Enter the process number

4

Enter the burst time of the process

2

Enter the priority of the process

3

Process	Burst Time	Priority	Waiting Time	Turn Around Time
P2	3	1	0	3

P1	21	2	3	24
P4	2	3	24	26
P3	6	4	26	32

The average waiting time = 13.250000

The average turn around time = 21.250000

EXPERIMENT 2

OBJECTIVE

Write programs using the I/O system calls of UNIX/LINUX operating system (open, read, write, close, fcntl, seek, stat, opendir, readdir)

DESCRIPTION

The interface between a process and an operating system is provided by system calls. In general, system calls are available as assembly language instructions. They are also included in the manuals used by the assembly level programmers. System calls are usually made when a process in user mode requires access to a resource. Then it requests the kernel to provide the resource via a system call.

In general, system calls are required in the following situations:

If a file system requires the creation or deletion of files. Reading and writing from files also require a system call.

Creation and management of new processes.

Network connections also require system calls. This includes sending and receiving packets.

Access to a hardware devices such as a printer, scanner etc. requires a system call.

Types of System Calls

There are mainly five types of system calls. These are explained in detail as follows:

Process Control

These system calls deal with processes such as process creation, process termination etc.

File Management

These system calls are responsible for file manipulation such as creating a file, reading a file, writing into a file etc.

Device Management

These system calls are responsible for device manipulation such as reading from device buffers, writing into device buffers etc.

Information Maintenance

These system calls handle information and its transfer between the operating system and the user program.

Communication

These system calls are useful for interprocess communication. They also deal with creating and deleting a communication connection.

open() Used to Create a new empty file.

int open (const char* Path, int flags [, int mode]);

Parameters

Path : is the name to the file to open.

flags : is used to define the file opening modes such as create, read, write modes.

mode : is used to define the file permissions.

Upon successful completion, the function shall open the file and return a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, -1 shall be returned and *errno* set to indicate the error. No files shall be created or modified if the function returns -1.

read() is used to read the content from the file

size_t read (intfd, void* buf, size_tcnt);

Parameters

fd: file descripter

buf: buffer to read data from

cnt: length of buffer

Returns how many bytes were actually read or -1 in case of error.

write() is used to write the content to the file.

size_t write (intfd, void* buf, size_tcnt);

Parameters

- **fd: file descripter**
- **buf: buffer to write data to**
- **cnt: length of buffer**

Returns how many bytes were actually writtenor -1 in case of error.

close() Tells the operating system you are done with a file descriptor and Close the file which pointed by fd.

int close(intfd);

Parameter

- **fd :file descriptor**

Returns 0 on success or -1 on error.

➢ **lseek()** System call that is used to change the location of the read/write pointer of a file descriptor. The location can be set either in absolute or relative terms.

off_t lseek(int fd, off_t offset, int ref);

Parameters

fd : file descriptor of the pointer that is going to be moved

offset : The offset of the pointer

ref : method in which offsetid to be interpreted

Returns the offset of the pointer (in bytes) from thebeginning of the file. If the return value is -1,then there was an error moving the pointer.

SEEK_CUR

The current file position of fd is set to its current value plus pos, which can be negative, zero, or positive. A pos of zero returns the current file position value

new file postion=current file position + offset(the pos argument to lseek)

SEEK_END

The current file position of fd is set to the current length of the file plus pos,which can be negative, zero, or positive. A pos of zero sets the offset to the endof the file.

new file postion= file position of End Of the File + offset

SEEK_SET

The current file position of fd is set to pos.the offset is measured from the beggining of the file. A pos of zero sets the offset to the beginning of the file.

new file postion= offset

The call returns the new file position on success. On error, it returns -1 and errno is set asappropriate.

ALGORITHM:

Step 1: Start the program.

Step 2: open a file for O_RDWR for R/W,O_CREATE for creating a file , O_TRUNC for truncate a file

Step 3: Using write command, write the msg array contents file.

Step 4:Using lseek command to position the pointer to the specified location.

Step 5: Then the file is opened for read only mode and read the characters and displayed it and close the file

Step 6: Stop the program

SOURCE CODE:

```
#include<stdio.h>
#include<fcntl.h>
#include<unistd.h>
int main()
{
intfd;
char buffer[80];
charmsg[50]="Hello PEC";
fd=open("ss.txt",O_RDWR|O_CREAT);
printf("fd=%d",fd);
if(fd!=-1)
   {
printf("\n ss.txt opened with read write access\n");
write(fd,msg,sizeof(msg));
lseek(fd,0,SEEK_SET);
read(fd,buffer,sizeof(msg));
printf("\n %s was written to my file\n",buffer);
close(fd);
   }
return 0;
   }
```

OUTPUT:

fd=3

ss.txt opened with read write access

Hello PEC was written to my file

➢ **stat()** Stat system call is a system call in Linux to check the status of a file such as to check when the file was accessed.

int stat(const char *path, struct stat *buf)

Parameters:

path (Input) A pointer to the null-terminated path name of the file from which information is required.

buf (Output) A pointer to the area to which the information should be written.

ALGORITHM:

Step 1: Start the program.

Step 2:Use the stat command to display information regarding the root folder.

Step 3: Stop the program

SOURCE CODE:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <time.h>
main()
{
struct stat info;
if (stat("/", &info) != 0)
        perror("stat() error");
else
{
puts("stat() returned the following information about root f/s:");
printf(" inode:  %d\n", (int) info.st_ino);
printf(" dev id: %d\n", (int) info.st_dev);
printf("  mode:  %08x\n",       info.st_mode);
printf("  links:  %d\n",         info.st_nlink);
printf("    uid: %d\n", (int) info.st_uid);
printf("    gid: %d\n",  (int) info.st_gid);
}
}
```

OUTPUT:

stat() returned the following information about root f/s:

inode: 578659

dev id: 2097301

mode: 000041ed

links: 1

uid: 0

gid: 0

- **opendir()**

#include <sys/types.h>

#include <dirent.h>

DIR *opendir(const char **dirname*);

The **opendir()** function opens a directory so that it can be read with the **readdir()** function. The variable *dirname* is a string giving the name of the directory to open. If the last component of *dirname* is a symbolic link, **opendir()** follows the symbolic link. As a result, the directory that the symbolic link refers to is opened. The functions **readdir()**, **rewinddir()**, and **closedir()** can be called after a successful call to **opendir()**. The first **readdir()** call reads the first entry in the directory.

Parameters

dirname

(Input) A pointer to the null-terminated path name of the directory to be opened.

Return Value

value

opendir() was successful. The value returned is a pointer to a DIR, representing an open directory stream. This DIR describes the directory and is used in subsequent operations on the directory using the readdir(), rewinddir(), and closedir() functions.

NULL pointer

opendir() was not successful. The errno global variable is set to indicate the error.

- **readdir()**

#include <sys/types.h>
#include <dirent.h>

struct dirent *readdir(DIR **dirp*);

readdir()reads one dirent structure from the directory pointed at by fd into the memory area pointed to by dirp. The parameter count is ignored; at most one dirent structure is read.

The dirent structure is declared as follows:

```
struct dirent
{
long d_ino;              /* inode number */
   off_t d_off;            /* offset to this dirent */
```

```
unsigned short d_reclen;     /* length of this d_name */
char d_name [NAME_MAX+1]; /* filename (null-terminated) */
long d_ino;                  /* inode number */
   off_t d_off;               /* offset to this dirent */
unsigned short d_reclen;    /* length of this d_name */
char d_name [NAME_MAX+1];  /* filename (null-terminated) */
}
```

d_ino is an inode number. d_off is the distance from the start of the directory to this dirent. d_reclen is the size of d_name, not counting the null terminator. d_name is a null-terminated filename.

Parameters

dirp

(Input) A pointer to a DIR that refers to the open directory stream to be read.

Return Value

value

readdir() was successful. The value returned is a pointer to a dirent structure describing the next directory entry in the directory stream.

ALGORITHM:

Step 1: Start the program

Step 2: Include header files

Step 3: The opendir() function shall open a directory stream corresponding to the directory named by the dirname argument(home directory)

Step 4: The readdir() function returns a pointer to a dirent structure representing the next directory entry in the directory stream pointed to by dr.

Step 5: The closedir() function closes the directory stream associated with dir. The directory stream descriptor dr is not available after this call.

Step 6: Stop the program

SOURCE CODE:

```
#include <stdio.h>
#include <dirent.h>
int main(void)
{
    struct dirent *de;  // Pointer for directory entry
    // opendir() returns a pointer of DIR type.
    DIR *dr = opendir(".");
    if (dr == NULL) // opendir returns NULL if couldn't open directory
   {
            printf("Could not open current directory" );
             return 0;
    }
    while ((de = readdir(dr)) != NULL)
            printf("%s\n", de->d_name);
   closedir(dr);
    return 0;
}
```

OUTPUT:

Opens the home directory and prints the contents of it.

EXPERIMENT 3

OBJECTIVE

Write a C program to simulate Bankers Algorithm for Deadlock Avoidance and Prevention.

DESCRIPTION

DEADLOCK AVOIDANCE

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a deadlock. Deadlock avoidance is one of the techniques for handling deadlocks. This approach requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, it can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process. Banker's algorithm is a deadlock avoidance algorithm that is applicable to a system with multiple instances of each resource type

ALGORITHM:

Step 1: Input the number of processes and number of resources.

Step 2: Input the Max matrix.

Step 3: Input the Allocation matrix.

Step 4:Input Available Resources after allocation.

Step 5:Calculation of Need Matrix

need[i][j]=max[i][j]-alloc[i][j]

Step 6:Apply the safety algorithm

complete[i] = 0; for i=1, 2, 3, 4….n

Step 7: if avail[j]<need[i][j] break

Step 8: else the process getsexecuted and add the process to safe sequence and

avail[j]+=alloc[process][j];

Step 9: If all the processes are completed the system is in safe sequence

Step 10: Otherwise the system is not in the safe state.

SOURCE CODE:

```
#include<stdio.h>
int main()
{
int max[10][10],need[10][10],alloc[10][10],avail[10],completed[10], safeSequence[10];
int p,r,I,j,process,count=0;
printf("\nEnter the no. of processes:");
scanf("%d",&p);
for(i=0;i<p;i++)
        completed[i]=0;
printf("\nEnter the no of resources:");
scanf("%d",&r);
//Input Max matrix
printf("\nEnter the Max Matrix for each process:");
for(i=0;i<p;i++)
{
        printf("\n For process %d:",i+1);
        for(j=0;j<r;j++)
                scanf("%d",&max[i][j]);
}
// Input Allocation Matrix
printf("\nEnter the allocation for each process:");
for(i=0;i<p;i++)
{
        printf("\n For process %d:",i+1);
        for(j=0;j<r;j++)
                scanf("%d",&alloc[i][j]);
}
// Input Available Resources after allocation
printf("\n\n Enter the Available Resources:");
```

```
for(i=0;i<r;i++)
        scanf("%d",&avail[i]);
// Calculation of Need Matrix
for(i=0;i<p;i++)
        for(j=0;j<r;j++)
                need[i][j]=max[i][j]-alloc[i][j];
do
{
printf(" Need Matrix :\n" );
for(i=0;i<p;i++)
{
        for(j=0;j<r;j++)
                printf("%d\t\t", need[i][j]);
printf("\n");
}
process=-1;
for(i=0;i<p;i++)
{
        if(completed[i]==0)
        {
        process=I;
        for(j=0;j<r;j++)
        {
                if(avail[j]<need[i][j])
                {
                process=-1;
                break;
                }
        }
}
```

```
if(process!=-1)
break;
}
if(process!=-1)
{
        printf("\n Process %d runs to completion!:", process+1);
        safeSequence[count]=process+1;
        count++;
        for(j=0;j<r;j++)
        {
                avail[j]+=alloc[process][j];
                alloc[process][j]=0;
                max[process][j]=0;
                need[process][j]=0;
                completed[process]=1;
                printf("\n Available resources %d \n", avail[j]);
        }
}
} while (count!=p && process!=-1);
if(count==p)
{
        printf("\n The system is in safe sate!\n");
        printf("Safe Sequence:<");
        for(i=0;i<p;i++)
                printf("%d", safeSequence[i]);
        printf(">\n");
}
else
        printf("The system is in unsafe state!");
}
```

OUTPUT:

Enter the no. of Processes:5

Enter the no of Resources:3

Enter the Max Matrix for each Process:

For process 1:7 5 3

For process 2:3 2 2

For process 3:9 0 2

For process 4:2 2 2

For process 5:4 3 3

Enter the allocation for each Process:

For process 1:0 1 0

For process 2:2 0 0

For process 3:9 0 2

For process 4:2 2 2

For process 5:4 3 3

Enter the Available Resources:3 3 2

Need Matrix :

7	4	3
1	2	2
6	0	0
0	1	1
4	3	1

Process 2 runs to completion!:

Available Resources 5

Available Resources 3

Available Resources 2

Need Matrix :

7	4	3
0	0	0

6 0 0

0 1 1

4 3 1

Process 4 runs to completion!:

Available Resources 7

Available Resources 4

Available Resources 3

Need Matrix :

7 4 3

0 0 0

6 0 0

0 0 0

4 3 1

Process 1 runs to completion!:

Available Resources 7

Available Resources 5

Available Resources 3

Need Matrix :

0 0 0

0 0 0

6 0 0

0 0 0

4 3 1

Process 3 runs to completion!:

Available Resources 10

Available Resources 5

Available Resources 5

Need Matrix :

0 0 0

0 0 0

0 0 0

0 0 0

4 3 1

Process 5 runs to completion!:

Available Resources 10

Available Resources 5

Available Resources 7

The system is in safe sate!

Safe Sequence:<24135>

EXPERIMENT 4

OBJECTIVE

Write a C program to implement the Producer – Consumer problem using semaphores using UNIX/LINUX system calls.

DESCRIPTION

Producer-consumer problem, is a common paradigm for cooperating processes. A producer process produces information that is consumed by a consumer process. One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, there must be available a buffer of items that can be filled by the producer and emptied by the consumer. This buffer will reside in a region of memory that is shared by the producer and consumer processes. A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

ALGORITHM:

Step 1: The Semaphore full & empty are initialized.

Step 2: In the case of producer process

i) Produce an item in to temporary variable.

ii) If there is empty space in the buffer check the mutex value for enter into the critical section.

iii) If the mutex value is 0, allow the producer to add value in the temporary variable to the buffer.

Step 3: In the case of consumer process

i) It should wait if the buffer is empty

ii) If there is any item in the buffer check for mutex value, if the mutex==0, remove item from buffer

iii) Signal the mutex value and reduce the empty value by 1.

iv) Consume the item.

Step 4: Print the result

SOURCE CODE:

```
#include<stdio.h>
#include<stdlib.h>
int full=0,empty=3,x=0;
main()
{
int n;
void  producer();
void consumer();
int wait(int);
int signal(int);
printf("\n1.PRODUCER\n2.CONSUMER\n3.EXIT\n");
while(1)
{
        printf("\nENTER YOUR CHOICE\n");
        scanf("%d",&n);
        switch(n)
        {
                case 1:
                        if(empty!=0)
                                producer();
                        else
                                printf("BUFFER IS FULL");
                 break;
                case 2:
                        if(full!=0)
                                consumer();
                        else
                                printf("BUFFER IS EMPTY");
                break;
                case 3:
                        exit(0);
                break;
```

```
        }
}
}

int wait(int s)
{
        return(--s);
}

int signal(int s)
{
        return(++s);
}

void producer()
{
        full=signal(full);
        empty=wait(empty);
        x++;
        printf("\n Producer produces the item%d \n",x);
}

void consumer()
{
        full=wait(full);
        empty=signal(empty);
        printf("\n Consumer consumes item%d \n",x);
        x--;
}
```

OUTPUT:

1. PRODUCER
2. CONSUMER
3. EXIT

ENTER YOUR CHOICE
1

Producer produces the item1
ENTER YOUR CHOICE
1

Producer produces the item2
ENTER YOUR CHOICE
1

Producer produces the item3
ENTER YOUR CHOICE
2

Consumer consumes item3
ENTER YOUR CHOICE
1

Producer produces the item 3
ENTER YOUR CHOICE
3

EXPERIMENT 5

OBJECTIVE

Write C programs to illustrate the following IPC mechanisms

a) Pipes b) FIFOs c) Message Queues d) Shared Memory

DESCRIPTION

a) PIPES

Pipes are unidirectional byte streams which connect the standard output from one process into the standard input of another process. A pipe is created using the system call pipe that returns a pair of file descriptors.

A pipe is created by the pipe() system call.

Synopsis

int pipe (int *filedes) ;

Description

Call to the pipe () function which returns an array of file descriptors fd[0] and fd [1].

fd [1] connects to the write end of the pipe, and

fd[0] connects to the read end of the pipe.

Anything can be written to the pipe, and read from the other end in the order it came in.

It can be used only between parent and child processes.

RETURNS: 0 on success

-1 on error: errno = EMFILE (no free descriptors)

EMFILE (system file table is full)

EFAULT (fd array is not valid)

fd[0] is set up for reading, fd[1] is set up for writing. i.e., the first integer in the array (element 0) is set up and opened for reading, while the second integer (element 1) is set up and opened for writing.

When we use fork in any process, file descriptors remain open across child process and also parent process. If we call fork after creating a pipe, then the parent and child can communicate via the pipe.

b) FIFOs

Pipes were meant for communication between related processes. We can achieve communication between unrelated processes using Named Pipes. Another name for named pipe is FIFO (First-In-First-Out).

A fifo is created by mkfifo () system call.

Synopsis

#include <sys/types.h>#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);

Description

mkfifo() makes a FIFO special file with name pathname.

mode specifies the FIFO's permissions. It is modified by the process's umask in the usual way: the permissions of the created file are (mode & ~umask).

Once you have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file. However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it.

Opening a FIFO for reading normally blocks until some other process opens the same FIFO for writing, and vice versa.

Return Value

On success mkfifo() returns 0. In the case of an error, -1 is returned (in which case, errno is set appropriately).

c) MESSAGE QUEUES

Message queues provide a form of message passing in which any process (given that it has the necessary permissions) can read a message from or write a message to any IPC message queue on the system. There are no requirements that a process be waiting to receive a message from a queue before another process sends one, or that a message exist on the queue before a process requests to receive one.

ftok()

This generates an IPC key on the basis of supplied filename and ID. The filename can be provided along with its complete path. The file name must refer to an existing file.

Synopsis

ket_t ftok(const char *filename,int id);

Description

This function will generate the same key value if the same filebame and the same ID is supplied. Upon successful completion ftok will return a key, Otherwise it will return -1.

msgget()

This is used for creating a new message queue and for accessing an existing queue that is related to the specified key. If this is executed successfully, the function returns the identifier of the message queue.

Synopsis

int msggt(key_t,int flag)

Description

key: this is a uniquekey value that is retrieved by invoking the ftok function

flag: this can be any of the following constants;

IPC_CREAT: Creates the message queue if it doesn't already exist and returns the message queue identifier for the newly created message queue. If the message queue already exists with the supplied key value, it returns its identifier.

IPC_EXCL: If both IPC_CREAT and IPC_EXCL are specified and the message queue does not exist, then it is created. However, if it already exists then function will fail.

msgrcv()

This is used for reading a message form the specified message queue whose identifier is supplied.

Synopsis

int msgrcv(int msqid,void *msgstruc,int msgsize,long typemsg,int flag);

Description

msqid: Represents the message queue identifier of the queue from which the message needs to be read.

msgstruc: This is the user-defined structure into which the read message is placed. The user-defined structure must contain two members. One is usually named mtype, which must be of

type long int that specifies the type of message and the second is usually called mesg, which should be of char type to store message.

msgsize: Represents the size of the text to be read from the message queue in terms of bytes. If the message that is read is larger than msgsize then it will be truncated to msgsize bytes.

typemesg: Specifies which message on the queue needs to be received:

If typemsg is 0, the first message o the queue needs to be received.

If typemsg is greater than 0, the first message whose mtype field is equal to the typemsg is received

If typemsg is less than 0, a message whose mtype field is less than or equal to the typemsg is received

flag: Determines the action to be taken if the desired message is not found in the queue. It keeps its value of 0 if you don't want to specify the flag.

The flag can have any of the following values:

IPC_NOWAIT : This makes the msgrcv function fail if there is no desired message in the queue, that it will not make the caller wait for the appropriated message on the queue. If flag is not set to IPC_NOWAIT, it will make the caller wait for an appropriate message on the queue instead of failing the function.

MSG_NOERROR: This allows you to receive text that is larger than the size that's specified in the msgsize argument It simply truncates the text and receives it. If the flag is not set, on receiving the larger text, the function will not receive it and will fail the function.

If the function is executed successfully, the function returns the number of bytes that were placed into the text field of he structure that is pointed to by msgstruc. On failure the function returns a value of -1.

msgsnd()

This is used for sending or delivering a message to the queue.

Synopsis

int msgsnd(int msqid,struct msgbuf &msgstruc,in msgsize,int flag);

Description

msqid: Represent the queue identifier of the message that we want to send. The queue identifier is usually retrieved by invoking msgget function.

msgstruc: This is a pointer to the user-defined structure. It is the mesg member that contains the message that wee want to send to the queue.

msgsize: Represents the size of the message in bytes.

flag: Determines the action to be taken on the message. If the flag value is set to IPC_NOAIT and if the message queue is full the message will not be written to the queue and the control is returned to the calling process. But if flag is not set and the message queue is full, then the calling process will suspend until a space becomes available in the queue. Usually the value of flag is set to 0

If this is executed successfully, the function returns 0, otherwise it returns -1.

d) SHARED MEMORY

Inter Process Communication through shared memory is a concept where two or more process can access the common memory. And communication is done via this shared memory where changes made by one process can be viewed by another process.

The problem with pipes, fifo and message queue – is that for two process to exchange information.

- The information has to go through the kernel.
- Server reads from the input file.
- The server writes this data in a message using either a pipe, fifo or message queue.
- The client reads the data from the IPC channel,again requiring the data to be copied from kernel's IPC buffer to the client's buffer.
- Finally the data is copied from the client's buffer.

A total of four copies of data are required (2 read and 2 write). So, shared memory provides a way by letting two or more processes share a memory segment. With Shared Memory the data is only copied twice – from input file into shared memory and from shared memory to the output file.

ftok(): is used to generate a unique key.

shmget()

The above system call creates or allocates a shared memory segment.

Synopsis

int shmget(key_t key,size_tsize,intshmflg);

Description

key: It recognizes the shared memory segment. The key can be either an arbitrary value or one that can be derived from the library function ftok(). The key can also be IPC_PRIVATE, means, running processes as server and client (parent and child relationship) i.e., inter-related process communication. If the client wants to use shared memory with this key, then

it must be a child process of the server. Also, the child process needs to be created after the parent has obtained a shared memory.

Upon successful completion, shmget() returns an identifier for the shared memory segment.

shmat():

The above system call performs shared memory operation for System V shared memory segment i.e., attaching a shared memory segment to the address space of the calling process. Before you can use a shared memory segment, you have to attach yourself to it using shmat().

Synopsis

void *shmat (int shmid, void *shmaddr,int shmflg);

Description

shmid : Itis the identifier of the shared memory segment. This id is the shared memory identifier, which is the return value of shmget() system call.

shmaddr : It is to specify the attaching address. If shmaddr is NULL, the system by default chooses the suitable address to attach the segment. If shmaddr is not NULL and SHM_RND is specified in shmflg, the attach is equal to the address of the nearest multiple of SHMLBA (Lower Boundary Address). Otherwise, shmaddr must be a page aligned address at which the shared memory attachment occurs/starts.

shmflg: specifies the required shared memory flag/s such as SHM_RND (rounding off address to SHMLBA) or SHM_EXEC (allows the contents of segment to be executed) or SHM_RDONLY (attaches the segment for read-only purpose, by default it is read-write) or SHM_REMAP (replaces the existing mapping in the range specified by shmaddr and continuing till the end of segment).

This call would return the address of attached shared memory segment on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

shmdt():

This system call performs shared memory operation for shared memory segment of detaching the shared memory segment from the address space of the calling process.

When you’re done with the shared memory segment, your program should detach itself from it using shmdt().

Synopsis

int shmdt(void *shmaddr);

Description

The argument, shmaddr, is the address of shared memory segment to be detached. The to-be-detached segment must be the address returned by the shmat() system call.

This call would return 0 on success and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

shmctl()

This system call performs control operation for a shared memory segment.

when you detach from shared memory,it is not destroyed. So, to destroy shmctl() is used.

Synopsis

shmctl(int shmid,IPC_RMID,NULL);

Description

shmid: This is the identifier of the shared memory segment. This id is the shared memory identifier, which is the return value of shmget() system call.

cmd: This is the command to perform the required control operation on the shared memory segment.

Valid values for cmd are −

IPC_STAT − Copies the information of the current values of each member of struct shmid_ds to the passed structure pointed by buf. This command requires read permission to the shared memory segment.

IPC_SET − Sets the user ID, group ID of the owner, permissions, etc. pointed to by structure buf.

IPC_RMID − Marks the segment to be destroyed. The segment is destroyed only after the last process has detached it.

IPC_INFO − Returns the information about the shared memory limits and parameters in the structure pointed by buf.

SHM_INFO − Returns a shm_info structure containing information about the consumed system resources by the shared memory.

buf : Thus is a pointer to the shared memory structure named struct shmid_ds. The values of this structure would be used for either set or get as per cmd.

This call returns the value depending upon the passed command. Upon success of IPC_INFO and SHM_INFO or SHM_STAT returns the index or identifier of the shared memory segment or 0 for other operations and -1 in case of failure. To know the cause of failure, check with errno variable or perror() function.

a) Pipes

ALGORITHM:

Step 1: Start the program

Step 2: Create a pipe using pipe() system call

Step 3: Create a child process.If the child process is created successfully then write the message into the pipe otherwise goto step2

Step 4: Read the message from the pipe and display the message

Step 5: Stop the program

SOURCE CODE:

```
#include<stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>
int main()
{
intfd[2],child;
char a[]=" Hello PEC\n";
pipe(fd);
child=fork();
if(! child)
{
        close(fd[0]);
        write(fd[1],a,strlen(a));
        wait(0);
}
else
{
        close(fd[1]);
        read(fd[0],a,sizeof(a));
        printf("\n\n The string retrieved from the pipe is %s",a);
}
return 0;
}
```

b) FIFOs

ALGORITHM:

Step 1: Create two processes, one is fifowriter and another one is fiforeader.

Step 2: Writer process performs the following −

- Creates a named pipe (using system call mkfifo()) with name “MYFIFO”, if not created.
- Opens the named pipe for write only purposes.
- Writes the data into the FIFO

Step 3: Reader process performs the following −

- Opens the named pipe for read only purposes.
- Reads the content from the FIFO and put in to the buffer.
- Writes the content out from the buffer on the screen.

SOURCE CODE:

```
/* Filename: fifowrite.c */
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define FIFO_FILE "MYFIFO"
int main()
{
int fd;
   char buffer[80]=”WRITER DATA”;
      /* Create the FIFO */
mkfifo(FIFO_FILE, 0666);
fd= open(FIFO_FILE, O_WRONLY);
write(fd,buffer,sizeof(buffer));
close(fd);
return 0;
}

/* Filename: fiforead.c */
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>

#define FIFO_FILE "MYFIFO"
int main()
{
```

```
int fd1;
charbufferr[80];
   fd1=open(FIFO_FILE,O_RDONLY);
read(fd1,buffer,sizeof(buffer));
printf("       has been sent by writer”);
write(1,buffer,sizeof(buffer));
close(fd1);
return 0;
}
```

OUTPUT:

Step 1: First compile the program for writing message into the fifo.

Step 2: Then run the program.

Step 3: Now open another terminal and compile the program for reading the message from fifo.

Step 4: Then run the program

Output of the program for reading the message from the fifo

WRITER DATA has been sent by writer

C) Message Queues

ALGORITHM:

Step 1: Generate an IPC key by invoking the ftok function. A filename and ID are supplied while creating the IPC key.

Step 2: Invoke the msgget function to create the message queue. The message queue is associated with the IPC key that was created in step1

Step 3: Define a structure with two members, mtype and mesg. Set the value of mtype member to 1

Step 4: Enter the message that's going to be added to the message queue. The string that's entered is assignedto the mesg member of the structure that was defined in step 3;

Step 5: Invoke the msgsnd function to send the entered message in to the message queue

SOURCE CODE:

```
// Program for writing the message to the message queue
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#inlude<stdio.h>
#inlucde<string.h>
#define MSGSIZE 255
struct msgstruc
{
long mtype;
char mesg[MSGSIZE];
};
void main()
{
int msqid,msglen;
key_t key;
struct msgstruc msgbuf;
system("touch messagefile");
if((key= ftok("messagefile",'a'))==-1)
{
perror("ftok");
exit(1);
}
if((msqid=msgget(key,0666|IPC_CREAT))==-1)
{
perror("msgget");
exit(1);
}
msgbus.mtype=1;
printf("Enter a message to add to message queue:");
scanf("%s",msgbuf.mesg);
msglen=strlen(msgbuf.mesg);
```

```
if(msgsnd(msqid,&msgbuf,msglen,IPC_NOWAIT)<0)
perror("msgsnd");
printf("the message sent is %s\n",msgbuf.mesg);
return 0;
}
```

ALGORITHM:

Step 1: Invoke the ftok function to generate the IPC key. The filename and ID are supplied while creating the IPC key. These must be the same as what were applied while generating the key for writing the message in the message queue.

Step 2: Invoke the msgget function to access the message queue that is associated with the IPC key. The message queue that's associated with this key already contains a message that we wrote through the previous program.

Step 3: Define a structure with two members, mtype and mesg.

Step 4: Invoke the msgrcv function to read the message from the associated message queue. The structure that was defined in step 3 is passed to this function.

Step 5:The read message is then displayed on the screen.

SOURCE CODE:

```
// Program for reading a message from the message queue
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#inlude<stdio.h>
#inlucde<string.h>
#inlcude<stdlib.h>
#define MSGSIZE 255

struct msgstruc
{
        long mtype;
        char mesg[MSGSIZE];
};
int main()
{
int msqid;
key_t key;
struct msgstruc rcvbuffer;
if((key= ftok("messagefile",'a'))==-1)
{
        perror("ftok");
        exit(1);
}
if((msqid=msgget(key,0666))<0)
{
        perror("msgget");
        exit(1);
}
if(msgrcv(msqid,&rcvbuffer,MSGSIZE,1,0)<0)
{
        perror("msgrcv");
```

```
        exit(1);
}

printf("The message received  is %s\n",rcvbuffer.mesg);
return 0;
}
```

OUTPUT:

Step 1: First compile the program for writing message in to the queue.

Step 2: Then run the program

Output of the program for writing the message into the message queue

Enter a message to add to message queue: GoodDay

The message id GoodDay

Step 3: Now open another terminal and compile the program for reading the message from the queue.

Step 4: Then run the program

Output of the program for reading the message from the message queue

The message received id GoodDay

d) SHARED MEMORY

ALGORITHM:

Step 1: Generate an IPC key by invoking the ftok function. A filename and ID are supplied while creating the IPC key.

Step 2: Invoke the shmget function to create the shared memory. The shared memory is associated with the IPC key that was created in step1.

Step 3: Invoke the shmat function to

Step 4: Enter the message that's going to be added to the shared memory. The string that's entered is assigned to the str.

Step 5: Invoke the shmdt function to

SOURCE CODE:

```
//Program for writing in to the shared memory
#include<stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
        char *str;
        int shmid;

        key_t key = ftok("sharedmem",'a');
        if ((shmid = shmget(key, 1024,0666|IPC_CREAT)) < 0) {
                perror("shmget");
                exit(1);
        }
        if ((str = shmat(shmid, NULL, 0)) == (char *) -1) {
                perror("shmat");
                exit(1);
        }
        printf("Enter the string to be written in memory : ");
        gets(str);
        printf("String written in memory: %s\n",str);
        shmdt(str);
        return 0;
}
```

ALGORITHM:

Step 1: Invoke the ftok function to generate the IPC key. The filename and ID are supplied while creating the IPC key. These must be the same as what were applied while generating the key for writing the message in the shared memory.

Step 2: Invoke the shmget function to access the shared memory that is associated with the IPC key. The message queue that's associated with this key already contains a message that we wrote through the previous program.

Step 3: Invoke the shmat function to

Step 4: Display the message that's being written to the shared memory.

Step 5: Invoke the shmdt function to

Step 6: Invoke the shmctl function to

SOURCE CODE:

```
//Program for reading from the memory
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>

int main()
{
        int shmid;
        char * str;

        key_t key = ftok("sharedmem",'a');
        if ((shmid = shmget(key, 1024,0666|IPC_CREAT)) < 0) {
                perror("shmget");
                exit(1);
        }
        if ((str = shmat(shmid, NULL, 0)) == (char *) -1) {
                perror("shmat");
                exit(1);
        }
        printf("Data read from memory: %s\n",str);
        shmdt(str);
        shmctl(shmid,IPC_RMID,NULL);
        return 0;
}
```

OUTPUT:

Step 1: First compile the program for writing message into the shared memeory.

Step 2: Then run the program

Output of the program for writing the message into the shared memory

Enter the string to be written in memory: GoodDay

String written in memory: GoodDay

Step 3: Now open another terminal and compile the program for reading the message from the shared memory.

Step 4: Then run the program

Output of the program for reading the message from the shared memory

Data read from memory: GoodDay

EXPERIMENT 6

OBJECTIVE

Write C programs to simulate the following memory management techniques

a) Paging b) Segmentation

PAGING

In computer operating systems, paging is one of the memory management schemes by which a computer stores and retrieves data from the secondary storage for use in main memory. In the paging memory-management scheme, the operating system retrieves data from secondary storage in same-size blocks called pages. Paging is a memory-management scheme that permits the physical address space a process to be noncontiguous. The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory into blocks of the same size called pages. When a process is to be executed, its pages are loaded into any available memory frames from their source.

SEGMENTATION

Like Paging, Segmentation is also a memory management scheme. It supports the user's view of the memory. The process is divided into the variable size segments and loaded to the logical memory address space.

The logical address space is the collection of variable size segments. Each segment has its name and length. For the execution, the segments from logical memory space are loaded to the physical memory space.The address specified by the user contain two quantities the segment name and the Offset. The segments are numbered and referred by the segment number instead of segment name. This segment number is used as an index in the segment table, and offset value decides the length or limit of the segment. The segment number and the offset together generates the address of the segment in the physical memory space.

ALGORITHM:

Step 1: Read the logical address, page size and physical address.

Step 2: calculate the number of pages and number of frames and display.

Step 3: Create the page table with the page number page and page address.

Step 4: Read the page number and offset value.

Step 5: If the page number and offset value is valid, add the offset value with the address corresponding to the page number and display the result.

Step 6: Display the page is not found or error message.

SOURCE CODE:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
main()
{
int page_size,no_of_pages,no_of_frames,logical_add;
int alloc[50],base[50],frame[50],page[50];
int i,f,n,physical_add,frame_sizes,pg_no,add,offset;
int temp;
int f1;
clrscr();
printf("\n\t\t PAGING\n");
printf("\n\t Enter the logical address space:");
scanf("%d",&logical_add);
printf("\n\t Enter the page size:");
scanf("%d",&page_size);
printf("\n\t Enter the physical address space:");
scanf("%d",&physical_add);
frame_sizes =page_size;
no_of_pages=logical_add/page_size;
no_of_frames=physical_add/frame_sizes;
printf("\n\t Number of pages = %d",no_of_pages);
printf("\n\t Number of frames = %d",no_of_frames);
for(i=0;i<no_of_frames;i++)
alloc[i]=0;
for(i=0;i<no_of_pages;i++)
{
temp=rand()%no_of_frames;
while(alloc[temp]==1)
temp=rand()%no_of_frames;
alloc[temp]=1;
frame[i]=temp;
```

```
}
printf("\n Page No \t Frame No \t Base address ");
for(i=0;i<no_of_pages;i++)
{
base[i]=frame[i]*page_size;
page[i]=i;
printf("%d\t %d\t %d\t",i,frame[i],base[i]);
}
printf("\n\t Enter the Page num and Offset : ");
scanf(" %d %d",&pg_no,&offset);
for(i=0;i<no_of_pages;i++)
{
if(pg_no ==page[i])
{
add=base[i]+offset;
f=1;
break;
}
}
if(offset>=page_size)
printf("\n\t Trying to access other page");
else
{
if(f==1)
printf("\n\t Physical Address = %d",physical_add);
else
printf("\n\t Page not found");
getch();
}
}
}
```

Enter the logical address space:500
Enter the page size:100

Enter the physical address space:1000

Number of pages =5

Number of frames=10

Page No	Frame No	Base address
0	6	600
1	0	0
2	2	200
3	7	700
4	5	500

Enter the page number and offset: 3 4

Physical address = 704

ALGORITHM:

Step 1: Read the segment numbers, limit address and base address of the segments.

Step 2: Create the segment table with the segment number and segment details.

Step 3: Read the logical address.

Step 4: If the logical address value is valid, add the base address with the logical address and display the result as physical address.

Step 5: Display the -1 for physical address if logical address is not valid.

SOURCE CODE:

```
#include<stdio.h>
void main()
{
        int i,n,seg[20],lt[20],base[20],log[20],phy[20];
        printf("\n Enter the number of segments:");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("\n Enter the segment no of %d :",i+1);
                scanf("%d",&seg[i]);
        }
        for(i=0;i<n;i++)
        {
                printf("\n Enter the limit address of %d :",i+1);
                scanf("%d",&lt[i]);
        }
for(i=0;i<n;i++)
        {
                printf("\n Enter the base address of %d :",i+1);
                scanf("%d",&base[i]);
        }
        printf("\n The segment no \t\t segment table \n");
        for(i=0;i<n;i++)
        {
                printf("\n %d \t %d \t %d",seg[i],lt[i],base[i]);
                printf("\n");
        }
        for(i=0;i<n;i++)
        {
                printf("\n Enter the logical no of %d :",i+1);
                scanf("%d",&log[i]);
        }
        for(i=0;i<n;i++)
        {
                if(log[i]<lt[i])
```

```
                {
                        phy[i]=log[i]+base[i];
                }
                else
                {
                        phy[i]=-1;
                }
        }
        printf("\n Physical address");
        for(i=0;<n;i++)
        {
                printf("\n %d",phy[i]);
        }
}
```

OUTPUT :

Enter the number of segments:5

Enter the segment no of 1 :1

Enter the segment no of 2 :2

Enter the segment no of 3 :3

Enter the segment no of 4 :4

Enter the segment no of 5 :5

Enter the limit address of 1 :5

Enter the limit address of 2 :4

Enter the limit address of 3 :6

Enter the limit address of 4 :4

Enter the limit address of 5 :7

Enter the base address of 1 :20

Enter the base address of 2 :15

Enter the base address of 3 :5

Enter the base address of 4 :28

Enter the base address of 5 :50

The segment no	segment table	
1	5	20
2	4	15
3	6	5
4	4	28
5	7	50

Enter the logical no of 1 :2

Enter the logical no of 2 :3

Enter the logical no of 3 :7

Enter the logical no of 4 :5

Enter the logical no of 4 :5

Physical address

22

18

-1

-1

55