



## **VIGNAN'S INSTITUTE OF MANAGEMENT AND TECHNOLOGY FOR WOMEN**

Sponsored by Lavu Educational Society

Affiliated to JNTU, Hyderabad & Approved by AICTE, New Delhi

Kondapur (V), Ghatkesar (M), Medchal-Malkajgiri (Dist.)-501301, Ph: 9652910002/3



# **Lab Manual**

## **OPERATING SYSTEMS**

**R22 B.Tech. CSE II Year II Sem.**

# **VISION&MISSIONOF THEINSTITUTE&DEPARTMENT**

## **VisionoftheDepartment&MissionoftheCollege**

### **VisionoftheCollege:**

Toempowerstudentswithprofessionaleducationusingcreative&innovativetechnicalpractices of global competence and research aptitude tobecomecompetitiveengineerswith ethical values and entrepreneurialskills.

### **Missionofthecollege:**

To impart value based professional education through creative and innovative teaching- learning processto face the globalchallenges ofthe new era technology.

To inculcate research aptitude and to bring out creativity in students by imparting engineering knowledge imbining interpersonal skills to promote innovation, research andentrepreneurship.

## **VisionoftheDepartment&MissionoftheDepartment**

### **VisionoftheDepartmentofCSE:**

Debouch as a center of excellence for computer science engineering by imparting social, moral and ethical values oriented education through advanced pedagogical techniques and produce technologically and highly competent professionals of global standards with capabilities of solving challenges of the time through innovative and creative solutions.

### **MissionoftheDepartmentofCSE:**

✚To envision inquisitive-driven advanced knowledge building among students to impart foundational knowledge of computer science and its applications of all spheres using the state-of-the-art facilities and software industry-instituteinteraction.

✚To advance the department industry collaborations through interaction with professional society through seminars/workshops/guest lectures and student internship programs.

To nurture students withleadershipqualities, communication skills andimbibe qualities to work asateammemberandaleaderfortheeconomicalandtechnologicaldevelopmentincuttingedge technologies in national and globalarena.

### **Program Educational Objectives (PEOs):**

**PEO1:** To provide graduates the foundational and essential knowledge in mathematics, science, computer science and engineering and interdisciplinary engineering to emerge as technocrats.

**PEO2:** To inculcate the capabilities to analyse, design and develop innovative solutions of computer support systems for benefits of the society, by diligence and teamwork.

**PEO3:** To drive the graduates toward employment/pursue higher studies/turn as entrepreneurs

### **Program 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 analyze 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 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 modeling 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 the 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 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 lifelong learning in the broadest context of technological change.

### **Program Specific Outcomes (PSOs):**

**PSO1: Foundation on Software Development:**

Ability to grasp the software development life cycle of software systems and possess competent skills and knowledge of software design process

**PSO2: Industrial Skills Ability:**

Ability to interpret fundamental concepts and methodology of computer systems so that students can understand the functionality of hardware and software aspects of computer systems

**PSO3: Ethical and Social Responsibility:**

Communicate effectively in both verbal and written form, will have knowledge of professional and ethical responsibilities and will show the understanding of impact

of engineering solutions on the society and also will be aware of contemporary issues.

**R22 B.Tech. CSE Syllabus JNTU Hyderabad**  
**OPERATING SYSTEMS LAB**  
**B.Tech. II Year II Sem.**

**Prerequisites:** A course on “Programming for Problem Solving”, A course on “Computer Organization and Architecture”.

**Co-requisite:** A course on “Operating Systems”.

**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
7. Write C programs to simulate Page replacement policies a) FCFS b) LRU c) Optimal

**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

## WEEK-1

### 1. Write C programs to simulate the following CPU Scheduling algorithms

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

#### a) FCFS(First come first serve) Scheduling algorithm:

In the "First come first serve" scheduling algorithm, as the name suggests, the process which arrives first, gets executed first, or we can say that the process which requests the CPU first, gets the CPU allocated first.

First Come First Serve, is just like FIFO(First in First out) Queue data structure, where the data element which is added to the queue first, is the one who leaves the queue first. This is used in Batch Systems. It's easy to understand and implement.

#### Algorithm for FCFS scheduling:

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: Set the waiting of the first process as '0' and its burst time as its turn around time

Step 5: for each process in the Ready Q calculate

Turn around time for Process(n)= Completion Time - Arrival Time

Waiting time for process(n)= Turnaround time - Burst Time

Step 6: Calculate

Average waiting time = Total waiting Time / Number of process

Average Turnaround time = Total Turnaround Time / Number of process

Step 7: Stop the process.

#### Program to Simulate First Come First Serve CPU Scheduling Algorithm :

```
#include<stdio.h>
#include<string.h>
void main()
{
    int i,j,n,bt[10],compt[10],at[10], wt[10],tat[10];
    float sumwt=0.0,sumtat=0.0,avgwt,avgtat;
    printf("Enter number of processes: ");
    scanf("%d",&n);
    printf("Enter the burst time of %d process\n", n);
    for(i=0;i<n;i++)
    {
        scanf("%d",&bt[i]);
```

```

    }
    printf("Enter the arrival time of %d process\n", n);
    for(i=0;i<n;i++)
    {
        scanf("%d",&at[i]);
    }
    compt[0]=bt[0]-at[0];
    for(i=1;i<n;i++)
    compt[i]=bt[i]+compt[i-1];
    for(i=0;i<n;i++)
    {
        tat[i]=compt[i]-at[i];
        wt[i]=tat[i]-bt[i];
        sumtat+=tat[i];
        sumwt+=wt[i]; }
    avgwt=sumwt/n;
    avgtat=sumtat/n;
    printf("----- \n");
    printf("PN\tBt\tCt\tTat\tWt\n");
    printf("----- \n");
    for(i=0;i<n;i++)
    {
        printf("%d\t%d\t%d\t%d\t%d\n",i,bt[i],compt[i],tat[i],wt[i]);
    }
    printf("-----\n");
    printf(" Avgwt = %.2f\tAvgtat = %.2f\n",avgwt,avgtat);
    printf("-----\n");
}

```

### **OUTPUT :**

```

Enter number of processes: 3
Enter the burst time of 3 process
24
3
3
Enter the arrival time of 3 process
0
0
0

```

```

-----
PN   Bt   Ct   Tat   Wt
-----
0    24   24   24    0
1    3    27   27    24
2    3    30   30    27
-----

```

Avgwt = 17.00 Avgtat = 27.00  
-----

### **b) SJF(Shortest Job First) Scheduling algorithm:**

Shortest Job First scheduling works on the process with the shortest **burst time** or **duration** first. This is the best approach to minimize waiting time. This is used in Batch Systems. It is of two types: Non Pre-emptive and Pre-emptive. To successfully implement it, the burst time/duration time of the processes should be known to the processor in advance, which is practically not feasible all the time. This scheduling algorithm is optimal if all the jobs/processes are available at the same time. (either Arrival time is 0 for all, or Arrival time is same for all).

The criteria of this algorithm are which process having the smallest CPU burst, CPU is assigned to that next process. If two process having the same CPU burst time FCFS is used to break the tie.

#### **Algorithm for SJF:**

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: For each process in the ready queue, calculate

Turn around time for Process(n)= Completion Time - Arrival Time

Waiting time for process (n)= Turnaround time - Burst Time

Step 6: Calculate

Average waiting time = Total waiting Time / Number of process

Average Turnaround time = Total Turnaround Time / Number of process

Step 7: Stop the process.

#### **Program to Simulate Shortest Job First CPU Scheduling Algorithm :**

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

```
#include<string.h>
```

```
int main()
```

```
{
```

```
    int i,j,n,bt[10],compt[10], wt[10],tat[10],temp;
```

```
    float sumwt=0.0,sumtat=0.0,avgwt,avgtat;
```

```
    printf("Enter number of processes: ");
```

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

```
    printf("Enter the burst time of %d process\n", n);
```

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

```
    {
```

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

```
    }
```

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

```
    for(j=i+1;j<n;j++)
```

```
    if(bt[i]>bt[j])
```

```
    {
```



```

        temp=bt[i];
        bt[i]=bt[j];
        bt[j]=temp;
    }
    compt[0]=bt[0];
    for(i=1;i<n;i++)
    compt[i]=bt[i]+compt[i-1];
    for(i=0;i<n;i++)
    {
        tat[i]=compt[i];
        wt[i]=tat[i]-bt[i];
        sumtat+=tat[i];
        sumwt+=wt[i];
    }
    avgwt=sumwt/n;
    avgtat=sumtat/n;
    printf("-----\n");
    printf("Bt\tCt\tTat\tWt\n");
    printf("-----\n");
    for(i=0;i<n;i++)
    {
        printf("%2d\t%2d\t%2d\t%2d\n",bt[i],compt[i],tat[i],wt[i]);

    }
    printf("-----\n");
    printf(" Avgwt = %.2f\tAvgtat = %.2f\n",avgwt,avgtat);
    printf("-----\n");
    getch();
}

```

### **OUTPUT :**

Enter number of processes: 4  
Enter the burst time of 4 process

6  
8  
7  
3

Bt	Ct	Tat	Wt
3	3	3	0
6	9	9	3
7	16	16	9
8	24	24	16

Avgwt = 7.00    Avgtat = 13.00

## Shortest Remaining Time First(SRTF) CPU Scheduling Algorithm

This Algorithm is the preemptive version of SJF scheduling. In SRTF, the execution of the process can be stopped after certain amount of time. At the arrival of every process, the short term scheduler schedules the process with the least remaining burst time among the list of available processes and the running process.

Once all the processes are available in the ready queue, No preemption will be done and the algorithm will work as SJF scheduling. The context of the process is saved in the Process Control Block when the process is removed from the execution and the next process is scheduled. This PCB is accessed on the next execution of this process.

### Program to Simulate Shortest Remaining Time First(SRTF) CPU Scheduling Algorithm :

```
#include <stdio.h>
int main()
{
    int arrival_time[10], burst_time[10], temp[10];
    int i, smallest, count = 0, time, n;
    double wait_time = 0, turnaround_time = 0, end;
    float average_waiting_time, average_turnaround_time;
    printf("\nEnter the Total Number of Processes:");
    scanf("%d", &n);
    printf("\nEnter Details of %d Processes", n);
    for(i = 0; i < n; i++)
    {
        printf("\nEnter Arrival Time:");
        scanf("%d", &arrival_time[i]);
        printf("\nEnter Burst Time:");
        scanf("%d", &burst_time[i]);
        temp[i] = burst_time[i];
    }
    burst_time[9] = 9999;
    for(time = 0; count != n; time++)
    {
        smallest = 9;
        for(i = 0; i < n; i++)
        {
            if(arrival_time[i] <= time && burst_time[i] < burst_time[smallest] && burst_time[i] > 0)
            {
                smallest = i;
            }
        }
        burst_time[smallest]--;
        if(burst_time[smallest] == 0)
        {
            count++;
        }
    }
}
```

```

        end = time + 1;
        wait_time = wait_time + end - arrival_time[smallest] - temp[smallest];
        turnaround_time = turnaround_time + end - arrival_time[smallest];
    }
}
average_waiting_time = wait_time / n;
average_turnaround_time = turnaround_time / n;
printf("Average Waiting Time:\t %lf\n", average_waiting_time);
printf("Average Turnaround Time:\t %lf\n", average_turnaround_time);
return 0;
}

```

---

### **OUTPUT :**

```

Enter the Total Number of Processes:4
Enter Details of 4 Processes
Enter Arrival Time:0
Enter Burst Time:5
Enter Arrival Time:1
Enter Burst Time:3
Enter Arrival Time:2
Enter Burst Time:4
Enter Arrival Time:4
Enter Burst Time:1
Average Waiting Time: 2.750000
Average Turnaround Time: 6.000000

```

---

### **c) Round Robin Scheduling algorithm:**

Round Robin(RR) scheduling algorithm is mainly designed for time-sharing systems. This algorithm is similar to FCFS scheduling, but in Round Robin(RR) scheduling, preemption is added which enables the system to switch between processes. A fixed time is allotted to each process, called a **quantum**, for execution. Once a process is executed for the given time period that process is preempted and another process executes for the given time period. Context switching is used to save states of preempted processes. This algorithm is simple and easy to implement and the most important thing is this algorithm is starvation-free as all processes get a fair share of CPU. It is

```

int i, limit, total = 0, x, counter = 0, time_quantum;
int wait_time = 0, turnaround_time = 0, arrival_time[10], burst_time[10], temp[10];
float average_wait_time, average_turnaround_time;
printf("\nEnter Total Number of Processes:\t");
scanf("%d", &limit);
x = limit;
for(i = 0; i < limit; i++)
{
    printf("\nEnter Details of Process[%d]\n", i + 1);
    printf("Arrival Time:\t");
    scanf("%d", &arrival_time[i]);
    printf("Burst Time:\t");
    scanf("%d", &burst_time[i]);
    temp[i] = burst_time[i];
}
printf("\nEnter Time Quantum:\t");
scanf("%d", &time_quantum);
printf("\nProcess ID\t\tBurst Time\t Turnaround Time\t Waiting Time\n");
for(total = 0; i = 0; x != 0;)
{
    if(temp[i] <= time_quantum && temp[i] > 0)
    {
        total = total + temp[i];
        temp[i] = 0;
        counter = 1;
    }
    else if(temp[i] > 0)
    {
        temp[i] = temp[i] - time_quantum;
        total = total + time_quantum;
    }
    if(temp[i] == 0 && counter == 1)
    {
        x--;
        printf("\nProcess[%d]\t\t\t%d\t\t %d\t\t %d", i + 1, burst_time[i], total - arrival_time[i], total - arrival_time[i] - burst_time[i]);
        wait_time = wait_time + total - arrival_time[i] - burst_time[i];
        turnaround_time = turnaround_time + total - arrival_time[i];
        counter = 0;
    }
}

```

```

        if(i == limit - 1)
        {
            i = 0;
        }
        else if(arrival_time[i + 1] <= total)
        {
            i++;
        }
        else
        {
            i = 0;
        }
    }
    average_wait_time = wait_time * 1.0 / limit;
    average_turnaround_time = turnaround_time * 1.0 / limit;
    printf("\nAverage Waiting Time:\t%f", average_wait_time);
    printf("\nAvg Turnaround Time:\t%f\n", average_turnaround_time);
    return 0;
}

```

### **OUTPUT:**

Enter Total Number of Processes: 3

Enter Details of Process[1]

Arrival Time: 0

Burst Time: 24

Enter Details of Process[2]

Arrival Time: 0

Burst Time: 3

Enter Details of Process[3]

Arrival Time: 0

Burst Time: 3

Enter Time Quantum: 4

Process ID	Burst Time	Turnaround Time	Waiting Time
Process[2]	3	7	4
Process[3]	3	10	7
Process[1]	24	30	6

Average Waiting Time: 5.666667

Avg Turnaround Time: 15.666667

Enter Total Number of Processes: 5

Enter Details of Process[1]

Arrival Time: 0

Burst Time: 5

Enter Details of Process[2]

Arrival Time: 1

Burst Time: 3

Enter Details of Process[3]

Arrival Time: 2

Burst Time: 1

Enter Details of Process[4]

Arrival Time: 3

Burst Time: 2

Enter Details of Process[5]

Arrival Time: 4

Burst Time: 3

Enter Time Quantum: 2

Process ID	Burst Time	Turnaround Time	Waiting Time
Process[3]	1	3	2
Process[4]	2	4	2
Process[2]	3	11	8
Process[5]	3	9	6
Process[1]	5	14	9

Average Waiting Time: 5.400000

Avg Turnaround Time: 8.200000

#### **d) PRIORITY SCHEDULING**

In a Priority based Scheduling Algorithm in Operating Systems, every process is assigned a **Priority Number**. Based on this Priority Number, the processes are executed. This scheduling algorithm is normally very useful in real-time systems. The process having the highest priority (1) is executed first and then priority 2, 3 and so on.

**Priority Scheduling:** These are of two types. One is internal priority, second is external priority. The cpu is allocated to the process with the highest priority. Equal priority processes are scheduled in the FCFS order. Priorities are generally some fixed range of numbers such as 0 to 409. The low numbers represent high priority.

**Algorithm for Priority Scheduling:**

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: 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 turn around time

Step 6: For each process in the Ready Q calculate

Turn around time for Process(n)= Completion Time - Arrival Time

Waiting time for process(n)= Turnaround time - Burst Time

Step 7: Calculate

Average waiting time = Total waiting Time / Number of process

Average Turnaround time = Total Turnaround Time / Number of process

Step 8: Stop the process.

**Program to Simulate Priority CPU Scheduling Algorithm:**

```
#include<stdio.h>
int main()
{
    int i,j,n,bt[10],p[10],compt[10], wt[10],tat[10],temp1,temp2; float
    sumwt=0.0,sumtat=0.0,avgwt,avgtat;
    printf("Enter number of processes: ");
    scanf("%d",&n);
    printf("Enter the burst time of %d process\n", n);
    for(i=0;i<n;i++)
    scanf("%d",&bt[i]);
    printf("Enter the priority of %d process\n", n);
    for(i=0;i<n;i++)
    scanf("%d",&p[i]);
    for(i=0;i<n;i++)
    for(j=i+1;j<n;j++)
    if(p[i]>p[j])
    {
        temp1=bt[i];
        bt[i]=bt[j];
        bt[j]=temp1;
        temp2=p[i];
        p[i]=p[j];
    }
}
```

```

p[j]=temp2;
}
compt[0]=bt[0]; wt[0]=0; for(i=1;i<n;i++) compt[i]=bt[i]+compt[i-1]; for(i=0;i<n;i++)
{
tat[i]=compt[i]; wt[i]=tat[i]-bt[i]; sumtat+=tat[i]; sumwt+=wt[i];
}
avgwt=sumwt/n; avgtat=sumtat/n;
printf("-----\n");
printf("Bt\tCt\tTat\tWt\n");
printf("-----\n");
for(i=0;i<n;i++)
{
printf("%2d\t%2d\t%2d\t%2d\n",bt[i],compt[i],tat[i],wt[i]);
}
printf("-----\n");
printf(" Avgwt = %.2f\tAvgtat = %.2f\n",avgwt,avgtat);
printf("----- \n");
}

```

### Output:

Enter number of processes: 4  
Enter the burst time of 4 process  
4  
5  
7  
8  
Enter the priority of 4 process  
4  
3  
1  
2

```

-----
Bt   Ct   Tat   Wt
-----
7    7    7     0
8    15   15    7
5    20   20   15
4    24   24   20
-----

```

Avgwt = 10.50 Avgtat = 16.50

### WEEK 2



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

## **System Calls:**

When a computer is turned on, the program that gets executed first is called the *operating system*. It controls pretty much all activity in the computer. This includes who logs in, how disks are used, how memory is used, how the CPU is used, and how you talk with other computers. The operating system we use is called "Unix".

The way that programs talk to the operating system is via *system calls*. A system call looks like a procedure call, but it's different -- **it is a request to the operating system to perform some activity**.

### **System Calls for I/O:**

There are 5 basic system calls that Unix provides for file I/O.

1. int open(char \*path, int flags [ , int mode ] );
2. int close(int fd);
3. int read(int fd, char \*buf, int size);
4. int write(int fd, char \*buf, int size);
5. off\_t lseek(int fd, off\_t offset, int whence);

**Aim:** C program using open, read, write, close system calls

### **Theory:**

There are 5 basic system calls that Unix provides for file I/O.

1. **Create:** Used to Create a new empty file  
**Syntax:** int creat(char \*filename, mode\_t mode)  
filename : name of the file which you want to create  
mode : indicates permissions of new file.

2. **open:** Used to Open the file for reading, writing or both.

**Syntax:** int open(char \*path, int flags [ , int mode ] );

Path : path to file which you want to use  
flags : How you like to use

O\_RDONLY: read only, O\_WRONLY: write only, O\_RDWR: read and write,  
O\_CREAT: create file if it doesn't exist, O\_EXCL: prevent creation if it already exists

3. **close:** Tells the operating system you are done with a file descriptor and Close the file which pointed by fd.

**Syntax:** int close(int fd);  
fd : file descriptor

4. **read:** From the file indicated by the file descriptor fd, the read() function reads cnt bytes of input into the memory area indicated by buf. A successful read() updates the access time for the file.

**Syntax:** int read(int fd, char \*buf, int size);  
fd: file descriptor

buf: buffer to  
read data from  
cnt: length of  
buffer

5. **write:** Writes cnt bytes from buf to the file or socket associated with fd. cnt should not be greater than INT\_MAX (defined in the limits.h header file). If cnt is zero, write() simply returns 0 without attempting any other action.

**Syntax:** int write(int fd, char

\*buf, int size); fd: file descriptor

buf: buffer to

write data to

cnt: length of

buffer

\***File descriptor** is integer that uniquely identifies an open file of the process.

### **Programs on process System calls fork() wait() exit() and exec():**

**// Fork() System call**

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

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

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

```
int main()
```

```
{
```

```
    fork();
```

```
    printf("Hello, My process Id=%d\n", getpid());
```

```
    return 0;
```

```
}
```

**Output:**

```
aiml_dept@desktop-vmtw:~$ vi sample1.c
```

```
aiml_dept@desktop-vmtw:~$ gcc sample1.c
```

```
aiml_dept@desktop-vmtw:~$ ./a.out
```

```
Hello, My process Id=93
```

```
Hello, My process Id=94
```

**// exec() System call**

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

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

```
#include<stdlib.h>
```

```
int main(int argc, char*argv[])
```

```
{
```

```
    printf("PID of ex1.c=%d\n", getpid());
```

```
    char *args[]={ "hello", "vmtw", "AIML", NULL};
```

```
    execv("./ex2", args);
```

```
    return 0;
```

```
}
```

**Output:**

```
aiml_dept@desktop-vmtw:~$ vi sample2.c
aiml_dept@desktop-vmtw:~$ gcc sample2.c
aiml_dept@desktop-vmtw:~$ ./a.out
PID of ex1.c=101
```

#### **// exec() System call**

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main(int argc, char*argv[])
{
    printf("we are in ex2.c\n");
    printf("PID of ex2.c=%d\n", getpid());
    return 0;
}
```

#### **Output:**

```
aiml_dept@desktop-vmtw:~$ vi sample3.c
aiml_dept@desktop-vmtw:~$ gcc sample3.c
aiml_dept@desktop-vmtw:~$ ./a.out
we are in ex2.c
PID of ex2.c=108
```

#### **// wait() and exit() System call**

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<sys/wait.h>
#include<unistd.h>
int main (int argc, char**argv)
{
    int pid_t, pid;
    pid=fork();
    if(pid==0)
    {
        printf("it is the child process and pid is %d\n", getpid());
        int i=0;
        for(i=0;i<8;i++)
        {
            printf("%d\n",i);
        }
        exit(0);
    }
    else if(pid>0)
    {
        printf("it is the parent process and pid is %d\n", getpid());
        int status;
```

```

        wait(&status);
        printf("child is reaped");
    }
    else
    {
        printf("error in forking...\n");
        exit(EXIT_FAILURE);
    }
    return 0;
}

```

### **Output:**

```

aiml_dept@desktop-vmtw:~$ vi sample4.c
aiml_dept@desktop-vmtw:~$ gcc sample4.c
aiml_dept@desktop-vmtw:~$ ./a.out
it is the parent process and pid is 115
it is the child process and pid is 116
0
1
2
3
4
5
6
7
child is reaped.

```

### **Algorithm**

1. Start the program.
2. Open a file for O\_RDWR for R/W, O\_CREATE for creating a file, O\_TRUNC for truncate a file.
3. Using getchar(), read the character and stored in the string[] array.
4. The string [] array is write into a file close it.
5. Then the first is opened for read only mode and read the characters and displayed it and close the file.
6. Stop the program.

### **Program on system calls read write open close.**

```

#include<sys/stat.h>
#include<stdio.h>
#include<fcntl.h>
#include<sys/types.h>
#include<unistd.h>
#include<fcntl.h>
int main()
{

```

```

int n,i=0;
int f1,f2;
char c,string[100];
f1=open("data",O_RDWR|O_CREAT|O_TRUNC);
while((c=getchar())!='\n')
{
    string[i++]=c;
}
string[i]='\0';
write(f1,string,i);
close(f1);
f2=open("data",O_RDONLY);
read(f2,string,0);
printf("\n%s\n",string);
close(f2);
return 0;
}

```

**Output:**

```

aiml_dept@desktop-vmtw:~$ vi sample5.c
aiml_dept@desktop-vmtw:~$ gcc sample5.c
aiml_dept@desktop-vmtw:~$ ./a.out
hello welcome to vmtw

```

```

hello welcome to vmtw

```

**b) Aim:** C program using lseek

**Theory:**

lseek is a 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.

**Syntax :** off\_t lseek(int fildes, off\_t offset, int whence);

int fildes : The file descriptor of the pointer that is going to be moved.  
off\_t offset : The offset of the pointer (measured in bytes).

int whence : Legal values for this variable are provided at the end which are SEEK\_SET (Offset is to be measured in absolute terms), SEEK\_CUR (Offset is to be measured relative to the current location of the pointer), SEEK\_END (Offset is to be measured relative to the end of the file)

**Algorithm:**

1. Start the program
2. Open a file in read mode
3. Read the contents of the file
4. Use lseek to change the position of pointer in the read process
5. Stop

**Program:**

```
#include<stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
int main()
{
    int file=0;
    if((file=open("testfile.txt",O_RDONLY)) < -1)
        return 1;
    char buffer[19];
    if(read(file,buffer,19) != 19) return 1;
    printf("%s\n",buffer);
    if(lseek(file,10,SEEK_SET) < 0)
        return 1;
    if(read(file,buffer,19) != 19)
        return 1;
    printf("%s\n",buffer);
    return 0;
}
```

### Output:

aiml\_dept@desktop-vmtw:~\$ vi testfile.txt

aiml\_dept@desktop-vmtw:~\$ cat testfile.txt


A system call is a mechanism that provides the interface between a process and the operating system.

It is a programmatic method in which a computer program requests a service from the kernel of the OS.

aiml\_dept@desktop-vmtw:~\$ vi sample6.c

aiml\_dept@desktop-vmtw:~\$ gcc sample6.c

aiml\_dept@desktop-vmtw:~\$ ./a.out

A system call is a 

all is a mechanism 

```
[188r1a0501@localhost ~]$ vi testfile.txt
[188r1a0501@localhost ~]$ cat testfile.txt
sdfghjkl;mnbbvrtuijnnb
ggtyujjg

[188r1a0501@localhost ~]$ gcc w2c.c
[188r1a0501@localhost ~]$ ./a.out
sdfghjkl;mnbbvrtyu
mnbbvrtuijnnb
ggty
```

c) **Aim:** C program using opendir(), closedir(), readdir()

**Theory:**

The following are the various operations using directories

1. Creating directories.

**Syntax :** int mkdir(const char \*pathname, mode\_t mode);

2. The 'pathname' argument is used for the name of the directory.

3. Opening directories

**Syntax :** DIR \*opendir(const char \*name);

4. Reading directories.

**Syntax:** struct dirent \*readdir(DIR \*dirp);

5. Removing directories.

**Syntax:** int rmdir(const char \*pathname);

6. Closing the directory.

**Syntax:** int closedir(DIR \*dirp);

7. Getting the current working directory.

**Syntax:** char \*getcwd(char \*buf, size\_t size);

**Algorithm:**

1. Start the program
2. Print a menu to choose the different directory operations
3. To create and remove a directory ask the user for name and create and remove the same respectively.
4. To open a directory check whether directory exists or not. If yes open the directory .If it does not exist print an error message.
5. Finally close the opened directory.
6. Stop

**Program:**

```
#include<stdio.h>
#include<unistd.h>
#include<fcntl.h>
#include<dirent.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<stdlib.h>
int main(int argc, char **argv)
{
    struct stat buf;
    int exists;
    DIR *d;
    struct dirent *de;
    d = opendir(".");
    if (d == NULL)
    {
        fprintf(stderr, "Couldn't open \".\"\\n");
        exit(1);
    }
```



```

}
for (de = readdir(d); de != NULL;
de = readdir(d))
{
    exists = stat(de->d_name, &buf);
    if (exists < 0)
    {
        fprintf(stderr, "%s not found\n", de->d_name);
    }
    else
    {
        printf("%s %ld\n", de->d_name, buf.st_size);
    }
}
closedir(d);
return 0;
}

```

### **Output:**

```

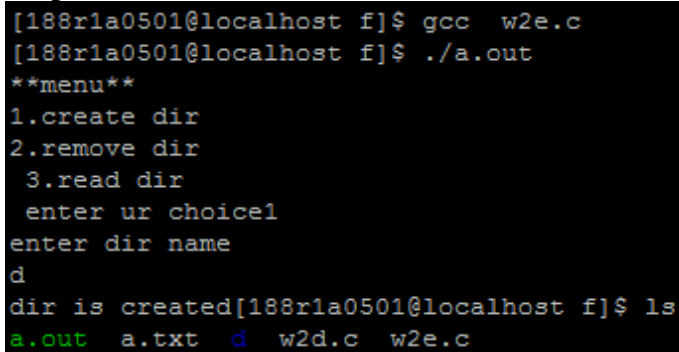
aiml_dept@desktop-vmtw:~$ vi sample7.c
aiml_dept@desktop-vmtw:~$ gcc sample7.c
aiml_dept@desktop-vmtw:~$ ./a.out
data 3
a1 0
a2 0
.sudo_as_admin_successful 0
sample5.c 444
.cache 4096
.bash_history 295
.config 4096
testfile.txt 204
.profile 807
sample3.c 174
.bash_logout 220
b2 0
.viminfo 8706
sample2.c 213
.bashrc 3771
sample1.c 143
sample4.c 541
b1 0
sample6.c 385
a.out 8744
.. 4096
. 4096
sample7.c 639

```

## Program:

```
#include<stdio.h>
#include<unistd.h>
#include<fcntl.h>
#include<dirent.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<stdlib.h>
int main(int argc, char **argv)
{
char d[10]; int c,op; DIR *e; struct dirent *sd;
printf("**menu**\n1.create dir\n2.remove dir\n 3.read dir\n enter ur choice"); scanf("%d",&op);
switch(op)
{
case 1: printf("enter dir name\n"); scanf("%s",&d); c=mkdir(d,777);
if(c==1)
printf("dir is not created"); else
printf("dir is created"); break;
case 2: printf("enter dir name\n"); scanf("%s",&d); c=rmdir(d);
if(c==1)
printf("dir is not removed"); else
printf("dir is removed"); break;
case 3: printf("enter dir name to open"); scanf("%s",&d);
e=opendir(d); if(e==NULL)
printf("dir does not exist"); else
{
printf("dir exist\n"); while((sd=readdir(e))!=NULL)
printf("%s\t",sd->d_name);
}
closedir(e); break;
}
}
```

## Output:



```
[188r1a0501@localhost f]$ gcc w2e.c
[188r1a0501@localhost f]$ ./a.out
**menu**
1.create dir
2.remove dir
 3.read dir
  enter ur choice1
enter dir name
d
dir is created[188r1a0501@localhost f]$ ls
a.out  a.txt  d    w2d.c  w2e.c
```

### **WEEK -3**

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

##### **a) Aim**

**Write a C program to simulate the Bankers Algorithm for Deadlock Avoidance.**

##### **Data structures**

1. n- Number of process, m-number of resource types.
2. Available: Available[j]=k, k – instance of resource type R<sub>j</sub> is available.
3. Max: If max [i, j]=k, P<sub>i</sub> may request at most k instances resource R<sub>j</sub>.
4. Allocation: If Allocation [i, j]=k, P<sub>i</sub> allocated to k instances of resource R<sub>j</sub>
5. Need: If Need[I, j]=k, P<sub>i</sub> may need k more instances of resource type R<sub>j</sub>,
6. Need [I, j] =Max [I, j]-Allocation [I, j];

##### **Safety Algorithm**

1. Work and Finish be the vector of length m and n respectively, Work=Available and Finish[i] =False.
2. Find an i such that both
3. Finish[i] =False
4. Need<=Work
5. If no such I exist go to step 4.
6. work=work+Allocation, Finish[i] =True;
7. If Finish [1] =True for all I, then the system is in safe state.

##### **Resource request algorithm**

1. Let Request i be request vector for the process P<sub>i</sub>, If request i=[j]=k, then process P<sub>i</sub> wants k instances of resource type R<sub>j</sub>.
2. If Request<=Need I go to step 2. Otherwise raise an error condition. 3. If Request<=Available go to step
3. Otherwise P<sub>i</sub> must since the resources are available.
4. Have the system pretend to have allocated the requested resources to process P<sub>i</sub> by modifying the state as follows;
5. Available=Available-Request i;
6. Allocation i =Allocation+Request i;
7. Need i=Need i-Request I;

If the resulting resource allocation state is safe, the transaction is completed and process P<sub>i</sub> is allocated its resources. However, if the state is unsafe, the P<sub>i</sub> must wait for Request i and the old resource-allocation state is restore.

##### **Algorithm:**

1. Start the program.
2. Get the values of resources and processes.
3. Get the avail value.
4. After allocation find the need value.
5. Check whether it is possible to allocate.
6. If it is possible then the system is in safe state.
7. Else system is not in safety state.
8. If the new request comes then check that the system is in safety.

9. Or not if we allow the request.
10. Stop the program.

```
#include<stdio.h>
int main ()
{
    int allocated[15][15], max[15][15], need[15][15], avail[15], tres[15],
    work[15], flag[15];
    int pno, rno, i, j, prc, count, t, total;
    count = 0;
    //clrscr ();
    printf ("\n Enter number of process:");
    scanf ("%d", &pno);
    printf ("\n Enter number of resources:");
    scanf ("%d", &rno);
    for (i = 1; i <= pno; i++)
    {
        flag[i] = 0;
    }
    printf ("\n Enter total numbers of each resources:");
    for (i = 1; i <= rno; i++)
        scanf ("%d", &tres[i]);
    printf ("\n Enter Max resources for each process:");
    for (i = 1; i <= pno; i++)
    {
        printf ("\n for process %d:", i);
        for (j = 1; j <= rno; j++)
            scanf ("%d", &max[i][j]);
    }
    printf ("\n Enter allocated resources for each process:");
    for (i = 1; i <= pno; i++)
    {
        printf ("\n for process %d:", i);
        for (j = 1; j <= rno; j++)
            scanf ("%d", &allocated[i][j]);
    }
    printf ("\n available resources:\n");
    for (j = 1; j <= rno; j++)
    {
        avail[j] = 0;
        total = 0;
        for (i = 1; i <= pno; i++)
        {
            total += allocated[i][j];
        }
        avail[j] = tres[j] - total;
    }
}
```

```

work[j] = avail[j];
printf (" %d \t", work[j]);
}
do
{
for (i = 1; i <= pno; i++)
{
for (j = 1; j <= rno; j++)
{
need[i][j] = max[i][j] - allocated[i][j];
}
}
}
printf ("\n Allocated matrix Max need");
for (i = 1; i <= pno; i++)
{
printf ("\n");
for (j = 1; j <= rno; j++)
{
printf ("%4d", allocated[i][j]);
}
printf ("|");
for (j = 1; j <= rno; j++)
{
printf ("%4d", max[i][j]);
}
printf ("|");
for (j = 1; j <= rno; j++)
{
printf ("%4d", need[i][j]);
}
}
}
prc = 0;
for (i = 1; i <= pno; i++)
{
if (flag[i] == 0)
{
prc = i;
for (j = 1; j <= rno; j++)
{
if (work[j] < need[i][j])
{
prc = 0;
break;
}
}
}
}
}

```

```

if (prc != 0)
    break;
}
if (prc != 0)
{
    printf ("\n Process %d completed", i);
    count++;
    printf ("\n Available matrix:");
    for (j = 1; j <= rno; j++)
    {
        work[j] += allocated[prc][j];
        allocated[prc][j] = 0;
        max[prc][j] = 0;
        flag[prc] = 1;
        printf (" %d", work[j]);
    }
}
}
while (count != pno && prc != 0);
if (count == pno)
    printf ("\nThe system is in a safe state!!");
else
    printf ("\nThe system is in an unsafe state!!");
return 0;
}

```

### Output:

Enter number of process:5

Enter number of resources:3

Enter total numbers of each resources:10

5 7

Enter Max resources 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 allocated resources for each process:

for process 1:0     1     0

for process 2:2     0     0

for process 3:3     0     2

for process 4:2     1     1

for process 5:0     0     2

available resources:

3     3     2

Allocated matrix Max need

0	1	0	7	5	3	7	4	3
2	0	0	3	2	2	1	2	2
3	0	2	9	0	2	6	0	0
2	1	1	2	2	2	0	1	1
0	0	2	4	3	3	4	3	1

Process 2 completed

Available matrix: 5 3 2

Allocated matrix Max need

0	1	0	7	5	3	7	4	3
0	0	0	0	0	0	0	0	0
3	0	2	9	0	2	6	0	0
2	1	1	2	2	2	0	1	1
0	0	2	4	3	3	4	3	1

Process 4 completed

Available matrix: 7 4 3

Allocated matrix Max need

0	1	0	7	5	3	7	4	3
0	0	0	0	0	0	0	0	0
3	0	2	9	0	2	6	0	0
0	0	0	0	0	0	0	0	0
0	0	2	4	3	3	4	3	1

Process 1 completed

Available matrix: 7 5 3

Allocated matrix Max need

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
3	0	2	9	0	2	6	0	0
0	0	0	0	0	0	0	0	0
0	0	2	4	3	3	4	3	1

Process 3 completed

Available matrix: 10 5 5

Allocated matrix Max need

0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	2	4	3	3	4	3	1

Process 5 completed

Available matrix: 10 5 7

The system is in a safe state!!

## Write a C program to simulate Bankers Algorithm for Deadlock Prevention?

### Algorithm:

1. Start
2. Attacking Mutex condition : never grant exclusive access. but this may not be possible for several resources.
3. Attacking preemption: not something you want to do.
4. Attacking hold and wait condition : make a process hold at the most 1 resource at a time. make all the requests at the beginning. All or nothing policy. If you feel, retry. eg. 2- phase locking 34
5. Attacking circular wait: Order all the resources. Make sure that the requests are issued in the correct order so that there are no cycles present in the resource graph. Resources numbered 1 ... n. Resources can be requested only in increasing order. ie. you cannot request a resource whose no is less than any you may be holding.
6. Stop

```
#include<stdio.h>
#include<conio.h>
int max[10][10],alloc[10][10],need[10][10],avail[10],i,j,p,r,finish[10]={0},flag=0;
void main( )
{

printf("\n\nSIMULATION OF DEADLOCK PREVENTION");
printf("Enter no. of processes, resources");
scanf("%d%d",&p,&r);printf("Enter allocation matrix");
for(i=0;i<p;i++)
for(j=0;j<r;j++)
scanf("%d",&alloc[i][j]);
printf("enter max matrix");
for(i=0;i<p;i++) /*reading the maximum matrix and availale matrix*/
for(j=0;j<r;j++)
scanf("%d",&max[i][j]);
printf("enter available matrix");
for(i=0;i<r;i++)
scanf("%d",&avail[i]);
for(i=0;i<p;i++)
```



```

for(j=0;j<r;j++)
need[i][j]=max[i][j]-alloc[i][j];
fun(); /*calling function*/
if(flag==0)
{
    if(finish[i]!=1)
    {
printf("\n\n Failing :Mutual exclusion");
for(j=0;j<r;j++)
{ /*checking for mutual exclusion*/
if(avail[j]<need[i][j])
avail[j]=need[i][j];
}fun();
printf("\n By allocating required resources to process %d dead lock is prevented ",i);
printf("\n\n lack of preemption");
for(j=0;j<r;j++)
{
    if(avail[j]<need[i][j])
avail[j]=need[i][j];
alloc[i][j]=0;
}
fun( );
printf("\n\n daed lock is prevented by allocating needed resources");
printf(" \n \n failing:Hold and Wait condition ");
for(j=0;j<r;j++)
{ /*checking hold and wait condition*/
if(avail[j]<need[i][j])
avail[j]=need[i][j];
}
fun( );
printf("\n AVOIDING ANY ONE OF THE CONDITION, U CAN PREVENT DEADLOCK");
}
}
getch( );
}
fun()
{
while(1)
{
for(flag=0,i=0;i<p;i++)
{
if(finish[i]==0)
{
for(j=0;j<r;j++)
{
if(need[i][j]<=avail[j])

```

```

continue;
else
break;
}
if(j==r)
{
for(j=0;j<r;j++)
avail[j]+=alloc[i][j];
flag=1;
finish[i]=1;
}
}
}
if(flag==0)
break;
}
}

```

**Output:**

### **SIMULATION OF DEADLOCK PREVENTION**

**Enter no. of processes, resources 3 2**

**Enter allocation matrix 2 4 5**

**3 4 5**

**enter max matrix 4 3 4**

**5 6 1**

**enter available matrix 1 5**

**Failing :Mutual exclusion**

**By allocating required resources to process 3 dead lock is prevented**

**lack of preemption**

**dead lock is prevented by allocating needed resources**

**Failing:Hold and Wait condition**

**AVOIDING ANY ONE OF THE CONDITION, U CAN PREVENT DEADLOCK**

## WEEK-4

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

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

### Algorithm:

1. The Semaphore mutex, full & empty are initialized.
2. In the case of producer process
3. Produce an item in to temporary variable. If there is empty space in the buffer check the mutex value for enter into the critical section. If the mutex value is 0, allow the producer to add value in the temporary variable to the buffer.
4. 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.
5. Print the result

```
#include<stdio.h>
#include<stdlib.h>
int mutex = 1, full = 0, empty = 3, x = 0;
int main () {
    int n;
    void producer ();
    void consumer ();
    int wait (int);
    int signal (int);
    printf (" \n1.Producer\n2.Consumer\n3.Exit");
    while (1)
    {
        printf ("\nEnter your choice:");
        scanf ("%d", &n);
        switch (n)
        {
        case 1:
            if ((mutex == 1) && (empty != 0))
                producer ();
            else
                printf ("Buffer is full!!");
            break;
```

```

case 2:
if ((mutex == 1) && (full != 0))
consumer ();
else
printf ("Buffer is empty!!");
break;
case 3:
exit (0);
break; }

}
return 0;
}
int wait (int s)
{
return (--s);
}
int signal (int s)
{
return (++s);
}
void producer ()
{
mutex = wait (mutex);
full = signal (full);
empty = wait (empty);
x++;
printf ("\nProducer produces the item %d", x);
mutex = signal (mutex);
}
void consumer ()
{
mutex = wait (mutex);
full = wait (full);
empty = signal (empty);
printf ("\nConsumer consumes item %d", x);
x--;
mutex = signal (mutex);
}

```

### **Output:**

**1.Producer**

**2.Consumer**

**3.Exit**

**Enter your choice:1**

**Producer produces the item 1**  
**Enter your choice:1**

**Producer produces the item 2**  
**Enter your choice:1**

**Producer produces the item 3**  
**Enter your choice:2**

**Consumer consumes item 3**  
**Enter your choice:2**

**Consumer consumes item 2**  
**Enter your choice:2**

**Consumer consumes item 1**  
**Enter your choice:2**  
**Buffer is empty!!**  
**Enter your choice:**

```
[188r1a0501@localhost ~]$ vi pc.c
[188r1a0501@localhost ~]$ gcc pc.c
[188r1a0501@localhost ~]$ ./a.out
```

```
1.Producer
2.Consumer
3.Exit
Enter your choice:1

Producer produces the item 1
Enter your choice:1

Producer produces the item 2
Enter your choice:1

Producer produces the item 3
Enter your choice:2

Consumer consumes item 3
Enter your choice:2

Consumer consumes item 2
Enter your choice:2

Consumer consumes item 1
Enter your choice:2
Buffer is empty!!
Enter your choice:3
[188r1a0501@localhost ~]$ █
```

## Week: 5

**Write C programs to illustrate the following IPC mechanisms**

**Aim: Write C programs to illustrate the following IPC mechanisms**

### **ALGORITHM:**

1. Start the program.
2. Declare the variables.
3. Read the choice.
4. Create a piping processing using IPC.
5. Assign the variable lengths
6. "strcpy" the message lengths.
7. To join the operation using IPC .
8. Stop the program.

**Program : ( PIPE PROCESSING)**

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define MSG_LEN 64
int main()
{
    int result;
    int fd[2];
    char message[MSG_LEN];
    char recvd_msg[MSG_LEN];
    result = pipe(fd);
    //Creating a pipe//fd[0] is for reading and fd[1] is for writing
    if (result < 0)
    {
        perror("pipe ");
        exit(1);
    }
    strncpy(message,"Linux World!! ",MSG_LEN);
    result=write(fd[1],message,strlen(message));
    if (result < 0)
    {
        perror("write");
        exit(2);
    }
    strncpy(message,"Understanding ",MSG_LEN);
    result=write(fd[1],message,strlen(message));
    if(result < 0)
    {
        perror("write");
        exit(2);
    }
    strncpy(message,"Concepts of ",MSG_LEN);
    result=write(fd[1],message,strlen(message));
    if (result < 0)
    {
        perror("write");
        exit(2);
    }
    strncpy(message,"Piping ", MSG_LEN);
    result=write(fd[1],message,strlen(message));
    if (result < 0){
        perror("write");
        exit(2);
    }
}
```

```

result=read(fd[0],recvd_msg,MSG_LEN);
if (result < 0)
{
perror("read");
exit(3);
}
printf("%s\n",recvd_msg);
return 0;
}

```

Output:

```

ithod@DESKTOP-0SCES8B:~$ vi sample.c
ithod@DESKTOP-0SCES8B:~$ gcc sample.c
ithod@DESKTOP-0SCES8B:~$ ./a.out
Linux World!! Understanding Concepts of Piping
ithod@DESKTOP-0SCES8B:~$ ithod@DESKTOP-0SCES8B:~$ vi sample.c
ithod@DESKTOP-0SCES8B:~$: command not found
ithod@DESKTOP-0SCES8B:~$ ithod@DESKTOP-0SCES8B:~$ gcc sample.c
ithod@DESKTOP-0SCES8B:~$: command not found
ithod@DESKTOP-0SCES8B:~$ ithod@DESKTOP-0SCES8B:~$ ./a.out
ithod@DESKTOP-0SCES8B:~$: command not found
ithod@DESKTOP-0SCES8B:~$ Linux World!! Understanding Concepts of Pipingithod@DESKTOP-0SCES8B:~$ vi sample.c
Linux Worldithod@DESKTOP-0SCES8B:~$ ./a.out Understanding Concepts of Pipingithod@DESKTOP-0SCES8B:~$ vi sample.c
Linux: command not found
ithod@DESKTOP-0SCES8B:~$ ithod@DESKTOP-0SCES8B:~$ gcc sample.c
ithod@DESKTOP-0SCES8B:~$: command not found
ithod@DESKTOP-0SCES8B:~$ ithod@DESKTOP-0SCES8B:~$ ./a.out
ithod@DESKTOP-0SCES8B:~$: command not found
ithod@DESKTOP-0SCES8B:~$ Linux World!! Understanding Concepts of Pipingithod@DESKTOP-0SCES8B:~$ vi sample.c
Linux Worldithod@DESKTOP-0SCES8B:~$ ./a.out Understanding Concepts of Pipingithod@DESKTOP-0SCES8B:~$ vi sample.c
Linux: command not found
ithod@DESKTOP-0SCES8B:~$ ithod@DESKTOP-0SCES8B:~$ gcc sample.c
ithod@DESKTOP-0SCES8B:~$: command not found
ithod@DESKTOP-0SCES8B:~$ ithod@DESKTOP-0SCES8B:~$ ./a.out
ithod@DESKTOP-0SCES8B:~$: command not found
ithod@DESKTOP-0SCES8B:~$ Linux World!! Understanding Concepts of Piping
Linux Worldithod@DESKTOP-0SCES8B:~$ ./a.out Understanding Concepts of Piping

```

#### a) FIFO

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>
#include <linux/stat.h>
#define FIFO_FILE "MYFIFO"
int main(void)
{
FILE *fp;
char readbuf[80];
/* Create the FIFO if it does not exist */
umask(0);
mknod(FIFO_FILE, S_IFIFO|0666, 0);
while(1)

```



```

{
fp = fopen(FIFO_FILE, "r");
fgets(readbuf, 80, fp);
printf("Received string: %s\n", readbuf);
fclose(fp);
}
return(0);
}

```

### Output:

```

ithod@DESKTOP-0SCES8B:~$ vi sample1.c
ithod@DESKTOP-0SCES8B:~$ gcc sample1.c
ithod@DESKTOP-0SCES8B:~$ ./a.out
hello
FIFO
gfjdsjdfkdsghjghfg

```

```

#include <stdio.h>
#include <stdlib.h>
#define FIFO_FILE "MYFIFO"
int main(int argc, char *argv[])
{
FILE *fp;
if ( argc != 2 ) {
printf("USAGE: fifoclient [string]\n");
exit(1);
}
if((fp = fopen(FIFO_FILE, "w")) == NULL) {
perror("fopen");
exit(1);
}
fputs(argv[1], fp);
fclose(fp);
return(0);
}

```

### Output:

```

ithod@DESKTOP-0SCES8B:~$ vi sample2.c
ithod@DESKTOP-0SCES8B:~$ gcc sample2.c
ithod@DESKTOP-0SCES8B:~$ ./a.out
USAGE: fifoclient [string]
ithod@DESKTOP-0SCES8B:~$

```

## C Program for Message Queue (Writer Process)

```

#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
// structure for message queue
struct msg_buffer {
    long msg_type;
    char msg_text[100];
} message;
int main()
{
    key_t key;
    int msgid;
    // ftok to generate unique key
    key = ftok("progfile", 65);
    // msgget creates a message queue
    // and returns identifier
    msgid = msgget(key, 0666 | IPC_CREAT);
    message.msg_type = 1;
    printf("Write Data : ");
    gets(message.msg_text);

    // msgsnd to send message
    msgsnd(msgid, &message, sizeof(message), 0);

    // display the message
    printf("Data send is : %s \n", message.msg_text);

    return 0;
}

```

### Output:

```

ithod@DESKTOP-0SCES8B:~$ vi sample3.c
ithod@DESKTOP-0SCES8B:~$ vi write.c
ithod@DESKTOP-0SCES8B:~$ gcc write.c
write.c: In function 'main':
write.c:23:15: warning: implicit declaration of function 'gets'; did you mean 'fgets'? [-Wimplicit-function-declaration]
    gets(message.msg_text);
    ^~~~
    fgets
/tmp/cc7AZsHP.o: In function `main':
write.c:(.text+0x57): warning: the `gets' function is dangerous and should not be used.
ithod@DESKTOP-0SCES8B:~$ ./a.out
Write Data : hello world
Data send is : hello world

```

### C Program for Message Queue (Reader Process)

```

#include<stdio.h>
#include<sys/ipc.h>
#include<sys/msg.h>
// structure for message queue
struct msg_buffer {
    long msg_type;
    char msg_text[100];
} message;
int main()
{
    key_t key;
    int msgid;
    // ftok to generate unique key
    key = ftok("progfile", 65);
    // msgget creates a message queue and returns identifier
    msgid = msgget(key, 0666 | IPC_CREAT);
    // msgrcv to receive message
    msgrcv(msgid, &message, sizeof(message), 1, 0);
    // display the message
    printf("Data Received is : %s \n",
        message.msg_text);
    // to destroy the message queue
    msgctl(msgid, IPC_RMID, NULL);
    return 0;
}

```

Output:

```

ithod@DESKTOP-0SCES8B:~$ vi reader1.c
ithod@DESKTOP-0SCES8B:~$ gcc reader1.c
ithod@DESKTOP-0SCES8B:~$ ./a.out
Data Received is :

```

## Week: 6

**Aim: Write C programs to simulate the following memory management techniques**

### a) Paging

AIM: To write a C program to implement memory management using paging technique.

ALGORITHM:

Step1 : Start the program.

Step2 : Read the base address, page size, number of pages and memory unit.

Step3 : If the memory limit is less than the base address display the memory limit is less than limit.

Step4 : Create the page table with the number of pages and page address.

Step5 : Read the page number and displacement value.

Step6 : If the page number and displacement value is valid, add the displacement value with the address corresponding to the page number and display the result.

Step7 : Display the page is not found or displacement should be less than page size.

Step8 : Stop the program.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int ms, ps, nop, np, rempages, i, j, x, y, pa, offset; int s[10], fno[10][20];
printf("\nEnter the memory size -- ");
scanf("%d",&ms);
printf("\nEnter the page size -- ");
scanf("%d",&ps);
nop = ms/ps;
printf("\nThe no. of pages available in memory are -- %d ",nop);
printf("\nEnter number of processes -- ");
scanf("%d",&np);
rempages = nop; for(i=1;i<=np;i++)
{
printf("\nEnter no. of pages required for p[%d]-- ",i);
scanf("%d",&s[i]);
if(s[i] > rempages)
{
printf("\nMemory is Full");
break;
}
rempages = rempages - s[i];
printf("\nEnter pagetable for p[%d] --- ",i);
for(j=0;j<s[i];j++)
scanf("%d",&fno[i][j]);
}
printf("\nEnter Logical Address to find Physical Address ");
printf("\nEnter process no. and pagenumber and offset -- ");
```

```

scanf("%d %d %d",&x,&y, &offset);
if(x>np || y>=s[i] || offset>=ps)
printf("\nInvalid Process or Page Number or offset");
else
{
pa=fno[x][y]*ps+offset;
printf("\nThe Physical Address is -- %d",pa);
}
getch();
}

```

### Output

Enter the memory size -- 1000

Enter the page size -- 200

The no. of pages available in memory are -- 5

Enter number of processes -- 2

Enter no. of pages required for p[1]-- 20

Memory is Full

Enter Logical Address to find Physical Address

Enter process no. and pagenumber and offset -- 1

2

6

The Physical Address is -- 906999302

### b) Segmentation

**Aim: To write a C program to implement memory management using segmentation**

#### Algorithm:

Step1 : Start the program.

Step2 : Read the base address, number of segments, size of each segment, memory limit.

Step3 : If memory address is less than the base address display "invalid memory limit".

Step4 : Create the segment table with the segment number and segment address and display it.

Step5 : Read the segment number and displacement.

Step6 : If the segment number and displacement is valid compute the real address and display the same.

Step7 : Stop the program.

#### Program:

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

```

#include <stdlib.h>
struct list
{
int seg;
int base;
int limit;
struct list *next;
} *p;
void insert(struct list *q,int base,int limit,int seg)
{
if(p==NULL)
{
p=malloc(sizeof(struct list));
p->limit=limit;
p->base=base;
p->seg=seg;
p->next=NULL;
}
else
{
while(q->next!=NULL)
{
q=q->next;
printf("yes");
}
q->next=malloc(sizeof(struct list));
q->next->limit=limit;
q->next->base=base;
q->next->seg=seg;
q->next->next=NULL;
}

}
int find(struct list *q,int seg)
{
while(q->seg!=seg)
{
q=q->next;
}
return q->limit;
}

int search(struct list *q,int seg)
{
while(q->seg!=seg)
{

```

```

q=q->next;
}
return q->base;
}

void main()
{
p=NULL;
int seg,offset,limit,base,c,s,physical;
printf("Enter segment table/n");
printf("Enter -1 as segment value for termination\n");
do{
printf("Enter segment number");
scanf("%d",&seg);
if(seg!=-1)
{
printf("Enter base value:");
scanf("%d",&base);
printf("Enter value for limit:");
scanf("%d",&limit);
insert(p,base,limit,seg);
}
}
while(seg!=-1);
printf("Enter offset:");
scanf("%d",&offset);
printf("Enter bsegmentation number:");
scanf("%d",&seg);
c=find(p,seg);
s=search(p,seg);
if(offset<c)
{
physical=s+offset;
printf("Address in physical memory %d\n",physical);
}
else
{
printf("error");
}
}

```

### **OUTPUT:**

Enter segment table

Enter

-1 as segmentation value for termination

Enter segment number:1  
Enter base value:2000  
Enter value for limit:100  
Enter segment number:2  
Enter base value:2500  
Enter value for limit:100  
Enter segmentation number:  
-  
1  
Enter offset:90  
Enter segment number:2  
Address in physical memory 2590  
Enter segment table  
Enter  
-1 as segmentation value for termination  
Enter segment number:1  
Enter base value:2000  
Enter value for limit:100  
Enter segment number:2  
Enter base value:2500  
Enter value for limit:100  
Enter segmentation number:  
-  
1  
Enter offset:90  
Enter segment number:1  
Address in physical memory 2



## Week: 7

**Aim:.** Write C programs to simulate Page replacement policies a) FCFS b) LRU c) Optimal.

### a) FCFS Page replacement

The operating system replaces an existing page from the main memory by bringing a new page from the secondary memory.

In such situations, the FIFO method is used, which is also refers to the First in First Out concept. This is the simplest page replacement method in which the operating system maintains all the pages in a queue. Oldest pages are kept in the front, while the newest is kept at the end. On a page fault, these pages from the front are removed first, and the pages in demand are added.

Algorithm for FIFO Page Replacement

- Step 1. Start to traverse the pages.
- Step 2. If the memory holds fewer pages, then the capacity else goes to step 5.
- Step 3. Push pages in the queue one at a time until the queue reaches its maximum capacity or all page requests are fulfilled.
- Step 4. If the current page is present in the memory, do nothing.
- Step 5. Else, pop the topmost page from the queue as it was inserted first.
- Step 6. Replace the topmost page with the current page from the string.
- Step 7. Increment the page faults.
- Step 8. Stop

// C program for FIFO page replacement algorithm

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

```
int main()
```

```
{
```

```
    int incomingStream[] = {4, 1, 2, 4, 5};
```

```
    int pageFaults = 0;
```

```
    int frames = 3;
```

```
    int m, n, s, pages;
```

```
    pages = sizeof(incomingStream)/sizeof(incomingStream[0]);
```

```
    printf("Incoming \t Frame 1 \t Frame 2 \t Frame 3");
```

```
    int temp[frames];
```

```
    for(m = 0; m < frames; m++)
```

```
    {
```

```
        temp[m] = -1;
```

```
    }
```

```
    for(m = 0; m < pages; m++)
```

```
    {
```

```
        s = 0;
```

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

```

    {
        if(incomingStream[m] == temp[n])
        {
            s++;
            pageFaults--;
        }
    }
    pageFaults++;

    if((pageFaults <= frames) && (s == 0))
    {
        temp[m] = incomingStream[m];
    }
    else if(s == 0)
    {
        temp[(pageFaults - 1) % frames] = incomingStream[m];
    }

    printf("\n");
    printf("%d\t\t\t",incomingStream[m]);
    for(n = 0; n < frames; n++)
    {
        if(temp[n] != -1)
            printf(" %d\t\t\t", temp[n]);
        else
            printf(" - \t\t\t");
    }
}

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

```

### **Output –**

Incoming Frame 1 Frame 2 Frame 3

```

4      4      -      -
1      4      1      -
2      4      1      2
4      4      1      2
5      5      1      2

```

Total Page Faults: 4

## **b) LRU Page replacement**

The Least Recently Used (LRU) algorithm is a popular page replacement strategy that operates on the principle of locality of reference. It assumes that if a page is referenced recently, it is likely to be referenced again in the near future.

To implement the LRU algorithm, the system maintains a list or a queue of pages in the main memory. When a page needs to be replaced, the algorithm selects the page that has the earliest access time, indicating that it has been the least recently used. This selected page is then swapped out with the new page.

### **Implementing LRU Algorithm in Operating Systems**

Operating systems often implement the LRU algorithm to manage virtual memory and disk caching. The LRU algorithm can be implemented using various data structures such as linked lists, hash tables, or priority queues.

These data structures enable efficient tracking of page access times and facilitate the selection of the least recently used page for replacement.

### **Pseudocode of LRU Algorithm in OS**

LRU Algorithm:

1. Initialize an empty cache of size N (number of frames) and an empty queue (used to keep track of the order of page references).
2. While processing page references:
  - a. Read the next page reference.
  - b. If the page is present in the cache (cache hit), move the corresponding entry to the front of the queue.
  - c. If the page is not present in the cache (cache miss):
    - i. If the cache is full (all frames are occupied):
      1. Remove the least recently used page (the page at the end of the queue).
      2. Add the new page to the cache and insert it at the front of the queue.
    - ii. If the cache has an empty frame:
      1. Add the new page to the cache and insert it at the front of the queue.
  - d. Repeat steps (a) to (c) until all page references are processed.
3. At the end of processing page references, the cache will contain the most frequently used pages.

### **PROGRAM:**

```
#include<stdio.h>
main()
{
int q[20],p[50],c=0,c1,d,f,i,j,k=0,n,r,t,b[20],c2[20];
printf("Enter no of pages:");
scanf("%d",&n);
printf("Enter the reference string:");
for(i=0;i<n;i++)
scanf("%d",&p[i]);
printf("Enter no of frames:");
scanf("%d",&f);
```

```

q[k]=p[k];
printf("\n\t%d\n",q[k]);
c++;
k++;
for(i=1;i<n;i++)
{
    c1=0;
    for(j=0;j<f;j++)
    {
        if(p[i]!=q[j])
            c1++;
    }
    if(c1==f)
    {
        c++;
        if(k<f)
        {
            q[k]=p[i];
            k++;
            for(j=0;j<k;j++)
                printf("\t%d",q[j]);
            printf("\n");
        }
        else
        {
            for(r=0;r<f;r++)
            {
                c2[r]=0;
                for(j=i-1;j<n;j--)
                {
                    if(q[r]!=p[j])
                        c2[r]++;
                    else
                        break;
                }
            }
            for(r=0;r<f;r++)
                b[r]=c2[r];
            for(r=0;r<f;r++)
            {
                for(j=r;j<f;j++)
                {
                    if(b[r]<b[j])
                    {
                        t=b[r];
                        b[r]=b[j];

```

```

        b[j]=t;
    }
}
for(r=0;r<f;r++)
{
    if(c2[r]==b[0])
    q[r]=p[i];
    printf("\t%d",q[r]);
}
printf("\n");
}
}
printf("\nThe no of page faults is %d",c);
}

```

### **OUTPUT:**

Enter no of pages:10

Enter the reference string:7 5 9 4 3 7 9 6 2 1

Enter no of frames:3

7		
7	5	
7	5	9
4	5	9
4	3	9
4	3	7
9	3	7
9	6	7
9	6	2
1	6	2

The no of page faults is 10

### **c) Optimal Page replacement**

In operating systems, whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults. In this algorithm, OS replaces the page that will not be used for the longest period of time in future.

The idea is simple, for every reference we do following :

1. If referred page is already present, increment hit count.
2. If not present, find if a page that is never referenced in future. If such a page exists, replace this page with new page. If no such page exists, find a page that is referenced

farthest in future. Replace this page with new page.

**Program Code:**

```
#include<stdio.h>
int main()
{
int n,pg[30],fr[10];
int count[10],i,j,k,fault,f,flag,temp,current,c,dist,max,m,cnt,p,x;
fault=0;
dist=0;
k=0;
printf("Enter the total no pages:\t");
scanf("%d",&n);
printf("Enter the sequence:");
for(i=0;i<n;i++)
scanf("%d",&pg[i]);
printf("\nEnter frame size:");
scanf("%d",&f);

for(i=0;i<f;i++)
{
count[i]=0;
fr[i]=-1;
}
for(i=0;i<n;i++)
{
flag=0;
temp=pg[i];
for(j=0;j<f;j++)
{
if(temp==fr[j])
{
flag=1;
break;
}
}
if((flag==0)&&(k<f))
{
fault++;
fr[k]=temp;
k++;
}
else if((flag==0)&&(k==f))
{
fault++;
for(cnt=0;cnt<f;cnt++)
```

```

{
current=fr[cnt];
for(c=i;c<n;c++)
{
if(current!=pg[c])
count[cnt]++;
else
break;
}
}
max=0;
for(m=0;m<f;m++)
{
if(count[m]>max)
{
max=count[m];
p=m;
}
}
fr[p]=temp;
}
printf("\npage %d frame\t",pg[i]);
for(x=0;x<f;x++)
{
printf("%d\t",fr[x]);
}
}
printf("\nTotal number of faults=%d",fault);
return 0;
}

```

**Output:**

Enter the total no pages: 10

Enter the sequence:0

1  
2  
3  
0  
1  
2  
3  
0  
1

Enter frame size:3

page	0	frame	0	-1	-1
page	1	frame	0	1	-1
page	2	frame	0	1	2
page	3	frame	0	1	3
page	0	frame	0	1	3
page	1	frame	0	1	3
page	2	frame	0	2	3
page	3	frame	0	2	3
page	0	frame	0	2	3
page	1	frame	0	1	3

Total number of faults=6\_